Ping SDKs

July 2, 2025

SDKS

Table of Contents

What is available?

Our mission is to hide the complexity of underlying protocols and simplify your experience of integrating with Ping products.

We offer products that help developers build secure digital experiences, bringing apps to market faster and reducing costs and risk.

New name for the ForgeRock SDKs

The SDKs are being optimized to support diverse use cases across the entire Ping portfolio.

With a unified, modular architecture the Ping SDKs empower developers to seamlessly integrate any service, feature, or functionality into their apps, enabling quick and efficient access to the full range of Ping capabilities.

Learn more about the new name for the SDKs.

Ping SDKs

Our software development kits (SDKs) help you build secure digital experiences faster, for Android, iOS, and in JavaScript.

The SDKs enable you to easily integrate authentication, OAuth 2.0, registration, and self-service into your apps.

Ping SDKs for Authentication Journeys

PingOne Advanced Identity Cloud | PingAM

Integrate the Ping SDKs with **PingOne Advanced Identity Cloud** or **PingAM** for an embedded (in-app) experience.

Learn more »

DaVinci client for DaVinci Flows

PingOne | DaVinci

Integrate with **DaVinci flows** by using a web or native app in **PingOne** for an embedded (in-app) experience.

Learn more »

Ping SDKs for OIDC (centralized) login

PingOne | PingOne Advanced Identity Cloud | PingAM | PingFederate | OpenID Connect 1.0

Login to your apps using a browser-redirect, leveraging your server's own UI, or by creating your own UI, in a centralized (single) location.

Can be used with any OIDC-compliant server, including **PingOne**, **PingOne Advanced Identity Cloud**, **PingAM**, or **PingFederate**.

Learn more »

Ping (ForgeRock) Login Widget

PingOne Advanced Identity Cloud | PingAM

The Ping (ForgeRock) Login Widget is an all-inclusive UI component to help you add authentication, user registration, and other self-service journeys into your web applications.

The Ping (ForgeRock) Login Widget is only compatible with **PingOne Advanced Identity Cloud** and **PingAM**.

You can use the Ping (ForgeRock) Login Widget within React, Vue, Angular and a number of other modern JavaScript frameworks, as well as vanilla JavaScript.

Ping (ForgeRock) Login Widget »

ForgeRock Authenticator

PingOne Advanced Identity Cloud | PingAM

ForgeRock Authenticator is a multi-factor authentication application.

Users can download the application for Android and iOS and use it as part of their **PingOne Advanced Identity Cloud** and **PingAM** authentication journeys.

ForgeRock Authenticator »

Token Vault (Plugin)

OAuth 2.0 | OpenID Connect 1.0

Implemented as a plugin for the Ping SDK for JavaScript, Token Vault provides a feature called *origin isolation*.

Token Vault provides an additional layer of security for storing and using OAuth 2.0 and OpenID Connect 1.0 tokens in your JavaScript single-page applications (SPAs).

Token Vault (Plugin) »

Ping (ForgeRock) Authenticator module

PingOne Advanced Identity Cloud PingAM

The Ping (ForgeRock) Authenticator module helps you build the functionality of the ForgeRock Authenticator application into your own Android and iOS apps. The ForgeRock Authenticator works with both **PingOne Advanced Identity Cloud** and **PingAM**.

The module supports **time-based one-time passwords** (TOTP), **HMAC-based one-time password** (HOTP), and **Push notifications**.

Ping (ForgeRock) Authenticator module »

New name for the ForgeRock SDKs

What is the new name of the SDKs?

The new name for the ForgeRock SDKs is the **Ping SDKs**.

Why is the name being changed?

The SDKs are being optimized to support diverse use cases across the entire Ping portfolio. With a unified, modular architecture the Ping SDKs empower developers to seamlessly integrate any service, feature, or functionality into their apps, enabling quick and efficient access to the full range of Ping capabilities.

Figure 1. ForgeRock SDKs now known as the Ping SDKs

This furthers the commitment of a combined product offering, so you can continue to create the solutions you need. This means continued support for PingOne Advanced Identity Cloud, PingAM, as well as enabling other solutions such as PingOne DaVinci.

What other changes should I be aware of?

The existing ForgeRock Login Widget and ForgeRock Authenticator Module continue to provide support for PingOne Advanced Identity Cloud and PingAM exclusively; however, to align with the new naming conventions as well as to keep the server it supports intuitive, the names are slightly changing:

- The ForgeRock Login Widget is now the **Ping (ForgeRock) Login Widget**.

- The ForgeRock Authenticator Module is now the **Ping (ForgeRock) Authenticator module**.

The ForgeRock Token Vault is now **Token Vault** as you can use it with the Ping SDK for JavaScript for any server type, as long as the server is OAuth 2.0/OIDC compliant. This means that the Token Vault, when used in conjunction with the Ping SDK for JavaScript, supports and can be used with PingOne, PingOne Advanced Identity Cloud, or PingAM.

Design a protected system

Authentication, sessions, cookies, OAuth 2.0, authorization code flow, and so on. This page explains how to make sense of the complexity.

The modern system

In the early days, we wrote a single application that did it all. The gorgeous monolith! It did everything: handled user requests, authenticated users, rendered UIs, queried data directly from the database, served files, managed user sessions... everything. This could have been an application built with Rails, Spring, Node.js, but that's no longer a representation of a "modern system".

We now live in a world where "monolith" is a bad word. Everything has been split out into microservices, SPAs (single-page web app), PWAs (Progressive Web App), native mobile apps, with other functionality delegated to a FaaS, PaaS, or SaaS (Functions, Platform or Software as a Service).

This new design has given us a greater sense of organization and tooling to focus on solving the unique, novel problems independently of the common ones. Experts can now be responsible for their relative domains within their own repository or project. If a company does not employ an expert of a required domain, it can now "outsource" it to be managed by another company.

Unfortunately, this new paradigm comes with its own set of problems. Architecture diagrams now illustrate a complex web of distributed components that are simple in isolation, but hard to reason about when viewed holistically. Due to this distributed nature, the system now comes with more surface area to protect from unwanted access.

In a world where everything is a tap or click-of-the-finger away, it's more important than ever to ensure the right fingers have access to the right data. Knowing the basics of a protected system is no longer optional. Developers, product managers, IT professionals, all need to have a good grasp of the fundamentals.

Let's cover the basics to ensure we keep our data convenient but private and our users happy but safe.

What is a protected system?

In most modern, enterprise cases, "the system" will consist of a diverse collection of entities, but let's start with the most simple use case (not quite a system, but bear with me): the monolith.

Single, "full-stack", server-side application

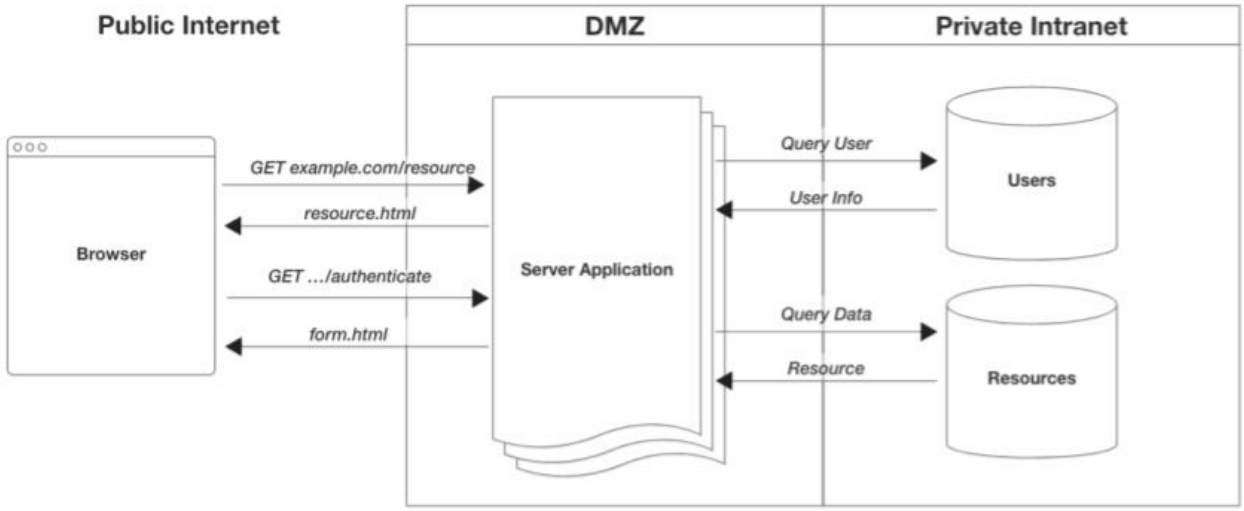


Figure 1. Architecture diagram of a monolith

This single application was responsible for everything, including identity and access management. These were applications common around the turn of the century. Though these "systems" still exist, they are becoming much less common as they are very hard to manage and engineer at large scale.

To take some baby steps, let’s consider one step up from this monolith, and separate out access management from the monolith.

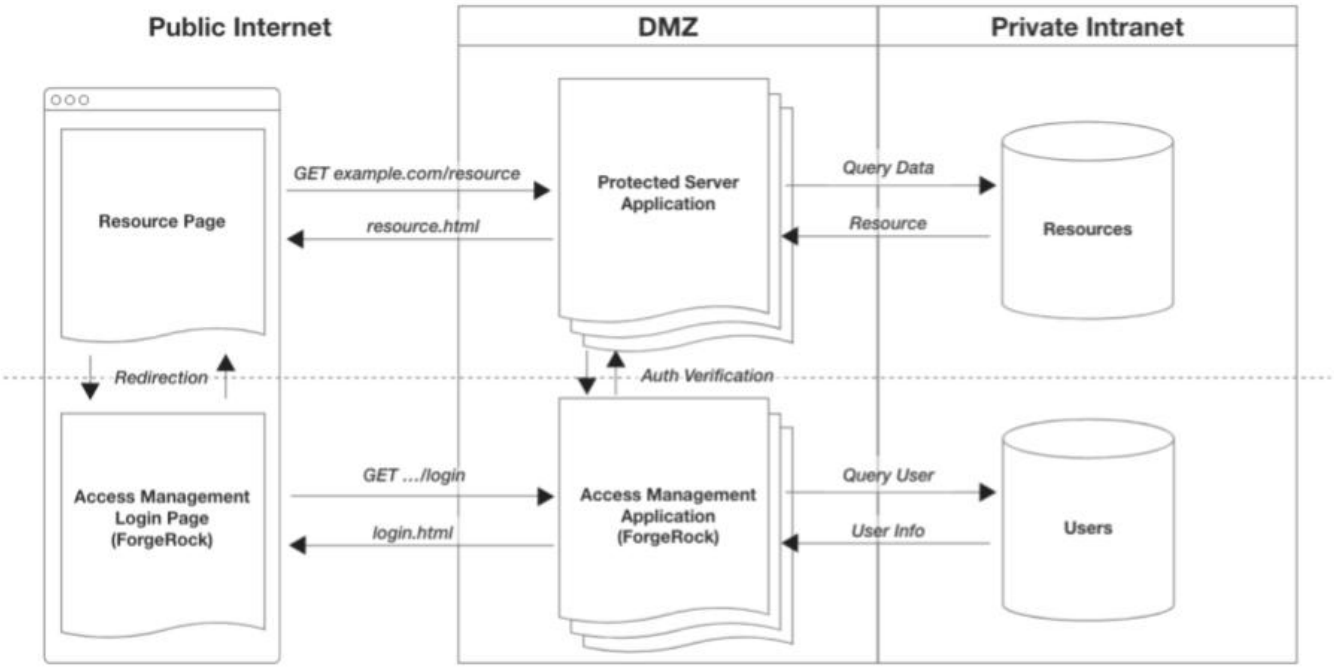


Figure 2. Architecture diagram of a monolith paired with access management app

In this design, you have two entities:

1. Protected, full-stack, server-side application: An application managing the resources you want protected.
2. **Access management application**: An application managing all identity and access concerns.

The beauty of this system is how it scales. If you decide to add another protected application to the system, you just delegate the access related needs to the access management application. (There are other great benefits to this, but let's save that for another article.) The new application introduced to the system could be a web app, mobile app, REST API service app, GraphQL app... anything that potentially serves up a protected resource.

In an effort to avoid having to rebuild such a vital function over and over with each new app, you "connect them" to your access management app. This dramatically reduces the surface area of risk in complex systems.

What's more, this serves the users better. It means they log in once, and have access to everything their role or privileges allow. With me so far? Okay, let's go a bit further.

What's a common system design?

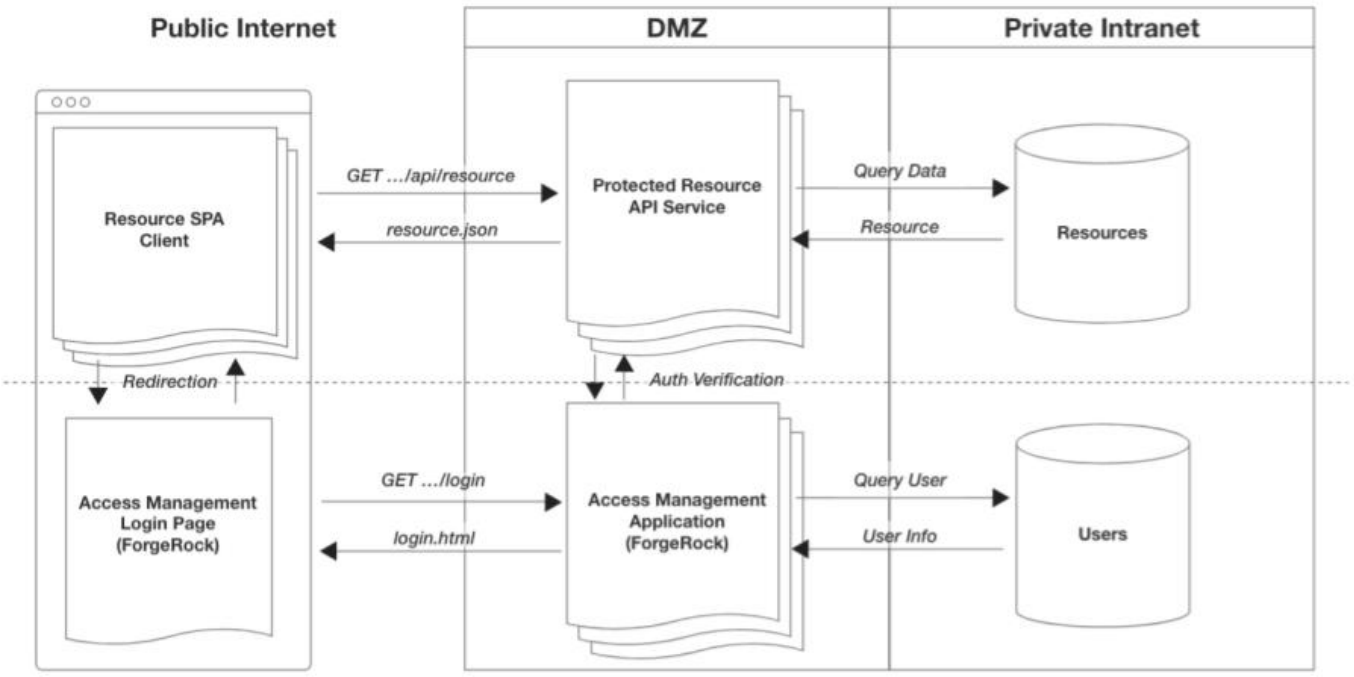
In modern system architectures, it's quite common to split the full-stack application into a backend with multiple, client-side apps, often one per platform: iOS, Android, Web. In these situations, it's advantageous to keep all data related concerns of our protected app within a central API server—often referred to as a "service". Each client app requests data via an API. This prevents business logic duplication across multiple applications and simplifies client-side development.

Let's add these multiple client-side apps as a generic entity to our system from above. We now have three distinct entity types as our "protected app" has been split into two application types:

1. **Protected client-side applications (Mobile and Web)**

2. Protected server-side, resource API service

3. Access management application



The main *access* responsibility of the protected client apps and the API service is to distinguish authenticated users from unauthenticated ones. This ensures those without access get denied, and converted to users with access by directing them to the access management app.

Let's break down the responsibility of each.

Client-side apps

The role of a protected client-side app is to not only distinguish between authenticated and unauthenticated users, but to assist in converting unauthenticated users into authenticated with as little friction as necessary.

An app will typically have both public and private portions. The simplest way to protect the private portion is by route, page or view. The protected routes will often have a reusable function that's run before any response is given, often referred to as "middleware". This function checks if the user has access by sending the access artifact, like a session cookie, to the access management app for validation. If the validation succeeds, the app continues processing the request; if not, the app will redirect the user to the login page.

This can be something as simple as this:

```
// Using a common client-side, middleware-style pattern (session-based example)
async function isAuthenticated(context, next) {
  const authResponse = await request(sessionValidateEndpoint);

  if (authResponse.valid) {
    next(); // continue with processing request
  } else {
    redirect(authenticationUrl); // send user to login
  }
}

routes('accounts/balances', isAuthenticated, (context) => {
  render(changePasswordForm);
});
```

> **Warning**
>
> It's important to know that protected client-side apps are not truly secure, and should not have embedded within them protected resources, secrets or private keys. They are inherently vulnerable as the entire codebase is sent to the user agent—a device outside of your control—to be executed, so all code is subject to manipulation.

Even though this client-side application cannot *guarantee* access protection, the implementation of such protection on the client increases user-experience and performance. It also reduces unnecessary requests to the underlying services.

Resource API services

The role of a protected resource API service is to be the final arbiter for protecting access to resources within the system. Since we can't *fully* trust our client-side applications, our resource API will need to duplicate the same check for authentication.

It will use the authentication artifact sent from the client with every request to validate the access to the requested resource. If validation passes, process the request. If it fails, send a 401 error message, and let the client-side app appropriately handle the issue:

```
// Using a Node.js middleware-style pattern (session-based example)
async function isAuthenticated(req, res, next) {
  const authResponse = await request(sessionValidateEndpoint);

  if (authResponse.valid) {
    next(); // continue with processing request
  } else {
    res.status(401).send(); // respond with 401 unauthorized
  }
}

routes.get('accounts/balances', isAuthenticated, (req, res) => {
  const balances = db.query('balances');

  res.json(balances);
});
```

> ⚠ **Warning**
>
> The above is route-level protection, which may not be granular enough for your system. Object-level protection, an increase in access control precision, may be required for your system but is outside the scope of this article.

Access management application

The access management (generic) application has the most important role in a protected system. It manages users, login, sessions, authorization, password management, and so on, all of which are vital functions.

At the simplest level, here are the main responsibilities of the application:

- Handles redirection from client-side apps for login, redirecting users back to the respective application upon completion.
- Provides an API for session/artifact validation.
- Provides an API for termination of session or artifact.

In situations where the above responsibilities exceed your level of comfort or skill set, it's often a good idea to delegate these responsibilities to a platform service provider, like ForgeRock. Our services and products allow you to focus on the novel aspects of your application development, and delegate the complexities of identity management (users, things, devices, and more), and access management (what those identities can do) to us.

Let's see how adopting Ping for our access management changes our system.

Integrate into a protected system

We provide a powerful, configurable Identity and Access Management solution out of the box. Whether it's an PingOne Advanced Identity Cloud tenant; self-hosted, cloud-ready container; or individual on-premise products, our products can provide a great solution for nearly any system. For simplicity, let's go with the PingOne Advanced Identity Cloud solution for the rest of this article.

PingOne Advanced Identity Cloud comes with its own login flow, registration and self-service journeys, as well as all the APIs needed for validation, refreshing, termination, authorization and more. This all-in-one solution works perfect for internal solutions or get-up-and-running quickly situations. But eventually, most companies want their user-facing experience to be fully customizable to suit their branding requirements.

> *If I'm redirecting my users to ForgeRock's platform, how do I provide a **fully** branded experience?*

In ForgeRock's PingOne Advanced Identity Cloud, you can choose how much control you want over your UIs. You can use it as-is, "theme" the provided UIs, or build your own UI using the underlying APIs and our open-source SDKs.

Build a branded UX with PingOne Advanced Identity Cloud and the SDKs

A fully branded experience means moving the responsibility of rendering the user authentication journey from PingOne Advanced Identity Cloud to an app that you will build. To facilitate this, we provide the Ping SDK for native Android and iOS apps, and for JavaScript application development. This allows you to easily integrate APIs into a new or existing app.

There are two choices for fully customizing the user experience:

1. Move the user authentication experience into each protected app, providing a native UX.
2. Move the user authentication experience into a single web app to centralize the login experience.

In both cases, our SDKs will help in developing these experiences. But, before we move on, it's important to know that your overall system has a significant impact on what choice suits you best.

What's your intended system design?

There are a few important points to consider when choosing how to protect your system:

1. How many client-side apps will need protecting?
2. Are all your apps and services served from a single domain?
3. Will there be any third-party apps or services that will need protection?

Let's dive into each concern and how it impacts your system.

How many client-side apps need protecting?

Say you have one app for each major platform: a web app, an iOS app, and an Android app. If the number of apps will not increase, you may want to develop the user experience for login, registration, and so on, within each app. This ensures that each app has full control over the best user experience for that platform.

By using our SDKs, you can more efficiently develop a dynamically responsive UI, handling each step within an authentication journey. This just slightly changes our client-side app's responsibilities.

Rather than redirecting unauthenticated users away from our application, we now just internally route the user to our native login experience. But, we will still continue to validate the user's session upon each navigation of our app.

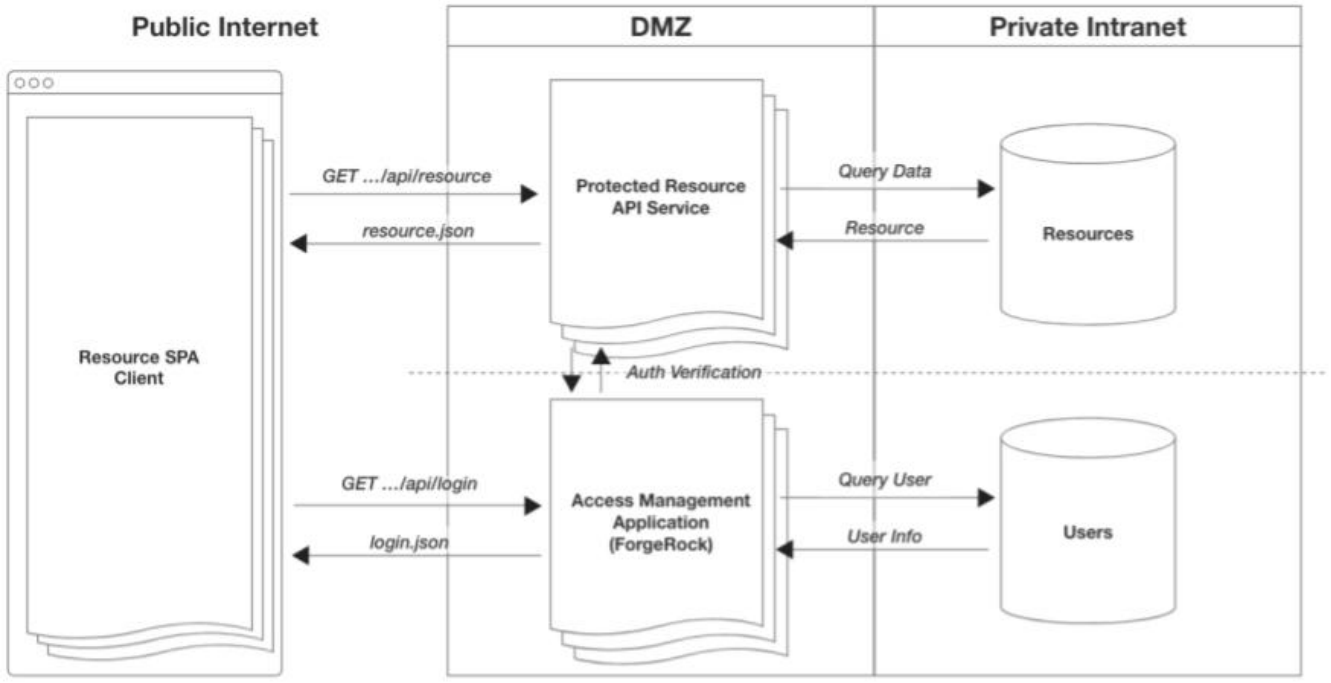


Figure 3. Architecture diagram of a SPA with Embedded Login and access management app

> *But, we have dozens/hundreds of client-side applications! We don't have the resources to update all of them.*

Now, if you have many apps, and each app needs to have within it a login (not to mention registration) flow, that's a lot of duplication. This will inevitably become a maintainability challenge, and a security liability as it increases your attack surface. Within this context, we need to go one step further.

To deal with this challenge, it's often recommended to extract the login (and possibly registration, self-service) related responsibilities out of the client-side apps, and build a single web app exclusively around this functionality. All front-end applications (mobile and web) can now redirect to this one, central application. This reduces your surface area for security liabilities as well as reduces duplication across your system.

Let's take a look at the system now:

1. Protected client-side apps (mobile & web)
2. Protected resource API services
3. **Authentication (login, registration & self-service) web app**
4. PingOne Advanced Identity Cloud

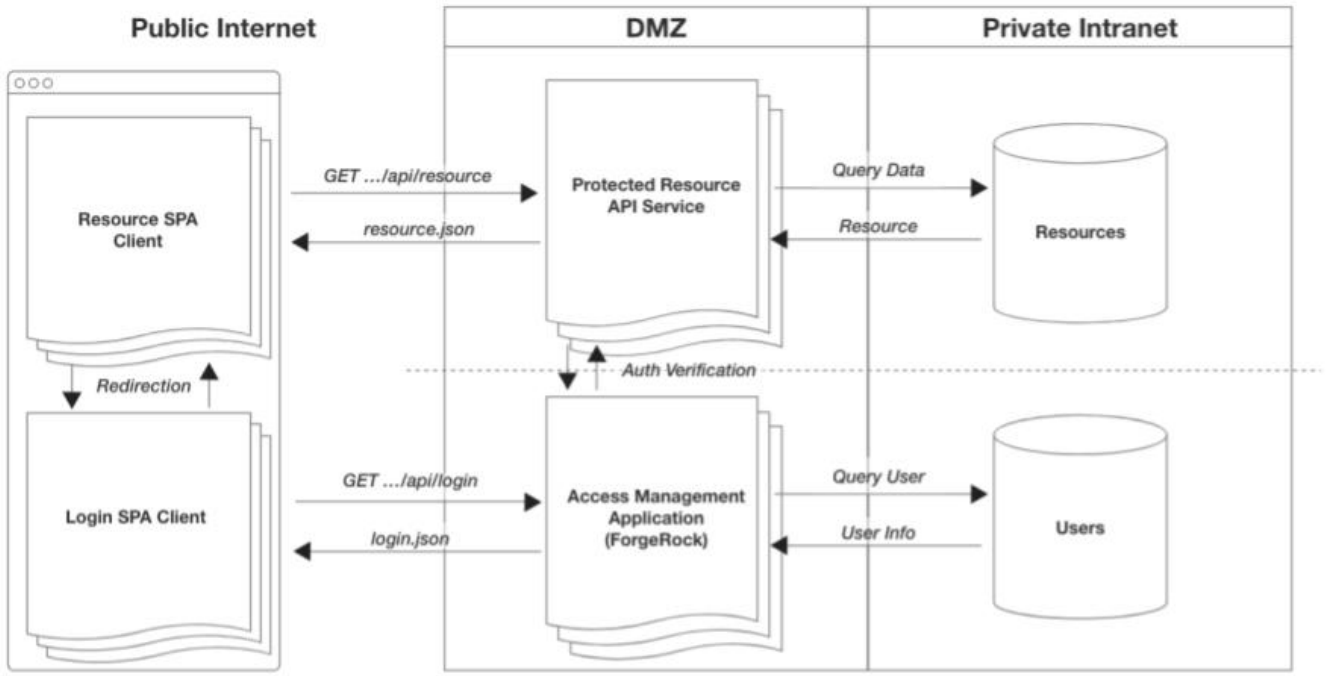


Figure 4. Architecture diagram of a SPA for resource app, a SPA for the login app, resource API server and access management app

With this design, we are now starting to organize the system components by scope of responsibility. For mobile applications, they'll have the availability of using the browser to authenticate, being redirected back to the native app when complete. Web apps will do a full redirect to the authentication app and a redirect back when done. Single sign-on functionality is provided out-of-the-box, as the browser is the shared platform for authentication between all apps, native or otherwise, on the user's device.

This provides a more scalable system that's optimized with apps having a more focused set of responsibilities while still providing full control over your brand and UX. Now that we have the core system design out of the way, let's discuss how all of these components will be hosted.

How are you hosting all these applications?

Simply put, are all the applications in the above system on the same host? For example:

- `mydomain.com/auth`
- `mydomain.com/app`
- `mydomain.com/api`

Another example would be the use of unique subdomains all on the same parent domain:

- `auth.mydomain.com`
- `app.mydomain.com`
- `api.mydomain.com`

If using either of the two patterns, a session-based system may work well for you. Sessions are frequently based on browser cookies, which are fundamentally restricted by the host or parent domain.

On the other hand, you may be using different hosts across your apps:

- `auth-server.com`
- `web-app.com`
- `rest-server.com`

This will constrain your options as session-based auth (driven by cookies) will be a challenge with apps on multiple hosts. An OAuth-based system is well-designed for this particular environment as it uses access tokens as the artifact passed around in the system, rather than a cookie.

But, before we dive into OAuth 2.0, let's discuss one more aspect of our system.

Any third-party companies involved?

Do you intend to extend access of your protected system to any third-party companies? For example, you may want to allow an application or service from an external company to interact with your protected system. For this, you likely want to restrict the scope of capabilities for these external entities, making an OAuth-based system a better choice.

What's OAuth and why is it better than session-based access with diverse hosting environments and third-party entities? Let's differentiate these two models.

Let's talk about access models (session v. OAuth)

To keep things simple, let's focus on two of the most common models of access: session-based and OAuth-based. Your system design, discussed above, should strongly influence the type of access model you want to implement, but it's not the only factor in making the choice.

Additional factors that can influence your access modeling are a bit more advanced and out of scope for this article, but they include:

1. Transaction authorization (aka policy enforcement)
2. Finer control over expiry times and access lifetimes
3. Finer control over scope of access or privileges

Look out for more information about these factors in a future article. For the rest of this article, let's talk a bit more about the basics of two foundational access models.

Session-based (cookies) access

The session-based model traditionally uses the *HTTP cookie* as its artifact. It's one of the oldest models for the web as the cookie was invented around the mid 1990's (though not originally for authentication). The HTTP cookie is a relatively simple way to persist data (a simple string of text) within a Web browser. This small piece of information is stored natively in the browser, and is tightly bound to the domain of the HTTP request the browser made to the server.

Let's use a simple example:

There's a web app running on `https://dashboard.example.com`, and an access management application running on `https://auth.example.com`. After making a request to the access management app to login, a "session cookie" gets added to the browser. This cookie is written because the server sent back a `Set-Cookie` header, so the cookie gets written to the full domain of the server, `auth.example.com`, or the parent domain, `example.com`.

Example of browser cookie storage:

```
-------------------- -------------------- --------------------
COOKIE NAME          VALUE                DOMAIN

session_id           AJi4QfFBCMzK3QFm...    .example.com
-------------------- -------------------- --------------------
```

Now that we have this cookie, all requests from that browser to `example.com` (even subdomains that share the same parent) will contain a `cookie` header with its value. It's worth noting that this "Just Works" as it's a seamless, almost invisible, mechanism of the browser.

Example of request with cookie:

```
GET https://auth.example.com/sessions/validate

HEADERS
content-type: application/json
cookie:       session_id=AJi4QfFBCMzK3Qc...s9dg7f6hyGHD
origin:       https://dashboard.example.com
```

This means you can have multiple apps running on multiple subdomains. As long the same parent or root domain is used, this session cookie will be sent automatically.

For example:

- `www.example.com`
- `accounts.example.com`
- `profile.example.com`
- `tasks.example.com`

With servers running on:

- `auth.example.com`
- `data.example.com`

As long as all apps, both client and server, are running on the same parent domain (`example.com`), you can configure cookies to work with this setup. In this case, we would configure the cookie to be written to the parent domain, `example.com` for the highest amount of flexibility. Your applications can then have their own subdomains and will still receive the cookie (many browsers store this as `.example.com`).

> **Note**
>
> The downside to this model is the tight coupling of cookies with their respective domains.

If you have apps running on different domains, say `auth.example.com` and `data.userbase.com` , this model unfortunately does not work. The cookies written for `auth.example.com` would not be sent to our `data.userbase.com` server. In this case, OAuth provides better support.

> **Warning**
>
> **Third-party cookies**: it's worth noting that there's still a nuance with cookies being written when browser-based apps (SPAs) are running on a different domain than the servers.
> These cookies are considered "Third-Party Cookies", and have been an important function of how the Web worked for years. Unfortunately, most browsers will disable this functionality within the next few years, so relying on it will be risky.
> Safari has already disabled third-party cookies by default.

OAuth 2.0-based access

OAuth is an industry standard for handling authorization and has been around since the late 2000's. OAuth 2.0 is the most recent specification of the protocol and is a large rework from the original. In this writing, any reference to OAuth will always refer to the 2.0 specification.

OAuth is a complex specification and has many variations and nuances. The details of which are beyond the scope of this article, so we will focus only on the basics.

The core artifact of OAuth is the *access token*, and like the value stored in a cookie for sessions, it is frequently just a simple string of text (sometimes called a JWT). But, unlike the cookie, the browser does not have a native concept of an access token, so obtaining and managing an Access Token doesn't automatically happen within a browser.

> **Note**
>
> There are other tokens frequently mentioned in texts about OAuth that are beyond the scope of this article, like refresh tokens and ID tokens. These tokens will not be covered in order to keep this article more introductory.

There are some choices about how to store and send the access tokens within a system. For the Web, as a simple example, `sessionStorage` or `localStorage` can often be used to store the token. Access Tokens are also not automatically sent along with all HTTP requests, so how one writes this token to HTTP requests is also something to be considered. Luckily, the industry has already standardized around *best practices*.

> *So, why is OAuth 2.0 better than session-based cookies in certain circumstances?*

OAuth is often mentioned in situations where you have third-party applications and services, or a multi-host setup with varying domains. This is because of its granularity of permissions (for security/privacy) and complete decoupling from domains. This provides more control over how it behaves. At the end of the day, an access token is just an opaque string that's passed around the system, frequently called a "bearer token", and written to the `Authorization` header of requests.

Example of request with authorization header:

```
GET https://rest.resource.com/activity

HEADERS
content-type:  application/json
authorization: Bearer 3QcIFmU6r0q43U...LJKf807
origin:        https://dashboard.example.com
```

Using OAuth doesn't dramatically change your system design. The basic principles of how it's used doesn't significantly diverge from the session-based model. You are still obtaining an access artifact from a server, passing it to APIs, and validating it where necessary. The additional responsibilities with Access Tokens are storing it and removing it as needed.

For example, here are some minor changes to the middleware example from above:

```
// Using Node.js middleware-style pattern (oauth-based example)
async function isAuthorized(req, res, next) {
  const authResponse = await request(oauthIntrospectionEndpoint);

  if (authResponse.access) {
    next(); // continue with processing request
  } else {
    res.redirect(authorizationUrl); // send to authorization
  }
}

routes.get('accounts/balances', isAuthorized, (req, res) => {
  res.render(changePasswordForm);
});
```

> **Note**
>
> Validating access tokens can also be done without a network request. We refer to these as "stateless" tokens. They can be introspected with a JWT decoding library for validation.

The only remaining difference between the OAuth and session-based model is the fact that an OAuth token has to be specially obtained from your access management application. The most common flow for attaining an access token is called the authorization code flow, and involves an additional interaction with the server after the user successfully authenticates.

The good thing is you do not have to reinvent the wheel to implement OAuth within your applications. PingOne Advanced Identity Cloud and SDKs abstract away the need for requesting, storing, sharing, and revoking the access token, leaving you with more time to build the novel aspects of your applications.

What's the best design to protect my system?

The answer is... well, *it depends*. As discussed above, there are quite a few important aspects to the kind of system we are discussing and the future plans for your products. Hopefully, after reading through the basics articulated above, you have a better, foundational understanding of what it means to design a protected system.

If things are still a bit fuzzy, don't worry. The good news is that Ping can help by providing the best tools and guidance to ensure you have the right information to make the best choice for you.

Security

The Ping SDKs are built from the ground up to use best practices for securing token material and data.

Security is a very broad subject, and every environment is different. Readers are expected to do their own research and complement the information found in these topics.

Tokens and keys

Learn how the Ping SDKs secure your session and OAuth 2.0-related tokens, and the encryption used.

Authentication

Discover the protocols the Ping SDKs use when your app authenticates your users.

Data

What data do the Ping SDKs use, and what security measures help to protect it.

OAuth 2.0

See how the Ping SDKs use Proof Key for Code Exchange (PKCE) to mitigate the risks of an OAuth 2.0 attack.

Token and key security

The Ping SDKs handle and store keys and tokens based on the security best practices of each platform.

Token storage

Depending on the authentication use case, the SDKs will potentially have to store and be able to retrieve the session cookie, ID tokens, access tokens, and refresh tokens.

Each token is serving a different use case, and as such how the SDKs handle them can be different.

The following sections cover how the SDKs handle different types of tokens.

Session tokens and cookies

- On Android and iOS, the session tokens are stored in either the Android keystore or iOS keychain after authentication completes. The tokens are encrypted using a hardware-backed security key when possible and can be retrieved by the SDK on request.
- When using the Ping SDK for JavaScript, cookies are stored in the browser's cookie storage. The cookie name matches the one provided by PingAM (such as `iPlanetDirectoryPro`) and its value is the actual session token. When making requests to PingAM, the value is passed as an authentication cookie. This cookie is configured with the `HTTPOnly` and `Secure` attributes, which provide additional layers of security.

ID, access, and refresh tokens

- On Android and iOS when authorization is completed any OAuth 2.0-related tokens are stored securely locally, encrypted using a hardware-backed security key when possible and can be retrieved by the SDK on request. Tokens **are not** configured as *cloud sharable* by default.
- When using the Ping SDK for JavaScript, the OAuth 2.0 Tokens are stored by using one of the web storage APIs provided by the browser. By default, this uses the browser's `localStorage`, but the SDK also supports `sessionStorage`.

> **Important**
>
> We recommend that JavaScript single-page applications do not use refresh tokens or any other long-running authorization elements due to the potentially unsecure nature of the storage mechanisms provided by browsers.

In addition to built-in storage schemes, Android and JavaScript app developers can provide a custom storage mechanism that can be passed to the SDK. Learn more in Customize storage on JavaScript.

Token lifecycle

The session and OAuth 2.0-related tokens the SDKs handle all have associated expiry times. When a token reaches its expiry time it becomes unusable.

A feature of the SDKs is that they manage the refresh of *OAuth 2.0 tokens*. The timing of the refresh is based on a threshold value to improve the end-user experience. The SDKs refresh tokens automatically when the token is requested from storage to be used in your application and its expiry is within the threshold.

In the case of access tokens, if a refresh token is present, then the Android and iOS SDKs will use it to obtain a new access token. If the refresh token cannot be used, is not present, or if it has expired, then the SDKs fall back to using the session token to start a new OAuth 2.0 flow.

> **Note**
>
> The SDKs do not handle the refresh of *session tokens*. If a session token has expired, the app needs to re-authenticate the user.

When an OAuth 2.0 or session token expires, the SDK removes any respective tokens from the secure storage and performs a cleanup. The Android and iOS SDKs also check if the current session token is the same one used to obtain the OAuth 2.0 tokens. In case of a mismatch, then these orphaned tokens are cleaned.

When using SDK logout methods to perform a `Logout` event, the SDKs revoke existing OAuth 2.0 tokens, revoke the session, and perform a local cleanup. If the SDKs are unable to revoke the session at the server—for example the network is unavailable—then the SDKs remove the tokens from local storage.

When using the Ping SDK for JavaScript, if an access token expires within the threshold limit or returns an HTTP `401 Unauthorized` error, the SDK attempts to renew it using the same session cookie that was performing the authorization code OAuth 2.0 flow.

The Ping SDK for JavaScript calls the `endSession` and `session?action=logout` endpoints during logout, as well as calling `revoke` whenever you use `FRUser.logout`. This ensures that the **server** invalidates the session cookie.

> **Note**
>
> The Ping SDK for JavaScript has no *direct* control over the session cookie; it can only make requests to the browser that may or may not be acted upon. Instead, it must rely on the server to manage the cookie removal.

Encryption key storage

On supported platforms and devices, the Ping SDKs generate Hardware-Backed encryption keys, and uses them to encrypt and store tokens. This provides an extra level of security against attacks.

- The Ping SDK for iOS uses the `kSecKeyAlgorithmECIESEncryptionCofactorX963SHA256AESGCM` encryption algorithm. The key is stored in the *Secure Enclave*.

  On unsupported devices, the SDK cannot not enforce hardware-backed encryption and will save the tokens in the iOS keychain.

- The Ping SDK for Android uses a number of different algorithms, depending on the OS version and device functionality. It supports the following encryptors:

  - `AndroidLEncryptor`: RSA
  - `AndroidMEncryptor`: AES
  - `AndroidNEncryptor`: Similar to ***M***, with the addition of setting `setInvalidatedByBiometricEnrollment` to `true`
  - `AndroidPEncryptor`: Similar to ***N***, with the addition of using *Android Strongbox*

Hardware-backed key storage and encryption

Both the Android and iOS SDKs use platform-provided methods to create hardware-backed encryption keys.

- On iOS the SDK creates keys within the `SecuredKey.swift` class. If `SecuredKey` generation fails, the `KeychainManager` generates the `KeychainService` with no `SecuredKey`. The values in this case will be added to the iOS keychain as `kSecClassGenericPassword` types.

  If `SecuredKey` creation is successful then the value is encrypted before being stored. The `SecuredKey.swift` class provides an `isAvailable()` public method that validates whether creation of the `SecuredKey` using Secure Enclave is available on the device or not.

  > **Tip**
  >
  > The SDKs also support devices that do not have Secure Enclave or other hardware-backed encryption functionality.

- On Android, the SDK uses `DefaultTokenManager` and `DefaultSingleSignOnManager` for storing tokens, in addition to `SecuredSharedPreferences` on supported devices.

  Depending on the Android version, the SDK can use more specific encryptors. For more information, see `getEncryptor`. For information about the different encryptor classes, see the `auth` folder in GitHub.

Authentication security

The Ping SDKs provide two methods for implementing authentication in your applications:

Auth journey (embedded) login

The app developer is responsible for building the login and registration UI.

Uses the Authorization code grant with PKCE flow, based on RFC7636.

When using auth journeys for authentication, the SDKs do not store user credentials on the device or in the browser.

OIDC (centralized) login

We provide a central login UI that app developers can use with a redirect for JavaScript apps, or by using an in-app browser in Android and iOS applications.

Android and iOS use the OAuth 2.0 for Native Apps, based on RFC8252, which is recommended way for third-party applications to authenticate in terms of security, as user credentials are never exposed to the third-party web or native application.

Both options have their merits and drawbacks, and the choice usually depends on your use case. For more information, refer to:

- Auth journey (embedded) login
- OIDC (centralized) login

The Ping SDKs also use the following protocols for authentication:

WebAuthn for Mobile and Web Biometrics

Based on the WebAuthn W3C spec.

- The Ping SDK for iOS uses a custom implementation of the protocol that has been created to offer backward compatibility older iOS versions including iOS 12. For more information, see Supported operating systems.
- The Ping SDK for Android uses the Google FIDO2 API.

Data security

The Ping SDKs do not save or load any user data, such as username or password, or personal information in memory. The only stored keys and data are the Session and OAuth 2.0 tokens required for authentication, and security-related certificates hashes.

The Ping SDKs for iOS and Android support *SSL Pinning*. The certificate information used is passed in the form of certificate key hashes in the SDKs configuration file. This means you do not have to bundle certificates with your iOS `.ipa` or Android `.apk` files.

OAuth 2.0 security with PKCE

Proof Key for Code Exchange (PKCE) mitigates the risks of an OAuth 2.0 attack. Without PKCE, a malicious application running in the same browser as your public client app could compromise the security of your app.

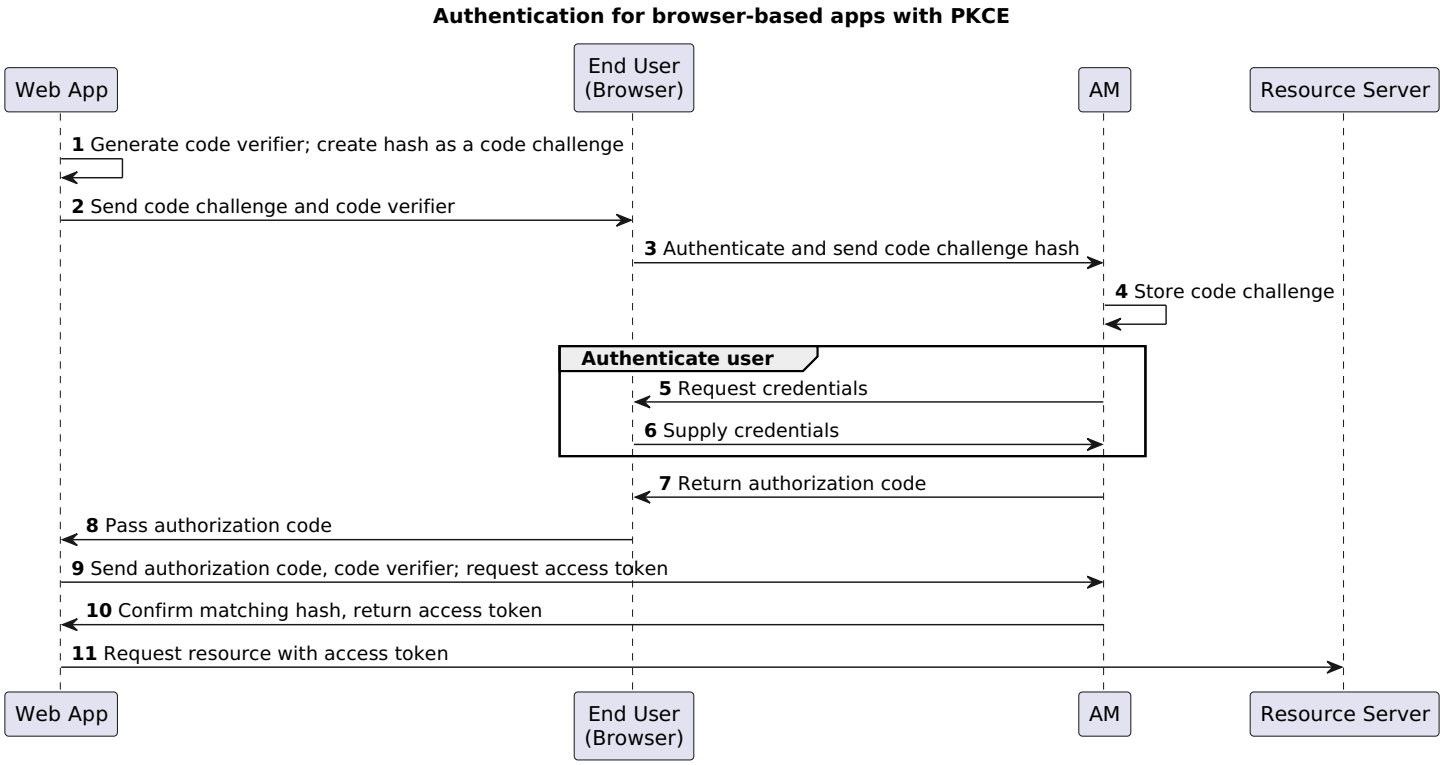
It is good practice to use PKCE for native apps and SPAs, because the code is stored on browsers and devices. Without PKCE, you'd have to include a client secret in those public-facing apps. For enhanced security, you should use PKCE whenever you have the option to use it.

How PKCE works

Your app, with the help of our code, generates a `code_verifier` (nonce). When a user make a request, your app creates a hash of that `code_verifier` as a `code_challenge`. ForgeRock, as an authorization server, saves the hash value.

After the hash is confirmed as valid, your app exchanges its authorization code grant for an access token. Your client app, as the bearer, can use the token to access to the user's resources.

This diagram depicts the authorization code grant flow in detail:



If you're familiar with OpenID Connect (OIDC) specifications, the web app is the *relying party*, and PingOne Advanced Identity Cloud or PingAM is the *authorization server*.

For more information on PKCE standards, see the following IETF document: Proof key for code exchange by OAuth public clients.

For more information on how we implement PKCE for native and SPA apps, refer to Authorization code grant with PKCE.

What's New

Subscribe to get automatic updates:

- Ping SDKs Changelog RSS feed
- Ping SDKs Changelog email notifications

> **Important**
>
> SDKs Renamed
> Prior to November 2024, the **Ping SDKs** were known as the **ForgeRock SDKs**.

Latest updates

SDK for Android 4.8.1 released NEW

25 June, 2025

A new version of the ForgeRock SDK for Android is now available with improvements and fixes over earlier versions:

- Improved performance by adding caching for KeyStore, Cipher, and Symmetric Key encryption and decryption.
- Added a `strongBoxPreferred=false` parameter to allow conditional use of StrongBox for key storage.

  Learn more in Preventing the Keystore System from using StrongBox.

Full changelog

SDK for Android 4.8.0 released

16 May, 2025

A new version of the ForgeRock SDK for Android is now available with improvements and fixes over earlier versions:

- Added the ability to update the Firebase Cloud Messaging (FCM) device token for existing devices registered for push notifications.

  Learn more in Updating device tokens for existing accounts.

- Added support for returning WebAuthn authenticator information in the updated WebAuthn authentication and registration callbacks introduced in PingAM and PingOne Advanced Identity Cloud.

  Learn more in Accessing WebAuthn authenticator information.

Full changelog

SDK for iOS 4.8.0 released NEW

16 May, 2025

A new version of the ForgeRock SDK for iOS is now available with improvements and fixes over earlier versions:

- Added the ability to update the Apple Push Notification Service (APNs) device token for existing devices registered for push notifications.

  Learn more in Updating device tokens for existing accounts.

- Added support for returning WebAuthn authenticator information in the updated WebAuthn authentication and registration callbacks introduced in PingAM and PingOne Advanced Identity Cloud.

  Learn more in Accessing WebAuthn authenticator information.

Full changelog

SDK for JavaScript 4.8.0 released NEW

16 May, 2025

A new version of the SDK for JavaScript is now available with improvements and fixes over earlier versions:

- Added a flag to skip immediately to the OAuth 2.0 flow rather than attempting to get tokens without redirecting.

  Learn more in Configure JavaScript apps for OIDC login.

- Added support for signing out of PingOne by using an ID token.

Full changelog

DaVinci Client for Android 1.1.0 released NEW

15 April, 2025

A new version of the DaVinci Client for Android is now available with improvements and fixes over earlier versions:

- Added social sign on with supported external IDPs.

  Learn more in Set up social sign on with external IDPs.

- Added `Accept-Language` header customization to support localization.

  Learn more in Localizing the user interface.

- Added support for additional PingOne Form fields.

  Learn more in Supported PingOne fields and collectors.

Full changelog

DaVinci Client for iOS 1.1.0 released NEW

15 April, 2025

A new version of the DaVinci Client for iOS is now available with improvements and fixes over earlier versions:

- Added social sign on with supported external IDPs.

Learn more in Set up social sign on with external IDPs.

- Added `Accept-Language` header customization to support localization.

  Learn more in Localizing the user interface.

- Added support for additional PingOne Form fields.

  Learn more in Supported PingOne fields and collectors.

- Added support for Swift 6.

Full changelog

DaVinci Client for JavaScript 1.1.0 released NEW

15 April, 2025

A new version of the DaVinci Client for JavaScript is now available with improvements and fixes over earlier versions:

- Added social sign on with supported external IDPs.

  Learn more in Set up social sign on with external IDPs.

- Added middleware support to alter `Accept-Language` header to support localization.

  Learn more in Localizing the user interface.

- Added support for additional PingOne Form fields.

  Learn more in Supported PingOne fields and collectors.

Full changelog

SDK for Android 4.7.0 released NEW

11 February, 2025

A new version of the ForgeRock SDK for Android is now available with improvements and fixes over earlier versions:

- Added support for user profile self-service.

  Learn more in Set up user profile self service.

- Added support for managing registered devices.

  Learn more in Set up registered device self service.

- Added support for signing-out of PingOne with an ID token.

Full changelog

SDK for iOS 4.7.0 released

11 February, 2025

A new version of the ForgeRock SDK for iOS is now available with improvements and fixes over earlier versions:

- Added support for user profile self-service.

  Learn more in Set up user profile self service.

- Added support for managing registered devices.

  Learn more in Set up registered device self service.

- Added support for signing-out of PingOne with an ID token.

Full changelog

SDK for JavaScript 4.7.0 released NEW

11 February, 2025

A new version of the SDK for JavaScript is now available with improvements and fixes over earlier versions:

Added

- Added support for managing registered devices.

  Learn more in Set up registered device self service.

Changed

- Prioritized `displayName` field over `userName` when saving a WebAuthn or passkey to an account. Previously the SDK displayed a UUID for saved credentials rather than the user's name.

Full changelog

DaVinci client 1.0.0 released

16 December, 2024

The first version of the DaVinci client for Android, iOS and JavaScript is now available.

> **Note**
>
> The Ping SDK DaVinci clients are constantly evolving to meet your business needs.
> Check back from time to time on latest updates and enhancements.

- Supports the **Custom HTML Template** capability of the **HTTP Connector**.
- Supports the following fields:
  - Text field
  - Password field
  - Submit button
  - Flow button

To learn more, refer to Supported DaVinci fields.

- Follow the DaVinci client tutorials to quickly setup a demo app to connect to your DaVinci flows.
- Read how to configure the DaVinci client to leverage DaVinci flows in your native or single-page apps.

Full changelog

SDK for Android 4.6.0 released

17 October, 2024

A new version of the ForgeRock SDK for Android is now available with improvements and fixes over earlier versions:

- Added the ability to customize how the ForgeRock SDK stores tokens and data.

  To learn more, refer to Customizing storage.
- Added support for Android App Links that use the http/https scheme for redirect URIs in centralized login apps.
- Added support for Android 15.
- Added support for the PingOne Protect Marketplace nodes.

Full changelog

SDK for iOS 4.6.0 released

17 October, 2024

A new version of the ForgeRock SDK for iOS is now available with improvements and fixes over earlier versions:

- Added support for the PingOne Protect Marketplace nodes.
- Exposed the realm, success URL and failure URL values within `Token`.
- Added client-side support for the upcoming `ReCaptchaEnterpriseCallback` callback.

Full changelog

SDK for JavaScript 4.6.0 released

17 October, 2024

A new version of the ForgeRock SDK for JavaScript is now available with improvements and fixes over earlier versions:

- Added centralized login support for PingFederate servers.
- Added client-side support for the upcoming `ReCaptchaEnterpriseCallback` callback.
- Added support for the PingOne Protect Marketplace nodes.

Full changelog

SDK for Android 4.5.0 released

9 July, 2024

A new version of the ForgeRock SDK for Android is now available with improvements and fixes over earlier versions:

- Added support for signing off from PingOne to the centralized login flow.

  To learn more, follow the Android tutorial for PingOne.

- Added the ability to dynamically configure the SDK by collecting values from a PingOne or server's OpenID Connect `.well-known` endpoint.

  To learn more, refer to Using the .well-known endpoint for dynamic configuration.

Full changelog

SDK for iOS 4.5.0 released

9 July, 2024

A new version of the ForgeRock SDK for iOS is now available with improvements and fixes over earlier versions:

- Added support for signing off from PingOne when using the centralized login flow with OAuth 2.0.

  To learn more, follow the iOS tutorial for PingOne.

- Added the ability to dynamically configure the SDK by collecting values from the server's OpenID Connect `.well-known` endpoint.

  To learn more, refer to Using the .well-known endpoint for dynamic configuration.

Full changelog

Ping (ForgeRock) Login Widget 1.3.0 released NEW

30 May, 2024

A new version of the ForgeRock SDK for JavaScript is now available with improvements and fixes over earlier versions:

- Added support for integration with PingOne Protect.
- Added the name of the device to the recovery codes page.

Full changelog

SDK for JavaScript 4.4.2 released

15 May, 2024

A new version of the ForgeRock SDK for JavaScript is now available with improvements and fixes over earlier versions:

- Added a `logoutRedirectUri` parameter to the `FRUser.logout()` method.

  Add the parameter to invoke a redirect flow, for revoking tokens and ending sessions created by a PingOne server.

To learn more, follow the JavaScript tutorial for PingOne.

- Added a `platformHeader` configuration property to control whether the SDK adds the `X-Requested-Platform` header to all outgoing connections.

Full changelog

SDK for iOS 4.4.1 released

25 April, 2024

A new version of the ForgeRock SDK for iOS is now available with improvements and fixes over earlier versions:

- Added privacy manifest files to ForgeRock SDK for iOS modules.

Full changelog

SDK for iOS 4.4.0 released

4 April, 2024

A new version of the ForgeRock SDK for iOS is now available with improvements and fixes over earlier versions:

- Added a new module for future integration with PingOne Protect.

  Learn more at Integrate with PingOne Protect for risk evaluations.

- Added an interface for customizing the biometric UI prompts when device binding or signing.

  Learn more at Bind and verify devices.

- Added support for the `TextInput` callback.

Full changelog

SDK for Android 4.4.0 released

28 March, 2024

A new version of the ForgeRock SDK for Android is now available with improvements and fixes over earlier versions:

- Added a new module for future integration with PingOne Protect.

  Learn more at Integrate with PingOne Protect for risk evaluations.

- Added an interface for customizing the biometric UI prompts when device binding or signing.

  Learn more at Bind and verify devices.

- Added support for the `TextInput` callback.

Full changelog

SDK for JavaScript 4.4.0 released

13 March, 2024

A new version of the ForgeRock SDK for JavaScript is now available with improvements and fixes over earlier versions:

- Added a new module for future integration with PingOne Protect.
- Added the ability to include the supplied device name when displaying recovery codes.

  Learn more at Using the device name.
- Added the ability to use values from an OpenID Connect `.well-known` URL to automatically configure the SDK paths.

  This simplifies using the SDKs with OIDC-compliant identity providers, such as **PingOne**.

  For more information, refer to the ForgeRock SDK for JavaScript PingOne tutorial.

Full changelog

SDK for Android 4.3.1 released

9 February, 2024

A new version of the ForgeRock SDK for Android is now available with improvements and fixes over earlier versions:

- Fixed an SDK crash during device binding on Android 9 devices.

Full changelog

Ping (ForgeRock) Login Widget 1.2.1 released

8 January, 2024

A new version of the Ping (ForgeRock) Web Login Framework is now available with improvements and fixes over earlier versions:

- Support for CAPTCHA nodes.

Full changelog

SDK for JavaScript 4.3.0 released

4 January, 2024

A new version of the ForgeRock SDK for JavaScript is now available with improvements and fixes over earlier versions:

- Added ability to override default prefix string given to storage keys.

  For more information, refer to `prefix` in the ForgeRock SDK for JavaScript Properties.
- Added an `FRQRCode` utility class to determine if a step has a QR code and handle the data to display.

  For more information, refer to Set up QR code handling.

Full changelog

SDK for Android 4.3.0 released

28 December, 2023

A new version of the ForgeRock SDK for Android is now available with improvements and fixes over earlier versions:

- Added ability to customize cookie headers in outgoing requests from the SDK.
- Added ability to add custom claims when verifying signatures from bound devices.
- Added client-side support for the upcoming `AppIntegrity` callback.

Full changelog

SDK for iOS 4.3.0 released

28 December, 2023

A new version of the ForgeRock SDK for iOS is now available with improvements and fixes over earlier versions:

- Added client-side support for the upcoming `AppIntegrity` callback.
- Added a new `ephemeralAuthSession` browser type for iOS13 and later.
- Added `iat` and `nbf` claims to the device binding JWS payload.
- Added ability to insert custom claims when performing device signing verification.
- Updated the detection of Jailbreak status.

Full changelog

SDK for Android 4.2.0 released

3 October, 2023

A new version of the ForgeRock SDK for Android is now available with improvements and fixes over earlier versions:

- Added Gradle 8 and JDK 17 support.
- Added Android 14 support.
- Added verification of key pairs during device binding enrollment by using Google Key Attestation.
- Added *issued at* (`iat`) and *not before* (`nbf`) claims to JSON Web tokens used for device binding and signing verification.

Full changelog

Token Vault 4.2.0 released

11 September, 2023

A new version of the Token Vault is now available with improvements and fixes over earlier versions:

- Added a requirement to declare a list of URLs in the Token Vault Proxy configuration. These generate an allowlist of origins to which the proxy can forward requests.

Full changelog

SDK for JavaScript 4.2.0 released

11 September, 2023

A new version of the ForgeRock SDK for JavaScript is now available with improvements and fixes over earlier versions:

- Added a `logLevel` configuration property to specify the level of logging the SDK performs.
- Added a `customLogger` configuration property to specify a replacement for the native `console.log` that the SDK uses by default.

  For example, you could write a replacement that captures SDK log output to services such as Relic or Rocket.

Full changelog

SDK for Android 4.1.0 released

31 July, 2023

A new version of the ForgeRock SDK for Android is now available with improvements and fixes over earlier versions:

- Added support for interceptors in the authenticator module
- Added an interface for refreshing access tokens
- Added support for policy advice from IG in JSON format

Full changelog

SDK for iOS 4.1.0 released

28 July, 2023

A new version of the ForgeRock SDK for iOS is now available with improvements and fixes over earlier versions:

- Added support for interceptors in the authenticator module.
- Added support for `mfauth` deep links in the authenticator sample app.
- Added an interface for refreshing access tokens.
- Added support for policy advice from IG in JSON format.

Full changelog

Token Vault 4.1.2 released

24 July, 2023

Token Vault provides an additional layer of security for storing and using OAuth 2.0 and OpenID Connect 1.0 tokens in your JavaScript single-page applications (SPAs).

Implemented as a plugin for the ForgeRock SDK for JavaScript, Token Vault provides a feature called *origin isolation*.

Full changelog

SDK for JavaScript 4.1.2 released

20 July, 2023

A new version of the ForgeRock SDK for JavaScript is now available with improvements and fixes over earlier versions:

- Added support in preparation for upcoming Token Vault.

Full changelog

Ping (ForgeRock) Login Widget 1.1.0 released

17 July, 2023

A new version of the Ping (ForgeRock) Web Login Framework is now available with improvements and fixes over earlier versions:

- Support for device profiling callbacks (`DeviceProfileCallback`)
- Support for web authentication (WebAuthn) journeys and trees.

Full changelog

SDK for JavaScript 4.1.1 released

29 June, 2023

A new version of the ForgeRock SDK for JavaScript is now available with improvements and fixes over earlier versions:

- Added support in the HTTPClient for receiving transactional authorization advice in JSON format.

Full changelog

SDK for iOS 4.0.0 released

9 June, 2023

A new version of the ForgeRock SDK for iOS is now available with improvements and fixes over earlier versions:

- Added support for Passkeys.
- Added the ability to provide a device name when registering WebAuthN devices.
- Added support for enforcing policies in the Authenticator SDK.
- Added SwiftUI quick start sample code.

Full changelog

> **Important**
>
> This release contains changes that could affect the functionality of your app.
> Refer to Incompatible changes.

SDK for Android 4.0.0 released

30 May, 2023

A new version of the ForgeRock SDK for Android is now available with improvements and fixes over earlier versions:

- Upgraded the Google Fido client to support Passkeys.
- Added the ability to provide a device name when registering WebAuthN devices.
- Added support for enforcing policies in the Authenticator SDK.

Full changelog

> **Important**
>
> This release contains changes that could affect the functionality of your app.
> Refer to Incompatible changes.

SDK for JavaScript 4.0.0 released

23 May, 2023

A new version of the ForgeRock SDK for JavaScript is now available with improvements and fixes over earlier versions:

- Added the ability to provide a device name when registering WebAuthN devices.
- Updated ESModule (ESM) bundle.
- Updated tags in the GitHub repo to be prefixed with the package name. For example, `javascript-sdk-${tag}`.

Full changelog

> **Important**
>
> This release contains changes that could affect the functionality of your app.
> Refer to the following pages for details:
>
> - Incompatible changes
> - Deprecations

Ping (ForgeRock) Login Widget 1.0.0 released

18 April, 2023

The Ping (ForgeRock) Login Widget is an all-inclusive UI component to help you add authentication, user registration, and other self-service journeys into your web applications.

The Ping (ForgeRock) Login Widget uses the Ping SDK for JavaScript internally, and adds a user interface and state management. This rendering layer helps eliminate the need to develop and maintain the UI components for providing complex authentication experiences.

Full changelog

Full changelogs

Ping SDK for Android

Ping SDK for iOS

Ping SDK for JavaScript

DaVinci client

Ping (ForgeRock) Login Widget

Token Vault

Legacy releases

Release timeline

Key:

N.N.N = Latest version

Release date	Platform	SDK version	Release type[1]
2025-MAY-16	Ping SDK for Android	4.8.0	Minor
2025-MAY-16	Ping SDK for JavaScript	4.8.0	Minor
2025-MAY-16	Ping SDK for iOS	4.8.0	Minor
2025-APR-15	DaVinci client for Android	1.1.0	Minor
2025-APR-15	DaVinci client for iOS	1.1.0	Minor
2025-APR-15	DaVinci client for JavaScript	1.1.0	Minor
2025-FEB-11	Ping SDK for Android	4.7.0	Minor
2024-FEB-11	Ping SDK for iOS	4.7.0	Minor
2025-FEB-11	Ping SDK for JavaScript	4.7.0	Minor
2024-DEC-16	DaVinci client	1.0.0	Major
2024-OCT-17	Ping SDK for Android	4.6.0	Minor
2024-OCT-17	Ping SDK for iOS	4.6.0	Minor
2024-OCT-17	Ping SDK for JavaScript	4.6.0	Minor
2024-JUL-11	Ping SDK for Android	4.5.0	Minor
2024-JUL-11	Ping SDK for iOS	4.5.0	Minor
2024-JUN-05	Login Widget	1.3.0	Minor
2024-MAY-15	Ping SDK for JavaScript	4.4.2	Patch
2024-APR-25	Ping SDK for iOS	4.4.1	Patch
2024-APR-04	Ping SDK for iOS	4.4.0	Minor
2024-MAR-28	Ping SDK for Android	4.4.0	Minor
2024-MAR-13	Ping SDK for JavaScript	4.4.0	Minor

Release date	Platform	SDK version	Release type[1]
2024-FEB-09	Ping SDK for Android	4.3.1	Patch
2024-JAN-08	Login Widget	1.2.1	Minor
2024-JAN-04	Ping SDK for JavaScript	4.3.0	Minor
2023-DEC-28	Ping SDK for Android	4.3.0	Minor
2023-DEC-28	Ping SDK for iOS	4.3.0	Minor
2023-JUL-31	Ping SDK for Android	4.2.0	Minor
2023-SEP-11	Token Vault	4.2.0	Minor
2023-SEP-11	Ping SDK for JavaScript	4.2.0	Minor
2023-JUL-31	Ping SDK for Android	4.1.0	Minor
2023-JUL-28	Ping SDK for iOS	4.1.0	Minor
2023-JUL-24	Token Vault	4.1.2	Major
2023-JUL-20	Ping SDK for JavaScript	4.1.2	Patch
2023-JUL-17	Login Widget	1.1.0	Minor
2023-JUN-29	Ping SDK for JavaScript	4.1.1	Patch
2023-JUN-09	Ping SDK for iOS	4.0.0	Major
2023-MAY-30	Ping SDK for Android	4.0.0	Major
2023-MAY-23	Ping SDK for JavaScript	4.0.0	Major
2023-APR-18	Login Widget	1.0.0	Major

Release date	Platform	SDK version	Release type[1]
2022-NOV-15	Ping SDK for iOS	3.4.1	Patch
2022-OCT-10	Ping SDK for JavaScript	3.4.0	Minor
2022-SEP-29	Ping SDK for Android	3.4.0	Minor
2022-SEP-22	Ping SDK for iOS	3.4.0	Minor
2022-JUN-22	Ping SDK for Android	3.3.3	Patch

Release date	Platform	SDK version	Release type[1]
2022-JUN-21	Ping SDK for Android	3.3.2	Patch
2022-JUN-20	Ping SDK for iOS	3.3.2	Patch
2022-JUN-08	Ping SDK for iOS	3.3.1	Patch
2022-MAY-19	Ping SDK for iOS	3.3.0	Minor
2022-MAY-18	Ping SDK for Android	3.3.0	Minor
2022-APR-25	Ping SDK for JavaScript	3.3.0	Minor
2022-JAN-27	Ping SDK for iOS	3.2.0	Minor
2022-JAN-26	Ping SDK for Android	3.2.0	Minor

Release date	Platform	SDK version	Release type[1]
2021-NOV-17	Ping SDK for iOS	3.1.1	Patch
2021-OCT-28	Ping SDK for Android	3.1.2	Patch
2021-SEP-25	Ping SDK for iOS	3.1.0	Minor
2021-SEP-09	Ping SDK for Android	3.1.1	Patch
2021-MAY-24	All	3.0.0	Major

Release date	Platform	SDK version	Release type[1]
2020-DEC-18	All	2.2.0	Minor
2020-AUG-21	All	2.1.0	Minor
2020-JUN-30	All	2.0.0	Major

Release date	Platform	SDK version	Release type[1]
2019-DEC-10	All	GA.12.10.2019	Technology Preview
2019-OCT-21	All	Beta.10.21.2019	Technology Preview

[1] For details about the scope of expected changes for different release types, see Interface stability.

What's New

Subscribe to get automatic updates:

- Ping SDKs Changelog RSS feed
- Ping SDKs Changelog email notifications

> **Important**
>
> SDKs Renamed
> Prior to November 2024, the **Ping SDKs** were known as the **ForgeRock SDKs**.

Latest updates

SDK for Android 4.8.1 released NEW

25 June, 2025

A new version of the ForgeRock SDK for Android is now available with improvements and fixes over earlier versions:

- Improved performance by adding caching for KeyStore, Cipher, and Symmetric Key encryption and decryption.
- Added a `strongBoxPreferred=false` parameter to allow conditional use of StrongBox for key storage.

  Learn more in Preventing the Keystore System from using StrongBox.

Full changelog

SDK for Android 4.8.0 released

16 May, 2025

A new version of the ForgeRock SDK for Android is now available with improvements and fixes over earlier versions:

- Added the ability to update the Firebase Cloud Messaging (FCM) device token for existing devices registered for push notifications.

  Learn more in Updating device tokens for existing accounts.

- Added support for returning WebAuthn authenticator information in the updated WebAuthn authentication and registration callbacks introduced in PingAM and PingOne Advanced Identity Cloud.

  Learn more in Accessing WebAuthn authenticator information.

Full changelog

SDK for iOS 4.8.0 released NEW

16 May, 2025

A new version of the ForgeRock SDK for iOS is now available with improvements and fixes over earlier versions:

- Added the ability to update the Apple Push Notification Service (APNs) device token for existing devices registered for push notifications.

  Learn more in Updating device tokens for existing accounts.

- Added support for returning WebAuthn authenticator information in the updated WebAuthn authentication and registration callbacks introduced in PingAM and PingOne Advanced Identity Cloud.

  Learn more in Accessing WebAuthn authenticator information.

Full changelog

SDK for JavaScript 4.8.0 released NEW

16 May, 2025

A new version of the SDK for JavaScript is now available with improvements and fixes over earlier versions:

- Added a flag to skip immediately to the OAuth 2.0 flow rather than attempting to get tokens without redirecting.

  Learn more in Configure JavaScript apps for OIDC login.

- Added support for signing out of PingOne by using an ID token.

Full changelog

DaVinci Client for Android 1.1.0 released NEW

15 April, 2025

A new version of the DaVinci Client for Android is now available with improvements and fixes over earlier versions:

- Added social sign on with supported external IDPs.

  Learn more in Set up social sign on with external IDPs.

- Added `Accept-Language` header customization to support localization.

  Learn more in Localizing the user interface.

- Added support for additional PingOne Form fields.

  Learn more in Supported PingOne fields and collectors.

Full changelog

DaVinci Client for iOS 1.1.0 released NEW

15 April, 2025

A new version of the DaVinci Client for iOS is now available with improvements and fixes over earlier versions:

- Added social sign on with supported external IDPs.

Learn more in Set up social sign on with external IDPs.

- Added `Accept-Language` header customization to support localization.

  Learn more in Localizing the user interface.

- Added support for additional PingOne Form fields.

  Learn more in Supported PingOne fields and collectors.

- Added support for Swift 6.

Full changelog

DaVinci Client for JavaScript 1.1.0 released NEW

15 April, 2025

A new version of the DaVinci Client for JavaScript is now available with improvements and fixes over earlier versions:

- Added social sign on with supported external IDPs.

  Learn more in Set up social sign on with external IDPs.

- Added middleware support to alter `Accept-Language` header to support localization.

  Learn more in Localizing the user interface.

- Added support for additional PingOne Form fields.

  Learn more in Supported PingOne fields and collectors.

Full changelog

SDK for Android 4.7.0 released NEW

11 February, 2025

A new version of the ForgeRock SDK for Android is now available with improvements and fixes over earlier versions:

- Added support for user profile self-service.

  Learn more in Set up user profile self service.

- Added support for managing registered devices.

  Learn more in Set up registered device self service.

- Added support for signing-out of PingOne with an ID token.

Full changelog

SDK for iOS 4.7.0 released

11 February, 2025

A new version of the ForgeRock SDK for iOS is now available with improvements and fixes over earlier versions:

- Added support for user profile self-service.

  Learn more in Set up user profile self service.

- Added support for managing registered devices.

  Learn more in Set up registered device self service.

- Added support for signing-out of PingOne with an ID token.

Full changelog

SDK for JavaScript 4.7.0 released NEW

11 February, 2025

A new version of the SDK for JavaScript is now available with improvements and fixes over earlier versions:

Added

- Added support for managing registered devices.

  Learn more in Set up registered device self service.

Changed

- Prioritized `displayName` field over `userName` when saving a WebAuthn or passkey to an account. Previously the SDK displayed a UUID for saved credentials rather than the user's name.

Full changelog

DaVinci client 1.0.0 released

16 December, 2024

The first version of the DaVinci client for Android, iOS and JavaScript is now available.

> **Note**
>
> The Ping SDK DaVinci clients are constantly evolving to meet your business needs.
> Check back from time to time on latest updates and enhancements.

- Supports the **Custom HTML Template** capability of the **HTTP Connector**.
- Supports the following fields:
  - Text field
  - Password field
  - Submit button
  - Flow button

To learn more, refer to Supported DaVinci fields.

- Follow the DaVinci client tutorials to quickly setup a demo app to connect to your DaVinci flows.
- Read how to configure the DaVinci client to leverage DaVinci flows in your native or single-page apps.

Full changelog

SDK for Android 4.6.0 released

17 October, 2024

A new version of the ForgeRock SDK for Android is now available with improvements and fixes over earlier versions:

- Added the ability to customize how the ForgeRock SDK stores tokens and data.

  To learn more, refer to Customizing storage.
- Added support for Android App Links that use the http/https scheme for redirect URIs in centralized login apps.
- Added support for Android 15.
- Added support for the PingOne Protect Marketplace nodes.

Full changelog

SDK for iOS 4.6.0 released

17 October, 2024

A new version of the ForgeRock SDK for iOS is now available with improvements and fixes over earlier versions:

- Added support for the PingOne Protect Marketplace nodes.
- Exposed the realm, success URL and failure URL values within `Token`.
- Added client-side support for the upcoming `ReCaptchaEnterpriseCallback` callback.

Full changelog

SDK for JavaScript 4.6.0 released

17 October, 2024

A new version of the ForgeRock SDK for JavaScript is now available with improvements and fixes over earlier versions:

- Added centralized login support for PingFederate servers.
- Added client-side support for the upcoming `ReCaptchaEnterpriseCallback` callback.
- Added support for the PingOne Protect Marketplace nodes.

Full changelog

SDK for Android 4.5.0 released

9 July, 2024

A new version of the ForgeRock SDK for Android is now available with improvements and fixes over earlier versions:

- Added support for signing off from PingOne to the centralized login flow.

  To learn more, follow the Android tutorial for PingOne.

- Added the ability to dynamically configure the SDK by collecting values from a PingOne or server's OpenID Connect `.well-known` endpoint.

  To learn more, refer to Using the .well-known endpoint for dynamic configuration.

Full changelog

SDK for iOS 4.5.0 released

9 July, 2024

A new version of the ForgeRock SDK for iOS is now available with improvements and fixes over earlier versions:

- Added support for signing off from PingOne when using the centralized login flow with OAuth 2.0.

  To learn more, follow the iOS tutorial for PingOne.

- Added the ability to dynamically configure the SDK by collecting values from the server's OpenID Connect `.well-known` endpoint.

  To learn more, refer to Using the .well-known endpoint for dynamic configuration.

Full changelog

Ping (ForgeRock) Login Widget 1.3.0 released `NEW`

30 May, 2024

A new version of the ForgeRock SDK for JavaScript is now available with improvements and fixes over earlier versions:

- Added support for integration with PingOne Protect.
- Added the name of the device to the recovery codes page.

Full changelog

SDK for JavaScript 4.4.2 released

15 May, 2024

A new version of the ForgeRock SDK for JavaScript is now available with improvements and fixes over earlier versions:

- Added a `logoutRedirectUri` parameter to the `FRUser.logout()` method.

  Add the parameter to invoke a redirect flow, for revoking tokens and ending sessions created by a PingOne server.

To learn more, follow the JavaScript tutorial for PingOne.

- Added a `platformHeader` configuration property to control whether the SDK adds the `X-Requested-Platform` header to all outgoing connections.

Full changelog

SDK for iOS 4.4.1 released

25 April, 2024

A new version of the ForgeRock SDK for iOS is now available with improvements and fixes over earlier versions:

- Added privacy manifest files to ForgeRock SDK for iOS modules.

Full changelog

SDK for iOS 4.4.0 released

4 April, 2024

A new version of the ForgeRock SDK for iOS is now available with improvements and fixes over earlier versions:

- Added a new module for future integration with PingOne Protect.

  Learn more at Integrate with PingOne Protect for risk evaluations.

- Added an interface for customizing the biometric UI prompts when device binding or signing.

  Learn more at Bind and verify devices.

- Added support for the `TextInput` callback.

Full changelog

SDK for Android 4.4.0 released

28 March, 2024

A new version of the ForgeRock SDK for Android is now available with improvements and fixes over earlier versions:

- Added a new module for future integration with PingOne Protect.

  Learn more at Integrate with PingOne Protect for risk evaluations.

- Added an interface for customizing the biometric UI prompts when device binding or signing.

  Learn more at Bind and verify devices.

- Added support for the `TextInput` callback.

Full changelog

SDK for JavaScript 4.4.0 released

13 March, 2024

A new version of the ForgeRock SDK for JavaScript is now available with improvements and fixes over earlier versions:

- Added a new module for future integration with PingOne Protect.
- Added the ability to include the supplied device name when displaying recovery codes.

  Learn more at Using the device name.
- Added the ability to use values from an OpenID Connect `.well-known` URL to automatically configure the SDK paths.

  This simplifies using the SDKs with OIDC-compliant identity providers, such as **PingOne**.

  For more information, refer to the ForgeRock SDK for JavaScript PingOne tutorial.

Full changelog

SDK for Android 4.3.1 released

9 February, 2024

A new version of the ForgeRock SDK for Android is now available with improvements and fixes over earlier versions:

- Fixed an SDK crash during device binding on Android 9 devices.

Full changelog

Ping (ForgeRock) Login Widget 1.2.1 released

8 January, 2024

A new version of the Ping (ForgeRock) Web Login Framework is now available with improvements and fixes over earlier versions:

- Support for CAPTCHA nodes.

Full changelog

SDK for JavaScript 4.3.0 released

4 January, 2024

A new version of the ForgeRock SDK for JavaScript is now available with improvements and fixes over earlier versions:

- Added ability to override default prefix string given to storage keys.

  For more information, refer to `prefix` in the ForgeRock SDK for JavaScript Properties.
- Added an `FRQRCode` utility class to determine if a step has a QR code and handle the data to display.

  For more information, refer to Set up QR code handling.

Full changelog

SDK for Android 4.3.0 released

28 December, 2023

A new version of the ForgeRock SDK for Android is now available with improvements and fixes over earlier versions:

- Added ability to customize cookie headers in outgoing requests from the SDK.
- Added ability to add custom claims when verifying signatures from bound devices.
- Added client-side support for the upcoming `AppIntegrity` callback.

Full changelog

SDK for iOS 4.3.0 released

28 December, 2023

A new version of the ForgeRock SDK for iOS is now available with improvements and fixes over earlier versions:

- Added client-side support for the upcoming `AppIntegrity` callback.
- Added a new `ephemeralAuthSession` browser type for iOS13 and later.
- Added `iat` and `nbf` claims to the device binding JWS payload.
- Added ability to insert custom claims when performing device signing verification.
- Updated the detection of Jailbreak status.

Full changelog

SDK for Android 4.2.0 released

3 October, 2023

A new version of the ForgeRock SDK for Android is now available with improvements and fixes over earlier versions:

- Added Gradle 8 and JDK 17 support.
- Added Android 14 support.
- Added verification of key pairs during device binding enrollment by using Google Key Attestation.
- Added *issued at* (`iat`) and *not before* (`nbf`) claims to JSON Web tokens used for device binding and signing verification.

Full changelog

Token Vault 4.2.0 released

11 September, 2023

A new version of the Token Vault is now available with improvements and fixes over earlier versions:

- Added a requirement to declare a list of URLs in the Token Vault Proxy configuration. These generate an allowlist of origins to which the proxy can forward requests.

Full changelog

SDK for JavaScript 4.2.0 released

11 September, 2023

A new version of the ForgeRock SDK for JavaScript is now available with improvements and fixes over earlier versions:

- Added a `logLevel` configuration property to specify the level of logging the SDK performs.
- Added a `customLogger` configuration property to specify a replacement for the native `console.log` that the SDK uses by default.

  For example, you could write a replacement that captures SDK log output to services such as Relic or Rocket.

Full changelog

SDK for Android 4.1.0 released

31 July, 2023

A new version of the ForgeRock SDK for Android is now available with improvements and fixes over earlier versions:

- Added support for interceptors in the authenticator module
- Added an interface for refreshing access tokens
- Added support for policy advice from IG in JSON format

Full changelog

SDK for iOS 4.1.0 released

28 July, 2023

A new version of the ForgeRock SDK for iOS is now available with improvements and fixes over earlier versions:

- Added support for interceptors in the authenticator module.
- Added support for `mfauth` deep links in the authenticator sample app.
- Added an interface for refreshing access tokens.
- Added support for policy advice from IG in JSON format.

Full changelog

Token Vault 4.1.2 released

24 July, 2023

Token Vault provides an additional layer of security for storing and using OAuth 2.0 and OpenID Connect 1.0 tokens in your JavaScript single-page applications (SPAs).

Implemented as a plugin for the ForgeRock SDK for JavaScript, Token Vault provides a feature called *origin isolation*.

Full changelog

SDK for JavaScript 4.1.2 released

20 July, 2023

A new version of the ForgeRock SDK for JavaScript is now available with improvements and fixes over earlier versions:

- Added support in preparation for upcoming Token Vault.

Full changelog

Ping (ForgeRock) Login Widget 1.1.0 released

17 July, 2023

A new version of the Ping (ForgeRock) Web Login Framework is now available with improvements and fixes over earlier versions:

- Support for device profiling callbacks (`DeviceProfileCallback`)
- Support for web authentication (WebAuthn) journeys and trees.

Full changelog

SDK for JavaScript 4.1.1 released

29 June, 2023

A new version of the ForgeRock SDK for JavaScript is now available with improvements and fixes over earlier versions:

- Added support in the HTTPClient for receiving transactional authorization advice in JSON format.

Full changelog

SDK for iOS 4.0.0 released

9 June, 2023

A new version of the ForgeRock SDK for iOS is now available with improvements and fixes over earlier versions:

- Added support for Passkeys.
- Added the ability to provide a device name when registering WebAuthN devices.
- Added support for enforcing policies in the Authenticator SDK.
- Added SwiftUI quick start sample code.

Full changelog

> **Important**
>
> This release contains changes that could affect the functionality of your app.
> Refer to Incompatible changes.

SDK for Android 4.0.0 released

30 May, 2023

A new version of the ForgeRock SDK for Android is now available with improvements and fixes over earlier versions:

- Upgraded the Google Fido client to support Passkeys.
- Added the ability to provide a device name when registering WebAuthN devices.
- Added support for enforcing policies in the Authenticator SDK.

Full changelog

> **Important**
>
> This release contains changes that could affect the functionality of your app.
> Refer to Incompatible changes.

SDK for JavaScript 4.0.0 released

23 May, 2023

A new version of the ForgeRock SDK for JavaScript is now available with improvements and fixes over earlier versions:

- Added the ability to provide a device name when registering WebAuthN devices.
- Updated ESModule (ESM) bundle.
- Updated tags in the GitHub repo to be prefixed with the package name. For example, `javascript-sdk-${tag}`.

Full changelog

> **Important**
>
> This release contains changes that could affect the functionality of your app.
> Refer to the following pages for details:
>
> - Incompatible changes
> - Deprecations

Ping (ForgeRock) Login Widget 1.0.0 released

18 April, 2023

The Ping (ForgeRock) Login Widget is an all-inclusive UI component to help you add authentication, user registration, and other self-service journeys into your web applications.

The Ping (ForgeRock) Login Widget uses the Ping SDK for JavaScript internally, and adds a user interface and state management. This rendering layer helps eliminate the need to develop and maintain the UI components for providing complex authentication experiences.

Full changelog

Full changelogs

Ping SDK for Android

Ping SDK for iOS

Ping SDK for JavaScript

DaVinci client

Ping (ForgeRock) Login Widget

Token Vault

Legacy releases

Release timeline

Key:

N.N.N = Latest version

Release date	Platform	SDK version	Release type[1]
2025-MAY-16	Ping SDK for Android	4.8.0	Minor
2025-MAY-16	Ping SDK for JavaScript	4.8.0	Minor
2025-MAY-16	Ping SDK for iOS	4.8.0	Minor
2025-APR-15	DaVinci client for Android	1.1.0	Minor
2025-APR-15	DaVinci client for iOS	1.1.0	Minor
2025-APR-15	DaVinci client for JavaScript	1.1.0	Minor
2025-FEB-11	Ping SDK for Android	4.7.0	Minor
2024-FEB-11	Ping SDK for iOS	4.7.0	Minor
2025-FEB-11	Ping SDK for JavaScript	4.7.0	Minor
2024-DEC-16	DaVinci client	1.0.0	Major
2024-OCT-17	Ping SDK for Android	4.6.0	Minor
2024-OCT-17	Ping SDK for iOS	4.6.0	Minor
2024-OCT-17	Ping SDK for JavaScript	4.6.0	Minor
2024-JUL-11	Ping SDK for Android	4.5.0	Minor
2024-JUL-11	Ping SDK for iOS	4.5.0	Minor
2024-JUN-05	Login Widget	1.3.0	Minor
2024-MAY-15	Ping SDK for JavaScript	4.4.2	Patch
2024-APR-25	Ping SDK for iOS	4.4.1	Patch
2024-APR-04	Ping SDK for iOS	4.4.0	Minor
2024-MAR-28	Ping SDK for Android	4.4.0	Minor
2024-MAR-13	Ping SDK for JavaScript	4.4.0	Minor

Release date	Platform	SDK version	Release type[1]
2024-FEB-09	Ping SDK for Android	4.3.1	Patch
2024-JAN-08	Login Widget	1.2.1	Minor
2024-JAN-04	Ping SDK for JavaScript	4.3.0	Minor
2023-DEC-28	Ping SDK for Android	4.3.0	Minor
2023-DEC-28	Ping SDK for iOS	4.3.0	Minor
2023-JUL-31	Ping SDK for Android	4.2.0	Minor
2023-SEP-11	Token Vault	4.2.0	Minor
2023-SEP-11	Ping SDK for JavaScript	4.2.0	Minor
2023-JUL-31	Ping SDK for Android	4.1.0	Minor
2023-JUL-28	Ping SDK for iOS	4.1.0	Minor
2023-JUL-24	Token Vault	4.1.2	Major
2023-JUL-20	Ping SDK for JavaScript	4.1.2	Patch
2023-JUL-17	Login Widget	1.1.0	Minor
2023-JUN-29	Ping SDK for JavaScript	4.1.1	Patch
2023-JUN-09	Ping SDK for iOS	4.0.0	Major
2023-MAY-30	Ping SDK for Android	4.0.0	Major
2023-MAY-23	Ping SDK for JavaScript	4.0.0	Major
2023-APR-18	Login Widget	1.0.0	Major

Release date	Platform	SDK version	Release type[1]
2022-NOV-15	Ping SDK for iOS	3.4.1	Patch
2022-OCT-10	Ping SDK for JavaScript	3.4.0	Minor
2022-SEP-29	Ping SDK for Android	3.4.0	Minor
2022-SEP-22	Ping SDK for iOS	3.4.0	Minor
2022-JUN-22	Ping SDK for Android	3.3.3	Patch

Release date	Platform	SDK version	Release type[1]
2022-JUN-21	Ping SDK for Android	3.3.2	Patch
2022-JUN-20	Ping SDK for iOS	3.3.2	Patch
2022-JUN-08	Ping SDK for iOS	3.3.1	Patch
2022-MAY-19	Ping SDK for iOS	3.3.0	Minor
2022-MAY-18	Ping SDK for Android	3.3.0	Minor
2022-APR-25	Ping SDK for JavaScript	3.3.0	Minor
2022-JAN-27	Ping SDK for iOS	3.2.0	Minor
2022-JAN-26	Ping SDK for Android	3.2.0	Minor

Release date	Platform	SDK version	Release type[1]
2021-NOV-17	Ping SDK for iOS	3.1.1	Patch
2021-OCT-28	Ping SDK for Android	3.1.2	Patch
2021-SEP-25	Ping SDK for iOS	3.1.0	Minor
2021-SEP-09	Ping SDK for Android	3.1.1	Patch
2021-MAY-24	All	3.0.0	Major

Release date	Platform	SDK version	Release type[1]
2020-DEC-18	All	2.2.0	Minor
2020-AUG-21	All	2.1.0	Minor
2020-JUN-30	All	2.0.0	Major

Release date	Platform	SDK version	Release type[1]
2019-DEC-10	All	GA.12.10.2019	Technology Preview
2019-OCT-21	All	Beta.10.21.2019	Technology Preview

[1] For details about the scope of expected changes for different release types, see Interface stability.

Ping SDK for Android changelog

Subscribe to get automatic updates:

- Ping SDKs Changelog RSS feed
- Ping SDKs Changelog email notifications

Ping SDK for Android 4.8.1

June 25, 2025

Added

- Added caching for KeyStore, Cipher, and Symmetric Key encryption and decryption, improving performance. [SDKS-4090]
- Added a `strongBoxPreferred=false` parameter to allow conditional use of StrongBox for key storage. [SDKS-4090]

Ping SDK for Android 4.8.0

May 16, 2025

Added

- Added support for returning WebAuthn authenticator information in the updated WebAuthn authentication and registration callbacks introduced in PingAM and PingOne Advanced Identity Cloud. [SDKS-3843]
- Added the ability to update the Firebase Cloud Messaging (FCM) device token for existing devices registered for push notifications. [SDKS-3684]

Updated

- Improved logging for errors and warning exceptions. [SDKS-3990]

Fixed

- Fixed an issue causing a crash when the killing the app process in the background during the OIDC (centralized login) flow. [SDKS-3993]

Ping SDK for Android 4.7.0

February 11, 2025

Added

- Added support for user profile self-service. [SDKS-3408]
- Added support for managing registered devices.
- Added support for signing-out of PingOne with an ID token. [SDKS-3423]

Updated

- Improved compatibility with certain devices by implementing a fallback mechanism that uses asymmetric key generation if symmetric key generation in the AndroidKeyStore fails. [SDKS-3467]

Fixed

- Fixed an issue that caused duplicate PUSH notifications in the Authenticator module. [SDKS-3533]

Ping SDK for Android 4.6.0

October 17, 2024

Added

- Added support for Android 15. [SDKS-3098]
- Added the ability to customize how the SDK stores tokens and data. [SDKS-3378]
- Added support for Android App Links that use the http/https scheme for redirect URIs in centralized login apps. [SDKS-3433]
- Added support for the PingOne Protect Marketplace nodes. [SDKS-3297]
- Exposed the realm and success URL values within `SSOToken`. [SDKS-3351]
- Added client-side support for the upcoming `ReCaptchaEnterpriseCallback` callback. [SDKS-2499]

Updated

- Updated the SDK to ignore any type 4 `TextOutputCallback` callbacks, as these contain JavaScript that Android cannot execute. [SDKS-3227]

Fixed

- Fixed a potential `ServiceConnection` leak in `CustomTabManager`. [SDKS-3346]

Ping SDK for Android 4.5.0

July 12, 2024

Added

- Added support for signing off from PingOne to the centralized login flow. [SDKS-3020]
- Added the ability to dynamically configure the SDK by collecting values from the server's OpenID Connect `.well-known` endpoint. [SDKS-3022]

Fixed

- Resolved security vulnerability warnings related to the `commons-io-2.6.jar` and `bcprov-jdk15on-1.68.jar` libraries. [SDKS-3072, SDKS-3073]
- Fixed a `NullPointerException` in the centralized login flow. [SDKS-3079]
- Improved multi-threaded performance when caching access tokens. [SDKS-3104]
- Synchronized the encryption and decryption block to avoid keystore crashes. [SDKS-3199]
- Fixed an issue related to handling `HiddenValueCallback` if `isMinifyEnabled` is set to `true`. [SDKS-3201]
- Fixed an issue where device binding using an application PIN was failing when Arabic language was used. [SDKS-3221]

- Fixed an issue where browser sessions were not properly signed out when a non-default browser was used in centralized login. [SDKS-3276]
- Fixed an unexpected behavior in the authentication flow caused by `AppAuthConfiguration` settings being ignored during centralized login. [SDKS-3277]
- Fixed the `FRUser.revokeAccessToken()` method to not end the user's session during the centralized login flow. [SDKS-3282]

Ping SDK for Android 4.4.0

March 28, 2024

Added

- Added a new module for integration with PingOne Protect. [SDKS-2900]
- Added support for the `TextInput` callback. [SDKS-545]
- Added an interface for customizing the biometric UI prompts when device binding or signing. [SDKS-2991]
- Added `x-requested-with: forgerock-sdk` and `x-requested-platform: android` immutable HTTP headers to each outgoing request. [SDKS-3033]

Fixed

- Addressed a null pointer exception during centralized login by using `ActivityResultContract` in place of the deprecated `onActivityResult` method. [SDKS-3079]
- Addressed `nimbus-jose-jwt:9.25` library security vulnerability (CVE-2023-52428). [SDKS-2988]

Ping SDK for Android 4.3.1

February 9, 2024

Fixed

- Fixed an issue where the SDK crashes during device binding on Android 9 devices. [SDKS-2948]

Ping SDK for Android 4.3.0

December 28, 2023

Added

- Added ability to customize cookie headers in outgoing requests from the SDK. [SDKS-2780]
- Added ability to add custom claims when verifying signatures from bound devices. [SDKS-2787]
- Added client-side support for the upcoming `AppIntegrity` callback. [SDKS-2631]

Updated

- The SDK now uses auth-per-use keys for Device Binding. [SDKS-2797]
- Improved handling of WebAuthn cancellations. [SDKS-2819]

- The `forgerock_url`, `forgerock_realm`, and `forgerock_cookie_name` parameters are now mandatory when dynamically configuring the SDK. [SDKS-2782]
- Addressed `woodstox-core:6.2.4` library security vulnerability CVE-2022-40152. [SDKS-2751]

Ping SDK for Android 4.2.0

October 3, 2023

Added

- Added Gradle 8 and JDK 17 support. [SDKS-2451]
- Added Android 14 support. [SDKS-2636]
- Added verification of key pairs during device binding enrollment by using Google Key Attestation. [SDKS-2412]
- Added *issued at* (`iat`) and *not before* (`nbf`) claims to JSON Web tokens used for device binding and signing verification. [SDKS-2747]

Ping SDK for Android 4.1.0

July 31, 2023

Added

- Added support for interceptors in the authenticator module. [SDKS-2544]
- Added an interface for refreshing access tokens. [SDKS-2567]
- Added support for policy advice from IG in JSON format. [SDKS-2240]

Fixed

- Fixed an issue with parsing the `issuer` value in the URI provided by the combined MFA registration node. [SDKS-2542]
- Added an error message about duplicated accounts while using the combined MFA registration node. [SDKS-2627]
- Fixed an issue that caused loss of WebAuthn credentials when upgrading the SDK from 4.0.0-beta4 to newer versions. [SDKS-2576]

Ping SDK for Android 4.0.0

May 30, 2023

Added

- Upgraded the Google Fido client to support Passkeys. [SDKS-2243]
- Added the `FRWebAuthn` interface to remove WebAuthn reference keys. [SDKS-2272]
- Added an interface to specify a device name during WebAuthn registration. [SDKS-2296]
- Added `DeviceBinding` callback support. [SDKS-1747]
- Added `DeviceSigningVerifier` callback support. [SDKS-2022]
- Added support for combined MFA registration in the Authenticator SDK. [SDKS-1972]

- Added support for enforcing policies in the Authenticator SDK. [SDKS-2166]

Fixed

- Fixed WebAuthn authentication on devices that use a full-screen biometric prompt. [SDKS-2340]
- Fixed functionality of the `NetworkCollector` method. [SDKS-2445]

Incompatible changes

- Removed support for native single sign-on (SSO).
- Changed the signature for a number of methods.

For more information, refer to Incompatible changes.

Ping SDK for Android 3.4.0

September 29, 2022

Added

- Dynamic SDK Configuration. [SDKS-1759]
- Android 13 support. [SDKS-1944]

Changed

- Changed activity type used as parameter in `PushNotification.accept`. [SDKS-1968]
- Updated deserialization of objects to use a class allowlist to prevent access to untrusted data. [SDKS-1818]
- Updated the `Authenticator` module and sample app to handle the new `POST_NOTIFICATIONS` permission in Android 13. [SDKS-2033]
- Fixed an issue where the `DefaultTokenManager` was not caching the `AccessToken` in memory upon retrieval from Shared Preferences. [SDKS-2066]
- Deprecated the `forgerock_enable_cookie` configuration. [SDKS-2069]
- Align `forgerock_logout_endpoint` configuration name with the Ping SDK for iOS. [SDKS-2085]
- Allow leading slash on custom endpoint path. [SDKS-2074]
- Fixed bug where the `state` parameter value was not being verified upon calling the `Authorize` endpoint. [SDKS-2078]

Ping SDK for Android 3.3.3

June 22, 2022

Changed

- Updated the version of the `com.squareup.okhttp3` library in the SDK to 4.10.0 [SDKS-1957]

Ping SDK for Android 3.3.2

June 21, 2022

Added

- Interface for log management [SDKS-1864]

Ping SDK for Android 3.3.0

May 18, 2022

Added

- Support SSL pinning [SDKS-80]
- Restore session token when it is out of sync with the session token that bound with the access token [SDKS-1664]
- Session token should be included in the header instead of request parameter for `/authorize` endpoint [SDKS-1670]
- Support to broadcast logout event to clear application tokens when user logout the app [SDKS-1663]
- Obtain timestamp from new `PushNotification` payload [SDKS-1666]
- Add new payload attributes to the `PushNotification` [SDKS-1776]
- Allow processing of push notifications without device token [SDKS-1844]

Fixed

- Dispose `AuthorizationService` when no longer required [SDKS-1636]
- Authenticator sample app crash after scanning push mechanism [SDKS-1454]

Ping SDK for Android 3.2.0

January 26, 2022

Features

- Google Sign-In Security Enhancement.
- Fix for WebAuthn Registration & Authentication prompt.

Ping SDK for Android 3.1.2

October 28, 2021

Features

- Disable native SSO when the SDK fails to access the Android AccountManager.

Ping SDK for Android 3.1.1

September 09, 2021

Features

- Support for Android 12.
- Unlocked device is not required for data decryption.

- Introduced `FRLifecycle` interface and exposed interfaces to allow custom native SSO implementation.

Ping SDK for iOS changelog

Subscribe to get automatic updates:

- Ping SDKs Changelog RSS feed
- Ping SDKs Changelog email notifications

Ping SDK for iOS 4.8.0

May 16, 2025

Added

- Added the ability to update the Apple Push Notification Service (APNs) device token for existing devices registered for push notifications. [SDKS-3684]
- Added support for returning WebAuthn authenticator information in the updated WebAuthn authentication and registration callbacks introduced in PingAM and PingOne Advanced Identity Cloud. [SDKS-3842]

Updated

- Upgraded ReCAPTCHA Enterprise to version 18.7.0 (from 18.6.0) [SDKS-3927]

Fixed

- Resolved an issue where updating device biometrics didn't enforce device re-binding as expected. [SDKS-3963]
- Corrected the missing `PingProtect` scheme. [SDKS-3856]
- Resolved a race condition in the device network collector that prevented `NetworkReachabilityMonitor` from completing. [SDKS-3827]

Ping SDK for iOS 4.7.0

February 11, 2025

Added

- Added support for user profile self-service. [SDKS-3409]
- Added support for managing registered devices.
- Added support for signing-out of PingOne with an ID token. [SDKS-3424]

Updated

- Updated jailbreak detectors to reduce false-positive detections. [SDKS-3693]

Fixed

- Fixed an issue that caused duplicate PUSH notifications in the Authenticator module. [SDKS-3533]

Ping SDK for iOS 4.6.0

October 17, 2024

Added

- Added support for the PingOne Protect Marketplace nodes. [SDKS-3296]
- Exposed the realm, success URL, and failure URL values within `Token`. [SDKS-3352]
- Added client-side support for the upcoming `ReCaptchaEnterpriseCallback` callback. [SDKS-3324]
- Added support for Device Binding in iOS simulators, by setting **Authentication Type** in the Device Binding node to **None**.

Updated

- Updated the SDK to skip any type 4 `TextOutputCallback` callbacks, as these contain JavaScript that iOS cannot execute. [SDKS-3226]
- Made `PolicyAdviceCreator` public. [SDKS-3349]

Fixed

- Fixed missing UIKit import issue for SPM. [SDKS-3348]
- Fixed an issued preventing SSL pinning from working with root certificates. [SDKS-3334]
- Fixed a build failure because `FRCore.swiftmodule` is not built for `arm64`. [SDKS-3347]

Ping SDK for iOS 4.5.0

July 12, 2024

Added

- Added support for signing off from PingOne when using the centralized login flow with OAuth 2.0. [SDKS-3021]
- Added the ability to dynamically configure the SDK by collecting values from the server's OpenID Connect `.well-known` endpoint. [SDKS-3023]

Fixed

- Fixed issue causing SSL pinning configuration to be ignored in `FRURLProtocol` class. [SDKS-3239]
- Removed scope validation from `AccessToken` initialization. [SDKS-3305]

Ping SDK for iOS 4.4.1

April 25, 2024

Added

- Added privacy manifest files to Ping SDK for iOS modules. [SDKS-3086]

Updated

- Updated PingOne Signals (Protect) SDK to version 5.2.3. [SDKS-3086]

- Updated Google SDK to version 7.1.0. [SDKS-3086]

Changed

- Removed `storage` field from the `HardwareCollector` class. [SDKS-3086]

Ping SDK for iOS 4.4.0

April 4, 2024

Added

- Added a new module for integration with PingOne Protect. [SDKS-2901]
  - Added support for the `TextInput` callback. [SDKS-546]
  - Added an interface for customizing the biometric UI prompts when device binding or signing. [SDKS-2990]
  - Added `x-requested-with: forgerock-sdk` and `x-requested-platform: ios` immutable HTTP headers to each outgoing request. [SDKS-2997]

Changed

- Prevented the operation of device binding and signing features on simulators. [SDKS-2995]

Ping SDK for iOS 4.3.0

December 15, 2023

Added

- Added client-side support for the upcoming `AppIntegrity` callback. [SDKS-2630/SDKS-2761]
- Added a new `ephemeralAuthSession` browser type for iOS13 and later. [SDKS-2707]
- Added `iat` and `nbf` claims to the device binding JWS payload. [SDKS-2748]
- Added ability to insert custom claims when performing device signing verification. [SDKS-2788]

Fixed

- Fixed an issue where the `issuer` parameter was not properly parsed when using PingAM 7.2.x. [SDKS-2653]
- Fixed an issue related to inadequate cache control. [SDKS-2700]
- Fixed an issue when the `sfViewController` setting in centralized login had `entersReaderIfAvailable` set to `true`. [SDKS-2746]
- Fixed an issue with the device profile collector that affected phones with multiple sim cards in iOS 16.3 and earlier. [SDKS-2776]
- Fixed an issue with device binding API access levels. [SDKS-2886]
- Fixed an issue with removing a `userkey` from the local device repo. [SDKS-2887]

Updated

- Updated the detection of Jailbreak status. [SDKS-2796]

- Improved unit and end-to-end tests. [SDKS-2637]

Ping SDK for iOS 4.1.0

July 28, 2023

Added

- Added support for interceptors in the authenticator module. [SDKS-2545]
- Added support for `mfauth` deep links in the authenticator sample app. [SDKS-2524]
- Added an interface for refreshing access tokens. [SDKS-2563]
- Added support for policy advice from IG in JSON format. [SDKS-2239]

Fixed

- Fixed an issue with parsing the `issuer` value in the URI provided by the combined MFA registration node. [SDKS-2542]
- Added an error message about duplicated accounts while using the combined MFA registration node. [SDKS-2627]

Ping SDK for iOS 4.0.0

June 9, 2023 `major`

Added

- Added support for Passkeys. [SDKS-2140]
- Added `DeviceBinding` callback support. [SDKS-1748]
- Added `DeviceSigningVerifier` callback support. [SDKS-2023]
- Added support for combined MFA registration in the Authenticator SDK. [SDKS-1972]
- Added support for enforcing policies in the Authenticator SDK. [SDKS-2166]
- Added an interface for listing and deleting WebAuthn credentials from the device. [SDKS-2279]
- Added an interface to specify a device name during WebAuthn registration. [SDKS-2297]
- Added a SwiftUI quick start example. [SDKS-2405]

Fixed

- Added error message description to the `WebAuthnError` enum. [SDKS-2226]
- Updated the order of presenting the registered WebAuthN keys on the device. [SDKS-2251]
- Updated Facebook SDK version to 16.0.1. [SDKS-1839]
- Updated Google SDK version to 7.0.0. [SDKS-2426]

Incompatible changes

- Changed the signature for a number of methods.

For more information, refer to Incompatible changes.

Ping SDK for iOS 3.4.1

November 15, 2022

Changed

- Updated legacy encryption algorithm used for generation of cryptographic keys stored in `Secure Enclave` [SDKS-1994]
- Fixed an issue related to push notifications timeout [SDKS-2164]
- Fixed an unexpected error occurring during the decoding of some push notifications [SDKS-2199]

Ping SDK for iOS 3.4.0

September 22, 2022

Added

- Dynamic SDK Configuration [SDKS-1760]
- iOS 16 Support [SDKS-1932]

Changed

- Fixed build errors on Xcode 14 [SDKS-2073]
- Fixed bug where the `state` parameter value was not verified upon calling the `Authorize` endpoint [SDKS-2077]

Ping SDK for iOS 3.3.2

June 20, 2022

Added

- Interface for log management [SDKS-1863]

Changed

- Fixed memory leak in the `NetworkCollector` class [SDKS-1931]

Ping SDK for iOS 3.3.1

June 08, 2022

Added

- Add `PushType.biometric` support and `BiometricAuthentication` class for biometric authentication. Updated sample app to handle new Push types [SDKS-1865]

Changed

- Fixed the bug when refreshing the access token we return the old token [SDKS-1824]
- Fixed bug when multiple threads are trying to access the same resource in the `deviceCollector` and `ProfileCollector` [SDKS-1912]

Ping SDK for iOS 3.3.0

May 19, 2022

Added

- SSL pinning support [SDKS-1627]
- Obtain timestamp from new push notification payload [SDKS-1665]
- Add new payload attributes in the push notification [SDKS-1775]
- Apple Sign In enhancements to get user profile info [SDKS-1632]

Changed

- Remove "Accept: application/x-www-form-urlencoded" header from /authorize endpoint for GET requests [SDKS-1729]
- Remove `iPlanetDirectoryPro` (or session cookie name) from the query parameter, and inject it into the header instead [SDKS-1708]
- Fix issue when expired push notification displayed as "Approved" in the notification history list [SDKS-1491]
- Fix issues with registering TOTP accounts with invalid period [SDKS-1405]

Ping SDK for iOS 3.2.0

January 27, 2022

Changed

- Updated GoogleSignIn library to the latest version `6.1.0`.
- FRGoogleSignIn is now available through SPM.

Ping SDK for iOS 3.1.1

November 17, 2021

Features

- Added custom implementation for `HTTPCookie` for iOS 11+ devices, to support NSSecureCoding for storing cookies.
- Changed all instances of Archiving/Unarchiving to use NSSecureCoding.
- `SecuredKey` initializer now supports passing a Keychain accessibility flag.
- `SecuredKey` now has the same default Keychain accessibility flag as the KeychainService ".afterFirstUnlock".

Ping SDK for iOS 3.1.0

September 25, 2021

Features

- Fixed an issue where the MetadataCallback was overriding the stage property of a node.
- Fixed an issue which was affecting the centralized login feature.

- Various bug fixes and enhancements for the Authenticator SDK.

Ping SDK for JavaScript changelog

Subscribe to get automatic updates:

- Ping SDKs Changelog RSS feed
- Ping SDKs Changelog email notifications

Ping SDK for JavaScript 4.8.0

May 16, 2025 `minor`

Added

- Added a flag to skip immediately to the OAuth 2.0 flow rather than attempting to get tokens without redirecting. [SDKS-3866]
- Added support for signing out of PingOne by using an ID token. [SDKS-3757]

Changed

- Removed an unneeded call to the `/session` endpoint. [SDKS-3757]

Ping SDK for JavaScript 4.7.0

February 11, 2025 `minor`

Added

- Added a device client module to manage registered devices.

Changed

- Prioritized `displayName` field over `userName` when saving a WebAuthn or passkey to an account. Previously the SDK displayed a UUID for saved credentials rather than the user's name. [SDKS-3473]

Ping SDK for JavaScript 4.6.0

October 17, 2024 `minor`

Added

- Added centralized login support for PingFederate servers. [SDKS-3250]
- Added client-side support for the upcoming `ReCaptchaEnterpriseCallback` callback. [SDKS-3326]
- Added support for the PingOne Protect Marketplace nodes. [SDKS-3298]

Changed

- Refactored authorize URL utilities for upcoming DaVinci module. [SDKS-3183]
- Updated allowed message list to include PingFederate "requires consent" response. [SDKS-3478]

- Changed the PKCE utility to return a storage function.

Ping SDK for JavaScript 4.4.2

May 15, 2024 `patch`

Added

- Added a `logoutRedirectUri` parameter to the `FRUser.logout()` method.

  Add the parameter to invoke a redirect flow, for revoking tokens and ending sessions created by a PingOne server.

  To learn more, follow the JavaScript tutorial for PingOne.

- Added a `platformHeader` configuration property to control whether the SDK adds the `X-Requested-Platform` header to all outgoing connections.

Updated

- Updated the embedded PingOne Signals (Protect) SDK to the latest version.
- Updated the SDK to import the PingOne Signals (Protect) SDK dynamically and start it with a method call rather than on load.
- Updated the build system to use Vite.

Fixed

- Wrapped the PingOne Signals (Protect) SDK to protect it from being called when running server-side.

Ping SDK for JavaScript 4.4.0

March 13, 2024 `minor`

Added

- Added a new module for integration with PingOne Protect. [SDKS-2902]
- Added the ability to include the supplied device name when displaying recovery codes. [SDKS-2536]
- Added the ability to use the OpenID Connect `.well-known` endpoint to override the default path configuration. [SDKS-2966]

  This simplifies using the SDKs with OIDC-compliant identity providers, such as **PingOne**.

  For more information, refer to the Ping SDK for JavaScript PingOne tutorial.

  > **Note**
  >
  > The SDK is currently unable to revoke PingOne-issued OIDC tokens when using Firefox and Safari, due to third-party cookie restrictions.

- Added `StepOptions` type to the public API.

Fixed

- Fixed a naming collision when using `sessionStorage` for tokens, state, and PKCE data and performing centralized login. [SDKS-2945]

Ping SDK for JavaScript 4.3.0

January 4, 2024 `minor`

Added

- Added ability to override default prefix string given to storage keys.

  For more information, refer to `prefix` in the Ping SDK for JavaScript Properties.

- Added an `FRQRCode` utility class to determine if a step has a QR code and handle the data to display.

  For more information, refer to Set up QR code handling.

Fixed

- Fixed undefined `main` and `module` fields in package.json.

Ping SDK for JavaScript 4.2.0

September 11, 2023 `minor`

Added

- Added a `logLevel` configuration property to specify the level of logging the SDK performs.

  For more information, refer to About the default Ping SDK for JavaScript logger.

- Added a `customLogger` configuration property to specify a replacement for the native `console.log` that the SDK uses by default.

  For example, you could write a replacement that captures SDK log output to services such as Relic or Rocket.

  For more information, refer to Customize the Ping SDK for JavaScript logger .

Ping SDK for JavaScript 4.1.2

July 20, 2023 `patch`

Added

- Added support in preparation for upcoming Token Vault.

Fixed

- Fixed an issue with the `getTokens()` method failing if no parameters are provided and you perform certain down-leveling of code in the build process.

Ping SDK for JavaScript 4.1.1

June 29, 2023 `minor`

Added

- Added support in the HTTPClient for receiving transactional authorization advice in JSON format.

Changed

- Improved types when using strict mode with TypeScript.

Ping SDK for JavaScript 4.0.0

May 23, 2023 `major`

Added

- Added the ability to provide a device name when registering WebAuthN devices.

Changed

- Updated ESModule (ESM) bundle.
- Updated tags in the GitHub repo to be prefixed with the package name. For example, `javascript-sdk-${tag}`.
- Inserted a `prompt=none` parameter into OAuth 2.0 calls to the `/authorize` endpoint to prevent console error about frames.

Incompatible changes

- No longer provides Universal Module Definition (UMD) support
- Updated Policy types
- Removed duplicate modules

For more information, refer to Incompatible changes.

Deprecated

- JavaScript `support` configuration property deprecated.

For more information, refer to Deprecations.

Ping SDK for JavaScript 3.4.0

October 10, 2022 `minor`

Changed

- Fixed HTTP headers by capitalizing all header names
- Added support for `TextInput` callback
- Updated device profile collection code:
  - Added optional chaining to protect object checks in both browser and node environments
  - Changed usage of `window.crypto` to `globalThis.crypto` to improve compatibility

Ping SDK for JavaScript 3.3.0

April 25, 2022 `minor`

Added

- Added Angular sample app.
- Added token threshold feature.

DaVinci client changelog

Subscribe to get automatic updates:

- Ping SDKs Changelog RSS feed
- Ping SDKs Changelog email notifications

DaVinci client for Android 1.1.0

April 17, 2025

Added

- Added support for additional PingOne Form fields. [SDKS-3649]
  - Label
  - Checkbox
  - Dropdown
  - Combobox
  - Radio list
  - Flow link
- Added an `external-idp` module to support social sign on with supported external IDPs by using browser redirects. [SDKS-3662]

  Supported external IDPs:
  - Apple
  - Facebook
  - Google
- Added `Accept-Language` header to support localization. [SDKS-3622]
- Added ability to validate PingOne Form fields. [SDKS-3649]
- Added support for default values in PingOne Form fields. [SDKS-3649]
- Added an interface to access `ErrorNode` and validation errors. [SDKS-3649]

- Added a `browser` module. [SDKS-3662]
- Added dynamic environment switching in the test sample app. [SDKS-3642]

Fixed

- Fixed an issue affecting the global logger when configuring a logger in DaVinci client configuration. [SDKS-3616]

DaVinci client for iOS 1.1.0

April 17, 2025

Added

- Added support for additional PingOne Form fields. [SDKS-3671, SDKS-3672]
  - Label
  - Checkbox
  - Dropdown
  - Combobox
  - Radio list
  - Flow link
- Added an `external-idp` module to support social sign on with supported external IDPs by using browser redirects. [SDKS-3720, SDKS-3920]

  Supported external IDPs:
  - Apple
  - Facebook
  - Google
- Added `Accept-Language` header to support localization. [SDKS-3623]
- Added ability to validate PingOne Form fields. [SDKS-3671, SDKS-3672]
- Added support for default values in PingOne Form fields. [SDKS-3674]
- Added a `PingBrowser` module. [SDKS-3920]
- Added Swift 6 support. [SDKS-3728]

DaVinci client for JavaScript 1.1.0

April 17, 2025

Added

- Added support for additional PingOne Form fields.
  - Label

    - Checkbox
    - Dropdown
    - Combobox
    - Radio list
    - Flow link
- Added support for social sign on with supported external IDPs.

  Supported external IDPs:

    - Apple
    - Facebook
    - Google
- Added the ability to call start with query parameters which the DaVinci client appends to the `/authorize` call.
- Added request middleware to amend outgoing HTTP requests, for example to override `Accept-Language` headers.
- Added ability to validate PingOne Form fields.
- Added support for default values in PingOne Form fields.

Updated

- Updated dependency on `@forgerock/javascript-sdk` to `4.7.0`.
- Updated error node to now be submittable to help the app recover from an error state.
- Updated the checks to determine what node state the DaVinci Client is in based on the response from PingOne.

DaVinci client 1.0.0

December 16, 2024

Added

- Initial release of the DaVinci client, for Android, iOS and JavaScript.

Login Widget changelog

Subscribe to get automatic updates:

- Ping SDKs Changelog RSS feed
- Ping SDKs Changelog email notifications

Ping (ForgeRock) Login Widget 1.3.0

June 5, 2024 `minor`

Added

- Added support for integration with PingOne Protect.
- Added the name of the device to the recovery codes page.

Fixed

- Corrected an issue that prevented use of the `logLevel` parameter in the Ping (ForgeRock) Login Widget configuration.
- Fixed an issue with configuration literals that caused `ZodError` messages in the console.

Ping (ForgeRock) Login Widget 1.2.1

January 8, 2024 `minor`

Added

- Support for CAPTCHA nodes.

Ping (ForgeRock) Login Widget 1.1

July 17, 2023 `minor`

Added

- Support for device profiling callbacks (`DeviceProfileCallback`)
- Support for web authentication (WebAuthn) journeys and trees.

Ping (ForgeRock) Login Widget 1.0

April 18, 2023 `major`

Changed

- First public release

Token Vault changelog

Subscribe to get automatic updates:

- Ping SDKs Changelog RSS feed
- Ping SDKs Changelog email notifications

Token Vault 4.2.0

September 11, 2023 `minor`

Added

- Added a requirement to declare a list of URLs in the Token Vault Proxy configuration. These generate an allowlist of origins to which the proxy can forward requests.

Token Vault 4.1.2

July 24, 2023 `major`

Added

- Initial release of Token Vault.

Limitations

This page lists the known issues and limitations of the Ping SDKs.

> **Important**
>
> SDKs Renamed
> Prior to November 2024, the **Ping SDKs** were known as the **ForgeRock SDKs**.

All platforms

- The Ping SDKs ***do not*** support authentication chains nor modules.
- The FRUI module is for prototyping your UI, and is not intended for production use, as-is.
- As of ForgeRock SDKs 3.0, the Identity Providers supported for social login are limited to Apple, Facebook, and Google.

Ping SDK for Android

- Displaying CAPTCHAs or using the Ping (ForgeRock) Authenticator module in your application requires the presence of the Google Play Services.
- The Authenticator module of the Ping SDK for Android only supports Firebase Cloud Messaging service as a Push Notification provider.
- Social Login requires PingAM 7.1 or the latest version of PingOne Advanced Identity Cloud.
- Calling `FRUser.logout()` will only sign out the session from PingAM but not the Social Identity Provider. Every subsequent, social login attempt will automatically log in without asking for credentials.
- Biometric authentication is only supported on Android 7.0 or newer.
- Biometric authentication requires PingAM 7.1 or the latest version of PingOne Advanced Identity Cloud.
- Biometric authentication requires the use of Google Play Services.
- When a biometric dialog, such as the *provide fingerprint* dialog, is dismissed, the application may become unresponsive.
- Biometric authentication does not distinguish individual biometrics (fingerprints or faces), but is limited to any registered for the device's current user account.
- As of ForgeRock SDKs 3.0, only platform authenticators can be used for WebAuthn; roaming/USB authenticators, like Yubikey, are not currently supported.

- Ping SDK for Android apps do not function correctly if they are minimized to picture-in-picture mode in Android custom tabs.

  The Ping SDK is not able to detect being minimized until API support from Google is available in Android.

Ping SDK for iOS

- Data encryption with Secure Enclave is only available for iOS 10+ devices with TouchID or FaceID.
- DeviceCollector customization is only available in Swift.
- JailbreakDetector customization is only available in Swift.
- HiddenValueCallback and SuspendedTextOutputCallback are not accessible in Objective-C.
- FRAuthenticator SDK is only available in Swift.
- Social Login requires PingAM 7.1 or the latest version of PingOne Advanced Identity Cloud.
- Calling `FRUser.logout()` will only sign out the session from PingAM but not the Social Identity Provider. Every subsequent, social login attempt will automatically log in without asking for credentials.
- The Google Sign-In SDK is only compatible with CocoaPods (Swift Package Manager is not supported).
- Sign In With Apple is only supported in iOS 13 and above.
- Biometric authentication requires PingAM 7.1 or the latest version of PingOne Advanced Identity Cloud.
- Biometric authentication does not distinguish between individual biometrics (fingerprints or faces), but is limited to the collection of biometrics registered for the device's current user account.
- For Biometric authentication, iOS only supports the ES256 signing algorithm, this is configured in the WebAuthn Registration node.
- For "usernameless" biometric authentication support, "limit registrations" must be disabled within the WebAuthn Registration node.
- As of ForgeRock SDKs 3.0, only the platform authenticator can be used for WebAuthn; roaming/USB authenticators, like Yubikey, are not supported.
- Device Binding is not supported on iOS simulators. You must use a physical device to test Device Binding.

Ping SDK for JavaScript

- The Ping SDK for JavaScript is currently unable to revoke PingOne-issued OIDC tokens when using Firefox and Safari, due to third-party cookie protection.
- When resources are protected by PingGateway, the Ping SDK for JavaScript can only support transactional authorization if PingAM and PingGateway are on the same origin.
- FireFox does not support Touch ID as a WebAuthn device on Mac therefore it limits some WebAuthn node configurations.
- The SDK requires polyfills to function in IE 11 and Legacy Edge.
- In WebKit for both macOS and iOS, the "Prevent Cross-site Tracking" option, which is enabled by default, can prevent the SDK from functioning when the app and PingAM are under different origins.

- Collecting location information requires the user's system preferences to allow browser access to location information.
- IndexedDB as a token storage strategy has a known issue with Firefox Private Mode. (Use `localStorage` as an alternative.)
- Social login with Apple requires the use of a form POST, so the "Redirect URL" cannot be an SPA as they are unable to handle a POST request; the use of the special PingAM endpoint explained in Set up social login is recommended.
- Calling `FRUser.logout()` will only sign out the session from PingAM but not the social identity provider. Every subsequent social login attempt will automatically log in without asking for credentials.

Ping (ForgeRock) Authenticator module

- The default storage client for Android that is built on `SharedPreferences` can behave unpredictably on devices from certain manufacturers that customize the Android operating system.

  For maximum compatibility with devices from different manufacturers we highly recommend that you implement your own custom storage client for Android devices.

Incompatible changes

Incompatible changes refer to changes that impact existing functionality and might have an effect on your deployment. Before you upgrade, review these lists and make the appropriate changes to your scripts and plugins.

> **Important**
>
> SDKs Renamed
> Prior to November 2024, the **Ping SDKs** were known as the **ForgeRock SDKs**.

ForgeRock SDK for iOS 4.0.0

Exception changes

- The `FRAClient.updateAccount()` method now throws `AccountError.accountLocked` when attempting to update a locked account.
- The `HOTPMechanism.generateCode()` and `TOTPMechanism.generateCode()` methods now throw AccountError.accountLocked when attempting to get an OATH token for a locked account.

Method signature changes

The signature of the following methods has changed:

`WebAuthnRegistrationCallback`

Old

```
public func register(
  node: Node? = nil,
  onSuccess: @escaping StringCompletionCallback,
  onError: @escaping ErrorCallback
)
```

New

```
public func register(
  node: Node? = nil,
  window: UIWindow? = UIApplication.shared.windows.first,
  deviceName: String? = nil,
  usePasskeysIfAvailable: Bool = false,
  onSuccess: @escaping StringCompletionCallback,
  onError: @escaping ErrorCallback
)
```

`WebAuthnAuthenticationCallback`

Old

```
public func authenticate(
  node: Node? = nil,
  onSuccess: @escaping StringCompletionCallback,
  onError: @escaping ErrorCallback
)
```

New

```
public func authenticate(
  node: Node? = nil,
  window: UIWindow? = UIApplication.shared.windows.first,
  preferImmediatelyAvailableCredentials: Bool = false,
  usePasskeysIfAvailable: Bool = false,
  onSuccess: @escaping StringCompletionCallback,
  onError: @escaping ErrorCallback
)
```

`FacebookSignInHandler`

Old

```
public static func handle(
  _ application: UIApplication,
  _ url: URL,
  _ options: [UIApplication.OpenURLOptionsKey : Any] = [:]
) -> Bool
```

New

```
public static func application(
  _ application: UIApplication,
  didFinishLaunchingWithOptions launchOptions: [UIApplication.LaunchOptionsKey : Any]? = nil
) -> Bool
```

> **Note**
>
> In ForgeRock SDK for Android 4.0.0 and later, make calls to the method using:
> `application(_ application:, didFinishLaunchingWithOptions launchOptions: )`
> Not the previous call:
> `application(_ app:, open url:, options:)`

ForgeRock SDK for Android 4.0.0

Removed support for native single sign-on (SSO)

The Android platform has deprecated `sharedUserId` that underpins the ForgeRock SDK for Android native SSO implementation.

This native SSO implementation will not be viable after `sharedUserId` is removed from the Android platform.

Due to this deprecation, ForgeRock SDK for Android 4.0.0 removes support for Android native single sign-on, as well as the following related changes:

- `AuthenticatorService` is removed. Remove `<service>` from your `AndroidManifest.xml` file.
- The ForgeRock SDK for Android no longer requires the following permissions:
  - `android.permission.AUTHENTICATE_ACCOUNTS`
  - `android.permission.GET_ACCOUNTS`
  - `android.permission.MANAGE_ACCOUNTS`
  - `android.permission.USE_CREDENTIALS`
- The ForgeRock SDK for Android no longer requires the following configuration properties:
  - `forgerock`
  - `forgerock_account_name`
  - `forgerock_webauthn_account_name`
  - `forgerock_webauthn_max_credential`
  - `forgerock_enable_sso`

Method signature changes

The signature of the following methods has changed:

`WebAuthnRegistrationCallback`

Old

```
public void register(Node node,FRListener<Void> listener)
```

New

```
suspend fun register(context: Context, node: Node)
```

`WebAuthAuthenticationCallback`

Old

```
public void authenticate(
  @NonNull Fragment fragment,
  @NonNull Node node,
  @Nullable WebAuthnKeySelector selector,
  FRListener<Void> listener
)
```

New

```
suspend fun authenticate(
  context: Context,
  node: Node,
  selector: WebAuthnKeySelector = WebAuthnKeySelector.DEFAULT
)
```

`org.forgerock.android.auth.FRAClient`

Old

```
public boolean updateAccount(@NonNull Account account)
```

New

```
public boolean updateAccount(@NonNull Account account)
  throws AccountLockException
```

`org.forgerock.android.auth.HOTPMechanism`

Old

```
public OathTokenCode getOathTokenCode()
  throws OathMechanismException
```

New

```
public OathTokenCode getOathTokenCode()
  throws OathMechanismException, AccountLockException
```

`org.forgerock.android.auth.OathMechanism`

Old

```
public abstract OathTokenCode getOathTokenCode()
  throws OathMechanismException
```

New

```
public abstract OathTokenCode getOathTokenCode()
  throws OathMechanismException, AccountLockException
```

`org.forgerock.android.auth.TOTPMechanism`

Old

```
public OathTokenCode getOathTokenCode()
  throws OathMechanismException
```

New

```
public OathTokenCode getOathTokenCode()
  throws OathMechanismException, AccountLockException
```

ForgeRock SDK for JavaScript 4.0.0

No longer provides Universal Module Definition (UMD) support

This version of the ForgeRock SDK for JavaScript does not provide a UMD bundle.

If you require UMD support, you can:

- Use an earlier version of the ForgeRock SDK for JavaScript, such as 3.4.0.
- Clone the repository with the latest source code and configure it locally to provide UMD support.

> **Note**
>
> Support for CommonJS (CJS) and ES Modules (ESM) is not affected and still provided in ForgeRock SDK for JavaScript 4.0.0

Removal of `indexedDB` token store

The `indexedDB` option has been removed from the `tokenStore` configuration property in ForgeRock SDK for JavaScript 4.0.0. The `indexedDB` option did not offer sufficient functionality or reliability when the browser is using a private or incognito window.

If you are using the `indexedDB` option after upgrading to ForgeRock SDK for JavaScript 4.0.0 it is ignored and the SDK defaults to using the `localStorage` option instead. A warning message is output to the browser console.

This change will not affect the functionality of your app.

For more information on options for the token store, refer to Configure the Ping SDKs for Auth Journeys.

Updated `Policy` types

Updated policy types so that a `PolicyRequirement` array is output from `failedPolicies`.

Removed duplicate modules

Removed the `FRUI` and `Event` modules from the ForgeRock SDK for JavaScript repository.

These modules were incorrectly duplicated from the `forgerock-javascript-sdk-ui` repository.

Deprecated

The functionality listed here is deprecated, and likely to be removed in a future release.

Deprecated since Ping SDK for JavaScript 4.0

JavaScript `support` configuration property

The `support` configuration property has been removed in Ping SDK for JavaScript 4.0.

This property could be used to change the way the SDK would make requests to the `/authorize` endpoint in OAuth 2.0 interactions.

If you configured the SDK to use the `modern` option, you might notice that your app uses the default iframe method to call the `/authorize` endpoint if you upgrade to this version of the SDK. This technical difference will not negatively impact your app's user-experience or require any code changes.

If you were using the `legacy` option or not providing a value for the `support` property at all, you will likely obtain improvements in latency and a reduction of errors in the logs when upgrading to Ping SDK for JavaScript 4.0.

Interface stability

Interfaces labelled as *Evolving* in the documentation may change without warning. In addition, the following rules apply:

- Interfaces that are not described in released product documentation should be considered *Internal/Undocumented*.
- Also refer to Deprecated features and Incompatible changes.

Product release levels

Ping Identity defines Major, Minor, Maintenance, and Patch product release levels. The version number reflects release level. The release level tells you what sort of compatibility changes to expect.

Release level definitions

Release Label	Version Numbers	Characteristics
Major	Version: x[.0.0] (trailing 0s are optional)	• Bring major new features, minor features, and bug fixes. • Can include changes even to Stable interfaces. • Can remove previously Deprecated functionality, and in rare cases remove Evolving functionality that has not been explicitly Deprecated. • Include changes present in previous Minor and Maintenance releases.
Minor	Version: x.y[.0] (trailing 0s are optional)	• Bring minor features, and bug fixes. • Can include backwards-compatible changes to Stable interfaces in the same Major release, and incompatible changes to Evolving interfaces. • Can remove previously Deprecated functionality. • Include changes present in previous Minor and Maintenance releases.
Maintenance, Patch	Version: x.y.z[.p] The optional *p* reflects a Patch version.	• Bring bug fixes • Are intended to be fully compatible with previous versions from the same Minor release.

Product stability labels

Ping Identity Platform software supports many features, protocols, APIs, GUIs, and command-line interfaces. Some of these are standard and very stable. Others offer new functionality that is continuing to evolve.

Ping Identity acknowledges you invest in these features and interfaces and so need to understand when they are expected to change. For that reason, we define stability labels and use these definitions in Ping Identity Platform products.

Stability label definitions

Stability Label	Definition
Stable	This documented feature or interface is expected to undergo backwards-compatible changes only for major releases. Changes may be announced at least one minor release before they take effect.
Evolving	This documented feature or interface is continuing to evolve and so is expected to change, potentially in backwards-incompatible ways even in a minor release. Changes are documented at the time of product release. While new protocols and APIs are still in the process of standardization, they are Evolving. This applies, for example, to recent Internet-Draft implementations and to newly developed functionality.

Stability Label	Definition
Legacy	This feature or interface has been replaced with an improved version, and is no longer receiving development effort from Ping Identity. You should migrate to the newer version, however the existing functionality will remain. Legacy features or interfaces will be marked as *Deprecated* if they are scheduled to be removed from the product.
Deprecated	This feature or interface is deprecated, and likely to be removed in a future release. For previously stable features or interfaces, the change was likely announced in a previous release. Deprecated features or interfaces will be removed from Ping Identity products.
Removed	This feature or interface was deprecated in a previous release, and has now been removed from the product.
Technology Preview	Technology previews provide access to new features that are considered as new technology that is not yet supported. Technology preview features may be functionally incomplete, and the function as implemented is subject to change without notice. *DO NOT DEPLOY A TECHNOLOGY PREVIEW INTO A PRODUCTION ENVIRONMENT.* Customers are encouraged to test drive the technology preview features in a non-production environment, and are welcome to make comments and suggestions about the features in the associated forums. Ping Identity does not guarantee that a technology preview feature will be present in future releases, the final complete version of the feature is liable to change between preview and the final version. Once a technology preview moves into the completed version, said feature will become part of Ping Identity Platform. Technology previews are provided on an "AS-IS" basis for evaluation purposes only, and Ping Identity accepts no liability or obligations for the use thereof.
Internal/ Undocumented	Internal and undocumented features or interfaces can change without notice. If you depend on one of these features or interfaces, contact support to discuss your needs.

Getting support

Ping Identity provides support services, professional services, training, and partner services to assist you in setting up and maintaining your deployments. For a general overview of these services, see https://www.pingidentity.com.

Ping Identity has staff members around the globe who support our international customers and partners. For details on Ping Identity's support offering, visit https://www.pingidentity.com/support.

Ping Identity publishes comprehensive documentation online:

- The Ping Identity Knowledge Base offers a large and increasing number of up-to-date, practical articles that help you deploy and manage Ping Identity Platform software.

  While many articles are visible to everyone, Ping Identity customers have access to much more, including advanced information for customers using Ping Identity Platform software in a mission-critical capacity.

- Ping Identity product documentation, such as this document, aims to be technically accurate and complete with respect to the software documented. It is visible to everyone and covers all product features and examples of how to use them.

Troubleshooting

For troubleshooting information, see the following articles in the Knowledge Base:

- Ping SDK for Android Troubleshooting
- Ping SDK for iOS Troubleshooting
- Ping SDK for JavaScript Troubleshooting

Additional Articles

- How do I troubleshoot issues with the CORS filter in PingAM/AM/OpenAM (All versions)?

Compatibility

Supported server versions

The Ping SDKs support the following server versions:

- PingOne
- PingOne Advanced Identity Cloud
- PingAM 6.5, 7.0, 7.1, 7.2, 7.3, 7.4, 7.5, 8.0, and later
- PingFederate

Supported operating systems and browsers

Select a platform below to view the supported operating systems and browsers.

Android

The Ping SDK for Android supports the following versions of the Android operating system:

Supported Android versions and original release dates

Release	API Levels	Released
Android 15	35	September, 2024
Android 14	34	October, 2023
Android 13	33	March, 2022
Android 12	31, 32	October, 2021
Android 11	30	September, 2020
Android 10	29	September, 2019
Android 9 (Pie)	28	August, 2018

Important

Since **March 1st, 2025**, the Ping SDKs support policy is as follows:

- Every public major release of Android within the last 6 years.

Supported browsers on Android

- Chrome - Two most recent major versions.

iOS

The Ping SDK for iOS supports the following versions of the iOS operating system:

Supported iOS versions and original release dates

Release	Released
iOS 18	September, 2024
iOS 17	September, 2023
iOS 16	September, 2022

Important

Since **March 1st, 2025**, the Ping SDKs support policy is as follows:

- Every public major release of iOS within the last 3 years.

Supported browsers on iOS

- Safari - Two most recent major versions.

JavaScript / Login Widget

The Ping SDK for JavaScript, and the Ping (ForgeRock) Login Widget support the desktop and mobile browsers listed below.

Minimum supported Desktop browser versions

- Chrome 83
- Firefox 77
- Safari 13
- Microsoft Edge 83 (Chromium)

Supported Mobile browsers

- iOS (Safari) - Two most recent major versions of the operating system.
- Android (Chrome) - Two most recent major versions of the operating system.

JavaScript Compatibility with WebViews

A WebView allows you to embed a web browser into your native Android or iOS application to display HTML pages, and run JavaScript apps.

For example, the Android system WebView is based on the Google Chrome engine, and the iOS WebView is based on the Safari browser engine.

However, it is important to note that WebViews do not implement the full feature set of their respective browsers. For example, some of the browser-provided APIs that the Ping SDK for JavaScript requires are not available in a WebView, such as the WebAuthn APIs.

In addition, there are concerns that a WebView does not provide the same level of security as their full browser counterparts.

As the SDK requires full, spec-compliant, browser-supplied APIs for full functionality we **do not** support usage within a WebView.

We also do not support or test usage with any wrappers around WebViews.

Whilst you might be able to implement simple use-cases using the Ping SDK for JavaScript within a WebView, we recommend that you use an alternative such as opening a full browser, or using an in-app instance of a full browser such as Custom Tabs for Android or SFSafariViewController for iOS.

Supported authentication journey callbacks

The Ping SDKs support the following authentication journey callbacks when using the following servers:

- PingOne Advanced Identity Cloud
- PingAM

Callback name	Callback description	Android	iOS	JavaScript
`BooleanAttributeInputCallback` `SDK 2.1`	Collects true or false.	☑	☑	☑
`ChoiceCallback`	Collects single user input from available choices, retrieves selected choice from user interaction.	☑	☑	☑
`ConfirmationCallback`	Retrieve a selected option from a list of options.	☑	☑	☑
`ConsentMappingCallback` `SDK 2.0`	Prompts the user to consent to share their profile data.	☑	✕	✕
`DeviceBindingCallback`	Cryptographically bind a mobile device to a user account.	☑ `SDK 4.0`	☑ `SDK 4.0`	✕

Callback name	Callback description	Android	iOS	JavaScript
`DeviceProfileCallback` `SDK 2.0`	Collects meta and/or location data about the authenticating device.	☑	☑	☑
`DeviceSigningVerifierCallback`	Verify ownership of a bound device by signing a challenge.	☑ `SDK 4.0`	☑ `SDK 4.0`	✕
`HiddenValueCallback`	Returns form values that are not visually rendered to the end user.	☑	☑	☑
`IdPCallback`	Provides the information required for connecting to an identity provider (IdP) for social sign-on.	☑	☑	☑
`KbaCreateCallback` `SDK 2.0`	Collects knowledge-based answers. For example, the name of your first pet.	☑	☑	☑
`MetadataCallback` (1)	Injects key-value metadata into the authentication process. For example, the WebAuthn nodes use this callback to return the data the SDK requires to perform authentication and registration.	☑	☑	☑
`NameCallback`	Collects a username.	☑	☑	☑
`NumberAttributeInputCallback` `SDK 2.1`	Collects a number.	☑	☑	☑
`PasswordCallback`	Collects a password or one-time pass code.	☑	☑	☑
`PingOneProtectEvaluationCallback` `SDK 4.4`	Collects captured contextual data from the client to perform risk evaluations.	☑	☑	☑
`PingOneProtectInitializeCallback` `SDK 4.4`	Instructs the client to start capturing contextual data for risk evaluations	☑	☑	☑
`PollingWaitCallback`	Instructs the client to wait for the given period and resubmit the request.	☑	☑	☑
`ReCaptchaCallback`	Provides data required to use a CAPTCHA in your apps.	☑ (2)	☑	☑

Callback name	Callback description	Android	iOS	JavaScript
`ReCaptchaEnterpriseCallback`	Provides data required to use reCAPTCHA Enterprise in your apps.	☑ (2) SDK 4.6	☑ SDK 4.6	☑ SDK 4.6
`RedirectCallback`	Redirects the user's browser or user-agent.	×	×	☑
`SelectIdPCallback`	Provides a list of identity providers (IdPs) users can choose from to perform social sign-on.	☑	☑	☑
`StringAttributeInputCallback` SDK 2.0	Collects the values of attributes for use elsewhere in a tree.	☑	☑	☑
`SuspendedTextOutputCallback` SDK 2.1	Pause and resume authentication, sometimes known as "magic links".	☑	☑	☑
`TermsAndConditionsCallback` SDK 2.0	Collects a user's acceptance of the configured Terms & Conditions.	☑	☑	☑
`TextInputCallback`	Collects text input from the end user. For example, a nickname for their account.	☑ SDK 4.4	☑ SDK 4.4	☑ SDK 3.4
`TextOutputCallback`	Provides a message to be displayed to a user with a given message type.	☑	☑	☑
`TextOutputCallback` *(`messageType` `===` `4`)*	Some nodes use the `TextOutputCallback` callback to include JavaScript that is intended to be run on the client. In this case the `mesageType` property equals `4`.	×	×	☑
`ValidatedPasswordCallback` SDK 2.0	Collects a password value with optional password policy validation.	☑	☑	☑
`ValidatedUsernameCallback` SDK 2.0	Collects a username value with optional username policy validation.	☑	☑	☑

The table below lists the nodes that might return supported callbacks.

The actual callbacks a node returns depends on its configuration. It might not return all the callbacks listed in this table.

Callback	Auth nodes that might return callback
`BooleanAttributeInputCallback`	• Attribute Collector node
`ChoiceCallback`	• Choice Collector node
`ConfirmationCallback`	• LDAP Decision node • Message node • MFA Registration Options node • OATH Token Verifier node • Polling Wait node • Push Wait node • WebAuthn Authentication node • OATH Registration node
`ConsentMappingCallback`	• Consent Collector node
`DeviceBindingCallback`	• Device Binding node
`DeviceProfileCallback`	• Device Profile Collector node
`DeviceSigningVerifierCallback`	• Device Signing Verifier node
`HiddenValueCallback`	• Amster Jwt Decision node • Push Wait node • WebAuthn Authentication node • WebAuthn Registration node
`IdPCallback`	• Social Provider Handler node
`KbaCreateCallback`	• KBA Definition node
`MetaDataCallback`	• WebAuthn Authentication node • WebAuthn Registration node

Callback	Auth nodes that might return callback
`NameCallback`	• Username Collector node • Datastore Decision node • OATH Token Verifier node • Platform Username node • Configuration Provider node
`NumberAttributeInputCallback`	• Attribute Collector node
`PasswordCallback`	• Create Password node • Password Collector node • Datastore Decision node • KBA Verification node • LDAP Decision node • One-time Password Collector Decision node • Platform Password node
`PingOneProtectEvaluationCallback`	• PingOne Protect Evaluation node
`PingOneProtectInitializeCallback`	• PingOne Protect Initialization node
`PollingWaitCallback`	• Combined MFA Registration node • Push Registration node
`ReCaptchaCallback`	• CAPTCHA node • Legacy CAPTCHA node (deprecated)
`ReCaptchaEnterpriseCallback`	• reCAPTCHA Enterprise node
`RedirectCallback`	• Provision IDM Account node • Identity Assertion node • Social Provider Handler node
`SelectIdPCallback`	• Select Identity Provider node

Callback	Auth nodes that might return callback
`StringAttributeInputCallback`	• Attribute Collector node
`SuspendedTextOutputCallback`	• Email Suspend node
`TermsAndConditionsCallback`	• Accept Terms and Conditions node
`TextInputCallback`	• Configuration Provider node
`TextOutputCallback`	• Create Password node • Display Username node • LDAP Decision node • Message node • MFA Registration Options node
`TextOutputCallback (messageType == 4)`	• WebAuthn Authentication node • WebAuthn Registration node
`ValidatedPasswordCallback`	• Platform Password node
`ValidatedUsernameCallback`	• Platform Username node

The table below lists the supported callbacks that a node might return.

The actual callbacks a node returns depends on its configuration. It might not return all the callbacks listed in this table.

Auth node	Callbacks the node might return
Accept Terms and Conditions node	`TermsAndConditionsCallback`
Amster Jwt Decision node	`HiddenValueCallback`

Attribute Collector node	`BooleanAttributeInputCallback`
	`NumberAttributeInputCallback`
	`StringAttributeInputCallback`

CAPTCHA node	`ReCaptchaCallback`

Choice Collector node	`ChoiceCallback`
Combined MFA Registration node	`PollingWaitCallback`
Configuration Provider node	`NameCallback` `TextInputCallback`
Consent Collector node	`ConsentMappingCallback`
Create Password node	`PasswordCallback` `TextOutputCallback`
Datastore Decision node	`NameCallback` `PasswordCallback`
Device Binding node	`DeviceBindingCallback`
Device Profile Collector node	`DeviceProfileCallback`
Device Signing Verifier node	`DeviceSigningVerifierCallback`
Display Username node	`TextOutputCallback`
Email Suspend node	`SuspendedTextOutputCallback`
Identity Assertion node	`RedirectCallback`
KBA Definition node	`KbaCreateCallback`
KBA Verification node	`PasswordCallback`

LDAP Decision node	`ConfirmationCallback`
	`PasswordCallback`
	`TextOutputCallback`

Legacy CAPTCHA node (deprecated)	`ReCaptchaCallback`
Message node	`ConfirmationCallback` `TextOutputCallback`
MFA Registration Options node	`ConfirmationCallback` `TextOutputCallback`
OATH Registration node	`ConfirmationCallback`
OATH Token Verifier node	`ConfirmationCallback` `NameCallback`
One-time Password Collector Decision node	`PasswordCallback`

Password Collector node	`PasswordCallback`
PingOne Protect Evaluation node	`PingOneProtectEvaluationCallback`
PingOne Protect Initialization node	`PingOneProtectInitializeCallback`
Platform Password node	`PasswordCallback` `ValidatedPasswordCallback`
Platform Username node	`NameCallback` `ValidatedUsernameCallback`
Polling Wait node	`ConfirmationCallback`
Provision IDM Account node	`RedirectCallback`
Push Registration node	`PollingWaitCallback`
Push Wait node	`ConfirmationCallback` `HiddenValueCallback`
reCAPTCHA Enterprise node	`ReCaptchaEnterpriseCallback`
Select Identity Provider node	`SelectIdPCallback`
Social Provider Handler node	`IdPCallback` `RedirectCallback`
Username Collector node	`NameCallback`

WebAuthn Authentication node	`ConfirmationCallback`
	`HiddenValueCallback`
	`MetaDataCallback`
	`TextOutputCallback (messageType == 4)`

WebAuthn Registration node	`HiddenValueCallback`
	`MetaDataCallback`
	`TextOutputCallback (messageType == 4)`

(1) The WebAuthn Authentication node and the WebAuthn Registration node both use a `MetaDataCallback` when the **Return challenge as JavaScript** is *NOT* enabled.

You must *not* enable this option when handling WebAuthn journeys with the Ping SDK for Android and iOS.

The Ping SDK for JavaScript handles either the `MetaDataCallback` or the JavaScript-based payload.

(2) Requires the presence of Google Play Services.

Supported PingOne fields and collectors

The DaVinci clients support the following connectors and capabilities when connecting to PingOne:

- PingOne Forms Connector
  - **Show Form** capability
- HTTP Connector
  - **Custom HTML** capability

PingOne Form Connector fields

Custom Fields support

Field (`Collector`)	Description	DaVinci module		
		Android	iOS	JavaScript
Text Input (`TextCollector`)	Collects a single text string.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`
Password (`PasswordCollector`)	Collects a single text string that cannot be read from the screen.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`
Dropdown (`SingleSelectCollector`)	Collects a value from a dropdown containing one or more text strings.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`
Combobox (`MultiSelectCollector`)	Collects a value from a dropdown containing one or more text strings, the user can enter their own text string.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`
Radio Button List (`SingleSelectCollector`)	Collects a value from one or radio buttons.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`
Checkbox List (`MultiSelectCollector`)	Collects the value of one or more checkboxes.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`

Toolbox support

Field (`Collector`)	Description	DaVinci module		
		Android	iOS	JavaScript
Flow Button (`FlowCollector`)	Presents a customized button.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`
Flow Link (`FlowCollector`)	Presents a customized link.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`

Translatable Rich Text (`TextCollector`)	Presents rich text that you can translate into multiple languages.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`
Social Login (`IdpCollector`)	Presents a button to allow users to authenticate using an external identity provider, such as Apple, Facebook, or Google.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`

HTTP Connector fields

- HTTP Connector field and collector support
- HTTP Connector SK-Component support

HTTP Connector field and collector support

Field (`Collector`)	Description	DaVinci module		
		Android	iOS	JavaScript
Text field (`TextCollector`)	Collects a single text string.	☑ `1.0.0`	☑ `1.0.0`	☑ `1.0.0`
Password field (`PasswordCollector`)	Collects a single text string that cannot be read from the screen.	☑ `1.0.0`	☑ `1.0.0`	☑ `1.0.0`
Submit Button (`SubmitCollector`)	Sends the collected data to PingOne to continue the DaVinci flow.	☑ `1.0.0`	☑ `1.0.0`	☑ `1.0.0`
Flow Button (`FlowCollector`)	Triggers an alternative flow without sending the data collected so far to PingOne.	☑ `1.0.0`	☑ `1.0.0`	☑ `1.0.0`
Label (`LabelCollector`)	Display a read-only text label.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`
Radio / Dropdown (`SingleSelectCollector`)	Collects a single value from a choice of multiple options.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`

HTTP Connector SK-Component support

SK-Component (`Collector`)	Description	DaVinci module		
		Android	iOS	JavaScript
skIDP (`IdpCollector`)	Presents a button to allow users to authenticate using an external identity provider, such as Apple, Facebook, or Google.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`

Unsupported features:

Verify that your flow does not depend on any *unsupported* elements:

SKPolling components

The SKPolling component cannot be processed by the DaVinci Client and should not be included in flows.

Features such as Magic Link authentication require the **SKPolling** component and therefore cannot be used with the DaVinci Client.

Images

Images included in the flow cannot be passed to the SDK.

Introducing the Ping SDKs for Authentication Journeys

Server support:

- ✕ PingOne
- ✓ PingOne Advanced Identity Cloud
- ✓ PingAM
- ✕ PingFederate

SDK support:

- ✓ Ping SDK for Android
- ✓ Ping SDK for iOS
- ✓ Ping SDK for JavaScript

The Ping SDKs can leverage PingOne Advanced Identity Cloud authentication journeys and PingAM authentication trees, and their associated callbacks.

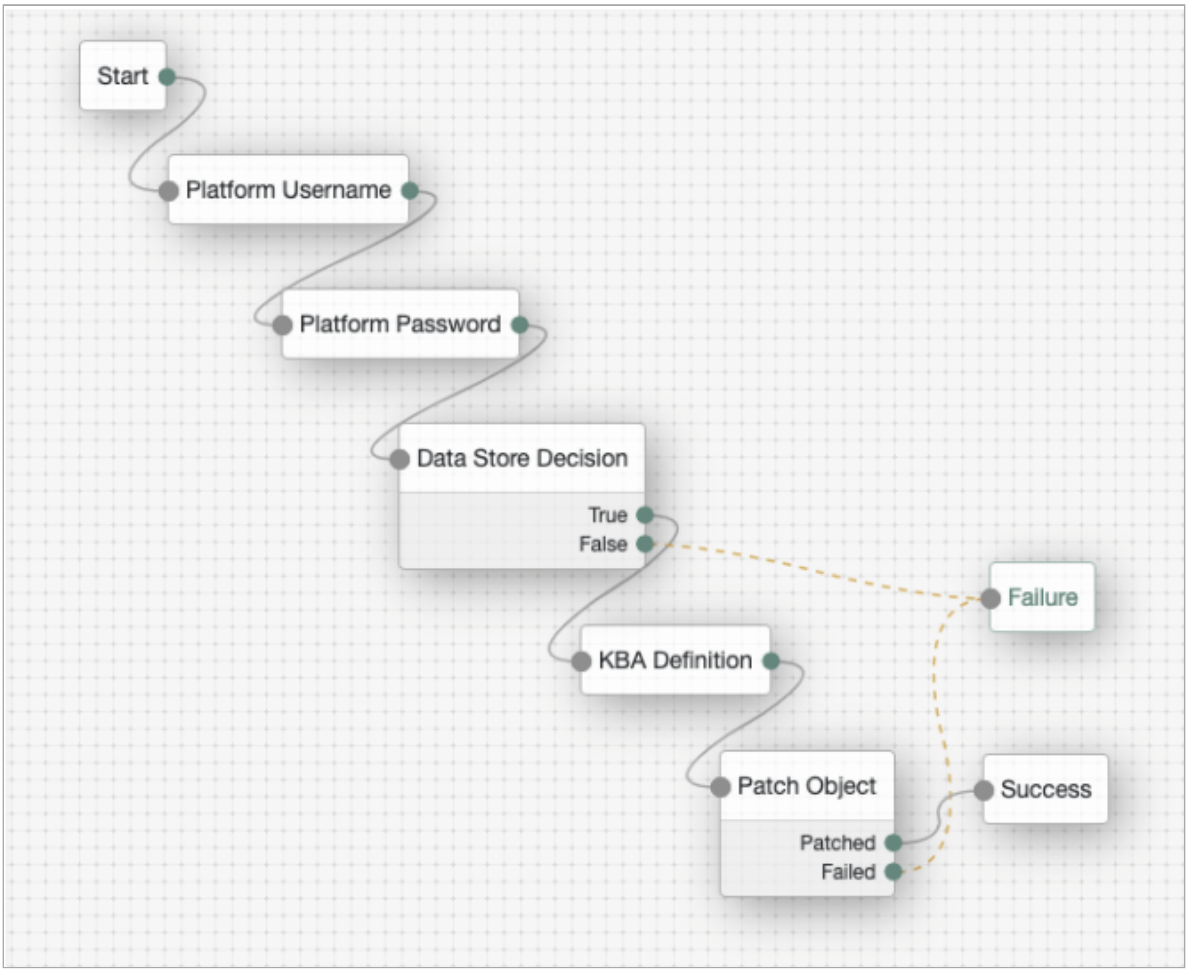
For more information, refer to Supported callbacks.

They let you step through each node in a journey, where you render the appropriate user interface to collect input from your users. The Ping SDKs then return the input and continue the journey.

For example, let's say you want to use this authentication flow:

1. Collect username and password.
2. Request KBA information.
3. Request the user to accept the terms and conditions.

You can use the SDK to make each callback call the next step in the tree. You don't have to traverse the REST APIs to call the next step.



Real time response to authentication tree changes

The Ping SDKs empower developers to build applications that can handle the changes to your authentication journeys in real time, without having to redeploy your app.

Token management

The Ping SDKs use the OAuth 2.0 auth code flow, and support PKCE.

This method is the best practice for first-party applications. The SDK automatically handles token exchange for you, and also securely stores the tokens.

Token refresh is automatically handled by the SDK, so you don't have to think about it.

Single sign-on (SSO)

In some scenarios, your company may have multiple native applications that customers have installed on their devices.

You can use the SDK to seamlessly sign users in to multiple applications on a device.

When the customer signs in to one application, they are automatically signed in to a second application on that device—without having to authenticate again.

Push authentication and OTP

The Ping SDKs can help you integrate push authentication or one-time password (OTP) capabilities into your mobile applications so your end users don't have to download and use a dedicated Authenticator application. The SDK's Authenticator module can support:

- ✓ Time-based one-time passwords (TOTP)
- ✓ HMAC-based one-time password (HOTP)
- ✓ Push notifications

Pluggability and extensibility

The SDK has a modular architecture and is designed with flexibility in mind.

Don't want to use our method for jailbreak detection? No problem! Just plug in your own method, or use any 3rd-party plug-in instead.

Device security profile

Using the SDK, you have the option to collect device profile information to use in your authentication journeys.

You might use this data to compare a user sign-in to a prior sign-in event.

If the device profile has changed too much from the prior event, you can deny the sign-in.

Jailbreak detection

Detecting whether a device is jailbroken or rooted assures developers that a device is managed by the authorized device owner.

Jailbroken devices may be running outdated OS versions, or could be missing security patches.

Detecting whether a device is jailbroken can provide valuable insight into the security posture of a device. You can feed that insight into your authentication journey.

The iOS and Android SDKs generate a score to determine if a device is jailbroken or rooted. There are a number of factors that go into creating this score. The score ranges from 0 to 1.0, where 1 indicates the device is an emulator.

You can use this information as part of an authentication flow to ask the user for another factor, or to deny access entirely.

Device ID and meta data

Ping SDKs can automatically generate a device ID for you. You can use the ID with PingIDM or PingAM to allow your users to manage their devices.

For example, you can insert the device ID and associated data into a user's profile. This lets them view their devices and set the devices as trusted. You can also decide to use a recognized device in an authentication flow to avoid asking a user for another factor.

> **Note**
>
> It is up to you what information you collect from users and devices.
> You should always use data responsibly and provide your users appropriate control over data they share with you.
> You are responsible for complying with any regulations or data protection laws.

Location information

You can collect latitude and longitude information from your users via the Android and iOS SDKs.

Apps that use location services must request location permissions from users.

Web biometrics

The Ping SDK for JavaScript supports web biometrics functionality provided by PingAM.

Web biometrics lets users authenticate by using an authenticator device; for example, the fingerprint scanner on their laptop or phone, or a USB key such as those provided by Yubico, or Google's Titan security keys.

Communication with authentication devices is handled by the SDK. PingAM requests that the SDK activates authenticators with certain criteria; for example, it must be built-in to the platform, or is a cross-platform roaming USB device. You can also specify that the device must verify the *identity* of the user, rather than simply that a user is present.

The Ping SDKs have two methods for handling web biometrics: one for registering devices, and another for authenticating using a registered device.

For more information, refer to Web biometrics.

Mobile biometric authentication

Mobile biometric authentication lets users authenticate by using a mobile device's biometric authentication. Communication with the platform authenticator, for example, with a fingerprint reader or facial recognition system, is handled by the Ping SDK.

The Ping SDK communicates with PingAM to perform biometric registration and authentication using the WebAuthn nodes. Similar to WebAuthn with the Ping SDK for JavaScript, you can configure the nodes in PingAM to request that the SDK activates authenticators with certain criteria.

The Ping SDKs enable passkey support on supported platforms. Passkeys can be synchronized across a user's devices and browsers, simplifying device registration and enabling passwordless experiences.

This feature is available in the ForgeRock SDK for Android 3.0 and the ForgeRock SDK for iOS v3.0 or later. It requires PingOne Advanced Identity Cloud, or PingAM 7.1 or later.

For more information, refer to What are mobile biometrics?.

Social authentication

You can authenticate by using a trusted Identity Provider (IdP), like Apple, Facebook, Google, and many others.

These IdPs are used for authentication and identity verification.

This is often referred to as Social Login or Social Authentication.

These IdPs return the necessary information to integrate user information into your user's profile.

Depending on the device platform (Android, Web or iOS), the user is redirected from the current web application or login page to the IdP's authorization server. Or, if on a native mobile app, the user is directed to the IdP's authentication SDK, if available.

Once on the IdP via a web page or SDK, the user will authenticate and provide the necessary consent required for sharing the information.

When complete, the user is redirected back to your app or to your server to complete the authentication journey.

For more information, refer to Set up social login.

Compatibility

Supported server versions

The Ping SDKs support the following server versions:

- PingOne
- PingOne Advanced Identity Cloud
- PingAM 6.5, 7.0, 7.1, 7.2, 7.3, 7.4, 7.5, 8.0, and later
- PingFederate

Supported operating systems and browsers

Select a platform below to view the supported operating systems and browsers.

Android

The Ping SDK for Android supports the following versions of the Android operating system:

Supported Android versions and original release dates

Release	API Levels	Released
Android 15	35	September, 2024
Android 14	34	October, 2023
Android 13	33	March, 2022
Android 12	31, 32	October, 2021
Android 11	30	September, 2020
Android 10	29	September, 2019
Android 9 (Pie)	28	August, 2018

Important

Since **March 1st, 2025**, the Ping SDKs support policy is as follows:

- Every public major release of Android within the last 6 years.

Supported browsers on Android

- Chrome - Two most recent major versions.

iOS

The Ping SDK for iOS supports the following versions of the iOS operating system:

Supported iOS versions and original release dates

Release	Released
iOS 18	September, 2024
iOS 17	September, 2023
iOS 16	September, 2022

Important

Since **March 1st, 2025**, the Ping SDKs support policy is as follows:

- Every public major release of iOS within the last 3 years.

Supported browsers on iOS

- Safari - Two most recent major versions.

JavaScript / Login Widget

The Ping SDK for JavaScript, and the Ping (ForgeRock) Login Widget support the desktop and mobile browsers listed below.

Minimum supported Desktop browser versions

- Chrome 83
- Firefox 77
- Safari 13
- Microsoft Edge 83 (Chromium)

Supported Mobile browsers

- iOS (Safari) - Two most recent major versions of the operating system.
- Android (Chrome) - Two most recent major versions of the operating system.

JavaScript Compatibility with WebViews

A WebView allows you to embed a web browser into your native Android or iOS application to display HTML pages, and run JavaScript apps.

For example, the Android system WebView is based on the Google Chrome engine, and the iOS WebView is based on the Safari browser engine.

However, it is important to note that WebViews do not implement the full feature set of their respective browsers. For example, some of the browser-provided APIs that the Ping SDK for JavaScript requires are not available in a WebView, such as the WebAuthn APIs.

In addition, there are concerns that a WebView does not provide the same level of security as their full browser counterparts.

As the SDK requires full, spec-compliant, browser-supplied APIs for full functionality we **do not** support usage within a WebView.

We also do not support or test usage with any wrappers around WebViews.

Whilst you might be able to implement simple use-cases using the Ping SDK for JavaScript within a WebView, we recommend that you use an alternative such as opening a full browser, or using an in-app instance of a full browser such as Custom Tabs for Android or SFSafariViewController for iOS.

Supported authentication journey callbacks

The Ping SDKs support the following authentication journey callbacks when using the following servers:

- PingOne Advanced Identity Cloud
- PingAM

Callback name	Callback description	Android	iOS	JavaScript
`BooleanAttributeInputCallback` `SDK 2.1`	Collects true or false.	☑	☑	☑
`ChoiceCallback`	Collects single user input from available choices, retrieves selected choice from user interaction.	☑	☑	☑
`ConfirmationCallback`	Retrieve a selected option from a list of options.	☑	☑	☑
`ConsentMappingCallback` `SDK 2.0`	Prompts the user to consent to share their profile data.	☑	✕	✕
`DeviceBindingCallback`	Cryptographically bind a mobile device to a user account.	☑ `SDK 4.0`	☑ `SDK 4.0`	✕

Callback name	Callback description	Android	iOS	JavaScript
`DeviceProfileCallback` `SDK 2.0`	Collects meta and/or location data about the authenticating device.	☑	☑	☑
`DeviceSigningVerifierCallback`	Verify ownership of a bound device by signing a challenge.	☑ `SDK 4.0`	☑ `SDK 4.0`	✕
`HiddenValueCallback`	Returns form values that are not visually rendered to the end user.	☑	☑	☑
`IdPCallback`	Provides the information required for connecting to an identity provider (IdP) for social sign-on.	☑	☑	☑
`KbaCreateCallback` `SDK 2.0`	Collects knowledge-based answers. For example, the name of your first pet.	☑	☑	☑
`MetadataCallback` (1)	Injects key-value metadata into the authentication process. For example, the WebAuthn nodes use this callback to return the data the SDK requires to perform authentication and registration.	☑	☑	☑
`NameCallback`	Collects a username.	☑	☑	☑
`NumberAttributeInputCallback` `SDK 2.1`	Collects a number.	☑	☑	☑
`PasswordCallback`	Collects a password or one-time pass code.	☑	☑	☑
`PingOneProtectEvaluationCallback` `SDK 4.4`	Collects captured contextual data from the client to perform risk evaluations.	☑	☑	☑
`PingOneProtectInitializeCallback` `SDK 4.4`	Instructs the client to start capturing contextual data for risk evaluations	☑	☑	☑
`PollingWaitCallback`	Instructs the client to wait for the given period and resubmit the request.	☑	☑	☑
`ReCaptchaCallback`	Provides data required to use a CAPTCHA in your apps.	☑ (2)	☑	☑

Callback name	Callback description	Android	iOS	JavaScript
`ReCaptchaEnterpriseCallback`	Provides data required to use reCAPTCHA Enterprise in your apps.	☑ (2) SDK 4.6	☑ SDK 4.6	☑ SDK 4.6
`RedirectCallback`	Redirects the user's browser or user-agent.	×	×	☑
`SelectIdPCallback`	Provides a list of identity providers (IdPs) users can choose from to perform social sign-on.	☑	☑	☑
`StringAttributeInputCallback` SDK 2.0	Collects the values of attributes for use elsewhere in a tree.	☑	☑	☑
`SuspendedTextOutputCallback` SDK 2.1	Pause and resume authentication, sometimes known as "magic links".	☑	☑	☑
`TermsAndConditionsCallback` SDK 2.0	Collects a user's acceptance of the configured Terms & Conditions.	☑	☑	☑
`TextInputCallback`	Collects text input from the end user. For example, a nickname for their account.	☑ SDK 4.4	☑ SDK 4.4	☑ SDK 3.4
`TextOutputCallback`	Provides a message to be displayed to a user with a given message type.	☑	☑	☑
`TextOutputCallback` *(`messageType` `===` `4`)*	Some nodes use the `TextOutputCallback` callback to include JavaScript that is intended to be run on the client. In this case the `mesageType` property equals `4`.	×	×	☑
`ValidatedPasswordCallback` SDK 2.0	Collects a password value with optional password policy validation.	☑	☑	☑
`ValidatedUsernameCallback` SDK 2.0	Collects a username value with optional username policy validation.	☑	☑	☑

The table below lists the nodes that might return supported callbacks.

The actual callbacks a node returns depends on its configuration. It might not return all the callbacks listed in this table.

Callback	Auth nodes that might return callback
`BooleanAttributeInputCallback`	• Attribute Collector node
`ChoiceCallback`	• Choice Collector node
`ConfirmationCallback`	• LDAP Decision node • Message node • MFA Registration Options node • OATH Token Verifier node • Polling Wait node • Push Wait node • WebAuthn Authentication node • OATH Registration node
`ConsentMappingCallback`	• Consent Collector node
`DeviceBindingCallback`	• Device Binding node
`DeviceProfileCallback`	• Device Profile Collector node
`DeviceSigningVerifierCallback`	• Device Signing Verifier node
`HiddenValueCallback`	• Amster Jwt Decision node • Push Wait node • WebAuthn Authentication node • WebAuthn Registration node
`IdPCallback`	• Social Provider Handler node
`KbaCreateCallback`	• KBA Definition node
`MetaDataCallback`	• WebAuthn Authentication node • WebAuthn Registration node

Callback	Auth nodes that might return callback
`NameCallback`	• Username Collector node • Datastore Decision node • OATH Token Verifier node • Platform Username node • Configuration Provider node
`NumberAttributeInputCallback`	• Attribute Collector node
`PasswordCallback`	• Create Password node • Password Collector node • Datastore Decision node • KBA Verification node • LDAP Decision node • One-time Password Collector Decision node • Platform Password node
`PingOneProtectEvaluationCallback`	• PingOne Protect Evaluation node
`PingOneProtectInitializeCallback`	• PingOne Protect Initialization node
`PollingWaitCallback`	• Combined MFA Registration node • Push Registration node
`ReCaptchaCallback`	• CAPTCHA node • Legacy CAPTCHA node (deprecated)
`ReCaptchaEnterpriseCallback`	• reCAPTCHA Enterprise node
`RedirectCallback`	• Provision IDM Account node • Identity Assertion node • Social Provider Handler node
`SelectIdPCallback`	• Select Identity Provider node

Callback	Auth nodes that might return callback
`StringAttributeInputCallback`	• Attribute Collector node
`SuspendedTextOutputCallback`	• Email Suspend node
`TermsAndConditionsCallback`	• Accept Terms and Conditions node
`TextInputCallback`	• Configuration Provider node
`TextOutputCallback`	• Create Password node • Display Username node • LDAP Decision node • Message node • MFA Registration Options node
`TextOutputCallback (messageType == 4)`	• WebAuthn Authentication node • WebAuthn Registration node
`ValidatedPasswordCallback`	• Platform Password node
`ValidatedUsernameCallback`	• Platform Username node

The table below lists the supported callbacks that a node might return.

The actual callbacks a node returns depends on its configuration. It might not return all the callbacks listed in this table.

Auth node	Callbacks the node might return
Accept Terms and Conditions node	`TermsAndConditionsCallback`
Amster Jwt Decision node	`HiddenValueCallback`

Attribute Collector node	`BooleanAttributeInputCallback`
	`NumberAttributeInputCallback`
	`StringAttributeInputCallback`

CAPTCHA node	`ReCaptchaCallback`

Choice Collector node	`ChoiceCallback`
Combined MFA Registration node	`PollingWaitCallback`
Configuration Provider node	`NameCallback` `TextInputCallback`
Consent Collector node	`ConsentMappingCallback`
Create Password node	`PasswordCallback` `TextOutputCallback`
Datastore Decision node	`NameCallback` `PasswordCallback`
Device Binding node	`DeviceBindingCallback`
Device Profile Collector node	`DeviceProfileCallback`
Device Signing Verifier node	`DeviceSigningVerifierCallback`
Display Username node	`TextOutputCallback`
Email Suspend node	`SuspendedTextOutputCallback`
Identity Assertion node	`RedirectCallback`
KBA Definition node	`KbaCreateCallback`
KBA Verification node	`PasswordCallback`

LDAP Decision node	`ConfirmationCallback`
	`PasswordCallback`
	`TextOutputCallback`

Legacy CAPTCHA node (deprecated)	`ReCaptchaCallback`
Message node	`ConfirmationCallback` `TextOutputCallback`
MFA Registration Options node	`ConfirmationCallback` `TextOutputCallback`
OATH Registration node	`ConfirmationCallback`
OATH Token Verifier node	`ConfirmationCallback` `NameCallback`
One-time Password Collector Decision node	`PasswordCallback`

Password Collector node	`PasswordCallback`
PingOne Protect Evaluation node	`PingOneProtectEvaluationCallback`
PingOne Protect Initialization node	`PingOneProtectInitializeCallback`
Platform Password node	`PasswordCallback` `ValidatedPasswordCallback`
Platform Username node	`NameCallback` `ValidatedUsernameCallback`
Polling Wait node	`ConfirmationCallback`
Provision IDM Account node	`RedirectCallback`
Push Registration node	`PollingWaitCallback`
Push Wait node	`ConfirmationCallback` `HiddenValueCallback`
reCAPTCHA Enterprise node	`ReCaptchaEnterpriseCallback`
Select Identity Provider node	`SelectIdPCallback`
Social Provider Handler node	`IdPCallback` `RedirectCallback`
Username Collector node	`NameCallback`

WebAuthn Authentication node	`ConfirmationCallback`
	`HiddenValueCallback`
	`MetaDataCallback`
	`TextOutputCallback (messageType == 4)`

WebAuthn Registration node	`HiddenValueCallback`
	`MetaDataCallback`
	`TextOutputCallback (messageType == 4)`

(1) The WebAuthn Authentication node and the WebAuthn Registration node both use a `MetaDataCallback` when the **Return challenge as JavaScript** is *NOT* enabled.

You must *not* enable this option when handling WebAuthn journeys with the Ping SDK for Android and iOS.

The Ping SDK for JavaScript handles either the `MetaDataCallback` or the JavaScript-based payload.

(2) Requires the presence of Google Play Services.

Configure the Ping SDKs for Auth Journeys

The Ping SDKs are designed to be flexible and can be customized to suit many different situations.

Learn more about configuring and customizing the Ping SDKs in the sections below:

Configure Ping SDK properties

Learn how to configure properties in the SDKs so they can connect to your authorization server to authenticate your users and obtain tokens.

Learn more »

Configure logging in the Ping SDKs

Utilize logging messages in the Ping SDKs during development an testing to identify, reproduce and fix issues you might encounter.

Customize the loggers to get exactly the right level of information, in the right formats.

Learn more »

Customize REST requests

Intercept the outgoing REST calls the Ping SDKs make to customize or add data that is important to you or your environment.

Learn more »

Customize how the Ping SDKs store data

There are use cases where you might need to customize how to store data. For example, you might be running on hardware that provides specialized security features.

For these cases, you can provide your own storage classes.

Learn more »

Verify servers with SSL/certificate pinning

The Ping SDKs support *SSL pinning*, sometimes referred to as *certificate pinning*.

SSL pinning is the security practice of validating the certificates presented by the server against known values, improving the security of your system.

Learn more »

Configure Ping SDK properties

Applies to:

- ✓ Ping SDK for Android
- ✓ Ping SDK for iOS
- ✓ Ping SDK for JavaScript

You need to configure certain settings in the SDKs so they can connect to your authorization server to authenticate your users and obtain tokens.

The method you use to configure these settings depends on which SDK you are using.

Ping SDK for Android

Configure Ping SDK for Android properties

Ping SDK for iOS

Configure Ping SDK for iOS properties

Ping SDK for JavaScript

Configure Ping SDK for JavaScript properties

Configure Ping SDK for Android properties

Applies to:

- ✓ Ping SDK for Android
- ✕ Ping SDK for iOS
- ✕ Ping SDK for JavaScript

To configure the Ping SDK for Android, use the `FROptionsBuilder` methods to build an `FROptions` object, and pass the object to the `FRAuth.start()` method.

Properties

The following properties are available for configuring the Ping SDK for Android:

Server

`FROptionsBuilder` *attribute*

`server`

Properties

Property name	Description	Required
Java `setUrl` ***Kotlin*** `url`	The base URL of the PingAM instance to connect to, including port and deployment path. *Identity Cloud example*: `https://openam-forgerock-sdks.forgeblocks.com/am` *Self-hosted example*: `https://openam.example.com:8443/openam`	✓[1]
Java `setRealm` ***Kotlin*** `realm`	The realm in which the OAuth 2.0 client profile and authentication journeys are configured. For example, `alpha`. Defaults to the self-hosted top-level realm `root`.	✓[1]
Java `setTimeout` ***Kotlin*** `timeout`	A timeout, in seconds, for each request that communicates with PingAM. *Default*: `30`	✗
Java `setCookieName` ***Kotlin*** `cookieName`	The name of the cookie that contains the session token. For example, with a self-hosted PingAM server this value might be `iPlanetDirectoryPro`. **Tip** PingOne Advanced Identity Cloud tenants use a random alpha-numeric string. To locate the cookie name in an PingOne Advanced Identity Cloud tenant, navigate to **Tenant settings** > **Global Settings**, and copy the value of the **Cookie** property. *Default*: `iPlanetDirectoryPro`	✓[1]
Java `setCookieCache` ***Kotlin*** `cookieCache`	Time, in seconds, to cache the session token cookie in memory. *Default*: `0`	✗

Journeys

`FROptionsBuilder` *attribute*

`service`

Properties

Property name	Description	Required
Java `setAuthService` ***Kotlin*** `authService`	The name of a user ***authentication*** tree configured in your server. For example, `sdkUsernamePasswordJourney`.	✕
Java `setRegistrationService` ***Kotlin*** `registrationService`	The name of a user ***registration*** tree configured in your server. For example, `sdkRegistrationJourney`.	✕

OAuth 2.0

`FROptionsBuilder` *attribute*

`oauth`

Properties

Property name	Description	Required
Java `setOauthClientId` ***Kotlin*** `oauthClientId`	The `client_id` of the OAuth 2.0 client profile to use. For example, `sdkNativeClient`.	✕ [1]
Java `setOauthRedirectUri` ***Kotlin*** `oauthRedirectUri`	The `redirect_uri` as configured in the OAuth 2.0 client profile. **Important** This value must match a value configured in your OAuth 2.0 client. For example, `org.forgerock.demo://oauth2redirect`.	✕ [1]
Java `setOauthSignOutRedirectUri` ***Kotlin*** `oauthSignOutRedirectUri`	The URI to redirect to after signing the user out of the authorization server. For example, `org.forgerock.demo://oauth2redirect`.	✕ [1]

Property name	Description	Required
Java `SetOauthScope` ***Kotlin*** `oauthScope`	A list of scopes to request when performing an OAuth 2.0 authorization flow, separated by spaces. For example, `openid profile email address`.	✕ [1]
Java `setOauthThreshold` ***Kotlin*** `oauthThreshold`	A threshold, in seconds, to refresh an OAuth 2.0 token before the `access_token` expires (defaults to `30` seconds).	✕
Java `setOauthCache` ***Kotlin*** `oauthCache`	Time, in seconds, to cache an OAuth 2.0 token in memory (defaults to `0` seconds).	✕

Storage

`FROptionsBuilder` *attribute*

`store`

Properties

Property name	Description	Required
Java `setOidcStorage` ***Kotlin*** `oidcStorage`	A custom class for the storage of OpenID Connect-related items, such as access tokens.	✕
Java `SetSsoTokenStorage` ***Kotlin*** `ssoTokenStorage`	A custom class for the storage of single sign-on-related items, such as SSO tokens.	✕

Property name	Description	Required
Java `SetCookiesStorage` ***Kotlin*** `cookiesStorage`	A custom class for the storage of cookies.	×

SSL pinning

`FROptionsBuilder` *attribute*

`sslPinning`

Properties

Property name	Description	Required
Java `setPins` ***Kotlin*** `pins`	An array of public key certificate hashes (strings) for trusted sites and services.	×
Java `setBuildSteps` ***Kotlin*** `buildSteps`	An array of `BuildStep` objects to provide additional SSL pinning parameters to `OkHttpClient` instances.	×

Endpoints

`FROptionsBuilder` *attribute*

`urlPath`

Properties

Property name	Description	Required
Java `setAuthenticateEndpoint` ***Kotlin*** `authenticateEndpoint`	Override the path to the authorization server's `authenticate` endpoint. *Default*: `/json/realms/{forgerock_realm}/authenticate`	✕
Java `setAuthorizeEndpoint` ***Kotlin*** `authorizeEndpoint`	Override the path to the authorization server's `authorize` endpoint. *Default*: `/oauth2/realms/{forgerock_realm}/authorize`	✕
Java `setTokenEndpoint` ***Kotlin*** `tokenEndpoint`	Override the path to the authorization server's `access_token` endpoint. *Default*: `/oauth2/realms/{forgerock_realm}/access_token`	✕
Java `setRevokeEndpoint` ***Kotlin*** `revokeEndpoint`	Override the path to the authorization server's `revoke` endpoint. *Default*: `/oauth2/realms/{forgerock_realm}/token/revoke`	✕
Java `setUserinfoEndpoint` ***Kotlin*** `userinfoEndpoint`	Override the path to the authorization server's `userinfo` endpoint. *Default*: `/oauth2/realms/{forgerock_realm}/userinfo`	✕
Java `setSessionEndpoint` ***Kotlin*** `sessionEndpoint`	Override the path to the authorization server's `sessions` endpoint.	✕

> **Note**
>
> Session and token lifecycle
> The SDK revokes and removes persisted tokens if you programmatically change any of the following properties:
>
> - `setUrl` / `url`
> - `setRealm` / `realm`
> - `setCookieName` / `cookieName`
> - `setOauthClientId` / `oauthClientId`
> - `setOauthRedirectUri` / `oauthRedirectUri`
> - `setOauthScope` / `oauthScope`

Examples

The following examples show how to configure the Ping SDK in your Android applications:

Android - Java

```
FROptions options = FROptionsBuilder.build(frOptionsBuilder -> {
    frOptionsBuilder.server(serverBuilder -> {
        serverBuilder.setUrl("https://tenant.forgeblocks.com/am");
        serverBuilder.setRealm("alpha");
        serverBuilder.setCookieName("46b42b4229cd7a3");
        return null;
    });
    frOptionsBuilder.oauth(oAuthBuilder -> {
        oAuthBuilder.setOauthClientId("androidClient");
        oAuthBuilder.setOauthRedirectUri("https://localhost:8443/callback");
        oAuthBuilder.setOauthScope("openid profile email address");
        return null;
    });
    frOptionsBuilder.service(serviceBuilder -> {
        serviceBuilder.setAuthServiceName("Login");
        serviceBuilder.setRegistrationServiceName("Registration");
        return null;
    });
    return null;
});
FRAuth.start(this, options);
```

Android - Kotlin

```
val options = FROptionsBuilder.build {
    server {
       url = "https://openam-forgerock-sdks.forgeblocks.com/am"
       realm = "alpha"
       cookieName = "iPlanetDirectoryPro"
    }
    oauth {
       oauthClientId = "sdkPublicClient"
       oauthRedirectUri = "https://localhost:8443/callback"
       oauthScope = "openid profile email address"
    }
    service {
       authServiceName = "Login"
       registrationServiceName = "Registration"
    }
}

FRAuth.start(this, options);
```

When the application calls `FRAuth.start()`, the `FRAuth` class checks for the presence of an `FROptions` object. If the object is not present, static initialization from `strings.xml` happens. If the object is present, the `FRAuth` class uses the options object and calls the same internal initialization method.

The app can call `FRAuth.start()` multiple times in its lifecycle:

- When the app calls `FRAuth.start()` for the first time in its lifecycle, the SDK checks for the presence of session and access tokens in the local storage. If an existing session is present, initialization does not log the user out.
- If the app calls `FRAuth.start()` again, the SDK checks whether session managers and token managers are initialized, and cleans the existing session and token storage. This ensures that changes to the app configuration remove and revoke existing sessions and tokens.

Using the .well-known endpoint

You can configure the SDKs to obtain many required settings from your authorization server's `.well-known` OpenID Connect endpoint.

Settings gathered from the endpoint include the paths to use for OAuth 2.0 authorization requests, and login endpoints.

Use the `FROptions.discover` method to use the `.well-known` endpoint to configure OAuth 2.0 paths:

```
val options =
    options.discover("https://openam-forgerock-sdks.forgeblocks.com/am/oauth2/realms/root/realms/alpha/.well-known/
openid-configuration")

FRAuth.start(context, options)
```

Configure Ping SDK for iOS properties

Applies to:

- ✕ Ping SDK for Android
- ✓ Ping SDK for iOS
- ✕ Ping SDK for JavaScript

Use the `FROptions` interface to build an options object and pass the object to the `FRAuth.start()` method.

Properties

The following properties are available for configuring the Ping SDK for iOS:

Server

Properties

Property name	Description	Required
FROptions `url` ***Properties file*** `forgerock_url`	The base URL of the PingAM instance to connect to, including port and deployment path. *Identity Cloud example*: `https://openam-forgerock-sdks.forgeblocks.com/am` *Self-hosted example*: `https://openam.example.com:8443/openam`	✓[1]
FROptions `realm` ***Properties file*** `forgerock_realm`	The realm in which the OAuth 2.0 client profile and authentication journeys are configured. For example, `alpha`. Defaults to the self-hosted top-level realm `root`.	✓[1]
FROptions `timeout` ***Properties file*** `forgerock_timeout`	A timeout, in seconds, for each request that communicates with PingAM. *Default*: `30`	✕

Property name	Description	Required
FROptions `cookieName` ***Properties file*** `forgerock_cookie_name`	The name of the cookie that contains the session token. For example, with a self-hosted PingAM server this value might be `iPlanetDirectoryPro`. **Tip** PingOne Advanced Identity Cloud tenants use a random alpha-numeric string. To locate the cookie name in an PingOne Advanced Identity Cloud tenant, navigate to **Tenant settings** > **Global Settings**, and copy the value of the **Cookie** property. *Default*: `iPlanetDirectoryPro`	✓[1]
FROptions `enableCookie` ***Properties file*** `forgerock_enable_cookie`	When `true`, enables cookie use. *Default*: `true`	✕

Journeys

Properties

FROptions `authServiceName` ***Properties file*** `forgerock_auth_service_name`	The name of a user ***authentication*** tree configured in your server. For example, `sdkUsernamePasswordJourney`.	✕
FROptions `registrationServiceName` ***Properties file*** `forgerock_registration_service_name`	The name of a user ***registration*** tree configured in your server. For example, `sdkRegistrationJourney`.	✕

OAuth 2.0

Properties

FROptions `oauthClientId` ***Properties file*** `forgerock_oauth_client_id`	The `client_id` of the OAuth 2.0 client profile to use. For example, `sdkNativeClient`.	✕ [1]
FROptions `oauthRedirectUri` ***Properties file*** `forgerock_oauth_redirect_uri`	The `redirect_uri` as configured in the OAuth 2.0 client profile. **Important** This value must match a value configured in your OAuth 2.0 client. For example, `org.forgerock.demo://oauth2redirect`.	✕ [1]
FROptions `oauthSignoutRedirectUri` ***Properties file*** `forgerock_oauth_sign_out_redirect_uri`	The URI to redirect to after signing the user out of the authorization server. For example, `org.forgerock.demo://oauth2redirect`.	✕
FROptions `oauthScope` ***Properties file*** `forgerock_oauth_scope`	A list of scopes to request when performing an OAuth 2.0 authorization flow, separated by spaces. For example, `openid profile email address`.	✕ [1]
FROptions `oauthThreshold` ***Properties file*** `forgerock_oauth_threshold`	A threshold, in seconds, to refresh an OAuth 2.0 token before the `access_token` expires (defaults to `30` seconds).	✕

SSL pinning

Properties

FROptions `sslPinningPublicKeyHashes` ***Properties file*** `forgerock_ssl_pinning_public_key_hashes`	An array of public key certificate hashes (strings) for trusted sites and services.	×
FROptions `keychainAccessGroup` ***Properties file*** `forgerock_keychain_access_group`	Keychain access group for the shared keychain.	×

Endpoints

Properties

FROptions `authenticateEndpoint` ***Properties file*** `forgerock_authenticate_endpoint`	Override the path to the authorization server's `authenticate` endpoint. *Default*: `/json/realms/{forgerock_realm}/authenticate`	×
FROptions `authorizeEndpoint` ***Properties file*** `forgerock_authorize_endpoint`	Override the path to the authorization server's `authorize` endpoint. *Default*: `/oauth2/realms/{forgerock_realm}/authorize`	×
FROptions `tokenEndpoint` ***Properties file*** `forgerock_token_endpoint`	Override the path to the authorization server's `access_token` endpoint. *Default*: `/oauth2/realms/{forgerock_realm}/access_token`	×

FROptions `revokeEndpoint` ***Properties file*** `forgerock_revoke_endpoint`	Override the path to the authorization server's `token/revoke` endpoint. *Default*: `/oauth2/realms/{forgerock_realm}/token/revoke`	✕
FROptions `userinfoEndpoint` ***Properties file*** `forgerock_userinfo_endpoint`	Override the path to the authorization server's `userinfo` endpoint. *Default*: `/oauth2/realms/{forgerock_realm}/userinfo`	✕
FROptions `sessionEndpoint` ***Properties file*** `forgerock_session_endpoint`	Override the path to the authorization server's `sessions` endpoint.	✕
FROptions `endSessionEndpoint` ***Properties file*** `forgerock_endsession_endpoint`	Override the path to the authorization server's `endSession` endpoint.	✕

Note

Session and token lifecycle
The SDK revokes and removes persisted tokens if you programmatically change any of the following properties:

- `url`
- `realm`
- `cookieName`
- `oauthClientId`
- `oauthRedirectUri`
- `oauthScope`

Example

The following Swift example shows how to configure the Ping SDK in your iOS applications:

```
let options = FROptions(
  url: "https://tenant.forgeblocks.com/am",
  realm: "alpha",
  cookieName: "46b42b4229cd7a3",
  oauthClientId: "sdkNativeClient",
  oauthRedirectUri: "org.forgerock.demo://oauth2redirect",
  oauthScope: "openid profile email address",
  authServiceName: "Login",
  registrationServiceName: "Register")
try FRAuth.start(options: options)
```

When the application calls `FRAuth.start()`, the `FRAuth` class checks for the presence of an `FROptions` object.

If the object is not present, the static initialization from `FRAuthConfig.plist` happens.

If the object is present, the `FRAuth` class converts it to a `[String, Any]` dictionary and calls the same internal initialization method.

The app can call `FRAuth.start()` multiple times in its lifecycle:

- When the app calls `FRAuth.start()` for the first time in its lifecycle, the SDK checks for the presence of session and access tokens in the local storage.

  If an existing session is present, initialization does not log the user out.

- If the app calls `FRAuth.start()` again, the SDK checks whether session managers and token managers are initialized, and cleans the existing session and token storage.

  This ensures that changes to the app configuration remove and revoke existing sessions and tokens.

Using the .well-known endpoint

You can configure the SDKs to obtain many required settings from your authorization server's `.well-known` OpenID Connect endpoint.

Settings gathered from the endpoint include the paths to use for OAuth 2.0 authorization requests, and login endpoints.

Use the `FROptions.discover` method to use the `.well-known` endpoint to configure OAuth 2.0 paths:

```
let options = try await FROptions(config: config).discover(
  discoveryURL: "https://openam-forgerock-sdks.forgeblocks.com/am/oauth2/realms/root/realms/alpha/.well-known/openid-
configuration")

try FRAuth.start(options: options)
```

Configure Ping SDK for JavaScript properties

Applies to:

- ✕ Ping SDK for Android
- ✕ Ping SDK for iOS
- ✓ Ping SDK for JavaScript

Configure SDK properties in your JavaScript app by editing a `serverConfig` object, a parameter of the `forgerock.Config.set()` function.

Properties

The following properties are available for configuring the Ping SDK for JavaScript:

Server

Properties

Property	Description
`serverConfig`	An interface for configuring how the SDK contacts the PingAM instance. Contains `baseUrl` and `timeout`.

`serverConfig: {baseUrl}`	The base URL of the server to connect to, including port and deployment path.
	Identity Cloud example:
	`https://openam-forgerock-sdks.forgeblocks.com/am`
	Self-hosted example:
	`https://openam.example.com:8443/openam`

`serverConfig: {wellknown}`	A URL to the server's `.well-known/openid-configuration` endpoint.
	Use the `Config.setAsync()` method to set SDK configuration using values derived from those provided at the URL.
	Example:
	`https://openam-forgerock-sdks.forgeblocks.com/am/oauth2/realms/root/realms/alpha/.well-known/openid-configuration`
	Self-hosted example:
	`https://openam.example.com:8443/openam/oauth2/realms/root/.well-known/openid-configuration`

`serverConfig: {timeout}`	A timeout, in milliseconds, for each request that communicates with your server. For example, for 30 seconds specify `30000`. Defaults to `5000` (5 seconds).

Property	Description
`realmPath`	The realm in which the OAuth 2.0 client profile and authentication journeys are configured. For example, `alpha`. Defaults to the self-hosted top-level realm `root`.
`tree`	The name of the user ***authentication*** tree configured in your server. For example, `sdkUsernamePasswordJourney`.

OAuth 2.0

Properties

Property	Description
`clientId`	The `client_id` of the OAuth 2.0 client profile to use.
`redirectUri`	The `redirect_uri` as configured in the OAuth 2.0 client profile. **Tip** The Ping SDK for JavaScript attempts to load the redirect page to capture the OAuth 2.0 `code` and `state` query parameters that the server appended to the redirect URL. If the page you redirect to does not exist, takes a long time to load, or runs any JavaScript you might get a timeout, delayed authentication, or unexpected errors. To ensure the best user experience, we ***highly recommend*** that you redirect to a static HTML page with minimal HTML and no JavaScript when obtaining OAuth 2.0 tokens. For example, `https://localhost:8443/callback.html`.
`scope`	A list of scopes to request when performing an OAuth 2.0 authorization flow, separated by spaces. For example, `openid profile email address`.
`oauthThreshold`	A threshold, in seconds, to refresh an OAuth 2.0 token before the `access_token` expires. Defaults to `30` seconds.

Storage

Properties

Property	Description
`tokenStore`	The API to use for storing tokens on the client: **`sessionStorage`** Store tokens using the `sessionStorage` API. The browser clears session storage when a page session ends. **`localStorage`** Store tokens using the `localStorage` API. The browser saves local storage data across browser sessions. This is the default setting, as it provides the highest browser compatibility. **`{{custom}}`** Specify a custom implementation that has functions that can set, retrieve, and remove, items from a custom storage scheme. Learn more in Customize storage on JavaScript.
`prefix`	Override the default `fr` prefix string applied to the keys used for storing data on the client, such as tokens, device IDs, and information about the steps in a journey. For example, the key used for storing tokens consists of the `prefix`, followed by the ID of the OAuth 2.0 client: `fr-sdkPublicClient`.

Logging

Properties

Property	Description
`logLevel`	Specify whether the SDK should output its log messages in the console and the level of messages to display. One of: • `none` (default) • `info` • `warn` • `error` • `debug`
`logger`	Specify a function to override the default logging behavior. Refer to Customize the Ping SDK for JavaScript logger.

General

Properties

Property	Description
`platformHeader`	Specify whether to include an `X-Requested-Platform` header in outgoing requests. The server can use the value of this header to alter the logic of an authentication flow. For example, if the value indicates a JavaScript web app, the journey could avoid device binding nodes, as they are only supported by Android and iOS apps. Defaults to `false`.

Endpoints

Properties

Property	Description
`serverConfig: { paths: { authenticate }}`	Override the path to the authorization server's `authenticate` endpoint. *Default*: `json/{realmPath}/authenticate`
`serverConfig: { paths: { authorize }}`	Override the path to the authorization server's `authorize` endpoint. *Default*: `oauth2/{realmPath}/authorize`
`serverConfig: { paths: { accessToken }}`	Override the path to the authorization server's `access_token` endpoint. *Default*: `oauth2/{realmPath}/access_token`
`serverConfig: { paths: { revoke }}`	Override the path to the authorization server's `revoke` endpoint. *Default*: `oauth2/{realmPath}/token/revoke`
`serverConfig: { paths: { userInfo }}`	Override the path to the authorization server's `userinfo` endpoint. *Default*: `oauth2/{realmPath}/userinfo`
`serverConfig: { paths: { sessions }}`	Override the path to the authorization server's `sessions` endpoint. *Default*: `json/{realmPath}/sessions`
`serverConfig: { paths: { endSession }}`	Override the path to the authorization server's `endSession` endpoint. *Default*: `oauth2/{realmPath}/connect/endSession`

Examples

The following examples show how to configure the Ping SDK in your JavaScript applications:

```
forgerock.Config.set({
    serverConfig: {
        baseUrl: 'https://openam-forgerock-sdks.forgeblocks.com/am',
        timeout: 3000,
        paths: {
            authenticate: 'iam/endpoints/authN',
            authorize: 'iam/endpoints/authZ'
        },
    },
    clientId: 'sdkPublicClient',
    scope: 'openid profile email address',
    redirectUri: `${window.location.origin}/callback.html`,
    realmPath: 'alpha'
});
```

Using the .well-known endpoint

You can configure the SDKs to obtain many required settings from your authorization server's `.well-known` OpenID Connect endpoint.

Settings gathered from the endpoint include the paths to use for OAuth 2.0 authorization requests, and login endpoints.

Use the `Config.setAsync` method to use the `.well-known` endpoint to configure OAuth 2.0 paths:

```
await Config.setAsync({
  serverConfig: {
    wellknown: 'https://openam-forgerock-sdks.forgeblocks.com/am/oauth2/realms/root/realms/alpha/.well-known/openid-
configuration'
  },
  clientId: 'sdkPublicClient',
  scope: 'openid profile email address',
  redirectUri: `${window.location.origin}/callback.html`
});
```

Configure logging

Applies to:

- ✓ Ping SDK for Android
- ✓ Ping SDK for iOS
- ✓ Ping SDK for JavaScript

This page covers how to use the default logging in the Ping SDKs, and how to customize logging.

Ping SDK for Android

Configure Ping SDK for Android logging

Ping SDK for iOS

Configure Ping SDK for iOS logging

Ping SDK for JavaScript

Configure Ping SDK for JavaScript logging

Android

Configure default Android logging

The Ping SDK for Android does all of its logging through a custom interface called `FRLogger`. The default implementation of this interface logs the messages through the native Android Log class. This displays messages from the SDK in real-time in the **Logcat** window in Android Studio.

The log severity levels defined in the Ping SDK for Android are as follows:

Log level	Description
`DEBUG`	Show debug log messages intended only for development, as well as the message levels lower in this list; `INFO`, `WARN`, and `ERROR`. In addition, all network activities of the SDK are included in the logs.
`INFO`	Show expected log messages for regular usage, as well as the message levels lower in this list, `WARN`, and `ERROR`.
`WARN`	Show possible issues that are not yet errors, as well as the messages of `ERROR` log level.
`ERROR`	Show issues that caused errors.

Log level	Description
`NONE`	No log messages are shown.

> **Note**
>
> The log levels are cumulative.
> If you select a lower severity level, all messages logged at higher severity levels are also included. For example, if you select the `DEBUG` level, the log includes all events logged at the `DEBUG`, `INFO`, `WARN`, and `ERROR` levels.
> By default, the log level of the Ping SDK for Android is set to `Logger.Level.WARN`.

Customize Android logging

The Ping SDK for Android allows developers to customize the default logger behavior:

1. Create a class that implements the `FRLogger` interface:

```
import androidx.annotation.Nullable;
import org.forgerock.android.auth.FRLogger;

public class MyCustomLogger implements FRLogger {
    @Override
    public void error(@Nullable String tag, @Nullable Throwable t, @Nullable String message, @Nullable
Object... values) {
        /// Custom error message handling...
    }

    @Override
    public void error(@Nullable String tag, @Nullable String message, @Nullable Object... values) {
        /// Custom error message handling...
    }

    @Override
    public void warn(@Nullable String tag, @Nullable String message, @Nullable Object... values) {
        /// Custom warning message handling...
    }

    @Override
    public void warn(@Nullable String tag, @Nullable Throwable t, @Nullable String message, @Nullable
Object... values) {
        /// Custom warning message handling...
    }

    @Override
    public void debug(@Nullable String tag, @Nullable String message, @Nullable Object... values) {
        /// Custom debug message handling...
    }

    @Override
    public void info(@Nullable String tag, @Nullable String message, @Nullable Object... values) {
        /// Custom info message handling...
    }

    @Override
    public void network(@Nullable String tag, @Nullable String message, @Nullable Object... values) {
        /// Custom network details handling...
    }

    @Override
    public boolean isNetworkEnabled() {
        return true; // include network call details in the logs
    }
}
```

2. In your application, set the custom logger and desired log level:

```
Logger.setCustomLogger(new MyCustomLogger()); // The default logger will no longer be active
Logger.set(Logger.Level.DEBUG);
```

3. You can now use the `Logger` interface in your app.

   For example:

```
String TAG = MainActivity.class.getSimpleName();
Logger.debug (TAG, "Happy logging!");
```

iOS

Configure the default iOS logging

The Ping SDK for iOS does all of its logging through a custom protocol called `FRLogger`. The default implementation of the `FRLogger` protocol logs the messages through the native iOS FRConsoleLogger class. This displays messages from the SDK in real-time in the console window in Xcode.

Each log message has an associated log level that describes the type and the severity of the message. Log levels are helpful tool for tracking and analyzing events that take place in your app.

The log severity levels defined in the Ping SDK for iOS are as follows:

Log level	Description
`none`	Prevent logging
`verbose`	Logs that are not important or can be ignored
`info`	Logs that maybe helpful or meaningful for debugging, or understanding the flow
`network`	Logs for network traffic, including request and response
`warning`	Logs that are a minor issue or an error that can be ignored
`error`	Logs that are a severe issue or a major error that impacts the SDK's functionality or flow
`all`	Logs at all levels

> **Note**
>
> The log levels are **not** cumulative. That is, you should explicitly specify all the log levels you want to record.
> For example, if you select the `debug` level, the output only includes events logged at `debug` level.
> To include other levels, you must specify an array of the required log levels.
> By default, the log level of the Ping SDK for iOS is set to `LogLevel.none`.

Customize iOS logging

The Ping SDK for iOS lets developers customize the default logger behavior:

1. Create a class that conforms to the `FRLogger` protocol:

```
class MyCustomLogger: FRLogger {
    func logVerbose(timePrefix: String, logPrefix: String, message: String) {
        /// Custom verbose message handling...
    }

    func logInfo(timePrefix: String, logPrefix: String, message: String) {
        /// Custom info message handling...
    }

    func logNetwork(timePrefix: String, logPrefix: String, message: String) {
        /// Custom network message handling...
    }

    func logWarning(timePrefix: String, logPrefix: String, message: String) {
        /// Custom warning message handling...
    }

    func logError(timePrefix: String, logPrefix: String, message: String) {
        /// Custom error message handling...
    }
}
```

2. In your application, set the custom logger and desired log level:

```
FRLog.setCustomLogger(MyCustomLogger()) // The default logger will no longer be active
FRLog.setLogLevel([.all])
```

3. You can now use the `FRLog` class in your app.

   For example:

```
FRLog.v("Happy logging!")
```

JavaScript

Configure the default JavaScript logging

The Ping SDK for JavaScript performs logging through the native `console` class. This displays messages from the SDK in real-time in the console window provided in many browsers.

The default `logLevel` is `none`, which prevents the Ping SDK for JavaScript from logging any messages to the console.

To enable the output of log messages from the Ping SDK for JavaScript, specify a `logLevel` value other than `none`.

For example, use the following code to specify the `debug` level:

Setting the log level in the Ping SDK for JavaScript configuration

```
Config.set({
  serverConfig: {
    baseUrl: 'https://openam-forgerock-sdks.forgeblocks.com/am/',
    timeout: 5000,
  },
  logLevel: 'debug',
});
```

The log severity levels defined in the Ping SDK for JavaScript are as follows:

Log level	Description
`debug`	Show debug log messages intended only for development, as well as the message levels lower in this list; `info`, `warn`, and `error`. In addition, all network activities of the SDK are included in the logs.
`info`	Show expected log messages for regular usage, as well as the message levels lower in this list, `warn`, and `error`.
`warn`	Show possible issues that are not yet errors, as well as the messages of `error` log level.
`error`	Show issues that caused errors.
`none`	No log messages are shown. This is the default setting.

Note

The log levels are cumulative. If you select a lower severity level, all messages logged at higher severity levels are also included.
For example, if you select the `debug` level, the output includes all events logged by the SDK at `debug`, `info`, `warn`, and `error` levels.

For more information on configuring the Ping SDK for JavaScript, refer to Ping SDK for JavaScript Properties

Customize JavaScript logging

The Ping SDK for JavaScript allows developers to customize the default logger behavior. For example, you might want to redirect the logs to an external service.

1. Create a function that implements the `LoggerFunctions` interface.

   For example, the following code adds a prefix to each log message from the SDK and logs it to the console:

```
const customLogger = {
  warn: (msg) => console.warn(`[FR SDK] ${msg}`),
  error: (msg) => console.error(`[FR SDK] ${msg}`),
  log: (msg) => console.log(`[FR SDK] ${msg}`),
  info: (msg) => console.info(`[FR SDK] ${msg}`),
};
```

The signature of the interface defaults to the following:

`(…msgs: unknown[]) ⇒ void`

You can pass your own type definition into the Generic if required. For example:

```
// typescript generic example
type YourAsyncLoggerType = LoggerFunctions<
  (...msgs: unknown[]) => Promise<void>,
  (...msgs: unknown[]) => Promise<void>,
  (...msgs: unknown[]) => Promise<void>,
  (...msgs: unknown[]) => Promise<void>
  >

const customLoggerWithApiCall: YourAsyncLoggerType = {
  warn: (msg) => yourAsyncLogFunction.warn(`[FR SDK] ${msg}`),
  error: (msg) => yourAsyncLogFunction.error(`[FR SDK] ${msg}`),
  log: (msg) => yourAsyncLogFunction.log(`[FR SDK] ${msg}`),
  info: (msg) => yourAsyncLogFunction.info(`[FR SDK] ${msg}`),
};
```

2. In the SDK configuration of your app, specify the custom logger and required log level:

```
Config.set({
    serverConfig: {
        baseUrl: 'https://openam-forgerock-sdks.forgeblocks.com/am/',
        timeout: '5000'
    },
    logLevel: 'error',
    logger: customLogger,
});
```

The SDK redirects its logging output to your custom handler.

Customize REST calls

Applies to:

- ✓ Ping SDK for Android
- ✓ Ping SDK for iOS
- ✓ Ping SDK for JavaScript

The Ping SDKs support modification of REST calls before they are sent.

For example, you can add or customize:

- Query parameters
- Headers
- Cookies
- Request URLs
- Request methods
- Body and post data

Request interceptors

Each SDK provides an interface that you can use to customize requests:

Android

```
public interface FRRequestInterceptor<Action> {
    @NonNull Request intercept(Request request, Action action);
}

public class Action {
    private String type;
    private JSONObject payload;
}
```

iOS

```
public protocol RequestInterceptor {
    func intercept(request: Request, action: Action) -> Request
}

public struct Action {
    public let type: String
    public let payload: [String: Any]?
}
```

JavaScript

```
type RequestMiddleware = (req: RequestObj, action: Action, next: () => RequestObj) => void;

interface RequestObj {
  url: URL;
  init: RequestInit;
}

interface Action {
  type: string;
  payload?: any; // optional data
}
```

Request interceptors have two inputs:

- The `Request` object
- The `Action` object

The `Request` object

Represents the original request, and has information about the body, method type, parameters, and more.

Android

```
public class Request {
    public URL url();
    public Iterator<Pair<String, String>> headers();
    public String header(String name);
    public List<String> headers(String name);
    public String method();
    public Object tag();
    public Body body();
    public Builder newBuilder();
}

// Use Build to build upon on existing Request
public class Builder {
   public Builder url(URL url);
   public Builder url(String url);
   public Builder header(String name, String value);
   public Builder addHeader(String name, String value);
   public Builder removeHeader(String name);
   public Builder get();
   public Builder put(Body body);
   public Builder post(Body body);
   public Builder delete(Body body);
   public Builder delete();
   public Builder patch(Body body);
   public Request build();

}
```

iOS

```
public struct Request {
    // Properties
    public let url: String
    public let method: HTTPMethod
    private(set) public var headers: [String: String]
    public let bodyParams: [String: Any]
    public let urlParams: [String: String]
    public let responseType: ContentType
    public let requestType: ContentType
    public let timeoutInterval: Double

    public enum ContentType: String {
        case plainText = "text/plain"
        case json = "application/json"
        case urlEncoded = "application/x-www-form-urlencoded"
    }

    public enum HTTPMethod: String {
        case GET = "GET"
        case PUT = "PUT"
        case POST = "POST"
        case DELETE = "DELETE"
    }

    public func build() -> URLRequest?
}
```

JavaScript

Refer to the native JavaScript `Request` object in the *MDN Web Docs*.

The `Action` object

Represents the type of operation the request performs:

Action	Description
`START_AUTHENTICATE`	Initial call to an authentication tree
`AUTHENTICATE`	Proceed through an authentication tree flow
`AUTHORIZE`	Obtain authorization token from PingAM
`EXCHANGE_TOKEN`	Exchange authorization code for an access token
`REFRESH_TOKEN`	Refresh an access token

Action	Description
`REVOKE_TOKEN`	Revoke a refresh or access token
`LOGOUT`	Log out a session
`USER_INFO`	Obtain information from the `userinfo` endpoint
`PUSH_REGISTER`	Register a push device with PingAM; for example, a call to `/json/push/sns/message?_action=register`
`PUSH_AUTHENTICATE`	Authenticate using push; for example, a call to `/json/push/sns/message?_action=authenticate`

Note

The `AUTHENTICATE` and `START_AUTHENTICATE` actions have a payload that contains:

`tree`
: The name of the authentication tree being called.

`type`
: Whether the call is to a `service`, or is in response to `composite_advice`.

The outcome of applying a request interceptor is the entire modified request object, ready to either be sent to PingAM, or to have additional request interceptors applied.

Examples

This section covers how to develop request interceptors, referred to as "middleware" in the Ping SDK for JavaScript, and apply them to outbound requests from your applications.

Ping SDK for Android

Query parameters and headers

The example sets the `ForceAuth` query parameter to `true`, and adds an `Accept-Language` header with a value of `en-GB` on all outgoing requests of the `START_AUTHENTICATE` type:

```
public class QueryParamsAndHeaderRequestInterceptor implements FRRequestInterceptor<Action> {
    @NonNull
    @Override
    public Request intercept(@NonNull Request request, Action tag) {
        if (tag.getType().equals(START_AUTHENTICATE)) {
            return request.newBuilder()
                    // Add query parameter:
                    .url(Uri.parse(request.url().toString())
                            .buildUpon()
                            .appendQueryParameter("ForceAuth", "true").toString())

                    // Add additional header:
                    .addHeader("Accept-Language", "en-GB")

                    // Construct the updated request:
                    .build();
        }
        return request;
    }
}
```

To register the request interceptor, use the `RequestInterceptorRegistry.getInstance().register()` method:

```
RequestInterceptorRegistry.getInstance().register(new QueryParamsAndHeaderRequestInterceptor())
```

Any calls the app makes to initiate authentication now have the query parameter `ForceAuth=true` appended, and include an `accept-language: en-GB` header added.

Cookies

The example adds a custom cookie to outgoing requests:

```
public class CustomCookieInterceptor implements FRRequestInterceptor<Action>, CookieInterceptor {
    @NonNull
    @Override
    public Request intercept(@NonNull Request request) {
       return request;
    }

    @NonNull
    @Override
    public Request intercept(@NonNull Request request, Action tag) {
        return request;
    }

    @NonNull
    @Override
    public List<Cookie> intercept(@NonNull List<Cookie> cookies) {
        List<Cookie> newCookies = new ArrayList<>();
        newCookies.addAll(cookies);
        newCookies.add(
          new Cookie.Builder()
            .domain("example.com")
            .name("member").value("gold")
            .httpOnly().secure().build()
        );

        return newCookies;
    }
}
```

Tip

You can register multiple request interceptors as follows:

```
RequestInterceptorRegistry.getInstance().register(
    new QueryParamsAndHeaderRequestInterceptor(),
    new CustomCookieInterceptor()
);
```

Ping SDK for iOS

Query parameters and headers

The example sets the `ForceAuth` query parameter to `true`, and adds an `Accept-Language` header with a value of `en-GB` on all outgoing requests of the `AUTHENTICATE` or `START_AUTHENTICATE` type:

```
class QueryParamsAndHeaderRequestInterceptor: RequestInterceptor {
    func intercept(request: Request, action: Action) -> Request {
        if action.type == "START_AUTHENTICATE" || action.type == "AUTHENTICATE" {
            // Add query parameter:
            var urlParams = request.urlParams
            urlParams["ForceAuth"] = "true"

            // Add additional header:
            var headers = request.headers
            headers["Accept-Language"] = "en-GB"

            // Construct the updated request:
            let newRequest = Request(
                url: request.url,
                method: request.method,
                headers: headers,
                bodyParams: request.bodyParams,
                urlParams: urlParams,
                requestType: request.requestType,
                responseType: request.responseType,
                timeoutInterval: request.timeoutInterval
            )
            return newRequest
        }
        else {
            return request
        }
    }
}
```

To register the request interceptor, use the `registerInterceptors()` method:

```
FRRequestInterceptorRegistry.shared.registerInterceptors(
    interceptors: [
        QueryParamsAndHeaderRequestInterceptor()
    ]
)
```

Any calls the app makes to initiate authentication now have the query parameter `ForceAuth=true` appended, and include an `accept-language: en-GB` header added.

Cookies

The example adds a custom cookie to outgoing requests:

```
class CookieInterceptor: RequestInterceptor {
    func intercept(request: Request, action: Action) -> Request {
        if action.type == "START_AUTHENTICATE" || action.type == "AUTHENTICATE" {

            var headers = request.headers
            headers["Cookie"] = "member=gold; level=2"

            let newRequest = Request(
                url: request.url,
                method: request.method,
                headers: headers,
                bodyParams: request.bodyParams,
                urlParams: request.urlParams,
                requestType: request.requestType,
                responseType: request.responseType,
                timeoutInterval: request.timeoutInterval)
            return newRequest
        }
        else {
            return request
        }
    }
}
```

Tip

You can register multiple request interceptors as follows:

```
FRRequestInterceptorRegistry.shared.registerInterceptor(
    interceptors: [
        QueryParamsAndHeaderRequestInterceptor(),
        CookieInterceptor()
    ]
)
```

Ping SDK for JavaScript

The example has two middleware configurations. One sets the `ForceAuth` query parameter to `true`, the other adds an `Accept-Language` header with a value of `en-GB` on all outgoing requests of the `START_AUTHENTICATE` type:

```
const forceAuthMiddleware = (
    req: RequestObj,
    action: Action,
    next: () => RequestObj
): void => {
    switch (action.type) {
        case 'START_AUTHENTICATE':
            req.url.searchParams.set('ForceAuth', 'true');
            break;
    }
    next();
};

const addHeadersMiddleware = (
    req: RequestObj,
    action: Action,
    next: () => RequestObj
): void => {
    switch (action.type) {
        case 'START_AUTHENTICATE':
            const headers = req.init.headers as Headers;
            headers.append('Accept-Language', 'en-GB');
            break;
    }
    next();
};
```

Apply the middleware in the `config`:

```
Config.set({
    clientId: 'sdkPublicClient',
    middleware: [
        forceAuthMiddleware,
        addHeadersMiddleware
    ],
    redirectUri: 'https://localhost:8443/callback.html',
    realmPath: 'alpha',
    scope: 'openid profile email address',
    serverConfig: {
        baseUrl: 'https://openam-forgerock-sdks.forgeblocks.com/am',
        timeout: 30000
    },
    tree: 'UsernamePassword'
});
```

Any calls the app makes to start authentication now have the query parameter and header added.

> **Note**
>
> You can only modify headers in certain types of request.
> For example `START_AUTHENTICATE` and `AUTHENTICATE` types, but not `AUTHORIZE` types as they occur in an iframe.

More information

- Authentication parameters
- Authenticate endpoint parameters
- SDK troubleshooting

Customize storage

Applies to:

- ✓ Ping SDK for Android
- ✕ Ping SDK for iOS
- ✓ Ping SDK for JavaScript

Depending on the authentication use case, the SDKs may need to store and retrieve session cookies, ID tokens, access tokens, and refresh tokens.

Each token is serving a different use case, and as such how the SDKs handle them can be different.

The SDKs employ identity best practices for storing data by default. To learn more about how the SDKs store different data, refer to Token and key security and Data security.

There are use cases where you might need to customize how to store data. For example, you might be running on hardware that provides specialized security features, or perhaps target older hardware that cannot handle the latest algorithms.

For these cases, you can provide your own storage classes.

Customize storage on Android

You can configure your Android apps to use customized storage for these types of data:

1. OAuth 2.0 / OpenID Connect 1.0 tokens
2. SSO data
3. Cookies

> **Tip**
>
> Depending on why you want to override storage mechanisms, you might prefer instead to prevent use of StrongBox. Learn more in Preventing the Keystore System from using StrongBox.

Implement storage override classes

Use the `Storage` interface to override the different types of storage as follows

OpenID Connect storage

`Storage<AccessToken>`

SSO token storage

`Storage<SSOToken>`

Cookie storage

`Storage<Collection<String>>`

You must implement the following functions in each storage class:

`save()`

Stores an item in the customized storage.

`get()`

Retrieves an item from the customized storage.

`delete()`

Removes an item from the customized storage.

Examples:

OpenID Connect storage

```
class MyCustomTokenStorage(context: Context) : Storage<AccessToken> {

    override fun save(item: AccessToken) {
        TODO("Implement save to storage functionality")
    }

    override fun get(): AccessToken? {
        TODO("Implement retrieve to storage functionality")
    }

    override fun delete() {
        TODO("Implement remove from storage functionality")
    }
}
```

SSO token storage

```
class MyCustomSSOTokenStorage(context: Context) : Storage<SSOToken> {

    override fun save(item: SSOToken) {
        TODO("Implement save to storage functionality")
    }

    override fun get(): SSOToken? {
        TODO("Implement retrieve to storage functionality")
    }

    override fun delete() {
        TODO("Implement remove from storage functionality")
    }

}
```

Cookie storage

```
class MyCustomCookiesStorage() : Storage<Collection<String>> {

    override fun save(item: Collection<String>) {
        TODO("Implement save to storage functionality")
    }

    override fun get(): Collection<String>? {
        TODO("Implement retrieve to storage functionality")
    }

    override fun delete() {
        TODO("Implement remove from storage functionality")
    }
}
```

The SDK includes a basic example of a customized storage class that places data temporarily in memory. Refer to MemoryStorage.kt in the **forgerock-android-sdk** GitHub repo.

Important

Apps you release that use customized storage will not be able to access existing data that was stored using a different method.
Previous users of your app will have to log in again after upgrading to an app that is using a different storage mechanism.
To prevent having to log in again your custom storage could manually migrate any existing data to the new storage during initialization.
For an example of migrating existing stored data, see `SSOTokenStorage.kt`

Configure storage overrides

Add a `store` key to the `FROptionsBuilder.build` parameters to specify which storage types to override, and the class you created above that provides the implementation:

```
val options = FROptionsBuilder.build {
    server {
       url = "https://openam-forgerock-sdks.forgeblocks.com/am"
       realm = "alpha"
       cookieName = "iPlanetDirectoryPro"
    }
    oauth {
       oauthClientId = "sdkPublicClient"
       oauthRedirectUri = "https://localhost:8443/callback"
       oauthScope = "openid profile email address"
    }
    service {
       authServiceName = "Login"
       registrationServiceName = "Registration"
    }
    store {
       // Default storage settings
       // Uses SecureSharedPreferences
       //    oidcStorage = TokenStorage(ContextProvider.context)
       //    ssoTokenStorage = SSOTokenStorage(ContextProvider.context)
       //    cookiesStorage = CookiesStorage(ContextProvider.context)

       oidcStorage = MyCustomTokenStorage(ContextProvider.context)
       ssoTokenStorage = MyCustomSSOTokenStorage(ContextProvider.context)
       cookiesStorage = MyCustomCookiesStorage(ContextProvider.context)
    }
}

FRAuth.start(this, options);
```

> **Note**
>
> You can only specify the `store` options when dynamically configuring the Ping SDK for Android. You cannot add the parameters to the `strings.xml` file.

Implement storage fallbacks

One use case for providing custom storage is when the device you are targeting might not support the default `SecureSharedPreferences` storage methods provided by the SDK.

In this case you can create a fallback mechanism such that if the default storage method produces an error, a second storage method attempts to save the data.

The following `CustomStorageWithFallback.kt` example file is available in the **forgerock-android-sdk** GitHub repo.

```
package com.example.app.storage

import android.content.Context
import kotlinx.serialization.Serializable
import org.forgerock.android.auth.AccessToken
import org.forgerock.android.auth.SSOToken
import org.forgerock.android.auth.storage.CookiesStorage
import org.forgerock.android.auth.storage.SSOTokenStorage
import org.forgerock.android.auth.storage.Storage
import org.forgerock.android.auth.storage.TokenStorage

/
 * A custom storage implementation that switches to a fallback storage when an error occurs.
 */
class CustomStorageWithFallback<T : @Serializable Any>(
    private val context: Context,
    private val flag: String, (1)
    primary: Storage<T>, (2)
    private val fallback: Storage<T> (3)
) : Storage<T> {

    @Volatile
    private var current: Storage<T> = primary (4)

    /
     * Save an item to the current storage. If an error occurs, switch to the fallback storage.
     *
     * @param item The item to be saved.
     */
    override fun save(item: T) {
        try {
            // Save the item to the current storage.
            current.save(item) (5)
        } catch (e: Throwable) {
            // If an error occurs, switch to the fallback storage.
            context.getSharedPreferences("storage-control", Context.MODE_PRIVATE).edit()
                .putInt(flag, 1).apply() (6)
            fallback.save(item) (7)
            current = fallback
        }
    }

    /
     * Retrieve an item from the current storage.
     *
     * @return The retrieved item, or null if no item is found.
     */
    override fun get(): T? {
        return current.get()
    }

    /
     * Delete an item from the current storage.
     */
    override fun delete() {
        current.delete()
    }
}

/
```

```
 * Load the SSO token storage with a fallback mechanism.
 *
 * @param context The application context.
 * @return The storage instance for SSO tokens.
 */
fun loadSSOTokenStorage(context: Context): Storage<SSOToken> {  (8)
    return loadStorage(
        context,
        "ssoStorage",
        { SSOTokenStorage(context) },
        { MemoryStorage() }
    )
}

/
 * Load the token storage with a fallback mechanism.
 *
 * @param context The application context.
 * @return The storage instance for tokens.
 */
fun loadTokenStorage(context: Context): Storage<AccessToken> { (9)
    return loadStorage(
        context,
        "tokenStorage",
        { TokenStorage(context) },
        { MemoryStorage() }
    )
}

/
 * Load the cookies storage with a fallback mechanism.
 *
 * @param context The application context.
 * @return The storage instance for cookies.
 */
fun loadCookiesStorage(context: Context): Storage<Collection<String>> { (10)
    return loadStorage(
        context,
        "cookiesStorage",
        { CookiesStorage(context) },
        { MemoryStorage() }
    )
}

/
 * Load a storage instance with a fallback mechanism.
 *
 * @param T The type of object to be stored.
 * @param context The application context.
 * @param flag A flag used to control the storage type.
 * @param primary A function to initialize the primary storage.
 * @param fallback A function to initialize the fallback storage.
 * @return The storage instance.
 */
inline fun <reified T : Any> loadStorage( (11)
    context: Context,
    flag: String,
    primary: () → Storage<T>,
    fallback: () → Storage<T>
): Storage<T> {
    val control = context.getSharedPreferences("storage-control", Context.MODE_PRIVATE)
    // Get the storage type from the control flag. 0: primary, 1: fallback.
```

```
        val storageType = control.getInt(flag, 0)
        return when (storageType) {
            // Use the primary storage.
            0 → CustomStorageWithFallback(context,
                flag,
                primary(),
                fallback())

            // Use the fallback storage.
            else → fallback()
        }
    }
```

1. Flag whether the code should use the primary storage mechanism, or the fallback
2. The class to use as the ***primary*** storage mechanism
3. The class to use as the ***fallback*** storage mechanism
4. Initially, set the primary mechanism as current
5. Attempt to save with the ***current*** mechanism
6. If it fails, set `flag` to `1`
7. Attempt to save with the ***fallback*** mechanism
8. Create an *SSO token* wrapper function to load the primary and fallback mechanisms
9. Create an *OIDC token* wrapper function to load the primary and fallback mechanisms
10. Create a *Cookie* wrapper function to load the primary and fallback mechanisms
11. Create a function to load the customized storage wrappers

Configure your SDK application as follows to use the customized storage with fallback functionality:

```
store {
    oidcStorage = loadTokenStorage(ContextProvider.context)
    ssoTokenStorage = loadSSOTokenStorage(ContextProvider.context)
    cookiesStorage = loadCookiesStorage(ContextProvider.context)
}
```

Preventing the Keystore System from using StrongBox

Devices running Android 9 or higher might be able to use a keystore system backed by StrongBox.

Storing keys, tokens, and secrets by using StrongBox offers the highest level of security for your app, and is the default option in the Ping SDK for Android.

However, using StrongBox can be slower, and more resource-intensive. When using StrongBox on certain devices the performance and responsiveness of your app may drop below acceptable levels. To learn more, refer to the device requirements for StrongBox in the Android Source documentation.

The Ping SDK for Android provides a `strongBoxPreferred` flag you can use to avoid the use of StrongBox if required. The flag only applies to the storage mechanisms built-in to the Ping SDK for Android. You do not have to provide your own custom storage to use the `strongBoxPreferred` flag.

Important

If your app is using customized storage and you switch to using the built-in storage mechanisms the app will not be able to access the existing tokens and keys.
To avoid this, first call `FRAuth.start` with the original configuration and the customized storage, then call it a second time with the new `store` configuration and `strongBoxPreferred` flag.

Use the following code to use the built-in storage mechanisms prevent use of StrongBox:

Preventing use of StrongBox for storage

```
val myConfig = FROptionsBuilder.build {
  server {
    ...
  }
  oauth {
    ...
  }
  store {
    oidcStorage = TokenStorage(
      encryptor = EncryptorDelegate(
        SecretKeyEncryptor {
          context = <application context> (1)
          keyAlias = "<key alias>" (2)
          strongBoxPreferred = false (3)
        }
      )
    )
    ssoTokenStorage = SSOTokenStorage(
      encryptor = EncryptorDelegate(
        SecretKeyEncryptor {
          context = <application context> (1)
          keyAlias = "<key alias>" (2)
          strongBoxPreferred = false (3)
        }
      )
    )
    cookiesStorage = CookiesStorage(
      encryptor = EncryptorDelegate(
        SecretKeyEncryptor {
          context = <application context> (1)
          keyAlias = "<key alias>" (2)
          strongBoxPreferred = false (3)
        }
      )
    )
  }
}
```

1 For *<application context>* enter the application context, such as `ContextProvider.context`.

2 For *<key alias>* enter a string used as the alias for the key the Ping SDK creates.
You can use any value that does not clash with any other key names. A common pattern is `<top-level-domain>.<company-name>.<version>.KEYS`.
For example, `com.example.v1.KEYS`.

3 To prevent use of StrongBox, set the `strongBoxPreferred` boolean to `false`.

If not specified or set to `true`, the Ping SDK for Android will use StrongBox when configured to use the built-in storage mechanisms.

Some devices implement StrongBox, but are not optimal. You can use the `Build` class to conditionally apply the `strongBoxPreferred` flag based on the device manufacturer, model, or other properties:

Conditionally applying flags based on device properties

```
store {
    if (Build.MANUFACTURER.contains("Example")) {
      oidcStorage = TokenStorage(
        encryptor = EncryptorDelegate(SecretKeyEncryptor {
          context = ContextProvider.context
          keyAlias = "com.example.v1.KEYS"
          strongBoxPreferred = false
        })
      )
    }
  }
```

Customize storage on JavaScript

The Ping SDK for JavaScript provides two built-in storage schemes for OAuth 2.0 tokens:

Session storage

Store tokens using the `sessionStorage` API.

The browser clears session storage when a page session ends.

Local storage

Store tokens using the `localStorage` API.

The browser saves local storage data across browser sessions. This is the default setting, as it provides the highest browser compatibility.

You can configure your JavaScript apps to use customized storage if required.

Implement storage functions

You must implement the following functions in your custom storage scheme:

`set(clientId, tokens)`

Store a tokens object in the customized storage for a particular client.

`get(clientId)`

Retrieves the tokens object from the customized storage for a particular client.

`remove(clientId)`

Remove all items from the customized storage for a particular client.

Example:

```
let inMemoryTokens;

myTokenStore = {
  get(clientId) {
    console.log('Custom token getter used.');
    // Return a promise that resolves to any tokens stored in memory
    return Promise.resolve(inMemoryTokens);
  },
  set(clientId, tokens) {
    console.log('Custom token setter used.');
    // Example of storing tokens in memory
    inMemoryTokens = tokens;
    return Promise.resolve(undefined);
  },
  remove(clientId) {
    console.log('Custom token remover used.');
    // Reset the in-memory store
    inMemoryTokens = undefined;
    return Promise.resolve(undefined);
  },
};
```

Enable the custom storage

Use the `tokenStore` configuration property to configure the Ping SDK for JavaScript to use your custom storage object:

```
forgerock.Config.set({
    serverConfig: {
        baseUrl: 'https://openam-forgerock-sdks.forgeblocks.com/am',
        timeout: 3000,
    },
    clientId: 'sdkPublicClient',
    scope: 'openid profile email address',
    redirectUri: ${window.location.origin}/callback.html,
    tokenStore: myTokenStore
});
```

Enable SSL pinning

Applies to:

- ✓ Ping SDK for Android
- ✓ Ping SDK for iOS
- ✕ Ping SDK for JavaScript

The Ping SDKs support *SSL pinning*, sometimes referred to as *certificate pinning*. SSL pinning is the security practice of validating the certificates presented by the server against known values.

When the SDK attempts to make an HTTPS connection to your authorization server, it first verifies that a hash of the server's public key (obtained from the server's SSL certificate) matches a set of hashes defined within your app. This SSL pinning reduces the chance of a man-in-the-middle (MITM) attack, improving the security of your app.

If the hash does not match, your app does not connect to the authorization server, and an error is returned instead. Note that if your public key changes, you will need to rebuild and re-release your app with the new hash included.

Get a hash of the public key from your server

To enable SSL pinning you need a hash of your server's public key. You can use the `openssl` tool to extract this from your server's SSL certificate and create the hash value.

In the following command, replace `<tenant-env-fqdn>` with the fully-qualified domain name of your server, for example, `my-company.forgeblocks.com`:

```
echo | openssl s_client -servername <tenant-env-fqdn> -connect <tenant-env-fqdn>:443 | openssl x509 -pubkey -noout |
openssl rsa -pubin -outform der | openssl dgst -sha256 -binary | openssl enc -base64
```

The command outputs a hash of the public key extracted from the certificate:

`S4kZuhQQ1DPcXBSWFQXD0gG+UW7usdbVx6roNWpRl65I=`

Use this value in the next steps to configure SSL pinning.

Ping SDK for Android

Configure SSL pinning in your Android application.

Ping SDK for iOS

Configure SSL pinning in your iOS application.

Configure SSL pinning in Android

To enable SSL pinning in the Ping SDK for Android, add the hash of the public keys for any PingAM authorization servers your application will contact to your app's configuration.

Add the hashes to an array named `forgerock_ssl_pinning_public_key_hashes` in your `strings.xml` file:

```
<string-array name="forgerock_ssl_pinning_public_key_hashes">
   <item>S4kZuhQQ1DPcXBSWFQXD0gG+UW7usdbVx6roNWpRl65I=</item>
</string-array>
```

If the public key you use to obtain SSL certificates for the PingAM servers change, you can update this property programmatically.

Override default implementation of SSL pinning for Android

You can override how the Ping SDK for Android performs SSL pinning by registering your own implementation.

To override the default SSL pinning, you create your own implementation of `checkServerTrusted()`:

```
try {
    final TrustManager myCustomTrustManager = new X509TrustManager() {
        @Override
        public void checkClientTrusted(java.security.cert.X509Certificate[] chain, String authType) {}

        @Override
        public void checkServerTrusted(java.security.cert.X509Certificate[] chain, String authType) {
            // Provide custom SSL Pinning handling
        }

        @Override
        public java.security.cert.X509Certificate[] getAcceptedIssuers() {
            return new java.security.cert.X509Certificate[] {};
        }
    };
    SSLContext sslContext = SSLContext.getInstance("SSL");
    sslContext.init(null, new TrustManager[] { myCustomTrustManager }, new java.security.SecureRandom());
    Config.getInstance().reset();
    Config.getInstance().init(this, null);
    Config.getInstance().setBuildSteps(Collections.singletonList(builder1 -> {
        builder1.sslSocketFactory(sslContext.getSocketFactory(), (X509TrustManager) myCustomTrustManager);
        builder1.hostnameVerifier((s, sslSession) -> true);
    }));

} catch (NoSuchAlgorithmException | KeyManagementException e) {
    runOnUiThread(() -> content.setText(e.getMessage()));
}
```

Alternatively, you can override the SDK's SSL pinning functionality:

```
val myCustomTrustManager: TrustManager = object : X509TrustManager {
    override fun checkClientTrusted(chain: Array<X509Certificate>, authType: String) {}
    override fun checkServerTrusted(chain: Array<X509Certificate>, authType: String) {
        // Provide custom SSL Pinning handling
    }
    override fun getAcceptedIssuers(): Array<X509Certificate> {
        return arrayOf()
    }
}
val sslContext = SSLContext.getInstance("SSL")
sslContext.init(null, arrayOf(myCustomTrustManager), SecureRandom())

val option = FROptionsBuilder.build {
    server {
        forgerock_url = "https://custom.example.com"
        forgerock_realm = "prod"
    }
    sslPinning {
        buildSteps = listOf(object: BuildStep<OkHttpClient.Builder> {
            override fun build(builder1: OkHttpClient.Builder) {
                builder1.sslSocketFactory(
                    sslContext.socketFactory,
                    myCustomTrustManager as X509TrustManager
                )
                builder1.hostnameVerifier { s, sslSession -> true }
            }
        })
        forgerock_ssl_pinning_public_key_hashes = emptyList()
    }
}
```

Configure SSL pinning in iOS

To enable SSL pinning in the Ping SDK for iOS, add the hash of the public keys for any PingAM authorization servers your application will contact to your app's configuration.

Add the hashes to an array named `forgerock_ssl_pinning_public_key_hashes` in your `FRAuthConfig.plist` file:

```
<key>forgerock_ssl_pinning_public_key_hashes</key>
<array>
    <string>S4kZuhQQ1DPcXBSWFQXD0gG+UW7usdbVx6roNWpRl65I=</string>
</array>
```

If the public key you use to obtain SSL certificates for the PingAM servers change, you can update this property programmatically.

Override default implementation of SSL pinning for iOS

You can override how the Ping SDK for iOS performs SSL pinning by registering your own implementation.

To override the default SSL pinning, create a new `CustomPinningHandler` subclass of the default `FRURLSessionSSLPinningHandler` class. Override the implementation of the `urlSession` functions:

```
class CustomPinningHandler: FRURLSessionSSLPinningHandler {
    override func urlSession(_ session: URLSession, didReceive challenge: URLAuthenticationChallenge,
completionHandler: @escaping (URLSession.AuthChallengeDisposition, URLCredential?) -> Void) {
        // Provide Custom SSL Pinning handling
    }

    override func urlSession(_ session: URLSession, task: URLSessionTask, didReceive challenge:
URLAuthenticationChallenge, completionHandler: @escaping (URLSession.AuthChallengeDisposition, URLCredential?) ->
Void) {
        // Provide Custom SSL Pinning handling
    }
}
```

Add your new custom handler as part of the configuration:

```
let customPinningHandler = CustomPinningHandler(frSecurityConfiguration: nil)
RestClient.shared.setURLSessionConfiguration(config: nil, handler: customPinningHandler)
```

Ping SDK for Auth Journey tutorials

Follow these tutorials integrate your apps with **Authentication journeys**, also known as ***Intelligent Authentication*** in the following servers:

- PingOne Advanced Identity Cloud
- PingAM

Ping SDK tutorials

Follow these core Ping SDK tutorials to integrate your apps with **Authentication journeys**, also known as ***Intelligent Authentication*** in the following servers:

Ping SDK for Android

Ping SDK for iOS

Ping SDK for JavaScript

Integrating Ping SDKs into other platforms

Follow these tutorials to leverage the Ping SDKs in other platforms or languages, to support **Authentication journeys**, also known as ***Intelligent Authentication*** in your apps.

Angular

Flutter (iOS)

ReactJS

React Native (iOS)

Ping SDK for Auth Journey tutorials

Follow these core Ping SDK tutorials to integrate your apps with **Authentication journeys**, also known as ***Intelligent Authentication*** in the following servers:

- PingOne Advanced Identity Cloud
- PingAM

Ping SDK for Android

Ping SDK for iOS

Ping SDK for JavaScript

Ping SDK for Android Auth Journey tutorials

Follow these tutorials integrate your Android apps with **Authentication journeys**, also known as ***Intelligent Authentication*** in the following servers:

- PingOne Advanced Identity Cloud
- PingAM

Ping SDK for Android Tutorials

Quick start

In this quick start tutorial you update one of our sample applications.

The app steps through a simple authentication journey and displays a basic prototype UI to gather user credentials.

Deep dive

This deep dive tutorial guides you through creating a Ping SDK-enabled Android app from beginning to end.

You'll step through the user authentication journey and display the appropriate user interface, meaning you get to implement the design to your specific requirements.

Authentication journey quick start for Android

Prepare › Download › Configure › Run

In this quick start tutorial you update one of our sample applications to connect to your PingOne Advanced Identity Cloud tenant or PingAM server to authenticate a user.

The app steps through a simple authentication journey and returns a session token. The app is then able to obtain user info from the server, and finally sign out to terminate the session.

> **Tip**
>
> To learn how to create an app from scratch to authenticate your users, try the Authentication journey deep-dive tutorial for Android.

Before you begin

Before you begin this tutorial ensure you have set up your PingOne Advanced Identity Cloud tenant or PingAM server with the required configuration.

For example, you will need an OAuth 2.0 client application set up, as well as an authentication journey for the app to navigate.

Complete prerequisites »

Step 1. Download the samples

To start this tutorial, you need to download the SDK sample apps repo, which contains the projects you will use.

Start step 1 »

Step 2. Configure connection properties

In this step, you configure the "kotlin-ui-prototype" sample app to connect to the OAuth 2.0 application you created in PingOne Advanced Identity Cloud or PingAM.

Start step 2 »

Step 3. Test the app

In this step, you will test your application.

You run it in the emulator or on your Android device, perform authentication with a demo user, obtain OAuth 2.0 tokens, and then log out the user.

Test app »

Before you begin

Prepare › Download › Configure › Run

To successfully complete this tutorial refer to the prerequisites and compatibility requirements in this section.

The tutorial also requires a configured server.

Compatibility

Android

This sample requires at least Android 9 (Pie) - API level 28.

For more information, refer to Supported operating systems and browsers.

Java

This sample requires at least Java 8 (v1.8).

Prerequisites

Android Studio

Download and install Android Studio, which is available for many popular operating systems.

An Android emulator or physical device

To try the quick start application as you develop it, you need an Android device. To add a virtual, emulated Android device to Android Studio, refer to Create and manage virtual devices, on the **Android Developers** website.

Server configuration

This tutorial requires you to configure one of the following servers:

PingOne Advanced Identity Cloud

PingAM

PingOne Advanced Identity Cloud

The samples and tutorials in this documentation often require that you have an identity set up so that you can test authentication.

To create a demo user in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. In the left panel, click **Identities** > **Manage**.
3. Click **+ New Alpha realm - User**.
4. Enter the following details:
    - **Username** = `demo`
    - **First Name** = `Demo`
    - **Last Name** = `User`
    - **Email Address** = `demo.user@example.com`
    - **Password** = `Ch4ng3it!`
5. Click **Save**.

Authentication journeys provide fine-grained authentication by allowing multiple paths and decision points throughout the flow. Authentication journeys are made up of nodes that define actions taken during authentication.

Each node performs a single task, such as collecting a username or making a simple decision. Nodes can have multiple outcomes rather than just success or failure. For details, see the Authentication nodes configuration reference in the PingAM documentation.

To create a simple journey for use when testing the Ping SDKs, follow these steps:

1. In your PingOne Advanced Identity Cloud tenant, navigate to **Journeys**, and click **+ New Journey**.
2. Enter a name, such as `sdkUsernamePasswordJourney` and click **Save**.

   The authentication journey designer appears.

3. Drag the following nodes into the designer area:
    - **Page Node**
    - **Platform Username**
    - **Platform Password**
    - **Data Store Decision**
4. Drag and drop the **Platform Username** and **Platform Password** nodes onto the **Page Node**, so that they both appear on the same page when logging in.
5. Connect the nodes as follows:

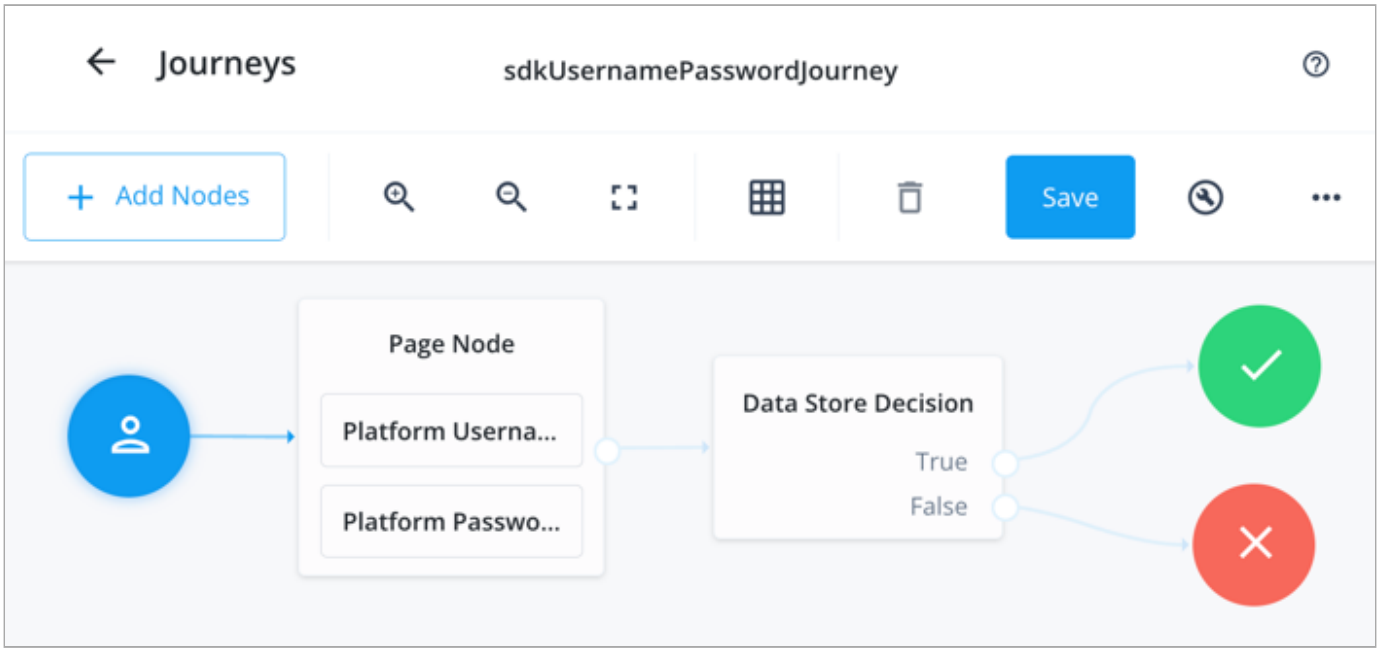


Figure 1. Example username and password authentication journey

6. Click **Save**.

Public clients do not use a client secret to obtain tokens because they are unable to keep them hidden. The Ping SDKs commonly use this type of client to obtain tokens, as they cannot guarantee safekeeping of the client credentials in a browser or on a mobile device.

To register a *public* OAuth 2.0 client application for use with the SDKs in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. In the left panel, click **Applications**.
3. Click **+ Custom Application**.
4. Select **OIDC - OpenId Connect** as the sign-in method, and then click **Next**.
5. Select **Native / SPA** as the application type, and then click **Next**.
6. In **Name**, enter a name for the application, such as `Public SDK Client`.

7. In **Owners**, select a user that is responsible for maintaining the application, and then click **Next**.

   > **Tip**
   >
   > When trying out the SDKs, you could select the `demo` user you created previously.

8. In **Client ID**, enter `sdkPublicClient`, and then click **Create Application**.

   PingOne Advanced Identity Cloud creates the application and displays the details screen.

9. On the **Sign On** tab:

   1. In **Sign-In URLs**, enter the following values:

      `org.forgerock.demo://oauth2redirect`

      > **Important**
      >
      > Also add any other domains where you host SDK applications.

   2. In **Grant Types**, enter the following values:

      `Authorization Code`

      `Refresh Token`

   3. In **Scopes**, enter the following values:

      `openid profile email address`

10. Click Show advanced settings, and on the **Authentication** tab:

    1. In **Token Endpoint Authentication Method**, select `none`.
    2. In **Client Type**, select `Public`.
    3. Enable the **Implied Consent** property.

11. Click **Save**.

The application is now configured to accept client connections from and issue OAuth 2.0 tokens to the example applications and tutorials covered by this documentation.

The provider specifies the supported OAuth 2.0 configuration options for a realm.

To ensure the PingOne Advanced Identity Cloud OAuth 2.0 provider service is configured for use with the Ping SDKs, follow these steps:

1. In your PingOne Advanced Identity Cloud tenant, navigate to **Native Consoles** > **Access Management**.
2. In the left panel, click **Services**.
3. In the list of services, click **OAuth2 Provider**.
4. On the **Core** tab, ensure **Issue Refresh Tokens** is enabled.
5. On the **Consent** tab, ensure **Allow Clients to Skip Consent** is enabled.

6. Click **Save Changes**.

PingAM

The samples and tutorials in this documentation often require that you have an identity set up so that you can test authentication.

To create a demo user in PingAM, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. Navigate to **Identities**, and then click **+ Add Identity**.
3. Enter the following details:
   - **User ID** = `demo`
   - **Password** = `Ch4ng3it!`
   - **Email Address** = `demo.user@example.com`
4. Click **Create**.

Authentication trees provide fine-grained authentication by allowing multiple paths and decision points throughout the authentication flow. Authentication trees are made up of nodes that define actions taken during authentication.

Each node performs a single task, such as collecting a username or making a simple decision. Nodes can have multiple outcomes rather than just success or failure. For details, see the Authentication nodes configuration reference in the PingAM documentation.

To create a simple tree for use when testing the Ping SDKs, follow these steps:

1. Under **Realm Overview**, click **Authentication Trees**, then click **Create Tree**.
2. Enter a tree name, for example `sdkUsernamePasswordJourney`, and then click **Create**.

   The authentication tree designer appears, showing the **Start** entry point connected to the **Failure** exit point.
3. Drag the following nodes from the **Components** panel on the left side into the designer area:
   - **Page Node**
   - **Username Collector**
   - **Password Collector**
   - **Data Store Decision**
4. Drag and drop the **Username Collector** and **Password Collector** nodes onto the **Page Node**, so that they both appear on the same page when logging in.
5. Connect the nodes as follows:

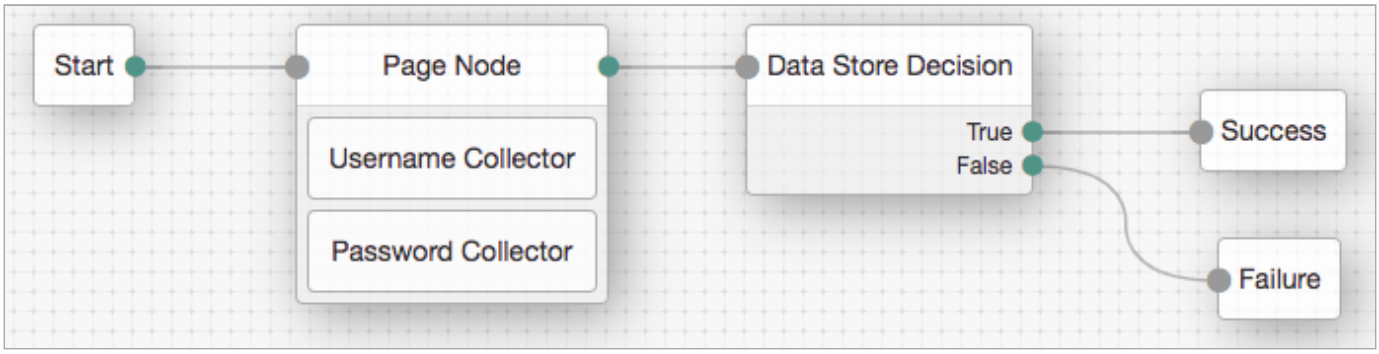


Figure 2. Example username and password authentication tree

6. Select the **Page Node**, and in the **Properties** pane, set the **Stage** property to `UsernamePassword`.

   > **Tip**
   >
   > You can configure the node properties by selecting a node and altering properties in the right-hand panel.

   One of the samples uses this specific value to determine the custom UI to display.

7. Click **Save**.

Public clients do not use a client secret to obtain tokens because they are unable to keep them hidden. The Ping SDKs commonly use this type of client to obtain tokens, as they cannot guarantee safekeeping of the client credentials in a browser or on a mobile device.

To register a *public* OAuth 2.0 client application for use with the SDKs in AM, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. Navigate to **Applications** > **OAuth 2.0** > **Clients**, and then click **+ Add Client**.
3. In **Client ID**, enter `sdkPublicClient`.
4. Leave **Client secret** empty.
5. In **Redirection URIs**, enter the following values:

   `org.forgerock.demo://oauth2redirect`

   > **Important**
   >
   > Also add any other domains where you will be hosting SDK applications.

6. In **Scopes**, enter the following values:

   `openid profile email address`

7. Click **Create**.

   PingAM creates the new OAuth 2.0 client, and displays the properties for further configuration.

8. On the **Core** tab:

   1. In **Client type**, select `Public`.

2. Disable **Allow wildcard ports in redirect URIs**.
3. Click **Save Changes**.

9. On the **Advanced** tab:
    1. In **Grant Types**, enter the following values:

    ```
    Authorization Code
    Refresh Token
    ```

    2. In **Token Endpoint Authentication Method**, select `None`.
    3. Enable the **Implied consent** property.
10. Click **Save Changes**.

The provider specifies the supported OAuth 2.0 configuration options for a realm.

To ensure the PingAM OAuth 2.0 provider service is configured for use with the Ping SDKs, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. In the left panel, click **Services**.
3. In the list of services, click **OAuth2 Provider**.
4. On the **Core** tab, ensure **Issue Refresh Tokens** is enabled.
5. On the **Consent** tab, ensure **Allow Clients to Skip Consent** is enabled.
6. Click **Save Changes**.

Step 1. Download the samples

Prepare › **Download** › Configure › Run

To start this tutorial, you need to download the SDK sample apps repo, which contains the projects you will use.

1. In a web browser, navigate to the SDK Sample Apps repository.
2. Download the source code using one of the following methods:

Download a ZIP file

1. Click **Code**, and then click **Download ZIP**.
2. Extract the contents of the downloaded ZIP file to a suitable location.

Use a Git-compatible tool to clone the repo locally

1. Click **Code**, and then copy the HTTPS URL.

2. Use the URL to clone the repository to a suitable location.

   For example, from the command-line you could run:

```
git clone https://github.com/ForgeRock/sdk-sample-apps.git
```

The result of these steps is a local folder named `sdk-sample-apps`.

Step 2. Configure connection properties

Prepare › Download › **Configure** › Run

In this step, you configure the "kotlin-ui-prototype" sample to connect to your server.

1. In Android Studio, open the `sdk-sample-apps/android/kotlin-ui-prototype` folder you cloned in the previous step.
2. In the **Project** pane, switch to the **Android** view.

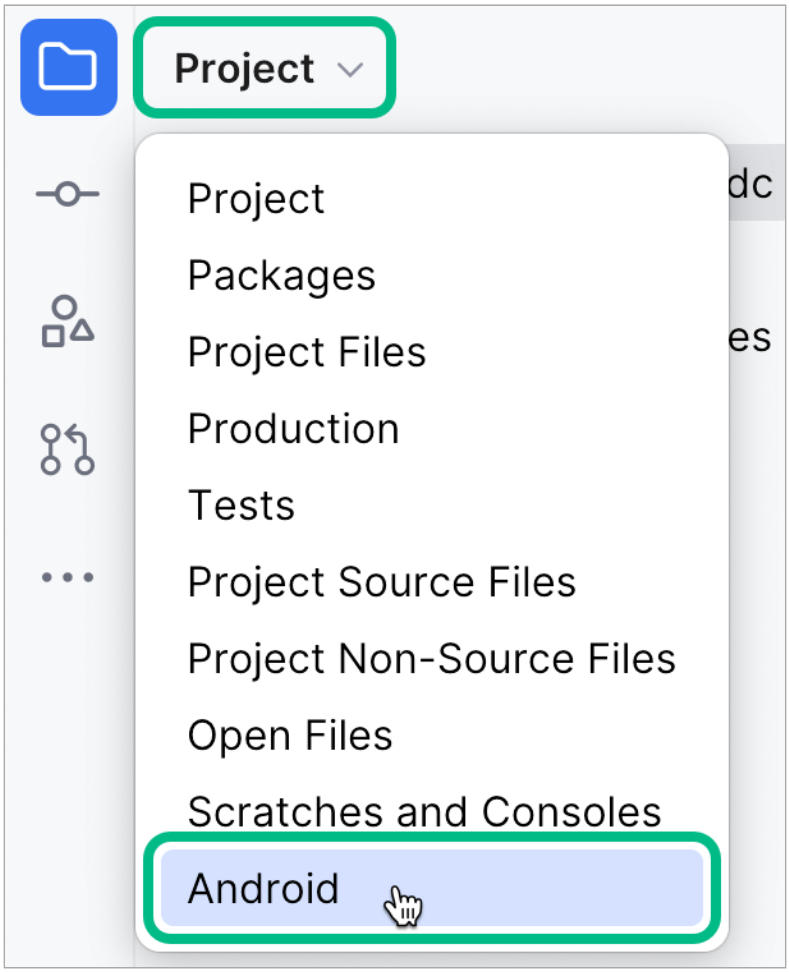


Figure 1. Switching the project pane to Android view.

3. In the **Android** view, navigate to **app > kotlin+java > com.example.app > env**, and open `EnvViewModel.kt`.

   This file has the server environments the sample app uses. Each specifies the properties using the `FROptionsBuilder.build` method.

```
// Example values for a PingAM instance
val PingAM = FROptionsBuilder.build {
    server {
        url = "https://openam.example.com:8443/openam"
        realm = "root"
        cookieName = "iPlanetDirectoryPro"
        timeout = 50
    }
    oauth {
        oauthClientId = "sdkPublicClient"
        oauthRedirectUri = "org.forgerock.demo://oauth2redirect"
        oauthScope = "openid profile email address"
        oauthSignOutRedirectUri = "org.forgerock.demo://oauth2redirect"
    }
    service {
        authServiceName = "sdkUsernamePasswordJourney"
    }
}

// Example values for a Ping Advanced Identity Cloud instance
val PingAdvancedIdentityCloud = FROptionsBuilder.build {
    server {
        url = "https://openam-forgerock-sdks.forgeblocks.com/am"
        realm = "alpha"
        cookieName = "29cd7a346b42b42"
        timeout = 50
    }
    oauth {
        oauthClientId = "sdkPublicClient"
        oauthRedirectUri = "org.forgerock.demo://oauth2redirect"
        oauthScope = "openid profile email address"
        oauthSignOutRedirectUri = "org.forgerock.demo://oauth2redirect"
    }
    service {
        authServiceName = "sdkUsernamePasswordJourney"
    }
}
```

4. Update the `PingAM` or `PingAdvancedIdentityCloud` example configuration values to match your server environment:

   url

   The URL of the server to connect to, including the deployment path of the Access Management component.

   Identity Cloud example:

   `https://openam-forgerock-sdks.forgeblocks.com/am`

   Self-hosted example:

   `https://openam.example.com:8443/openam`

realm

The realm in which the OAuth 2.0 client profile and authentication journeys are configured.

Usually, `root` for AM and `alpha` or `bravo` for Advanced Identity Cloud.

cookieName

The name of the cookie that contains the session token.

For example, with a self-hosted PingAM server this value might be `iPlanetDirectoryPro`.

> **Tip**
>
> PingOne Advanced Identity Cloud tenants use a random alpha-numeric string.
> To locate the cookie name in an PingOne Advanced Identity Cloud tenant, navigate to **Tenant settings** > **Global Settings**, and copy the value of the **Cookie** property.

oauthClientId

The client ID of your OAuth 2.0 application in PingOne Advanced Identity Cloud or PingAM.

For example, `sdkPublicClient`

oauthRedirectUri

The redirect URI or sign-in URL as configured in the OAuth 2.0 client profile.

> **Note**
>
> This value must exactly match a value configured in your OAuth 2.0 client.

For example, `org.forgerock.demo://oauth2redirect`.

oauthScope

The scopes you added to your OAuth 2.0 application in PingOne Advanced Identity Cloud.

For example, `openid profile email address`

oauthRedirectUri

The sign-out URL as configured in the OAuth 2.0 client profile.

> **Note**
>
> This value must exactly match a value configured in your OAuth 2.0 client.

For example, `org.forgerock.demo://oauth2redirect`.

authServiceName

The authentication tree or journey you created earlier.

For example, `sdkUsernamePasswordJourney`

The result will resemble the following:

```
// Example values for a Ping Advanced Identity Cloud instance
val PingAdvancedIdentityCloud = FROptionsBuilder.build {
    server {
        url = "https://openam-forgerock-sdks.forgeblocks.com/am"
        realm = "alpha"
        cookieName = "ch15fefc5407912"
        timeout = 50
    }
    oauth {
        oauthClientId = "sdkPublicClient"
        oauthRedirectUri = "org.forgerock.demo://oauth2redirect"
        oauthScope = "openid profile email address"
        oauthSignOutRedirectUri = "org.forgerock.demo://oauth2redirect"
    }
    service {
        authServiceName = "sdkUsernamePasswordJourney"
    }
}
```

5. Save your changes.

With the sample configured, you can proceed to Step 3. Test the app.

Step 3. Test the app

Prepare › Download › Configure › **Run**

In this step, you run and test the sample app.

You run it in the emulator or on your Android device and perform authentication with a demo user.

1. In Android Studio, select **Run > Run 'app'**.
2. In the sample app, on the **Environment** page, select the environment you configured in the last step.
3. Tap the menu icon (☰), tap 🚀 **Launch Journey**, and then tap **Submit**.
4. Sign on as a demo user:
    - **Name**: `demo`
    - **Password**: `Ch4ng3it!`
5. After successful authentication the app displays the session token:

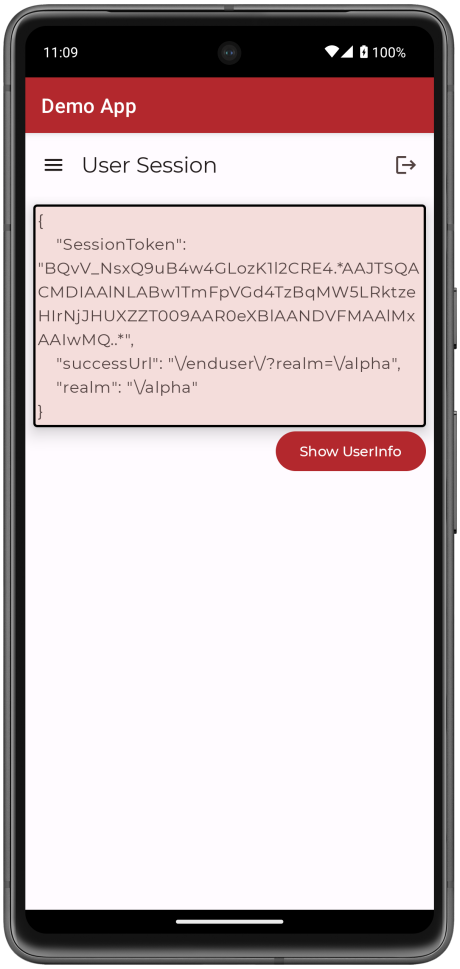


Figure 1. Viewing a user's session token in the Android sample app.

6. Tap **Show Userinfo**.

   The app displays the user info for the account.

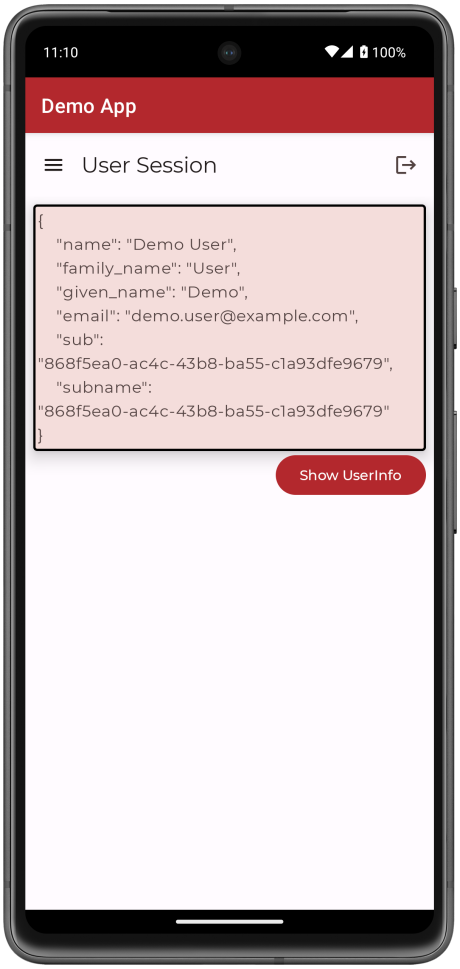


Figure 2. Viewing userinfo for an account in the Android sample app.

7. Tap the menu icon (☰), and then tap **Show Token**.

   The app displays the access, refresh, and ID tokens for the account. You can also view the scopes granted to the user.

Figure 3. Viewing OAuth 2.0 tokens for an account in the Android sample app.

8. Tap the menu icon (☰), and then tap **Logout**.

   The app terminates the user session and returns to the **Launch Journey** page.

Authentication journey deep-dive tutorial for Android

This tutorial guides you through creating a Ping SDK-enabled Android app from beginning to end. The app connects to a PingOne Advanced Identity Cloud tenant or PingAM server to authenticate a user using an authentication journey.

You'll step through the user authentication journey and display the appropriate user interface, meaning you get to implement the design to your requirements.

> **Tip**
>
> To get up and running in the shortest time, try the Authentication journey quick start for Android.

Before you begin

Before you begin this tutorial ensure you have set up your PingOne Advanced Identity Cloud tenant or PingAM server with the required configuration.

For example, you will need an OAuth 2.0 client application set up, as well as an authentication journey for the app to navigate.

Complete prerequisites »

Step 1. Configure the development environment

In this step, you set up your environment to create Android applications using the freely-available Android Studio IDE.

You then create a new application project and configure it to use the Ping SDK for Android.

Start step 1 »

Step 2. Configure connection properties

In this step, you provide your application with the settings it needs to connect to your PingOne Advanced Identity Cloud or PingAM instance.

For example, which authentication tree to use, and the realm it is a part of.

Start step 2 »

Step 3. Initialize the SDK

In this step, you enable debug logging during development.

You then and add a call to the `FRAuth.start()` method, which initializes the SDK and loads the configuration you have defined in the previous step.

Start step 3 »

Step 4. Create a status view

In this step, you create a layout and add buttons to log in and log out your user, as well as a text view field to show their current authentication status.

You also add the code to update the value displayed in the text view.

Start step 4 »

Step 5. Add login and logout calls

In this step, you update the app with the `NodeListener` interface, which manages the client side of the authentication journey.

Start step 5 »

Step 6. Create UI to handle the callbacks

In this step, you add a UI fragment to obtain credentials from the user, and code to open that fragment when the callback is received.

You also add code to populate the callback with the credentials and return it to the server, completing the authentication journey.

Start step 6 »

Step 7. Test the app

In this step, you will test your application.

You run it in the emulator or on your Android device, perform authentication with a demo user, check the log for success messages, and then log out the user.

Test app »

Before you begin

To successfully complete this tutorial refer to the prerequisites and compatibility requirements in this section.

The tutorial also requires a configured server.

Compatibility

Android

This sample requires at least Android API 23 (Android 6.0)

Java

This sample requires at least Java 8 (v1.8).

Prerequisites

Android Studio

Download and install Android Studio, which is available for many popular operating systems.

An Android emulator or physical device

To try the quick start application as you develop it, you need an Android device. To add a virtual, emulated Android device to Android Studio, refer to Create and manage virtual devices, on the **Android Developers** website.

Server configuration

This tutorial requires you to configure one of the following servers:

PingOne Advanced Identity Cloud

PingOne Advanced Identity Cloud

PingAM

PingAM

PingOne Advanced Identity Cloud

The samples and tutorials in this documentation often require that you have an identity set up so that you can test authentication.

To create a demo user in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. In the left panel, click **Identities** > **Manage**.
3. Click **+ New Alpha realm - User**.

4. Enter the following details:
    - **Username** = `demo`
    - **First Name** = `Demo`
    - **Last Name** = `User`
    - **Email Address** = `demo.user@example.com`
    - **Password** = `Ch4ng3it!`
5. Click **Save**.

Authentication journeys provide fine-grained authentication by allowing multiple paths and decision points throughout the flow. Authentication journeys are made up of nodes that define actions taken during authentication.

Each node performs a single task, such as collecting a username or making a simple decision. Nodes can have multiple outcomes rather than just success or failure. For details, see the Authentication nodes configuration reference in the PingAM documentation.

To create a simple journey for use when testing the Ping SDKs, follow these steps:

1. In your PingOne Advanced Identity Cloud tenant, navigate to **Journeys**, and click **+ New Journey**.
2. Enter a name, such as `sdkUsernamePasswordJourney` and click **Save**.

   The authentication journey designer appears.
3. Drag the following nodes into the designer area:
    - **Page Node**
    - **Platform Username**
    - **Platform Password**
    - **Data Store Decision**
4. Drag and drop the **Platform Username** and **Platform Password** nodes onto the **Page Node**, so that they both appear on the same page when logging in.
5. Connect the nodes as follows:

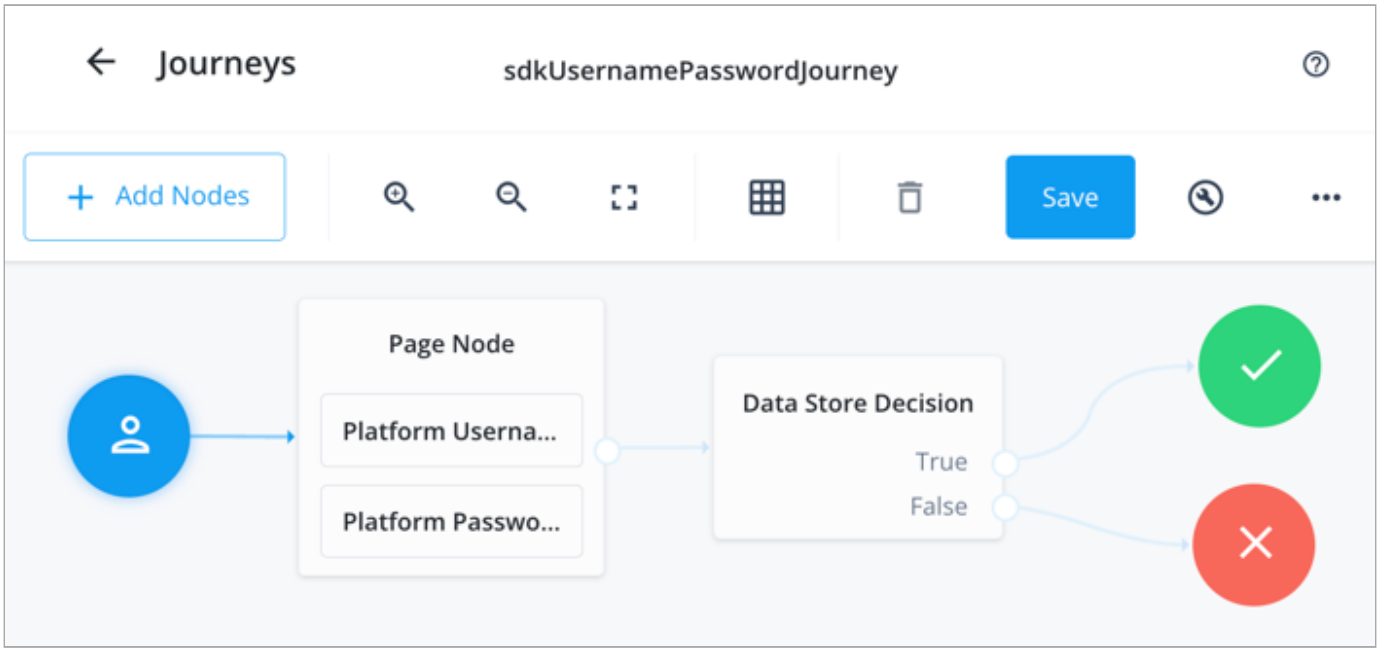


Figure 1. Example username and password authentication journey

6. Click **Save**.

Public clients do not use a client secret to obtain tokens because they are unable to keep them hidden. The Ping SDKs commonly use this type of client to obtain tokens, as they cannot guarantee safekeeping of the client credentials in a browser or on a mobile device.

To register a *public* OAuth 2.0 client application for use with the SDKs in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. In the left panel, click **Applications**.
3. Click **+ Custom Application**.
4. Select **OIDC - OpenId Connect** as the sign-in method, and then click **Next**.
5. Select **Native / SPA** as the application type, and then click **Next**.
6. In **Name**, enter a name for the application, such as `Public SDK Client`.
7. In **Owners**, select a user that is responsible for maintaining the application, and then click **Next**.

> **Tip**
>
> When trying out the SDKs, you could select the `demo` user you created previously.

8. In **Client ID**, enter `sdkPublicClient`, and then click **Create Application**.

   PingOne Advanced Identity Cloud creates the application and displays the details screen.

9. On the **Sign On** tab:

   1. In **Sign-In URLs**, enter the following values:

      `org.forgerock.demo://oauth2redirect`

> **Important**
>
> Also add any other domains where you host SDK applications.

2. In **Grant Types**, enter the following values:

   `Authorization Code`

   `Refresh Token`

3. In **Scopes**, enter the following values:

   `openid profile email address`

10. Click Show advanced settings, and on the **Authentication** tab:

    1. In **Token Endpoint Authentication Method**, select `none`.
    2. In **Client Type**, select `Public`.
    3. Enable the **Implied Consent** property.

11. Click **Save**.

The application is now configured to accept client connections from and issue OAuth 2.0 tokens to the example applications and tutorials covered by this documentation.

The provider specifies the supported OAuth 2.0 configuration options for a realm.

To ensure the PingOne Advanced Identity Cloud OAuth 2.0 provider service is configured for use with the Ping SDKs, follow these steps:

1. In your PingOne Advanced Identity Cloud tenant, navigate to **Native Consoles** > **Access Management**.
2. In the left panel, click **Services**.
3. In the list of services, click **OAuth2 Provider**.
4. On the **Core** tab, ensure **Issue Refresh Tokens** is enabled.
5. On the **Consent** tab, ensure **Allow Clients to Skip Consent** is enabled.
6. Click **Save Changes**.

PingAM

The samples and tutorials in this documentation often require that you have an identity set up so that you can test authentication.

To create a demo user in PingAM, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. Navigate to **Identities**, and then click **+ Add Identity**.
3. Enter the following details:
   - **User ID** = `demo`

- **Password** = `Ch4ng3it!`
- **Email Address** = `demo.user@example.com`

4. Click **Create**.

Authentication trees provide fine-grained authentication by allowing multiple paths and decision points throughout the authentication flow. Authentication trees are made up of nodes that define actions taken during authentication.

Each node performs a single task, such as collecting a username or making a simple decision. Nodes can have multiple outcomes rather than just success or failure. For details, see the Authentication nodes configuration reference in the PingAM documentation.

To create a simple tree for use when testing the Ping SDKs, follow these steps:

1. Under **Realm Overview**, click **Authentication Trees**, then click **Create Tree**.
2. Enter a tree name, for example `sdkUsernamePasswordJourney`, and then click **Create**.

   The authentication tree designer appears, showing the **Start** entry point connected to the **Failure** exit point.
3. Drag the following nodes from the **Components** panel on the left side into the designer area:
   - **Page Node**
   - **Username Collector**
   - **Password Collector**
   - **Data Store Decision**
4. Drag and drop the **Username Collector** and **Password Collector** nodes onto the **Page Node**, so that they both appear on the same page when logging in.
5. Connect the nodes as follows:

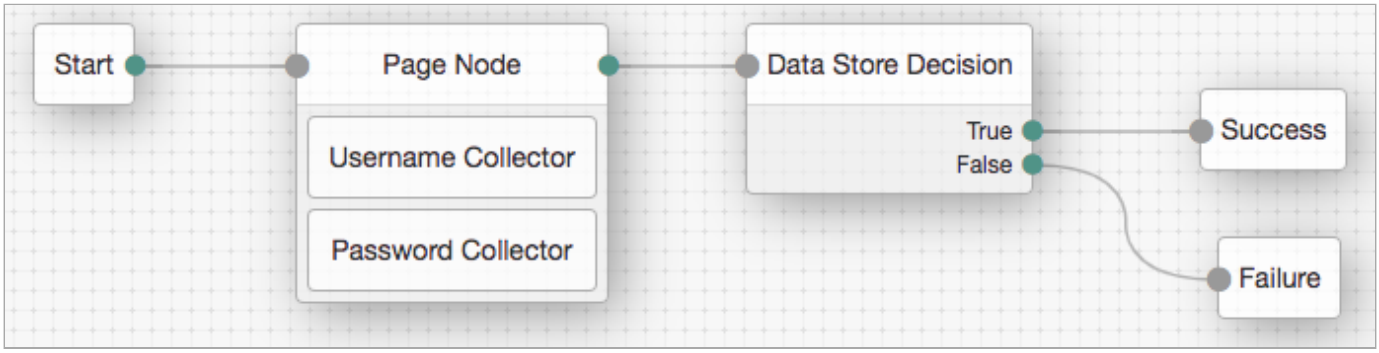


Figure 2. Example username and password authentication tree

6. Select the **Page Node**, and in the **Properties** pane, set the **Stage** property to `UsernamePassword`.

> **Tip**
>
> You can configure the node properties by selecting a node and altering properties in the right-hand panel.

One of the samples uses this specific value to determine the custom UI to display.

7. Click **Save**.

Public clients do not use a client secret to obtain tokens because they are unable to keep them hidden. The Ping SDKs commonly use this type of client to obtain tokens, as they cannot guarantee safekeeping of the client credentials in a browser or on a mobile device.

To register a *public* OAuth 2.0 client application for use with the SDKs in AM, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. Navigate to **Applications** > **OAuth 2.0** > **Clients**, and then click **+ Add Client**.
3. In **Client ID**, enter `sdkPublicClient`.
4. Leave **Client secret** empty.
5. In **Redirection URIs**, enter the following values:

   `org.forgerock.demo://oauth2redirect`

   > **Important**
   >
   > Also add any other domains where you will be hosting SDK applications.

6. In **Scopes**, enter the following values:

   `openid profile email address`

7. Click **Create**.

   PingAM creates the new OAuth 2.0 client, and displays the properties for further configuration.

8. On the **Core** tab:
   1. In **Client type**, select `Public`.
   2. Disable **Allow wildcard ports in redirect URIs**.
   3. Click **Save Changes**.
9. On the **Advanced** tab:
   1. In **Grant Types**, enter the following values:

      ```
      Authorization Code
      Refresh Token
      ```

   2. In **Token Endpoint Authentication Method**, select `None`.
   3. Enable the **Implied consent** property.
10. Click **Save Changes**.

The provider specifies the supported OAuth 2.0 configuration options for a realm.

To ensure the PingAM OAuth 2.0 provider service is configured for use with the Ping SDKs, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. In the left panel, click **Services**.
3. In the list of services, click **OAuth2 Provider**.
4. On the **Core** tab, ensure **Issue Refresh Tokens** is enabled.
5. On the **Consent** tab, ensure **Allow Clients to Skip Consent** is enabled.
6. Click **Save Changes**.

Step 1. Configure the development environment

In this step, you set up your environment to create Android applications using the freely-available Android Studio IDE.

You then create a new application project and configure it to use the Ping SDK for Android.

Prerequisites

Android Studio

Download and install Android Studio, which is available for many popular operating systems.

An Android emulator or physical device

To try the quick start application as you develop it, you need an Android device. To add a virtual, emulated Android device to Android Studio, refer to Create and manage virtual devices, on the **Android Developers** website.

Create a new project

1. In Android Studio, select **File > New > New Project**.
2. On the **New Project** screen, select **Empty Views Activity**, and then click **[Next]**.
3. On the next screen:
   - In the **Name** field, enter `Ping SDK for Android Quick Start`.
   - In the **Package name** field, enter `com.example.quickstart`.
   - In the **Save location** field, enter the location in which to create the project.
   - In the **Language** drop-down, select `Java`.
   - In the **Minimum SDK** drop-down, select `API 23: Android 6.0 (Marshmallow)`.

- Click **[Finish]**.

  Android Studio creates a simple application that you can now configure to use the Ping SDK for Android.

Configure compile options

The Ping SDK for Android requires *at least* Java 8 (v1.8).

Configure compile options in your project to use this version of Java, or later:

1. In the **Android** view of your project, right-click **app**, and then click **Open module settings**.
2. In the **Project Structure** dialog, navigate to **Modules > app > Properties**.
3. In the **Source Compatibility** and **Target compatibility** drop-downs, select the version of Java to use for the project:

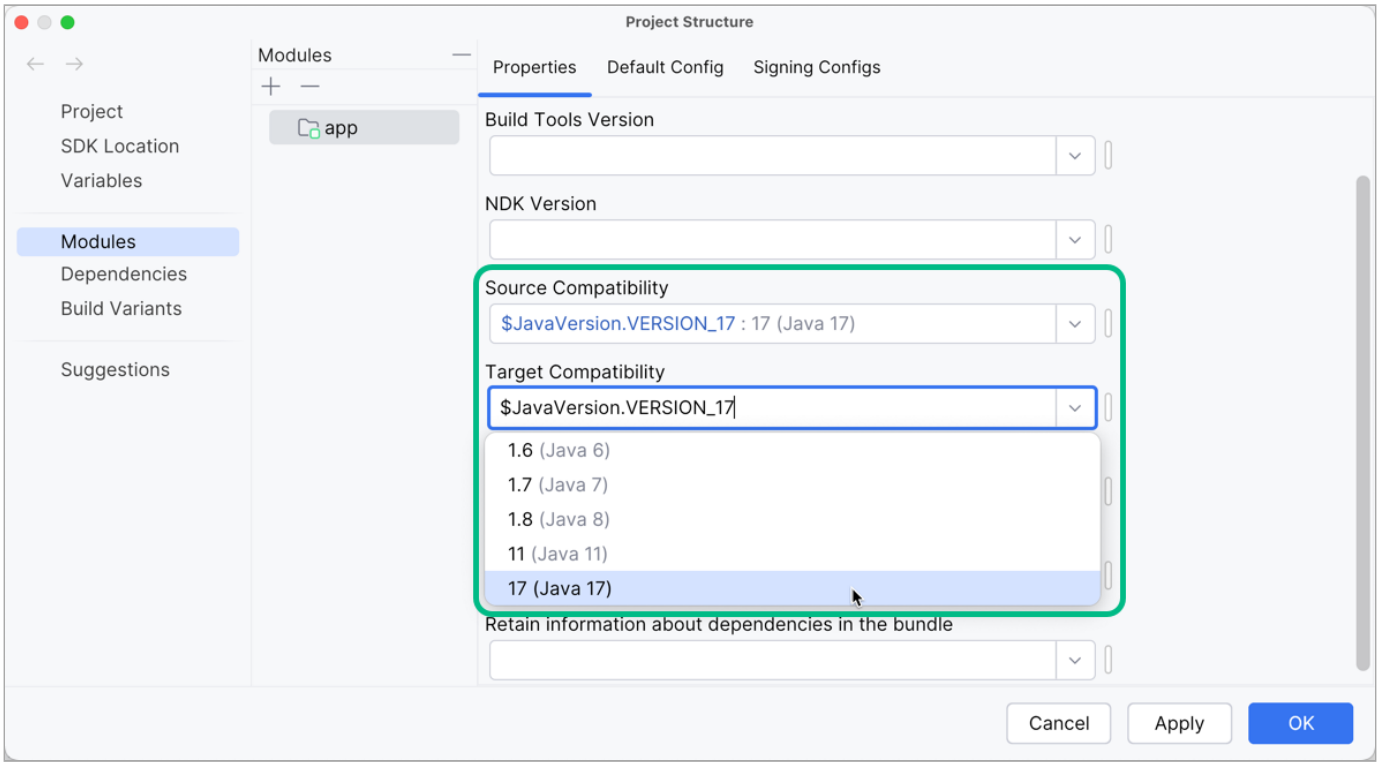


Figure 1. Selecting the Java version for a project in Android Studio

4. Click **OK**.

Add build dependencies

To use the Ping SDK for Android, add the relevant dependencies to your project:

1. In the **Project** tree view of your Android Studio project, open the `Gradle Scripts/build.gradle` file for the *module*.

2. In the `dependencies` section, add the following:

```
implementation 'org.forgerock:forgerock-auth:4.8.1'
```

Example of the dependencies section after editing:

```
dependencies {
    implementation 'org.forgerock:forgerock-auth:4.8.1'
    ...
    implementation 'androidx.appcompat:appcompat:1.6.1'
    implementation 'com.google.android.material:material:1.8.0'
    implementation 'androidx.constraintlayout:constraintlayout:2.1.4'
}
```

(Optional) Enable optional clear traffic and location support

If you are not using the PingOne Advanced Identity Cloud but rather a local PingAM server that does not use the HTTPS protocol, you can edit your project manifest file to allow cleartext connections.

> **Important**
>
> You should only configure this property during development against a local PingAM server.
> Do **not** configure this property in your production applications.

1. Open the project manifest file.

   For example, **app > manifests > AndroidManifest.xml**.

2. Add an `android:usesCleartextTraffic="true"` attribute to the `<application>` element.

(Optional) Enable location permissions

If you intend for your application to use any of the Android location services; for example, the SDK's location matching or geofencing features, add one of the relevant properties to the project manifest file

1. Open the project's manifest file.

   For example, **app > manifests > AndroidManifest.xml**.

2. Add the relevant properties as a child of the `<manifest>` element:

    1. Coarse location access

```
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
```

2. Fine location access (requires both permissions)

```
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
```

For information about which permission to use, see Location permissions in the *Google Developer Documentation*.

Example completed manifest file

The following shows an example `AndroidManifest.xml` file with support for cleartext traffic and fine location access enabled:

`AndroidManifest.xml`

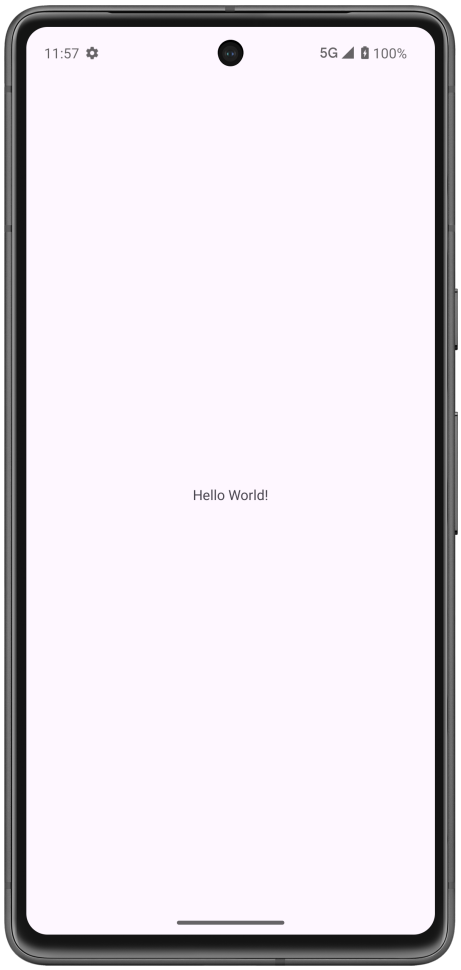
```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.ForgeRockSDKForAndroidQuickStart"
        tools:targetApi="31"
        android:usesCleartextTraffic="true">
        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>

            <meta-data
                android:name="android.app.lib_name"
                android:value="" />
        </activity>
    </application>

    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>

</manifest>
```

Check point

In Android Studio, select **Run** > **Run 'app'**.

Android Studio builds the application and runs it in the default emulator.

As you have not yet added any UI, the app displays only "Hello World!".



You have now configured your Android app development environment, created a new project, and configured it with the required dependencies and build options.

In the next step, you configure your application with the settings it needs to connect to your PingOne Advanced Identity Cloud or PingAM instance.

Step 2. Configure connection properties

In this step, you provide your application with the settings it needs to connect to your PingOne Advanced Identity Cloud or PingAM instance.

For example, which authentication tree to use and the realm it is a part of.

For this quick start guide, you must provide at least the following properties:

Property	Description
`forgerock_oauth_client_id`	The `client_id` of the OAuth 2.0 client profile to use.
`forgerock_oauth_redirect_uri`	The `redirect_uri` as configured in the OAuth 2.0 client profile. This value must match a value configured in your OAuth 2.0 client, but is not actually used by the Android application.
`forgerock_oauth_scope`	A list of scopes to request when performing an OAuth 2.0 authorization flow.
`forgerock_url`	The URL of the PingOne Advanced Identity Cloud or PingAM instance. For example, `https://openam-forgerock-sdks.forgeblocks.com/am` If you are *not* using PingOne Advanced Identity Cloud, specify the port and deployment path. For example, `https://openam.example.com:8443/openam`.
`forgerock_realm`	The realm in which the OAuth 2.0 client profile is configured. For example, `alpha` If you are *not* using PingOne Advanced Identity Cloud, specify the default PingAM the top-level realm; `root`.
`forgerock_auth_service`	The name of the journey to use for authentication. For example, `sdkUsernamePasswordJourney`
`forgerock_cookie_name`	The name of the cookie that contains the session token. To obtain the name of the cookie in the PingOne Advanced Identity Cloud: 1. Click your user in the top-right corner and select **Tenant settings**. 2. On the **Global Settings** tab, copy the value of the **Cookie** property. The value is a random string of characters, such as `29cd7a346b42b42`. If you are *not* using PingOne Advanced Identity Cloud, the cookie name is usually `iPlanetDirectoryPro`.

Property	Description
`forgerock_oauth_threshold`	A threshold, in seconds, to refresh an OAuth 2.0 token before the `access_token` expires (defaults to `30` seconds).
`forgerock_timeout`	A timeout, in seconds, for each request that communicates with PingAM.

Add required connection settings to your app

1. In the **Project** tree view of your Android Studio project, navigate to **app** > **res** > **values**, and then open the `strings.xml` file.
2. Inside the `<resources>` element, add the following elements, adjusting the values for your deployment:

```
<!-- OAuth 2.0 client details -->
<string name="forgerock_oauth_client_id" translatable="false">sdkPublicClient</string>
<string name="forgerock_oauth_redirect_uri" translatable="false">https://sdkapp.example.com:8443/callback</
string>
<string name="forgerock_oauth_scope" translatable="false">openid profile email address</string>

<!-- PingOne Advanced Identity Cloud details -->
<string name="forgerock_url" translatable="false">https://openam-forgerock-sdks.forgeblocks.com/am</string>
<string name="forgerock_cookie_name" translatable="false">iPlanetDirectoryPro</string>
<string name="forgerock_realm" translatable="false">alpha</string>

<!-- Journey details -->
<string name="forgerock_auth_service" translatable="false">sdkUsernamePasswordJourney</string>
```

Check point

You have now configured your application with the settings it needs to connect to your PingOne Advanced Identity Cloud or PingAM instance.

In the next step, you add debug logging and initialize the SDK.

Step 3. Initialize the SDK

In this step, you enable debug logging during development.

You then add a call to the `FRAuth.start()` method, which initializes the SDK and loads the configuration you have defined in the previous step.

Enable debug logging and initialize the SDK

1. Open the project's `MainActivity` class file.

   For example, **app > java > com.example.quickstart > MainActivity**.
2. Enable debug logging and initialize the SDK in the `onCreate()` method after the generated code:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
 // Add these lines:
   Logger.set(Logger.Level.DEBUG);
    FRAuth.start(this);
}
```

3. Add the required import statements for `org.forgerock.android.auth.FRAuth` and `org.forgerock.android.auth.Logger`.

```
import org.forgerock.android.auth.FRAuth;
import org.forgerock.android.auth.Logger;
```

Check point

You have now added debug logging to your app and initialized the SDK.

1. To test the setup so far, in Android Studio, select **Run > Run 'app'**.

   If everything is configured correctly, the app builds, and the default emulator will run the application.

2. Open the **Logcat** pane. The SDK will generate output similar to the following if everything is configured correctly:

```
-- PROCESS STARTED (14305) for package com.example
[4.8.1] [DefaultTokenManager]: Using SharedPreference: StorageDelegate
```

> **Tip**
>
> In the Logcat filter bar, enter `tag:ForgeRock` to only view output from the SDK.

If you get errors when running the app, check the **app > res > values > strings.xml** has the correct values. Refer to Step 2. Configure connection properties.

In the next step, you create the initial user interface to display the current authentication status, and add buttons to log in and log out.

Step 4. Create a status view

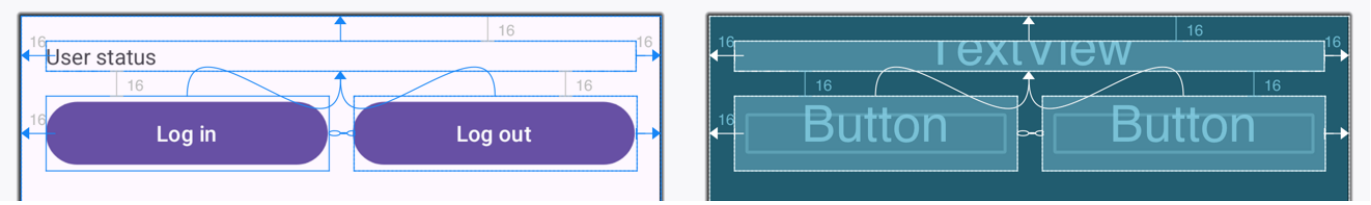
In this step, you create a layout and add buttons to log in and log out your user, as well as a text view field to show their current authentication status.

You also add the code to update the value displayed in the text view.

Create a layout for the status view

1. Navigate to **app > res > layout** and open `activity_main.xml`.
2. Select and delete the existing `TextView` element that contains the text `Hello World!`.
3. From the **Palette** pane, drag a new `TextView` element to the canvas:
   - **id**: `textViewUserStatus`
   - **text**: `User status`
4. From the **Palette** pane, drag a new `Button` element to the canvas:
   - **id**: `buttonLogin`
   - **text**: `Log in`
5. From the **Palette** pane, drag a second new `Button` element to the canvas:
   - **id**: `buttonLogout`
   - **text**: `Log out`
6. Layout the elements on the canvas to your liking.

   The following screenshot shows one possibility:



```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:id="@+id/textViewUserStatus"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="16dp"
        android:layout_marginTop="16dp"
        android:layout_marginEnd="16dp"
        android:text="User status"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <Button
        android:id="@+id/buttonLogin"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="16dp"
        android:layout_marginTop="16dp"
        android:layout_marginEnd="8dp"
        android:text="Log in"
        app:layout_constraintEnd_toStartOf="@+id/buttonLogout"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/textViewUserStatus" />

    <Button
        android:id="@+id/buttonLogout"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="8dp"
        android:layout_marginTop="16dp"
        android:layout_marginEnd="16dp"
        android:text="Log out"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toEndOf="@+id/buttonLogin"
        app:layout_constraintTop_toBottomOf="@+id/textViewUserStatus" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Add a function to update the status view

1. Open the project's `MainActivity` class file.

   For example, **app > java > com.example.quickstart > MainActivity**.

2. Add the following statements *before* the definition of the `onCreate()` function:

```
private TextView status;
private Button loginButton;
private Button logoutButton;
```

3. Add import statements for the `FRUser` module, and for `android.widget.Button` and `android.widget.TextView`.

```
import org.forgerock.android.auth.FRUser;
import android.widget.Button;
import android.widget.TextView;
```

4. In the `onCreate()` function, after the call to `FRAuth.start()`, add references to the elements on the status view layout:

```
// Add references to status view elements
status = findViewById(R.id.textViewUserStatus);
loginButton = findViewById(R.id.buttonLogin);
logoutButton = findViewById(R.id.buttonLogout);
updateStatus();
```

5. Add the following function *after* the existing `onCreate()` function:

```
private void updateStatus() {
    runOnUiThread(() -> {
        if (FRUser.getCurrentUser() == null) {
            status.setText("User is not authenticated.");
            loginButton.setEnabled(true);
            logoutButton.setEnabled(false);
        } else {
            status.setText("User is authenticated.");
            loginButton.setEnabled(false);
            logoutButton.setEnabled(true);
        }
    });
}
```

Check point

In Android Studio, select **Run > Run 'app'**.

If everything is configured correctly, the app builds, and the default emulator runs the application.

The app shows the **[Log in]** and **[Log out]** buttons, as well as a text view element that displays `User is not authenticated`:

In the next step, you create and attach functions to the buttons to start the authentication journey, or begin the logout process.

Step 5. Add login and logout calls

In this step, you update the app with the `NodeListener` interface, which manages the client side of the authentication journey.

The interface provides methods to handle the results of the authentication journey:

`onSuccess()`

The authentication journey is complete and an `FRUser` object is now available for further use.

For example, you could display the user's name in your app.

`onCallbackReceived()`

Recursively handle each step within the authentication journey, by completing and returning any callbacks received.

For example, in this quick start guide we receive `NameCallback` and `PasswordCallback` callbacks. In the next step, we create the UI to request these credentials from the user.

`onException()`

Handle any errors.

Implement `NodeListener` and methods

1. Edit the `MainActivity` class so that it implements `NodeListener<FRUser>`:

```
public class MainActivity extends AppCompatActivity implements NodeListener<FRUser> {
```

2. Add import statements for `org.forgerock.android.auth.NodeListener` and `org.forgerock.android.auth.Node`:

```
import org.forgerock.android.auth.NodeListener;
import org.forgerock.android.auth.Node;
```

3. At the bottom of the `MainActivity` class, add the handler methods from the `NodeListener` interface:

```
public class MainActivity extends AppCompatActivity implements NodeListener<FRUser> {

  // …
  // …
  // …

  @Override
    public void onSuccess(FRUser result) {
      updateStatus();
    }

    @Override
    public void onCallbackReceived(Node node) {
      // Display appropriate UI to handle callbacks
    }

    @Override
    public void onException(Exception e) {
      Logger.error(TAG, e.getMessage(), e);
    }
}
```

4. Attach `FRUser.login()` and `FRUser.logout()` calls to the appropriate buttons, after the `updateStatus()` call:

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Logger.set(Logger.Level.DEBUG);
    FRAuth.start(this);
    // Add references to status view elements
    status = findViewById(R.id.textViewUserStatus);
    loginButton = findViewById(R.id.buttonLogin);
    logoutButton = findViewById(R.id.buttonLogout);
    updateStatus();

    // Attach 'FRUser.login()' to 'loginButton'
    loginButton.setOnClickListener(view → FRUser.login(getApplicationContext(), this));

    // Attach 'FRUser.getCurrentUser().logout()' to 'logoutButton'
    logoutButton.setOnClickListener(view → {
        FRUser.getCurrentUser().logout();
        updateStatus();
    });
}
```

Check point

1. In Android Studio, select **Run > Run 'app'**.

   If everything is configured correctly, the app builds, and the emulator runs the application.

2. In the Emulator, click the **[Log in]** button.

   In the **Run** pane, you should see the following to indicate that the journey was found and the callbacks were returned. In our case, a `NameCallback` and `PasswordCallback` callback, as configured in the page node:

   ```
   [4.8.1] [AuthServiceResponseHandler]: Journey callback(s) received.
   ```

In the next step, you add a UI fragment to obtain credentials from the user, and code to open that fragment when the callback is received.

You also add code to populate the callback with the credentials and return it to the server, completing the authentication journey.

Step 6. Create UI to handle the callbacks

In this step, you add a UI fragment to obtain credentials from the user, and code to open that fragment when a callback is received.

The authentication journey in this quick start guide sends the `NameCallback` and `PasswordCallback` callbacks.

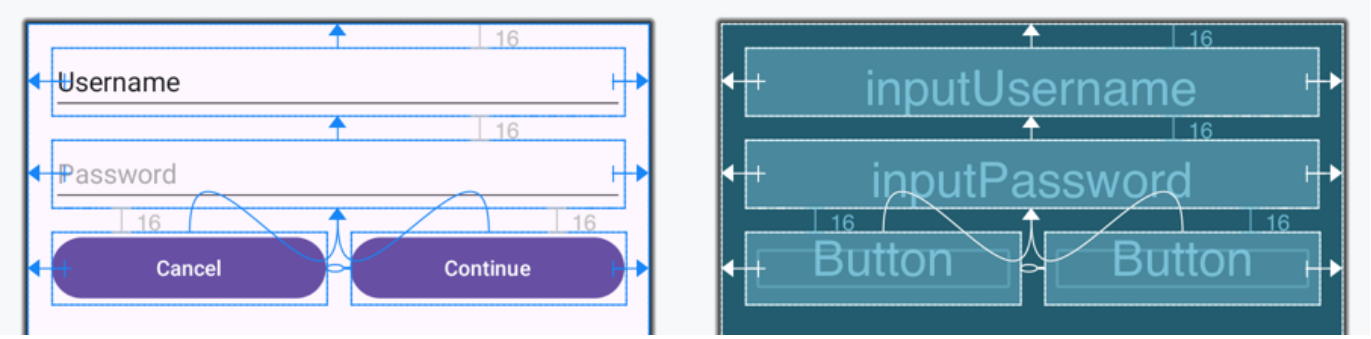
For demonstration purposes, this application uses a `DialogFragment` to collect the username and password.

You also add code to populate the callback with the credentials and return it to the server, completing the authentication journey.

Create a UI fragment

1. Navigate to **app > res**.
2. Right-click **layout** and select **New > Fragment > Fragment (Blank)**.
3. In the **New Android Component** dialog, enter the following values, and then click **[Finish]**:
    - **Fragment Name**: `usernamePasswordFragment`
    - **Fragment Layout Name**: `fragment_username_password`
    - **Source Language**: `Java`
4. Navigate to **app > res > layout** and open `fragment_username_password.xml`.
5. Select and delete the existing `TextView` element that contains the text `Hello blank fragment`.
6. In the **Component Tree** pane, right-click the **FrameLayout** component, select **Convert FrameLayout to ConstraintLayout**, and then click **[OK]**.
7. In the **Palette** pane, from the **Text** category drag a `Plain Text` input element to the canvas:
    - **id**: `inputUsername`
    - **text**: `Username`
8. Drag a `Password` element to the canvas:
    - **id**: `inputPassword`
    - **hint**: `Password`
9. In the **Palette** pane, from the **Button** category, drag a `Button` element to the canvas:
    - **id**: `buttonCancel`
    - **text**: `Cancel`
10. Drag a second `Button` element to the canvas:
    - **id**: `buttonContinue`
    - **text**: `Continue`
11. Layout the elements on the canvas to your liking.

    The following screenshot shows one possibility:



```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/frameLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".usernamePasswordFragment">

    <EditText
        android:id="@+id/inputUsername"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="16dp"
        android:layout_marginTop="16dp"
        android:layout_marginEnd="16dp"
        android:ems="10"
        android:inputType="text"
        android:text="Username"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <EditText
        android:id="@+id/inputPassword"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="16dp"
        android:layout_marginTop="16dp"
        android:layout_marginEnd="16dp"
        android:ems="10"
        android:hint="Password"
        android:inputType="textPassword"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/inputUsername" />

    <Button
        android:id="@+id/buttonCancel"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="16dp"
        android:layout_marginTop="16dp"
        android:layout_marginEnd="8dp"
        android:text="Cancel"
        app:layout_constraintEnd_toStartOf="@+id/buttonContinue"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/inputPassword"
        tools:text="Cancel" />

    <Button
        android:id="@+id/buttonContinue"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="8dp"
        android:layout_marginTop="16dp"
        android:layout_marginEnd="16dp"
```

```
        android:text="Continue"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toEndOf="@+id/buttonCancel"
        app:layout_constraintTop_toBottomOf="@+id/inputPassword" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Configure the fragment code

1. Open `usernamePasswordFragment.java`

   For example, **app > java > com.example.quickstart > usernamePasswordFragment**.

2. Update the class to extend `DialogFragment` rather than `Fragment`, which makes opening and closing the fragment easier:

   ```
   public class usernamePasswordFragment extends DialogFragment {
   ```

3. Add import statements for `androidx.fragment.app.DialogFragment`:

   ```
   import androidx.fragment.app.DialogFragment;
   ```

4. Within the `usernamePasswordFragment` class, initialize required variables:

   ```
   private MainActivity listener;
   private Node node;
   ```

5. Update the `newInstance` method to accept a node object as its only parameter:

   ```
   public static usernamePasswordFragment newInstance(Node node) {
       usernamePasswordFragment fragment = new usernamePasswordFragment();
       Bundle args = new Bundle();
       args.putSerializable("NODE", node);
       fragment.setArguments(args);
       return fragment;
   }
   ```

6. Insert an `onResume()` method below the `newInstance()` method. This correctly sizes the fragment dialog when displayed:

```
@Override
public void onResume() {
    super.onResume();
    ViewGroup.LayoutParams params = getDialog().getWindow().getAttributes();
    params.width = ViewGroup.LayoutParams.MATCH_PARENT;
    params.height = ViewGroup.LayoutParams.WRAP_CONTENT;
    getDialog().getWindow().setAttributes((android.view.WindowManager.LayoutParams) params);
}
```

7. Delete the `onCreate()` function.

8. Update the `onCreateView` method to capture the values from the fields in the fragment and populate the callbacks the node returned:

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
                         Bundle savedInstanceState) {
    // Inflate the layout for this fragment
    View view = inflater.inflate(R.layout.fragment_username_password, container, false);
    node = (Node) getArguments().getSerializable("NODE");
    AppCompatEditText username = view.findViewById(R.id.inputUsername);
    AppCompatEditText password = view.findViewById(R.id.inputPassword);
    Button next = view.findViewById(R.id.buttonContinue);
    next.setOnClickListener(v -> {
        dismiss();
        node.getCallback(NameCallback.class)
                .setName(username.getText().toString());
        node.getCallback(PasswordCallback.class)
                .setPassword(password.getText().toString().toCharArray());
        node.next(getContext(), listener);
    });
    Button cancel = view.findViewById(R.id.buttonCancel);
    cancel.setOnClickListener(v -> {
        dismiss();
    });
    return view;
}
```

9. Add an `onAttach()` method after the `onCreateView()` method. This ensures the fragment is correctly connected to the main activity:

```
@Override
public void onAttach(@NonNull Context context) {
    super.onAttach(context);
    if (context instanceof MainActivity) {
        listener = (MainActivity) context;
    }
}
```

10. Add any missing required import statements:

```
import android.content.Context;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.Button;

import androidx.annotation.NonNull;
import androidx.appcompat.widget.AppCompatEditText;
import androidx.fragment.app.DialogFragment;

import org.forgerock.android.auth.Node;
import org.forgerock.android.auth.callback.NameCallback;
import org.forgerock.android.auth.callback.PasswordCallback;
```

```
package com.example.quickstart;

import android.os.Bundle;
import androidx.fragment.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import androidx.fragment.app.DialogFragment;
import android.content.Context;
import android.widget.Button;
import androidx.annotation.NonNull;
import androidx.appcompat.widget.AppCompatEditText;
import org.forgerock.android.auth.Node;
import org.forgerock.android.auth.callback.NameCallback;
import org.forgerock.android.auth.callback.PasswordCallback;

/**
 * A simple {@link Fragment} subclass.
 * Use the {@link usernamePasswordFragment#newInstance} factory method to
 * create an instance of this fragment.
 */
public class usernamePasswordFragment extends DialogFragment {

    private MainActivity listener;
    private Node node;

    public usernamePasswordFragment() {
        // Required empty public constructor
    }

    public static usernamePasswordFragment newInstance(Node node) {
        usernamePasswordFragment fragment = new usernamePasswordFragment();
        Bundle args = new Bundle();
        args.putSerializable("NODE", node);
        fragment.setArguments(args);
        return fragment;
    }

    @Override
    public void onResume() {
        super.onResume();
        ViewGroup.LayoutParams params = getDialog().getWindow().getAttributes();
        params.width = ViewGroup.LayoutParams.MATCH_PARENT;
        params.height = ViewGroup.LayoutParams.WRAP_CONTENT;
        getDialog().getWindow().setAttributes((android.view.WindowManager.LayoutParams) params);
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        // Inflate the layout for this fragment
        View view = inflater.inflate(R.layout.fragment_username_password, container, false);
        node = (Node) getArguments().getSerializable("NODE");
        AppCompatEditText username = view.findViewById(R.id.inputUsername);
        AppCompatEditText password = view.findViewById(R.id.inputPassword);
        Button next = view.findViewById(R.id.buttonContinue);
        next.setOnClickListener(v -> {
            dismiss();
            node.getCallback(NameCallback.class)
                    .setName(username.getText().toString());
```

```
                node.getCallback(PasswordCallback.class)
                        .setPassword(password.getText().toString().toCharArray());
                node.next(getContext(), listener);
            });
            Button cancel = view.findViewById(R.id.buttonCancel);
            cancel.setOnClickListener(v -> {
                dismiss();
            });
            return view;
        }

        @Override
        public void onAttach(@NonNull Context context) {
            super.onAttach(context);
            if (context instanceof MainActivity) {
                listener = (MainActivity) context;
            }
        }
    }
```

Open the fragment when receiving callbacks

1. Open the project's `MainActivity` class file.

   For example, **app > java > com.example.quickstart > MainActivity**.

2. Update the `onCallbackReceived()` method to open the fragment to gather the credentials:

   ```
   @Override
   public void onCallbackReceived(Node node) {
       usernamePasswordFragment fragment = usernamePasswordFragment.newInstance(node);
       fragment.show(getSupportFragmentManager(), usernamePasswordFragment.class.getName());
   }
   ```

Check point

You have now completed the quick start application.

You added a UI fragment to obtain credentials from the user, and code to open that fragment when the callback is received.

You also added code to populate the callback with the credentials and return it to the server, completing the authentication journey.

In the next step, you test the application by authenticating a user, checking the logs, and then logging out.

Step 7. Test the app

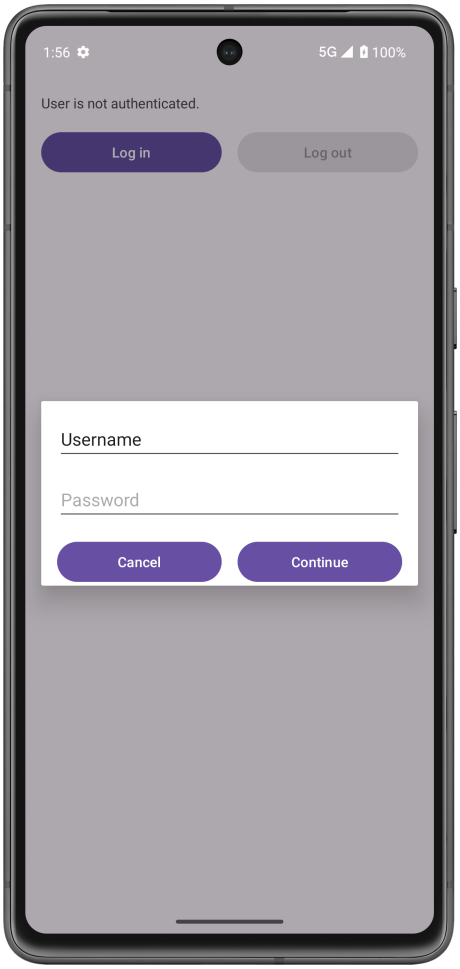
In this step, you run and test your application.

You run it in the emulator or on your Android device, perform authentication with a demo user, check the log for success messages, and then log out the user.

Log in as a demo user

1. In Android Studio, select **Run > Run 'app'**.
2. Click **[Log in]**.

   The fragment dialog appears, with fields for both name and password, as well as continue and cancel buttons:



3. Sign on as a demo user:
   - **Name**: `demo`
   - **Password**: `Ch4ng3it!`

4. Click **[Continue]**.

   If authentication is successful, the application returns to the main screen, and displays `User is authenticated`.

5. Open the **Logcat** pane.

   If authentication was successful, the log contains entries similar to the following:

   ```
   [4.8.1] [AuthServiceResponseHandler]: Journey finished with Success outcome.
   [4.8.1] [AuthServiceResponseHandler]: SSO Token received.
   ```

   If authentication fails:

   - Check the credentials you are using are correct.

     For example, attempt to log directly into your ID Cloud or PingAM instance using them.

   - Check your `strings.xml` has the correct values for your environment

6. Click the **[Log out]** button.

   If logout is successful, the application displays `User is not authenticated`.

   The **Logcat** pane contains entries similar to the following:

   ```
   [4.6.0] [OAuth2ResponseHandler]: Revoke success
   [4.6.0] [DefaultTokenManager]: Revoking AccessToken & Refresh Token Success
   ```

You have successfully completed the tutorial.

Next Steps

- Update your app to handle additional supported callbacks.
- Improve the security of your application by adding SSL pinning.
- Add the ability to update your configuration without reinstalling the app.
- Offer "magic links" to your users by adding support for suspending and resuming authentication.

Authentication journey tutorial for iOS

Prepare › Download › Configure › Run

In this tutorial you update a sample app to step through an authentication journey, meaning you get to design and implement the user interface to your requirements.

The sample navigates through a simple authentication journey, and obtains OAuth 2.0 tokens for the user.

Before you begin

Before you begin this tutorial ensure you have set up your PingOne Advanced Identity Cloud tenant or PingAM server with the required configuration.

For example, you will need an OAuth 2.0 client application set up, as well as an authentication journey for the app to navigate.

Complete prerequisites »

Step 1. Download the samples

To start this tutorial, you need to download the SDK sample apps repo, which contains the projects you will use.

Start step 1 »

Step 2. Configure connection properties

In this step, you configure the "uikit-quickstart" sample app to connect to the OAuth 2.0 application you created in PingOne Advanced Identity Cloud or PingAM.

Start step 2 »

Step 3. Test the app

In this step, you will test your application.

You run it in the emulator or on your iOS device, perform authentication with a demo user, check the log for success messages, and then log out the user.

Test app »

Before you begin

Prepare › Download › Configure › Run

To successfully complete this tutorial refer to the prerequisites and compatibility requirements in this section.

The tutorial also requires a configured server.

Compatibility

iOS

This sample app is compatible with iOS 12 and later.

Prerequisites

Xcode

You can download the latest version for free from https://developer.apple.com/xcode/.

Server configuration

This tutorial requires you to configure one of the following servers:

PingOne Advanced Identity Cloud

PingAM

PingOne Advanced Identity Cloud

The samples and tutorials in this documentation often require that you have an identity set up so that you can test authentication.

To create a demo user in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. In the left panel, click **Identities** > **Manage**.
3. Click **+ New Alpha realm - User**.
4. Enter the following details:
    - **Username** = `demo`
    - **First Name** = `Demo`
    - **Last Name** = `User`
    - **Email Address** = `demo.user@example.com`
    - **Password** = `Ch4ng3it!`
5. Click **Save**.

Authentication journeys provide fine-grained authentication by allowing multiple paths and decision points throughout the flow. Authentication journeys are made up of nodes that define actions taken during authentication.

Each node performs a single task, such as collecting a username or making a simple decision. Nodes can have multiple outcomes rather than just success or failure. For details, see the Authentication nodes configuration reference in the PingAM documentation.

To create a simple journey for use when testing the Ping SDKs, follow these steps:

1. In your PingOne Advanced Identity Cloud tenant, navigate to **Journeys**, and click **+ New Journey**.
2. Enter a name, such as `sdkUsernamePasswordJourney` and click **Save**.

   The authentication journey designer appears.
3. Drag the following nodes into the designer area:
   - **Page Node**
   - **Platform Username**
   - **Platform Password**
   - **Data Store Decision**
4. Drag and drop the **Platform Username** and **Platform Password** nodes onto the **Page Node**, so that they both appear on the same page when logging in.
5. Connect the nodes as follows:

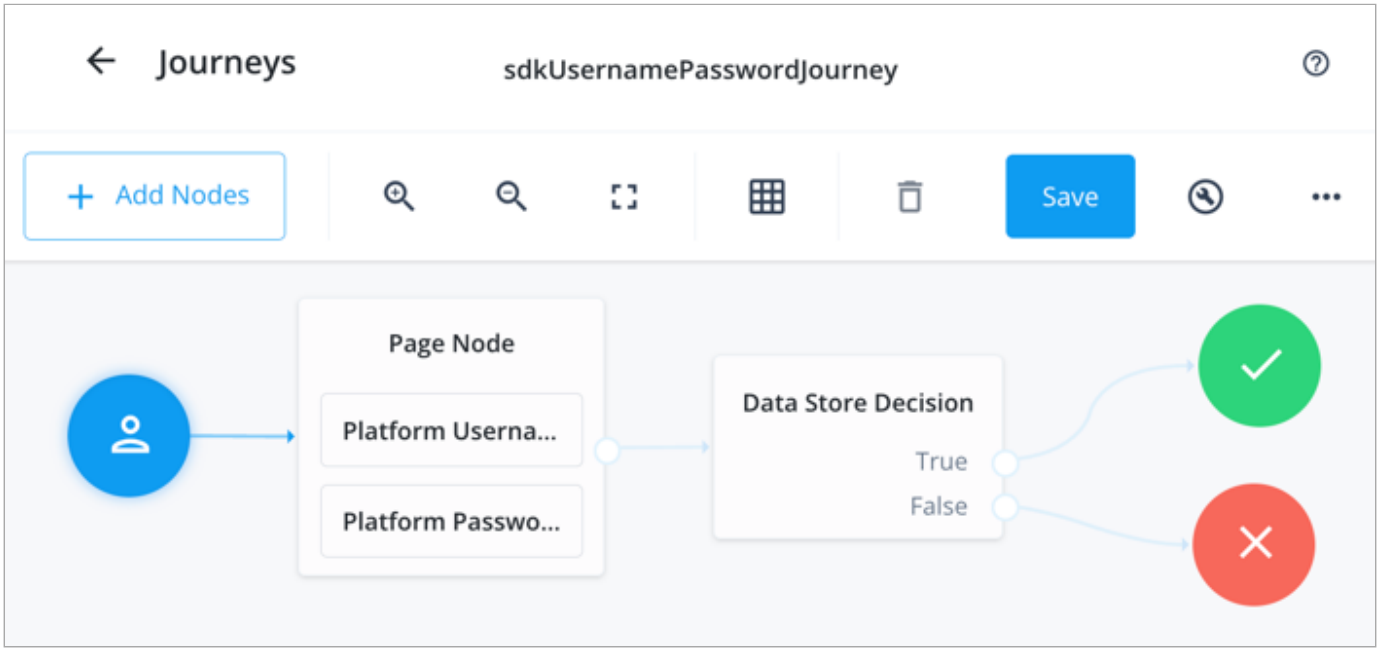


Figure 1. Example username and password authentication journey

6. Click **Save**.

Public clients do not use a client secret to obtain tokens because they are unable to keep them hidden. The Ping SDKs commonly use this type of client to obtain tokens, as they cannot guarantee safekeeping of the client credentials in a browser or on a mobile device.

To register a *public* OAuth 2.0 client application for use with the SDKs in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. In the left panel, click **Applications**.
3. Click **+ Custom Application**.
4. Select **OIDC - OpenId Connect** as the sign-in method, and then click **Next**.
5. Select **Native / SPA** as the application type, and then click **Next**.
6. In **Name**, enter a name for the application, such as `Public SDK Client`.
7. In **Owners**, select a user that is responsible for maintaining the application, and then click **Next**.

   > **Tip**
   >
   > When trying out the SDKs, you could select the `demo` user you created previously.

8. In **Client ID**, enter `sdkPublicClient`, and then click **Create Application**.

   PingOne Advanced Identity Cloud creates the application and displays the details screen.
9. On the **Sign On** tab:
   1. In **Sign-In URLs**, enter the following values:

      `org.forgerock.demo://oauth2redirect`

      > **Important**
      >
      > Also add any other domains where you host SDK applications.

   2. In **Grant Types**, enter the following values:

      `Authorization Code`

      `Refresh Token`

   3. In **Scopes**, enter the following values:

      `openid profile email address`
10. Click Show advanced settings, and on the **Authentication** tab:
    1. In **Token Endpoint Authentication Method**, select `none`.
    2. In **Client Type**, select `Public`.
    3. Enable the **Implied Consent** property.
11. Click **Save**.

The application is now configured to accept client connections from and issue OAuth 2.0 tokens to the example applications and tutorials covered by this documentation.

The provider specifies the supported OAuth 2.0 configuration options for a realm.

To ensure the PingOne Advanced Identity Cloud OAuth 2.0 provider service is configured for use with the Ping SDKs, follow these steps:

1. In your PingOne Advanced Identity Cloud tenant, navigate to **Native Consoles** > **Access Management**.
2. In the left panel, click **Services**.
3. In the list of services, click **OAuth2 Provider**.
4. On the **Core** tab, ensure **Issue Refresh Tokens** is enabled.
5. On the **Consent** tab, ensure **Allow Clients to Skip Consent** is enabled.
6. Click **Save Changes**.

PingAM

The samples and tutorials in this documentation often require that you have an identity set up so that you can test authentication.

To create a demo user in PingAM, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. Navigate to **Identities**, and then click **+ Add Identity**.
3. Enter the following details:
   - **User ID** = `demo`
   - **Password** = `Ch4ng3it!`
   - **Email Address** = `demo.user@example.com`
4. Click **Create**.

Authentication trees provide fine-grained authentication by allowing multiple paths and decision points throughout the authentication flow. Authentication trees are made up of nodes that define actions taken during authentication.

Each node performs a single task, such as collecting a username or making a simple decision. Nodes can have multiple outcomes rather than just success or failure. For details, see the Authentication nodes configuration reference in the PingAM documentation.

To create a simple tree for use when testing the Ping SDKs, follow these steps:

1. Under **Realm Overview**, click **Authentication Trees**, then click **Create Tree**.
2. Enter a tree name, for example `sdkUsernamePasswordJourney`, and then click **Create**.

The authentication tree designer appears, showing the **Start** entry point connected to the **Failure** exit point.

3. Drag the following nodes from the **Components** panel on the left side into the designer area:
    - **Page Node**
    - **Username Collector**
    - **Password Collector**
    - **Data Store Decision**
4. Drag and drop the **Username Collector** and **Password Collector** nodes onto the **Page Node**, so that they both appear on the same page when logging in.
5. Connect the nodes as follows:

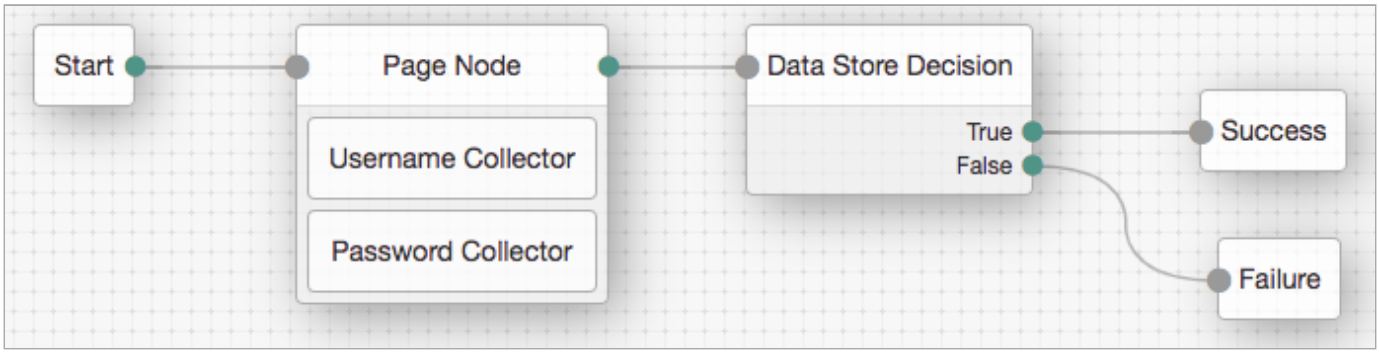


Figure 2. Example username and password authentication tree

6. Select the **Page Node**, and in the **Properties** pane, set the **Stage** property to `UsernamePassword`.

> **Tip**
>
> You can configure the node properties by selecting a node and altering properties in the right-hand panel.

One of the samples uses this specific value to determine the custom UI to display.

7. Click **Save**.

Public clients do not use a client secret to obtain tokens because they are unable to keep them hidden. The Ping SDKs commonly use this type of client to obtain tokens, as they cannot guarantee safekeeping of the client credentials in a browser or on a mobile device.

To register a *public* OAuth 2.0 client application for use with the SDKs in AM, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. Navigate to **Applications** > **OAuth 2.0** > **Clients**, and then click **+ Add Client**.
3. In **Client ID**, enter `sdkPublicClient`.
4. Leave **Client secret** empty.
5. In **Redirection URIs**, enter the following values:

```
org.forgerock.demo://oauth2redirect
```

> **Important**
>
> Also add any other domains where you will be hosting SDK applications.

6. In **Scopes**, enter the following values:

   `openid profile email address`

7. Click **Create**.

   PingAM creates the new OAuth 2.0 client, and displays the properties for further configuration.

8. On the **Core** tab:

   1. In **Client type**, select `Public`.
   2. Disable **Allow wildcard ports in redirect URIs**.
   3. Click **Save Changes**.

9. On the **Advanced** tab:

   1. In **Grant Types**, enter the following values:

      ```
      Authorization Code
      Refresh Token
      ```

   2. In **Token Endpoint Authentication Method**, select `None`.
   3. Enable the **Implied consent** property.

10. Click **Save Changes**.

The provider specifies the supported OAuth 2.0 configuration options for a realm.

To ensure the PingAM OAuth 2.0 provider service is configured for use with the Ping SDKs, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. In the left panel, click **Services**.
3. In the list of services, click **OAuth2 Provider**.
4. On the **Core** tab, ensure **Issue Refresh Tokens** is enabled.
5. On the **Consent** tab, ensure **Allow Clients to Skip Consent** is enabled.
6. Click **Save Changes**.

Step 1. Download the samples

Prepare › Download › Configure › Run

To complete this tutorial, you need to download the SDK sample apps repo, which contains the projects you will use.

1. In a web browser, navigate to the SDK Sample Apps repository.
2. Download the source code using one of the following methods:

Download a ZIP file

1. Click **Code**, and then click **Download ZIP**.
2. Extract the contents of the downloaded ZIP file to a suitable location.

Use a Git-compatible tool to clone the repo locally

1. Click **Code**, and then copy the HTTPS URL.
2. Use the URL to clone the repository to a suitable location.

   For example, from the command-line you could run:

```
git clone https://github.com/ForgeRock/sdk-sample-apps.git
```

The result of these steps is a local folder named `sdk-sample-apps`.

Step 2. Configure connection properties

Prepare › Download › **Configure** › Run

In this step, you configure the "uikit-quickstart" sample app to connect to the OAuth 2.0 application you created in PingOne Advanced Identity Cloud or PingAM.

1. In Xcode, on the **File** menu, click **Open**.
2. Navigate to the `sdk-sample-apps` folder you cloned in the previous step, navigate to `iOS` > `uikit-quickstart` > `Quickstart.xcodeproj`, and then click **Open**.
3. Choose how you want to configure the sample app. You can either configure the sample by using dynamic configuration, or by updating an Apple PLIST file.

Dynamic configuration

1. In the **Project Navigator** pane, navigate to **Quickstart > Quickstart**, and open the `LoginViewController` file.
2. Replace the call to `try FRAuth.start()` with the following code:

```
let options = FROptions(
  url: "{as_url}",
  cookieName: "{cookie_name}",
  realm: "{realm_path}",
  oauthClientId: "{oauth2_client_id}",
  oauthRedirectUri: "{oauth2_redirect}",
  oauthScope: "{oauth2_scopes}",
  authServiceName: "Login",
  registrationServiceName: "Register")

try FRAuth.start(options: options)
```

3. Replace the following strings with the values you obtained when you registered the OAuth 2.0 application:

{as_url}

The base URL of the server to connect to.

Identity Cloud example:

`https://openam-forgerock-sdks.forgeblocks.com/am`

Self-hosted example:

`https://openam.example.com:8443/openam`

{cookie_name}

The name of the cookie that contains the session token.

For example, with a self-hosted PingAM server this value might be `iPlanetDirectoryPro`.

> **Tip**
>
> PingOne Advanced Identity Cloud tenants use a random alpha-numeric string.
> To locate the cookie name in an PingOne Advanced Identity Cloud tenant, navigate to **Tenant settings** > **Global Settings**, and copy the value of the **Cookie** property.
> For example, `ch15fefc5407912`

{realm_path}

The realm in which the OAuth 2.0 client profile and authentication journeys are configured.

Usually, `root` for AM and `alpha` or `bravo` for Advanced Identity Cloud.

{oauth2_client_id}

The client ID of your OAuth 2.0 application in PingOne Advanced Identity Cloud or PingAM.

For example, `sdkNativeClient`

{oauth2_redirect}

The `redirect_uri` as configured in the OAuth 2.0 client profile.

> **Note**
>
> This value must exactly match a value configured in your OAuth 2.0 client.

For example, `org.forgerock.demo://oauth2redirect`.

{oauth2_scopes}

The scopes you added to your OAuth 2.0 application in PingOne Advanced Identity Cloud.

For example, `address email openid phone profile`

The result resembles the following:

```
let options = FROptions(
  url: "https://openam-forgerock-sdks.forgeblocks.com/am",
  cookieName: "ch15fefc5407912",
  realm: "alpha",
  oauthClientId: "sdkPublicClient",
  oauthRedirectUri: "org.forgerock.demo://oauth2redirect",
  oauthScope: "openid profile email address",
  authServiceName: "Login",
  registrationServiceName: "Register")

try FRAuth.start(options: options)
```

PLIST file

1. In the navigator pane in Xcode, right-click `FRAuthConfig` and select **Open As** > **Source Code**.
2. Replace the existing file content with the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/
PropertyList-1.0.dtd">
<plist version="1.0">
  <dict>
     <key>forgerock_url</key>
     <string>{as_url}</string>
     <key>forgerock_cookie_name</key>
     <string>{cookie_name}</string>
     <key>forgerock_realm</key>
     <string>{realm_path}</string>
     <key>forgerock_oauth_client_id</key>
     <string>{oauth2_client_id}</string>
     <key>forgerock_oauth_redirect_uri</key>
     <string>{oauth2_redirect}</string>
     <key>forgerock_oauth_scope</key>
     <string>openid profile email address</string>
     <key>forgerock_oauth_threshold</key>
     <string>60</string>
     <key>forgerock_timeout</key>
     <string>60</string>
     <key>forgerock_auth_service_name</key>
     <string>sdkUsernamePasswordJourney</string>
     <key>forgerock_registration_service_name</key>
     <string>Registration</string>
  </dict>
</plist>
```

3. Replace the following strings with the values you obtained when you registered the OAuth 2.0 application:

{as_url}

The base URL of the server to connect to.

Identity Cloud example:

`https://openam-forgerock-sdks.forgeblocks.com/am`

Self-hosted example:

`https://openam.example.com:8443/openam`

{cookie_name}

The name of the cookie that contains the session token.

For example, with a self-hosted PingAM server this value might be `iPlanetDirectoryPro`.

> **Tip**
>
> PingOne Advanced Identity Cloud tenants use a random alpha-numeric string.
> To locate the cookie name in an PingOne Advanced Identity Cloud tenant, navigate to **Tenant settings** > **Global Settings**, and copy the value of the **Cookie** property.

{realm_path}

The realm in which the OAuth 2.0 client profile and authentication journeys are configured.

Usually, `root` for AM and `alpha` or `bravo` for Advanced Identity Cloud.

{oauth2_client_id}

The client ID of your OAuth 2.0 application in PingOne Advanced Identity Cloud or PingAM.

For example, `sdkNativeClient`

{oauth2_redirect}

The `redirect_uri` as configured in the OAuth 2.0 client profile.

> **Note**
>
> This value must exactly match a value configured in your OAuth 2.0 client.

For example, `org.forgerock.demo://oauth2redirect`.

The result resembles the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/
PropertyList-1.0.dtd">
<plist version="1.0">
  <dict>
     <key>forgerock_url</key>
     <string>https://openam.example.com:8443/openam</string>
     <key>forgerock_cookie_name</key>
     <string>iPlanetDirectoryPro</string>
     <key>forgerock_realm</key>
     <string>alpha</string>
     <key>forgerock_oauth_client_id</key>
     <string>sdkNativeClient</string>
     <key>forgerock_oauth_redirect_uri</key>
     <string>org.forgerock.demo://oauth2redirect</string>
     <key>forgerock_oauth_scope</key>
     <string>openid profile email address</string>
     <key>forgerock_oauth_threshold</key>
     <string>60</string>
     <key>forgerock_timeout</key>
     <string>60</string>
     <key>forgerock_auth_service_name</key>
     <string>sdkUsernamePasswordJourney</string>
     <key>forgerock_registration_service_name</key>
     <string>Registration</string>
  </dict>
</plist>
```

4. Save your changes.

With the sample configured, you can proceed to Step 3. Test the app.

Step 3. Test the app

Prepare › Download › Configure › **Run**

In the following procedure, you run the sample app that you configured in the previous step. The app steps through the login journey, rendering a UI to collect the required data for each node, for example a username and password node, together inside a page node.

1. In Xcode, select **Product > Run**.

   Xcode launches the sample app in the iPhone simulator.

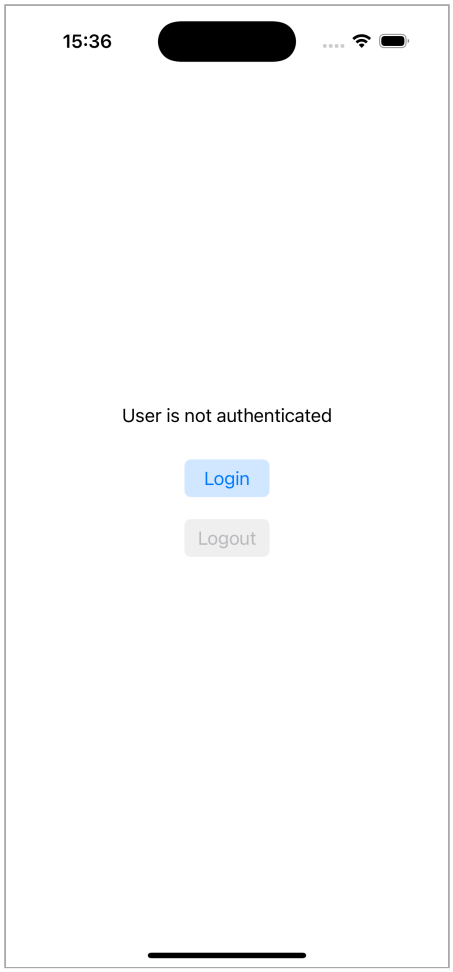


Figure 1. Starting the quickstart app in an iOS simulator

2. In the sample app on the iPhone simulator, tap the **Login** button.

   The app displays fields to input the user's credentials:

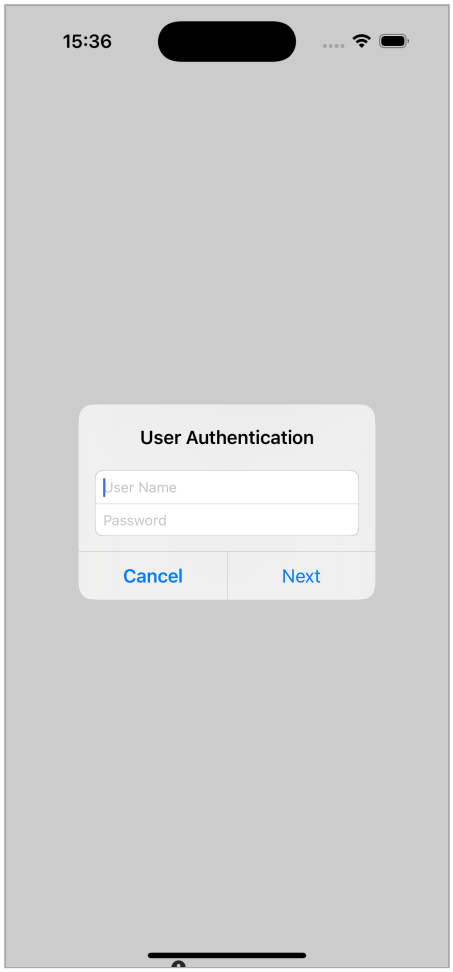


Figure 2. Login as your demo user

3. Sign on using the credentials of your demo user, and then click **Next**. For example:
    - **User Name**: `demo`
    - **Password**: `Ch4ng3it!`

    If authentication is successful the app displays a message that the user is authenticated and enables the **Logout** button:

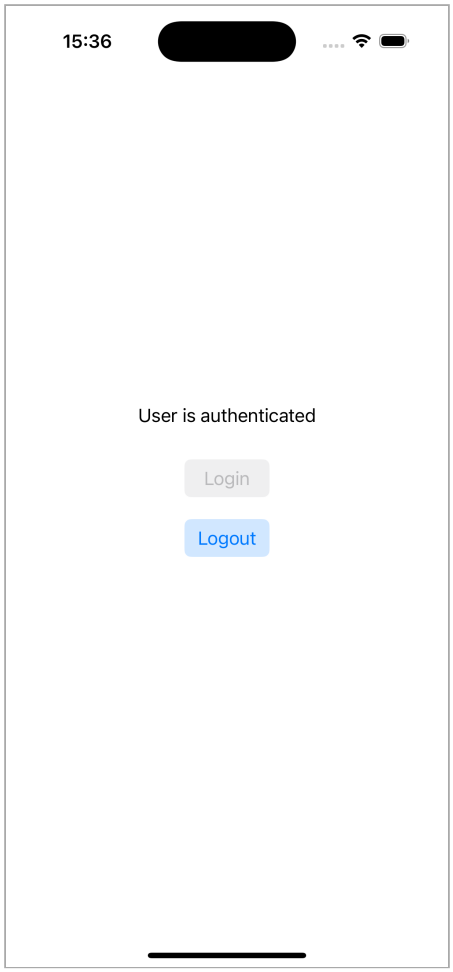


Figure 3. Login as your demo user

The console in Xcode outputs the access token, as well as a message that the user is authenticated:

```
[FRCore][4.8.0] [🌐 - Network] Response | [☑ 200] :
https://openam-forgerock-sdks.forgeblocks.com/am/oauth2/realms/alpha/access_token in 77 ms
        Response Header: [
    AnyHashable("x-forgerock-transactionid"): 670c-c3e9,
    AnyHashable("Content-Type"): application/json;charset=UTF-8,
    AnyHashable("Date"): Tue, 12 Nov 2024 15:36:27 GMT,
          Response Data: {
        "access_token":"eyJ0…Pc8k",
        "refresh_token":"eyJ0…dnA4",
        "scope":"address openid profile email",
        "id_token":"eyJ0…czKQ",
        "token_type":"Bearer",
        "expires_in":3598
    }
  ]
User is authenticated
```

4. Tap **Logout** to revoke the tokens, end the session, and return to the initial screen.

   The console in Xcode outputs the calls to the `/sessions` and `/revoke` endpoints:

```
Logout button is pressed

[FRCore][4.8.0] [🌐 - Network] Request | [POST]
https://openam-forgerock-sdks.forgeblocks.com/am/json/realms/alpha/sessions
    Additional Headers: [
        "accept-api-version": "resource=3.1",
        "8a92ca506c38f08": "qc7z…MQ.."]
        URL Parameters: ["_action": "logout"]
        Body Parameters: ["tokenId": "qc7z…MQ.."]

[FRCore][4.8.0] [🌐 - Network] Request | [POST]
https://openam-forgerock-sdks.forgeblocks.com/am/oauth2/realms/alpha/token/revoke
    Body Parameters: [
        "token": "eyJ0…dnA4",
        "client_id": "sdkPublicClient"]

[FRCore][4.8.0] [🌐 - Network] Response | [☑ 200] :
https://openam-forgerock-sdks.forgeblocks.com/am/json/realms/alpha/sessions?_action=logout in 117 ms
    Response Data: {
        "result":"Successfully logged out"}

[FRCore][4.8.0] [🌐 - Network] Response | [☑ 200] :
https://openam-forgerock-sdks.forgeblocks.com/am/oauth2/realms/alpha/token/revoke in 113 ms
        Response Data: {}
```

Authentication journey tutorial for JavaScript

Prepare › Download › Install › Configure › Run

In this tutorial you update a sample app to step through an authentication journey, meaning you get to design and implement the user interface to your requirements.

The sample navigates through a simple authentication journey, and obtains OAuth 2.0 tokens for the user.

Before you begin

Before you begin this tutorial ensure you have set up your PingOne Advanced Identity Cloud tenant or PingAM server with the required configuration.

For example, you will need to configure CORS, have an OAuth 2.0 client application set up, as well as an authentication journey for the app to navigate.

Complete prerequisites »

Step 1. Download the samples

To start this tutorial, you need to download the SDK sample apps repo, which contains the projects you will use.

Start step 1 »

Step 2. Install the dependencies

The sample projects need a number of dependencies that you can install by using the `npm` command.

For example, the Ping SDK for JavaScript itself is one of the dependencies.

Start step 2 »

Step 3. Configure connection properties

In this step, you configure the "embedded-login" sample app to connect to the OAuth 2.0 application you created in PingOne Advanced Identity Cloud or PingAM.

Start step 3 »

Step 4. Test the app

The final step is to run the sample app. The sample connects to your server and walks through your authentication journey or tree.

After successful authentication, the sample obtains an OAuth 2.0 access token and displays the related user information.

Test app »

Before you begin

Prepare › Download › Install › Configure › Run

To successfully complete this tutorial refer to the prerequisites in this section.

The tutorial also requires a configured server.

Prerequisites

Node and NPM

The SDK requires a minimum Node.js version of `18`, and is tested on versions `18` and `20`. To get a supported version of Node.js, refer to the Node.js download page.

You will also need `npm` to build the code and run the samples.

Server configuration

This tutorial requires you to configure one of the following servers:

PingOne Advanced Identity Cloud

PingAM

PingOne Advanced Identity Cloud

Cross-origin resource sharing (CORS) lets user agents make cross-domain server requests. In PingOne Advanced Identity Cloud, you can configure CORS to allow browsers from trusted domains to access PingOne Advanced Identity Cloud protected resources. For example, you might want a custom web application running on your own domain to get an end-user's profile information using the PingOne Advanced Identity Cloud REST API.

The Ping SDK for JavaScript samples and tutorials use `https://localhost:8443` as the host domain, which you should add to your CORS configuration.

If you are using a different domain for hosting SDK applications, ensure you add them to the CORS configuration as accepted origin domains.

To update the CORS configuration in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. At the top right of the screen, click your name, and then select **Tenant settings**.
3. On the **Global Settings** tab, click **Cross-Origin Resource Sharing (CORS)**.
4. Perform one of the following actions:
    - If available, click **ForgeRockSDK**.
    - If you haven't added any CORS configurations to the tenant, click **+ Add a CORS Configuration**, select **Ping SDK**, and then click **Next**.
5. Add `https://localhost:8443` and any DNS aliases you use to host your Ping SDK for JavaScript applications to the **Accepted Origins** property.
6. Complete the remaining fields to suit your environment.

This documentation assumes the following configuration, required for the tutorials and sample applications:

Property	Values
`Accepted Origins`	`https://localhost:8443`
`Accepted Methods`	`GET` `POST`

`Accepted Headers`	`accept-api-version`
	`x-requested-with`
	`content-type`
	`authorization`
	`if-match`
	`x-requested-platform`
	`iPlanetDirectoryPro` [1]
	`ch15fefc5407912` [2]

`Exposed Headers`	`authorization` `content-type`
`Enable Caching`	`True`
`Max Age`	`600`
`Allow Credentials`	`True`

> **Tip**
>
> Click **Show advanced settings** to be able to edit all available fields.

7. Click **Save CORS Configuration**.

The samples and tutorials in this documentation often require that you have an identity set up so that you can test authentication.

To create a demo user in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. In the left panel, click **Identities** > **Manage**.
3. Click **+ New Alpha realm - User**.
4. Enter the following details:
    - **Username** = `demo`
    - **First Name** = `Demo`
    - **Last Name** = `User`
    - **Email Address** = `demo.user@example.com`

- **Password** = `Ch4ng3it!`

5. Click **Save**.

Authentication journeys provide fine-grained authentication by allowing multiple paths and decision points throughout the flow. Authentication journeys are made up of nodes that define actions taken during authentication.

Each node performs a single task, such as collecting a username or making a simple decision. Nodes can have multiple outcomes rather than just success or failure. For details, see the Authentication nodes configuration reference in the PingAM documentation.

To create a simple journey for use when testing the Ping SDKs, follow these steps:

1. In your PingOne Advanced Identity Cloud tenant, navigate to **Journeys**, and click **+ New Journey**.
2. Enter a name, such as `sdkUsernamePasswordJourney` and click **Save**.

   The authentication journey designer appears.
3. Drag the following nodes into the designer area:
   - **Page Node**
   - **Platform Username**
   - **Platform Password**
   - **Data Store Decision**
4. Drag and drop the **Platform Username** and **Platform Password** nodes onto the **Page Node**, so that they both appear on the same page when logging in.
5. Connect the nodes as follows:

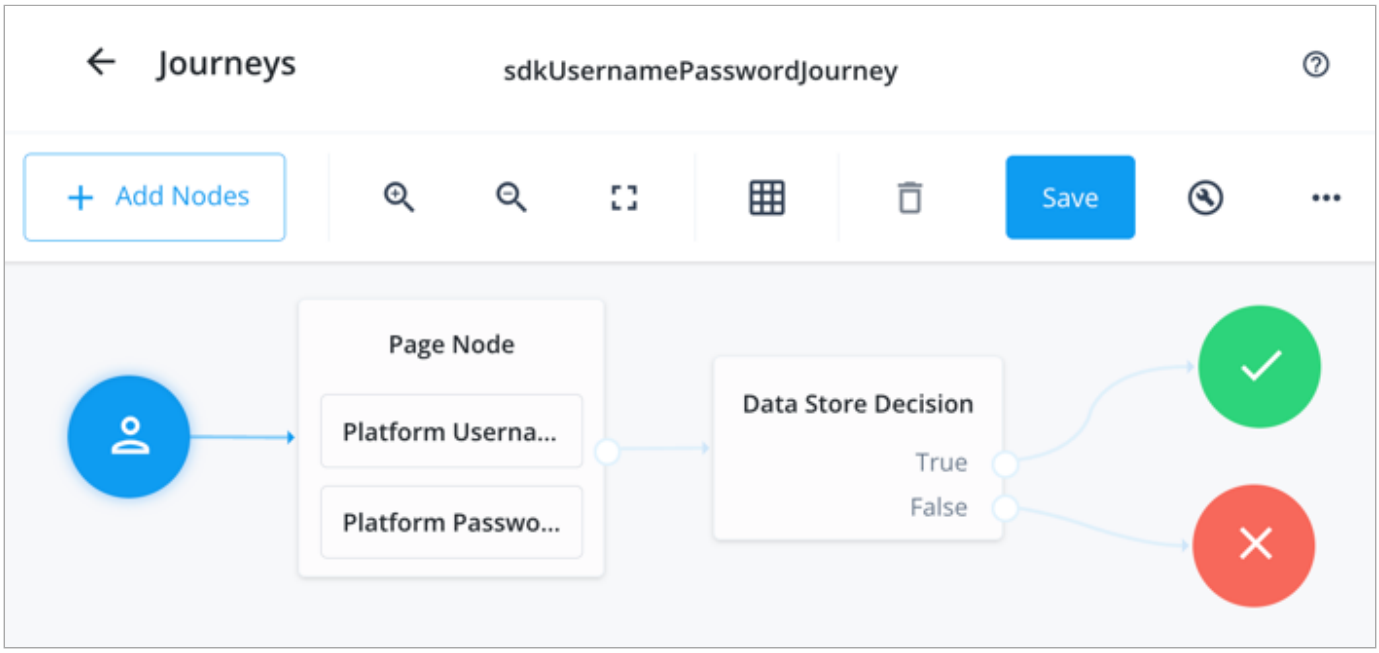


Figure 1. Example username and password authentication journey

6. Click **Save**.

Public clients do not use a client secret to obtain tokens because they are unable to keep them hidden. The Ping SDKs commonly use this type of client to obtain tokens, as they cannot guarantee safekeeping of the client credentials in a browser or on a mobile device.

To register a *public* OAuth 2.0 client application for use with the SDKs in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. In the left panel, click **Applications**.
3. Click **+ Custom Application**.
4. Select **OIDC - OpenId Connect** as the sign-in method, and then click **Next**.
5. Select **Native / SPA** as the application type, and then click **Next**.
6. In **Name**, enter a name for the application, such as `Public SDK Client`.
7. In **Owners**, select a user that is responsible for maintaining the application, and then click **Next**.

   > **Tip**
   >
   > When trying out the SDKs, you could select the `demo` user you created previously.

8. In **Client ID**, enter `sdkPublicClient`, and then click **Create Application**.

   PingOne Advanced Identity Cloud creates the application and displays the details screen.
9. On the **Sign On** tab:
   1. In **Sign-In URLs**, enter the following values:

      `https://localhost:8443/callback.html`

      > **Important**
      >
      > Also add any other domains where you host SDK applications.

   2. In **Grant Types**, enter the following values:

      `Authorization Code`

      `Refresh Token`

   3. In **Scopes**, enter the following values:

      `openid profile email address`
10. Click Show advanced settings, and on the **Authentication** tab:
    1. In **Token Endpoint Authentication Method**, select `none`.
    2. In **Client Type**, select `Public`.
    3. Enable the **Implied Consent** property.
11. Click **Save**.

The application is now configured to accept client connections from and issue OAuth 2.0 tokens to the example applications and tutorials covered by this documentation.

The provider specifies the supported OAuth 2.0 configuration options for a realm.

To ensure the PingOne Advanced Identity Cloud OAuth 2.0 provider service is configured for use with the Ping SDKs, follow these steps:

1. In your PingOne Advanced Identity Cloud tenant, navigate to **Native Consoles** > **Access Management**.
2. In the left panel, click **Services**.
3. In the list of services, click **OAuth2 Provider**.
4. On the **Core** tab, ensure **Issue Refresh Tokens** is enabled.
5. On the **Consent** tab, ensure **Allow Clients to Skip Consent** is enabled.
6. Click **Save Changes**.

PingAM

Cross-origin resource sharing (CORS) lets user agents make cross-domain server requests. In PingAM, you can configure CORS to allow browsers from trusted domains to access PingAM protected resources. For example, you might want a custom web application running on your own domain to get an end-user's profile information using the PingAM REST API.

The Ping SDK for JavaScript samples and tutorials all use `https://localhost:8443` as the host domain, which you should add to your CORS configuration.

If you are using a different URL for hosting SDK applications, ensure you add them to the CORS configuration as accepted origin domains.

To enable CORS in PingAM, and create a CORS filter to allow requests from your configured domain names, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. Navigate to **Configure** > **Global Services** > **CORS Service** > **Configuration**, and set the **Enable the CORS filter** property to `true`.

   > **Important**
   >
   > If this property is not enabled, CORS headers are not added to responses from PingAM, and CORS is disabled entirely.

3. On the **Secondary Configurations** tab, click **Click Add a Secondary Configuration**.
4. In the **Name** field, enter `ForgeRockSDK`.
5. in the **Accepted Origins** field, enter any DNS aliases you use for your SDK apps.

   This documentation assumes the following configuration:

Property	Values
`Accepted Origins`	`https://localhost:8443`
`Accepted Methods`	`GET` `POST`

`Accepted Headers`	`accept-api-version`
	`x-requested-with`
	`content-type`
	`authorization`
	`if-match`
	`x-requested-platform`
	`iPlanetDirectoryPro` [1]
	`ch15fefc5407912` [2]

`Exposed Headers`	`authorization` `content-type`

6. Click **Create**.

   PingAM displays the configuration of your new CORS filter.

7. On the CORS filter configuration page:

   1. Ensure **Enable the CORS filter** is enabled.
   2. Set the **Max Age** property to `600`
   3. Ensure **Allow Credentials** is enabled.

8. Click **Save Changes**.

The samples and tutorials in this documentation often require that you have an identity set up so that you can test authentication.

To create a demo user in PingAM, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. Navigate to **Identities**, and then click **+ Add Identity**.
3. Enter the following details:
   - **User ID** = `demo`
   - **Password** = `Ch4ng3it!`
   - **Email Address** = `demo.user@example.com`
4. Click **Create**.

Authentication trees provide fine-grained authentication by allowing multiple paths and decision points throughout the authentication flow. Authentication trees are made up of nodes that define actions taken during authentication.

Each node performs a single task, such as collecting a username or making a simple decision. Nodes can have multiple outcomes rather than just success or failure. For details, see the Authentication nodes configuration reference in the PingAM documentation.

To create a simple tree for use when testing the Ping SDKs, follow these steps:

1. Under **Realm Overview**, click **Authentication Trees**, then click **Create Tree**.
2. Enter a tree name, for example `sdkUsernamePasswordJourney`, and then click **Create**.

   The authentication tree designer appears, showing the **Start** entry point connected to the **Failure** exit point.
3. Drag the following nodes from the **Components** panel on the left side into the designer area:
   - **Page Node**
   - **Username Collector**
   - **Password Collector**
   - **Data Store Decision**
4. Drag and drop the **Username Collector** and **Password Collector** nodes onto the **Page Node**, so that they both appear on the same page when logging in.
5. Connect the nodes as follows:

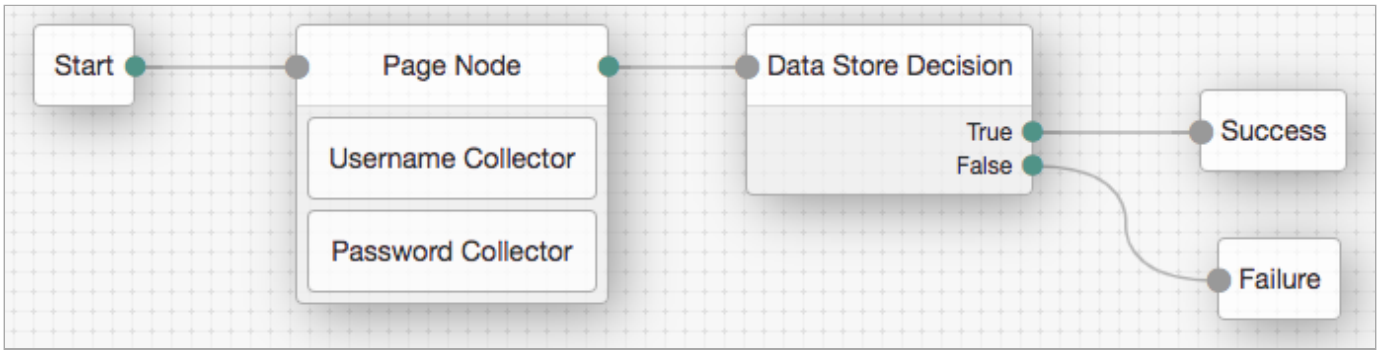


Figure 2. Example username and password authentication tree

6. Select the **Page Node**, and in the **Properties** pane, set the **Stage** property to `UsernamePassword`.

   > **Tip**
   >
   > You can configure the node properties by selecting a node and altering properties in the right-hand panel.

   One of the samples uses this specific value to determine the custom UI to display.
7. Click **Save**.

Public clients do not use a client secret to obtain tokens because they are unable to keep them hidden. The Ping SDKs commonly use this type of client to obtain tokens, as they cannot guarantee safekeeping of the client credentials in a browser or on a mobile device.

To register a *public* OAuth 2.0 client application for use with the SDKs in AM, follow these steps:

1. Log in to the PingAM admin UI as an administrator.

2. Navigate to **Applications** > **OAuth 2.0** > **Clients**, and then click **+ Add Client**.
3. In **Client ID**, enter `sdkPublicClient`.
4. Leave **Client secret** empty.
5. In **Redirection URIs**, enter the following values:

   `https://localhost:8443/callback.html`

   > **Tip**
   >
   > The Ping SDK for JavaScript attempts to load the redirect page to capture the OAuth 2.0 `code` and `state` query parameters that the server appended to the redirect URL.
   > If the page you redirect to does not exist, takes a long time to load, or runs any JavaScript you might get a timeout, delayed authentication, or unexpected errors.
   > To ensure the best user experience, we ***highly recommend*** that you redirect to a static HTML page with minimal HTML and no JavaScript when obtaining OAuth 2.0 tokens.

   > **Important**
   >
   > Also add any other domains where you will be hosting SDK applications.

6. In **Scopes**, enter the following values:

   `openid profile email address`

7. Click **Create**.

   PingAM creates the new OAuth 2.0 client, and displays the properties for further configuration.

8. On the **Core** tab:
   1. In **Client type**, select `Public`.
   2. Disable **Allow wildcard ports in redirect URIs**.
   3. Click **Save Changes**.
9. On the **Advanced** tab:
   1. In **Grant Types**, enter the following values:

      ```
      Authorization Code
      Refresh Token
      ```

   2. In **Token Endpoint Authentication Method**, select `None`.
   3. Enable the **Implied consent** property.
10. Click **Save Changes**.

The provider specifies the supported OAuth 2.0 configuration options for a realm.

To ensure the PingAM OAuth 2.0 provider service is configured for use with the Ping SDKs, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. In the left panel, click **Services**.
3. In the list of services, click **OAuth2 Provider**.
4. On the **Core** tab, ensure **Issue Refresh Tokens** is enabled.
5. On the **Consent** tab, ensure **Allow Clients to Skip Consent** is enabled.
6. Click **Save Changes**.

---

1. Cookie name value in PingAM servers.
2. In PingOne Advanced Identity Cloud tenants, go to **Tenant Settings > Global Settings > Cookie** to find this dynamic cookie name value.

Step 1. Download the samples

Prepare › **Download** › Install › Configure › Run

Tip

Check that you have completed the prerequisites before starting the tutorial.

To start this tutorial, you need to download the SDK sample apps repo, which contains the projects you will use.

1. In a web browser, navigate to the SDK Sample Apps repository.
2. Download the source code using one of the following methods:

Download a ZIP file

1. Click **Code**, and then click **Download ZIP**.
2. Extract the contents of the downloaded ZIP file to a suitable location.

Use a Git-compatible tool to clone the repo locally

1. Click **Code**, and then copy the HTTPS URL.
2. Use the URL to clone the repository to a suitable location.

   For example, from the command-line you could run:

```
git clone https://github.com/ForgeRock/sdk-sample-apps.git
```

The result of these steps is a local folder named `sdk-sample-apps`.

Step 2. Install the dependencies

Prepare › Download › **Install** › Configure › Run

In the following procedure, you install the required modules and dependencies, including the Ping SDK for JavaScript.

1. In a terminal window, navigate to the `sdk-sample-apps/javascript` folder.
2. To install the required packages, enter the following:

   ```
   npm install
   ```

   The `npm` tool downloads the required packages, and places them inside a `node_modules` folder.

Step 3. Configure connection properties

Prepare › Download › Install › **Configure** › Run

In this step, you configure the sample app to connect to the authentication tree/journey you created when setting up your server configuration.

1. Choose how you want to configure the sample app. You can either configure the sample by using dynamic configuration, or by create a `.env` file.

Dynamic configuration

1. Open the `/sdk-sample-apps/javascript/embedded-login/src/main.js` file.
2. Replace the call to `forgerock.Config.set()` with the following code:

```
await forgerock.Config.setAsync({
  serverConfig: {
    wellknown: '{WELL_KNOWN}'
  },
  clientId: '{WEB_OAUTH_CLIENT}',
  tree: '{TREE}',
  scope: '{SCOPE}',
  redirectUri: `${window.location.origin}/callback.html`
});
```

3. Replace the placeholder strings with the values you obtained when preparing your environment.

{WELL_KNOWN}

The `.well-known` endpoint of your server.

PingOne Advanced Identity Cloud example:

`https://openam-forgerock-sdks.forgeblocks.com/am/oauth2/realms/root/realms/alpha/.well-known/openid-configuration`

You can view the `.well-known` endpoint for an OAuth 2.0 client in the PingOne Advanced Identity Cloud admin console:

1. Log in to your PingOne Advanced Identity Cloud administration console.
2. Click **Applications**, and then select the OAuth 2.0 client you created earlier. For example, sdkPublicClient.
3. On the **Sign On** tab, in the **Client Credentials** section, copy the **Discovery URI** value.

PingAM example:

`https://openam.example.com:8443/openam/oauth2/realms/root/.well-known/openid-configuration`

{WEB_OAUTH_CLIENT}

The client ID from your OAuth 2.0 application.

For example, `sdkPublicClient`

{TREE}

The simple login journey or tree you created earlier.

For example `sdkUsernamePasswordJourney`.

{SCOPE}

The scopes you added to your OAuth 2.0 application.

For example, `address email openid phone profile`

The result resembles the following:

main.js

```
await Config.setAsync({
  serverConfig: {
    wellknown: 'https://openam-forgerock-sdks.forgeblocks.com/am/oauth2/realms/root/realms/
alpha/.well-known/openid-configuration'
  },
  clientId: 'sdkPublicClient',
  tree: 'sdkUsernamePasswordJourney',
  scope: 'openid profile email address',
  redirectUri: `${window.location.origin}/callback.html`
});
```

Create a .env file

1. Copy the `.env.example` file in the `/sdk-sample-apps/javascript/embedded-login` folder and save it with the name `.env` within this same directory.

   Your `.env` file has the following initial contents:

   Initial `.env` file

   ```
   SERVER_URL=$SERVER_URL
   REALM_PATH=$REALM_PATH
   SCOPE=$SCOPE
   TIMEOUT=$TIMEOUT
   TREE=$TREE
   WEB_OAUTH_CLIENT=$WEB_OAUTH_CLIENT
   ```

2. Replace the placeholder strings with the values you obtained when preparing your environment.

   $SERVER_URL

   The base URL of the server to connect to.

   Identity Cloud example:

   `https://openam-forgerock-sdks.forgeblocks.com/am`

   Self-hosted example:

   `https://openam.example.com:8443/openam`

   $REALM_PATH

   The realm in your server.

   Usually, `root` for AM and `alpha` or `bravo` for Advanced Identity Cloud.

   $SCOPE

   The scopes you added to your OAuth 2.0 application.

   For example, `address email openid phone profile`

   $TIMEOUT

   The simple login journey or tree you created earlier, for example `sdkUsernamePasswordJourney`.

   $TREE

   The simple login journey or tree you created earlier, for example `sdkUsernamePasswordJourney`.

   $WEB_OAUTH_CLIENT

   The simple login journey or tree you created earlier, for example `sdkUsernamePasswordJourney`.

Here's an example; your values may vary:

```
AM_URL=https://openam-forgerock-sdks.forgeblocks.com/am
REALM_PATH=alpha
SCOPE=openid profile email address
TIMEOUT=5000
TREE=sdkUsernamePasswordJourney
WEB_OAUTH_CLIENT=sdkPublicClient
```

Here are descriptions for some of the values:

TREE

The simple login journey or tree you created earlier, for example `sdkUsernamePasswordJourney`.

REALM_PATH

The realm of your server.

Usually, `root` for AM and `alpha` or `bravo` for Advanced Identity Cloud.

Step 4. Test the app

Prepare › Download › Install › Configure › **Run**

In the following procedure, you run the sample app that you configured in the previous step. The sample connects to your server and walks through the authentication journey you created in an earlier step.

After successful authentication, the sample obtains an OAuth 2.0 access token and displays the related user information.

1. In a terminal window, navigate to the `/javascript` folder in your `sdk-sample-apps` project.
2. To run the embedded login sample, enter the following:

```
npm run start:embedded-login
```

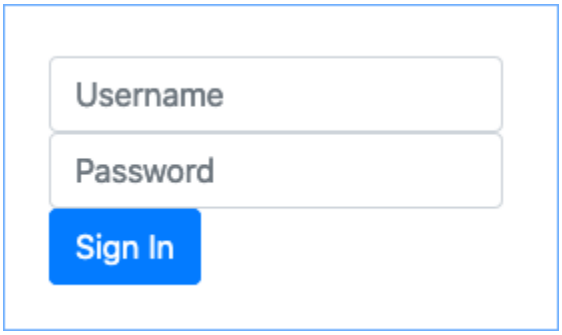
3. In a web browser:
    1. Ensure you are *NOT* currently logged into the server instance.

> **Note**
>
> If you are logged into the PingAM instance in the browser, the sample will not work. Logout of the PingAM instance *before* you run the sample.

2. Navigate to the following URL:

   `https://localhost:8443`

   A form appears with *"Username"* and *"Password"* fields, as defined by the page node in the `sdkUsernamePasswordJourney` you created in a previous step:



3. Authenticate as a non-administrative user, and click **Sign In**.

   Default login credentials:

   - "Username" - `demo`
   - "Password" - `Ch4ng3it!`

   If the app displays the user information, authentication was successful:

   > **Tip**
   >
   > To see the application calling the `authorize` and `authenticate` endpoints, open the **Network** tab of your browser's developer tools.

4. To revoke the OAuth 2.0 token, click the **Sign Out** button.

   The application calls the `endSession` endpoint to revoke the OAuth 2.0 token, and returns to the sign-in form.

Recap

Congratulations!

You have now used the Ping SDK for JavaScript to authenticate to your server instance.

You have seen how to obtain OAuth 2.0 tokens, view the related user information, and log a user out of the server.

More information

- API reference: `FRAuth`
- API reference: `TokenManager`
- API reference: `UserManager`

Ping SDKs platform integrations for auth journeys

Follow these tutorials to leverage the Ping SDKs in other platforms or languages, to support **Authentication journeys**, also known as ***Intelligent Authentication*** in your apps.

These tutorials support the following servers:

- PingOne Advanced Identity Cloud
- PingAM

Angular

Flutter (iOS)

ReactJS

React Native (iOS)

Authentication journey tutorial for Angular

In this tutorial you build out a sample Angular SPA and make use of a Node.js REST API server sample app.

This guide uses the Ping SDK for JavaScript to implement the following application features:

- Dynamic authentication form for login.
- OAuth/OIDC token acquisition through the Authorization Code Flow with PKCE.
- Protected client-side routing.
- Resource requests to a protected REST API.
- Log out - revoke tokens and end session.

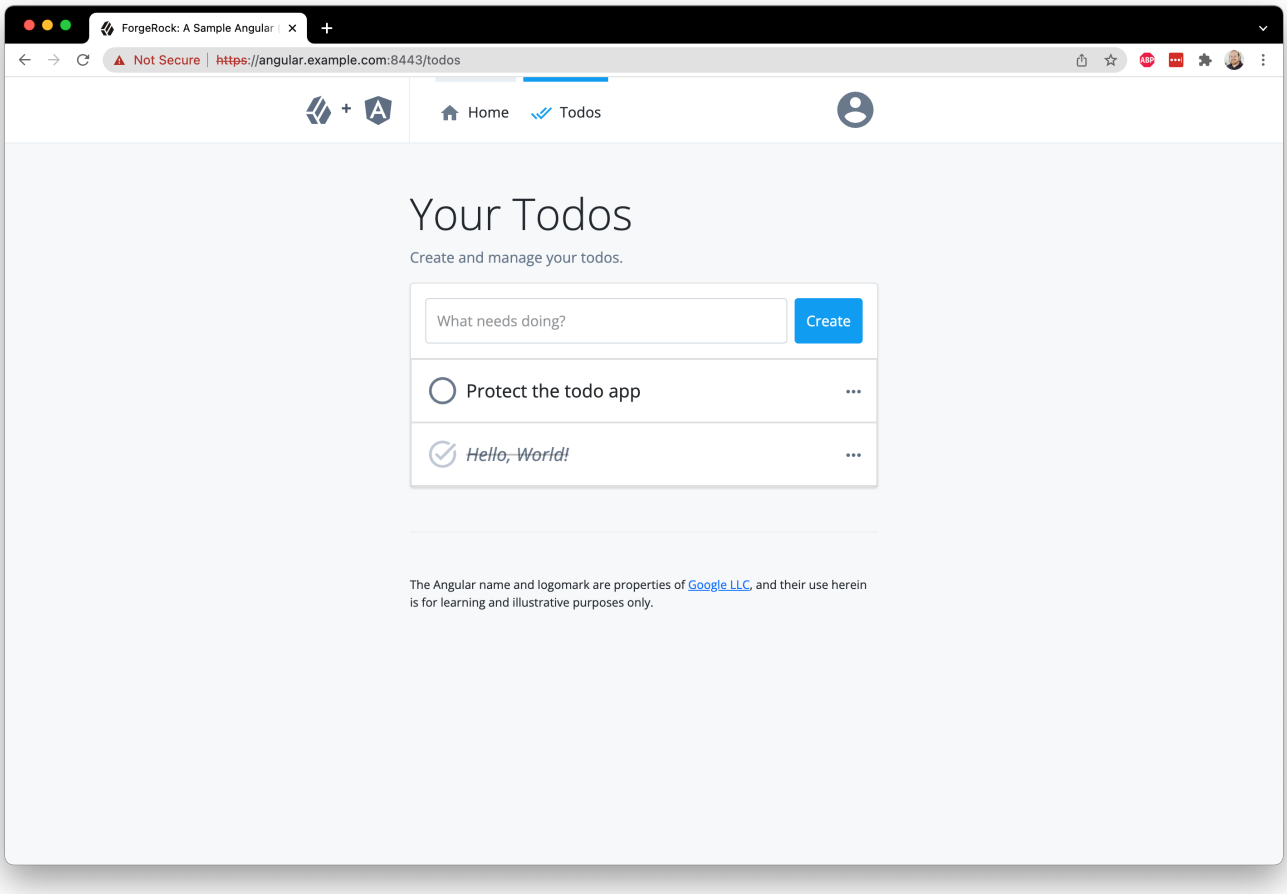


Figure 1. The Todo page of the sample app.

Before you begin

Before you begin this tutorial ensure you have set up your PingOne Advanced Identity Cloud tenant or PingAM server with the required configuration.

For example, you will need to configure CORS, have an OAuth 2.0 client application set up, as well as an authentication journey for the app to navigate.

Complete prerequisites »

Step 1. Download the samples

To start this tutorial, you need to download the SDK sample apps repo, which contains the projects you will use.

Start step 1 »

Step 2. Configure connection properties

Configure both the **Todo** client app, and the API backend server app to connect to the OAuth 2.0 application you created in PingOne Advanced Identity Cloud or PingAM.

Start step 2 »

Step 3. Build and run the projects

In this step you build and run the API backend server app, and then the **Todo** client app.

There are also troubleshooting tips if the apps do not start as expected.

Start step 3 »

Step 4. Implement the Ping SDK

In this final step you implement the Ping SDK into the **Todo** client app, so that it handles the responses from your PingOne Advanced Identity Cloud tenant or PingAM server, can get tokens and user information, and supports logging out.

Start step 4 »

Before you begin

To successfully complete this tutorial refer to the prerequisites in this section.

The tutorial also requires a configured server.

Prerequisites

Node and NPM

The SDK requires a minimum Node.js version of `18`, and is tested on versions `18` and `20`. To get a supported version of Node.js, refer to the Node.js download page.

You will also need `npm` version 7 or newer to build the code and run the samples.

Server configuration

This tutorial requires you to configure one of the following servers:

PingOne Advanced Identity Cloud

PingAM

PingOne Advanced Identity Cloud

Cross-origin resource sharing (CORS) lets user agents make cross-domain server requests. In PingOne Advanced Identity Cloud, you can configure CORS to allow browsers from trusted domains to access PingOne Advanced Identity Cloud protected resources. For example, you might want a custom web application running on your own domain to get an end-user's profile information using the PingOne Advanced Identity Cloud REST API.

The Ping SDK for JavaScript samples and tutorials use `https://localhost:8443` as the host domain, which you should add to your CORS configuration.

If you are using a different domain for hosting SDK applications, ensure you add them to the CORS configuration as accepted origin domains.

For example, for this tutorial you should also add the host domain used by the todo API backend server, which defaults to `http://localhost:9443`.

To update the CORS configuration in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. At the top right of the screen, click your name, and then select **Tenant settings**.
3. On the **Global Settings** tab, click **Cross-Origin Resource Sharing (CORS)**.

4. Perform one of the following actions:
   - If available, click **ForgeRockSDK**.
   - If you haven't added any CORS configurations to the tenant, click **+ Add a CORS Configuration**, select **Ping SDK**, and then click **Next**.
5. Add `https://localhost:8443` and any DNS aliases you use to host your Ping SDK for JavaScript applications to the **Accepted Origins** property.
6. Add the URL used by the todo API backend server, which defaults to `http://localhost:9443`.
7. Complete the remaining fields to suit your environment.

   This documentation assumes the following configuration, required for the tutorials and sample applications:

Property	Values
`Accepted Origins`	`https://localhost:8443` `http://localhost:9443`
`Accepted Methods`	`GET` `POST`

`Accepted Headers`	`accept-api-version`
	`x-requested-with`
	`content-type`
	`authorization`
	`if-match`
	`x-requested-platform`
	`iPlanetDirectoryPro` [1]
	`ch15fefc5407912` [2]

`Exposed Headers`	`authorization` `content-type`
`Enable Caching`	`True`
`Max Age`	`600`
`Allow Credentials`	`True`

> **Tip**
>
> Click **Show advanced settings** to be able to edit all available fields.

8. Click **Save CORS Configuration**.

The samples and tutorials in this documentation often require that you have an identity set up so that you can test authentication.

To create a demo user in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. In the left panel, click **Identities** > **Manage**.
3. Click **+ New Alpha realm - User**.
4. Enter the following details:
    - **Username** = `demo`
    - **First Name** = `Demo`
    - **Last Name** = `User`
    - **Email Address** = `demo.user@example.com`
    - **Password** = `Ch4ng3it!`
5. Click **Save**.

Authentication journeys provide fine-grained authentication by allowing multiple paths and decision points throughout the flow. Authentication journeys are made up of nodes that define actions taken during authentication.

Each node performs a single task, such as collecting a username or making a simple decision. Nodes can have multiple outcomes rather than just success or failure. For details, see the Authentication nodes configuration reference in the PingAM documentation.

To create a simple journey for use when testing the Ping SDKs, follow these steps:

1. In your PingOne Advanced Identity Cloud tenant, navigate to **Journeys**, and click **+ New Journey**.
2. Enter a name, such as `sdkUsernamePasswordJourney` and click **Save**.

   The authentication journey designer appears.
3. Drag the following nodes into the designer area:
    - **Page Node**
    - **Platform Username**
    - **Platform Password**
    - **Data Store Decision**
4. Drag and drop the **Platform Username** and **Platform Password** nodes onto the **Page Node**, so that they both appear on the same page when logging in.
5. Connect the nodes as follows:

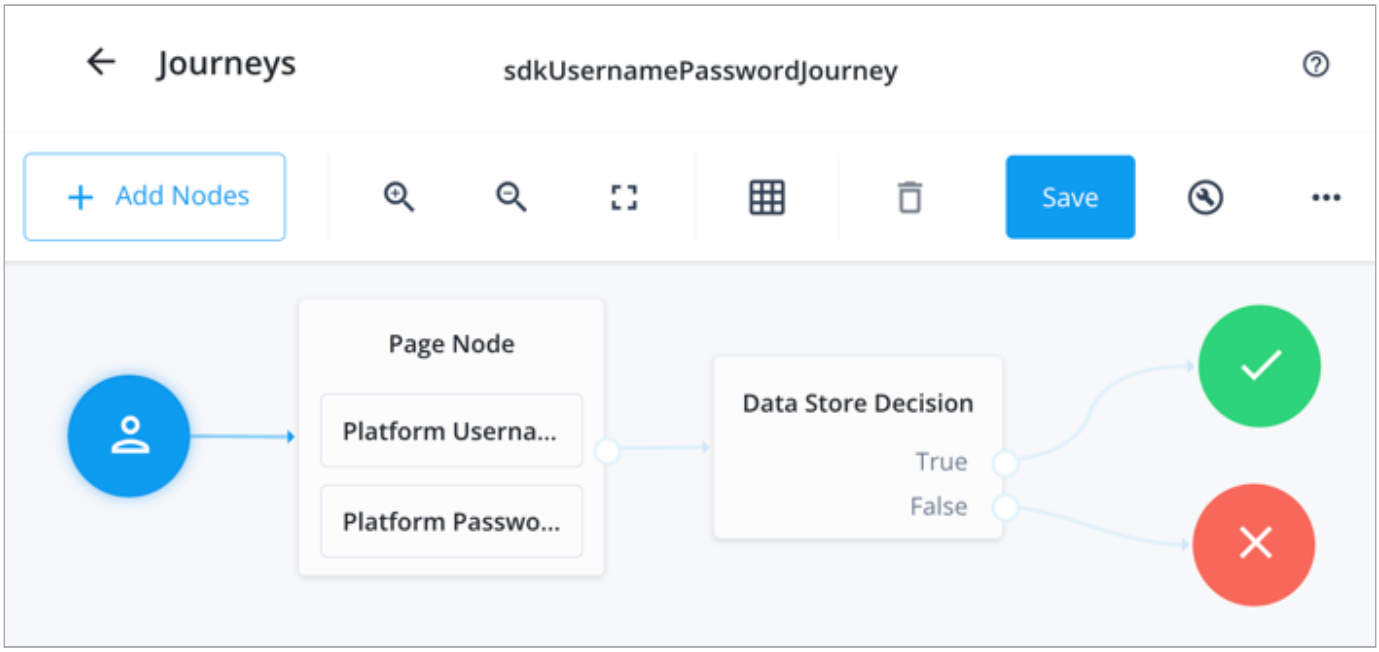


Figure 1. Example username and password authentication journey

6. Click **Save**.

Public clients do not use a client secret to obtain tokens because they are unable to keep them hidden. The Ping SDKs commonly use this type of client to obtain tokens, as they cannot guarantee safekeeping of the client credentials in a browser or on a mobile device.

To register a *public* OAuth 2.0 client application for use with the SDKs in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. In the left panel, click **Applications**.
3. Click **+ Custom Application**.
4. Select **OIDC - OpenId Connect** as the sign-in method, and then click **Next**.
5. Select **Native / SPA** as the application type, and then click **Next**.
6. In **Name**, enter a name for the application, such as `Public SDK Client`.
7. In **Owners**, select a user that is responsible for maintaining the application, and then click **Next**.

> **Tip**
>
> When trying out the SDKs, you could select the `demo` user you created previously.

8. In **Client ID**, enter `sdkPublicClient`, and then click **Create Application**.

   PingOne Advanced Identity Cloud creates the application and displays the details screen.

9. On the **Sign On** tab:

   1. In **Sign-In URLs**, enter the following values:

      `https://localhost:8443/callback`

> **Important**
>
> Also add any other domains where you host SDK applications.

2. In **Grant Types**, enter the following values:

   `Authorization Code`

   `Refresh Token`

3. In **Scopes**, enter the following values:

   `openid profile email address`

10. Click Show advanced settings, and on the **Authentication** tab:

    1. In **Token Endpoint Authentication Method**, select `none`.
    2. In **Client Type**, select `Public`.
    3. Enable the **Implied Consent** property.

11. Click **Save**.

The application is now configured to accept client connections from and issue OAuth 2.0 tokens to the example applications and tutorials covered by this documentation.

Confidential clients are able to securely store credentials and are commonly used for server-to-server communication. For example, the "Todo" API backend provided with the SDK samples uses a confidential client to obtain tokens.

The following tutorials and integrations require a *confidential* client:

- Authentication journey tutorial for Angular
- Authentication journey tutorial for ReactJS
- Build advanced token security in a JavaScript SPA

To register a *confidential* OAuth 2.0 client application for use with the SDKs in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. In the left panel, click **Applications**.
3. Click **+ Custom Application**.
4. Select **OIDC - OpenId Connect** as the sign-in method, and then click **Next**.
5. Select **Web** as the application type, and then click **Next**.
6. In **Name**, enter a name for the application, such as `Confidential SDK Client`.
7. In **Owners**, select a user responsible for maintaining the application, and then click **Next**.

> **Tip**
>
> When trying out the SDKs, you could select the `demo` user you created previously.

8. On the **Web Settings** page:
    1. In **Client ID**, enter `sdkConfidentialClient`
    2. In **Client Secret**, enter a strong password and make a note of it for later use.

    > **Important**
    > The client secret is not available to view after this step.
    > If you forget it, you must reset the secret and reconfigure any connected clients.

    3. Click **Create Application**.

    PingOne Advanced Identity Cloud creates the application and displays the details screen.
9. On the **Sign On** tab, click **Show advanced settings**, and on the **Access** tab:
    1. In **Default Scopes**, enter `am-introspect-all-tokens`.
10. Click **Save**.

The provider specifies the supported OAuth 2.0 configuration options for a realm.

To ensure the PingOne Advanced Identity Cloud OAuth 2.0 provider service is configured for use with the Ping SDKs, follow these steps:

1. In your PingOne Advanced Identity Cloud tenant, navigate to **Native Consoles** > **Access Management**.
2. In the left panel, click **Services**.
3. In the list of services, click **OAuth2 Provider**.
4. On the **Core** tab, ensure **Issue Refresh Tokens** is enabled.
5. On the **Consent** tab, ensure **Allow Clients to Skip Consent** is enabled.
6. Click **Save Changes**.

PingAM

Cross-origin resource sharing (CORS) lets user agents make cross-domain server requests. In PingAM, you can configure CORS to allow browsers from trusted domains to access PingAM protected resources. For example, you might want a custom web application running on your own domain to get an end-user's profile information using the PingAM REST API.

The Ping SDK for JavaScript samples and tutorials all use `https://localhost:8443` as the host domain, which you should add to your CORS configuration.

If you are using a different URL for hosting SDK applications, ensure you add them to the CORS configuration as accepted origin domains.

For example, for this tutorial you should also add the host domain used by the todo API backend server, which defaults to `http://localhost:9443`.

To enable CORS in PingAM, and create a CORS filter to allow requests from your configured domain names, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. Navigate to **Configure** > **Global Services** > **CORS Service** > **Configuration**, and set the **Enable the CORS filter** property to `true`.

   > **Important**
   >
   > If this property is not enabled, CORS headers are not added to responses from PingAM, and CORS is disabled entirely.

3. On the **Secondary Configurations** tab, click **Click Add a Secondary Configuration**.
4. In the **Name** field, enter `ForgeRockSDK`.
5. in the **Accepted Origins** field, enter any DNS aliases you use for your SDK apps.

   This documentation assumes the following configuration:

Property	Values
`Accepted Origins`	`https://localhost:8443` `http://localhost:9443`
`Accepted Methods`	`GET` `POST`

`Accepted Headers`	`accept-api-version`
	`x-requested-with`
	`content-type`
	`authorization`
	`if-match`
	`x-requested-platform`
	`iPlanetDirectoryPro` [1]
	`ch15fefc5407912` [2]

`Exposed Headers`	`authorization` `content-type`

6. Click **Create**.

   PingAM displays the configuration of your new CORS filter.

7. On the CORS filter configuration page:
   1. Ensure **Enable the CORS filter** is enabled.
   2. Set the **Max Age** property to `600`
   3. Ensure **Allow Credentials** is enabled.
8. Click **Save Changes**.

The samples and tutorials in this documentation often require that you have an identity set up so that you can test authentication.

To create a demo user in PingAM, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. Navigate to **Identities**, and then click **+ Add Identity**.
3. Enter the following details:
    - **User ID** = `demo`
    - **Password** = `Ch4ng3it!`
    - **Email Address** = `demo.user@example.com`
4. Click **Create**.

Authentication trees provide fine-grained authentication by allowing multiple paths and decision points throughout the authentication flow. Authentication trees are made up of nodes that define actions taken during authentication.

Each node performs a single task, such as collecting a username or making a simple decision. Nodes can have multiple outcomes rather than just success or failure. For details, see the Authentication nodes configuration reference in the PingAM documentation.

To create a simple tree for use when testing the Ping SDKs, follow these steps:

1. Under **Realm Overview**, click **Authentication Trees**, then click **Create Tree**.
2. Enter a tree name, for example `sdkUsernamePasswordJourney`, and then click **Create**.

   The authentication tree designer appears, showing the **Start** entry point connected to the **Failure** exit point.
3. Drag the following nodes from the **Components** panel on the left side into the designer area:
    - **Page Node**
    - **Username Collector**
    - **Password Collector**
    - **Data Store Decision**
4. Drag and drop the **Username Collector** and **Password Collector** nodes onto the **Page Node**, so that they both appear on the same page when logging in.
5. Connect the nodes as follows:

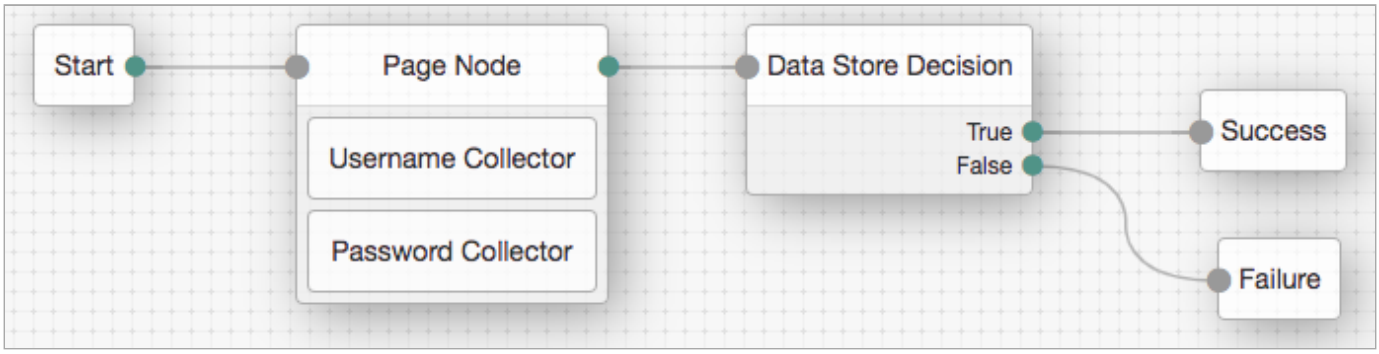


Figure 2. Example username and password authentication tree

6. Select the **Page Node**, and in the **Properties** pane, set the **Stage** property to `UsernamePassword`.

   > **Tip**
   >
   > You can configure the node properties by selecting a node and altering properties in the right-hand panel.

   One of the samples uses this specific value to determine the custom UI to display.

7. Click **Save**.

Public clients do not use a client secret to obtain tokens because they are unable to keep them hidden. The Ping SDKs commonly use this type of client to obtain tokens, as they cannot guarantee safekeeping of the client credentials in a browser or on a mobile device.

To register a *public* OAuth 2.0 client application for use with the SDKs in AM, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. Navigate to **Applications** > **OAuth 2.0** > **Clients**, and then click **+ Add Client**.
3. In **Client ID**, enter `sdkPublicClient`.
4. Leave **Client secret** empty.
5. In **Redirection URIs**, enter the following values:

   `https://localhost:8443/callback`

   > **Important**
   >
   > Also add any other domains where you will be hosting SDK applications.

6. In **Scopes**, enter the following values:

   `openid profile email address`

7. Click **Create**.

   PingAM creates the new OAuth 2.0 client, and displays the properties for further configuration.

8. On the **Core** tab:

   1. In **Client type**, select `Public`.

2. Disable **Allow wildcard ports in redirect URIs**.
3. Click **Save Changes**.

9. On the **Advanced** tab:
    1. In **Grant Types**, enter the following values:

    ```
    Authorization Code
    Refresh Token
    ```

    2. In **Token Endpoint Authentication Method**, select `None`.
    3. Enable the **Implied consent** property.
10. Click **Save Changes**.

The provider specifies the supported OAuth 2.0 configuration options for a realm.

To ensure the PingAM OAuth 2.0 provider service is configured for use with the Ping SDKs, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. In the left panel, click **Services**.
3. In the list of services, click **OAuth2 Provider**.
4. On the **Core** tab, ensure **Issue Refresh Tokens** is enabled.
5. On the **Consent** tab, ensure **Allow Clients to Skip Consent** is enabled.
6. Click **Save Changes**.

---

1. Cookie name value in PingAM servers.
2. In PingOne Advanced Identity Cloud tenants, go to **Tenant Settings > Global Settings > Cookie** to find this dynamic cookie name value.

Step 1. Download the samples

To start this tutorial, you need to download the SDK sample apps repo, which contains the projects you will use.

Within the repo are two branches related to this tutorial:

`build-protected-app/start`

Contains all the source files you need to follow this tutorial, but without the actual implementation of the Ping SDK functionality.

Use this branch if you want to complete the tutorial step-by-step, adding the code the tutorial provides.

`build-protected-app/complete`

The same source files but with the Ping SDK code already implemented.

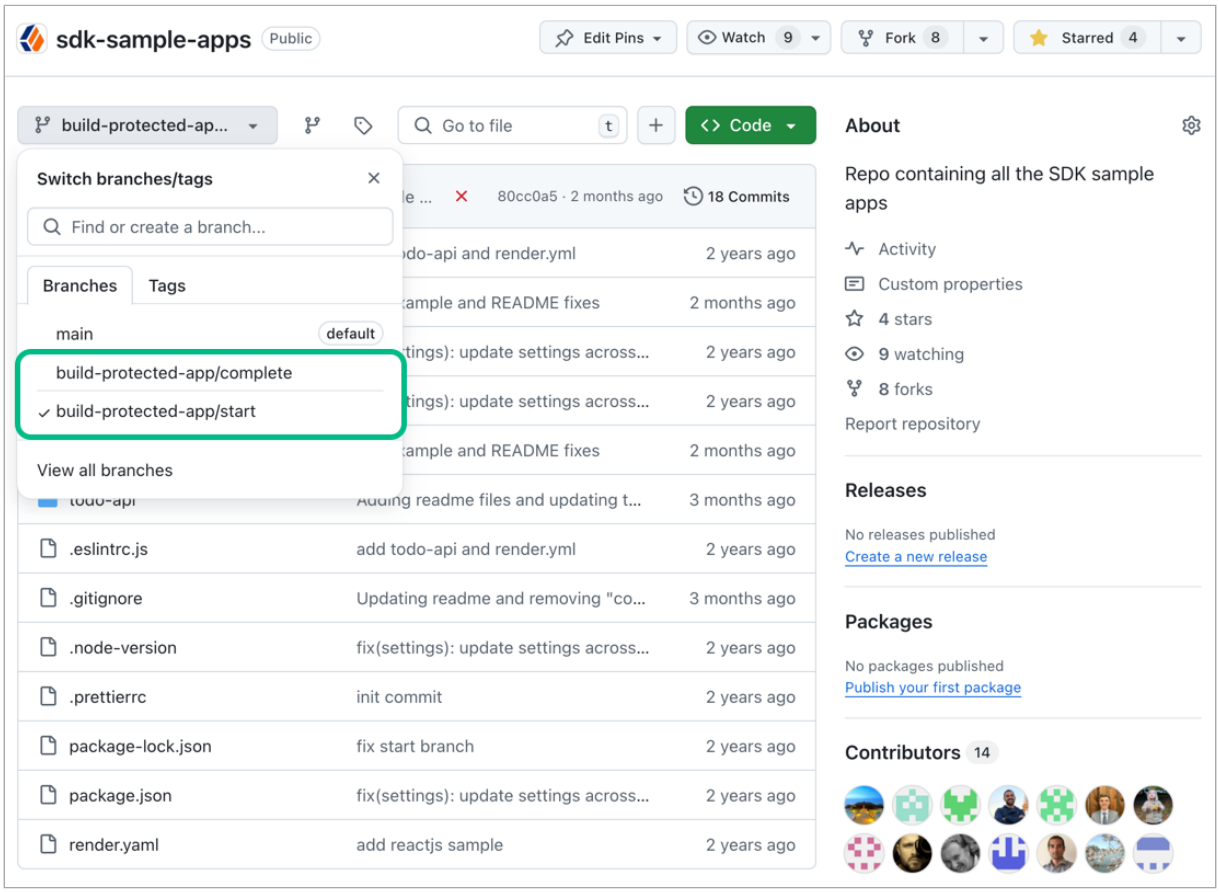
Use this branch if you want to skip ahead of the tutorial, or if you want to compare your work with the completed version for troubleshooting.

To get a copy of the tutorial source code:

1. In a web browser, navigate to the SDK Sample Apps repository.
2. Download the source code using one of the following methods:

Download a ZIP file

1. Select which branch to download:



2. Click **Code**, and then click **Download ZIP**.
3. Extract the contents of the downloaded ZIP file to a suitable location.

Use a Git-compatible tool to clone the repo locally

1. Click **Code**, and then copy the HTTPS URL.
2. Use the URL to clone the repository to a suitable location.

   For example, from the command-line you could run:

```
git clone https://github.com/ForgeRock/sdk-sample-apps.git
```

3. Checkout which branch you want to work on.

   For example, from the command-line you could run:

   ```
   git checkout build-protected-app/start
   ```

The result of these steps is a local folder named `sdk-sample-apps`.

Step 2. Configure connection properties

There are two projects in this tutorial that require configuration:

Client Angular app

The front-end client app, written in Angular, that handles the UI and authentication journeys.

Backend API server

A backend REST API server that uses a confidential OAuth 2.0 client to contact the authorization server. The API server handles storage and retrieval of your personal "Todo" items.

Configure the Angular client app

Copy the `.env.example` file in the `sdk-sample-apps/angular-todo` folder and save it with the name `.env` within this same directory.

Add your relevant values to this new file because it provides all the important configuration settings to your applications.

Example client `.env` file

```
AM_URL=https://openam-forgerock-sdks.forgeblocks.com/am
APP_URL=https://localhost:8443
API_URL=http://localhost:9443
DEBUGGER_OFF=true
JOURNEY_LOGIN=sdkUsernamePasswordJourney
JOURNEY_REGISTER=Registration
REALM_PATH=alpha
WEB_OAUTH_CLIENT=sdkPublicClient
PORT=9443
REST_OAUTH_CLIENT=sdkConfidentialClient
REST_OAUTH_SECRET=ch4ng3it!
```

Here are descriptions for some of the values:

- `DEBUGGER_OFF`: set to `true`, to disable `debug` statements in the app.

  These statements are for learning the integration points at runtime in your browser.

  When you open the browser's developer tools, the app pauses at each integration point. Code comments are placed above each one explaining their use.

- `JOURNEY_LOGIN`: The simple login journey or tree you created earlier, for example `sdkUsernamePasswordJourney`.
- `JOURNEY_REGISTER`: The registration journey or tree.

  You can use the default built-in `Registration` journey.
- `REALM_PATH`: The realm of your server.

  Usually, `root` for AM and `alpha` or `bravo` for Advanced Identity Cloud.

Configure the API server app

Copy the `.env.example` file in the `sdk-sample-apps/todo-api` folder and save it with the name `.env` within this same directory.

Add your relevant values to this new file as it will provide all the important configuration settings to your applications.

Example API server `.env` file

```
AM_URL=https://openam-forgerock-sdks.forgeblocks.com/am
DEVELOPMENT=true
PORT=9443
REALM_PATH=alpha
REST_OAUTH_CLIENT=sdkConfidentialClient
REST_OAUTH_SECRET=ch4ng3it!
```

Step 3. Build and run the projects

In this step you build and run the API backend, and the "Todo" client app project.

1. Open a terminal window at the root directory of the SDK samples repo, and then run the following command to start both the API backend server and the "Todo" client:

   ```
   npm run start:angular-todo
   ```

2. In a *different* browser than the one you are using to administer the server, visit the following URL: `https://localhost:8443`.

   The app renders a home page explaining the purpose of the project:

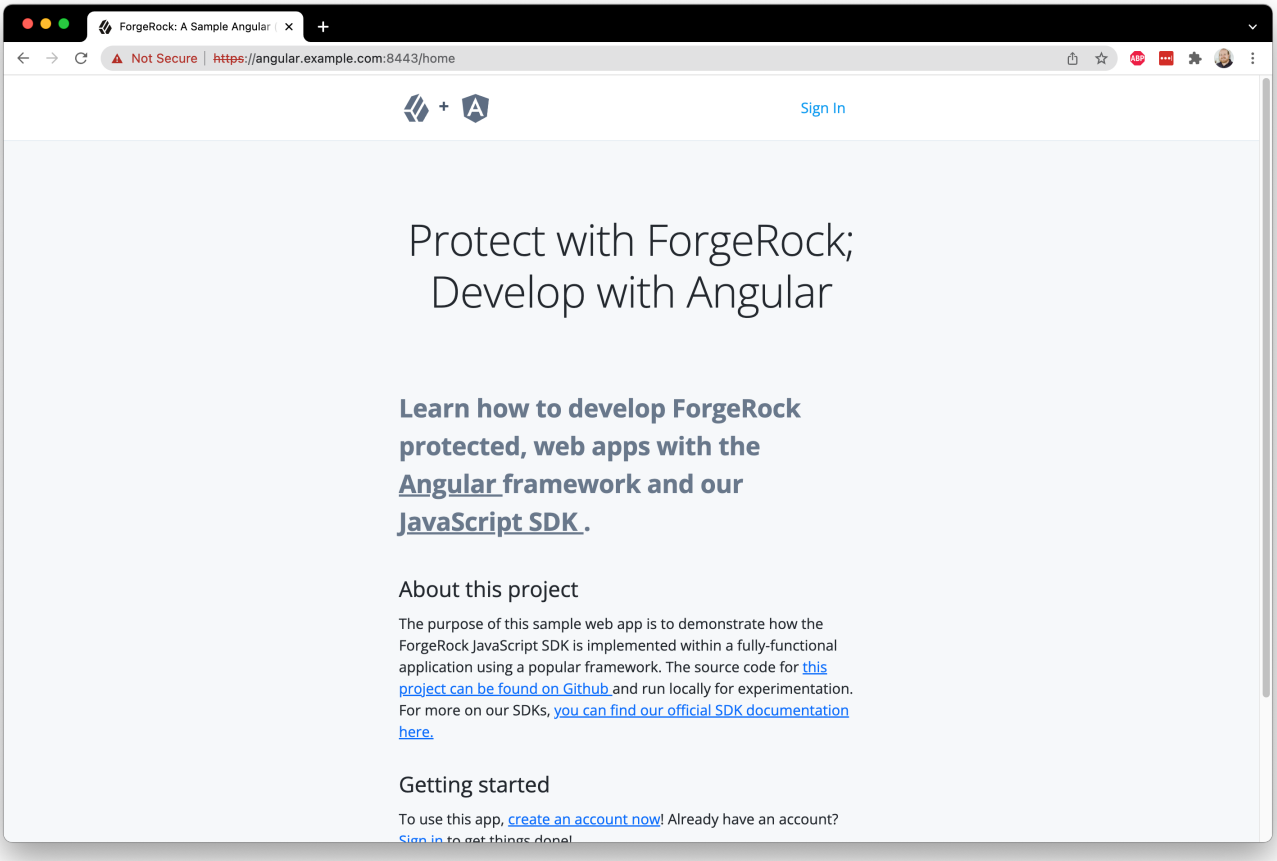


Figure 1. The home page of the sample app.

> **Note**
>
> Only the home page renders successfully. The login page functionality is not yet functional. You will develop this functionality later in this tutorial.

Troubleshooting

If the home page doesn't render due to errors, here are a few tips:

- Visit `http://localhost:9443/healthcheck` in the same browser you use for the Angular app; ensure it responds with "OK".
- Ensure your `hosts` file has the correct aliases.
- Look for error output in the terminal that is running the `start` command.
- Ensure you are not logged into the server within the same browser as the sample app; logout if you are and use a different browser.

Step 4. Implement the Ping SDK

Now that you have the environment and servers setup you can build the Ping SDK into the app to handle callbacks, display UI, and other tasks.

Set configuration from the ENV file

Within your IDE of choice, navigate to the `sdk-sample-apps/angular-todo` directory. This directory is where you will spend the rest of your time.

First, open up the `src/app/app.component.ts` file, import the `Config` object from the Ping SDK for JavaScript and call the `set` function on this object.

To import the `Config` object, modify the list of imports as follows:

```
  import { Component, OnInit } from '@angular/core';
  import { environment } from '../environments/environment';
  import { UserService } from './services/user.service';
+ import { Config, UserManager } from '@forgerock/javascript-sdk';

@@ collapsed @@
```

Now configure the SDK using the `set` function by adding the following code to the `ngOnInit` function:

```
@@ collapsed @@
  async ngOnInit(): Promise<void> {
+   Config.set({
+     clientId: environment.WEB_OAUTH_CLIENT,
+     redirectUri: environment.APP_URL,
+     scope: 'openid profile email address',
+     serverConfig: {
+     baseUrl: environment.AM_URL,
+     timeout: 30000, // 90000 or less
+   },
+   realmPath: environment.REALM_PATH,
+   tree: environment.JOURNEY_LOGIN,
+ });
@@ collapsed @@
```

The use of `set()` should always be the first SDK method called and is frequently done at the application's top-level file.

To configure the SDK to communicate with the journeys, OAuth clients, and realms of the appropriate server, pass a configuration object with the appropriate values.

The configuration object you are using in this instance pulls most of its values out of the `.env` variables you previously setup.

The variables map to constants within the `environment.ts` file generated when the project is built.

Go back to your browser and refresh the home page. There should be no change to what's rendered, and no errors in the console. Now that the app is configured to your server, let's wire up the simple login page!

Build the login page

Consider how the application renders the home page:

`HomeComponent` consists of `src/app/views/home/home.component.html` (HTML template with Angular directives), and `src/app/views/home/home.component.ts` (Angular component).

For the login page, the same pattern applies:

`LoginComponent` consists of `src/app/views/login/login.component.html` and `src/app/views/login/login.component.ts`. This is a simple view component, which includes `FormComponent` which actually invokes the SDK - more on that shortly.

Navigate to the app's login page within your browser. You should see a "loading" spinner and message that's persistent since it doesn't have the data needed to render the form. To ensure the correct form is rendered, the initial data needs to be retrieved from the server. That is the first task.

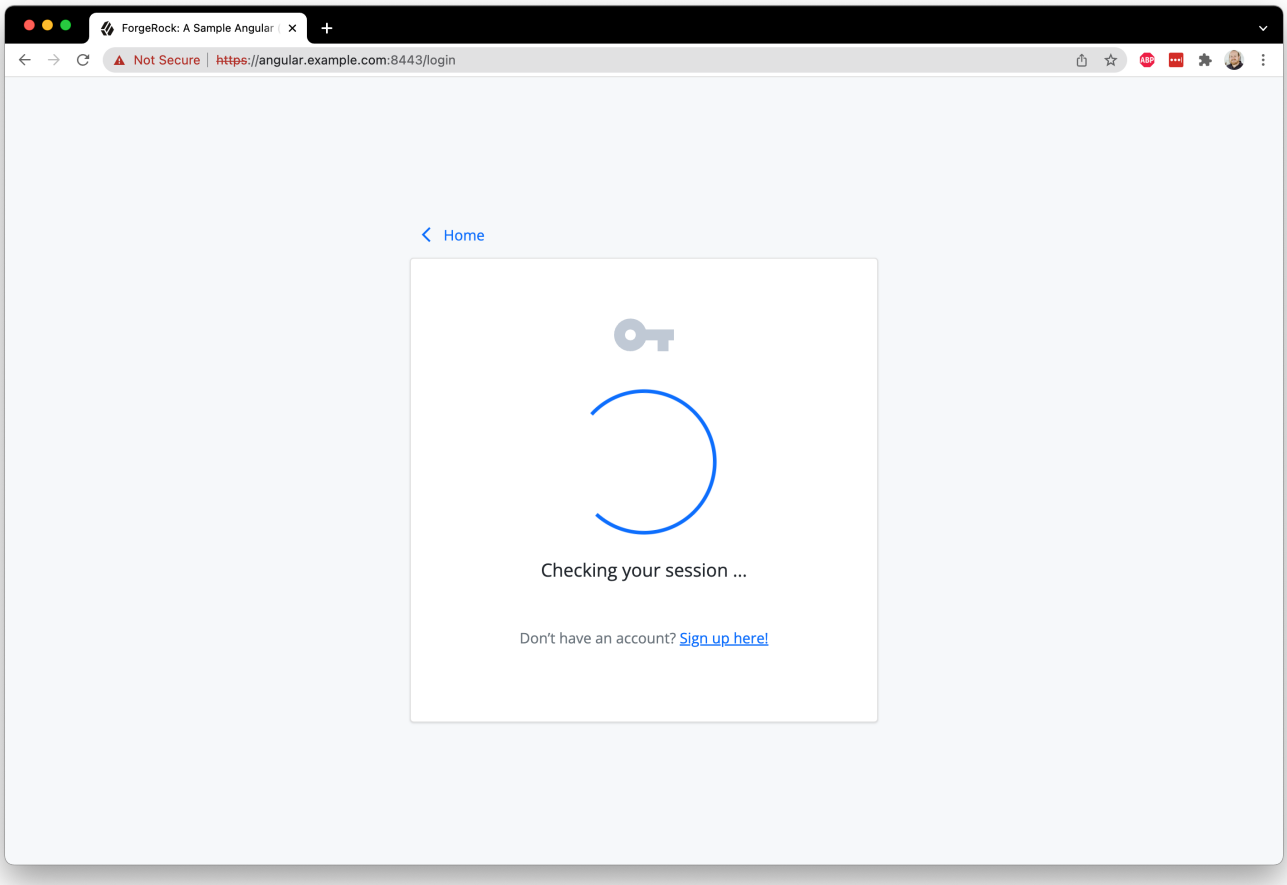


Figure 1. Login page with spinner

Since most of the action is taking place in `src/app/features/journey/form/form.component.html` and `src/app/features/journey/form/form.component.ts`, open both and add the SDK import to `form.component.ts`:

```
  import { Component, Input, OnInit } from '@angular/core';
  import { Router } from '@angular/router';
  import { environment } from '../../../../environments/environment';
  import { UserService } from 'src/app/services/user.service';
- import { FRLoginFailure, FRLoginSuccess, FRStep } from '@forgerock/javascript-sdk';
+ import { FRAuth, FRLoginFailure, FRLoginSuccess, FRStep } from '@forgerock/javascript-sdk';
@@ collapsed @@
```

`FRAuth` is the first object used as it provides the necessary methods for authenticating a user against the Login **Journey**/**Tree**. Use the `start()` method of `FRAuth` as it returns data we need for rendering the form.

Add the following code to the `nextStep` function to call the start function, initiating the authentication attempt using the SDK:

```
@@ collapsed @@
  async nextStep(step?: FRStep): Promise<void> {
    this.submittingForm = true;
+   try {
+     let nextStep = await FRAuth.next(step, { tree: this.tree });
+   } catch (err) {
+       console.log(err);
+   } finally {
+       this.submittingForm = false;
+   }
  }
@@ collapsed @@
```

The result of this initial request is stored in a variable named `nextStep`. We now need to work out whether this is a login failure, success, or step with instructions for what needs to be rendered to the user for input collection.

To handle these outcomes, add the following code after the code you added above:

```
@@ collapsed @@
  async nextStep(step?: FRStep): Promise<void> {
    this.submittingForm = true;

    try {
      let nextStep = await FRAuth.next(step, { tree: this.tree });

+     switch (nextStep.type) {
+       case 'LoginFailure':
+           this.handleFailure(nextStep);
+           break;
+         case 'LoginSuccess':
+           this.handleSuccess(nextStep);
+           break;
+         case 'Step':
+           this.handleStep(nextStep);
+           break;
+         default:
+           this.handleFailure();
+     }
    } catch (err) {
        console.log(err);
    } finally {
        this.submittingForm = false;
    }
  }
@@ collapsed @@
```

Since the `nextStep` type is likely a `Step` with instructions for rendering and collecting user input, we call the `handleStep()` function. We also set the `step` variable on the component ready for the template to process.

To process the `step`, we build a form that uses the `*ngFor` and `ngSwitch` directives to iterate over the callbacks and switch based on the callback type. This lets us use the appropriate component to render something to the user. Once the user provides their input and submits the form, we catch the submission and invoke the nextStep function again.

So starting with the form submission, we add the following code inside the `<div id="callbacks">` tag in the `FormComponent` template (`src/app/features/journey/form/form.component.html`)

```
@@ collapsed @@
  <div id="callbacks">
+   <form #callbackForm (ngSubmit)="nextStep(step)" ngNativeValidate class="cstm_form">
+     <app-button [buttonText]="buttonText" [submittingForm]="submittingForm">
+     </app-button>
+   </form>
  </div>
@@ collapsed @@
```

The form should now catch submissions. To iterate through the callbacks, add the following code inside the `<form>` tag you just added, just before the `<app-button>` tag:

```
@@ collapsed @@
  <div id="callbacks">
    <form #callbackForm (ngSubmit)="nextStep(step)" ngNativeValidate class="cstm_form">
+     <div *ngFor="let callback of step?.callbacks" v-bind:key="callback.payload._id">
+     </div>
      <app-button [buttonText]="buttonText" [submittingForm]="submittingForm">
      </app-button>
    </form>
  </div>
@@ collapsed @@
```

To switch based on the type of the callback, add the following code within the `<div>` tag you just added:

```
@@ collapsed @@
  <div *ngFor="let callback of step?.callbacks" v-bind:key="callback.payload._id">
+   <container-element [ngSwitch]="callback.getType()">
+   </container-element>
  </div>
@@ collapsed @@
```

Finally, to render something appropriate to the user based on the callback type (and handle unknown callbacks), add the below code within the `<container-element>` tag you just added.

```
@@ collapsed @@
  <container-element [ngSwitch]="callback.getType()">
+   <app-text *ngSwitchCase="'NameCallback'" [callback]="$any(callback)" [name]="callback?.payload?.input?.
[0]?.name" (updatedCallback)="$any(callback).setName($event)">
+   </app-text>

+   <app-password *ngSwitchCase="'PasswordCallback'" [callback]="$any(callback)" [name]="callback?.payload?.input?.
[0]?.name" (updatedCallback)="$any(callback).setPassword($event)">
+   </app-password>

+   <app-unknown *ngSwitchDefault [callback]="callback"></app-unknown>
  </container-element>
@@ collapsed @@
```

Refresh the page, and you should now have a dynamic form that reacts to the callbacks returned from our initial call to PingAM or PingOne Advanced Identity Cloud.

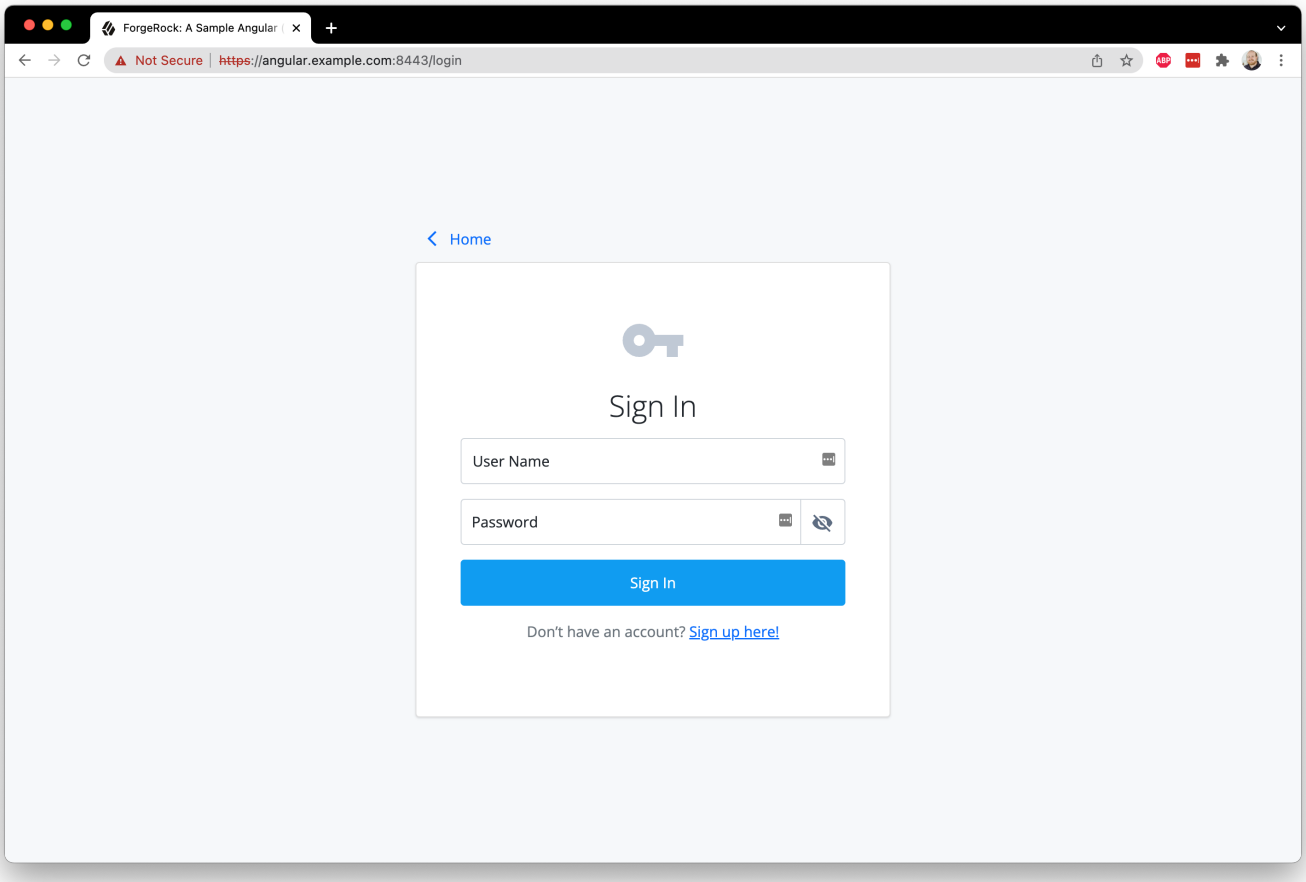


Figure 2. Login page form

Refresh the login page and use the test user to login. You should get a mostly blank login page if the user's credentials are valid and the journey completes. You can verify this by going to the Network panel within the developer tools and inspecting the last `/authenticate` request. It should have a `tokenId` and `successUrl` property.

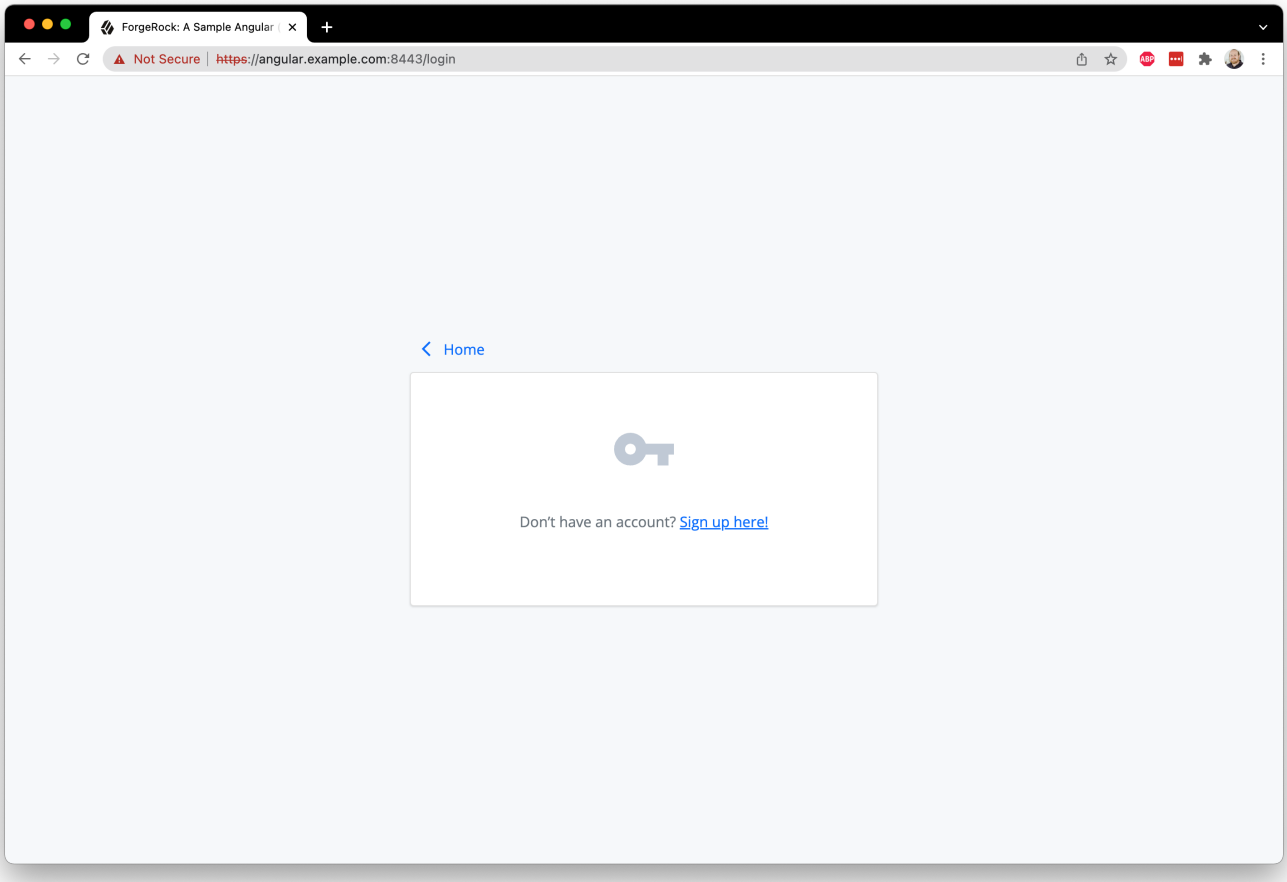


Figure 3. Successful request without handling render

You may ask, "How are the user's input values added to the `step` object?" Let's take a look at the component for rendering the username input. Open up the `Text` component: `src/app/features/journey/text/text.component.ts` and `src/app/features/journey/text/text.component.html`:

```
<input
  @@ collapsed @@
  (input)="updateValue($event)"
  @@ collapsed @@
/>
```

When the user changes the value of the input, the `(input)` event fires and calls `updateValue()`. This in turn uses the `EventEmitter` defined in the `@Output` directive to emit the updated value to the parent component - in this case, the `FormComponent`. From here, the `FormComponent` calls the appropriate convenience method in the SDK to set the value for the callback. This final piece is shown below (this is already in your project so no need to copy it):

```
<app-text *ngSwitchCase="'NameCallback'" [callback]="$any(callback)" [name]="callback?.payload?.input?.
[0]?.name" (updatedCallback)="$any(callback).setName($event)"
</app-text>
```

Each callback type has its own collection of methods for getting and setting data in addition to a base set of generic callback methods. The SDK automatically adds these methods to the callback prototype. For more information about these callback methods, see our API documentation^, or the source code in GitHub, for more details.

Now that the form is rendering and submitting, add conditions to the `FormComponent` template (`src/app/features/journey/form/form.component.html`), to handle the success and error response from PingAM or PingOne Advanced Identity Cloud. This code should be inserted towards the top of the file, inside the `<ng-container>` tag:

```
  <ng-container
  [ngTemplateOutlet]="success ? successMessage : failure ? failureMessage : step ? callbacks : loading"
>
    <ng-template #successMessage>
+     <app-loading [message]="'Success! Redirecting ...'"></app-loading>
    </ng-template>

    <ng-template #failureMessage>
+     <app-alert [message]="failure?.getMessage()" [type]="'error'"></app-alert>
    </ng-template>
@@ collapsed @@
```

Once you handle the success and error condition, return back to the browser and remove all cookies created from any previous logins. Refresh the page and login with your test user created in the Setup section above. You should see a “Success!” alert message. Congratulations, you are now able to authenticate users!

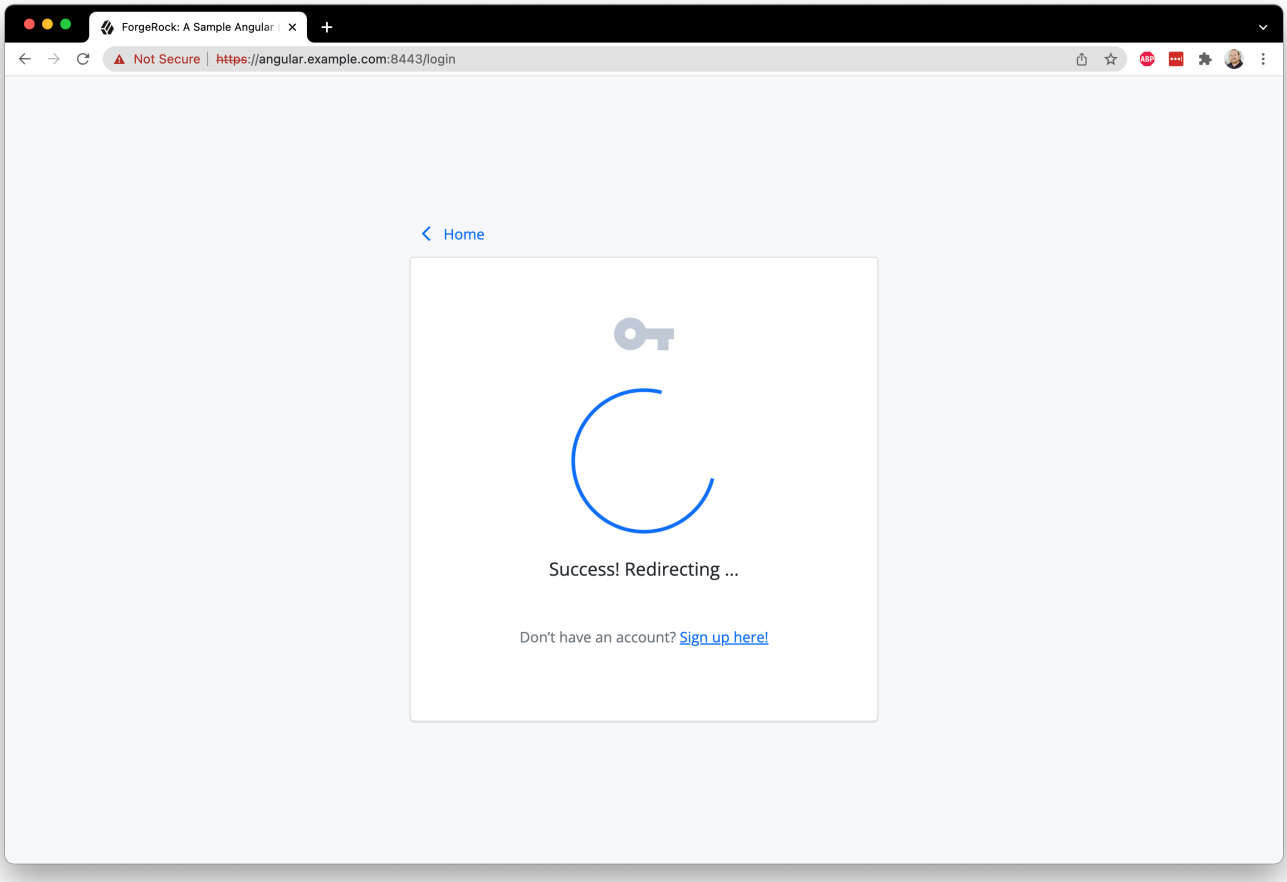


Figure 4. Login page with successful authentication

Continue to the OAuth 2.0 flow

At this point, the user is authenticated. The session has been created and a session cookie has been written to the browser. This is "session-based authentication", and is viable when your system (apps and services) can rely on cookies as the access artifact. However, there are increasing limitations with the use of cookies. In response to this, and other reasons, it's common to add an additional step to your authentication process: the "OAuth" or "OIDC flow".

The goal of this flow is to attain a separate set of tokens, replacing the need for cookies as the shared access artifact. The two common tokens are the access token and the ID Token. We focus on the access token in this guide. The specific flow that the SDK uses to acquire these tokens is called the Authorization Code Flow with PKCE.

To start, import the `TokenManager` and `UserManager` objects from the Ping SDK into the same `src/app/features/journey/form.component.ts` file - replace the import you added earlier with the following code:

```
  import { Component, Input, OnInit } from '@angular/core';
  import { Router } from '@angular/router';
  import { environment } from '../../../../environments/environment';
  import { UserService } from 'src/app/services/user.service';
- import { FRAuth, FRLoginFailure, FRLoginSuccess, FRStep } from '@forgerock/javascript-sdk';
+ import { FRAuth, FRLoginFailure, FRLoginSuccess, FRStep, TokenManager, UserManager, } from '@forgerock/javascript-
sdk';
@@ collapsed @@
```

In addition to the components that we were already importing, we have now imported the `TokenManager` and `UserManager` from the SDK.

Only an authenticated user that has a valid session can successfully request OAuth/OIDC tokens. We must therefore make sure we make this asynchronous token request after we get a `'LoginSuccess'` back from the authentication journey. In the code we wrote in the previous section, our processing of the response means that a `'LoginSuccess'` results in a call to the currently-empty function `handleSuccess`.

Let's invoke the OAuth 2.0 flow from here. Note that since the `getTokens` request is asynchronous, `handleSuccess` has been marked `async`.

Add the following code to the try block within `handleSuccess` to start the flow:

```
@@ collapsed @@
  async handleSuccess(success?: FRLoginSuccess) {
    this.success = success;

+   try {
+     await TokenManager.getTokens({ forceRenew: true });
+   } catch (err) {
+     console.error(err);
+   }
  }
@@ collapsed @@
```

Once the changes are made, return back to your browser and remove all cookies created from any previous logins. Refresh the page and verify the login form is rendered. If the success message continues to display, make sure "third-party cookies" are also removed.

Login with your test user. You should get a success message like you did before, but now check your browser's console log. You should see an additional entry of an object that contains your `idToken` and `accessToken`. Since the SDK handles storing these tokens for you, which are in `localStorage`, you have completed a full login and OAuth/OIDC flow.

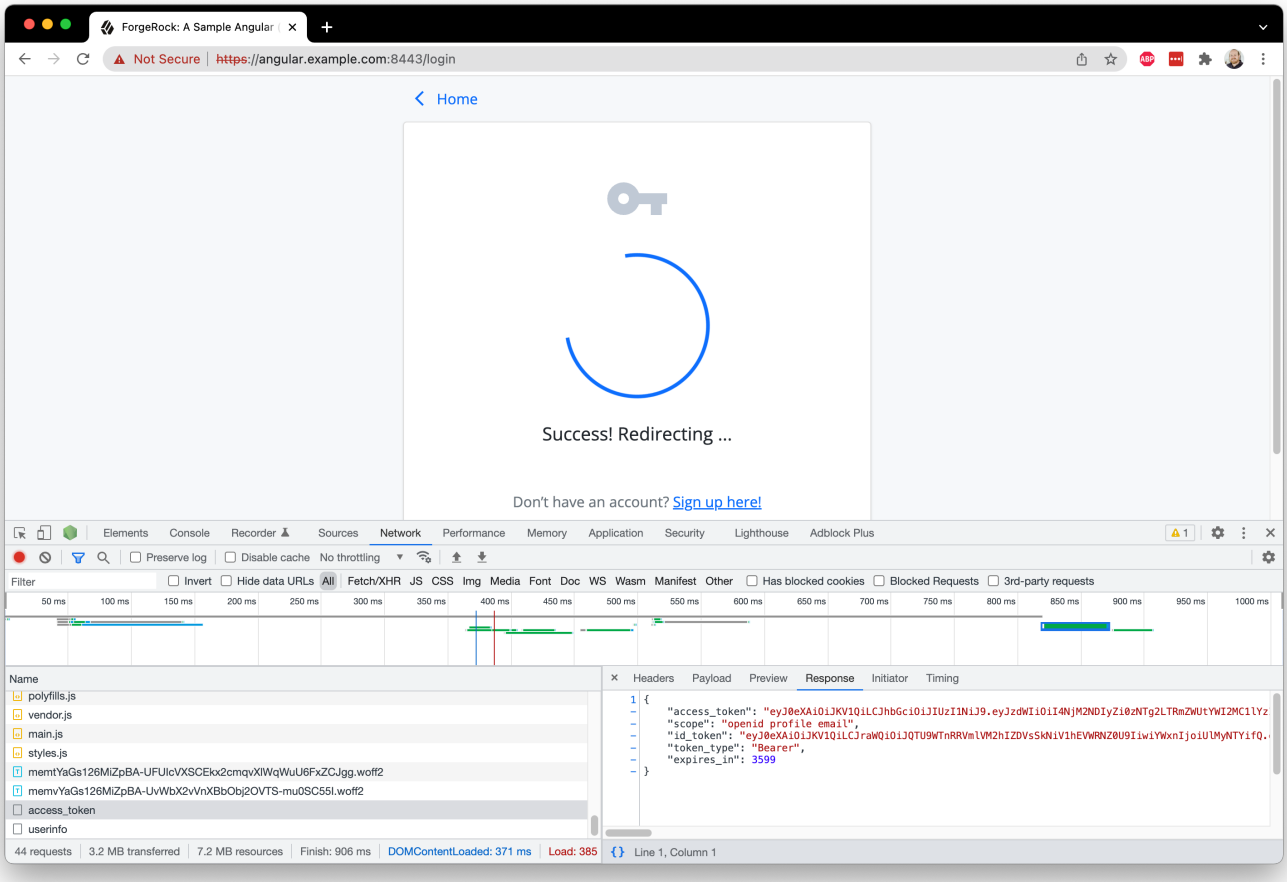


Figure 5. Login page with OAuth success

Request user information

Now that the user is authenticated and an access token is attained, you can now make your first authenticated request.

The SDK provides a convenience method for calling the `/userinfo` endpoint, a standard OAuth endpoint for requesting details about the current user. The data returned from this endpoint correlates with the "scopes" set within the SDK configuration.

The scopes `profile` and `email` allow the inclusion of user's first and last name as well as their email address.

To retrieve user information, add another single line of code to invoke the `getCurrentUser()` function of the SDK, underneath the `getTokens()` call:

```
@@ collapsed @@
  async handleSuccess(success?: FRLoginSuccess) {
    this.success = success;

    try {
      await TokenManager.getTokens({ forceRenew: true });

+     let info = await UserManager.getCurrentUser();
    } catch (err) {
      console.error(err);
    }
  }
@@ collapsed @@
```

We want to store the fact that the user is authenticated, together with the user information we retrieved, in a state that can be shared with other Angular components in our app. To do this, we have injected the service `UserService` into `FormComponent`. This service is also injected into other components that should need access to authentication status and user information.

To update the `UserService` and redirect the user to the home page, add the following code below the `getCurrentUser()` call:

```
@@ collapsed @@
  async handleSuccess(success?: FRLoginSuccess) {
    this.success = success;

    try {
      await TokenManager.getTokens({ forceRenew: true });

      let info = await UserManager.getCurrentUser();
+     this.userService.info = info;
+     this.userService.isAuthenticated = true;

+     this.router.navigateByUrl('/');
    } catch (err) {
      console.error(err);
    }
  }
@@ collapsed @@
```

Revisit the browser, clear out all cookies, storage and cache, and log in with your test user. Once you have landed on the home page you should notice that the page looks slightly different with an added success alert and message with the user's full name. This is due to the app "reacting" to the state in the `UserService` that we set just before the redirection.

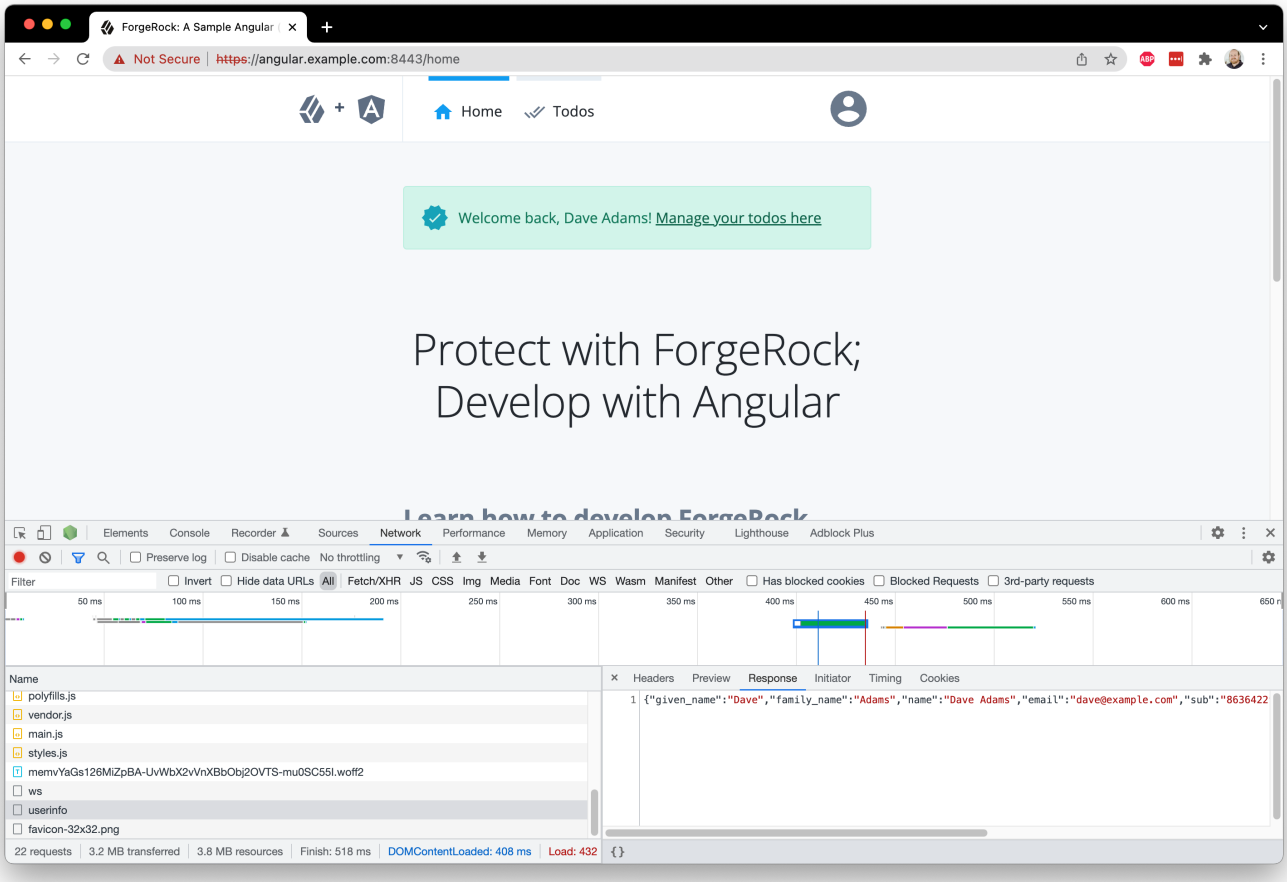


Figure 6. Home page with userinfo

React to the presence of the access token

To ensure your app provides a good user-experience, it's important to have a recognizable, authenticated experience, even if the user refreshes the page or closes and reopens the browser tab. This makes it clear to the user that they are logged in.

Currently, if you refresh the page, the authenticated experience is lost. Let's fix that!

If the user is logged in, there are tokens in the browser. To ensure the tokens are valid and the user information is available to the rest of the page, we use the `getCurrentUser()` function of the SDK. The function determines if the tokens are still valid. The function also retrieves the user information for use in the rest of the app.

To do this, add the following code to the `ngOnInit()` function in the main component - `src/app/app.component.ts`. This should provide what we need to re-initialise the user's authentication status:

```
@@ collapsed @@
  async ngOnInit(): Promise<void> {

      Config.set({
        clientId: environment.WEB_OAUTH_CLIENT,
        redirectUri: environment.APP_URL,
        scope: 'openid profile email address',
        serverConfig: {
          baseUrl: environment.AM_URL,
          timeout: 30000, // 90000 or less
        },
        realmPath: environment.REALM_PATH,
        tree: environment.JOURNEY_LOGIN,
      });

+     try {
+       const tokens: Tokens = await TokenStorage.get();
+       if (tokens !== undefined) {
+         // Assume user is likely authenticated if there are tokens
+         const info = await UserManager.getCurrentUser();
+         this.userService.isAuthenticated = true;
+         this.userService.info = info;
+       }
+     } catch (err) {
+       // User likely not authenticated
+       console.log(err);
+     }
  }
@@ collapsed @@
```

With a global state API available to the app using `UserService`, different components can pull this state in and use it to conditionally render a set of UI elements. Navigation elements and the displaying of profile data are good examples of such conditional rendering. Examples of this can be found by reviewing `src/app/layout/header/header.component.ts` and `src/app/views/home/home.component.ts`.

Validate the access token

The presence of the access token can be a good *hint* for authentication, but it doesn't mean the token is actually valid. Tokens can expire or be revoked on the server-side.

We are now focusing on protecting a particular page in our app (`todos`), so we may want to be sure that the user has valid tokens. We are currently just checking that there are tokens in the browser and redirecting to the login page. This is a reasonable approach and is quick since there are no network requests involved. However we have no assurance that the tokens are still valid. We could ensure that the tokens are still valid with the use of `getCurrentUser()` method as we do in the main component. However as this now requires a network request to complete before the page loads, it could impact on the speed at which the page loads. This is a decision that you must make for your implementation, depending on your requirements.

In this example, instead of just checking for presence of tokens, we prioritize security over speed by making sure that the token is valid before the page is rendered.

To protect a route by ensuring the user has a valid access token, open the `src/app/auth/auth.guard.ts` file which uses the `CanActivate` interface, and import the `UserManager` from the SDK:

```
@@ collapsed @@
  import { UserService } from '../services/user.service';
- import { Tokens, TokenStorage } from '@forgerock/javascript-sdk';
+ import { Tokens, TokenStorage, UserManager } from '@forgerock/javascript-sdk';
@@ collapsed @@
```

Then, replace the code within `canActivate` as follows:

```
@@ collapsed @@
    // Assume user is likely authenticated if there are tokens
    const tokens: Tokens = await TokenStorage.get();
+   const info = await UserManager.getCurrentUser();
-   if (tokens === undefined) {
+   if (tokens === undefined || info === undefined) {
    return loginUrl;
@@ collapsed @@
```

Revisit the browser and refresh the page. Navigate to the Todos page. You should notice a quick spinner and text communicating that the app is "verifying access". Once the server responds, the Todos page renders. The consequence of this is the protected route now has to wait for the server to respond, but the user's access has been verified by the server.

Request protected resources with an access token

Once the Todos page renders, notice how the the Todo collection appears empty. This is due to the request function in the `TodoService` being incomplete.

To make resource requests to a protected endpoint, we have an `HttpClient` module that provides a simple wrapper around the native `fetch()` method of the browser. When you call the `request()` method, it should retrieve the user's access token, and attach it as a Bearer Token to the request as an `authorization` header. This is what the resource server uses to make its own request to the server to validate the user's access token.

All requests to the Todos backend live in the `TodoService`, which is injected into the `TodosComponent` which renders the `/todos` page. Each of the functions dedicated to a particular backend request, call the convenience function `request()`, which needs to use the Ping SDK `HttpClient`.

To use the `HttpClient`, add the following import statement to the top of `src/app/services/todo.service.ts`:

```
  import { Injectable } from '@angular/core';
  import { Todo } from '../features/todo/todo';
  import { environment } from '../../environments/environment';
+ import { HttpClient } from '@forgerock/javascript-sdk';
@@ collapsed @@
```

Now, complete the `request()` function to use the `HttpClient` to make requests to the Todos backend - replace the existing return statement with the following:

```
@@ collapsed @@
    request(resource: string, method: string, data?: Todo): Promise<Response> {
-     return new Promise((resolve, reject) => reject('Method not implemented'));
+     return HttpClient.request({
+       url: resource,
+       init: {
+         headers: {
+           'Content-Type': 'application/json',
+         },
+         body: JSON.stringify(data),
+         method: method,
+       },
+       timeout: 5000,
+     });
    }
@@ collapsed @@
```

At this point, the user can login, request access tokens, and access the page of the protected resources (todos). Now, revisit the browser and clear out all cookies, storage, and cache. Keeping the developer tools open and on the network tab, log in with you test user. Once you have been redirected to the home page, do the following:

1. Click on the “Todos” item in the navigation bar - you should see that a lot of network activity should be listed.
2. Find the network call to the `/todos` endpoint (`http://localhost:9443/todos`).
3. Click on that network request and view the request headers.
4. Notice the `authorization` header with the bearer token; that’s the `HttpClient` in action.

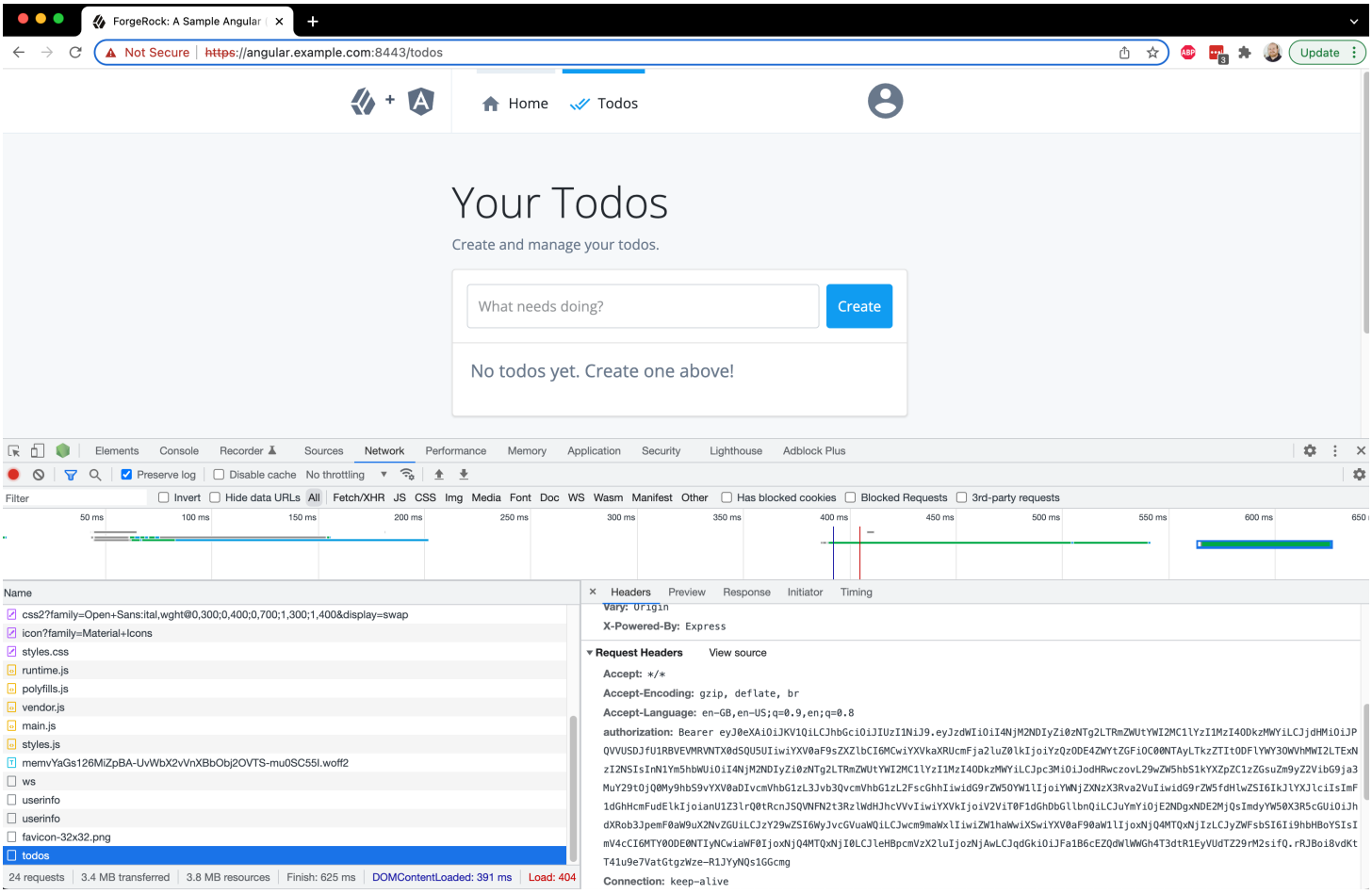


Figure 7. Todos page with successful request

Handle logout request

Of course, you can't have a protected app without providing the ability to log out. Luckily, this is a fairly easy task using the SDK.

Open up the `LogoutComponent` file `src/app/features/logout/logout.component.ts` and import `FRUser` from the Ping SDK:

```
@@ collapsed @@
  import { Component, OnInit } from '@angular/core';
  import { Router } from '@angular/router';
  import { UserService } from '../../services/user.service';
+ import { FRUser } from '@forgerock/javascript-sdk';
@@ collapsed @@
```

Logging the user out and revoking their tokens is easy using the `logout()` function of `FRUser`. Once this async call returns, we then remove any user information from `UserService` (and therefore other parts of the application since this is injected in other components). To do this, add the following code to `logout()`:

```
@@ collapsed @@
  async logout() {
+   try {
+     await FRUser.logout();
+     this.userService.info = undefined;
+     this.userService.isAuthenticated = false;
+     setTimeout(() => this.redirectToHome(), 1000);
+   } catch (err) {
+     console.error(`Error: logout did not successfully complete; ${err}`);
+   }
  }
@@ collapsed @@
```

Test the app

To test the app return to your browser, empty the local storage and cache, and reload the page.

You should now be able to log in with the demo user, navigate to the **Todos** page, add and edit some "Todos", and logout by selecting the profile icon in the top-right and clicking **Sign Out**.

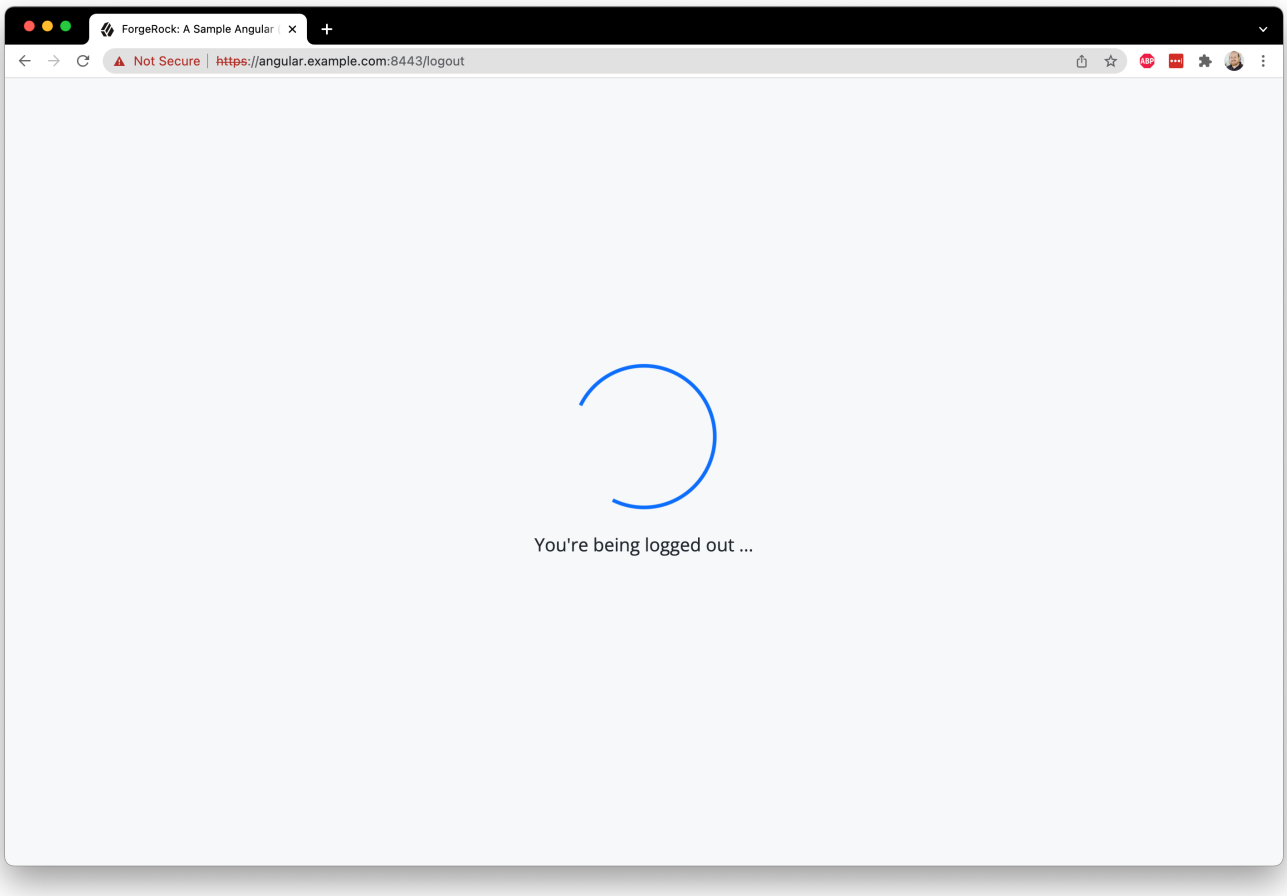


Figure 8. Logout page

Congratulations, you just built a protected app with Angular!

Authentication journey tutorial for an iOS Flutter app

This tutorial covers the basics of developing a protected mobile app with Flutter. It focuses on developing the iOS bridge code along with a minimal Flutter UI to authenticate a user.

Bridge code development is a concept common to mobile apps built using hybrid technologies. Hybrid is a term used when a portion of the mobile app uses a language that is not native to the platform (Android and Java or iOS and Swift).

Flutter is an open source framework by Google for building beautiful, natively compiled, multi-platform applications from a single codebase. Flutter requires this bridging code to provide the hybrid layer (Dart) access to native APIs (Swift in this case) or dependencies.

This guide uses the Ping SDK to implement the following application features:

1. Authentication through a simple journey/tree.
2. Requesting OAuth/OIDC tokens.
3. Requesting user information.
4. Logging a user out.

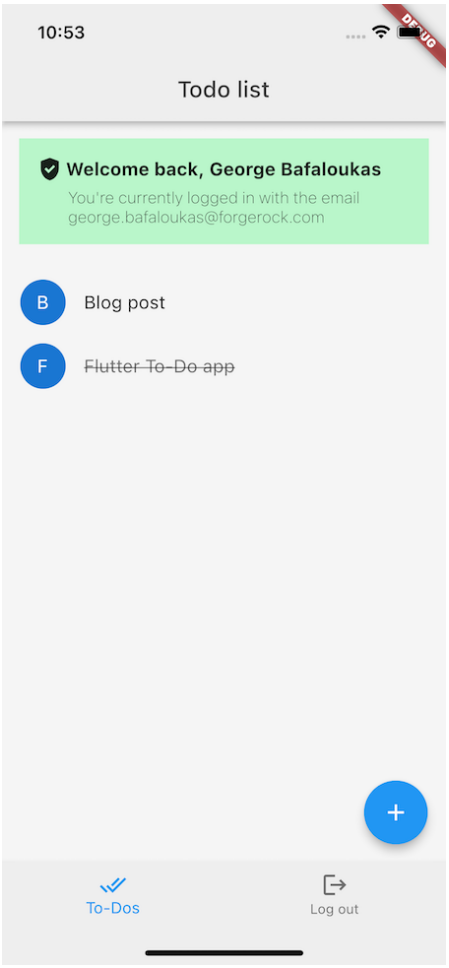


Figure 1. The to-do sample app

Before you begin

Before you begin this tutorial ensure you have set up your PingOne Advanced Identity Cloud tenant or PingAM server with the required configuration.

For example, you will need to configure CORS, have an OAuth 2.0 client application set up, as well as an authentication journey for the app to navigate.

Complete prerequisites »

Step 1. Download the samples

To start this tutorial, you need to download the SDK sample apps repo, which contains the projects you will use.

Start step 1 »

Step 2. Configure the projects

In this step you install the dependencies the projects require, and configure the connection properties.

Start step 2 »

Step 3. Configure connection properties

In this step, you configure the samples to connect to the authentication tree/journey and OAuth 2.0 client you created when setting up your server configuration.

Start step 3 »

Step 4. Build and run the project

Build and run the apps, and learn about *Hot Module Reloading*.

Start step 4 »

Step 5. Implement the iOS bridge code

In this step you implement the bridge code and add methods for starting the Ping SDK, logging a user in, stepping through a journey, and finally logging a user out.

Start step 5 »

Step 6. Implement the UI in Flutter

In this final step you implement the user interface for logging in, and code for submitting the forms. You will also handle returning to the list view, requesting user info, and handling logout triggers.

This is also the moment you can try out the fully functioning app.

Start step 6 »

Before you begin

To successfully complete this tutorial refer to the prerequisites and compatibility requirements in this section.

The tutorial also requires a configured server.

Compatibility

iOS

This sample app is compatible with iOS 12 and later.

Prerequisites

Xcode

You can download the latest version for free from https://developer.apple.com/xcode/.

Swift Packager Manager

This project requires use of the Swift Package Manager (SPM).

Dart

Configure Dart in Xcode.

Flutter

Install the latest version of Flutter.

You will also need an IDE so that you can work with the Flutter UI. To learn more about the IDEs that Flutter supports, refer to Set up an editor in the *Flutter* documentation.

Server configuration

This tutorial requires you to configure one of the following servers:

PingOne Advanced Identity Cloud

PingAM

PingOne Advanced Identity Cloud

The samples and tutorials in this documentation often require that you have an identity set up so that you can test authentication.

To create a demo user in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. In the left panel, click **Identities** > **Manage**.
3. Click **+ New Alpha realm - User**.
4. Enter the following details:
    - **Username** = `demo`
    - **First Name** = `Demo`
    - **Last Name** = `User`
    - **Email Address** = `demo.user@example.com`
    - **Password** = `Ch4ng3it!`
5. Click **Save**.

Authentication journeys provide fine-grained authentication by allowing multiple paths and decision points throughout the flow. Authentication journeys are made up of nodes that define actions taken during authentication.

Each node performs a single task, such as collecting a username or making a simple decision. Nodes can have multiple outcomes rather than just success or failure. For details, see the Authentication nodes configuration reference in the PingAM documentation.

To create a simple journey for use when testing the Ping SDKs, follow these steps:

1. In your PingOne Advanced Identity Cloud tenant, navigate to **Journeys**, and click **+ New Journey**.
2. Enter a name, such as `sdkUsernamePasswordJourney` and click **Save**.

The authentication journey designer appears.

3. Drag the following nodes into the designer area:
    - **Page Node**
    - **Platform Username**
    - **Platform Password**
    - **Data Store Decision**
4. Drag and drop the **Platform Username** and **Platform Password** nodes onto the **Page Node**, so that they both appear on the same page when logging in.
5. Connect the nodes as follows:

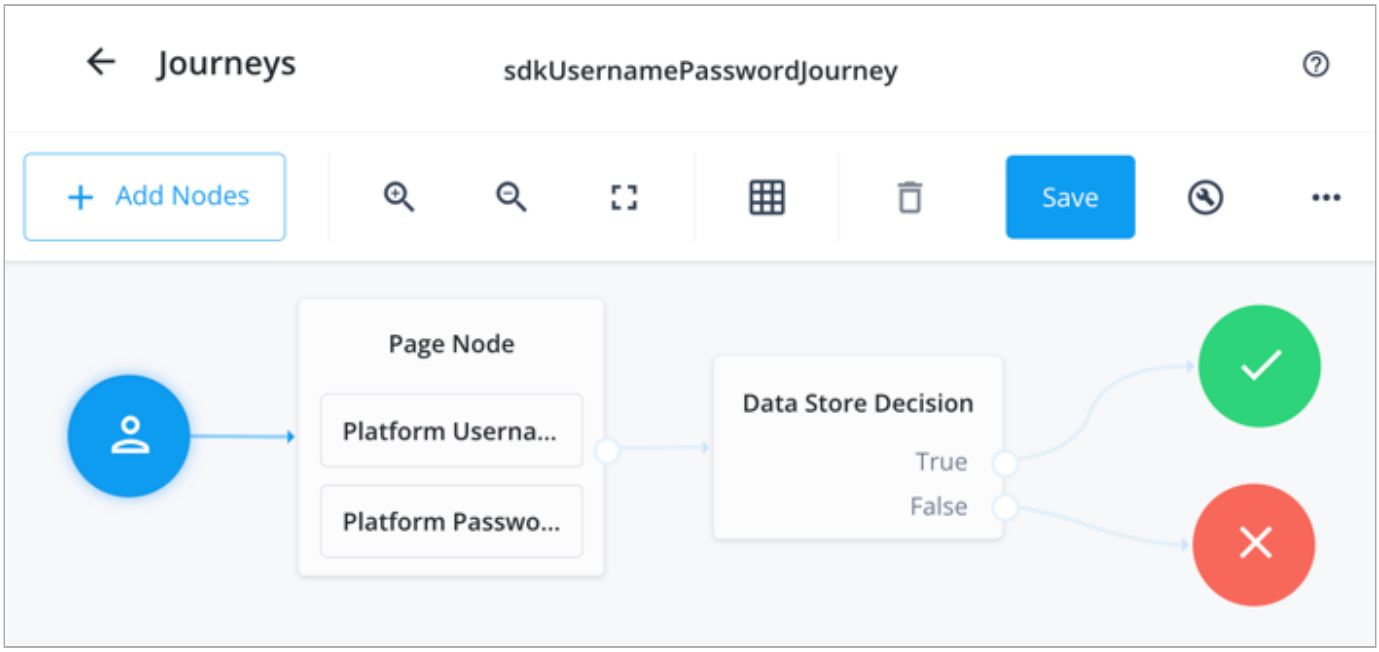


Figure 1. Example username and password authentication journey

6. Click **Save**.

Public clients do not use a client secret to obtain tokens because they are unable to keep them hidden. The Ping SDKs commonly use this type of client to obtain tokens, as they cannot guarantee safekeeping of the client credentials in a browser or on a mobile device.

To register a *public* OAuth 2.0 client application for use with the SDKs in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. In the left panel, click **Applications**.
3. Click **+ Custom Application**.
4. Select **OIDC - OpenId Connect** as the sign-in method, and then click **Next**.
5. Select **Native / SPA** as the application type, and then click **Next**.

6. In **Name**, enter a name for the application, such as `Public SDK Client`.
7. In **Owners**, select a user that is responsible for maintaining the application, and then click **Next**.

> **Tip**
>
> When trying out the SDKs, you could select the `demo` user you created previously.

8. In **Client ID**, enter `sdkPublicClient`, and then click **Create Application**.

   PingOne Advanced Identity Cloud creates the application and displays the details screen.
9. On the **Sign On** tab:
   1. In **Sign-In URLs**, enter the following values:

      `https://com.example.flutter.todo/callback`

      > **Important**
      >
      > Also add any other domains where you host SDK applications.

   2. In **Grant Types**, enter the following values:

      `Authorization Code`

      `Refresh Token`

   3. In **Scopes**, enter the following values:

      `openid profile email address`
10. Click Show advanced settings, and on the **Authentication** tab:
    1. In **Token Endpoint Authentication Method**, select `none`.
    2. In **Client Type**, select `Public`.
    3. Enable the **Implied Consent** property.
11. Click **Save**.

The application is now configured to accept client connections from and issue OAuth 2.0 tokens to the example applications and tutorials covered by this documentation.

The provider specifies the supported OAuth 2.0 configuration options for a realm.

To ensure the PingOne Advanced Identity Cloud OAuth 2.0 provider service is configured for use with the Ping SDKs, follow these steps:

1. In your PingOne Advanced Identity Cloud tenant, navigate to **Native Consoles** > **Access Management**.
2. In the left panel, click **Services**.
3. In the list of services, click **OAuth2 Provider**.
4. On the **Core** tab, ensure **Issue Refresh Tokens** is enabled.

5. On the **Consent** tab, ensure **Allow Clients to Skip Consent** is enabled.
6. Click **Save Changes**.

PingAM

The samples and tutorials in this documentation often require that you have an identity set up so that you can test authentication.

To create a demo user in PingAM, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. Navigate to **Identities**, and then click **+ Add Identity**.
3. Enter the following details:
   - **User ID** = `demo`
   - **Password** = `Ch4ng3it!`
   - **Email Address** = `demo.user@example.com`
4. Click **Create**.

Authentication trees provide fine-grained authentication by allowing multiple paths and decision points throughout the authentication flow. Authentication trees are made up of nodes that define actions taken during authentication.

Each node performs a single task, such as collecting a username or making a simple decision. Nodes can have multiple outcomes rather than just success or failure. For details, see the Authentication nodes configuration reference in the PingAM documentation.

To create a simple tree for use when testing the Ping SDKs, follow these steps:

1. Under **Realm Overview**, click **Authentication Trees**, then click **Create Tree**.
2. Enter a tree name, for example `sdkUsernamePasswordJourney`, and then click **Create**.

   The authentication tree designer appears, showing the **Start** entry point connected to the **Failure** exit point.
3. Drag the following nodes from the **Components** panel on the left side into the designer area:
   - **Page Node**
   - **Username Collector**
   - **Password Collector**
   - **Data Store Decision**
4. Drag and drop the **Username Collector** and **Password Collector** nodes onto the **Page Node**, so that they both appear on the same page when logging in.
5. Connect the nodes as follows:

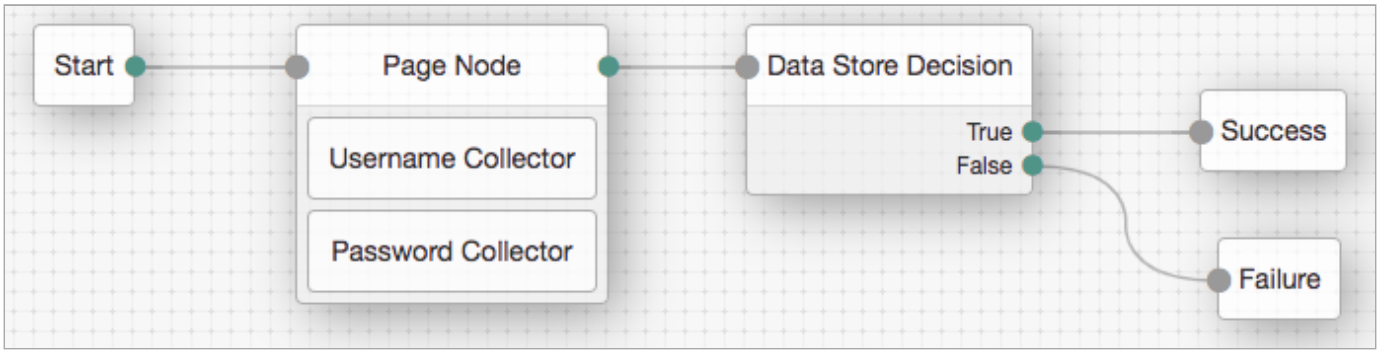


Figure 2. Example username and password authentication tree

6. Select the **Page Node**, and in the **Properties** pane, set the **Stage** property to `UsernamePassword`.

   > **Tip**
   >
   > You can configure the node properties by selecting a node and altering properties in the right-hand panel.

   One of the samples uses this specific value to determine the custom UI to display.

7. Click **Save**.

Public clients do not use a client secret to obtain tokens because they are unable to keep them hidden. The Ping SDKs commonly use this type of client to obtain tokens, as they cannot guarantee safekeeping of the client credentials in a browser or on a mobile device.

To register a *public* OAuth 2.0 client application for use with the SDKs in AM, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. Navigate to **Applications** > **OAuth 2.0** > **Clients**, and then click **+ Add Client**.
3. In **Client ID**, enter `sdkPublicClient`.
4. Leave **Client secret** empty.
5. In **Redirection URIs**, enter the following values:

   `https://com.example.flutter.todo/callback`

   > **Important**
   >
   > Also add any other domains where you will be hosting SDK applications.

6. In **Scopes**, enter the following values:

   `openid profile email address`

7. Click **Create**.

   PingAM creates the new OAuth 2.0 client, and displays the properties for further configuration.

8. On the **Core** tab:

   1. In **Client type**, select `Public`.

2. Disable **Allow wildcard ports in redirect URIs**.
3. Click **Save Changes**.

9. On the **Advanced** tab:
    1. In **Grant Types**, enter the following values:

    ```
    Authorization Code
    Refresh Token
    ```

    2. In **Token Endpoint Authentication Method**, select `None`.
    3. Enable the **Implied consent** property.
10. Click **Save Changes**.

The provider specifies the supported OAuth 2.0 configuration options for a realm.

To ensure the PingAM OAuth 2.0 provider service is configured for use with the Ping SDKs, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. In the left panel, click **Services**.
3. In the list of services, click **OAuth2 Provider**.
4. On the **Core** tab, ensure **Issue Refresh Tokens** is enabled.
5. On the **Consent** tab, ensure **Allow Clients to Skip Consent** is enabled.
6. Click **Save Changes**.

Step 1. Download the samples

To start this tutorial, you need to download the Flutter sample app repo, which contains the projects you will use.

1. In a web browser, navigate to the Flutter Sample App repository.
2. Download the source code using one of the following methods:

Download a ZIP file

1. Click **Code**, and then click **Download ZIP**.
2. Extract the contents of the downloaded ZIP file to a suitable location.

Use a Git-compatible tool to clone the repo locally

1. Click **Code**, and then copy the HTTPS URL.
2. Use the URL to clone the repository to a suitable location.

   For example, from the command-line you could run:

```
git clone https://github.com/ForgeRock/forgerock-flutter-sample.git
```

The result of these steps is a local folder named `forgerock-flutter-sample`.

Step 2. Configure the projects

In this step you install the dependencies the projects require.

Install the Ping SDK for iOS

This Flutter app requires the native Ping SDK for iOS. Install this by using Swift Package Manager (SPM) on the generated iOS project:

1. Navigate to the iOS project, `forgerock-flutter-sample/Flutter_To_Do_app/flutter_todo_app/ios`.
2. In Xcode, open `Runner.xcworkspace`.
3. Select the **Runner** project and navigate to **Package Dependencies**.
4. Click the + sign, and add the Ping SDK for iOS repository, `https://github.com/ForgeRock/forgerock-ios-sdk.git`.
5. Add the `FRCore` and `FRAuth` libraries to the project.

Install Flutter

Next, we need to open Android Studio and build the project.

If you haven't configured Android Studio for Flutter, please follow the guide in the Flutter documentation.

Don't forget to set the Dart SDK path in Android Studio. You can find that in the folder where you downloaded the Flutter SDK. (For example, `~/flutter/bin/cache/dart-sdk`.)

In Android Studio, click **File** > **Open**, and navigate to `forgerock-flutter-sample/Flutter_To_Do_app/`.

When Android Studio loads the project and is ready, install any gradle dependencies, and select the iOS simulator to build and run the project.

Install API server dependencies

Install the **TODO** Node.JS API server app dependencies by using `npm`:

1. In a Terminal window, navigate to the root folder, `forgerock-flutter-sample`.
2. Enter `npm install`.

Step 3. Configure connection properties

In this step, you configure the sample app to connect to the authentication tree/journey you created when setting up your server configuration.

Using the server settings from above, edit the `.env.js` file within the project. This can be found the root folder of the project.

Add your relevant values to configure all the important server settings in the project. Not all variables will need values at this time.

You can list the file in the Terminal by doing `ls -a`, and edit it using a text editor like `nano` or `vi`.

Example `.env.js` file

```
/**
 * Avoid trailing slashes in the URL string values below.
 */
const AM_URL = 'https://openam-forgerock-sdks.forgeblocks.com/am'; // Required; enter _your_ PingAM URL
const DEBUGGER_OFF = true;
const DEVELOPMENT = true;
const API_URL = 'https://api.example.com:9443'; // (your resource API server's URL)
const JOURNEY_LOGIN = 'sdkUsernamePasswordJourney'; // (name of journey/tree for Login)
const JOURNEY_REGISTER = 'Registration'; // (name of journey/tree for Register)
const SEC_KEY_FILE = './updatedCerts/api.example.com.key';
const SEC_CERT_FILE = './updatedCerts/api.example.com.crt';
const REALM_PATH = ''; //Required (ex: alpha)
const REST_OAUTH_CLIENT = ''; // (name of private OAuth 2.0 client/application)
const REST_OAUTH_SECRET = ''; // (the secret for the private OAuth 2.0 client/application)
const WEB_OAUTH_CLIENT = 'sdkPublicClient'; // (the name of the public OAuth 2.0 client/application)
const PORT = '9443';
```

Descriptions of relevant values:

`AM_URL`

The URL that references PingAM itself (for PingOne Advanced Identity Cloud, the URL is likely `https://<tenant-name>.forgeblocks.com/am`).

`API_PORT` *and* `API_BASE_URL`

These just need to be "truthy" (not 0 or an empty string) right now to avoid errors, and we will use them in a future part of this series.

`DEBUGGER_OFF`

When `true`, this disables the `debugger` statements in the JavaScript layer. These debugger statements are for learning the integration points at runtime in your browser. When the browser's developer tools are open, the app pauses at each integration point. Code comments above each integration point explain its use.

`REALM_PATH`

The realm of your server (likely `root`, `alpha`, or `bravo`).

`REST_OAUTH_CLIENT` *and* `REST_OAUTH_SECRET`

We will use these values in a future part of this series, so any string value will do.

Step 4. Build and run the project

Now that everything is set up, build and run the to-do app project.

1. Go back to the iOS project (`forgerock-flutter-sample/Flutter_To_Do_app/flutter_todo_app/ios`).
2. If the project is not already open in Xcode double-click `Runner.xcworkspace` .
3. Once Xcode is ready, select iPhone 11 or higher as the target for the device simulator on which to run the app.
4. Now, click the **build/play** button to build and run this application in the target simulator.

With everything up and running, you will need to rebuild the project with Xcode when you modify the bridge code (Swift files). But, when modifying the Flutter code, it will use "hot module reloading" to automatically reflect the changes in the app without having to manually rebuild the project.

Troubleshooting

1. Under the **General** tab, make sure that the `FRAuth` and `FRCore` frameworks are added to your target's **Frameworks, Libraries, and Embedded Content**.
2. Bridge code has been altered, so be aware of API name changes.

Using Xcode and iOS Simulator

We recommend the use of iPhone 11 or higher as the target for the iOS simulator. When you first run the build command in Xcode (clicking the **Play** button), it takes a while for the app to build, the OS to load, and app to launch within the simulator. Once the app is launched, rebuilding it is much faster if the changes are not automatically "hot reloaded" when made in the Flutter layer.

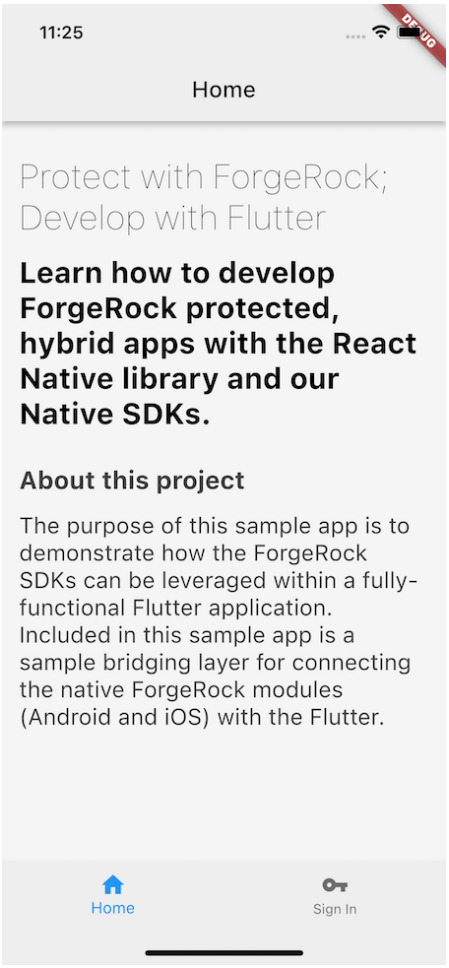


Figure 1. To-do app home screen

> **Note**
>
> Only the home screen will render successfully at this moment. If you click the **Sign In** button, it won't be fully functional. This is **intended** as you will develop this functionality throughout this tutorial.

Once the app is built and running, you will have access to all the logs in the Xcode output console.

Figure 2. Xcode log output

Step 5. Implement the iOS bridge code

Review the files that allow for the "bridging" between the Flutter project and the native Ping SDK.

In Xcode, navigate to the **Runner/Runner** directory, and you will see a few important files:

`FRAuthSampleBridge.swift`

The main Swift bridging code that provides the callable methods for the Flutter layer.

`FRAuthSampleStructs.swift`

Provides the structs for the Swift bridging code.

`FRAuthSampleHelpers.swift`

Provides the extensions to often used objects within the bridge code.

`FRAuthConfig`

A `.plist` file that configures the Ping SDK for iOS to the appropriate authorization server.

> **Note**
>
> The remainder of the files within the workspace are automatically generated when you create a Flutter project with the CLI command, so you can ignore them.

Configure your `.plist` file

In the Xcode directory/file list section, also known as the **Project Navigator**, complete the following:

1. Find `FRAuthConfig.plist` file within the `ios/Runner` directory.
2. Add the name of your PingOne Advanced Identity Cloud or PingAM cookie.
3. Add the OAuth client you created from above.
4. Add your authorization server URLs.
5. Add the login tree you created above.

A hypothetical example (your values may vary):

```
  <dict>
    <key>forgerock_cookie_name</key>
-   <string></string>
+   <string>e1babb394ea5130</string>
    <key>forgerock_enable_cookie</key>
    <true/>
    <key>forgerock_oauth_client_id</key>
    <string>flutterOAuthClient</string>
    <key>forgerock_oauth_redirect_uri</key>
    <string>https://com.example.flutter.todo/callback</string>
    <key>forgerock_oauth_scope</key>
    <string>openid profile email address</string>
    <key>forgerock_oauth_url</key>
-   <string></string>
+   <string>https://auth.forgerock.com/am</string>
    <key>forgerock_oauth_threshold</key>
    <string>60</string>
    <key>forgerock_url</key>
-   <string></string>
+   <string>https://auth.forgerock.com/am</string>
    <key>forgerock_realm</key>
-   <string></string>
+   <string>alpha</string>
    <key>forgerock_timeout</key>
    <string>60</string>
    <key>forgerock_keychain_access_group</key>
    <string>com.forgerock.flutterTodoApp</string>
    <key>forgerock_auth_service_name</key>
-   <string></string>
+   <string>UsernamePassword</string>
    <key>forgerock_registration_service_name</key>
-   <string></string>
+  <string>Registration</string>
</dict>
```

Descriptions of relevant values:

`forgerock_cookie_name`

If you have an PingOne Advanced Identity Cloud tenant, you can find this random string value under the **Tenant Settings** in the top-right dropdown in the admin UI. If you have your own installation of PingAM, this is often `iPlanetDirectoryPro`.

`forgerock_url` *and* `forgerock_oauth_url`

The URL of PingAM within your server installation.

`forgerock_realm`

The realm of your server (likely `root`, `alpha`, or `bravo`).

`forgerock_auth_service_name`

This is the journey/tree that you use for login.

`forgerock_registration_service_name`

This is the journey/tree that you use for registration, but it will not be used until a future part of this tutorial series.

Write the `start()` method

Staying within the `Runner` directory, find the `FRAuthSampleBridge` file and open it. We have parts of the file already stubbed out and the dependencies are already installed. All you need to do is write the functionality.

For the SDK to initialize with the `FRAuth.plist` configuration from Step 2, write the `start` function as follows:

```
   import Foundation
   import FRAuth
   import FRCore
   import Flutter

   public class FRAuthSampleBridge {
       var currentNode: Node?
       private let session = URLSession(configuration: .default)

    @objc func frAuthStart(result: @escaping FlutterResult) {

+      /**
+       * Set log level to all
+       */
+      FRLog.setLogLevel([.all])
+
+      do {
+        try FRAuth.start()
+        let initMessage = "SDK is initialized"
+        FRLog.i(initMessage)
+        result(initMessage)
+      } catch {
+        FRLog.e(error.localizedDescription)
+        result(FlutterError(code: "SDK Init Failed",
+                            message: error.localizedDescription,
+                            details: nil))
+      }
    }
```

The `start()` function above calls the Ping SDK for iOS's `start()` method on the `FRAuth` class. There's a bit more that may be required within this function for a production app. We'll get more into this in a separate part of this series, but for now, let's keep this simple.

Write the `login()` method

Once the `start()` method is called, and it has initialized, the SDK is ready to handle user requests. Let's start with `login()`.

Just underneath the `start()` method we wrote above, add the `login()` method.

```
@@ collapsed @@

@objc func frAuthStart(result: @escaping FlutterResult) {
    // Set log level according to your needs
    FRLog.setLogLevel([.all])

    do {
      try FRAuth.start()
        result("SDK Initialised")
        FRUser.currentUser?.logout()
    }
    catch {
      FRLog.e(error.localizedDescription)
        result(FlutterError(code: "SDK Init Failed",
                            message: error.localizedDescription,
                            details: nil))
    }
  }

 @objc func login(result: @escaping FlutterResult) {
+   FRUser.login { (user, node, error) in
+     self.handleNode(user, node, error, completion: result)
+   }
 }

@@ collapsed @@
```

This `login()` function initializes the journey/tree specified for authentication. You call this method without arguments as it does not log the user in. This initial call to the server will return the first set of callbacks that represents the first node in your journey/tree to collect user data.

Also, notice that we have a special "handler" function within the callback of `FRUser.login()`. This `handleNode()` method serializes the `node` object that the Ping SDK for iOS returns in a JSON string. Data passed between the "native" layer and the Flutter layer is limited to serialized objects. This method can be written in many ways and should be written in whatever way is best for your application.

Write the `next()` method

To finalize the functionality needed to complete user authentication, we need a way to iteratively call `next()` until the tree completes successfully or fails. In the bridge file, add a private method called `handleNode()`.

First, we will write the decoding of the JSON string and prepare the node for submission.

```
@@ collapsed @@

@objc func login(result: @escaping FlutterResult) {
      FRUser.login { (user, node, error) in
          self.handleNode(user, node, error, completion: result)
      }
  }

 @objc func next(_ response: String, completion: @escaping FlutterResult) {
+   let decoder = JSONDecoder()
+   let jsonData = Data(response.utf8)
+   if let node = self.currentNode {
+     var responseObject: Response?
+     do {
+       responseObject = try decoder.decode(Response.self, from: jsonData)
+     } catch  {
+       FRLog.e(String(describing: error))
+       completion(FlutterError(code: "Error",
+                                    message: error.localizedDescription,
+                                details: nil))
+     }
+
+     let callbacksArray = responseObject!.callbacks ?? []
+
+     for (outerIndex, nodeCallback) in node.callbacks.enumerated() {
+       if let thisCallback = nodeCallback as? SingleValueCallback {
+         for (innerIndex, rawCallback) in callbacksArray.enumerated() {
+           if let inputsArray = rawCallback.input, outerIndex == innerIndex,
+             let value = inputsArray.first?.value {
+
+             thisCallback.setValue(value.value as! String)
+           }
+         }
+       }
+     }
+
+     //node.next logic goes here
+
+
+   } else {
+     completion(FlutterError(code: "Error",
+                                 message: "UnkownError",
+                                 details: nil))
+   }
 }

@@ collapsed @@
```

Now that you've prepared the data for submission, introduce the `node.next()` call from the Ping SDK for iOS. Then, handle the subsequent `node` returned from the `next()` call, or process the success or failure representing the completion of the journey/tree.

```
@@ collapsed @@

      for (outerIndex, nodeCallback) in node.callbacks.enumerated() {
        if let thisCallback = nodeCallback as? SingleValueCallback {
          for (innerIndex, rawCallback) in callbacksArray.enumerated() {
            if let inputsArray = rawCallback.input, outerIndex == innerIndex,
              let value = inputsArray.first?.value {

              thisCallback.setValue(value)
            }
          }
        }
      }

     //node.next logic goes here
+     node.next(completion: { (user: FRUser?, node, error) in
+       if let node = node {
+         self.handleNode(user, node, error, completion: completion)
+       } else {
+         if let error = error {
+           completion(FlutterError(code: "LoginFailure",
+                                   message: error.localizedDescription,
+                                  details: nil))
+           return
+         }
+
+         let encoder = JSONEncoder()
+         encoder.outputFormatting = .prettyPrinted
+         do {
+             if let user = user, let token = user.token, let data = try? encoder.encode(token), let jsonAccessToken
= String(data: data, encoding: .utf8) {
+                 completion(try ["type": "LoginSuccess", "sessionToken": jsonAccessToken].toJson())
+             } else {
+                 completion(try ["type": "LoginSuccess", "sessionToken": ""].toJson())
+             }
+           }
+           catch {
+             completion(FlutterError(code: "Serializing Response failed",
+                                   message: error.localizedDescription,
+                                   details: nil))
+           }
+       }
+     })
    } else {
      completion(FlutterError(code: "Error",
                              message: "UnkownError",
                              details: nil))
    }
  }

@@ collapsed @@
```

The above code handles a limited number of callback types. Handling full authentication and registration journeys/trees requires additional callback handling. To keep this tutorial simple, we'll focus just on `SingleValueCallback` type.

Write the `logout()` bridge method

Finally, add the following lines of code to enable logout for the user:

```
@@ collapsed @@

    } else {
      completion(FlutterError(code: "Error",
                              message: "UnkownError",
                              details: nil))
    }

  @objc func frLogout(result: @escaping FlutterResult) {
+        FRUser.currentUser?.logout()
+        result("User logged out")
  }

@@ collapsed @@
```

Step 6. Implement the UI in Flutter

Review how the application renders the *home* view.

In Android Studio, navigate to the Flutter project, **flutter_todo_app** > **java/main.dart**.

Open up the second file in the above sequence, the `java/main.dart` file, and notice the following:

1. The use of `import 'package:flutter/material.dart';` from the Flutter library.
2. The `TodoApp` class extending `StatefulWidget`.
3. The `_TodoAppState` class extending `State<TodoApp>`.
4. Building the UI for the navigation bar.

```
import 'package:flutter/material.dart';
import 'package:flutter_todo_app/home.dart';
import 'package:flutter_todo_app/login.dart';
import 'package:flutter_todo_app/todolist.dart';

void main() => runApp(
  new TodoApp(),
);

class TodoApp extends StatefulWidget {
  @override
  _TodoAppState createState() => new _TodoAppState();
}

class _TodoAppState extends State<TodoApp> {
  int _selectedIndex = 0;

  final _pageOptions = [
    HomePage(),
    LoginPage(),
    TodoList(),
  ];

  void _onItemTapped(int index) {
    setState(() {
      _selectedIndex = index;
    });
  }

  @override
  Widget build(BuildContext context) {
    return new MaterialApp(
      home: Scaffold(
        body: _pageOptions[_selectedIndex],
        bottomNavigationBar: BottomNavigationBar(
          items: const <BottomNavigationBarItem>[
            BottomNavigationBarItem(
              icon: Icon(Icons.home),
              label: 'Home',
            ),
            BottomNavigationBarItem(
              icon: Icon(Icons.vpn_key),
              label: 'Sign In',
            ),
          ],
          currentIndex: _selectedIndex,
          selectedItemColor: Colors.blueAccent[800],
          onTap: _onItemTapped,
          backgroundColor: Colors.grey[200],
        )
      ),
    );
  }
}
```

Flutter uses something called `MethodChannel` to communicate between Flutter and the Native layer. In this application we will define a `MethodChannel` with the following identifier: `'forgerock.com/SampleBridge'`.

The same identifier will be used in the iOS FRSampleBridge so that the two layers communicate and pass information. To initialize the Ping SDK when the log in view first loads, we call the `frStart()` method on the bridge code.

> **Note**
>
> It's important to initialize the SDK as early as possible. Call this initialization step, so it resolves before any other native SDK methods can be used.

Building the login view

Navigate to the app's login view within the Simulator. You should see an empty screen with a button, since the app doesn't have the data needed to render the form. To render the correct form, retrieve the initial data from the server. This is our first task.

Since most of the action is taking place in `flutter_todo_app/Java/login.dart`, open it and add the following:

1. Import `FRNode.dart` from the Dart helper classes provided for improved ergonomics for handling callbacks:

   ```
   import 'package:flutter_todo_app/FRNode.dart';
   ```

2. If not already there, import `async`, `convert`, `scheduler`, `services` from the `flutter` package. Add the following:

   ```
   import 'dart:async';
   import 'dart:convert';
   import 'package:flutter/scheduler.dart';
   import 'package:flutter/services.dart';
   ```

3. Create a static reference for the method channel

   ```
   MethodChannel('forgerock.com/SampleBridge')
   ```

4. Override the `initState` Flutter lifecycle method and initialize the SDK.

```
class _LoginPageState extends State<LoginPage> {
+  static const platform = MethodChannel('forgerock.com/SampleBridge'); //Method channel as defined in the
native Bridge code

@@ collapsed @@

 //Lifecycle Methods
+  @override
+  void initState() {
+    super.initState();
+    SchedulerBinding.instance?.addPostFrameCallback((_) => {
+      //After creating the first controller that uses the SDK, call the 'frAuthStart' method to initialize
the native SDKs.
+      _startSDK()
+    });
+  }

 // SDK Calls -  Note the promise type responses. Handle errors on the UI layer as required
   Future<void> _startSDK() async {
+    String response;
+    try {
+
+      //Start the SDK. Call the frAuthStart channel method to initialise the native SDKs
+      final String result = await platform.invokeMethod('frAuthStart');
+      response = 'SDK Started';
+      _login();
+    } on PlatformException catch (e) {
+      response = "SDK Start Failed: '${e.message}'.";
+    }
  }

@@ collapsed @@
```

To develop the login functionality, we first need to use the `login()` method from the bridge code to get the first set of callbacks, and then render the form appropriately. This `login()` method is an asynchronous method. Let's get started!

Compose the data gathering process using the following:

1. After the SDK initialization is complete, call the `_login()` method.
2. Use the `platform` reference to call the Bridge login method `platform.invokeMethod('login')`.
3. Parse the response and call `_handleNode()` method.
4. Handle any errors that might be returned from the `Bridge`.

```
@@ collapsed @@

  Future<void> _login() async {
+    try {
+      //Call the default login tree.
+      final String result = await platform.invokeMethod('login');
+      Map<String, dynamic> frNodeMap = jsonDecode(result);
+      var frNode = FRNode.fromJson(frNodeMap);
+      currentNode = frNode;
+
+      //Upon completion, a node with callbacks will be returned, handle that node and present the callbacks
to UI as needed.
+      _handleNode(frNode);
+    } on PlatformException catch (e) {
+      debugPrint('SDK Error: $e');
+      Navigator.pop(context);
+    }
  }
```

The above code is expected to return either a `Node` with a set of `Callback` objects, or a success/error message. We need to handle any exceptions thrown from the bridge on the catch block. Typically, when we begin the authentication journey/tree, this returns a `Node`. Using the `FRNode` helper object, we parse the result in a native Flutter `FRNode` object.

In the next step we are going to "handle" this node, and produce our UI.

```
@@ collapsed @@
  // Handling methods
  void _handleNode(FRNode frNode) {
+    // Go through the node callbacks and present the UI fields as needed. To determine the required UI element,
check the callback type.
+    frNode.callbacks.forEach((frCallback) {
+      final controller = TextEditingController();
+      final field = TextField(
+        controller: controller,
+        obscureText: frCallback.type == "PasswordCallback", // If the callback type is 'PasswordCallback', make this
a 'secure' textField.
+        enableSuggestions: false,
+        autocorrect: false,
+        decoration: InputDecoration(
+          border: OutlineInputBorder(),
+          labelText: frCallback.output[0].value,
+        ),
+      );
+      setState(() {
+        _controllers.add(controller);
+        _fields.add(field);
+      });
+    });
  }
```

The `_handleNode()` method focuses on the `callbacks` property. This property contains instructions about what to render to collect user input.

The previous code processes the Node callbacks and generates two `TextField` objects:

- A `TextField` for the username.
- A `TextField` for the password.

Use the `frCallback.type` to differentiate between the two `TextField` objects and obscure the text of each `TextField`. Next, add the `TextField` objects to the `List` and create the accompanying TextEditingControllers.

Run the app again, and you should see a dynamic form that reacts to the callbacks returned from our initial call to ForgeRock.

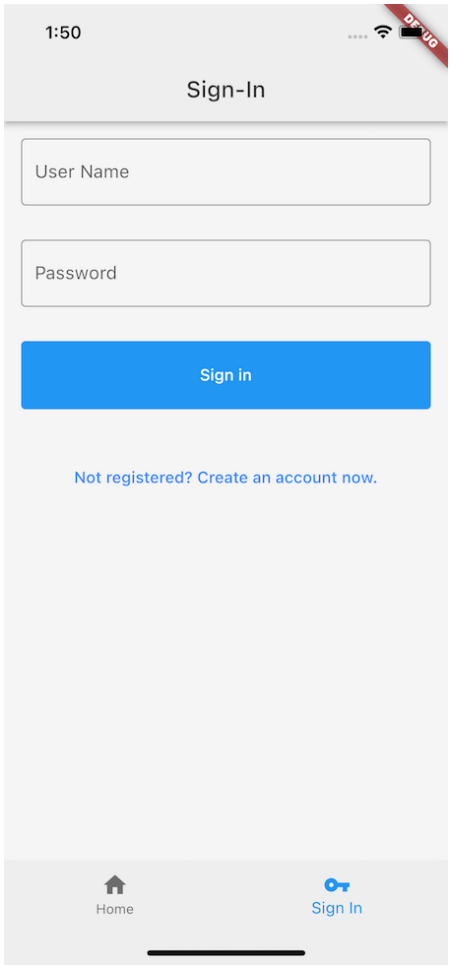


Figure 1. Login screen form

Submitting the login form

Since a form that can't submit anything isn't very useful, we'll now handle the submission of the user input values to ForgeRock. Continuing in `login.dart`, edit the current `_okButton` element, adding an `onPressed` handler calling the `_next()` function. This function should do the following:

1. Go through the `_controllers` array to capture the values of the form elements.
2. Update the `Node callbacks` with those values.
3. Submit the results to ForgeRock.

4. Check the response for a `LoginSuccess` message, or if a new node is returned, handle this in a similar way and resubmit the user inputs as needed.
5. Handle errors with a generic failure message.

```
@@ collapsed @@

Widget _okButton() {
  return Container(
    color: Colors.transparent,
    width: MediaQuery.of(context).size.width,
    margin: EdgeInsets.all(15.0),
    height: 60,
    child: TextButton(
      style: ButtonStyle(backgroundColor: MaterialStateProperty.all(Colors.blue)),
      onPressed: () async {
        showAlertDialog(context);
+       _next();
      },
      child:
      Text(
        "Sign in",
        style: TextStyle(color: Colors.white),
      ),
    ),
  );
}

@@ collapsed @@

 Future<void> _next() async {
   // Capture the User Inputs from the UI, populate the currentNode callbacks and submit back to {am_name}
+  currentNode.callbacks.asMap().forEach((index, frCallback) {
+    _controllers.asMap().forEach((controllerIndex, controller) {
+      if (controllerIndex == index) {
+        frCallback.input[0].value = controller.text;
+      }
+    });
+  });
+  String jsonResponse = jsonEncode(currentNode);
+  try {
+    // Call the SDK next method, to submit the User Inputs to {am_name}. This will return the next Node or a
Success/Failure
+    String result = await platform.invokeMethod('next', jsonResponse);
+    Navigator.pop(context);
+    Map<String, dynamic> response = jsonDecode(result);
+    if (response["type"] == "LoginSuccess") {
+      _navigateToNextScreen(context);
+    } else  {
+      //If a new node is returned, handle this in a similar way and resubmit the user inputs as needed.
+      Map<String, dynamic> frNodeMap = jsonDecode(result);
+      var frNode = FRNode.fromJson(frNodeMap);
+      currentNode = frNode;
+      _handleNode(frNode);
+    }
+  } catch (e) {
+    Navigator.pop(context);
+    debugPrint('SDK Error: $e');
+  }
 }

@@ collapsed @@
```

After the app refreshes, use the test user to login. If successful, you should see a success message. Congratulations, you are now able to authenticate users!

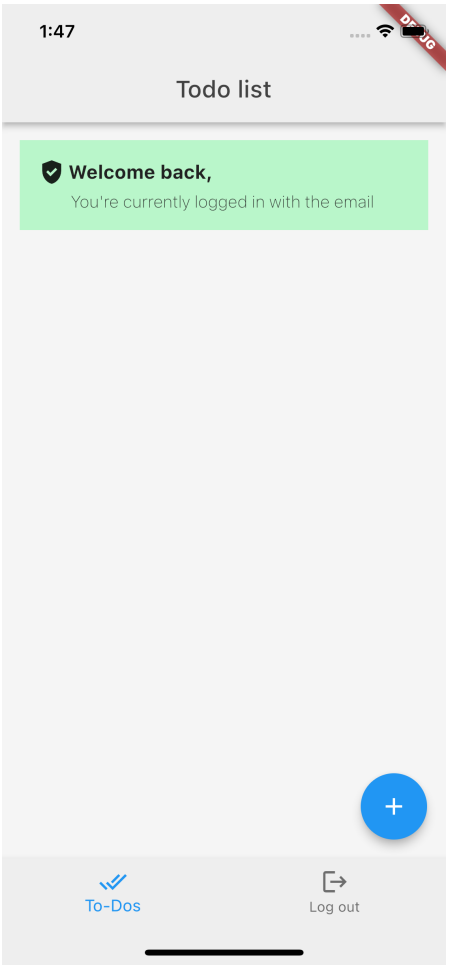


Figure 2. Login screen with successful authentication

What's more, you can verify the authentication details by going to the Xcode or Android Studio log, and observing the result of the last call to the server. It should have a type of `LoginSuccess` with token information.

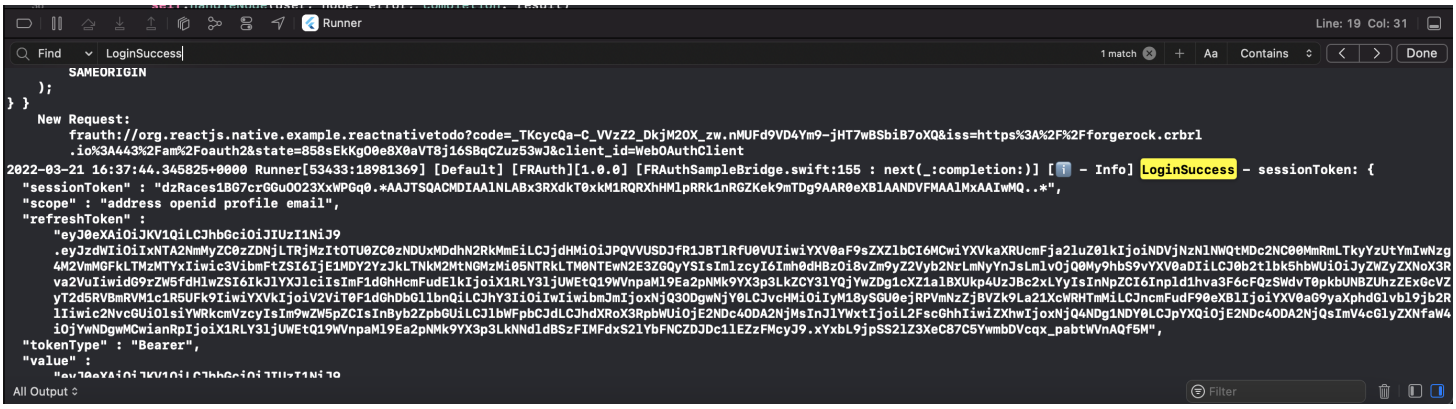


Figure 3. Successful login response from Xcode

Handling the user provided values

You may ask, "How did the user's input values get added to the `Node` object?" Let's take a look at the component for handling the user input submission. Notice how we loop through the `Node Callbacks` and the `_controllers` array. Each input is set on the `frCallback.input[0].value`, and then we call `FRSampleBridge.next()` method.

```
@@ collapsed @@

// Capture the User Inputs from the UI, populate the currentNode callbacks and submit back to {am_name}
  currentNode.callbacks.asMap().forEach((index, frCallback) {
    _controllers.asMap().forEach((controllerIndex, controller) {
      if (controllerIndex == index) {
        frCallback.input[0].value = controller.text;
      }
    });
  });

  String jsonResponse = jsonEncode(currentNode);

@@ collapsed @@

  try {
    // Call the SDK next method, to submit the User Inputs to {am_name}. This will return the next Node or a Success/
Failure
    String result = await platform.invokeMethod('next', jsonResponse);

@@ collapsed @@

  } catch (e) {
    Navigator.pop(context);
    debugPrint('SDK Error: $e');
  }
```

There are two important items to focus on regarding the `FRCallback` object.

`callback.type`

Retrieves the call back `type` so that can identify how to present the callback in the UI.

`callback.input`

The input array that contains the inputs that you need to set the values for.

Since the `NameCallback` and `PasswordCallback` only have one input, you can set the value of them by calling `frCallback.input[0].value = controller.text;`. Some other callbacks might contain multiple inputs, so some extra code will be required to set the values of those.

Each callback type has its own collection of inputs and outputs. Those are exposed as arrays that the developer can loop through and act upon. Many callbacks have common base objects in iOS and Android, like the `SingleValueCallback`, but appear as different types `NameCallback` or `PasswordCallback` to allow for easier differentiation in the UI layer. You can find a full list of the supported callbacks of the SDKs here.

Redirecting to the TodoList screen and requesting user info

Now that the user can log in, let's go one step further and redirect to the TodoList screen. After we get the `LoginSccess` message we can call the `_navigateToNextScreen()` method. This will navigate to the `TodoList` class. When the `TodoList` initializes, we want to request information about the authenticated user to display their name and other information. We will now utilize the existing `FRAuthSampleBridge.getUserInfo()` method already included in the bridge code.

Let's do a little setup before we make the request to the server:

1. Override the `initState()` method in the `_TodoListState` class in `todolist.dart`.
2. Create a `SchedulerBinding.instance?.addPostFrameCallback` to execute some code when the state is loaded.
3. Call `_getUserInfo()`.

```
@@ collapsed @@
//Lifecycle methods

+ @override
+ void initState() {
+   super.initState();
+   SchedulerBinding.instance?.addPostFrameCallback((_) => {
+     //Calling the userinfo endpoint is going to give use some user profile information to enrich our UI.
Additionally, verifies that we have a valid access token.
+     _getUserInfo()
+   });
+ }

@@ collapsed @@
```

With the setup complete, implement the request to your server for the user's information. Within this empty `_getUserInfo()`, add an `async` function to make that call `FRAuthSampleBridge.getUserInfo()` and parse the response.

```
@@ collapsed @@

Future<void> _getUserInfo() async {
  showAlertDialog(context);
  String response;
+  try {
+    final String result = await platform.invokeMethod('getUserInfo');
+    Map<String, dynamic> userInfoMap = jsonDecode(result);
+    response = result;
+    header = userInfoMap["name"];
+    subtitle = userInfoMap["email"];
+    Navigator.pop(context);
+    setState(() {
+      _getTodos();
+    });
+  } on PlatformException catch (e) {
+    response = "SDK Start Failed: '${e.message}'.";
+    Navigator.pop(context);
+  }
+  debugPrint('SDK: $response');
}

@@ collapsed @@
```

In the code above, we collected the user information and set the name and email of the user in some variables. In addition to updating the user info, we will call the `_getTodos()` method in order to retrieve ToDos from the server. Notice that we use the `setState()` function. This ensures that our UI is updated based on the newly received information.

When you test this in the Simulator, completing a successful authentication results in the home screen being rendered with a success message. The user's name and email are included for visual validation. You can also view the console in Xcode and see more complete logs.

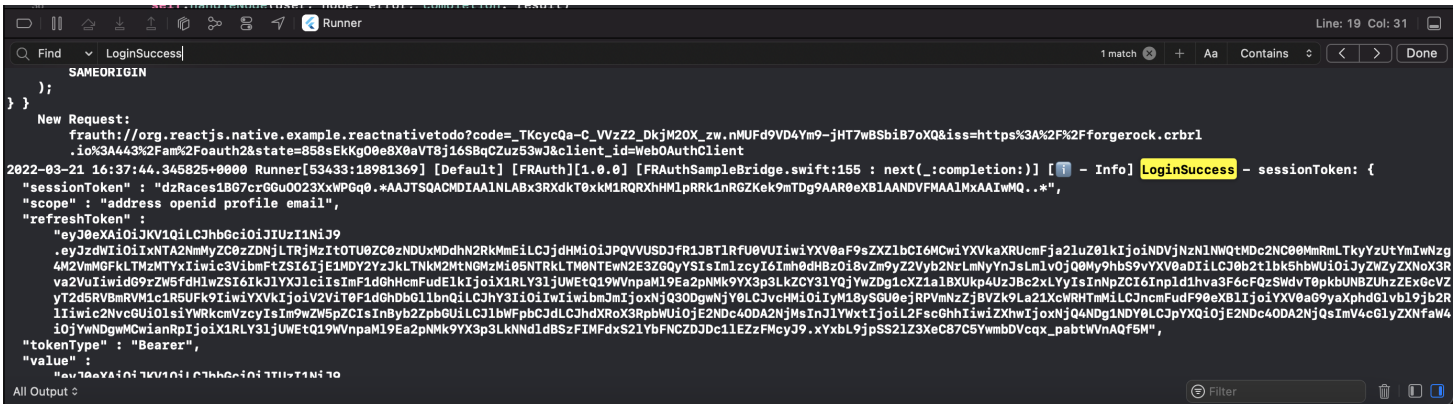


Figure 4. Home screen after successful authentication

Adding logout functionality

Clicking the **Sign Out** button results in creating and rendering an alert view asking you if you are sure you want to log out with two options (yes/no). Clicking **yes** does nothing at the moment. We will now implement that missing logic.

To add the logic into the view to call this new Swift method:

1. Open the `todolist.dart` file, and add the following:

```
@@ collapsed @@

  TextButton(
    child: const Text('Yes'),
+       onPressed: () {
+           Navigator.of(context).pop();
+           _logout();
+       },
  ),

@@ collapsed @@

  Future<void> _logout() async {
+    final String result = await platform.invokeMethod('logout');
+    _navigateToNextScreen(context);
  }
```

2. Revisit the app within the Simulator, and tap the **Sign Out** button.

   This time around when clicking **Yes** will dispose of the alert and log you out, returning you back to the log in screen.

   If you tap **No**, you will return to the main list screen.

Testing the app

You should now be able to successfully authenticate a user, display the user's information, and log a user out.

Congratulations, you just built a protected iOS app with Flutter!

Authentication journey tutorial for ReactJS

In this tutorial you build out a sample ReactJS SPA and make use of a Node.js REST API server sample app.

This guide uses the Ping SDK for JavaScript to implement the following application features:

- Dynamic authentication form for login.
- OAuth/OIDC token acquisition through the Authorization Code Flow with PKCE.
- Protected client-side routing.
- Resource requests to a protected REST API.
- Log out - revoke tokens and end session.

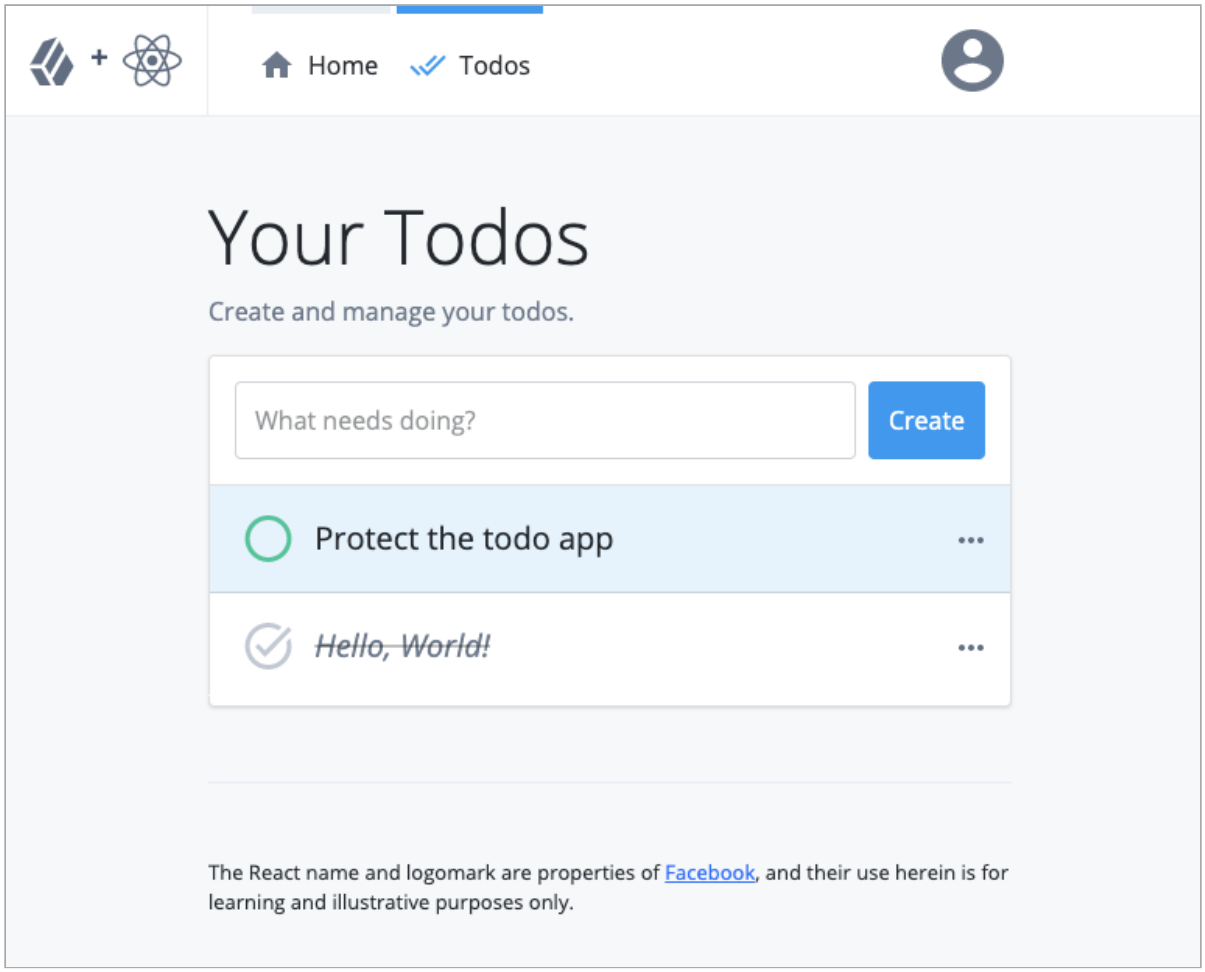


Figure 1. Screenshot of the to-do page of the sample app

Before you begin

Before you begin this tutorial ensure you have set up your PingOne Advanced Identity Cloud tenant or PingAM server with the required configuration.

For example, you will need to configure CORS, have an OAuth 2.0 client application set up, as well as an authentication journey for the app to navigate.

Complete prerequisites »

Step 1. Download the samples

To start this tutorial, you need to download the SDK sample apps repo, which contains the projects you will use.

Start step 1 »

Step 2. Configure connection properties

Configure both the **Todo** client app, and the API backend server app to connect to the OAuth 2.0 application you created in PingOne Advanced Identity Cloud or PingAM.

Start step 2 »

Step 3. Build and run the projects

In this step you build and run the API backend server app, and then the **Todo** client app.

There are also troubleshooting tips if the apps do not start as expected.

Start step 3 »

Step 4. Implement authentication using the Ping SDK

In this step you implement the Ping SDK into the **Todo** client app, so that it authenticates a user and handles the responses from your PingOne Advanced Identity Cloud tenant or PingAM server.

Start step 4 »

Step 5. Start an OAuth 2.0 flow

In this step you use the session token you received in the previous step to start an Oauth 2.0 flow.

Start step 5 »

Step 6. Manage access tokens

In this step you implement code to handle the presence of an access token, and getting user info from the OAuth 2.0 endpoint.

Start step 6 »

Step 7. Handle logout requests

In this step you implement code to terminate the session and revoke tokens.

Start step 7 »

Step 8. Test the app

In this final step you run the completed sample application.

Test it out »

Before you begin

To successfully complete this tutorial refer to the prerequisites in this section.

The tutorial also requires a configured server.

Prerequisites

Node and NPM

The SDK requires a minimum Node.js version of `18`, and is tested on versions `18` and `20`. To get a supported version of Node.js, refer to the Node.js download page.

You will also need `npm` version 7 or newer to build the code and run the samples.

Server configuration

This tutorial requires you to configure one of the following servers:

PingOne Advanced Identity Cloud

PingOne Advanced Identity Cloud

PingAM

PingAM

PingOne Advanced Identity Cloud

Cross-origin resource sharing (CORS) lets user agents make cross-domain server requests. In PingOne Advanced Identity Cloud, you can configure CORS to allow browsers from trusted domains to access PingOne Advanced Identity Cloud protected resources. For example, you might want a custom web application running on your own domain to get an end-user's profile information using the PingOne Advanced Identity Cloud REST API.

The Ping SDK for JavaScript samples and tutorials use `https://localhost:8443` as the host domain, which you should add to your CORS configuration.

If you are using a different domain for hosting SDK applications, ensure you add them to the CORS configuration as accepted origin domains.

For example, for this tutorial you should also add the host domain used by the todo API backend server, which defaults to `http://localhost:9443`.

To update the CORS configuration in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. At the top right of the screen, click your name, and then select **Tenant settings**.
3. On the **Global Settings** tab, click **Cross-Origin Resource Sharing (CORS)**.
4. Perform one of the following actions:
    - If available, click **ForgeRockSDK**.
    - If you haven't added any CORS configurations to the tenant, click **+ Add a CORS Configuration**, select **Ping SDK**, and then click **Next**.
5. Add `https://localhost:8443` and any DNS aliases you use to host your Ping SDK for JavaScript applications to the **Accepted Origins** property.
6. Add the URL used by the todo API backend server, which defaults to `http://localhost:9443`.
7. Complete the remaining fields to suit your environment.

   This documentation assumes the following configuration, required for the tutorials and sample applications:

Property	Values
`Accepted Origins`	`https://localhost:8443` `http://localhost:9443`
`Accepted Methods`	`GET` `POST`

Property	Values

`Accepted Headers`	`accept-api-version`
	`x-requested-with`
	`content-type`
	`authorization`
	`if-match`
	`x-requested-platform`
	`iPlanetDirectoryPro` [1]
	`ch15fefc5407912` [2]

`Exposed Headers`	`authorization` `content-type`
`Enable Caching`	`True`
`Max Age`	`600`
`Allow Credentials`	`True`

> **Tip**
>
> Click **Show advanced settings** to be able to edit all available fields.

8. Click **Save CORS Configuration**.

The samples and tutorials in this documentation often require that you have an identity set up so that you can test authentication.

To create a demo user in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. In the left panel, click **Identities** > **Manage**.
3. Click **+ New Alpha realm - User**.
4. Enter the following details:
    - **Username** = `demo`
    - **First Name** = `Demo`
    - **Last Name** = `User`
    - **Email Address** = `demo.user@example.com`
    - **Password** = `Ch4ng3it!`
5. Click **Save**.

Authentication journeys provide fine-grained authentication by allowing multiple paths and decision points throughout the flow. Authentication journeys are made up of nodes that define actions taken during authentication.

Each node performs a single task, such as collecting a username or making a simple decision. Nodes can have multiple outcomes rather than just success or failure. For details, see the Authentication nodes configuration reference in the PingAM documentation.

To create a simple journey for use when testing the Ping SDKs, follow these steps:

1. In your PingOne Advanced Identity Cloud tenant, navigate to **Journeys**, and click **+ New Journey**.
2. Enter a name, such as `sdkUsernamePasswordJourney` and click **Save**.

   The authentication journey designer appears.
3. Drag the following nodes into the designer area:
   - **Page Node**
   - **Platform Username**
   - **Platform Password**
   - **Data Store Decision**
4. Drag and drop the **Platform Username** and **Platform Password** nodes onto the **Page Node**, so that they both appear on the same page when logging in.
5. Connect the nodes as follows:

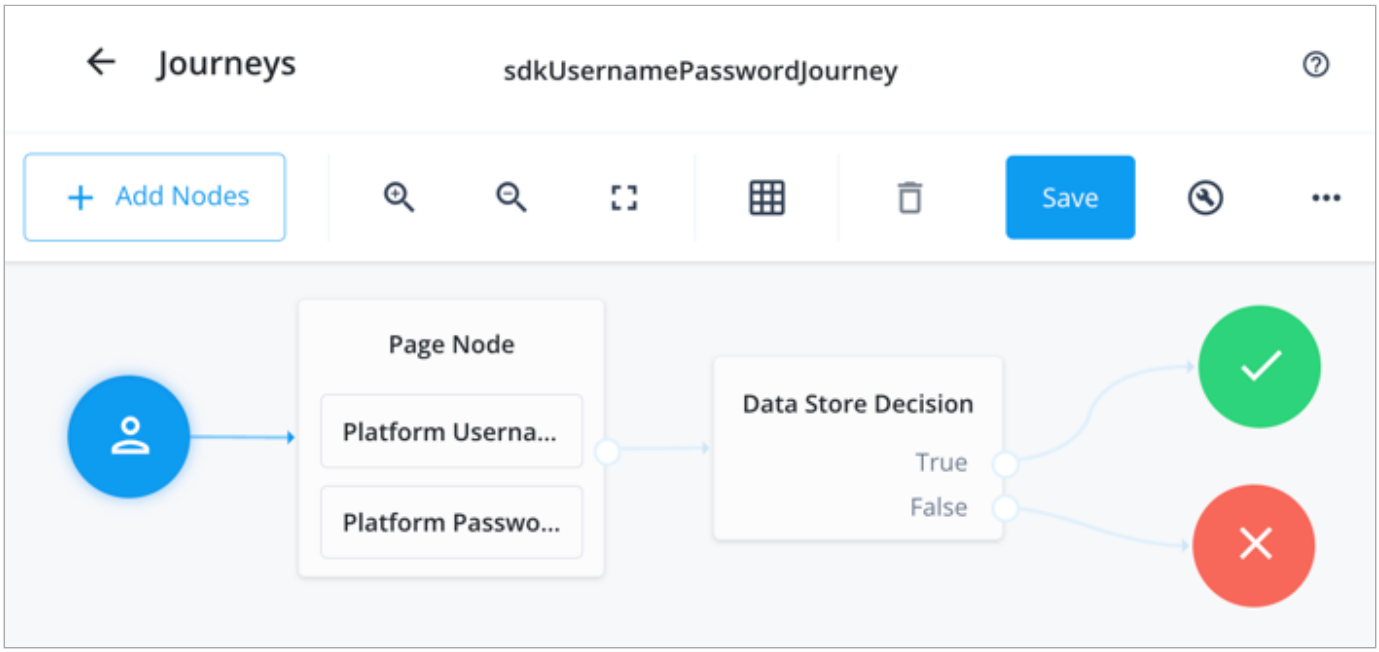


Figure 1. Example username and password authentication journey

6. Click **Save**.

Public clients do not use a client secret to obtain tokens because they are unable to keep them hidden. The Ping SDKs commonly use this type of client to obtain tokens, as they cannot guarantee safekeeping of the client credentials in a browser or on a mobile device.

To register a *public* OAuth 2.0 client application for use with the SDKs in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. In the left panel, click **Applications**.
3. Click **+ Custom Application**.
4. Select **OIDC - OpenId Connect** as the sign-in method, and then click **Next**.
5. Select **Native / SPA** as the application type, and then click **Next**.
6. In **Name**, enter a name for the application, such as `Public SDK Client`.
7. In **Owners**, select a user that is responsible for maintaining the application, and then click **Next**.

   > **Tip**
   >
   > When trying out the SDKs, you could select the `demo` user you created previously.

8. In **Client ID**, enter `sdkPublicClient`, and then click **Create Application**.

   PingOne Advanced Identity Cloud creates the application and displays the details screen.

9. On the **Sign On** tab:

   1. In **Sign-In URLs**, enter the following values:

      `https://localhost:8443/callback`

      > **Important**
      >
      > Also add any other domains where you host SDK applications.

   2. In **Grant Types**, enter the following values:

      `Authorization Code`

      `Refresh Token`

   3. In **Scopes**, enter the following values:

      `openid profile email address`

10. Click Show advanced settings, and on the **Authentication** tab:

    1. In **Token Endpoint Authentication Method**, select `none`.
    2. In **Client Type**, select `Public`.
    3. Enable the **Implied Consent** property.

11. Click **Save**.

The application is now configured to accept client connections from and issue OAuth 2.0 tokens to the example applications and tutorials covered by this documentation.

Confidential clients are able to securely store credentials and are commonly used for server-to-server communication. For example, the "Todo" API backend provided with the SDK samples uses a confidential client to obtain tokens.

The following tutorials and integrations require a *confidential* client:

- Authentication journey tutorial for Angular
- Authentication journey tutorial for ReactJS
- Build advanced token security in a JavaScript SPA

To register a *confidential* OAuth 2.0 client application for use with the SDKs in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. In the left panel, click **Applications**.
3. Click **+ Custom Application**.
4. Select **OIDC - OpenId Connect** as the sign-in method, and then click **Next**.
5. Select **Web** as the application type, and then click **Next**.
6. In **Name**, enter a name for the application, such as `Confidential SDK Client`.
7. In **Owners**, select a user responsible for maintaining the application, and then click **Next**.

   > **Tip**
   >
   > When trying out the SDKs, you could select the `demo` user you created previously.

8. On the **Web Settings** page:
   1. In **Client ID**, enter `sdkConfidentialClient`
   2. In **Client Secret**, enter a strong password and make a note of it for later use.

      > **Important**
      >
      > The client secret is not available to view after this step.
      > If you forget it, you must reset the secret and reconfigure any connected clients.

   3. Click **Create Application**.

      PingOne Advanced Identity Cloud creates the application and displays the details screen.
9. On the **Sign On** tab, click **Show advanced settings**, and on the **Access** tab:
   1. In **Default Scopes**, enter `am-introspect-all-tokens`.
10. Click **Save**.

The provider specifies the supported OAuth 2.0 configuration options for a realm.

To ensure the PingOne Advanced Identity Cloud OAuth 2.0 provider service is configured for use with the Ping SDKs, follow these steps:

1. In your PingOne Advanced Identity Cloud tenant, navigate to **Native Consoles** > **Access Management**.
2. In the left panel, click **Services**.
3. In the list of services, click **OAuth2 Provider**.
4. On the **Core** tab, ensure **Issue Refresh Tokens** is enabled.
5. On the **Consent** tab, ensure **Allow Clients to Skip Consent** is enabled.
6. Click **Save Changes**.

PingAM

Cross-origin resource sharing (CORS) lets user agents make cross-domain server requests. In PingAM, you can configure CORS to allow browsers from trusted domains to access PingAM protected resources. For example, you might want a custom web application running on your own domain to get an end-user's profile information using the PingAM REST API.

The Ping SDK for JavaScript samples and tutorials all use `https://localhost:8443` as the host domain, which you should add to your CORS configuration.

If you are using a different URL for hosting SDK applications, ensure you add them to the CORS configuration as accepted origin domains.

For example, for this tutorial you should also add the host domain used by the todo API backend server, which defaults to `http://localhost:9443`.

To enable CORS in PingAM, and create a CORS filter to allow requests from your configured domain names, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. Navigate to **Configure** > **Global Services** > **CORS Service** > **Configuration**, and set the **Enable the CORS filter** property to `true`.

   > **Important**
   >
   > If this property is not enabled, CORS headers are not added to responses from PingAM, and CORS is disabled entirely.

3. On the **Secondary Configurations** tab, click **Click Add a Secondary Configuration**.
4. In the **Name** field, enter `ForgeRockSDK`.
5. in the **Accepted Origins** field, enter any DNS aliases you use for your SDK apps.

   This documentation assumes the following configuration:

Property	Values
`Accepted Origins`	`https://localhost:8443` `http://localhost:9443`
`Accepted Methods`	`GET` `POST`

`Accepted Headers`	`accept-api-version`
	`x-requested-with`
	`content-type`
	`authorization`
	`if-match`
	`x-requested-platform`
	`iPlanetDirectoryPro` [1]
	`ch15fefc5407912` [2]

`Exposed Headers`	`authorization` `content-type`

6. Click **Create**.

   PingAM displays the configuration of your new CORS filter.

7. On the CORS filter configuration page:

   1. Ensure **Enable the CORS filter** is enabled.
   2. Set the **Max Age** property to `600`
   3. Ensure **Allow Credentials** is enabled.

8. Click **Save Changes**.

The samples and tutorials in this documentation often require that you have an identity set up so that you can test authentication.

To create a demo user in PingAM, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. Navigate to **Identities**, and then click **+ Add Identity**.
3. Enter the following details:
   - **User ID** = `demo`
   - **Password** = `Ch4ng3it!`
   - **Email Address** = `demo.user@example.com`
4. Click **Create**.

Authentication trees provide fine-grained authentication by allowing multiple paths and decision points throughout the authentication flow. Authentication trees are made up of nodes that define actions taken during authentication.

Each node performs a single task, such as collecting a username or making a simple decision. Nodes can have multiple outcomes rather than just success or failure. For details, see the Authentication nodes configuration reference in the PingAM documentation.

To create a simple tree for use when testing the Ping SDKs, follow these steps:

1. Under **Realm Overview**, click **Authentication Trees**, then click **Create Tree**.
2. Enter a tree name, for example `sdkUsernamePasswordJourney`, and then click **Create**.

   The authentication tree designer appears, showing the **Start** entry point connected to the **Failure** exit point.
3. Drag the following nodes from the **Components** panel on the left side into the designer area:
   - **Page Node**
   - **Username Collector**
   - **Password Collector**
   - **Data Store Decision**
4. Drag and drop the **Username Collector** and **Password Collector** nodes onto the **Page Node**, so that they both appear on the same page when logging in.
5. Connect the nodes as follows:

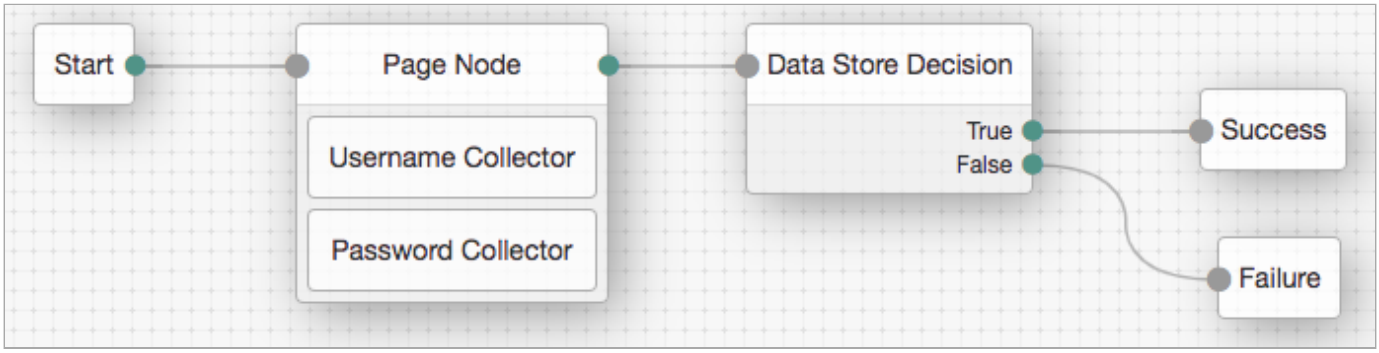


Figure 2. Example username and password authentication tree

6. Select the **Page Node**, and in the **Properties** pane, set the **Stage** property to `UsernamePassword`.

   > **Tip**
   >
   > You can configure the node properties by selecting a node and altering properties in the right-hand panel.

   One of the samples uses this specific value to determine the custom UI to display.
7. Click **Save**.

Public clients do not use a client secret to obtain tokens because they are unable to keep them hidden. The Ping SDKs commonly use this type of client to obtain tokens, as they cannot guarantee safekeeping of the client credentials in a browser or on a mobile device.

To register a *public* OAuth 2.0 client application for use with the SDKs in AM, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. Navigate to **Applications** > **OAuth 2.0** > **Clients**, and then click **+ Add Client**.
3. In **Client ID**, enter `sdkPublicClient`.
4. Leave **Client secret** empty.
5. In **Redirection URIs**, enter the following values:

   `https://localhost:8443/callback`

   > **Important**
   >
   > Also add any other domains where you will be hosting SDK applications.

6. In **Scopes**, enter the following values:

   `openid profile email address`

7. Click **Create**.

   PingAM creates the new OAuth 2.0 client, and displays the properties for further configuration.

8. On the **Core** tab:
   1. In **Client type**, select `Public`.
   2. Disable **Allow wildcard ports in redirect URIs**.
   3. Click **Save Changes**.
9. On the **Advanced** tab:
   1. In **Grant Types**, enter the following values:

      ```
      Authorization Code
      Refresh Token
      ```

   2. In **Token Endpoint Authentication Method**, select `None`.
   3. Enable the **Implied consent** property.
10. Click **Save Changes**.

Confidential clients are able to store credentials securely and are commonly used for server-to-server communication.

The following tutorials and integrations require a *confidential* client:

- Authentication journey tutorial for Angular
- Authentication journey tutorial for ReactJS
- Build advanced token security in a JavaScript SPA

To register a *confidential* OAuth 2.0 client application for use with the SDKs in AM, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. Navigate to **Applications** > **OAuth 2.0** > **Clients**, and then click **+ Add Client**.
3. In **Client ID**, enter `sdkConfidentialClient`.
4. In **Client Secret**, enter a strong password and make a note of it for later use.

   > **Important**
   >
   > The client secret is not available to view after this step.
   > If you forget it, you must reset the secret and reconfigure any connected clients.

5. In **Default Scopes**, enter `am-introspect-all-tokens`.

   PingAM creates the new OAuth 2.0 client and displays the properties for further configuration.
6. On the **Advanced** tab:
   1. Enable the **Implied consent** property.
7. Click **Save Changes**.

The provider specifies the supported OAuth 2.0 configuration options for a realm.

To ensure the PingAM OAuth 2.0 provider service is configured for use with the Ping SDKs, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. In the left panel, click **Services**.
3. In the list of services, click **OAuth2 Provider**.
4. On the **Core** tab, ensure **Issue Refresh Tokens** is enabled.
5. On the **Consent** tab, ensure **Allow Clients to Skip Consent** is enabled.
6. Click **Save Changes**.

---

1. Cookie name value in PingAM servers.
2. In PingOne Advanced Identity Cloud tenants, go to **Tenant Settings > Global Settings > Cookie** to find this dynamic cookie name value.

Step 1. Download the samples

To start this tutorial, you need to download the SDK sample apps repo, which contains the projects you will use.

Within the repo are two branches related to this tutorial:

`build-protected-app/start`

Contains all the source files you need to follow this tutorial, but without the actual implementation of the Ping SDK functionality.

Use this branch if you want to complete the tutorial step-by-step, adding the code the tutorial provides.

`build-protected-app/complete`

The same source files but with the Ping SDK code already implemented.

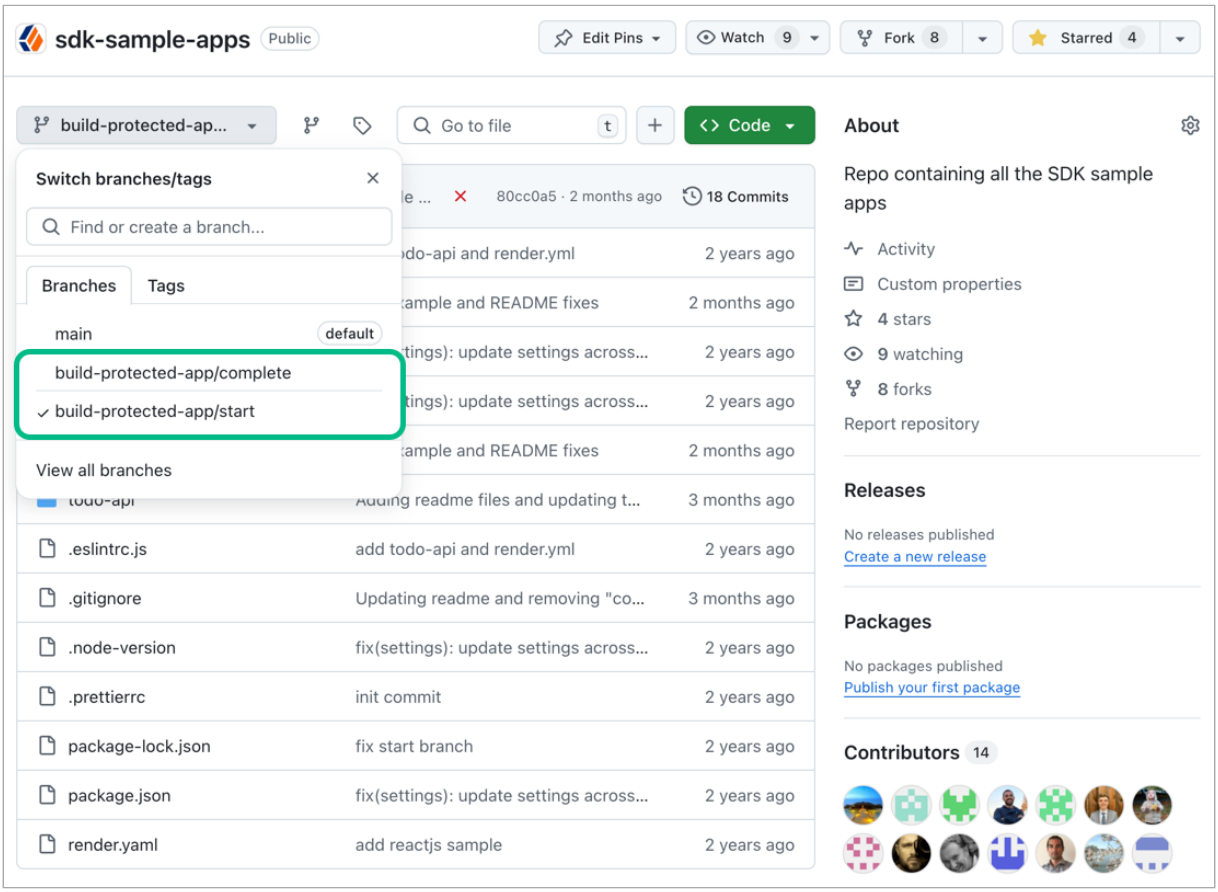
Use this branch if you want to skip ahead of the tutorial, or if you want to compare your work with the completed version for troubleshooting.

To get a copy of the tutorial source code:

1. In a web browser, navigate to the SDK Sample Apps repository.
2. Download the source code using one of the following methods:

Download a ZIP file

1. Select which branch to download:



2. Click **Code**, and then click **Download ZIP**.
3. Extract the contents of the downloaded ZIP file to a suitable location.

Use a Git-compatible tool to clone the repo locally

1. Click **Code**, and then copy the HTTPS URL.
2. Use the URL to clone the repository to a suitable location.

For example, from the command-line you could run:

```
git clone https://github.com/ForgeRock/sdk-sample-apps.git
```

3. Checkout which branch you want to work on.

   For example, from the command-line you could run:

```
git checkout build-protected-app/start
```

The result of these steps is a local folder named `sdk-sample-apps`.

Step 2. Configure connection properties

There are two projects in this tutorial that require configuration:

Client ReactJS app

The front-end client app, written in React, that handles the UI and authentication journeys.

Backend API server

A backend REST API server that uses a confidential OAuth 2.0 client to contact the authorization server. The API server handles storage and retrieval of your personal "Todo" items.

Configure the React client app

Copy the `.env.example` file in the `sdk-sample-apps/reactjs-todo` folder and save it with the name `.env` within this same directory.

Add your relevant values to this new file because it provides all the important configuration settings to your applications.

Example client `sdk-sample-apps/reactjs-todo/.env` file

```
API_URL=http://localhost:9443
DEBUGGER_OFF=true
DEVELOPMENT=true
JOURNEY_LOGIN=sdkUsernamePasswordJourney
JOURNEY_REGISTER=Registration
PORT=8443
WEB_OAUTH_CLIENT=sdkPublicClient
WELLKNOWN_URL=https://openam-forgerock-sdks.forgeblocks.com/am/oauth2/realms/root/realms/alpha/.well-known/openid-configuration
```

Here are descriptions for some of the values:

`DEBUGGER_OFF`

Set to `true`, to disable `debug` statements in the app.

These statements are for learning the integration points at runtime in your browser.

When you open the browser's developer tools, the app pauses at each integration point. Code comments are placed above each one explaining their use.

`DEVELOPMENT`

When `true`, this provides better debugging during development.

`JOURNEY_LOGIN`

The simple login journey or tree you created earlier, for example `sdkUsernamePasswordJourney`.

`JOURNEY_REGISTER`

The registration journey or tree.

You can use the default builtin `Registration` journey.

`WELLKNOWN_URL`

The URL to your server's `.well-known/openid-configuration` endpoint.

Example:

`https://openam-forgerock-sdks.forgeblocks.com/am/oauth2/realms/root/realms/alpha/.well-known/openid-configuration`

Self-hosted example:

`https://openam.example.com:8443/openam/oauth2/realms/root/.well-known/openid-configuration`

Configure the API server app

Copy the `.env.example` file in the `sdk-sample-apps/todo-api` folder and save it with the name `.env` within this same directory.

Add your relevant values to this new file as it will provide all the important configuration settings to your applications.

Example API server `sdk-sample-apps/todo-api/.env` file

```
AM_URL=https://openam-forgerock-sdks.forgeblocks.com/am
DEVELOPMENT=true
SERVER_PORT=9443
REALM_PATH=alpha
REST_OAUTH_CLIENT=sdkConfidentialClient
REST_OAUTH_SECRET=ch4ng3it!
```

Step 3. Build and run the projects

In this step you build and run the API backend, and the "Todo" client app project.

1. Open a terminal window at the root of the `sdk-sample-apps` directory and install the dependencies using the `npm install` command:

   ```
   npm install
   ```

2. In the same directory run the following command to start both the API backend server and the "Todo" client:

   ```
   npm run start:reactjs-todo
   ```

3. In a *different* browser than the one you are using to administer the server, visit the following URL: `https://localhost:8443`.

   The app renders a home page explaining the purpose of the project:

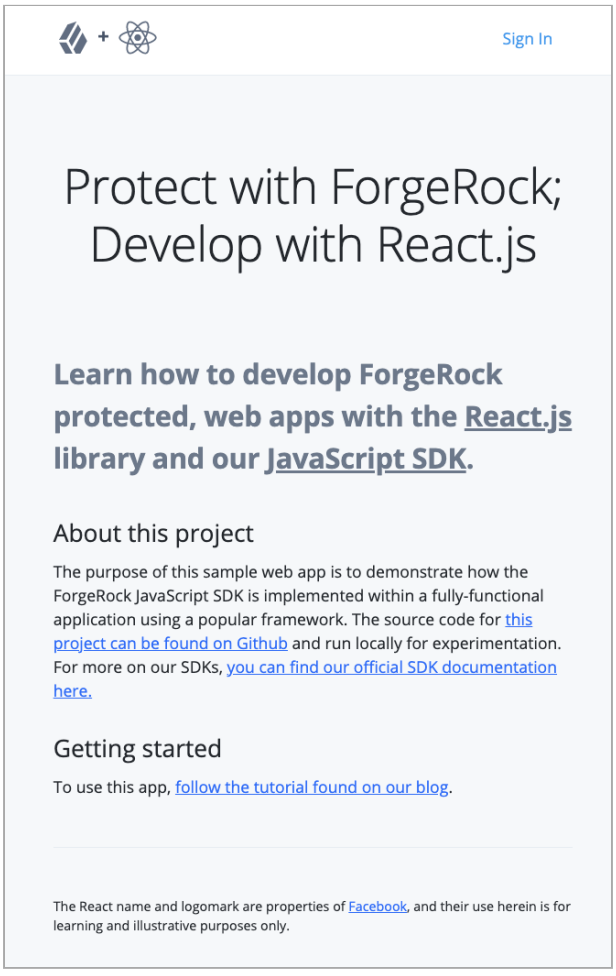


Figure 1. Screenshot of the home page of the sample app

> **Note**
>
> Only the home page renders successfully. The login page functionality is not yet functional. You will develop this functionality later in this tutorial.

Troubleshooting

If the home page doesn't render due to errors, here are a few tips:

- Visit `http://localhost:9443/healthcheck` in the same browser you use for the ReactJS app to test the API backend is running. The API backend should respond with `Ok`.
- Look for error output in the terminal that is running the `npm run start:reactjs-todo` command.
- Ensure you are not logged into your PingOne Advanced Identity Cloud tenant or PingAM server in the same browser as the sample app; log out if you are, or use a different browser or an incognito window.

Step 4. Implement authentication using the Ping SDK

Now that we have our environment and servers setup, let's jump into the application! Within your IDE of choice, navigate to the `reactjs-todo/client` directory. This directory is where you will spend the rest of your time.

First, open up the `index.js` file, import the `Config` module from the Ping SDK for JavaScript and call the `setAsync()` method on this object:

/reactjs-todo/client/index.js

```
+ import { Config } from '@forgerock/javascript-sdk';
  import React from 'react';
  import { createRoot } from 'react-dom/client';

  import Router from './router';
  import { WELLKNOWN_URL, APP_URL, JOURNEY_LOGIN, WEB_OAUTH_CLIENT } from './constants';
  import { AppContext, useGlobalStateMgmt } from './global-state';

  import './styles/index.scss';

+ const urlParams = new URLSearchParams(window.location.search);
+ const journeyParam = urlParams.get('journey');
+
+ await Config.setAsync();

  /**
  * Initialize the React application
  * This is an IIFE (Immediately Invoked Function Expression),
  * so it calls itself.
  */
  (async function initAndHydrate() {

@@ collapsed @@
```

The use of `setAsync()` should always be the first SDK method called and is frequently done at the application's top-level file. To configure the SDK to communicate with the journeys, OAuth clients, and realms of the appropriate server, pass a configuration object with the appropriate values.

The configuration object you will use in this instance will pull most of its values out of the `.env` variables previously setup, which are mapped to constants within our `constants.js` file.

Here's an example config for an PingOne Advanced Identity Cloud tenant:

/reactjs-todo/client/index.js

```
  import { Config } from '@forgerock/javascript-sdk';
  import React from 'react';
  import { createRoot } from 'react-dom/client';

  import Router from './router';
  import { WELLKNOWN_URL, APP_URL, JOURNEY_LOGIN, WEB_OAUTH_CLIENT } from './constants';
  import { AppContext, useGlobalStateMgmt } from './global-state';

  import './styles/index.scss';

  const urlParams = new URLSearchParams(window.location.search);
  const journeyParam = urlParams.get('journey');

  await Config.setAsync(
+   {
+     clientId: WEB_OAUTH_CLIENT,
+     redirectUri: `${window.location.origin}/callback`,
+     scope: 'openid profile email address',
+     serverConfig: {
+       wellknown: WELLKNOWN_URL,
+       timeout: 3000,
+     },
+     tree: `${journeyParam || JOURNEY_LOGIN}`,
+   }
  );

@@ collapsed @@
```

Go back to your browser and refresh the home page. There should be no change to what's rendered, and no errors in the console. Now that the app is configured to your server, let's wire up the simple login page.

Building the login page

First, let's review how the application renders the home page:

`index.js` > `router.js` > `views/home.js` > inline code + components (`components/`)

For the login page, the same pattern applies except it has less code within the view file:

`index.js` > `router.js` > `views/login.js` > `components/journey/form.js`

In the top-right of the home page, click **Sign In** to open the login page.

You should see a "loading" spinner and message that's persistent since it doesn't have the callbacks from your server that are needed to render the form. Obtaining these callbacks is the first task.

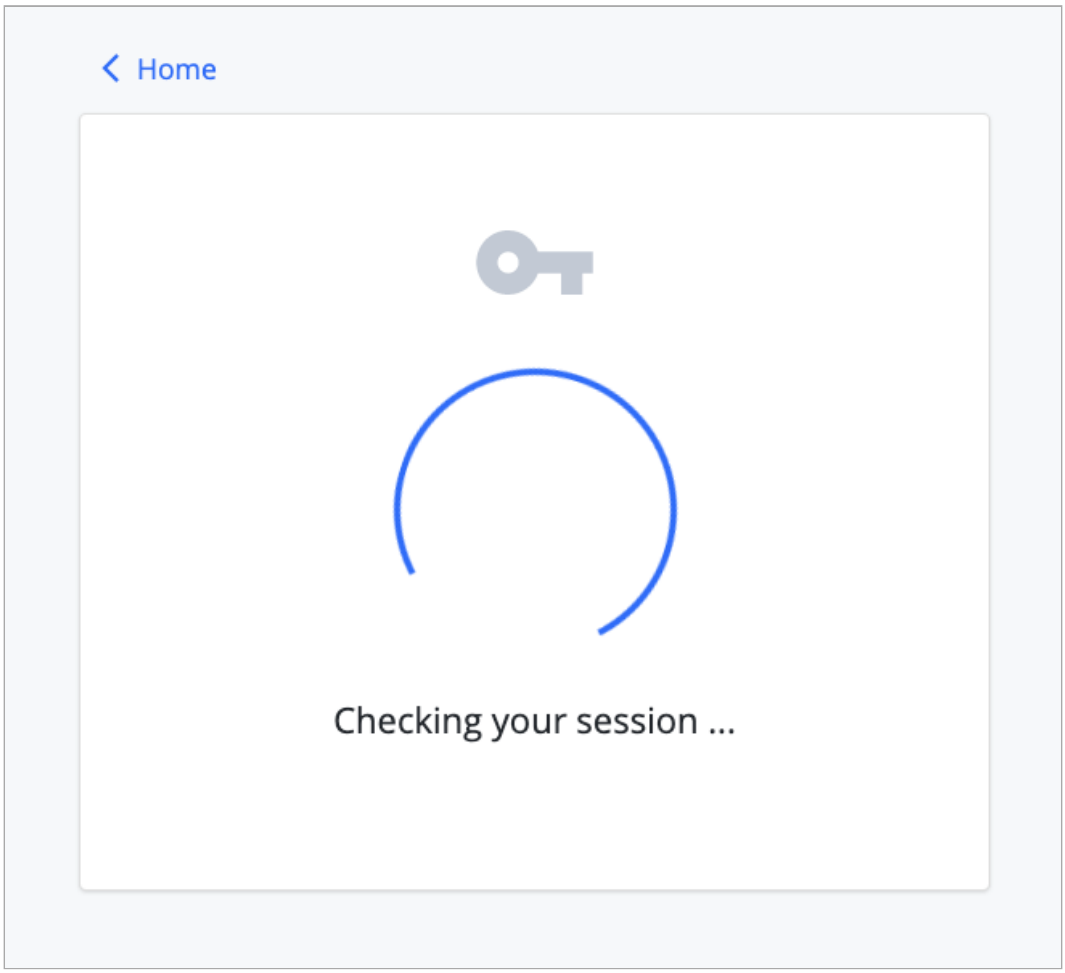


Figure 1. Screenshot of the todo app's login page with spinner.

Since most of the action is taking place in `reactjs-todo/client/components/journey/form.js`, open it and add the `FRAuth` module from the Ping SDK for JavaScript:

reactjs-todo/client/components/journey/form.js

```
+ import { FRAuth } from '@forgerock/javascript-sdk';
import React from 'react';

import Loading from '../utilities/loading';

@@ collapsed @@
```

`FRAuth` is the first object used as it provides the necessary methods for authenticating a user by using an authentication journey or tree. Use the `start()` method of `FRAuth` as it returns data we need for rendering the form.

You will need to add new imports. Add `useContext` and `useState` from the **React** package.

You'll use the `useState()` method for managing the data received from the server, and the `useEffect` is needed due to the `FRAuth.start()` method resulting in a network request.

reactjs-todo/client/components/journey/form.js

```
  import { FRAuth } from '@forgerock/javascript-sdk';
- import React from 'react';
+ import React, { useEffect, useState } from 'react';

  import Loading from '../utilities/loading';

  export default function Form() {
+   const [step, setStep] = useState(null);

+   useEffect(() => {
+     async function getStep() {
+       try {
+         const initialStep = await FRAuth.start();
+         console.log(initialStep);
+         setStep(initialStep);
+       } catch (err) {
+         console.error(`Error: request for initial step; ${err}`);
+       }
+     }
+     getStep();
+   }, []);
    return <Loading message="Checking your session ..." />;
  }
```

> **Note**
>
> We are passing an empty array as the second argument into `useEffect`. This instructs the `useEffect` to only run once after the component mounts.
> To learn more, refer to What an Effect with empty dependencies means in the React Developer Documentation.

This code prints the response to starting the journey to the debug console in the browser. This response contains the first step of the journey and its `callbacks`. These callbacks are the instructions for what needs to be rendered to the user to collect their input.

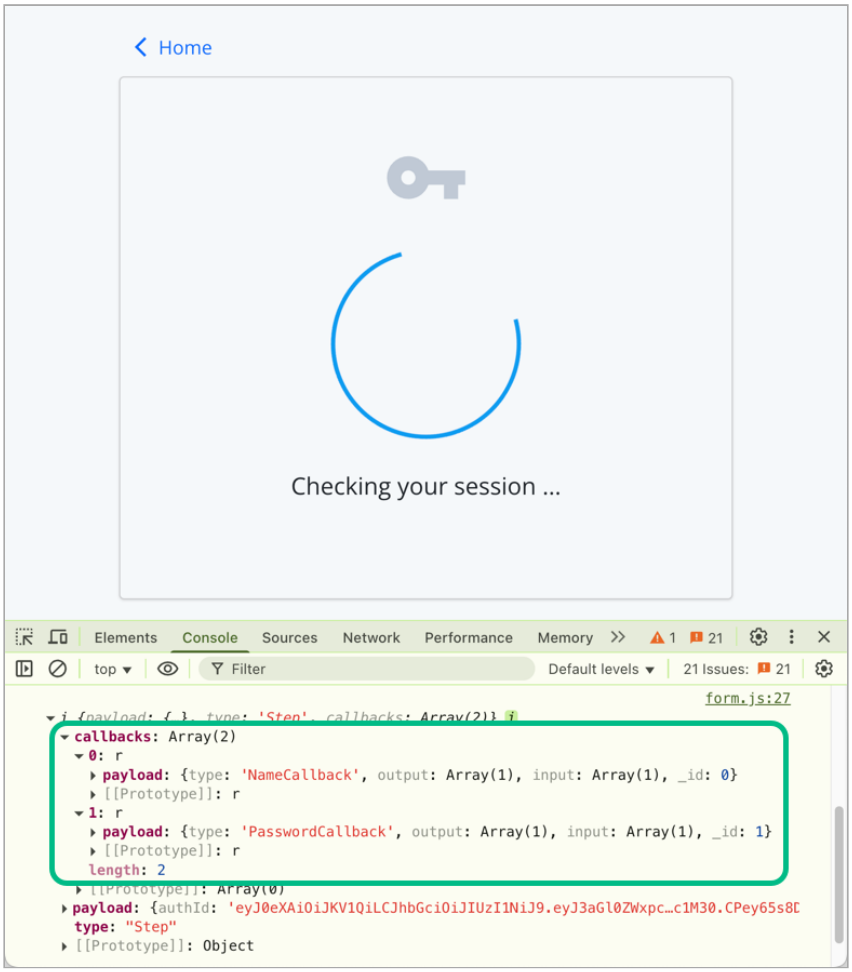


Figure 2. Screenshot of browser console showing first step of the journey and the `callbacks` returned from the server.

Below is a summary of what you'll do to get the form to react to the new callback data:

1. Import the needed form-input components
2. Create a function to map received `callbacks` to the appropriate component
3. Use the components to render the appropriate UI for each callback in the response from the server

First, import the `Alert`, `AppContext`, `Password`, `Text`, and `Unknown` components.

reactjs-todo/client/components/journey/form.js

```
  import { FRAuth } from '@forgerock/javascript-sdk';
  import React, { useEffect, useState } from 'react';

+ import Alert from './alert';
+ import Password from './password';
+ import Text from './text';
+ import Unknown from './unknown';
  import Loading from '../utilities/loading';

@@ collapsed @@
```

Next, within the `Form` function body, create the function that maps these imported components to their appropriate callback.

reactjs-todo/client/components/journey/form.js

```
@@ collapsed @@

  export default function Form() {
    const [step, setStep] = useState(null);

@@ collapsed @@

+   function mapCallbacksToComponents(cb, idx) {
+     const name = cb?.payload?.input?.[0].name;
+     switch (cb.getType()) {
+       case 'NameCallback':
+         return <Text callback={cb} inputName={name} key='username' />;
+       case 'PasswordCallback':
+         return <Password callback={cb} inputName={name} key='password' />;
+       default:
+         // If current callback is not supported, render a warning message
+         return <Unknown callback={cb} key={`unknown-${idx}`} />;
+     }
+   }
    return <Loading message="Checking your session ..." />;
  }
```

Finally, check for the presence of the `step.callbacks`, and if they exist, map over them with the function from above. Replace the single return of `<Loading message="Checking your session ..." />` with the following:

reactjs-todo/client/components/journey/form.js

```
@@ collapsed @@

+   if (!step) {
      return <Loading message='Checking your session ...' />;
+   } else if (step.callbacks?.length) {
+     return (
+       <form className='cstm_form'>
+         {step.callbacks.map(mapCallbacksToComponents)}
+         <button className='btn btn-primary w-100' type='submit'>
+           Sign In
+         </button>
+       </form>
+     );
+   } else {
+     return <Alert message={step.payload.message} />;
*   }
  }
```

Refresh the page, and you should now have a dynamic form that reacts to the callbacks returned from our initial call to ForgeRock.

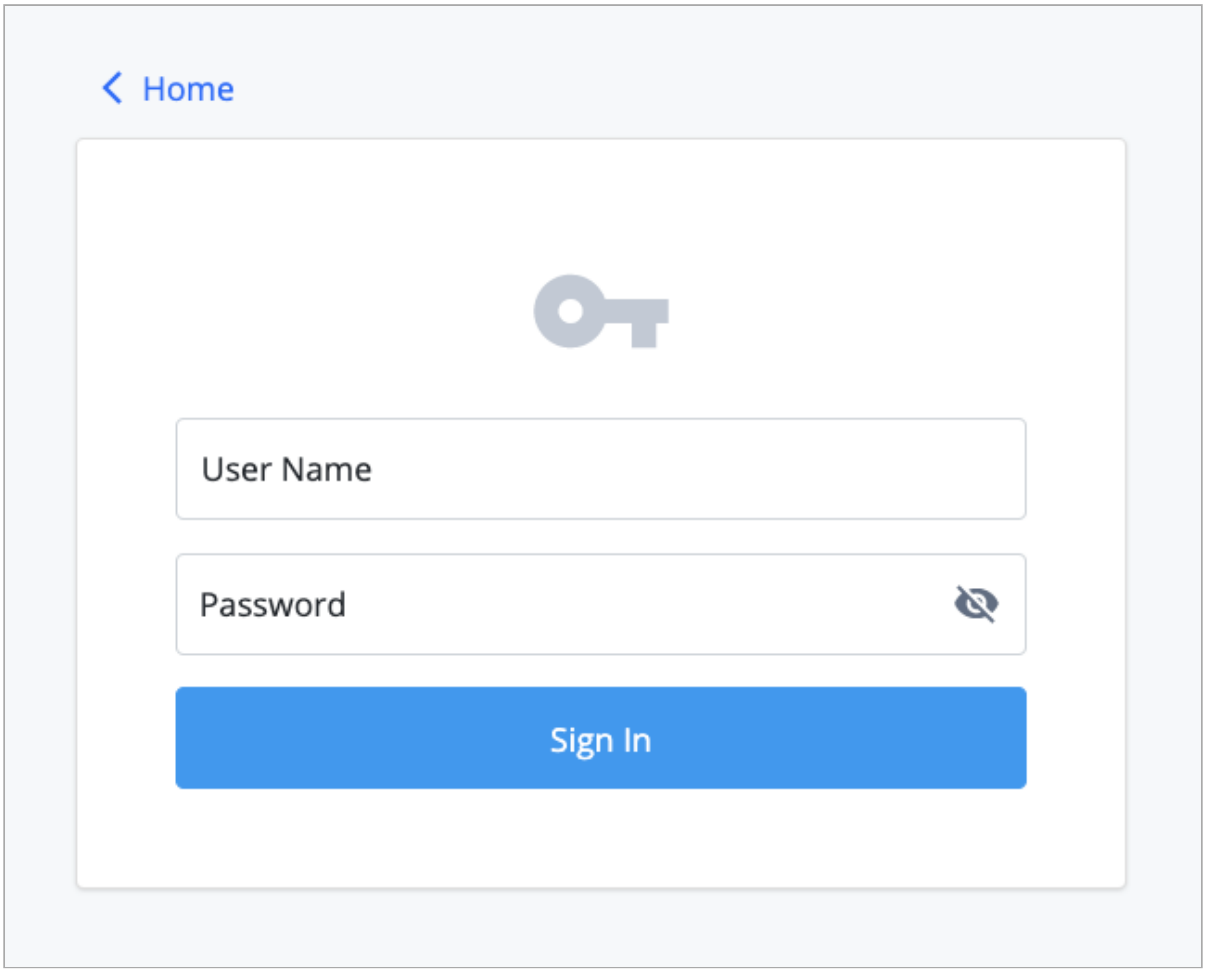


Figure 3. Screenshot of login page with rendered form

Handling the login form submission

Since a form that can't submit anything isn't very useful, we'll now handle the submission of the user input values to ForgeRock. First, let's edit the current form element, `<form className="cstm_form">`, and add an `onSubmit` handler with a simple, inline function.

reactjs-todo/client/components/journey/form.js

```
@@ collapsed @@

- <form className='cstm_form'>
+ <form
+   className="cstm_form"
+   onSubmit={(event) => {
+     event.preventDefault();
+     async function getStep() {
+       try {
+         const nextStep = await FRAuth.next(step);
+         console.log(nextStep);
+         setStep(nextStep);
+       } catch (err) {
+         console.error(`Error: form submission; ${err}`);
+       }
+     }
+     getStep();
+   }}
+ >
```

Refresh the login page and use the test user to login. You will get a mostly blank login page if the user's credentials are valid and the journey completes. You can verify this by going to the Network panel within the developer tools and inspect the last `/authenticate` request. It should have a `tokenId` and `successUrl` property.

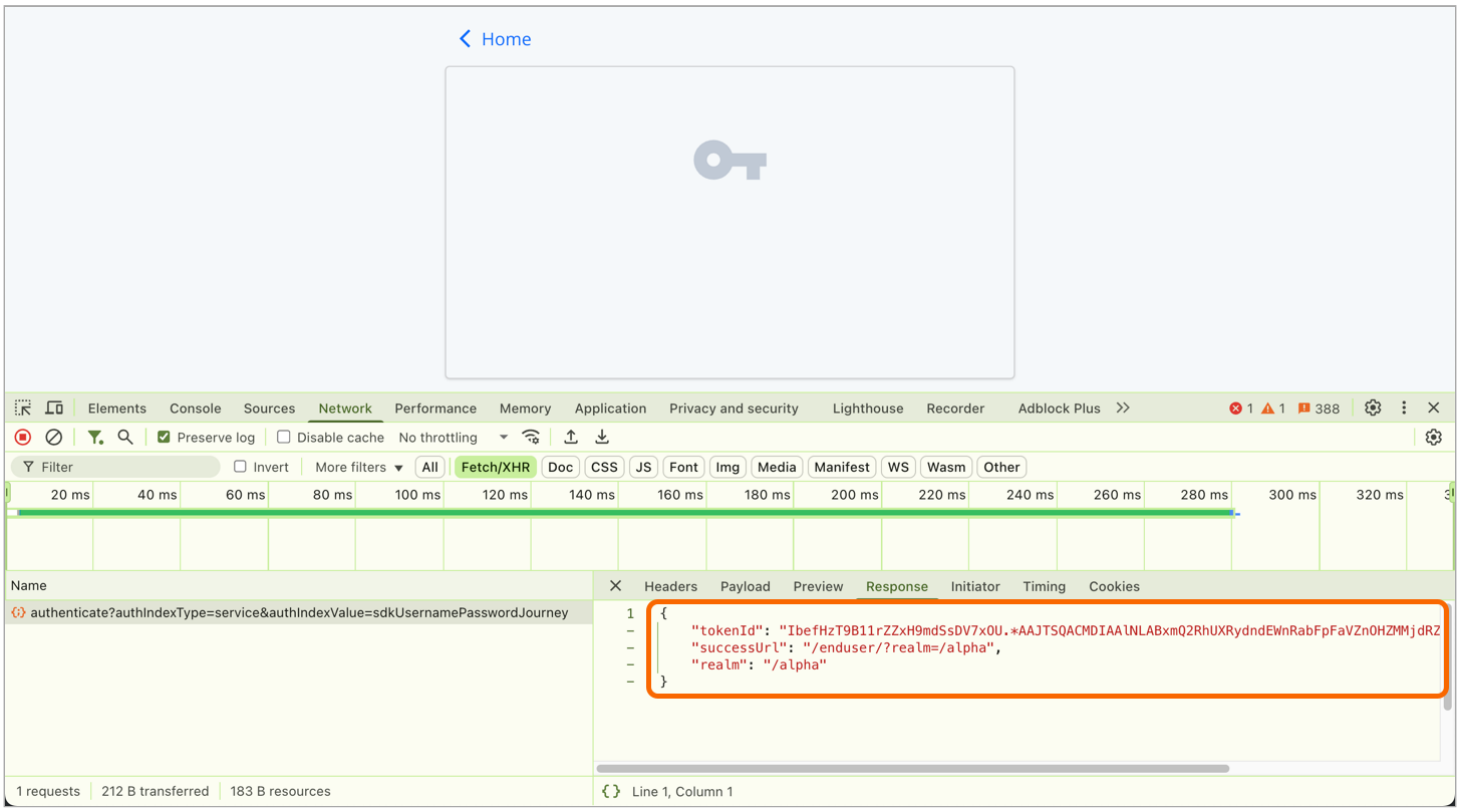


Figure 4. Screenshot of empty login form & network request showing success data

You may ask, "How did the user's input values get added to the `step` object?" Let's take a look at the component for rendering the username input. Open up the `Text` component: `components/journey/text.js`. Notice how special methods are being used on the callback object. These are provided as convenience methods by the SDK for getting and setting data.

reactjs-todo/client/components/journey/text.js

```
@@ collapsed @@

  export default function Text({ callback, inputName }) {
    const [state] = useContext(AppContext);
    const existingValue = callback.getInputValue();

    const textInputLabel = callback.getPrompt();
    function setValue(event) {
      callback.setInputValue(event.target.value);
    }

    return (
      <div className={`cstm_form-floating form-floating mb-3`}>
        <input
          className={`cstm_form-control form-control ${validationClass} bg-transparent ${state.theme.textClass} $
{state.theme.borderClass}`}
          defaultValue={existingValue}
          id={inputName}
          name={inputName}
          onChange={setValue}
          placeholder={textInputLabel}
        />
        <label htmlFor={inputName}>{textInputLabel}</label>
      </div>
    );
  }
```

The two important items to focus on are the `callback.getInputValue()` and the `callback.setInputValue()`. The `getInputValue` retrieves any existing value that may be provided by ForgeRock, and the `setInputValue` sets the user's input on the callback while they are typing (i.e. `onChange`). Since the `callback` is passed from the `Form` to the components by "reference" (not by "value"), any mutation of the `callback` object within the `Text` (or `Password`) component is also contained within the `step` object in `Form`.

> **Note**
>
> You may think, "That's not very idiomatic React! Shared, mutable state is bad." And, yes, you are correct, but we are taking advantage of this to keep everything simple (and this guide from being too long), so I hope you can excuse the pattern.

Each callback type has its own collection of methods for getting and setting data in addition to a base set of generic callback methods. These methods are added to the callback prototype by the SDK automatically. For more information about these callback methods, see our API documentation, or the source code in GitHub, for more details.

Now that the form is rendering and submitting, add conditions to the `Form` component for handling the success and error response from ForgeRock. This condition handles the success result of the authentication journey.

reactjs-todo/client/components/journey/form.js

```
@@ collapsed @@

  if (!step) {
    return <Loading message='Checking your session ...' />;
+ } else if (step.type === 'LoginSuccess') {
+   return <Alert message="Success! You're logged in." type='success' />;
  } else if (step.callbacks?.length) {

@@ collapsed @@
```

Once you handle the success and error condition, return to the browser and remove all cookies created from any previous logins. Refresh the page and login with your test user created in the Setup section above. You should see a "Success!" alert message. Congratulations, you are now able to authenticate users!

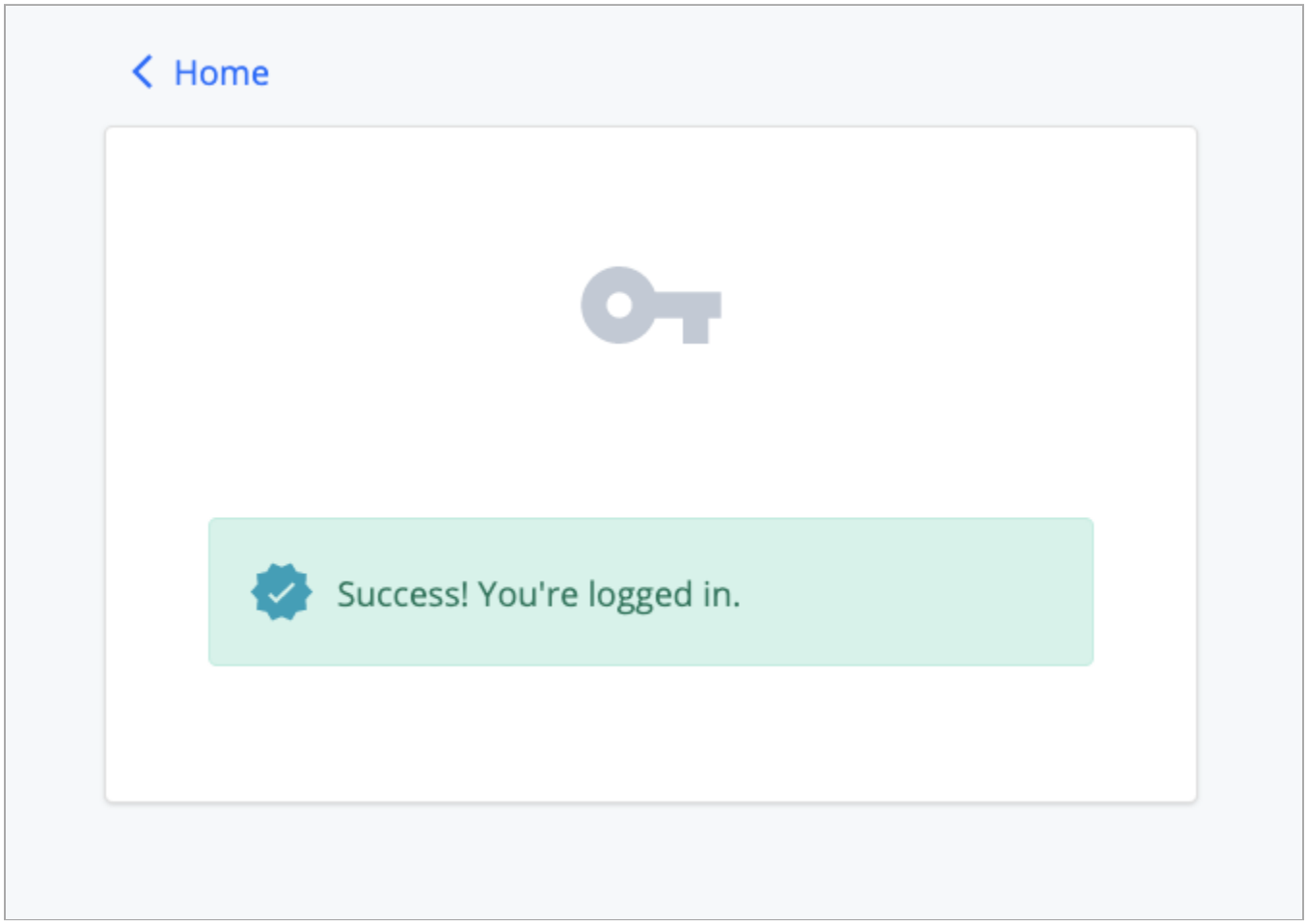


Figure 5. Screenshot of login page with success alert

Step 5. Start an OAuth 2.0 flow

At this point, the user is authenticated. The session has been created and a session cookie has been written to the browser. This is "session-based authentication", and is viable when your system (apps and services) can rely on cookies as the access artifact.

However, there are increasing limitations with the use of cookies. In response to this, and other reasons, it's common to add a step to your authentication process; the "OAuth" or "OIDC flow".

The goal of this flow is to attain a separate set of tokens, replacing the need for cookies as the shared access artifact. The two common tokens are the Access Token and the ID Token. We will focus on the access token in this guide. The specific flow that the SDK uses to acquire these tokens is called the Authorization Code Flow with PKCE.

To start, import the `TokenManager` object from the Ping SDK into the same `form.js` file.

reactjs-todo/client/components/journey/form.js

```
- import { FRAuth } from '@forgerock/javascript-sdk';
+ import { FRAuth, TokenManager } from '@forgerock/javascript-sdk';

@@ collapsed @@
```

Only an authenticated user that has a valid session can successfully request OAuth/OIDC tokens. Make sure we make this token request after we get a `'LoginSuccess'` back from the authentication journey. This is an asynchronous call to the server. There are multiple ways to handle this, but we'll use the `useEffect` and `useState` hooks.

Add a `useState` to the top of the function body to create a simple boolean flag of the user's authentication state.

reactjs-todo/client/components/journey/form.js

```
@@ collapsed @@

  export default function Form() {
    const [step, setStep] = useState(null);
+   const [isAuthenticated, setAuthentication] = useState(false);

@@ collapsed @@
```

Now, add a new `useEffect` hook to allow us to work with another asynchronous request. Unlike our first `useEffect`, this one will be dependent on the state of `isAuthenticated`. To do this, add `isAuthenticated` to the array passed in as the second argument. This instructs React to run the `useEffect` function when the value of `isAuthenticated` is changed.

reactjs-todo/client/components/journey/form.js

```
 @@ collapsed @@

   useEffect(() => {
     async function getStep() {
       try {
         const initialStep = await FRAuth.start();
         setStep(initialStep);
       } catch (err) {
         console.error(`Error: request for initial step; ${err}`);
       }
     }
     getStep();
   }, []);

 + useEffect(() => {
 +   async function oauthFlow() {
 +     try {
 +       const tokens = await TokenManager.getTokens();
 +       console.log(tokens);
 +     } catch (err) {
 +       console.error(`Error: token request; ${err}`);
 +     }
 +   }
 +   if (isAuthenticated) {
 +     oauthFlow();
 +   }
 + }, [isAuthenticated]);

   @@ collapsed @@
```

Finally, we need to conditionally set this authentication flag when we have a success response from our authentication journey. In your `form` element's `onSubmit` handler, add a simple conditional and set the flag to `true`.

reactjs-todo/client/components/journey/form.js

```
@@ collapsed @@

  <form
    className="cstm_form"
    onSubmit={(event) => {
      event.preventDefault();
      async function getStep() {
        try {
          const nextStep = await FRAuth.next(step);
+         if (nextStep.type === 'LoginSuccess') {
+           setAuthentication(true);
+         }
          console.log(nextStep);
          setStep(nextStep);
        } catch (err) {
          console.error(`Error: form submission; ${err}`);
        }
      }
      getStep();
    }}
  >

@@ collapsed @@
```

Once the changes are made, return to your browser and remove all cookies created from any previous logins. Refresh the page and verify the login form is rendered. If the success message continues to display, make sure "third-party cookies" are also removed.

Login with your test user. You should get a success message like you did before, but now check your browser's console log. You should see an additional entry of an object that contains your `idToken` and `accessToken`. Since the SDK handles storing these tokens for you, which are in `localStorage`, you have completed a full login and OAuth/OIDC flow.

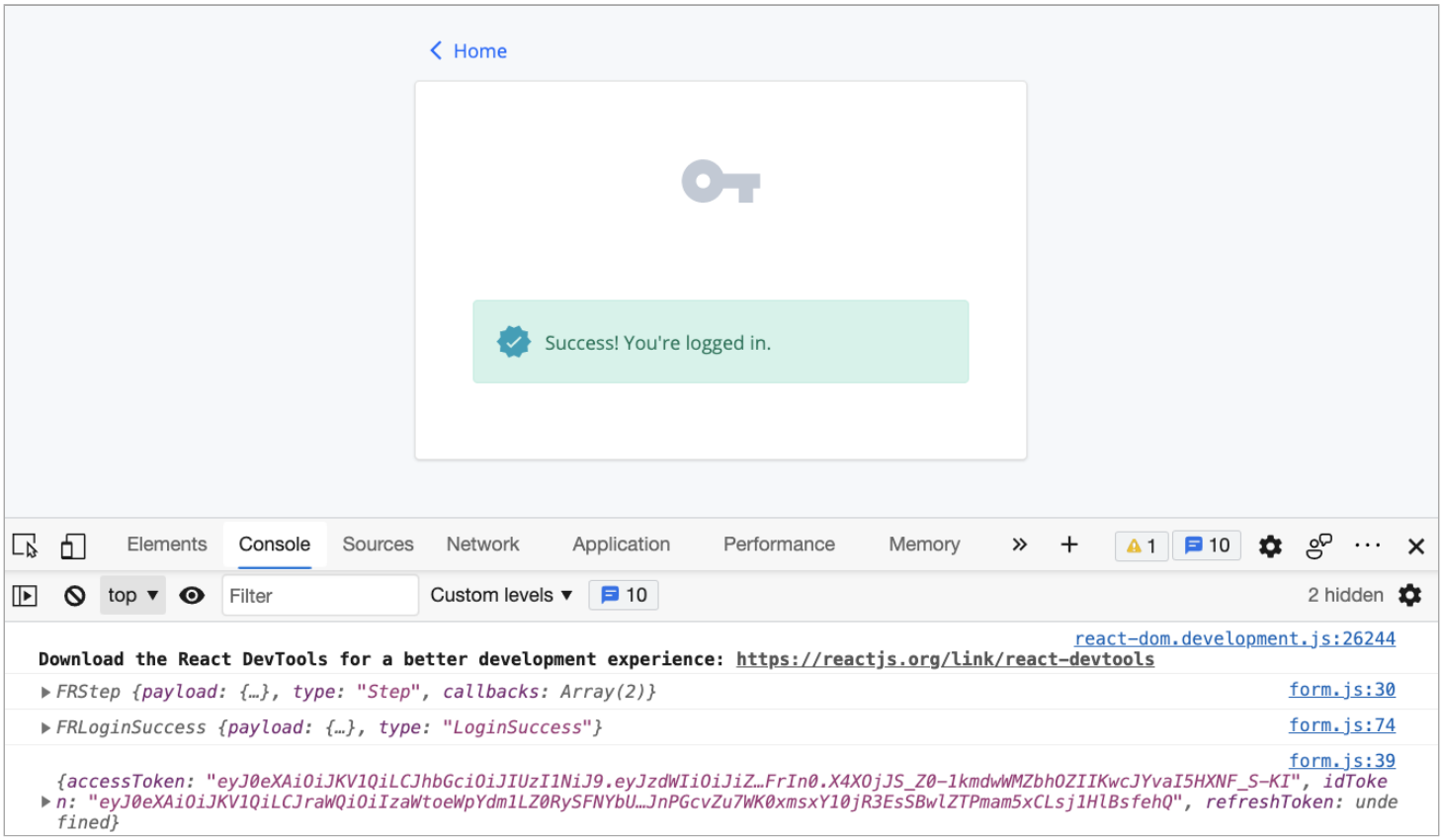


Figure 1. Screenshot of login page with success alert and console log

Request user information

Now that the user is authenticated and an access token is attained, you can now make your first authenticated request! The SDK provides a convenience method for calling the `/userinfo` endpoint, a standard OAuth endpoint for requesting details about the current user. The data returned from this endpoint correlates with the "scopes" set within the SDK configuration. The scopes `profile` and `email` will allow the inclusion of user's first and last name as well as their email address.

Within the `form.js` file, add the `UserManager` object to our Ping SDK import statement.

reactjs-todo/client/components/journey/form.js

```
- import { FRAuth, TokenManager } from '@forgerock/javascript-sdk';
+ import { FRAuth, TokenManager, UserManager } from '@forgerock/javascript-sdk';

@@ collapsed @@
```

The `getCurrentUser()` method on this new object will request the user's data and validate the existing access token. After the `TokenManager.getTokens()` method call, within the `oauthFlow()` function from above, add this new method.

reactjs-todo/client/components/journey/form.js

```
@@ collapsed @@

  try {
    const tokens = await TokenManager.getTokens();
    console.log(tokens);
+   const user = await UserManager.getCurrentUser();
+   console.log(user);

@@ collapsed @@
```

If the access token is valid, the user information will be logged to the console, just after the tokens. Before we move on from the `form.js` file, set a small portion of this state to the global context for application-wide state access. Add the remaining imports for setting the state and redirecting back to the home page: `useContext`, `AppContext` and `useNavigate`.

reactjs-todo/client/components/journey/form.js

```
- import React, { useEffect, useState } from 'react';
+ import React, { useContext, useEffect, useState } from 'react';
+ import { useNavigate } from 'react-router-dom';

+ import { AppContext } from '../../global-state';

@@ collapsed @@
```

At the top of the `Form` function body, use the `useContext()` method to get the app's global `state` and `methods`. Call the `useNavigate()` method to get the `navigation` object.

reactjs-todo/client/components/journey/form.js

```
  export default function Form() {
    const [step, setStep] = useState(null);
    const [isAuthenticated, setAuthentication] = useState(false);
+   const [_, methods] = useContext(AppContext);
+   const navigate = useNavigate();

@@ collapsed @@
```

After the `UserManager.getCurrentUser()` call, set the new user information to the global state and redirect to the home page.

reactjs-todo/client/components/journey/form.js

```
@@ collapsed @@

  const user = await UserManager.getCurrentUser();
  console.log(user);

+ methods.setUser(user.name);
+ methods.setEmail(user.email);
+ methods.setAuthentication(true);

+ navigate('/');

@@ collapsed @@
```

Revisit the browser, clear out all cookies, storage and cache, and log in with you test user. If you landed on the home page and the logs in the console show tokens and user data, you have successfully used the access token for retrieving use data. Notice that the home page looks a bit different with an added success alert and message with the user's full name. This is due to the app "reacting" to the global state that we set just before the redirection.

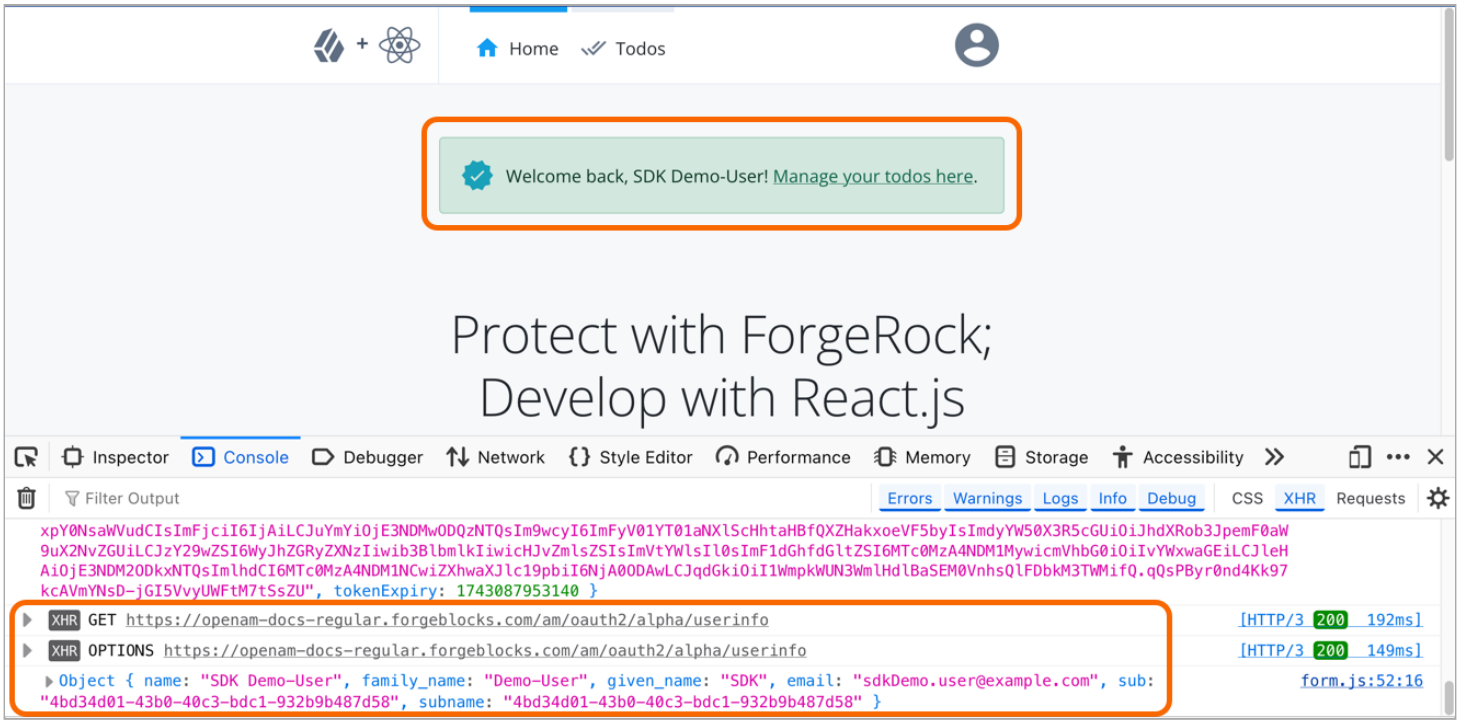


Figure 2. Screenshot of home page with successful login and user info

Step 6. Manage access tokens

To ensure your app provides a good user experience, it's important to have a recognizable, authenticated experience, even if the user refreshes the page or closes and reopens the browser tab. This makes it clear to the user that they are *logged in*.

Currently, if you refresh the page, the authenticated experience is lost. Let's fix that!

Because the SDK stores the tokens in localStorage, you can use their presence as a *hint* for their authentication status without requiring a network request. This allows for quickly rendering the appropriate navigational elements and content to the screen.

To do this, add the `TokenStorage.get` method to the `index.js` file as it will provide what we need to rehydrate the user's authentication status. First, import `TokenStorage` into the file. Use the `TokenStorage.get()` method within the `initAndHydrate` function. Second, add these values to the `useGlobalStateMgmt` function call.

reactjs-todo/client/index.js

```
- import { Config } from '@forgerock/javascript-sdk';
+ import { Config, TokenStorage } from '@forgerock/javascript-sdk';

  (async function initAndHydrate() {
    let isAuthenticated;
+   try {
+     isAuthenticated = !!(await TokenStorage.get());
+   } catch (err) {
+     console.error(`Error: token retrieval for hydration; ${err}`);
+   }

@@ collapsed @@

  function Init() {
    const stateMgmt = useGlobalStateMgmt({
      email,
+     isAuthenticated,
      prefersDarkTheme,
      username,
    });

@@ collapsed @@
```

With a global state API available throughout the app, different components can pull this state in and use it to conditionally render one set of UI elements versus another. Navigation elements and the displaying of profile data are good examples of such conditional rendering. Examples of this can be found by reviewing `components/layout/header.js` and `views/home.js`.

Validating the access token

The presence of the access token can be a good *hint* for authentication, but it doesn't mean the token is actually valid. Tokens can expire or be revoked on the server-side.

You can ensure the token is still valid with the use of `getCurrentUser()` method from earlier. This is optional, depending on your product requirements. If needed, you can protect routes with a token validation check before rendering portions of your application. This can prevent a potentially jarring experience of partial rendering UI that may be ejected due to an invalid token.

To validate a token for protecting a route, open the `router.js` file, import the `ProtectedRoute` module and replace the regular `<Route path="todos">` with the new `ProtectedRoute` wrapper.

reactjs-todo/client/router.js

```
@@ collapsed @@

  import Register from './views/register';
+ import { ProtectedRoute } from './utilities/route';
  import Todos from './views/todos';

@@ collapsed @@

<Route
  path="todos"
  element={
-   <>
+   <ProtectedRoute>
      <Header />
      <Todos />
      <Footer />
-   </>
+   </ProtectedRoute>
  }
/>

@@ collapsed @@
```

Let's take a look at what this wrapper does. Open `utilities/route.js` file and focus just on the `validateAccessToken` function within the `useEffect` function. Currently, it's just checking for the existence of the tokens with `TokenStorage.get`, which may be fine for some situations. We can optionally call the `UserManager.getCurrentUser()` method to ensure the stored tokens are still valid.

To do this, import `UserManager` into the file, and then replace `TokenStorage.get` with `UserManager.getCurrentUser`.

reactjs-todo/client/utilities/route.js

```
  import React, { useContext, useEffect, useState } from 'react';
  import { Route, Redirect } from 'react-router-dom';
- import { TokenStorage } from '@forgerock/javascript-sdk';
+ import { UserManager } from '@forgerock/javascript-sdk';

@@ collapsed @@

  useEffect(() => {
    async function validateAccessToken() {
     if (auth) {
        try {
-         await TokenStorage.get();
+         await UserManager.getCurrentUser();
          setValid('valid');
        } catch (err) {

@@ collapsed @@
```

In the code above, we are reusing the `getCurrentUser()` method to validate the token. If it succeeds, we can be sure our token is valid and call `setValid` to `'valid'`. If it fails, we know it is not valid and call `setValid` to `'invalid'`. We set that outcome with our `setValid()` state method and the routing will know exactly where to redirect the user.

Revisit the browser and refresh the page. Navigate to the todos page. You will notice a quick spinner and text communicating that the app is "verifying access". Once the server responds, the Todos page renders. As you can see, the consequence of this is the protected route now has to wait for the server to respond, but the user's access is being verified against the server.

At this point, that verification fails, as we aren't including the access token in the request.

Request protected resources with an access token

Once the Todos page renders, notice how the todo collection has a persistent spinner to indicate the process of requesting todos. This is due to the `fetch` request not having an `authorization` header, so the request does not succeed.

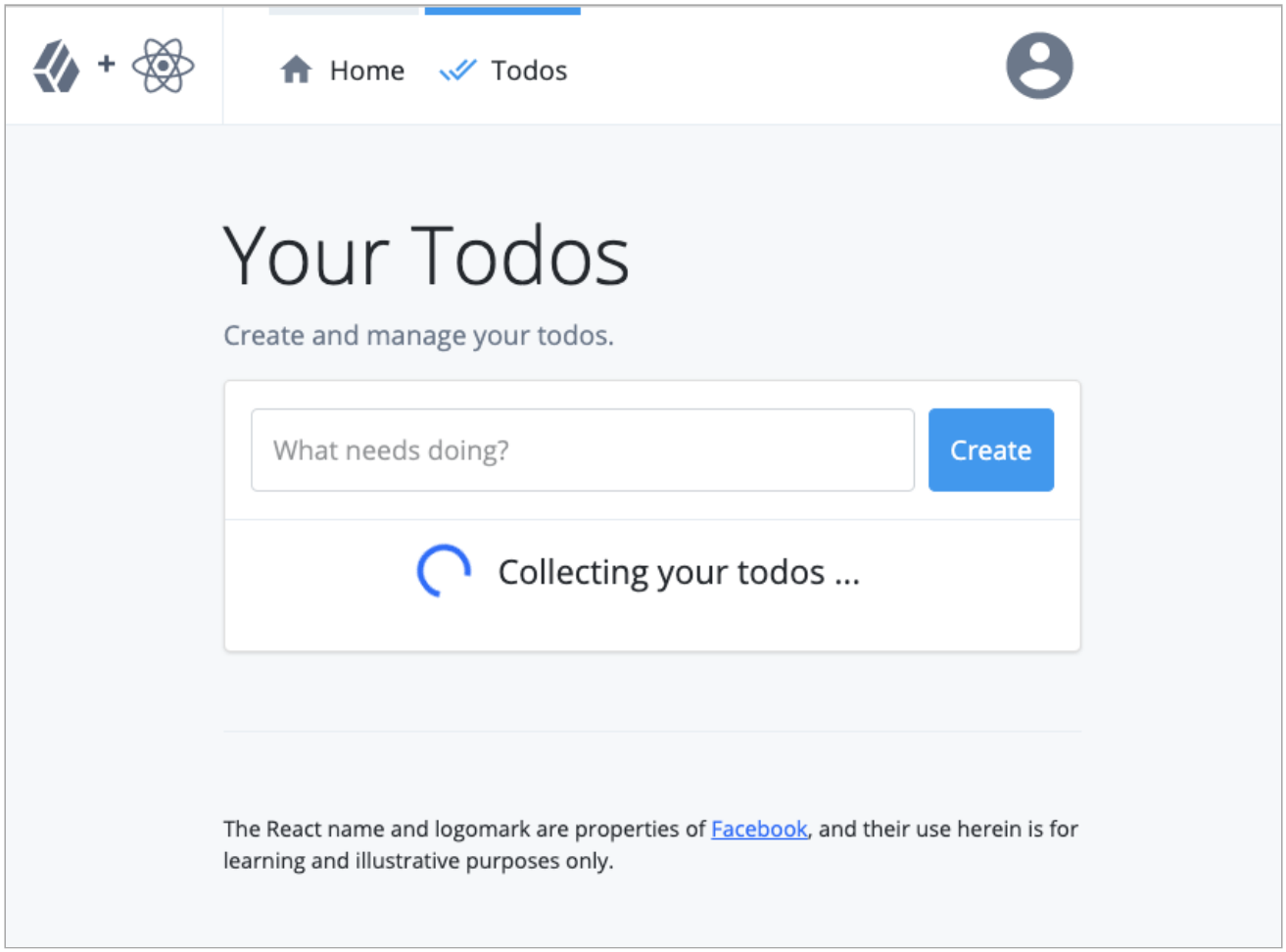


Figure 1. Screenshot of to-dos page with persistent spinner

To make resource requests to a protected endpoint use the `HttpClient` module. This module provides a simple wrapper around the native `fetch` method of the browser.

When you call the `HttpClient.request()` method the Ping SDK retrieves the user's access token and attaches it to the request in an `authorization` header as a **Bearer** token.

When the API backend server receives the request with the `authorization` header it calls your PingOne Advanced Identity Cloud tenant or PingAM server to validate the enclosed access token, and grants access to the route if successful.

To attach the user's access token to outgoing requests, open `utilities/request.js` and import the `HttpClient` from the Ping SDK. Then, replace the native fetch method with the `HttpClient.request()` method:

reactjs-todo/client/utilities/request.js

```
+ import { HttpClient } from '@forgerock/javascript-sdk';
  import { API_URL, DEBUGGER } from '../constants';

  export default async function apiRequest(resource, method, data) {
    let json;
    try {
-     const response = await fetch(`${API_URL}/${resource}`, {
+     const response = await HttpClient.request({
+       url: `${API_URL}/${resource}`,
+       init: {
          body: data && JSON.stringify(data),
          headers: {
            'Content-Type': 'application/json',
          },
          method: method,
+       },
      });

@@ collapsed @@
```

The `init` object in the above maps directly to the `init` options object seen in the official `Request` documentation in the Mozilla Web Docs.

The interface of the response from the request also maps directly to the official `Response` object seen in the Mozilla Web Doc.

At this point, the user can log in, request access tokens, and access the page of the protected resources (the "todos").

Now, revisit the browser and clear out all cookies, storage and cache. Keeping the developer tools open and on the network tab, log in with you test user. Once you have been redirected to the home page, do the following:

1. Click on the "Todos" item in the nav; a lot of network activity will be listed
2. Find the network call to the `/todos` endpoint (`http://localhost:9443/todos`)
3. Click on that network request and view the request headers
4. Notice the `authorization` header with the bearer token; that's the `HttpClient` in action

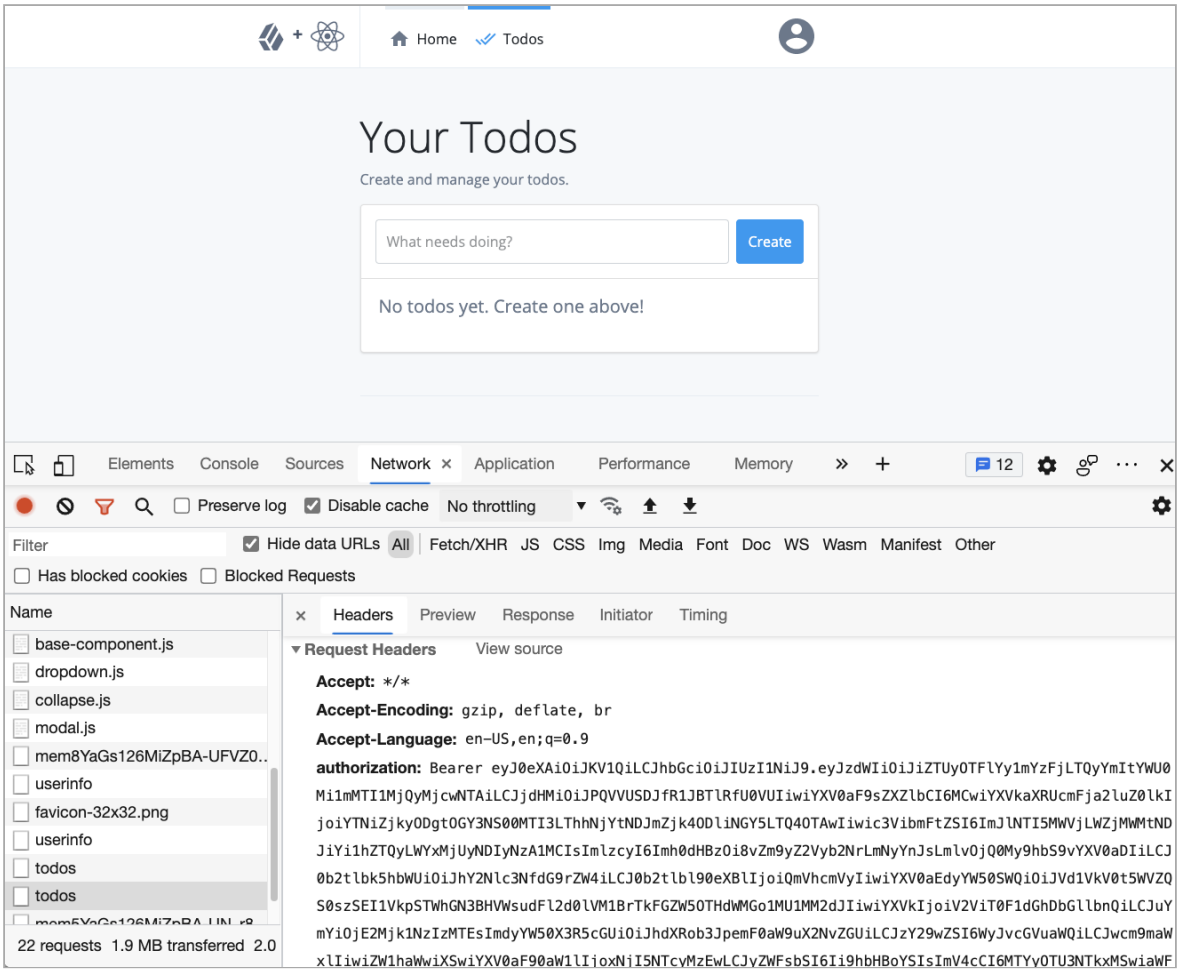


Figure 2. Screenshot of successful request for to-dos with Network panel open

Step 7. Handle logout requests

Of course, you can't have a protected app without providing the ability to log out. Luckily, this is a fairly easy task.

Open up the `views/logout.js` file and import the following:

1. `FRUser` from the Ping SDK
2. `useEffect` and `useContext` from React
3. `useNavigate` from React Router
4. `AppContext` from the global state module.

reactjs-todo/client/views/logout.js

```
+ import { FRUser } from '@forgerock/javascript-sdk';
- import React from 'react';
+ import React, { useContext, useEffect } from 'react';
+ import { useNavigate } from 'react-router-dom';

+ import { AppContext } from '../global-state';

@@ collapsed @@
```

Since logging out requires a network request, we need to wrap it in a `useEffect` and pass in a callback function with the following functionality:

reactjs-todo/client/views/logout.js

```
@@ collapsed @@

  export default function Logout() {
+   const [_, { setAuthentication, setEmail, setUser }] = useContext(AppContext);
+   const navigate = useNavigate();

+   useEffect(() => {
+     async function logout() {
+       try {
+         await FRUser.logout();

+         setAuthentication(false);
+         setEmail('');
+         setUser('');

+         navigate('/');
+       } catch (err) {
+         console.error(`Error: logout; ${err}`);
+       }
+     }
+     logout();
+   }, []);

    return <Loading classes="pt-5" message="You're being logged out ..." />;
  }
```

Since we only want to call this method once, after the component mounts, we will pass in an empty array as a second argument for `useEffect()`. After `FRUser.logout()` completes, we just empty or falsify the global state to clean up and redirect back to the home page.

You have now completed the coding part of this tutorial, and can proceed to the final step, Test the app.

Step 8. Test the app

Once all the previous steps are complete you can run the app end-to-end to see the flow.

1. In your browser, empty the local storage and cache.
2. Ensure that the client and API apps are running.

   You can run both apps with a single command:

   ```
   npm run start:reactjs-todo
   ```

3. In your browser, visit the home page of the client app at https://localhost:8443.

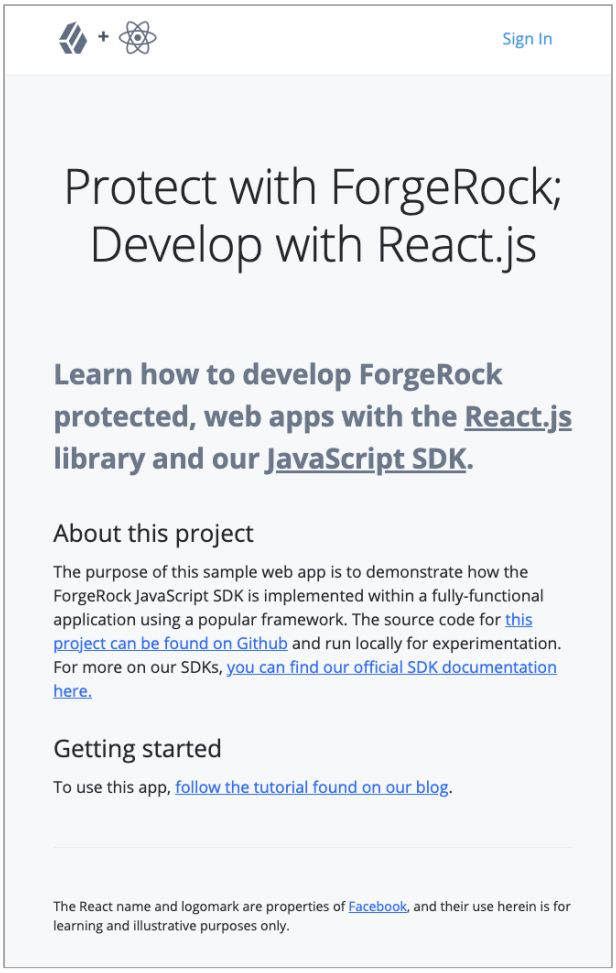


Figure 1. Screenshot of the home page

> **Note**
>
> You may need to dismiss warning from your browser about the self-signed certificate the client app uses.

4. Click **Sign In**, and enter the credentials of the demo user you created earlier.

   The app displays a welcome message, and outputs the data retrieved from the `/userinfo` OAuth 2.0 endpoint.

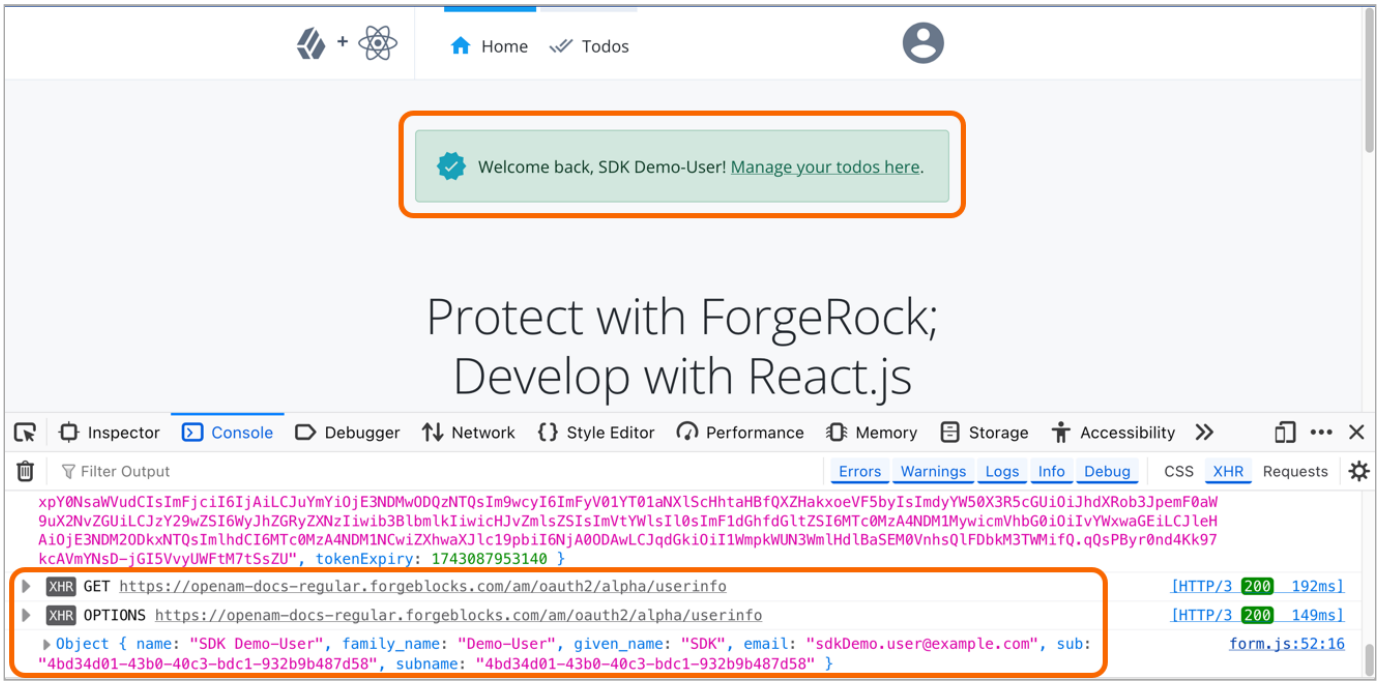


Figure 2. Screenshot of a user signed in to the home page, with `userinfo` data in the console.

5. Click **Todos**.

   The app opens the protected `/todos` route and inserts the access token as a bearer token in the authorization header. If the access token is valid the app displays an empty list of todo items.

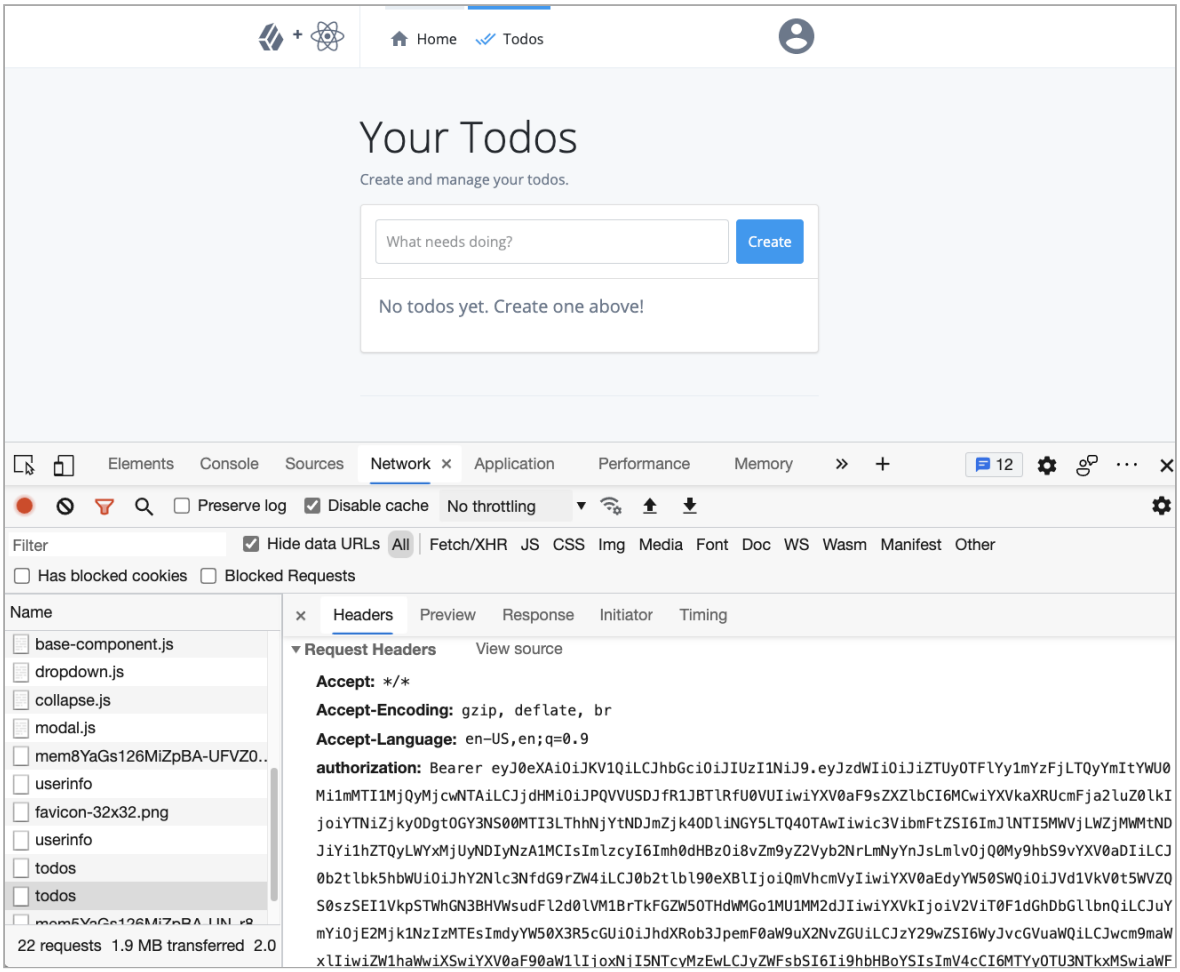


Figure 3. Screenshot of the todo page and console output showing the bearer token.

6. Click the user icon, and then click **Sign out**.

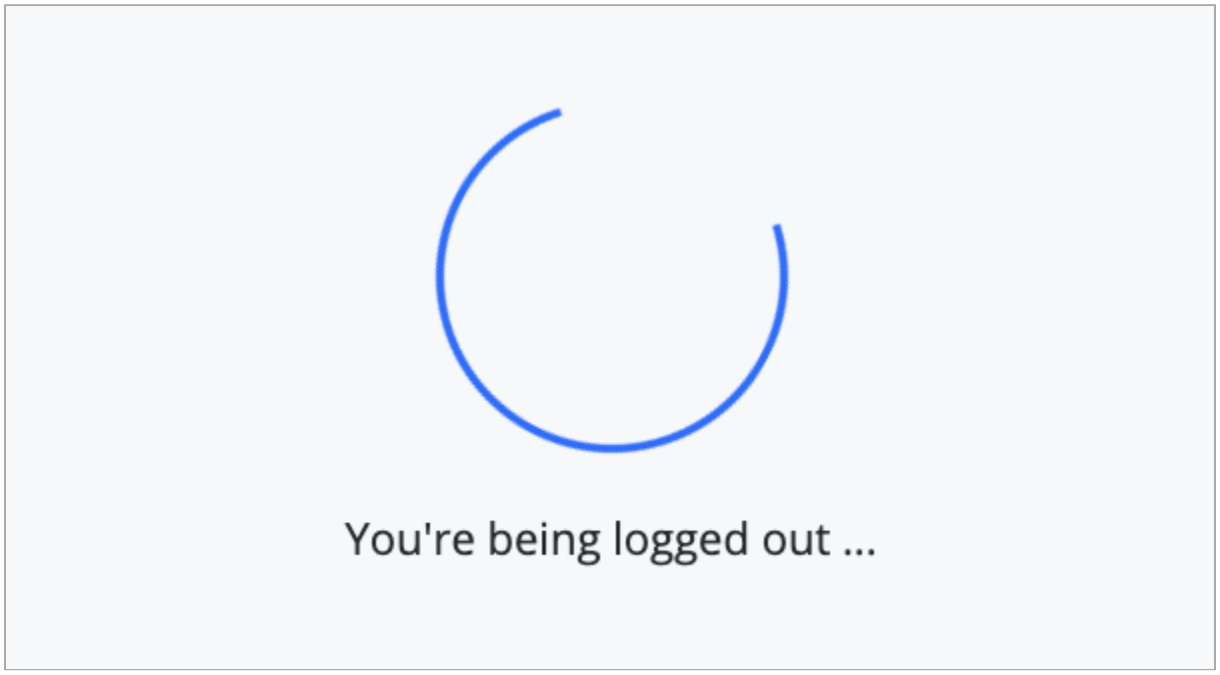


Figure 4. Screenshot of logout page with spinner

The app revokes the access token and removes the session cookies from storage, before returning the user to the home page.

Congratulations, you just built a protected app with ReactJS.

Authentication journey tutorial for an iOS React Native app

This tutorial covers the basics of developing a protected mobile app with React Native. You will develop the iOS bridge code along with a minimal React UI to authenticate a user.

Ping does not provide a React Native version of the Ping SDK. Instead we present this how-to as a guide to basic development of "bridge code" for connecting the Ping SDK for iOS to the React Native layer.

This guide covers how to implement the following application features using the Ping SDK for iOS and Ping SDK for JavaScript:

1. Authentication through a simple journey/tree.
2. Requesting OAuth/OIDC tokens.
3. Requesting user information.
4. Logging a user out.

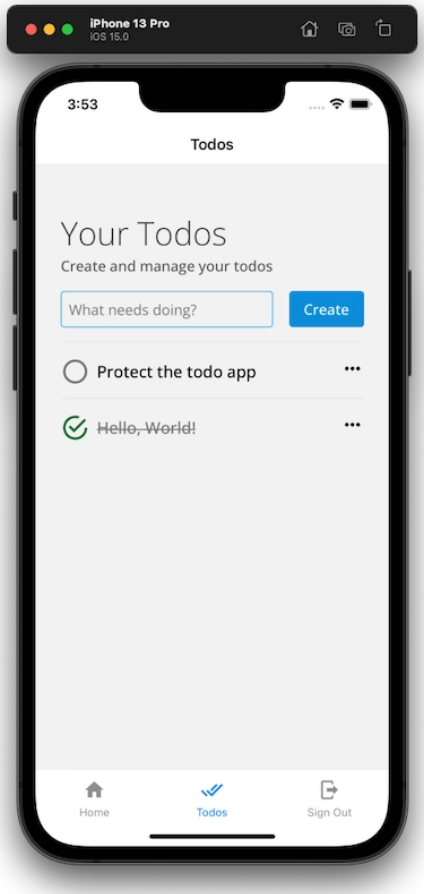


Figure 1. The to-do sample app

Before you begin

Before you begin this tutorial ensure you have set up your PingOne Advanced Identity Cloud tenant or PingAM server with the required configuration.

For example, you will need to configure an OAuth 2.0 client application, as well as an authentication journey for the app to navigate.

Complete prerequisites »

Step 1. Download the samples

To start this tutorial, you need to download the SDK sample apps repo, which contains the projects you will use.

Start step 1 »

Step 2. Configure the projects

In this step you install the dependencies the projects require, and configure the connection properties.

Start step 2 »

Step 3. Configure connection properties

In this step, you configure the samples to connect to the authentication tree/journey and OAuth 2.0 client you created when setting up your server configuration.

Start step 3 »

Step 4. Build and run the project

Build and run the apps, and learn about *Hot Module Reloading*.

Start step 4 »

Step 5. Implement the iOS bridge code

In this step you implement the bridge code and add methods for starting the Ping SDK, logging a user in, stepping through a journey, and finally logging a user out.

Start step 5 »

Step 6. Implement the UI in React Native

In this final step you implement the user interface for logging in, and code for submitting the forms. You will also handle returning to the list view, requesting user info, and handling logout triggers.

This is also the moment you can try out the fully functioning app.

Start step 6 »

Before you begin

To successfully complete this tutorial refer to the prerequisites and compatibility requirements in this section.

The tutorial also requires a configured server.

Compatibility

iOS

This sample app is compatible with iOS 12 and later.

Prerequisites

Xcode

You can download the latest version for free from https://developer.apple.com/xcode/.

Node.js

This tutorial requires Node.js 14 or higher and npm 7 or higher

You can check your version with `node -v` and `npm -v`.

Server configuration

This tutorial requires you to configure one of the following servers:

PingOne Advanced Identity Cloud

PingAM

PingOne Advanced Identity Cloud

The samples and tutorials in this documentation often require that you have an identity set up so that you can test authentication.

To create a demo user in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. In the left panel, click **Identities** > **Manage**.
3. Click **+ New Alpha realm - User**.
4. Enter the following details:
    - **Username** = `demo`
    - **First Name** = `Demo`
    - **Last Name** = `User`

- **Email Address** = `demo.user@example.com`
- **Password** = `Ch4ng3it!`

5. Click **Save**.

Authentication journeys provide fine-grained authentication by allowing multiple paths and decision points throughout the flow. Authentication journeys are made up of nodes that define actions taken during authentication.

Each node performs a single task, such as collecting a username or making a simple decision. Nodes can have multiple outcomes rather than just success or failure. For details, see the Authentication nodes configuration reference in the PingAM documentation.

To create a simple journey for use when testing the Ping SDKs, follow these steps:

1. In your PingOne Advanced Identity Cloud tenant, navigate to **Journeys**, and click **+ New Journey**.
2. Enter a name, such as `sdkUsernamePasswordJourney` and click **Save**.

   The authentication journey designer appears.
3. Drag the following nodes into the designer area:
   - **Page Node**
   - **Platform Username**
   - **Platform Password**
   - **Data Store Decision**
4. Drag and drop the **Platform Username** and **Platform Password** nodes onto the **Page Node**, so that they both appear on the same page when logging in.
5. Connect the nodes as follows:

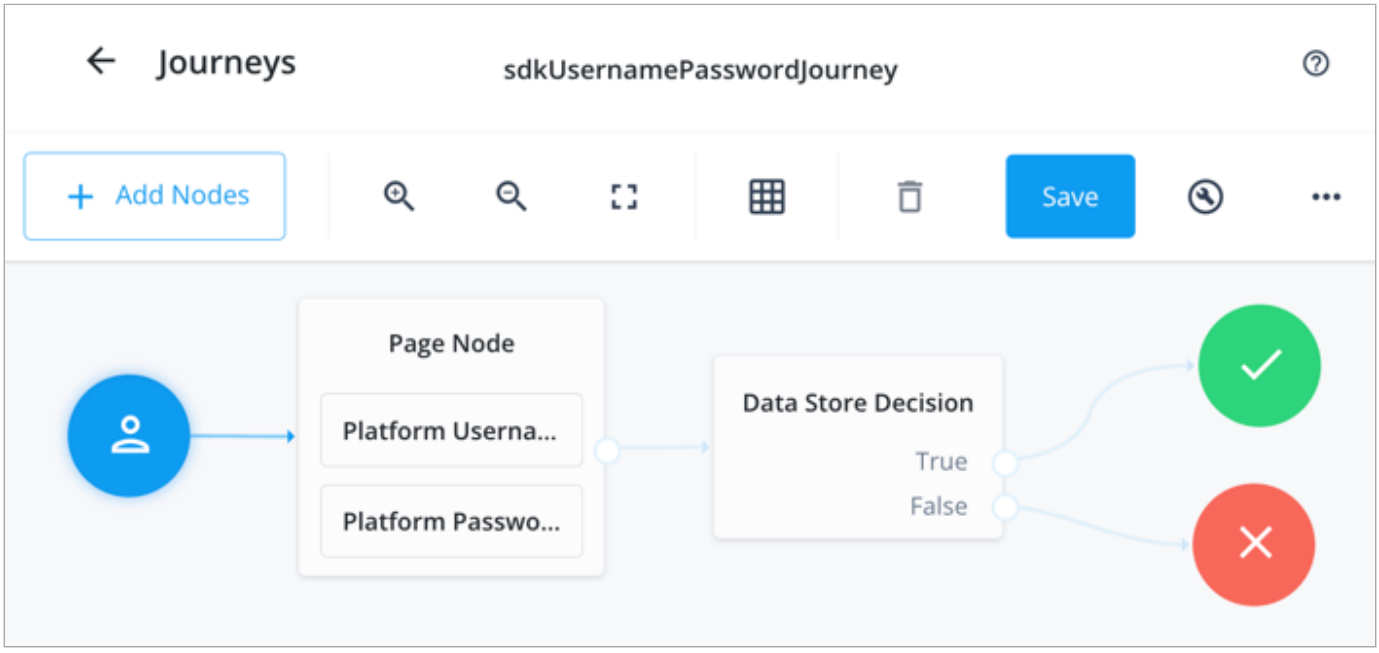


Figure 1. Example username and password authentication journey

6. Click **Save**.

Public clients do not use a client secret to obtain tokens because they are unable to keep them hidden. The Ping SDKs commonly use this type of client to obtain tokens, as they cannot guarantee safekeeping of the client credentials in a browser or on a mobile device.

To register a *public* OAuth 2.0 client application for use with the SDKs in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. In the left panel, click **Applications**.
3. Click **+ Custom Application**.
4. Select **OIDC - OpenId Connect** as the sign-in method, and then click **Next**.
5. Select **Native / SPA** as the application type, and then click **Next**.
6. In **Name**, enter a name for the application, such as `Public SDK Client`.
7. In **Owners**, select a user that is responsible for maintaining the application, and then click **Next**.

   > **Tip**
   >
   > When trying out the SDKs, you could select the `demo` user you created previously.

8. In **Client ID**, enter `sdkPublicClient`, and then click **Create Application**.

   PingOne Advanced Identity Cloud creates the application and displays the details screen.

9. On the **Sign On** tab:

   1. In **Sign-In URLs**, enter the following values:

      `https://com.example.reactnative.todo/callback`

      > **Important**
      >
      > Also add any other domains where you host SDK applications.

   2. In **Grant Types**, enter the following values:

      `Authorization Code`

      `Refresh Token`

   3. In **Scopes**, enter the following values:

      `openid profile email address`

10. Click Show advanced settings, and on the **Authentication** tab:

    1. In **Token Endpoint Authentication Method**, select `none`.
    2. In **Client Type**, select `Public`.
    3. Enable the **Implied Consent** property.

11. Click **Save**.

The application is now configured to accept client connections from and issue OAuth 2.0 tokens to the example applications and tutorials covered by this documentation.

The provider specifies the supported OAuth 2.0 configuration options for a realm.

To ensure the PingOne Advanced Identity Cloud OAuth 2.0 provider service is configured for use with the Ping SDKs, follow these steps:

1. In your PingOne Advanced Identity Cloud tenant, navigate to **Native Consoles** > **Access Management**.
2. In the left panel, click **Services**.
3. In the list of services, click **OAuth2 Provider**.
4. On the **Core** tab, ensure **Issue Refresh Tokens** is enabled.
5. On the **Consent** tab, ensure **Allow Clients to Skip Consent** is enabled.
6. Click **Save Changes**.

PingAM

The samples and tutorials in this documentation often require that you have an identity set up so that you can test authentication.

To create a demo user in PingAM, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. Navigate to **Identities**, and then click **+ Add Identity**.
3. Enter the following details:
    - **User ID** = `demo`
    - **Password** = `Ch4ng3it!`
    - **Email Address** = `demo.user@example.com`
4. Click **Create**.

Authentication trees provide fine-grained authentication by allowing multiple paths and decision points throughout the authentication flow. Authentication trees are made up of nodes that define actions taken during authentication.

Each node performs a single task, such as collecting a username or making a simple decision. Nodes can have multiple outcomes rather than just success or failure. For details, see the Authentication nodes configuration reference in the PingAM documentation.

To create a simple tree for use when testing the Ping SDKs, follow these steps:

1. Under **Realm Overview**, click **Authentication Trees**, then click **Create Tree**.

2. Enter a tree name, for example `sdkUsernamePasswordJourney`, and then click **Create**.

   The authentication tree designer appears, showing the **Start** entry point connected to the **Failure** exit point.

3. Drag the following nodes from the **Components** panel on the left side into the designer area:

   - **Page Node**
   - **Username Collector**
   - **Password Collector**
   - **Data Store Decision**

4. Drag and drop the **Username Collector** and **Password Collector** nodes onto the **Page Node**, so that they both appear on the same page when logging in.

5. Connect the nodes as follows:

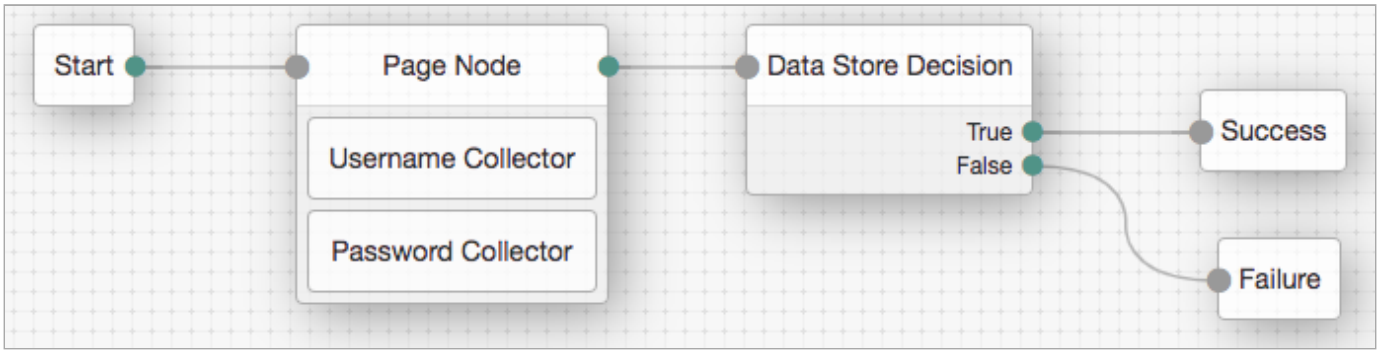


Figure 2. Example username and password authentication tree

6. Select the **Page Node**, and in the **Properties** pane, set the **Stage** property to `UsernamePassword`.

   > **Tip**
   >
   > You can configure the node properties by selecting a node and altering properties in the right-hand panel.

   One of the samples uses this specific value to determine the custom UI to display.

7. Click **Save**.

Public clients do not use a client secret to obtain tokens because they are unable to keep them hidden. The Ping SDKs commonly use this type of client to obtain tokens, as they cannot guarantee safekeeping of the client credentials in a browser or on a mobile device.

To register a *public* OAuth 2.0 client application for use with the SDKs in AM, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. Navigate to **Applications** > **OAuth 2.0** > **Clients**, and then click **+ Add Client**.
3. In **Client ID**, enter `sdkPublicClient`.
4. Leave **Client secret** empty.
5. In **Redirection URIs**, enter the following values:

`https://com.example.reactnative.todo/callback`

> **Important**
>
> Also add any other domains where you will be hosting SDK applications.

6. In **Scopes**, enter the following values:

   `openid profile email address`

7. Click **Create**.

   PingAM creates the new OAuth 2.0 client, and displays the properties for further configuration.

8. On the **Core** tab:

   1. In **Client type**, select `Public`.
   2. Disable **Allow wildcard ports in redirect URIs**.
   3. Click **Save Changes**.

9. On the **Advanced** tab:

   1. In **Grant Types**, enter the following values:

      ```
      Authorization Code
      Refresh Token
      ```

   2. In **Token Endpoint Authentication Method**, select `None`.
   3. Enable the **Implied consent** property.

10. Click **Save Changes**.

The provider specifies the supported OAuth 2.0 configuration options for a realm.

To ensure the PingAM OAuth 2.0 provider service is configured for use with the Ping SDKs, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. In the left panel, click **Services**.
3. In the list of services, click **OAuth2 Provider**.
4. On the **Core** tab, ensure **Issue Refresh Tokens** is enabled.
5. On the **Consent** tab, ensure **Allow Clients to Skip Consent** is enabled.
6. Click **Save Changes**.

Step 1. Download the samples

To start this tutorial, you need to download the React Native sample app repo, which contains the projects you will use.

1. In a web browser, navigate to the React Native Sample App repository.
2. Download the source code using one of the following methods:

Download a ZIP file

1. Click **Code**, and then click **Download ZIP**.
2. Extract the contents of the downloaded ZIP file to a suitable location.

Use a Git-compatible tool to clone the repo locally

1. Click **Code**, and then copy the HTTPS URL.
2. Use the URL to clone the repository to a suitable location.

   For example, from the command-line you could run:

```
git clone https://github.com/ForgeRock/forgerock-react-native-sample.git
```

The result of these steps is a local folder named `forgerock-react-native-sample`.

Step 2. Configure the projects

In this step you install the dependencies the projects require.

This React Native app requires two types of dependencies:

1. JavaScript and its Node package modules
2. Swift dependencies, using CocoaPods.

First, let's install the JavaScript dependencies. Within the project directory: `forgerock-react-native-sample/reactnative-todo/` (file and directory references are from this location), use the following command:

```
npm install
```

When the command finishes, `cd` into the `ios` directory and install the needed CocoaPods dependencies.

```
cd ios
pod install
```

When done, you can return to the project directory.

```
cd ..
```

Step 3. Configure connection properties

In this step, you configure the sample app to connect to the authentication tree/journey you created when setting up your server configuration.

Using the server settings from earlier, create a `.env.js` file within the project, using the `.env.js.template` as a source. This can be found the root folder of the project.

Add your relevant values to configure all the important server settings in the project. Not all variables will need values at this time.

You can list the file in the Terminal by doing `ls -a`, and edit it using a text editor like `nano` or `vi`.

Example `.env.js` file

```
/**
 * Avoid trailing slashes in the URL string values below
 */
const AM_URL = 'https://openam-forgerock-sdks.forgeblocks.com/am'; // Required; enter _your_ PingAM URL
const API_PORT = 8080; // Required; default port is 8080
const API_BASE_URL = 'http://localhost'; // Required; default domain is http://localhost
const DEBUGGER_OFF = true;
const REALM_PATH = 'alpha'; // Required
const REST_OAUTH_CLIENT = 'sdkPublicClient';
const REST_OAUTH_SECRET = '';
```

Descriptions of relevant values:

`AM_URL`

The URL that references PingAM itself (for PingOne Advanced Identity Cloud, the URL is likely `https://<tenant-name>.forgeblocks.com/am`).

`API_PORT` *and* `API_BASE_URL`

These just need to be "truthy" (not 0 or an empty string) right now to avoid errors, and we will use them in a future part of this series.

`DEBUGGER_OFF`

When `true`, this disables the `debugger` statements in the JavaScript layer. These debugger statements are for learning the integration points at runtime in your browser. When the browser's developer tools are open, the app pauses at each integration point. Code comments above each integration point explain its use.

`REALM_PATH`

The realm of your server (likely `root`, `alpha`, or `bravo`).

`REST_OAUTH_CLIENT` *and* `REST_OAUTH_SECRET`

We will use these values in a future part of this series, so any string value will do.

Step 4. Build and run the project

Now that everything is set up, build and run the to-do app project.

1. Open your Finder application and find the following file `ios/reactnativetodo.xcworkspace`.
2. Double click this file to open and load the project within Xcode.
3. Once Xcode is ready, select iPhone 11 or higher as the target for the device simulator on which to run the app.
4. Now, click the build/play button to build and run this application in the target simulator.

With everything up and running, you will need to rebuild the project with Xcode when you modify the bridge code (Swift files). But, when modifying the React Native code, it will use "hot module reloading" to automatically reflect the changes in the app without having to manually rebuild the project.

Troubleshooting

1. Make sure `libFRAuth.a` is added to your Target's **Frameworks, Libraries, and Embedded Content** under the **General** tab.
2. Make sure the Metro server is running; `npx react-native start` if you want to run it manually.
3. Bridge code has been altered, so be aware of API name changes.
4. If you get the error, `[!] CocoaPods could not find compatible versions for pod "FRAuth"`, run `pod repo update` then `pod install`.

Xcode, iOS Simulator and Safari dev tools

We recommend the use of iPhone 11 or higher as the target for the iOS Simulator. When you first run the build command in Xcode (clicking the "play" button), it takes a while for the app to build, the OS to load, and app to launch within the Simulator. Once the app is launched, rebuilding it is much faster if the changes are not automatically "hot reloaded" when made in the React layer.

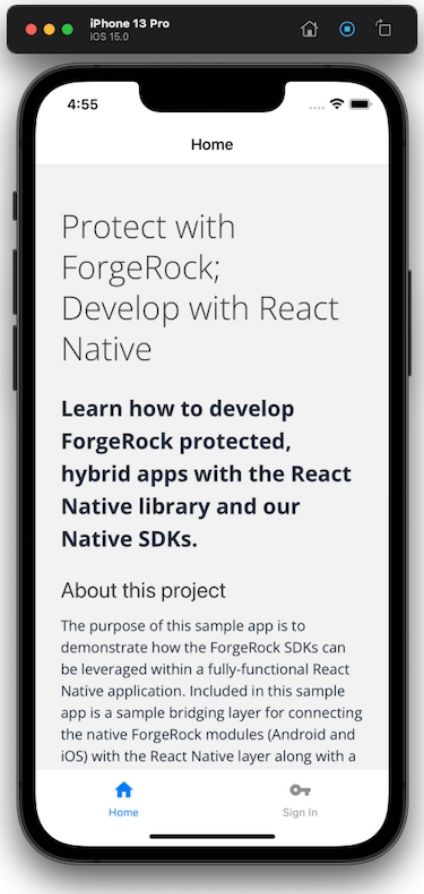


Figure 1. To-do app home screen

> **Note**
>
> Only the home screen will render successfully at this moment. If you click on the **Sign In** button, it won't be fully functional. This is **intended** as you will develop this functionality throughout this tutorial.

Once the app is built and running, you will have access to all the logs within Xcode's output console. Both the native and JavaScript logs display here. Because of this, there's quite a lot of output, so you may want to use it only when the Safari console does not provide enough information for debugging purposes.

Figure 2. Xcode log output

For additional tooling, click "Device" within the top menu, and then select "Shake". This triggers the React Native dev tools, allowing you to reload the app, inspect the UI, as well as other actions.

> **Warning**
>
> Due to a particular confusing bug in React Native, we do not recommend using the Chrome debugger, but recommend using Safari for debugging. To use Safari for debugging the React code, follow the instructions found in the React Native docs.

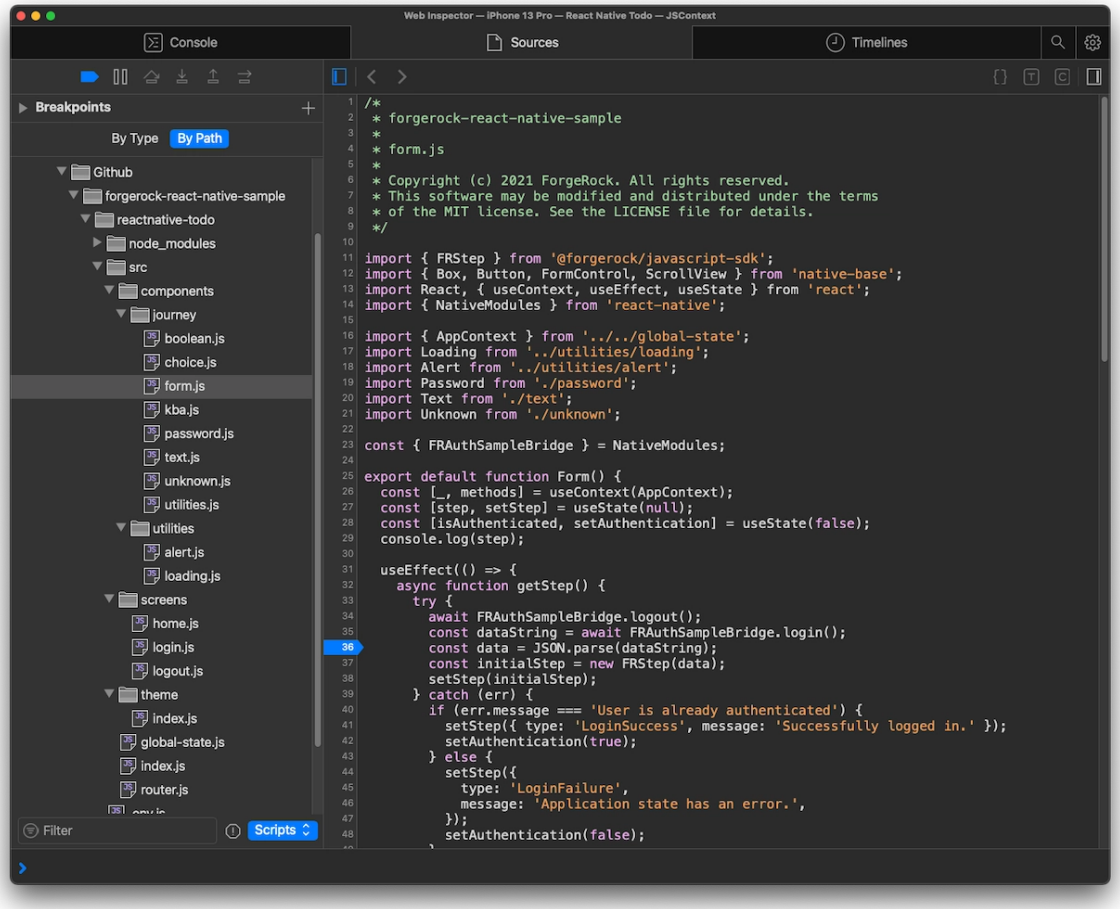


Figure 3. Safari dev tools

Tips if the home screen doesn't render

1. Restart the app (in Xcode) and Metro (in terminal).
2. Didn't work? Using Xcode, clean the build folder and rebuild/rerun the app.
3. If that doesn't work, remove the following from the `reactnative-todo` directory: `node_modules`, `package-lock.json`, `ios/.Pods`, `ios/Podfile.lock`, and then reinstall dependencies with `npm i` and within the `ios/` directory `pod install`.
4. If you're still having issues, within the simulator, click the **Home** button and long press the React Todo application to .remove it. Then, restart from the project Xcode.
5. You can also use **Device** > **Erase All Content and Settings** if the problem persists.

Step 5. Implement the iOS bridge code

Review the files that allow for the "bridging" between the React Native project and the native Ping SDK.

Within Xcode, navigate to the `forgerock-react-native-sample/reactnative-todo/ios` directory, and you will see a few important files:

- `reactnativetodo-Bridging-Header.h`: Header file that exposes the React Native bridging module and the FRAuth module into the Swift context.
- `reactnativetodo/FRAuthSampleBridge.m`: The module file that defines the exported interfaces of our bridging code.
- `reactnativetodo/FRAuthSampleBridge.swift`: The main Swift bridging code that provides the callable methods for the React Native layer.
- `reactnativetodo/FRAuthSampleStructs.swift`: Provides the structs for the Swift bridging code.
- `reactnativetodo/FRAuthSampleHelpers.swift`: Provides the extensions to often used objects within the bridge code.
- `reactnativetodo/FRAuthConfig.plist`: The `.plist` file that configures the Ping SDK for iOS to the appropriate authorization server.

We provide the header file as-is. The file's creation, naming and use requires very specific conventions that are outside the scope of this tutorial. You will not need to modify it.

> **Note**
>
> The remainder of the files within the workspace are automatically generated when you create a React Native project with the CLI command, so you can ignore them.

Configure your `.plist` file

Within Xcode's directory/file list section (aka Project Navigator), complete the following:

1. Find `FRAuthConfig.plist` file within the `ios/reactnativetodos` directory.
2. Add the name of your PingOne Advanced Identity Cloud or PingAM cookie.
3. Add the OAuth client you created from above.
4. Add your authorization server URLs.
5. Add the login tree you created above.

A hypothetical example (your values may vary):

```
  <dict>
    <key>forgerock_cookie_name</key>
-   <string></string>
+   <string>iPlanetDirectoryPro</string>
    <key>forgerock_enable_cookie</key>
    <true/>
    <key>forgerock_oauth_client_id</key>
    <string>ReactNativeOAuthClient</string>
    <key>forgerock_oauth_redirect_uri</key>
    <string>https://com.example.reactnative.todo/callback</string>
    <key>forgerock_oauth_scope</key>
    <string>openid profile email</string>
    <key>forgerock_oauth_url</key>
-   <string></string>
+   <string>https://auth.forgerock.com/am</string>
    <key>forgerock_oauth_threshold</key>
    <string>60</string>
    <key>forgerock_url</key>
-   <string></string>
+   <string>https://auth.forgerock.com/am</string>
    <key>forgerock_realm</key>
-   <string></string>
+   <string>alpha</string>
    <key>forgerock_timeout</key>
    <string>60</string>
    <key>forgerock_keychain_access_group</key>
    <string>org.reactjs.native.example.reactnativetodo</string>
    <key>forgerock_auth_service_name</key>
-   <string></string>
+   <string>UsernamePassword</string>
    <key>forgerock_registration_service_name</key>
-   <string></string>
+  <string>Registration</string>
</dict>
```

Descriptions of relevant values:

- `forgerock_cookie_name`: If you have PingOne Advanced Identity Cloud, you can find this random string value under the **Tenant Settings** found in the top-right dropdown in the admin UI. If you have your own installation of PingAM, this is often `iPlanetDirectoryPro`.
- `forgerock_url` & `forgerock_oauth_url`: The URL of PingAM within your server installation.
- `forgerock_realm`: The realm of your server (likely `root`, `alpha` or `bravo`).
- `forgerock_auth_service_name`: This is the journey/tree that you use for login.
- `forgerock_registration_service_name`: This is the journey/tree that you use for registration, but it will not be used until a future part of this tutorial series.

Write the `start()` method

Staying within the `reactnativetodo` directory, find the `FRAuthSampleBridge` file and open it. We have some of the files already stubbed out and the dependencies are already installed. All you need to do is write the functionality.

For the SDK to initialize with the `FRAuth.plist` configuration from Step 2, write the `start()` function as follows:

```
  import Foundation
  import FRAuth
  import FRCore
  import UIKit

  @objc(FRAuthSampleBridge)
  public class FRAuthSampleBridge: NSObject {
    var currentNode: Node?

    @objc static func requiresMainQueueSetup() -> Bool {
      return false
    }

+   @objc func start(
+     _ resolve: @escaping RCTPromiseResolveBlock,
+     rejecter reject: @escaping RCTPromiseRejectBlock) {

+     /**
+      * Set log level to all
+      */
+     FRLog.setLogLevel([.all])
+
+     do {
+       try FRAuth.start()
+       let initMessage = "SDK is initialized"
+       FRLog.i(initMessage)
+       resolve(initMessage)
+     } catch {
+       FRLog.e(error.localizedDescription)
+       reject("Error", "SDK Failed to initialize", error)
+     }
+   }

    /**
     * Method for calling the `getUserInfo` to retrieve the user information from
     * the OIDC endpoint
     */
    @objc func getUserInfo(
      _ resolve: @escaping RCTPromiseResolveBlock,
      rejecter reject: @escaping RCTPromiseRejectBlock) {

@@ collapsed @@
```

The `start()` function above calls the Ping SDK for iOS's `start()` method on the `FRAuth` class. There's a bit more that may be required within this function for a production app. We'll get more into this in a separate part of this series, but for now, let's keep this simple.

Write the `login()` method

Once the `start()` method is called and it has initialized, the SDK is now ready to handle user requests. Let's start with `login()`.

Just underneath the `start()` method we wrote above, add the `login()` method.

```
@@ collapsed @@

  @objc func start(
    _ resolve: @escaping RCTPromiseResolveBlock,
    rejecter reject: @escaping RCTPromiseRejectBlock) {

    /**
     * Set log level according to  all
     */
    FRLog.setLogLevel([.all])

    do {
      try FRAuth.start()
      let initMessage = "SDK is initialized"
      FRLog.i(initMessage)
      resolve(initMessage)
    } catch {
      FRLog.e(error.localizedDescription)
      reject("Error", "SDK Failed to initialize", error)
    }
  }

+ @objc func login(
+   _ resolve: @escaping RCTPromiseResolveBlock,
+   rejecter reject: @escaping RCTPromiseRejectBlock) {
+
+   FRUser.login { (user, node, error) in
+     self.handleNode(user, node, error, resolve: resolve, rejecter: reject)
+   }
+ }

@@ collapsed @@
```

This `login()` function initializes the journey/tree specified for authentication. You call this method without arguments as it does not login the user. This initial call to the server will return the first set of callbacks that represents the first node in your journeyt/tree to collect user data.

Also, notice that we have a special "handler" function within the callback of `FRUser.login()`. This `handleNode()` method serializes the `node` object that the Ping SDK for iOS returns in a JSON string. Data passed between the "native" layer and the React layer is limited to strings. This method can be written in many ways and should be written in whatever way is best for your application. However, a unique use of the Ping SDK for JavaScript to convert this basic JSON of data into a decorated object for better ergonomics is used in this tutorial.

Write the `next()` method

To finalize the functionality needed to complete user authentication, we need a way to iteratively call `next` until the tree completes successfully or fails. To do this, continue in the bridge file, and add a private method called `handleNode()`.

First, we will write the decoding of the JSON string and prepare the node for submission.

```
@@ collapsed @@

  @objc func login(
    _ resolve: @escaping RCTPromiseResolveBlock,
    rejecter reject: @escaping RCTPromiseRejectBlock) {

    FRUser.login { (user, node, error) in
      self.handleNode(user, node, error, resolve: resolve, rejecter: reject)
    }
  }

+ @objc func next(
+   _ response: String,
+   resolve: @escaping RCTPromiseResolveBlock,
+   rejecter reject: @escaping RCTPromiseRejectBlock) {
+
+   let decoder = JSONDecoder()
+   let jsonData = Data(response.utf8)
+   if let node = self.currentNode {
+     var responseObject: Response?
+     do {
+       responseObject = try decoder.decode(Response.self, from: jsonData)
+     } catch  {
+       FRLog.e(String(describing: error))
+       reject("Error", "UnknownError", error)
+     }
+
+     let callbacksArray = responseObject!.callbacks ?? []
+
+     for (outerIndex, nodeCallback) in node.callbacks.enumerated() {
+       if let thisCallback = nodeCallback as? SingleValueCallback {
+         for (innerIndex, rawCallback) in callbacksArray.enumerated() {
+           if let inputsArray = rawCallback.input, outerIndex == innerIndex,
+             let value = inputsArray.first?.value {
+
+             thisCallback.setValue(value.value as! String)
+           }
+         }
+       }
+     }
+   } else {
+     reject("Error", "UnknownError", nil)
+   }
+ }

@@ collapsed @@
```

Now that you've prepared the data for submission, introduce the `node.next` call from the Ping SDK for iOS. Then, handle the subsequent `node` returned from the `next` call, or process the success or failure representing the completion of the journey/tree.

```
@@ collapsed @@

      for (outerIndex, nodeCallback) in node.callbacks.enumerated() {
        if let thisCallback = nodeCallback as? SingleValueCallback {
          for (innerIndex, rawCallback) in callbacksArray.enumerated() {
            if let inputsArray = rawCallback.input, outerIndex == innerIndex,
              let value = inputsArray.first?.value {

              thisCallback.setValue(value)
            }
          }
        }
      }

+     node.next(completion: { (user: FRUser?, node, error) in
+       if let node = node {
+         self.handleNode(user, node, error, resolve: resolve, rejecter: reject)
+       } else {
+         if let error = error {
+           reject("Error", "LoginFailure", error)
+           return
+         }
+
+         let encoder = JSONEncoder()
+         encoder.outputFormatting = .prettyPrinted
+         if let user = user,
+           let token = user.token,
+           let data = try? encoder.encode(token),
+           let accessInfo = String(data: data, encoding: .utf8) {
+
+           resolve(["type": "LoginSuccess", "accessInfo": accessInfo])
+         } else {
+           resolve(["type": "LoginSuccess", "accessInfo": ""])
+         }
+       }
+     })
    } else {
      reject("Error", "UnknownError", nil)
    }
  }

@@ collapsed @@
```

The above code handles a limited number of callback types. Handling full authentication and registration journeys/trees requires additional callback handling. To keep this tutorial simple, we'll focus just on `SingleValueCallback` type.

Write the `logout()` bridge method

Finally, add the following lines of code to enable logout for the user:

```
@@ collapsed @@

    } else {
      reject("Error", "UnknownError", nil)
    }

+   @objc func logout() {
+     FRUser.currentUser?.logout()
+   }
  }

@@ collapsed @@
```

Step 6. Implement the UI in React Native

Let's review how the application renders the *home* view:

`index.js` > `src/index.js` > `src/router.js` > `src/screens/home.js`

Open up the second file in the above sequence, the `src/index.js` file, and write the following:

1. Import `useEffect` from the React library.
2. Import `NativeModules` from the `react-native` package.
3. Pull `FRAuthSampleBridge` from the `NativeModules` object.
4. Write an `async` function within the `useEffect` callback to call the SDK `start()` method.

```
- import React from 'react';
+ import React, { useEffect } from 'react';
+ import { NativeModules } from 'react-native';
  import { SafeAreaProvider } from 'react-native-safe-area-context';

  import Theme from './theme/index';
  import { AppContext, useGlobalStateMgmt } from './global-state';
  import Router from './router';

+ const { FRAuthSampleBridge } = NativeModules;

  export default function App() {
    const stateMgmt = useGlobalStateMgmt({});

+   useEffect(() => {
+     async function start() {
+       await FRAuthSampleBridge.start();
+     }
+     start();
+   }, []);

    return (
      <Theme>
        <AppContext.Provider value={stateMgmt}>
          <SafeAreaProvider>
            <Router />
          </SafeAreaProvider>
        </AppContext.Provider>
      </Theme>
    );
  }
```

`FRAuthSampleBridge` is the JavaScript representation of the Swift bridge code we developed earlier. Any public methods added to the Swift class within the bridge code are available in the `FRAuthSampleBridge` object.

> **Note**
>
> It's important to initialize the SDK at a root level. Call this initialization step, so it resolves before any other native SDK methods can be used.

Build the login view

Navigate to the app's login view within the Simulator. You should see a "loading" spinner and a message that's persistent, since the app doesn't have the data needed to render the form. To ensure the correct form is rendered, the initial data needs to be retrieved from the server. That will be the first task.

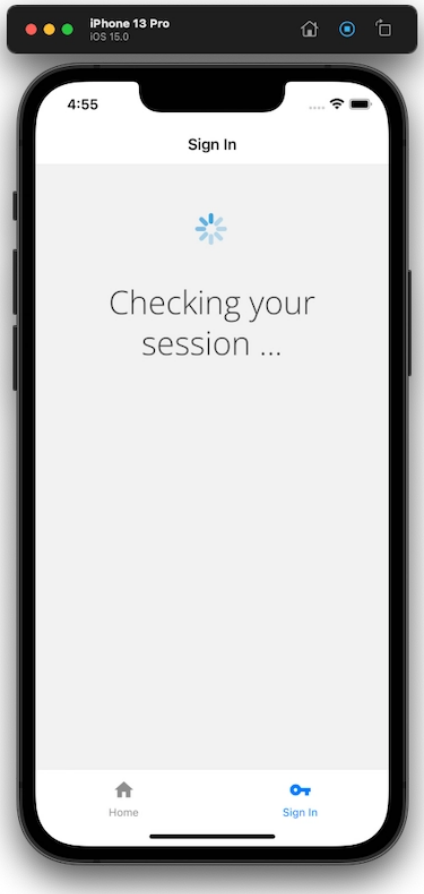


Figure 1. Login screen with spinner

Since most of the action is taking place in `src/components/journey/form.js`, open it and add the following:

1. Import `FRStep` from the `@forgerock/javascript-sdk` for improved ergonomics for handling callbacks.
2. Import `NativeModules` from the `react-native` package.
3. Pull `FRAuthSampleBridge` from the `NativeModules` object.

```
+ import { FRStep } from '@forgerock/javascript-sdk';
  import React from 'react';
+ import { NativeModules } from 'react-native';

  import Loading from '../utilities/loading';

+ const { FRAuthSampleBridge } = NativeModules;

@@ collapsed @@
```

To develop the login functionality, we first need to use the `login()` method from the bridge code to get the first set of callbacks, and then render the form appropriately. This `login()` method is an asynchronous method, so import a few additional packages from React to encapsulate this "side effect". Let's get started!

Import two new modules from React: `useState` and `useEffect`. The `useState`() method is for managing the data received from the server, and the `useEffect` is for the `FRAuthSampleBridge.login()` method's asynchronous, network request.

Compose the data gathering process using the following:

1. Import `useEffect` from the React library.
2. Write the `useEffect` function inside the component function.
3. Write an `async` function within the `useEffect` for calling `login`.
4. Write an `async` `logout` function to ensure user if fully logged out before attempting to `login`.
5. Call `FRAuthSampleBridge.login()` to initiate the call to the login journey/tree.
6. When the `login()` call returns with the data, parse the JSON string.
7. Assign that data to our component state via the `setState()` method.
8. Lastly, call this new method to execute this process.

```
  import { FRStep } from '@forgerock/javascript-sdk';
- import React from 'react';
+ import React, { useEffect, useState } from 'react';
  import { NativeModules } from 'react-native';

  import Loading from '../utilities/loading';

  const { FRAuthSampleBridge } = NativeModules;

  export default function Form() {
+   const [step, setStep] = useState(null);
+   console.log(step);
+
+   useEffect(() => {
+     async function getStep() {
+       try {
+         await FRAuthSampleBridge.logout();
+         const dataString = await FRAuthSampleBridge.login();
+         const data = JSON.parse(dataString);
+         const initialStep = new FRStep(data);
+         setStep(initialStep);
+       } catch (err) {
+         setStep({
+           type: 'LoginFailure',
+           message: 'Application state has an error.',
+         });
+       }
+     }
+     getStep();
+   }, []);

    return <Loading message="Checking your session ..." />;
  }
```

> **Tip**
>
> We are passing an empty array as the second argument into `useEffect`. This instructs the `useEffect` to only run once after the component mounts. This is functionally is equivalent to a class component using `componentDidMount` to run an asynchronous method after the component mounts.

The above code will result in two logs to your console:

1. `null`
2. An object with a few properties.

The property to focus on is the `callbacks` property. This property contains the instructions for what needs to be rendered to the user for input collection.

Import the components from NativeBase as well as the custom, local components within this `journey/` directory:

```
  import { FRStep } from '@forgerock/javascript-sdk';
+ import { Box, Button, FormControl, ScrollView } from 'native-base';
  import React, { useEffect, useState } from 'react';
  import { NativeModules } from 'react-native';

  import Loading from '../utilities/loading';
+ import Alert from '../utilities/alert';
+ import Password from './password';
+ import Text from './text';
+ import Unknown from './unknown';

@@ collapsed @@
```

Now, within the `Form` function body, create the function that maps these imported components to their appropriate callbacks.

```
@@ collapsed @@

  export default function Form() {
    const [step, setStep] = useState(null);
    console.log(step);

@@ collapsed @@

+   function mapCallbacksToComponents(cb, idx) {
+     const name = cb?.payload?.input?.[0].name;
+     switch (cb.getType()) {
+       case 'NameCallback':
+         return <Text callback={cb} inputName={name} key="username" />;
+       case 'PasswordCallback':
+         return <Password callback={cb} inputName={name} key="password" />;
+       default:
+         // If current callback is not supported, render a warning message
+         return <Unknown callback={cb} key={`unknown-${idx}`} />;
+     }
+   }

    return <Loading message="Checking your session ..." />;
  }
```

Finally, return the appropriate component for the following states:

- If there is no `step` data, render the `Loading` component to indicate the request is still processing.
- If there is `step` data, and it is of type `'Step'`, then map over `step.callbacks` with the function from above.
- If there is `step` data, but the type is `'LoginSuccess'` or `'LoginFailure'`, render an alert.

```
@@ collapsed @@

+ if (!step) {
    return <Loading message='Checking your session ...' />;
+ } else if (step.type === 'Step') {
+   return (
+     <ScrollView>
+       <Box safeArea flex={1} p={2} w="90%" mx="auto">
+         <FormControl>
+           {step.callbacks?.map(mapCallbacksToComponents)}
+           <Button>Sign In</Button>
+         </FormControl>
+       </Box>
+     </ScrollView>
+   );
+ } else {
+   // Handle success or failure of the journey/tree
+   return (
+     <Box safeArea flex={1} p={2} w="90%" mx="auto">
+       <Alert message={step.message} type={step.type} />
+     </Box>
+   );
+ }
```

Refresh the page, and you should now have a dynamic form that reacts to the callbacks returned from our initial call to ForgeRock.

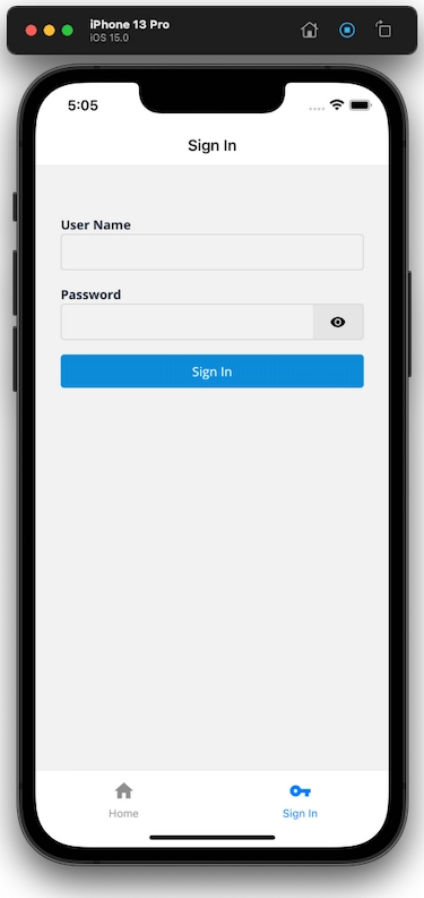


Figure 2. Login screen form

Handle the login form submission

Since a form that can't submit anything isn't very useful, we'll now handle the submission of the user input values to ForgeRock. First, add a second `useState` to track whether the user is authenticated or not, and then edit the current `Button` element, adding an `onPress` handler with a simple, inline function. This function should do the following:

1. Submit the modified `step` data to the server with the `FRAuthSampleBridge.next()` method.
2. Test if the response property `type` has the value of `'LoginSuccess'`.
3. If successful, parse the `response` JSON.
4. Call `setStep()` with the new object parsed from the JSON (this is mostly just for logging the step to the console).
5. Call `setAuthentication()` to true, which is a global state method that triggers the app to react (pun intended!) to the new user state.
6. Handle errors with a generic failure message.

```
@@ collapsed @@

  export default function Form() {
    const [step, setStep] = useState(null);
+   const [isAuthenticated, setAuthentication] = useState(false);
    console.log(step);

@@ collapsed @@

    return (
      <ScrollView>
        <Box safeArea flex={1} p={2} w="90%" mx="auto">
          <FormControl>
            {step.callbacks?.map(mapCallbacksToComponents)}
-         <Button>Sign In</Button>
+         <Button
+           onPress={() => {
+             async function getNextStep() {
+               try {
+                 const response = await FRAuthSampleBridge.next(
+                   JSON.stringify(step.payload),
+                 );
+                 if (response.type === 'LoginSuccess') {
+                   const accessInfo = JSON.parse(response.accessInfo);
+                   setStep({
+                     accessInfo,
+                     message: 'Successfully logged in.',
+                     type: 'LoginSuccess',
+                   });
+                   setAuthentication(true);
+                 } else {
+                   setStep({
+                     message: 'There has been a login failure.',
+                     type: 'LoginFailure',
+                   });
+                 }
+               } catch (err) {
+                 console.error(`Error: form submission; ${err}`);
+               }
+             }
+             getNextStep();
+           }}
+         >
+           Sign In
+         </Button>
        </FormControl>
      </Box>
    </ScrollView>
  );

@@ collapsed @@
```

After the app refreshes, use the test user to login. If successful, you should see a success message. Congratulations, you are now able to authenticate users!

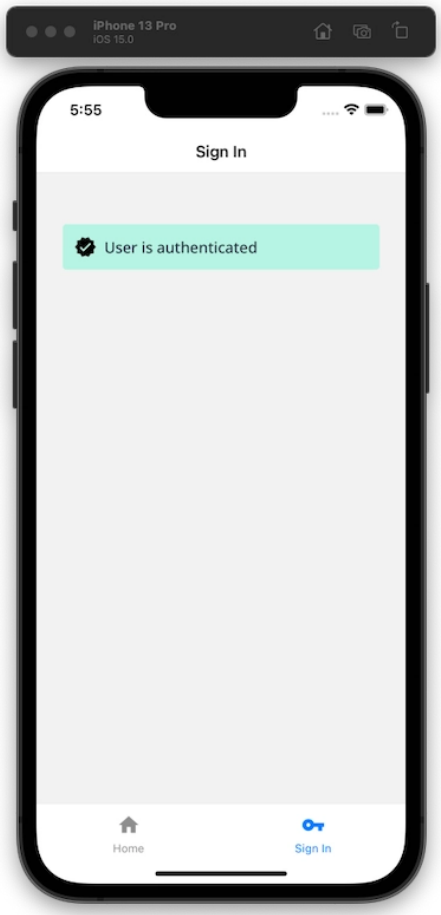


Figure 3. Login screen with successful authentication

What's more, you can verify the authentication details by going to the Xcode or Safari log and observing the result of the last call to the server. It should have a type of `"LoginSuccess"` along with token information.

Figure 4. Successful login response from Xcode

> **Note**
>
> If you got a login failure, you can re-attempt the login by going to the **Device** menu on the Simulator and selecting "Shake". This will allow you to reload the app, providing a fresh login form.

Handle the user provided values

You may ask, "How did the user's input values get added to the `step` object?" Let's take a look at the component for rendering the username input. Open up the `Text` component: `components/journey/text.js`. Notice how special methods are being used on the callback object. These are provided as convenience methods by the Ping SDK for JavaScript for getting and setting data.

```
@@ collapsed @@

  export default function Text({ callback }) {

@@ collapsed @@

    const error = handleFailedPolicies(
      callback.getFailedPolicies ? callback.getFailedPolicies() : [],
    );
    const isRequired = callback.isRequired ? callback.isRequired() : false;
    const label = callback.getPrompt();
    const setText = (text) => callback.setInputValue(text);
    return (
      <FormControl isRequired={isRequired} isInvalid={error}>
        <FormControl.Label mb={0}>{label}</FormControl.Label>
        <Input
          autoCapitalize="none"
          autoComplete="off"
          autoCorrect={false}
          onChangeText={setText}
          size="lg"
          type="text"
        />
        <FormControl.ErrorMessage>
          {error.length ? error : ''}
        </FormControl.ErrorMessage>
      </FormControl>
    );
  }
```

There are two important items to focus on

- `callback.getPrompt()` : Retrieves the input's label to be used in the UI.
- `callback.setInputValue()` : Sets the user's input on the callback while they are typing (i.e. `onChangeText`).

Since the `callback` is passed from the `Form` to the components by "reference" (not by "value"), any mutation of the `callback` object within the `Text` (or `Password`) component is also contained in the `step` object in the `Form` component.

> **Note**
>
> You may think, "That's not very idiomatic React! Shared, mutable state is bad." And, yes, you are correct, but we are taking advantage of this to keep everything simple (and this guide from being too long), so I hope you can excuse the pattern.

Each callback type has its own collection of methods for getting and setting data in addition to a base set of generic callback methods. The SDK automatically adds these methods to the callback's prototype. For more information about these callback methods, see our API documentation, or the source code in GitHub, for more details.

Request user info and redirecting to home screen

Now that the user can login, let's go one step further and request information about the authenticated user to display their name and other information. We will now utilize the existing `FRAuthSampleBridge.getUserInfo()` method already included in the bridge code.

Let's do a little setup before we make the request to the server:

1. Add `useContext` to the import from React so that we have access to the global state.
2. Import `AppContext` from the `global-state` module.
3. Call `useContext` with our `AppContext` to provide access to the setter methods.
4. Add one more `useEffect` function to detect the change of the user's authentication.

```
  import { FRStep } from '@forgerock/javascript-sdk';
  import { Box, Button, FormControl, ScrollView } from 'native-base';
- import React, { useEffect, useState } from 'react';
+ import React, { useContext, useEffect, useState } from 'react';
  import { NativeModules } from 'react-native';

+ import { AppContext } from '../../global-state';
@@ collapsed @@

  export default function Form() {
+   const [_, methods] = useContext(AppContext);
    const [step, setStep] = useState(null);
    const [isAuthenticated, setAuthentication] = useState(false);
    console.log(step);

    useEffect(() => {
      async function getStep() {
        try {
          await FRAuthSampleBridge.logout();
          const dataString = await FRAuthSampleBridge.login();
          const data = JSON.parse(dataString);
          const initialStep = new FRStep(data);
          setStep(initialStep);
        } catch (err) {
          console.error(`Error: request for initial step; ${err}`);
        }
      }
      getStep();
    }, []);

+   useEffect(() => {
+
+   }, [isAuthenticated]);

@@ collapsed @@
```

It's worth noting that the `isAuthenticated` declared in the array communicates to React that this `useEffect` should only execute if the state of that variable changes. This prevents unnecessary code execution since the value is initially `false`, and continues to be `false` until the user completes authentication.

With the setup complete, implement the request to the server for the user's information. Within this empty `useEffect`, add an `async` function to make that call to `FRAuthSampleBridge.getUserInfo()` and call it only when `isAuthenticated` is `true`.

```
@@ collapsed @@

    useEffect(() => {
+     async function getUserInfo() {
+       const userInfo = await FRAuthSampleBridge.getUserInfo();
+       console.log(userInfo);
+
+       methods.setName(userInfo.name);
+       methods.setEmail(userInfo.email);
+       methods.setAuthentication(true);
+     }
+
+     if (isAuthenticated) {
+       getUserInfo();
+     }
    }, [isAuthenticated]);

@@ collapsed @@
```

In the code above, we collected the user information and set a few values to the global state to allow the app to react to this information. In addition to updating the global state, the React Navigation also reacts to the global state change and renders the new screens and tab navigation.

When you test this in the Simulator, completing a successful authentication results in the home screen being rendered with a success message. The user's name and email are included for visual validation. You can also view the console in Safari and see the user's information logged.

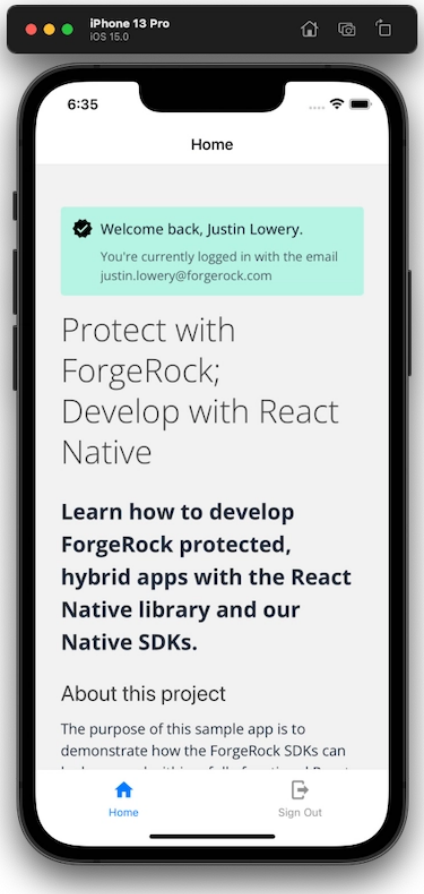


Figure 5. Home screen after successful authentication

Add logout functionality to our bridge and React Native code

Clicking the **Sign Out** button within the navigation results in the logout page rendering with a persistent "loading" spinner and message. This is due to the missing logic that we'll add now.

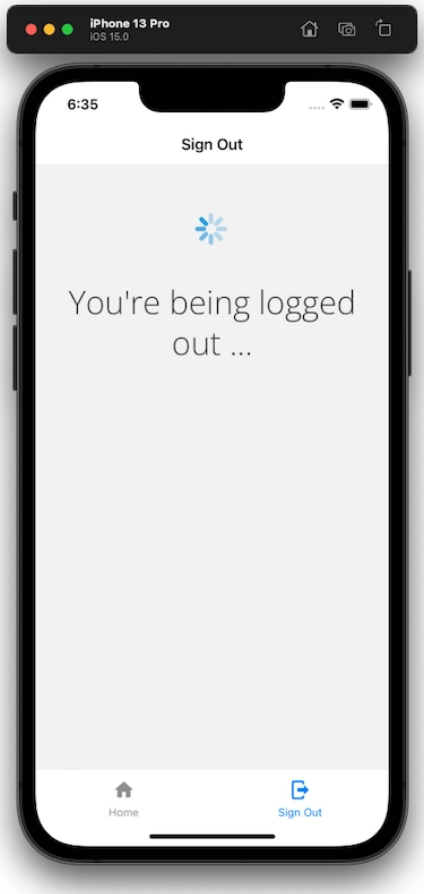


Figure 6. Logout screen with spinner

To add the logic into the view to call this new Swift method:

1. Open up the `screens/logout.js` file and import the following:
    1. `useEffect` and `useContext` from React
    2. `useHistory` from React Router
    3. `AppContext` from the global state module

```
- import React from 'react';
+ import React, { useContext, useEffect } from 'react';
+ import { NativeModules } from 'react-native';

+ import { AppContext } from '../global-state';
  import { Loading } from '../components/utilities/loading';

+ const { FRAuthSampleBridge } = NativeModules;

@@ collapsed @@
```

2. Since logging out requires an `async`, network request, we need to wrap it in a `useEffect` and pass in a callback function with the following functionality:

```
@@ collapsed @@

  export default function Logout() {
+   const [_, { setAuthentication }] = useContext(AppContext);
+
+   useEffect(() => {
+     async function logoutUser() {
+       try {
+         await FRAuthSampleBridge.logout();
+       } catch (err) {
+         console.error(`Error: logout; ${err}`);
+       }
+       setAuthentication(false);
+     }
+     logoutUser();
+   }, []);
+
    return <Loading message="You're being logged out ..." />;
  }
```

Since we only want to call this method once, after the component mounts, we will pass in an empty array as a second argument for `useEffect()`. The use of the `setAuthentication()` method empties or falsifies the global state to clean up and re-renders the home screen.

3. Revisit the app within the Simulator, and tap the **Sign Out** button.

   You should see a quick flash of the loading screen, and then the home screen should be displayed with the logged out UI state.

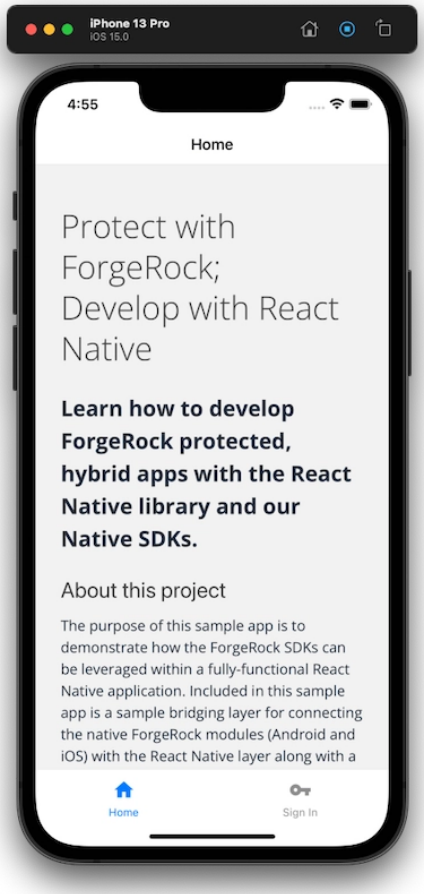


Figure 7. Logged out home screen

Testing the app

You should now be able to successfully authenticate a user, display the user's information, and log a user out.

Congratulations, you just built a protected iOS app with React Native.

Implement your use cases with the Ping SDKs

The SDKs enable you to implement many authentication, registration, and self-service use cases into your mobile and web apps.

Visit the following pages for more information on implementing different use cases using the Ping SDKs:

Set up PingOne Protect for risk evaluations

Applies to: Android | iOS | JavaScript

The Ping SDKs can integrate with PingOne Protect to evaluate the risk involved in a transaction.

Find out how to configure your application to use PingOne Protect.

Read more »

Set up user profile self service

Applies to: Android | iOS | JavaScript

View and edit user profile information, such as name, address, and marketing preferences.

Read more »

Set up registered device self service

Applies to: Android | iOS| JavaScript

View, rename, and delete user-registered devices.

Read more »

Set up mobile biometrics

Applies to: Android | iOS

Discover how to allow users to authenticate by using an authenticator device. For example, the fingerprint scanner on their laptop or a phone.

Leverage passkey support to synchronize across multiple devices.

Read more »

Set up web biometrics

Applies to: JavaScript

Discover how to allow users to authenticate by using WebAuthn.

Leverage passkey support to synchronize across multiple devices.

Read more »

Set up Device Profiling

Applies to: Android | iOS | JavaScript

Instruct your client applications to collect device profile information for decision-making in authentication journeys.

Read more »

Set up Social Login

Applies to: Android | iOS | JavaScript

Add support for authenticating to your apps by using trusted Identity Providers (IdP), like Apple, Facebook, and Google.

Read more »

Set up Magic Links

Applies to: Android | iOS | JavaScript

Learn how to pause a user's progress through an authentication tree, and later resume from the same point.

Read more »

Set up Transactional Authorization

Applies to: Android | iOS | JavaScript

Configure transactional authorization support in your app. Transactional authorization requires a user to authorize individual access attempts to specific protected resources.

It is part of an PingAM policy that grants single-use or one-shot access.

Read more »

Set up QR Code handling

Applies to: JavaScript

Learn how to handle callbacks that require a QR code to be displayed.

A number of journeys make use of QR codes, such as device registration for multi-factor authentication.

Read more »

Set up Google reCAPTCHA Enterprise

Applies to: Android | iOS | JavaScript

This tutorial shows how integrate with Google reCAPTCHA Enterprise.

Read more »

Integrate with PingOne Protect for risk evaluations

Applies to:

- ✓ Ping SDK for Android
- ✓ Ping SDK for iOS
- ✓ Ping SDK for JavaScript

The Ping SDKs can integrate with PingOne Protect to evaluate the risk involved in a transaction.

Important

PingOne Protect is supported in the following servers:

Advanced Identity Cloud

Use the official PingOne Protect nodes

PingAM 7.5 and later

Use the official PingOne Protect nodes

PingAM 7.2 - 7.4

Use the marketplace PingOne Protect nodes

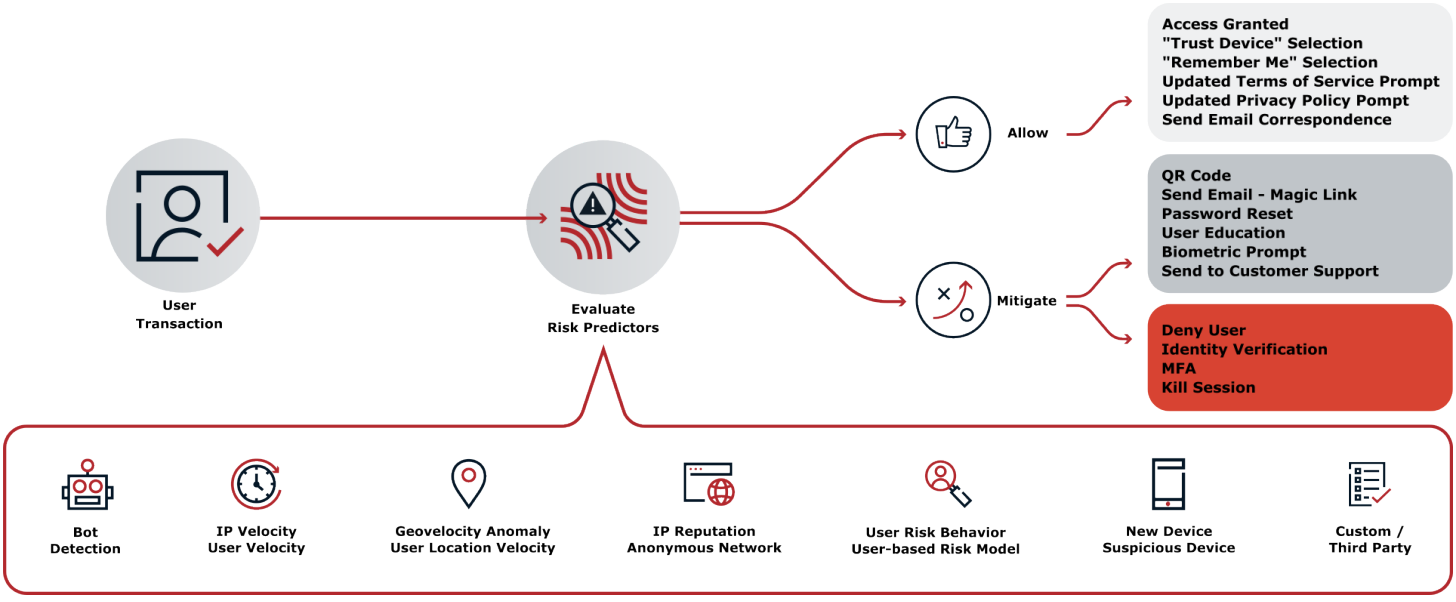


Figure 1. A flowchart illustrating how risk predictors evaluate many different data points.

You can instruct the Ping SDKs to use the embedded PingOne Signals SDK to gather information during a transaction. Your authentication journeys can then gather this information together and request a risk evaluation from PingOne.

Based on the response, you can choose whether to allow or deny the transaction or perform additional mitigation, such as bot detection measures.

You can use the audit functionality in PingOne to view the risk evaluations:

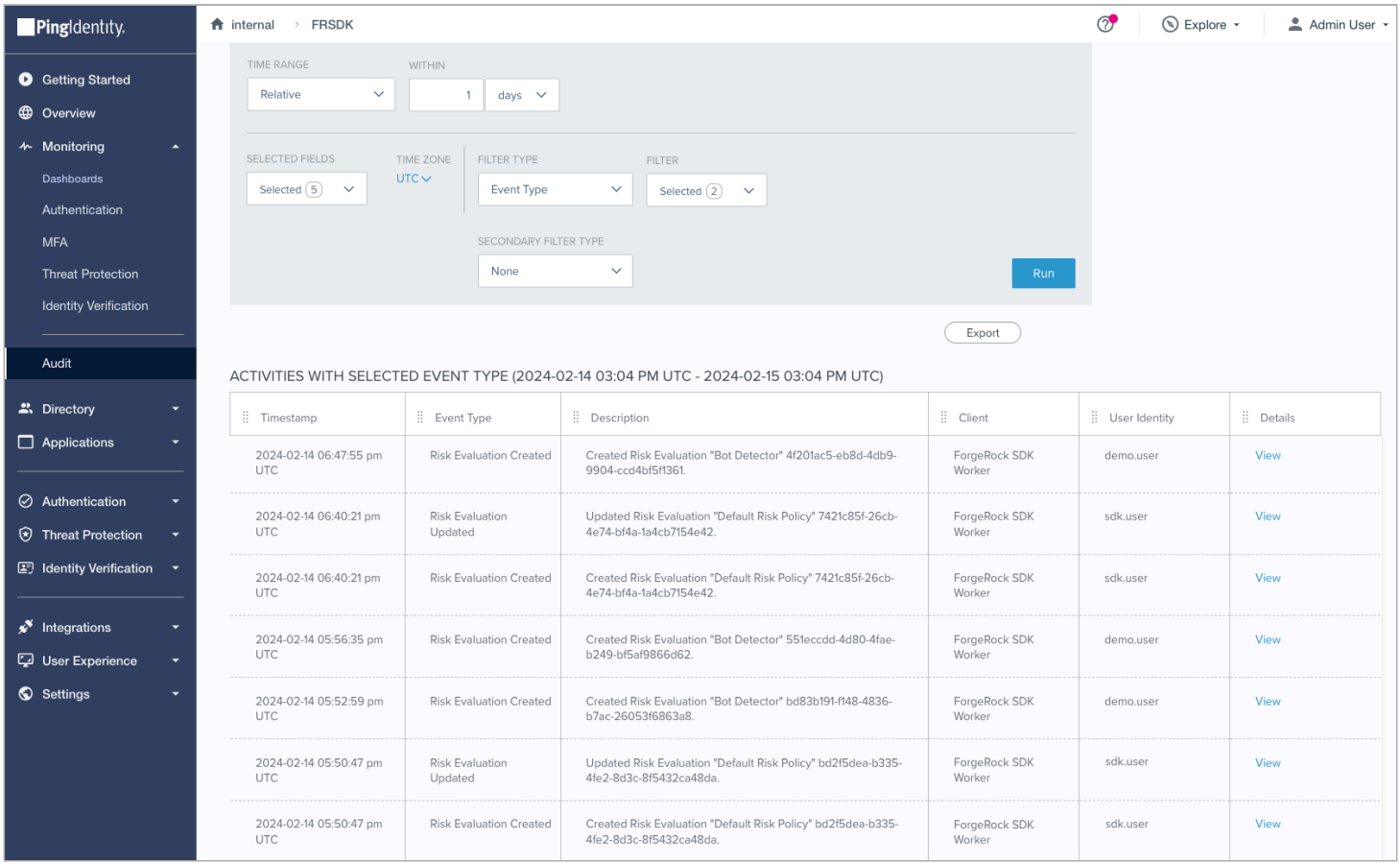


Figure 2. Risk evaluation records in the PingOne audit viewer.

Steps

Step 1. Set up the servers

In this step, you set up your PingOne Advanced Identity Cloud or PingAM server, and your PingOne instance to perform risk evaluations.

For example, you create a worker application in PingOne and configure your server to access it. You also create an authentication journey that uses the relevant nodes.

Step 2. Install dependencies

In this step, you add the required PingOne Protect module and dependencies to your project.

We provide instructions for Android, iOS, and JavaScript projects.

Step 3. Develop the client app

With everything prepared, you can now add Ping SDK code to your client application to evaluate risk by using PingOne Protect.

You'll learn how to initialize the collection of contextual data, gather and send it to the server for a risk evaluation, and how to pause and resume behavioral data collection.

Step 1. Set up the servers

In this step, you set up your PingOne Advanced Identity Cloud or PingAM server, and your PingOne instance to perform risk evaluations.

1. Create a worker application in PingOne
2. Configure the PingOne Worker service in your server
3. Configure a journey to perform PingOne Protect risk evaluations

Create a worker application in PingOne

To allow your server to access the PingOne administration API you must create a worker application in PingOne.

The worker application provides the client credentials your server uses to communicate with the PingOne admin APIs using the OpenID Connect protocol.

To create a worker application in PingOne:

1. In the PingOne administration console, navigate to **Applications** > **Applications**, and then click **Add** (+).
2. In the **Add Application** panel:
    1. In **Application name**, enter a unique identifier for the worker application.

       For example, `Ping SDK Worker`.
    2. Optionally, enter a **Description** for the application and select an **Icon**.

       These do not affect the operation of the worker application but do help you identify it in the list.
    3. In **Application Type**, select **Worker**.
    4. Click **Save**.
3. In the application properties panel for the worker application you created:
    1. On the **Roles** tab, click **Grant Roles**.
    2. On the **Available responsibilities** tab, select the **Identity Data Admin** row, and ensure the environment is correct.
    3. Click **Save**.
    4. On the **Overview** tab, ensure your worker application resembles the following image, and then enable it by using the toggle (1):

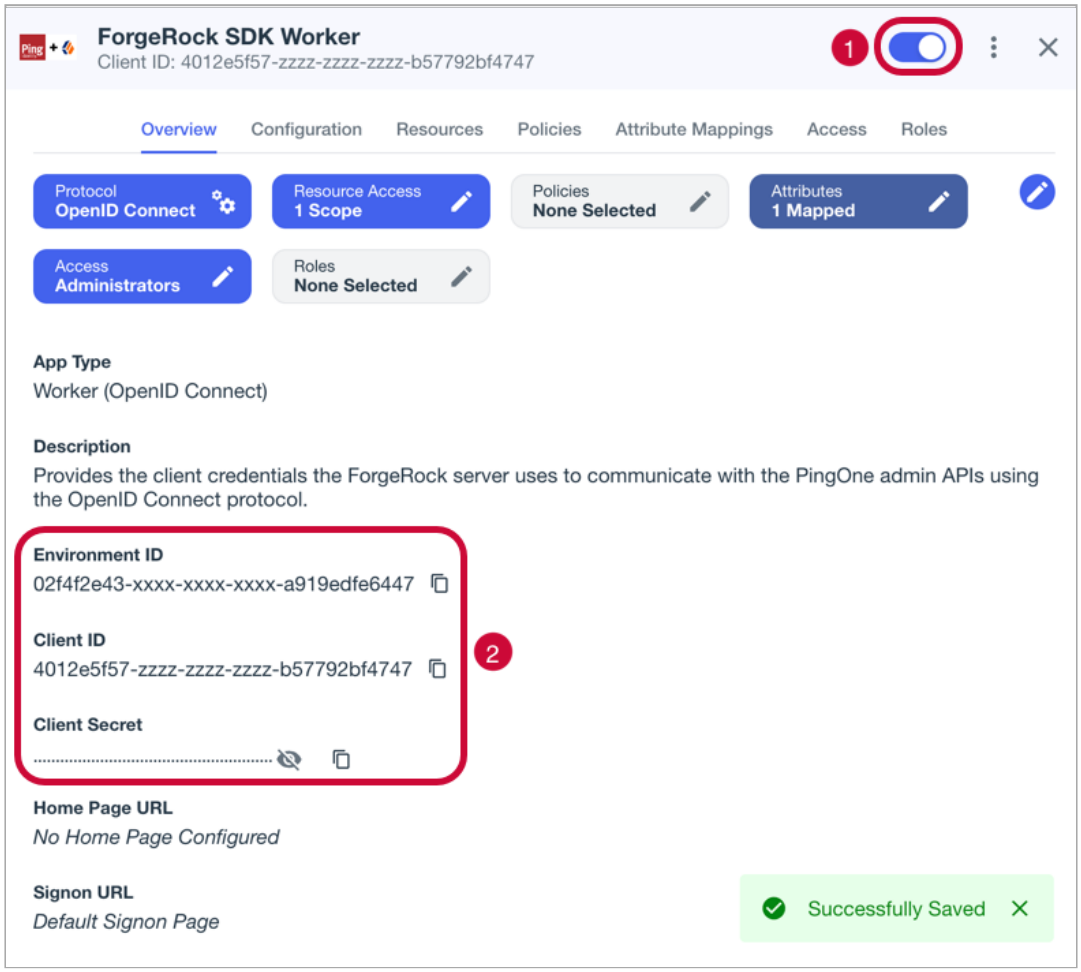


Figure 1. Example worker application in PingOne

5. Make a note of the **Environment ID**, **Client ID**, and **Client Secret** values (2).

   You need these values in the next step when you Configure the PingOne Worker service in your server.

Configure the PingOne Worker service in your server

After you create a worker application in PingOne, you must configure the PingOne Worker service in your server with the credentials.

Prerequisites

You need the following values from the PingOne Worker application you created in PingOne:

Client ID

Client ID of the worker application in PingOne.

Example: `6c7eb89a-66e9-ab12-cd34-eeaf795650b2`

Client Secret

Client secret of the worker application in PingOne.

> **Tip**
>
> Use the **Secret Mask** (👁) or **Copy to Clipboard** (📋) buttons to obtain the value in the PingOne administration console.

Example: `Ch15~o5Hm8N4_eS_m8~ARrV0KQAIQS6d.sJWe8TMXurEb~KWexY_p0gelR`

Environment ID

Identifier of the environment that contains the worker application in PingOne.

Example: `3072206d-c6ce-ch15-m0nd-f87e972c7cc3`

> **Important**
>
> The PingOne Worker Service requires a configured OAuth2 provider service in your server.
>
> - If you are using a self-managed AM server, you must configure the OAuth2 Provider service in a realm to expose the OAuth 2.0 endpoints and OAuth 2.0 administration REST endpoints.
> - The **OAuth2 provider service** is preconfigured in Advanced Identity Cloud.

Register the client secret in the server

You need to make the client secret of the worker application in PingOne available for use in the PingOne worker service.

Advanced Identity Cloud

If you are using Advanced Identity Cloud you will need to create an environment secret to hold the client secret value, as follows:

1. In the Advanced Identity Cloud admin UI, go to ⚙ **Tenant Settings** > **Global Settings** > **Environment Secrets & Variables**.
2. Click the **Secrets** tab.
3. Click **+ Add Secret**.
4. In the **Add a Secret** modal window, enter the following information:

Name	Enter a secret name. For example, `ping-protect-client-secret`. **Note** Secret names cannot be modified after the secret has been created.
Description	(optional) Enter a description of the purpose of the secret.
Value	Enter the **Client Secret** value you obtained when creating the worker application in PingOne. For example, `Ch15~o5Hm8N4_eS_m8~ARrV0KQAIQS6d.sJWe8TMXurEb~KWexY_p0gelR`. The field obscures the secret value by default. You can optionally click the visibility toggle (👁) to view the secret value as you enter it.

5. Click **Save** to create the variable.
6. Click **View Update**, check the details of the new secret, and then click **Apply Update**.

   Advanced Identity Cloud displays a final confirmation page.

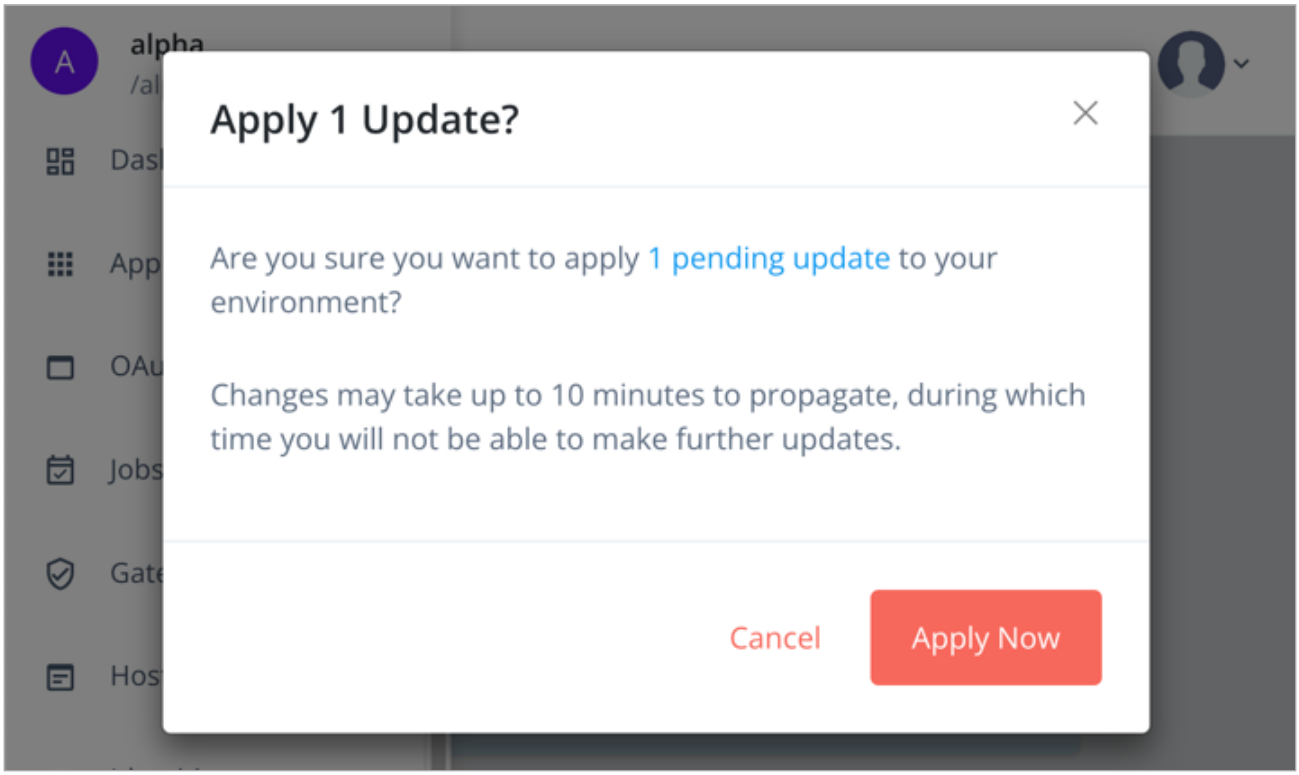


Figure 2. Apply updated secrets in Advanced Identity Cloud

7. Click **Apply Now**.

   Advanced Identity Cloud propagates the new secret and its value to all servers. You **must** wait until the secrets have propagated throughout the environment before attempting to use the secret.

   The **Environment Secrets & Variables** page displays the following message while the update is in progress:

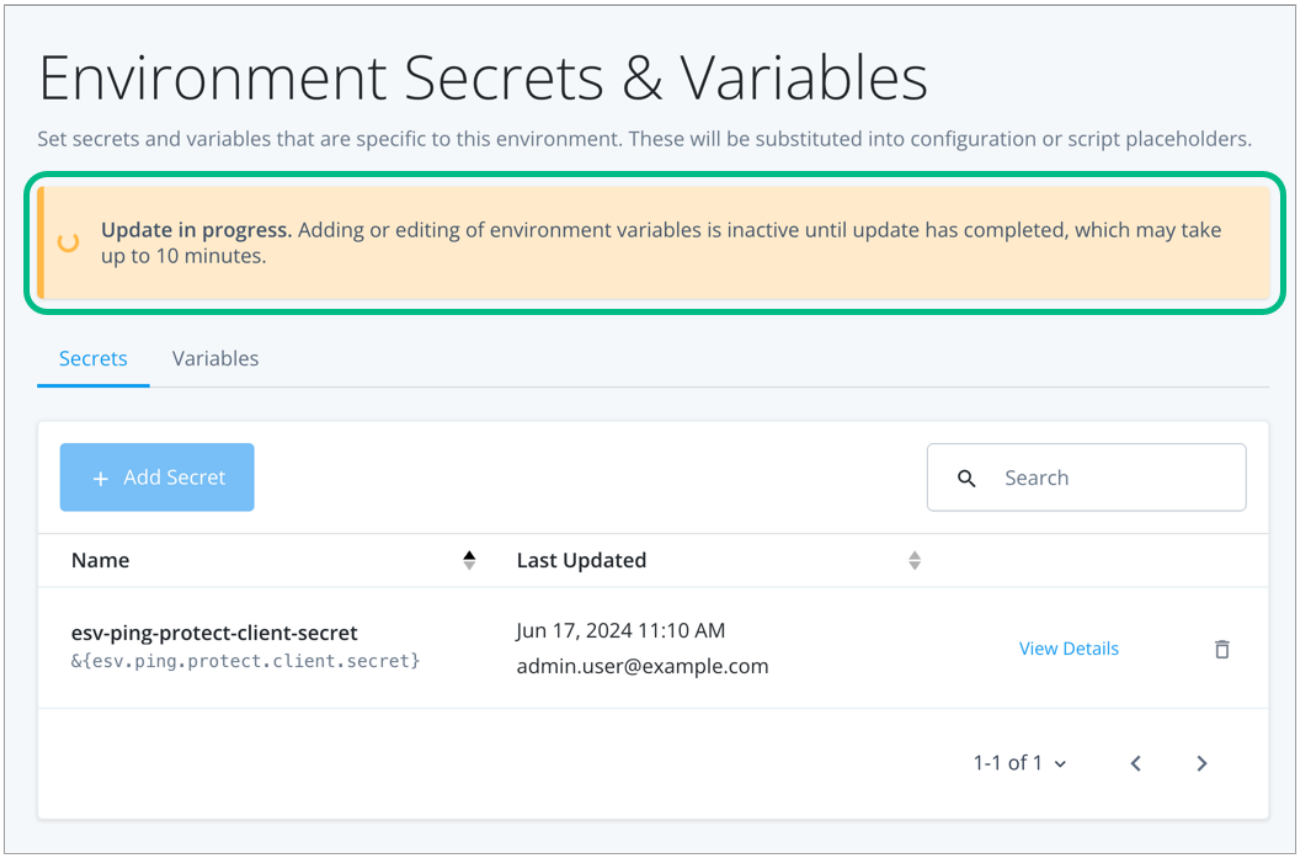


Figure 3. Propagating secrets in progress in Advanced Identity Cloud.

Self-managed AM

For information on adding secret values for use in services in a self-managed AM instance, refer to Create key aliases in the AM documentation.

Configure the PingOne worker service

To configure the PingOne worker service:

1. If you are using PingOne Advanced Identity Cloud, in the administration console navigate to **Native Consoles** > **Access Management**.
2. In the AM admin UI, click **Services**.
3. If the **PingOne Worker Service** is in the list of services, select it.
4. If you do not yet have a **PingOne Worker Service**:
    1. Click **+ Add a Service**.
    2. In **Choose a service type**, select `PingOne Worker Service`, and then click **Create**.
5. On the **Secondary Configurations** tab, click **+ Add a Secondary Configuration**.
6. On the **New workers configuration** page:
    1. Enter a **Name** for the configuration.

For example, `SDK PingOne Worker`.

You use this value when you configure an authentication journey that performs risk evaluations.

2. In **Client ID**, enter the client ID of the PingOne Worker application you created earlier.
3. In **Client Secret Label Identifier**, enter an identifier to create a specific secret label to represent the client secret of the worker application.

   For example, `workerAppClientSecret`.

   The secret label uses the template `am.services.pingone.worker.identifier.clientsecret` where identifier is the **Client Secret Label Identifier** value.

   This field can only contain characters `a-z`, `A-Z`, `0-9`, and `.` and can't start or end with a period.
4. In **Environment ID**, enter the environment ID containing the PingOne Worker application you created earlier.
5. Click **Create**

7. On the **Workers Configuration** page, ensure that the **PingOne API Server URL** and **PingOne Authorization Server URL** are correct for the region of your PingOne servers:

PingOne URLs by region

Region	Authorization URL	API URL
North America (Excluding Canada)	`https://auth.pingone.com`	`https://api.pingone.com/v1`
Canada	`https://auth.pingone.ca`	`https://api.pingone.ca/v1`
Europe	`https://auth.pingone.eu`	`https://api.pingone.eu/v1`
Asia-Pacific	`https://auth.pingone.asia`	`https://api.pingone.asia/v1`

8. Confirm your configuration resembles the image below, and then click **Save changes**.

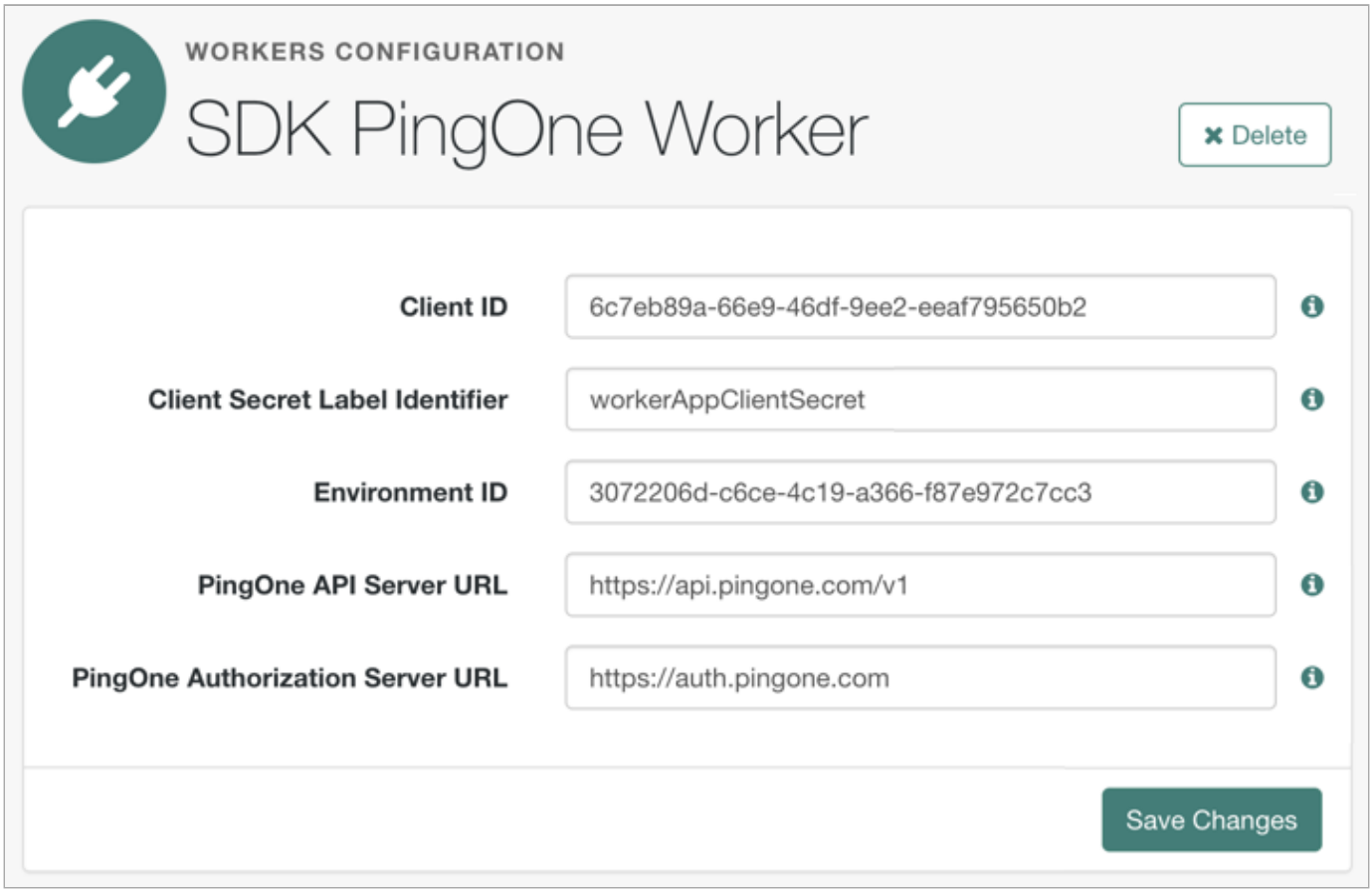


Figure 4. Example worker application in PingOne

Map the Client Secret Label Identifier to a secret

To make the client secret available to the PingOne Worker Service, you must map the secret to the ID created.

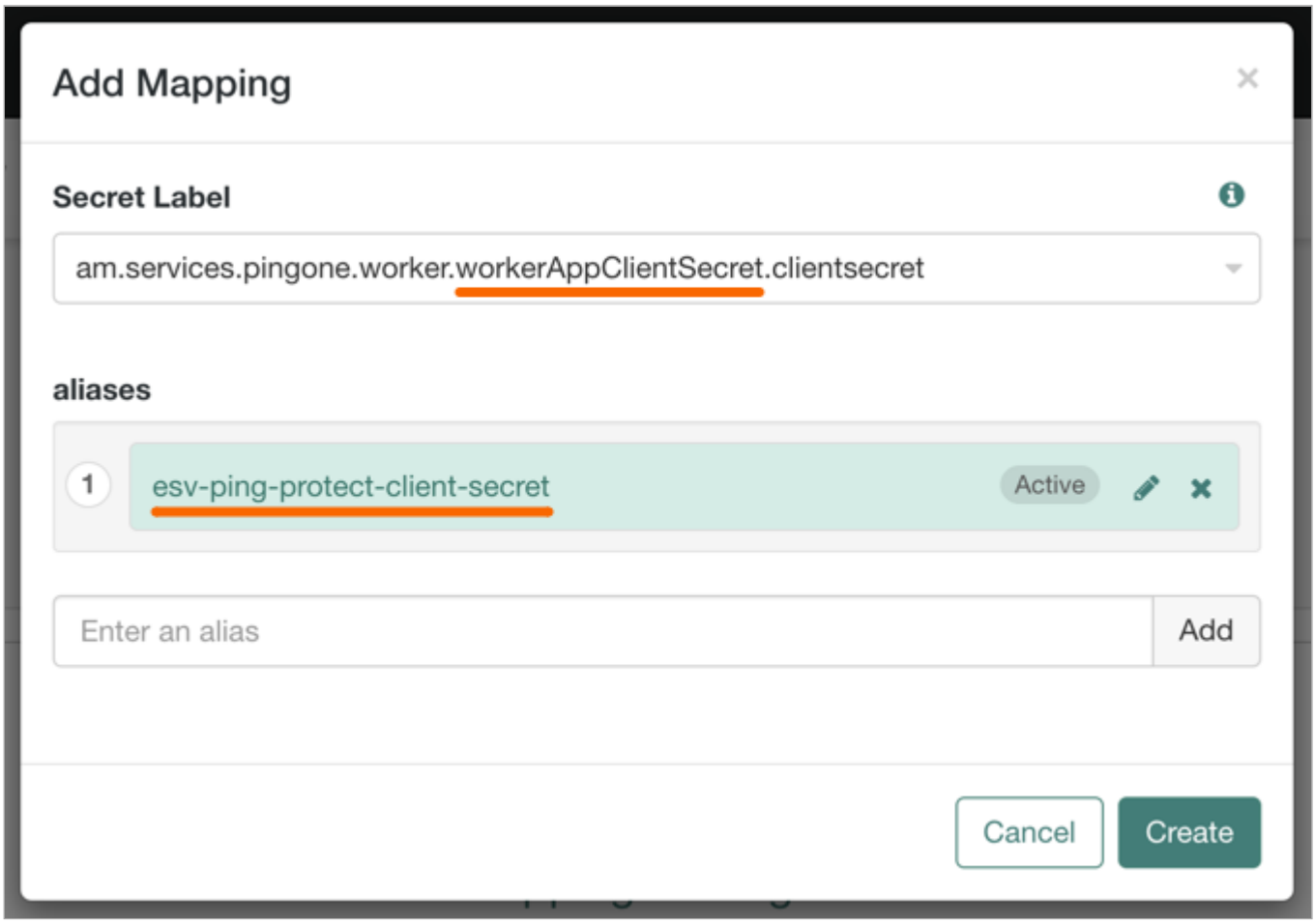
Map secrets in Advanced Identity Cloud

1. In the Advanced Identity Cloud admin UI, click **Native Consoles** > **Access Management**.
2. In the AM admin UI (native console), go to **Realm** > **Secret Stores**.
3. Click the **ESV** secret store, then click **Mappings**.
4. Click **+ Add Mapping**.
    1. In **Secret Label**, select the label generated when you entered the **Client Secret Label Identifier** previously.

       For example, `am.services.pingone.worker.workerAppClientSecret.clientsecret`.
    2. In **aliases**, enter the name of the ESV secret you created earlier, including the `esv-` prefix, and then click **Add**.

       For example, `esv-ping-protect-client-secret`

The result resembles the following:



5. Click **Create**.

To learn more about mapping secrets and label identifiers in Advanced Identity Cloud, refer to Secret labels.

Map secrets in self-managed AM

To learn about mapping secrets in self-managed AM, refer to Map and rotate secrets.

You have now configured the PingOne Worker service in your server. You can now Configure a journey to perform PingOne Protect risk evaluations.

Configure a journey to perform PingOne Protect risk evaluations

To make risk evaluations in PingOne, you must configure an authentication journey in your server.

The following table covers the authentication nodes and callbacks for integrating your authentication journeys with PingOne Protect.

Node	Callback	Description
PingOne Protect Initialization node	PingOneProtectInitiateCallback	Instruct the embedded PingOne Signals SDK to start gathering contextual information.
PingOne Protect Evaluation node	PingOneProtectEvaluationCallback	Returns contextual information that the server can send to your PingOne Protect instance to perform a risk evaluation.
PingOne Protect Result node	Non-interactive	Inform the PingOne Protect instance about the status of the transaction.

Note

These official PingOne Protect nodes are available in PingAM 7.5 and later, as well as PingOne Advanced Identity Cloud.
If you are using PingAM versions 7.2 to 7.4, you should instead use the equivalent PingOne Protect Marketplace nodes.
The PingOne Protect marketplace nodes use a `MetadataCallback` callback. The SDK recognizes the specific configuration the marketplace nodes place in this callback and can use it for use with PingOne Protect.

In your server, log in as an administrator and create a new authentication journey similar to the following example:

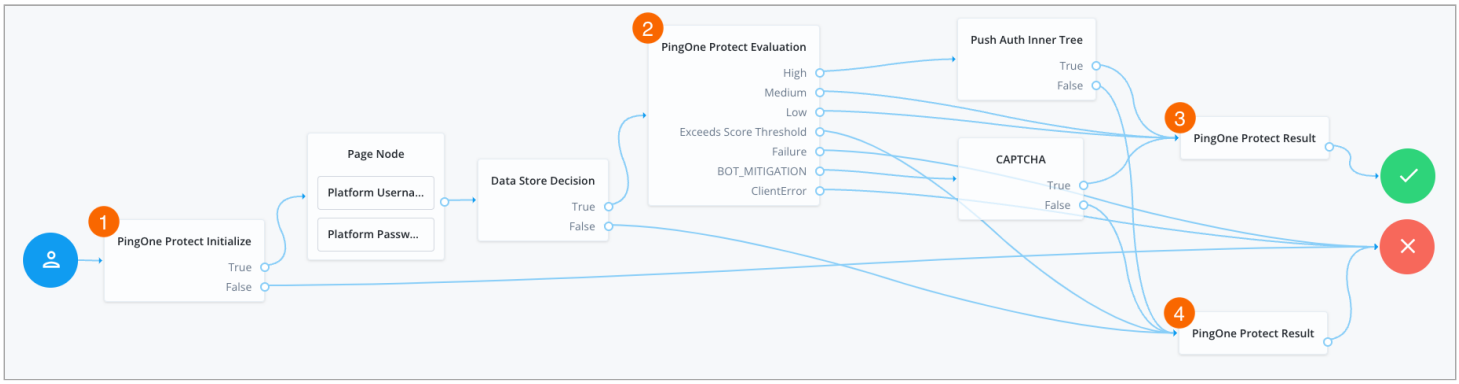


Figure 5. An example PingOne Protect journey

- The PingOne Protect Initialize node 1 instructs the SDK to initialize the PingOne Protect Signals API with the configured properties.

  Initialize the PingOne Protect Signals API as early in the journey as possible, before any user interaction.

  This enables it to gather sufficient contextual data to make an informed risk evaluation.

  Tip

  You can initialize the PingOne Protect Signals API whenever you want to start collecting data. This could be at application startup, or when a particular page or view is visited.
  Learn more at initializing data collection.

- The user enters their credentials, which are verified against the identity store.
- The PingOne Protect Evaluation node 2 performs a risk evaluation against a risk policy in PingOne.

  The example journey continues depending on the outcome:

  `High`

  The journey requests that the user respond to a push notification.

  `Medium` *or* `Low`

  The risk is not significant, so no further authentication factors are required.

  `Exceeds Score Threshold`

  The score returned is higher than the configured threshold and is considered too risky to complete successfully.

  `Failure`

  The risk evaluation could not be completed, so the authentication attempt continues to the **Failure** node.

  `BOT_MITIGATION`

  The risk evaluation returned a recommended action to check for the presence of a human, so the journey continues to a CAPTCHA node.

  `ClientError`

  The client returned an error when attempting to capture the data to perform a risk evaluation, so the authentication attempt continues to the **Failure** node.

- An instance of the PingOne Protect Result node 3 returns the `Success` result to PingOne, which can be viewed in the audit console to help with analysis and risk policy tuning.
- A second instance of the PingOne Protect Result node 4 returns the `Failed` result to PingOne, which can be viewed in the audit console to help with analysis and risk policy tuning.

You have now configured a suitable authentication journey in your server. You can now proceed to Step 2. Install dependencies.

Step 2. Install dependencies

To capture contextual data and perform risk evaluations, you must add the PingOne Protect module to your Ping SDK project.

Select your platform below for instructions on installing the required modules or dependencies:

Ping SDK for Android

Add the PingOne Protect dependencies to your Android project.

Ping SDK for iOS

Add the PingOne Protect dependencies to your iOS project by using Cocoapods or Swift Package Manager.

Ping SDK for JavaScript

Add the PingOne Protect dependencies to your JavaScript project by using npm.

Add Android dependencies

To add the PingOne Protect dependencies to your Android project:

1. In the **Project** tree view of your Android Studio project, open the `Gradle Scripts/build.gradle` file for the *module*.
2. In the `dependencies` section, add the required dependencies:

   Example `dependencies` section after editing:

```
dependencies {
    // Ping SDK main module
    implementation 'org.forgerock:forgerock-auth:4.8.1'

    // PingOne Protect module
    implementation 'org.forgerock:ping-protect:4.8.1'
}
```

After installing the module, you can proceed to Step 3. Develop the client app.

Add iOS dependencies

You can use CocoaPods or the Swift Package Manager to add the PingOne Protect dependencies to your iOS project.

Add dependencies using CocoaPods

1. If you do not already have CocoaPods, install the latest version.
2. If you do not already have a Podfile, in a terminal window, run the following command to create a new Podfile:

```
pod init
```

3. Add the following lines to your Podfile:

```
pod 'PingProtect' // Add-on for {p1p_name}
```

4. Run the following command to install pods:

```
pod install
```

Add dependencies using Swift Package Manager

1. With your project open in **Xcode**, select **File** > **Add Package Dependencies**.
2. In the search bar, enter the Ping SDK for iOS repository URL: `https://github.com/ForgeRock/forgerock-ios-sdk`.
3. Select the `forgerock-ios-sdk` package, and then click **Add Package**.
4. In the **Choose Package Products** dialog, ensure that the `PingProtect` library is added to your target project.
5. Click **Add Package**.
6. In your project, import the library:

```
// Import the {p1p_name} library
import PingProtect
```

After installing the module, you can proceed to Step 3. Develop the client app.

Add JavaScript dependencies

Install the PingOne Protect module by using npm:

```
npm install @forgerock/ping-protect
```

After installing the module, you can proceed to Step 3. Develop the client app.

Step 3. Develop the client app

Integrating your application with PingOne Protect enables you to perform risk evaluations during your customer's journey.

Add code for the following tasks to fully integrate with PingOne Protect:

1. Initialize data collection
2. Pause and resume behavioral data capture
3. Return collected data for a risk evaluation

Initialize data collection

You must initialize the PingOne Signals SDK so that it collects the data needed to evaluate risk.

The earlier you can initialize the PingOne Signals SDK, the more data it can collect to make a risk evaluation.

You can initialize the PingOne Signals SDK by using the `start()` method, which supports the following parameters:

Parameter			Description
Android	*iOS*	*JavaScript*	
`envID`			Required. Your PingOne environment identifier.
`deviceAttributesToIgnore`			Optional. A list of device attributes to ignore when collecting device signals. For example, `AUDIO_OUTPUT_DEVICES` or `IS_ACCEPT_COOKIES`.
`isBehavioralDataCollection`	`behavioralDataCollection`		When `true`, collect behavioral data. Default is `true`.
`isConsoleLogEnabled`	`consoleLogEnabled`		When `true`, output SDK log messages in the developer console. Default is `false`.
`isLazyMetadata`	`lazyMetadata`		When `true`, calculate metadata on demand rather than automatically after calling `start`. Default is `false`.
N/A		`deviceKeyRsyncIntervals`	Number of days that device attestation can rely upon the device fallback key. Default: `14`
N/A		`disableHub`	When `true`, the client stores device data in the browser's `localStorage` only. When `false` the client uses an iframe. Default is `false`.

N/A	`disableTags`	When `true`, the client does not collect tag data. Tags are used to record the pages the user visited, forming a browsing history. Default is `false`.
N/A	`enableTrust`	When `true`, tie the device payload to a non-extractable crypto key stored in the browser for content authenticity verification. Default is `false`.
N/A	`externalIdentifiers`	Optional. A list of custom identifiers that are associated with the device entity in PingOne Protect.
N/A	`hubUrl`	Optional. The iframe URL to use for cross-storage device IDs.
N/A	`waitForWindowLoad`	When `true`, initialize the SDK on the `load` event, instead of the `DOMContentLoaded` event. Default is `true`.

There are two options for initializing the PingOne Signals SDK:

1. Initialize manually
2. Initialize based on a callback

Initialize manually

Call the `start()` method before users start interacting with your application to gather the most data and make the most informed risk evaluations.

Pass in the configuration parameters as required.

Android

```
try {
  val params =
    PIInitParams(
      envId = "3072206d-c6ce-ch15-m0nd-f87e972c7cc3",
    )
  PIProtect.start(context, params)
  Logger.info("Settings Protect", "Initialize succeeded")
} catch (e: Exception) {
  Logger.error("Initialize Error", e.message)
  throw e
}
```

iOS

```
let initParams = PIInitParams(envId: "3072206d-c6ce-ch15-m0nd-f87e972c7cc3")
  PIProtect.start(initParams: initParams) { error in
    if let error = error as? NSError {
        FRLog.e("Initialize error: \(error.localizedDescription)")
    } else {
        FRLog.i("Initialize succeeded")
    }
  }
```

JavaScript

```
import { PIProtect } from '@forgerock/ping-protect';

try {
  // Initialize PingOne Protect with manual configuration
  PIProtect.start({ envId: '3072206d-c6ce-ch15-m0nd-f87e972c7cc3' });
} catch (err) {
  console.error(err);
}
```

Initialize based on a callback

Not all authentication journeys perform risk evaluations, and therefore do not need to initialize data collection. You can choose to initialize capture of data on receipt of the `PingOneProtectInitializeCallback` callback rather than during app start up.

The callback also provides the configuration parameters.

Android

```
try {
  val callback =
    node.getCallback(PingOneProtectInitializeCallback::class.java)
  callback.start(context)
} catch (e: PingOneProtectInitException) {
  Logger.error("PingOneInitException", e, e.message)
} catch (e: Exception) {
  Logger.error("PingOneInitException", e, e.message)
  callback.setClientError(e.message);
}
node.next()
```

iOS

```
if callback.type == "PingOneProtectInitializeCallback",
  let pingOneProtectInitCallback = callback as? PingOneProtectInitializeCallback
{
  pingOneProtectInitCallback.start { result in
    DispatchQueue.main.async {
      var initResult = ""
      switch result {
      case .success:
        initResult = "Success"
      case .failure(let error):
        initResult = "Error: \(error.localizedDescription)"
      }
      FRLog.i("{p1p_name} Initialize Result: \n\(initResult)")
      handleNode(node)
    }
  }
  return
}
```

JavaScript

```
import { PIProtect } from '@forgerock/ping-protect';

if (step.getCallbacksOfType('PingOneProtectInitializeCallback')) {
  const callback = step.getCallbackOfType('PingOneProtectInitializeCallback');

  // Obtain config properties from the callback
  const config = callback.getConfig();

  console.log(JSON.stringify(config));

  try {
    // Initialize {p1p_name} with configuration from callback
    await PIProtect.start(config);
  } catch (err) {
    // Add any errors to the callback
    callback.setClientError(err.message);
  }
}

FRAuth.next(step);
```

Pause and resume behavioral data capture

The PingOne Protect Signals SDK can capture behavioral data, such as how the user interacts with the app, to help when performing evaluations.

There are scenarios where you might want to pause the collection of behavioral data. For example, the user might not be interacting with the app, or you only want to use device attribute data to be considered when performing PingOne Protect evaluations. You can then resume behavioral data collection when required.

The SDKs provide the `pauseBehavioralData()` and `resumeBehavioralData()` methods for pausing and resuming the capture of behavioral data.

The `PingOneProtectEvaluationCallback` callback can include a flag to pause or resume behavioral capture that you should respond to as follows:

Android

```
val callback =
  node.getCallback(PingOneProtectEvaluationCallback::class.java)

const shouldPause = callback.pauseBehavioralData

Logger.info("PingOneProtectEvaluationCallback", "getPauseBehavioralData: ${shouldPause}")

if (shouldPause) {
  PIProtect.pauseBehavioralData()
}
```

iOS

```
if callback.type == "PingOneProtectEvaluationCallback",
  let pingOneProtectEvaluationCallback = callback as? PingOneProtectEvaluationCallback
{
  if let shouldPause = pingOneProtectEvaluationCallback.pauseBehavioralData, shouldPause {
    PIProtect.pauseBehavioralData()
  }
}
```

JavaScript

```
const callback = step.getCallbackOfType('PingOneProtectEvaluationCallback');
const shouldPause = callback.getPauseBehavioralData();

console.log(`getPauseBehavioralData: ${shouldPause}`);

if (shouldPause) {
  PIProtect.pauseBehavioralData();
}
```

Return collected data for a risk evaluation

To perform risk evaluations, the PingOne server requires the captured data.

On receipt of a `PingOneProtectEvaluationCallback` callback, use the `getData()` method to populate the response with the captured data.

Android

```
try {
  val callback =
    node.getCallback(PingOneProtectEvaluationCallback::class.java)
  callback.getData(context)
} catch (e: PingOneProtectEvaluationException) {
  Logger.error("PingOneRiskEvaluationCallback", e, e.message)
} catch (e: Exception) {
  Logger.error("PingOneRiskEvaluationCallback", e, e.message)
}
```

iOS

```
if callback.type == "PingOneProtectEvaluationCallback",
  let pingOneProtectEvaluationCallback = callback as? PingOneProtectEvaluationCallback
{
  pingOneProtectEvaluationCallback.getData { result in
    DispatchQueue.main.async {
      var evaluationResult = ""
      switch result {
      case .success:
        evaluationResult = "Success"
      case .failure(let error):
        evaluationResult = "Error: \(error.localizedDescription)"
      }
      FRLog.i("{p1p_name} Evaluation Result: \n\(evaluationResult)")
      handleNode(node)
    }
  }
  return
}
```

JavaScript

```
let data;

if (step.getCallbacksOfType('PingOneProtectEvaluationCallback')) {
  const callback = step.getCallbackOfType('PingOneProtectEvaluationCallback');
  try {
    // Asynchronous call
    data = await PIProtect.getData();
  } catch (err) {
    // Add any errors to the callback
    callback.setClientError(err.message);
  }
}
callback.setData(data);
FRAuth.next(step);
```

Set up user profile self service

Applies to:

- ✓ Ping SDK for Android
- ✓ Ping SDK for iOS
- ✓ Ping SDK for JavaScript

The Ping SDKs support many of the callbacks used by authentication journeys, including ones that enable your users to manage their own profile information, such as name, address, phone numbers and marketing preferences.

To update a user's profile information you must have already authenticated them and issued a session token. You can then use that session token to start a new journey which allows the user to update their profile data.

Compatibility

PingIDM is responsible for profile management. Therefore this tutorial is only compatible with the following server environments:

- PingOne Advanced Identity Cloud
- PingAM and PingIDM deployed together as the Ping Identity Platform (ForgeRock Identity Platform)
- PingAM and PingIDM deployed together by using ForgeRock DevOps (ForgeOps)

Before you begin

You must create an authentication journey that checks for the presence of a user session and then displays the user profile fields for editing. The journey must also update the profile with any changed values.

Create a user profile management journey

Follow the steps below to create a user profile management journey:

1. Create a new journey or tree and give it a name:
    1. In your PingOne Advanced Identity Cloud tenant, navigate to **Journeys**, and click **+ New Journey**.
    2. Enter a name, such as `sdkProfileManagement` and click **Save**.

       The authentication journey designer appears.

    1. Under **Realm Overview**, click **Authentication Trees**, then click **Create Tree**.
    2. Enter a tree name, for example `sdkProfileManagement`, and then click **Create**.

       The authentication tree designer appears, showing the **Start** entry point connected to the **Failure** exit point.
2. Drag the following nodes into the designer area:
    - **Get Session Data**
    - **Attribute Collector**
    - **Patch Object**
    - **Data Store Decision**
3. Connect the nodes as follows:

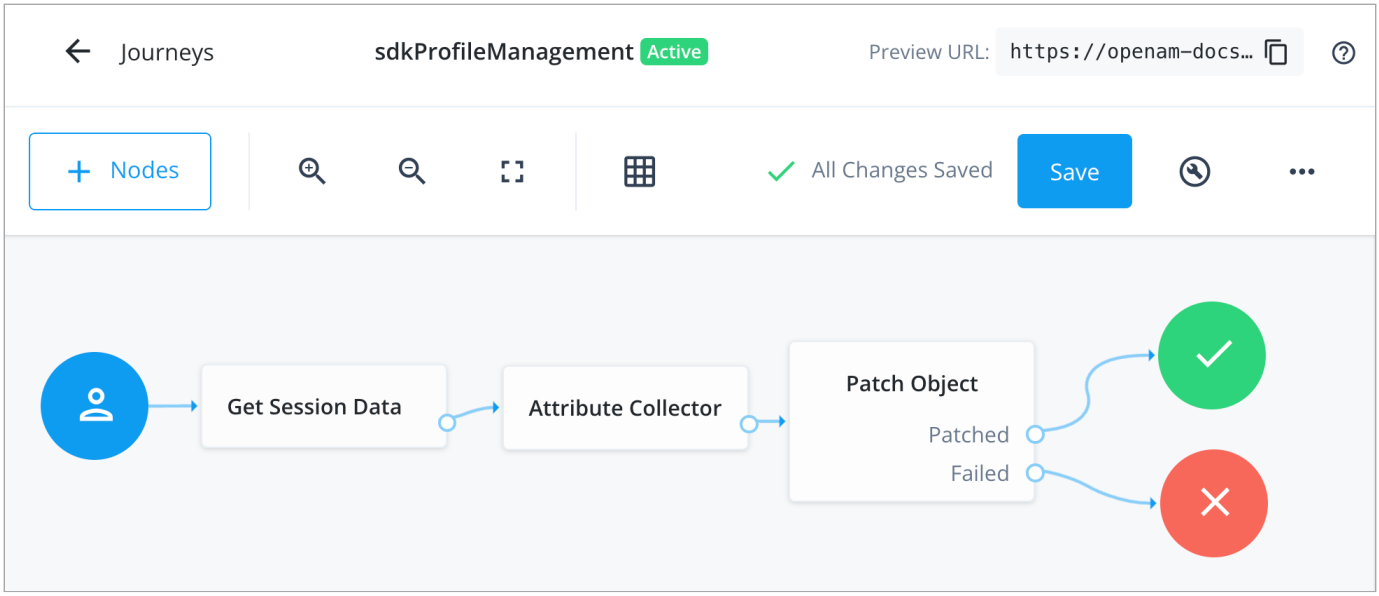


Figure 1. Example profile management authentication journey

4. Select the **Get Session Data** node and configure it to obtain the user's account name from the session and store it in shared state, as follows:
    1. In **Session Data Key**, enter `UserToken`.

> **Important**
>
> This field is case-sensitive. The value must exactly match the name of a property in the user's session. For a list of properties, refer to Get Session Data node.

2. In **Shared State Key**, enter `userName`.

The result resembles the following:

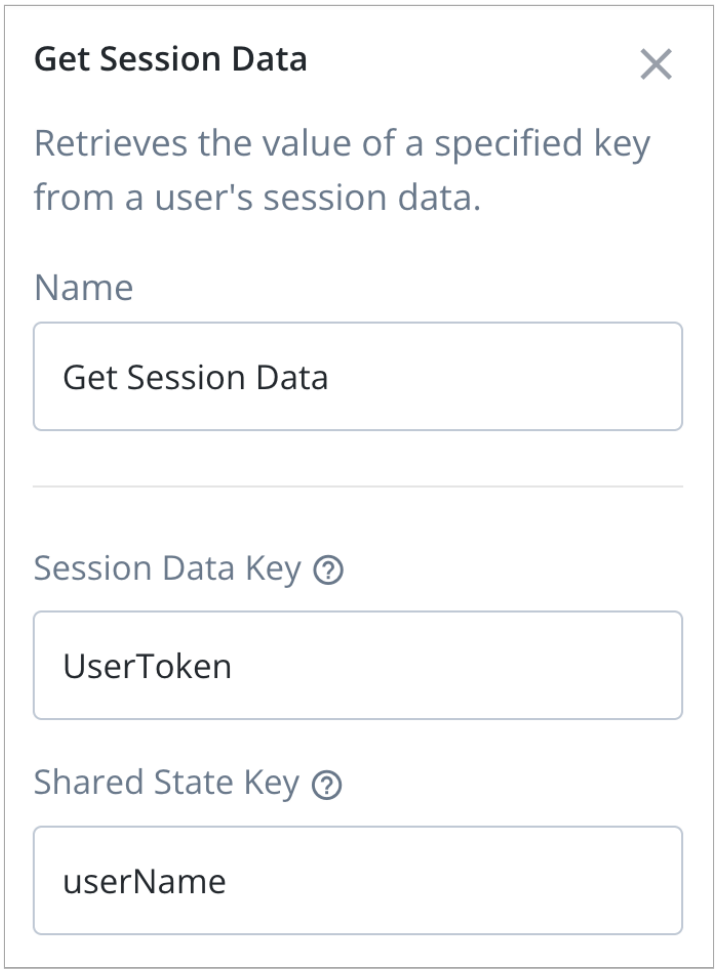


Figure 2. Configure the Get Session Data node for profile management.

5. Select the **Attribute Collector** node and configure it with the profile attributes you want the user to view and edit:

   1. In **Attributes to Collect**, enter the profile attributes to display. For example:

      - `givenName`
      - `sn`
      - `mail`
      - `telephoneNumber`
      - `postalAddress`
      - `city`

- `country`
- `preferences/marketing`

2. In **Identity Attribute**, enter `userName`.

The result resembles the following:

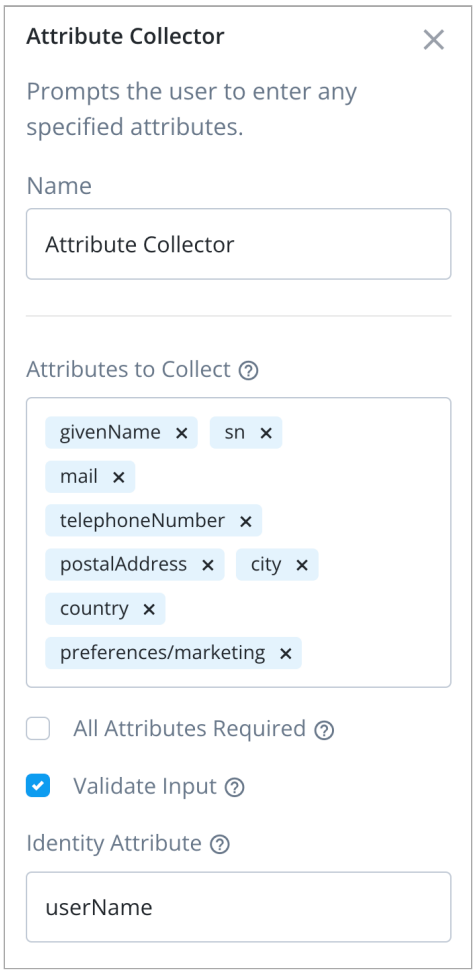


Figure 3. Configure the Attribute Collector node for profile management.

6. Select the **Patch Object** node and configure it to update the user's profile:

1. In **Identity Resource**, enter `managed/alpha_user`.
2. In **Identity Attribute**, enter `userName`.

The result resembles the following:

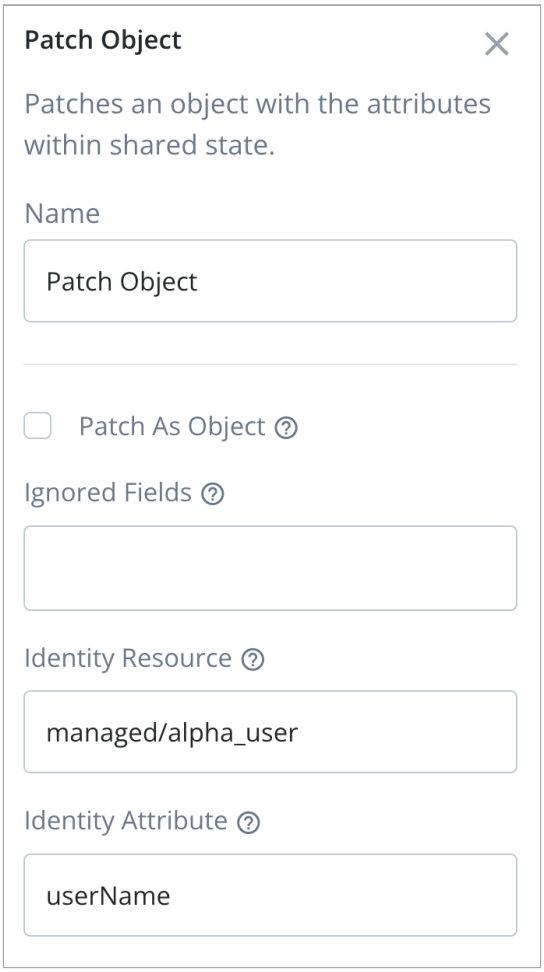


Figure 4. Configure the Patch Object node for profile management.

7. Click **Save**.

Server configuration

This tutorial requires you to configure one of the following servers:

PingOne Advanced Identity Cloud

PingOne Advanced Identity Cloud

PingAM

PingAM

PingOne Advanced Identity Cloud

The samples and tutorials in this documentation often require that you have an identity set up so that you can test authentication.

To create a demo user in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. In the left panel, click **Identities** > **Manage**.

3. Click **+ New Alpha realm - User**.
4. Enter the following details:
    - **Username** = `demo`
    - **First Name** = `Demo`
    - **Last Name** = `User`
    - **Email Address** = `demo.user@example.com`
    - **Password** = `Ch4ng3it!`
5. Click **Save**.

Authentication journeys provide fine-grained authentication by allowing multiple paths and decision points throughout the flow. Authentication journeys are made up of nodes that define actions taken during authentication.

Each node performs a single task, such as collecting a username or making a simple decision. Nodes can have multiple outcomes rather than just success or failure. For details, see the Authentication nodes configuration reference in the PingAM documentation.

To create a simple journey for use when testing the Ping SDKs, follow these steps:

1. In your PingOne Advanced Identity Cloud tenant, navigate to **Journeys**, and click **+ New Journey**.
2. Enter a name, such as `sdkUsernamePasswordJourney` and click **Save**.

   The authentication journey designer appears.
3. Drag the following nodes into the designer area:
    - **Page Node**
    - **Platform Username**
    - **Platform Password**
    - **Data Store Decision**
4. Drag and drop the **Platform Username** and **Platform Password** nodes onto the **Page Node**, so that they both appear on the same page when logging in.
5. Connect the nodes as follows:

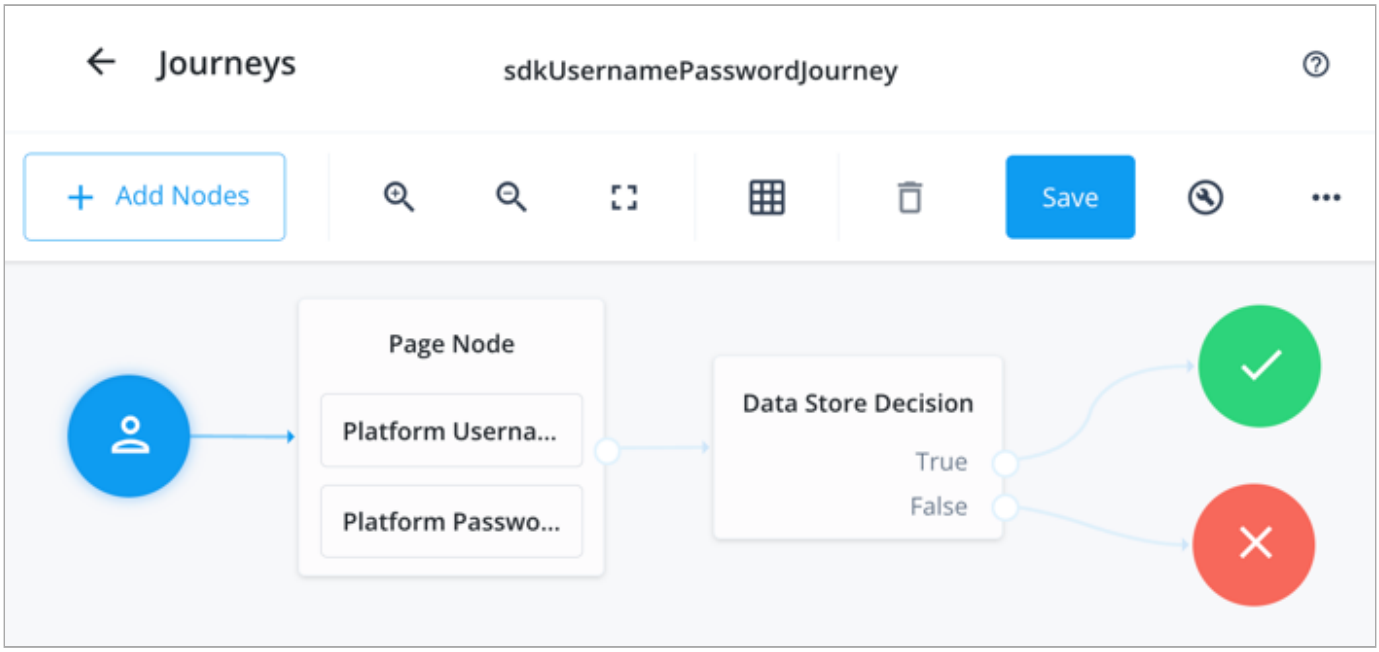


Figure 5. Example username and password authentication journey

6. Click **Save**.

Public clients do not use a client secret to obtain tokens because they are unable to keep them hidden. The Ping SDKs commonly use this type of client to obtain tokens, as they cannot guarantee safekeeping of the client credentials in a browser or on a mobile device.

To register a *public* OAuth 2.0 client application for use with the SDKs in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. In the left panel, click **Applications**.
3. Click **+ Custom Application**.
4. Select **OIDC - OpenId Connect** as the sign-in method, and then click **Next**.
5. Select **Native / SPA** as the application type, and then click **Next**.
6. In **Name**, enter a name for the application, such as `Public SDK Client`.
7. In **Owners**, select a user that is responsible for maintaining the application, and then click **Next**.

> **Tip**
>
> When trying out the SDKs, you could select the `demo` user you created previously.

8. In **Client ID**, enter `sdkPublicClient`, and then click **Create Application**.

   PingOne Advanced Identity Cloud creates the application and displays the details screen.

9. On the **Sign On** tab:

   1. In **Sign-In URLs**, enter the following values:

      `org.forgerock.demo://oauth2redirect`

> **Important**
>
> Also add any other domains where you host SDK applications.

2. In **Grant Types**, enter the following values:

   `Authorization Code`

   `Refresh Token`

3. In **Scopes**, enter the following values:

   `openid profile email address`

10. Click Show advanced settings, and on the **Authentication** tab:
    1. In **Token Endpoint Authentication Method**, select `none`.
    2. In **Client Type**, select `Public`.
    3. Enable the **Implied Consent** property.
11. Click **Save**.

The application is now configured to accept client connections from and issue OAuth 2.0 tokens to the example applications and tutorials covered by this documentation.

The provider specifies the supported OAuth 2.0 configuration options for a realm.

To ensure the PingOne Advanced Identity Cloud OAuth 2.0 provider service is configured for use with the Ping SDKs, follow these steps:

1. In your PingOne Advanced Identity Cloud tenant, navigate to **Native Consoles** > **Access Management**.
2. In the left panel, click **Services**.
3. In the list of services, click **OAuth2 Provider**.
4. On the **Core** tab, ensure **Issue Refresh Tokens** is enabled.
5. On the **Consent** tab, ensure **Allow Clients to Skip Consent** is enabled.
6. Click **Save Changes**.

PingAM

The samples and tutorials in this documentation often require that you have an identity set up so that you can test authentication.

To create a demo user in PingAM, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. Navigate to **Identities**, and then click **+ Add Identity**.
3. Enter the following details:
   - **User ID** = `demo`

- **Password** = `Ch4ng3it!`
- **Email Address** = `demo.user@example.com`

4. Click **Create**.

Authentication trees provide fine-grained authentication by allowing multiple paths and decision points throughout the authentication flow. Authentication trees are made up of nodes that define actions taken during authentication.

Each node performs a single task, such as collecting a username or making a simple decision. Nodes can have multiple outcomes rather than just success or failure. For details, see the Authentication nodes configuration reference in the PingAM documentation.

To create a simple tree for use when testing the Ping SDKs, follow these steps:

1. Under **Realm Overview**, click **Authentication Trees**, then click **Create Tree**.
2. Enter a tree name, for example `sdkUsernamePasswordJourney`, and then click **Create**.

   The authentication tree designer appears, showing the **Start** entry point connected to the **Failure** exit point.
3. Drag the following nodes from the **Components** panel on the left side into the designer area:
   - **Page Node**
   - **Username Collector**
   - **Password Collector**
   - **Data Store Decision**
4. Drag and drop the **Username Collector** and **Password Collector** nodes onto the **Page Node**, so that they both appear on the same page when logging in.
5. Connect the nodes as follows:

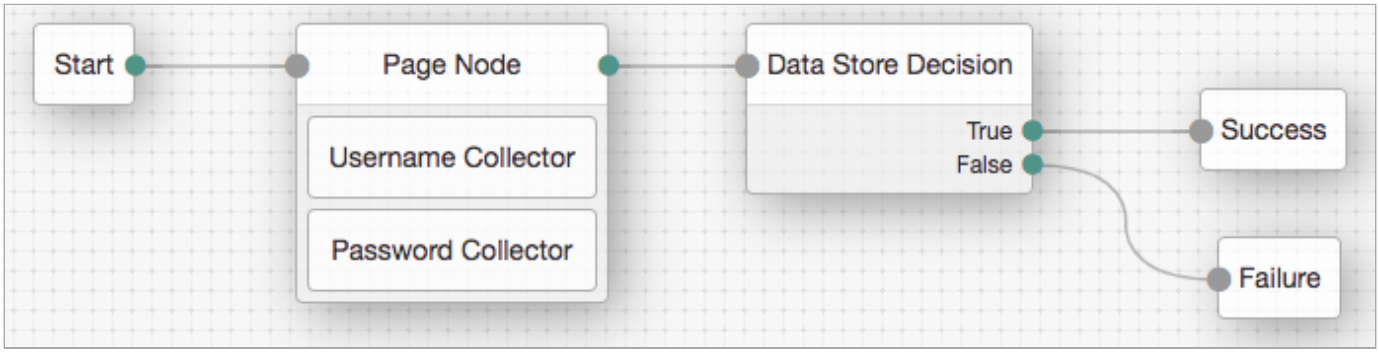


Figure 6. Example username and password authentication tree

6. Select the **Page Node**, and in the **Properties** pane, set the **Stage** property to `UsernamePassword`.

> **Tip**
>
> You can configure the node properties by selecting a node and altering properties in the right-hand panel.

One of the samples uses this specific value to determine the custom UI to display.

7. Click **Save**.

Public clients do not use a client secret to obtain tokens because they are unable to keep them hidden. The Ping SDKs commonly use this type of client to obtain tokens, as they cannot guarantee safekeeping of the client credentials in a browser or on a mobile device.

To register a *public* OAuth 2.0 client application for use with the SDKs in AM, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. Navigate to **Applications** > **OAuth 2.0** > **Clients**, and then click **+ Add Client**.
3. In **Client ID**, enter `sdkPublicClient`.
4. Leave **Client secret** empty.
5. In **Redirection URIs**, enter the following values:

   `org.forgerock.demo://oauth2redirect`

   > **Important**
   >
   > Also add any other domains where you will be hosting SDK applications.

6. In **Scopes**, enter the following values:

   `openid profile email address`

7. Click **Create**.

   PingAM creates the new OAuth 2.0 client, and displays the properties for further configuration.

8. On the **Core** tab:
   1. In **Client type**, select `Public`.
   2. Disable **Allow wildcard ports in redirect URIs**.
   3. Click **Save Changes**.
9. On the **Advanced** tab:
   1. In **Grant Types**, enter the following values:

      ```
      Authorization Code
      Refresh Token
      ```

   2. In **Token Endpoint Authentication Method**, select `None`.
   3. Enable the **Implied consent** property.
10. Click **Save Changes**.

The provider specifies the supported OAuth 2.0 configuration options for a realm.

To ensure the PingAM OAuth 2.0 provider service is configured for use with the Ping SDKs, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. In the left panel, click **Services**.
3. In the list of services, click **OAuth2 Provider**.
4. On the **Core** tab, ensure **Issue Refresh Tokens** is enabled.
5. On the **Consent** tab, ensure **Allow Clients to Skip Consent** is enabled.
6. Click **Save Changes**.

Try it out

Follow the steps below to configure and run one of our sample applications to test profile self-management.

Step 1. Download the sample apps

To start this tutorial, you need to download the Ping SDK sample apps repo, which contains the projects you will use.

1. In a web browser, navigate to the Ping SDK sample apps repository.
2. Download the source code using one of the following methods:

Download a ZIP file

1. Click **Code**, and then click **Download ZIP**.
2. Extract the contents of the downloaded ZIP file to a suitable location.

Use a Git-compatible tool to clone the repo locally

1. Click **Code**, and then copy the HTTPS URL.
2. Use the URL to clone the repository to a suitable location.

   For example, from the command-line you could run:

```
git clone https://github.com/ForgeRock/sdk-sample-apps.git
```

The result of these steps is a local folder named `sdk-sample-apps`.

Step 2. Configure sample apps

Depending on the platform you are using, follow the steps below to configure a sample application to connect to your server.

Android

In this step, you configure the "kotlin-ui-prototype" sample to connect to your server.

1. In Android Studio, open the `sdk-sample-apps/android/kotlin-ui-prototype` folder you cloned in the previous step.
2. In the **Project** pane, switch to the **Android** view.
3. In the **Android** view, navigate to **app > kotlin+java > com.example.app > env**, and open `EnvViewModel.kt`.

   This file has the server environments the sample app uses. Each specifies the properties using the `FROptionsBuilder.build` method.
4. Update the `PingAM` or `PingAdvancedIdentityCloud` example configuration values to match your server environment:

url

The URL of the server to connect to.

Identity Cloud example:

`https://openam-forgerock-sdks.forgeblocks.com/am`

Self-hosted example:

`https://openam.example.com:8443/openam`

cookieName

The name of the cookie that contains the session token.

For example, with a self-hosted PingAM server this value might be `iPlanetDirectoryPro`.

> **Tip**
>
> PingOne Advanced Identity Cloud tenants use a random alpha-numeric string.
> To locate the cookie name in an PingOne Advanced Identity Cloud tenant, navigate to **Tenant settings** > **Global Settings**, and copy the value of the **Cookie** property.

realm

The realm in which the OAuth 2.0 client profile and authentication journeys are configured.

Usually, `root` for AM and `alpha` or `beta` for Advanced Identity Cloud.

oauthClientId

The client ID of your OAuth 2.0 application in PingOne Advanced Identity Cloud or PingAM.

For example, `sdkPublicClient`

oauthRedirectUri

The `redirect_uri` as configured in the OAuth 2.0 client profile.

> **Note**
>
> This value must exactly match a value configured in your OAuth 2.0 client.

For example, `org.forgerock.demo://oauth2redirect`.

5. Update the `USER_PROFILE_JOURNEY` variable with the name of the profile management journey you created earlier.

   For example, `sdkProfileManagement`

6. Save your changes.

iOS

In this step, you configure the "FRExample" sample app to connect to your server.

1. In Xcode, on the **File** menu, click **Open**.
2. Navigate to the `sdk-sample-apps` folder you cloned in the previous step, navigate to `iOS` > `uikit-frexamples` > `FRExample` > `FRExample.xcodeproj`, and then click **Open**.
3. In the navigator pane in Xcode, right-click `FRExample/Configs/FRAuthConfig` and select **Open As** > **Source Code**.
4. Update the following key values to match your server environment:

forgerock_url

The URL of the server to connect to.

Identity Cloud example:

```
https://openam-forgerock-sdks.forgeblocks.com/am
```

Self-hosted example:

```
https://openam.example.com:8443/openam
```

forgerock_cookie_name

The name of the cookie that contains the session token.

For example, with a self-hosted PingAM server this value might be `iPlanetDirectoryPro`.

> **Tip**
>
> PingOne Advanced Identity Cloud tenants use a random alpha-numeric string.
> To locate the cookie name in an PingOne Advanced Identity Cloud tenant, navigate to **Tenant settings** > **Global Settings**, and copy the value of the **Cookie** property.

forgerock_realm

The realm in which the OAuth 2.0 client profile and authentication journeys are configured.

Usually, `root` for AM and `alpha` or `beta` for Advanced Identity Cloud.

forgerock_oauth_client_id

The client ID of your OAuth 2.0 application in PingOne Advanced Identity Cloud or PingAM.

For example, `sdkPublicClient`

forgerock_oauth_redirect_uri

The `redirect_uri` as configured in the OAuth 2.0 client profile.

> **Note**
>
> This value must exactly match a value configured in your OAuth 2.0 client.

For example, `org.forgerock.demo://oauth2redirect`.

5. Save your changes.

Step 3. Run the sample app

Depending on the platform you are using, follow the steps below to run the sample application, obtain a session token, and use it to complete the self-registration journey you created earlier.

Android

1. In Android Studio, select **Run > Run 'app'**.
2. Tap the menu icon (☰), and then tap **Launch Journey**.
3. In **Journey Name** enter the name of a journey that will authenticate the user and issue a session, and then click **Submit**.

   For example, enter `sdkUsernamePasswordJourney` to use the authentication tree you created earlier.
4. Sign on as a demo user:
   - **Name**: `demo`
   - **Password**: `Ch4ng3it!`
5. After successful authentication, tap the menu icon (☰), and then tap **User Profile**.

   The app sends the session token to the journey which extracts the username and returns their profile information:

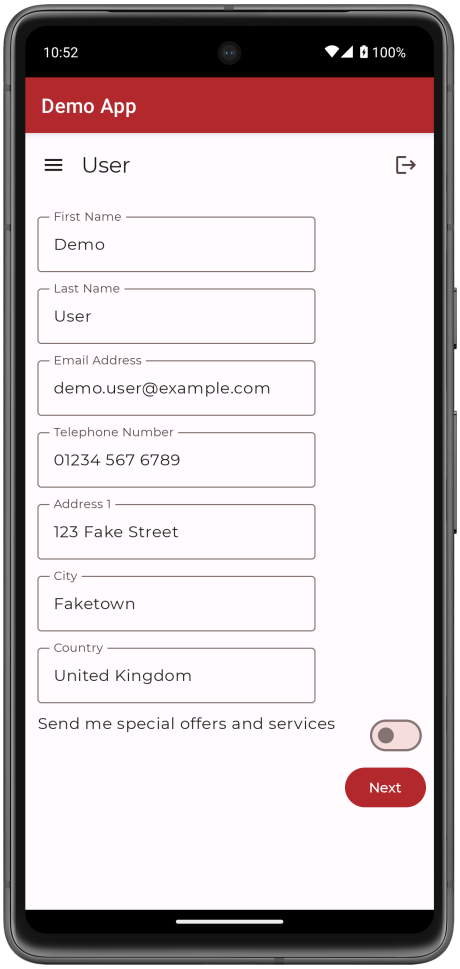


Figure 7. Viewing a user's profile information in an Android sample app.

6. Update any of the presented properties and then click **Next**.
7. To verify the profile was updated, tap the menu icon (☰), and then tap **User Profile**.

The app displays the updated profile values.

iOS

1. In Xcode, select **Product > Run**.

   Xcode launches the `FRExample` app in the iPhone simulator.

2. In the sample app on the iPhone simulator, in the **Select an action** menu, select **Login with UI (FRUser)**, and then click **Perform Action**.

3. Sign on as a demo user:

   - **Name**: `demo`
   - **Password**: `Ch4ng3it!`

4. After successful authentication, in the **Select an action** menu, select **FRSession.authenticate with UI (Token)**, and then click **Perform Action**.

5. In the popup window, enter the name of the profile management journey you created earlier, and then click **Continue**.

   For example, `sdkProfileManagement`

6. Update any of the presented properties and then click **Next**.

   The app sends the session token to the journey which extracts the username and returns their profile information:

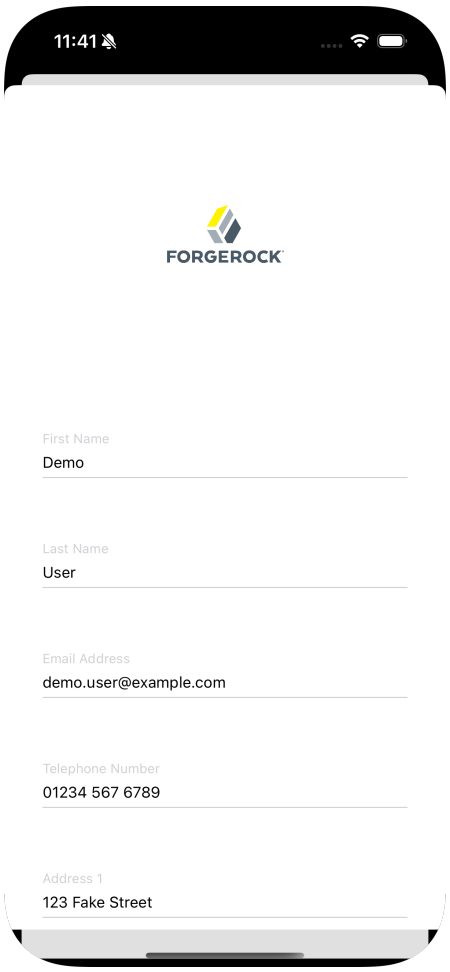


Figure 8. Viewing a user's profile information in an iOS sample app.

7. Update any of the presented properties and then click **Next**.
8. To verify the profile was updated, tap **Perform Action** again, enter the name of your profile management tree and then click **Continue**.

   The app displays the updated profile values.

Set up registered device self service

Applies to:

- ✓ Ping SDK for Android
- ✓ Ping SDK for iOS
- ✓ Ping SDK for JavaScript

The Ping SDKs can retrieve a list of the devices your users register to their accounts. Users can then manage their own devices, for example delete or rename them.

PingOne Advanced Identity Cloud and PingAM support registration of many different device types to support your multi-factor authentication journeys:

OATH devices

The registered device generates a one-time passcode that your users enter into the authentication journey.

Register OATH devices using the OATH Registration node.

To learn more about implementation, refer to Implement MFA using OATH one-time passwords.

PUSH devices

The registered device receives a PUSH notification from the server that the user must approve to continue their authentication journey.

Register PUSH devices using the Push Registration node.

To learn more about implementation, refer to Implement MFA using push notifications.

WebAuthn devices

The registered device acts as an *authenticator* and uses public-key cryptography to securely sign an assertion from the server.

Register WebAuthn devices using the WebAuthn Registration node.

To learn more about implementation, refer to Implement mobile biometrics and Implement web biometrics.

Device binding

The registered device generates a key pair and a key ID. The Ping SDKs send the *public* key and key ID to PingOne Advanced Identity Cloud or PingAM for storage in the user's profile.

Bind devices using the Device Binding node.

To learn more about implementation, refer to Bind and verify user devices.

Device profiling

The Ping SDKs collect specific data about the registered device to create a profile that helps to identify it during authentication journeys.

Profile devices using the Device Profile Collector node.

To learn more about implementation, refer to Device profile client configuration.

The Ping SDKs provide utility methods for managing each type of registered device:

Methods for managing devices

Type	Get	Update	Delete
OATH	`oath.get()`	Not supported	`oath.delete(device)`
PUSH	`push.get()`	Not supported	`push.delete(device)`
WebAuthn	`webAuthn.get()`	`webAuthn.update(device)`	`webAuthn.delete(device)`
Device binding	`bound.get()`	`bound.update(device)`	`bound.delete(device)`
Device profiles	`profile.get()`	`profile.update(device)`	`profile.delete(device)`

Getting lists of devices

To get a list of devices you must have an active session for the user. The Ping SDKs include the session token when making calls to the device management endpoints.

Session tokens often have a short duration and expire after 5 minutes. If the client does not have an active session token you should trigger an authentication journey to obtain a new session token before attempting to manage registered devices.

> **Note**
>
> Device management is not available when using OIDC login, as the Ping SDKs do not have direct access to the session token, which remains in the embedded browser.

Examples

Android

1. Import and initialize `deviceClient`:

```
import org.forgerock.android.auth.selfservice.Device

private val deviceClient = DeviceClient()
```

2. Call the `get()` method to retrieve lists of devices:

```
"Oath" -> deviceClient.oath.get()
"Push" -> deviceClient.push.get()
"WebAuthn" -> deviceClient.webAuthn.get()
"Binding" -> deviceClient.bound.get()
"Profile" -> deviceClient.profile.get()
```

iOS

1. Import `FRAuth` and initialize `deviceClient`:

```
import FRAuth

let deviceClient = DeviceClient()
```

2. Call the `get()` method to retrieve lists of devices:

```
let oathDevices = try await deviceClient.oath.get()
let pushDevices = try await deviceClient.push.get()
let webAuthnDevices = try await deviceClient.webAuthn.get()
let bindingDevices = try await deviceClient.bound.get()
let profileDevices = try await deviceClient.profile.get()
```

JavaScript

1. Import the `deviceClient` module from the Ping SDK for JavaScript, and provide a `ConfigOptions` object to initialize device self-service functionality:

```
import { deviceClient } from '@forgerock/javascript-sdk/device-client';
import { type ConfigOptions } from '@forgerock/javascript-sdk';

const config: ConfigOptions = {
  serverConfig: {
    baseUrl: 'https://openam-forgerock-sdks.forgeblocks.com/am',
  },
  realmPath: 'alpha',
};

const deviceClient = deviceClient(config);
```

2. Call the `get()` method, with optional query, to retrieve a list of devices.

   For example, the following code gets a list of all the user's OATH devices:

```
const oathQuery: RetrieveOathQuery = {
  /* your query parameters */
};

deviceClient.oath
  .get(oathQuery)
  .then((response) => {
    console.log('OATH Devices:', response);
  })
  .catch((error) => {
    console.error('Error fetching OATH devices:', error);
  });
```

Renaming a device

You can rename some types of registered device, with the following caveats:

1. You can only rename these device types:
   - Bound devices
   - Device profiles
   - WebAuthn devices
2. The authentication journey that provided the users' session must fulfil one or more of the following criteria:
   - Used same multi-factor authentication method as the device you want to rename.

For example, to rename a WebAuthn device the authentication journey that created the session must also authenticate using a WebAuthn device.

Or:

- Used the Enable Device Management node that alters the **Device Check Enforcement Strategy**.

Examples

Android

```
fun update(device: Device) {
    viewModelScope.launch {
        try {
            when (device) {
                is WebAuthnDevice -> deviceClient.webAuthn.update(device)
                is BoundDevice -> deviceClient.bound.update(device)
                is ProfileDevice -> deviceClient.profile.update(device)
                else -> throw IllegalArgumentException("Unsupported Device Type")
            }
            fetch(selectedType)
        } catch (e: Exception) {
            yield()
            state.update { it.copy(devices = emptyList(), throwable = e) }
        }
    }
}
```

iOS

```
let alert = UIAlertController(title: "Edit Device Name",
                              message: device.id,
                              preferredStyle: .alert)

alert.addTextField { textField in
    textField.text = device.deviceName
}

let okAction = UIAlertAction(title: "Submit", style: .default) {  [unowned alert] _ in
    let updateDeviceName = alert.textFields![0].text!
    Task {
        do {
            if var device = device as? BoundDevice {
                device.deviceName = updateDeviceName
                try await self.deviceClient.bound.update(device)
            } else if var device = device as? ProfileDevice {
                device.deviceName = updateDeviceName
                try await self.deviceClient.profile.update(device)
            } else if var device = device as? WebAuthnDevice {
                device.deviceName = updateDeviceName
                try await self.deviceClient.webAuthn.update(device)
            }
        } catch AuthApiError.apiFailureWithMessage(let reason, let message, let code, _) {
            self.showAlert(title: reason, message: message + " -  \(String(describing: code ?? 0))")
        }
        self.reloadAllDevices()
    }
}
```

JavaScript

```
const updateWebAuthnQuery: WebAuthnQueryWithUUID & WebAuthnBody = {
  /* your update query */
};

deviceClient.webAuthn
  .update(updateWebAuthnQuery)
  .then((response) => {
    console.log('Updated WebAuthn Device:', response);
  })
  .catch((error) => {
    console.error('Error updating WebAuthn device:', error);
  });
```

Deleting a device

You can delete or deregister a device, with the following caveats:

1. The authentication journey that provided the users' session must fulfil one or more of the following criteria:
    - Used same multi-factor authentication method as the device you want to delete.

      For example, to delete a WebAuthn device the authentication journey that created the session must also authenticate using a WebAuthn device.

   Or:

    - Used the Enable Device Management node that alters the **Device Check Enforcement Strategy**.

Examples

Android

```
fun delete(device: Device) {
    viewModelScope.launch {
        try {
            when (device) {
                is OathDevice -> deviceClient.oath.delete(device)
                is PushDevice -> deviceClient.push.delete(device)
                is WebAuthnDevice -> deviceClient.webAuthn.delete(device)
                is BoundDevice -> deviceClient.bound.delete(device)
                is ProfileDevice -> deviceClient.profile.delete(device)
                else -> throw IllegalArgumentException("Unsupported Device Type")
            }
            fetch(selectedType)
        } catch (e: Exception) {
            yield()
            state.update { it.copy(devices = emptyList(), throwable = e) }

        }
    }
}
```

iOS

```
let delete = UIContextualAction(style: .destructive, title: "Delete") { (action, view, completion) in
    let alert = UIAlertController(title: "Delete Device", message: "Are you sure you want to delete device
\"\(device.deviceName)\"?", preferredStyle: .alert)

    alert.addAction(UIAlertAction(title: "Yes", style: .destructive, handler: { (alert: UIAlertAction!) in
        Task {
            do {
              if let device = device as? BoundDevice {
                  try await self.deviceClient.bound.delete(device)
              } else if let device = device as? ProfileDevice {
                  try await self.deviceClient.profile.delete(device)
              } else if let device = device as? WebAuthnDevice {
                  try await self.deviceClient.webAuthn.delete(device)
              } else if let device = device as? OathDevice {
                  try await self.deviceClient.oath.delete(device)
              } else if let device = device as? PushDevice {
                  try await self.deviceClient.push.delete(device)
              }
            } catch AuthApiError.apiFailureWithMessage(let reason, let message, let code, _) {
                self.showAlert(title: reason, message: message + " -  \(String(describing: code ?? 0))")
            }
            self.reloadAllDevices()
        }
        completion(true)
    }))
```

JavaScript

```
const deleteWebAuthnQuery: WebAuthnQueryWithUUID & WebAuthnBody = {
  /* your delete query */
};

deviceClient.webauthn
  .delete(deleteWebAuthnQuery)
  .then((response) => {
    console.log('Deleted WebAuthn Device:', response);
  })
  .catch((error) => {
    console.error('Error deleting WebAuthn device:', error);
  });
```

Limitations

- The SDK for JavaScript does not apply the following customizations when using the device management endpoints:
  - Customized REST calls using interceptors
  - Customized logging behaviors

What are mobile biometrics?

Applies to:

- ✓ Ping SDK for Android
- ✓ Ping SDK for iOS
- ✕ Ping SDK for JavaScript

Mobile biometric authentication lets users authenticate by using a mobile device's biometric authenticator. Communication with the platform authenticator, such as fingerprint reader or facial recognition system, is handled by the SDK. The SDK communicates with PingAM to perform biometric registration and authentication using WebAuthn nodes. You can configure the nodes in PingAM to request that the SDK activates authenticators with certain criteria.

To enable mobile biometrics, the user's authenticator must first be registered through an authentication journey with the WebAuthn Registration node. Registration involves the selected authenticator creating a key pair. This key pair is specific to the origin of the application performing the authentication. The private key is used to sign the challenge from PingAM and create attestation for the authenticator.

The public key of the pair is sent to PingAM and stored in the user's profile. The private key is securely stored within the mobile device's and never leaves the device at any time.

When authenticating using mobile biometrics, the registered user encounters the WebAuthn Authentication node via an authentication journey. A challenge from PingAM is created and sent to the user's device. The device then signs an assertion from that challenge with its stored, private key. This assertion is then sent to PingAM for verification using the public key stored in the user's profile. If the data is verified as being from the registered authenticator and passes attestation checks, the authentication is considered successful.

Differences between device binding and WebAuthn

There are many similarities between WebAuthn and Device Binding and JWS verification. We provide authentication nodes to implement both technologies in your journeys.

Both can be used for usernameless and passwordless authentication, they both use public key cryptography, and both can be used as part of a multi-factor authentication journey.

One major difference is that with device binding, the private key never leaves the device.

With WebAuthn, there is a possibility that the private key is synchronized across client devices because of Passkey support, which may be undesirable for your organization.

For more details of the differences, refer to the following table:

Comparison of WebAuthn and Device Binding/JWS Verification

Feature	WebAuthn / FIDO	Device Binding / JWS Verifier	Details
Industry-standards based	☑	×	You can refer to the WebAuthn W3C specification. Device binding and JWS verification are proprietary implementations.
Public key cryptography	☑	☑	Both methods use Public key cryptography.
Usernameless support	☑	☑	After registration, the username can be stored in the device and obtained during authentication without the user having to enter their credentials.
Keys are bound to the device	×	☑	With WebAuthn, if Passkeys are used, they can be shared across devices. With device binding, the private keys do not leave the device.
Sign custom data	×	☑	With device binding, you can: • Customize the challenge that the device must sign. For example, you could include details of a transaction, such as the amount in dollars. • Add custom claims to the payload when signing a challenge. This gives additional context that the server can make use of by using a scripted node. Refer to Add custom claims when signing.
Format of signed data	WebAuthn authenticator data	JSON Web Signature (JWS)	
Integration	×	☑	With device binding, after verification, the signed JWT is available in: • Audit Logs • Transient node state This enables the data within to be used for integration into your processes and business logic.

Feature	WebAuthn / FIDO	Device Binding / JWS Verifier	Details
Platform support	☑ Android ☑ iOS ☑ Web browsers	☑ Android ☑ iOS ✕ Web browsers	As it is challenging to store secure data in a browser as a client app, device binding is not supported in web browsers.
Authenticator support	Determined by the platform. Configuration limited to: • Biometric with Fallback to Device Pin	Determined by the authentication node. Full configuration options: • Biometric Authentication • Biometric with Fallback to Device Pin • Application Pin • Silent	With device binding, you can specify what authentication action the user must perform to get access to the private keys. This provides greater flexibility in your security implementation and can reduce authentication friction for your users.
Key storage	Web browsers and iOS synchronize to the cloud. Android has the option to synchronize to the cloud.	***Android*** KeyStore ***iOS*** Secure enclave: hardware-backed and not synchronized to the cloud.	Both technologies store the private keys securely on the client. WebAuthn supports synchronizing the private keys to the cloud for use on other devices. This can reduce authentication friction for your users but may also increase the risk of a breach.
Managing device keys	Managed by the device OS. Apps cannot delete *local* client keys programmatically and do not have a reference to the *remote* server key for deletion.	Managed by the Ping SDKs. Provides an interface to delete local client and remote server keys.	The ability to programmatically delete both client and server keys can greatly simplify the process of registering a new device if an old device is lost or stolen.
Passkey Support	☑	✕	WebAuthn supports synchronizing the private keys to the cloud for use on other devices. Device binding keeps the private key locked in the device.

Feature	WebAuthn / FIDO	Device Binding / JWS Verifier	Details
App integrity verification	***Android*** Requires an `assetlinks.json` file. ***iOS*** Requires `apple-app-site-association` file.	Not provided by the device binding or verification nodes. It can be added as part of the journey by using app integrity nodes.	App integrity verification helps ensure your users are only using a supported app rather than a third-party or potentially malicious version.
Key attestation	***Android*** SafteyNet ***iOS*** None	***Android*** Uses hardware-backed key pairs with Key Attestation. ***iOS*** It can be added as part of the journey by using app integrity nodes to support key attestation.	Key attestation verifies that the private key is valid and correct, is not forged, and was not created in an insecure manner.
Complexity	Medium	Low	WebAuthn requires a bit more configuration, for example, creating and uploading the `assetlinks.json` and `apple-app-site-association` files. Device binding only requires the journey and the SDK built into your app.

Prerequisites

To create a journey with mobile biometric authentication, you need the following:

1. A server that supports the WebAuthn nodes (PingOne Advanced Identity Cloud or PingAM 7.1 or later).
2. A mobile device that has biometric authentication, such as Face ID or a fingerprint reader, and the user has registered their biometrics on the device.
3. An application with the latest native mobile SDK that includes the Biometric Authentication API (v3 or later).

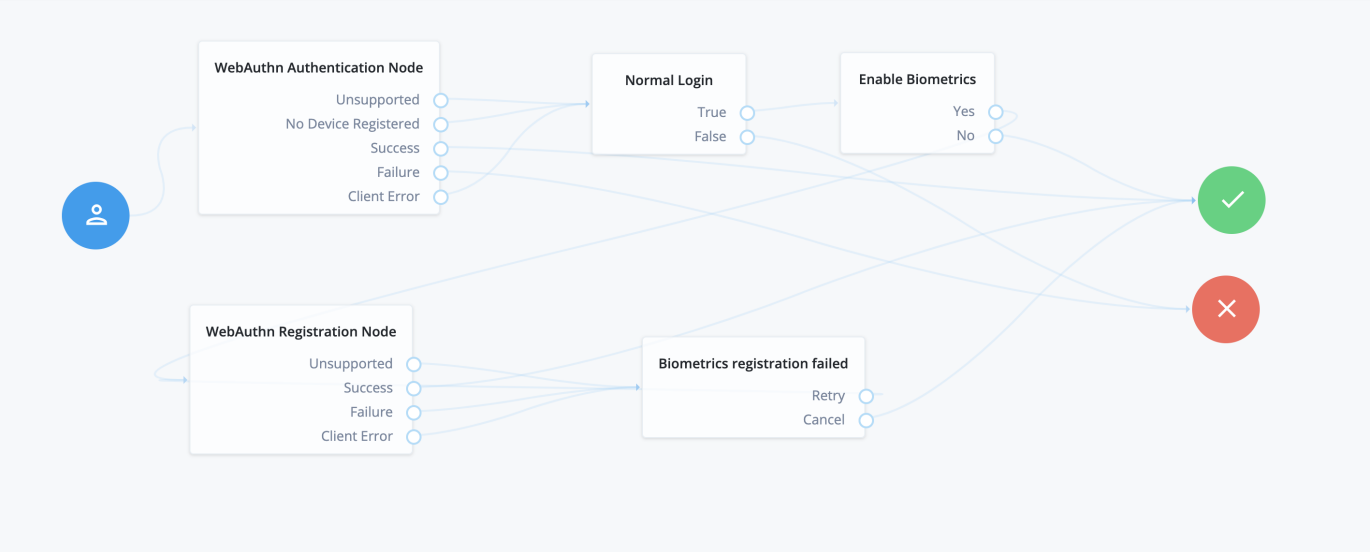
> **Note**
>
> The SDKs can only support the WebAuthn Registration and WebAuthn Authentication nodes when the `Return challenge as JavaScript` option is disabled.
> When this option is disabled, the WebAuthn nodes return a `MetadataCallback`, which the SDK converts to a `WebAuthnRegistrationCallback` or a `WebAuthnAuthenticationCallback`.

Prepare the server

The following example journey covers the "usernameless" authentication case. This is a simple prototype flow that does not cover all edge cases that might be present in a production environment. If WebAuthn authentication is not possible for any reason, the flow falls back to a normal login journey.

To access this configuration, you need to log in to PingOne Advanced Identity Cloud or PingAM as an administrator, and create a new journey.

1. In the editor drag the following nodes into the journey:
   - WebAuthn Registration node
   - WebAuthn Authentication node
   - Inner Tree Evaluator node
   - Two Choice Collector nodes
2. Connect the nodes similar to the following example:



3. Configure the nodes:
   - In both the WebAuthn Registration and WebAuthn Authentication nodes, the `Return challenge as JavaScript` option must be **disabled**.
   - In the WebAuthn Registration node, `Authentication attachment` **must be** either `UNSPECIFIED` or `PLATFORM`.
   - In the WebAuthn Registration node, enable the `Username to device` option.

- In the WebAuthn Authentication node, enable the `Username from device` option.
- Use Choice Collector nodes to handle the user input in case of registration failure, and to give users the option to enable biometrics for this journey.
- Set the `Relying party identifier` to the domain of your PingAM instance.

  For example: `openam.example.com`.

If you want your users to provide a username, deactivate the `Username from/to device` options, and add a Username Collector node before the WebAuthn Registration node and WebAuthn Authentication node.

Accessing WebAuthn authenticator information

The Ping SDKs send WebAuthn assertion or attestation information back to the server when they encounter a **WebAuthn Authentication node** or a **WebAuthn Registration Node**.

For example, whether the authenticator used is `platform` based, such as a built-in fingerprint reader, or is `cross-platform`, such as a USB security key that could be used on multiple clients.

The Ping SDKs also include a number of flags about the authenticator used, as defined in Web Authentication: An API for accessing Public Key Credentials Level 2.

The authentication nodes store this data in transient state, so that you can use the information to alter the course of the authentication journey, if required.

WebAuthnRegistration node

Stores the attestation information in a `webauthnAttestationInfo` object in transient state:

Example `webauthnAttestationInfo` object

```
{
  "authenticatorAttachment": "platform",
  "flags": {
    "UP": true,
    "UV": true,
    "ED": false,
    "AT": false,
    "BE": true,
    "BS": true
  }
}
```

WebAuthnAuthentication node

Stores the assertion information in a `webauthnAttestationInfo` object in transient state:

Example `webauthnAttestationInfo` object

```
{
  "authenticatorAttachment": "cross-platform",
  "flags": {
    "UP": true,
    "UV": true,
    "ED": false,
    "AT": false,
    "BE": false,
    "BS": false
  }
}
```

Catching client errors

The WebAuthn Registration and WebAuthn Authentication nodes might result in a `Client Error`. Client errors can happen for a number of reasons.

In order to parse the error and act upon it, make use of a Scripted Decision node to access the shared state within the journey, and read the `WebAuthenticationDOMException` thrown.

For more information regarding the use of the Scripted Decision node, see Scripted Decision Node API Functionality in the PingAM documentation.

Biometrics using the Ping SDK for Android

This section covers how to implement mobile biometric authentication using the Ping SDK for Android.

Support for mobile biometrics lets users authenticate through the WebAuthn Registration and WebAuthn Authentication nodes to register the user's device, and use the device as an authenticator.

Associate your app with your server

To associate your server with your Android app you need to make public, verifiable statements by using a Digital Asset Links JSON file (`assetlinks.json`).

Example `assetlinks.json` file

```
[
    {
        "relation": [
            "delegate_permission/common.handle_all_urls",
            "delegate_permission/common.get_login_creds"
        ],
        "target": {
            "namespace": "android_app",
            "package_name": "com.example.app",
            "sha256_cert_fingerprints": [
                "E6:5A:5D:37:22:FC...22:99:20:03:E6:47"
            ]
        }
    }
]
```

Get SHA-256 fingerprint of your signing certificates

The `assetlinks.json` file includes SHA-256 fingerprints of the certificates you use to sign your Android applications. The steps for obtaining the fingerprint depend on the method you use to distribute your application.

Android App Bundles

If you are using Android App Bundles to distribute your apps, then the hashes of the certificate used to sign your application are available in the Android Developer console.

Follow these steps to obtain the SHA-256 hash of your signing certificate:

1. Configure your Android App Bundle for signing. Google has a number of methods for managing the signing certificates, including uploading your own or having Google manage them for you.

   For information on how to set up signing, refer to Sign your app in the Google Developer Documentation.

2. In the Google Play Console:

   1. Select the app that will be supporting mobile biometrics.

   2. Navigate to **Setup** > **App integrity** > **App signing**.

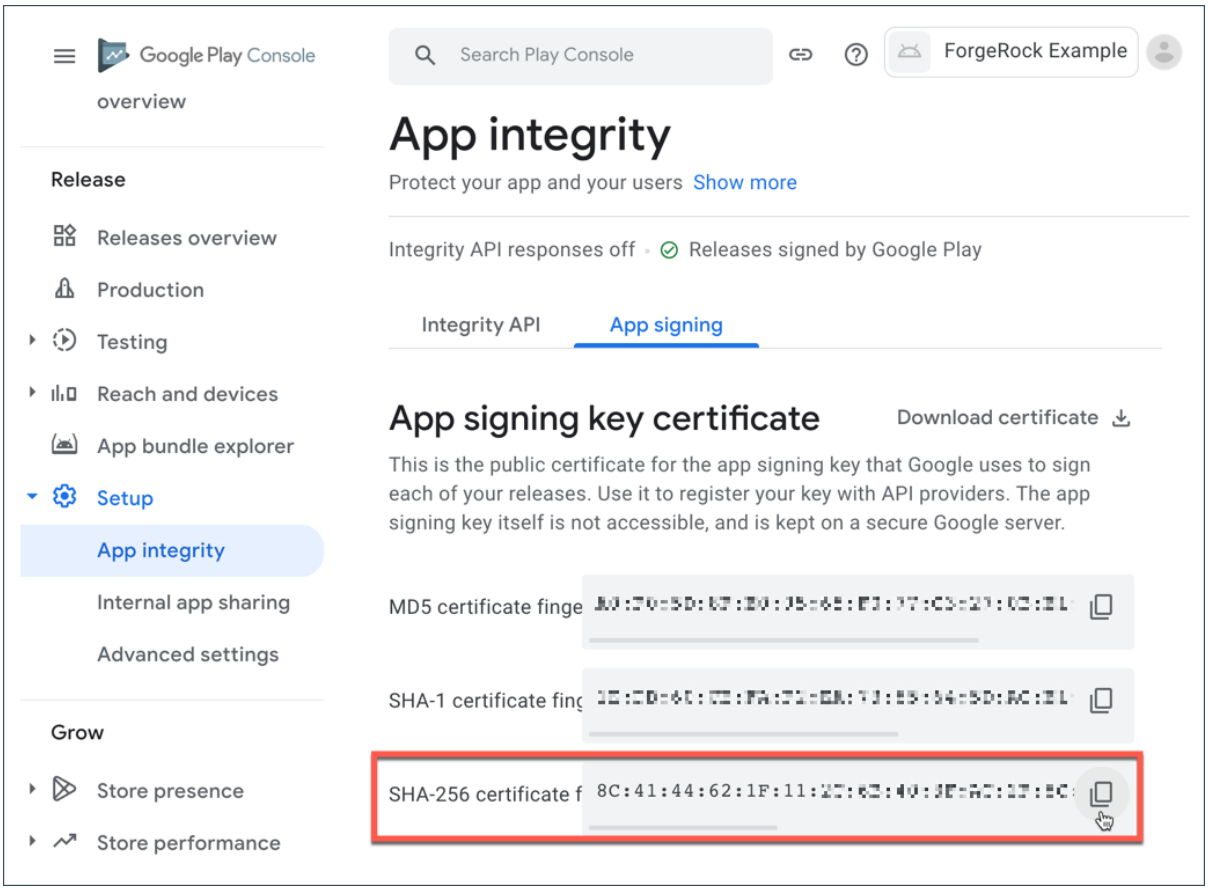


Figure 1. App signing keys in the Google Play Console

   3. In the **App signing key certificate** section, copy the **SHA-256 certificate fingerprint** value.

> **Tip**
>
> In the **Digital Asset Links JSON** section is a file that you can copy with the SHA-256 fingerprint already in place.

3. Create or update an `assetlinks.json` with the values copied from the Google Play Console for your app.

For more information on creating an `assetlinks.json` file, refer to Google Digital Asset Links.

Local debug keys

You must manually generate a SHA-256 fingerprint of your signing key in the following scenarios:

- You are signing your APK with the default debug.jks that Android Studio created for the project
- You are signing your APK with your own keys that you have generated that have not been uploaded to the Google Play Console

Follow these steps to obtain the SHA-256 hash of your signing certificate:

1. In the `build.gradle` file for your application, check the settings defined in the `signingConfigs` property:

   Example signingConfigs when using the default debug.jks

   ```
   signingConfigs {
       debug {
           storeFile file('../debug.jks')
           storePassword 'android'
           keyAlias 'androiddebugkey'
           keyPassword 'android'
       }
   }
   ```

2. In a terminal window, navigate to the location of the JKS file, and then run the following command:

   ```
   keytool -list -v -alias <keyAlias> -keystore <storeFile> | grep SHA256
   ```

   Important

   Swap the *<keyAlias>* and *<storeFile>* placeholders with the values you obtained from your project. For example:
   `keytool -list -v -alias "androiddebugkey" -keystore "./debug.jks" | grep SHA256`

3. When requested, enter the keystore password, as specified in the `keyPassword` property in the `build.gradle` file.

   The command prints the SHA-256 fingerprint of the signing key:

   ```
   Enter keystore password:  android
   SHA256: E6:5A:5D:37:22:FC...22:99:20:03:E6:47
   Signature algorithm name: SHA256withRSA
   ```

4. Create or update an `assetlinks.json` with the SHA-256 fingerprint, and the details of your app.

For more information on creating an `assetlinks.json` file, refer to Google Digital Asset Links.

Host the digital asset links JSON file

- For PingOne Advanced Identity Cloud deployments, refer to Upload an Android assetlinks.json file.
- For self-managed deployments, host the file at `https://<your domain>/.well-known/assetlinks.json`.

Summary

You have now created and uploaded a digital asset links JSON file.

You can now proceed to Configure biometric authentication journeys.

Configure biometric authentication journeys

To use mobile biometrics with the Ping SDK for Android configure the authentication nodes in your journeys as follows:

1. In each **WebAuthn Registration node** and **WebAuthn Authentication node**:
    - Ensure the **Return challenge as JavaScript** option is **not enabled**

      The SDK expects a JSON response from these nodes, enabling this option would cause the journey to fail
    - Set the **Relying party identifier** option to be the domain hosting the `assetlinks.json` file

      For example, `openam-docs.forgeblocks.com`

      You do not need the protocol or the path.
2. In each **WebAuthn Registration node**
    - Set the **Authentication attachment** option to either `UNSPECIFIED` or `PLATFORM`
    - Ensure the **Accepted signing algorithms** option includes either `ES256` or `RS256`
    - Ensure the **Limit registrations** option is **not enabled**

Configure origin domains

To enable WebAuthn on Android devices, you must configure the nodes with the base64-encoded SHA-256 hash of the signing certificate as the origin domain.

The steps for obtaining the base64-encoded SHA-256 hash depend on the method you use to distribute your application.

Android App Bundles

Follow these steps to download the app signing certificate, and then generate a base64-encoded SHA-256 hash:

1. In the Google Play Console:
    1. Select the app that will be supporting mobile biometrics.
    2. Navigate to **Setup** > **App integrity** > **App signing**.
    3. In the **App signing key certificate** section, click **Download certificate**.

       This downloads a local copy of the signing certificate, named `deployment_cert.der`.
2. In a terminal window, navigate to the location of the `deployment_cert.der` file, and then run the following command:

    ```
    cat deployment_cert.der | openssl sha256 -binary | openssl base64 | tr '/+' '_-' | tr -d '='
    ```

    The command prints the base64-encode SHA-256 fingerprint of the signing key:

    ```
    jEFEYh80K55iHYkxsBRLGtAP6wvjOS5Pj-ZKHHjwi0k
    ```
3. Add a prefix of `android:apk-key-hash:` to the base64-encode SHA-256 fingerprint. For example:

    ```
    android:apk-key-hash:jEFEYh80K55iHYkxsBRLGtAP6wvjOS5Pj-ZKHHjwi0k
    ```
4. In each **WebAuthn Registration node** and **WebAuthn Authentication node**, set the **Origin domains** option to the value created in the previous step:

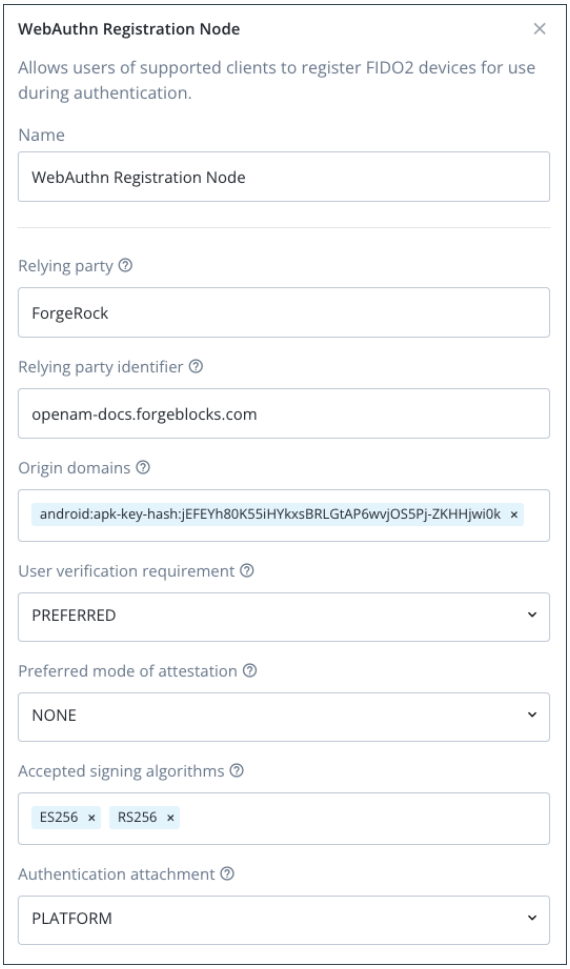


Figure 1. Example WebAuthn Registration node configuration

Local debug keys

Follow these steps to extract the app signing certificate from the JKS and generate a base64-encoded SHA-256 hash:

1. In the `build.gradle` file for your application, check the settings defined in the `signingConfigs` property:

   Example signingConfigs when using the default debug.jks

   ```
   signingConfigs {
       debug {
           storeFile file('../debug.jks')
           storePassword 'android'
           keyAlias 'androiddebugkey'
           keyPassword 'android'
       }
   }
   ```

2. In a terminal window, navigate to the location of the JKS file, and then run the following command:

   ```
   keytool -exportcert -alias <keyAlias> -keystore <storeFile> | openssl sha256 -binary | openssl base64
   | tr '/+' '_-' | tr -d '='
   ```

   Important

   Swap the *<keyAlias>* and *<storeFile>* placeholders with the values you obtained from your project. For example:
   `keytool -exportcert -alias "androiddebugkey" -keystore "./debug.jks" | openssl sha256 -binary | openssl base64 | tr '/+' '_-' | tr -d '='`

3. When requested, enter the keystore password, as specified in the `keyPassword` property in the `build.gradle` file.

   The command prints the base64-encoded SHA-256 fingerprint of the signing key:

   ```
   Enter keystore password:  android
   jEFEYh80K55iHYkxsBRLGtAP6wvjOS5Pj-ZKHHjwi0k
   ```

4. Add a prefix of `android:apk-key-hash:` to the base64-encode SHA-256 fingerprint. For example:

   ```
   android:apk-key-hash:jEFEYh80K55iHYkxsBRLGtAP6wvjOS5Pj-ZKHHjwi0k
   ```

5. In each **WebAuthn Registration node** and **WebAuthn Authentication node**, set the **Origin domains** option to the value created in the previous step:

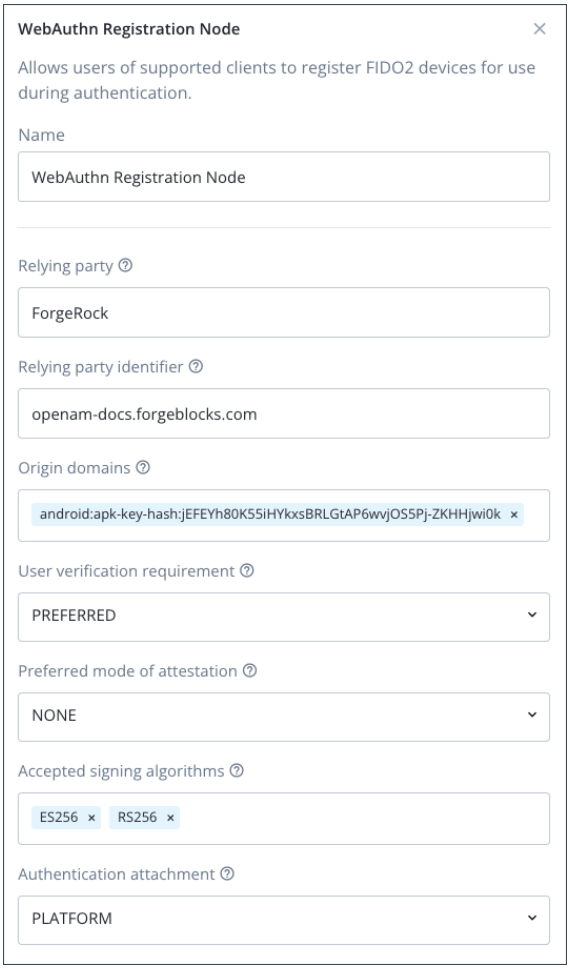


Figure 2. Example WebAuthn Registration node configuration

Summary

You have now configured your WebAuthn journey for use with the Ping SDK for Android.

You can now proceed to Configure the Ping SDK for Android for WebAuthn.

Configure the Ping SDK for Android for WebAuthn

1. Add the following dependency to the `build.gradle` file:

```
implementation 'com.google.android.gms:play-services-fido:20.0.1'
```

2. Link to `assetlinks.json` in the Android app, adding the following line to the manifest file under your application:

```
<meta-data android:name="asset_statements" android:resource="@string/asset_statements" />
```

3. Add an `asset_statements` string resource to the `string.xml` file:

```
<string name="asset_statements" translatable="false">
[{
  \"include\": \"https://<custom-domain-fqdn>/.well-known/assetlinks.json\"
}]
</string>
```

Register a WebAuthn device

To register a WebAuthn device on receipt of a `WebAuthnRegistrationCallback` from the server, use the `register()` method.

Optionally, use the `deviceName` parameter to assign a name to the device to help the user identify it.

Android - Java

```
WebAuthnRegistrationCallback callback =
    node.getCallback(WebAuthnRegistrationCallback.class);

callback.register(requireContext(), deviceName, node, new FRListener<Void>() {
    @Override
    public void onSuccess(Void result) {
        // Registration is successful
        // Continue the journey by calling next()
    }

    @Override
    public void onException(Exception e) {
        // An error occurred during the registration process
        // Continue the journey by calling next()
    }
});
```

Android - Kotlin

```
fun WebAuthnRegistrationCallback(
    callback: WebAuthnRegistrationCallback,
    node: Node,
    onCompleted: () -> Unit
) {

    val context = LocalContext.current
    var deviceName by remember { mutableStateOf(Build.MODEL) }

    try {
        callback.register(context, deviceName, node)
        // Registration is successful
        currentOnCompleted()
    } catch (e: CancellationException) {
        // User cancelled registration
    } catch (e: Exception) {
        // An error occurred during the registration process
        currentOnCompleted()
    }
}
```

Passkey support

The Ping SDK for Android supports passkeys when the app is running on **Android P** or later. For more information on passkeys, refer to Passkey support on Android and Chrome.

If the WebAuthn Registration node has the **Username to device** option enabled and the app is running on **Android P** or later, then the SDK sets the `RESIDENT_KEY_REQUIRED` flag and enables passkeys for WebAuthn.

In this case, the user is asked to create a new passkey on their device and is required to perform biometric authentication to confirm. The device syncs the generated passkey to the user's Google Account for use on their supported devices.

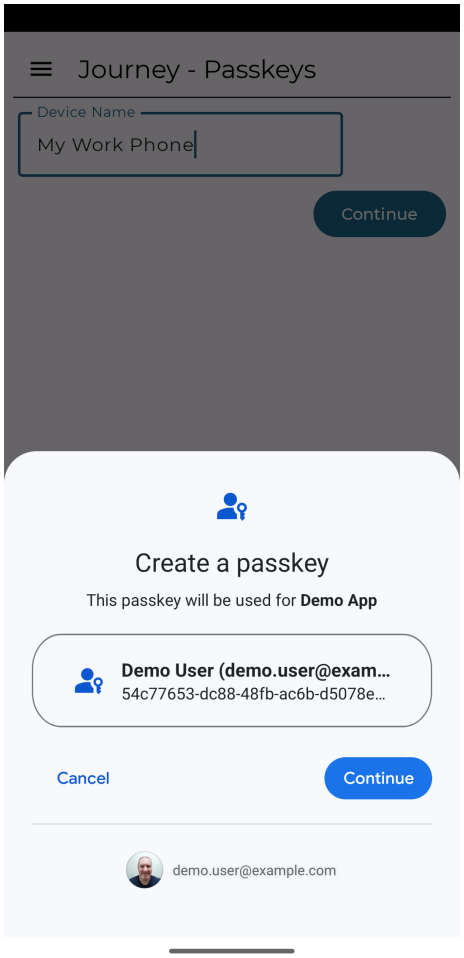


Figure 1. Creating a new passkey on Android

If the device is not running **Android P** or later, the SDK sets the `RESIDENT_KEY_DISCOURAGED` flag, meaning passkeys are not used nor synchronized to the Google Account.

For more information about resident keys and client-side discoverable credentials, refer to ResidentKeyRequirement in the Google developer documentation.

Override passkey support

You can use the `setResidentKeyRequirement()` method to override the automatic behavior. For example, if you do not want to use passkeys on **Android P** devices, you might use the following code:

```
callback.setResidentKeyRequirement(ResidentKeyRequirement.RESIDENT_KEY_DISCOURAGED)
```

Authenticate by using a WebAuthn device

After the user registers their mobile device they can use it as an authenticator, with its registered key pair, through the WebAuthn Authentication node, which the Ping SDK for Android returns as a `WebAuthnAuthenticationCallback`.

If the device supports passkeys, the operating system displays a list of available passkeys:

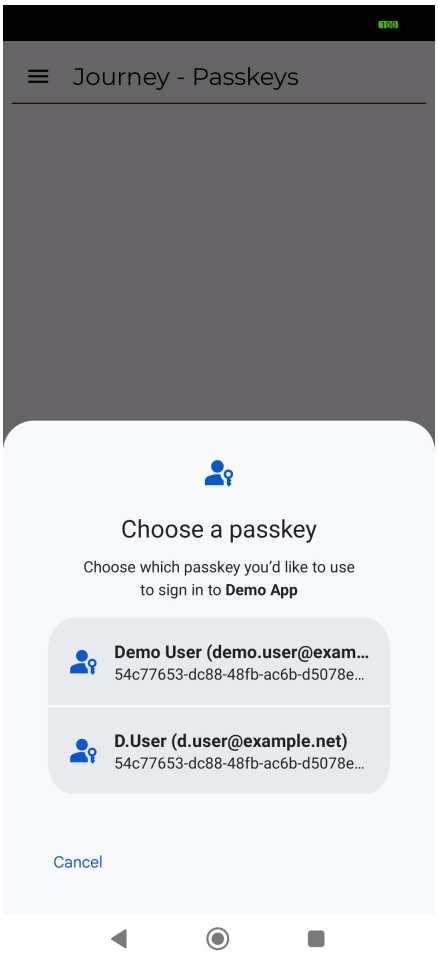


Figure 1. Select the passkey to use for WebAuthn

Note that removing credentials stored on the client device does not remove the associated data from the server. You will need to register the device again after removing credentials from the client.

As part of authentication process, the SDK provides the `WebAuthnAuthenticationCallback` for authenticating the device as a credential.

Android - Java

```
WebAuthnAuthenticationCallback callback = node.getCallback(WebAuthnAuthenticationCallback.class);
callback.authenticate(requireContext(), node, webAuthnKeySelector.DEFAULT, new FRListener<Void>() {
    @Override
    public void onSuccess(Void result) {
        // Authentication is successful
        // Continue the journey by calling next()
    }

    @Override
    public void onException(Exception e) {
        // An error occurred during the authentication process
        // Continue the journey by calling next()
    }
});
```

Android - Kotlin

```
fun WebAuthnAuthenticationCallback(
    callback: WebAuthnAuthenticationCallback,
    node: Node,
    onCompleted: () -> Unit
) {

    val context = LocalContext.current

    try {
        callback.authenticate(context, node)
        // Authentication successful
        currentOnCompleted()
    } catch (e: CancellationException) {
        // User cancelled authentication
    } catch (e: Exception) {
        // An error occurred during the authentication process
        currentOnCompleted()
    }
}
```

Note

The `WebAuthnAuthenticationCallback.authenticate()` method has a parameter, `Node`.
If the current node has both `WebAuthnAuthenticationCallback` and `HiddenValueCallback` callbacks then the SDK automatically sets the outcome of the authentication process for both success and failure to the designated `HiddenValueCallback`.

WebAuthnKeySelector

An optional `WebAuthnKeySelector` parameter can be provided for authentication.

The `WebAuthnKeySelector.select()` method is invoked when `Username from device` is enabled in the WebAuthn Authentication node. This feature requires that `Username to device` is enabled in the WebAuthn Registration node as well. With these options enabled, the registered key pair is associated with the username, and the SDK can present a list of registered keys to the user to continue the authentication process without collecting a username.

> **Note**
>
> The `sourceList` is a list of `PublicKeyCredentialSource` constructed during registration. You can alter the string value and present the altered value to the user; however, you **must** return the selected `PublicKeyCredentialSource` as it was provided in the original list to the provided `listener`.

```
callback.authenticate(this, node, new WebAuthnKeySelector() {
    @Override
    public void select(@NonNull FragmentManager fragmentManager,
                       @NonNull List<PublicKeyCredentialSource> sourceList,
                       @NonNull FRListener<PublicKeyCredentialSource> listener) {
        //Always pick the first one.
        listener.onSuccess(sourceList.get(0));
    }
}, new FRListener<Void>() {
    @Override
    public void onSuccess(Void result) {
        //...
    }

    @Override
    public void onException(Exception e) {
        //...
    }
});
```

Handle WebAuthn errors

When an error occurs during the registration or authentication process, the Ping SDK for Android returns the `WebAuthnResponseException` exception. In most cases, errors are returned as per the specification. The error code can be found from `WebAuthnResponseExcetpion.getErrorCode()`.

Convert exceptions for handling by the PingAM server

When you use `WebAuthnRegistrationCallback.register()` or `WebAuthnAuthenticationCallback.authenticate()`, the SDK automatically parses the error into the appropriate format for PingAM. When PingAM receives the completed callback from the SDK the authentication flow follows the WebAuthn registration process to reach the appropriate outcome.

However, if the error has to be handled manually, the `WebAuthnResponseException` class provides a convenience method called `toServerError()` to convert the error into the appropriate format.

```
callback.register(this, node, new FRListener<Void>() {
    @Override
    public void onSuccess(Void result) {
        next();
    }

    @Override
    public void onException(Exception e) {
        if (e instanceof WebAuthnResponseException) {
            WebAuthnResponseException exception = (WebAuthnResponseException) e;
            exception.getErrorCode(); // Do something with the error or proceed to the next node.
        }
    }
});
```

> **Note**
>
> `WebAuthnResponseExcetpion.getErrorCode() == com.google.android.gms.fido.fido2.api.common.ErrorCode#NOT_SUPPORTED_ERR` results in an `Unsupported` outcome in both WebAuthn Registration node and WebAuthn Authentication node.
> Any other `WebAuthnResponseExcetpion.getErrorCode()` results in a `Client Error` outcome in the nodes.

Unregister a WebAuthn device

To unregister a WebAuthn device from a user's profile, use the `deleteCredentials` function in your application. The function requires the `publicKeyCredentialSource` as a parameter.

Use the `loadAllCredentials` method and pass in the relying party identifier (rpId) string to return an array of `publicKeyCredentialSource` values. The rpId string must match the configuration you used when you configured the authentication journeys earlier.

> **Note**
>
> You can only remove a device if it has the username embedded in the profile.
> You must enable the **Username to Device** option in the WebAuthn Registration node to be able to remove the device from a user's profile on the server using the SDKs.

The SDK attempts to delete the record of the device from the server. If that succeeds, it will then remove the local keys held by the client device. If it fails to remove the records from the server, it will not remove the local keys by default.

However, you can pass the `forceDelete: true` boolean parameter to the function to delete the local keys even if the call to the server fails.

```
val rpId = "openam-docs.forgeblocks.com"

frWebAuthn.loadAllCredentials(rpId).let {
    frWebAuthn.deleteCredentials(it.first(), true)
}
```

Removing keys from either the server or the device means you will need to register it again for WebAuthn journeys. Refer to Register a WebAuthn device.

More information

- `deleteCredentials` API reference
- `loadAllCredentials` API reference

Biometrics using the Ping SDK for iOS

This section covers how to implement mobile biometric authentication using the Ping SDK for iOS.

Support for mobile biometrics lets users authenticate through the WebAuthn Registration and WebAuthn Authentication nodes to register the user's device, and use the device as an authenticator.

Prepare an apple-app-site-association file

You can create an `apple-app-site-association` file that creates a secure association between your domain and your app. This allows you to share credentials, and use universal links to open your app from your website.

To create the secure association, you upload the `apple-app-site-association` file to your domain, and add matching Associated Domains Entitlement keys to your app.

1. Prepare an `apple-app-site-association` file. For example:

```
{
  "applinks": {
    "details": [
      {
        "appIDs": [
          "XXXXXXXXXX.com.example.AppName"
        ],
        "components": [
          {
            "/": "/reset/*",
            "comment": "Success after reset password journey"
          }
        ]
      }
    ]
  },
  "webcredentials": {
    "apps": [
      "XXXXXXXXXX.com.example.AppName"
    ]
  }
}
```

For more information, refer to Supporting associated domains.

2. Host the file at your domain.
    - For PingOne Advanced Identity Cloud deployments, refer to Upload an apple-app-site-association file in the *Identity Cloud documentation*.
    - For self-managed deployments, host the file at `https://<your domain>/.well-known/apple-app-site-association`.
3. Configure the associated domains entitlement key in your app.

    For more information, refer to Associated Domains Entitlement.

Configure biometric authentication journeys

To use mobile biometrics with the Ping SDK for iOS configure the authentication nodes in your journeys as follows:

1. In each **WebAuthn Registration node** and **WebAuthn Authentication node**:
    - Ensure the **Return challenge as JavaScript** option is **not enabled**.

      The SDK expects a JSON response from these nodes; enabling the **Return challenge as JavaScript** option would cause the journey to fail.
    - Set the **Relying party identifier** option to be the domain hosting the `apple-app-site-association` file; for example, `openam-docs.forgeblocks.com`.

      You do not need the protocol or the path.
    - To enable passkey support, enable **Username to device** in the **WebAuthn Registration node**, and **Username from device** in the **WebAuthn Authentication node**.
2. In each **WebAuthn Registration node**:
    - Set the **Authentication attachment** option to either `UNSPECIFIED` or `PLATFORM`.
    - Ensure the **Accepted signing algorithms** option includes `ES256`.
    - Ensure the **Limit registrations** option is **not enabled**.

Configure origin domains

To enable WebAuthn on iOS devices, you must configure the nodes with a specially-formatted string containing the bundle identifier of your application, which you can find in XCode, on the **Signing & Capabilities** tab of your apps target page:

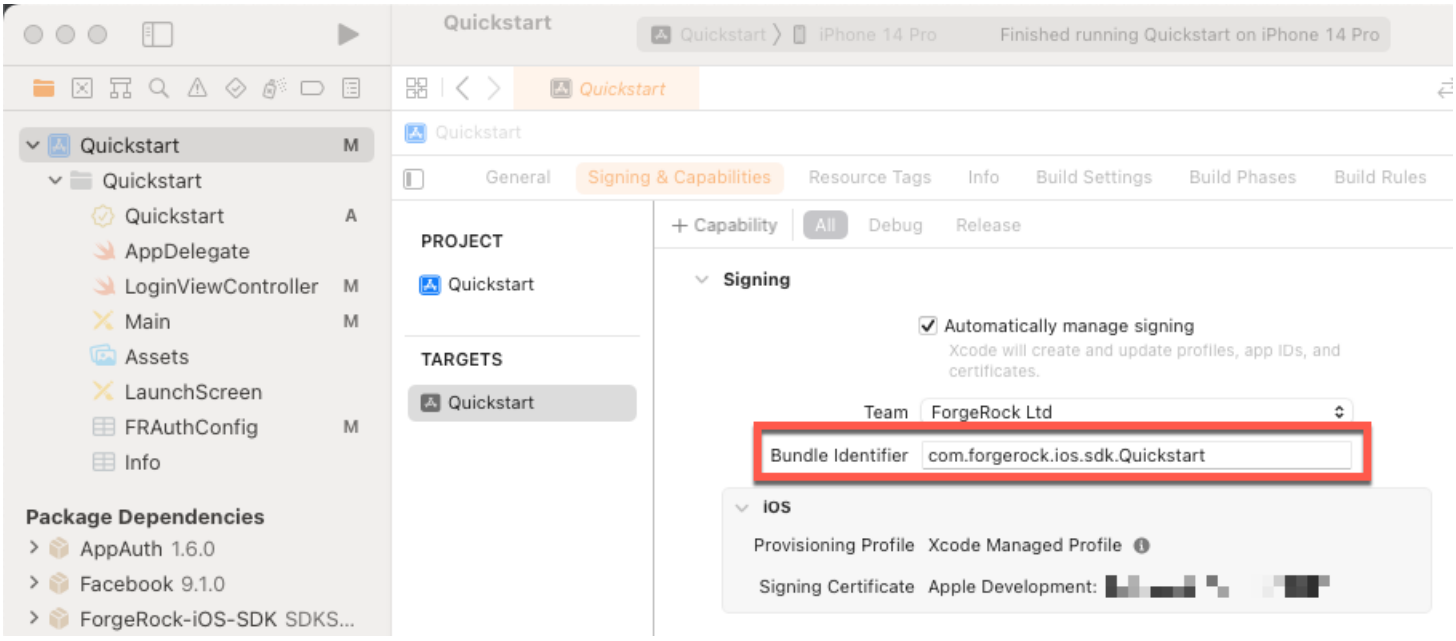


Figure 1. Bundle identifier field in XCode

Prefix this value with the string `ios:bundle-id:`. For example:

```
ios:bundle-id:com.forgerock.ios.sdk.Quickstart
```

To enable passkey support, add the fully-qualified domain name of the PingOne Advanced Identity Cloud or PingAM instance as an origin domain. For example, `https://openam-docs.forgeblocks.com`.

Add these values to the **Origin domains** property in each **WebAuthn Registration node** and **WebAuthn Authentication node** in the journey.

Register a WebAuthn device

To register a WebAuthn device on receipt of a `WebAuthnRegistrationCallback` from the server, use the `register()` method.

```
if let registrationCallback = callback as? WebAuthnRegistrationCallback {

    registrationCallback.delegate = self

    registrationCallback.register(
        node: node,
        window: UIApplication.shared.windows.first,
        deviceName: UIDevice.current.name,
        usePasskeysIfAvailable: false)
    { (attestation) in
        // Registration is successful
        // Submit the Node using Node.next()
    } onError: { (error) in
        // An error occurred during the registration process
        // Submit the Node using Node.next()
    }
}
```

Use the optional `deviceName` parameter to assign a name to the device to help the user identify it.

Set the `usePasskeysIfAvailable` parameter to `true` to enable passkeys on supported devices.

Enable Passkey support

The Ping SDK for iOS supports passkeys when the app is running on **iOS 16** or later, or recent versions of macOS. For more information, refer to Passkeys in the Apple developer documentation.

To enable the use of passkeys during registration, you should: - In PingAM, enable the **Username to device** option in the WebAuthn Registration node in your authentication journeys. - In the SDK, set the `usePasskeysIfAvailable` parameter to `true` in the `registrationCallback.register` function. - Run your app on a passkey-enabled version of iOS or macOS.

When passkeys are enabled the user is asked to create a new passkey on their device and is required to perform biometric authentication to confirm. The device syncs the generated passkey to the user's iCloud account for use on their supported devices.

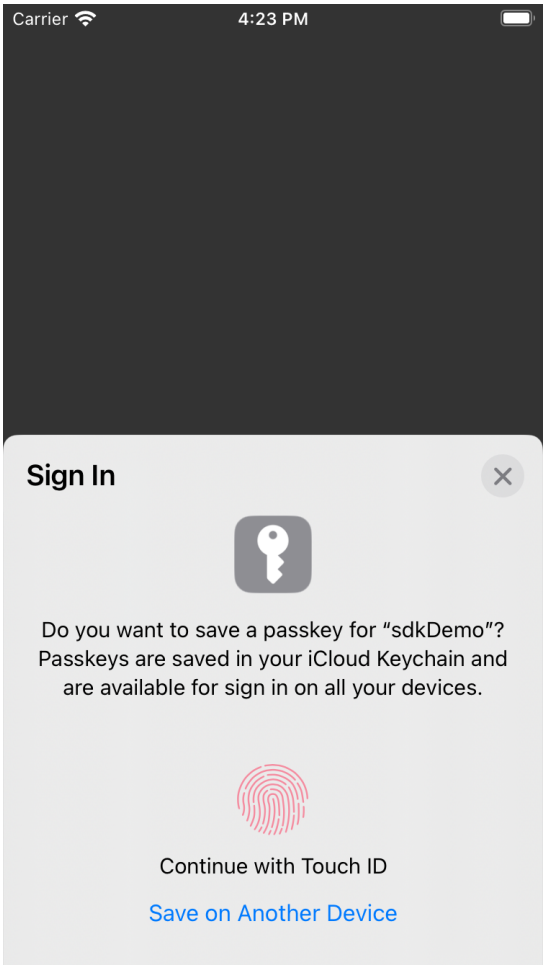


Figure 1. Creating a new passkey on iOS

Detect WebAuthn keys on passkey-enabled devices

The `localKeyExistsAndPasskeysAreAvailable()` method is invoked when the SDK detects an existing WebAuthn key on the device, and that the device supports passkeys.

```
// MARK: PlatformAuthenticatorAuthenticationDelegate
func localKeyExistsAndPasskeysAreAvailable() {
    // You can prompt the user to Register a new Key to use with Apple Passkeys. From then on, use the `register` and
`authenticate` methods passing the `usePasskeysIfAvailable: true`.
}
```

You can use this delegate method to offer a journey that reregisters the device. Make sure you set `usePasskeysIfAvailable` to `true`.

Request consent

You might need to ask the user for consent to perform certain actions depending on the configuration of the authentication journey.

The Ping SDK for iOS provides the `PlatformAuthenticatorRegistrationDelegate` protocol for requesting user consent:

```
public protocol PlatformAuthenticatorRegistrationDelegate {
    func excludeCredentialDescriptorConsent(consentCallback: @escaping WebAuthnUserConsentCallback)
    func createNewCredentialConsent(keyName: String, rpName: String, rpId: String?, userName: String,
userDisplayName: String, consentCallback: @escaping WebAuthnUserConsentCallback)
}
```

Request consent when credentials already exist for the device

The SDK invokes the `excludeCredentialDescriptorConsent()` method when `Limit registrations` is **enabled** in the WebAuthn Registration node.

This setting prevents a device from being registered if the server has a set of matching keys already stored for it.

During registration, the server returns a list of key descriptor identifiers that the SDK compares with its stored keys. If there is a match, you must get consent from the user to generate a new set of identifiers without explaining the reason, which is they already exist.

For more information, refer to section (6.3.2.3) in the WebAuthn specification.

The following example shows how to request consent:

```
func excludeCredentialDescriptorConsent(consentCallback: @escaping WebAuthnUserConsentCallback) {
    let alert = UIAlertController(title: "Create Credentials", message: nil, preferredStyle: .alert)
    let cancelAction = UIAlertAction(title: "Cancel", style: .cancel, handler: { (_) in
        consentCallback(.reject)
    })
    let allowAction = UIAlertAction(title: "Allow", style: .default) { (_) in
        consentCallback(.allow)
    }
    alert.addAction(cancelAction)
    alert.addAction(allowAction)

    guard let vc = self.viewController else {
        return
    }

    DispatchQueue.main.async {
        viewController.present(alert, animated: true, completion: nil)
    }
}
```

If the user selects **Allow**, the SDK returns `WebAuthnError.notAllowed`. If the user selects **Cancel**, the SDK returns `WebAuthnError.invalidState`.

Request consent to create new credentials

The SDK invokes the `createNewCredentialConsent()` method to obtain user consent prior to the SDK generating a key-pair.

In addition to the consent, the SDK might prompt for biometric authentication if the WebAuthn Registration node's `User verification requirement` is set to `PREFERRED` or `REQUIRED`.

For more information, refer to section 6.3.2.6 in the WebAuthn specification.

The following example shows how to request consent:

```
func createNewCredentialConsent(
    keyName: String,
    rpName: String,
    rpId: String?,
    userName: String,
    userDisplayName: String,
    consentCallback: @escaping WebAuthnUserConsentCallback)
    {
        let alert = UIAlertController(
            title: "Create Credentials",
            message: "KeyName: \(keyName) | Relying Party Name: \(rpName) | User Name: \(userName)",
            preferredStyle: .alert)

        let cancelAction = UIAlertAction(
            title: "Cancel",
            style: .cancel,
            handler: { (_) in
                consentCallback(.reject)
        })

        let allowAction = UIAlertAction(
            title: "Allow",
            style: .default) { (_) in
                consentCallback(.allow)
        }
        alert.addAction(cancelAction)
        alert.addAction(allowAction)

        guard let vc = self.viewController else {
            return
        }

        DispatchQueue.main.async {
            viewController.present(alert, animated: true, completion: nil)
        }
    }
```

If the user selects **Allow**, the SDK creates the key pair and performs the attestation. If the user selects **Cancel**, the SDK returns `WebAuthnError.cancelled`.

Authenticate by using a WebAuthn device

After the user's mobile device has been registered in PingAM, the device can be used as an authenticator with its registered key pair through the WebAuthn Authentication node, which is returned as a `WebAuthnAuthenticationCallback` by the Ping SDK for iOS.

If the device supports Passkeys, the operating system displays passkeys that can be used:

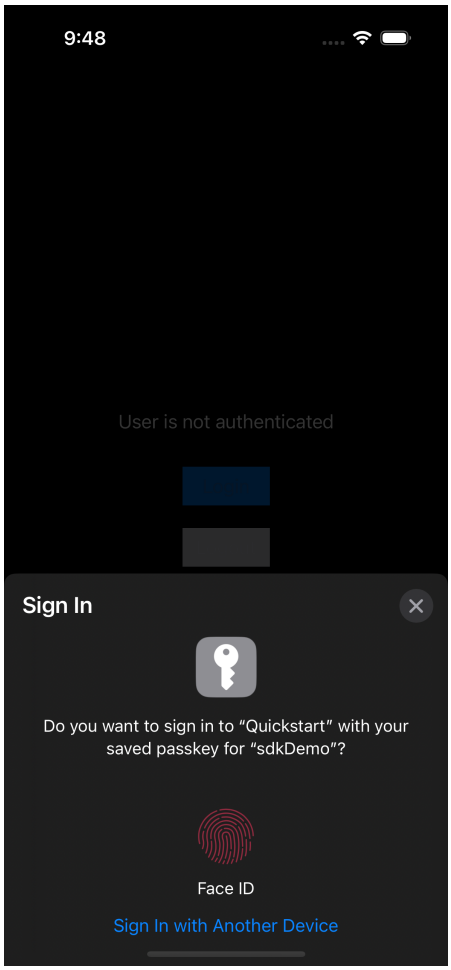


Figure 1. Select a passkey to use for WebAuthn

Note that removing credentials stored on the client device does not remove the associated data from the server. You will need to register the device again after removing credentials from the client.

With `WebAuthnAuthenticationCallback`, you must implement the following protocol method to handle the authentication process:

```
public protocol PlatformAuthenticatorAuthenticationDelegate {
    func selectCredential(keyNames: [String], selectionCallback: @escaping WebAuthnCredentialsSelectionCallback)
}
```

As part of authentication process, the SDK provides the `WebAuthnAuthenticationCallback` for authenticating the device as a credential.

```
if let authenticationCallback = callback as? WebAuthnAuthenticationCallback {

    authenticationCallback.delegate = self

    // Note that the `Node` parameter in `.authenticate()` is an optional parameter.
    // If the node is provided, the SDK automatically returns the assertion
    // in the HiddenValueCallback.

    authenticationCallback.authenticate(
        node: node,
        window: UIApplication.shared.windows.first,
        preferImmediatelyAvailableCredentials: false,
        usePasskeysIfAvailable: true
    ) { (assertion) in
        // Authentication is successful
        // Submit the Node using Node.next()
    } onError: { (error) in
        // An error occurred during the authentication process
        // Submit the Node using Node.next()
    }
}
```

Set the `usePasskeysIfAvailable` parameter to `true` to enable passkeys on supported devices.

When passkeys are enabled, the device offers the ability to sign in using passkeys stored an a separate supported device, by first scanning a QR code.

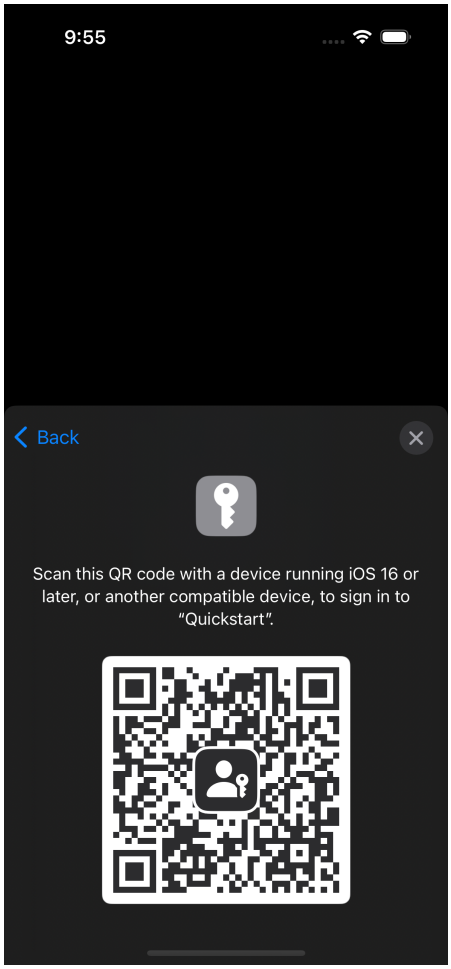


Figure 2. Use a stored passkey from another device by first scanning the QR code

To prevent this behavior and only accept passkeys stored on the initial client device, set the `preferImmediatelyAvailableCredentials` parameter to `true`.

> **Note**
>
> The `WebAuthnAuthenticationCallback.authenticate()` method has an optional parameter, `Node`.
> If the current node contains both `WebAuthnAuthenticationCallback` and `HiddenValueCallback` callbacks, and this node is passed as a parameter to the `WebAuthnAuthenticationCallback.authenticate()` method, then the SDK automatically returns the outcome of the authentication process for both success and failure into the designated `HiddenValueCallback`.
> If the node is not provided, the assertion or error outcome **must** be set manually.

Select credentials

The `func selectCredential()` method is invoked when `Username from device` is enabled in the WebAuthn Authentication node. This feature requires that `Username to device` is enabled in the WebAuthn Registration node as well. With these options enabled, the registered key pair is associated with the username, and the SDK can present a list of registered keys to the user to continue the authentication process without collecting a username.

> **Note**
>
> The `keyName` is an array of strings constructed as `<User's displayName> <Registered Timestamp>`.
> You may alter the string value, and present the altered value to the user, but you **must** return the key name string as it was provided in the original array.

```
func selectCredential(keyNames: [String], selectionCallback: @escaping WebAuthnCredentialsSelectionCallback) {
    let actionSheet = UIAlertController(title: "Select Credentials", message: nil, preferredStyle: .actionSheet)

    for keyName in keyNames {
        actionSheet.addAction(UIAlertAction(title: keyName, style: .default, handler: { (action) in
            selectionCallback(keyName)
        }))
    }

    actionSheet.addAction(UIAlertAction(title: "Cancel", style: .cancel, handler: { (action) in
        selectionCallback(nil)
    }))

    guard let vc = self.viewController else {
        return
    }

    if actionSheet.popoverPresentationController != nil {
        actionSheet.popoverPresentationController?.sourceView = self
        actionSheet.popoverPresentationController?.sourceRect = self.bounds
    }

    DispatchQueue.main.async {
        viewController.present(actionSheet, animated: true, completion: nil)
    }
}
```

Error handling

When an error occurs during the registration or authentication process, the Ping SDK for iOS returns the following error. In most cases, errors are returned as per the specification:

```
public enum WebAuthnError: FRError {
    case badData
    case badOperation
    case invalidState
    case constraint
    case cancelled
    case timeout
    case notAllowed
    case unsupported
    case unknown
}
```

Error handling with PingAM node outcome

When you use `WebAuthnRegistrationCallback.register()` or `WebAuthnAuthenticationCallback.authenticate()`, the SDK automatically parses the error into the appropriate format for PingAM. When PingAM receives the node, the authentication flow follows the WebAuthn registration process to reach the appropriate outcome.

However, if the error has to be handled manually, the `WebAuthnError` provides a convenience method called `WebAuthnError.convertToWebAuthnOutcome()` to convert the error into an appropriate format.

> **Note**
>
> `WebAuthnError.unsupported` results in `Unsupported` outcome in both the WebAuthn Registration and WebAuthn Authentication nodes.
> Any other `WebAuthnError` results in a `Client Error` outcome in the nodes.

```
// keep HiddenValueCallback to set the value later
let hiddenValueCallback = HiddenValueCallback
if let registrationCallback = callback as? WebAuthnRegistrationCallback {
    //  Perform registration
    registrationCallback.register { (attestation) in
        // Registration successful

        // only if HiddenValueCallback is designated for WebAuthn outcome
        if hiddenValueCallback.isWebAuthnOutcome {
            // set attestation manually
            hiddenValueCallback.setValue(attestation)
        }
        // Submit the Node using Node.next()
    } onError: { (error) in
        // only if HiddenValueCallback is designated for WebAuthn outcome
        // and, the error is WebAuthnError
        if hiddenValue.isWebAuthnOutcome, let webAuthnError = error as? WebAuthnError {
            // set error outcome manually
            hiddenValueCallback.setValue(webAuthnError.convertToWebAuthnOutcome())
        }
        // Submit the Node using Node.next()
    }
}
```

Unregister a WebAuthn device

To unregister a WebAuthn device from a user's profile, use the `deleteCredential` function in your application. The function requires the `publicKeyCredentialSource` as a parameter.

Use the `loadAllCredentials` method and pass in the relying party identifier (rpId) string to return an array of `publicKeyCredentialSource` values. The rpId string must match the configuration you used when you configured the authentication journeys earlier.

> **Note**
>
> You can only remove a device if it has the username embedded in the profile.
> You must enable the **Username to Device** option in the WebAuthn Registration node to be able to remove the device from a user's profile on the server using the SDKs.

The SDK attempts to delete the record of the device from the server. If that succeeds, it will then remove the local keys held by the client device. If it fails to remove the records from the server, it will not remove the local keys by default.

However, you can pass the `forceDelete: true` boolean parameter to the function to delete the local keys even if the call to the server fails.

```
let rpId = "openam-docs.forgeblocks.com"

if let credentialSource = FRWebAuthn.loadAllCredentials(
  by: rpId
).first {
  try? FRWebAuthn.deleteCredential(
    publicKeyCredentialSource: credentialSource,
    forceDelete: true
  )
}
```

Removing keys from either the server or the device means you will need to register it again for WebAuthn journeys. Refer to Register a WebAuthn device.

More information

- `deleteCredential` API reference
- `loadAllCredentials` API reference

Web biometrics

Applies to:

- ✕ Ping SDK for Android
- ✕ Ping SDK for iOS
- ✓ Ping SDK for JavaScript

What are web biometrics?

Web biometrics let users authenticate by using an authenticator device. For example:

- The fingerprint scanner on a laptop or phone.
- Face ID.

Communication with the authentication devices is handled by the SDK. You can configure the nodes in PingAM to request that the SDK activates authenticators with certain criteria. For example, the authenticator can be:

- Built into the platform.
- A cross-platform USB device.

- Bluetooth.
- NFC.

You can also configure PingAM to request that the device verify the identity of the user, or that a user is present.

To use web biometrics, users must first register their authenticators. If recovery codes are enabled, users must also make a copy of their codes.

Registration involves the selected authenticator creating or minting, a key pair. The key pair is specific to the origin of the site that uses it. This helps fight against phishing attacks.

The public key of the pair is sent to PingAM and stored in the user's profile. The private key is stored securely, either in the authenticator itself or in the platform managing the authenticators. The private key does not leave the client at any time.

When authenticating using web biometrics, PingAM sends a challenge to the authenticator, expecting it to use this challenge to create a signed assertion with its stored, private key.

The assertion is then sent to PingAM for verification using the public key stored in the user's profile. If the data is verified as being from the correct device, and passes any attestation checks, the authentication is successful.

Differences between device binding and WebAuthn

There are many similarities between WebAuthn and Device Binding and JWS verification. We provide authentication nodes to implement both technologies in your journeys.

Both can be used for usernameless and passwordless authentication, they both use public key cryptography, and both can be used as part of a multi-factor authentication journey.

One major difference is that with device binding, the private key never leaves the device.

With WebAuthn, there is a possibility that the private key is synchronized across client devices because of Passkey support, which may be undesirable for your organization.

For more details of the differences, refer to the following table:

Comparison of WebAuthn and Device Binding/JWS Verification

Feature	WebAuthn / FIDO	Device Binding / JWS Verifier	Details
Industry-standards based	☑	×	You can refer to the WebAuthn W3C specification. Device binding and JWS verification are proprietary implementations.
Public key cryptography	☑	☑	Both methods use Public key cryptography.
Usernameless support	☑	☑	After registration, the username can be stored in the device and obtained during authentication without the user having to enter their credentials.

Feature	WebAuthn / FIDO	Device Binding / JWS Verifier	Details
Keys are bound to the device	✕	☑	With WebAuthn, if Passkeys are used, they can be shared across devices. With device binding, the private keys do not leave the device.
Sign custom data	✕	☑	With device binding, you can: • Customize the challenge that the device must sign. For example, you could include details of a transaction, such as the amount in dollars. • Add custom claims to the payload when signing a challenge. This gives additional context that the server can make use of by using a scripted node. Refer to Add custom claims when signing.
Format of signed data	WebAuthn authenticator data	JSON Web Signature (JWS)	
Integration	✕	☑	With device binding, after verification, the signed JWT is available in: • Audit Logs • Transient node state This enables the data within to be used for integration into your processes and business logic.
Platform support	☑ Android ☑ iOS ☑ Web browsers	☑ Android ☑ iOS ✕ Web browsers	As it is challenging to store secure data in a browser as a client app, device binding is not supported in web browsers.

Feature	WebAuthn / FIDO	Device Binding / JWS Verifier	Details
Authenticator support	Determined by the platform. Configuration limited to: • Biometric with Fallback to Device Pin	Determined by the authentication node. Full configuration options: • Biometric Authentication • Biometric with Fallback to Device Pin • Application Pin • Silent	With device binding, you can specify what authentication action the user must perform to get access to the private keys. This provides greater flexibility in your security implementation and can reduce authentication friction for your users.
Key storage	Web browsers and iOS synchronize to the cloud. Android has the option to synchronize to the cloud.	***Android*** KeyStore ***iOS*** Secure enclave: hardware-backed and not synchronized to the cloud.	Both technologies store the private keys securely on the client. WebAuthn supports synchronizing the private keys to the cloud for use on other devices. This can reduce authentication friction for your users but may also increase the risk of a breach.
Managing device keys	Managed by the device OS. Apps cannot delete *local* client keys programmatically and do not have a reference to the *remote* server key for deletion.	Managed by the Ping SDKs. Provides an interface to delete local client and remote server keys.	The ability to programmatically delete both client and server keys can greatly simplify the process of registering a new device if an old device is lost or stolen.
Passkey Support	☑	×	WebAuthn supports synchronizing the private keys to the cloud for use on other devices. Device binding keeps the private key locked in the device.

Feature	WebAuthn / FIDO	Device Binding / JWS Verifier	Details
App integrity verification	***Android*** Requires an `assetlinks.json` file. ***iOS*** Requires `apple-app-site-association` file.	Not provided by the device binding or verification nodes. It can be added as part of the journey by using app integrity nodes.	App integrity verification helps ensure your users are only using a supported app rather than a third-party or potentially malicious version.
Key attestation	***Android*** SafteyNet ***iOS*** None	***Android*** Uses hardware-backed key pairs with Key Attestation. ***iOS*** It can be added as part of the journey by using app integrity nodes to support key attestation.	Key attestation verifies that the private key is valid and correct, is not forged, and was not created in an insecure manner.
Complexity	Medium	Low	WebAuthn requires a bit more configuration, for example, creating and uploading the `assetlinks.json` and `apple-app-site-association` files. Device binding only requires the journey and the SDK built into your app.

Before using web biometrics

- The device must have a biometric (platform) authenticator.
- The OS must provide access to the platform authenticator via the API.
- The browser must support WebAuthn capabilities, and support the OS's platform authenticator's API.

If any of the above prerequisites is missing, web biometrics will not work.

For more information on support in PingAM, see Minimum Web Authentication User Agent Versions in the PingAM documentation.

Prepare for web biometrics

You must create an authentication journey in PingAM to authenticate users by using a web biometrics device. The journey also allows users to register a device if they have not already done so.

For more information, see Creating Trees for Web Authentication (WebAuthn) in the PingAM documentation.

Handle web biometrics

The Ping SDK for JavaScript has the following methods for handling web biometrics:

`FRWebAuthn.register(step, deviceName)`

For registering new devices. Optionally, assign a name to the device to help the user identify it. If you do not provide a custom name, the server assigns a generic value such as **New Security Key**.

`FRWebAuthn.authenticate(step)`

For authenticating using a previously registered device.

Use the `FRWebAuthn.getWebAuthnStepType()` convenience method to determine which method to use:

```
// Determine if a step is a web biometrics step
const stepType = FRWebAuthn.getWebAuthnStepType(step);

if (stepType === WebAuthnStepType.Registration) {
  // Registering a new device, with optional device name
  step = await FRWebAuthn.register(step, 'myDeviceName');
} else if (stepType === WebAuthnStepType.Authentication) {
  // Authenticating with a registered device
  step = await FRWebAuthn.authenticate(step);
}

// `step` has now been populated with the web biometrics credentials

// Send this new step to the {fr_server}
nextStep = FRAuth.next(step);
```

Using the device name

The device name is available for display in other callbacks received from a journey.

For example, you can get the device name when displaying recovery codes as follows:

```
// Determine if step is a display recovery codes step
const isDisplayRecoveryCodesStep = FRRecoveryCodes.isDisplayStep(step);

if (isDisplayRecoveryCodesStep) {
  // Obtain recovery codes
  const recoveryCodes = FRRecoveryCodes.getCodes(step);

  // Obtain device display name
  const deviceName = FRRecoveryCodes.getDisplayName(step);

  // Display `recoveryCodes` and `deviceName` to the user
}
```

Tip

The Ping (ForgeRock) Login Widget has built-in support and associated UI for displaying the device name alongside the recovery codes.

Set up passwordless authentication with passkeys

What is passwordless?

Passwordless authentication is the term used to describe a group of identity verification methods that don't rely on users entering passwords. There are many ways to go passwordless, such as supporting biometrics, hardware security keys, or the use of specialized mobile applications like authenticators that can all provide a secure alternative to inputting a password and sending it over the network.

We offer extensive support for passwordless, including OTPs either via email or SMS, an authenticator application, push notifications with number challenges and biometric unlock, magic links, WebAuthn, and more.

All these methods can be used in passwordless scenarios or as additional factors of authentication (2FA/MFA) to secure your systems further. Some of these require an authenticator application, such as the ForgeRock Authenticator, while others just on existing channels like email or SMS. Using the Ping SDKs, developers can include the functionality of the ForgeRock Authenticator within their own applications.

In this blog post, we focus more on using biometrics for going passwordless. The Ping SDKs support the WebAuthn protocol, offering out-of-the-box nodes in both PingAM and PingOne Advanced Identity Cloud. Furthermore, using the SDKs, developers can utilize the power of passkeys on every supported platform.

Biometrics and WebAuthn

What are these technologies, and how can we use them? Let's dive a bit deeper into that.

WebAuthn is an abbreviation of *Web Authentication*. It is a specification issued by W3C. It specifies a set of interfaces for browsers and apps to implement.

To use the WebAuthn protocol, the user requires access to a strong authenticator. Newer laptops and most Android and iOS mobile devices include biometric sensors that can be used for this. Those biometric scanners, more commonly known by their marketing names such as FaceID or TouchID, are used to register the user's biometric data with the mobile operating system. They can be used to unlock the device itself, unlock information stored in the secure storage, and more.

WebAuthn requires two distinct ceremonies:

Registration

During registration, the device (called "authenticator") generates a cryptographic keypair. The public key is sent to the server, and the private key is safely stored locally.

Authentication

During authentication, the server asks the client to sign a message with a nonce (called "challenge") with its private key.

This signed message with the nonce can then be verified by the server using the public key obtained through registration.

This has really strong security properties because the private key is hardware bound and never leaves the device. The signing of the message is done directly by the *authenticator*, the device, and protected by some form of local user verification (PIN or Biometrics).

Differences between WebAuthn keys and Passkeys

So, what is the difference between WebAuthn keys, as described above, and passkeys? Until now, the private key created during the registration process was stored on the device. This has one shortcoming; if the user changes the device or loses it, they cannot authenticate again. Moreover, the server needs to allow for registration of more than one key if users have multiple devices for authenticating to a website or service.

Apple, Google, and Microsoft chose *passkeys*, an implementation of WebAuthn with the additional feature of storing the user's private keys in their respective cloud services. That means that those "passkeys" are available to use on all the devices logged in to the same cloud account.

This means Apple, Google, and Microsoft are responsible for keeping the user's private keys safe. Also, it is up to the user to ensure their account on these providers is secure by using strong passwords, MFA, and so on.

Although this makes the attack vector broader, this way of handling keys makes the whole passwordless experience more accessible and, therefore, more likely to be used by the everyday user. Additionally, it makes account recovery due to a single lost device a thing of the past.

Differences between WebAuthn and Device Binding

There are many similarities between WebAuthn and Device Binding and JWS verification. We provide authentication nodes to implement both technologies in your journeys.

Both can be used for usernameless and passwordless authentication, they both use public key cryptography, and both can be used as part of a multi-factor authentication journey.

One major difference is that with device binding, the private key never leaves the device.

With WebAuthn, there is a possibility that the private key is synchronized across client devices because of Passkey support, which may be undesirable for your organization.

For more details of the differences, refer to the following table:

Comparison of WebAuthn and Device Binding/JWS Verification

Feature	WebAuthn / FIDO	Device Binding / JWS Verifier	Details
Industry-standards based	☑	×	You can refer to the WebAuthn W3C specification. Device binding and JWS verification are proprietary implementations.
Public key cryptography	☑	☑	Both methods use Public key cryptography.
Usernameless support	☑	☑	After registration, the username can be stored in the device and obtained during authentication without the user having to enter their credentials.
Keys are bound to the device	×	☑	With WebAuthn, if Passkeys are used, they can be shared across devices. With device binding, the private keys do not leave the device.
Sign custom data	×	☑	With device binding, you can: • Customize the challenge that the device must sign. For example, you could include details of a transaction, such as the amount in dollars. • Add custom claims to the payload when signing a challenge. This gives additional context that the server can make use of by using a scripted node. Refer to Add custom claims when signing.
Format of signed data	WebAuthn authenticator data	JSON Web Signature (JWS)	
Integration	×	☑	With device binding, after verification, the signed JWT is available in: • Audit Logs • Transient node state This enables the data within to be used for integration into your processes and business logic.

Feature	WebAuthn / FIDO	Device Binding / JWS Verifier	Details
Platform support	☑ Android ☑ iOS ☑ Web browsers	☑ Android ☑ iOS ✕ Web browsers	As it is challenging to store secure data in a browser as a client app, device binding is not supported in web browsers.
Authenticator support	Determined by the platform. Configuration limited to: • Biometric with Fallback to Device Pin	Determined by the authentication node. Full configuration options: • Biometric Authentication • Biometric with Fallback to Device Pin • Application Pin • Silent	With device binding, you can specify what authentication action the user must perform to get access to the private keys. This provides greater flexibility in your security implementation and can reduce authentication friction for your users.
Key storage	Web browsers and iOS synchronize to the cloud. Android has the option to synchronize to the cloud.	***Android*** KeyStore ***iOS*** Secure enclave: hardware-backed and not synchronized to the cloud.	Both technologies store the private keys securely on the client. WebAuthn supports synchronizing the private keys to the cloud for use on other devices. This can reduce authentication friction for your users but may also increase the risk of a breach.
Managing device keys	Managed by the device OS. Apps cannot delete *local* client keys programmatically and do not have a reference to the *remote* server key for deletion.	Managed by the Ping SDKs. Provides an interface to delete local client and remote server keys.	The ability to programmatically delete both client and server keys can greatly simplify the process of registering a new device if an old device is lost or stolen.
Passkey Support	☑	✕	WebAuthn supports synchronizing the private keys to the cloud for use on other devices. Device binding keeps the private key locked in the device.

Feature	WebAuthn / FIDO	Device Binding / JWS Verifier	Details
App integrity verification	***Android*** Requires an `assetlinks.json` file. ***iOS*** Requires `apple-app-site-association` file.	Not provided by the device binding or verification nodes. It can be added as part of the journey by using app integrity nodes.	App integrity verification helps ensure your users are only using a supported app rather than a third-party or potentially malicious version.
Key attestation	***Android*** SafteyNet ***iOS*** None	***Android*** Uses hardware-backed key pairs with Key Attestation. ***iOS*** It can be added as part of the journey by using app integrity nodes to support key attestation.	Key attestation verifies that the private key is valid and correct, is not forged, and was not created in an insecure manner.
Complexity	Medium	Low	WebAuthn requires a bit more configuration, for example, creating and uploading the `assetlinks.json` and `apple-app-site-association` files. Device binding only requires the journey and the SDK built into your app.

How to implement Passkeys using Ping SDKs

The first step is having access to a PingAM instance or a PingOne Advanced Identity Cloud tenant, as well as an existing application that uses these for authentication. In this example, we use PingOne Advanced Identity Cloud.

Download the sample app

We provide a sample app we've built that implements authentication using the Ping SDK for iOS. You can download the full iOS project from the SDK Sample Apps repo on GitHub.

Create WebAuthn registration and authentication journeys

In PingOne Advanced Identity Cloud, we will create new journeys for both WebAuthn device registration and authentication.

Tip

To speed up the process of creating the required journeys, we provide a pre-configured JSON file that you can import into your ID Cloud tenant.

Download

Click here to download the JSON file that you can import into your tenant to create the required journeys.

This automatically creates both of the required journeys, as well as the scripts you require in the scripted nodes.
To import the JSON file, in the PingOne Advanced Identity Cloud admin UI, go to **Journeys**, and then click **Import**.
After successfully importing the journeys into your tenant, skip ahead and Configure the WebAuthn nodes in the journeys.
For more information on importing journeys, refer to Import journeys.

Create the registration journey manually

If you decided not to import the journey file into your tenant, you need to create the journey manually.

Using the Journey editor, create a new journey in the alpha realm and name it `BlogWebAuthnRegistration`. Then, drag the following nodes from the list and connect them as displayed on the screenshot below:

- Four scripted Decision nodes
- WebAuthn Registration Node
- Get Session Data Node
- Username Collector Node
- Password Collector Node
- Data Store Decision Node

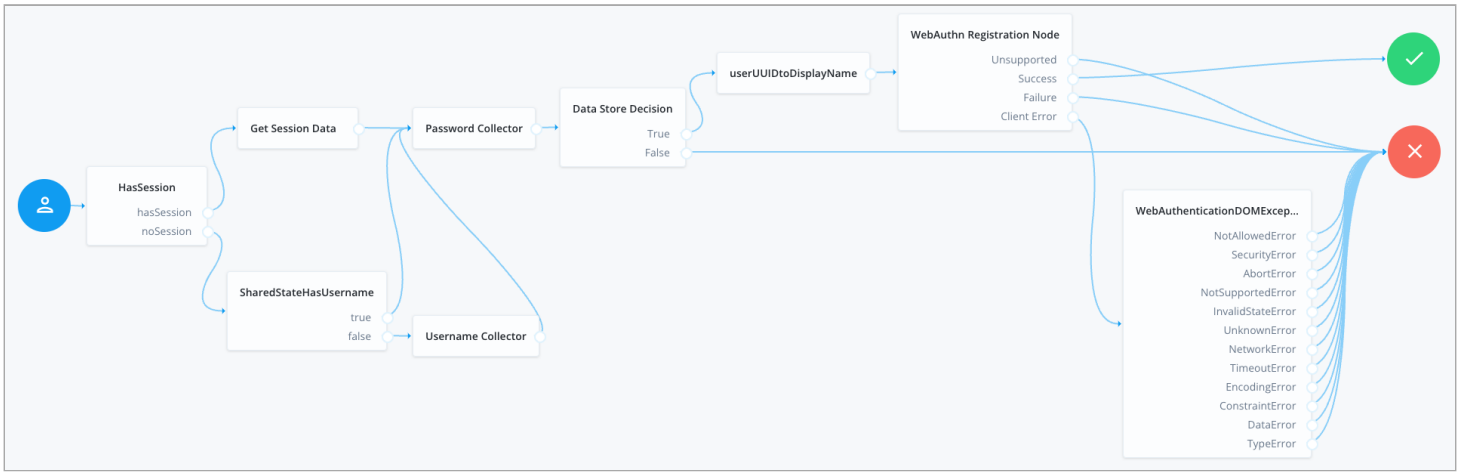


Figure 1. The **`BlogWebAuthnRegistration`** *journey*

We need to assign the scripts for the scripted decision nodes. First, `setUUIDtoDisplayName`, and second, `WebAuthnErrorHandler`. The first one ensures the creation of a user-friendly name for our passkey, and the second allows developers to handle the WebAuthn client error cases in more detail.

Source for the `userUUIDtoDisplayName` script

```
var user = nodeState.get('username').asString();
nodeState.putShared('displayName', user.toString());

outcome = 'true';
```

Source for the `WebAuthnErrorHandler` script

```
// Error format example:
// ERROR::InvalidStateError:No Credential is registered

var error = sharedState.get("WebAuthenticationDOMException");
logger.message(error);

// Match word or phrase between "::" and ":"
var result = error.match(/::([\w\s]{1,}):{0,}/);
outcome = result ? result[1] : 'UnknownError';

logger.message("Outcome: " + outcome + "| ERROR: " + error);
```

Next, we set up the "HasSession" and "SharedStateHasUsername" scripts.

Source for the `HasSession` script

```
if (typeof existingSession !== 'undefined') {
  outcome = "hasSession";
} else {
  outcome = "noSession";
}
```

Source for the `SharedStateHasUsername` script

```
var user = nodeState.get('username');

if (user != null) {
  outcome = 'true';
} else {
  outcome = 'false';
}
```

Create the authentication journey manually

If you decided not to import the journey file into your tenant, you need to create the journey manually.

Using the journey editor, create a new journey named `BlogWebAuthnAuthentication`. For this journey, use the following nodes:

- Scripted Decision node
- WebAuthn Authentication node
- Inner Tree Evaluator node (this calls the `BlogWebAuthnRegistration` you created previously)

Connect them as follows:

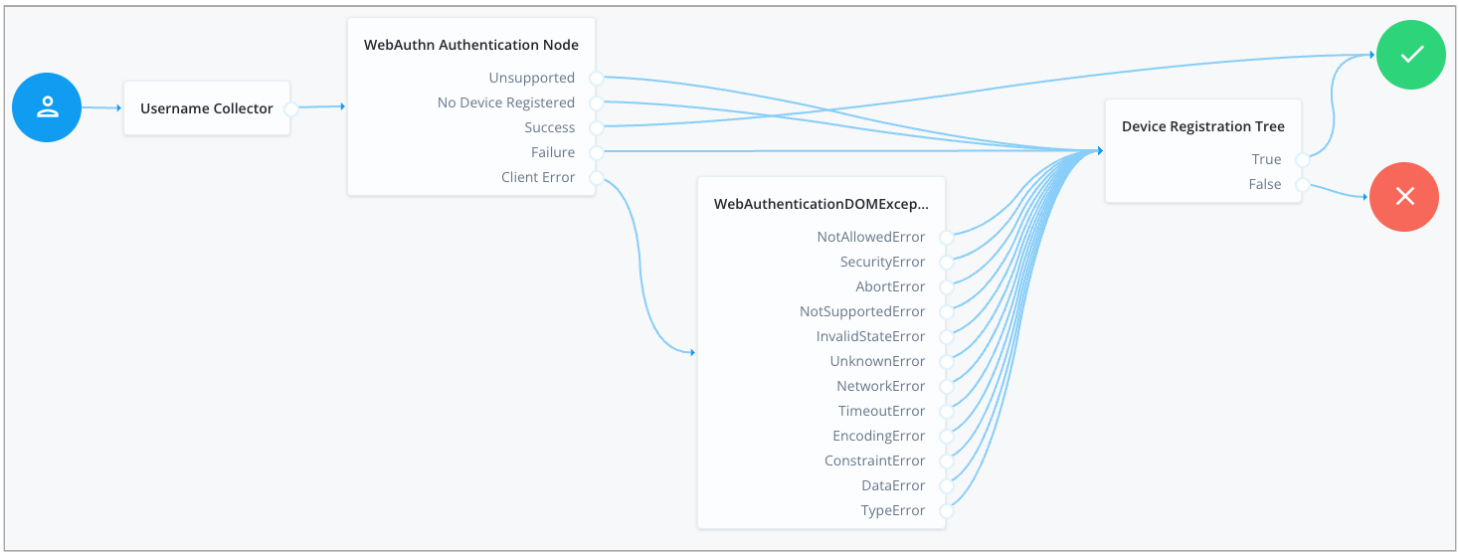


Figure 2. The `BlogWebAuthnAuthentication` journey

Configure the WebAuthn nodes in the journeys

You must configure the **WebAuthn Registration** node and the **WebAuthn Authentication** node in the new journeys with values matching your environment.

Open each newly created tree and configure the WebAuthn nodes within to use identical configuration.

The important configuration options in this case are the following fields:

Relying party identifier

This needs to be set to the domain that will be the "Relying Party" for the registration. Set this to the domain name of your tenant. If you are using custom domains, set this to match the custom domain configured for the realm.

Origin domains

This needs to be set to the origin of the application that registers passkeys.

For iOS and Android, this involves special configuration depending on the platform and whether you use plain WebAuthn keys or passkeys.

For more information, refer to the following:

- Configure Android origin domains
- Configure iOS origin domains

When implementing passkeys, set this to the origin that serves the `apple-app-site-association` file.

Return challenge as JavaScript

Ensure this is **NOT** enabled.

Shared state attribute for display name

Set to `displayName` as indicated by the script above.

Lastly, in order to allow the application to register and authenticate against the server using passkeys, we need to configure and upload the `apple-app-site-association` file.

For more details on how to do this in PingOne Advanced Identity Cloud, refer to Prepare an apple-app-site-association file.

With both journeys configured, the server is able to register a device for passkeys and authentication.

In the `BlogWebAuthnAuthentication` journey, you will notice that if the authentication step fails, the user proceeds to the registration step automatically. This is acceptable based on the requirements for the scope we have in this blogpost, but in other scenarios allowing the user to authenticate with other means such as a password or OTP is advisable.

The `BlogWebAuthnRegistration` journey is built in a way that allows applications to call it directly when a user session exists or call it internally from another journey.

Test the journeys in a browser

Using the out-of-the-box platform user interface you can test the functionality in a browser. Start by copying the **Preview URL** from the journey editor for the `BlogWebAuthnAuthentication` journey.

Running for the first time, the flow should look something like this:

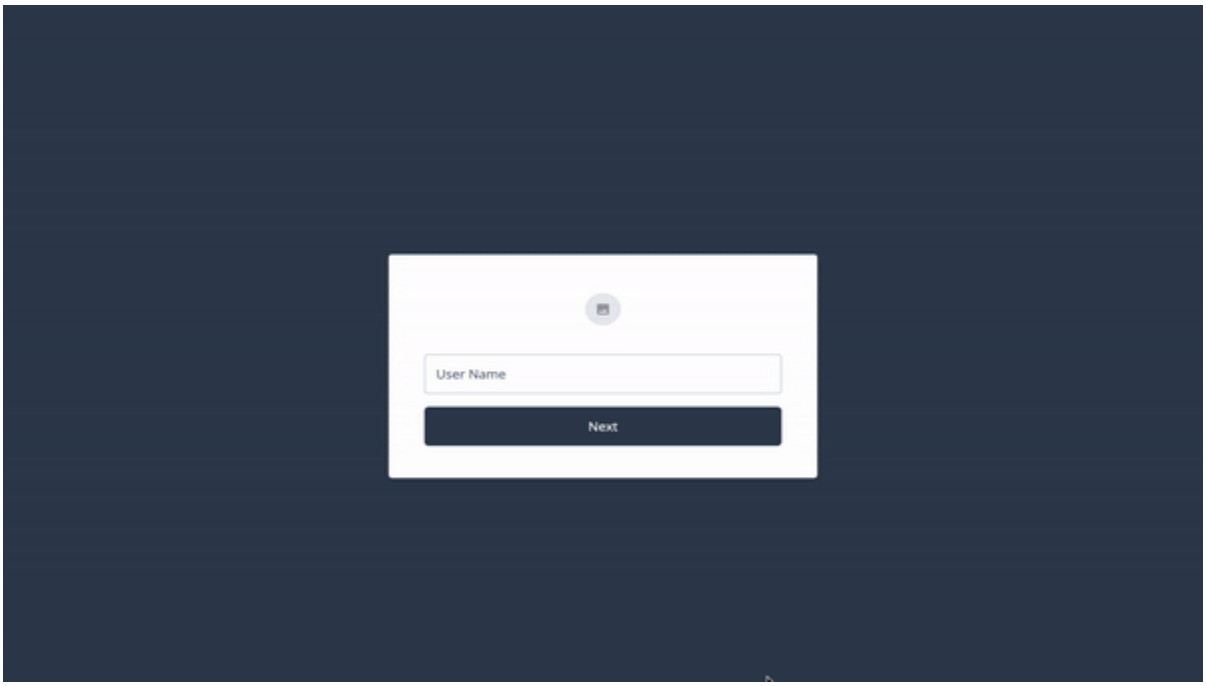


Figure 3. Registering a new passkey on the first attempt to authenticate

In subsequent authentication attempts, you are able to authenticate using your newly created passkey:

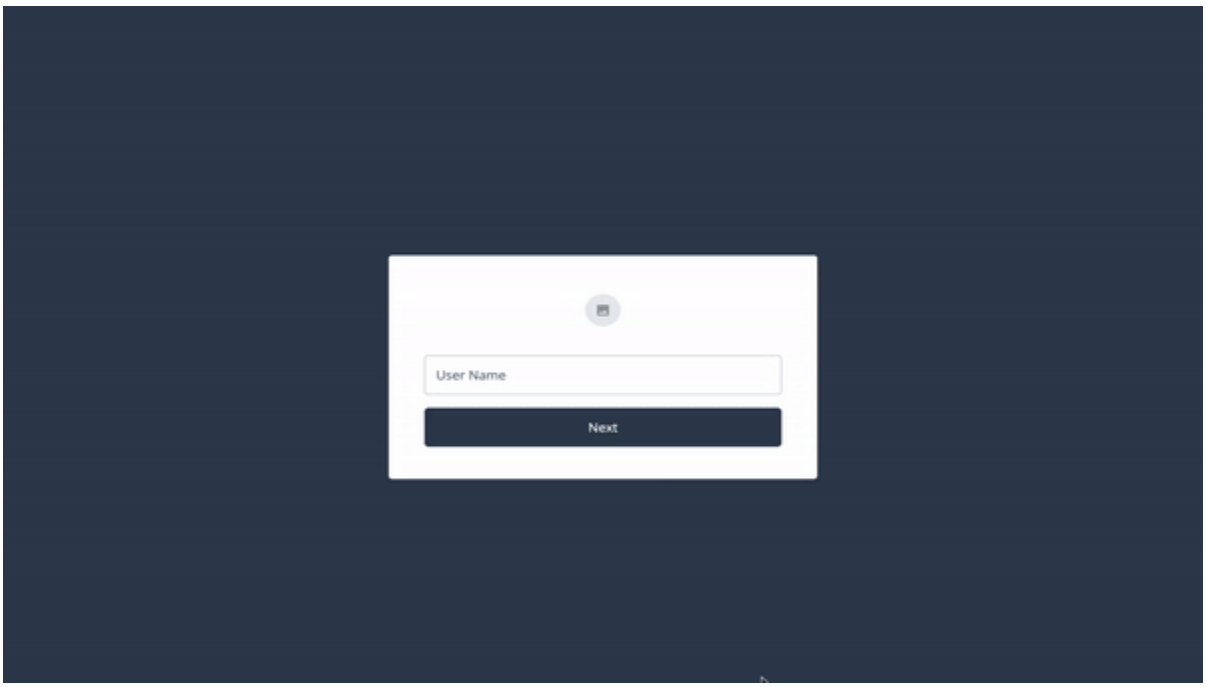


Figure 4. Using the new passkey on a subsequent attempt to authenticate

Using Passkeys with the Ping SDK for iOS

At this point it is advisable to download the complete project from GitHub. Open the project in Xcode and have a look at the `LoginViewController` and `SettingsViewController` class. The logic described below can be found in those two controllers. If you have an existing project using the Ping SDK, the code should look familiar. This tutorial focuses on the logic regarding passkey (WebAuthn) authentication and registration.

Add support for the callbacks

In order to use passkeys with the Ping SDK for Android or iOS, developers need to handle the `WebAuthnAuthentication` and `WebAuthnRegistration` callbacks.

The first node the app needs to handle on the authentication journey is the `NameCallback` from the username node. We assume your iOS application already handles basic authentication with username and password, so we expect this to be implemented.

The first *new* callback to be handled is the `WebAuthnAuthentication` callback. In the `handleNode` method add some code to do so:

Handling the `WebAuthnAuthentication` callback

```
else if let authenticationCallback = callback as? WebAuthnAuthenticationCallback {
  authenticationCallback.delegate = self
  ...
  ...
}
```

In order to start the WebAuthn authentication flow, we need to call the `authenticationCallback.authenticate()` method:

Starting the WebAuthn flow with Passkey support

```
authenticationCallback.authenticate(node: node, preferImmediatelyAvailableCredentials: false, usePasskeysIfAvailable:
self.usePasskeysIfAvailable) { (assertion) in
     // Authentication is successful
     // Submit the Node using Node.next()
     node.next { (user: FRUser?, node, error) in
           self.handleNode(user: user, node: node, error: error)
     }
  } onError: { (error) in
     // An error occurred during the authentication process
     // Submit the Node using Node.next()
     node.next { (user: FRUser?, node, error) in
           self.handleNode(user: user, node: node, error: error)
     }
  }
```

The full code should look something like this:

Code for handling the authentication journey callbacks

```
if let authenticationCallback = callback as? WebAuthnAuthenticationCallback {
  authenticationCallback.delegate = self

  // Note that the `Node` parameter in `.authenticate()` is an optional parameter.
  // If the node is provided, the SDK automatically sets the assertion to the designated HiddenValueCallback
  authenticationCallback.authenticate(
    node: node,
    usePasskeysIfAvailable: PebbleBankUtilities.usePasskeysIfAvailable
  ) { (assertion) in
    // Authentication is successful
    // Submit the Node using Node.next()
    node.next { (token: Token?, node, error) in
      self.handleNode(token: token, node: node, error: error)
    }
  } onError: { (error) in
    // An error occurred during the authentication process
    // Submit the Node using Node.next()
    let alert = UIAlertController(
      title: "WebAuthnError",
      message: "Something went wrong authenticating the device",
      preferredStyle: .alert
    )
    let okAction = UIAlertAction(
      title: "OK",
      style: .default,
      handler: { (action) in
        node.next { (token: Token?, node, error) in
          self.handleNode(token: token, node: node, error: error)
        }
      }
    )
    alert.addAction(okAction)
    DispatchQueue.main.async {
      self.present(alert, animated: true, completion: nil)
    }
  }
}
```

In a similar way, we need to add support for the `WebAuthnRegistration` callbacks.

Code for handling the registration journey callbacks

```
if let registrationCallback = callback as? WebAuthnRegistrationCallback {
  registrationCallback.delegate = self

  // Note that the `Node` parameter in `.register()` is an optional parameter.
  // If the node is provided, the SDK automatically sets the error outcome or attestation to the designated
HiddenValueCallback
  registrationCallback.register(
    node: node,
    deviceName: UIDevice.current.name,
    usePasskeysIfAvailable: PebbleBankUtilities.usePasskeysIfAvailable
  ) { (attestation) in
    // Registration is successful
    // Submit the Node using Node.next()
    node.next { (token: Token?, node, error) in
      self.handleNode(token: token, node: node, error: error)
    }
  } onError: { (error) in
    // An error occurred during the registration process
    // Submit the Node using Node.next()
    let alert = UIAlertController(
      title: "WebAuthnError",
      message: "Something went wrong registering the device",
      preferredStyle: .alert
    )
    let okAction = UIAlertAction(
      title: "OK",
      style: .default,
      handler: { (action) in
        node.next { (token: Token?, node, error) in
          self.handleNode(token: token, node: node, error: error)
        }
      }
    )
    alert.addAction(okAction)
    DispatchQueue.main.async {
      self.present(alert, animated: true, completion: nil)
    }
  }
}
```

The application can now handle the callbacks returned by each of the nodes that appear on the journey. The full list of expected callbacks is as follows:

- `NameCallback`
- `PasswordCallback`
- `WebAuthnAuthenticationCallback`
- `WebAuthnRegistrationCallback`

Furthermore, we allow the iOS application to call different journeys based on the situation. For example, when the users haven't registered for biometrics, the app calls the default `Login` journey. When the user has followed the `BlogWebAuthnRegistration` journey and has registered for biometrics, the app uses the `BlogWebAuthnAuthentication` journey for authentication.

> **Note**
>
> The sample app has been implemented to call the `BlogWebAuthnRegistration` journey directly from its settings screen.
> This allows the user to register the device for biometrics after successfully authenticating.

Call the journeys

When using the SDKs, we can call a journey directly by using the `FRSession.authenticate` method. In order to call the passkey *registration* journey, we can use the following code:

Calling the passkey registration journey

```
FRSession.authenticate(authIndexValue: "BlogWebAuthnRegistration") { result, node, error in
    self.handleNode(token: result, node: node, error: error)
}
```

In order to call the passkey *authentication* journey:

Calling the passkey authentication journey

```
FRSession.authenticate(authIndexValue: "BlogWebAuthnAuthentication") { result, node, error in
       self.handleNode(token: result, node: node, error: error)
}
```

In the iOS app, upon successful completion of the `BlogWebAuthnRegistration` journey, the SDK saves a flag on the iOS device (in `UserDefaults`) noting that this device is now registered with a passkey.

This client side logic allows us to swap to a passkey authentication journey as the main way of authenticating from this device.

Configure the project

With the sample project open, select the `PebbleBankUtilities` file. This file contains the SDK configuration options. Configure these to point to your environment.

Additionally, this file contains the `ForceAuthInterceptorBiometricRegistration` request interceptor. When using the SDK, developers have the option to create request interceptors that enrich the REST calls the SDK makes. In this case we have added the following:

- A URL query parameter to force the use of the journey despite the presence of an existing valid session:

  `ForceAuth=true`

- A header to inject the session cookie:

  `[Cookie Name]: <SessionToken>`

This request interceptor is only used when the app calls `BlogWebRegistrationJourney`, and injects the existing user session and the `ForceAuth` parameter.

Lastly, the Xcode project needs to be configured to allow `WebCredentials` based on your server configuration. We also need to create an `apple-app-association` file and upload it to PingOne Advanced Identity Cloud.

You can find more details on how to configure Xcode in the Apple developer docs. You can also find more details on how to configure and upload the `apple-app-association` file in the SDK documentation.

Test the app

With the Xcode project fully configured, we can now run and test the flow. A reminder that a complete version of this project can be found on GitHub. Complete documentation on mobile biometrics for iOS and Android can be found in the SDK documentation.

Below is a complete demonstration of the functionality using the demo app:

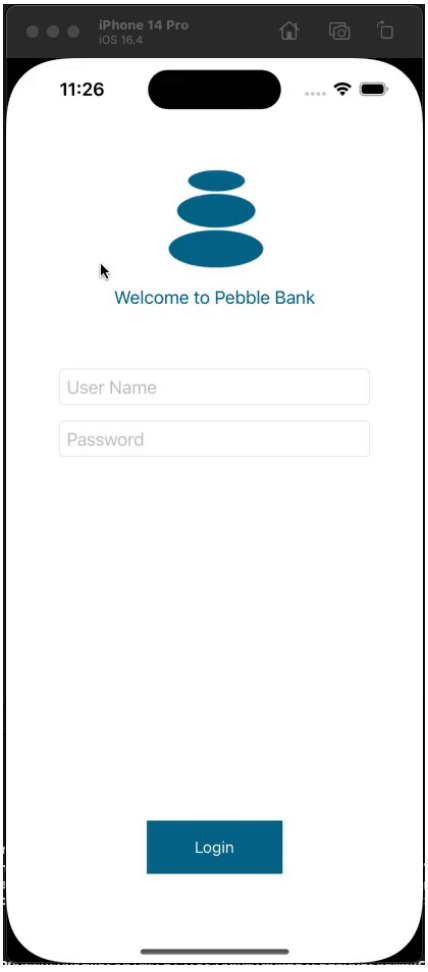


Figure 5. Complete demo of app using passkeys

Summary

Building passwordless authentication for users is not trivial, as they will need to register a device that will act as an authenticator, replacing the password.

Furthermore, users need to be driven down a passwordless journey by choice, or automatically if they have enabled this option in the app.

Using PingOne Advanced Identity Cloud and the Ping SDKs for Android and iOS, developers have a set of tools to make the experience as frictionless as possible and by writing minimal code.

When registering a device with passkeys to replace the traditional username and password, the following considerations should come to mind:

- Is the user or the device registering a valid and authenticated user, or is it a bad actor attempting an account take over?
- What happens if the user attempts to authenticate on a device that does not have the passkey? Will there be an offering for traditional username and password paths?
- Is the flow clear and easy to understand for all users?
- Could the use case support *usernameless* authentication? A step further to make this flow even smoother for end users

Passkeys are here to stay and seem to be a great stepping stone for replacing passwords. Improvements on the user experience from the operating systems and browsers are sure to come in the future.

As this post shows, using the tools provided your applications are ready to go passwordless today!

Bind and verify user devices

Applies to:

- ✓ Ping SDK for Android
- ✓ Ping SDK for iOS
- ✕ Ping SDK for JavaScript

The Ping SDKs for Android and iOS can cryptographically bind a mobile device to a user account.

Registered devices generate a key pair and a key ID. The SDK sends the *public* key and key ID to AM for storage in the user's profile.

The SDK stores the private key on the device in either the Android KeyStore, or the iOS Secure Enclave. Access to the private keys is protected by either biometric security or a PIN.

A user can bind multiple devices to their account, and each device can bind to multiple users.

After binding a device your authentication journeys in AM can verify ownership of the bound device by requesting that it signs a challenge using the private key.

Differences between device binding and WebAuthn

There are many similarities between WebAuthn and Device Binding and JWS verification. We provide authentication nodes to implement both technologies in your journeys.

Both can be used for usernameless and passwordless authentication, they both use public key cryptography, and both can be used as part of a multi-factor authentication journey.

One major difference is that with device binding, the private key never leaves the device.

With WebAuthn, there is a possibility that the private key is synchronized across client devices because of Passkey support, which may be undesirable for your organization.

For more details of the differences, refer to the following table:

Comparison of WebAuthn and Device Binding/JWS Verification

Feature	WebAuthn / FIDO	Device Binding / JWS Verifier	Details
Industry-standards based	☑	×	You can refer to the WebAuthn W3C specification. Device binding and JWS verification are proprietary implementations.
Public key cryptography	☑	☑	Both methods use Public key cryptography.
Usernameless support	☑	☑	After registration, the username can be stored in the device and obtained during authentication without the user having to enter their credentials.
Keys are bound to the device	×	☑	With WebAuthn, if Passkeys are used, they can be shared across devices. With device binding, the private keys do not leave the device.
Sign custom data	×	☑	With device binding, you can: • Customize the challenge that the device must sign. For example, you could include details of a transaction, such as the amount in dollars. • Add custom claims to the payload when signing a challenge. This gives additional context that the server can make use of by using a scripted node. Refer to Add custom claims when signing.
Format of signed data	WebAuthn authenticator data	JSON Web Signature (JWS)	

Feature	WebAuthn / FIDO	Device Binding / JWS Verifier	Details
Integration	✕	☑	With device binding, after verification, the signed JWT is available in: • Audit Logs • Transient node state This enables the data within to be used for integration into your processes and business logic.
Platform support	☑ Android ☑ iOS ☑ Web browsers	☑ Android ☑ iOS ✕ Web browsers	As it is challenging to store secure data in a browser as a client app, device binding is not supported in web browsers.
Authenticator support	Determined by the platform. Configuration limited to: • Biometric with Fallback to Device Pin	Determined by the authentication node. Full configuration options: • Biometric Authentication • Biometric with Fallback to Device Pin • Application Pin • Silent	With device binding, you can specify what authentication action the user must perform to get access to the private keys. This provides greater flexibility in your security implementation and can reduce authentication friction for your users.
Key storage	Web browsers and iOS synchronize to the cloud. Android has the option to synchronize to the cloud.	***Android*** KeyStore ***iOS*** Secure enclave: hardware-backed and not synchronized to the cloud.	Both technologies store the private keys securely on the client. WebAuthn supports synchronizing the private keys to the cloud for use on other devices. This can reduce authentication friction for your users but may also increase the risk of a breach.
Managing device keys	Managed by the device OS. Apps cannot delete *local* client keys programmatically and do not have a reference to the *remote* server key for deletion.	Managed by the Ping SDKs. Provides an interface to delete local client and remote server keys.	The ability to programmatically delete both client and server keys can greatly simplify the process of registering a new device if an old device is lost or stolen.

Feature	WebAuthn / FIDO	Device Binding / JWS Verifier	Details
Passkey Support	☑	✕	WebAuthn supports synchronizing the private keys to the cloud for use on other devices. Device binding keeps the private key locked in the device.
App integrity verification	***Android*** Requires an `assetlinks.json` file. ***iOS*** Requires `apple-app-site-association` file.	Not provided by the device binding or verification nodes. It can be added as part of the journey by using app integrity nodes.	App integrity verification helps ensure your users are only using a supported app rather than a third-party or potentially malicious version.
Key attestation	***Android*** SafteyNet ***iOS*** None	***Android*** Uses hardware-backed key pairs with Key Attestation. ***iOS*** It can be added as part of the journey by using app integrity nodes to support key attestation.	Key attestation verifies that the private key is valid and correct, is not forged, and was not created in an insecure manner.
Complexity	Medium	Low	WebAuthn requires a bit more configuration, for example, creating and uploading the `assetlinks.json` and `apple-app-site-association` files. Device binding only requires the journey and the SDK built into your app.

Relevant authentication nodes and callbacks

The following table covers the authentication nodes and callbacks that AM provides for creating device binding journeys.

Node	Callback	Description
Device Binding node	DeviceBindingCallback	Registers a device to the user and optionally stores the public key and key ID in the user's profile
Device Binding Storage node	Non-interactive	Stores the public key and key ID in the user's profile if they were stored in node state
Device Signing Verifier node	DeviceSigningVerifierCallback	Verifies ownership of a device by requesting it signs a challenge and verifying the result

The SDKs support the default **Authentication Type** options provided by the authentication nodes. These options define how the user must authenticate on their device to gain access to the private keys stored on it:

`Biometric only`

Request that the client secures access to private keys with biometric security, such as a fingerprint.

`Biometric with PIN fallback`

Request that the client secures access to the private keys with biometric security, such as a fingerprint, but allow use of the ***device PIN*** if biometric is unavailable.

`Application PIN`

Request that the client secures access to the private keys with an ***application-specific PIN***.

On Android devices, the private keys used for binding and verification are stored in a keystore file protected by the application PIN specified by the user - it does not use hardware-backed encryption. However, this keystore file is encrypted using keys from the hardware-backed `AndroidKeyStore`.

> **Important**
>
> The ***application-specific PIN*** applies only to your app, and is not linked to the ***device PIN*** used to unlock the device.
> The application-specific PIN is stored only on the client device and is not sent to AM.
> If the user forgets their application-specific PIN, they must bind the device again.

None

The user does not need to authenticate to gain access to the private keys on their device.

The SDKs provide the UI to handle these application types by default. You can also override the default UI and provide your own implementations. Refer to Custom authentication UI.

Add device binding dependencies

To bind a device and perform signing verification, you must add the device binding module to your project.

Add Android dependencies

To add the device binding dependencies to your Android project:

1. In the **Project** tree view of your Android Studio project, open the `Gradle Scripts/build.gradle` file for the *module*.
2. In the `dependencies` section, add the required dependencies:

   Example `dependencies` section after editing:

```
dependencies {
    implementation 'org.forgerock:forgerock-auth:4.8.1'

    // Device binding core dependencies
    implementation 'com.nimbusds:nimbus-jose-jwt:9.23'
    implementation 'androidx.security:security-crypto:1.0.0'

    // BIOMETRIC_ONLY, BIOMETRIC_WITH_FALLBACK
    implementation 'androidx.biometric:biometric-ktx:1.2.0-alpha04'

    // APPLICATION_PIN
    implementation 'com.madgag.spongycastle:bcpkix-jdk15on:1.58.0.0'
}
```

Add iOS dependencies

You can use CocoaPods or the Swift Package Manager to add the device binding dependencies to your iOS project.

Add dependencies using CocoaPods

1. If you do not already have CocoaPods, install the latest version.
2. If you do not already have a Podfile, in a terminal window run the following command to create a new Podfile:

```
pod init
```

3. Add the following lines to your Podfile:

```
pod 'FRDeviceBinding' // Add-on for Device Binding feature
```

4. Run the following command to install pods:

```
pod install
```

Add dependencies using Swift Package Manager

1. With your project open in **Xcode**, select **File** > **Add Package Dependencies**.
2. In the search bar, enter the Ping SDK for iOS repository URL: `https://github.com/ForgeRock/forgerock-ios-sdk`.
3. Select the `forgerock-ios-sdk` package, and then click **Add Package**.

4. In the **Choose Package Products** dialog, ensure that the `FRDeviceBinding` library is added to your target project.
5. Click **Add Package**.
6. In your project, import the library:

```
// Import the library
import FRDeviceBinding
```

Handle device binding callbacks

To bind a device on receipt of a `DeviceBindingCallback`, use the `DeviceBindingCallback.bind()` function.

This binds the device to the account using the default implementation.

Examples

Android - Java

```
DeviceBindingCallback callback = node.getCallback(DeviceBindingCallback.class);
callback.setDeviceName("My Android Device");
callback.bind(this.getActivity(), new FRListener<Void>() {
    @Override
    public void onSuccess(Void result) {
        // Proceed to the next node
        node.next();
    }

    @Override
    public void onException(Exception e) {
        // Proceed to the next node
        node.next();
    }
});
```

Android - Kotlin

```
try {
    // Provide a friendly name for the device
    callback.setDeviceName("My Android Device")
    // Bind the device
    callback.bind(context)
    // Proceed to the next node
} catch (e: CancellationException) {
    // Ignore, due to configuration change
} catch (e: DeviceBindingException) {
    // Proceed to the next node
}
```

iOS - Swift

```
// Provide a friendly name for the device
callback.setDeviceName("My iOS Device")

// Bind the device
callback.bind() { result in
    switch result {
        case .success:
            // Proceed to the next node
        case .failure(let error):
            // Handle the error and proceed to the next node
    }
}
```

Important

Device Binding is not supported on iOS simulators.
You must use a physical device to test Device Binding on iOS.

Note

The examples above use the default user interface for authenticating users in order to create and securely store private keys. For information on providing your own UI for authenticating access to the private keys, refer to Implement custom UI.

Handle device signing verifier callbacks

To sign the challenge on receipt of a `DeviceSigningVerifierCallback`, use the `DeviceBindingCallback.sign()` function.

Examples

Android - Java

```
callback.sign(requireContext(), new FRListener<Void>() {
    @Override
    public void onSuccess(Void result) {
        // Proceed to the next node
    }

    @Override
    public void onException(Exception e) {
        // Proceed to the next node
    }
});
```

Android - Kotlin

```
try {
    callback.sign(context)
    // Proceed to the next node
} catch (e: CancellationException) {
    // Ignore, due to configuration change
} catch (e: DeviceBindingException) {
    // Map custom client errors:
    when (e.status) {
        is UnRegister -> {
            callback.setClientError("UnReg")
        }
        is UnAuthorize -> {
            callback.setClientError("UnAuth")
        }
    }
    // Proceed to the next node
}
```

iOS - Swift

```
callback.sign() { result in
    switch result {
        case .success:
            // Proceed to the next node
        case .failure(let error):
            // Handle the error and proceed to the next node
    }
}
```

> **Note**
>
> The examples above use the default user interface for authenticating users in order to access the private keys. For information on providing your own UI for authenticating access to the private keys, refer to Implement custom UI.

Add custom claims when signing

When signing a challenge on receipt of a `DeviceSigningVerifierCallback`, you can also add custom claims to the payload to provide additional context to the server.

A script in your authentication journey can access these claims and use them to implement additional functionality or logic.

The Device Signing Verifier node places the contents of the signed JWT in shared state in a variable named `DeviceSigningVerifierNode.JWT`.

Examples

Android - Java

```
Map<String, String> customClaims  = new HashMap<>() {{
    put("os", "value1");
}};

callback.sign(context, customClaims, new FRListener<Void>() {
    @Override
    public void onSuccess(Void result) {
        // Proceed to the next node
    }

    @Override
    public void onException(Exception e) {
        // Check for DeviceBindingErrorStatus.InvalidCustomClaims status
        // and fix invalid custom claims if needed
        // Proceed to the next node
    }
});
```

Android - Kotlin

```
try {
    callback.sign(context, mapOf("os" to "android"))
    // Proceed to the next node
} catch (e: CancellationException) {
    // Ignore, due to configuration change
} catch (e: DeviceBindingException) {
    // Map custom client errors:
    when (e.status) {
        is UnRegister -> {
            callback.setClientError("UnReg")
        }
        is UnAuthorize -> {
            callback.setClientError("UnAuth")
        }
        is DeviceBindingErrorStatus.InvalidCustomClaims -> {
          // Fix the invalid custom claims
        }
    }
    // Proceed to the next node
}
```

iOS - Swift

```
callback.sign(
  customClaims: [
    "platform": "iOS",
    "isCompanyPhone": true,
    "lastUpdated": Int(Date().timeIntervalSince1970)
  ]
) { result in
      switch result
      {
          case .success:
              // Proceed to the next node
          case .failure(let error):
              // Handle the error and proceed to the next node
              if error == .invalidCustomClaims {
                  // Fix the invalid custom claims
                  print(error.errorMessage)
                  return
              }
      }
  }
```

Unbind devices by deleting keys

Registered devices store a public key and key ID on the AM server, and the private key in either the Android KeyStore or the iOS Secure Enclave.

To completely unbind a device from a user, you must delete the keys from both the client device and the server.

The following table outlines scenarios where the client deletes the **local** keys:

Scenario	Android Device	iOS Device
User deletes the client application	Local key is deleted	Local key is ***NOT*** deleted. Reinstall an app with the same AppID and signature to gain access to the original keys. The device is still bound to the user.
User factory resets the client device	Local key is deleted	Local key is deleted
User restores a backup from the original device to a new device	Local keys are not exported to the cloud during backup and cannot be restored to another device. New device will require new keys. Keys remain on the original device.	Local keys are stored in Secure Enclave and are not exported to the cloud during backup and cannot be restored to another device. New device will require new keys. Keys remain on the original device.

Important

Removing keys from the client device manually does not remove the keys from the server.
Use the SDK to remove both sets of keys from within your application, or an Administrator can remove server keys by using the REST API.

To completely unbind a device from a user, use the SDK `delete` method to contact the AM server to delete the keys.

When the keys are successfully removed from the server, the SDK removes the private keys from the device.

Step 1. Retrieve a list of keys

Call the `FRUserKeys().loadAll()` method to obtain a list of keys that are stored on the device:

Android - Kotlin

```
val frUserKeys = FRUserKeys(context)
var keys: List<UserKey> = frUserKeys.loadAll()
```

iOS - Swift

```
let userKeys = FRUserKeys().loadAll()
```

Step 2. Delete the key from both the server and the device

Call the `FRUserKeys().delete(userKey, forceDelete)` method to delete a key from the server.

The parameters are as follows:

`userKey`

Which key to delete.

`forceDelete`

Whether to delete the local key if deleting the key from the server fails.

When set to `true`, the local key is deleted even if removal from the server was not successful.

Defaults to `false`, meaning the local key is not deleted if removal from the server fails.

Example:

Android - Kotlin

```
val frUserKeys = FRUserKeys(context)
frUserKeys.delete(userKey, false)
```

iOS - Swift

```
do {
    try FRUserKeys().delete(
        userKey: userKey,
        forceDelete: false
    )
}

catch {
    print("Failed to delete public key from server")
}
```

After deleting keys, the user needs to rebind the device for use in authentication journeys.

Implement custom UI

To ease implementation, the OS and Ping SDKs provide default user interfaces for authenticating to access private keys, and also for selecting the private key to use if there is more than one.

The user interface for authenticating to access the private keys uses the text strings returned by the callback. You can configure these strings in the configuration of the relevant nodes on the server, or you can override these values using the SDK for providing the prompts.

You can also implement your own user interface for requesting an application PIN, and key selection when there are multiple available.

Customize authentication prompts

Customize or localize the text prompts that appear when accessing the private keys.

Customize authentication UI

Learn how to implement your own user interface for accessing the private keys when requesting an application PIN.

Customize key selection UI

Discover how to implement a user interface for choosing between multiple available keys.

Custom authentication prompts

The default text strings used when prompting the user to authenticate to gain access to the private keys come from the callbacks.

You can override or localize these strings by using the `Prompt` object. You can then pass the customized object into the `deviceBindingCallback.bind` and `deviceSigningVerifierCallback.sign` calls.

Android - Kotlin

Binding:

```
val deviceBindingCallback = node.getCallback(DeviceBindingCallback::class.java)

deviceBindingCallback.bind(
    activity,
    prompt = Prompt("Custom Title", "Custom Subtitle", "Custom Description"),
    listener =
        object : FRListener<Void?> {
            override fun onSuccess(result: Void?) {
                node.next(activity, activity)
            }

            override fun onException(e: java.lang.Exception) {
                node.next(activity, activity)
            }
        }
)
```

Signing:

```
val deviceSigningVerifierCallback = node.getCallback(DeviceSigningVerifierCallback::class.java)

deviceSigningVerifierCallback.sign(
    activity,
    prompt = Prompt("Custom Title", "Custom Subtitle", "Custom Description"),
    listener =
        object : FRListener<Void?> {
            override fun onSuccess(result: Void?) {
                node.next(activity, activity)
            }

            override fun onException(e: java.lang.Exception) {
                node.next(activity, activity)
            }
        }
)
```

iOS

Binding:

```
if callback.type == "DeviceBindingCallback",
let deviceBindingCallback = callback as? DeviceBindingCallback {
    let customPrompt = Prompt (
        title: "Custom Title",
        subtitle: "Custom Subtitle",
        description: "Custom Description"
    )
    deviceBindingCallback.bind (prompt: customPrompt)
    { result in
        /// process the result
    }
    return
}
```

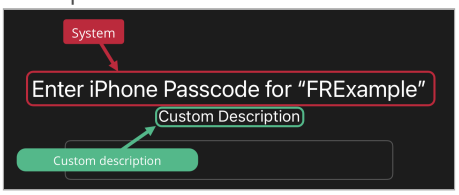
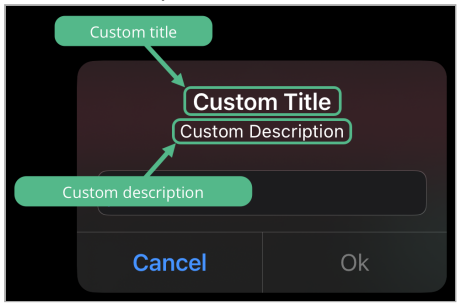
Signing:

```
if callback.type == "DeviceSigningVerifierCallback",
let deviceSigningVerifierCallback = callback as? DeviceSigningVerifierCallback {
    let customPrompt = Prompt (
        title: "Custom Title",
        subtitle: "Custom Subtitle",
        description: "Custom Description"
    )
    deviceSigningVerifierCallback.sign (prompt: customPrompt)
    { result in
        /// process the result
    }
    return
}
```

Apple iOS restrictions on custom prompts

On iOS devices, some of the prompts displayed to the user are system controlled and cannot be customized.

The following table outlines the situations where iOS uses your customized prompt:

Biometric Only	Biometric with allow fallback	Application PIN
FaceID-registered devices display no system-provided or custom text:  TouchID-registered devices show a system-provided title and the custom description text: 	When allow fallback is enabled, the biometric prompts match the biometric-only display. If authentication falls back to using the *device* PIN, then the device shows a system-provided title and the custom description text: 	When using an *application* PIN, the device shows both the custom title and custom description text: 

Custom authentication UI

When binding a device or verifying ownership of a device with signing, the user is asked to authorize access to their private keys.

For biometric-backed authentication such as touch or face ID, the UI is provided by the OS. When using `APPLICATION_PIN` as the authentication method you can customize the UI as required.

For example, the Ping SDK for Android uses the following UI when requesting an application PIN:

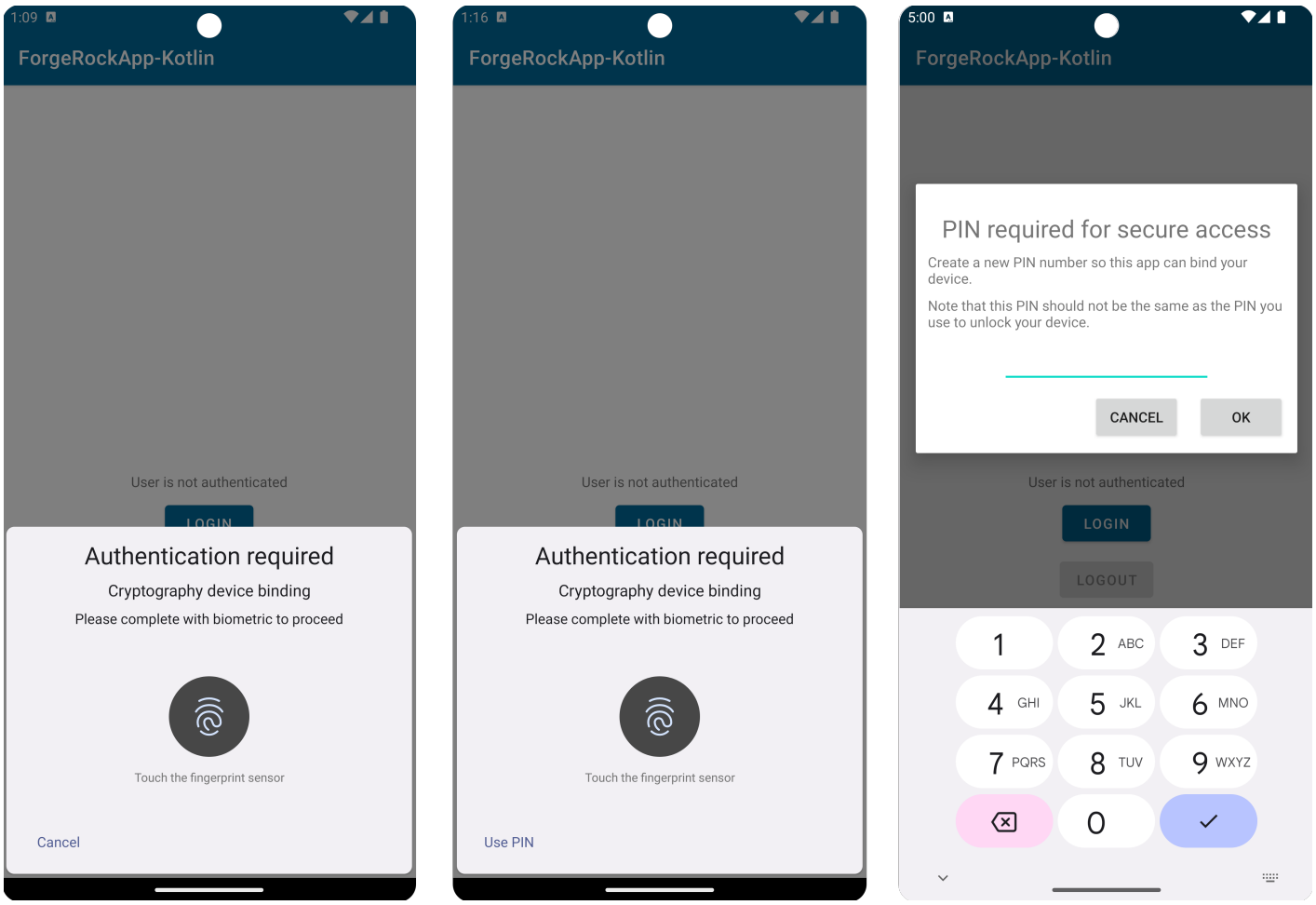


Figure 1. Android OS UI for `BIOMETRIC_ONLY`, `BIOMETRIC_ALLOW_FALLBACK`, *and a custom* `APPLICATION_PIN`

When providing your own application PIN UI, you can use the same mechanism for both binding and signing.

The following code shows how to implement a custom application PIN UI:

Android - Java

```
callback.bind(requireContext(), deviceBindingAuthenticationType -> {
    switch (deviceBindingAuthenticationType) {
        case APPLICATION_PIN: {
            return new CustomAppPinDeviceAuthenticator();
        }
        default:
            return callback.getDeviceAuthenticator(deviceBindingAuthenticationType);
    }
}, new FRListener<Void>() {
    @Override
    public void onSuccess(Void result) {
      // Proceed to the next node
    }

    @Override
    public void onException(Exception e) {
      // Proceed to the next node
    }
});
```

```
public class CustomAppPinDeviceAuthenticator extends ApplicationPinDeviceAuthenticator {

    public CustomAppPinDeviceAuthenticator() {
        super((prompt, fragmentActivity, $completion) -> {
            $completion.resumeWith("1234".toCharArray());
            return IntrinsicsKt.getCOROUTINE_SUSPENDED();
        });
    }
}
```

Android - Kotlin

```
class CustomPinCollector: PinCollector {
    override suspend fun collectPin(prompt: Prompt, fragmentActivity: FragmentActivity): CharArray {}
}

class CustomAppPinDeviceAuthenticator: ApplicationPinDeviceAuthenticator(CustomPinCollector())

callback.bind(context) {
    when (it) {
        // Implement your custom app PIN UI...
        APPLICATION_PIN -> CustomAppPinDeviceAuthenticator()
        else -> {
            callback.getDeviceAuthenticator(it)
        }
    }
}
```

iOS - Swift

```
callback.bind(deviceAuthenticator: { type in
    switch type {
        case .applicationPin:
            return ApplicationPinDeviceAuthenticator(pinCollector: CustomPinCollector())
        default:
            return callback.getDeviceAuthenticator(type: type)
    }
}, completion: { result in
    switch result {
        case .success:
            // Proceed to the next node
        case .failure(let error):
            // Handle the error and proceed to the next node
    }
})
```

```
class CustomPinCollector: PinCollector {
    func collectPin(prompt: Prompt, completion: @escaping (String?) -> Void) {
        // Implement your custom app PIN UI...
        completion("1234")
    }
}
```

Custom key selection UI

When verifying ownership of a device using signing, the user could be asked to select which private key to use if they have more than one on their device.

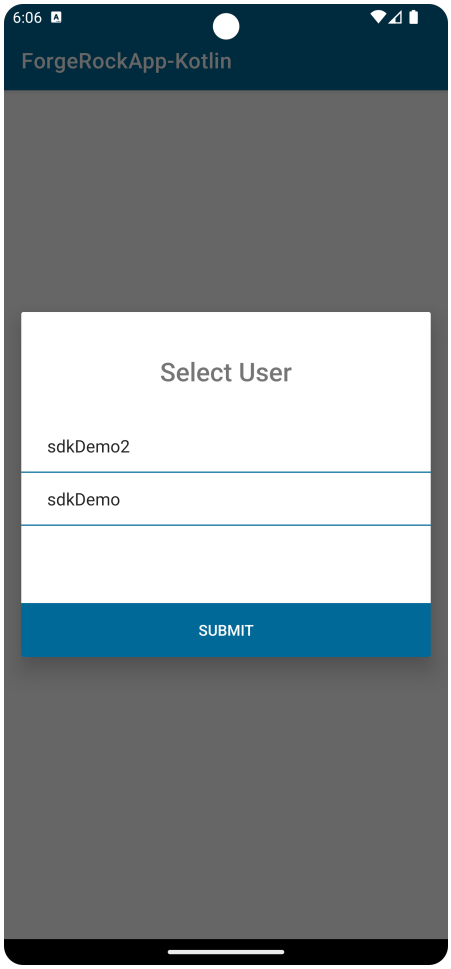


Figure 2. Default Android UI for selecting the private key

You can override the default key selection UI to implement your own.

Android - Java

```
callback.sign(requireContext(), new CustomUserKeySelector(),  new FRListener<Void>() {
   @Override
   public void onSuccess(Void result) {

   }

   @Override
   public void onException(Exception e) {

   }
});
```

```
// Custom user selector that always returns the most recently created key
public class CustomUserKeySelector implements UserKeySelector {

    @Nullable
    @Override
    public Object selectUserKey(@NonNull UserKeys userKeys, @NonNull FragmentActivity fragmentActivity,
@NonNull Continuation<? super UserKey> $completion) {
        $completion.resumeWith(userKeys.getItems().get(0));
        return IntrinsicsKt.getCOROUTINE_SUSPENDED();
    }
}
```

Android - Kotlin

```
callback.sign(context, CustomUserKeySelector())
```

```
// Custom user selector that always returns the most recently created key
class CustomUserKeySelector : UserKeySelector {
    override suspend fun selectUserKey(userKeys: UserKeys,
                                       fragmentActivity: FragmentActivity): UserKey {
        return userKeys.items[0]
    }
}
```

iOS - Swift

```
callback.sign(userKeySelector: CustomUserKeySelector()) { result in
    switch result {
        case .success:
            // Proceed to the next node
        case .failure(let error):
            // Handle the error and proceed to the next node
    }
}
```

```
// Custom user selector that always returns the most recently created key
class CustomUserKeySelector: UserKeySelector {
    func selectUserKey(userKeys: [UserKey], selectionCallback: @escaping UserKeySelectorCallback) {
        selectionCallback(userKeys.first)
    }
}
```

Error handling

If an error occurs when binding a device or signing a secret for verification, the SDK raises an exception. Check the `status` property of the exception for information about the problem.

The following table lists the possible values, and the outcomes these map to in the authentication nodes:

Description	Exception status	Mapped node outcome
The client device does not support device binding. For example, it does not provide biometric sensors, or the SDK cannot generate the required key pair. **Important** Device Binding is not supported on iOS simulators. You must use a physical device to test Device Binding on iOS.	`Unsupported`	Unsupported
Binding or signing did not complete before the timeout expired.	`Timeout`	Timeout
The user cancelled binding or signing before completion.	`Abort`	Abort

Description	Exception status	Mapped node outcome
The SDK could not locate an existing private key. Either the device has not yet been bound, or the private key was removed.	`UnRegister`	Unsupported
The user failed the authentication required to access the private key. For example, they used an unrecognized fingerprint, or the wrong application PIN.	`UnAuthorize`	Unsupported
An unknown, unexpected error occurred.	`Abort`	Abort

You can map exceptions to custom client error outcomes in the nodes. For example, the following code maps the `UnRegister` status to an outcome named `CustomUnReg` in the node:

Android - Kotlin

```
deviceBindingCallback.bind(activity, object : FRListener<Void> {
    override fun onSuccess(result: Void?) {
        node.next(activity, activity)
    }

    override fun onException(e: java.lang.Exception?) {
        // Custom Error
        if (e is DeviceBindingException) {
            if (e.status is UnRegister) {
                deviceBindingCallback.setClientError("CustomUnReg")
            }
        }
        node.next(activity, activity)
    }
})
```

iOS - Swift

```
// Bind the device
callback.bind() { result in
    switch result {
        case .success:
            // Proceed to the next node
        case .failure(let error):
            // Custom Error
            if error == DeviceBindingStatus.unRegister {
                callback.setClientError("CustomUnReg")
            }
    }
}
```

Important

You must add the name of the custom client error, for example `CustomUnReg`, to the **Client error outcomes** property in the node configuration:

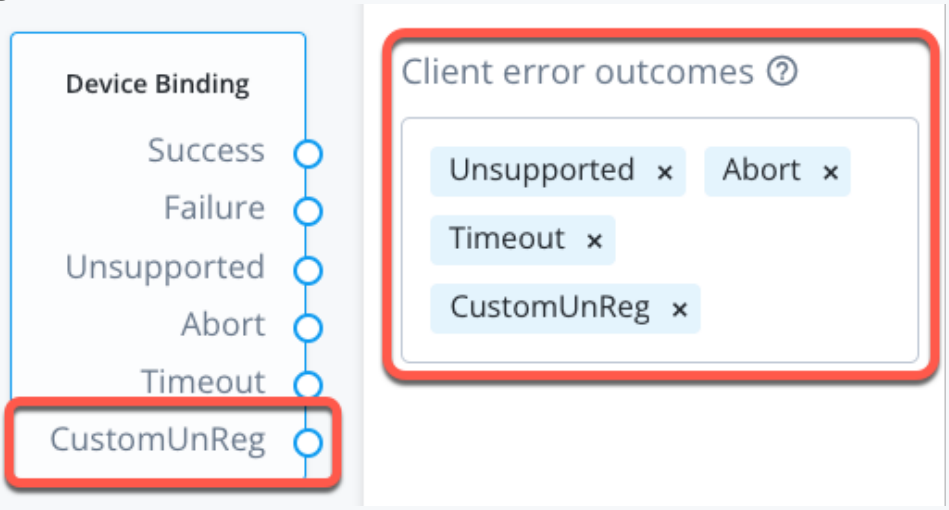


Figure 3. Custom client error outcome in the device binding node.

Device profile client configuration

Applies to:

- ✓ Ping SDK for Android
- ✓ Ping SDK for iOS
- ✓ Ping SDK for JavaScript

PingOne Advanced Identity Cloud and PingAM 7 and later include nodes that instruct your client applications to collect device profile information for decision-making in authentication journeys.

Device profile information can help you build authentication journeys that:

- Allow easy-pass for users on trusted devices.
- Introduce stronger authentication requirements, or deny access, for users on suspicious devices.

This section explains how to work with device profile information in your client app.

Prepare the server

In this step, you set up your server to perform device profiling.

Configure a journey to perform device profiling

To profile devices you must configure an authentication journey in your server.

The following table covers the authentication nodes and callbacks available for profiling devices in your authentication journeys.

Node	Callback	Description
Device Profile Collector node	DeviceProfileCallback	Gather location data and other metadata from the client device.
Device Match node	Non-interactive	Compare collected device data with that stored in the user's profile.
Device Profile Save node	Non-interactive	Persist collected device data to a user's profile.

In your server, log in as an administrator and create a new authentication journey similar to the following example:

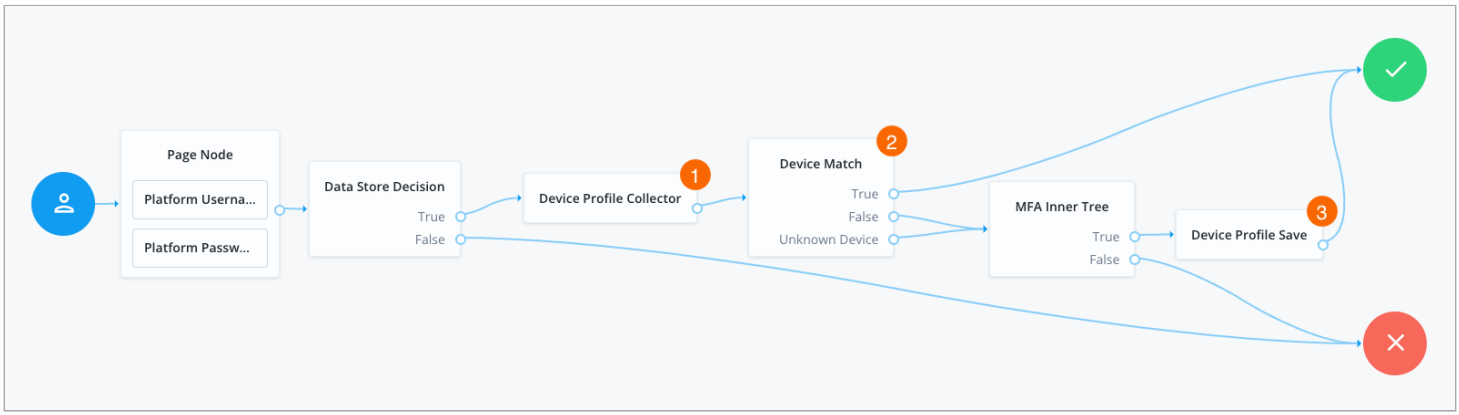


Figure 1. An example device profiling journey

- You must identify the user to be able to retrieve any stored device profiles they already have.

  In this example, we ask for their username and password, and verify the credentials against the data store.

- The Device Profile Collector node 1 instructs the SDK to collect and return metadata from the client device.
- The Device Match node 2 compares the collected metadata against any stored in the user's profile.

> **Tip**
>
> You can write scripts to customize how the node compares captured and stored metadata.
> For a complete sample script, as well as instructions for its use and a development toolkit, refer to https://github.com/ForgeRock/forgerock-device-match-script on GitHub.

- The example journey continues after comparing metadata, depending on the outcome:

`True`

The client device matches a device they have saved previously, and authentication succeeds without further interaction.

`False` *or* `Unknown Device`

The client device does not match, or they have not saved a device profile, so authentication requires additional steps, such as MFA verification.

- After successful MFA verification, the Device Profile Save node 3 offers to save the metadata to the user's profile, marking the client device as trusted for future journeys.

Customize device profile matching

The default Device Match node logic matches devices based on the number of differences in the captured attribute values. You can modify the threshold for difference by using the **Acceptable Variance** field in the node configuration.

The node also supports a custom matching script, where you can customize or write your own logic for matching device profiles. The script type must be **Decision node script for authentication trees** in self-managed AM servers or **Journey Decision Node** in Advanced Identity Cloud deployments.

Learn more about creating scripts:

- Script with JavaScript in Advanced Identity Cloud
- Scripting in AM

Download and modify a sample device match script

ForgeRock provides a sample repository that builds a device profile matching script you can download and customize for use in your environment.

The code is available on GitHub: https://github.com/ForgeRock/forgerock-device-match-script

Requirements

You require the following prerequisites to build this project:

- Node v13.10 or higher
- NPM v6 or higher

Customize the script

Without any modifications, the sample script performs matching on both metadata and location data.

You can modify `src/index.js` to prevent either of the matching types from running:

Perform both metadata and location matching

```
// Metadata and location matching
const [ metadataMatch, locationMatch ] = deviceMatcher();
const isMetadataMatching = metadataMatch(client.metadata, stored.metadata);
const isLocationMatching = locationMatch(client.location, stored.location);
```

Perform only metadata matching

```
const [ metadataMatch ] = deviceMatcher();
const isMetadataMatching = metadataMatch(client.metadata, stored.metadata);
```

Perform only location matching

```
const [ _, locationMatch ] = deviceMatcher();
const isLocationMatching = locationMatch(client.location, stored.location);
```

Tip

To modify the *logic* the script uses, edit the following files:

- Metadata matching: `src/metadata.js`
- Location matching: `src/location.js`

Configure the matching

The logic for both metadata and location matching can be configured according to your requirements.

Metadata matching

The metadata matching script uses recursive iteration to compare the metadata obtained from the client with the stored metadata. It is written with small to moderately large sized JavaScript objects in mind, and not optimized for very large or very deep structures.

When the recursion reaches a primitive value in the JSON, such as a string, number, or boolean, it does a comparison of the associated values. If there's a mismatch, it increments a counter.

You can modify this behavior as follows:

1. Configure the `maxUnmatchedAttrs` parameter to specify the maximum allowed number of allowed mismatches.
2. Alter the "weight" of specific attributes in the JSON, by using the `attrWeights` parameter.

   The weightings assigned work alongside the configured maximum number of mismatches. For example, if `maxUnmatchedAttrs` is set to 2, this could be exceeded by having three attributes with the *default* weight of 1 that do not match (n=1+1+1), or you could have a single property with a weight of 3 that does not match (n=3).

   Configuring the weighting means you can assign lower importance to certain profile properties, like display width or height, which might vary if the user changes displays from the prior authentication.

   If you're less concerned with the display properties because they can easily change, and more concerned with things that will remain unchanged, like the device's memory, you can assign greater weight to them as appropriate.

Example:

```
const config = {
  attrWeights: {
    // Custom weights for metadata attributes (object keys)
    deviceMemory: 3 // type `number`
    // ... as many attributes as you want
    // all attributes default to 1
  },
  maxUnmatchedAttrs: 2, // type `number`; default to 0 (exact match)
};
const [ metadataMatch ] = deviceMatcher(config);
const isMetadataMatching = metadataMatch(client.metadata, stored.metadata);
```

Location matching

The location matching script does not internally compare geolocation coordinates. It uses an external library called "geolib", which is well-built and very powerful.

The script uses the `getDistance()` function from the library, and wraps it with a comparison of the distance between two points to that of the maximum allowed radius.

Specify the maximum radius by using the `allowedRadius` parameter. All measurements are in meters. **Example**:

```
const config = {
  allowedRadius: 250, // type `number`; defaults to 100 (meters)
};
const [ _, locationMatch ] = deviceMatcher(config);
const isLocationMatching = locationMatch(client.location, stored.location);
```

Build the script

To build the sample device match script, follow these steps:

1. Download the device match script project from the GitHub repository:

   ```
   git clone https://github.com/ForgeRock/forgerock-device-match-script.git
   ```

2. In a terminal window, navigate to the root of the device match script project:

   ```
   cd forgerock-device-match-script
   ```

3. Run `npm` to download and install the required packages and modules:

   ```
   npm install
   ```

4. Build the device match script with `npm`:

   ```
   npm run build:widget
   ```

5. Copy and paste the contents of the `dist/script.js` file into your ForgeRock server.

   The script type must be **Decision node script for authentication trees** in self-managed AM servers or **Journey Decision Node** in Advanced Identity Cloud deployments.

6. In your Device Match node configuration, select **Use Custom Matching Script**, and in the **Custom Matching Script** field, select the script you created in the previous step.

7. Click **Save**.

For information on testing the script as well as some frequently asked questions, refer to the README.md in the repo.

Uniquely identifying devices

The Device Match node looks up a user's stored device profiles using a device identifier as a key. The Ping SDKs generate the device identifier as part of the device profile that it returns to the Device Profile Collector node as part of the JSON payload.

For example:

```
{
  "identifier": "d50cdb5ce8d055a3-86bd35e1b975a14d76b40940112c2380264c8efd",
  ....
}
```

Device identifier generation

This section covers the identifiers used on each platform, and how they are generated.

Android

On Android, a static device ID is not possible.

Static device ID

An ID that never changes, even during a factory reset or app re-installation.

Instance ID

An identifier for an instance of an application.

Instead of using a device ID, Android uses an instance ID. The instance ID provides a unique identifier for each instance of app, or app group.

Instance ID generation algorithm:

1. Generate a public/private key pair, and store the `KeyPair` in the `AndroidKeyStore` (Shared Storage).
2. Hash the public key with SHA1.
3. Encode with Base64.
4. Compile the `ANDROID_ID` with the hashed public key.

iOS

On iOS, `FRDeviceIdentifier` provides a unique identifier for each device that is defined in same Shared Keychain Access Group.

`FRDeviceIdentifier` provides a secure mechanism to uniquely generate, persist, and manage the identifier.

Device ID generation algorithm:

1. Generate an RSA key pair with key size of 2048.
2. Persist RSA keys in the Shared Keychain Service.
3. Hash the public key with SHA1.
4. Convert the hashed data into a hex string.

To view code that shows how iOS generates the device ID, see FRDeviceIdentifier.swift.

JavaScript

In JavaScript, the browser's crypto library generates the device ID. The ID is stored in the browser's `localStorage`.

To view code that shows how JavaScript generates the device ID, see index.ts in the `forgerock-javascript-sdk` repository.

When can identifiers change?

If the identifier changes, the Device Match node will be unable to match any stored device profiles.

If this happens, your journey must collect and store a new device profile, which contains the new identifier.

This section explains what can cause an identifier to change on each platform.

Android

In Android, the instance ID is deleted or changes if any of the following occurs:

- An app is restored on a new device.
- The user uninstalls and re-installs the app.
- The user clears app data.

iOS

On iOS, the device ID is stored in the Keychain. This means the ID persists when the app is removed.

However, the device ID is deleted or changes if any of the following occurs:

- The user wipes or factory resets the phone.
- The user migrates to a new phone.
- The keychain is programmatically deleted from the phone.
- The device ID is programmatically deleted from the Keychain.
- The keychain identifier in the `forgerock_keychain_access_group` configuration property changes.

JavaScript

In JavaScript, the device ID is deleted or changes if any of the following occurs:

- The browser window creates the device ID while in "private" or "incognito" mode. Closing the browser removes the ID.
- The browser removes the ID when cleaning up old data to make room for new data.
- The browser is uninstalled and reinstalled. The ID is removed.
- The user removes the device ID by clearing the browser data.

Set up device profiling in Android apps

This page shows how to detect the `DeviceProfileCallback`, how to collect the device profile, and how to send the profile to PingAM.

Important

PingAM includes collected device information in its audit logs by default.
To configure PingAM to filter out this information and ensure no personally identifiable information (PII) is written to the audit logs, refer to Prevent auditing of device data.

Handle a device profile callback

If an authentication journey uses the device profile node, the SDK returns `DeviceProfileCallback` to collect device attributes.

You use various SDK methods to handle the callback.

Use the default device profile callback

Call the `DeviceProfileCallback.execute()` method to collect the device profile:

```
callback.execute(context, new FRListener<Void>() {
@Override
public void onSuccess(Void result) {
// call next

}

@Override
public void onException(Exception e) {
}
});
```

Customize the device profile callback

1. Extend the callback that you want to override, providing two constructors that match the parent constructors.

2. Annotate the constructor with the `@Keep` annotation.

3. Override the default implementation:

```
public class MyCustomDeviceProfileCallback extends DeviceProfileCallback {

 public MyCustomDeviceProfileCallback() {
 }

 @Keep
 public MyCustomDeviceProfileCallback(JSONObject jsonObject, int index) {
     super(jsonObject, index);
 }

 @Override
 public void execute(Context context, FRListener<Void> listener) {
     super.execute(context, listener);
 }
}
```

4. Register the callback:

```
CallbackFactory.getInstance().register(MyCustomDeviceProfileCallback.class);
```

Manually collect device profile information

Instead of responding to a device callback, your app can get the device profile using default collectors.

You can also modify the default collectors. A set of collectors are predefined.

The `FRDevice` uses the default predefined collector to collect device profile:

The following code collects the device profile using the default collectors:

```
FRDeviceCollector.DEFAULT.collect(context, listener);
```

Default collectors

Collector name	Description
`FRDeviceCollector`	Main collector that includes other collectors and provides a collector version.
`BluetoothCollector`	Collect BLE support information of the device.
`BrowserCollector`	Collect browser information of the device; specifically, the `User-Agent`.
`CameraCollector`	Collect camera information of the device.
`DisplayCollector`	Collect display information of the device.

Collector name	Description
`HardwareCollector`	Collect hardware-related information, such as the number of CPUs, number of active CPUs, and so on.
`LocationCollector`	Collect location Information of the device.
`NetworkCollector`	Collect network information of the device.
`PlatformCollector`	Collect platform-related information, such as device jailbreak status, time zone/locale, OS version, device name, and device model.
`TelephonyCollector`	Collect telephony information of the device.

Sample device profile


```
{
  "identifier": "d50cdb5ce8d055a3-86bd35e1b975a14d76b40940112c2380264c8efd",
  "metadata": {
    "platform": {
      "platform": "Android",
      "version": 31,
      "device": "emulator64_x86_64_arm64",
      "deviceName": "sdk_gphone64_x86_64",
      "model": "sdk_gphone64_x86_64",
      "brand": "google",
      "locale": "en_US",
      "timeZone": "America/Vancouver",
      "jailBreakScore": 1
    },
    "hardware": {
      "hardware": "ranchu",
      "manufacturer": "Google",
      "storage": 5951,
      "memory": 1968,
      "cpu": 4,
      "display": {
        "width": 1080,
        "height": 2148,
        "orientation": 1
      },
      "camera": {
        "numberOfCameras": 2
      }
    },
    "browser": {
      "userAgent": "Mozilla/5.0 (Linux; Android 12; sdk_gphone64_x86_64 Build/SPB5.210812.003; wv) AppleWebKit/537.36
(KHTML, like Gecko) Version/4.0 Chrome/91.0.4472.114 Mobile Safari/537.36"
    },
    "bluetooth": {
      "supported": true
    },
    "network": {
      "connected": true
    },
    "telephony": {
      "networkCountryIso": "us",
      "carrierName": "T-Mobile"
    }
  },
  "location": {
    "latitude": 37.4219711,
    "longitude": -122.0849955
  },
  "lastSelectedDate": 1634068456582,
  "alias": "sdk_gphone64_x86_64"
}
```


Create a custom collector

1. Create a custom "DeviceCollector" class that implements the `DeviceCollector` interface:

```
public class MyCustomMetadataCollector implements DeviceCollector {

 private static final List<DeviceCollector> COLLECTORS = new ArrayList<>();

 static {
     //Pick from existing Collector or implement your own collector
     COLLECTORS.add(new PlatformCollector());
     COLLECTORS.add(new NetworkCollector());
     COLLECTORS.add(new TelephonyCollector());
 }

 @Override
 public String getName() {
     return "metadata";
 }

 @Override
 public void collect(Context context, FRListener<JSONObject> listener) {
     collect(context, listener, new JSONObject(), COLLECTORS);
 }
}
```

2. Use `FRDeviceCollectorBuilder` to add your custom `Collector`:

```
public class MyCustomDeviceProfileCallback extends DeviceProfileCallback {

 public MyCustomDeviceProfileCallback() {
 }

 @Keep
 public MyCustomDeviceProfileCallback(JSONObject jsonObject, int index) {
     super(jsonObject, index);
 }

 @Override
 public void execute(Context context, FRListener<Void> listener) {
     FRDeviceCollector.FRDeviceCollectorBuilder builder = FRDeviceCollector.builder();
     if (isMetadata()) {
         builder.collector(new MyCustomMetadataCollector());
     }
     if (isLocation()) {
         builder.collector(new LocationCollector());
     }

     builder.build().collect(context, new FRListener<JSONObject>() {
         @Override
         public void onSuccess(JSONObject result) {
             setValue(result.toString());
             Listener.onSuccess(listener, null);
         }

         @Override
         public void onException(Exception e) {
             Listener.onException(listener, null);
         }
     });
 }
}
```

Device profile attributes

By default, the Ping SDK collects the following device attributes:

Attribute	Value
`identifier`	A unique ID for the device. To learn more about the device identifier, refer to Uniquely identifying devices.
`location`	The location of a device (longitude and latitude values). This is configured in the node and requires user permissions.

Attribute	Value
`metadata`	Metadata for the device, including: **`platform`** The device OS, such as Android or iOS. **`deviceName`** The name of the device. **`locale`** The locale of the device, such as `en`. **`timeZone`** The time zone of the device, such as `Africa/Johannesburg`. **`brand`** The brand of the device, such as `Apple`. **`jailBreakScore`** A value between 0 and 1 that denotes the tampering level for a device.

Obtain user permission for the device location

Your app requires the user's authorization to access the device location.

For information about how to request the authorization, refer to Request location permissions.

Implement default jailbreak/rooted device detection

The `FRRootDetector` class is responsible for analyzing whether the device is tampered.

The class analyzes the device by using multiple device tamper detectors, and returns the highest score in the range between `0.0` to `1.0` from all the detectors.

> **Note**
>
> You can customize the `metadata.platform.jailBreakScore` with `Root Detector`.

Sample using default tamper detection:

```
// The DEFAULT Detector uses all available detectors in the SDK to determine if the device is rooted.
RootDetector rootDetector = FRRootDetector.DEFAULT;

// Check if device is rooted
double rootedScore = rootDetector.isRooted(context)

// Evaluate the result
if rootedScore == 0.0 {
    // Detectors score result with 0.0, likely device is not rooted
}
else if rootedScore <= 0.5 {
    // Some of the detectors returned a possible positive result that indicates the device might be rooted
}
else {
    // Most of the detectors returned a possible positive result that indicates the device is likely rooted
}
```

Customize jailbreak/rooted detection

The SDKs provide a set of industry-standard detectors that allow you to customize the detectors to use.

Sample custom tamper detection code:

```
// Using Builder to choose two detectors
RootDetector rootDetector = FRRootDetector.builder()
        .detector(new SuCommandDetector())
        .detector(new RootAppDetector())
        .build();

// Get result
double rootedScore = rootDetector.isRooted(context)
```

Implement custom detectors

You can provide your own detectors by implementing the `RootDetector` interface on Android. The interface represents the definition of an individual analyzer for detecting when the device is rooted or jailbroken.

Each detector determines whether the device is rooted or jailbroken. Each collector returns a result score as a `Double`, within the range of `0.0` to `1.0`.

Sample custom detector code:

```
// Add custom detector to RootDetector
RootDetector rootDetector = FRRootDetector.builder()
        .detectors(FRRootDetector.DEFAULT_DETECTORS)
        .detector(new RootDetector() {
            @Override
            public double isRooted(Context context) {
                return 0;
            }
        })
        .build();

// Get result
double rootedScore = rootDetector.isRooted(context);
```

More information

API reference: `FRDevice`

Set up device profiling in iOS apps

This page shows how to detect the `DeviceProfileCallback`, how to collect the device profile, and how to send the profile to PingAM.

> **Important**
>
> PingAM includes collected device information in its audit logs by default.
> To configure PingAM to filter out this information and ensure no personally identifiable information (PII) is written to the audit logs, refer to Prevent auditing of device data.

Handle a device profile callback

If an authentication journey uses the device profile node, the SDK returns `DeviceProfileCallback` to collect device attributes.

You use various SDK methods to handle the callback.

Use the default device profile callback

```
deviceProfileCallback.execute { _ in node.next
  { (user: FRUser?, node, error) in
   self.handleNode(user: user, node: node, error: error) //Handle the node
  }

}
```

Customize the device profile callback

1. The `DeviceCollector` protocol is a baseline class implementation protocol for the `FRDeviceCollector`.

   Create a new class inheriting the protocol and implement the protocol methods:

```
public class CustomCollector: DeviceCollector {
      public var name: String = "customCollector"
      public func collect(completion: @escaping DeviceCollectorCallback) {
        var result: [String: Any] = [:]
        result["customID"] = "MyCustomIDValue"
        completion(result)
      }
 }
```

2. Add the custom collector in the `DeviceProfileCallback` array of `ProfileCollectors`:

```
deviceProfileCallback.profileCollector.collectors.append(CustomCollector())
```

3. To collect the device profile and submit the node, use the `DeviceProfileCallback.execute()` method:

```
deviceProfileCallback.execute { _ in node.next
  { (user: FRUser?, node, error) in
    self.handleNode(user: user, node: node, error: error) // Handle the node
  }
}
```

Tip

If you want to remove any of the default profile collectors, access the profile collectors array of the `DeviceProfileCallback` node. The default list contains the following collectors:

- `PlatformCollector()`
- `HardwareCollector()`
- `BrowserCollector()`
- `TelephonyCollector()`
- `NetworkCollector()`

Manually collect device profile information

Instead of responding to a device callback, your app can get the device profile using default collectors.

The `FRAuth` SDK provides a device profile feature that enables you to identify and obtain details about the device.

The `FRDevice` class includes methods for getting information about a device.

Use the `FRDevice.currentDevice?.getProfile()` method to return device profile:

1. In your app, add the following code after the SDK initialization:

```
FRDevice.currentDevice?.getProfile(completion: { (deviceProfile) in
    print(deviceProfile)
})
```

2. When the above code is triggered, the app prints JSON text, including the device profile, in the console.

> **Note**
>
> The device `identifier` is not the same as Apple's device vendor identifier. As long as the Keychain Service configuration remains unchanged, the device identifier *should* remain the same for this device, even if you delete and reinstall the application.

Default collectors

Collector name	Description
`FRDeviceCollector`	Main collector that includes other collectors and provides a collector version.
`BluetoothCollector`	Collect BLE support information of the device.
`BrowserCollector`	Collect browser information from the device; specifically, the `User-Agent`.
`CameraCollector`	Collect camera information of the device.
`DisplayCollector`	Collect display information of the device.
`HardwareCollector`	Collect hardware-related information such as the number of CPUs, number of active CPUs, and so on.
`LocationCollector`	Collect location information of the device.
`NetworkCollector`	Collect network information of the device.
`PlatformCollector`	Collect platform-related information, such as device jailbreak status, time zone/locale, OS version, device name, and device model.
`TelephonyCollector`	Collect telephony information of the device.

Sample device profile

```
{
    "identifier": "uiCToe3h2kdfYLOzvHpaloxsKVE=",
    "version": "1.0",
    "platform": {
        "jailbreakScore": 0.0,
        "systemInfo": {
            "sysname": "Darwin",
            "release": "18.6.0",
            "version": "Darwin Kernel Version 18.6.0: Thu Apr 25 22:14:08 PDT 2019; root:xnu-4903.262.2~2/
RELEASE_ARM64_T8015",
            "nodename": "James-Go-iPhone-X",
            "machine": "iPhone10,6"
        },
        "timezone": "America/Vancouver",
        "model": "iPhone10,6",
        "version": "12.3.1",
        "locale": "en",
        "platform": "iOS",
        "device": "iPhone",
        "brand": "Apple",
        "deviceName": "James Go iPhone X"
    },
    "hardware": {
        "cpu": 6,
        "storage": 60975.11328125,
        "manufacturer": "Apple",
        "activeCPU": 6,
        "display": {
            "orientation": 1,
            "width": 375.0,
            "height": 812.0
        },
        "memory": 2823.0,
        "multitaskSupport": true,
        "camera": {
            "numberOfCameras": 4
        }
    },
    "bluetooth": {
        "supported": true
    },
    "browser": {
        "agent": "FRExample/1.0 (com.forgerock.frexample; iPhone; build:1; iOS 12.3.1) CFNetwork/978.0.7 Darwin/
18.6.0 FRAuth/1.0"
    },
    "telephony": {
        "mobileCountryCode": "302",
        "voipEnabled": true,
        "isoCountryCode": "ca",
        "mobileNetworkCode": "220",
        "carrierName": "TELUS"
    },
    "network": {
        "connected": true
    },
    "location": {
        "latitute": 37.7873589,
        "longitude": -122.408227
    }
}
```

Modify the default collectors

You can collect the device profile using default collectors. You can also modify the default collectors.

The following code collects the device profile using the default collectors:

```
FRDeviceCollector.shared.collect { (result) in
    // Result is returned as [String: Any]
}
```

The collectors array in the FRDeviceCollector class is a public property. You can add, remove, or change any of the collectors:

```
@objc
  public var collectors: [DeviceCollector]
```

Create a custom collector

```
class CustomCollector: DeviceCollector {
    var name: String = "custom"

    func collect(completion: @escaping DeviceCollectorCallback) {
        var result: [String: Any] = [:]

        // Perform logic to collect any device profile
        result["key"] = "value"

        completion(result)
    }
    }
}
```

Device profile attributes

By default, the Ping SDK collects the following device attributes:

Attribute	Value
`identifier`	A unique ID for the device. To learn more about the device identifier, refer to Uniquely identifying devices.
`location`	The location of a device (longitude and latitude values). This is configured in the node and requires user permissions.

Attribute	Value
`metadata`	Metadata for the device, including: **`platform`** The device OS, such as Android or iOS. **`deviceName`** The name of the device. **`locale`** The locale of the device, such as `en`. **`timeZone`** The time zone of the device, such as `Africa/Johannesburg`. **`brand`** The brand of the device, such as `Apple`. **`jailBreakScore`** A value between 0 and 1 that denotes the tampering level for a device.

Obtain user permission for the device location

Your app requires the user's authorization to access the device location.

For information about how to request the authorization, refer to Requesting Authorization for Location Services.

Implement default jailbreak/rooted device detection

The `FRJailbreakDetector` class is responsible for analyzing whether the device is tampered.

The class analyzes the device by using multiple device tamper detectors and returns the highest score in the range between `0.0` to `1.0` from all the detectors.

> **Warning**
>
> On iOS, a device simulator always returns `1.0`.

Sample using default tamper detection:

```
// Check if device is jailbroken
let jailbrokenScore = FRJailbreakDetector.shared.analyze()

// Evaluate the result
if jailbrokenScore == -1.0 {
    // no detectors found
}
else if jailbrokenScore == 0.0 {
    // Means that the detectors score result is 0.0
}
else {
    // Some detectors returned a possible positive result that indicates the device might be jailbroken
}
```

Customize jailbreak/rooted detection

The SDKs provide a set of well-known detectors. You can choose the detectors to use.

Sample custom tamper detection code:

```
// Remove everything and only add your selected detectors
FRJailbreakDetector.shared.detectors.removeAll()
FRJailbreakDetector.shared.detectors.append(BashDetector())
FRJailbreakDetector.shared.detectors.append(SSHDetector())

// Or loop through detectors and remove specific detector from default
for detector in FRJailbreakDetector.shared.detectors {
    // Remove specific detector if required
}

// Get result
let jailbrokenScore = FRJailbreakDetector.shared.analyze()
```

Implement custom detectors

Developers can implement their own detectors by extending the `JailbreakDetector` protocol on iOS. This protocol represents the definition of a individual analyzer for detecting if a device is rooted or jailbroken.

Each detector should analyze its logic to determine whether the device is rooted or jailbroken. Each detector returns the result score as a `Double`, within the range of `0.0` to `1.0`.

Sample custom detector code:

```
// Custom detector
class CustomDetector: JailbreakDetector {
    public func analyze() -Double {
        var result = 0.0
        // do the custom logic
        return result
    }
}

// Add custom detector to JailbreakDetector
FRJailbreakDetector.shared.detectors.append(CustomDetector())

// Get the result
let jailbrokenScore = FRJailbreakDetector.shared.analyze()
```

Known limitations

For ForgeRock SDK for iOS v2.2.0 and earlier, the iOS SDK has discrepancies in attribute names inside the JSON payload. ForgeRock SDK for iOS v3.0.0 and later correct the attribute names to align with PingAM's expectation and other platforms. The following attribute names were changed as of 3.0.0:

Attribute Names for 2.2.0 and earlier	Attribute Names for 3.0.0 and later
`timezone`	`timeZone`
`jailbreakScore`	`jailBreakScore`

> **Note**
>
> If your application still uses iOS SDK 2.2.0 or older, an adjustment in the `Custom Matching Script` is required in the `Device Match Node` to collect and analyze correct attribute values from the SDK.

More information

- `FRDevice` class

Set up device profiling in JavaScript apps

This page shows how to detect the `DeviceProfileCallback`, how to collect the device profile, and how to send the profile to your server.

> **Important**
>
> The server includes collected device information in its audit logs by default.
> To configure PingAM to filter out this information and ensure no personally identifiable information (PII) is written to the audit logs, refer to Prevent auditing of device data.

Handle a device profile callback

1. Get the callback and any messages to display to the user:

```
const deviceCollectorCb = step.getCallbackOfType('DeviceProfileCallback');
const message = deviceCollectorCb.getMessage();
```

2. From the callback, determine if the intention is to collect the device location, the device metadata, or both:

```
const isLocationRequired = deviceCollectorCb.isLocationRequired();
const isMetadataRequired = deviceCollectorCb.isMetadataRequired();
```

3. Create a new instance of the `FRDevice` class.

```
const device = new FRDevice();
```

> **Tip**
>
> To return specific device data, pass in configuration options to the `FRDevice` constructor. For example, you can return device platform, display, browser, and hardware data.

Manually collect device profile information

Instead of responding to a device callback, your app can get the device profile using default collectors.

1. Call the `getProfile()` method of the `FRDevice` class, passing the boolean values from the callback that indicate if device location and/or device metadata is required:

```
const profile = await device.getProfile({
 location: isLocationRequired,
 metadata: isMetadataRequired,
});
```

> **Note**
>
> To return specific device data, override the `FRDevice` class methods for getting the device metadata, device browser plugins, device name, device hardware, and so on.

2. Set the device profile information for the step:

```
step.getCallbackOfType('DeviceProfileCallback').setProfile(profile);
```

Default collectors

Collector name	Description
`fontNames`	Collect font information.
`displayProps`	Collect display information of the device.
`browserProps`	Collect browser information of the device.
`hardwareProps`	Collect hardware related information.
`platformProps`	Collect platform related information

Sample device profile


```
{
  "identifier": "714524572-2799534390-3707617532",
  "metadata": {
    "hardware": {
      "cpuClass": null,
      "deviceMemory": 8,
      "hardwareConcurrency": 16,
      "maxTouchPoints": 0,
      "oscpu": null,
      "display": {
        "width": 1080,
        "height": 1920,
        "pixelDepth": 24,
        "angle": 270
      }
    },
    "browser": {
      "userAgent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/
80.0.3987.163 Safari/537.36 Edg/80.0.361.111",
      "appName": "Netscape",
      "appCodeName": "Mozilla",
      "appVersion": "5.0 (Macintosh; Intel Mac OS X 10_14_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/
80.0.3987.163 Safari/537.36 Edg/80.0.361.111",
      "appMinorVersion": null,
      "buildID": null,
      "product": "Gecko",
      "productSub": "20030107",
      "vendor": "Google Inc.",
      "vendorSub": "",
      "browserLanguage": null,
      "plugins": "internal-pdf-viewer;mhjfbmdgcfjbbpaeojofohoefgiehjai;internal-nacl-plugin;"
    },
    "platform": {
      "deviceName": "Mac (Browser)"
      "language": "en-US",
      "platform": "MacIntel",
      "userLanguage": null,
      "systemLanguage": null,
      "fonts": "cursive;monospace;sans-serif;fantasy;Arial;Arial Black;Arial Narrow;Arial Rounded MT Bold;Comic Sans
MS;Courier;Courier New;Georgia;Impact;Papyrus;Tahoma;Trebuchet MS;Verdana;",
      "timezone": 300
    }
  },
  "location": {
    "latitude": 30.49843,
    "longitude": -97.639371
  }
}
```


Create a custom collector

1. Create an instance of `FRDevice` and customize the profile:

```
const device = new FRDevice({
  // Example customization:
  // Collect just the presence of Arial and Helvetica
  fontNames: [ 'Arial', 'Helvetica' ],
  // Do not collect any display properties
  displayProps: [],
  // Just collect User Agent
  browserProps: [ 'userAgent' ]
});
```

2. Replace the methods for further customization:

```
device.getHardwareMeta = function () {
  let obj;
  // Custom logic to collect hardware profile obj
  return obj;
}
```

3. Run the `getProfile()` method using the custom configuration and collector methods:

```
const profile = await device.getProfile({
  location: isLocationRequired,
  metadata: isMetadataRequired,
});
```

Device profile attributes

By default, the Ping SDK collects the following device attributes:

Attribute	Value
`identifier`	A unique ID for the device. To learn more about the device identifier, refer to Uniquely identifying devices.
`location`	The location of a device (longitude and latitude values). This is configured in the node and requires user permissions.

Attribute	Value
`metadata`	Metadata for the device, including: **`platform`** The device OS, such as Android or iOS. **`deviceName`** The name of the device. **`locale`** The locale of the device, such as `en`. **`timeZone`** The time zone of the device, such as `Africa/Johannesburg`. **`brand`** The brand of the device, such as `Apple`. **`jailBreakScore`** A value between `0.0` and `1.0` that denotes the tampering level for a device.

Obtain user permission for the device location

Your app requires the user's authorization to access the device location.

If the user denies location access, the SDK still collects the device profile data; however, the collected data will not include any location coordinates.

If the user provides the permission, the SDK collects the location coordinates.

Known limitations

- Location access requires user permissions. If the user denies permission or doesn't respond to the browser's request in time, geolocation coordinates are not collected and the profile is generated without it.
- Generating the profile can take time, especially when the location is requested. If this is a step all to itself, showing a spinner with message is recommended.
- To reduce authentication round-trips/latency, you can collect the device profile at the same time you collect other information.
- The device profile ID is generated if one doesn't exist, and stored in the browser's `localStorage`. If this is done within a browser's "private" or "incognito" mode, the ID does not persist once that window is closed. This creates a new ID, and therefore a new profile, when the profile is generated again.
- As JavaScript runs within a browser, it is more a browser profile than a device profile. A different browser on the same device produces a substantially different profile.
- Some profile attributes are more volatile than others. Plugging an external display into a laptop, for example, alters the generated profile.

More information

- API Reference: `FRDevice`

Prevent device data from appearing in audit logs

When using device profiling as part of your authentication journeys, the captured information is included in the audit logs by default.

You can configure PingAM to filter out this information to ensure no personally identifiable information (PII) is written to the audit logs.

The following JSON is a sample audit log entry, from the `authentication` topic:


```
{
  "_id": "c12f6ef2-262e-4263-b924-ed2236365d1a-1276",
  "timestamp": "2020-07-01T16:57:43.565Z",
  "eventName": "{am_name}-NODE-LOGIN-COMPLETED",
  "transactionId": "c12f6ef2-262e-4263-b924-ed2236365d1a-1274",
  "trackingIds": [
    "c12f6ef2-262e-4263-b924-ed2236365d1a-1259"
  ],
  "principal": [
    "bjensen"
  ],
  "entries": [
    {
      "info": {
        "nodeOutcome": "outcome",
        "treeName": "Test",
        "displayName": "Device Profile Collector",
        "nodeType": "DeviceProfileCollectorNode",
        "nodeId": "b9c49dc6-e557-4f98-bb05-504cd715e8d9",
        "authLevel": "0",
        "nodeExtraLogging": {
          "forgeRock.device.profile": {
            "identifier": "f505e455f33004c9-01ab094b8797382b1fab71cc8b3753ffb2bd774b",
            "version": "1.0",
            "metadata": {
              "platform": {
                "platform": "Android",
                ...
```


In the sample above, you can see the start of the device profile data, under the `nodeExtraLogging` entry.

You can filter this out of the audit logs, by using JSON pointer-like syntax:

1. Log in to the PingAM console as an administrator, for example `amAdmin`.
2. Navigate to **Configure** > **Global Services** > **Audit Logging**.
3. In the **Field blacklist filters** list, add an entry that starts with the relevant topic, and then a JSON-pointer like syntax to specify the data to exclude.

   For example, to exclude the device data from audit logs, enter:

`/authentication/entries/0/info/nodeExtraLogging/forgeRock.device.profile`

4. Save your changes.

   Device profile data will no longer appear in the `authentication` audit logs.

More information

- Set up audit logging

Set up social login

Applies to:

- ✓ Ping SDK for Android
- ✓ Ping SDK for iOS
- ✓ Ping SDK for JavaScript

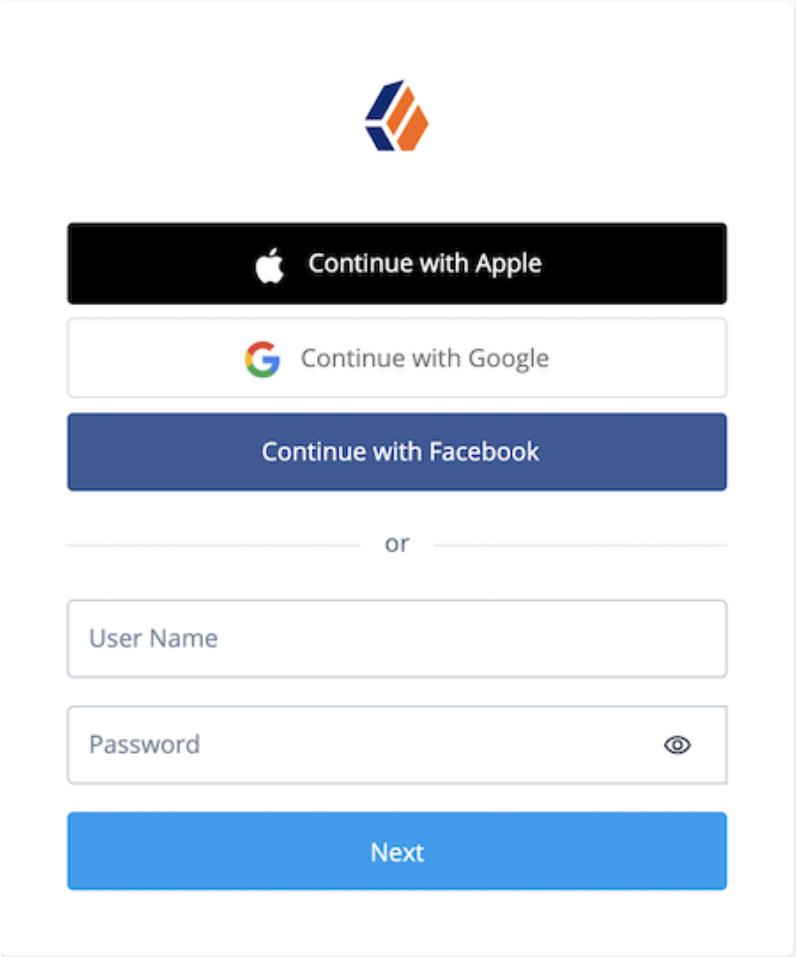
What is social login?

We provide the capability within authentication journeys/trees to support trusted Identity Providers (IdP), like Apple, Facebook, Google, and many others, for authentication and identity verification on behalf of ForgeRock. This is often referred to as *social login* or *social authentication*. These IdPs return the necessary user information to your server.

Depending on the device platform—Android, iOS, or web—the user is either redirected from the current web application, or the login page to the IdP's authorization server, or, if on a native mobile app, the user is directed to the IdP's authentication SDK, if available. Once on the IdP through a web page or the SDK, the user authenticates, and provides the necessary consent required for sharing the information with ForgeRock. When complete, the user is redirected back to your app or to the server to complete the authentication journey.

It's common to offer these social login options in addition to traditional authentication with username and password, but they can be used alone.

A screen capture of a login page with a common combination of methods:



Limitations

Before implementing social login, read Limitations.

Support matrices

Platform matrix

	Ping SDK for JavaScript	Ping SDK for Android	Ping SDK for iOS
SDK Version	3.0.0 and above	3.0.0 and above	3.0.0 and above
PingAM Version	6.5.2 and above	7.1.0 and above	7.1.0 and above
Platform Setup	Not required	Required	Required

Callback matrix

	Ping SDK for JavaScript	Ping SDK for Android	Ping SDK for iOS
`SelectIdPCallback`	Yes	Yes	Yes
`IdPCallback`	No	Yes	Yes
`RedirectCallback`	Yes	No	No

Supported providers matrix

The Ping SDK social login feature supports the following providers:

	Ping SDK for JavaScript	Ping SDK for Android	Ping SDK for iOS
Google	Yes	Yes	Yes
Facebook	Yes	Yes	Yes
Apple	Yes	Yes	Yes

Sign in with Apple is only supported on iOS 12+ devices.

Instructions

This how-to covers setting up an PingOne Advanced Identity Cloud tenant with the IdPs that the SDKs support: Apple, Facebook, and Google.

PingOne Advanced Identity Cloud supports additional IdPs.

Configure social login identity providers

In this section, you set your identity providers (IdP) to work with your apps through PingOne Advanced Identity Cloud.

Apple

Configure a Sign in with Apple client.

Facebook

Configure Facebook for use as a social identity provider.

Google

Configure Google for use as a social identity provider.

Create an Apple client

Sign up for an Apple developer account

You must enroll in the Apple Developer program.

> **Note**
>
> *Apple Developer Enterprise Program* accounts are not able to configure Sign in with Apple.

Set up application redirection

After Apple processes the initial authorization request and the user is successfully authenticated, Sign in with Apple sends an HTTP POST request to PingOne Advanced Identity Cloud or PingAM containing the authorization results.

For a web application (SPA) or an Android device, the POST request is sent to a dynamically created endpoint, specified in the Apple Sign In configuration as the redirect URL.

The redirect URL

To complete Apple client set up, you need the full redirect URL. This URL is not made available until you fully set up the provider in PingAM. If you have already set up your Apple provider, the redirect URL resembles the following:

```
https://<tenant-env-fqdn>/am/oauth2/<realm>/client/form_post/<secondary-configuration-name>
```

Set up Apple sign in

Create an app ID

1. Log in to your Apple developer account.
2. In the **Program resources** category, under **Certificates, Identifiers & Profiles**, click **Identifiers**.
3. Click the plus button (+) next to the **Identifiers** header.
4. Select **App IDs**, and click **Continue**.
5. Select **App type**, and click **Continue**.
6. Type a description of your app, and provide a `Bundle ID` using `reverse-domain name style`.

   For example `com.forgerock.ios.sdk.example`.
7. Select `Sign in with Apple`, and click **Continue**.
8. Review your entry, and click **Register**.

Create a service ID

1. On the **Identifiers** page, click the plus button (+) next to the **Identifier** header.
2. Select **Service IDs**, and click **Continue**.
3. Enter a description of your service.
4. Enter an `Identifier` that is similar to your app ID.

   For example, `<app-id>.service`.
5. Click **Continue**.
6. Review your entry, and click **Register**.

Configure the Apple sign in service

1. On the **Identifiers** page, click the dropdown next to the magnifying glass icon, and then select **Services IDs**.
2. Select the service ID you created.
3. Next to `Sign in with Apple`, click **Configure**.
4. Click the plus button next to the **Website URLs** header.
5. In **Domains and Subdomains**:
   - For JavaScript apps, enter the domains that host your app.

     For example, `sdkapp.example.org`

     > **Tip**
     >
     > During testing, do not use the `example.com` domain to host your application. Apple treats this domain differently than other domains, which can cause unexpected issues.
     > Using `example.org` or any other domain does not present these same difficulties.

   - For native Android and iOS apps, enter the domain of your PingOne Advanced Identity Cloud or PingAM instance.

     For example, `openam-forgerock-sdks.forgeblocks.com`
6. In **Return URLs**, enter the URL that Apple redirects users to after authentication.

   Users must be redirected back to PingOne Advanced Identity Cloud or PingAM to continue their authentication journey.

   The URL to use is dynamically created by PingOne Advanced Identity Cloud or PingAM when you configure identity providers, and uses the following syntax:

   Advanced Identity Cloud

   ```
   https://<tenant-env-fqdn>/am/oauth2/<realm>/client/form_post/<secondary-configuration-name>
   ```

   PingAM

   ```
   https://<am-fqdn>/openam/oauth2/client/form_post/<secondary-configuration-name>
   ```
7. Click **Next**.
8. Review, and click **Done**.

Create a key

Store your key in a safe location. You cannot download keys more than once.

1. On the developer account page, in the left navigation panel, click **Keys**.
2. Click the plus button next to the **Keys** header.
3. Enter your key name, and select **Sign in with Apple**.

4. Click **Configure**, select your primary app ID, and click **Save**.
5. Click **Continue**.
6. Review, and click **Register**.

Generate a client secret

The client secret for Apple sign is a JSON Web token (JWT). The JWT is more complex than a simple string. A common way of generating the JWT is to use the jwt/ruby-jwt library.

Before you create the JWT, you need to understand certain requirements. To learn about these requirements, see Apple's documentation about generating and validating tokens.

Configure the client ID

- For **Native iOS**: The `client_id` should be the `AppID` (bundle identifier) from the Apple Development portal.
- For **Web** or **Android**: The `client_id` should be the `ServiceID` from the Apple Development portal.

Example signing script:

```
require "jwt"

key_file = [Key file name]
team_id = [Team ID]
client_id = [AppID or Service ID]
key_id = [Key ID]
validity_period = 180 # In days. Max 180 (6 months) according to Apple docs.

private_key = OpenSSL::PKey::EC.new IO.read key_file

token = JWT.encode(
    {
        iss: team_id,
        iat: Time.now.to_i,
        exp: Time.now.to_i + 86400 * validity_period,
        aud: "https://appleid.apple.com",
        sub: client_id
    },
    private_key,
    "ES256",
    header_fields=
    {
        kid: key_id
    }
)
puts token
```

Create a Facebook client

To use Facebook as an Identity Provider, visit the Facebook for Developers page, and follow these steps:

1. Click the **Create App** button.
2. Select **Consumer** for app type, and click **Next**.

3. Enter your app's display name and contact email.
4. Click the **Create app** button.
5. On the **Add products to your app** page, under **Facebook Login**, click **Set up**.
6. In the left navigation panel, click **Settings** > **Basic**.
7. Take note of the **App ID** and **App secret** values.

Generate a key hash

The default password for Android Studio is `android`.

1. To generate a key hash value, in a terminal window, enter the following command:

   ```
   keytool -exportcert -alias androiddebugkey -keystore ~/.android/debug.keystore | openssl sha1 -binary |
   openssl base64
   ```

2. Note the key hash value for later use.

Configure an Android app

1. On the developer apps page, double-click an Android app.
2. In the left navigation panel, select **Settings** > **Basic**.
3. At the bottom of the page, click **Add platform**, select **Android**, and click **Next**.
4. In the **Select Android Store** dialog, select a store.

   For example, Google Play.
5. Click **Next**.
6. Scroll down to the Android section.
7. In the **Key hashes** field, enter the key hash value you generated earlier.
8. In the **Package Names** field, enter your app's **Google Play Package Name**.

   The name is often a reverse domain name, such as `com.example.app`.
9. In the **Class Name** field, enter your app's class name.
10. Click **Save changes**.

Configure an iOS app

1. On the developer apps page, select iOS.
2. Click **Next**.
3. Enter your **Bundle ID**.

   The name is often a reverse domain name, such as `com.example.app`.
4. Click **Save changes**.

5. In the left navigation panel, under **Facebook Login**, select **Quickstart**.
6. Click **iOS**.
7. Read the information, and select your package manager.
8. Click **Next**.
9. Enter your **Bundle ID**.
10. Click **Save**.
11. Click **Continue**.
12. Select your single sign-on settings.
13. When you get to **Configure Your info.plist**, configure your `info.plist` file with the XML snippet that contains data for your app.

Create a Google client

To use Google as an IdP, visit Google's API Dashboard, and follow these steps:

1. In the left navigation, click **Credentials**.
2. Click **CREATE CREDENTIALS** > **OAuth client ID**.

For an Android app

1. Select `Android` as the value for **Application Type**.
2. In the **Name** field, type a name for this application.
3. Enter the package name from the `AndroidManifest.xml` file.
4. Enter the SHA-1 certificate fingerprint.

   Use the following command to get the fingerprint:

   ```
   keytool -keystore path-to-debug-or-production-keystore -list -v
   ```

5. Click **Create**.

For an iOS app

1. Select `iOS` as the value for **Application Type**.
2. In the **Name** field, type a name for this application.
3. Enter the bundle id as listed in the app's `Info.plist` file.
4. If the app is listed in the Apple App Store, enter the Apple ID of the app.
5. Enter the Team ID that Apple assigned to your team.
6. Click **Create**.

For a JavaScript app

1. Select `Web application` as the value for **Application Type**.
2. In the **Name** field, type a name for this application.
3. Under **Authorized JavaScript Origins**, add the origins of the apps that use Google as an IdP.

   Origins include scheme, domain, and port.
4. Under **Authorized redirect URIs**, add the full redirect URLs of your apps that handle the redirection from Google after user login.
5. Click **Create**.

Native Android social authentication

To enable native Android social authentication, you must create two OAuth 2.0 clients in the Google API console:

1. Create an OAuth 2.0 client for the Android application.

   See the step for an Android app above.
2. Create an OAuth 2.0 client for PingAM to communicate with the Google APIs.

   See the step for a JavaScript app above.

Set up PingOne Advanced Identity Cloud for social login

This page explains how to configure your PingOne Advanced Identity Cloud tenant to work with your social IdP.

Enable IdPs

1. To access the configuration, log in to the PingOne Advanced Identity Cloud tenant as an administrator.
2. Click **Native Consoles**.
3. Click **Access Management**.
4. Click **Services**.
5. Click **Social Identity Providers Service**, or create it by clicking **Add a Service**.

   Once in the service, ensure it is enabled.
6. To manage your IdPs, click the **Secondary Configurations** tab.

Setting up your configuration for providers is mostly the same, but there are a few differences with Apple that are described below.

You will likely need a configuration for each platform that you develop. Use a naming scheme that's easy to remember, such as `google_web`, `google_ios`, `google_android`.

Configure Google and Facebook

1. Under **Secondary Configuration**, click **Add a Secondary Configuration**, and choose the provider to configure.

   The following fields are required:

   #### *Client ID*

   This is the ID for the client registered with the IdP. For native Android, set up the client with the second OAuth set of credentials generated using the web configuration. For details, see Set up social login in Android apps.

   #### *Client Secret*

   This is the secret for the client registered with the IdP.

   #### *Facebook*

   Facebook provides a client secret.

   #### *Android*

   You must set up the secret with the second OAuth set of credentials generated using the Web configuration. For details, see Set up social login in Android apps.

   #### *iOS*

   The Google Credentials do not provide a client secret.

   #### *Redirect URL*

   The redirect URL/URI for your application after the user authenticates with the provider.

   #### *Scope Delimiter*

   " " (a literal space character)

2. For native iOS and Android apps, set **Enable Native Nonce** to `OFF`.

   The option is `ON` by default.

3. Click **Create**.

4. Ensure that the configuration is enabled, and the `Transform Script` is set to `Google Profile Normalization`, or `Facebook Profile Normalization`.

Configure Apple

1. Under **Secondary Configuration**, click **Add a Secondary Configuration**, and choose the Apple Provider Configuration.

   The following fields are required:

   #### *Client ID*

   For native iOS, use the app ID created in the Apple developer page.

For Android or Web, use the service ID.

Client Secret

The JWT client secret you generated.

Redirect URL

Native iOS

Use the `<application-uri>` of your app.

The `<application-uri>` can vary as it does not impact the flow.

Android or Web

For PingAM, use `<forgerock-url>/am/oauth2/client/form_post/<configuration-name>`. For example, the configuration name here is `apple_web`. You can also add the redirect URI from the OAuth2.0 client settings.

For IDC, use `<forgerock-url>/am/oauth2/<realmName>/client/form_post/<configuration-name>`.

(Optional) Redirect after form post URL

This is only needed for configurations that use an PingOne Advanced Identity Cloud or PingAM endpoint for handling the POST redirection from Apple. The value of this field is the URL/URI of the Android or web app handling the remainder of the login flow.

Scope Delimiter

" " (a literal space character)

OAuth Scopes

Include the value `email` in this field to include the user's Apple ID email in the JWT response and the identity token.

If you add the email scope after a user agrees to provide scopes, the JWT response and the identity token will not include the user's Apple ID email.

> **Tip**
>
> To troubleshoot when the JWT response does not include the user's Apple ID email.
>
> 1. Sign in with your Apple ID used for authentication at the Apple ID page.
> 2. Under **Security** > **Sign in with Apple**, click **Manage apps & websites** > **Apps & websites using Apple ID**.
> 3. Select the application you are developing using Sign in with Apple.
> 4. Click **Stop using Apple ID**.
> 5. Attempt to re-authenticate in your app.
>
> For more information, read Sign in with Apple - Updated Scope Not Reflected in JWT Claims.

Well Known Endpoint

`https://appleid.apple.com/.well-known/openid-configuration`

Issuer

`https://appleid.apple.com`

2. Click **Create** to save your configuration.
3. Add these values to complete the configuration:

Response Mode

Native iOS

Use `Default`.

Android or Web

Use `FORM_POST`.

Transform Script

Ensure `Apple Profile Normalization` is selected.

Create your authentication journey

After configuring the provider services, the next step is to build your authentication journey. There are nearly unlimited number of ways to compose authentication journeys. This page covers the basics.

We have two different *sets* of authentication nodes that provide social login capabilities. This page focuses on the node sets supported by the Ping SDKs:

The Select Identity Provider node

The Select Identity Provider node has the following important settings.

Include local authentication

When enabled, this includes an additional "provider" in the `SelectIdPCallback` called `localAuthentication`. This also enables a different outcome of the node that you can connect to a local authentication path. The node will now have two outcomes: `Social Authentication`, and `Local Authentication`.

Filter enabled providers

This setting further reduces the number of providers passed to your application. This is often used when you have providers configured for different platforms, and you only want the platform specific providers sent to the application.

For more information, see the Select Identity Provider node reference.

Social Provider Handler node

The Social Provider Handler node has the following important settings:

Transformation Script

Set `Normalized Profile to Managed User`.

Client Type

If the browser is used for IdP authentication (always the case for JavaScript), `BROWSER` is the correct value. If you are using the IdP's SDK with native mobile for authentication, then `NATIVE` is the correct choice.

For more information, see the Social Provider Handler node reference.

Create users for a specific realm

To create users from a specific realm, link the Create node and the Patch node to the IDM users and their realm.

For example, for users in the alpha realm, set the Create node and the Patch node as follows:

Create Object node

Set **Identity Resource** to `managed/alpha_user`.

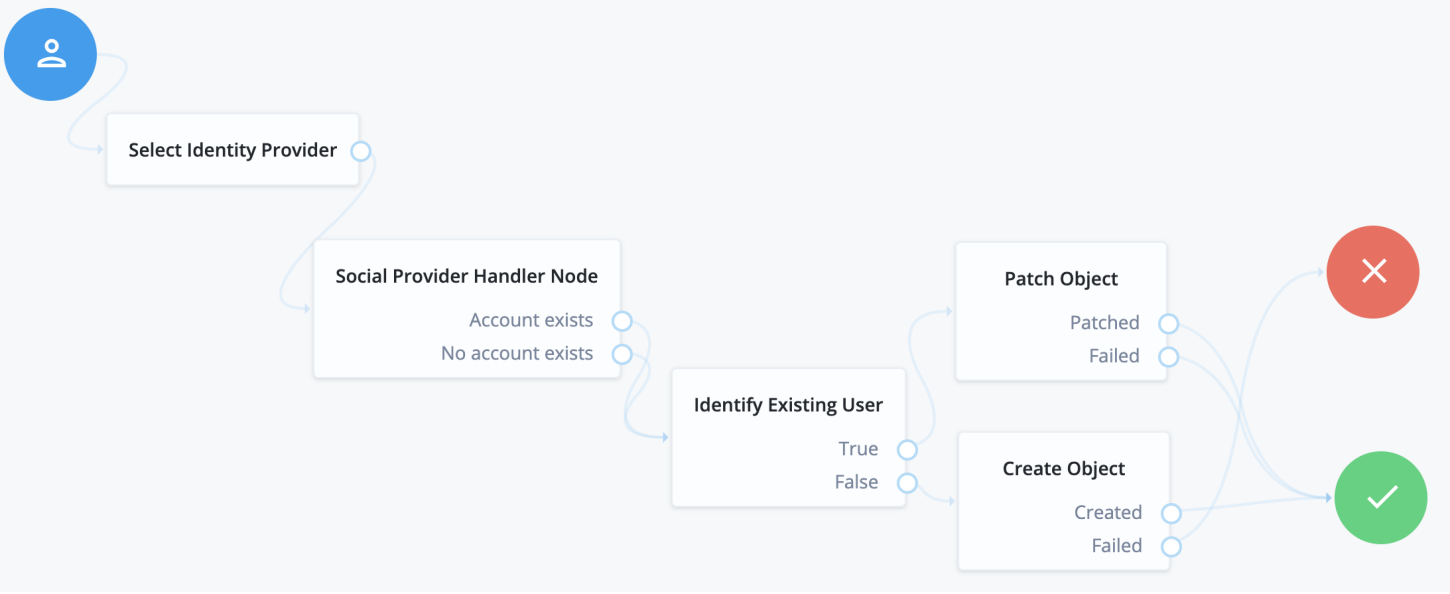
Patch Object node

Set **Identity Resource** to `managed/alpha_user`.

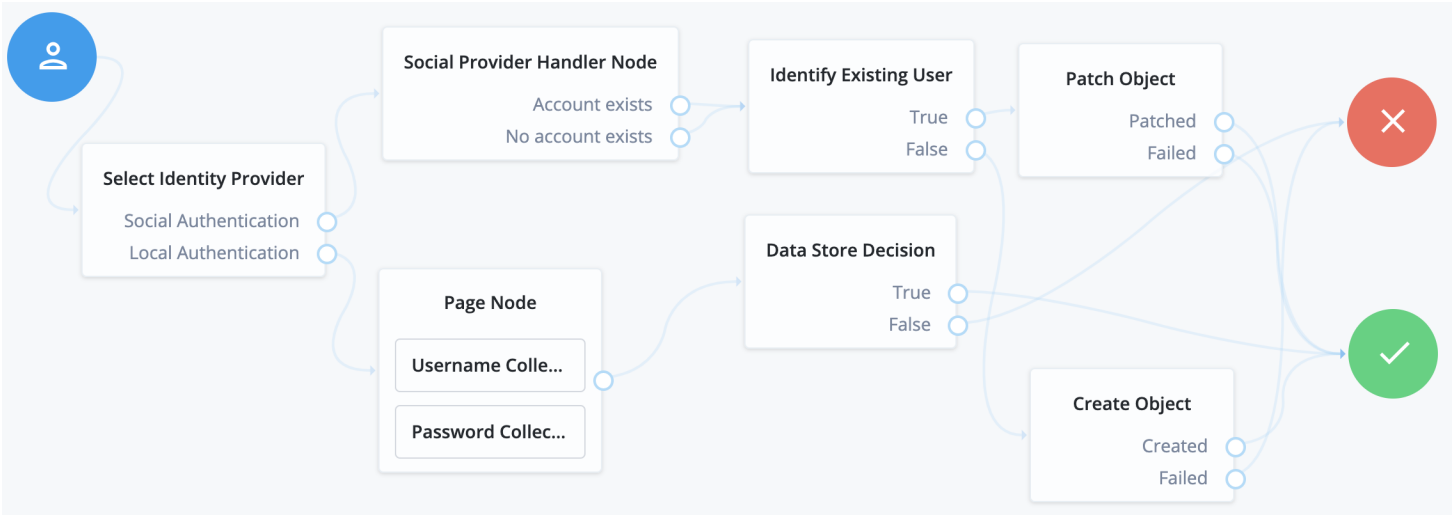
For more information, see the Create Object node, and Patch Object node references.

A simple social authentication journey

Start with the simplest journey. A user is presented with a choice of providers, the choice is made, and the user is taken to the IdP to authenticate. Upon the user's return, the user management capabilities read the identity information from the provider to verify, allow, or deny access to the system.

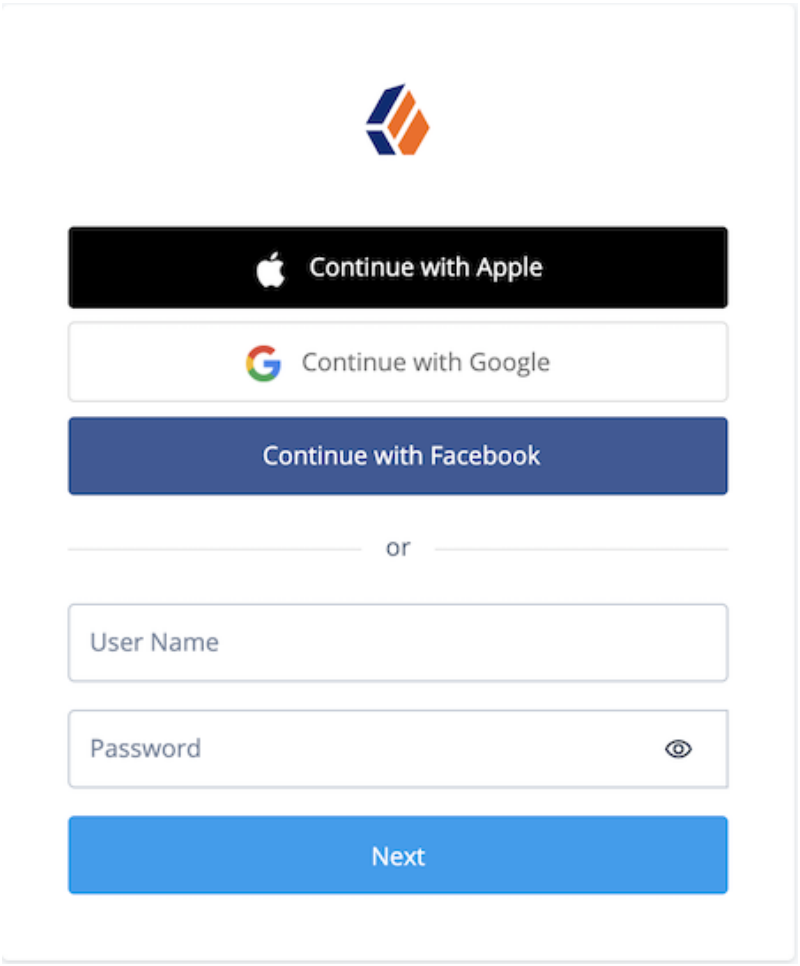


To add a choice of local authentication, enable the feature in the Select Identity Provider node, mentioned above, to create an alternative path for local credentials.



Social authentication and local username-password journey

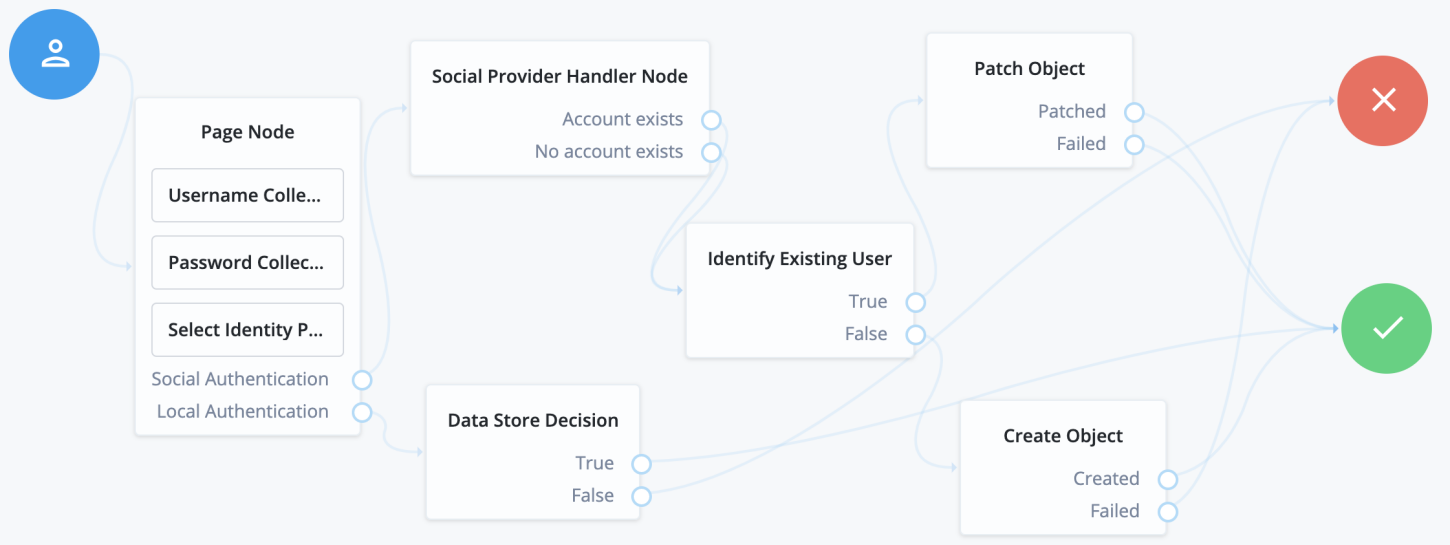
This is one of the more common authentication journeys seen across the web. This is a choice between using a third-party identity provider, or using first-party credentials with username and password directly. The UI for this combination looks like this:



To compose this type of journey, use a Page node to combine the following nodes:

- Password Collector/Platform Password

• Select Identity Provider (enable `Include local authentication`)



Set up social login in Android apps

This page shows how to use the Ping SDK for Android with authentication journeys that provide social login and registration.

The first callback your app encounters is the `SelectIdPCallback`, which lets the user choose their IdP. You use the `getProviders()` method to display the available providers, and `setValue` when the user makes a choice:

```
List<SelectIdPCallback.IdPValue> providers = callback.getProviders();
callback.setValue(chosenProvider);
```

The next callback is the `IdPCallback`. You call the `signIn()` method on the `IdPCallback` class:

```
callback.signIn(null, new FRListener<Void>() {
  @Override
  public void onSuccess(Void result) {
    //proceed to next node
    node.next();
  }

  @Override
  public void onException(Exception e) {
    //handle error
  }
});
```

This method directs the user to authenticate with the IdP. When the user authenticates with that provider, the result is automatically added to the `IdPCallback` with the following methods:

```
idPCallback.setTokenType(String tokenType)
idPCallback.setToken(String token)
```

Note

In order to override the automatic provider detection, and identify the returned provider manually, you must check the `IdPClient` provider value in the `IdPCallback` returned:

```
IdPHandler idPHandler = null;
  switch (callback.getProvider()) {
    case "facebook-android":
      idPHandler = new FacebookSignInHandler();
      break;
    case "google-android":
      idPHandler = new GoogleSignInHandler();
      break;
    case "apple-android":
      idPHandler = new AppleSignInHandler();
      break;
    default:
      //error
  }

//  If the handler has been found and initialised, call the following to perform login
callback.signIn(idPHandler, new FRListener<Void>() {
  ...
  //  Social Login flow is completed
}
```

SDK configuration

Google

For Google Sign-In, add the following dependency to the `build.gradle` file:

```
implementation 'com.google.android.gms:play-services-auth:20.5.0'
```

Facebook

For Facebook Login:

1. Add the Facebook dependency to the `build.gradle` file:

   ```
   implementation 'com.facebook.android:facebook-login:16.0.0'
   ```

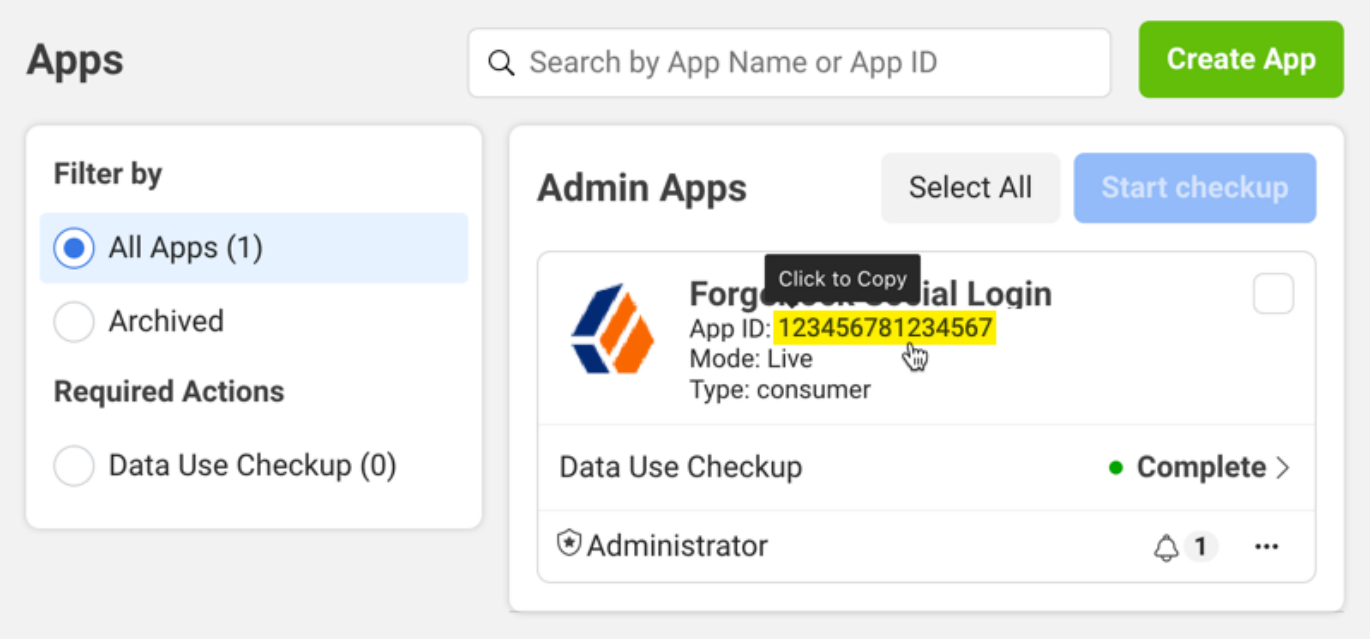
2. Add the following to the `AndroidManifest.xml` file:

```
<meta-data android:name="com.facebook.sdk.ApplicationId"
    android:value="@string/facebook_app_id"/>

<meta-data android:name="com.facebook.sdk.ClientToken"
    android:value="@string/facebook_client_token"/>

<activity android:name="com.facebook.FacebookActivity"
    android:configChanges=
    "keyboard|keyboardHidden|screenLayout|screenSize|orientation"
    android:label="@string/app_name" />
<activity
    android:name="com.facebook.CustomTabActivity"
    android:exported="true">
    <intent-filter>
      <action android:name="android.intent.action.VIEW" />
      <category android:name="android.intent.category.DEFAULT" />
      <category android:name="android.intent.category.BROWSABLE" />
      <data android:scheme="@string/fb_login_protocol_scheme" />
    </intent-filter>
</activity>
```

3. Add the following attributes to the `string.xml` file:

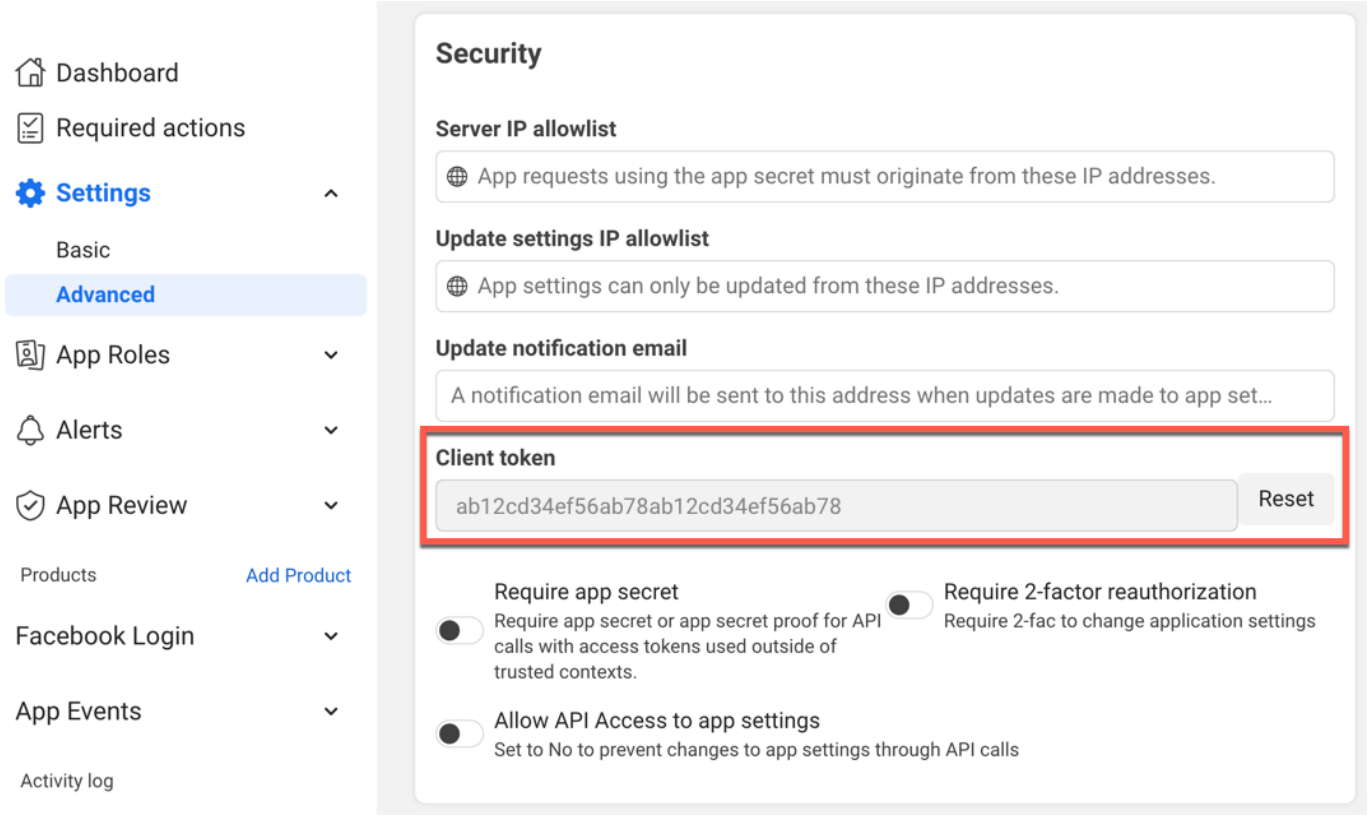
```
<string name="facebook_app_id">Your Facebook App ID</string>
<string name="fb_login_protocol_scheme">"fb" + Your Facebook App ID</string>
<string name="facebook_client_token">Your Facebook client token</string>
```

To find the values for `facebook_app_id` and `fb_login_protocol_scheme`, go to the Facebook Developer Console, and copy the App ID value:



> **Tip**
>
> Prefix your App ID value with `fb` to get the `fb_login_protocol_scheme` value.
> For example, if your **App ID** is `123456781234567`, your `fb_login_protocol_scheme` is `fb123456781234567`.

To find the value for `facebook_client_token`, go to the Facebook Developer Console, select your app, then navigate to **Settings** > **Advanced** > **Security**. Copy the **Client token** value:



For example:

```
<string name="facebook_app_id">123456781234567</string>
<string name="fb_login_protocol_scheme">fb123456781234567</string>
<string name="facebook_client_token">ab12cd34ef56ab78ab12cd34ef56ab78</string>
```

Apple

For Sign in with Apple:

1. Add the `AppAuth` dependency to the `build.gradle` file:

```
implementation 'net.openid:appauth:0.7.1'
```

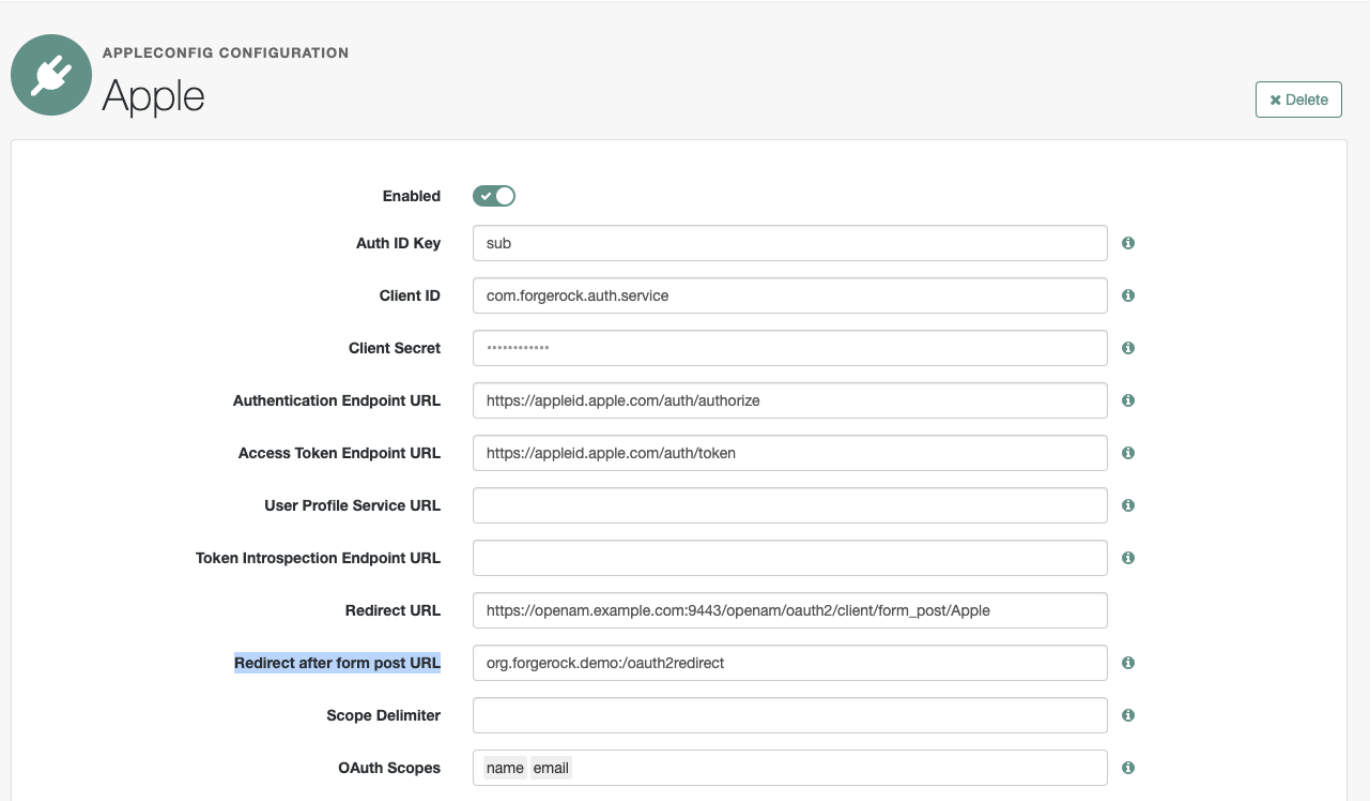
2. Add the following to the `AndroidManifest.xml` file:

```
<activity
    android:name="net.openid.appauth.RedirectUriReceiverActivity"
    tools:node="replace"
    android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" />
        <data android:scheme="@string/apple_scheme" />
    </intent-filter>
</activity>
```

3. Add the following attribute to the `string.xml` file:

```
<string name="apple_scheme">Your Redirect after form post URL Scheme</string>
```

To find the value of `apple_sheme`, go to **Services** > **Social Identity Provider Service** > **Secondary Configurations** > **Apple**:



In this example, the scheme would be `org.forgerock.demo`.

Set up social login in iOS apps

This page shows how to use the Ping SDK for iOS with authentication journeys that provide social login and registration.

Setup the social providers

Configure Facebook

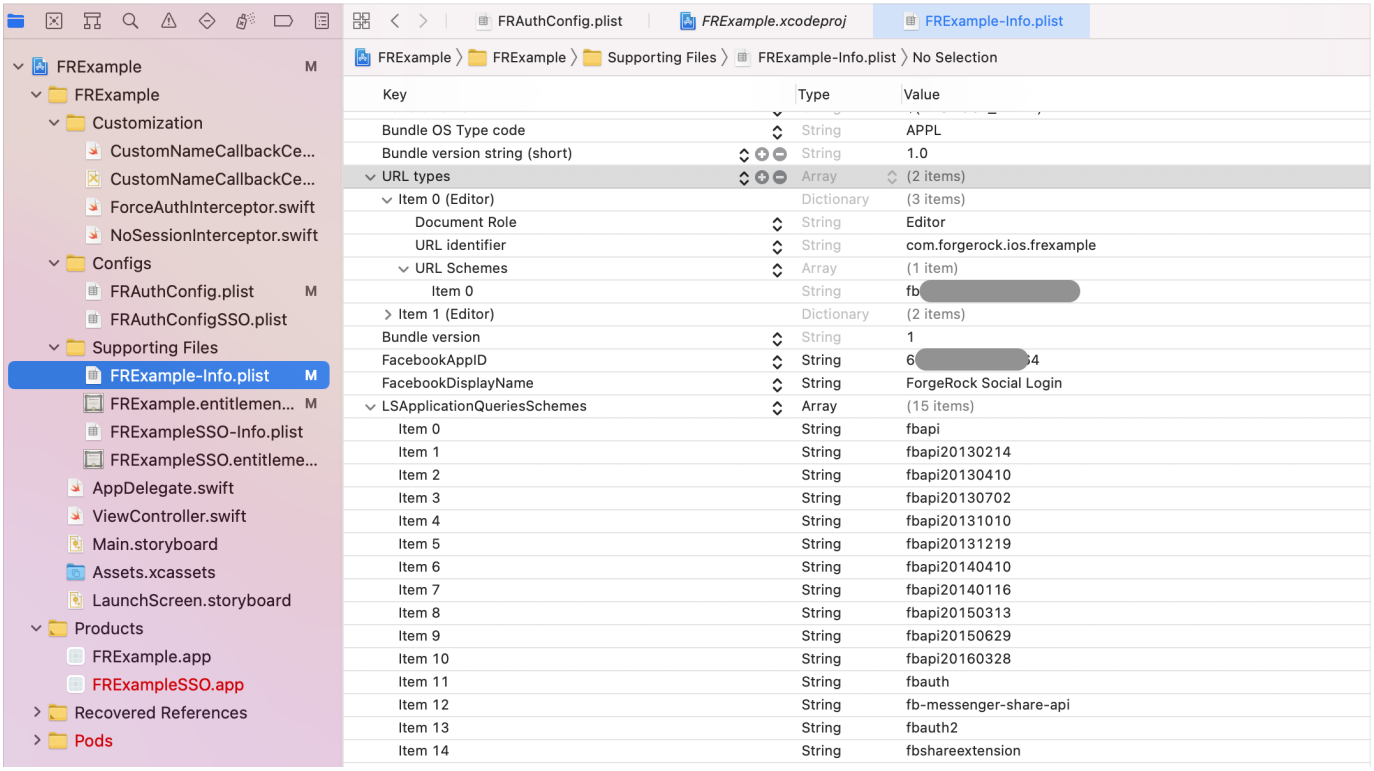
1. Create a Facebook client for iOS.

   For details, see Configure social login identity providers.

   Facebook provides you with the `.plist` configuration.

2. Follow the instructions on the page and copy the values in your app's `Info.plist` in Xcode.

   The final `Info.plist` file in your project, containing the Facebook generated `Custom URL Scheme`, and the `LSApplicationQueriesSchemes`, should look something like this:



3. Include the `FRFacebookSignIn` module in your project.

   The `FRFacebookSignIn` is a new module that is distributed separately of `FRAuth`.

   Assuming you are using CocoaPods, add the following lines in your projects `Podfile`:

```
pod 'FRAuth'
pod 'FRFacebookSignIn'
...
... Other Pods
...
```

4. Run the following command to install pods:

```
pod install
```

> **Note**
>
> Alternatively, you can add the `FRFacebookSignIn` module to your project using the Swift Package Manager in Xcode.

5. Initialize the Facebook sign-in handler in your app's `AppDelegate` file:

    1. Locate the following method:

    ```
    func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions:
    [UIApplication.LaunchOptionsKey: Any]?)
    ```

    2. Add a call to the `FacebookSignInHandler.application(_:didFinishLaunchingWithOptions:)` method, before the `return true` line:

    ```
    FacebookSignInHandler.application(application, didFinishLaunchingWithOptions: launchOptions)
    ```

    The result might resemble the following:

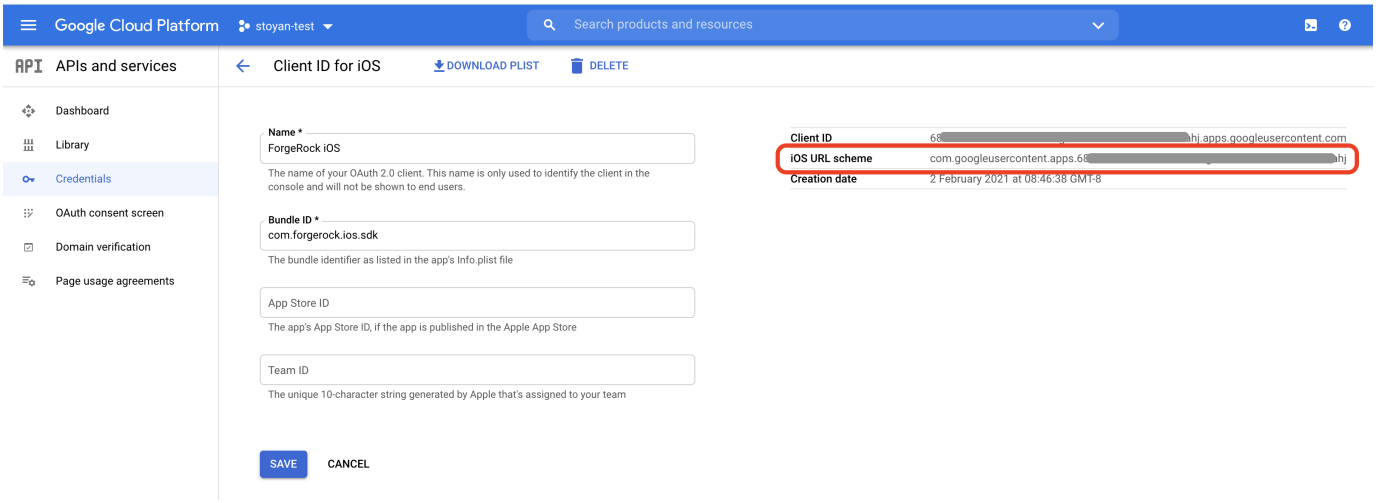
    ```
    func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions:
    [UIApplication.LaunchOptionsKey: Any]?) -> Bool {

        // Enable logs for all levels
        FRLog.setLogLevel([ .all])

        // Initialize the Facebook sign-in handler
        FacebookSignInHandler.application(application, didFinishLaunchingWithOptions: launchOptions)

        return true
    }
    ```

Configure Google

1. Create a Google client for iOS.

    For details, refer to Create a Google client.

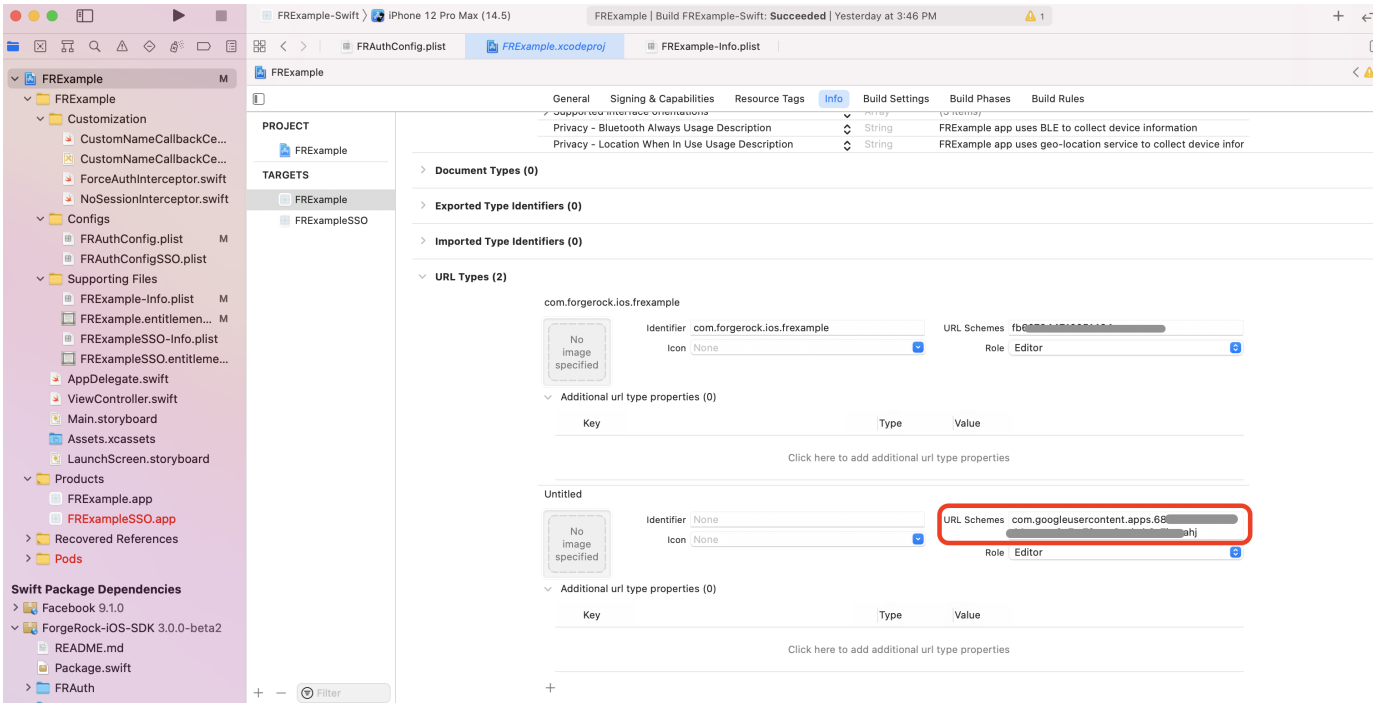
2. Access the client in the Google Console, and make a note of the generated custom iOS URL scheme:



3. Configure your Xcode project with the Google generated custom iOS URL scheme:

    1. Select your project file, select the app target, and in the **Info** pane, expand the **URL Types** option.
    2. Click on the + icon to add a new custom URL scheme, and paste the generated URL scheme in the **URL Scheme** field.

    The configuration should look something like this:



4. Include the `FRGoogleSignIn` module in your project.

    The `FRGoogleSignIn` is a new module that is distributed separately of `FRAuth`.

    Assuming you are using CocoaPods, add the following lines in your projects `Podfile`:

```
pod 'FRAuth'
pod 'FRGoogleSignIn'
...
... Other Pods
...
```

5. Run the following command to install pods:

```
pod install
```

> **Note**
>
> The `FRGoogleSignIn` module is not available through the Swift Package Manager.

Configure Apple

1. Create an Apple Client for iOS.

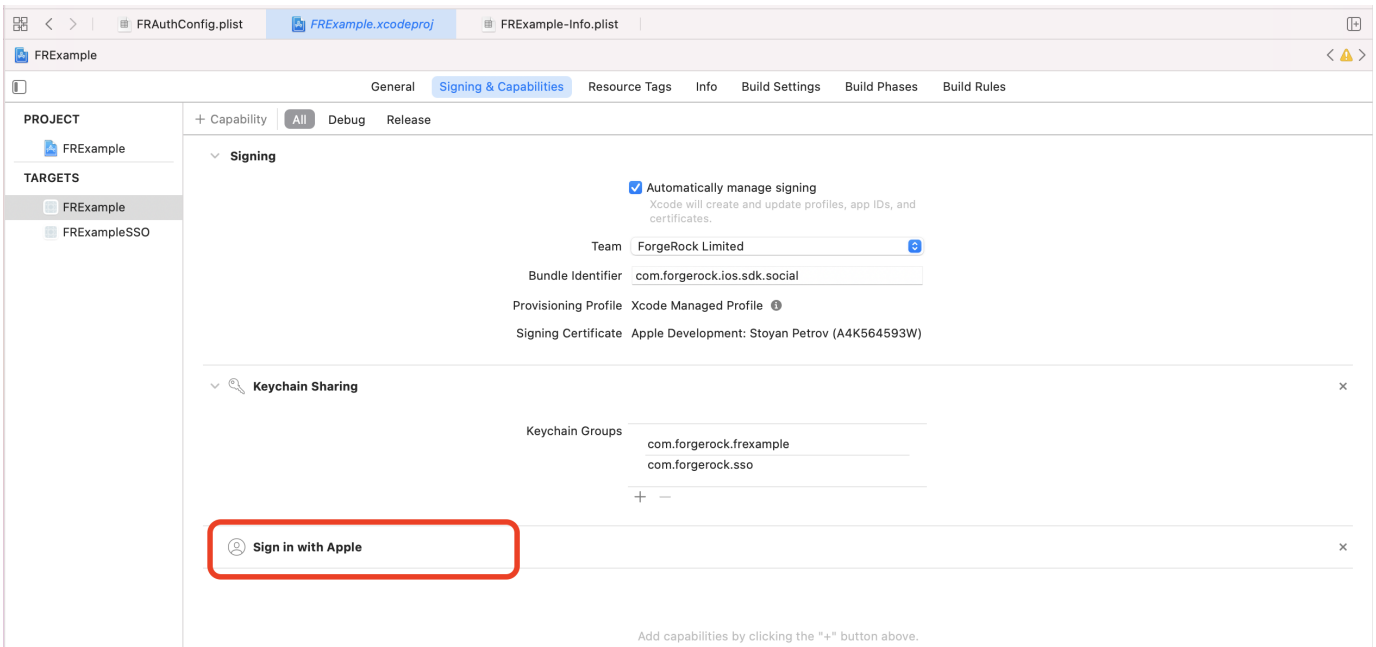
   For details, refer to Create an Apple client.

2. Configure your Xcode project with the Google generated custom iOS URL scheme.

3. Select your project file, select the app target and go to the **Signing & Capabilities** tab in Xcode.

4. Click the **+ Capability** button, and search for `Sign In with Apple`.

   After enabling the capability the Xcode page should look something like this.



Handle social login with the Ping SDK for iOS

After configuring social providers in PingOne Advanced Identity Cloud, and configuring your Xcode project to work with Facebook, Google, and Apple IdPs, you are ready to use the Ping SDK for iOS to authenticate.

The first callback your app encounters is the `SelectIdPCallback`, which lets the user choose their IdP. Use the `providers` array to display the available providers, and `setProvider()` method when the user makes a choice:

```
// Within your login flow
let selectIdPCallback = callback as? SelectIdPCallback
let providersArray = selectIdPCallback.providers

// display providers
// user makes choice

// Sets provider on the callback within `selectIdPCallback`
selectIdPCallback.setProvider(provider: providersArray[self.selectedIndex])
node.next { (user: FRUser?, node, error) in

    // Handle node

}
```

The next callback returned is the `IdPCallback`.

The SDK automatically identifies the correct IdP for authentication as long as the `IdPClient`, derived from the **Social Identity Provider Service** configuration in PingAM, contains `facebook`, `google` or `apple`. Detection is case-insensitive.

```
//  Node is returned with IdPCallback
let idpCallback = node.callbacks.first as! IdPCallback

//  Call the following to perform login
idpCallback.signIn(handler: nil, presentingViewController: self) {
  (token: String?, tokenType: String?, error: Error?) in

  //  Social Login flow is completed
  node.next { (user: FRUser?, node, error) in
    // Handle node
  }
}
```

Note

To override the automatic provider detection and identify the returned provider manually, check the `IdPClient` provider value in the returned `IdPCallback` as shown in the example below:

```
//  Node is returned with IdPCallback
let idpCallback = node.callbacks.first as! IdPCallback
//  Based on IdPClient in IdPCallback, choose the correct handler
var handler: IdPHandler?
if idpCallback.idpClient.provider == "facebook-ios" {
  handler = FacebookSignInHandler()
} else if idpCallback.idpClient.provider == "google-ios" {
  handler = GoogleSignInHandler()
} else if idpCallback.idpClient.provider == "apple-ios" {
  handler = AppleSignInHandler()
} else {
  throw error
}

//  If the handler has been found and initialized, call the following to perform login
idpCallback.signIn(handler: handler, presentingViewController: self) {
  (token: String?, tokenType: String?, error: Error?) in

  //  Social Login flow is completed
  node.next { (user: FRUser?, node, error) in
    // Handle node
  }
}
```

Set up social login in JavaScript apps

This page shows how to use the Ping SDK for JavaScript with authentication journeys that provide social login and registration.

The first callback your app encounters is the `SelectIdPCallback`, which lets the user choose their IdP. Use the `getProviders()` method to display the available providers, and `setProvider()` when the user makes a choice:

```
// Within your login flow
const cb = step.getCallbackOfType('SelectIdPCallback')
const providers = cb.getProviders();

// display providers
// user makes choice

// Sets provider on the callback within `step`
cb.setProvider(chosenProvider);

FRAuth.next(step);
```

The next callback is the `RedirectCallback`. Detect the presence of the callback, and call a special `redirect()` method on the `FRAuth` class, passing the whole step object as the argument:

```
// Within your login flow
if (step.getCallbackOfType('RedirectCallback')) {
  FRAuth.redirect(step);
}
```

This triggers a full browser redirect to the IdP. When the user authenticates with the IdP, they are redirected back to the app. When your application handles this redirect, check for the query parameters `code` and `state` for Facebook and Google, or `form_post_entry` for Apple. If they are present, call the `resume()` method on the `FRAuth` class:

```
// Application route handler for redirection from provider
// `code`, `state` and `form_post_entry` are "variablized" from URL
if ((code && state) || form_post_entry) {
  step = FRAuth.resume(window.location.href);
}
```

The `resume()` method gathers the appropriate URL information, and information from the previous step saved to browser storage prior to the redirect in order to properly resume the authentication journey.

Suspend and resume authentication with magic links

Applies to:

- ✓ Ping SDK for Android
- ✓ Ping SDK for iOS
- ✓ Ping SDK for JavaScript

The Ping SDKs support the *Suspended Authentication* feature provided by PingAM.

Suspended authentication lets you pause a user's progress through an authentication tree, and later resume from the same point.

Any input provided during authentication is saved when the authentication tree is suspended, and restored when the authentication tree is resumed. This lets the authentication tree continue after closing the browser, using a different browser, or even on a different device.

When suspending an authentication tree, you provide the user with a URL that contains a unique ID that lets them resume their authentication. The unique identifier for retrieving the saved progress can only be used once. These URLs are sometimes referred to as "magic links".

Note that the "magic link" represents a users' authentication journey up to the point it was paused. Ensure appropriate additional authentication is used in the remainder of a suspended authentication journey.

Typical use cases include password-less authentication, and email verification during progressive profile completion.

Prepare for suspended authentication

To use suspended authentication within your application, configure your server as follows:

- Enable outgoing email.

  Configure PingIDM to be able to send outbound email.

  > **Tip**
  >
  > When targeting applications that use the Ping SDK for JavaScript, alter the email template to include a URI that points to the application, rather than an instance of PingAM.
  >
  > Your app can then handle the URI the user clicks, and route it appropriately to resume the authentication.

  For more information, see Configure outbound email in the IDM documentation.

- Add an "Email Suspend Node" into an authentication tree.

  Enable suspended authentication by adding the node to a tree.

  For more information, see Suspended authentication in the PingAM documentation.

Handle the suspended authentication callback

The Ping SDKs receive a `SuspendedTextOutputCallback` when a "Email Suspend Node" is reached:

```
{
    "type": "SuspendedTextOutputCallback",
    "output": [{
        "name": "message",
        "value": "An email has been sent to the address you entered. Click the link in that email to proceed."
    }, {
        "name": "messageType",
        "value": "0"
    }]
}
```

Your application should display the `message` field, which instructs the user on how to proceed.

Authentication is now suspended. The Ping SDKs can resume authentication by using the `suspendedId` parameter that was emailed to the user.

Capture the resume URI

Your application must be able to capture the URI that the user is emailed. That URI contains the `suspendedId` parameter that is used to resume the user's authentication or registration journey.

Exact details on how to capture or intercept the URI from the email are beyond the scope of this documentation. However, the following resources may prove useful:

Android:

- Create Deep Links to App Content
- Verify Android App Links

iOS:

- Universal Links for Developers

Ensure your application is able to intercept the resume URI, and obtain the value of the `suspendedId` parameter it contains, before continuing to the next section.

Resume a suspended authentication

After obtaining the value of the `suspendedId` parameter, use it in you application to continue the user's authentication or registration journey, as follows:

Resume authentication in an Android app

The `FRSession` interface accepts the resume URI, including the `suspendedId` parameter:

```
FRSession.authenticate(Context, Uri, NodeListener<FRSession>)
```

> **Note**
>
> If the specified URI scheme, host, or port does not match with those of the PingAM instance configured for the app, the SDK throws an exception.

For example, you could retrieve the resume URI from the 'intent', and pass it into the SDK as follows:

```
Uri resumeURI = getIntent().getData();
//Note that SuspendedAuthSessionException (401) is returned if suspendedId is invalid or expired
FRSession.authenticate(Context context, Uri resumeUri, NodeListener<FRSession>)
```

Resume authentication in an iOS app

The `FRSession` interface accepts the resume URI, including the `suspendedId` parameter:

```
@objc public class FRSession: NSObject {
    public static func authenticate<T>(resumeURI: URL, completion:@escaping NodeCompletion<T>)
}
```

Examples:

AppDelegate.swift

```
class AppDelegate: UIResponder, UIApplicationDelegate {

  // This method is one of AppDelegate protocol that is invoked when
  // iOS tries to open the app using the app's dedicated URL
  func application(_ app: UIApplication, open url: URL, options: [UIApplication.OpenURLOptionsKey: Any] = [:]) ->
Bool {

    // validate the resumeURI contains 'suspendedId' parameter
    let resumeURL = url

    //  With given resumeURI, use FRSession to resume authenticate flow
    FRSession.authenticate(resumeURI: resumeURL) { (token: Token?, node, error) in
        //  Handle Node, or the result of continuing the the authentication flow
    }
  }
}
```

SceneDelegate.swift

```
class SceneDelegate: UIResponder, UIWindowSceneDelegate {

  func scene(_ scene: UIScene, openURLContexts URLContexts: Set<UIOpenURLContext>) {
    if let url = URLContexts.first?.url {
      let resumeURL = url  // validate the resumeURI contains 'suspendedId' parameter

      //  With given resumeURI, use FRSession to resume authenticate flow
      FRSession.authenticate(resumeURI: resumeURL) { (token: Token?, node, error) in
        //  Handle Node, or the result of continuing the the authentication flow
      }
    }
  }
}
```

Resume authentication in a JavaScript app

The method `next()` in the `FRAuth` class has been updated to accept the `suspendedID` value:

```
interface StepOptions extends ConfigOptions {
  query: {
    suspendedId: string; // you must have captured the suspendedId from the users' resumeURI
  };
}

abstract class FRAuth {
  public static async next(
    previousStep?: FRStep,
    options?: StepOptions,
  ): Promise<FRStep | FRLoginSuccess | FRLoginFailure> {
    const nextPayload = await Auth.next(previousStep ? previousStep.payload : undefined, options);

    // ... continue as normal
  }
}
```

For example, your app code may resemble the following:

```
const step = await FRAuth.next(null, {query: {suspendedId: 'i1PUHHWq6bTi3HxNjFGIqEask4g'}});
```

More information

- Suspended authentication
- Configure outbound email
- Email suspend node

Set up transactional authorization

Applies to:

- ✓ Ping SDK for Android
- ✓ Ping SDK for iOS
- ✓ Ping SDK for JavaScript

The Ping SDKs have builtin support for transactional authorization.

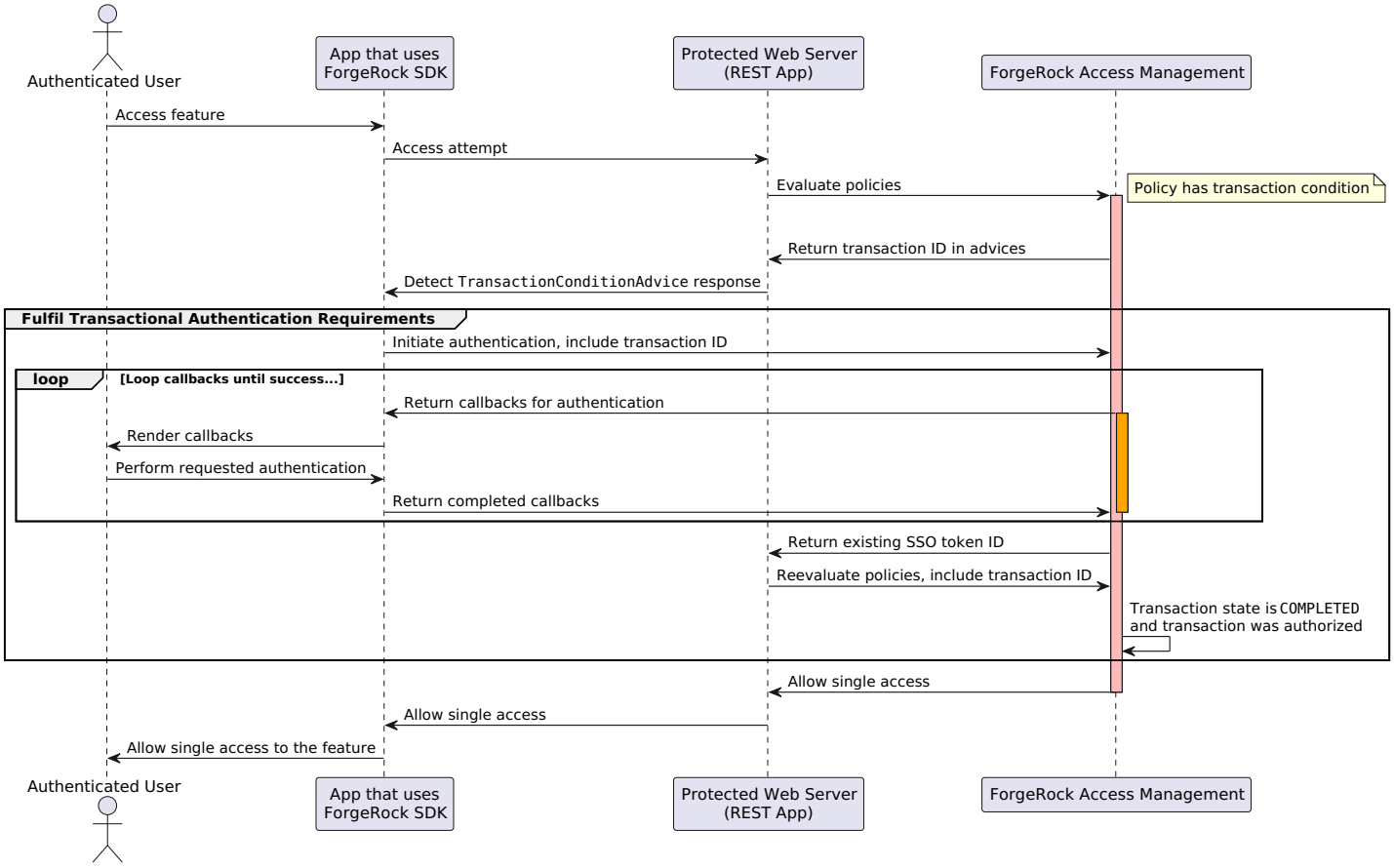
Transactional authorization requires a user to authorize every access to a resource. It is part of an PingAM policy that grants single-use or one-shot access.

For example, a user might approve a financial transaction with a one-time password (OTP) sent to their device, or respond to a push notification to confirm that they have indeed signed on from an unexpected location.

Performing the additional action successfully grants access to the protected resource but only once. Additional attempts to access the resource require the user to perform the configured actions again.

How does transactional authorization work?

The following diagram shows the flow used during transactional authorization:



When the Ping SDKs attempt to access a resource protected with transactional authorization, PingAM returns JSON that has an empty `actions` attribute. A unique transaction ID (TxId) is also included under `/advices/TransactionConditionAdvice`.

For example:

```
{
    "resource": "https://app-backend.example.com:8000/protected/feature/",
    "actions": {},
    "attributes": {},
    "advices": {
        "TransactionConditionAdvice": [
            "7b8bfd4c-60fe-4271-928d-d09b94496f84"
        ]
    },
    "ttl": 0
}
```

The Ping SDKs detect that transactional authorization is required, and make a call to the `/authenticate` endpoint to begin to fulfil the requirements specified in the policy protecting the resource. The call must include the `TxId` value originally received from PingAM.

PingAM responds to the request with a series of required callbacks to fulfil the policy.

Each callback is handled by the SDK; for example, by rendering UI for the user to complete, or responding to a push notification.

When all the callbacks have been completed, the SDK attempts to access the protected resource again, using the same session or OAuth 2.0 token as before. The SDK adds the transaction ID into the policy evaluation as an `environment` property:

```
{
    "resources" : ["https://app-backend.example.com:8000/protected/feature/"],
    "application" : "iPlanetAMWebAgentService",
    "subject" : {
        "ssoToken" : "AQIC5w....*AJTMQAA*"
    },
    "environment": {
        "TxId": ["77b8bfd4c-60fe-4271-928d-d09b94496f84"]
    }
}
```

As the transaction ID matches an entry in PingAM's completed transaction list, PingAM returns a new policy evaluation result, including the actions the SDK-based application can now perform:

```
{
    "resource": "https://app-backend.example.com:8000/protected/feature/",
    "actions": {
        "POST": true,
        "GET": true
    },
    "attributes": {},
    "advices": {},
    "ttl": 0
}
```

For more information on transactional authorization, and how to set up PingAM to use it, see Transactional authorization in the PingAM documentation.

Handle transactions in an Android app

In this example, an API is protected by PingGateway and PingAM.

The example adds a `x-authenticate-response` header to the access request. This header causes IG to return the advice as JSON in a header named `Www-Authenticate`.

The SDK provides interceptors for handling the returned advice:

- `AdviceHandler`
- `IdentityGatewayAdviceInterceptor`

After obtaining the advice, pass it to the `authenticate()` method of the `FRSession` class.

The following example shows how to use the interceptors to handle the advice using `OKHttpClient`:

```
val builder: OkHttpClient.Builder = OkHttpClient.Builder()

builder.addInterceptor(object: IdentityGatewayAdviceInterceptor() {
  override fun getAdviceHandler(advice: PolicyAdvice): AdviceHandler {
    return object: AdviceHandler {
      override suspend fun onAdviceReceived(
        context: Context,
        advice: PolicyAdvice) {
        // Authenticate the advice with
        // FRSession.getCurrentSession().authenticate(context, advice, ...)
      }
    }
  }
})

builder.cookieJar(SecureCookieJar.builder().context(context).build())

val client: OkHttpClient = builder.build()
val requestBuilder: Request.Builder = Request.Builder().url(api)

requestBuilder.addHeader("x-authenticate-response", "header");

val request = requestBuilder.build()

client.newCall(request).enqueue(object: Callback {
  override fun onFailure(call: Call, e: IOException) {
    // Handle Failure
  }

  override fun onResponse(call: Call, response: Response) {
    // Handle Response
  }
})
```

Handle transactions in an iOS app

The following steps demonstrate how to handle transactional authorization in the Ping SDK for iOS.

This example assumes interaction directly with PingAM.

If the resource server is protected by IG, and routes are configured for protected resources, the optional steps are not required, as the SDK is able to deal directly with the responses from IG.

1. Create an `AuthorizationPolicy` with the URL to evaluate policies against, and a delegate of `AuthorizationPolicyDelegate`:

   ```
   let authPolicy = AuthorizationPolicy(
     validatingURL: ["https://protectedendpoint"],
     delegate: self
   )
   ```

2. Add `AuthorizationPolicy` to `FRURLProtocol`

```
FRURLProtocol.authorizationPolicy = authPolicy
```

3. Register the `FRURLProtocol` class:

```
URLProtocol.registerClass(FRURLProtocol.self)
```

4. Create a `URLSessionConfiguration` with `FRURLProtocol`, and create a `URLSession` with the configuration.

   (*Optional*) If using IG, add the `x-authenticate-response` header so that IG returns the advice response as JSON in a header named `Www-Authenticate`, rather than as a redirect with query parameters:

```
// Configure FRURLProtocol for HTTP client
let config = URLSessionConfiguration.default
config.protocolClasses = [FRURLProtocol.self]

var request = URLRequest(url: "https://protectedendpoint")
request.setValue("header", forHTTPHeaderField: "x-authenticate-response")

self.urlSession = URLSession(configuration: config)
self.urlSession.dataTask(with: request) { (data, response, error) in }.resume()
```

5. (*Optional*) If the SDK is unable to parse the response into `policyAdvice`, construct it with the given response by implementing the `AuthorizationPolicyDelegate.evaluateAuthorizationPolicy()` method:

```
extension YourClass: AuthorizationPolicyDelegate {
  func evaluateAuthorizationPolicy(
    responseData: Data?,
    response: URLResponse?,
    error: Error?
  ) -> PolicyAdvice? {
    if let httpResponse = response as? HTTPURLResponse,
      httpResponse.statusCode == 401,
      let json = httpResponse.allHeaderFields["Www-Authenticate"] as? String,
      let policyAdvice = PolicyAdviceCreator().parseAsBase64(advice: json)
    {
      return policyAdvice
    } else {
      // If PolicyAdvice cannot be constructed, return 'nil' to stop authZ
      return nil
    }
  }
}
```

6. Initiate authentication tree flow, including `policyAdvice`, by implementing the `AuthorizationPolicyDelegate.onPolicyAdviseReceived()` method:

```
extension YourClass: AuthorizationPolicyDelegate {
  func onPolicyAdviseReceived(
    policyAdvice: PolicyAdvice, completion: @escaping FRCompletionResultCallback
  ) {
    FRSession.authenticate(policyAdvice: policyAdvice) { (token: Token?, node, error) in
      if error != nil {
        //Authentication failed
        completion(false)
        return
      }
      if let _ = token {
        completion(true)
      } else {
        // Handle node.
        // At the end of the authentication, you should get back a Token.
        // In this case you will need to call the completion handler
      }
    }
  }
}
```

7. (*Optional*) Decorate the original `URLRequest` object with updated information, by implementing the `AuthorizationPolicyDelegate.updateRequest()` method:

```
extension YourClass: AuthorizationPolicyDelegate {
  func updateRequest(originalRequest: URLRequest, txId: String?) -> URLRequest {
    // append txId into the request
    return request
  }
}
```

If a delegation method is not defined, the SDK appends the `_txid` query parameter automatically to the URL.

Handle transactions in a JavaScript app

Transactional authorization is built into the `HttpClient` module of the Ping SDK for JavaScript.

The `HttpClient` module detects when transactional authorization is enabled, and depending on your setup, initiates interaction with either PingAM or IG.

Ensure you specify `credentials: 'include'`, so that the request includes the necessary cookies.

When transactional authorization is enabled and callbacks are returned, your client app must implement the necessary user interaction. Ensure that you iterate through returned callbacks until you receive a success or failure response.

When you do receive a success response, make a new request to the initial resource endpoint, which will now be authorized.

The following code shows a sample JavaScript implementation. The included `authorization` middleware handles the callbacks that the configured transactional authorization returns:

```
console.log('Make a $200 withdrawal from account');
return HttpClient.request({
    init: {
        method: 'POST',
        body: JSON.stringify({ amount: '200' }),
        credentials: 'include',
    },
    authorization: {
        handleStep: async (step) => {
            console.log('Withdrawal endpoint is set up for transational authorization...');
            step.getCallbackOfType('ValidatedCreateUsernameCallback').setName(un);
            step.getCallbackOfType('ValidatedCreatePasswordCallback').setPassword(pw);
            return Promise.resolve(step);
        },
    },
    timeout: 0,
    url: `${resourceUrl}/withdraw`,
});
```

More information

- Transactional authorization
- MFA: Push authentication
- PingGateway
- Web agents
- Java agents

Set up QR code handling

Applies to:

- ✕ Ping SDK for Android
- ✕ Ping SDK for iOS
- ✓ Ping SDK for JavaScript

The Ping SDK for JavaScript has the following methods for handling QR codes:

`FRQRCode.isQRCodeStep(step)`

For determining if a step requires a QR code.

`FRQRCode.getQRCodeData(step)`

For extracting the QR code data from the step, such as the URI.

Example:

```
// Import the module
import { FRQRCode } from '@forgerock/javascript-sdk';

// Determine if a step is a QR code step
const isQRCode: boolean = FRQRCode.isQRCodeStep(step);

if (isQRCode) {
  // Extract QR code data
  const data: {
    message: string,
    use: string,
    uri: string
  } = FRQRCode.getQRCodeData(step);

  // Render a QR code using the `data` from the step

}
```

Integrate with Google reCAPTCHA Enterprise

Applies to:

- ✓ ForgeRock SDK for Android
- ✓ ForgeRock SDK for iOS
- ✓ ForgeRock SDK for JavaScript

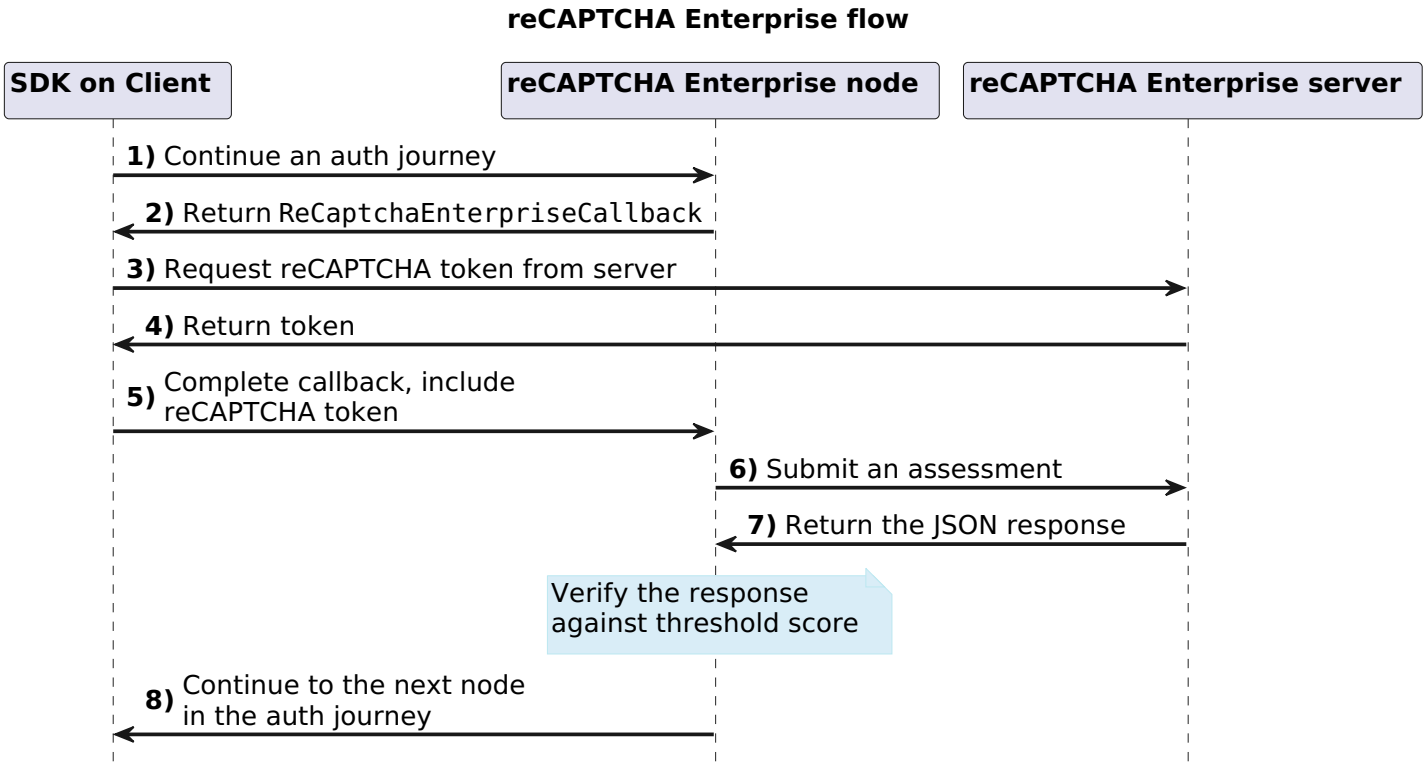
Google ReCAPTCHA Enterprise uses advanced risk analysis techniques to distinguish between humans and bots. reCAPTCHA Enterprise is useful when you want to detect automated attacks or threats against your website or mobile apps.

These threats typically originate from scripts, mobile emulators, bot software, or humans.

Google reCAPTCHA Enterprise offers enhanced detection compared to earlier versions, with more granular scores, reason codes for risky events, mobile app SDKs, password breach and leak detection, multi-factor authentication (MFA), and the ability to tune your site-specific model to protect enterprise businesses.

Understand how the SDKs work with Google reCAPTCHA Enterprise

The following diagram outlines the flow of information between the client that is using the Ping SDKs, the reCAPTCHA Enterprise node in your journey, and the Google reCAPTCHA servers:



1. The SDKs start or continue an authentication journey by visiting the `/authorize` endpoint
2. The next step in the journey is the **reCAPTCHA Enterprise** node, so the journey returns the a `ReCaptchaEnterpriseCallback` to the client.
3. The client uses the values in the callback to request a token from the reCAPTCHA server.
4. The reCAPTCHA server returns a unique token for the transaction to the client.
5. The client adds the token to the callback and returns it to the node.

   You can also add custom data to the payload that forms the assessment. See Customizing the assessment payload.
6. The node submits the data collected on the client to the reCAPTCHA server for assessment.
7. The reCAPTCHA server returns the response to the request to the node.

   > **Tip**
   >
   > You can enable to the **Store reCAPTCHA assessment JSON** option in the node to save this data in the state for additional processing later in the journey.
   > The node stores the JSON in a variable named `CaptchaEnterpriseNode.ASSESSMENT_RESULT`.

   The node verifies the response against the **Score threshold** you configure in the node, and continues the journey along the relevant outcome, `true` or `false`.
8. The client handles the next node as the journey continues.

Set up a journey for reCAPTCHA Enterprise

To enable reCAPTCHA Enterprise, add the reCAPTCHA Enterprise node to a journey.

This node returns a `ReCaptchaEnterpriseCallback` callback that the SDKs can handle to perform the reCAPTCHA Enterprise assessment.

Prepare your app for reCAPTCHA Enterprise

To add support for reCAPTCHA Enterprise to your apps, complete the following prerequisite tasks.

Android

Add the following to your `build.gradle` configuration file:

```
implementation("com.google.android.recaptcha:recaptcha:18.x.x")
```

iOS

You can add the dependencies using Cocoapods or Swift Package Manager (SPM).

1. If you do not already have CocoaPods, install the latest version.
2. If you do not already have a Podfile, in a terminal window, run the following command to create a new Podfile:

```
pod init
```

3. Add the following lines to your Podfile:

```
pod 'FRCaptchaEnterprise'
```

4. Run the following command to install pods:

```
pod install
```

1. With your project open in **Xcode**, select **File** > **Add Package Dependencies**.
2. In the search bar, enter the ForgeRock SDK for iOS repository URL: `https://github.com/ForgeRock/forgerock-ios-sdk`.
3. Select the `forgerock-ios-sdk` package, and then click **Add Package**.
4. In the **Choose Package Products** dialog, set the **Add to Target** field for the `FRCaptchaEnterprise` library top be the name of your project.
5. Click **Add Package**.
6. In your project, import the library:

```
// Import the reCAPTCHA Enterprise library
import FRCaptchaEnterprise
```

JavaScript

In a JavaScript app you must complete the following steps before using reCAPTCHA Enterprise:

1. Load the reCAPTCHA Enterprise JavaScript.

   To learn more, refer to Install score-based keys on websites in the Google Cloud documentation.

2. Obtain a reCAPTCHA Enterprise token.

   To learn more, refer to Retrieve a token in the Google Cloud documentation.

Handling the callback with the SDK

Use code similar to the following to handle the `ReCaptchaEnterpriseCallback` in your client-side code using the Ping SDKs:

Android

```
ReCaptchaEnterpriseCallback callback;
try {
  callback.execute(application = application)
}
catch (e: Exception) {
  if(e is RecaptchaException) {
    Logger.error(
      "RecaptchaException",
      "${e.errorCode}:${e.message}"
    )
  }
  Logger.error("RecaptchaException", e.message)
}
```

iOS

```
var captcha: ReCaptchaEnterpriseCallback?
if #available(iOS 13.0, *) {
  Task {
    do {
      try await captcha.execute(action: "login")
    }
    catch let error as RecaptchaError {
      // Handle errors
    }
  }
}
```

JavaScript

Return the `reCAPTCHA_Enterprise_Token` you obtained earlier in the callback as follows:

```
const callback = step.getCallbackOfType<forgerock.ReCaptchaEnterpriseCallback>(
  forgerock.CallbackType.ReCaptchaEnterpriseCallback,
) as forgerock.ReCaptchaEnterpriseCallback;

callback.setResult(reCAPTCHA_Enterprise_Token);

return forgerock.FRAuth.next(step);
```

Customizing the assessment payload

You can add additional data to customize the payload that the server sends to the Google reCAPTCHA Enterprise for assessment.

Add data to the payload to leverage additional functionality provided by reCAPTCHA Enterprise.

The JSON format the payload expects is as follows:

```
{
  "token": string,
  "siteKey": string,
  "userAgent": string,
  "userIpAddress": string,
  "expectedAction": string,
  "hashedAccountId": string,
  "express": boolean,
  "requestedUri": string,
  "wafTokenAssessment": boolean,
  "ja3": string,
  "headers": [
    string
  ],
  "firewallPolicyEvaluation": boolean,
  "transactionData": {
    object (TransactionData)
  },
  "userInfo": {
    object (UserInfo)
  },
  "fraudPrevention": enum (FraudPrevention)
}
```

By default, the SDK or the node itself populates the following fields:

- `token`
- `siteKey`
- `userAgent`

- `userIpAddress`
- `expectedAction`

You can however also override these values if it suits your use case.

> **Note**
>
> You can add custom payload data as part of an authentication journey that includes the reCAPTCHA Enterprise node. Custom data in the journey overrides any custom data added by the client.

To learn more about the payload, refer to Project Assessments - Event in the Google Developer documentation.

To add custom data for an assessment, use the `setPayload()` method:

Android

```
// Optional payload values for customization
callback.setPayload(
    JSONObject().put("firewallPolicyEvaluation",false)
)
```

iOS

```
// Optional payload values for customization
callback.setPayload([
  "firewallPolicyEvaluation", false
])
```

JavaScript

```
// Optional payload values for customization
callback.setPayload({
  "firewallPolicyEvaluation": false
});
```

Returning custom error codes

You can return a custom error code to the node, which can then continue the journey based on the values:

Android

```
// Optional custom error code
callback.setClientError("custom_client_error")
```

iOS

```
// Optional custom error code
callback.setClientError("custom_client_error")
```

JavaScript

```
// Optional custom error code
callback.setClientError('custom_client_error');
```

Ping SDKs API reference

View API references for the different modules provided in the Ping SDKs.

Ping SDK for Android

- forgerock-auth
- forgerock-auth-ui
- forgerock-authenticator
- forgerock-core
- ping-protect

Ping SDK for iOS

- FRAuth
- FRAuthenticator
- FRCore
- FRDeviceBinding
- FRFacebookSignIn
- FRProximity
- FRUI
- PingOne Protect

Ping SDK for JavaScript

- FR Core
- FR Core with UI
- PingOne Protect module

More resources

- Ping (ForgeRock) Authenticator module API reference
- Ping (ForgeRock) Login Widget API reference

SDK troubleshooting

Ping SDK for JavaScript

Troubleshoot your JavaScript apps

Knowledge base

Browse our extensive list of articles regarding diagnosing and fixing issues you might encounter when using the Ping SDKs.

Getting support

Discover how to contact our support team for specific assistance not covered in the documentation or knowledge base.

Troubleshooting your JavaScript app

This section contains information on how to diagnose issues within the application, or with communication to the server.

How to fix cross-domain, authenticated-request failures?

As browsers continue to implement stricter privacy and security configurations, third-party cookies are commonly being disabled by default. Third-party cookies are defined by a web app and corresponding server sharing cookies with mismatched domains: `example.app` and `server.com`.

This is common when the PingOne Advanced Identity Cloud is running on the provided domain `forgeblocks.com`, and a web app is running on a separately hosted domain, `company.com`. When third-party cookies are disabled by the browser, the set cookie instruction sent by PingOne Advanced Identity Cloud on `forgeblocks.com`, such as the session cookie, to the web app on `company.com` will be ignored by the browser. Hence, the cookie will not be written. This will lead to subsequent requests requiring authentication to fail due to the missing cookie.

The Authorization Code Flow (OAuth 2.0) is one such request and the resulting error will resemble this:

```
SecurityError: Blocked a frame with origin "http://example.com" from accessing a cross-origin frame.
```

> **Note**
>
> This isn't a CORS configuration issue, even though this mentions "cross-origin". In most cases, this is due to a request to the `/authorize` endpoint resulting in a redirection within the iframe to the login page, due to the missing session cookie, which is not allowed to be rendered in an iframe for security reasons.

Use a shared parent domain

The best action to take is to ensure all entities of your system are on the same parent domain. This can be done by the use of subdomains with a shared parent domain often using a reverse proxy that routes a request to the appropriate application server. An example would be `store.example.com`, the web app, and `accounts.example.com`, the IAM server.

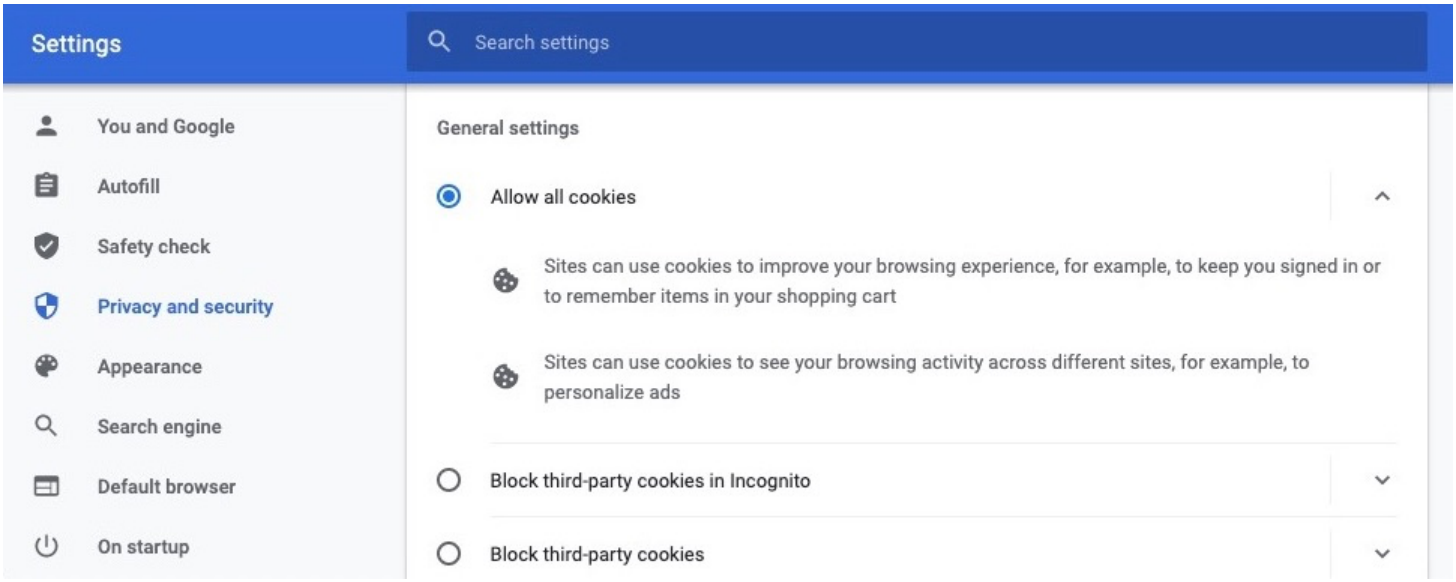
If a shared domain is not possible, the browser configuration will have to be changed.

macOS and iOS Safari workaround

In Apple's Safari, third-party cookies are disabled by default. Disabling **Prevent Cross-site Tracking** in Safari's privacy settings lets Safari use third-party cookies.

Google Chrome and Incognito Mode workaround

Currently, in Google Chrome, third-party cookies are disabled when in Incognito mode by default. This can be fixed by changing the cookie setting from **Block third-party cookies in Incognito** to **Allow all cookies** in Chrome settings under **Privacy and security** > **Cookies and other site data**.



How do I enable "platform authenticators" in Safari (iOS and macOS)?

Apple's latest operating systems (iOS 14 and macOS 11 as of this writing) place a strict requirement for the use of their Face ID and Touch ID (platform authenticators) on the web. The requirement is that a "user gesture" needs to be the trigger prior to, and directly associated with, accessing `navigator.credentials`, the Web Authentication (aka WebAuthn) API. If there's not a clear connection between a "user activated event" and the access of the API, Apple will log a warning similar to the following:

```
User gesture is not detected.
To use the platform authenticator, call 'navigator.credentials.get' within user activated events.
```

This is essentially communicating that the call site of the credentials API needs to happen within the callback function of a user-activated event, like click, touch, keydown, and so on, as opposed to events like DOM load or routing-related events. If there's too much elapsed time between the event and the API access, or the connection, scope or context of the event is cleared from the DOM or runtime, Apple will not allow the use of its platform authenticators.

> **Note**
>
> Access to non-platform authenticators like roaming (USB) keys will still be available regardless of user activated events.

Solution

First, make sure the "user activated event," such as click, touch, keydown, and so on, precedes the calling of the credentials API, and the call site of the API happens within the event handler's callback function. Calling the API after a routing event or a DOM load event will not work.

If the issue persists, ensure the DOM element with which the user interacts (button or link), the associated DOM event's callback function, and its relationship with the credentials API call site is all preserved. Once the API has successfully been triggered, the DOM and event handler-related code can be cleared.

For more information, see this Webkit blog article: Meet Face ID and Touch ID for the Web.

How do I fix "a mutation operation was attempted on a database that did not allow mutations"?

This error message can occur if you are using Firefox's Private Window mode, which prevents write operations to IndexedDB. For more information, see IndexedDB does not function in private browsing mode.

> **Note**
>
> In Ping SDK for JavaScript 2.2 and earlier, `indexedDB` is the default storage mechanism for OAuth/OIDC tokens.

Solution

Because IndexedDB is not writable within Firefox's private mode, we recommend using either `localStorage` or `sessionStorage`. You can configure the use of either by setting the `tokenStore` value on the config object to your preferred option.

Here's an example:

```
Config.set({
  // standard config settings ...
  tokenStore: 'localStorage', // Or, 'sessionStorage'
});
```

> **Note**
>
> The `indexedDB` option was removed in the ForgeRock SDK for JavaScript v4.0.0. Use `sessionStorage` or `localStorage` instead.

Knowledge base

The Knowledge Base contains information on how to diagnose issues within your application, or with communication to the authorization server.

For troubleshooting information, see the following articles in the Knowledge Base:

- Ping SDK for Android Troubleshooting
- Ping SDK for iOS Troubleshooting
- Ping SDK for JavaScript Troubleshooting

Additional Articles

- How do I troubleshoot issues with the CORS filter in PingAM/OpenAM (All versions)?

Getting support

Ping Identity provides support services, professional services, training, and partner services to assist you in setting up and maintaining your deployments. For a general overview of these services, see https://www.pingidentity.com.

Ping Identity has staff members around the globe who support our international customers and partners. For details on Ping Identity's support offering, visit https://www.pingidentity.com/support.

Ping Identity publishes comprehensive documentation online:

- The Ping Identity Knowledge Base offers a large and increasing number of up-to-date, practical articles that help you deploy and manage Ping Identity Platform software.

  While many articles are visible to everyone, Ping Identity customers have access to much more, including advanced information for customers using Ping Identity Platform software in a mission-critical capacity.

- Ping Identity product documentation, such as this document, aims to be technically accurate and complete with respect to the software documented. It is visible to everyone and covers all product features and examples of how to use them.

Introducing the DaVinci client for DaVinci flows

Server support:

- ✓ PingOne
- ✗ PingOne Advanced Identity Cloud
- ✗ PingAM
- ✗ PingFederate

SDK support:

- ✓ Ping SDK for Android
- ✓ Ping SDK for iOS
- ✓ Ping SDK for JavaScript

The DaVinci clients provide powerful orchestration capabilities with PingOne DaVinci. They enable you to consume DaVinci flows to meet your use cases, all while providing a native Android or iOS, or a single-page app JavaScript experience.

You have complete control of your UI, so you can create the tailored experience you desire for your end users, all while leaving the DaVinci client to do the heavy lifting of communication between your app and your DaVinci flows.

UI Development

You are in charge of the experience your end users have in your Android, iOS, or JavaScript (SPA) applications.

Theme and brand your app the way you want to, focusing on your application logic and letting the DaVinci client communicate with PingOne DaVinci.

Dynamically Updating of DaVinci Flows

The DaVinci client works with PingOne DaVinci server-driven orchestration. This means that as long as you don't hardcode UI elements in your application, and you appropriately handle the collector types, then you can update your DaVinci flow without needing to update your app or redeploy to the app stores.

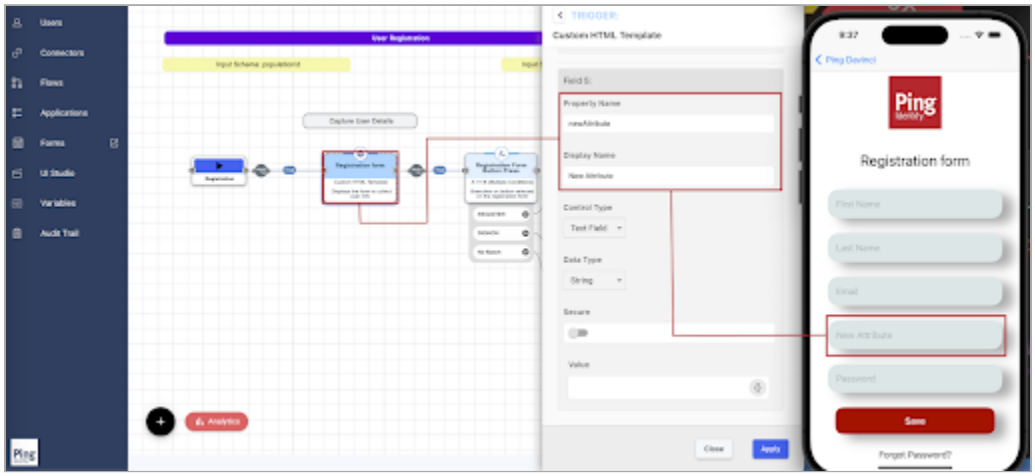


Figure 1. Your app responds to changes in your DaVinci flows, without redeploying.

This saves time and enables you to control your application orchestration experience without unnecessary burden.

Flow collectors

DaVinci sends requests to the DaVinci client from UI-related connectors. This enables you to step through each piece of information, and gather information from your end users appropriately.

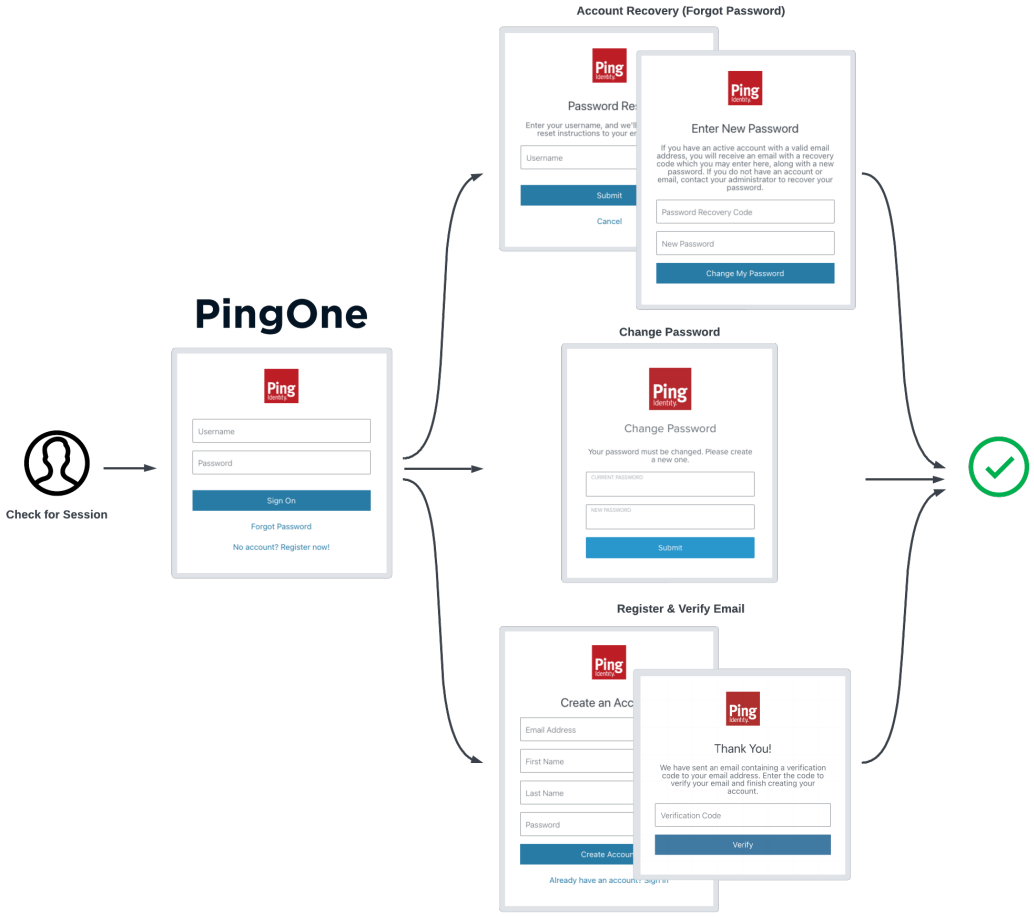


Figure 2. Step through flows and and collect input from your users.

For example, if you created a DaVinci flow to register a user, and you used the HTTP Connector - Custom HTML Template capability:

- First name, last name, and email address would be sent as the `TextCollector` type to your app, and you would determine the way in which you represented this field in your UI.
- The password field would be sent as the `PasswordCollector` type.
- Once the flow completes, PingOne sends back the necessary tokens, and the DaVinci client automatically stores, retrieves, and refreshes tokens as needed.

Token Management

The DaVinci client uses the OAuth 2.0 auth code flow, and support PKCE.

This method is the best practice for first-party applications. The SDK automatically handles token exchange for you, and also securely stores the tokens.

The DaVinci client handles token refresh automatically, so you don’t have to think about it.

Compatibility

Supported operating systems and browsers

Select a platform below to view the supported operating systems and browsers.

Android

The Ping SDK for Android supports the following versions of the Android operating system:

Supported Android versions and original release dates

Release	API Levels	Released
Android 15	35	September, 2024
Android 14	34	October, 2023
Android 13	33	March, 2022
Android 12	31, 32	October, 2021
Android 11	30	September, 2020
Android 10	29	September, 2019
Android 9 (Pie)	28	August, 2018

Important

We are updating how we determine which Android versions form our support policy for the Ping SDK for Android.
From **March 1st, 2025**, the support policy is as follows:

- Every public major release of Android within the last 6 years.
  For example, this would mean support for **Android 9** (API level 28) and later versions.

Supported browsers on Android

- Chrome - Two most recent major versions.

iOS

The Ping SDK for iOS supports the following versions of the iOS operating system:

Supported iOS versions and original release dates

Release	Released
iOS 18	September, 2024
iOS 17	September, 2023
iOS 16	September, 2022

Important

We are updating how we determine which iOS versions form our support policy for the Ping SDK for iOS. From **March 1st, 2025**, the support policy is as follows:

- Every public major release of iOS within the last 3 years.
  For example, this would mean support for **iOS 16** and later versions.

Supported browsers on iOS

- Safari - Two most recent major versions.

JavaScript / Login Widget

The Ping SDK for JavaScript, and the Ping (ForgeRock) Login Widget support the desktop and mobile browsers listed below.

Minimum supported Desktop browser versions

- Chrome 83
- Firefox 77
- Safari 13
- Microsoft Edge 83 (Chromium)

Supported Mobile browsers

- iOS (Safari) - Two most recent major versions of the operating system.
- Android (Chrome) - Two most recent major versions of the operating system.

JavaScript Compatibility with WebViews

A WebView allows you to embed a web browser into your native Android or iOS application to display HTML pages, and run JavaScript apps.

For example, the Android system WebView is based on the Google Chrome engine, and the iOS WebView is based on the Safari browser engine.

However, it is important to note that WebViews do not implement the full feature set of their respective browsers. For example, some of the browser-provided APIs that the Ping SDK for JavaScript requires are not available in a WebView, such as the WebAuthn APIs.

In addition, there are concerns that a WebView does not provide the same level of security as their full browser counterparts.

As the SDK requires full, spec-compliant, browser-supplied APIs for full functionality we **do not** support usage within a WebView.

We also do not support or test usage with any wrappers around WebViews.

Whilst you might be able to implement simple use-cases using the Ping SDK for JavaScript within a WebView, we recommend that you use an alternative such as opening a full browser, or using an in-app instance of a full browser such as Custom Tabs for Android or SFSafariViewController for iOS.

Supported PingOne fields and collectors

The DaVinci clients support the following connectors and capabilities:

- PingOne Forms Connector
  - **Show Form** capability
- HTTP Connector
  - **Custom HTML** capability

PingOne Form Connector fields

- Custom Fields support
- Toolbox support

Custom Fields support

Field (`Collector`)	Description	DaVinci module		
		Android	iOS	JavaScript
Text Input (`TextCollector`)	Collects a single text string.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`
Password (`PasswordCollector`)	Collects a single text string that cannot be read from the screen.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`
Dropdown (`SingleSelectCollector`)	Collects a value from a dropdown containing one or more text strings.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`
Combobox (`MultiSelectCollector`)	Collects a value from a dropdown containing one or more text strings, the user can enter their own text string.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`
Radio Button List (`SingleSelectCollector`)	Collects a value from one or radio buttons.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`
Checkbox List (`MultiSelectCollector`)	Collects the value of one or more checkboxes.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`

Toolbox support

Field (`Collector`)	Description	DaVinci module		
		Android	iOS	JavaScript
Flow Button (`FlowCollector`)	Presents a customized button.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`
Flow Link (`FlowCollector`)	Presents a customized link.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`

Translatable Rich Text (`TextCollector`)	Presents rich text that you can translate into multiple languages.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`
Social Login (`IdpCollector`)	Presents a button to allow users to authenticate using an external identity provider, such as Apple, Facebook, or Google.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`

HTTP Connector fields

- HTTP Connector field and collector support
- HTTP Connector SK-Component support

HTTP Connector field and collector support

Field (`Collector`)	Description	DaVinci module		
		Android	iOS	JavaScript
Text field (`TextCollector`)	Collects a single text string.	☑ `1.0.0`	☑ `1.0.0`	☑ `1.0.0`
Password field (`PasswordCollector`)	Collects a single text string that cannot be read from the screen.	☑ `1.0.0`	☑ `1.0.0`	☑ `1.0.0`
Submit Button (`SubmitCollector`)	Sends the collected data to PingOne to continue the DaVinci flow.	☑ `1.0.0`	☑ `1.0.0`	☑ `1.0.0`
Flow Button (`FlowCollector`)	Triggers an alternative flow without sending the data collected so far to PingOne.	☑ `1.0.0`	☑ `1.0.0`	☑ `1.0.0`
Label (`LabelCollector`)	Display a read-only text label.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`
Radio / Dropdown (`SingleSelectCollector`)	Collects a single value from a choice of multiple options.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`

HTTP Connector SK-Component support

SK-Component (`Collector`)	Description	DaVinci module		
		Android	iOS	JavaScript
skIDP (`IdpCollector`)	Presents a button to allow users to authenticate using an external identity provider, such as Apple, Facebook, or Google.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`

Unsupported features:

Verify that your flow does not depend on any *unsupported* elements:

SKPolling components

The SKPolling component cannot be processed by the DaVinci Client and should not be included in flows.

Features such as Magic Link authentication require the **SKPolling** component and therefore cannot be used with the DaVinci Client.

Images

Images included in the flow cannot be passed to the SDK.

Default DaVinci client headers

Applies to:

- ✓ DaVinci client for Android
- ✓ DaVinci client for iOS
- ✓ DaVinci client for JavaScript

The DaVinci clients send a number of header values to the server with every outgoing request.

These headers can help you identify the client in your flows and help you correlate actions to a transaction in DaVinci audit logs. You can also use these values to alter the course of a DaVinci flow.

The default headers the DaVinci client always include are as follows:

`x-requested-with`

Identifies that the request comes from an app built with the Ping DaVinci client.

Default value: `ping-sdk`

`x-requested-platform`

Identifies the platform the DaVinci client is running on.

Default values:

Platform	Value
Android	`android`
iOS	`ios`

Platform	Value
JavaScript	`javascript`

`interactionId`

Returns the `interactionId` value provided by the server to help trace the transaction in server audit logs and dashboards.

Example value: `18484499-c551-4d99-c415-b01c79bedb47`

`interactionToken`

Returns the `interactionToken` value provided by the server to help trace the transaction in server audit logs and dashboards.

Example value:
`437783552aa3a5a8f0041028d5b8dac2d72f7e7ebd7f88a966fb690402f6571b964c3df8897cbe542e62721070b3f6fcc946f4dd2bc80b9df332d39657fcaaad4651884093a786910d6f1337bd8dda17b4fca48e8fa481469ce0df1f676e46d1a6fc30577d910010d4a2530f2d02e69f436d610992c79fcb0ca87131d0df3f9a`

Getting started with the DaVinci client

Applies to:

- ✓ DaVinci client for Android
- ✓ DaVinci client for iOS
- ✓ DaVinci client for JavaScript

The DaVinci client is designed to be flexible and can be customized to suit many different situations.

Learn more about installing, configuring, and customizing the DaVinci client in the sections below:

Install the DaVinci client

Learn how to install the DaVinci client into your applications.

Learn more »

Configure DaVinci client properties

Learn how to configure properties in the DaVinci clients so they can connect to your authorization server to authenticate your users and obtain tokens.

Learn more »

Localize DaVinci client UI

Learn how you can leverage the languages feature in PingOne to localize your client applications for different audiences.

Learn more »

Installing the DaVinci client

Applies to:

- ✓ DaVinci client for Android
- ✓ DaVinci client for iOS
- ✓ DaVinci client for JavaScript

You need to install the DaVinci client module into your application to use its functionality.

The method you use to install the module depends on which platform you are using.

Android

Installing the DaVinci client for Android

iOS

Installing the DaVinci client for iOS

JavaScript

Installing the DaVinci client for JavaScript

Installing the DaVinci client for Android

To install the DaVinci client into your Android app:

1. In the **Project** tree view of your Android Studio project, open the `build.gradle.kts` file.
2. In the `dependencies` section, add the following:

```
// Ping SDK social sign-on dependencies
implementation("com.pingidentity.sdks:davinci:1.1.0")
```

Installing the DaVinci client for iOS

You can use Swift Package Manager (SPM) or CocoaPods to add the DaVinci client to your iOS project.

Swift Package Manager

1. With your project open in **Xcode**, select **File** > **Add Package Dependencies**.
2. In the search bar, enter the Ping SDK for iOS repository URL: `https://github.com/ForgeRock/ping-ios-sdk`.
3. Select the `ping-ios-sdk` package, and then click **Add Package**.
4. In the **Choose Package Products** dialog, ensure that the `PingDavinci` and `PingExternalIdp` libraries are added to your target project.
5. Click **Add Package**.
6. In your project, import the library:

```
import PingDavinci
```

CocoaPods

1. If you do not already have CocoaPods, install the latest version.
2. If you do not already have a Podfile, in a terminal window, run the following command to create a new Podfile:

```
pod init
```

3. Add the following lines to your Podfile:

```
pod 'PingDavinci'
```

4. Run the following command to install pods:

```
pod install
```

Installing the DaVinci client for JavaScript

The DaVinci client for JavaScript as available as an **npm** module at @forgerock/davinci-client.

To install the module into your JavaScript project, run the following `npm` command:

```
npm install @forgerock/davinci-client@1.1.0
```

After installation, import the module into your app as follows:

```
import { davinci } from '@forgerock/davinci-client';
```

Next Steps

After installing the DaVinci client, you should configure it to connect to your server.

Lear more in Getting started with the DaVinci client.

Configure DaVinci client properties

Applies to:

- ✓ DaVinci client for Android
- ✓ DaVinci client for iOS
- ✓ DaVinci client for JavaScript

You need to configure certain settings so that the DaVinci client can connect to your PingOne instance to step through your DaVinci flows and authenticate your users.

The method you use to configure these settings depends on which platform you are using.

Android

Configure DaVinci client for Android properties

iOS

Configure DaVinci client for iOS properties

JavaScript

Configure DaVinci client for JavaScript properties

Configure DaVinci client for Android properties

Applies to:

- ✓ DaVinci client for Android
- ✕ DaVinci client for iOS
- ✕ DaVinci client for JavaScript

Configure DaVinci client for Android properties to connect to PingOne and step through an associated DaVinci flow.

Create an instance of the `DaVinci` object and use the underlying `Oidc` module to set configuration properties.

The following properties are available for configuring the DaVinci client for Android:

Properties

Property	Description	Required?
`discoveryEndpoint`	Your PingOne server's `.well-known/openid-configuration` endpoint. *Example*: `https://auth.pingone.com/3072206d-c6ce-ch15-m0nd-f87e972c7cc3/as/.well-known/openid-configuration`	Yes
`clientId`	The `client_id` of the OAuth 2.0 client profile to use. For example, `6c7eb89a-66e9-ab12-cd34-eeaf795650b2`	Yes
`scopes`	A set of scopes to request when performing an OAuth 2.0 authorization flow. For example, `"openid", "profile", "email", "address", "revoke"`.	Yes
`redirectUri`	The `redirect_uri` as configured in the OAuth 2.0 client profile. **Note** This value must match a value configured in your OAuth 2.0 client. For example, `org.forgerock.demo://oauth2redirect`.	Yes
`timeout`	A timeout, in seconds, for each request that communicates with the server. Default is `30` seconds.	No
`acrValues`	Request which flow the PingOne server uses by adding an Authentication Context Class Reference (ACR) parameter. Enter a single DaVinci policy by using its flow policy ID. Example: `"d1210a6b0b2665dbaa5b652221badba2"`	No
`additionalParameters`	Add additional key-pair parameters as query strings to the initial OAuth 2.0 call to the `/authorize` endpoint. For example, `additionalParameters = mapOf("customKey" to "customValue")` **Tip** You can access these additional OAuth 2.0 parameters in your DaVinci flows by using the `authorizationRequest.<customParameter>` property. Learn more in Referencing PingOne data in the flow.	No

Example

The following shows an example DaVinci client configuration, using the underlying `Oidc` module:

Configure DaVinci client connection properties

```
import com.pingidentity.davinci.DaVinci
import com.pingidentity.davinci.module.Oidc

val daVinci = DaVinci {
    module(Oidc) {
        clientId = "6c7eb89a-66e9-ab12-cd34-eeaf795650b2"
        discoveryEndpoint = "https://auth.pingone.com/3072206d-c6ce-ch15-m0nd-f87e972c7cc3/" +
            "as/.well-known/openid-configuration"
        scopes = mutableSetOf("openid", "profile", "email", "address", "revoke")
        redirectUri = "org.forgerock.demo://oauth2redirect"
        additionalParameters = mapOf("customKey" to "customValue")
    }
}
```

Configure DaVinci client for iOS properties

Applies to:

- ✕ DaVinci client for Android
- ✓ DaVinci client for iOS
- ✕ DaVinci client for JavaScript

Configure DaVinci client module for iOS properties to connect to PingOne and step through an associated DaVinci flow.

Create an instance of the `DaVinci` class by passing configuration to the `createDaVinci` method. This uses the underlying `Oidc` module to set configuration properties.

The following properties are available for configuring the DaVinci client for iOS:

Properties

Property	Description	Required?
`discoveryEndpoint`	Your PingOne server's `.well-known/openid-configuration` endpoint. *Example*: `https://auth.pingone.com/3072206d-c6ce-ch15-m0nd-f87e972c7cc3/as/.well-known/openid-configuration`	Yes
`clientId`	The `client_id` of the OAuth 2.0 client profile to use. For example, `6c7eb89a-66e9-ab12-cd34-eeaf795650b2`	Yes

Property	Description	Required?
`scopes`	A set of scopes to request when performing an OAuth 2.0 authorization flow. For example, `"openid", "profile", "email", "address", "revoke"`.	Yes
`redirectUri`	The `redirect_uri` as configured in the OAuth 2.0 client profile. **Note** This value must match a value configured in your OAuth 2.0 client. For example, `org.forgerock.demo://oauth2redirect`.	Yes
`timeout`	A timeout, in seconds, for each request that communicates with the server. Default is `30` seconds.	No
`acrValues`	Request which flow the PingOne server uses by adding an Authentication Context Class Reference (ACR) parameter. Enter a single DaVinci policy by using its flow policy ID. Example: `"d1210a6b0b2665dbaa5b652221badba2"`	No
`additionalParameters`	Add additional key-pair parameters as query strings to the initial OAuth 2.0 call to the `/authorize` endpoint. For example, `myConfig.additionalParameters = ["customKey":"customValue"]` **Tip** You can access these additional OAuth 2.0 parameters in your DaVinci flows by using the `authorizationRequest.<customParameter>` property. Learn more in Referencing PingOne data in the flow.	No

Example

The following shows an example DaVinci client configuration, using the underlying `Oidc` module:

Configure DaVinci client connection properties

```
let daVinci = DaVinci.createDaVinci { config in
    // Oidc as module
    config.module(OidcModule.config) { oidcValue in
        oidcValue.clientId = "6c7eb89a-66e9-ab12-cd34-eeaf795650b2"
        oidcValue.discoveryEndpoint = "https://auth.pingone.com/3072206d-c6ce-ch15-m0nd-f87e972c7cc3/as/.well-known/
openid-configuration"
        oidcValue.scopes = ["openid", "profile", "email", "address", "revoke"]
        oidcValue.redirectUri = "org.forgerock.demo://oauth2redirect"
        oidcValue.additionalParameters = ["customKey":"customValue"]
    }
}
```

Configure DaVinci client for JavaScript properties

Applies to:

- ✕ DaVinci client for Android
- ✕ DaVinci client for iOS
- ✓ DaVinci client for JavaScript

Configure DaVinci client properties to connect to PingOne and step through an associated DaVinci flow.

Pass a `config` object into `davinci` to initialize a client with the required connection properties.

The following properties are available for configuring the DaVinci client for JavaScript:

Properties

Property	Description	Required?
`serverConfig`	An interface for configuring how the SDK contacts the PingAM instance. Contains `wellknown` and `timeout`.	Yes
`serverConfig: {wellknown}`	Your PingOne server's `.well-known/openid-configuration` endpoint. *Example*: `https://auth.pingone.com/3072206d-c6ce-ch15-m0nd-f87e972c7cc3/as/.well-known/openid-configuration`	Yes
`serverConfig: {timeout}`	A timeout, in milliseconds, for each request that communicates with your server. For example, for 30 seconds specify `30000`. Defaults to `5000` (5 seconds).	No

Property	Description	Required?
`clientId`	The `client_id` of the OAuth 2.0 client profile to use. For example, `6c7eb89a-66e9-ab12-cd34-eeaf795650b2`	Yes
`scope`	A list of scopes to request when performing an OAuth 2.0 authorization flow, separated by spaces. For example, `openid profile email phone`.	No
`responseType`	The type of OAuth 2.0 flow to use, either `code` or `token`. Defaults to `code`.	No

Example

The following shows a full DaVinci client configuration:

Configure DaVinci client connection properties

```
import { davinci } from '@forgerock/davinci';

const davinciClient = await davinci({
  config: {
    clientId: '6c7eb89a-66e9-ab12-cd34-eeaf795650b2',
    serverConfig: {
      wellknown: 'https://auth.pingone.com/3072206d-c6ce-ch15-m0nd-f87e972c7cc3/as/.well-known/openid-configuration',
      timeout: 3000,
    },
    scope: 'openid profile email phone',
    responseType: 'code',
  },
});
```

Localizing the user interface

Applies to:

- ✓ DaVinci client for Android
- ✓ DaVinci client for iOS
- ✓ DaVinci client for JavaScript

You can leverage the languages feature in PingOne to localize your client applications for different audiences.

The DaVinci clients automatically send the preferred languages configured in the browser or mobile device to PingOne so that it can return the appropriate language.

Console output from an iOS client showing `Accept-Language` header

```
[Ping SDK 1.1.0] ⬆
Request URL: https://auth.pingone.com/c2a6...1602/as/authorize?response_mode=pi.flow&client_id=85ff...
6791&response_type=code&scope=openid&redirect_uri=http://localhost:
5829&code_challenge=m8BD...rhPM&code_challenge_method=S256
Request Method: GET
Request Headers: [
    "x-requested-platform": "ios",
    "Content-Type": "application/json",
    "x-requested-with": "ping-sdk",
    "Accept-Language": "en-GB, en;q=0.9"
]
Request Timeout: 15.0
```

You can also override the configured settings directly in your code if required.

Before you begin

You must configure PingOne to support multiple languages that your client apps can use:

1. In PingOne, enable the built-in languages you want to support.

   Learn more in Enabling or disabling a language.

   > **Tip**
   >
   > You can also add your own languages and regions.
   > Learn more in Adding a language.

2. Ensure your language has the required localized strings for your clients to use.

   Learn more in Modifying translatable keys.

   > **Tip**
   >
   > You can also add your own keys to a language for use in your client applications.
   > Learn more in Adding a custom key for DaVinci.

3. Add your localized strings to your chosen implementation:

   ### *PingOne forms*

   Update the fields in your PingOne forms to use translatable keys.

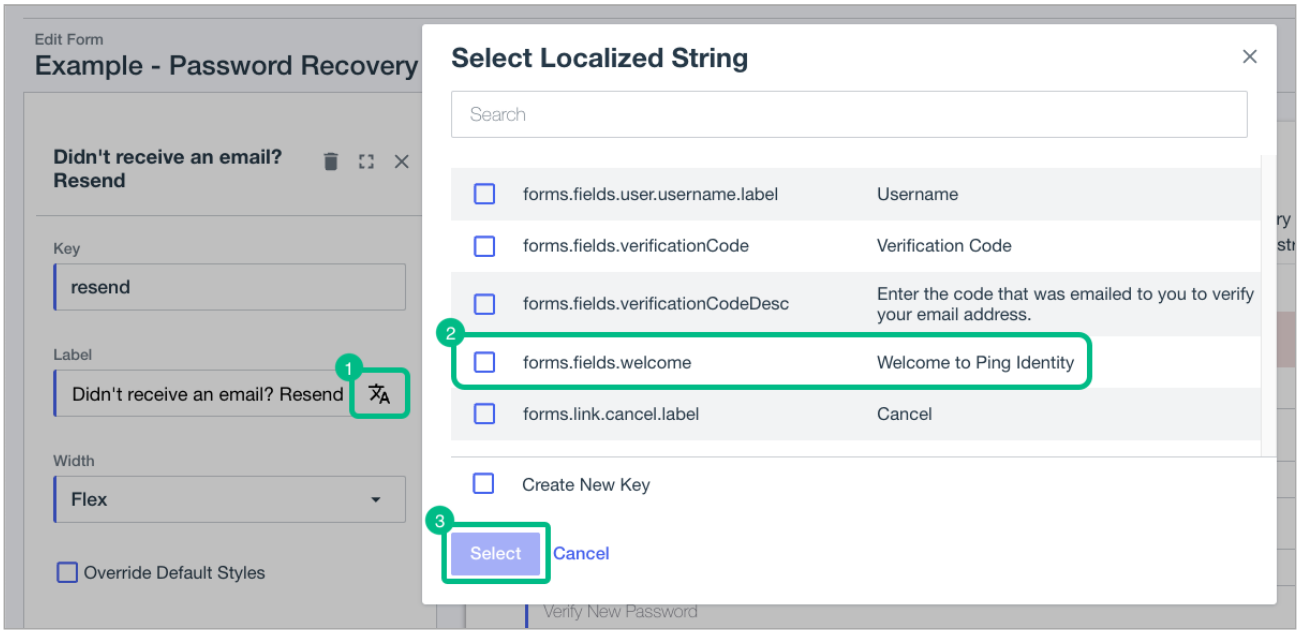


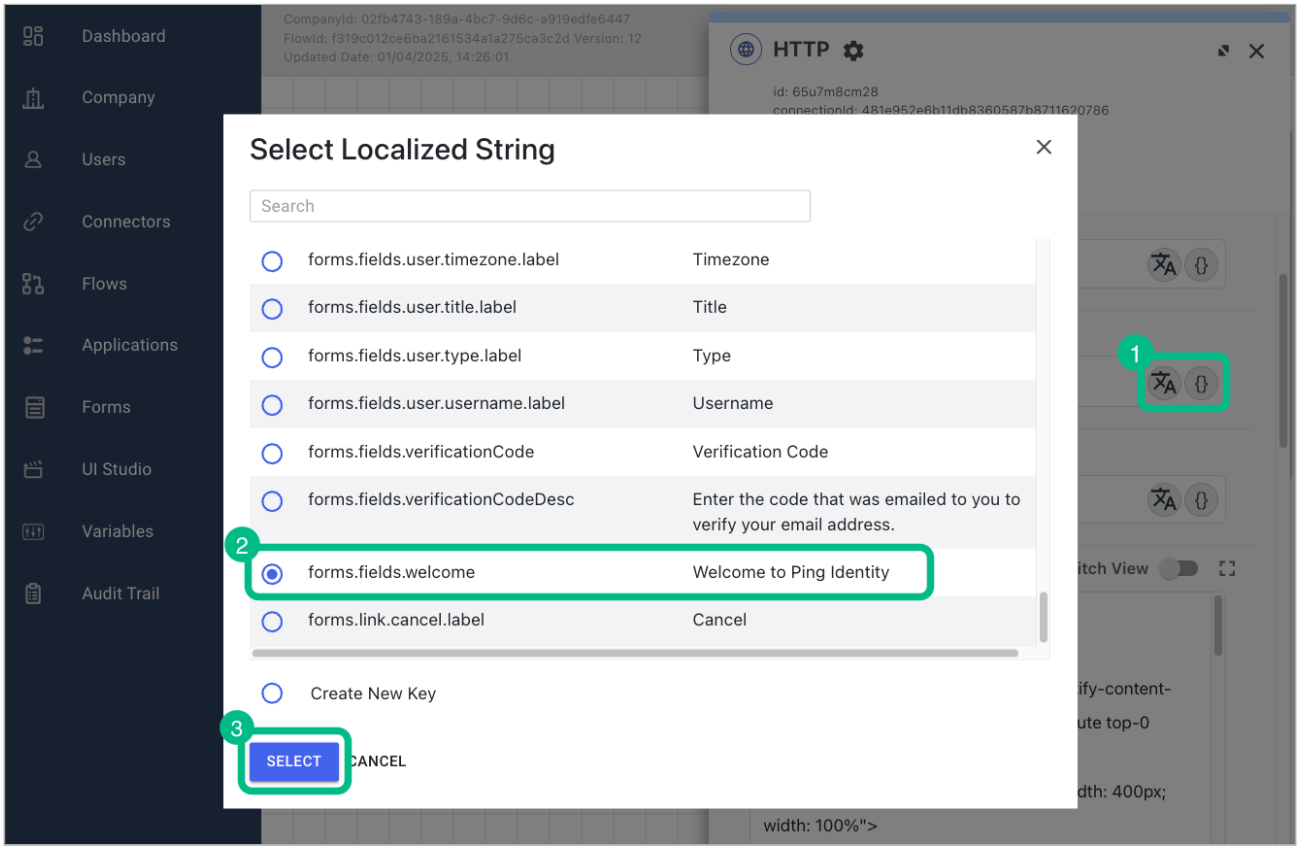
Figure 1. Adding localized strings to a PingOne form

Learn more in Using translatable keys.

Custom HTTP Connector

Update the **Output Fields** in your custom HTML to use your localized strings.

Adding localized strings to a custom HTTP connector



Learn more in HTTP Connector.

Configuring a DaVinci client to send the language header

The DaVinci clients automatically send the `Accept-Language` header when making requests to the PingOne server. This header includes each of the languages configured on the client device or in the browser, and maintains the order of preference.

The DaVinci client for Android and iOS also add generic versions of sub-dialects configured on the device, to make it more likely that PingOne can fall back to a similar language if the specific sub-dialect is unavailable.

For example, configuring **English (British)** (`en-GB`) as a preferred languages causes the DaVinci client to also send **English** (`en`) as a fallback option:

```
"Accept-Language": "en-GB, en;q=0.9"
```

Overriding the automatically-added languages

You can override the default behavior of automatically sending configured languages.

DaVinci client for Android

To provide your own values for the `Accept-Language` header, use the `CustomHeader` module.

Add the module to your `DaVinci` configuration as follows:

Using the `CustomHeader` module to override default language behavior

```
import com.pingidentity.davinci.DaVinci
import com.pingidentity.davinci.module.Oidc
import com.pingidentity.davinci.module.CustomHeader

val daVinci = DaVinci {
    module(Oidc) {
        clientId = "6c7eb89a-66e9-ab12-cd34-eeaf795650b2"
        discoveryEndpoint = "https://auth.pingone.com/3072206d-c6ce-ch15-m0nd-f87e972c7cc3/as/.well-known/
openid-configuration"
        scopes = mutableSetOf("openid", "profile", "email", "address", "revoke")
        redirectUri = "org.forgerock.demo://oauth2redirect"
        additionalParameters = mapOf("customKey" to "customValue")
    }

    // Add French as the preferred language, before default options
    module(CustomHeader, priority = 5, mode = OverrideMode.APPEND) {
        header(name = "Accept-Language", value = "fr")
    }
}
```

`priority`

Default behavior of the DaVinci client is provided by a number of built-in modules. These modules all run with a `priority` value of `10`.

- To run your module before the default modules ensure your module has a `priority` value less than the default of `10`.
- To run your module after the default modules, set the `prioriy` value to greater than `10`.

`mode`

You can choose how the `CustomHeader` module applies the modification by using the `mode` parameter:

`OverrideMode.APPEND`

The DaVinci client combines any additional parameters you provide with any parameters the default behavior adds.

The order is determined by the `priority` order of the modules.

`OverrideMode.OVERRIDE`

Any additional parameters you provide replace any parameters the default behavior would have added.

DaVinci client for iOS

To provide your own values for the `Accept-Language` header, use the `CustomHeader` module.

Add the module to your `DaVinci` configuration as follows:

Using the `CustomHeader` module to override default language behavior

```
import PingDavinci

public let davinci = DaVinci.createDaVinci { config in
    let currentConfig = ConfigurationManager.shared.currentConfigurationViewModel
    config.module(OidcModule.config) { oidcValue in
        oidcValue.clientId = "6c7eb89a-66e9-ab12-cd34-eeaf795650b2"
        oidcValue.scopes = ["openid", "profile", "email", "address", "revoke"]
        oidcValue.redirectUri = "org.forgerock.demo://oauth2redirect"
        oidcValue.discoveryEndpoint = "https://auth.pingone.com/3072206d-c6ce-ch15-m0nd-f87e972c7cc3/
as/.well-known/openid-configuration"
    }

    // Add French as the preferred language, before default options
    config.module(CustomHeader.config, priority: 5, mode: .append) { customHeaderValue in
        customHeaderValue.header(name: "Accept-Language", value: "fr")
    }
}
```

`priority`

Default behavior of the DaVinci client is provided by a number of built-in modules. These modules all run with a `priority` value of `10`.

- To run your module before the default modules ensure your module has a `priority` value less than the default of `10`.
- To run your module after the default modules, set the `prioriy` value to greater than `10`.

`mode`

You can choose how the `CustomHeader` module applies the modification by using the `mode` parameter:

`.append`

The DaVinci client combines any additional parameters you provide with any parameters the default behavior adds.

The order is determined by the `priority` order of the modules.

`.override`

Any additional parameters you provide replace any parameters the default behavior would have added.

DaVinci client for JavaScript

To override the default browser behavior and provide your own values for the `Accept-Language` header, use the `RequestMiddleware` type.

Call your request middleware when creating the DaVinci client as follows:

Using the `CustomHeader` module to override default language behavior

```
import { davinci } from '@forgerock/davinci';
import type { RequestMiddleware } from '@forgerock/davinci-client/types';

const requestMiddleware: RequestMiddleware[] = [
  (fetchArgs, action, next) => {
    fetchArgs.headers?.set('Accept-Language', 'fr-FR, fr;q=0.9');
    next();
  },
];

const davinciClient = await davinci({
  config: {
    clientId: '6c7eb89a-66e9-ab12-cd34-eeaf795650b2',
    serverConfig: {
      wellknown: 'https://auth.pingone.com/3072206d-c6ce-ch15-m0nd-f87e972c7cc3/as/.well-known/openid-
configuration',
      timeout: 3000,
    },
    scope: 'openid profile email phone',
    responseType: 'code',
  },
  requestMiddleware
});
```

Ping SDK for PingOne DaVinci tutorials

These tutorials teach you how to connect your apps to a PingOne tenant to authenticate a user using a DaVinci flow, such as the PingOne sign-on with sessions flow.

This flow allows users to register, authenticate, and verify their email address with PingOne. It combines the PingOne Sign On and Password Reset flow, which allows users to reset or recover their passwords, with the PingOne Registration and Email Verification flow, which allows users to register and verify their email. In addition, this flow checks for an active user session before prompting for authentication.

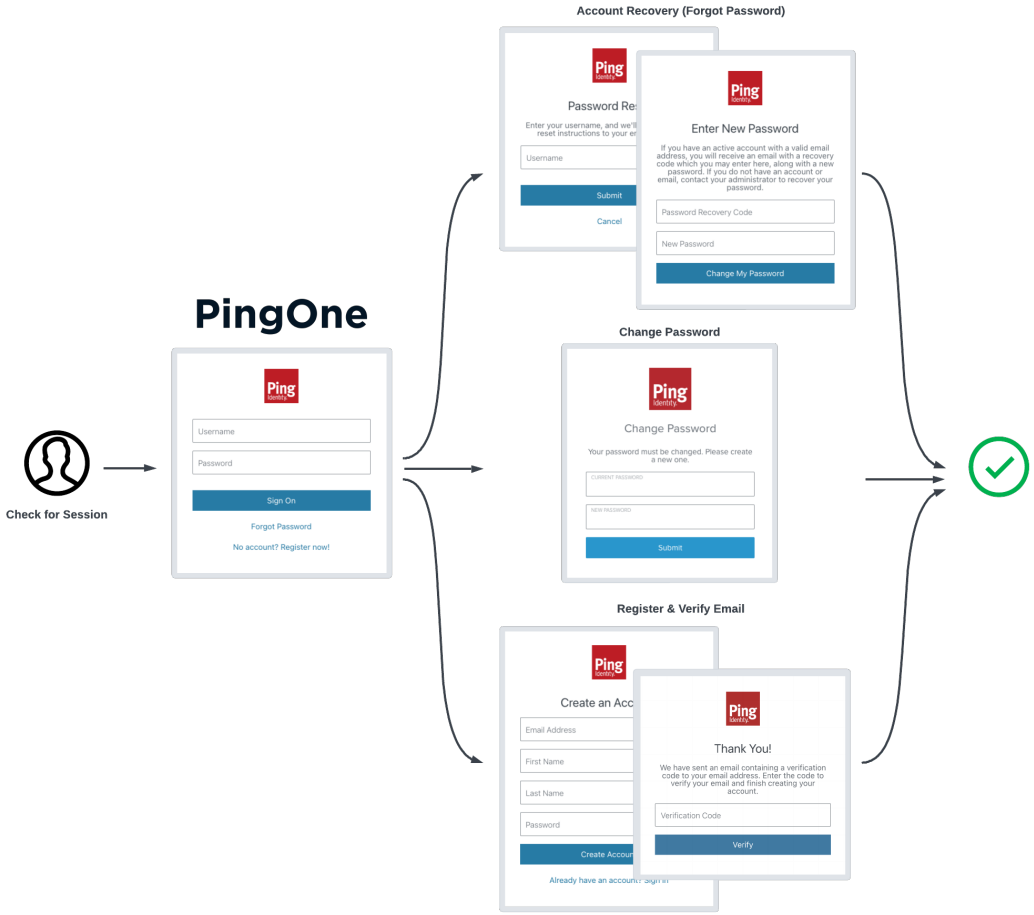


Figure 1. Overview of the PingOne sign-on with sessions flow

The tutorials also support DaVinci flows that use the **PingOne Forms connector**.

> **Tip**
>
> Each tutorial provides a list of the supported capabilities and fields.
> Check that your flows use only supported fields before attempting to use them with the Ping SDKs.

The tutorials cover topics such as:

- Configuring an app with the connection settings for your PingOne instance.
- Starting a DaVinci flow
- Rendering UI depending on the type of node encountered in the flow

- Returning responses to the server for different nodes encountered
- Getting an access token for the user on completion of the flow
- Getting the user's details from PingOne, such as their full name and email address
- Signing the user out of PingOne

To start, choose the platform to host your app:

Android

Android DaVinci flow login tutorials

iOS

iOS DaVinci flow login tutorials

JavaScript

JavaScript DaVinci flow login tutorials

DaVinci Client for Android tutorials

Follow these tutorials integrate your Android apps with DaVinci flows in your PingOne instance.

DaVinci Client for Android Tutorials

Quick start

In this quick start tutorial you update one of our sample applications.

The app steps through a DaVinci flow and displays a basic prototype UI to gather user credentials.

Deep dive

This deep dive tutorial covers the steps to take to integrate an Android app with a DaVinci flow in PingOne.

You'll learn about installing and importing the DaVinci Client, stepping through DaVinci flows, and how to leverage Jetpack Compose to help you integrate DaVinci flows into your Android apps.

DaVinci client quick start for Android

Prepare › Download › Configure › Run

This tutorial walks you through updating a provided sample app so that it connects to a PingOne tenant to authenticate a user using the PingOne sign-on with sessions DaVinci flow.

This flow allows users to register, authenticate, and verify their email address with PingOne.

Before you begin

Before you begin this tutorial ensure you have set up your PingOne server with the required configuration.

For example, you will need to have an OAuth 2.0 client application set up.

Complete prerequisites »

Step 1. Download the samples

To start this tutorial, you need to download the SDK sample apps repo, which contains the projects you will use.

Start step 1 »

Step 2. Configure the sample app

In this step, you configure the sample app to connect to the OAuth 2.0 application you created in DaVinci.

Start step 2 »

Step 3. Test the app

To test the app, run the sample that you configured in the previous step.

The sample connects to your PingOne server to obtain the correct URIs to authenticate the user, and redirects the browser to your PingOne server.

After authentication, PingOne redirects the browser back to your application, which then obtains an OAuth 2.0 access token and displays the related user information.

Test app »

Before you begin

Prepare › Download › Configure › Run

To successfully complete this tutorial refer to the prerequisites in this section.

The tutorial also requires a configured PingOne instance.

Compatibility

PingOne

- Your PingOne instance must have DaVinci enabled.

DaVinci flows

Ensure your flows only use supported connectors, capabilities and fields for user interactions:

- **HTTP Connector**
  - **Custom HTML** capability
    - HTTP Connector field and collector support
    - HTTP Connector SK-Component support

HTTP Connector field and collector support

Field (`Collector`)	Description	DaVinci module		
		Android	iOS	JavaScript
Text field (`TextCollector`)	Collects a single text string.	☑ `1.0.0`	☑ `1.0.0`	☑ `1.0.0`
Password field (`PasswordCollector`)	Collects a single text string that cannot be read from the screen.	☑ `1.0.0`	☑ `1.0.0`	☑ `1.0.0`
Submit Button (`SubmitCollector`)	Sends the collected data to PingOne to continue the DaVinci flow.	☑ `1.0.0`	☑ `1.0.0`	☑ `1.0.0`
Flow Button (`FlowCollector`)	Triggers an alternative flow without sending the data collected so far to PingOne.	☑ `1.0.0`	☑ `1.0.0`	☑ `1.0.0`
Label (`LabelCollector`)	Display a read-only text label.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`
Radio / Dropdown (`SingleSelectCollector`)	Collects a single value from a choice of multiple options.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`

HTTP Connector SK-Component support

SK-Component (`Collector`)	Description	DaVinci module		
		Android	iOS	JavaScript

skIDP (`IdpCollector`)	Presents a button to allow users to authenticate using an external identity provider, such as Apple, Facebook, or Google.	☑ 1.1.0	☑ 1.1.0	☑ 1.1.0

Verify that your flow does not depend on any *unsupported* elements:

SKPolling components

The SKPolling component cannot be processed by the DaVinci Client and should not be included in flows.

Features such as Magic Link authentication require the **SKPolling** component and therefore cannot be used with the DaVinci Client.

Images

Images included in the flow cannot be passed to the SDK.

For example, the PingOne sign-on with sessions DaVinci flow.

- **PingOne Form Connector**
  - **Show Form** capability
    - Custom Fields support
    - Toolbox support

Custom Fields support

Field (`Collector`)	Description	DaVinci module		
		Android	iOS	JavaScript
Text Input (`TextCollector`)	Collects a single text string.	☑ 1.1.0	☑ 1.1.0	☑ 1.1.0
Password (`PasswordCollector`)	Collects a single text string that cannot be read from the screen.	☑ 1.1.0	☑ 1.1.0	☑ 1.1.0
Dropdown (`SingleSelectCollector`)	Collects a value from a dropdown containing one or more text strings.	☑ 1.1.0	☑ 1.1.0	☑ 1.1.0
Combobox (`MultiSelectCollector`)	Collects a value from a dropdown containing one or more text strings, the user can enter their own text string.	☑ 1.1.0	☑ 1.1.0	☑ 1.1.0

Radio Button List (`SingleSelectCollector`)	Collects a value from one or radio buttons.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`
Checkbox List (`MultiSelectCollector`)	Collects the value of one or more checkboxes.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`

Toolbox support

Field (`Collector`)	Description	DaVinci module		
		Android	iOS	JavaScript
Flow Button (`FlowCollector`)	Presents a customized button.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`
Flow Link (`FlowCollector`)	Presents a customized link.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`
Translatable Rich Text (`TextCollector`)	Presents rich text that you can translate into multiple languages.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`
Social Login (`IdpCollector`)	Presents a button to allow users to authenticate using an external identity provider, such as Apple, Facebook, or Google.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`

Android

This sample requires at least Android API 23 (Android 6.0)

Java

This sample requires at least Java 8 (v1.8).

Prerequisites

Android Studio

Download and install Android Studio, which is available for many popular operating systems.

An Android emulator or physical device

To try the quick start application as you develop it, you need an Android device. To add a virtual, emulated Android device to Android Studio, refer to Create and manage virtual devices, on the **Android Developers** website.

Server configuration

You must configure your PingOne instance for use with the DaVinci client.

Ask your PingOne administrator to complete the following tasks:

- Configure a DaVinci flow
- Create a DaVinci application
- Configure PingOne for DaVinci flow invocation

To learn how to complete these steps, refer to Launching a flow with a Ping SDK in the *PingOne DaVinci documentation*.

Step 1. Download the samples

Prepare › **Download** › Configure › Run

To start this tutorial, you need to download the ForgeRock SDK sample apps repo, which contains the projects you will use.

1. In a web browser, navigate to the SDK Sample Apps repository.
2. Download the source code using one of the following methods:

Download a ZIP file

1. Click **Code**, and then click **Download ZIP**.
2. Extract the contents of the downloaded ZIP file to a suitable location.

Use a Git-compatible tool to clone the repo locally

1. Click **Code**, and then copy the HTTPS URL.
2. Use the URL to clone the repository to a suitable location.

   For example, from the command-line you could run:

```
git clone https://github.com/ForgeRock/sdk-sample-apps.git.git
```

The result of these steps is a local folder named `sdk-sample-apps`.

Step 2. Configure the sample app

Prepare › Download › **Configure** › Run

In this section you open the sample project in Android Studio, and view the integration points in the **TODO** pane.

You'll visit each integration point in the sample app to understand how to complete a DaVinci flow, including handling the different nodes and their collectors, obtaining an access token and user information, and finally signing out of the session.

1. In *Android Studio*, click **Open**, navigate to the `sdk-sample-apps/android/kotlin-davinci` folder that has the downloaded sample source code in, and then click **Open**.

   Android Studio opens and loads the DaVinci tutorial project.

2. In the **Project** pane, navigate to **samples** > **app**.

3. On the **View** menu, select **Tool Windows**, and then click **TODO**.

   The sample code is annotated with `TODO` comments to help you locate the integration points, where code changes are required.

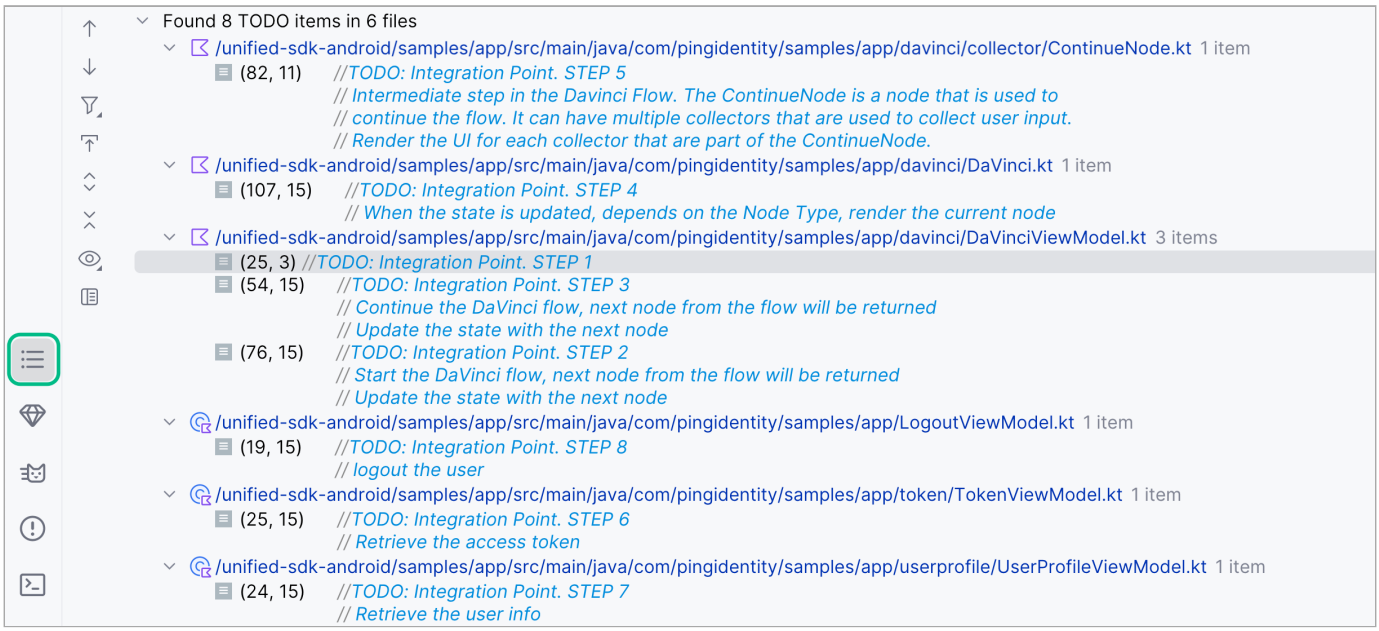


Figure 1. Integration points annotated with `TODO` comments

4. In the **TODO** pane, double-click the **STEP 1** line.

   Android Studio opens `DaVinciViewModel.kt`:

DaVinciViewModel.kt

```
//TODO: Integration Point. STEP 1
val daVinci = DaVinci {
    logger = Logger.STANDARD

    // Oidc as module
    module(Oidc) {
        clientId = "<Client ID>"
        discoveryEndpoint = "<Discovery Endpoint>"
        scopes = mutableSetOf("<scope1>", "<scope2>", "…")
        redirectUri = "<Redirect URI>"
    }
}
```

This snippet initializes the `DaVinci` module, and leverages the OpenID Connect (OIDC) module to configure the settings to connect to your PingOne instance.

1. Replace *<Client ID>* with the ID of the client you are connecting to in PingOne.

   Example:

   ```
   clientId = "6c7eb89a-66e9-ab12-cd34-eeaf795650b2"
   ```

   Refer to Get configuration values from PingOne for instructions of where to find this value.

2. Replace *<Discovery Endpoint>* with the OIDC Discovery Endpoint value from the client you are connecting to in PingOne.

   Example:

   ```
   discoveryEndpoint = "https://auth.pingone.ca/3072206d-c6ce-ch15-m0nd-f87e972c7cc3/as/.well-known/openid-configuration"
   ```

   Refer to Get configuration values from PingOne for instructions of where to find this value.

3. In the `scopes` property, add the scopes you want to assign users who complete authentication using the client.

   Example:

   ```
   scopes = mutableSetOf("openid", "email", "profile")
   ```

4. Replace *<Redirect URI>* with the application ID of your sample app, followed by `://oauth2redirect`.

   Example:

   ```
   redirectUri = "org.forgerock.demo://oauth2redirect"
   ```

   > **Important**
   >
   > The `redirectUri` value you use must exactly match one of the **Redirect URIs** value you enter in the native OAuth 2.0 application you created earlier.

5. Optionally, delete the `TODO` comment to remove it from the list.

The result resembles the following:

DaVinciViewModel.kt

```
val daVinci = DaVinci {
    logger = Logger.STANDARD

    // Oidc as module
    module(Oidc) {
        clientId = "6c7eb89a-66e9-ab12-cd34-eeaf795650b2"
        discoveryEndpoint = "https://auth.pingone.ca/3072206d-c6ce-ch15-m0nd-f87e972c7cc3/as/.well-known/
openid-configuration"
        scopes = mutableSetOf("openid", "email", "profile")
        redirectUri = "org.forgerock.demo://oauth2redirect"
    }
}
```

5. In the **TODO** pane, double-click the **STEP 2** line.

   Android Studio opens `DaVinciViewModel.kt`:

DaVinciViewModel.kt

```
//TODO: Integration Point. STEP 2
// Start the DaVinci flow, next node from the flow will be returned
// Update the state with the next node

/*
val next = daVinci.start()

state.update {
    it.copy(prev = next, node = next)
}
*/
```

   This snippet calls `start()` to start the DaVinci flow, and assigns the returned node to the variable `next`.

   It also updates the app's state to store the response as both `prev` and `node`.

   1. Uncomment the highlighted text.
   2. Optionally, delete the `TODO` comment to remove it from the list.

6. In the **TODO** pane, double-click the **STEP 3** line.

   Android Studio opens `DaVinciViewModel.kt`:

DaVinciViewModel.kt

```
//TODO: Integration Point. STEP 3
// Continue the DaVinci flow, next node from the flow will be returned
// Update the state with the next node

/*
val next = current.next()

state.update {
    it.copy(prev = current, node = next)
}
*/
```

This snippet calls `next()` to *continue* the DaVinci flow, by proceeding to the next available node. It assigns the newly returned node to the variable `next`.

It also updates the app's state to store the new response as `node`, and the *current* node as `prev`.

1. Uncomment the highlighted text.
2. Optionally, delete the `TODO` comment to remove it from the list.

7. In the **TODO** pane, double-click the **STEP 4** line.

   Android Studio opens `DaVinci.kt`:

DaVinci.kt

```
//TODO: Integration Point. STEP 4
// Render the current node depending on its type

/*
when (val node = state.node) {
    is ContinueNode → {
        Render(node = node, onNodeUpdated, onStart) { onNext(node) }
    }
    is FailureNode → {
        Log.e("DaVinci", node.cause.message, node.cause)
        Render(node = node)
    }
    is ErrorNode → {
        Render(node)
        // Render the previous node
        if (state.prev is ContinueNode) {
            Render(node = state.prev, onNodeUpdated, onStart) { onNext(state.prev) }
        }
    }
    is SuccessNode → {
        LaunchedEffect(true) { onSuccess?.let { onSuccess() } }
    }
    else → {}
}
*/
```

This snippet watches for a change in state, and takes an action based on the ode type returned by DaVinci.

For example, if it is a `FailureNode` log an error message. If the node is a `ContinueNode` that continues the flow, call the render function to display the necessary fields on the screen.

1. Uncomment the highlighted text.
2. Optionally, delete the `TODO` comment to remove it from the list.

8. In the **TODO** pane, double-click the **STEP 5** line.

   Android Studio opens `ContinueNode.kt`:

ContinueNode.kt

```
//TODO: Integration Point. STEP 5
// Intermediate step in the Davinci Flow. The ContinueNode is a node that is used to
// continue the flow. It can have multiple collectors that are used to collect user input.
// Render the UI for each collector that are part of the ContinueNode.

/*
continueNode.collectors.forEach {
    when (it) {
        is FlowCollector -> {
            hasAction = true
            FlowButton(it, onNext)
        }

        is PasswordCollector -> {
            Password(it, onNodeUpdated)
        }
        is SubmitCollector -> {
            hasAction = true
            SubmitButton(it, onNext)
        }

        is TextCollector -> Text(it, onNodeUpdated)
    }
}
*/
```

This snippet handles the various collectors that are returned by the current node.

Loop through all of the collectors in the node and render an appropriate field in your app.

For example, this snippet handles text and password fields, and two types of button, a submit and a flow type.

1. Uncomment the highlighted text.
2. Optionally, delete the `TODO` comment to remove it from the list.

9. In the **TODO** pane, double-click the **STEP 6** line.

   Android Studio opens `TokenViewModel.kt`:

TokenViewModel.kt

```
//TODO: Integration Point. STEP 6
// Retrieve the access token

/*
User.user()?.let {
    when (val result = it.token()) {
        is Failure -> {
            state.update {
                it.copy(token = null, error = result.value)
            }
        }

        is Success -> {
            state.update {
                it.copy(token = result.value, error = null)
            }
        }
    }
} ?: run {
    state.update {
        it.copy(token = null, error = null)
    }
}
*/
```

This snippet gets called when the flow reaches a `SuccessNode` and gets an access token on behalf of the authenticated user.

1. Uncomment the highlighted text.
2. Optionally, delete the `TODO` comment to remove it from the list.

10. In the **TODO** pane, double-click the **STEP 7** line.

    Android Studio opens `UserProfileViewModel.kt`:

UserProfileViewModel.kt

```
//TODO: Integration Point. STEP 7
// Retrieve the user info

/*
User.user()?.let { user →
    when (val result = user.userinfo(false)) {
        is Result.Failure →
            state.update { s →
                s.copy(user = null, error = result.value)
            }

        is Result.Success →
            state.update { s →
                s.copy(user = result.value, error = null)
            }
    }
}
*/
```

This snippet gets the user's info from DaVinci, for example their preferred name for display within the sample app.

1. Uncomment the highlighted text.
2. Optionally, delete the `TODO` comment to remove it from the list.

11. In the **TODO** pane, double-click the **STEP 8** line.

    Android Studio opens `LogoutViewModel.kt`:

LogoutViewModel.kt

```
//TODO: Integration Point. STEP 8
// logout the user

/*
User.user()?.logout()
*/
```

This snippet calls the `logout()` function on the `User` object to sign the user out of the app, and end their session in DaVinci.

1. Uncomment the highlighted text.
2. Optionally, delete the `TODO` comment to remove it from the list.

Step 3. Test the app

Prepare › Download › Configure › Run

In the following procedure, you run the sample app that you configured in the previous step.

1. Add or connect a device to Android Studio.

   Learn more about devices in Android Studio in the Android Developer documentation:

   - Create and manage virtual devices
   - Run apps on a hardware device

2. On the **Run** menu, click **Run 'samples.app'**.

   Android Studio starts the sample application on the simulated or connected device.

   The app automatically starts the DaVinci flow:

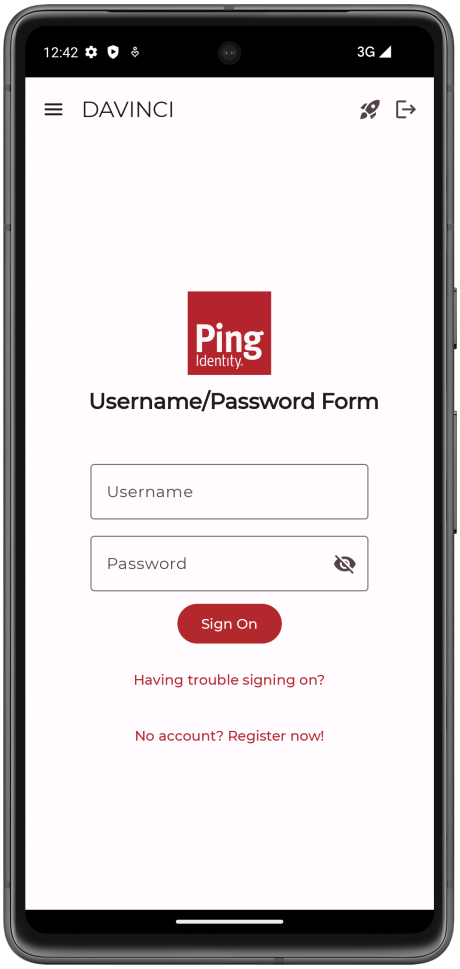


Figure 1. The DaVinci sample app first screen with fields and buttons.

3. Optionally, to register a new identity in PingOne, tap the **No Account? Register now!** link.

> **Tip**
>
> This link is an example of a `FlowButton`.

The app displays the registration screen:

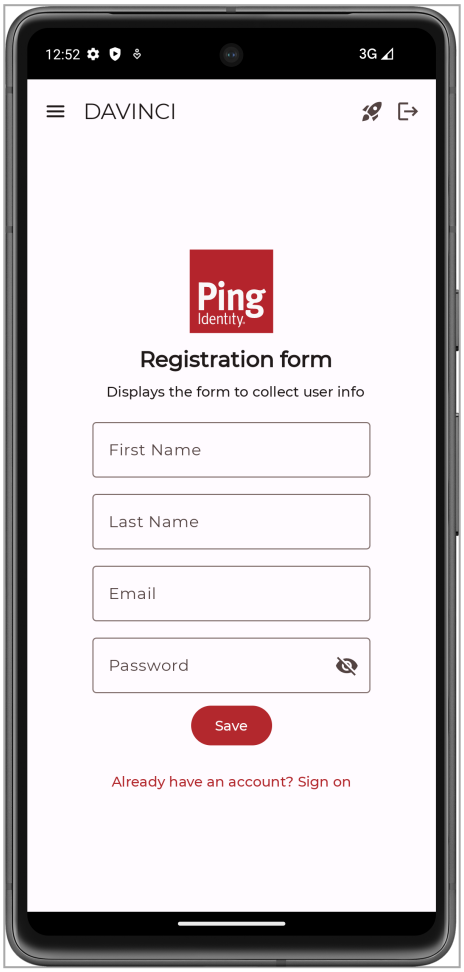


Figure 2. The DaVinci sample app registration screen.

1. Enter the details of the new identity, and then click **Save**.

   The app creates the new identity in PingOne and returns to the sign on screen.

4. Enter the username and password of a PingOne identity, and then click **Sign On**.

   The app sends the credentials to PingOne for validation, and on success displays the user's info:

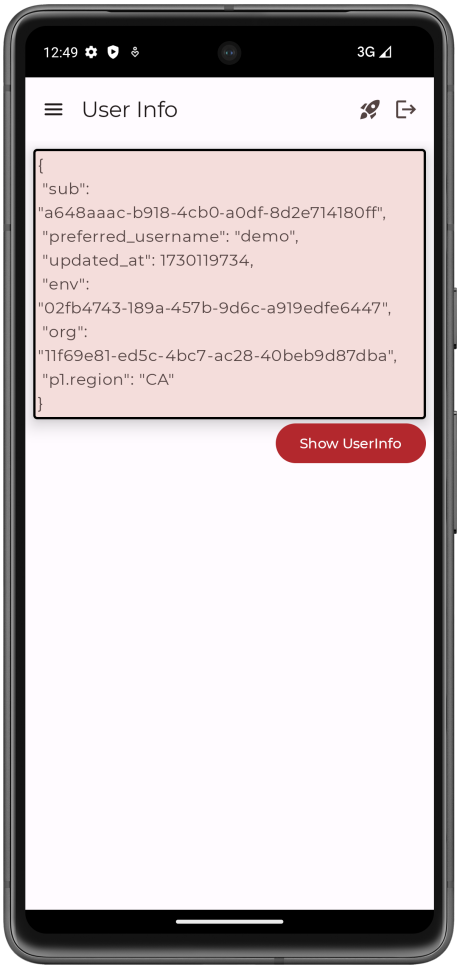


Figure 3. The DaVinci sample app displaying user info

5. Tap the menu (☰) icon, and then tap **Show Token**.

   The app shows the access token obtained on behalf of the user.

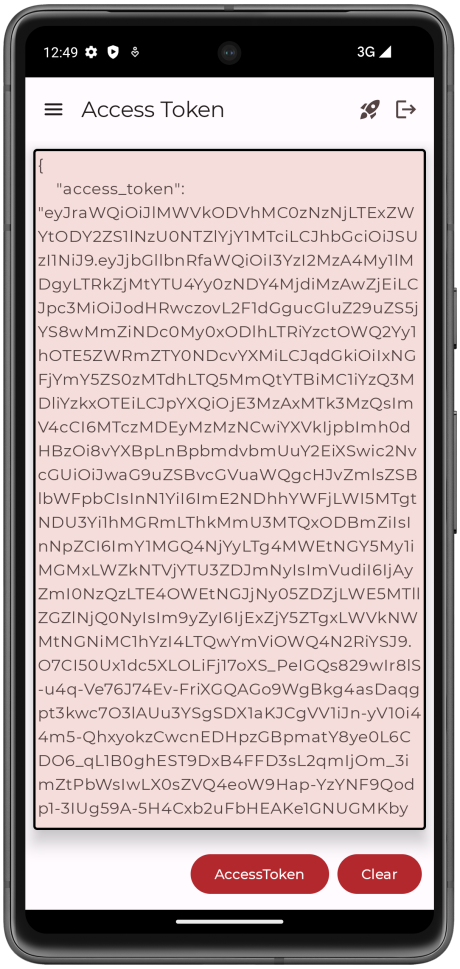


Figure 4. The DaVinci sample app displaying a user's access token

6. Tap the menu (☰) icon, and then tap **Logout**.

   The app revokes the existing tokens and ends the session in PingOne.

Troubleshooting

This section contains help if you encounter an issue running the sample code.

What can cause validation errors in the request?

When starting the app you might see an error message as follows:

> *The request could not be completed. One or more validation errors were in the request.*

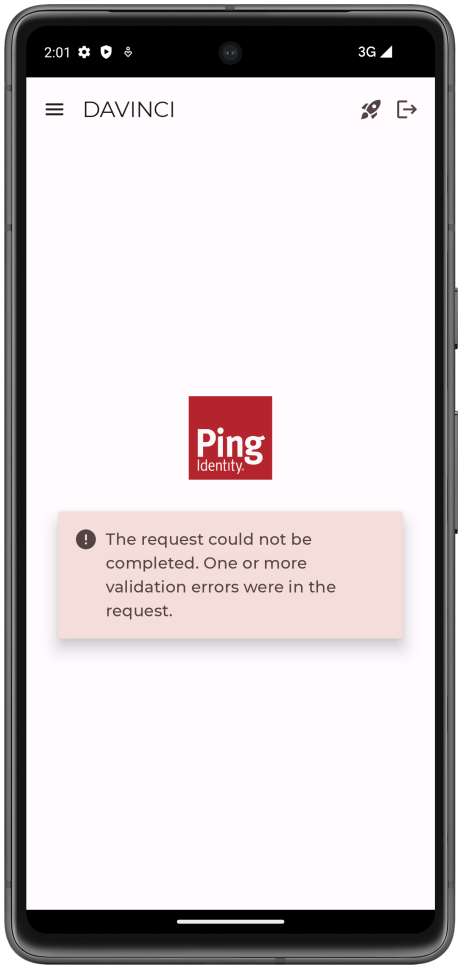


Figure 5. Validation error when starting the app

The **Logcat** pane in Android Studio often contains additional information about the error:

```
{
  "id" : "2a72bf00-5f20-4b78-a7d0-ad8d95e9b11b",
  "code" : "INVALID_DATA",
  "message" : "The request could not be completed. One or more validation errors were in the request.",
  "details" : [
    {
      "code" : "INVALID_VALUE",
      "target" : "redirect_uri",
      "message" : "Redirect URI mismatch"
    }
  ]
}
```

In this case the cause is `Redirect URI mismatch`.

Ensure that your `redirectUri` value in `com/pingidentity/samples/app/davinci/DaVinciViewModel.kt` exactly matches one of the values you entered in the **Redirect URIs** field in your OAuth 2.0 application in PingOne:

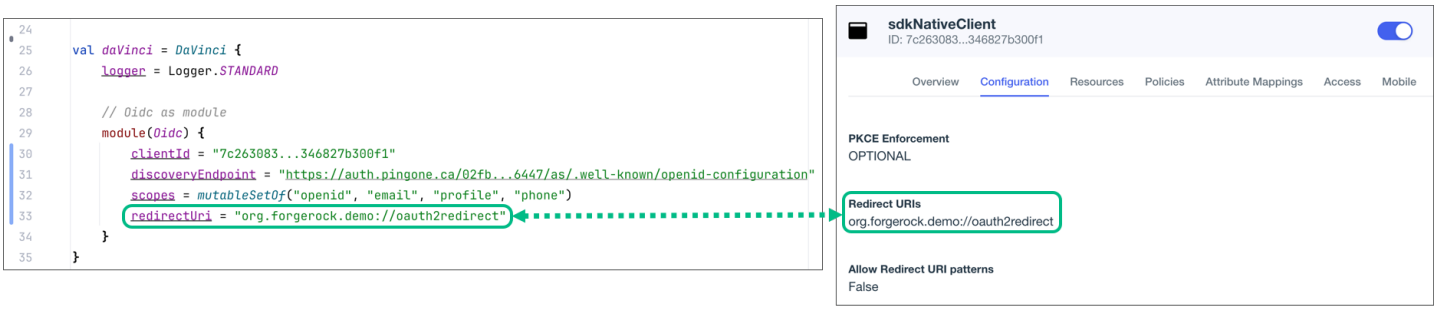


Figure 6. Match the redirect URIs in the sample app and PingOne configuration

DaVinci client deep dive for Android

Applies to:

- ✓ DaVinci client for Android
- ✕ DaVinci client for iOS
- ✕ DaVinci client for JavaScript

Configure DaVinci client for Android properties to connect to PingOne and step through an associated DaVinci flow.

Installing and importing the DaVinci client

To use the DaVinci client for Android, add the relevant dependencies to your project:

1. In the **Project** tree view of your Android Studio project, open the `Gradle Scripts/build.gradle.kts` file for the DaVinci *module*.
2. In the `dependencies` section, add the following:

```
implementation("com.pingidentity.sdks:davinci:1.1.0")
```

Example of the `dependencies` section after editing:

```
dependencies {

    val composeBom = platform(libs.androidx.compose.bom)
    implementation(composeBom)

    // DaVinci client
    implementation("com.pingidentity.sdks:davinci:1.1.0")

    ...

    implementation(libs.androidx.core.ktx)
    implementation(libs.androidx.appcompat)
    implementation(libs.material)
}
```

Configuring the DaVinci client

Configure DaVinci client for Android properties to connect to PingOne and step through an associated DaVinci flow.

The following shows an example DaVinci client configuration, using the underlying `Oidc` module:

Configure DaVinci client connection properties

```
import com.pingidentity.davinci.DaVinci
import com.pingidentity.davinci.module.Oidc

val daVinci = DaVinci {
    module(Oidc) {
        clientId = "6c7eb89a-66e9-ab12-cd34-eeaf795650b2"
        discoveryEndpoint = "https://auth.pingone.com/3072206d-c6ce-ch15-m0nd-f87e972c7cc3/" +
            "as/.well-known/openid-configuration"
        scopes = mutableSetOf("openid", "profile", "email", "address", "revoke")
        redirectUri = "org.forgerock.demo://oauth2redirect"
        additionalParameters = mapOf("customKey" to "customValue")
    }
}
```

For information on the properties available, refer to Configure DaVinci client for Android properties.

Stepping through DaVinci flows

To authenticate your users the DaVinci client for Android must start the flow, and step through each **node**.

> **Tip**
>
> For information on which connectors and fields the DaVinci client supports, refer to Compatibility.

Starting a DaVinci flow

To start a DaVinci flow, call the `start()` method:

Start a DaVinci flow

```
val node = daVinci.start()
```

Determining DaVinci flow node type

Each step of the flow returns one of four node types:

`ContinueNode`

This type indicates there is input required from the client. The `node` object for this type contains a `collector` object, which describes the information it requires from the client.

Learn more in Handling DaVinci flow collectors in continue nodes.

`SuccessNode`

This type indicates the flow is complete, and authentication was successful.

Learn more in Handling DaVinci flow success nodes.

`ErrorNode`

This type indicates an error in the data sent to the server. For example, an email address in an incorrect format, or a password that does not meet complexity requirements.

You can correct the error and resubmit to continue the flow.

Learn more in Handling DaVinci flow error nodes.

`FailureNode`

This type indicates that the flow could not be completed and must be restarted. This can be caused by a server error, or a timeout.

Learn more in Handling DaVinci flow failure nodes.

You can use the helper functions to determine which node type the server has returned:

Determine `node` type.

```
when (node) {
    is ContinueNode -> {}
    is ErrorNode -> {}
    is FailureNode -> {}
    is SuccessNode -> {}
}
```

Handling DaVinci flow collectors in continue nodes

The `ContinueNode` type contains `collectors`. These collectors define what information or action to request from the user, or client device.

> **Tip**
>
> For a list of supported collectors, refer to Supported PingOne fields and collectors.

There are specific collector types. For example there are `TextCollector` and `PasswordCollector` types.

To complete a DaVinci flow we recommend that you implement a component for each connector type you will encounter in the flow. Then you can iterate through the flow and handle each collector as you encounter it.

Access collectors in a `ContinueNode`

```
node.collectors.forEach {
    when(it) {
        is TextCollector → it.value = "My First Name"
        is PasswordCollector → it.value = "My Password"
        is SubmitCollector → it.value = "click me"
        is FlowCollector → it.value = "Forgot Password"
    }
}
```

Continuing a DaVinci flow

After collecting the data for a node you can proceed to the next node in the flow by calling the `next()` method on your current `node` object.

Continue a DaVinci flow using `next()`

```
val next = node.next()
```

> **Note**
>
> You do not need to pass any parameters into the `next` method as the DaVinci client internally stores the updated object, ready to return to the PingOne server.

The server responds with a new `node` object, just like when starting a flow initially.

Loop again through conditional checks on the new node's type to render the appropriate UI or take the appropriate action.

Handling DaVinci flow error nodes

DaVinci flows return the `ErrorNode` type when it receives data that is incorrect, but you can fix the data and resubmit. For example, an email value submitted in an invalid format or a new password that is too short.

You can retrieve the error message by using `node.message()`, and the raw JSON response with `node.input`.

Displaying the reason for an error

```
val node = daVinci.start() // Start the flow

//Determine the Node Type
when (node) {
    is ContinueNode -> {}
    is ErrorNode -> {
        node.message() // Retrieve the cause of the error
    }
    is FailureNode -> {}
    is SuccessNode -> {}
}
```

> **Note**
>
> This is different than a `FailureNode` type, which you cannot resubmit and must restart the entire flow.

You can retain a reference to the `node` you submit in case the next `node` you receive is an `ErrorNode` type. If so, you can re-render the previous form, and inject the error information from the new `ErrorNode` node.

After the user revises the data call `next()` as you did before.

Handling DaVinci flow failure nodes

DaVinci flows return the `FailureNode` type if there has been an issue that prevents the flow from continuing. For example, the flow times out or suffers a server error.

You can retrieve the cause of the failure by using `node.cause()`, which is a `Throwable` object.

Handling receipt of a `FailureNode` type

```
val node = daVinci.start() // Start the flow

//Determine the Node Type
when (node) {
    is ContinueNode -> {}
    is ErrorNode -> {}
    is FailureNode -> {
        node.cause() // Retrieve the error message
    }
    is SuccessNode -> {}
}
```

You should offer to restart the flow on receipt of a `FailureNode` type.

Handling DaVinci flow success nodes

DaVinci flows return the `SuccessNode` type when the user completes the flow and PingOne issues them a session.

To retrieve the existing session, you can use the following code:

Handling receipt of a `SuccessNode` type

```
val user: User? = daVinci.user()

user?.let {
    it.accessToken()
    it.revoke()
    it.userinfo()
    it.logout()
}
```

Leverage Jetpack Compose

The following shows how you could use the DaVinci client with Jetpack Compose:

ViewModel

```
// Define State that listen by the View
var state = MutableStateFlow<Node>(Empty)

//Start the DaVinci flow
val next = daVinci.start()

// Update the state
state.update {
    next
}

fun next(node: ContinueNode) {
    viewModelScope.launch {
        val next = node.next()
        state.update {
            next
        }
    }
}
```

View

```
when (val node = state.node) {
    is ContinueNode -> {}
    is ErrorNode -> {}
    is FailureNode -> {}
    is SuccessNode -> {}
}
```

DaVinci Client for iOS tutorials

Follow these tutorials integrate your iOS apps with DaVinci flows in your PingOne instance.

DaVinci Client for iOS Tutorials

Quick start

In this quick start tutorial you update one of our sample applications.

The app steps through a DaVinci flow and displays a basic prototype UI to gather user credentials.

Deep dive

This deep dive tutorial covers the steps to take to integrate an iOS app with a DaVinci flow in PingOne.

You'll learn about installing and importing the DaVinci Client, stepping through DaVinci flows, and how to leverage SwiftUI to help you integrate DaVinci flows into your iOS apps.

DaVinci client quick start for iOS

Prepare › Download › Configure › Run

This tutorial walks you through updating a provided sample app so that it connects to a PingOne tenant to authenticate a user using the PingOne sign-on with sessions DaVinci flow.

This flow allows users to register, authenticate, and verify their email address with PingOne.

Before you begin

Before you begin this tutorial ensure you have set up your PingOne server with the required configuration.

For example, you will need to have an OAuth 2.0 client application set up.

Complete prerequisites »

Step 1. Download the samples

To start this tutorial, you need to download the SDK sample apps repo, which contains the projects you will use.

Start step 1 »

Step 2. Configure the sample app

In this step, you configure the sample app to connect to the OAuth 2.0 application you created in DaVinci.

Start step 2 »

Step 3. Test the app

To test the app, run the sample that you configured in the previous step.

The sample connects to your PingOne server to obtain the correct URIs to authenticate the user, and redirects the browser to your PingOne server.

After authentication, PingOne redirects the browser back to your application, which then obtains an OAuth 2.0 access token and displays the related user information.

Test app »

Before you begin

Prepare › Download › Configure › Run

To successfully complete this tutorial refer to the prerequisites in this section.

The tutorial also requires a configured PingOne instance.

Compatibility

PingOne

- Your PingOne instance must have DaVinci enabled.

DaVinci flows

Ensure your flows only use supported connectors, capabilities and fields for user interactions:

- **HTTP Connector**
  - **Custom HTML** capability
    - HTTP Connector field and collector support
    - HTTP Connector SK-Component support

HTTP Connector field and collector support

Field (`Collector`)	Description	DaVinci module		
		Android	iOS	JavaScript
Text field (`TextCollector`)	Collects a single text string.	☑ `1.0.0`	☑ `1.0.0`	☑ `1.0.0`
Password field (`PasswordCollector`)	Collects a single text string that cannot be read from the screen.	☑ `1.0.0`	☑ `1.0.0`	☑ `1.0.0`
Submit Button (`SubmitCollector`)	Sends the collected data to PingOne to continue the DaVinci flow.	☑ `1.0.0`	☑ `1.0.0`	☑ `1.0.0`
Flow Button (`FlowCollector`)	Triggers an alternative flow without sending the data collected so far to PingOne.	☑ `1.0.0`	☑ `1.0.0`	☑ `1.0.0`
Label (`LabelCollector`)	Display a read-only text label.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`
Radio / Dropdown (`SingleSelectCollector`)	Collects a single value from a choice of multiple options.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`

HTTP Connector SK-Component support

SK-Component (`Collector`)	Description	DaVinci module		
		Android	iOS	JavaScript

skIDP (`IdpCollector`)	Presents a button to allow users to authenticate using an external identity provider, such as Apple, Facebook, or Google.	☑ 1.1.0	☑ 1.1.0	☑ 1.1.0

Verify that your flow does not depend on any *unsupported* elements:

SKPolling components

The SKPolling component cannot be processed by the DaVinci Client and should not be included in flows.

Features such as Magic Link authentication require the **SKPolling** component and therefore cannot be used with the DaVinci Client.

Images

Images included in the flow cannot be passed to the SDK.

For example, the PingOne sign-on with sessions DaVinci flow.

- **PingOne Form Connector**
  - **Show Form** capability
    - Custom Fields support
    - Toolbox support

Custom Fields support

Field (`Collector`)	Description	DaVinci module		
		Android	iOS	JavaScript
Text Input (`TextCollector`)	Collects a single text string.	☑ 1.1.0	☑ 1.1.0	☑ 1.1.0
Password (`PasswordCollector`)	Collects a single text string that cannot be read from the screen.	☑ 1.1.0	☑ 1.1.0	☑ 1.1.0
Dropdown (`SingleSelectCollector`)	Collects a value from a dropdown containing one or more text strings.	☑ 1.1.0	☑ 1.1.0	☑ 1.1.0
Combobox (`MultiSelectCollector`)	Collects a value from a dropdown containing one or more text strings, the user can enter their own text string.	☑ 1.1.0	☑ 1.1.0	☑ 1.1.0

Radio Button List (`SingleSelectCollector`)	Collects a value from one or radio buttons.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`
Checkbox List (`MultiSelectCollector`)	Collects the value of one or more checkboxes.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`

Toolbox support

Field (`Collector`)	Description	DaVinci module		
		Android	iOS	JavaScript
Flow Button (`FlowCollector`)	Presents a customized button.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`
Flow Link (`FlowCollector`)	Presents a customized link.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`
Translatable Rich Text (`TextCollector`)	Presents rich text that you can translate into multiple languages.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`
Social Login (`IdpCollector`)	Presents a button to allow users to authenticate using an external identity provider, such as Apple, Facebook, or Google.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`

iOS

This sample app is compatible with iOS 12 and later.

Prerequisites

Xcode

You can download the latest version for free from https://developer.apple.com/xcode/.

Server configuration

You must configure your PingOne instance for use with the DaVinci client.

Ask your PingOne administrator to complete the following tasks:

- Configure a DaVinci flow
- Create a DaVinci application
- Configure PingOne for DaVinci flow invocation

To learn how to complete these steps, refer to Launching a flow with a Ping SDK in the *PingOne DaVinci documentation*.

Step 1. Download the samples

Prepare › **Download** › Configure › Run

To start this tutorial, you need to download the ForgeRock SDK sample apps repo, which contains the projects you will use.

1. In a web browser, navigate to the SDK Sample Apps repository.
2. Download the source code using one of the following methods:

   Download a ZIP file

   1. Click **Code**, and then click **Download ZIP**.
   2. Extract the contents of the downloaded ZIP file to a suitable location.

   Use a Git-compatible tool to clone the repo locally

   1. Click **Code**, and then copy the HTTPS URL.
   2. Use the URL to clone the repository to a suitable location.

      For example, from the command-line you could run:

      ```
      git clone https://github.com/ForgeRock/sdk-sample-apps.git.git
      ```

The result of these steps is a local folder named `sdk-sample-apps`.

Step 2. Configure the sample app

Prepare › Download › **Configure** › Run

In this section you open the sample project in Xcode, and view the integration points in the **TODO** pane.

You'll visit each integration point in the sample app to understand how to complete a DaVinci flow, including handling the different nodes and their collectors, obtaining an access token and user information, and finally signing out of the session.

1. In *Xcode*, on the **File** menu, click **Open**.
2. Navigate to the `sdk-sample-apps` folder you cloned in the previous step, navigate to `iOS` > `swiftui-davinci` > `Davinci.xcworkspace`, and then click **Open**.

   Xcode opens and loads the DaVinci tutorial project.

3. Open `DavinciViewModel` and locate the `DaVinci.createDaVinci` call:

The `DaVinci.createDaVinci` call in `DavinciViewModel`

```
public let davinci = DaVinci.createDaVinci { config in
  //TODO: Provide here the Server configuration. Add the PingOne server Discovery Endpoint and the OAuth 2.0
client details
  config.module(OidcModule.config) { oidcValue in
    oidcValue.clientId = "Client ID"
    oidcValue.scopes = ["scope1", "scope2", "scope3"]
    oidcValue.redirectUri = "Redirect URI"
    oidcValue.discoveryEndpoint = "Discovery Endpoint"
  }
}
```

This snippet initializes the `DaVinci` module, and leverages the OpenID Connect (OIDC) module to configure the settings to connect to your PingOne instance.

1. In the `oidcValue.clientId` property, enter the ID of the client you are connecting to in PingOne.

   Example:

   `clientId = "6c7eb89a-66e9-ab12-cd34-eeaf795650b2"`

   Refer to Get configuration values from PingOne for instructions of where to find this value.

2. In the `oidcValue.scopes` property, add the scopes you want to assign users who complete authentication using the client.

   Example:

   `scopes = mutableSetOf("openid", "email", "profile")`

3. In the `oidcValue.redirectUri` property, enter the application ID of your sample app, followed by `://oauth2redirect`.

   Example:

   `redirectUri = "org.forgerock.demo://oauth2redirect"`

   > **Important**
   >
   > The `redirectUri` value you use must exactly match one of the **Redirect URIs** value you enter in the native OAuth 2.0 application you created earlier.

4. In the `oidcValue.discoveryEndpoint` property, enter the OIDC Discovery Endpoint value from the client you are connecting to in PingOne.

   Example:

   `discoveryEndpoint = "https://auth.pingone.ca/3072206d-c6ce-ch15-m0nd-f87e972c7cc3/as/.well-known/openid-configuration"`

   Refer to Get configuration values from PingOne for instructions of where to find this value.

5. Optionally, delete the `TODO` comment to remove it from the list.

The result resembles the following:

DavinciViewModel

```
public let davinci = DaVinci.createDaVinci { config in
  //TODO: Provide here the Server configuration. Add the PingOne server Discovery Endpoint and the OAuth2.0
client details
  config.module(OidcModule.config) { oidcValue in
    oidcValue.clientId = "6c7eb89a-66e9-ab12-cd34-eeaf795650b2"
    oidcValue.scopes = ["openid", "email", "profile"]
    oidcValue.redirectUri = "org.forgerock.demo://oauth2redirect"
    oidcValue.discoveryEndpoint = "https://auth.pingone.ca/3072206d-c6ce-ch15-m0nd-f87e972c7cc3/as/.well-
known/openid-configuration"
  }
}
```

Step 3. Test the app

Prepare › Download › Configure › **Run**

In the following procedure, you run the sample app that you configured in the previous step. The app performs a centralized login on your PingOne instance.

Log in as a demo user

1. In Xcode, select **Product** > **Run**.

   Xcode launches the sample app in the iPhone simulator.

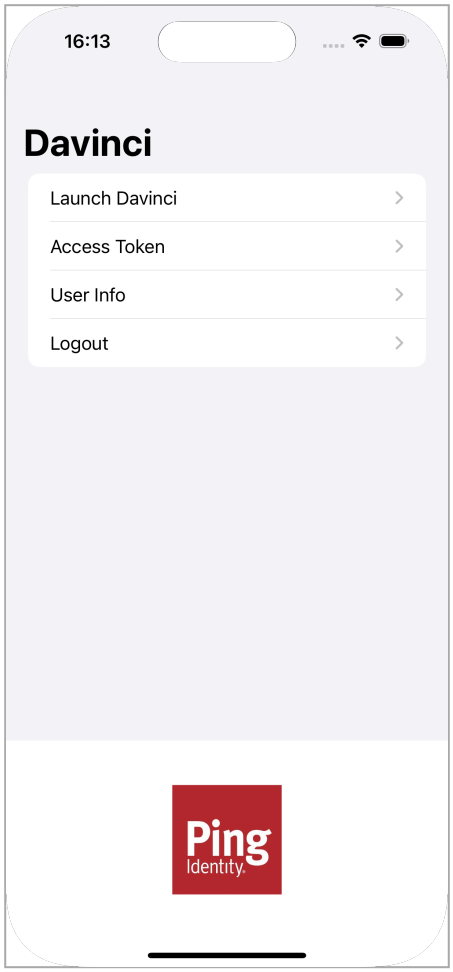


Figure 1. The iOS DaVinci sample main menu

2. In the sample app on the iPhone simulator, tap **Launch Davinci**.

   The sample app launches the DaVinci flow configured in the OAuth 2.0 profile.

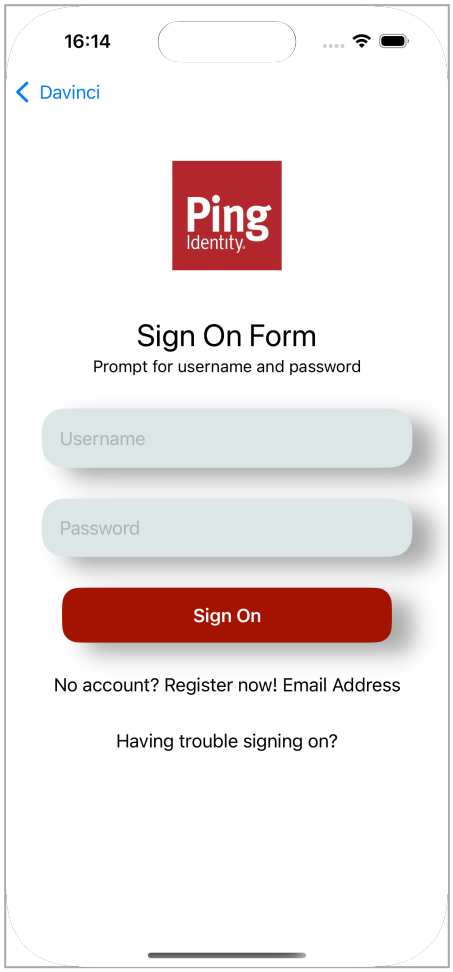


Figure 2. The DaVinci sample app first screen with fields and buttons.

3. Optionally, to register a new identity in PingOne:

    1. Tap the **No Account? Register now!** link.

    > **Tip**
    >
    > This link is an example of a `FlowButton`.

    The app displays the registration screen:

Figure 3. The DaVinci sample app registration screen.

2. Enter the details of the new identity, and then click **Save**.

   The app creates the new identity in PingOne and returns to the sign on screen.

4. Sign on as a demo user:

   - **Name**: `demo`
   - **Password**: `Ch4ng3it!`

If authentication is successful, the application displays the tokens issued by PingOne.

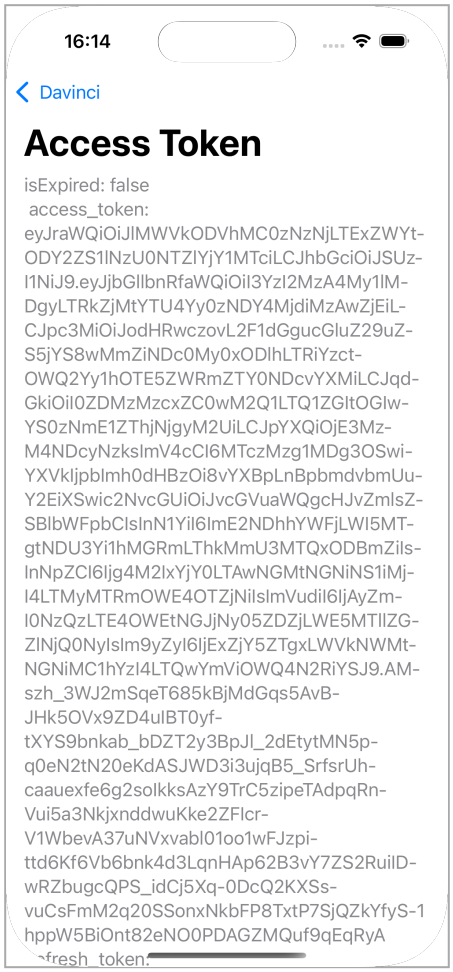


Figure 4. The token issues by the DaVinci flow

5. Tap **< Davinci** to go back to the main menu, and then tap **User Info**.

   The app retrieves the user's info from the `/userinfo` endpoint and displays it on screen:

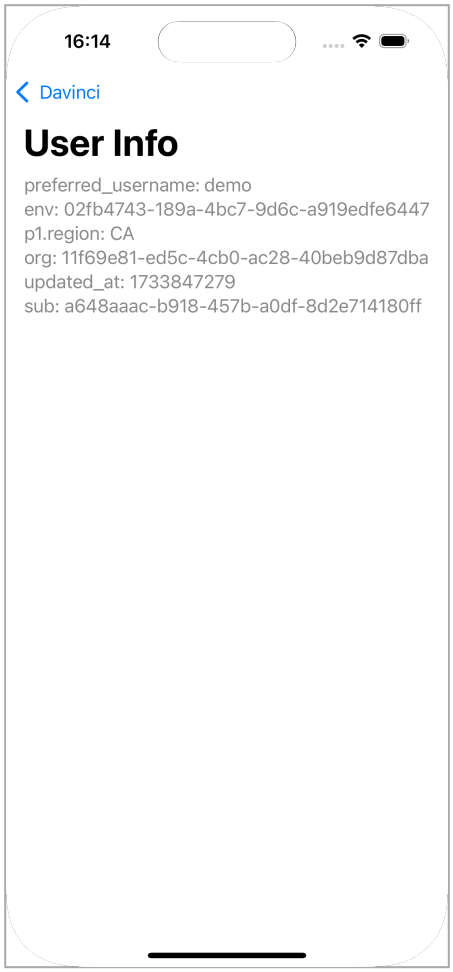


Figure 5. User info retrieved from PingOne

6. Tap < **Davinci** to go back to the main menu, and then tap **Logout**.

   The sample app displays a logout button.

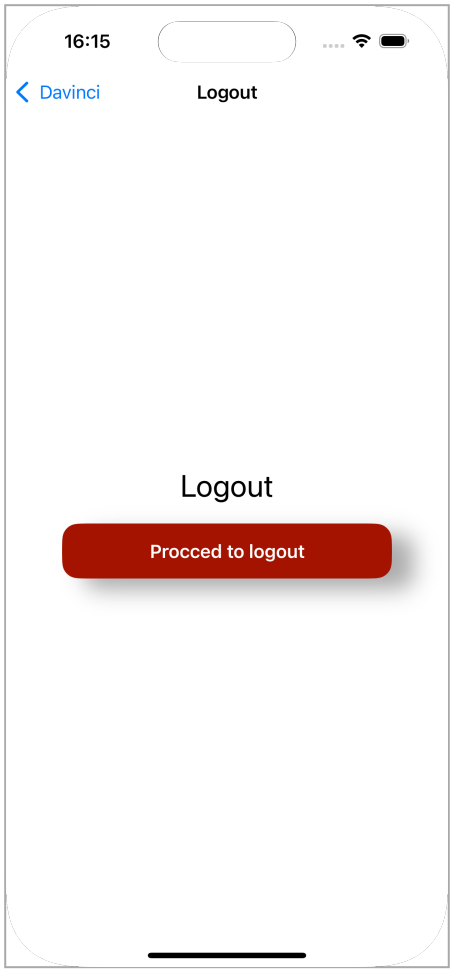


Figure 6. Logout button in the iOS sample app

7. Tap **Proceed to logout**

   The app revokes the existing tokens and ends the session in PingOne.

Tip

To verify the user is signed out:

1. In the PingOne administration console, navigate to **Directory > Users**.
2. Select the user you signed in as.
3. From the **Sevices** dropdown, select **Authentication**:

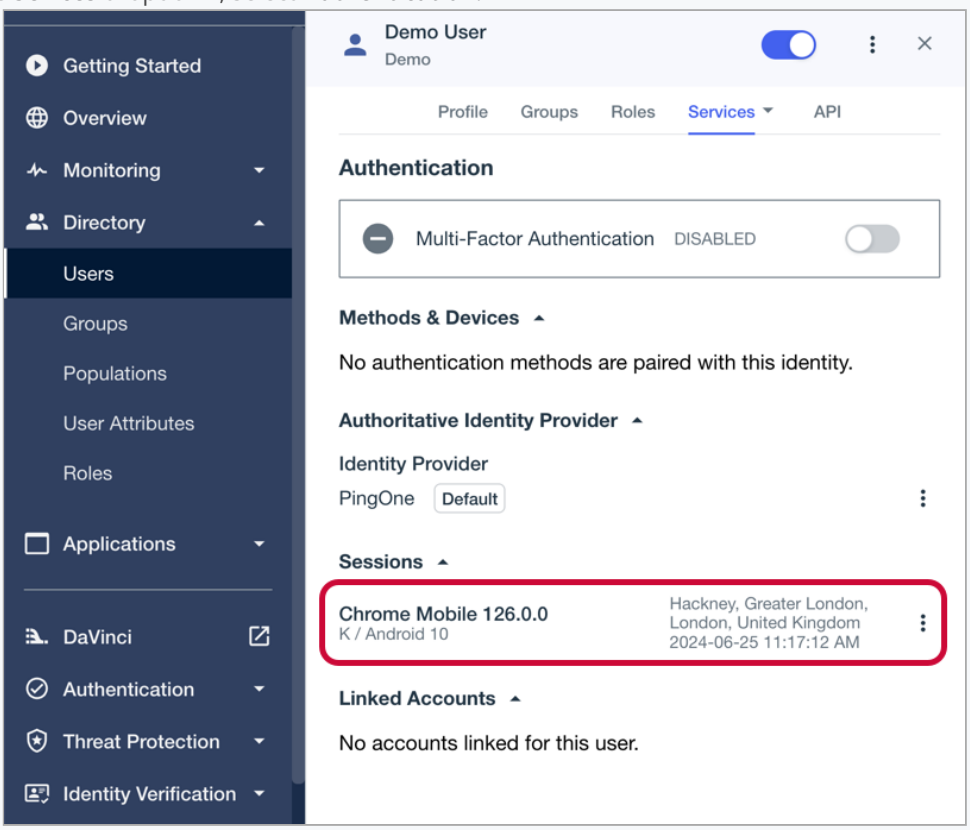


Figure 7. Checking a user's sessions in PingOne.

The **Sessions** section displays any existing sessions the user has, and whether they originate from a mobile device.

DaVinci client deep dive for iOS

Applies to:

- ✕ DaVinci client for Android
- ✓ DaVinci client for iOS
- ✕ DaVinci client for JavaScript

Configure DaVinci client module for iOS properties to connect to PingOne and step through an associated DaVinci flow.

Installing and importing the DaVinci client

To use the DaVinci client for iOS, use Swift Package Manager (SPM) or Cocoapods to add the dependencies to your project.

Add dependencies using SPM (Swift Package Manager)

You can install this by using SPM (Swift Package Manager) on the generated iOS project.

1. In Xcode,in the Project Navigator, right-click your project, and then click **Add Package Dependencies...**.
2. In the **Search or Enter Package URL** field, enter the URL of the repo containing the DaVinci Client for iOS, `https://github.com/ForgeRock/ping-ios-sdk.git`.
3. In **Add to Project**, select the name of your project, and then click **Add Package**.

   Xcode shows a dialog containing the libraries available in the Ping SDK for iOS.
4. Select the `PingDavinci` library, and in the **Add to Target** column select the name of your project.
5. Repeat the previous step for any other Ping SDK libraries you want to add to your project.
6. Click **Add Package**.

   Xcode displays the chosen libraries and any prerequisites they might have in the **Package Dependencies** pane of the Project Navigator:

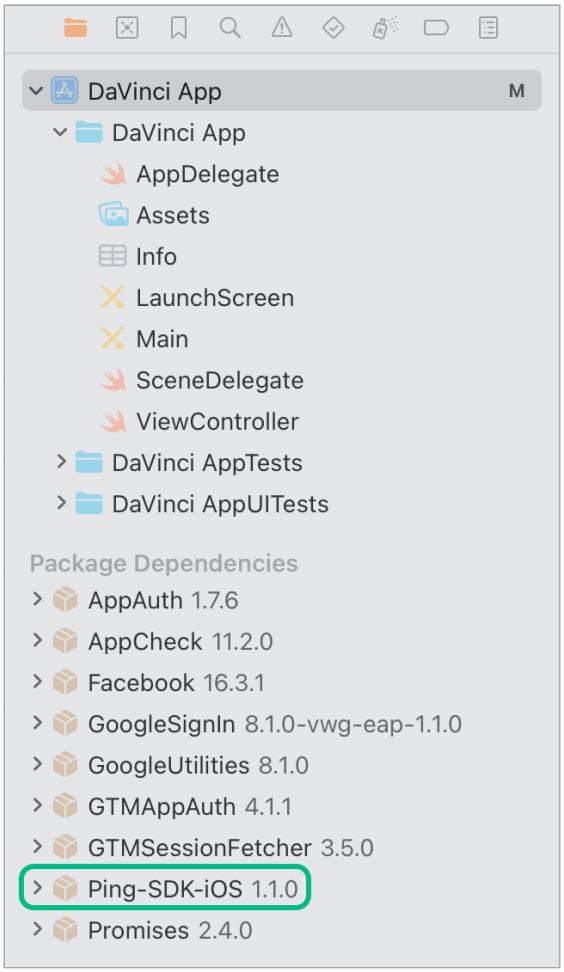


Figure 1. Package dependencies in the Xcode package navigator pane.

Add dependencies using CocoaPods

1. If you do not already have CocoaPods, install the latest version.
2. If you do not already have a Podfile, in a terminal window, run the following command to create a new Podfile:

```
pod init
```

3. Add the following lines to your Podfile:

```
pod 'PingDavinci'
```

4. Run the following command to install pods:

```
pod install
```

Configuring the DaVinci client

Configure DaVinci client for iOS properties to connect to PingOne and step through an associated DaVinci flow.

The following shows an example DaVinci client configuration, using the underlying `Oidc` module:

Configure DaVinci client connection properties

```
let daVinci = DaVinci.createDaVinci { config in
    // Oidc as module
    config.module(OidcModule.config) { oidcValue in
        oidcValue.clientId = "6c7eb89a-66e9-ab12-cd34-eeaf795650b2"
        oidcValue.discoveryEndpoint = "https://auth.pingone.com/3072206d-c6ce-ch15-m0nd-f87e972c7cc3/as/.well-known/
openid-configuration"
        oidcValue.scopes = ["openid", "profile", "email", "address", "revoke"]
        oidcValue.redirectUri = "org.forgerock.demo://oauth2redirect"
        oidcValue.additionalParameters = ["customKey":"customValue"]
    }
}
```

For information on the properties available, refer to Configure DaVinci client for iOS properties.

Stepping through DaVinci flows

To authenticate your users the DaVinci client for iOS must start the flow, and step through each **node**.

> **Tip**
>
> For information on which connectors and fields the DaVinci client supports, refer to Compatibility.

Starting a DaVinci flow

To start a DaVinci flow, call the `start()` method:

Start a DaVinci flow

```
var node = await daVinci.start()
```

Determining DaVinci flow node type

Each step of the flow returns one of four node types:

`ContinueNode`

This type indicates there is input required from the client. The `node` object for this type contains a `collector` object, which describes the information it requires from the client.

Learn more in Handling DaVinci flow collectors in continue nodes.

`SuccessNode`

This type indicates the flow is complete, and authentication was successful.

Learn more in Handling DaVinci flow success nodes.

`ErrorNode`

This type indicates an error in the data sent to the server. For example, an email address in an incorrect format, or a password that does not meet complexity requirements.

You can correct the error and resubmit to continue the flow.

Learn more in Handling DaVinci flow error nodes.

`FailureNode`

This type indicates that the flow could not be completed and must be restarted. This can be caused by a server error, or a timeout.

Learn more in Handling DaVinci flow failure nodes.

You can use the helper functions to determine which node type the server has returned:

Determine `node` type.

```
switch (node) {
  case is ContinueNode: do {}
  case is ErrorNode: do {}
  case is FailureNode: do {}
  case is SuccessNode: do {}
}
```

Handling DaVinci flow collectors in continue nodes

The `ContinueNode` type contains `collectors`. These collectors define what information or action to request from the user, or client device.

> **Tip**
>
> For a list of supported collectors, refer to Supported PingOne fields and collectors.

There are specific collector types. For example there are `TextCollector` and `PasswordCollector` types.

To complete a DaVinci flow we recommend that you implement a component for each connector type you will encounter in the flow. Then you can iterate through the flow and handle each collector as you encounter it.

Access collectors in a `ContinueNode`

```
node.collectors.forEach { item in
    switch(item) {
    case is TextCollector:
        (item as! TextCollector).value = "My First Name"
    case is PasswordCollector:
        (item as! PasswordCollector).value = "My Password"
    case is SubmitCollector:
        (item as! SubmitCollector).value = "click me"
    case is FlowCollector:
        (item as! FlowCollector).value = "Forgot Password"
    }
}
```

Continuing a DaVinci flow

After collecting the data for a node you can proceed to the next node in the flow by calling the `next()` method on your current `node` object.

Continue a DaVinci flow using `next()`

```
let next = node.next()
```

> **Note**
>
> You do not need to pass any parameters into the `next` method as the DaVinci client internally stores the updated object, ready to return to the PingOne server.

The server responds with a new `node` object, just like when starting a flow initially.

Loop again through conditional checks on the new node's type to render the appropriate UI or take the appropriate action.

Handling DaVinci flow error nodes

DaVinci flows return the `ErrorNode` type when it receives data that is incorrect, but you can fix the data and resubmit. For example, an email value submitted in an invalid format or a new password that is too short.

You can retrieve the error message by using `node.message()`, and the raw JSON response with `node.input`.

Displaying the reason for an error

```
let node = await daVinci.start() //Start the flow

//Determine the Node Type
switch (node) {
  case is ContinueNode: do {}
  case is FailureNode: do {}
  case is ErrorNode:
      (node as! ErrorNode).message //Retrieve the error message
  case is SuccessNode: do {}
}
```

> **Note**
>
> This is different than a `FailureNode` type, which you cannot resubmit and must restart the entire flow.

You can retain a reference to the `node` you submit in case the next `node` you receive is an `ErrorNode` type. If so, you can re-render the previous form, and inject the error information from the new `ErrorNode` node.

After the user revises the data call `next()` as you did before.

Handling DaVinci flow failure nodes

DaVinci flows return the `FailureNode` type if there has been an issue that prevents the flow from continuing. For example, the flow times out or suffers a server error.

You can retrieve the cause of the failure by using `node.cause()`, which is an `Error` instance.

Handling receipt of a `FailureNode` type

```
let node = await daVinci.start() //Start the flow

//Determine the Node Type
switch (node) {
  case is ContinueNode: do {}
  case is FailureNode:
    (node as! FailureNode).cause //Retrieve the cause of the Failure
  case is ErrorNode: do {}
  case is SuccessNode: do {}
}
```

You should offer to restart the flow on receipt of a `FailureNode` type.

Handling DaVinci flow success nodes

DaVinci flows return the `SuccessNode` type when the user completes the flow and PingOne issues them a session.

To retrieve the existing session, you can use the following code:

Handling receipt of a `SuccessNode` type

```
let user: User? = await daVinci.user()

_ = await user?.token()
await user?.revoke()
_ = await user?.userinfo(cache: false)
await user?.logout()
```

Leverage SwiftUI

The following shows how you could use the DaVinci client with SwiftUI:

ViewModel

```
//Define State that listen by the View

@Published var state: Node = EmptyNode()

//Start the DaVinci flow
let next = await daVinci.start()

//Update the state
state = next

func next(node: ContinueNode) {
   val next = await node.next()
   state = next

}
```

View

```
if let node = state.node {
    switch node {
    case is ContinueNode:
        // Handle ContinueNode case
        break
    case is ErrorNode:
        // Handle Error case
        break
    case is FailureNode:
        // Handle Failure case
        break
    case is SuccessNode:
        // Handle Success case
        break
    default:
        break
    }
}
```

DaVinci Client for JavaScript tutorials

Follow these tutorials integrate your JavaScript apps with DaVinci flows in your PingOne instance.

DaVinci Client for JavaScript Tutorials

Quick start

In this quick start tutorial you update one of our sample applications.

The app steps through a DaVinci flow and displays a basic prototype UI to gather user credentials.

Deep dive

This deep dive tutorial covers the steps to take to integrate an JavaScript app with a DaVinci flow in PingOne.

You'll learn about installing and importing the DaVinci Client and stepping through DaVinci flows.

DaVinci client quick start for JavaScript

Prepare › Download › Install › Configure › Run

This tutorial walks you through updating a provided sample app so that it connects to a PingOne tenant to authenticate a user using the PingOne sign-on with sessions DaVinci flow.

This flow allows users to register, authenticate, and verify their email address with PingOne.

Before you begin

Before you begin this tutorial ensure you have set up your PingOne instance with the required configuration.

For example, you will need an OAuth 2.0 client application set up.

Complete prerequisites »

Step 1. Download the samples

To start this tutorial, you need to download the SDK sample apps repo, which contains the projects you will use.

Start step 1 »

Step 2. Install the dependencies

The sample projects need a number of dependencies that you can install by using the `npm` command.

For example, the Ping SDK for JavaScript itself is one of the dependencies.

Start step 2 »

Step 3. Configure connection properties

In this step, you configure the "embedded-login" sample app to connect to the OAuth 2.0 application you created in PingOne Advanced Identity Cloud or PingAM.

Start step 3 »

Step 4. Test the app

The final step is to run the sample app. The sample connects to your server and walks through your authentication journey or tree.

After successful authentication, the sample obtains an OAuth 2.0 access token and displays the related user information.

Test app »

Before you begin

Prepare › Download › Install › Configure › Run

To successfully complete this tutorial refer to the prerequisites in this section.

The tutorial also requires a configured server.

Compatibility

PingOne

- Your PingOne instance must have DaVinci enabled.

DaVinci flows

Ensure your flows only use supported connectors, capabilities and fields for user interactions:

- **HTTP Connector**
  - **Custom HTML** capability
    - HTTP Connector field and collector support
    - HTTP Connector SK-Component support

HTTP Connector field and collector support

Field (`Collector`)	Description	DaVinci module		
		Android	iOS	JavaScript
Text field (`TextCollector`)	Collects a single text string.	☑ `1.0.0`	☑ `1.0.0`	☑ `1.0.0`
Password field (`PasswordCollector`)	Collects a single text string that cannot be read from the screen.	☑ `1.0.0`	☑ `1.0.0`	☑ `1.0.0`
Submit Button (`SubmitCollector`)	Sends the collected data to PingOne to continue the DaVinci flow.	☑ `1.0.0`	☑ `1.0.0`	☑ `1.0.0`
Flow Button (`FlowCollector`)	Triggers an alternative flow without sending the data collected so far to PingOne.	☑ `1.0.0`	☑ `1.0.0`	☑ `1.0.0`
Label (`LabelCollector`)	Display a read-only text label.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`
Radio / Dropdown (`SingleSelectCollector`)	Collects a single value from a choice of multiple options.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`

HTTP Connector SK-Component support

SK-Component (`Collector`)	Description	DaVinci module		
		Android	iOS	JavaScript

skIDP (`IdpCollector`)	Presents a button to allow users to authenticate using an external identity provider, such as Apple, Facebook, or Google.	☑ 1.1.0	☑ 1.1.0	☑ 1.1.0

Verify that your flow does not depend on any *unsupported* elements:

SKPolling components

The SKPolling component cannot be processed by the DaVinci Client and should not be included in flows.

Features such as Magic Link authentication require the **SKPolling** component and therefore cannot be used with the DaVinci Client.

Images

Images included in the flow cannot be passed to the SDK.

For example, the PingOne sign-on with sessions DaVinci flow.

- **PingOne Form Connector**
  - **Show Form** capability
    - Custom Fields support
    - Toolbox support

Custom Fields support

Field (`Collector`)	Description	DaVinci module		
		Android	iOS	JavaScript
Text Input (`TextCollector`)	Collects a single text string.	☑ 1.1.0	☑ 1.1.0	☑ 1.1.0
Password (`PasswordCollector`)	Collects a single text string that cannot be read from the screen.	☑ 1.1.0	☑ 1.1.0	☑ 1.1.0
Dropdown (`SingleSelectCollector`)	Collects a value from a dropdown containing one or more text strings.	☑ 1.1.0	☑ 1.1.0	☑ 1.1.0
Combobox (`MultiSelectCollector`)	Collects a value from a dropdown containing one or more text strings, the user can enter their own text string.	☑ 1.1.0	☑ 1.1.0	☑ 1.1.0

Radio Button List (`SingleSelectCollector`)	Collects a value from one or radio buttons.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`
Checkbox List (`MultiSelectCollector`)	Collects the value of one or more checkboxes.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`

Toolbox support

Field (`Collector`)	Description	DaVinci module		
		Android	iOS	JavaScript
Flow Button (`FlowCollector`)	Presents a customized button.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`
Flow Link (`FlowCollector`)	Presents a customized link.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`
Translatable Rich Text (`TextCollector`)	Presents rich text that you can translate into multiple languages.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`
Social Login (`IdpCollector`)	Presents a button to allow users to authenticate using an external identity provider, such as Apple, Facebook, or Google.	☑ `1.1.0`	☑ `1.1.0`	☑ `1.1.0`

Prerequisites

Node and NPM

The SDK requires a minimum Node.js version of `18`, and is tested on versions `18` and `20`. To get a supported version of Node.js, refer to the Node.js download page.

You will also need `npm` to build the code and run the samples.

Server configuration

You must configure your PingOne instance for use with the DaVinci client.

Ask your PingOne administrator to complete the following tasks:

- Configure a DaVinci flow
- Create a DaVinci application
- Configure PingOne for DaVinci flow invocation

To learn how to complete these steps, refer to Launching a flow with a Ping SDK in the *PingOne DaVinci documentation*.

Step 1. Download the samples

Prepare › **Download** › Install › Configure › Run

To start this tutorial, you need to download the SDK sample apps repo, which contains the projects you will use.

1. In a web browser, navigate to the SDK Sample Apps repository.
2. Download the source code using one of the following methods:

Download a ZIP file

1. Click **Code**, and then click **Download ZIP**.
2. Extract the contents of the downloaded ZIP file to a suitable location.

Use a Git-compatible tool to clone the repo locally

1. Click **Code**, and then copy the HTTPS URL.
2. Use the URL to clone the repository to a suitable location.

   For example, from the command-line you could run:

```
git clone https://github.com/ForgeRock/sdk-sample-apps.git
```

The result of these steps is a local folder named `sdk-sample-apps`.

Step 2. Install the dependencies

Prepare › Download › **Install** › Configure › Run

In the following procedure, you install the required modules and dependencies, including the Ping SDK for JavaScript.

1. In a terminal window, navigate to the `sdk-sample-apps/javascript/embedded-login-davinci` folder.
2. To install the required packages, enter the following:

```
npm install
```

   The `npm` tool downloads the required packages, and places them inside a `node_modules` folder.

Step 3. Configure connection properties

Prepare › Download › Install › **Configure** › Run

In this step, you configure the sample app to connect to the authentication tree/journey you created when setting up your server configuration.

1. Copy the `.env.example` file in the `sdk-sample-apps/javascript/embedded-login-davinci` folder and save it with the name `.env` within this same directory.
2. Open the `.env` file and edit the property values to match the settings you configured in previous steps:

```
VITE_SCOPE=$SCOPE
VITE_WEB_OAUTH_CLIENT=$WEB_OAUTH_CLIENT
VITE_WELLKNOWN_URL=$WELLKNOWN_URL
```

   1. In the *VITE_SCOPE* property, enter the scopes you want to assign users who complete authentication using the client, separated by spaces.

      Example:

      `VITE_SCOPE="openid profile email address"`

   2. In *VITE_WEB_OAUTH_CLIENT*, property, enter the ID of the client you are connecting to in PingOne.

      Example:

      `VITE_WEB_OAUTH_CLIENT="6c7eb89a-66e9-ab12-cd34-eeaf795650b2"`

      Refer to Get configuration values from PingOne for instructions of where to find this value.

   3. In the *VITE_WELLKNOWN_URL* property, enter the OIDC Discovery Endpoint value from the client you are connecting to in PingOne.

      Example:

      `discoveryEndpoint = "https://auth.pingone.ca/3072206d-c6ce-ch15-m0nd-f87e972c7cc3/as/.well-known/openid-configuration"`

      Refer to Get configuration values from PingOne for instructions of where to find this value.

The result resembles the following:

```
VITE_SCOPE="openid profile email address"
VITE_WEB_OAUTH_CLIENT="sdkPublicClient"
VITE_WELLKNOWN_URL="https://auth.pingone.ca/3072206d-c6ce-ch15-m0nd-f87e972c7cc3/as/.well-known/openid-configuration"
```

Step 4. Test the app

Prepare › Download › Install › Configure › **Run**

In the following procedure, you run the sample app that you configured in the previous step. The sample connects to your server and walks through the authentication journey you created in an earlier step.

After successful authentication, the sample obtains an OAuth 2.0 access token and displays the related user information.

1. In a terminal window, navigate to the `sdk-sample-apps/javascript/embedded-login-davinci` folder.
2. Use `npm` to run the sample:

   ```
   npm run dev
   ```

3. In a web browser:
   1. Ensure you are *NOT* currently logged into the PingOne instance.

      > **Note**
      >
      > If you are logged into the PingOne instance in the browser, the sample will not work. Logout of the PingAM instance *before* you run the sample.

   2. Navigate to the following URL:

      `http://localhost:5173`

      A form appears with *"Username"* and *"Password"* fields:

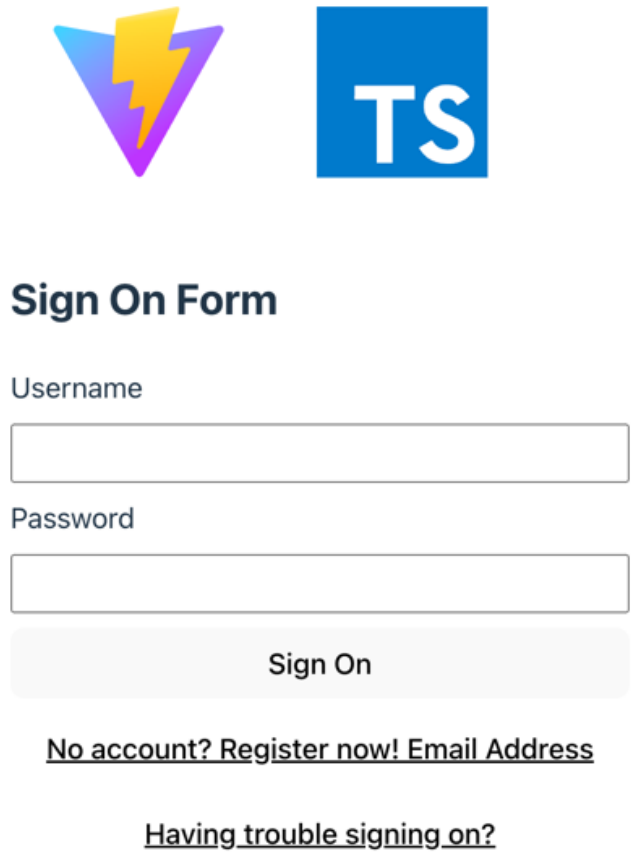


Figure 1. The login page of the JavaScript DaVinci client sample.

4. Optionally, to register a new identity in PingOne, tap the **No Account? Register now!** link.

> **Tip**
>
> This link is an example of a `FlowButton`.

The app displays the registration screen:

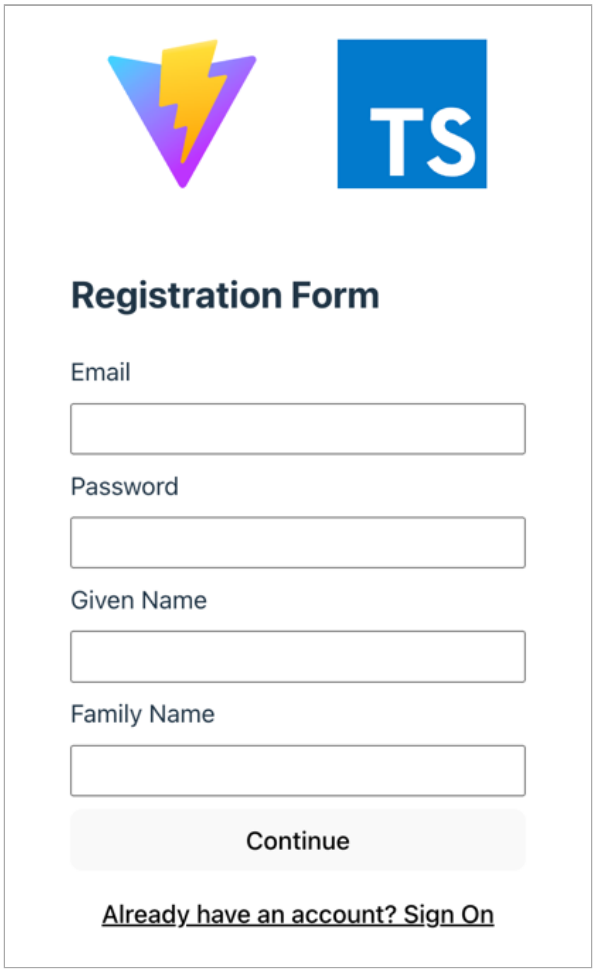


Figure 2. The DaVinci sample app registration screen.

1. Enter the details of the new identity, and then click **Save**.

   The app creates the new identity in PingOne and returns to the sign on screen.

5. Enter the username and password of a PingOne identity, and then click **Sign On**.

   The app sends the credentials to PingOne for validation, and on success displays the user's session info:

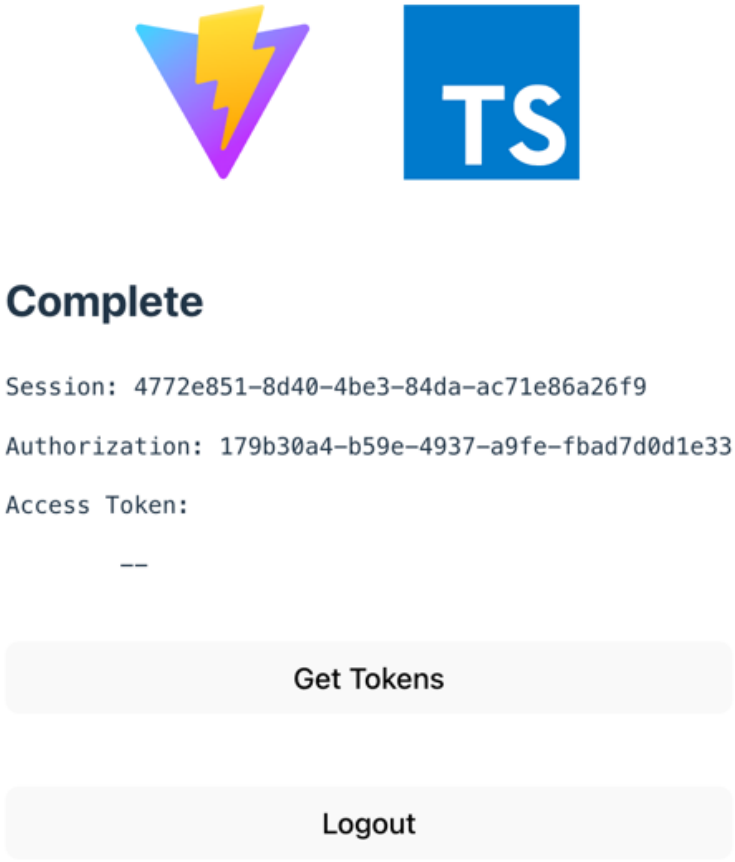


Figure 3. The DaVinci sample app displaying session info

6. To get an access token for the user, click **Get Tokens**.

Figure 4. The DaVinci sample app displaying tokens

7. To revoke the OAuth 2.0 token, click the **Logout** button.

   The application calls the `/as/revoke` endpoint to revoke the OAuth 2.0 token, and returns to the sign-in form.

DaVinci client deep dive for JavaScript

Applies to:

- ✕ DaVinci client for Android
- ✕ DaVinci client for iOS
- ✓ DaVinci client for JavaScript

Configure DaVinci client properties to connect to PingOne and step through an associated DaVinci flow.

Installing and importing the DaVinci client

To install and import the DaVinci client:

1. Install the DaVinci client into your JavaScript apps using `npm`:

Install the DaVinci client

```
npm install @forgerock/davinci-client
```

2. Import the DaVinci client as a named import:

Import the DaVinci client

```
import { davinci } from '@forgerock/davinci-client';
```

Configuring the DaVinci client

Configure DaVinci client for JavaScript properties to connect to PingOne and step through an associated DaVinci flow.

The following shows a full DaVinci client configuration:

Configure DaVinci client connection properties

```
import { davinci } from '@forgerock/davinci';

const davinciClient = await davinci({
  config: {
    clientId: '6c7eb89a-66e9-ab12-cd34-eeaf795650b2',
    serverConfig: {
      wellknown: 'https://auth.pingone.com/3072206d-c6ce-ch15-m0nd-f87e972c7cc3/as/.well-known/openid-configuration',
      timeout: 3000,
    },
    scope: 'openid profile email phone',
    responseType: 'code',
  },
});
```

For information on the properties available, refer to Configure DaVinci client for JavaScript properties.

Stepping through DaVinci flows

To authenticate your users the Ping SDK for JavaScript DaVinci client must start the flow, and step through each **node**.

> **Tip**
>
> For information on which connectors and fields the DaVinci client supports, refer to Compatibility.

Starting a DaVinci flow

To start a DaVinci flow, call the `start()` method on your new client object:

Start a DaVinci flow

```
let node = await davinciClient.start();
```

Adding custom parameters

When starting a DaVinci client you can add additional key-pair parameters. The DaVinci client will append these parameters as query strings to the initial OAuth 2.0 call to the `/authorize` endpoint.

> **Tip**
>
> You can access these additional OAuth 2.0 parameters in your DaVinci flows by using the `authorizationRequest.<customParameter>` property.
> Learn more in Referencing PingOne data in the flow.

To add parameters when starting the client, create an object of the key-value pairs and pass it as a `query` parameter to the `start()` function:

```
const query = {
  customKey: 'customValue'
}

let node = await davinciClient.start(query);
```

> **Tip**
>
> You can add any parameters to the request as required. For example, you could add `acr_values` to the request to the `/authorize` endpoint.

Determining DaVinci flow node type

Each step of the flow returns one of four node types:

`continue`

This type indicates there is input required from the client. The `node` object for this type contains a list of `collector` objects, which describe the information it requires from the client.

Learn more in Handling DaVinci flow collectors in continue nodes.

`success`

This type indicates the flow is complete, and authentication was successful.

Learn more in Handling DaVinci flow success nodes.

`error`

This type indicates an error in the data sent to the server. For example, an email address in an incorrect format, or a password that does not meet complexity requirements.

You can correct the error and resubmit to continue the flow.

Learn more in Handling DaVinci flow error nodes.

`failure`

This type indicates that the flow could not be completed and must be restarted. This can be caused by a server error, or a timeout.

Learn more in Handling DaVinci flow failure nodes.

Use `node.status` to determine which node type the server has returned:

Determine node type using the `node.status` property

```
let node = await davinciClient.start();

switch (node.status) {
  case 'continue':
    return renderContinue();
  case 'success':
    return renderSuccess();
  case 'error':
    return renderError();
  default: // Handle 'failure' node type
    return renderFailure();
}
```

Handling DaVinci flow collectors in continue nodes

The `continue` node type contains a list of `collector` objects. These collectors define what information or action to request from the user, or browser.

> **Tip**
>
> For a list of supported collectors, refer to Supported PingOne fields and collectors.

The Ping SDK for JavaScript groups collectors that have similar traits together into categories. For example, collectors that only require a single primitive value to be returned to the server, such as a username or password string, or a single value from a drop-down list are grouped together in a single value collectors category.

To complete a DaVinci flow, we recommend that you either implement a component for a *category* of collectors, or implement a component for each collector *type* that you will encounter in the flow.

Your app iterates through the flow and handles each collector as you encounter it.

Example of iterating collectors and using components

```
const collectors = davinciClient.getCollectors();
collectors.forEach((collector) => {
  if (collector.type === 'TextCollector') {
    textComponent(
      collector, // Object with the collector details
      davinciClient.update(collector), // Returns an update function for this collector
      davinciClient.validate(collector), // Returns a validate function for this collector
    );
  } else if (collector.type === 'PasswordCollector') {
    // eslint-disable-next-line @typescript-eslint/no-unused-expressions
    collector;
    passwordComponent(
      collector, // Object with the collector details
      davinciClient.update(collector), // Returns an update function for this collector
    );
  } else if (collector.type === 'SubmitCollector') {
    submitButtonComponent(
      collector, // Object with the collector details
    );
  } else if (collector.type === 'FlowCollector') {
    flowLinkComponent(
      collector, // Object with the collector details
      davinciClient.flow({
        // Returns a function to call the flow from within component
        action: collector.output.key,
      }),
      renderForm, // Enable re-rendering the form
    );
  }
});
```

Example 1. Handling `TextCollector` with a component

This example shows how to update a collector with a value gathered from your user.

Pass both a `collector` and `updater` object into a component that renders the appropriate user interface, captures the user's input, and then updates the collector, ready to return to the server.

Example `TextCollector` mapping

```
const collectors = davinciClient.getCollectors();
collectors.map((collector) => {
  if (collector.type === 'TextCollector') {
    renderTextCollector(collector, davinciClient.update(collector));
  }
});
```

> **Note**
>
> Mutating the `node` object, the `collectors` array, or any other properties does not alter the internal state of the DaVinci client.
> The internal data the client stores is immutable and can only be updated using the provided APIs, not through property assignment.

Your `renderTextCollector` would resemble the following:

Example `TextCollector` updater component

```
function renderTextCollector(collector, updater) {
  // ... component logic

  function onClick(event) {
    updater(event.target.value);
  }

  // render code
}
```

Example 2. Handling `FlowCollector` with a component

This example shows how change from the current flow to an alternate flow, such as a reset password or registration flow.

To switch flows, call the `flow` method on the `davinciClient` passing the `key` property to identify the new flow.

Example `FlowCollector` mapping

```
const collectors = davinciClient.getCollectors();
collectors.map((collector) => {
  if (collector.type === 'FlowCollector') {
    renderFlowCollector(collector, davinciClient.flow(collector));
  }
});
```

This returns a function you can call when the user interacts with it.

Example `flowCollector` component

```
function renderFlowCollector(collector, startFlow) {
  // ... component logic

  function onClick(event) {
    startFlow();
  }

  // render code
}
```

Example 3. Handling `SubmitCollector` with a component

This example shows how submit the current node and its collected values back to the server. The collection of the data is already complete so an updater component is not required. This collector only renders the button for the user to submit the collected data.

Example `SubmitCollector` mapping

```
const collectors = davinciClient.getCollectors();
collectors.map((collector) => {
  if (collector.type === 'SubmitCollector') {
    renderSubmitCollector(
      collector, // This is the only argument you will need to pass
    );
  }
});
```

Continuing a DaVinci flow

After collecting the data for a node you can proceed to the next node in the flow by calling the `next()` method on your DaVinci client object.

This can be the result of a user clicking on the button rendered from the `SubmitCollector`, from the `submit` event of an HTML form, or by programmatically triggering the submission in the application layer.

Continue a DaVinci flow using `next()`

```
let nextStep = davinciClient.next();
```

> **Note**
>
> You do not need to pass any parameters into the `next` method as the DaVinci client internally stores the updated object, ready to return to the PingOne server.

The server responds with a new `node` object, just like when starting a flow initially.

Loop again through conditional checks on the new node's type to render the appropriate UI or take the appropriate action.

Handling DaVinci flow error nodes

DaVinci flows return the `error` node type when it receives data that is incorrect, but you can fix the data and resubmit. For example, an email value submitted in an invalid format or a new password that is too short.

This is different than a `failure` node type which you cannot resubmit and instead you must restart the entire flow.

You can retain a reference to the `node` you submit in case the next `node` you receive is an `error` type. If so, you can re-render the previous form, and inject the error information from the new `error` node.

After the user revises the data call `next()` as you did before.

Handling DaVinci flow failure nodes

DaVinci flows return the `failure` node type if there has been an issue that prevents the flow from continuing. For example, the flow times out or suffers an HTTP 500 server error.

You should offer to restart the flow on receipt of a `failure` node type.

Restart a DaVinci flow on receipt of a `failure` node type

```
const node = await davinciClient.next();

if (node.status === 'failure') {
  const error = davinciClient.getError();
  renderError(error);

  // ... user clicks button to restart flow
  const freshNode = davinciClient.start();
}
```

Handling DaVinci flow success nodes

DaVinci flows return the `success` node type when the user completes the flow and PingOne issues them a session.

On receipt of a `success` node type you should use the OAuth 2.0 authorization Code and state properties from the node and use them to obtain an access token on behalf of the user.

To obtain an access token, leverage the Ping SDK for JavaScript.

Example of obtaining an access token using the Ping SDK for JavaScript

```
// ... other imports
import { davinci } from '@forgerock/davinci-client';
import { Config, TokenManager } from '@forgerock/javascript-sdk';

// ... other config or initialization code

// This Config.set accepts the same config schema as the davinci function
Config.set(config);

const node = await davinciClient.next();

if (node.status === 'success') {
  const clientInfo = davinciClient.getClient();

  const code = clientInfo.authorization?.code || '';
  const state = clientInfo.authorization?.state || '';

  const tokens = await TokenManager.getTokens({
    query: {
      code, state
    }
  });

  // user now has session and OIDC tokens
}
```

Implement your use cases with the DaVinci client

The DaVinci client enables you to implement many authentication, registration, and self-service use cases into your mobile and web apps.

Visit the following pages for more information on implementing different use cases using the DaVinci client:

Set up Social Login

Applies to: Android | iOS | JavaScript

Add support for authenticating to your apps by using trusted Identity Providers (IdP), such as Apple, Facebook, and Google.

Read more »

Set up social sign on with external IDPs

PingOne supports trusted Identity Providers (IdP), like Apple, Facebook, Google, and many others, providing authentication and identity verification on behalf of Ping Identity.

This is often referred to as *social sign on* or *social authentication*. These IdPs return the necessary user information for creating or validating accounts to your PingOne server.

The user is redirected from the client application to the IdP's authorization server. Once on the IdP, the user authenticates, and provides the necessary consent required for sharing the information with PingOne. When the IdP authenticates your user, they are redirected back to your server to complete the flow. When PingOne completes the flow, it redirects the user to your app, where they are now signed on.

It's common to offer these social login options in addition to traditional authentication with username and password, but they can be used alone.

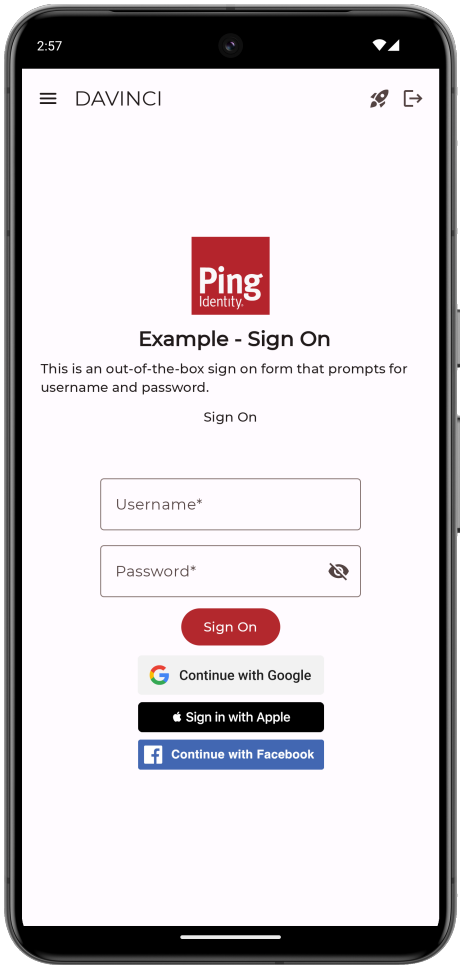


Figure 1. An Android app with a combination of authentication methods

Steps

Complete the following steps to integrate social login into your client applications:

Before you begin

Before you begin this tutorial, ensure you have set up your PingOne instance with the required configuration.

For example, you will need an OAuth 2.0 client application setup.

Complete prerequisites »

Configure client apps for social sign on

Learn how to set up your Android, iOS, and JavaScript client apps to handle social sign on.

Start step 1 »

Before you begin

To complete this tutorial, refer to the prerequisites in this section.

The tutorial also requires a configured server.

Compatibility

PingOne

- Your PingOne instance must have DaVinci enabled.
- Only **PingOne External IdPs** are supported.
  - Identity providers configured using a **DaVinci Service Connector** are not supported.

Connecting external identity providers in PingOne

In this section, you configure PingOne with details about the social login identity providers you want to integrate into your client apps.

The Ping SDKs are compatible with any OpenID Connect 1.0-compliant Identity Provider, such as those available by default in PingOne.

Note

You must configure the identity provider as a **PingOne External IdP**. Learn more in External IdPs.
Identity providers configured by using a **DaVinci Service Connector** are not supported.

Ping Identity has tested the steps in this tutorial with the Identity Providers listed below. Select a provider to view the PingOne documentation with instructions on how to configure an external IdP in PingOne:

Apple

Adding an identity provider - Apple

Facebook

Adding an identity provider - Facebook

Google

Adding an identity provider - Google

Configuring DaVinci Flows for social sign on

After connecting your chosen external identity providers to PingOne, the next step is to configure a DaVinci flow to display buttons on your login pages so that users can choose to authenticate using the external IdP.

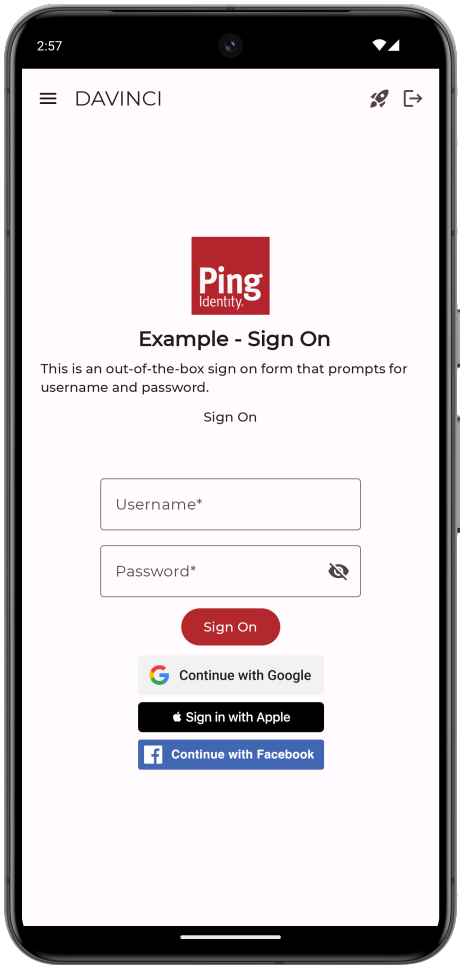


Figure 1. An Android app with three external IdP options: Google, Apple, and Facebook.

The Ping SDKs support two options for adding social sign on to your DaVinci flows:

DaVinci Forms

DaVinci Forms is a drag-and-drop form builder that allows you to create custom forms without having to write HTML.

HTTP Connector

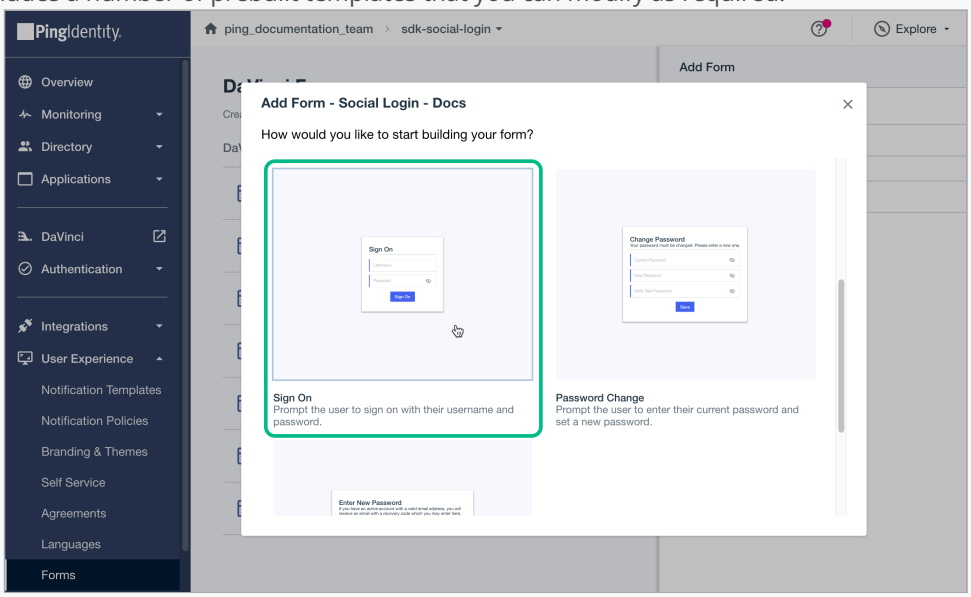
This powerful and versatile connector lets you show custom HTML pages in your DaVinci orchestration flows.

Option A. Configuring DaVinci Forms for social sign on

Complete the following steps to integrate external IdPs with PingOne using DaVinci Forms.

Creating a DaVinci Form

1. Create a form to display your selected external identity providers.

> **Tip**
>
> PingOne includes a number of prebuilt templates that you can modify as required.
>
> 
>

2. To add external identity providers to the form:

    1. From the **Toolbox** tab, drag a **Social Login** field onto the form for each external identity provider you want to display.

    2. In **PingOne External Identity Provider**, select the external IdP you created earlier. For example, Google.

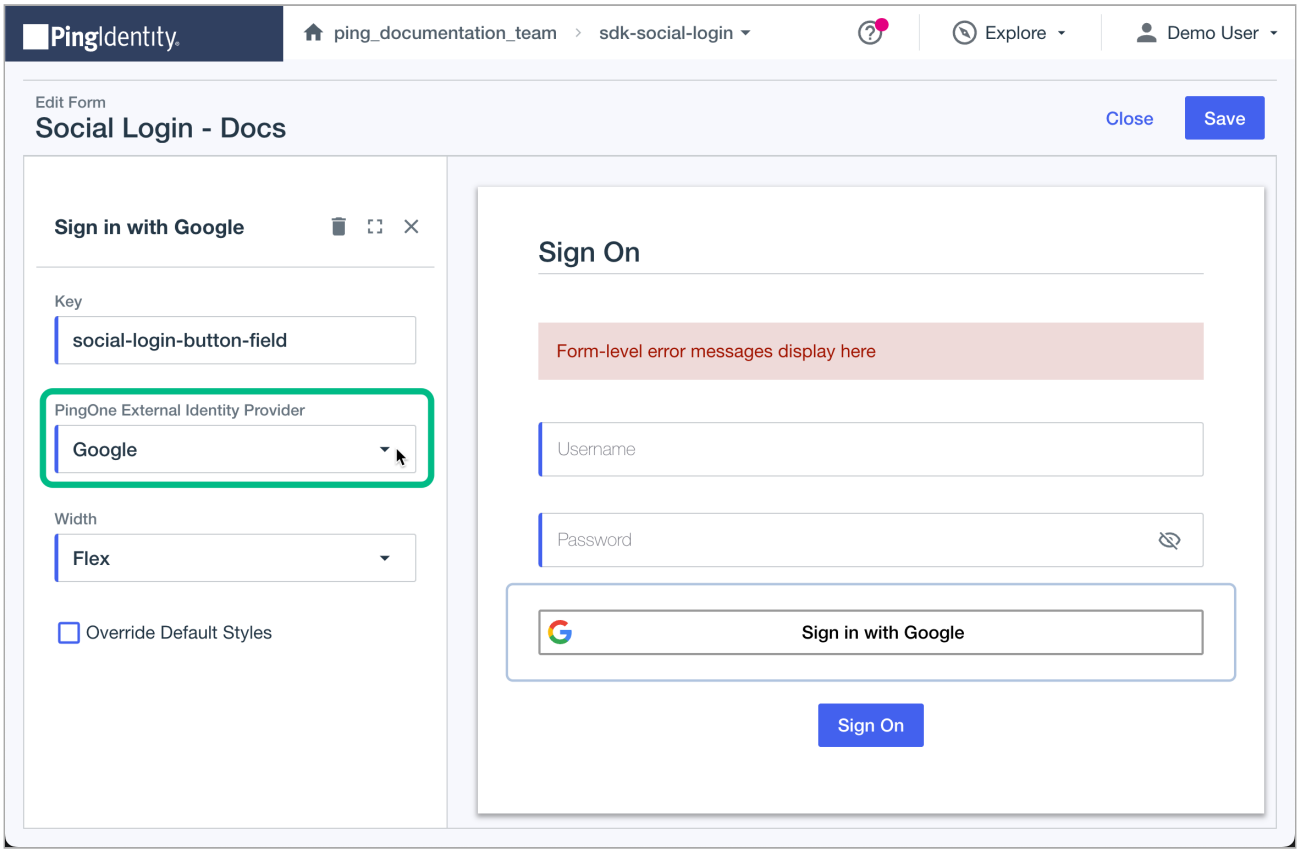


Figure 2. Configuring a Social Login field to use Google as the external IdP.

3. Save your changes.

Learn more in Creating a form in the PingOne documentation.

Adding a form to a DaVinci flow

When you have added your external identity providers to your form, you must now include it as part of your DaVinci flow.

1. Add the form you created for external IdPs to a flow by using the PingOne Forms connector.

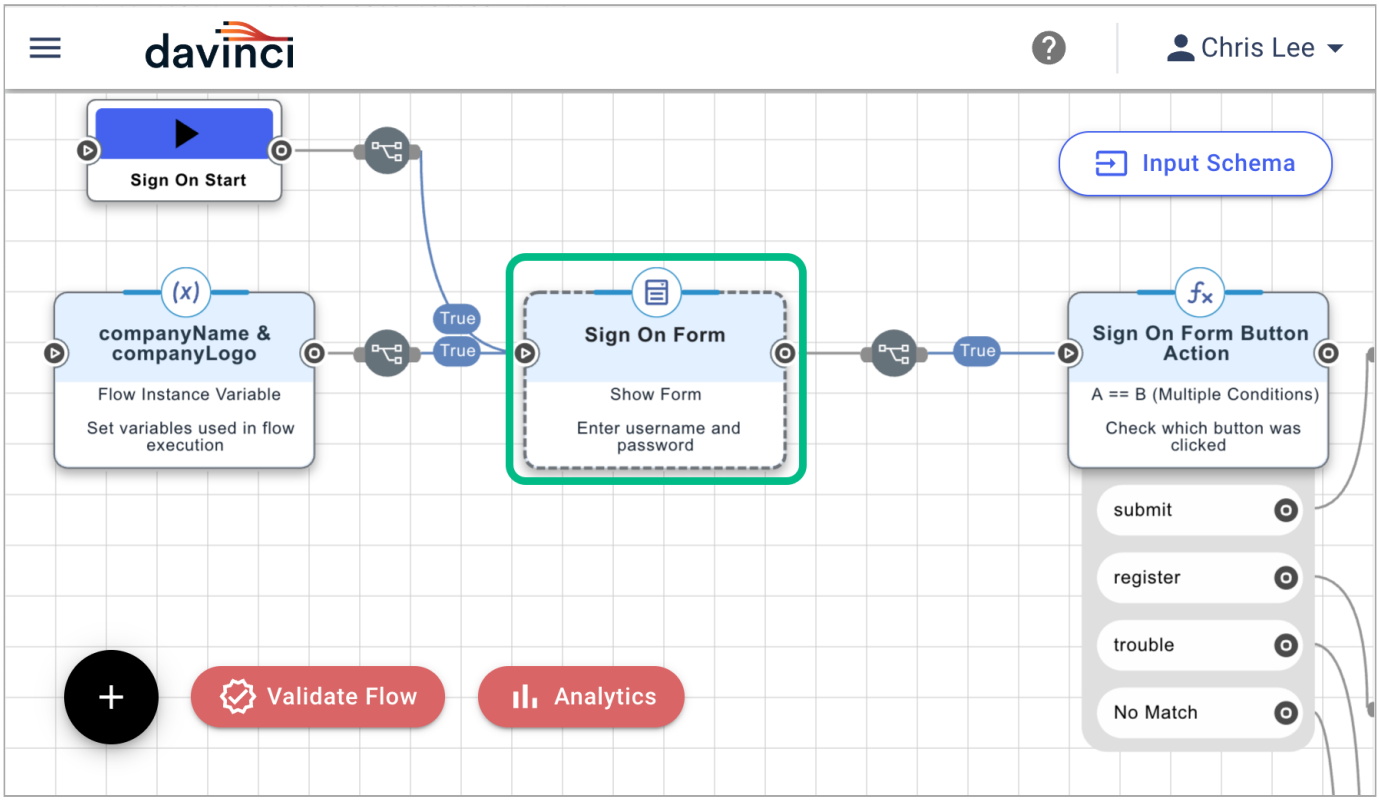


Figure 3. Example of a Forms Connector in a DaVinci flow.

2. To ensure the server can redirect back to an Android or iOS mobile app you must add a custom URI scheme.

> **Note**
>
> This is not required if you are only implementing a JavaScript client

Select the PingOne Forms connector you just added, click the **General** tab, and in **Application Return URL**, enter a custom URI scheme for redirecting users to your client app after social sign on.

If you are implementing Android or iOS clients for this tutorial, use `myapp://example.com`.

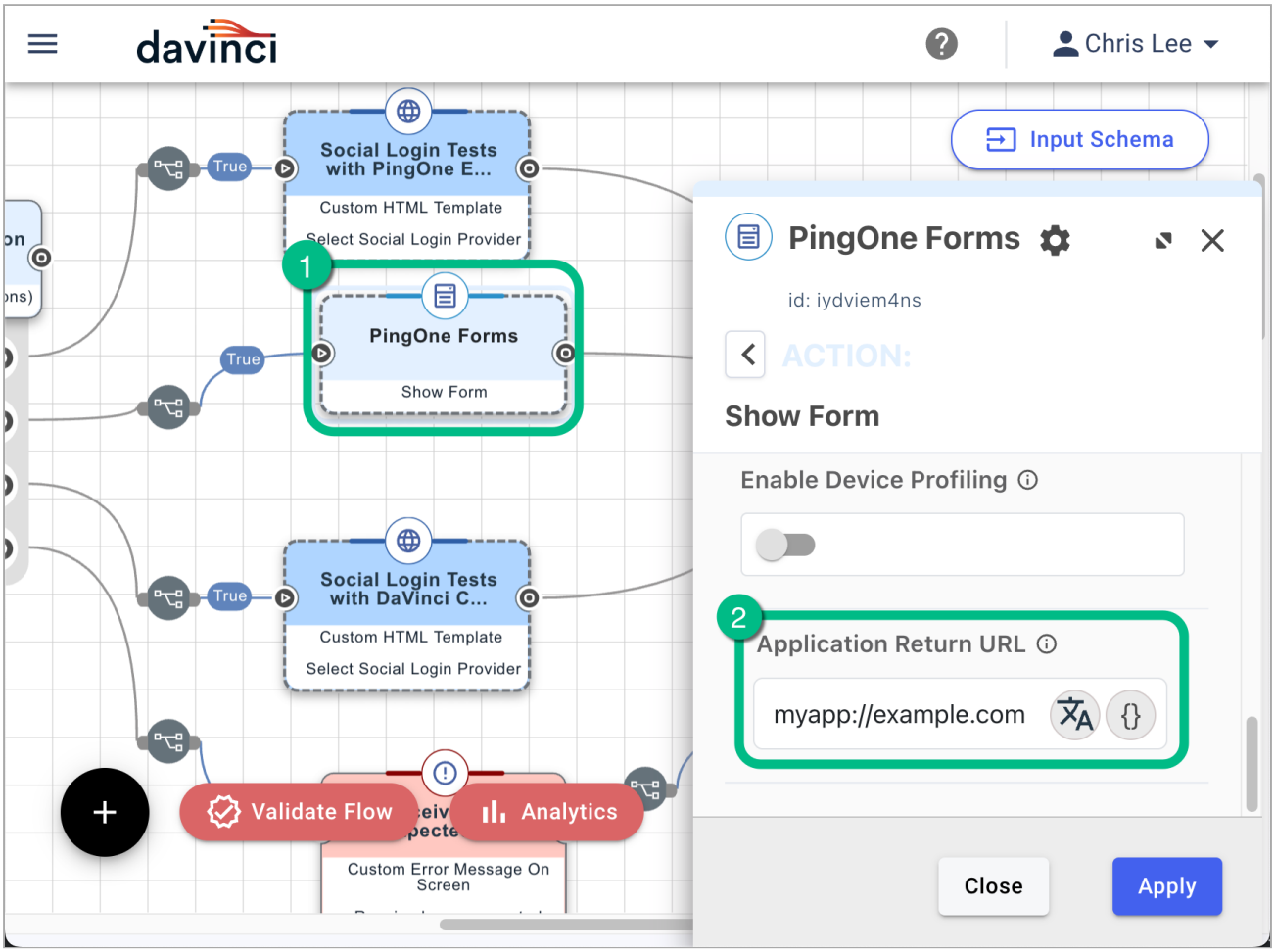


Figure 4. Configuring a return URL in the PingOne Form Connector.

3. Apply your changes.

You can now proceed to Configuring a DaVinci flow to be launched by the Ping SDKs.

Option B: Configuring the HTTP Connector for social sign on

Complete the following steps to integrate external IdPs with PingOne by adding the HTTP Connector to a DaVinci flow.

Adding the HTTP Connector to a DaVinci flow

1. You must add the **HTTP** connector to your DaVinci flow so that it can display your custom HTML sign-on page.

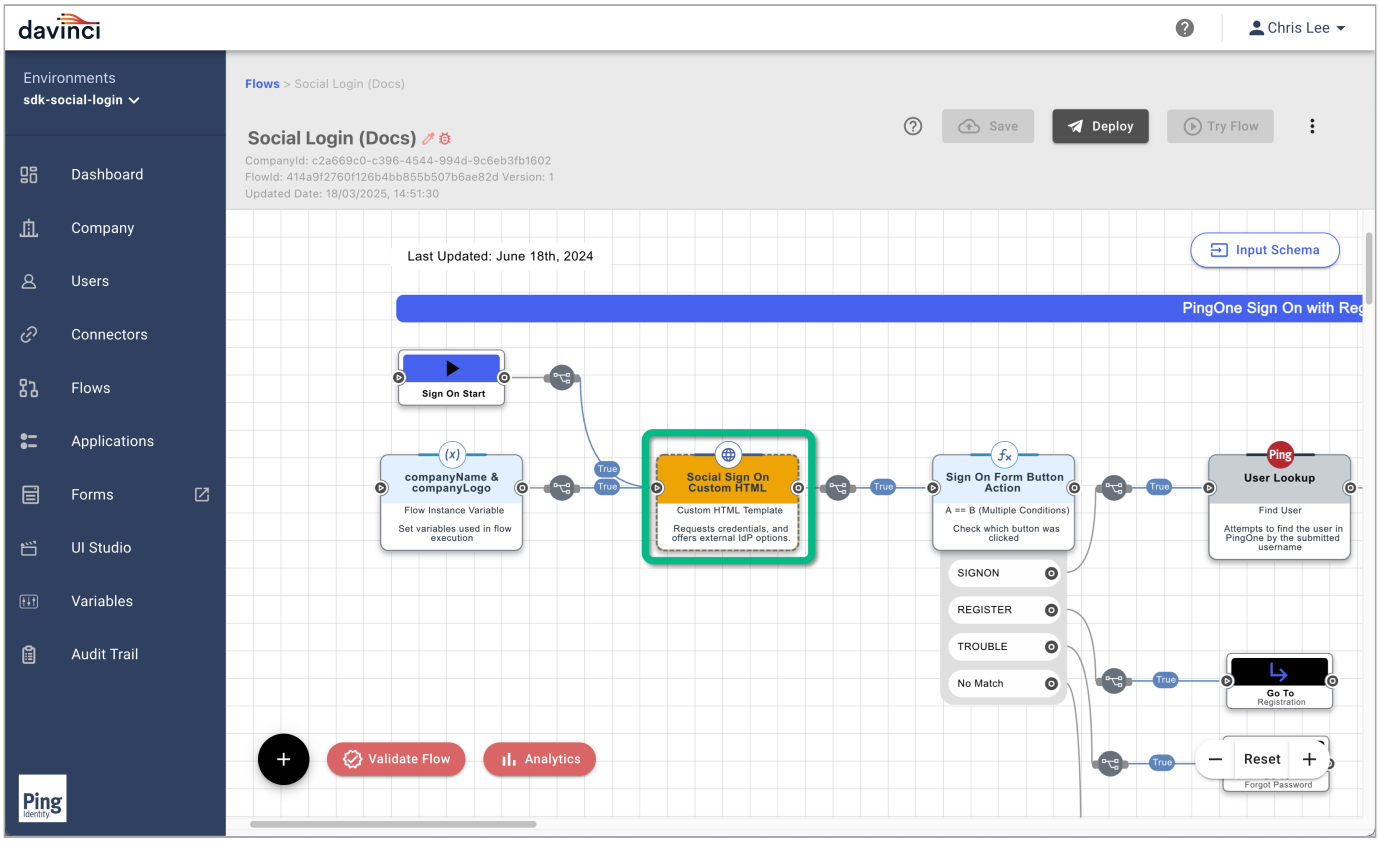


Figure 5. An HTTP connector added to a DaVinci flow.

To learn more, refer to Adding a connector.

Building a custom HTML sign-on page

With the HTTP Collector in place in the flow, you can now add custom HTML to display the sign-on page.

1. Select the HTTP Connector you added to your DaVinci flow, and add custom HTML to display a sign-on form.

HTML Template Switch View

```
<div class="end-user-nano">
  <div
    class="bg-light d-flex flex-column justify-content-center align-items-center position-absolute top-0 start-0 bottom-0 end-0 overflow-auto">
    <div style="max-width: 400px; min-width: 400px; width: 100%">
      <div class="card shadow mb-5">
        <main aria-labelledby="title">
          <div class="card-body p-5 d-flex flex-column">
            <img class="companyLogo align-self-center mb-5" src="{{global.variables.companyLogo}}" alt="{{global.variables.companyName}}">

            {{#if title}}
            <h1 id="title" class="text-center mb-4">{{title}}</h1>
            {{/if}}

            {{#if textOne}}
            <p class="text-muted text-center">{{textOne}}</p>
            {{/if}}

            {{#if textTwo}}
            <p class="text-muted text-center">{{textTwo}}</p>
            {{/if}}

            <!-- Generic Error Message -->
            <p class="text-danger mdi mdi-alert-circle text-center" id="feedbackError" data-id="feedback"
              data-skcomponent="skerror" aria-live="assertive"></p>

            <!-- Field Validation Error Messages -->
            <p class="text-danger mdi mdi-alert-circle text-center" id="usernameError" data-skerrorid="username"
              data-skcomponent="skerrormessage" aria-live="assertive"></p>
            <p class="text-danger mdi mdi-alert-circle text-center" id="passwordError" data-skerrorid="password"
              data-skcomponent="skerrormessage" aria-live="assertive"></p>

            <form id="signonForm" aria-labelledby="title" data-id="signonForm">
              <div class="mb-4 form-floating">
                <input class="form-control" id="username" name="username" placeholder="Username"
                  autocomplete="" data-id="username-input" aria-required="true"
                  aria-describedby="usernameLabel" />
                <label id="usernameLabel" for="username">Username</label>
              </div>
              <div id="passwordDiv" class="mb-4 form-floating">
                <input class="form-control" type="password" id="password" name="password" placeholder="Password"
                  autocomplete="" data-id="password-input" aria-required="true"
                  aria-describedby="passwordLabel" />
                <label id="passwordLabel" for="password">Password</label>
              </div>
```

Figure 6. Example custom HTML form in an HTTP connector.

To learn more about adding custom HTML, refer to Building a custom page.

2. Add an skIDP component to your custom HTML for each external IdP option you want to display.

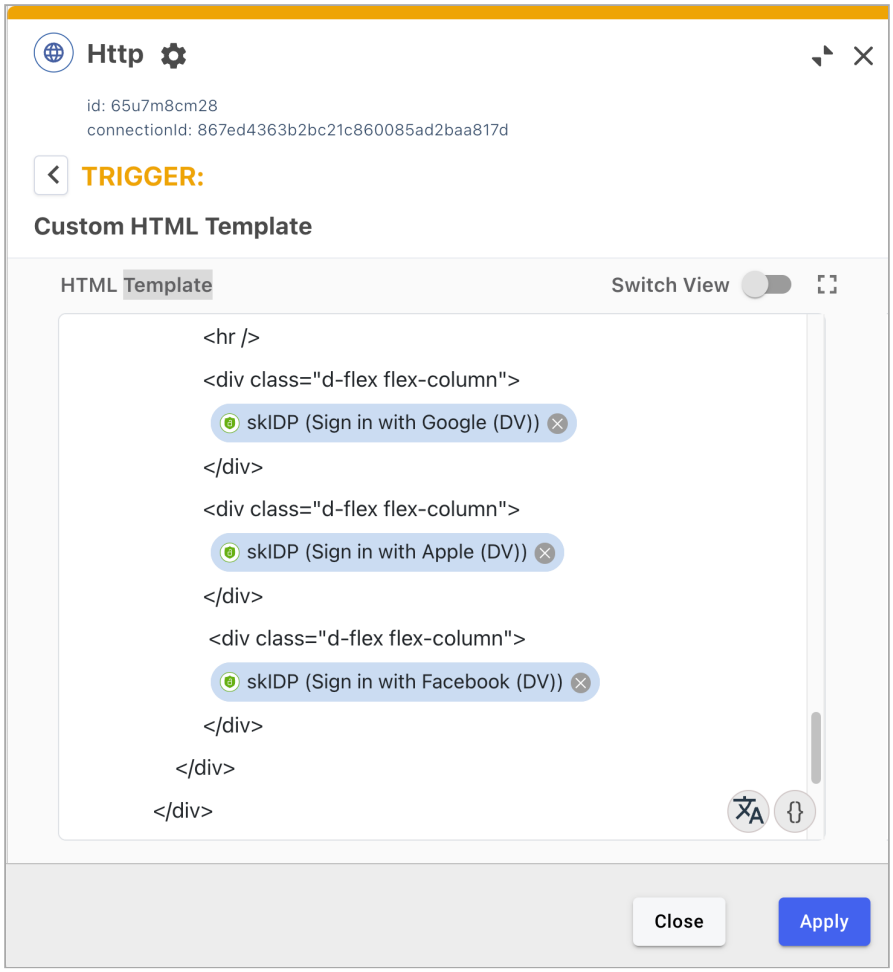


Figure 7. An HTTP connector with custom HTML showing three skIDP components.

To learn more, refer to Adding SK-Components to a connector.

3. Configure the **skIDP** component to use an external IdP:

    1. In the **HTML Template** field, select an **skIDP** component to view the **Update Component** modal.
    2. Select the **Identity Provider** tab.
    3. In **Identity Provider Connector**, select `PingOne Authentication`.
    4. In **PingOne External Identity Provider**, select one of the external IdPs you configured earlier.
    5. Enable **Link with PingOne User**.

        Failure to enable this option causes errors when attempting to use the flow with the Ping SDKs.

    6. To ensure the server can redirect back to an Android or iOS mobile app you must add a custom URI scheme.

        > **Note**
        >
        > This is not required if you are only implementing a JavaScript client

        In **Application Return to Url**, enter a custom URI scheme for redirecting users to your client app after social sign on.

If you are implementing Android or iOS clients for this tutorial, use `myapp://example.com`.

The result will resemble the following:

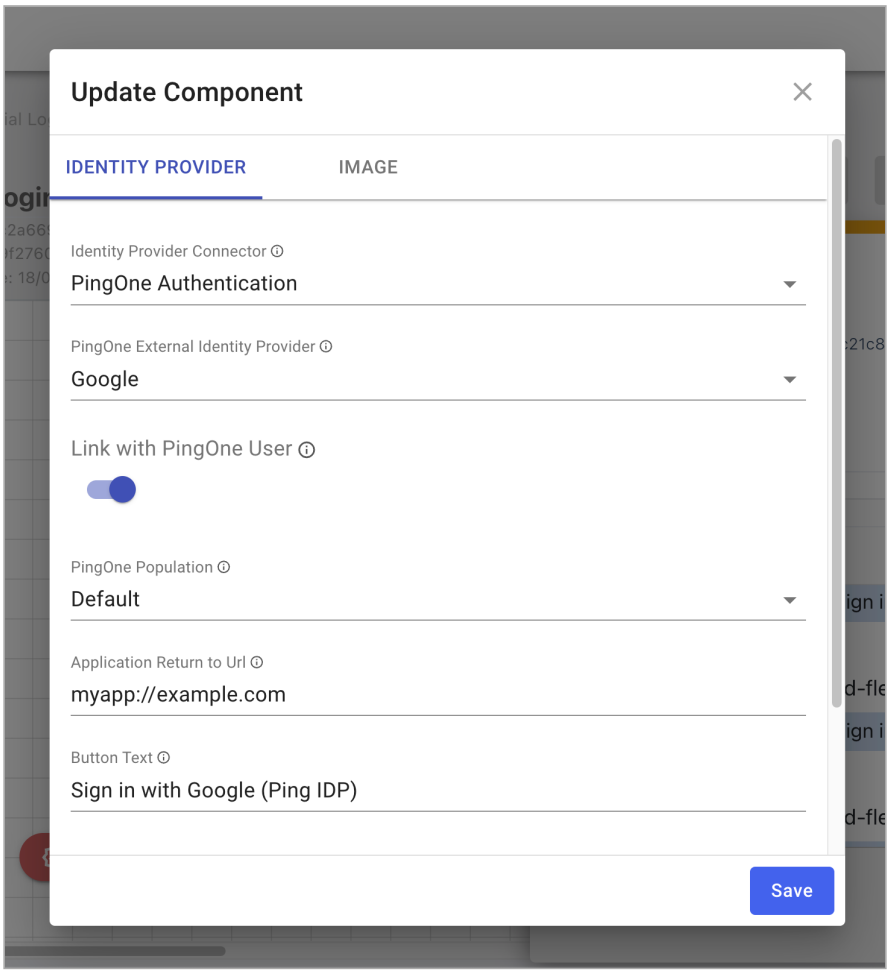


Figure 8. Configuring an skIDP component in an HTTP connector.

4. Save your changes.

You can now proceed to Configuring a DaVinci flow to be launched by the Ping SDKs.

Configuring a DaVinci flow to be launched by the Ping SDKs

Now that your DaVinci flow is configured to display your selected external IdPs you must configure PingOne so that you can launch the flow by using the Ping SDKs.

This involves performing the following high-level steps:

1. Checking that your DaVinci flow uses only compatible connectors and fields.
2. Creating an application in DaVinci to connect PingOne to the DaVinci flow.
3. Creating an application in PingOne that the Ping SDKs can connect to and access the DaVinci application and its PingOne Flow Policy.

To learn how to complete the steps, refer to Launching a flow with a Ping SDK in the DaVinci documentation.

Next Steps

Now that you have configured PingOne with external IdPs, added them to a DaVinci flow, and configured applications so that the Ping SDKs, you are ready to connect the Ping SDKs.

Related links

- Adding a connector
- HTTP Connector
- DaVinci Forms
  - Adding a form to a DaVinci flow
  - skIDP component

Configure client apps for social sign on

Select your platform to discover how to configure your client application to perform social sign on with an external IdP in PingOne.

Android

Configure an Android app for social sign on

iOS

Configure an iOS app for social sign on

JavaScript

Configure a JavaScript app for social sign on

Configure an Android app for social sign on

The Ping SDK for Android supports two methods for performing social sign on:

Redirect

The SDK redirects your user to the IdP's login user interface to authenticate. The IdP redirects them back to your app to continue complete the flow.

This is the default option and offers the widest compatibility.

Native

The SDK uses the native Android libraries of supported IdPs to handle social sign in using an embedded experience.

Perform the following steps to configure an Android app for social sign on using PingOne.

Step 1. Add core dependencies

Whichever method you choose to perform social sign on, you must add the core Ping SDKs `davinci` and `external-idp` modules in your project.

These dependencies provide support for redirecting users to the IdP for social sign-on. To use native libraries for supported IdPs refer to [Optional] Step 2. Add native SDK library dependencies.

1. In the **Project** tree view of your Android Studio project, open the `build.gradle.kts` file.
2. In the `dependencies` section, add the following:

```
// Ping SDK social sign-on dependencies
implementation("com.pingidentity.sdks:davinci:1.1.0")
implementation("com.pingidentity.sdks:external-idp:1.1.0")
```

[Optional] Step 2. Add native SDK library dependencies

Optionally, you can use an IdP's native SDK libraries to handle social sign on directly rather than redirecting the user in a web browser.

This can provide a smoother, more integrated experience for your users than the redirect method.

> **Note**
>
> If an IdP's native SDK libraries are not included in your app then the Ping SDKs fall back to use a browser redirect for social sign on.

The Ping SDK for Android supports the following native libraries:

- Facebook
- Google

Implementing the Facebook native sign-in SDK

To use Facebook's native SDK in your Android app, complete the following tasks:

Add Facebook SDK for Android dependencies

1. In addition to the Ping SDKs dependencies, add Facebook's SDK library to the `dependencies` section of your `build.gradle.kts` file:

```
// Facebook native sign-on SDK for Android
implementation("com.facebook.android:facebook-login:18.0.3")
```

2. In Android Studio, open your `/res/values/strings.xml` file, and add the following to the `<resources>` element:

```
<resources>

    <!-- Other resources... -->

    <string name="facebook_app_id">app_id</string>
    <string name="fb_login_protocol_scheme">fb[app_id]</string>
    <string name="facebook_client_token">client_token</string>

</resources>
```

Replace the placeholders with values from the application you created in the Meta Developer site.

app_id

Click the **App ID** label in the header bar of the Meta Developer site for your app to copy the value.

Add your application ID to both the `facebook_app_id` and `fb_login_protocol_scheme` properties.

client_token

In the Meta Developer site for your app, navigate to **App Settings** > **Advanced** > **Security**, and copy the **Client token** value.

> **Important**
>
> Do not use the **App secret** value found in **App settings** > **Basic** in your client applications.

The result resembles the following:

```
<resources>

    <!-- Other resources... -->

    <string name="facebook_app_id">1085352047332439</string>
    <string name="fb_login_protocol_scheme">fb[1085352047332439]</string>
    <string name="facebook_client_token">3399464ch1515ace3c9782ad1fbeef101</string>

</resources>
```

3. In your `/app/manifests/AndroidManifest.xml` file, add the following:

```
<activity
    android:name="com.facebook.CustomTabActivity"
    android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.VIEW"/>

        <category android:name="android.intent.category.DEFAULT"/>
        <category android:name="android.intent.category.BROWSABLE"/>

        <data android:scheme="@string/fb_login_protocol_scheme"/>
    </intent-filter>
</activity>
```

Learn more in Facebook Login for Android - Quickstart in the Meta Developers documentation.

Add key hashes to your Facebook App ID

You need to associate your Android app with your Facebook App ID. You can update your existing Facebook App ID in the Meta Developer console by adding the key hashes of your signing certificates.

1. In the Meta for Developers console, select your app, and then navigate to **App Settings** > **Basic**.
2. In the **Android** section, in the **Key hashes** field, enter the 28-character hash of your project's signing certificates.

   You should add both debug and production hashes.

Tip

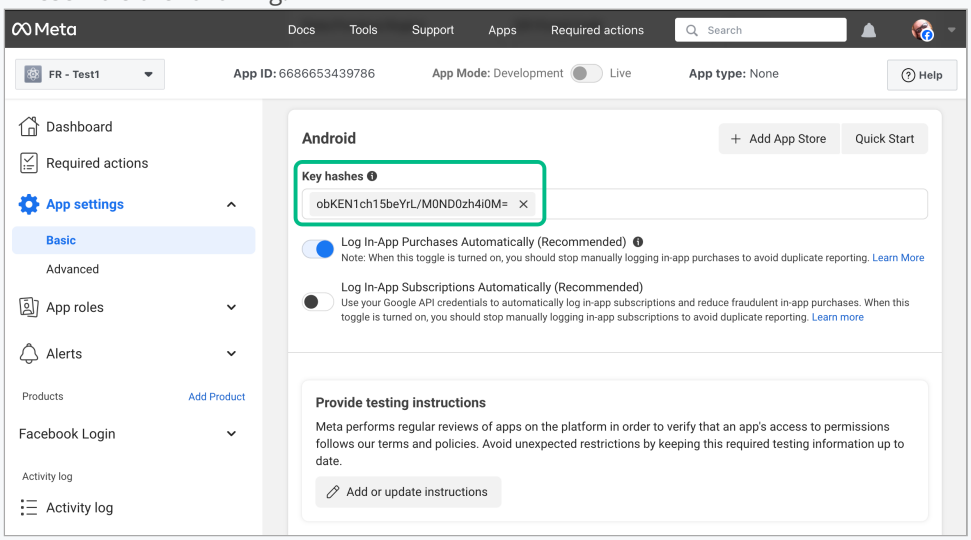
If you are self-signing your app, use the following command in a terminal window in the root of your Android project to generate the hash:

```
keytool -exportcert -alias key-alias -keystore path-to-debug-or-production-keystore | openssl sha1
-binary | openssl base64
```

You can get the values for *key-alias* and *path-to-debug-or-production-keystore* from the `signingConfigs` object in your project's `build.gradle.kts` file:

```
signingConfigs {
  getByName("debug") {
    storeFile = file("../debug.jks")
    storePassword = "android"
    keyAlias = "androiddebugkey"
    keyPassword = "android"
  }
}
```

The result will resemble the following:



Learn more in Create a Development Key Hash and Create a Release Key Hash in the Facebook SDK for Android documentation.

Implementing the Sign in with Google native SDK

To use the Sign in with Google native SDK in your Android app, complete the following tasks:

Add Sign in with Google dependencies

1. In addition to the core Ping SDK for Android dependencies, add Google's SDK libraries to the `dependencies` section of your `build.gradle.kts` file:

```
// Google native sign-on SDK for Android
implementation("com.google.android.libraries.identity.googleid:googleid:1.1.1")

// Needed by Android 13 and earlier
implementation("androidx.credentials:credentials-play-services-auth:1.5.0")
```

> **Note**
>
> If your application will only support Android 14 and later you can remove the `credentials-play-services-auth` dependency.
> Only apps running on Android 13 and earlier require that dependency to support social sign on.

Create client ID credentials for Android

1. In a browser, navigate to the Google's API Dashboard.
2. In the left navigation, click **Credentials**.
3. Click **CREATE CREDENTIALS**, and from the drop-down list, select `OAuth client ID`.
4. In the **Application Type** drop-down list, select `Android`.
5. In the **Name** field, enter a name for your app.
6. In the **Package name** field, enter the package name of your app.

   For example, `com.pingidentity.samples.app`.
7. In the **SHA-1 certificate fingerprint** field, enter the SHA-1 fingerprint of your project's signing certificate.

> **Tip**
>
> If you are self-signing your app, use the following command in a terminal window to get the fingerprint:
>
> ```
> keytool -keystore path-to-debug-or-production-keystore -list -v
> ```
>
> Learn more in Authenticating Your Client in the Google Play Store documentation.

The result will resemble the following:

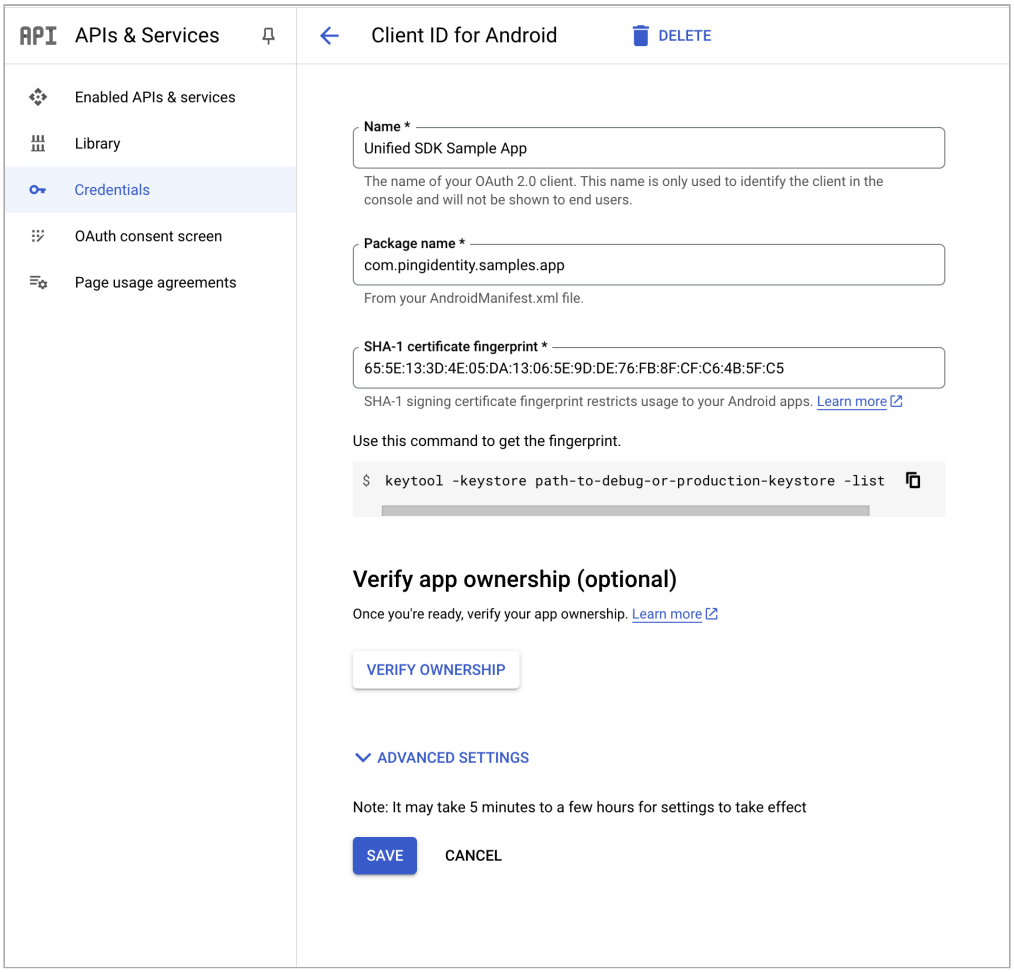


Figure 1. Configuring a client ID for Android apps in Google Cloud

8. Click **Create**.

> **Note**
>
> You do not need to configure your Android app or the PingOne server with the details of this client ID. The certificate fingerprint associates your app with the client ID.

Step 3. Handle the redirect scheme

You must configure your Android app to open when the server redirects the user to the custom URI scheme you entered when setting up PingOne.

For this tutorial, the custom URI scheme is `myapp://example.com`.

To configure your app to open when using the custom URI scheme:

1. In the **Project** tree view of your Android Studio project, open the `build.gradle.kts` file.
2. In the `android.defaultConfig` section, add a manifest placeholder for the `appRedirectUriScheme` property that specifies the protocol of the custom schema:

```
android {
    defaultConfig {
        manifestPlaceholders["appRedirectUriScheme"] = "myapp"
    }
}
```

Step 4. Handling IdpCollector nodes

Your app must handle the `IdpCollector` node type that DaVinci sends when a user attempts to authenticate using an external IdP.

When encountering an `IdpCollector` node type, call `idpCollector.authorize()` to begin authentication with the external IdP:

```
var node = daVinci.start()

if (node is ContinueNode) {
    node.collectors.forEach {
        when (it) {
            is IdpCollector -> {
                when (val result = idpCollector.authorize()) {
                    is Success -> {
                        // When success, move to next Node
                        node.next()
                    }
                    is Failure -> {
                        // Handle the failure
                    }
                }
            }
        }
    }
}
```

The `idpCollector.authorize()` method returns a `Success` result when authentication with the external IdP completes successfully. If not, it returns `Failure` and `Throwable` which shows the root cause of the issue.

```
val result = idpCollector.authorize()

result.onSuccess {
    // Move to next Node
}
result.onFailure {
    it // The Throwable
}
```

The result resembles the following:

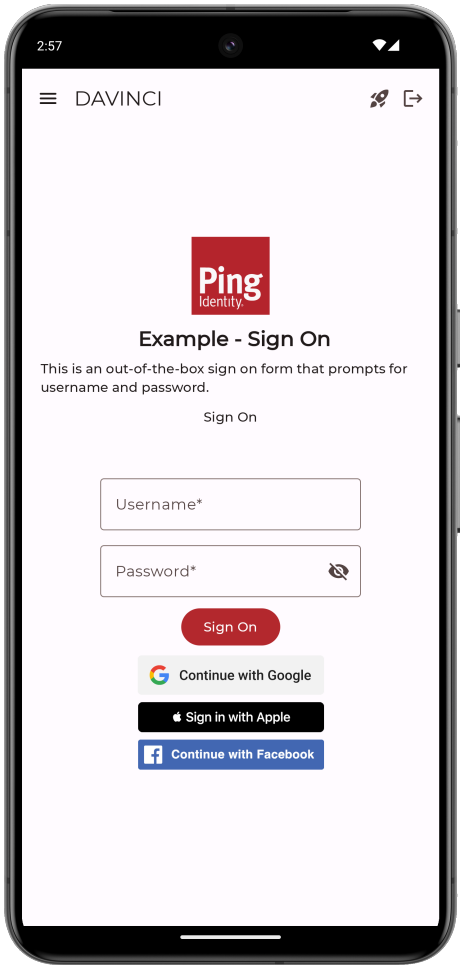


Figure 2. An Android app with three external IdP options: Google, Apple, and Facebook.

Configure an iOS app for social sign on

Perform the following steps to configure an iOS app for social sign on using PingOne.

Step 1. Add the dependencies

You must add the `davinci` and `external-idp` modules to your project. You can use either Cocoapods or Swift Package Manager (SPM) to add the dependencies.

You can use CocoaPods or the Swift Package Manager to add the PingOne Protect dependencies to your iOS project.

Add dependencies using CocoaPods

1. If you do not already have CocoaPods, install the latest version.
2. If you do not already have a Podfile, in a terminal window, run the following command to create a new Podfile:

```
pod init
```

3. Add the following lines to your Podfile:

```
pod 'PingDavinci'
pod 'External-idp'
```

4. Run the following command to install pods:

```
pod install
```

Add dependencies using Swift Package Manager

1. With your project open in **Xcode**, select **File** > **Add Package Dependencies**.
2. In the search bar, enter the Ping SDK for iOS repository URL: `https://github.com/ForgeRock/ping-ios-sdk`.
3. Select the `ping-ios-sdk` package, and then click **Add Package**.
4. In the **Choose Package Products** dialog, ensure that the `PingDavinci` and `PingExternalIdp` libraries are added to your target project.
5. Click **Add Package**.
6. In your project, import the library:

```
import PingDavinci
import PingExternalIdp
```

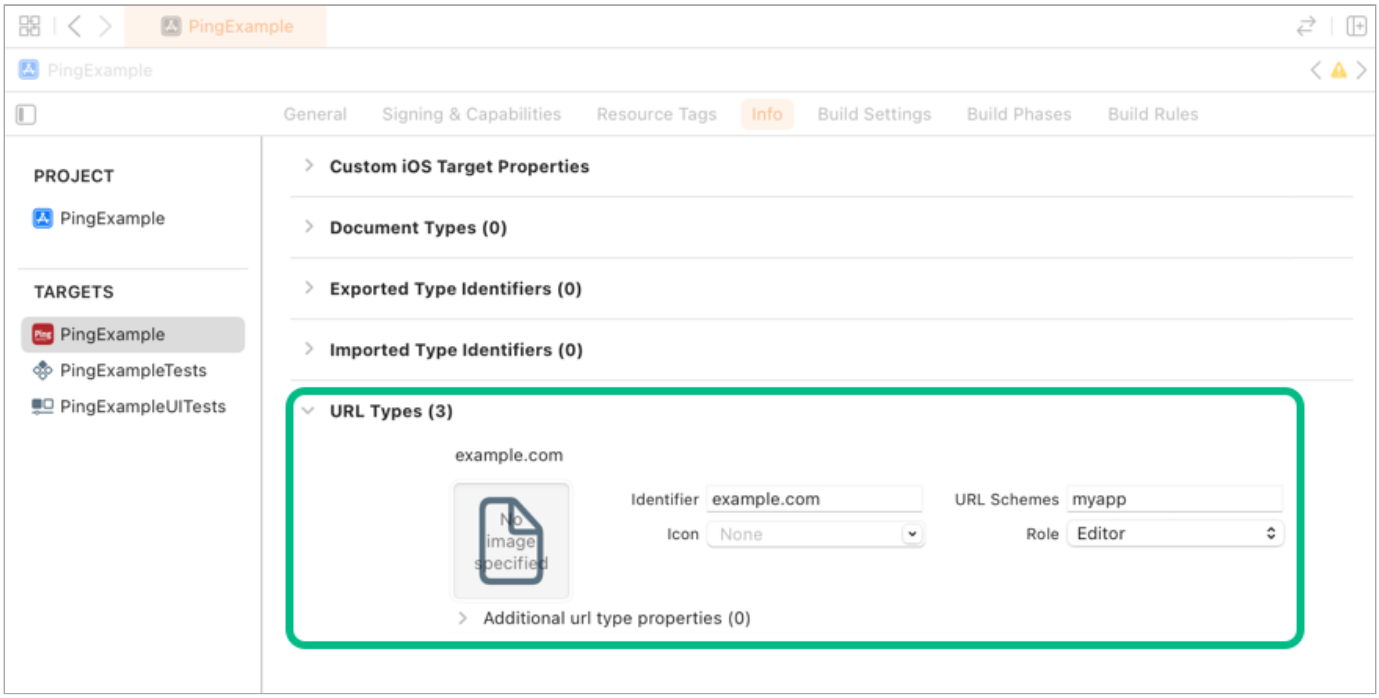
Step 2. Handle the redirect scheme

You must configure your iOS app to open when the server redirects the user to the custom URI scheme you entered when setting up PingOne.

For this tutorial, the custom URI scheme is `myapp://example.com`.

To configure your app to open when using the custom URI scheme:

1. In Xcode, in the **Project Navigator**, double-click your application to open the **Project** pane.
2. On the **Info** tab, in the **URL Types** panel, configure your custom URL scheme:



Step 3. Handling IdpCollector nodes

Your app must handle the `IdpCollector` node type that DaVinci sends when a user attempts to authenticate using an external IdP.

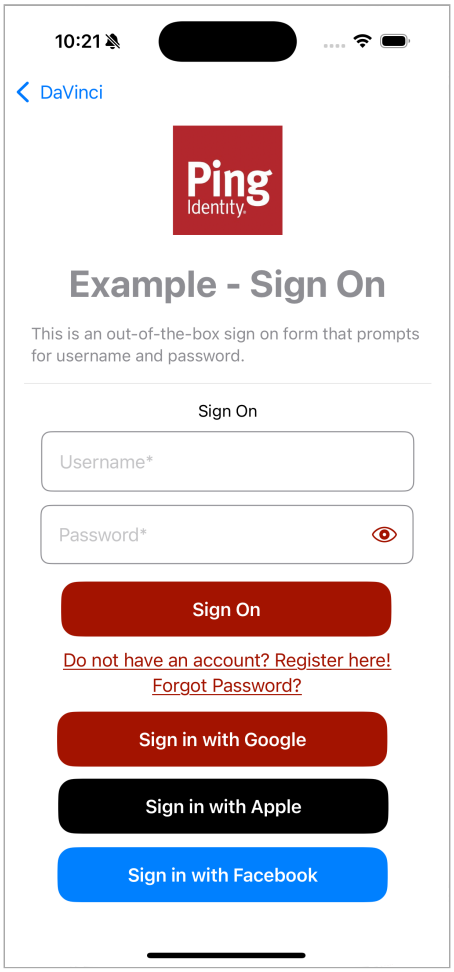
When encountering an `IdpCollector` node type, call `idpCollector.authorize()` to launch an in-app browser and begin authentication with the external IdP:

```
public class SocialButtonViewModel: ObservableObject {
    @Published public var isComplete: Bool = false
    public let idpCollector: IdpCollector

    public init(idpCollector: IdpCollector) {
        self.idpCollector = idpCollector
    }

    public func startSocialAuthentication() async → Result<Bool, IdpExceptions> {
        return await idpCollector.authorize()
    }

    public func socialButtonText() → some View {
        let bgColor: Color
        switch idpCollector.idpType {
        case "APPLE":
            bgColor = Color.appleButtonBackground
        case "GOOGLE":
            bgColor = Color.googleButtonBackground
        case "FACEBOOK":
            bgColor = Color.facebookButtonBackground
        default:
            bgColor = Color.themeButtonBackground
        }
        let text = Text(idpCollector.label)
            .font(.headline)
            .foregroundColor(.white)
            .padding()
            .frame(width: 300, height: 50)
            .background(bgColor)
            .cornerRadius(15.0)

        return text
    }
}
```

The `idpCollector.authorize()` method returns a `Success` result when authentication with the external IdP completes successfully. If not, it returns `Failure` and `IdpExceptions`, which shows the root cause of the issue.

```
Task {
    let result = await socialButtonViewModel.startSocialAuthentication()
    switch result {
    case .success(_):
        onNext(true)
    case .failure(let error): //<- Exception
        onStart()
    }
}
```

The result resembles the following:



Configure a JavaScript app for social sign on

Perform the following steps to configure a JavaScript app for social sign on using PingOne.

Step 1. Add the module

You must add the `davinci` module to your project:

```
import { davinci } from '@forgerock/davinci-client';
```

Step 2. Handle the redirect back from the IdP

You must configure your JavaScript app to continue a flow when the server redirects the user back from the IdP.

Use the `davinciClient.resume` method to continue an existing flow, rather than start a new one.

```
const davinciClient = await davinci({ config });
const continueToken = urlParams.get('continueToken');
let resumed: any;

if (continueToken) {
  // Continue an existing flow
  resumed = await davinciClient.resume({ continueToken });
} else {
  // Setup configuration for a new flow
  await Config.setAsync(config);
}
```

Step 3. Handling IdpCollector nodes

Your app must handle the `IdpCollector` node type that DaVinci sends. The node contains details of the button to render and the URL, for example.

Use the `davinciClient.externalIdp()` method to obtain the details from the collector:

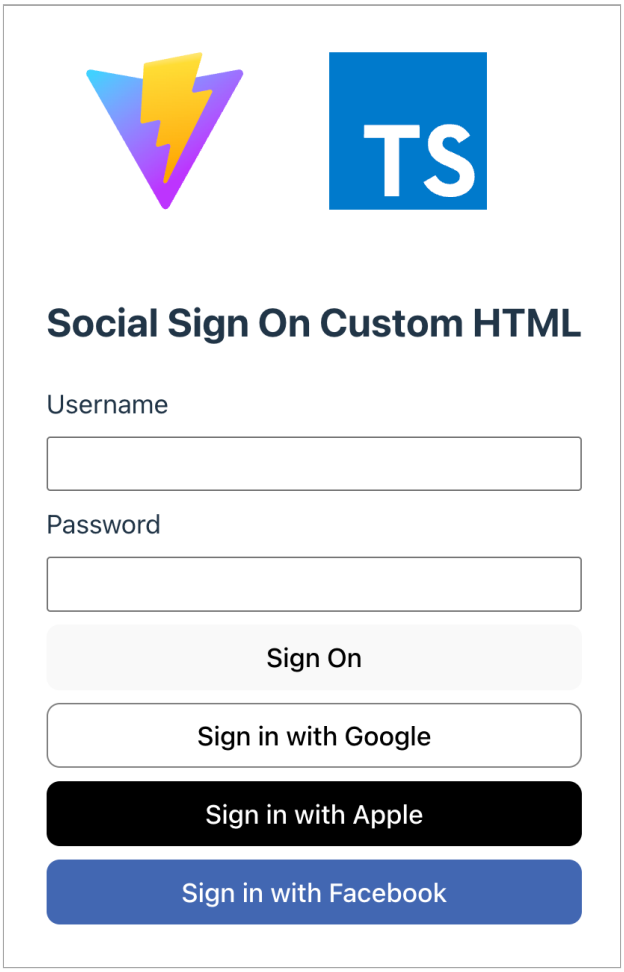
```
const collectors = davinciClient.getCollectors();

collectors.forEach((collector) => {
  if (collector.type === 'IdpCollector') {
    socialLoginButtonComponent(formEl, collector, davinciClient.externalIdp(collector));
  }
}
```

In this example, a `socialLoginButtonComponent` handles rendering the button:

Example `social-login-button.ts` file to render social sign-on buttons

```
import type { IdpCollector } from "@forgerock/davinci-client/types";

export default function submitButtonComponent(
  formEl: HTMLFormElement,
  collector: IdpCollector,
  updater: () => void
) {
  const button = document.createElement("button");

  button.value = collector.output.label;
  button.innerHTML = collector.output.label;

  if (collector.output.label.toLowerCase().includes('google')) {
    button.style.background = 'white'
    button.style.borderColor = 'grey'
  } else if (collector.output.label.toLowerCase().includes('facebook')) {
    button.style.color = 'white'
    button.style.background = 'royalblue'
    button.style.borderColor = 'royalblue'
  } else if (collector.output.label.toLowerCase().includes('apple')) {
    button.style.color = 'white'
    button.style.background = 'black'
    button.style.borderColor = 'black'
  }

  button.onclick = () => updater();

  formEl?.appendChild(button);
}
```

The result resembles the following:



DaVinci client API reference

View API references for the different modules provided in the DaVinci client.

Android

- DaVinci Client for Android API Reference

iOS

- Browser module
- DaVinci module
- External IDP module
- Logger module
- OIDC module
- Orchestrate module
- Storage module

JavaScript

- DaVinci Client for JavaScript API Reference

Introducing the Ping SDKs for OIDC login

Server support:

- ✓ PingOne
- ✓ PingOne Advanced Identity Cloud
- ✓ PingAM
- ✓ PingFederate

SDK support:

- ✓ Ping SDK for Android
- ✓ Ping SDK for iOS
- ✓ Ping SDK for JavaScript

The Ping SDKs can help you to login to your authorization server using an OpenID Connect flow, and leveraging the server's own UI to authenticate your users in your apps.

We call this **OIDC login**, but it was previously known as **centralized login**.

With this option, you reuse the same, centralized UI for login requests in multiple apps and sites.

When a user attempts to log in to your app they are redirected to your server's central login UI. After the user authenticates, they are redirected back to your app.

Changes to authentication journeys or DaVinci flows are immediately reflected in all apps that use OIDC login without the need to rebuild or update the client app.

Likewise, any features your server's UI supports are also available for use in your web or mobile apps.

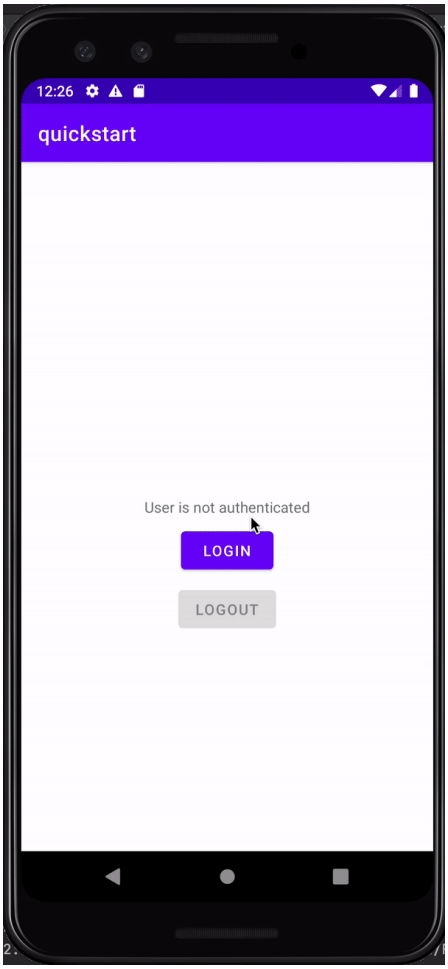


Figure 1. Centralized login in Android

Use cases

✓ If you require a consistent UI and user experience (UX) in all your apps and sites, using centralized login may be the best option.

✓ Simple branding and control over your authentication UX is sufficient.

✓ Your mobile apps use browser-based single sign-on.

Security considerations

- ✓ Using centralized login in apps built by a third party is safer than using embedded login.
- ✓ Third parties cannot access user credentials.
- ✓ User credentials are authenticated in one domain/origin and not sent elsewhere for authentication.
- ✓ Your apps and sites can use browser-based single sign-on.

Next steps

Configure the Ping SDKs for OIDC login

Tutorials

Configure the Ping SDKs for OIDC login

The Ping SDKs are designed to be flexible and can be customized to suit many different situations.

Learn more about configuring and customizing the Ping SDKs in the sections below:

Configure OIDC login

Learn how to configure your apps to use your authorization server's UI or your own web application for login requests.

Learn more »

Configure ACR parameters

Utilize an Authentication Context Class Reference (ACR) parameter to choose which authentication journey or DaVinci flow the user should complete.

Learn more »

Configure OIDC login

You can configure your apps to use your authorization server's UI or your own web application for login requests.

When a user attempts to log in to your app it redirects them to the central login UI. After the user authenticates, the authorization server redirects them back to your application or site.

Changes to authentication journeys or flows on your authorization server are available to all your apps that use the OIDC login method. Your app does not need to access user credentials directly, just the result of the authentication from the server - usually an access token.

Select your platform below to learn how to configure your app to use OIDC login:

Android

Configure Ping SDK for Android apps to perform OIDC login.

iOS

Configure Ping SDK for iOS apps to perform OIDC login.

JavaScript

Configure Ping SDK for JavaScript apps to perform OIDC login.

Configure Android apps for OIDC login

This section describes how to configure your Ping SDK for Android application to use centralized login by leveraging the `AppAuth` library:

1. Add the build dependency to the `build.gradle` file:

```
implementation 'net.openid:appauth:0.11.1'
```

2. Associate your application with the scheme your redirect URIs use.

   To ensure that only your app is able to obtain authorization tokens during centralized login we recommend you configure it to use Android App Links.

   If you do not want to implement Android App Links, you can instead use a custom scheme for your redirect URIs.

Android App Links

Complete the following steps to configure App Links:

1. In your application, configure the AppAuth library to use the HTTP scheme for capturing redirect URIs, by adding an `<intent-filter>` for `AppAuth.RedirectUriReceiverActivity` to your `AndroidManifest.xml`:

   AndroidManifest.xml

   ```
   <activity
      android:name="net.openid.appauth.RedirectUriReceiverActivity"
      android:exported="true"
      tools:node="replace">
      <intent-filter android:autoVerify="true">
          <action android:name="android.intent.action.VIEW" />

          <category android:name="android.intent.category.DEFAULT" />
          <category android:name="android.intent.category.BROWSABLE" />

          <data android:scheme="https" />
          <data android:host="android.example.com" />
          <data android:path="/oauth2redirect" />
      </intent-filter>
   </activity>
   ```

   - You must set `android:autoVerify` to `true`. This instructs Android to verify the against the `assetlinks.json` file you update in the next step.
   - Specify the `scheme`, `hosts`, and `path` parameters that will be used in your redirect URIs. The host value must match the domain where you upload the `assetlinks.json` file.

   To learn more about intents, refer to Add intent filters in the Android Developer documentation.

   To learn more about redirects and the AppAuth library, refer to Capturing the authorization redirect.

2. For Android 11 or higher, add the following to the `AndroidManfest.xml` file:

   ```
   <queries>
        <intent>
            <action android:name="android.intent.action.VIEW" />
            <category android:name="android.intent.category.BROWSABLE" />
            <data android:scheme="https" />
        </intent>
    </queries>
   ```

3. Create or update a Digital Asset Links (`assetlinks.json`) file that associates your app with the domain.

   You must host the file in a `.well-known` folder on the same host that you entered in the intent filter earlier.

The file will resemble the following:

https://android.example.com/.well-known/assetlinks.json

```
[
  {
    "relation": [
      "delegate_permission/common.handle_all_urls",
    ],
    "target": {
      "namespace": "android_app",
      "package_name": "com.example.app",
      "sha256_cert_fingerprints": [
        "c4:15:c8:f1:...:fe:ce:d7:37"
      ]
    }
  }
]
```

- To learn more, refer to Associate your app with your website in the Android Developer documentation.

4. Upload the completed file to the domain that matches the host value you configured in the earlier step.

   For information on uploading an `assetLinks.json` file to an Advanced PingOne Advanced Identity Cloud instance, refer to Upload an Android assetlinks.json file.

5. Add the following to the `strings.xml` file:

```
<string name="forgerock_oauth_redirect_uri" translatable="false">https://android.example.com/
oauth2redirect</string>
```

6. Add the App Link to the **Redirection URIs** property of your OAuth 2.0 client. For example, `https://android.example.com/oauth2redirect`

Custom Scheme

Complete the following steps to configure a custom scheme:

1. Configure the AppAuth library to use the custom scheme for capturing redirect URIs by using either of these two methods:

   - Add the custom scheme your app will use to your `build.gradle` file:

   ```
   android.defaultConfig.manifestPlaceholders = [
       'appAuthRedirectScheme': 'com.forgerock.android'
   ]
   ```

   Or:

   - Add an `<intent-filter>` for `AppAuth.RedirectUriReceiverActivity` to your `AndroidManifest.xml`:

   ```
   <activity
      android:name="net.openid.appauth.RedirectUriReceiverActivity"
      tools:node="replace">
       <intent-filter>
          <action android:name="android.intent.action.VIEW"/>
          <category android:name="android.intent.category.DEFAULT"/>
          <category android:name="android.intent.category.BROWSABLE"/>
          <data android:scheme="com.forgerock.android"/>
       </intent-filter>
   </activity>
   ```

   For more information, refer to Capturing the authorization redirect.

2. For Android 11 or higher, add the following to the `AndroidManfest.xml` file:

   ```
   <queries>
        <intent>
            <action android:name="android.intent.action.VIEW" />
            <category android:name="android.intent.category.BROWSABLE" />
            <data android:scheme="com.forgerock.android" />
        </intent>
    </queries>
   ```

3. Configure your application to use the redirect URI, either in the `strings.xml` file, or by using `FROptions`:

   strings.xml:

   ```
   <string name="forgerock_oauth_redirect_uri" translatable="false">org.forgerock.demo://
   oauth2redirect</string>
   ```

FROptions:

```
let options = FROptions(
    ...,
    oauthRedirectUri: "org.forgerock.demo://oauth2redirect",
    ...,
)
```

4. Add the custom scheme to the **Redirection URIs** property of your OAuth 2.0 client. For example, `org.forgerock.demo://oauth2redirect`

3. Configure your application to use browser mode:

```
// Use FRUser.browser() to enable browser mode:
FRUser.browser().login(context, new FRListener<FRUser>());

// Use standard SDK interface to retrieve an AccessToken:
FRUser.getCurrentUser().getAccessToken()

// Use standard SDK interface to logout a user:
FRUser.getCurrentUser().logout()
```

Tip

The SDK uses the OAuth 2.0 parameters you configured in your application.
You can amend the example code above to customize the integration with AppAuth; for example, adding OAuth 2.0 or OpenID Connect parameters, and browser colors:

```
FRUser.browser().appAuthConfigurer()
    .authorizationRequest(r -> {
        // Add a login hint parameter about the user:
        r.setLoginHint("demo@example.com");
        // Request that the user re-authenticates:
        r.setPrompt("login");
    })
    .customTabsIntent(t -> {
        // Customize the browser:
        t.setShowTitle(true);
        t.setToolbarColor(getResources().getColor(R.color.colorAccent));
    }).done()
    .login(this, new FRListener<FRUser>() {
        @Override
        public void onSuccess(FRUser result) {
            //success
        }

        @Override
        public void onException(Exception e) {
            //fail
        }
    });
```

Configure iOS apps for OIDC login

This section describes how to configure your Ping SDK for iOS application to use centralized login:

1. Associate your application with the scheme your redirect URIs use.

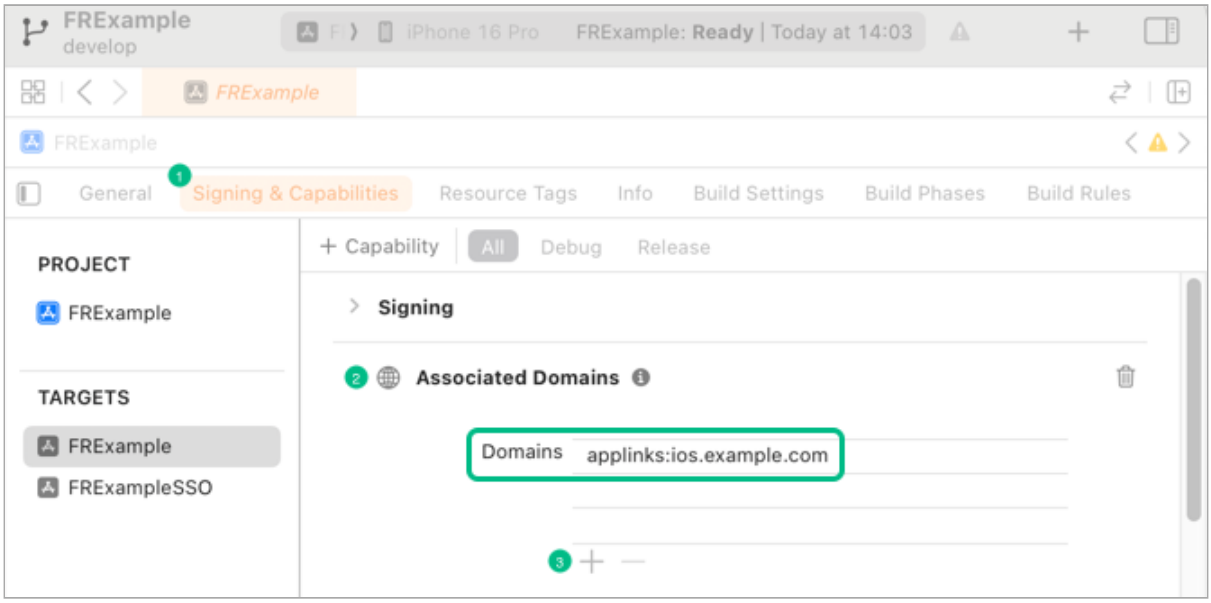
   To ensure that only your app is able to obtain authorization tokens during centralized login we recommend you configure it to use Universal Links.

   If you do not want to implement Universal Links, you can instead use a custom scheme for your redirect URIs.

Apple Universal Links

Complete the following steps to configure Universal Links:

1. In Xcode, in the **Project Navigator**, double-click your application to open the **Project** pane.
2. On the **Signing & Capabilities** tab, click **+ Capability**, type `Associated Domains`, and then double click the result to add the capability.
3. In **Domains**, click the **Add** (+) button, and enter `applinks:`, followed by the hostname that will be used in your redirect URIs.



   The host value must match the domain where you upload the `apple-app-site-association` file.

4. Create or update an `apple-app-site-association` file that associates your app with the domain.

   You must host the file in a `.well-known` folder on the same host that you entered in the intent filter earlier.

The file will resemble the following:

https://ios.example.com/.well-known/apple-app-site-association

```
{
  "applinks": {
      "details": [
           {
             "appIDs": [ "XXXXXXXXXX.com.example.AppName" ],
             "components": [
               {
                  "/": "/oauth2redirect",
                  "comment": "Associate my app with the OAuth 2.0 redirect URI."
               }
             ]
           }
       ]
   }
}
```

5. Upload the completed file to the domain that matches the host value you configured in the earlier step.

   For information on uploading an `apple-app-site-association` file to an Advanced PingOne Advanced Identity Cloud instance, refer to Upload an iOS apple-app-site-association file.

   For learn more information about Universal Links and associating domains, refer to the following in the Apple Developer documentation:

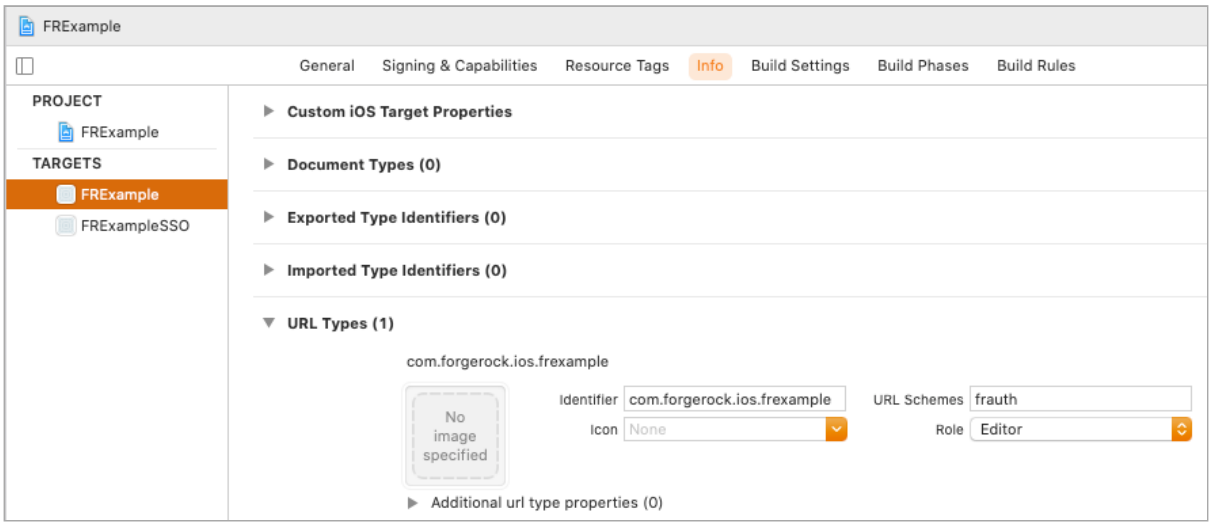
   - Supporting universal links in your app
   - Supporting associated domains

6. Add the Universal Link to the **Redirection URIs** property of your OAuth 2.0 client. For example, `https://ios.example.com/oauth2redirect`

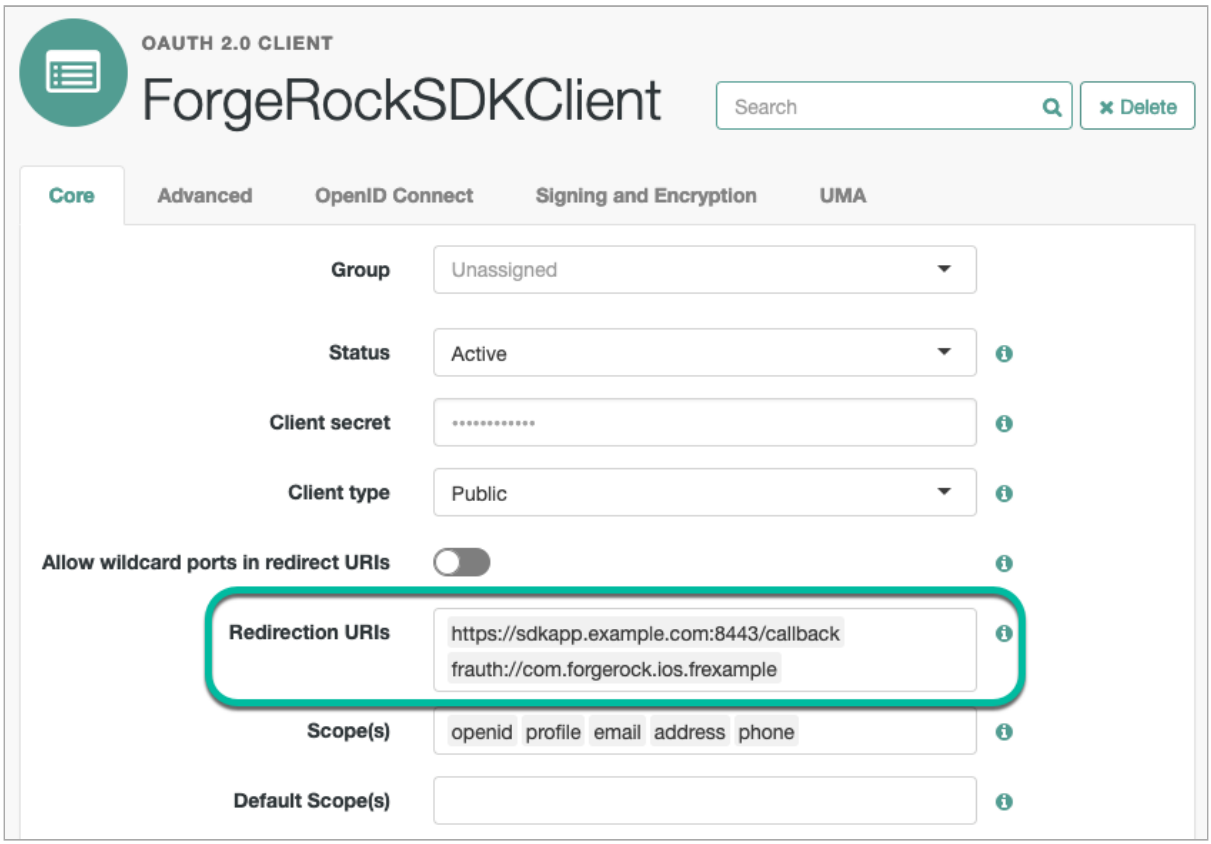
Custom scheme

Configure a custom URL type, for example `frauth`, so that users are redirected to your application:

1. In Xcode, in the **Project Navigator**, double-click your application to open the **Project** pane.
2. On the **Info** tab, in the **URL Types** panel, configure your custom URL scheme:



3. Add the custom URL scheme to the Redirection URIs property of your OAuth 2.0 client:



2. Update your application to call the `validateBrowserLogin()` function:

    1. In your `AppDelegate.swift` file, call the `validateBrowserLogin()` function:

    AppDelegate.swift

    ```
    class AppDelegate: UIResponder, UIApplicationDelegate {

      func application(_ app: UIApplication, open url: URL, options: [UIApplication.OpenURLOptionsKey: Any]
    = [:]) -> Bool {
        // Parse and validate URL, extract authorization code, and continue the flow:
        Browser.validateBrowserLogin(url)
      }
    }
    ```

    2. If you are using Universal Links, also add code similar to the following to set the URL:

    AppDelegate.swift

    ```
    func application(
      _ application: UIApplication,
      continue userActivity: NSUserActivity,
      restorationHandler:
      @escaping ([UIUserActivityRestoring]?) -> Void) -> Bool
      {
        // Get URL components from the incoming user activity.
        guard userActivity.activityType == NSUserActivityTypeBrowsingWeb,
        let incomingURL = userActivity.webpageURL else {
          return false
        }
        Browser.validateBrowserLogin(url)
      }
    )
    ```

    3. If your application is using `SceneDelegate`, in your `SceneDelegate.swift` file call the `validateBrowserLogin()` function:

    SceneDelegate.swift

    ```
    class SceneDelegate: UIResponder, UIWindowSceneDelegate {

      func scene(_ scene: UIScene, openURLContexts URLContexts: Set<UIOpenURLContext>) {
        if let url = URLContexts.first?.url {
          Browser.validateBrowserLogin(url)
        }
      }
    }
    ```

3. To enable centralized login, add code similar to the following to your app:

```
//  BrowserBuilder
let browserBuilder = FRUser.browser()
browserBuilder.set(presentingViewController: self)
browserBuilder.set(browserType: .authSession)
browserBuilder.setCustomParam(key: "custom_key", value: "custom_val")

//  Browser
let browser = browserBuilder.build()

// Login
browser.login{ (user, error) in
  if let error = error {
    // Handle error
  }
  else if let user = user {
    // Handle authenticated status
  }
}
```

You can specify what type of browser the client iOS device opens to handle centralized login.

Each browser has slightly different characteristics, which make them suitable to different scenarios, as outlined in this table:

Browser type	Characteristics
`.authSession`	Opens a web authentication session browser. Designed specifically for authentication sessions, however it prompts the user before opening the browser with a modal that asks them to confirm the domain is allowed to authenticate them. This is the default option in the Ping SDK for iOS.
`.ephemeralAuthSession`	Opens a web authentication session browser, but enables the `prefersEphemeralWebBrowserSession` parameter. This browser type *does not* prompt the user before opening the browser with a modal. The difference between this and `.authSession` is that the browser does not include any existing data such as cookies in the request, and also discards any data obtained during the browser session, including any session tokens. When is `ephemeralAuthSession` suitable: ◦ ✕ `ephemeralAuthSession` is *not* suitable when you require single sign-on (SSO) between your iOS apps, as the browser will not maintain session tokens. ◦ ✕ `ephemeralAuthSession` is *not* suitable when you require a session token to log a user out of the server, for example for logging out of PingOne, as the browser will not maintain session tokens. ◦ ✓ Use `ephemeralAuthSession` when you do not want the user's existing sessions to affect the authentication.

Browser type	Characteristics
`.nativeBrowserApp`	Opens the installed browser that is marked as the default by the user. Often Safari. The browser opens without any interaction from the user. However, the browser does display a modal when returning to your application.
`.sfViewController`	Opens a Safari view controller browser. Your client app is ***not*** able to interact with the pages in the `sfViewController` or access the data or browsing history. The view controller opens within your app without any interaction from the user. As the user does not leave your app, the view controller does not need to display a warning modal when authentication is complete and control returns to your application.

Configure JavaScript apps for OIDC login

This section describes how to configure your Ping SDK for JavaScript application with centralized login:

1. To initiate authentication using the OIDC-based centralized login method, call the `getTokens()` function:

```
const tokens = TokenManager.getTokens({
  login: 'redirect', // Redirect to the server or the web app that handles authentication
  forceRenew: false, // Immediately return stored tokens, if they exist
  skipBackgroundRequest: true // Regardless of session status, redirect to the authorization server to
initiate the OAuth 2.0 flow
});
```

The parameters for `getTokens()` are as follows:

`login`

Specifies the method the Ping SDK for JavaScript uses to handle authentication.

Supported values are as follows:

Setting	Description
`redirect`	Your app uses a redirect to your server, or another web application, to handle authentication. Use this option to perform centralized login.
`embedded`	Your app handles authentication natively using SDK functionality, and provides the user interface.

If you do not specify a value, `embedded` is assumed, for backwards-compatibility.

`forceRenew`

When `true`, the Ping SDK for JavaScript revokes and deletes any existing tokens it has and contacts your authorization server to obtain new tokens.

When `false`, or not specified, if existing tokens are present the Ping SDK returns these immediately for use. Otherwise it contacts your authorization server to obtain new tokens.

`skipBackgroundRequest`

When `true`, the Ping SDK for JavaScript redirects users immediately to your authorization server.

When `false`, or not specified, the Ping SDK for JavaScript attempts to obtain OAuth 2.0 tokens within an iframe to prevent unnecessary page redirects.

However requesting tokens in an iframe can cause errors with some authorization servers in certain environments. In these cases, we recommend setting `skipBackgroundRequest` to `true`.

2. When the user is returned to your app, complete the OAuth 2.0 flow by passing in the `code` and `state` values that were returned.

   Use the `query` property to complete the flow:

```
const tokens = TokenManager.getTokens({
  query: {
    code: 'lFJQYdoQG1u7nUm8 ... ', // Authorization code from redirect URL
    state: 'MTY2NDkxNTQ2Nde3D ... ', // State from redirect URL
  },
});
```

Specifying auth journeys/flows using ACR values

The Ping SDKs for Android, iOS, and JavaScript leverage the standards-based authorization code flow with PKCE.

When using OIDC login the client app can request which journey or flow the authorization server uses by adding an Authentication Context Class Reference (ACR) parameter during the process.

In the OpenID Connect specification the ACR parameter identifies a set of criteria that the user must satisfy when authenticating to the OpenID provider. For example, which authentication journey or DaVinci flow the user should complete.

Adding ACR parameters

Select your platform below to learn how to add an ACR parameter to your applications.

Android

In the `FRUser.browser()` method, use the `setAdditionalParameters` function to add an `acr_values` parameter, and one or more ACR values:

```
FRUser.browser().appAuthConfigurer()
 .authorizationRequest(r → {
     Map<String, String> additionalParameters = new HashMap<>();
     additionalParameters.put("acr_values", "RegistrationJourney");
     r.setAdditionalParameters(additionalParameters)
 })
 .done()
 .login(this, new FRListener<FRUser>() {
     @Override
     public void onSuccess(FRUser result) {
         userinfo();
     }

     @Override
     public void onException(Exception e) {
         System.out.println(e);
     }
 });
```

Replace *RegistrationJourney* with the ACR key that your authorization server requires.

PingOne

Enter a single DaVinci policy, by using its flow policy ID, or one or more PingOne policies by specifying the policy names, separated by spaces or the encoded space character `%20`.

Examples:

DaVinci flow policy ID

`"d1210a6b0b2665dbaa5b652221badba2"`

PingOne policy names

`"Single_Factor%20Multi_Factor"`

PingOne Advanced Identity Cloud or PingAM

Enter one or more of the ACR mapping keys as configured in the OAuth 2.0 provider service.

To learn more, refer to Configure acr claims.

> **Tip**
>
> You can list the available keys by inspecting the `acr_values_supported` property in the output of your OAuth 2.0 client's `/oauth2/.well-known/openid-configuration` endpoint.

iOS

In the `FRUser.browser()` method, use the `setCustomParam` function to add an `acr_values` key parameter, and one or more ACR values:

```
func performCentralizedLogin() {
     FRUser.browser()?
         .set(presentingViewController: self)
         .set(
             browserType: .authSession)
         #.setCustomParam(
             key: "acr_values",
             value: "RegistrationJourney")
         .build().login { (user, error) in
             self.displayLog("User: \(String(describing: user)) || Error: \(String(describing: error))")
     }
     return
 }
```

Replace *RegistrationJourney* with the ACR key that your authorization server requires.

PingOne

Enter a single DaVinci policy, by using its flow policy ID, or one or more PingOne policies by specifying the policy names, separated by spaces or the encoded space character `%20`.

Examples:

DaVinci flow policy ID

`"d1210a6b0b2665dbaa5b652221badba2"`

PingOne policy names

`"Single_Factor%20Multi_Factor"`

PingOne Advanced Identity Cloud or PingAM

Enter one or more of the ACR mapping keys as configured in the OAuth 2.0 provider service.

To learn more, refer to Configure acr claims.

> **Tip**
>
> You can list the available keys by inspecting the `acr_values_supported` property in the output of your OAuth 2.0 client's `/oauth2/.well-known/openid-configuration` endpoint.

JavaScript

In the `TokenManager.getTokens()` method, add an `acr_values` query parameter, and one or more ACR values:

```
await TokenManager.getTokens({
  login: 'redirect',
  query: {
    acr_values: "RegistrationJourney"
  }
});
```

Replace *RegistrationJourney* with the ACR key that your authorization server requires.

PingOne

Enter a single DaVinci policy, by using its flow policy ID, or one or more PingOne policies by specifying the policy names, separated by spaces or the encoded space character `%20`.

Examples:

DaVinci flow policy ID

```
"d1210a6b0b2665dbaa5b652221badba2"
```

PingOne policy names

```
"Single_Factor%20Multi_Factor"
```

PingOne Advanced Identity Cloud or PingAM

Enter one or more of the ACR mapping keys as configured in the OAuth 2.0 provider service.

To learn more, refer to Configure acr claims.

> **Tip**
>
> You can list the available keys by inspecting the `acr_values_supported` property in the output of your OAuth 2.0 client's `/oauth2/.well-known/openid-configuration` endpoint.

Ping SDK OIDC login tutorials

Learn how your apps can authenticate users with an OpenID Connect flow.

The tutorials show how your apps can leverage the user interface each server provides, centralizing the experience across your apps.

To start, choose the platform to host your app:

Android

Android OIDC login tutorials

iOS

iOS OIDC login tutorials

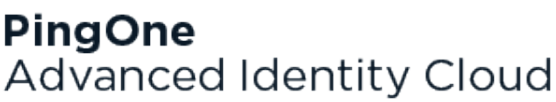
JavaScript

JavaScript OIDC login tutorials

Android OIDC login tutorials

Follow these Android tutorials to integrate your apps using OpenID Connect login to the following servers:

PingOne



PingOne Advanced Identity Cloud

PingAM

PingFederate

OIDC login to PingOne tutorial for Android

Prepare › Download › Configure › Run

In this tutorial you update a sample app that uses OIDC-based login to obtain tokens by redirecting to the PingOne UI for authentication.

The sample connects to the `.well-known` endpoint of your PingOne server to obtain the correct URIs to authenticate the user, and redirects to your PingOne server's login UI.

After authentication, PingOne redirects the browser back to your application, which then obtains an OAuth 2.0 access token and displays the related user information.

Before you begin

Before you begin this tutorial ensure you have set up your PingOne server with the required configuration.

For example, you will need to have an OAuth 2.0 client application set up, and a demo user to authenticate.

Complete prerequisites »

Step 1. Download the samples

To start this tutorial, you need to download the SDK sample apps repo, which contains the projects you will use.

Start step 1 »

Step 2. Configure connection properties

In this step, you configure the sample app to connect to the OAuth 2.0 application you created in PingOne.

Start step 2 »

Step 3. Test the app

To test the app, run the sample that you configured in the previous step.

The sample connects to your PingOne server to obtain the correct URIs to authenticate the user, and redirects the browser to your PingOne server.

After authentication, PingOne redirects the browser back to your application, which then obtains an OAuth 2.0 access token and displays the related user information.

Test app »

Before you begin

Prepare › Download › Configure › Run

To successfully complete this tutorial refer to the prerequisites in this section.

The tutorial also requires a configured PingOne instance.

Compatibility

Android

This sample requires at least Android API 23 (Android 6.0)

Java

This sample requires at least Java 8 (v1.8).

Prerequisites

Android Studio

Download and install Android Studio, which is available for many popular operating systems.

An Android emulator or physical device

To try the quick start application as you develop it, you need an Android device. To add a virtual, emulated Android device to Android Studio, refer to Create and manage virtual devices, on the **Android Developers** website.

Server configuration

This tutorial requires you to configure your PingOne server as follows:

The samples and tutorials in this documentation often require that you have an identity set up so that you can test authentication.

To create a demo user in PingOne, follow these steps:

1. Log in to your PingOne administration console.
2. In the left panel, navigate to **Directory** > **Users**.
3. Next to the **Users** label, click the plus icon (+).

   PingOne displays the **Add User** panel.
4. Enter the following details:
   - **Given Name** = `Demo`
   - **Family Name** = `User`
   - **Username** = `demo`
   - **Email** = `demo.user@example.com`
   - **Population** = `Default`
   - **Password** = `Ch4ng3it!`
5. Click **Save**.

To register a *public* OAuth 2.0 client application in PingOne for use with the Ping SDKs for Android and iOS, follow these steps:

1. Log in to your PingOne administration console.
2. In the left panel, navigate to **Applications** > **Applications**.
3. Next to the **Applications** label, click the plus icon (+).

   PingOne displays the **Add Application** panel.
4. In **Application Name**, enter a name for the profile, for example `sdkNativeClient`
5. Select **Native** as the **Application Type**, and then click **Save**.

6. On the **Configuration** tab, click the pencil icon (✎).
    1. In **Grant Type**, select the following values:

       `Authorization Code`

       `Refresh Token`
    2. In **Redirect URIs**, enter the following value:

       `org.forgerock.demo://oauth2redirect`
    3. In **Token Endpoint Authentication Method**, select `None`.
    4. In the **Advanced Settings** section, enable **Terminate User Session by ID Token**.
    5. Click **Save**.
7. On the **Resources** tab, next to **Allowed Scopes**, click the pencil icon (✎).
    1. In **Scopes**, select the following values:

       `email`

       `phone`

       `profile`

       > **Note**
       >
       > The `openid` scope is selected by default.

       The result resembles the following:

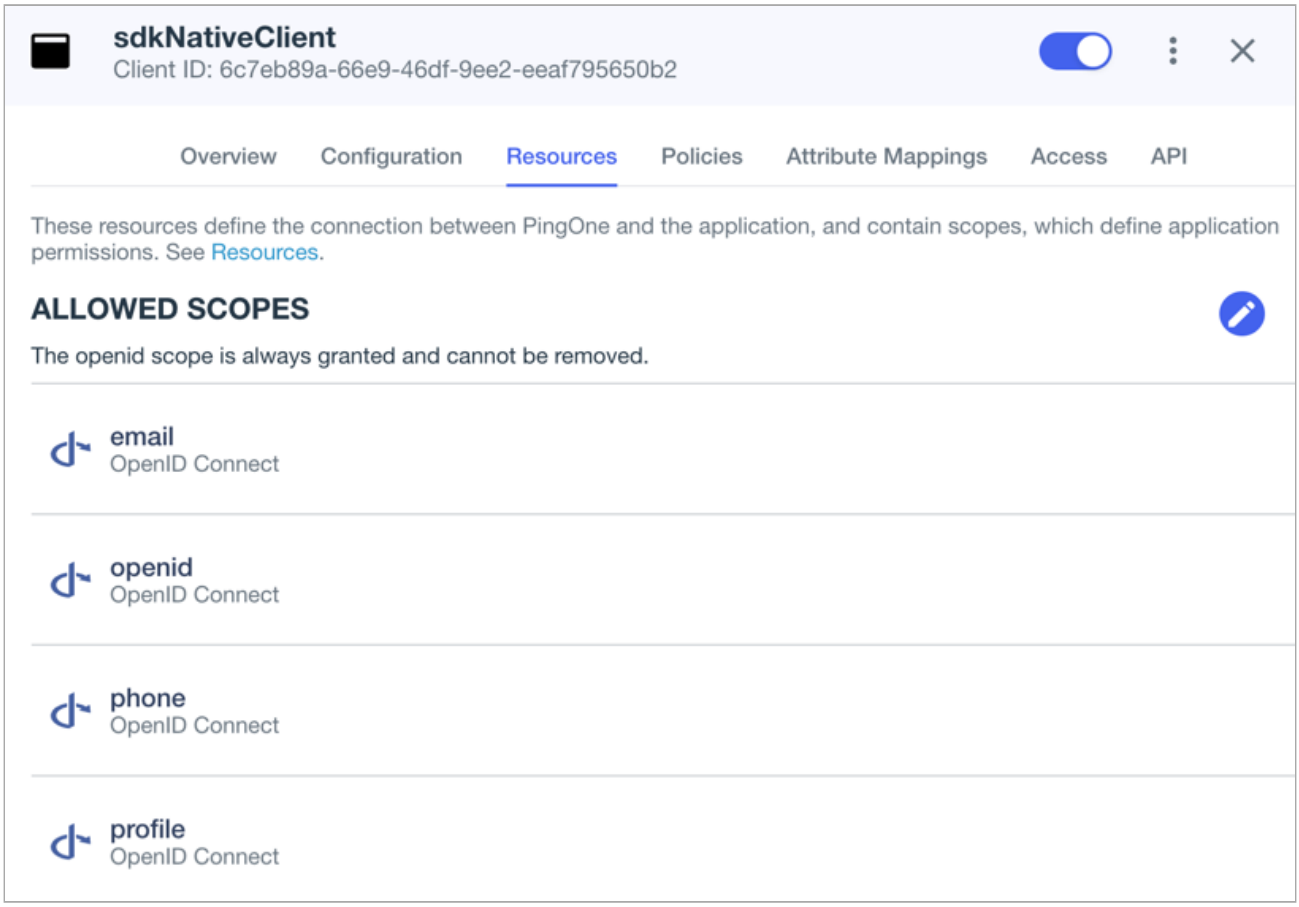


Figure 1. Adding scopes to an application.

8. Optionally, on the **Policies** tab, click the pencil icon (✎) to select the authentication policies for the application.

> **Note**
>
> Applications that have no authentication policy assignments use the environment's default authentication policy to authenticate users.

If you have a DaVinci license, you can select PingOne policies or DaVinci Flow policies, but not both. If you do not have a DaVinci license, the page only displays PingOne policies.

To use a *PingOne policy*:

1. Click **+ Add policies** and then select the policies that you want to apply to the application.
2. Click **Save**.

   PingOne applies the policies in the order in which they appear in the list. PingOne evaluates the first policy in the list first. If the requirements are not met, PingOne moves to the next one.

   For more information, see Authentication policies for applications.

To use a *DaVinci Flow policy*:

1. You must clear all PingOne policies. Click **Deselect all PingOne Policies**.
2. In the confirmation message, click **Continue**.

3. On the **DaVinci Policies** tab, select the policies that you want to apply to the application.
4. Click **Save**.

   PingOne applies the first policy in the list.

9. Click **Save**.
10. Enable the OAuth 2.0 client application by using the toggle next to its name:

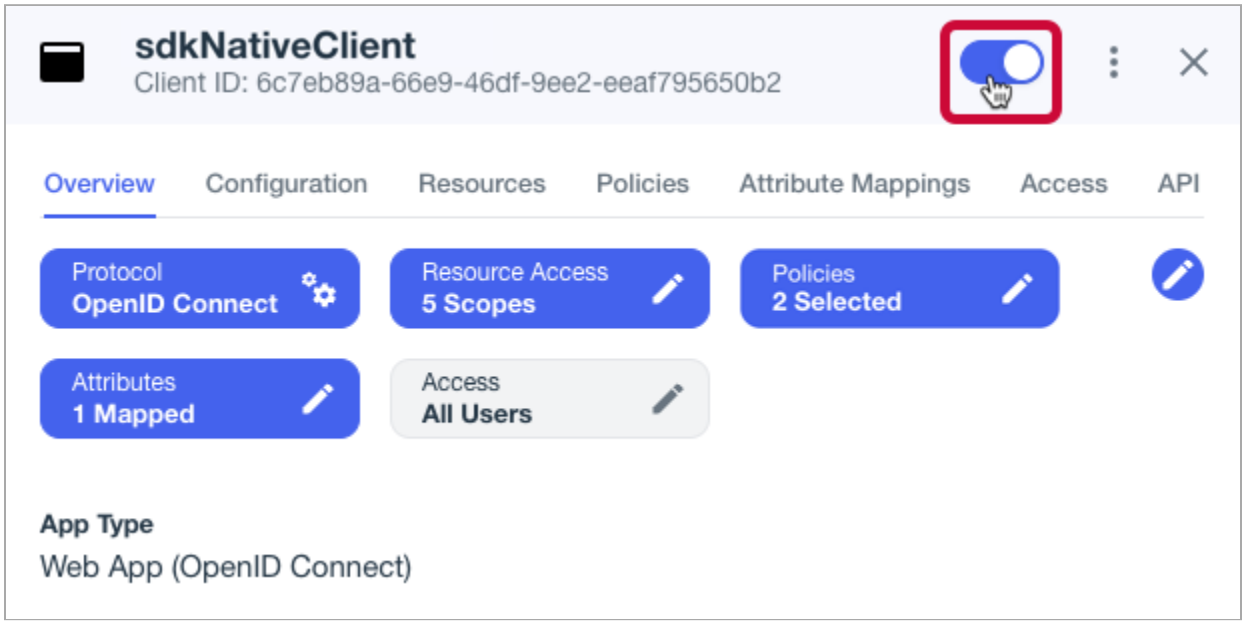


Figure 2. Enable the application using the toggle.

The application is now configured to accept client connections from and issue OAuth 2.0 tokens to the Android and iOS PingOne example applications and tutorials covered by this documentation.

Step 1. Download the samples

Prepare › **Download** › Configure › Run

To start this tutorial, you need to download the ForgeRock SDK sample apps repo, which contains the projects you will use.

1. In a web browser, navigate to the SDK Sample Apps repository.
2. Download the source code using one of the following methods:

Download a ZIP file

1. Click **Code**, and then click **Download ZIP**.
2. Extract the contents of the downloaded ZIP file to a suitable location.

Use a Git-compatible tool to clone the repo locally

1. Click **Code**, and then copy the HTTPS URL.
2. Use the URL to clone the repository to a suitable location.

   For example, from the command-line you could run:

```
git clone https://github.com/ForgeRock/sdk-sample-apps.git
```

The result of these steps is a local folder named `sdk-sample-apps`.

Step 2. Configure connection properties

Prepare › Download › **Configure** › Run

In this step, you configure the **kotlin-central-login-oidc** sample to connect to the OAuth 2.0 application you created in PingOne, using OIDC login.

1. In Android Studio, open the `sdk-sample-apps/android/kotlin-central-login-oidc` project you cloned in the previous step.
2. In the **Project** pane, switch to the **Android** view.

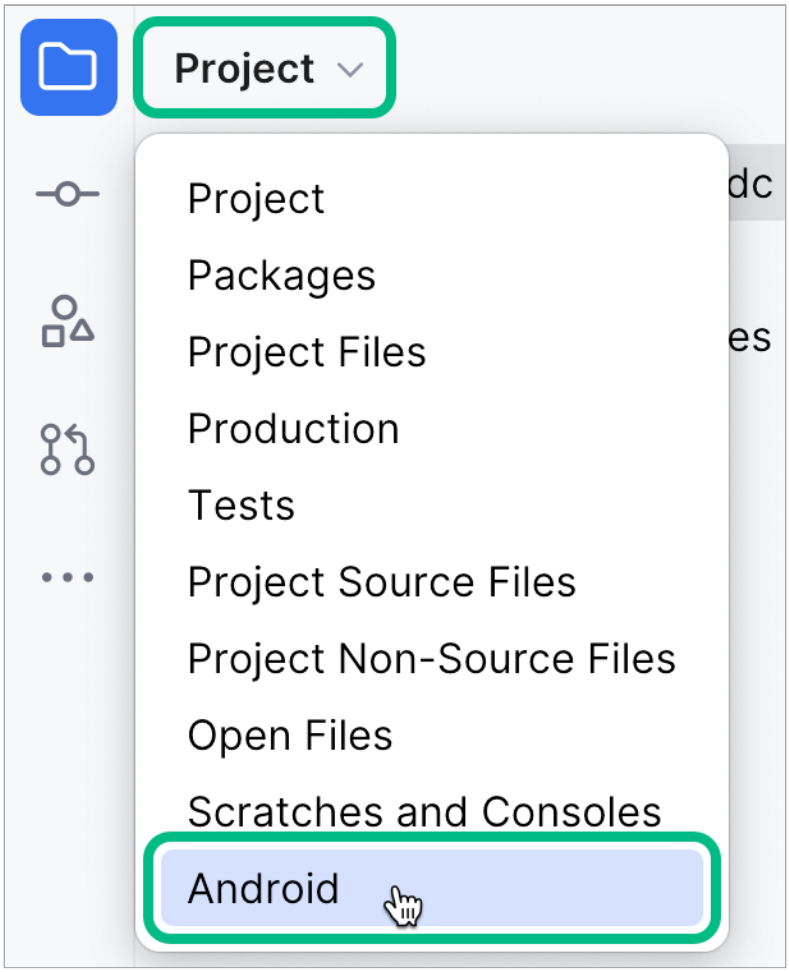


Figure 1. Switching the project pane to Android view.

3. In the **Android** view, navigate to **app > kotlin+java > com.example.app**, and open `Config.kt`.

4. Edit the default values provided in the `PingConfig` class with the values from your PingOne server:

`PingConfig` class default values

```
data class PingConfig(
    var discoveryEndpoint: String = "https://openam-sdks.forgeblocks.com/am/oauth2/realms/alpha/.well-known/
openid-configuration",
    var oauthClientId: String = "AndroidTest",
    var oauthRedirectUri: String = "org.forgerock.demo:/oauth2redirect",
    var oauthSignOutRedirectUri: String = "",
    var cookieName: String = "5421aeddf91aa20",
    var oauthScope: String = "openid profile email address")
)
```

discoveryEndpoint

The `.well-known` endpoint from your OAuth 2.0 application in PingOne.

To find the `.well-known` endpoint for an OAuth 2.0 client in PingOne:

1. Log in to your PingOne administration console.
2. Go to **Applications** > **Applications**, and then select the OAuth 2.0 client you created earlier.

   For example, sdkPublicClient.
3. On the **Configuration** tab, expand the **URLs** section, and then copy the **OIDC Discovery Endpoint** value.

For example, `https://auth.pingone.com/3072206d-c6ce-ch15-m0nd-f87e972c7cc3/as/.well-known/openid-configuration`

oauthClientId

The client ID from your OAuth 2.0 application in PingOne.

For example, `6c7eb89a-66e9-ab12-cd34-eeaf795650b2`

oauthRedirectUri

The `redirect_uri` as configured in the OAuth 2.0 client profile.

This value must exactly match a value configured in your OAuth 2.0 client.

For example, `org.forgerock.demo://oauth2redirect`

oauthSignOutRedirectUri

Leave this property empty.

It signals that the SDK can use the ID token to end the user's session, and does not need to open and return from a web page to perform log out.

> **Note**
>
> You must have enabled the **Terminate User Session by ID Token** setting when creating the OAuth 2.0 client in PingOne if you leave this property empty.

cookieName

Set this property to an empty string. PingOne servers do not require this setting.

oauthScope

The scopes you added to your OAuth 2.0 application in PingOne.

For example, `openid profile email phone`

The result resembles the following:

`PingConfig` class example values

```
data class PingConfig(
    var discoveryEndpoint: String = "https://auth.pingone.com/3072206d-c6ce-ch15-m0nd-f87e972c7cc3/as/.well-known/openid-configuration",
    var oauthClientId: String = "6c7eb89a-66e9-ab12-cd34-eeaf795650b2",
    var oauthRedirectUri: String = "org.forgerock.demo://oauth2redirect",
    var oauthSignOutRedirectUri: String = "",
    var cookieName: String = "",
    var oauthScope: String = "openid profile email phone"
)
```

5. Optionally, specify which of the configured policies PingOne uses to authenticate users.

   In `/app/kotlin+java/com.example.app/centralize/CentralizeLoginViewModel`, in the `login(fragmentActivity: FragmentActivity)` function, add an `acr_values` parameter to the authorization request by using the `setAdditionalParameters()` method:

```
fun login(fragmentActivity: FragmentActivity) {
  FRUser.browser().appAuthConfigurer()
    // Add acr values to the authorization request
    .authorizationRequest{
      it.setAdditionalParameters(
        mapOf(
          "acr_values" to "<Policy IDs>"
        )
      )
    }
    .customTabsIntent {
      it.setColorScheme(CustomTabsIntent.COLOR_SCHEME_DARK)
    }.appAuthConfiguration { appAuthConfiguration → }
    .done()
    .login(fragmentActivity,
        object : FRListener<FRUser> {
            override fun onSuccess(result: FRUser) {
              state.update {
                  it.copy(user = result, exception = null)
              }
            }

            override fun onException(e: Exception) {
                state.update {
                    it.copy(user = null, exception = e)
                }
            }
        }
    )
}
```

   Replace *<Policy IDs>* with either a single DaVinci policy by using its flow policy ID, or one or more PingOne policies by specifying the policy names, separated by spaces or the encoded space character `%20`.

   Examples:

DaVinci flow policy ID

```
"acr_values" to "d1210a6b0b2665dbaa5b652221badba2"
```

PingOne policy names

```
"acr_values" to "Single_Factor%20Multi_Factor"
```

For more information, refer to Editing an application - OIDC.

6. Save your changes.

Step 3. Test the app

Prepare › Download › Configure › Run

In the following procedure, you run the sample app that you configured in the previous step. The app performs a centralized login on your PingOne instance.

Log in as a demo user

1. In Android Studio, select **Run > Run 'ping-oidc.app'**.
2. On the **Environment** screen, ensure the PingOne environment you added earlier is correct.

   You can edit any of the values in the app if required.

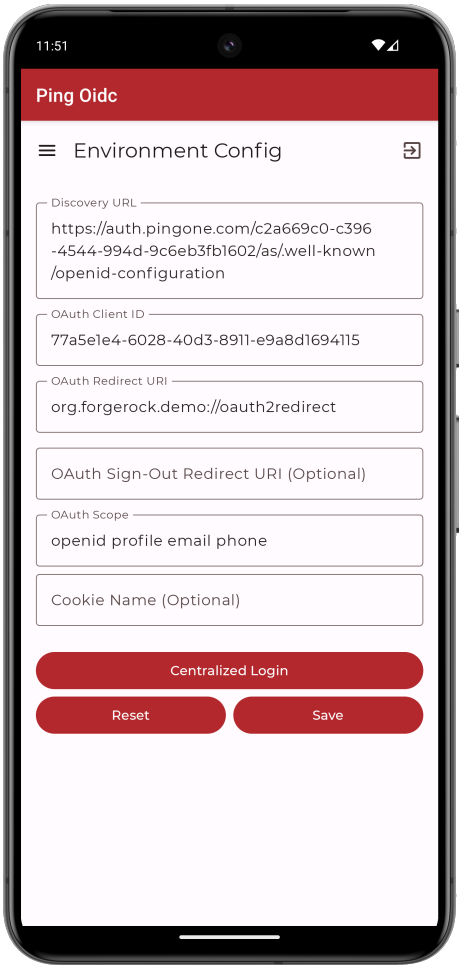


Figure 1. Confirm the PingOne connection properties

3. Tap **Centralized Login**.

   The app launches a web browser and redirects to your PingOne environment:

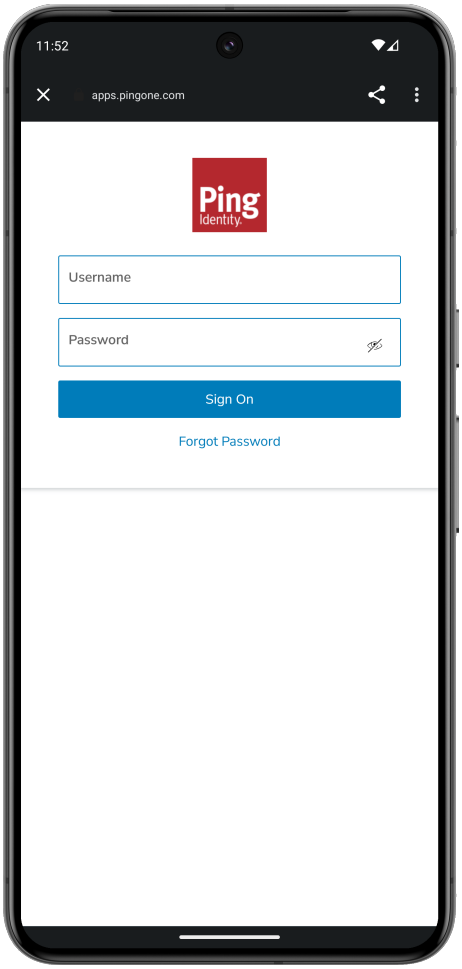


Figure 2. Browser launched and redirected to PingOne

4. Sign on as a demo user:
    - **Name**: `demo`
    - **Password**: `Ch4ng3it!`

   If authentication is successful, the application returns to the access token screen.

5. Tap the menu icon (☰), and then tap **User Profile**:

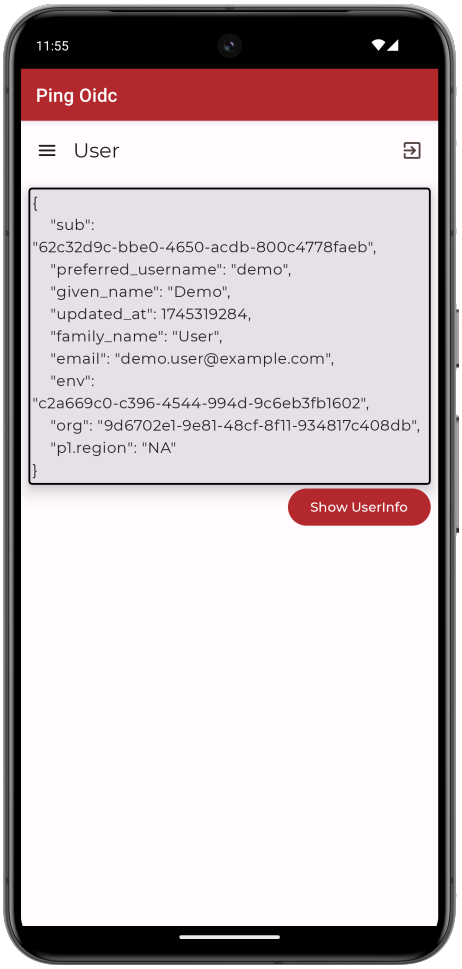


Figure 3. User info of the demo user

6. Tap the menu icon (☰), and then tap **Logout**.

   The app logs the user out of PingOne, revokes the tokens, and returns to the config page.

Tip

To verify the user is logged out:

1. In the PingOne administration console, navigate to **Directory > Users**.
2. Select the user you signed in as.
3. From the **Sevices** dropdown, select **Authentication**:

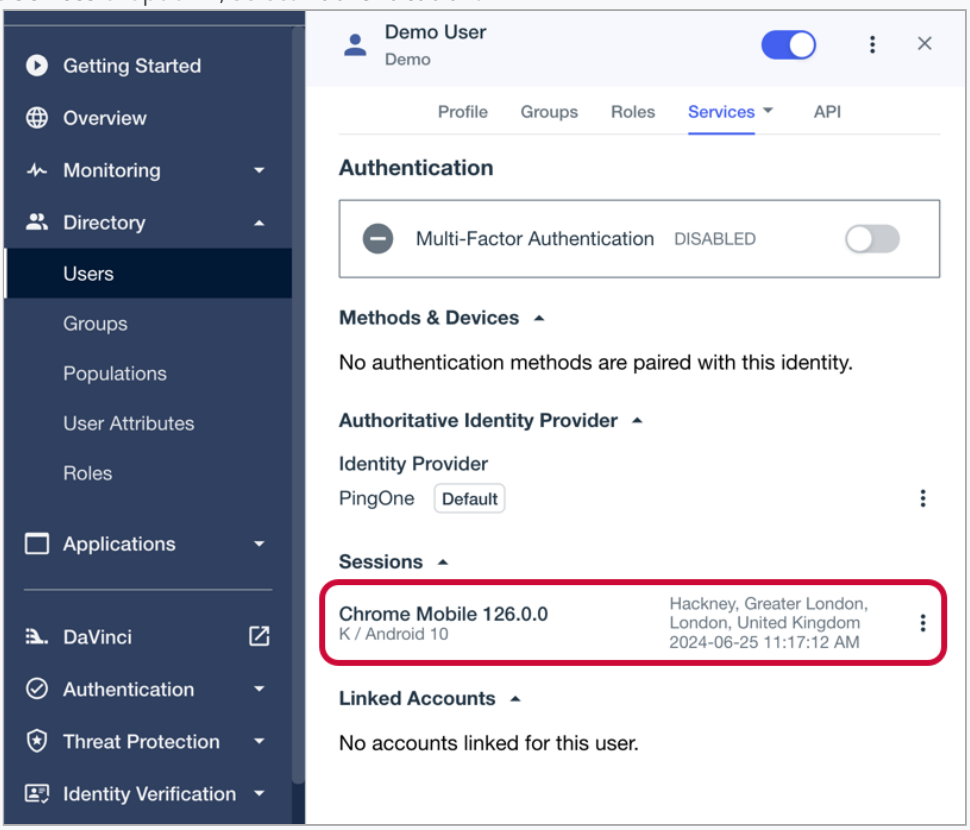


Figure 4. Checking a user's sessions in PingOne.

The **Sessions** section displays any existing sessions the user has, and whether they originate from a mobile device.

OIDC login to PingOne Advanced Identity Cloud tutorial for Android

Prepare › Download › Configure › Run

In this tutorial you update a sample app that uses OIDC-based login to obtain tokens by redirecting to the PingOne Advanced Identity Cloud UI for authentication.

The sample connects to the `.well-known` endpoint of your PingOne Advanced Identity Cloud server to obtain the correct URIs to authenticate the user, and redirects to your PingOne Advanced Identity Cloud server's login UI.

After authentication, PingOne Advanced Identity Cloud redirects the browser back to your application, which then obtains an OAuth 2.0 access token and displays the related user information.

Before you begin

Before you begin this tutorial ensure you have set up your PingOne Advanced Identity Cloud server with the required configuration.

For example, you will need to have an OAuth 2.0 client application set up, and a demo user to authenticate.

Complete prerequisites »

Step 1. Download the samples

To start this tutorial, you need to download the SDK sample apps repo, which contains the projects you will use.

Start step 1 »

Step 2. Configure connection properties

In this step, you configure the sample app to connect to the OAuth 2.0 application you created in PingOne Advanced Identity Cloud.

Start step 2 »

Step 3. Test the app

To test the app, run the sample that you configured in the previous step.

The sample connects to your PingOne Advanced Identity Cloud server to obtain the correct URIs to authenticate the user, and redirects the browser to your PingOne Advanced Identity Cloud server.

After authentication, PingOne Advanced Identity Cloud redirects the browser back to your application, which then obtains an OAuth 2.0 access token and displays the related user information.

Test app »

Before you begin

Prepare › Download › Configure › Run

To successfully complete this tutorial refer to the prerequisites in this section.

The tutorial also requires a configured PingOne Advanced Identity Cloud tenant.

Compatibility

Android

This sample requires at least Android API 23 (Android 6.0)

Java

This sample requires at least Java 8 (v1.8).

Prerequisites

Android Studio

Download and install Android Studio, which is available for many popular operating systems.

An Android emulator or physical device

To try the quick start application as you develop it, you need an Android device. To add a virtual, emulated Android device to Android Studio, refer to Create and manage virtual devices, on the **Android Developers** website.

Server configuration

This tutorial requires you to configure your PingOne Advanced Identity Cloud tenant as follows:

The samples and tutorials in this documentation often require that you have an identity set up so that you can test authentication.

To create a demo user in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. In the left panel, click **Identities** > **Manage**.
3. Click **+ New Alpha realm - User**.
4. Enter the following details:
    - **Username** = `demo`
    - **First Name** = `Demo`
    - **Last Name** = `User`
    - **Email Address** = `demo.user@example.com`
    - **Password** = `Ch4ng3it!`
5. Click **Save**.

Authentication journeys provide fine-grained authentication by allowing multiple paths and decision points throughout the flow. Authentication journeys are made up of nodes that define actions taken during authentication.

Each node performs a single task, such as collecting a username or making a simple decision. Nodes can have multiple outcomes rather than just success or failure. For details, see the Authentication nodes configuration reference in the PingAM documentation.

To create a simple journey for use when testing the Ping SDKs, follow these steps:

1. In your PingOne Advanced Identity Cloud tenant, navigate to **Journeys**, and click **+ New Journey**.
2. Enter a name, such as `sdkUsernamePasswordJourney` and click **Save**.

   The authentication journey designer appears.
3. Drag the following nodes into the designer area:
   - **Page Node**
   - **Platform Username**
   - **Platform Password**
   - **Data Store Decision**
4. Drag and drop the **Platform Username** and **Platform Password** nodes onto the **Page Node**, so that they both appear on the same page when logging in.
5. Connect the nodes as follows:

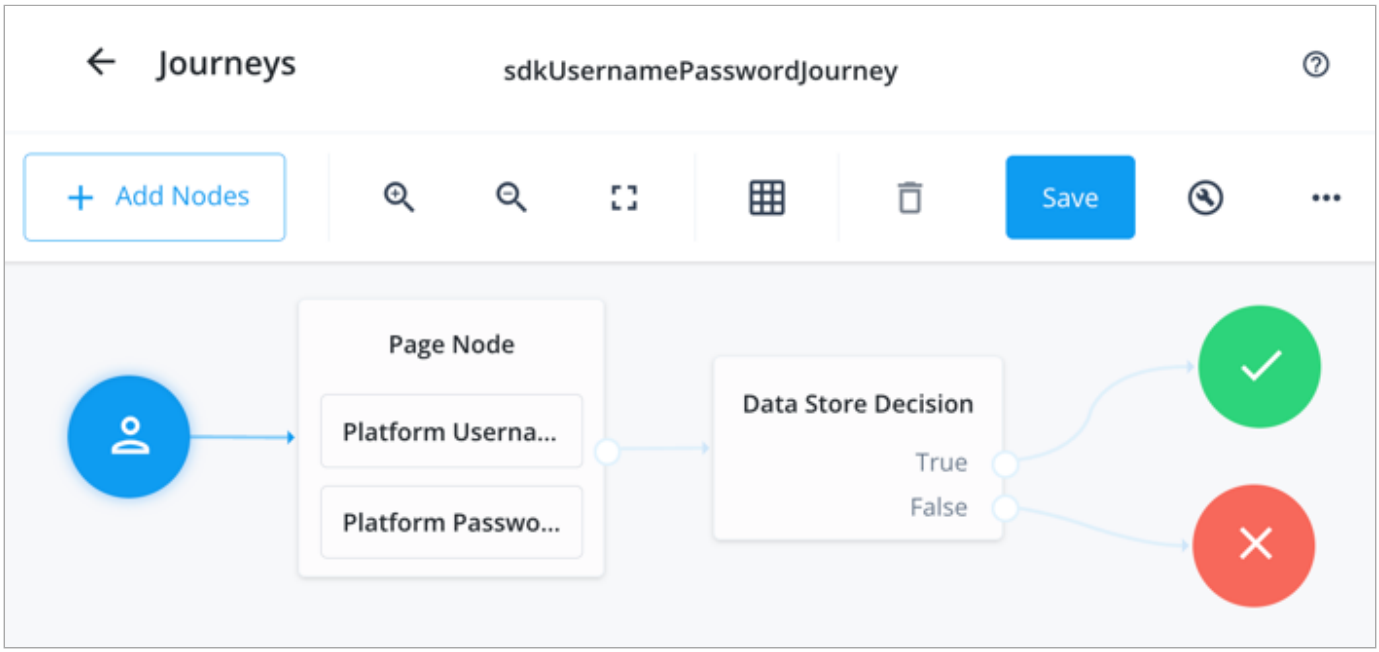


Figure 1. Example username and password authentication journey

6. Click **Save**.

Public clients do not use a client secret to obtain tokens because they are unable to keep them hidden. The Ping SDKs commonly use this type of client to obtain tokens, as they cannot guarantee safekeeping of the client credentials in a browser or on a mobile device.

To register a *public* OAuth 2.0 client application for use with the SDKs in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. In the left panel, click **Applications**.
3. Click **+ Custom Application**.
4. Select **OIDC - OpenId Connect** as the sign-in method, and then click **Next**.
5. Select **Native / SPA** as the application type, and then click **Next**.
6. In **Name**, enter a name for the application, such as `Public SDK Client`.
7. In **Owners**, select a user that is responsible for maintaining the application, and then click **Next**.

   > **Tip**
   >
   > When trying out the SDKs, you could select the `demo` user you created previously.

8. In **Client ID**, enter `sdkPublicClient`, and then click **Create Application**.

   PingOne Advanced Identity Cloud creates the application and displays the details screen.

9. On the **Sign On** tab:

   1. In **Sign-In URLs**, enter the following values:

      `org.forgerock.demo://oauth2redirect`

      > **Important**
      >
      > Also add any other domains where you host SDK applications.

   2. In **Grant Types**, enter the following values:

      `Authorization Code`

      `Refresh Token`

   3. In **Scopes**, enter the following values:

      `openid profile email address`

10. Click Show advanced settings, and on the **Authentication** tab:

    1. In **Token Endpoint Authentication Method**, select `none`.
    2. In **Client Type**, select `Public`.
    3. Enable the **Implied Consent** property.

11. Click **Save**.

The application is now configured to accept client connections from and issue OAuth 2.0 tokens to the example applications and tutorials covered by this documentation.

The provider specifies the supported OAuth 2.0 configuration options for a realm.

To ensure the PingOne Advanced Identity Cloud OAuth 2.0 provider service is configured for use with the Ping SDKs, follow these steps:

1. In your PingOne Advanced Identity Cloud tenant, navigate to **Native Consoles** > **Access Management**.
2. In the left panel, click **Services**.
3. In the list of services, click **OAuth2 Provider**.
4. On the **Core** tab, ensure **Issue Refresh Tokens** is enabled.
5. On the **Consent** tab, ensure **Allow Clients to Skip Consent** is enabled.
6. Click **Save Changes**.

Step 1. Download the samples

Prepare › **Download** › Configure › Run

To start this tutorial, you need to download the ForgeRock SDK sample apps repo, which contains the projects you will use.

1. In a web browser, navigate to the SDK Sample Apps repository.
2. Download the source code using one of the following methods:

   Download a ZIP file

   1. Click **Code**, and then click **Download ZIP**.
   2. Extract the contents of the downloaded ZIP file to a suitable location.

   Use a Git-compatible tool to clone the repo locally

   1. Click **Code**, and then copy the HTTPS URL.
   2. Use the URL to clone the repository to a suitable location.

      For example, from the command-line you could run:

      ```
      git clone https://github.com/ForgeRock/sdk-sample-apps.git
      ```

The result of these steps is a local folder named `sdk-sample-apps`.

Step 2. Configure connection properties

Prepare › Download › **Configure** › Run

In this step, you configure the **kotlin-central-login-oidc** sample to connect to the OAuth 2.0 application you created in PingOne Advanced Identity Cloud, using OIDC login.

1. In Android Studio, open the `sdk-sample-apps/android/kotlin-central-login-oidc` project you cloned in the previous step.
2. In the **Project** pane, switch to the **Android** view.
3. In the **Android** view, navigate to **app > kotlin+java > com.example.app**, and open `Config.kt`.
4. Edit the default values provided in the `PingConfig` class with the values from your PingOne Advanced Identity Cloud tenant:

```
data class PingConfig(
    var discoveryEndpoint: String = "https://openam-sdks.forgeblocks.com/am/oauth2/realms/alpha/.well-known/
openid-configuration",
    var oauthClientId: String = "AndroidTest",
    var oauthRedirectUri: String = "org.forgerock.demo://oauth2redirect",
    var oauthSignOutRedirectUri: String = "",
    var cookieName: String = "5421aeddf91aa20",
    var oauthScope: String = "openid profile email address"
)
```

discoveryEndpoint

The `.well-known` endpoint from your PingOne Advanced Identity Cloud tenant.

You can view the `.well-known` endpoint for an OAuth 2.0 client in the PingOne Advanced Identity Cloud admin console:

1. Log in to your PingOne Advanced Identity Cloud administration console.
2. Click **Applications**, and then select the OAuth 2.0 client you created earlier. For example, sdkPublicClient.
3. On the **Sign On** tab, in the **Client Credentials** section, copy the **Discovery URI** value.

For example, `https://openam-forgerock-sdks.forgeblocks.com/am/oauth2/alpha/.well-known/openid-configuration`

oauthClientId

The client ID from your OAuth 2.0 application in PingOne Advanced Identity Cloud.

For example, `sdkPublicClient`

oauthRedirectUri

The `redirect_uri` as configured in the OAuth 2.0 client profile.

This value must exactly match a value configured in your OAuth 2.0 client.

For example, `org.forgerock.demo://oauth2redirect`

oauthSignOutRedirectUri

Leave this property empty.

It signals that the SDK does not need to open and return from a web page to perform log out.

cookieName

The name of the cookie your PingOne Advanced Identity Cloud tenant uses to store SSO tokens in client browsers.

To locate the cookie name in an PingOne Advanced Identity Cloud tenant:

1. Navigate to **Tenant settings** > **Global Settings**
2. Copy the value of the **Cookie** property.

For example, `ch15fefc5407912`

oauthScope

The scopes you added to your OAuth 2.0 application in PingOne Advanced Identity Cloud.

For example, `openid profile email address`

The result resembles the following:

```
data class PingConfig(
    var discoveryEndpoint: String = "https://openam-forgerock-sdks.forgeblocks.com/am/oauth2/realms/
alpha/.well-known/openid-configuration",
    var oauthClientId: String = "sdkNativeClient",
    var oauthRedirectUri: String = "org.forgerock.demo://oauth2redirect",
    var oauthSignOutRedirectUri: String = "",
    var cookieName: String = "ch15fefc5407912",
    var oauthScope: String = "openid profile email address"
)
```

5. Save your changes.

Step 3. Test the app

Prepare › Download › Configure › **Run**

In the following procedure, you run the sample app that you configured in the previous step. The app performs a centralized login on your PingOne Advanced Identity Cloud instance.

Log in as a demo user

1. In Android Studio, select **Run > Run 'ping-oidc.app'**.
2. On the **Environment** screen, ensure the PingOne Advanced Identity Cloud environment you added earlier is correct.

   You can edit any of the values in the app if required.

Figure 1. Confirm the PingOne Advanced Identity Cloud connection properties

3. Tap **Centralized Login**.

   The app launches a web browser and redirects to your PingOne Advanced Identity Cloud environment:

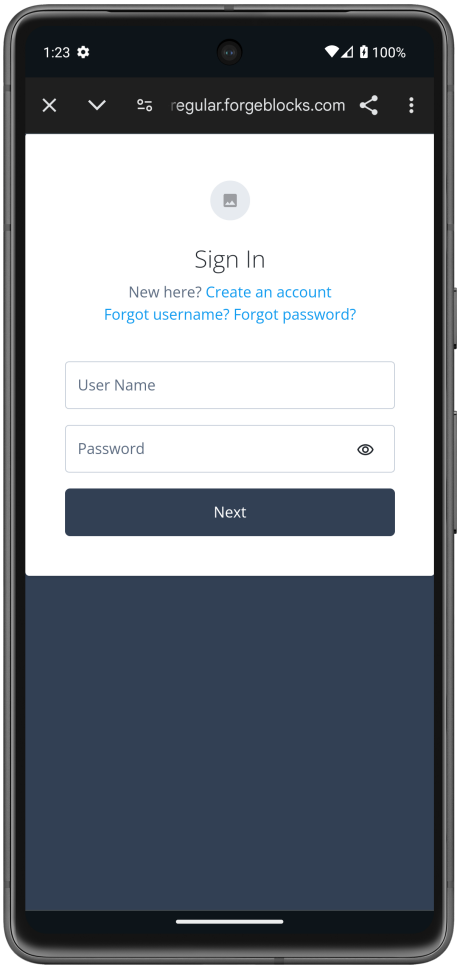


Figure 2. Browser launched and redirected to PingOne Advanced Identity Cloud

4. Sign on as a demo user:
    - **Name**: `demo`
    - **Password**: `Ch4ng3it!`

   If authentication is successful, the application returns to the access token screen.

5. Tap the menu icon (☰), and then tap **User Profile**:

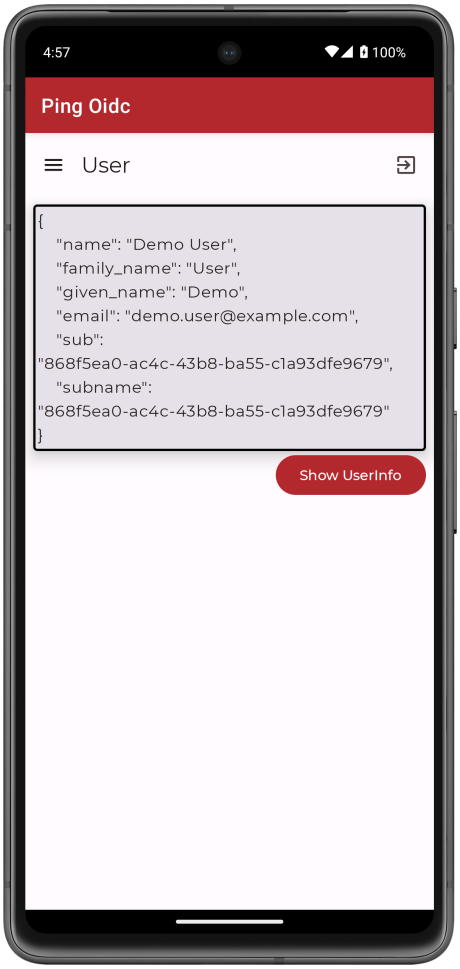


Figure 3. User info of the demo user

6. Tap the menu icon (☰), and then tap **Logout**.

   The app logs the user out of PingOne Advanced Identity Cloud, revokes the tokens, and returns to the config page.

OIDC login to PingAM tutorial for Android

Prepare › Download › Configure › Run

In this tutorial you update a sample app that uses OIDC-based login to obtain tokens by redirecting to the PingAM UI for authentication.

The sample connects to the `.well-known` endpoint of your PingAM server to obtain the correct URIs to authenticate the user, and redirects to your PingAM server's login UI.

After authentication, PingAM redirects the browser back to your application, which then obtains an OAuth 2.0 access token and displays the related user information.

Before you begin

Before you begin this tutorial ensure you have set up your PingAM server with the required configuration.

For example, you will need to have an OAuth 2.0 client application set up, and a demo user to authenticate.

Complete prerequisites »

Step 1. Download the samples

To start this tutorial, you need to download the SDK sample apps repo, which contains the projects you will use.

Start step 1 »

Step 2. Configure connection properties

In this step, you configure the sample app to connect to the OAuth 2.0 application you created in PingAM.

Start step 2 »

Step 3. Test the app

To test the app, run the sample that you configured in the previous step.

The sample connects to your PingAM server to obtain the correct URIs to authenticate the user, and redirects the browser to your PingAM server.

After authentication, PingAM redirects the browser back to your application, which then obtains an OAuth 2.0 access token and displays the related user information.

Test app »

Before you begin

Prepare › Download › Configure › Run

To successfully complete this tutorial refer to the prerequisites in this section.

The tutorial also requires a configured PingAM server.

Compatibility

Android

This sample requires at least Android API 23 (Android 6.0)

Java

This sample requires at least Java 8 (v1.8).

Prerequisites

Android Studio

Download and install Android Studio, which is available for many popular operating systems.

An Android emulator or physical device

To try the quick start application as you develop it, you need an Android device. To add a virtual, emulated Android device to Android Studio, refer to Create and manage virtual devices, on the **Android Developers** website.

Server configuration

This tutorial requires you to configure your PingAM server as follows:

The samples and tutorials in this documentation often require that you have an identity set up so that you can test authentication.

To create a demo user in PingAM, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. Navigate to **Identities**, and then click **+ Add Identity**.
3. Enter the following details:
   - **User ID** = `demo`
   - **Password** = `Ch4ng3it!`
   - **Email Address** = `demo.user@example.com`
4. Click **Create**.

Authentication trees provide fine-grained authentication by allowing multiple paths and decision points throughout the authentication flow. Authentication trees are made up of nodes that define actions taken during authentication.

Each node performs a single task, such as collecting a username or making a simple decision. Nodes can have multiple outcomes rather than just success or failure. For details, see the Authentication nodes configuration reference in the PingAM documentation.

To create a simple tree for use when testing the Ping SDKs, follow these steps:

1. Under **Realm Overview**, click **Authentication Trees**, then click **Create Tree**.

2. Enter a tree name, for example `sdkUsernamePasswordJourney`, and then click **Create**.

   The authentication tree designer appears, showing the **Start** entry point connected to the **Failure** exit point.

3. Drag the following nodes from the **Components** panel on the left side into the designer area:

   - **Page Node**
   - **Username Collector**
   - **Password Collector**
   - **Data Store Decision**

4. Drag and drop the **Username Collector** and **Password Collector** nodes onto the **Page Node**, so that they both appear on the same page when logging in.

5. Connect the nodes as follows:

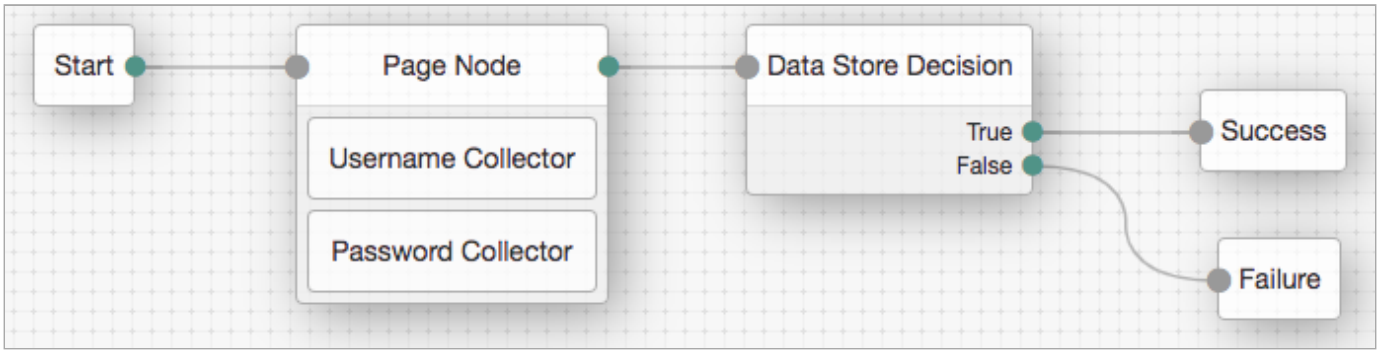


Figure 1. Example username and password authentication tree

6. Select the **Page Node**, and in the **Properties** pane, set the **Stage** property to `UsernamePassword`.

   > **Tip**
   >
   > You can configure the node properties by selecting a node and altering properties in the right-hand panel.

   One of the samples uses this specific value to determine the custom UI to display.

7. Click **Save**.

Public clients do not use a client secret to obtain tokens because they are unable to keep them hidden. The Ping SDKs commonly use this type of client to obtain tokens, as they cannot guarantee safekeeping of the client credentials in a browser or on a mobile device.

To register a *public* OAuth 2.0 client application for use with the SDKs in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. In the left panel, click **Applications**.
3. Click **+ Custom Application**.
4. Select **OIDC - OpenId Connect** as the sign-in method, and then click **Next**.
5. Select **Native / SPA** as the application type, and then click **Next**.

6. In **Name**, enter a name for the application, such as `Public SDK Client`.
7. In **Owners**, select a user that is responsible for maintaining the application, and then click **Next**.

> **Tip**
>
> When trying out the SDKs, you could select the `demo` user you created previously.

8. In **Client ID**, enter `sdkPublicClient`, and then click **Create Application**.

   PingOne Advanced Identity Cloud creates the application and displays the details screen.
9. On the **Sign On** tab:
   1. In **Sign-In URLs**, enter the following values:

      `org.forgerock.demo://oauth2redirect`

      > **Important**
      >
      > Also add any other domains where you host SDK applications.

   2. In **Grant Types**, enter the following values:

      `Authorization Code`

      `Refresh Token`

   3. In **Scopes**, enter the following values:

      `openid profile email address`
10. Click Show advanced settings, and on the **Authentication** tab:
    1. In **Token Endpoint Authentication Method**, select `none`.
    2. In **Client Type**, select `Public`.
    3. Enable the **Implied Consent** property.
11. Click **Save**.

The application is now configured to accept client connections from and issue OAuth 2.0 tokens to the example applications and tutorials covered by this documentation.

The provider specifies the supported OAuth 2.0 configuration options for a realm.

To ensure the PingOne Advanced Identity Cloud OAuth 2.0 provider service is configured for use with the Ping SDKs, follow these steps:

1. In your PingOne Advanced Identity Cloud tenant, navigate to **Native Consoles** > **Access Management**.
2. In the left panel, click **Services**.
3. In the list of services, click **OAuth2 Provider**.
4. On the **Core** tab, ensure **Issue Refresh Tokens** is enabled.

5. On the **Consent** tab, ensure **Allow Clients to Skip Consent** is enabled.
6. Click **Save Changes**.

Step 1. Download the samples

Prepare › **Download** › Configure › Run

To start this tutorial, you need to download the ForgeRock SDK sample apps repo, which contains the projects you will use.

1. In a web browser, navigate to the SDK Sample Apps repository.
2. Download the source code using one of the following methods:

   ##### *Download a ZIP file*

   1. Click **Code**, and then click **Download ZIP**.
   2. Extract the contents of the downloaded ZIP file to a suitable location.

   ##### *Use a Git-compatible tool to clone the repo locally*

   1. Click **Code**, and then copy the HTTPS URL.
   2. Use the URL to clone the repository to a suitable location.

      For example, from the command-line you could run:

      ```
      git clone https://github.com/ForgeRock/sdk-sample-apps.git
      ```

The result of these steps is a local folder named `sdk-sample-apps`.

Step 2. Configure connection properties

Prepare › Download › **Configure** › Run

In this step, you configure the **kotlin-central-login-oidc** sample to connect to the OAuth 2.0 application you created in PingOne Advanced Identity Cloud, using OIDC login.

1. In Android Studio, open the `sdk-sample-apps/android/kotlin-central-login-oidc` project you cloned in the previous step.
2. In the **Project** pane, switch to the **Android** view.
3. In the **Android** view, navigate to **app > kotlin+java > com.example.app**, and open `Config.kt`.

4. Edit the default values provided in the `PingConfig` class with the values from your PingOne Advanced Identity Cloud tenant:

```
data class PingConfig(
    var discoveryEndpoint: String = "https://openam-sdks.forgeblocks.com/am/oauth2/realms/alpha/.well-known/
openid-configuration",
    var oauthClientId: String = "AndroidTest",
    var oauthRedirectUri: String = "org.forgerock.demo://oauth2redirect",
    var oauthSignOutRedirectUri: String = "",
    var cookieName: String = "5421aeddf91aa20",
    var oauthScope: String = "openid profile email address"
)
```

discoveryEndpoint

The `.well-known` endpoint from your PingOne Advanced Identity Cloud tenant.

You can view the `.well-known` endpoint for an OAuth 2.0 client in the PingOne Advanced Identity Cloud admin console:

1. Log in to your PingOne Advanced Identity Cloud administration console.
2. Click **Applications**, and then select the OAuth 2.0 client you created earlier. For example, sdkPublicClient.
3. On the **Sign On** tab, in the **Client Credentials** section, copy the **Discovery URI** value.

For example, `https://openam-forgerock-sdks.forgeblocks.com/am/oauth2/alpha/.well-known/openid-configuration`

oauthClientId

The client ID from your OAuth 2.0 application in PingOne Advanced Identity Cloud.

For example, `sdkPublicClient`

oauthRedirectUri

The `redirect_uri` as configured in the OAuth 2.0 client profile.

This value must exactly match a value configured in your OAuth 2.0 client.

For example, `org.forgerock.demo://oauth2redirect`

oauthSignOutRedirectUri

Leave this property empty.

It signals that the SDK does not need to open and return from a web page to perform log out.

cookieName

The name of the cookie your PingOne Advanced Identity Cloud tenant uses to store SSO tokens in client browsers.

To locate the cookie name in an PingOne Advanced Identity Cloud tenant:

1. Navigate to **Tenant settings** > **Global Settings**

2. Copy the value of the **Cookie** property.

   For example, `ch15fefc5407912`

oauthScope

The scopes you added to your OAuth 2.0 application in PingOne Advanced Identity Cloud.

For example, `openid profile email address`

The result resembles the following:

```
data class PingConfig(
    var discoveryEndpoint: String = "https://openam-forgerock-sdks.forgeblocks.com/am/oauth2/realms/
alpha/.well-known/openid-configuration",
    var oauthClientId: String = "sdkNativeClient",
    var oauthRedirectUri: String = "org.forgerock.demo://oauth2redirect",
    var oauthSignOutRedirectUri: String = "",
    var cookieName: String = "ch15fefc5407912",
    var oauthScope: String = "openid profile email address"
)
```

5. Save your changes.

Step 3. Test the app

Prepare › Download › Configure › **Run**

In the following procedure, you run the sample app that you configured in the previous step. The app performs a centralized login on your PingAM instance.

Log in as a demo user

1. In Android Studio, select **Run > Run 'ping-oidc.app'**.
2. On the **Environment** screen, ensure the PingAM environment you added earlier is correct.

   You can edit any of the values in the app if required.

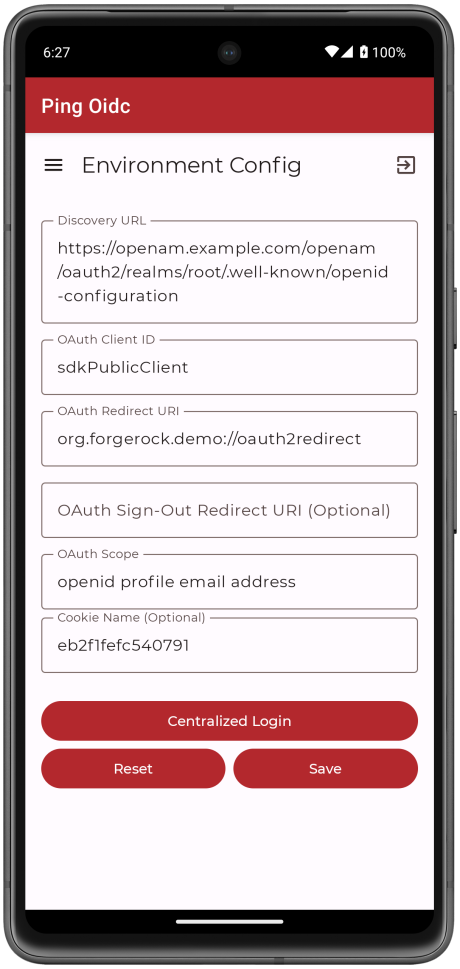


Figure 1. Confirm the PingAM connection properties

3. Tap **Centralized Login**.

   The app launches a web browser and redirects to your PingAM environment:

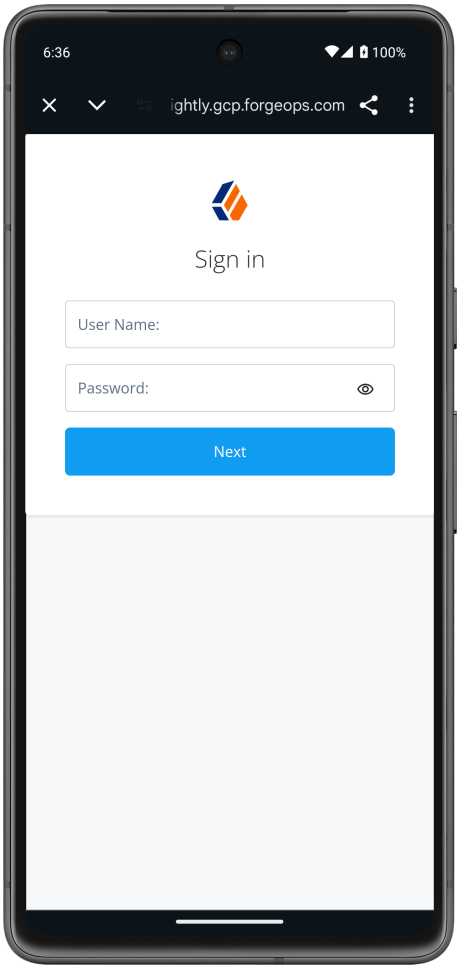


Figure 2. Browser launched and redirected to PingAM

4. Sign on as a demo user:
    - **Name**: `demo`
    - **Password**: `Ch4ng3it!`

   If authentication is successful, the application returns to the access token screen.

5. Tap the menu icon (☰), and then tap **User Profile**:

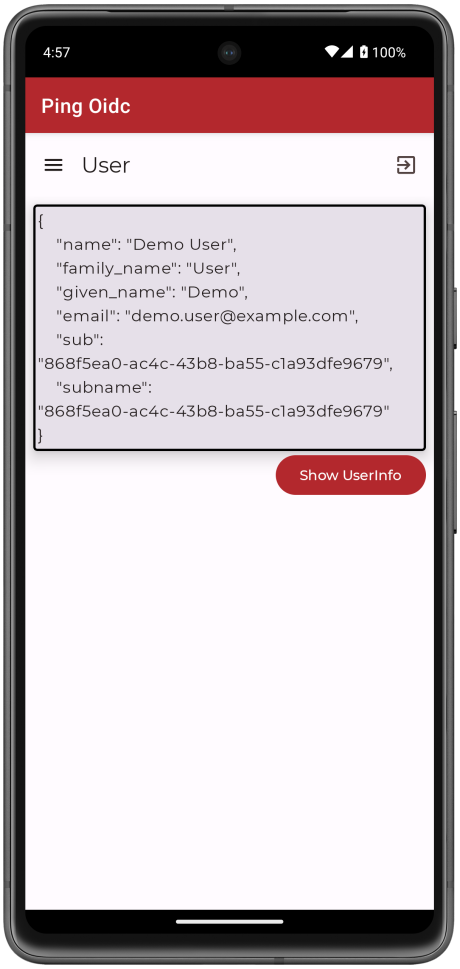


Figure 3. User info of the demo user

6. Tap the menu icon (☰), and then tap **Logout**.

   The app logs the user out of PingAM, revokes the tokens, and returns to the config page.

OIDC login to PingFederate tutorial for Android

Prepare › Download › Configure › Run

In this tutorial you update a sample app that uses OIDC-based login to obtain tokens by redirecting to the PingFederate UI for authentication.

The sample connects to the `.well-known` endpoint of your PingFederate server to obtain the correct URIs to authenticate the user, and redirects to your PingFederate server's login UI.

After authentication, PingFederate redirects the browser back to your application, which then obtains an OAuth 2.0 access token and displays the related user information.

Before you begin

Before you begin this tutorial ensure you have set up your PingFederate server with the required configuration.

For example, you will need to configure an OAuth 2.0 client application.

Complete prerequisites »

Step 1. Download the samples

To start this tutorial, you need to download the SDK sample apps repo, which contains the projects you will use.

Start step 1 »

Step 2. Configure connection properties

In this step, you configure the sample app to connect to the OAuth 2.0 application you created in PingFederate.

Start step 3 »

Step 3. Test the app

In the following procedure, you run the sample app that you configured in the previous step.

The sample connects to your PingFederate server to obtain the correct URIs to authenticate the user, and redirects the browser to your PingFederate server.

After authentication, PingFederate redirects the browser back to your application, which then obtains an OAuth 2.0 access token and displays the related user information.

Test app »

Before you begin

Prepare › Download › Configure › Run

To successfully complete this tutorial refer to the prerequisites and compatibility requirements in this section.

The tutorial also requires a configured PingFederate server.

Compatibility

Android

This sample requires at least Android API 23 (Android 6.0)

Java

This sample requires at least Java 8 (v1.8).

Prerequisites

Android Studio

Download and install Android Studio, which is available for many popular operating systems.

An Android emulator or physical device

To try the quick start application as you develop it, you need an Android device. To add a virtual, emulated Android device to Android Studio, refer to Create and manage virtual devices, on the **Android Developers** website.

Server configuration

This tutorial requires you to configure your PingFederate server as follows:

OAuth 2.0 client application profiles define how applications connect to PingFederate and obtain OAuth 2.0 tokens.

To allow the Ping SDKs to connect to PingFederate and obtain OAuth 2.0 tokens, you must register an OAuth 2.0 client application:

1. Log in to the PingFederate administration console as an administrator.
2. Navigate to **Applications** > **OAuth** > **Clients**.
3. Click **Add Client**.

   PingFederate displays the **Clients** | **Client** page.
4. In **Client ID** and **Name**, enter a name for the profile, for example `sdkPublicClient`

   Make a note of the **Client ID** value, you will need it when you configure the sample code.
5. In **Client Authentication**, select `None`.
6. In **Redirect URIs**, add the following values:

   `org.forgerock.demo://oauth2redirect`

   > **Important**
   >
   > Also add any other URLs where you host SDK applications.
   > Failure to add redirect URLs that exactly match your client app's values can cause PingFederate to display an error message such as `Redirect URI mismatch` when attempting to end a session by redirecting from the SDK.

7. In **Allowed Grant Types**, select the following values:

   `Authorization Code`

   `Refresh Token`

8. In the **OpenID Connect** section:

   1. In **Logout Mode**, select **Ping Front-Channel**

   2. In **Front-Channel Logout URIs**, add the following values:

      `org.forgerock.demo://oauth2redirect`

      > **Important**
      >
      > Also add any other URLs that redirect users to PingFederate to end their session.
      > Failure to add sign off URLs that exactly match your client app's values can cause PingFederate to display an error message such as `invalid post logout redirect URI` when attempting to end a session by redirecting from the SDK.

   3. In **Post-Logout Redirect URIs**, add the following values:

      `org.forgerock.demo://oauth2redirect`

9. Click **Save**.

   > **Important**
   >
   > After changing PingFederate configuration using the administration console, you must replicate the changes to each server node in the cluster before they take effect.
   > In the PingFederate administration console, navigate to **System > Server > Cluster Management**, and click **Replicate**.

   The application is now configured to accept client connections from and issue OAuth 2.0 tokens to the Ping SDK PingFederate example applications and tutorials covered by this documentation.

Cross-origin resource sharing (CORS) lets user agents make cross-domain server requests. In PingFederate, you can configure CORS to allow browsers or apps from trusted domains to access protected resources.

To configure CORS in PingFederate follow these steps:

1. Log in to the PingFederate administration console as an administrator.

2. Navigate to **System > OAuth Settings > Authorization Server Settings**.

3. In the **Cross-Origin Resource Sharing Settings** section, in the **Allowed Origin** field, enter any DNS aliases you use for your SDK apps.

   This documentation assumes the following configuration:

Property	Values
`Allowed Origin`	`org.forgerock.demo://oauth2redirect`

4. Click **Save**.

> **Important**
>
> After changing PingFederate configuration using the administration console, you must replicate the changes to each server node in the cluster before they take effect.
> In the PingFederate administration console, navigate to **System > Server > Cluster Management**, and click **Replicate**.

Your PingFederate server is now able to accept connections from origins hosting apps built with the Ping SDKs.

Step 1. Download the samples

Prepare › **Download** › Configure › Run

To start this tutorial, you need to download the projects you will use.

1. In a web browser, navigate to the SDK Sample Apps repository.
2. Download the source code using one of the following methods:

Download a ZIP file

1. Click **Code**, and then click **Download ZIP**.
2. Extract the contents of the downloaded ZIP file to a suitable location.

Use a Git-compatible tool to clone the repo locally

1. Click **Code**, and then copy the HTTPS URL.
2. Use the URL to clone the repository to a suitable location.

   For example, from the command-line you could run:

```
git clone https://github.com/ForgeRock/sdk-sample-apps.git
```

The result of these steps is a local folder named `sdk-sample-apps`.

Step 2. Configure connection properties

Prepare › Download › **Configure** › Run

In this step, you configure the **kotlin-central-login-oidc** sample to connect to the OAuth 2.0 application you created in PingFederate, using OIDC login.

1. In Android Studio, open the `sdk-sample-apps/android/kotlin-central-login-oidc` project you cloned in the previous step.
2. In the **Project** pane, switch to the **Android** view.

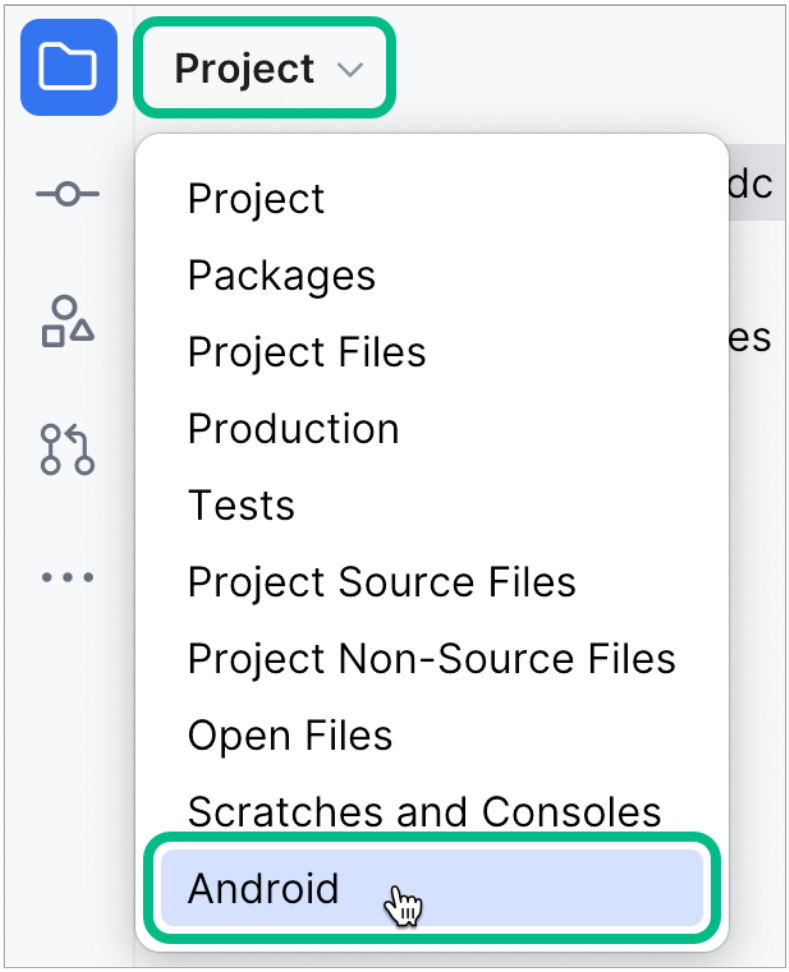


Figure 1. Switching the project pane to Android view.

3. In the **Android** view, navigate to **app > kotlin+java > com.example.app**, and open `Config.kt`.
4. Edit the default values provided in the `PingConfig` class with the values from your PingFederate server:

`PingConfig` class default values

```
data class PingConfig(
    var discoveryEndpoint: String = "https://openam-sdks.forgeblocks.com/am/oauth2/realms/alpha/.well-known/
openid-configuration",
    var oauthClientId: String = "AndroidTest",
    var oauthRedirectUri: String = "org.forgerock.demo:/oauth2redirect",
    var oauthSignOutRedirectUri: String = "",
    var cookieName: String = "5421aeddf91aa20",
    var oauthScope: String = "openid profile email address")
)
```

discoveryEndpoint

The `.well-known` endpoint of your PingFederate server.

To form the `.well-known` endpoint for a PingFederate server:

1. Log in to your PingFederate administration console.
2. Navigate to **System** > **Server** > **Protocol Settings**.
3. Make a note of the **Base URL** value.

   For example, `https://pingfed.example.com`

   > **Note**
   >
   > Do not use the admin console URL.

4. Append `/.well-known/openid-configuration` after the base URL value to form the `.well-known` endpoint of your server.

   For example, `https://pingfed.example.com/.well-known/openid-configuration`.

   The SDK reads the OAuth 2.0 paths it requires from this endpoint.

For example, `https://pingfed.example.com/.well-known/openid-configuration`

oauthClientId

The client ID from your OAuth 2.0 application in PingFederate.

For example, `sdkPublicClient`

oauthRedirectUri

The **Redirect URIs** as configured in the OAuth 2.0 client profile.

This value must exactly match a value configured in your OAuth 2.0 client.

For example, `org.forgerock.demo://oauth2redirect`

oauthSignOutRedirectUri

The **Front-Channel Logout URIs** as configured in the OAuth 2.0 client profile.

This value must exactly match a value configured in your OAuth 2.0 client.

For example, `org.forgerock.demo://oauth2redirect`

cookieName

Set this property to an empty string. PingFederate servers do not require this setting.

oauthScope

The scopes you added to your OAuth 2.0 application in PingFederate.

For example, `openid profile email phone`

The result resembles the following:

`PingConfig` class example values

```
data class PingConfig(
    var discoveryEndpoint: String = "https://pingfed.example.com/.well-known/openid-configuration",
    var oauthClientId: String = "sdkPublicClient",
    var oauthRedirectUri: String = "org.forgerock.demo://oauth2redirect",
    var oauthSignOutRedirectUri: String = "org.forgerock.demo://oauth2redirect",
    var cookieName: String = "",
    var oauthScope: String = "openid profile email phone"
)
```

5. Save your changes.

Step 3. Test the app

Prepare › Download › Configure › Run

In the following procedure, you run the sample app that you configured in the previous step. The app performs a centralized login on your PingFederate instance.

Log in as a demo user

1. In Android Studio, select **Run > Run 'ping-oidc.app'**.
2. On the **Environment** screen, ensure the PingFederate environment you added earlier is correct.

   You can edit any of the values in the app if required.

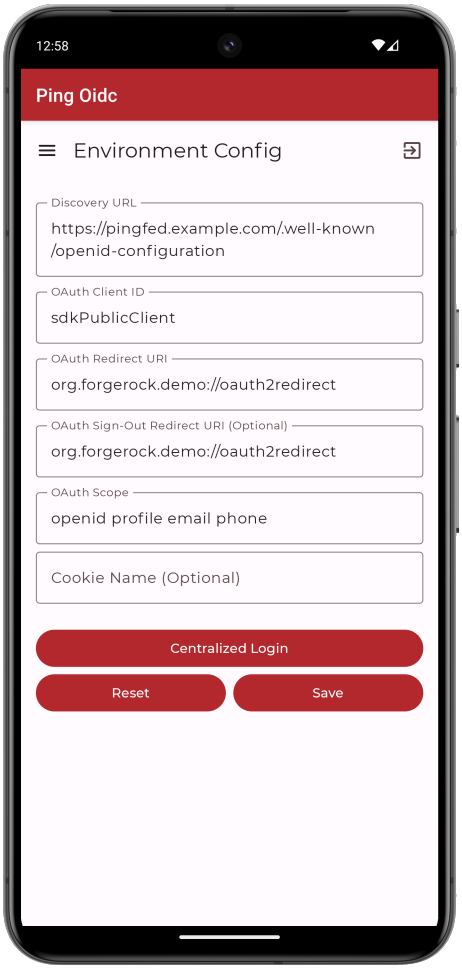


Figure 1. Confirm the PingFederate connection properties

3. Tap **Centralized Login**.

   The app launches a web browser and redirects to your PingFederate environment:

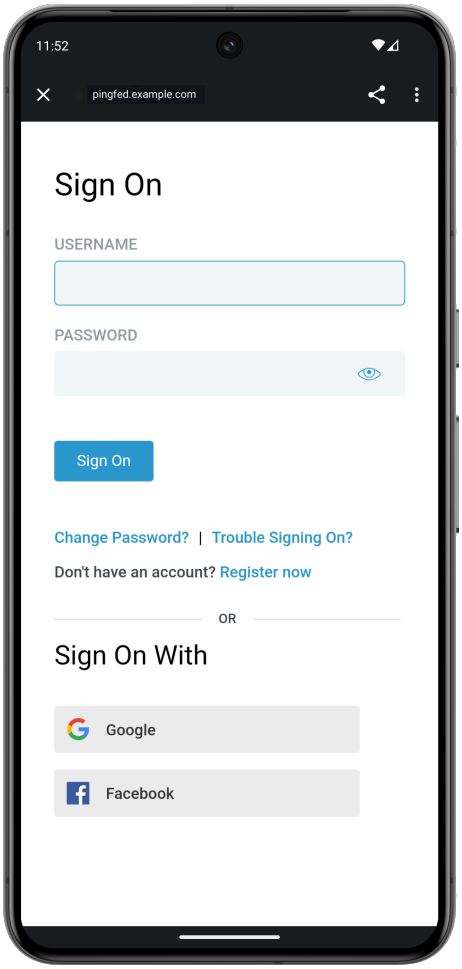


Figure 2. Browser launched and redirected to PingFederate

4. Sign on as a demo user:
    - **Name**: `demo`
    - **Password**: `Ch4ng3it!`

   If authentication is successful, the application returns to the access token screen.

5. Tap the menu icon (☰), and then tap **User Profile**:

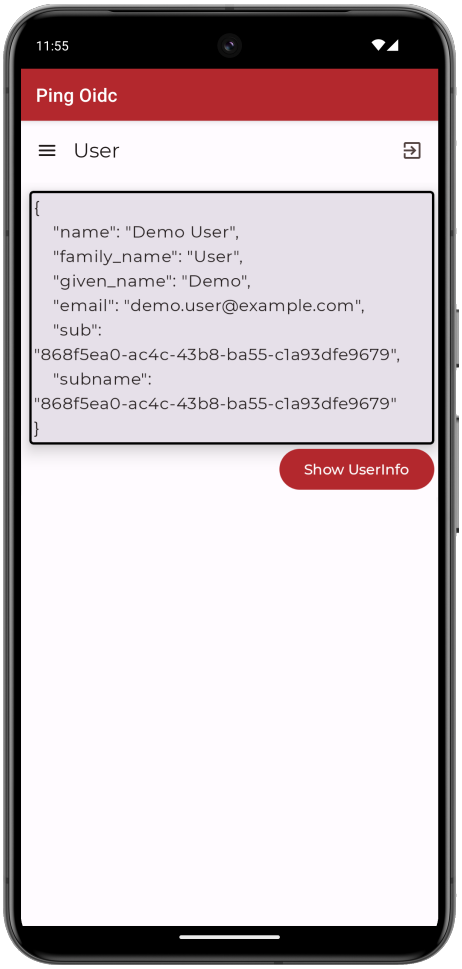


Figure 3. User info of the demo user

6. Tap the menu icon (☰), and then tap **Logout**.

   The app opens a browser momentarily to log the user out of PingFederate, and revoke the tokens.

Apple iOS OIDC login tutorials

Follow these iOS tutorials to integrate your apps using OpenID Connect login to the following servers:

PingOne

PingOne

PingOne
Advanced Identity Cloud

PingOne Advanced Identity Cloud

PingAM

PingAM

PingFederate

PingFederate

Authentication journey tutorial for iOS

Prepare › Download › Configure › Run

In this tutorial you update a sample app that uses OIDC-based login to obtain tokens by redirecting to the PingOne UI for authentication.

The sample connects to the `.well-known` endpoint of your PingOne server to obtain the correct URIs to authenticate the user, and redirects to your PingOne server's login UI.

After authentication, PingOne redirects the browser back to your application, which then obtains an OAuth 2.0 access token and displays the related user information.

Before you begin

Before you begin this tutorial ensure you have set up your PingOne server with the required configuration.

For example, you will need to have an OAuth 2.0 client application set up, and a demo user to authenticate.

Complete prerequisites »

Step 1. Download the samples

To start this tutorial, you need to download the SDK sample apps repo, which contains the projects you will use.

Start step 1 »

Step 2. Configure connection properties

In this step, you configure the sample app to connect to the OAuth 2.0 application you created in PingOne.

Start step 2 »

Step 3. Test the app

To test the app, run the sample that you configured in the previous step.

The sample connects to your PingOne server to obtain the correct URIs to authenticate the user, and redirects the browser to your PingOne server.

After authentication, PingOne redirects the browser back to your application, which then obtains an OAuth 2.0 access token and displays the related user information.

Test app »

Before you begin

Prepare › Download › Configure › Run

To successfully complete this tutorial refer to the prerequisites and compatibility requirements in this section.

The tutorial also requires a configured PingOne instance.

Compatibility

iOS

This sample app is compatible with iOS 12 and later.

Prerequisites

Xcode

You can download the latest version for free from https://developer.apple.com/xcode/.

Server configuration

This tutorial requires you to configure your PingOne server as follows:

The samples and tutorials in this documentation often require that you have an identity set up so that you can test authentication.

To create a demo user in PingOne, follow these steps:

1. Log in to your PingOne administration console.
2. In the left panel, navigate to **Directory** > **Users**.
3. Next to the **Users** label, click the plus icon (+).

   PingOne displays the **Add User** panel.
4. Enter the following details:
   - **Given Name** = `Demo`
   - **Family Name** = `User`
   - **Username** = `demo`
   - **Email** = `demo.user@example.com`
   - **Population** = `Default`
   - **Password** = `Ch4ng3it!`
5. Click **Save**.

To register a *public* OAuth 2.0 client application in PingOne for use with the Ping SDKs for Android and iOS, follow these steps:

1. Log in to your PingOne administration console.
2. In the left panel, navigate to **Applications** > **Applications**.
3. Next to the **Applications** label, click the plus icon (+).

   PingOne displays the **Add Application** panel.
4. In **Application Name**, enter a name for the profile, for example `sdkNativeClient`
5. Select **Native** as the **Application Type**, and then click **Save**.
6. On the **Configuration** tab, click the pencil icon (✎).
   1. In **Grant Type**, select the following values:

      `Authorization Code`

`Refresh Token`

2. In **Redirect URIs**, enter the following value:

   `org.forgerock.demo://oauth2redirect`

3. In **Token Endpoint Authentication Method**, select `None`.
4. In the **Advanced Settings** section, enable **Terminate User Session by ID Token**.
5. Click **Save**.

7. On the **Resources** tab, next to **Allowed Scopes**, click the pencil icon (✎).

   1. In **Scopes**, select the following values:

      `email`

      `phone`

      `profile`

      > **Note**
      >
      > The `openid` scope is selected by default.

      The result resembles the following:

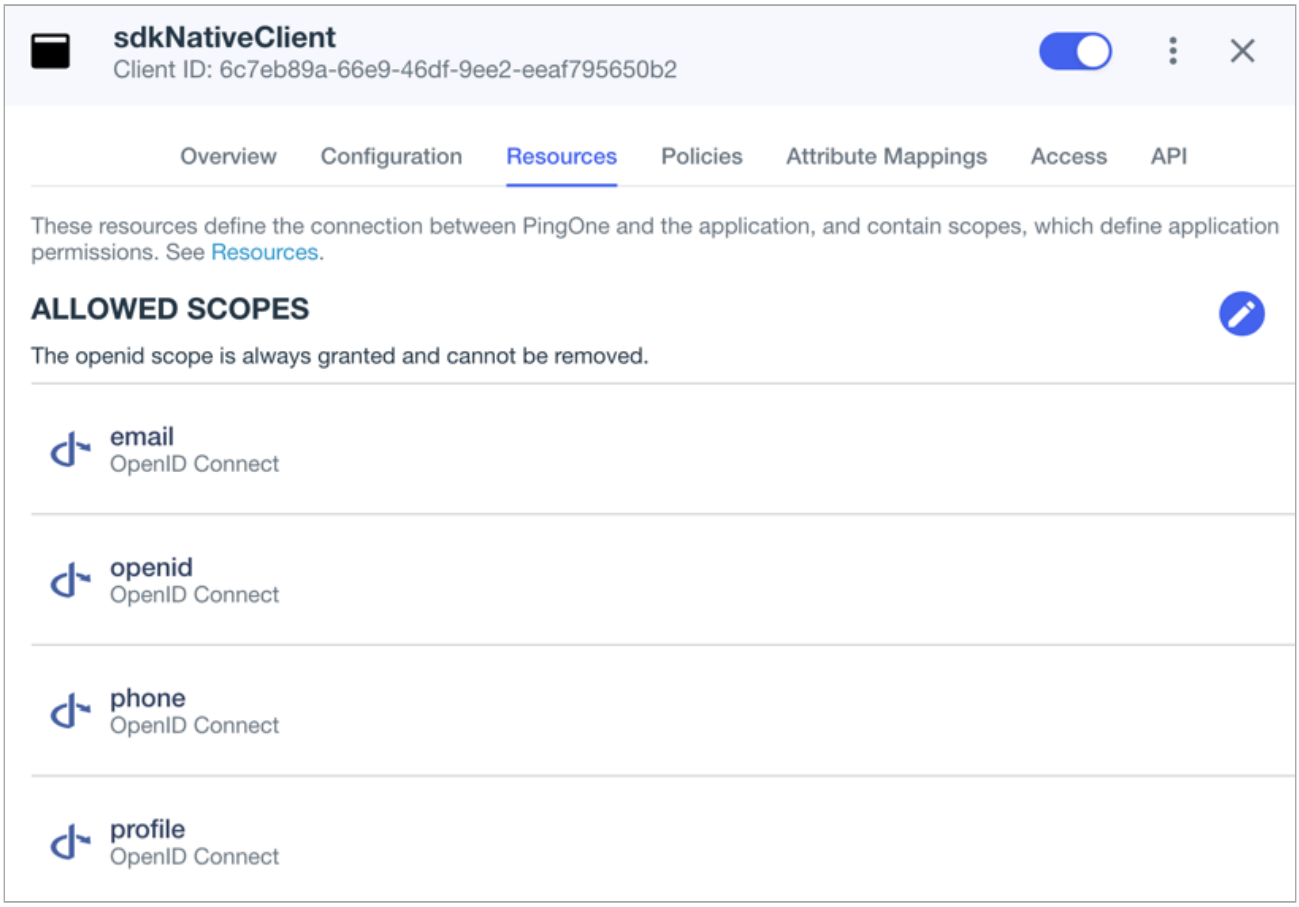


Figure 1. Adding scopes to an application.

8. Optionally, on the **Policies** tab, click the pencil icon (✎) to select the authentication policies for the application.

> **Note**
>
> Applications that have no authentication policy assignments use the environment's default authentication policy to authenticate users.

If you have a DaVinci license, you can select PingOne policies or DaVinci Flow policies, but not both. If you do not have a DaVinci license, the page only displays PingOne policies.

To use a *PingOne policy*:

1. Click **+ Add policies** and then select the policies that you want to apply to the application.
2. Click **Save**.

   PingOne applies the policies in the order in which they appear in the list. PingOne evaluates the first policy in the list first. If the requirements are not met, PingOne moves to the next one.

   For more information, see Authentication policies for applications.

To use a *DaVinci Flow policy*:

1. You must clear all PingOne policies. Click **Deselect all PingOne Policies**.
2. In the confirmation message, click **Continue**.

3. On the **DaVinci Policies** tab, select the policies that you want to apply to the application.
4. Click **Save**.

   PingOne applies the first policy in the list.

9. Click **Save**.
10. Enable the OAuth 2.0 client application by using the toggle next to its name:

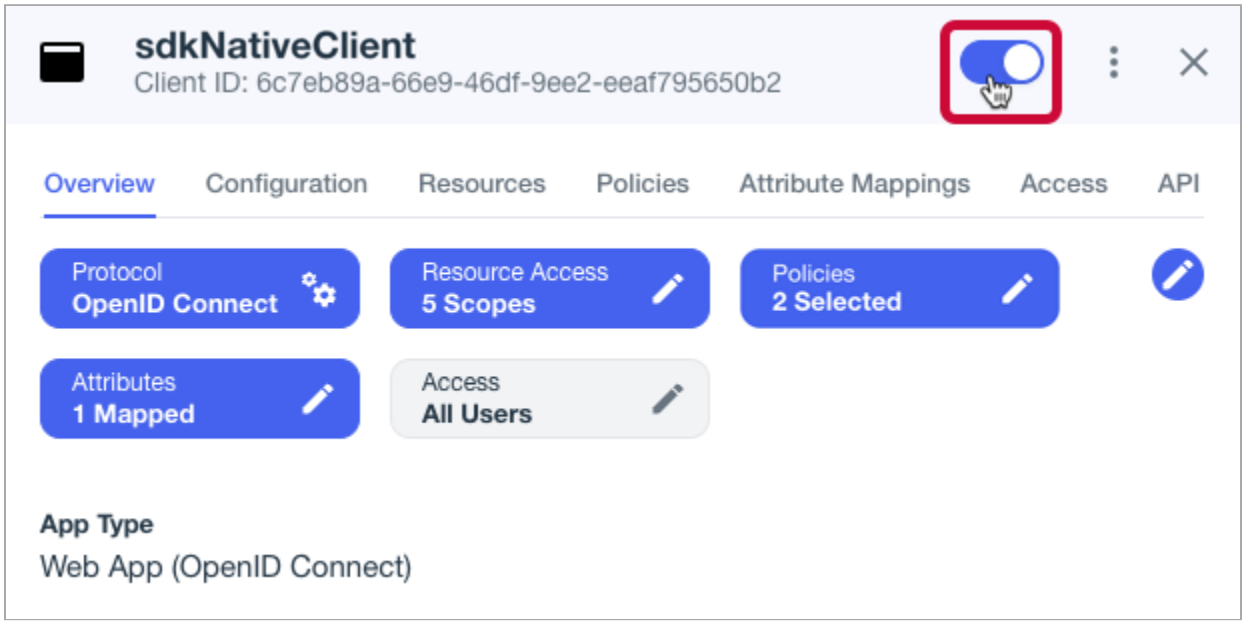


Figure 2. Enable the application using the toggle.

The application is now configured to accept client connections from and issue OAuth 2.0 tokens to the Android and iOS PingOne example applications and tutorials covered by this documentation.

Step 1. Download the samples

Prepare › **Download** › Configure › Run

To start this tutorial, you need to download the ForgeRock SDK sample apps repo, which contains the projects you will use.

1. In a web browser, navigate to the SDK Sample Apps repository.
2. Download the source code using one of the following methods:

Download a ZIP file

1. Click **Code**, and then click **Download ZIP**.
2. Extract the contents of the downloaded ZIP file to a suitable location.

Use a Git-compatible tool to clone the repo locally

1. Click **Code**, and then copy the HTTPS URL.
2. Use the URL to clone the repository to a suitable location.

   For example, from the command-line you could run:

```
git clone https://github.com/ForgeRock/sdk-sample-apps.git
```

The result of these steps is a local folder named `sdk-sample-apps`.

Step 2. Configure connection properties

Prepare › Download › **Configure** › Run

In this step, you configure the "swiftui-oidc" app to connect to the OAuth 2.0 application you created in PingOne, and display the login UI of the server.

1. In Xcode, on the **File** menu, click **Open**.
2. Navigate to the `sdk-sample-apps` folder you cloned in the previous step, navigate to `iOS` > `swiftui-oidc` > `PingExample` > `PingExample.xcodeproj`, and then click **Open**.
3. In the **Project Navigator** pane, navigate to **PingExample > PingExample > Utilities**, and open the `ConfigurationManager` file.
4. Locate the `ConfigurationViewModel` function which contains placeholder configuration properties.

   Tip

   The function is commented with `//TODO:` in the source to make it easier to locate.

```
return ConfigurationViewModel(
    clientId: "[CLIENT ID]",
    scopes: ["openid", "email", "address", "phone", "profile"],
    redirectUri: "[REDIRECT URI]",
    signOutUri: "[SIGN OUT URI]",
    discoveryEndpoint: "[DISCOVERY ENDPOINT URL]",
    environment: "[ENVIRONMENT - EITHER AIC OR PingOne]",
    cookieName: "[COOKIE NAME - OPTIONAL (Applicable for AIC only)]",
    browserSeletorType: .authSession
)
```

5. In the `ConfigurationViewModel` function, update the following properties with the values you obtained when preparing your environment.

clientId

The client ID from your OAuth 2.0 application in PingOne.

For example, `6c7eb89a-66e9-ab12-cd34-eeaf795650b2`

scopes

The scopes you added to your OAuth 2.0 application in PingOne.

For example, `openid profile email phone`

redirectUri

The `redirect_uri` to return to after logging in with the server UI, for example the URI to your client app.

> **Note**
>
> This value must exactly match a value configured in your OAuth 2.0 client.

For example, `org.forgerock.demo://oauth2redirect`.

signOutUri

Leave this property empty.

It signals that the SDK can use the ID token to end the user's session, and does not need to open and return from a web page to perform log out.

> **Note**
>
> You must have enabled the **Terminate User Session by ID Token** setting when creating the OAuth 2.0 client in PingOne if you leave this property empty.

discoveryEndpoint

The `.well-known` endpoint from your PingOne tenant.

To find the `.well-known` endpoint for an OAuth 2.0 client in PingOne:

1. Log in to your PingOne administration console.
2. Go to **Applications** > **Applications**, and then select the OAuth 2.0 client you created earlier.

   For example, sdkPublicClient.
3. On the **Configuration** tab, expand the **URLs** section, and then copy the **OIDC Discovery Endpoint** value.

For example, `https://auth.pingone.com/3072206d-c6ce-ch15-m0nd-f87e972c7cc3/as/.well-known/openid-configuration`

environment

Ensures the sample app uses the correct behavior for the different servers it supports, for example what logout parameters to use.

For PingOne and PingAM servers, specify `AIC`.

cookieName

Set this property to an empty string.

For example, `""`.

browserSeletorType

You can specify what type of browser the client iOS device opens to handle centralized login.

Each browser has slightly different characteristics, which make them suitable to different scenarios, as outlined in this table:

Browser type	Characteristics
`.authSession`	Opens a web authentication session browser. Designed specifically for authentication sessions, however it prompts the user before opening the browser with a modal that asks them to confirm the domain is allowed to authenticate them. This is the default option in the Ping SDK for iOS.
`.ephemeralAuthSession`	Opens a web authentication session browser, but enables the `prefersEphemeralWebBrowserSession` parameter. This browser type *does not* prompt the user before opening the browser with a modal. The difference between this and `.authSession` is that the browser does not include any existing data such as cookies in the request, and also discards any data obtained during the browser session, including any session tokens. When is `ephemeralAuthSession` suitable: ◦ ✕ `ephemeralAuthSession` is *not* suitable when you require single sign-on (SSO) between your iOS apps, as the browser will not maintain session tokens. ◦ ✕ `ephemeralAuthSession` is *not* suitable when you require a session token to log a user out of the server, for example for logging out of PingOne, as the browser will not maintain session tokens. ◦ ✓ Use `ephemeralAuthSession` when you do not want the user's existing sessions to affect the authentication.
`.nativeBrowserApp`	Opens the installed browser that is marked as the default by the user. Often Safari. The browser opens without any interaction from the user. However, the browser does display a modal when returning to your application.

Browser type	Characteristics
`.sfViewController`	Opens a Safari view controller browser. Your client app is ***not*** able to interact with the pages in the `sfViewController` or access the data or browsing history. The view controller opens within your app without any interaction from the user. As the user does not leave your app, the view controller does not need to display a warning modal when authentication is complete and control returns to your application.

The result resembles the following:

```
return ConfigurationViewModel(
    clientId: "6c7eb89a-66e9-ab12-cd34-eeaf795650b2",
    scopes: ["openid", "email", "phone", "profile"],
    redirectUri: "org.forgerock.demo://oauth2redirect",
    signOutUri: "",
    discoveryEndpoint: "https://auth.pingone.com/3072206d-c6ce-ch15-m0nd-f87e972c7cc3/as/.well-known/openid-
configuration",
    environment: "AIC",
    cookieName: "",
    browserSeletorType: .authSession
)
```

6. Optionally, specify ACR values to choose which authentication journey the server uses.

    1. Navigate to **PingExample > PingExample > ViewModels**, and open the `OIDCViewModel` file.

    2. In the `startOIDC()` function, add an `acr_values` parameter to the authorization request by using the `setCustomParam()` method:

```
public func startOIDC() async throws → FRUser {
    return try await withCheckedThrowingContinuation({
        (continuation: CheckedContinuation<FRUser, Error>) in
        Task { @MainActor in
            FRUser.browser()?
                .set(presentingViewController: self.topViewController!)
                .set(browserType:
ConfigurationManager.shared.currentConfigurationViewModel?.getBrowserType() ?? .authSession)
                .setCustomParam(key: "acr_values", value: "sdkUsernamePasswordJourney")
                .build().login { (user, error) in
                    if let frUser = user {
                        Task { @MainActor in
                            self.status = "User is authenticated"
                        }
                        continuation.resume(returning: frUser)
                    } else {
                        Task { @MainActor in
                            self.status = error?.localizedDescription ?? "Error was nil"
                        }
                        continuation.resume(throwing: error!)
                    }
                }
        }
    })
}
```

Enter one or more of the ACR mapping keys as configured in the OAuth 2.0 provider service.

To learn more, refer to Choose journeys with ACR values.

> **Tip**
>
> You can list the available keys by inspecting the `acr_values_supported` property in the output of your `/oauth2/.well-known/openid-configuration` endpoint.

Step 3. Test the app

Prepare › Download › Configure › **Run**

In this step, run the sample app that you configured in the previous step. The app performs OIDC login to your PingOne instance.

1. In Xcode, select **Product > Run**.

   Xcode launches the sample app in the iPhone simulator.

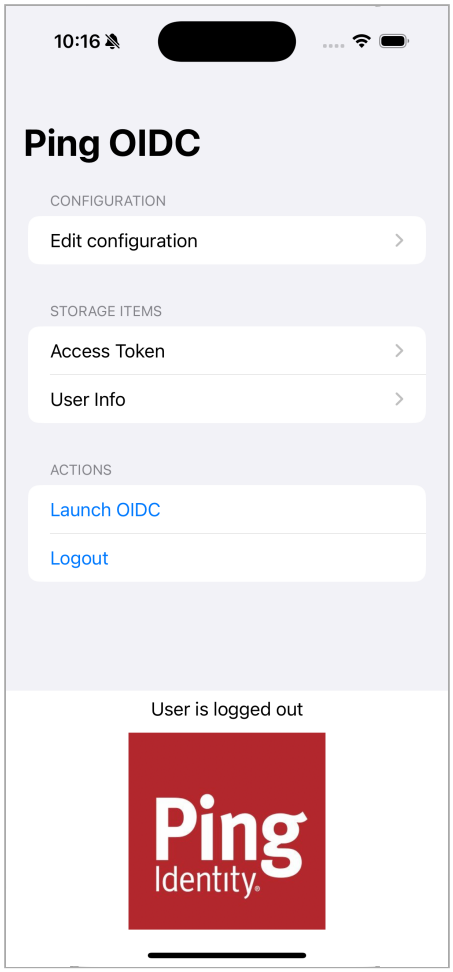


Figure 1. iOS OIDC login sample home screen

2. In the sample app on the iPhone simulator, tap **Edit configuration**, and verify or edit the configuration you entered in the previous step.

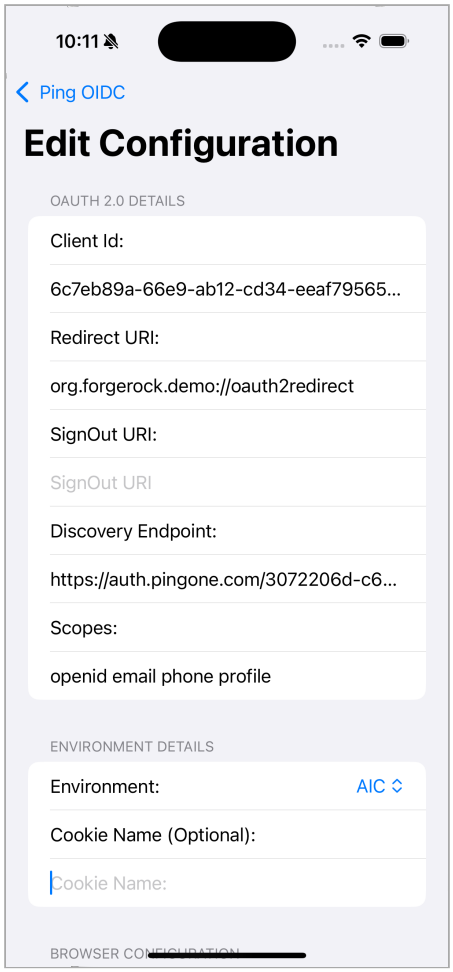


Figure 2. Verify the configuration settings

3. Tap < **Ping OIDC** to go back to the main menu, and then tap **Launch OIDC**.

> **Note**
>
> You might see a dialog asking if you want to open a browser. If you do, tap **Continue**.

The app launches a web browser and redirects to your PingOne login UI:

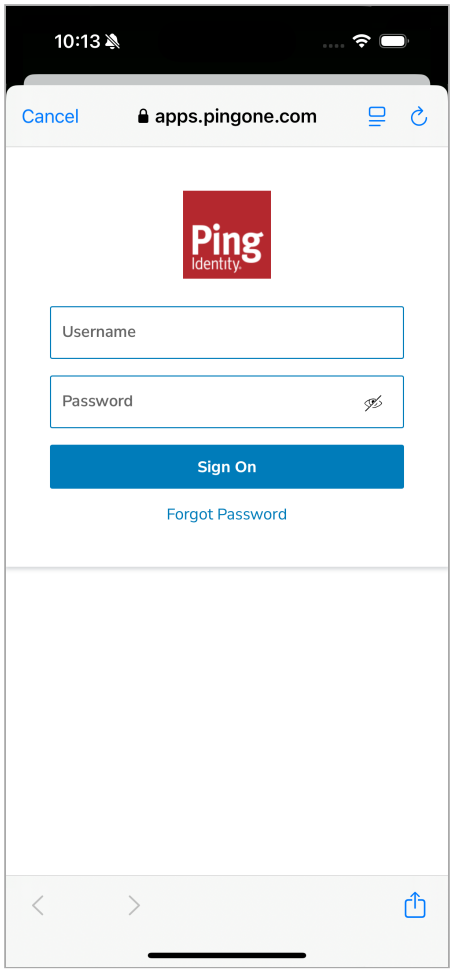


Figure 3. Browser launched and redirected to PingOne

4. Sign on as a demo user:
    - **Name**: `demo`
    - **Password**: `Ch4ng3it!`

If authentication is successful, the application displays the access token issued by PingOne.

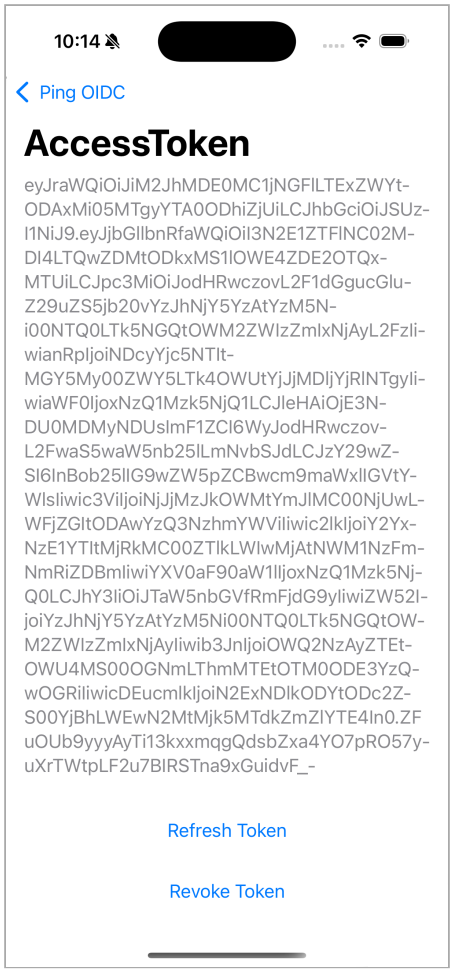


Figure 4. Access token after successful authentication

5. Tap < **Ping OIDC** to go back to the main menu, and then tap **User Info**. The app displays the user information relating to the access token:

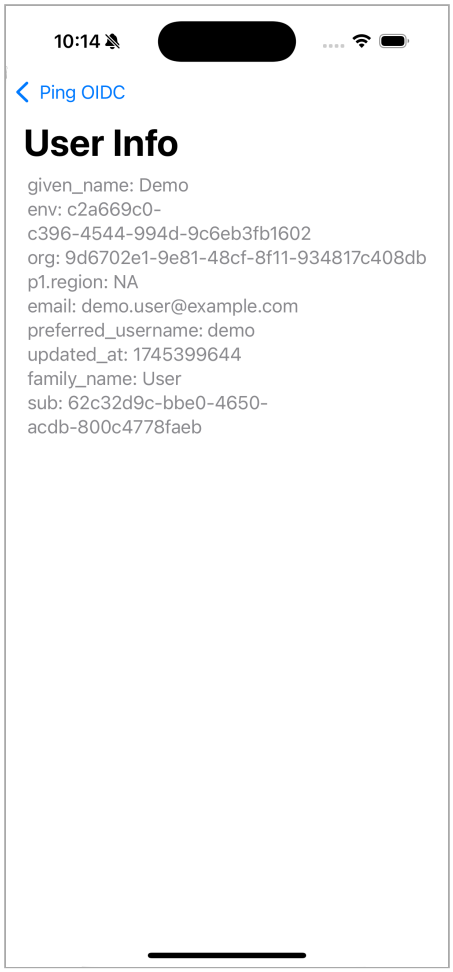


Figure 5. User info relating to the access token

6. Tap < **Ping OIDC** to go back to the main menu, and then tap **Logout**.

   The app logs the user out of the authorization server.

Tip

To verify the user is signed out:

1. In the PingOne administration console, navigate to **Directory > Users**.
2. Select the user you signed in as.
3. From the **Sevices** dropdown, select **Authentication**:

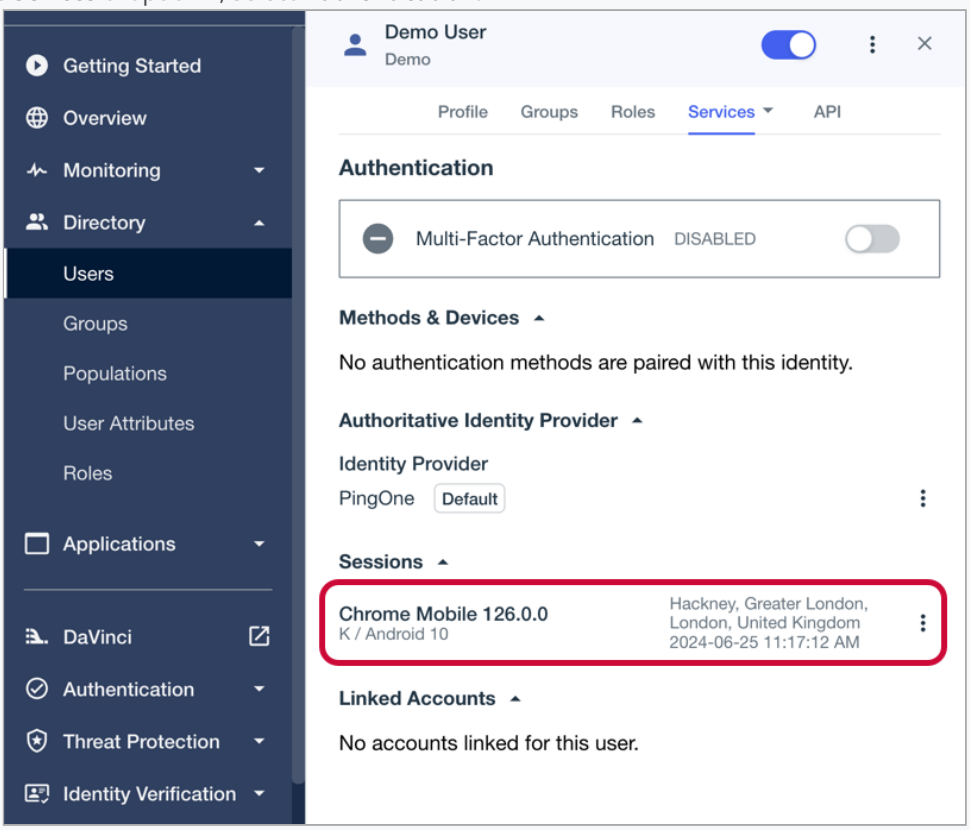


Figure 6. Checking a user's sessions in PingOne.

The **Sessions** section displays any existing sessions the user has, and whether they originate from a mobile device.

OIDC login to PingOne Advanced Identity Cloud tutorial for Android

Prepare › Download › Configure › Run

In this tutorial you update a sample app that uses OIDC-based login to obtain tokens by redirecting to the PingOne Advanced Identity Cloud UI for authentication.

The sample connects to the `.well-known` endpoint of your PingOne Advanced Identity Cloud server to obtain the correct URIs to authenticate the user, and redirects to your PingOne Advanced Identity Cloud server's login UI.

After authentication, PingOne Advanced Identity Cloud redirects the browser back to your application, which then obtains an OAuth 2.0 access token and displays the related user information.

Before you begin

Before you begin this tutorial ensure you have set up your PingOne Advanced Identity Cloud server with the required configuration.

For example, you will need to have an OAuth 2.0 client application set up, and a demo user to authenticate.

Complete prerequisites »

Step 1. Download the samples

To start this tutorial, you need to download the SDK sample apps repo, which contains the projects you will use.

Start step 1 »

Step 2. Configure connection properties

In this step, you configure the sample app to connect to the OAuth 2.0 application you created in PingOne Advanced Identity Cloud.

Start step 2 »

Step 3. Test the app

To test the app, run the sample that you configured in the previous step.

The sample connects to your PingOne Advanced Identity Cloud server to obtain the correct URIs to authenticate the user, and redirects the browser to your PingOne Advanced Identity Cloud server.

After authentication, PingOne Advanced Identity Cloud redirects the browser back to your application, which then obtains an OAuth 2.0 access token and displays the related user information.

Test app »

Before you begin

Prepare › Download › Configure › Run

To successfully complete this tutorial refer to the prerequisites in this section.

The tutorial also requires a configured PingOne Advanced Identity Cloud tenant.

Compatibility

iOS

This sample app is compatible with iOS 12 and later.

Prerequisites

Xcode

You can download the latest version for free from https://developer.apple.com/xcode/.

Server configuration

This tutorial requires you to configure your PingOne Advanced Identity Cloud tenant as follows:

The samples and tutorials in this documentation often require that you have an identity set up so that you can test authentication.

To create a demo user in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. In the left panel, click **Identities** > **Manage**.
3. Click **+ New Alpha realm - User**.
4. Enter the following details:
    - **Username** = `demo`
    - **First Name** = `Demo`
    - **Last Name** = `User`
    - **Email Address** = `demo.user@example.com`
    - **Password** = `Ch4ng3it!`
5. Click **Save**.

Authentication journeys provide fine-grained authentication by allowing multiple paths and decision points throughout the flow. Authentication journeys are made up of nodes that define actions taken during authentication.

Each node performs a single task, such as collecting a username or making a simple decision. Nodes can have multiple outcomes rather than just success or failure. For details, see the Authentication nodes configuration reference in the PingAM documentation.

To create a simple journey for use when testing the Ping SDKs, follow these steps:

1. In your PingOne Advanced Identity Cloud tenant, navigate to **Journeys**, and click **+ New Journey**.
2. Enter a name, such as `sdkUsernamePasswordJourney` and click **Save**.

The authentication journey designer appears.

3. Drag the following nodes into the designer area:
    - **Page Node**
    - **Platform Username**
    - **Platform Password**
    - **Data Store Decision**
4. Drag and drop the **Platform Username** and **Platform Password** nodes onto the **Page Node**, so that they both appear on the same page when logging in.
5. Connect the nodes as follows:

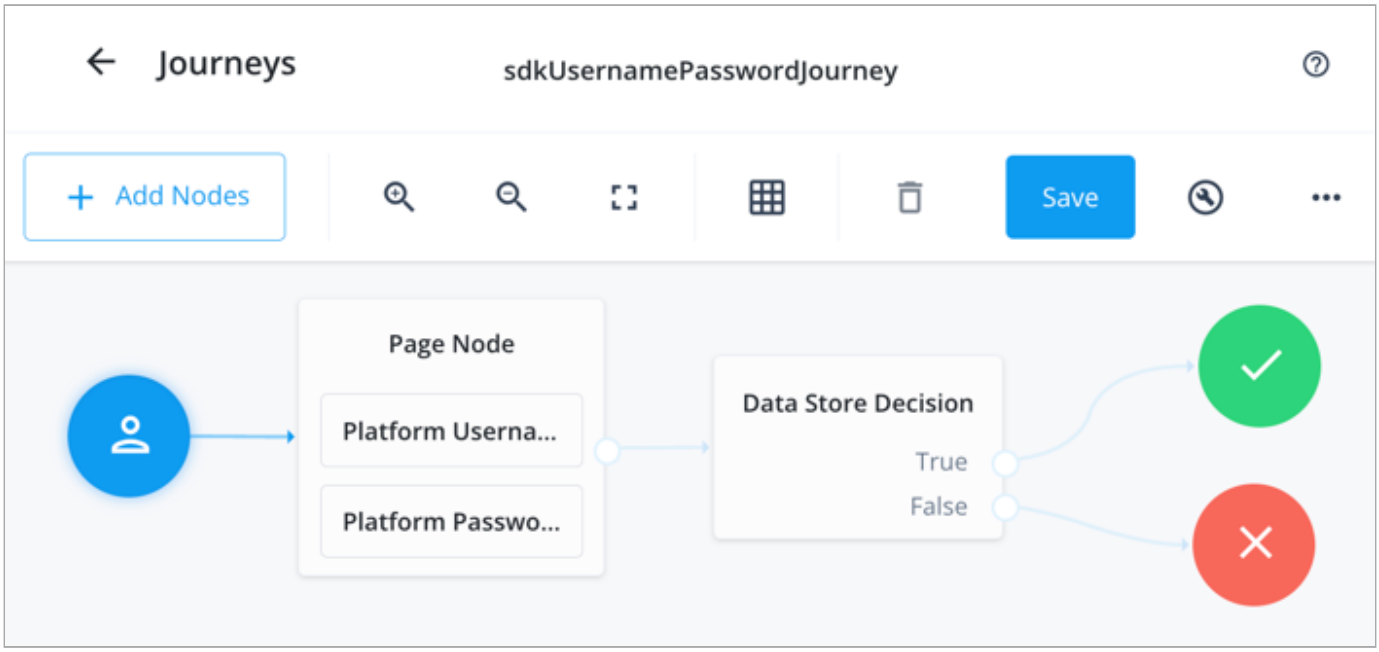


Figure 1. Example username and password authentication journey

6. Click **Save**.

Public clients do not use a client secret to obtain tokens because they are unable to keep them hidden. The Ping SDKs commonly use this type of client to obtain tokens, as they cannot guarantee safekeeping of the client credentials in a browser or on a mobile device.

To register a *public* OAuth 2.0 client application for use with the SDKs in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. In the left panel, click **Applications**.
3. Click **+ Custom Application**.
4. Select **OIDC - OpenId Connect** as the sign-in method, and then click **Next**.
5. Select **Native / SPA** as the application type, and then click **Next**.

6. In **Name**, enter a name for the application, such as `Public SDK Client`.
7. In **Owners**, select a user that is responsible for maintaining the application, and then click **Next**.

> **Tip**
>
> When trying out the SDKs, you could select the `demo` user you created previously.

8. In **Client ID**, enter `sdkPublicClient`, and then click **Create Application**.

   PingOne Advanced Identity Cloud creates the application and displays the details screen.
9. On the **Sign On** tab:
   1. In **Sign-In URLs**, enter the following values:

      `org.forgerock.demo://oauth2redirect`

      > **Important**
      >
      > Also add any other domains where you host SDK applications.

   2. In **Grant Types**, enter the following values:

      `Authorization Code`

      `Refresh Token`

   3. In **Scopes**, enter the following values:

      `openid profile email address`
10. Click Show advanced settings, and on the **Authentication** tab:
    1. In **Token Endpoint Authentication Method**, select `none`.
    2. In **Client Type**, select `Public`.
    3. Enable the **Implied Consent** property.
11. Click **Save**.

The application is now configured to accept client connections from and issue OAuth 2.0 tokens to the example applications and tutorials covered by this documentation.

The provider specifies the supported OAuth 2.0 configuration options for a realm.

To ensure the PingOne Advanced Identity Cloud OAuth 2.0 provider service is configured for use with the Ping SDKs, follow these steps:

1. In your PingOne Advanced Identity Cloud tenant, navigate to **Native Consoles** > **Access Management**.
2. In the left panel, click **Services**.
3. In the list of services, click **OAuth2 Provider**.
4. On the **Core** tab, ensure **Issue Refresh Tokens** is enabled.

5. On the **Consent** tab, ensure **Allow Clients to Skip Consent** is enabled.
6. Click **Save Changes**.

Step 1. Download the samples

Prepare › **Download** › Configure › Run

To start this tutorial, you need to download the ForgeRock SDK sample apps repo, which contains the projects you will use.

1. In a web browser, navigate to the SDK Sample Apps repository.
2. Download the source code using one of the following methods:

   Download a ZIP file

   1. Click **Code**, and then click **Download ZIP**.
   2. Extract the contents of the downloaded ZIP file to a suitable location.

   Use a Git-compatible tool to clone the repo locally

   1. Click **Code**, and then copy the HTTPS URL.
   2. Use the URL to clone the repository to a suitable location.

      For example, from the command-line you could run:

      ```
      git clone https://github.com/ForgeRock/sdk-sample-apps.git
      ```

The result of these steps is a local folder named `sdk-sample-apps`.

Step 2. Configure connection properties

Prepare › Download › **Configure** › Run

In this step, you configure the "swiftui-oidc" app to connect to the OAuth 2.0 application you created in PingOne Advanced Identity Cloud, and display the login UI of the server.

1. In Xcode, on the **File** menu, click **Open**.
2. Navigate to the `sdk-sample-apps` folder you cloned in the previous step, navigate to `iOS` > `swiftui-oidc` > `PingExample` > `PingExample.xcodeproj`, and then click **Open**.
3. In the **Project Navigator** pane, navigate to **PingExample > PingExample > Utilities**, and open the `ConfigurationManager` file.

4. Locate the `ConfigurationViewModel` function which contains placeholder configuration properties.

> **Tip**
>
> The function is commented with `//TODO:` in the source to make it easier to locate.

```
return ConfigurationViewModel(
    clientId: "[CLIENT ID]",
    scopes: ["openid", "email", "address", "phone", "profile"],
    redirectUri: "[REDIRECT URI]",
    signOutUri: "[SIGN OUT URI]",
    discoveryEndpoint: "[DISCOVERY ENDPOINT URL]",
    environment: "[ENVIRONMENT - EITHER AIC OR PingOne]",
    cookieName: "[COOKIE NAME - OPTIONAL (Applicable for AIC only)]",
    browserSeletorType: .authSession
)
```

5. In the `ConfigurationViewModel` function, update the following properties with the values you obtained when preparing your environment.

clientId

The client ID from your OAuth 2.0 application in PingOne Advanced Identity Cloud.

For example, `sdkPublicClient`

scopes

The scopes you added to your OAuth 2.0 application in PingOne Advanced Identity Cloud.

For example, `address email openid phone profile`

redirectUri

The `redirect_uri` to return to after logging in with the server UI, for example the URI to your client app.

> **Note**
>
> This value must exactly match a value configured in your OAuth 2.0 client.

For example, `org.forgerock.demo://oauth2redirect`.

signOutUri

The URI to redirect to after logging out of the authorization server, for example the URI to your client app.

> **Note**
>
> This value must exactly match a value configured in your OAuth 2.0 client.

For example, `org.forgerock.demo://oauth2redirect`.

discoveryEndpoint

The `.well-known` endpoint from your PingOne Advanced Identity Cloud tenant.

You can view the `.well-known` endpoint for an OAuth 2.0 client in the PingOne Advanced Identity Cloud admin console:

1. Log in to your PingOne Advanced Identity Cloud administration console.
2. Click **Applications**, and then select the OAuth 2.0 client you created earlier. For example, sdkPublicClient.
3. On the **Sign On** tab, in the **Client Credentials** section, copy the **Discovery URI** value.

For example, `https://openam-forgerock-sdks.forgeblocks.com/am/oauth2/alpha/.well-known/openid-configuration`

environment

Ensures the sample app uses the correct behavior for the different servers it supports, for example what logout parameters to use.

For PingOne Advanced Identity Cloud and PingAM servers, specify `AIC`.

cookieName

The name of the cookie your PingOne Advanced Identity Cloud tenant uses to store SSO tokens in client browsers.

To locate the cookie name in an PingOne Advanced Identity Cloud tenant:

1. Navigate to **Tenant settings** > **Global Settings**
2. Copy the value of the **Cookie** property.

For example, `ch15fefc5407912`

browserSeletorType

You can specify what type of browser the client iOS device opens to handle centralized login.

Each browser has slightly different characteristics, which make them suitable to different scenarios, as outlined in this table:

Browser type	Characteristics
`.authSession`	Opens a web authentication session browser. Designed specifically for authentication sessions, however it prompts the user before opening the browser with a modal that asks them to confirm the domain is allowed to authenticate them. This is the default option in the Ping SDK for iOS.

Browser type	Characteristics
`.ephemeralAuthSession`	Opens a web authentication session browser, but enables the `prefersEphemeralWebBrowserSession` parameter. This browser type *does not* prompt the user before opening the browser with a modal. The difference between this and `.authSession` is that the browser does not include any existing data such as cookies in the request, and also discards any data obtained during the browser session, including any session tokens. When is `ephemeralAuthSession` suitable: ◦ ✕ `ephemeralAuthSession` is *not* suitable when you require single sign-on (SSO) between your iOS apps, as the browser will not maintain session tokens. ◦ ✕ `ephemeralAuthSession` is *not* suitable when you require a session token to log a user out of the server, for example for logging out of PingOne, as the browser will not maintain session tokens. ◦ ✓ Use `ephemeralAuthSession` when you do not want the user's existing sessions to affect the authentication.
`.nativeBrowserApp`	Opens the installed browser that is marked as the default by the user. Often Safari. The browser opens without any interaction from the user. However, the browser does display a modal when returning to your application.
`.sfViewController`	Opens a Safari view controller browser. Your client app is ***not*** able to interact with the pages in the `sfViewController` or access the data or browsing history. The view controller opens within your app without any interaction from the user. As the user does not leave your app, the view controller does not need to display a warning modal when authentication is complete and control returns to your application.

The result resembles the following:

```
return ConfigurationViewModel(
    clientId: "sdkPublicClient",
    scopes: ["openid", "email", "address", "phone", "profile"],
    redirectUri: "org.forgerock.demo://oauth2redirect",
    signOutUri: "org.forgerock.demo://oauth2redirect",
    discoveryEndpoint: "https://openam-forgerock-sdks.forgeblocks.com/am/oauth2/alpha/.well-known/openid-
configuration",
    environment: "AIC",
    cookieName: "ch15fefc5407912",
    browserSeletorType: .authSession
)
```

6. Optionally, specify ACR values to choose which authentication journey the server uses.

    1. Navigate to **PingExample > PingExample > ViewModels**, and open the `OIDCViewModel` file.

    2. In the `startOIDC()` function, add an `acr_values` parameter to the authorization request by using the `setCustomParam()` method:

```
public func startOIDC() async throws → FRUser {
    return try await withCheckedThrowingContinuation({
        (continuation: CheckedContinuation<FRUser, Error>) in
        Task { @MainActor in
            FRUser.browser()?
                .set(presentingViewController: self.topViewController!)
                .set(browserType:
ConfigurationManager.shared.currentConfigurationViewModel?.getBrowserType() ?? .authSession)
                .setCustomParam(key: "acr_values", value: "sdkUsernamePasswordJourney")
                .build().login { (user, error) in
                    if let frUser = user {
                        Task { @MainActor in
                            self.status = "User is authenticated"
                        }
                        continuation.resume(returning: frUser)
                    } else {
                        Task { @MainActor in
                            self.status = error?.localizedDescription ?? "Error was nil"
                        }
                        continuation.resume(throwing: error!)
                    }
                }
        }
    })
}
```

    Enter one or more of the ACR mapping keys as configured in the OAuth 2.0 provider service.

    To learn more, refer to Choose journeys with ACR values.

    > **Tip**
    >
    > You can list the available keys by inspecting the `acr_values_supported` property in the output of your `/oauth2/.well-known/openid-configuration` endpoint.

Step 3. Test the app

Prepare › Download › Configure › **Run**

In this step, run the sample app that you configured in the previous step. The app performs OIDC login to your PingOne Advanced Identity Cloud instance.

1. In Xcode, select **Product > Run**.

    Xcode launches the sample app in the iPhone simulator.

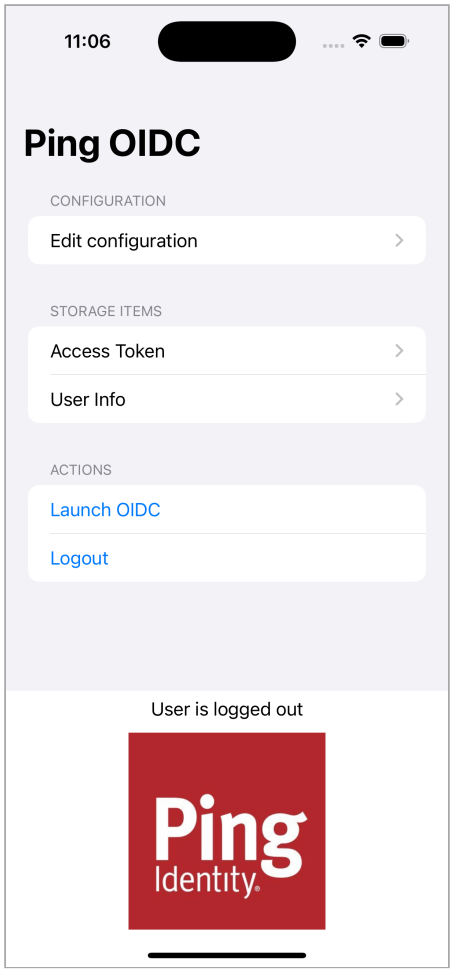


Figure 1. iOS OIDC login sample home screen

2. In the sample app on the iPhone simulator, tap **Edit configuration**, and verify or edit the configuration you entered in the previous step.

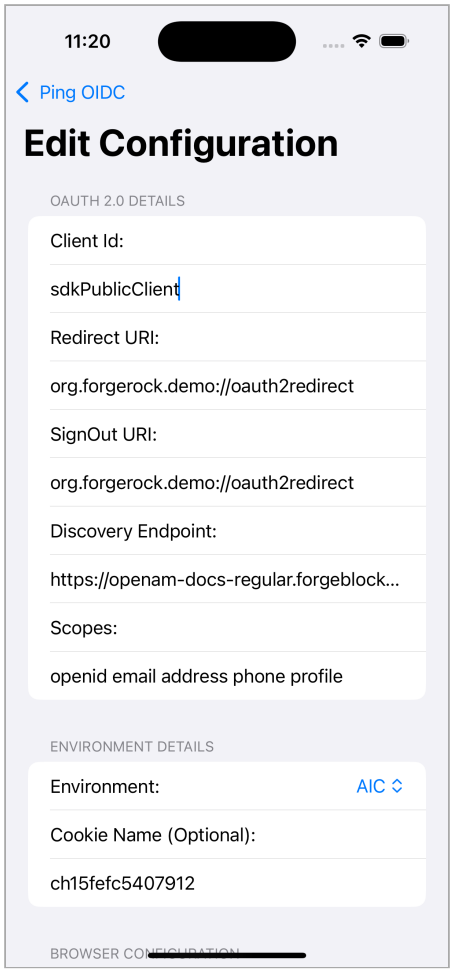


Figure 2. Verify the configuration settings

3. Tap < **Ping OIDC** to go back to the main menu, and then tap **Launch OIDC**.

> **Note**
>
> You might see a dialog asking if you want to open a browser. If you do, tap **Continue**.

The app launches a web browser and redirects to your PingOne Advanced Identity Cloud login UI:

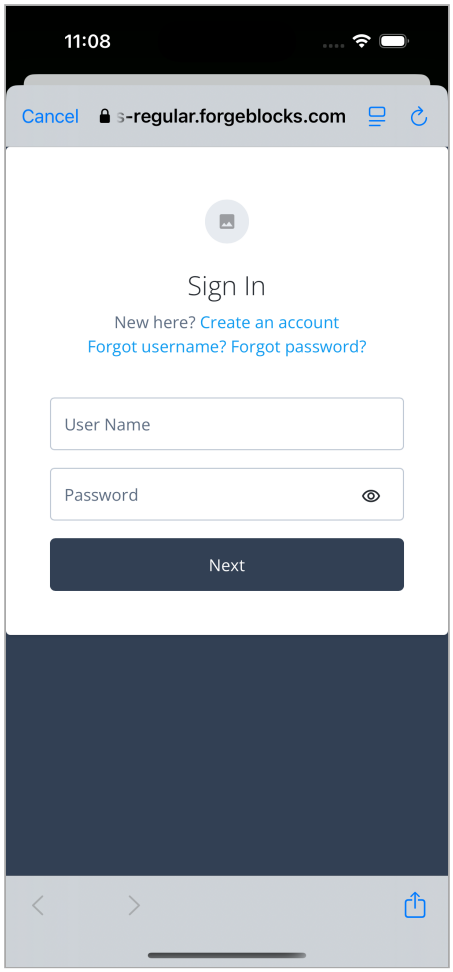


Figure 3. Browser launched and redirected to PingOne Advanced Identity Cloud

4. Sign on as a demo user:
    - **Name**: `demo`
    - **Password**: `Ch4ng3it!`

If authentication is successful, the application displays the access token issued by PingOne Advanced Identity Cloud.

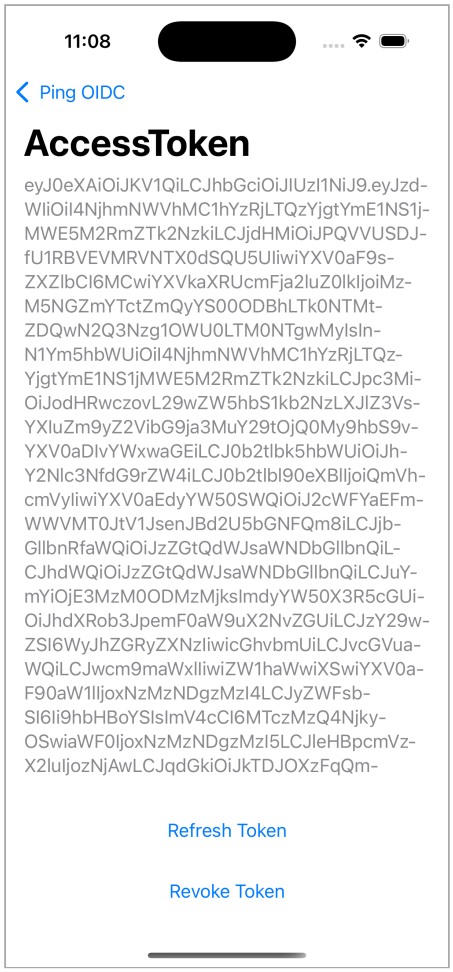


Figure 4. Access token after successful authentication

5. Tap < **Ping OIDC** to go back to the main menu, and then tap **User Info**.

   The app displays the information relating to the access token:

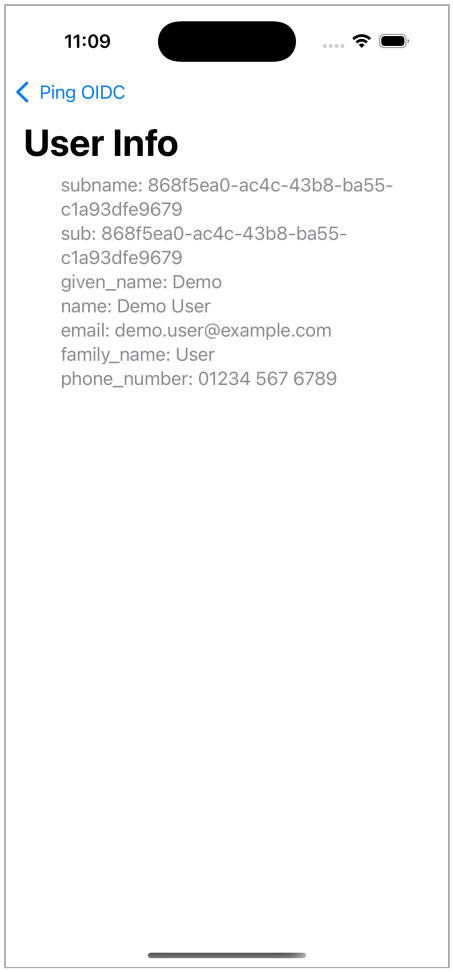


Figure 5. User info relating to the access token

6. Tap < **Ping OIDC** to go back to the main menu, and then tap **Logout**.

   The app logs the user out of the authorization server and prints a message to the Xcode console:

```
[FRCore][4.8.0] [🌐 - Network] Response | [☑ 204] :
  https://openam-forgerock-sdks.forgeblocks.com:443/am/oauth2/alpha/connect/endSession?
id_token_hint=eyJ0...sbrA&client_id=sdkPublicClient in 34 ms
[FRAuth][4.8.0] [FRUser.swift:211 : logout()] [Verbose]
  Invalidating OIDC Session successful
```

OIDC login to PingAM tutorial for Android

Prepare › Download › Configure › Run

In this tutorial you update a sample app that uses OIDC-based login to obtain tokens by redirecting to the PingAM UI for authentication.

The sample connects to the `.well-known` endpoint of your PingAM server to obtain the correct URIs to authenticate the user, and redirects to your PingAM server's login UI.

After authentication, PingAM redirects the browser back to your application, which then obtains an OAuth 2.0 access token and displays the related user information.

Before you begin

Before you begin this tutorial ensure you have set up your PingAM server with the required configuration.

For example, you will need to have an OAuth 2.0 client application set up, and a demo user to authenticate.

Complete prerequisites »

Step 1. Download the samples

To start this tutorial, you need to download the SDK sample apps repo, which contains the projects you will use.

Start step 1 »

Step 2. Configure connection properties

In this step, you configure the sample app to connect to the OAuth 2.0 application you created in PingAM.

Start step 2 »

Step 3. Test the app

To test the app, run the sample that you configured in the previous step.

The sample connects to your PingAM server to obtain the correct URIs to authenticate the user, and redirects the browser to your PingAM server.

After authentication, PingAM redirects the browser back to your application, which then obtains an OAuth 2.0 access token and displays the related user information.

Test app »

Before you begin

Prepare › Download › Configure › Run

To successfully complete this tutorial refer to the prerequisites in this section.

The tutorial also requires a configured PingAM server.

Compatibility

iOS

This sample app is compatible with iOS 12 and later.

Prerequisites

Xcode

You can download the latest version for free from https://developer.apple.com/xcode/.

Server configuration

This tutorial requires you to configure your PingAM server as follows:

The samples and tutorials in this documentation often require that you have an identity set up so that you can test authentication.

To create a demo user in PingAM, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. Navigate to **Identities**, and then click **+ Add Identity**.
3. Enter the following details:
   - **User ID** = `demo`
   - **Password** = `Ch4ng3it!`
   - **Email Address** = `demo.user@example.com`
4. Click **Create**.

Authentication trees provide fine-grained authentication by allowing multiple paths and decision points throughout the authentication flow. Authentication trees are made up of nodes that define actions taken during authentication.

Each node performs a single task, such as collecting a username or making a simple decision. Nodes can have multiple outcomes rather than just success or failure. For details, see the Authentication nodes configuration reference in the PingAM documentation.

To create a simple tree for use when testing the Ping SDKs, follow these steps:

1. Under **Realm Overview**, click **Authentication Trees**, then click **Create Tree**.

2. Enter a tree name, for example `sdkUsernamePasswordJourney`, and then click **Create**.

   The authentication tree designer appears, showing the **Start** entry point connected to the **Failure** exit point.

3. Drag the following nodes from the **Components** panel on the left side into the designer area:

   - **Page Node**
   - **Username Collector**
   - **Password Collector**
   - **Data Store Decision**

4. Drag and drop the **Username Collector** and **Password Collector** nodes onto the **Page Node**, so that they both appear on the same page when logging in.

5. Connect the nodes as follows:

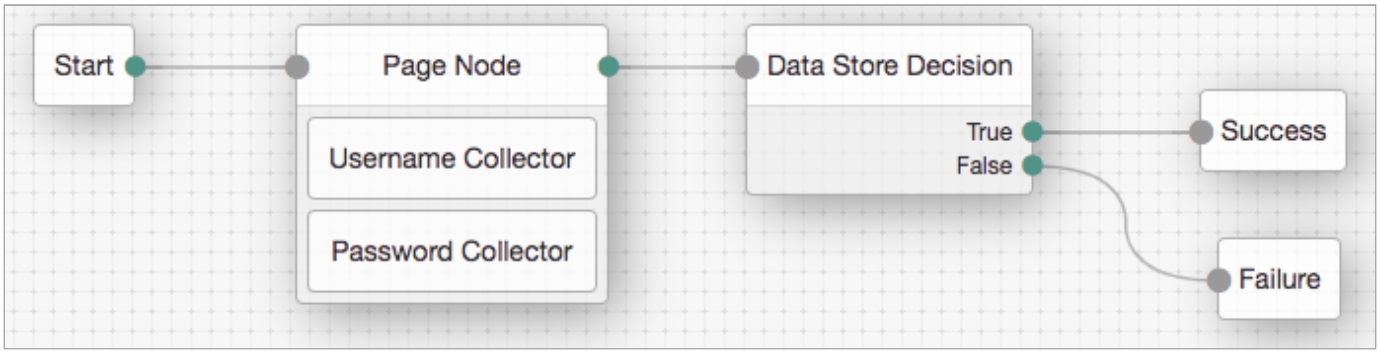


Figure 1. Example username and password authentication tree

6. Select the **Page Node**, and in the **Properties** pane, set the **Stage** property to `UsernamePassword`.

   > **Tip**
   >
   > You can configure the node properties by selecting a node and altering properties in the right-hand panel.

   One of the samples uses this specific value to determine the custom UI to display.

7. Click **Save**.

Public clients do not use a client secret to obtain tokens because they are unable to keep them hidden. The Ping SDKs commonly use this type of client to obtain tokens, as they cannot guarantee safekeeping of the client credentials in a browser or on a mobile device.

To register a *public* OAuth 2.0 client application for use with the SDKs in AM, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. Navigate to **Applications** > **OAuth 2.0** > **Clients**, and then click **+ Add Client**.
3. In **Client ID**, enter `sdkPublicClient`.
4. Leave **Client secret** empty.
5. In **Redirection URIs**, enter the following values:

`org.forgerock.demo://oauth2redirect`

> **Important**
>
> Also add any other domains where you will be hosting SDK applications.

6. In **Scopes**, enter the following values:

   `openid profile email address`

7. Click **Create**.

   PingAM creates the new OAuth 2.0 client, and displays the properties for further configuration.

8. On the **Core** tab:

   1. In **Client type**, select `Public`.
   2. Disable **Allow wildcard ports in redirect URIs**.
   3. Click **Save Changes**.

9. On the **Advanced** tab:

   1. In **Grant Types**, enter the following values:

      ```
      Authorization Code
      Refresh Token
      ```

   2. In **Token Endpoint Authentication Method**, select `None`.
   3. Enable the **Implied consent** property.

10. Click **Save Changes**.

The provider specifies the supported OAuth 2.0 configuration options for a realm.

To ensure the PingAM OAuth 2.0 provider service is configured for use with the Ping SDKs, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. In the left panel, click **Services**.
3. In the list of services, click **OAuth2 Provider**.
4. On the **Core** tab, ensure **Issue Refresh Tokens** is enabled.
5. On the **Consent** tab, ensure **Allow Clients to Skip Consent** is enabled.
6. Click **Save Changes**.

Step 1. Download the samples

Prepare › **Download** › Configure › Run

To start this tutorial, you need to download the ForgeRock SDK sample apps repo, which contains the projects you will use.

1. In a web browser, navigate to the SDK Sample Apps repository.
2. Download the source code using one of the following methods:

Download a ZIP file

1. Click **Code**, and then click **Download ZIP**.
2. Extract the contents of the downloaded ZIP file to a suitable location.

Use a Git-compatible tool to clone the repo locally

1. Click **Code**, and then copy the HTTPS URL.
2. Use the URL to clone the repository to a suitable location.

   For example, from the command-line you could run:

```
git clone https://github.com/ForgeRock/sdk-sample-apps.git
```

The result of these steps is a local folder named `sdk-sample-apps`.

Step 2. Configure connection properties

Prepare › Download › **Configure** › Run

In this step, you configure the "swiftui-oidc" app to connect to the OAuth 2.0 application you created in PingAM, and display the login UI of the server.

1. In Xcode, on the **File** menu, click **Open**.
2. Navigate to the `sdk-sample-apps` folder you cloned in the previous step, navigate to `iOS` > `swiftui-oidc` > `PingExample` > `PingExample.xcodeproj`, and then click **Open**.
3. In the **Project Navigator** pane, navigate to **PingExample > PingExample > Utilities**, and open the `ConfigurationManager` file.
4. Locate the `ConfigurationViewModel` function which contains placeholder configuration properties.

> **Tip**
>
> The function is commented with `//TODO:` in the source to make it easier to locate.

```
return ConfigurationViewModel(
    clientId: "[CLIENT ID]",
    scopes: ["openid", "email", "address", "phone", "profile"],
    redirectUri: "[REDIRECT URI]",
    signOutUri: "[SIGN OUT URI]",
    discoveryEndpoint: "[DISCOVERY ENDPOINT URL]",
    environment: "[ENVIRONMENT - EITHER AIC OR PingOne]",
    cookieName: "[COOKIE NAME - OPTIONAL (Applicable for AIC only)]",
    browserSeletorType: .authSession
)
```

5. In the `ConfigurationViewModel` function, update the following properties with the values you obtained when preparing your environment.

clientId

The client ID from your OAuth 2.0 application in PingAM.

For example, `sdkPublicClient`

scopes

The scopes you added to your OAuth 2.0 application in PingAM.

For example, `address email openid phone profile`

redirectUri

The `redirect_uri` to return to after logging in with the server UI, for example the URI to your client app.

> **Note**
>
> This value must exactly match a value configured in your OAuth 2.0 client.

For example, `org.forgerock.demo://oauth2redirect`.

signOutUri

The URI to redirect to after logging out of the authorization server, for example the URI to your client app.

> **Note**
>
> This value must exactly match a value configured in your OAuth 2.0 client.

For example, `org.forgerock.demo://oauth2redirect`.

discoveryEndpoint

The `.well-known` endpoint from your PingAM server.

For example, `https://openam.example.com:8443/openam/oauth2/.well-known/openid-configuration`

environment

Ensures the sample app uses the correct behavior for the different servers it supports, for example what logout parameters to use.

For PingAM servers, specify `AIC`.

cookieName

The name of the cookie your PingAM server uses to store SSO tokens in client browsers.

For example, `iPlanetDirectoryPro`

browserSeletorType

You can specify what type of browser the client iOS device opens to handle centralized login.

Each browser has slightly different characteristics, which make them suitable to different scenarios, as outlined in this table:

Browser type	Characteristics
`.authSession`	Opens a web authentication session browser. Designed specifically for authentication sessions, however it prompts the user before opening the browser with a modal that asks them to confirm the domain is allowed to authenticate them. This is the default option in the Ping SDK for iOS.
`.ephemeralAuthSession`	Opens a web authentication session browser, but enables the `prefersEphemeralWebBrowserSession` parameter. This browser type *does not* prompt the user before opening the browser with a modal. The difference between this and `.authSession` is that the browser does not include any existing data such as cookies in the request, and also discards any data obtained during the browser session, including any session tokens. When is `ephemeralAuthSession` suitable: ◦ ✕ `ephemeralAuthSession` is *not* suitable when you require single sign-on (SSO) between your iOS apps, as the browser will not maintain session tokens. ◦ ✕ `ephemeralAuthSession` is *not* suitable when you require a session token to log a user out of the server, for example for logging out of PingOne, as the browser will not maintain session tokens. ◦ ✓ Use `ephemeralAuthSession` when you do not want the user's existing sessions to affect the authentication.

Browser type	Characteristics
`.nativeBrowserApp`	Opens the installed browser that is marked as the default by the user. Often Safari. The browser opens without any interaction from the user. However, the browser does display a modal when returning to your application.
`.sfViewController`	Opens a Safari view controller browser. Your client app is ***not*** able to interact with the pages in the `sfViewController` or access the data or browsing history. The view controller opens within your app without any interaction from the user. As the user does not leave your app, the view controller does not need to display a warning modal when authentication is complete and control returns to your application.

The result resembles the following:

```
return ConfigurationViewModel(
    clientId: "sdkPublicClient",
    scopes: ["openid", "email", "address", "phone", "profile"],
    redirectUri: "org.forgerock.demo://oauth2redirect",
    signOutUri: "org.forgerock.demo://oauth2redirect",
    discoveryEndpoint: "https://openam.example.com:8443/openam/oauth2/.well-known/openid-configuration",
    environment: "AIC",
    cookieName: "iPlanetDirectoryPro",
    browserSeletorType: .authSession
)
```

6. Optionally, specify ACR values to choose which authentication journey the server uses.

    1. Navigate to **PingExample > PingExample > ViewModels**, and open the `OIDCViewModel` file.

    2. In the `startOIDC()` function, add an `acr_values` parameter to the authorization request by using the `setCustomParam()` method:

```
public func startOIDC() async throws → FRUser {
    return try await withCheckedThrowingContinuation({
        (continuation: CheckedContinuation<FRUser, Error>) in
        Task { @MainActor in
            FRUser.browser()?
                .set(presentingViewController: self.topViewController!)
                .set(browserType:
ConfigurationManager.shared.currentConfigurationViewModel?.getBrowserType() ?? .authSession)
                .setCustomParam(key: "acr_values", value: "sdkUsernamePasswordJourney")
                .build().login { (user, error) in
                    if let frUser = user {
                        Task { @MainActor in
                            self.status = "User is authenticated"
                        }
                        continuation.resume(returning: frUser)
                    } else {
                        Task { @MainActor in
                            self.status = error?.localizedDescription ?? "Error was nil"
                        }
                        continuation.resume(throwing: error!)
                    }
                }
        }
    })
}
```

Enter one or more of the ACR mapping keys as configured in the OAuth 2.0 provider service.

To learn more, refer to Choose journeys with ACR values.

> **Tip**
>
> You can list the available keys by inspecting the `acr_values_supported` property in the output of your `/oauth2/.well-known/openid-configuration` endpoint.

Step 3. Test the app

Prepare › Download › Configure › **Run**

In this step, run the sample app that you configured in the previous step. The app performs OIDC login to your PingAM server.

1. In Xcode, select **Product > Run**.

   Xcode launches the sample app in the iPhone simulator.

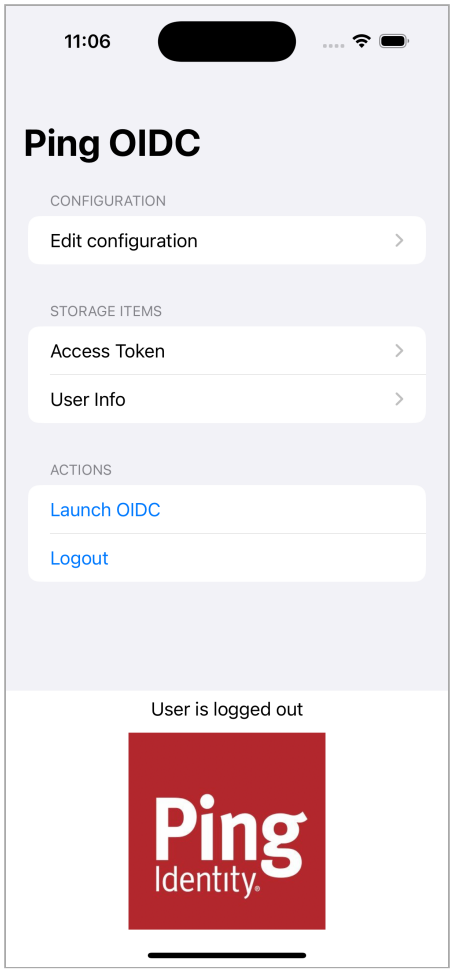


Figure 1. iOS OIDC login sample home screen

2. In the sample app on the iPhone simulator, tap **Edit configuration**, and verify or edit the configuration you entered in the previous step.

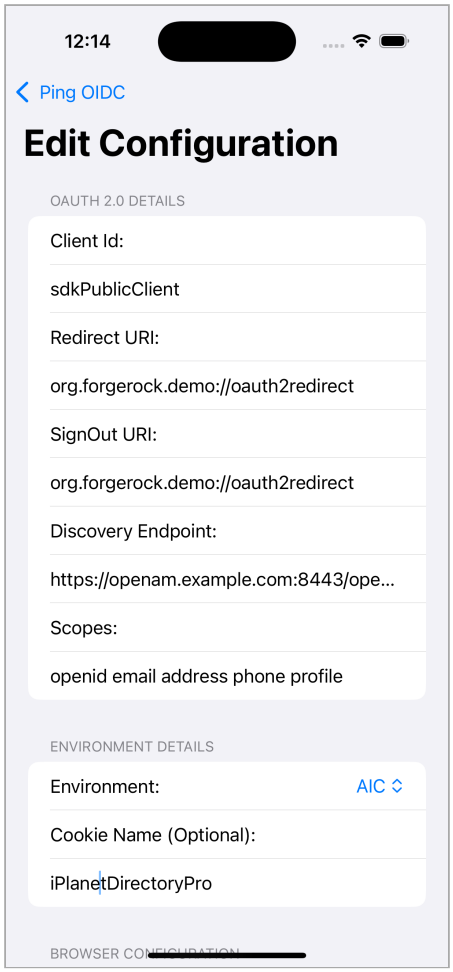


Figure 2. Verify the configuration settings

3. Tap < **Ping OIDC** to go back to the main menu, and then tap **Launch OIDC**.

> **Note**
>
> You might see a dialog asking if you want to open a browser. If you do, tap **Continue**.

The app launches a web browser and redirects to your PingAM login UI:

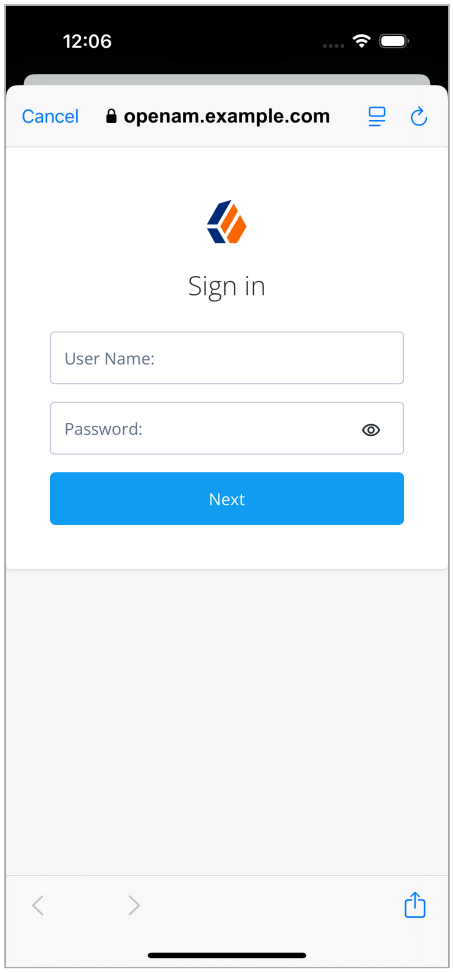


Figure 3. Browser launched and redirected to PingAM

4. Sign on as a demo user:
    - **Name**: `demo`
    - **Password**: `Ch4ng3it!`

If authentication is successful, the application displays the access token issued by PingAM.

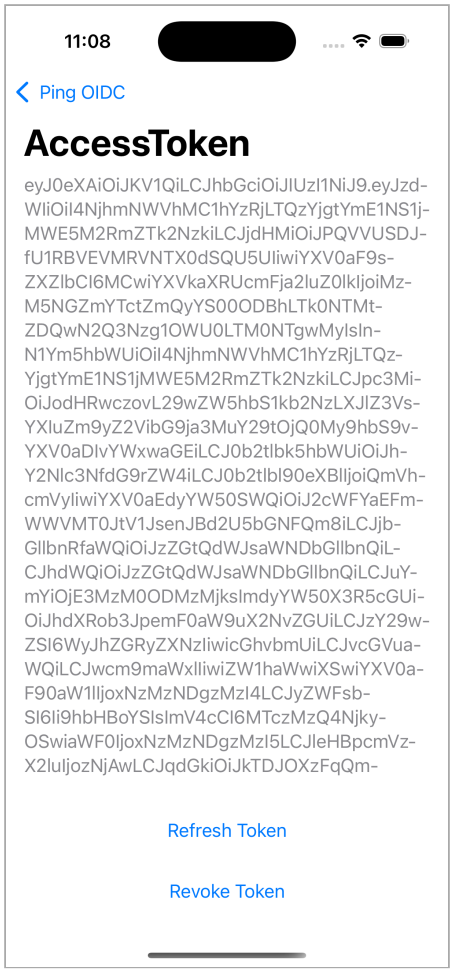


Figure 4. Access token after successful authentication

5. Tap < **Ping OIDC** to go back to the main menu, and then tap **User Info**.

   The app displays the information relating to the access token:

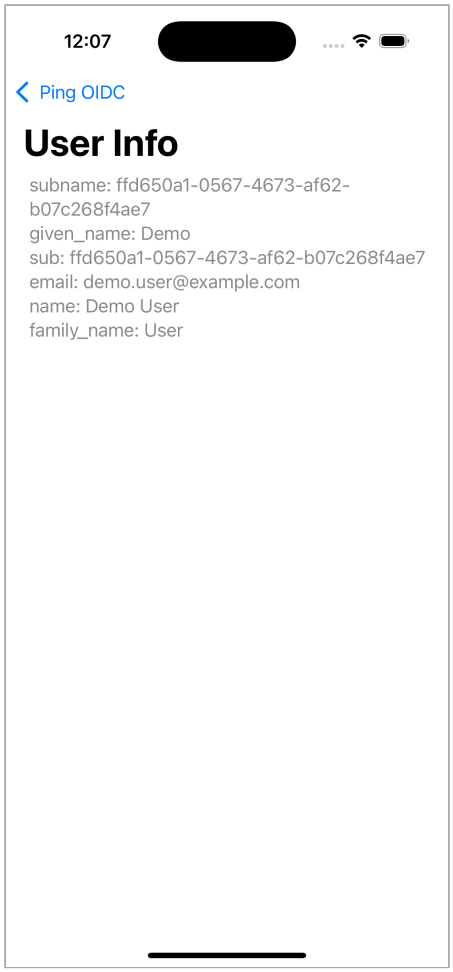


Figure 5. User info relating to the access token

6. Tap < **Ping OIDC** to go back to the main menu, and then tap **Logout**.

   The app logs the user out of the authorization server and prints a message to the Xcode console:

```
[FRCore][4.8.0] [🌐 - Network] Response | [☑ 204] :
  https://openam.example.com:443/am/oauth2/connect/endSession?
id_token_hint=eyJ0...sbrA&client_id=sdkPublicClient in 34 ms
[FRAuth][4.8.0] [FRUser.swift:211 : logout()] [Verbose]
  Invalidating OIDC Session successful
```

Authentication journey tutorial for iOS

Prepare › Download › Configure › Run

In this tutorial you update a sample app that uses OIDC-based login to obtain tokens by redirecting to the PingFederate UI for authentication.

The sample connects to the `.well-known` endpoint of your PingFederate server to obtain the correct URIs to authenticate the user, and redirects to your PingFederate server's login UI.

After authentication, PingFederate redirects the browser back to your application, which then obtains an OAuth 2.0 access token and displays the related user information.

Before you begin

Before you begin this tutorial ensure you have set up your PingFederate server with the required configuration.

For example, you will need to configure an OAuth 2.0 client application.

Complete prerequisites »

Step 1. Download the samples

To start this tutorial, you need to download the SDK sample apps repo, which contains the projects you will use.

Start step 1 »

Step 2. Configure connection properties

In this step, you configure the sample app to connect to the OAuth 2.0 application you created in PingFederate.

Start step 2 »

Step 3. Test the app

In the following procedure, you run the sample app that you configured in the previous step.

The sample connects to your PingFederate server to obtain the correct URIs to authenticate the user, and redirects the browser to your PingFederate server.

After authentication, PingFederate redirects the browser back to your application, which then obtains an OAuth 2.0 access token and displays the related user information.

Test app »

Before you begin

Prepare › Download › Configure › Run

To successfully complete this tutorial refer to the prerequisites and compatibility requirements in this section.

The tutorial also requires a configured PingFederate server.

Compatibility

iOS

This sample app is compatible with iOS 12 and later.

Prerequisites

Xcode

You can download the latest version for free from https://developer.apple.com/xcode/.

Server configuration

This tutorial requires you to configure your PingFederate server as follows:

OAuth 2.0 client application profiles define how applications connect to PingFederate and obtain OAuth 2.0 tokens.

To allow the Ping SDKs to connect to PingFederate and obtain OAuth 2.0 tokens, you must register an OAuth 2.0 client application:

1. Log in to the PingFederate administration console as an administrator.
2. Navigate to **Applications** > **OAuth** > **Clients**.
3. Click **Add Client**.

   PingFederate displays the **Clients** | **Client** page.
4. In **Client ID** and **Name**, enter a name for the profile, for example `sdkPublicClient`

   Make a note of the **Client ID** value, you will need it when you configure the sample code.
5. In **Client Authentication**, select `None`.
6. In **Redirect URIs**, add the following values:

   `org.forgerock.demo://oauth2redirect`

> **Important**
>
> Also add any other URLs where you host SDK applications.
> Failure to add redirect URLs that exactly match your client app's values can cause PingFederate to display an error message such as `Redirect URI mismatch` when attempting to end a session by redirecting from the SDK.

7. In **Allowed Grant Types**, select the following values:

   `Authorization Code`

   `Refresh Token`

8. In the **OpenID Connect** section:

   1. In **Logout Mode**, select **Ping Front-Channel**
   2. In **Front-Channel Logout URIs**, add the following values:

      `org.forgerock.demo://oauth2redirect`

      > **Important**
      >
      > Also add any other URLs that redirect users to PingFederate to end their session.
      > Failure to add sign off URLs that exactly match your client app's values can cause PingFederate to display an error message such as `invalid post logout redirect URI` when attempting to end a session by redirecting from the SDK.

   3. In **Post-Logout Redirect URIs**, add the following values:

      `org.forgerock.demo://oauth2redirect`

9. Click **Save**.

> **Important**
>
> After changing PingFederate configuration using the administration console, you must replicate the changes to each server node in the cluster before they take effect.
> In the PingFederate administration console, navigate to **System > Server > Cluster Management**, and click **Replicate**.

The application is now configured to accept client connections from and issue OAuth 2.0 tokens to the Ping SDK PingFederate example applications and tutorials covered by this documentation.

Cross-origin resource sharing (CORS) lets user agents make cross-domain server requests. In PingFederate, you can configure CORS to allow browsers or apps from trusted domains to access protected resources.

To configure CORS in PingFederate follow these steps:

1. Log in to the PingFederate administration console as an administrator.
2. Navigate to **System > OAuth Settings > Authorization Server Settings**.
3. In the **Cross-Origin Resource Sharing Settings** section, in the **Allowed Origin** field, enter any DNS aliases you use for your SDK apps.

This documentation assumes the following configuration:

Property	Values
`Allowed Origin`	`org.forgerock.demo://oauth2redirect`

4. Click **Save**.

> **Important**
>
> After changing PingFederate configuration using the administration console, you must replicate the changes to each server node in the cluster before they take effect.
> In the PingFederate administration console, navigate to **System > Server > Cluster Management**, and click **Replicate**.

Your PingFederate server is now able to accept connections from origins hosting apps built with the Ping SDKs.

Step 1. Download the samples

Prepare › **Download** › Configure › Run

To start this tutorial, you need to download the ForgeRock SDK sample apps repo, which contains the projects you will use.

1. In a web browser, navigate to the SDK Sample Apps repository.
2. Download the source code using one of the following methods:

Download a ZIP file

1. Click **Code**, and then click **Download ZIP**.
2. Extract the contents of the downloaded ZIP file to a suitable location.

Use a Git-compatible tool to clone the repo locally

1. Click **Code**, and then copy the HTTPS URL.
2. Use the URL to clone the repository to a suitable location.

   For example, from the command-line you could run:

```
git clone https://github.com/ForgeRock/sdk-sample-apps.git
```

The result of these steps is a local folder named `sdk-sample-apps`.

Step 2. Configure connection properties

Prepare › Download › **Configure** › Run

In this step, you configure the "swiftui-oidc" app to connect to the OAuth 2.0 application you created in PingFederate, and display the login UI of the server.

1. In Xcode, on the **File** menu, click **Open**.
2. Navigate to the `sdk-sample-apps` folder you cloned in the previous step, navigate to `iOS` > `swiftui-oidc` > `PingExample` > `PingExample.xcodeproj`, and then click **Open**.
3. In the **Project Navigator** pane, navigate to **PingExample > PingExample > Utilities**, and open the `ConfigurationManager` file.
4. Locate the `ConfigurationViewModel` function which contains placeholder configuration properties.

   Tip

   The function is commented with `//TODO:` in the source to make it easier to locate.

   ```
   return ConfigurationViewModel(
       clientId: "[CLIENT ID]",
       scopes: ["openid", "email", "address", "phone", "profile"],
       redirectUri: "[REDIRECT URI]",
       signOutUri: "[SIGN OUT URI]",
       discoveryEndpoint: "[DISCOVERY ENDPOINT URL]",
       environment: "[ENVIRONMENT - EITHER AIC OR PingOne]",
       cookieName: "[COOKIE NAME - OPTIONAL (Applicable for AIC only)]",
       browserSeletorType: .authSession
   )
   ```

5. In the `ConfigurationViewModel` function, update the following properties with the values you obtained when preparing your environment.

   clientId

   The client ID from your OAuth 2.0 application in PingFederate.

   For example, `sdkPublicClient`

   scopes

   The scopes you want to assign in PingFederate.

   For example, `openid profile email phone`

   redirectUri

   The **Redirect URIs** as configured in the OAuth 2.0 client profile.

   This value must exactly match a value configured in your OAuth 2.0 client.

For example, `org.forgerock.demo://oauth2redirect`

signOutUri

The **Front-Channel Logout URIs** as configured in the OAuth 2.0 client profile.

This value must exactly match a value configured in your OAuth 2.0 client.

For example, `org.forgerock.demo://oauth2redirect`

discoveryEndpoint

The `.well-known` endpoint from your PingFederate tenant.

To form the `.well-known` endpoint for a PingFederate server:

1. Log in to your PingFederate administration console.
2. Navigate to **System** > **Server** > **Protocol Settings**.
3. Make a note of the **Base URL** value.

   For example, `https://pingfed.example.com`

   > **Note**
   >
   > Do not use the admin console URL.

4. Append `/.well-known/openid-configuration` after the base URL value to form the `.well-known` endpoint of your server.

   For example, `https://pingfed.example.com/.well-known/openid-configuration`.

   The SDK reads the OAuth 2.0 paths it requires from this endpoint.

For example, `https://pingfed.example.com/.well-known/openid-configuration`

environment

Ensures the sample app uses the correct behavior for the different servers it supports, for example what logout parameters to use.

For PingFederate specify `PingOne`.

cookieName

Set this property to an empty string.

For example, `""`.

browserSeletorType

You can specify what type of browser the client iOS device opens to handle centralized login.

Each browser has slightly different characteristics, which make them suitable to different scenarios, as outlined in this table:

Browser type	Characteristics
`.authSession`	Opens a web authentication session browser. Designed specifically for authentication sessions, however it prompts the user before opening the browser with a modal that asks them to confirm the domain is allowed to authenticate them. This is the default option in the Ping SDK for iOS.
`.ephemeralAuthSession`	Opens a web authentication session browser, but enables the `prefersEphemeralWebBrowserSession` parameter. This browser type *does not* prompt the user before opening the browser with a modal. The difference between this and `.authSession` is that the browser does not include any existing data such as cookies in the request, and also discards any data obtained during the browser session, including any session tokens. When is `ephemeralAuthSession` suitable: ◦ ✕ `ephemeralAuthSession` is *not* suitable when you require single sign-on (SSO) between your iOS apps, as the browser will not maintain session tokens. ◦ ✕ `ephemeralAuthSession` is *not* suitable when you require a session token to log a user out of the server, for example for logging out of PingOne, as the browser will not maintain session tokens. ◦ ✓ Use `ephemeralAuthSession` when you do not want the user's existing sessions to affect the authentication.
`.nativeBrowserApp`	Opens the installed browser that is marked as the default by the user. Often Safari. The browser opens without any interaction from the user. However, the browser does display a modal when returning to your application.
`.sfViewController`	Opens a Safari view controller browser. Your client app is ***not*** able to interact with the pages in the `sfViewController` or access the data or browsing history. The view controller opens within your app without any interaction from the user. As the user does not leave your app, the view controller does not need to display a warning modal when authentication is complete and control returns to your application.

The result resembles the following:

```
return ConfigurationViewModel(
    clientId: "sdkPublicClient",
    scopes: ["openid", "email", "phone", "profile"],
    redirectUri: "org.forgerock.demo://oauth2redirect",
    signOutUri: "org.forgerock.demo://oauth2redirect",
    discoveryEndpoint: "https://pingfed.example.com/.well-known/openid-configuration",
    environment: "PingOne",
    cookieName: "",
    browserSeletorType: .authSession
)
```

With the sample configured, you can proceed to Step 3. Test the app.

Step 3. Test the app

Prepare › Download › Configure › **Run**

In this step, run the sample app that you configured in the previous step. The app performs OIDC login to your PingFederate instance.

1. In Xcode, select **Product** > **Run**.

   Xcode launches the sample app in the iPhone simulator.

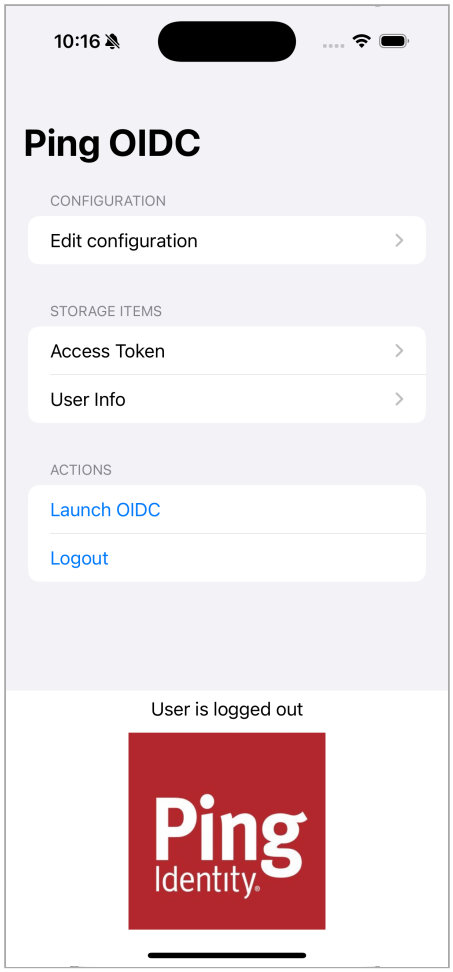


Figure 1. iOS OIDC login sample home screen

2. In the sample app on the iPhone simulator, tap **Edit configuration**, and verify or edit the configuration you entered in the previous step.

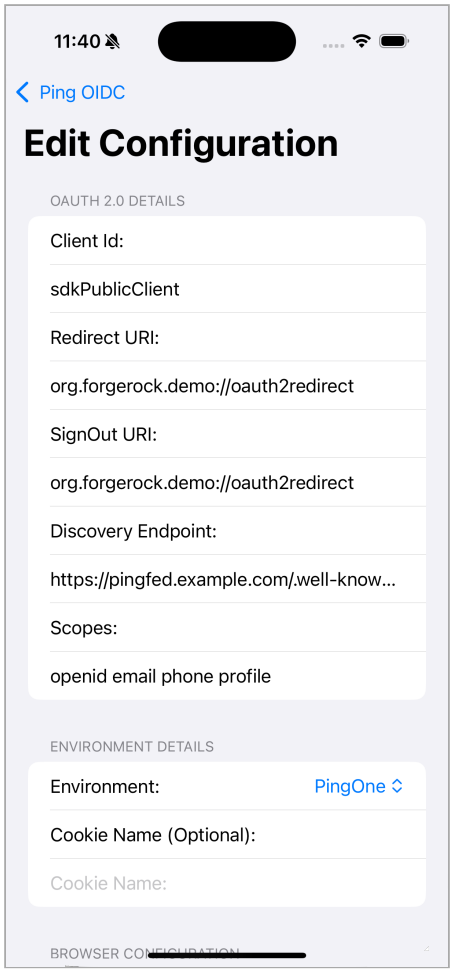


Figure 2. Verify the configuration settings

3. Tap < **Ping OIDC** to go back to the main menu, and then tap **Launch OIDC**.

> **Note**
>
> You might see a dialog asking if you want to open a browser. If you do, tap **Continue**.

The app launches a web browser and redirects to your PingFederate login UI:

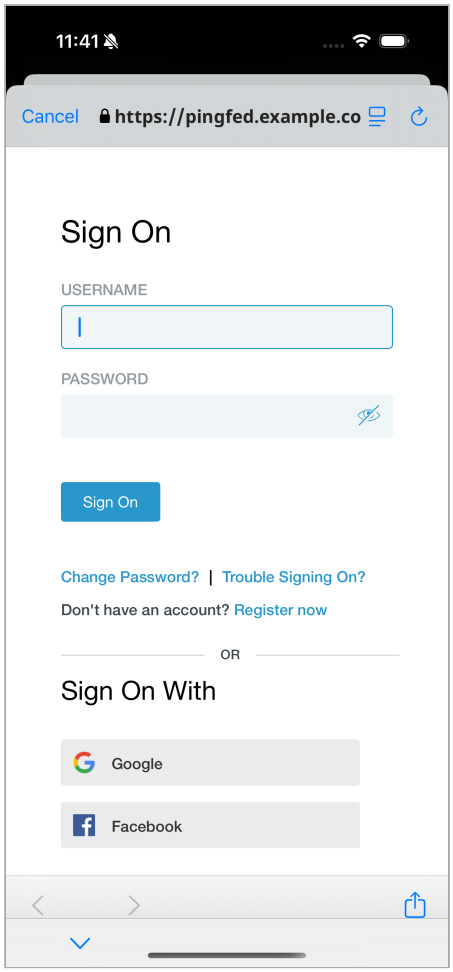


Figure 3. Browser launched and redirected to PingFederate

4. Sign on as a demo user:
    - **Name**: `demo`
    - **Password**: `Ch4ng3it!`

If authentication is successful, the application displays the access token issued by PingFederate.

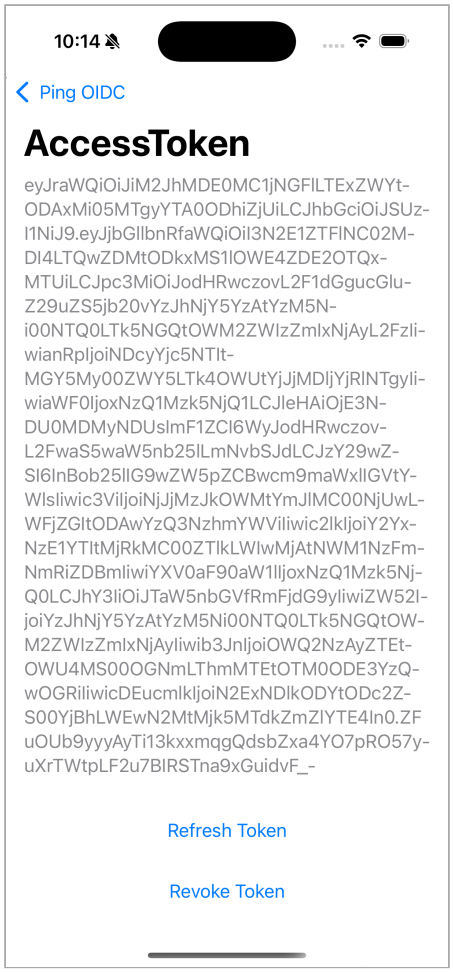


Figure 4. Access token after successful authentication

5. Tap < **Ping OIDC** to go back to the main menu, and then tap **User Info**. The app displays the user information relating to the access token:

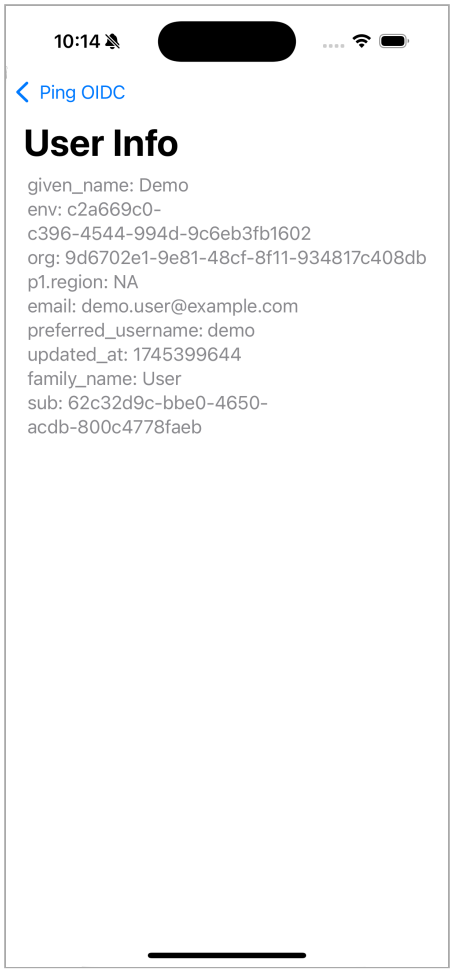


Figure 5. User info relating to the access token

6. Tap < **Ping OIDC** to go back to the main menu, and then tap **Logout**.

   The app briefly opens a browser to sign the user out of PingFederate, and revoke the tokens.

JavaScript OIDC login tutorials

Follow these JavaScript tutorials to integrate your apps using OpenID Connect login to the following servers:

PingOne

PingOne

PingOne Advanced Identity Cloud

PingOne Advanced Identity Cloud

PingAM

PingAM

PingFederate

PingFederate

OIDC login to PingOne tutorial for JavaScript

Prepare › Download › Install › Configure › Run

In this tutorial you update a sample app that uses OIDC-based login to obtain tokens by redirecting to the PingOne UI for authentication.

The sample connects to the `.well-known` endpoint of your PingOne server to obtain the correct URIs to authenticate the user, and redirects to your PingOne server's login UI.

After authentication, PingOne redirects the browser back to your application, which then obtains an OAuth 2.0 access token and displays the related user information.

Before you begin

Before you begin this tutorial ensure you have set up your PingOne server with the required configuration.

For example, you will need to have an OAuth 2.0 client application set up, and a demo user to authenticate.

Complete prerequisites »

Step 1. Download the samples

To start this tutorial, you need to download the SDK sample apps repo, which contains the projects you will use.

Start step 1 »

Step 2. Install the Ping SDK

The sample projects need a number of dependencies that you can install by using the `npm` command.

For example, the Ping SDK for JavaScript itself is one of the dependencies.

Start step 2 »

Step 3. Configure connection properties

In this step, you configure the sample app to connect to the OAuth 2.0 application you created in PingOne.

Start step 3 »

Step 4. Test the app

To test the app, run the sample that you configured in the previous step.

The sample connects to your PingOne server to obtain the correct URIs to authenticate the user, and redirects the browser to your PingOne server.

After authentication, PingOne redirects the browser back to your application, which then obtains an OAuth 2.0 access token and displays the related user information.

Test app »

Before you begin

Prepare › Download › Install › Configure › Run

To successfully complete this tutorial refer to the prerequisites in this section.

The tutorial also requires a configured PingOne instance.

Prerequisites

Node and NPM

This sample requires a minimum Node.js version of `18`, and is tested on versions `18` and `20`. To get a supported version of Node.js, refer to the Node.js download page.

You will also need `npm` to build the code and run the samples.

Server configuration

This tutorial requires you to configure your PingOne server as follows:

The samples and tutorials in this documentation often require that you have an identity set up so that you can test authentication.

To create a demo user in PingOne, follow these steps:

1. Log in to your PingOne administration console.
2. In the left panel, navigate to **Directory** > **Users**.
3. Next to the **Users** label, click the plus icon (+).

   PingOne displays the **Add User** panel.
4. Enter the following details:
   - **Given Name** = `Demo`
   - **Family Name** = `User`
   - **Username** = `demo`
   - **Email** = `demo.user@example.com`
   - **Population** = `Default`
   - **Password** = `Ch4ng3it!`
5. Click **Save**.

To register a *public* OAuth 2.0 client application in PingOne for use with the Ping SDK for JavaScript, follow these steps:

1. Log in to your PingOne administration console.
2. In the left panel, navigate to **Applications** > **Applications**.
3. Next to the **Applications** label, click the plus icon (+).

   PingOne displays the **Add Application** panel.
4. In **Application Name**, enter a name for the profile, for example `sdkPublicClient`
5. Select **OIDC Web App** as the **Application Type**, and then click **Save**.

6. On the **Configuration** tab, click the pencil icon (✎).

    1. In **Grant Type**, select the following values:

        `Authorization Code`

        `Refresh Token`

    2. In **Redirect URIs**, enter the following value:

        `https://localhost:8443`

    > **Important**
    >
    > Also add any other URLs where you host SDK applications.
    > Failure to add redirect URLs that exactly match your client app's values can cause PingOne to display an error message such as `Redirect URI mismatch` when attempting to end a session by redirecting from the SDK.

    1. In **Token Endpoint Authentication Method**, select `None`.

    2. In **Signoff URLs**, enter the following value:

        `https://localhost:8443`

        > **Important**
        >
        > Also add any other URLs that redirect users to PingOne to end their session.
        > Failure to add sign off URLs that exactly match your client app's values can cause PingOne to display an error message such as `invalid post logout redirect URI` when attempting to end a session by redirecting from the SDK.

    3. In **CORS Settings**, in the drop-down select **Allow specific origins**, and in the **Allowed Origins** field, enter the URL where you will be running the sample app.

        For example:

        `https://localhost:8443`

    4. In the **Advanced Settings** section, enable **Terminate User Session by ID Token**.

    5. Click **Save**.

7. On the **Resources** tab, next to **Allowed Scopes**, click the pencil icon (✎).

    1. In **Scopes**, select the following values:

        `email`

        `phone`

        `profile`

        > **Note**
        >
        > The `openid` scope is selected by default.

The result resembles the following:

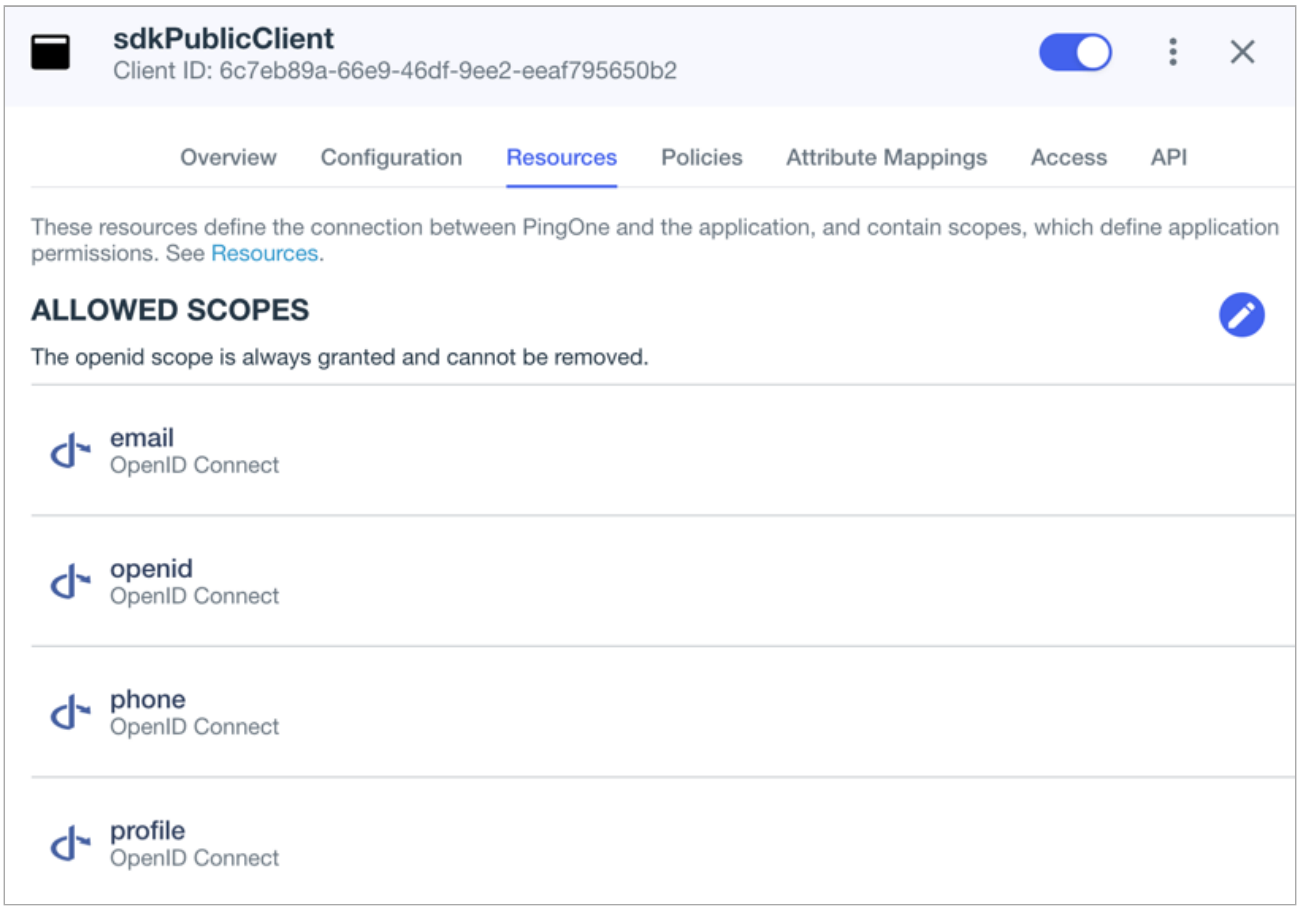


Figure 1. Adding scopes to an application.

8. Optionally, on the **Policies** tab, click the pencil icon (✎) to select the authentication policies for the application.

> **Note**
>
> Applications that have no authentication policy assignments use the environment's default authentication policy to authenticate users.

If you have a DaVinci license, you can select PingOne policies or DaVinci Flow policies, but not both. If you do not have a DaVinci license, the page only displays PingOne policies.

To use a *PingOne policy*:

1. Click **+ Add policies** and then select the policies that you want to apply to the application.
2. Click **Save**.

   PingOne applies the policies in the order in which they appear in the list. PingOne evaluates the first policy in the list first. If the requirements are not met, PingOne moves to the next one.

   For more information, see Authentication policies for applications.

To use a *DaVinci Flow policy*:

1. You must clear all PingOne policies. Click **Deselect all PingOne Policies**.

2. In the confirmation message, click **Continue**.
3. On the **DaVinci Policies** tab, select the policies that you want to apply to the application.
4. Click **Save**.

   PingOne applies the first policy in the list.

9. Click **Save**.
10. Enable the OAuth 2.0 client application by using the toggle next to its name:

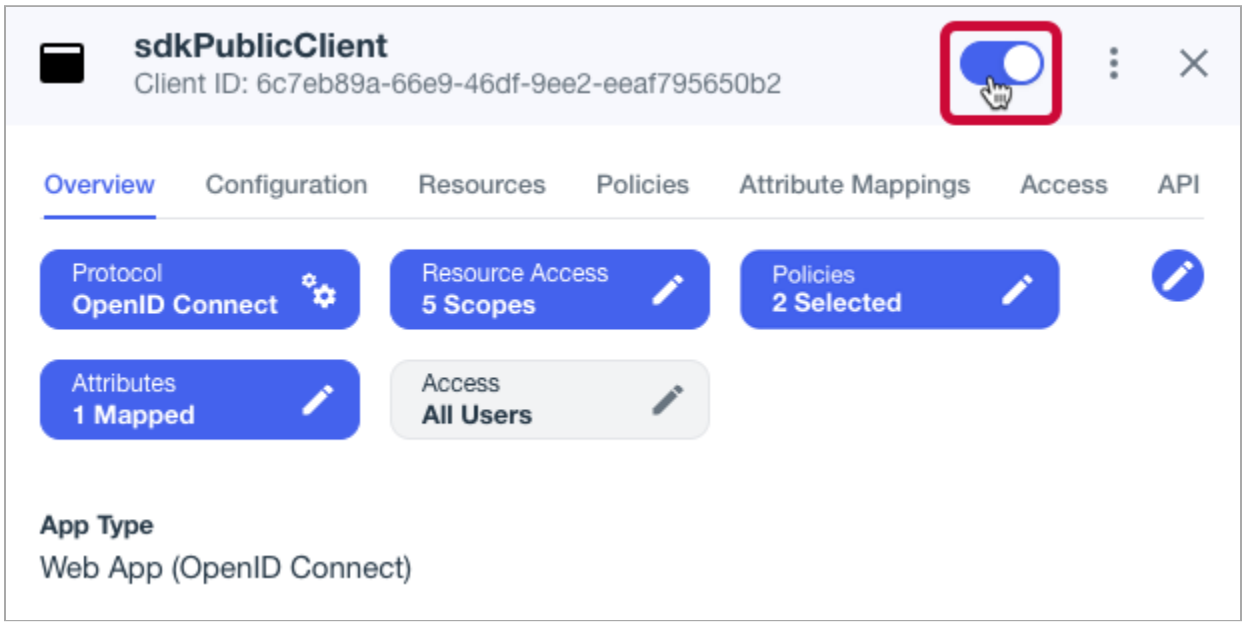


Figure 2. Enable the application using the toggle.

The application is now configured to accept client connections from and issue OAuth 2.0 tokens to the JavaScript example PingOne applications and tutorials covered by this documentation.

Step 1. Download the samples

Prepare › **Download** › Install › Configure › Run

To start this tutorial, you need to download the Ping SDK sample apps repo, which contains the projects you will use.

1. In a web browser, navigate to the Ping SDK sample apps repository.
2. Download the source code using one of the following methods:

Download a ZIP file

1. Click **Code**, and then click **Download ZIP**.
2. Extract the contents of the downloaded ZIP file to a suitable location.

Use a Git-compatible tool to clone the repo locally

1. Click **Code**, and then copy the HTTPS URL.
2. Use the URL to clone the repository to a suitable location.

   For example, from the command-line you could run:

   ```
   git clone https://github.com/ForgeRock/sdk-sample-apps.git
   ```

The result of these steps is a local folder named `sdk-sample-apps`.

Step 2. Install the Ping SDK

Prepare › Download › **Install** › Configure › Run

In the following procedure, you install the Ping SDK for JavaScript.

1. In a terminal window, navigate to the `sdk-sample-apps` folder.
2. To install the required packages, enter the following:

   ```
   npm install
   ```

   The `npm` tool downloads the required packages, and places them inside a `node_modules` folder.

More information

- Cloning a repository | GitHub Help
- Get npm | npm

Step 3. Configure connection properties

Prepare › Download › Install › **Configure** › Run

In this step, you configure the sample app to connect to the OAuth 2.0 application you created in PingOne.

1. In the IDE of your choice, open the `sdk-sample-apps` folder you cloned in the previous step.
2. Open the `/javascript/central-login-oidc/src/main.js` file.
3. Locate the `forgerock.Config.setAsync()` method and update the properties to match your PingOne environment:

```
await forgerock.Config.setAsync({
  clientId: process.env.WEB_OAUTH_CLIENT, // e.g. 'ForgeRockSDKClient' or PingOne Services Client GUID
  redirectUri: `${window.location.origin}`, // Redirect back to your app, e.g. 'https://localhost:8443' or the
domain your app is served.
  scope: process.env.SCOPE, // e.g. 'openid profile email address phone revoke' When using PingOne services
revoke scope is required
  serverConfig: {
    wellknown: process.env.WELL_KNOWN,
    timeout: process.env.TIMEOUT, // Any value between 3000 to 5000 is good, this impacts the redirect time to
login. Change that according to your needs.
  },
});
```

Replace the following strings with the values you obtained when you registered an OAuth 2.0 application in PingOne:

process.env.WEB_OAUTH_CLIENT

The client ID from your OAuth 2.0 application in PingOne.

For example, `6c7eb89a-66e9-ab12-cd34-eeaf795650b2`

process.env.SCOPE

The scopes you added to your OAuth 2.0 application in PingOne.

For example, `openid profile email phone revoke`

process.env.WELL_KNOWN

The `.well-known` endpoint from your OAuth 2.0 application in PingOne.

To find the `.well-known` endpoint for an OAuth 2.0 client in PingOne:

1. Log in to your PingOne administration console.
2. Go to **Applications > Applications**, and then select the OAuth 2.0 client you created earlier.

   For example, sdkPublicClient.
3. On the **Configuration** tab, expand the **URLs** section, and then copy the **OIDC Discovery Endpoint** value.

For example, `https://auth.pingone.com/3072206d-c6ce-ch15-m0nd-f87e972c7cc3/as/.well-known/openid-configuration`

process.env.TIMEOUT

Enter how many milliseconds to wait before timing out the OAuth 2.0 flow.

For example, `3000`

The result resembles the following:

```
await forgerock.Config.setAsync({
  clientId: "6c7eb89a-66e9-ab12-cd34-eeaf795650b2",
  redirectUri: `${window.location.origin}`,
  scope: "openid profile email phone revoke",
  serverConfig: {
    wellknown: "https://auth.pingone.com/3072206d-c6ce-ch15-m0nd-f87e972c7cc3/as/.well-known/openid-
configuration",
    timeout: 3000
  },
});
```

4. Alter the `logout` object as follows:

    1. In the `forgerock.FRUser.logout()` method, add the following parameter:

       `{logoutRedirectUri: `${window.location.origin}`}`

       The presence of this parameter causes the SDK to use a redirect flow for ending the session and revoking the tokens, which PingOne servers require.

    2. Remove or comment out the following line:

       `location.assign(`${document.location.origin}/`);`

    The result resembles the following:

```
const logout = async () => {
  try {
    await FRUser.logout({
      logoutRedirectUri: `${window.location.origin}`
    });
    // location.assign(`${document.location.origin}/`);
  } catch (error) {
    console.error(error);
  }
};
```

5. Optionally, specify which of the configured policies PingOne uses to authenticate users.

    In the `/javascript/central-login-oidc/src/main.js` file, find each instance of the `getTokens` method that has a `login: 'redirect'` parameter and add an additional `acr_values` query parameter:

```
await TokenManager.getTokens({
  login: 'redirect',
  query: {
    acr_values: "<Policy IDs>"
  }
});
```

    Replace *<Policy IDs>* with either a single DaVinci policy, by using its flow policy ID, or one or more PingOne policies by specifying the policy names, separated by spaces or the encoded space character `%20`.

Examples:

DaVinci flow policy ID

```
acr_values: "d1210a6b0b2665dbaa5b652221badba2"
```

PingOne policy names

```
acr_values: "Single_Factor%20Multi_Factor"
```

For more information, refer to Authentication policies.

Step 4. Test the app

Prepare › Download › Install › Configure › Run

In the following procedure, you run the sample app that you configured in the previous step.

The sample connects to your PingOne server to obtain the correct URIs to authenticate the user, and redirects the browser to your PingOne server.

After authentication, PingOne redirects the browser back to your application, which then obtains an OAuth 2.0 access token and displays the related user information.

Run the sample

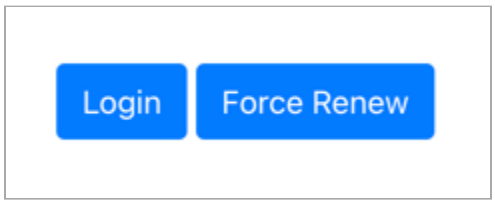
1. In a terminal window, navigate to the `/javascript` folder in your `sdk-sample-apps` project.
2. To run the embedded login sample, enter the following:

   ```
   npm run start:central-login-oidc
   ```

3. In a web browser, navigate to the following URL:

   `https://localhost:8443`

   The sample displays a page with two buttons:

4. Click **Login**.

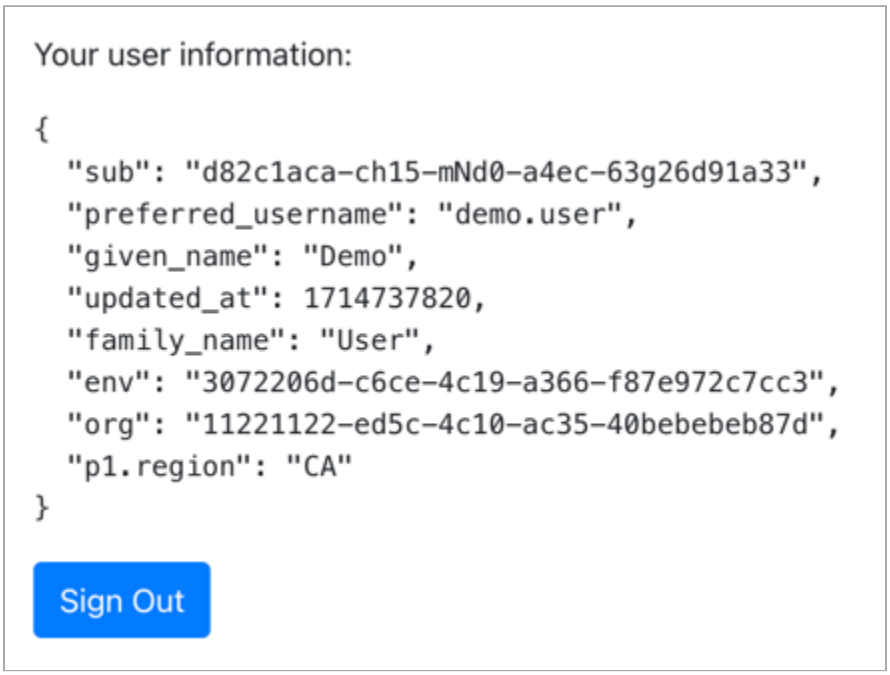
   The sample app redirects the browser to your PingOne instance.

> **Tip**
>
> To see the application calling the `authorize` endpoint, and the redirect back from PingOne with the `code` and `state` OAuth 2.0 parameters, open the **Network** tab of your browser's developer tools.

5. Authenticate as a known user in your PingOne system.

   After successful authentication, PingOne redirects the browser to the client application.

   If the app displays the user information, authentication was successful:



6. To revoke the OAuth 2.0 token, click the **Sign Out** button.

   The application redirects to the PingOne server to revoke the OAuth 2.0 token and end the session, and then returns to the URI specified by the `logoutRedirectUri` parameter of the `logout` method.

   In this tutorial, PingOne redirects users back to the client application, ready to authenticate again.

Recap

Congratulations!

You have now used the Ping SDK for JavaScript to obtain an OAuth 2.0 access token on behalf of a user from your PingOne server.

You have seen how to obtain OAuth 2.0 tokens, and view the related user information.

More information

- API reference: `TokenManager`
- API reference: `UserManager`

OIDC login to PingOne Advanced Identity Cloud tutorial for Android

Prepare › Download › Install › Configure › Run

In this tutorial you update a sample app that uses OIDC-based login to obtain tokens by redirecting to the PingOne Advanced Identity Cloud UI for authentication.

The sample connects to the `.well-known` endpoint of your PingOne Advanced Identity Cloud tenant to obtain the correct URIs to authenticate the user, and redirects to your PingOne Advanced Identity Cloud tenant's login UI.

After authentication, PingOne Advanced Identity Cloud redirects the browser back to your application, which then obtains an OAuth 2.0 access token and displays the related user information.

Before you begin

Before you begin this tutorial ensure you have set up your PingOne Advanced Identity Cloud tenant with the required configuration.

For example, you will need to have an OAuth 2.0 client application set up, and a demo user to authenticate.

Complete prerequisites »

Step 1. Download the samples

To start this tutorial, you need to download the SDK sample apps repo, which contains the projects you will use.

Start step 1 »

Step 2. Install the Ping SDK

The sample projects need a number of dependencies that you can install by using the `npm` command.

For example, the Ping SDK for JavaScript itself is one of the dependencies.

Start step 2 »

Step 3. Configure connection properties

In this step, you configure the sample app to connect to the OAuth 2.0 application you created in PingOne Advanced Identity Cloud.

Start step 3 »

Step 4. Test the app

To test the app, run the sample that you configured in the previous step.

The sample connects to your PingOne Advanced Identity Cloud tenant to obtain the correct URIs to authenticate the user, and redirects the browser to your PingOne Advanced Identity Cloud tenant.

After authentication, PingOne Advanced Identity Cloud redirects the browser back to your application, which then obtains an OAuth 2.0 access token and displays the related user information.

Test app »

Before you begin

Prepare › Download › Install › Configure › Run

To successfully complete this tutorial refer to the prerequisites in this section.

The tutorial also requires a configured PingOne Advanced Identity Cloud tenant.

Node and NPM

This sample requires a minimum Node.js version of `18`, and is tested on versions `18` and `20`. To get a supported version of Node.js, refer to the Node.js download page.

You will also need `npm` to build the code and run the samples.

Server configuration

This tutorial requires you to configure your PingOne Advanced Identity Cloud tenant as follows:

The samples and tutorials in this documentation often require that you have an identity set up so that you can test authentication.

To create a demo user in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.

2. In the left panel, click **Identities** > **Manage**.
3. Click **+ New Alpha realm - User**.
4. Enter the following details:
    - **Username** = `demo`
    - **First Name** = `Demo`
    - **Last Name** = `User`
    - **Email Address** = `demo.user@example.com`
    - **Password** = `Ch4ng3it!`
5. Click **Save**.

Authentication journeys provide fine-grained authentication by allowing multiple paths and decision points throughout the flow. Authentication journeys are made up of nodes that define actions taken during authentication.

Each node performs a single task, such as collecting a username or making a simple decision. Nodes can have multiple outcomes rather than just success or failure. For details, see the Authentication nodes configuration reference in the PingAM documentation.

To create a simple journey for use when testing the Ping SDKs, follow these steps:

1. In your PingOne Advanced Identity Cloud tenant, navigate to **Journeys**, and click **+ New Journey**.
2. Enter a name, such as `sdkUsernamePasswordJourney` and click **Save**.

   The authentication journey designer appears.
3. Drag the following nodes into the designer area:
    - **Page Node**
    - **Platform Username**
    - **Platform Password**
    - **Data Store Decision**
4. Drag and drop the **Platform Username** and **Platform Password** nodes onto the **Page Node**, so that they both appear on the same page when logging in.
5. Connect the nodes as follows:

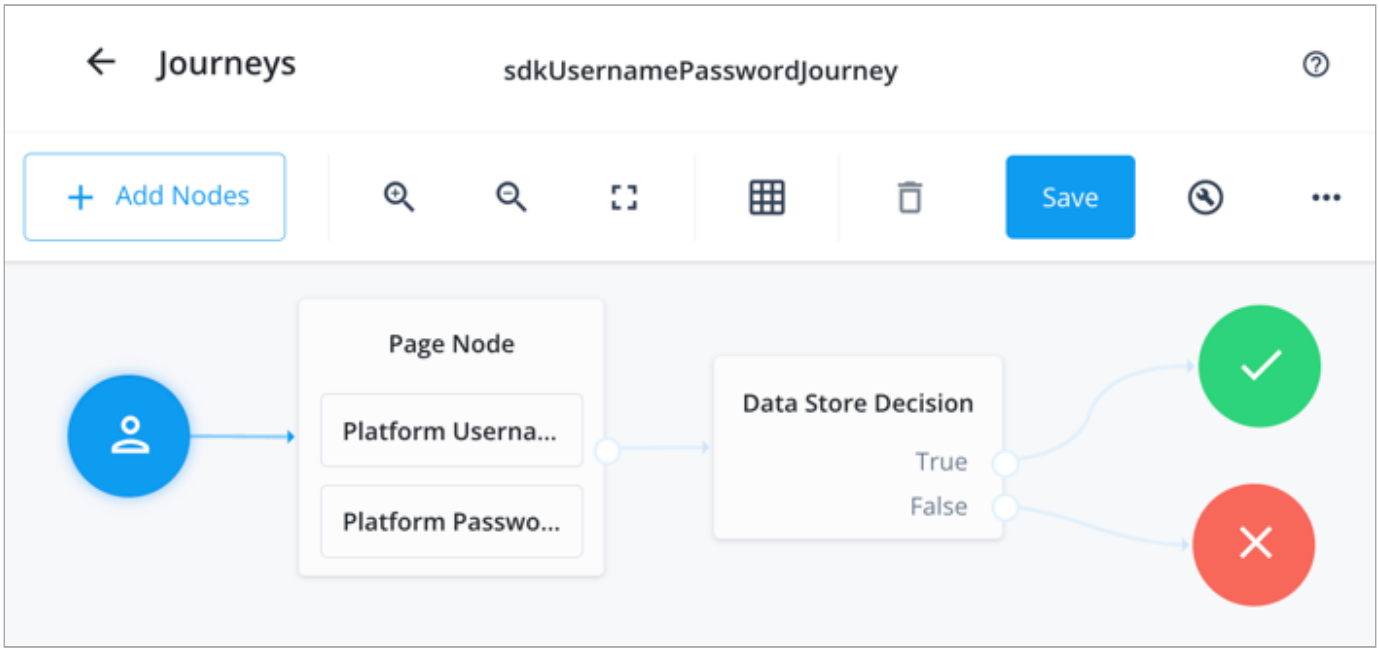


Figure 1. Example username and password authentication journey

6. Click **Save**.

Public clients do not use a client secret to obtain tokens because they are unable to keep them hidden. The Ping SDKs commonly use this type of client to obtain tokens, as they cannot guarantee safekeeping of the client credentials in a browser or on a mobile device.

To register a *public* OAuth 2.0 client application for use with the SDKs in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. In the left panel, click **Applications**.
3. Click **+ Custom Application**.
4. Select **OIDC - OpenId Connect** as the sign-in method, and then click **Next**.
5. Select **Native / SPA** as the application type, and then click **Next**.
6. In **Name**, enter a name for the application, such as `Public SDK Client`.
7. In **Owners**, select a user that is responsible for maintaining the application, and then click **Next**.

> **Tip**
>
> When trying out the SDKs, you could select the `demo` user you created previously.

8. In **Client ID**, enter `sdkPublicClient`, and then click **Create Application**.

   PingOne Advanced Identity Cloud creates the application and displays the details screen.

9. On the **Sign On** tab:

   1. In **Sign-In URLs**, enter the following values:

      `https://localhost:8443/callback.html`

> **Important**
>
> Also add any other domains where you host SDK applications.

2. In **Grant Types**, enter the following values:

   `Authorization Code`

   `Refresh Token`

3. In **Scopes**, enter the following values:

   `openid profile email address`

10. Click Show advanced settings, and on the **Authentication** tab:

    1. In **Token Endpoint Authentication Method**, select `none`.
    2. In **Client Type**, select `Public`.
    3. Enable the **Implied Consent** property.

11. Click **Save**.

The application is now configured to accept client connections from and issue OAuth 2.0 tokens to the example applications and tutorials covered by this documentation.

The provider specifies the supported OAuth 2.0 configuration options for a realm.

To ensure the PingOne Advanced Identity Cloud OAuth 2.0 provider service is configured for use with the Ping SDKs, follow these steps:

1. In your PingOne Advanced Identity Cloud tenant, navigate to **Native Consoles** > **Access Management**.
2. In the left panel, click **Services**.
3. In the list of services, click **OAuth2 Provider**.
4. On the **Core** tab, ensure **Issue Refresh Tokens** is enabled.
5. On the **Consent** tab, ensure **Allow Clients to Skip Consent** is enabled.
6. Click **Save Changes**.

Cross-origin resource sharing (CORS) lets user agents make cross-domain server requests. In PingOne Advanced Identity Cloud, you can configure CORS to allow browsers from trusted domains to access PingOne Advanced Identity Cloud protected resources. For example, you might want a custom web application running on your own domain to get an end-user's profile information using the PingOne Advanced Identity Cloud REST API.

The Ping SDK for JavaScript samples and tutorials use `https://localhost:8443` as the host domain, which you should add to your CORS configuration.

If you are using a different domain for hosting SDK applications, ensure you add them to the CORS configuration as accepted origin domains.

To update the CORS configuration in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.

2. At the top right of the screen, click your name, and then select **Tenant settings**.
3. On the **Global Settings** tab, click **Cross-Origin Resource Sharing (CORS)**.
4. Perform one of the following actions:
    - If available, click **ForgeRockSDK**.
    - If you haven't added any CORS configurations to the tenant, click **+ Add a CORS Configuration**, select **Ping SDK**, and then click **Next**.
5. Add `https://localhost:8443` and any DNS aliases you use to host your Ping SDK for JavaScript applications to the **Accepted Origins** property.
6. Complete the remaining fields to suit your environment.

   This documentation assumes the following configuration, required for the tutorials and sample applications:

Property	Values
`Accepted Origins`	`https://localhost:8443`
`Accepted Methods`	`GET` `POST`

`Accepted Headers`	`accept-api-version`
	`x-requested-with`
	`content-type`
	`authorization`
	`if-match`
	`x-requested-platform`
	`iPlanetDirectoryPro` [1]
	`ch15fefc5407912` [2]

`Exposed Headers`	`authorization` `content-type`
`Enable Caching`	`True`
`Max Age`	`600`
`Allow Credentials`	`True`

> **Tip**
>
> Click **Show advanced settings** to be able to edit all available fields.

7. Click **Save CORS Configuration**.

1. Cookie name value in PingAM servers.
2. In PingOne Advanced Identity Cloud tenants, go to **Tenant Settings > Global Settings > Cookie** to find this dynamic cookie name value.

Step 1. Download the samples

Prepare › **Download** › Install › Configure › Run

To start this tutorial, you need to download the Ping SDK sample apps repo, which contains the projects you will use.

1. In a web browser, navigate to the Ping SDK sample apps repository.
2. Download the source code using one of the following methods:

Download a ZIP file

1. Click **Code**, and then click **Download ZIP**.
2. Extract the contents of the downloaded ZIP file to a suitable location.

Use a Git-compatible tool to clone the repo locally

1. Click **Code**, and then copy the HTTPS URL.
2. Use the URL to clone the repository to a suitable location.

   For example, from the command-line you could run:

```
git clone https://github.com/ForgeRock/sdk-sample-apps.git
```

The result of these steps is a local folder named `sdk-sample-apps`.

Step 2. Install the Ping SDK

Prepare › Download › **Install** › Configure › Run

In the following procedure, you install the Ping SDK for JavaScript.

1. In a terminal window, navigate to the `sdk-sample-apps` folder.
2. To install the required packages, enter the following:

```
npm install
```

   The `npm` tool downloads the required packages, and places them inside a `node_modules` folder.

More information

- API reference: `Ping SDK for JavaScript`

- Cloning a repository | GitHub Help
- Get npm | npm

Step 3. Configure connection properties

Prepare › Download › Install › **Configure** › Run

In this step, you configure the sample app to connect to the OAuth 2.0 application you created in PingOne Advanced Identity Cloud.

1. In the IDE of your choice, open the `sdk-sample-apps` folder you cloned in the previous step.
2. Make a copy of the `/javascript/central-login-oidc/.env.example` file, and name it `.env`.

   The `.env` file provides the values used by the `forgerock.Config.setAsync()` method in `javascript/central-login-oidc/src/main.js`.
3. Update the `.env` file with the details of your PingOne Advanced Identity Cloud instance.

   ```
   SCOPE="$SCOPE"
   TIMEOUT=$TIMEOUT
   WEB_OAUTH_CLIENT="$WEB_OAUTH_CLIENT"
   WELL_KNOWN="$WELL_KNOWN"
   SERVER_TYPE="$SERVER_TYPE"
   ```

   Replace the following strings with the values you obtained when preparing your environment.

   $SCOPE

   The scopes you added to your OAuth 2.0 application in PingOne Advanced Identity Cloud.

   For example, `address email openid phone profile`

   $TIMEOUT

   How long to wait for OAuth 2.0 timeouts, in milliseconds.

   For example, `3000`

   $WEB_OAUTH_CLIENT

   The client ID from your OAuth 2.0 application in PingOne Advanced Identity Cloud.

   For example, `sdkPublicClient`

   $WELL_KNOWN

   The `.well-known` endpoint from your PingOne Advanced Identity Cloud tenant.

You can view the `.well-known` endpoint for an OAuth 2.0 client in the PingOne Advanced Identity Cloud admin console:

1. Log in to your PingOne Advanced Identity Cloud administration console.
2. Click **Applications**, and then select the OAuth 2.0 client you created earlier. For example, sdkPublicClient.
3. On the **Sign On** tab, in the **Client Credentials** section, copy the **Discovery URI** value.

For example, `https://openam-forgerock-sdks.forgeblocks.com/am/oauth2/alpha/.well-known/openid-configuration`

$SERVER_TYPE

Ensures the sample app uses the correct behavior for the different servers it supports, for example what logout parameters to use.

For PingOne Advanced Identity Cloud and PingAM servers, specify `AIC`.

The result resembles the following:

.env

```
SCOPE="address email openid phone profile"
TIMEOUT=3000
WEB_OAUTH_CLIENT="sdkPublicClient"
WELL_KNOWN="https://openam-forgerock-sdks.forgeblocks.com/am/oauth2/alpha/.well-known/openid-configuration"
SERVER_TYPE="AIC"
```

Step 4. Test the app

Prepare › Download › Install › Configure › **Run**

In the following procedure, you run the sample app that you configured in the previous step.

The sample connects to your PingOne Advanced Identity Cloud tenant to obtain the correct URIs to authenticate the user, and redirects the browser to your PingOne Advanced Identity Cloud tenant.

After authentication, PingOne Advanced Identity Cloud redirects the browser back to your application, which then obtains an OAuth 2.0 access token and displays the related user information.

Run the sample

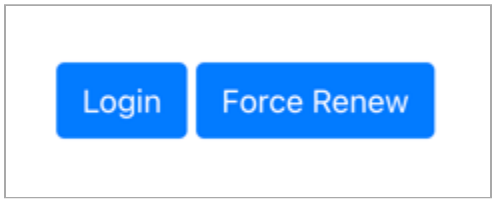
1. In a terminal window, navigate to the `/javascript` folder in your `sdk-sample-apps` project.
2. To run the embedded login sample, enter the following:

```
npm run start:central-login-oidc
```

3. In a web browser, navigate to the following URL:

   `https://localhost:8443`

   The sample displays a page with two buttons:



4. Click **Login**.

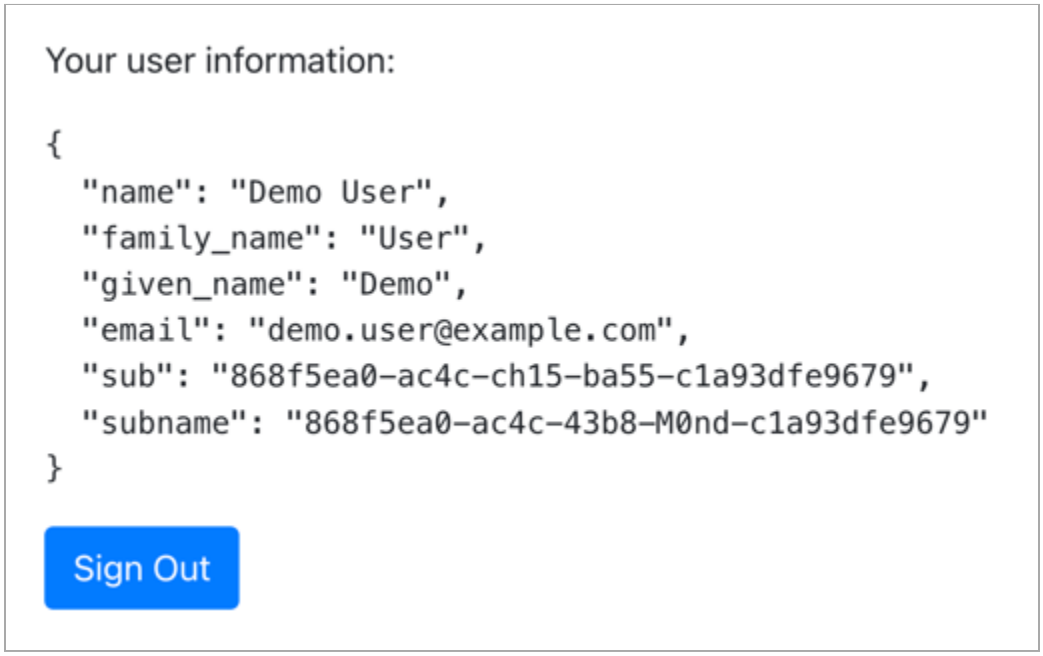
   The sample app redirects the browser to your PingOne Advanced Identity Cloud instance.

> **Tip**
>
> To see the application calling the `authorize` endpoint, and the redirect back from PingOne Advanced Identity Cloud with the `code` and `state` OAuth 2.0 parameters, open the **Network** tab of your browser's developer tools.

5. Authenticate as a known user in your PingOne Advanced Identity Cloud tenant.

   After successful authentication, PingOne Advanced Identity Cloud redirects the browser to the client application.

   If the app displays the user information, authentication was successful:



6. To revoke the OAuth 2.0 token, click the **Sign Out** button.

In this tutorial, PingOne Advanced Identity Cloud redirects users back to the client application, ready to authenticate again.

Recap

Congratulations!

You have now used the Ping SDK for JavaScript to obtain an OAuth 2.0 access token on behalf of a user from your PingOne Advanced Identity Cloud tenant.

You have seen how to obtain OAuth 2.0 tokens, and view the related user information.

More information

- API reference: `TokenManager`
- API reference: `UserManager`

OIDC login to PingAM tutorial for Android

Prepare › Download › Install › Configure › Run

In this tutorial you update a sample app that uses OIDC-based login to obtain tokens by redirecting to the PingAM UI for authentication.

The sample connects to the `.well-known` endpoint of your PingAM server to obtain the correct URIs to authenticate the user, and redirects to your PingAM server's login UI.

After authentication, PingAM redirects the browser back to your application, which then obtains an OAuth 2.0 access token and displays the related user information.

Before you begin

Before you begin this tutorial ensure you have set up your PingAM server with the required configuration.

For example, you will need to have an OAuth 2.0 client application set up, and a demo user to authenticate.

Complete prerequisites »

Step 1. Download the samples

To start this tutorial, you need to download the SDK sample apps repo, which contains the projects you will use.

Start step 1 »

Step 2. Install the Ping SDK

The sample projects need a number of dependencies that you can install by using the `npm` command.

For example, the Ping SDK for JavaScript itself is one of the dependencies.

Start step 2 »

Step 3. Configure connection properties

In this step, you configure the sample app to connect to the OAuth 2.0 application you created in PingAM.

Start step 3 »

Step 4. Test the app

To test the app, run the sample that you configured in the previous step.

The sample connects to your PingAM server to obtain the correct URIs to authenticate the user, and redirects the browser to your PingAM server.

After authentication, PingAM redirects the browser back to your application, which then obtains an OAuth 2.0 access token and displays the related user information.

Test app »

Before you begin

Prepare › Download › Install › Configure › Run

To successfully complete this tutorial refer to the prerequisites in this section.

The tutorial also requires a configured PingAM server.

Node and NPM

This sample requires a minimum Node.js version of `18`, and is tested on versions `18` and `20`. To get a supported version of Node.js, refer to the Node.js download page.

You will also need `npm` to build the code and run the samples.

Server configuration

This tutorial requires you to configure your PingAM server as follows:

The samples and tutorials in this documentation often require that you have an identity set up so that you can test authentication.

To create a demo user in PingAM, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. Navigate to **Identities**, and then click **+ Add Identity**.
3. Enter the following details:
    - **User ID** = `demo`
    - **Password** = `Ch4ng3it!`
    - **Email Address** = `demo.user@example.com`
4. Click **Create**.

Authentication trees provide fine-grained authentication by allowing multiple paths and decision points throughout the authentication flow. Authentication trees are made up of nodes that define actions taken during authentication.

Each node performs a single task, such as collecting a username or making a simple decision. Nodes can have multiple outcomes rather than just success or failure. For details, see the Authentication nodes configuration reference in the PingAM documentation.

To create a simple tree for use when testing the Ping SDKs, follow these steps:

1. Under **Realm Overview**, click **Authentication Trees**, then click **Create Tree**.
2. Enter a tree name, for example `sdkUsernamePasswordJourney`, and then click **Create**.

   The authentication tree designer appears, showing the **Start** entry point connected to the **Failure** exit point.
3. Drag the following nodes from the **Components** panel on the left side into the designer area:
    - **Page Node**
    - **Username Collector**
    - **Password Collector**
    - **Data Store Decision**
4. Drag and drop the **Username Collector** and **Password Collector** nodes onto the **Page Node**, so that they both appear on the same page when logging in.
5. Connect the nodes as follows:

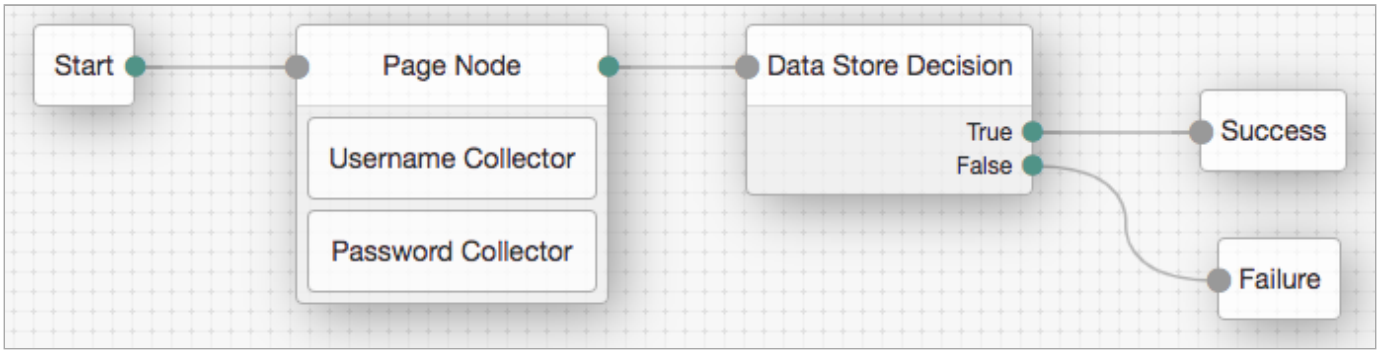


Figure 1. Example username and password authentication tree

6. Select the **Page Node**, and in the **Properties** pane, set the **Stage** property to `UsernamePassword`.

> **Tip**
>
> You can configure the node properties by selecting a node and altering properties in the right-hand panel.

One of the samples uses this specific value to determine the custom UI to display.

7. Click **Save**.

Public clients do not use a client secret to obtain tokens because they are unable to keep them hidden. The Ping SDKs commonly use this type of client to obtain tokens, as they cannot guarantee safekeeping of the client credentials in a browser or on a mobile device.

To register a *public* OAuth 2.0 client application for use with the SDKs in AM, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. Navigate to **Applications** > **OAuth 2.0** > **Clients**, and then click **+ Add Client**.
3. In **Client ID**, enter `sdkPublicClient`.
4. Leave **Client secret** empty.
5. In **Redirection URIs**, enter the following values:

   `https://localhost:8443/callback.html`

> **Tip**
>
> The Ping SDK for JavaScript attempts to load the redirect page to capture the OAuth 2.0 `code` and `state` query parameters that the server appended to the redirect URL.
> If the page you redirect to does not exist, takes a long time to load, or runs any JavaScript you might get a timeout, delayed authentication, or unexpected errors.
> To ensure the best user experience, we ***highly recommend*** that you redirect to a static HTML page with minimal HTML and no JavaScript when obtaining OAuth 2.0 tokens.

+

> **Important**
>
> Also add any other domains where you will be hosting SDK applications. . In **Scopes**, enter the following values:

+ `openid profile email address` . Click **Create**.

+ PingAM creates the new OAuth 2.0 client, and displays the properties for further configuration. . On the **Core** tab: .. In **Client type**, select `Public` . .. Disable **Allow wildcard ports in redirect URIs**. .. Click **Save Changes**. . On the **Advanced** tab: .. In **Grant Types**, enter the following values:

+

```
Authorization Code
Refresh Token
```

1. In **Token Endpoint Authentication Method**, select `None` .
2. Enable the **Implied consent** property.
    1. Click **Save Changes**.

The provider specifies the supported OAuth 2.0 configuration options for a realm.

To ensure the PingAM OAuth 2.0 provider service is configured for use with the Ping SDKs, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. In the left panel, click **Services**.
3. In the list of services, click **OAuth2 Provider**.
4. On the **Core** tab, ensure **Issue Refresh Tokens** is enabled.
5. On the **Consent** tab, ensure **Allow Clients to Skip Consent** is enabled.
6. Click **Save Changes**.

Cross-origin resource sharing (CORS) lets user agents make cross-domain server requests. In PingAM, you can configure CORS to allow browsers from trusted domains to access PingAM protected resources. For example, you might want a custom web application running on your own domain to get an end-user's profile information using the PingAM REST API.

The Ping SDK for JavaScript samples and tutorials all use `https://localhost:8443` as the host domain, which you should add to your CORS configuration.

If you are using a different URL for hosting SDK applications, ensure you add them to the CORS configuration as accepted origin domains.

To enable CORS in PingAM, and create a CORS filter to allow requests from your configured domain names, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. Navigate to **Configure** > **Global Services** > **CORS Service** > **Configuration**, and set the **Enable the CORS filter** property to `true` .

> **Important**
>
> If this property is not enabled, CORS headers are not added to responses from PingAM, and CORS is disabled entirely.

3. On the **Secondary Configurations** tab, click **Click Add a Secondary Configuration**.
4. In the **Name** field, enter `ForgeRockSDK`.
5. in the **Accepted Origins** field, enter any DNS aliases you use for your SDK apps.

   This documentation assumes the following configuration:

Property	Values
`Accepted Origins`	`https://localhost:8443`
`Accepted Methods`	`GET` `POST`

`Accepted Headers`	`accept-api-version`
	`x-requested-with`
	`content-type`
	`authorization`
	`if-match`
	`x-requested-platform`
	`iPlanetDirectoryPro` [1]
	`ch15fefc5407912` [2]

`Exposed Headers`	`authorization` `content-type`

6. Click **Create**.

   PingAM displays the configuration of your new CORS filter.
7. On the CORS filter configuration page:
   1. Ensure **Enable the CORS filter** is enabled.
   2. Set the **Max Age** property to `600`
   3. Ensure **Allow Credentials** is enabled.
8. Click **Save Changes**.

1. Cookie name value in PingAM servers.
2. In PingOne Advanced Identity Cloud tenants, go to **Tenant Settings > Global Settings > Cookie** to find this dynamic cookie name value.

Step 1. Download the samples

Prepare › **Download** › Install › Configure › Run

To start this tutorial, you need to download the Ping SDK sample apps repo, which contains the projects you will use.

1. In a web browser, navigate to the Ping SDK sample apps repository.
2. Download the source code using one of the following methods:

Download a ZIP file

1. Click **Code**, and then click **Download ZIP**.
2. Extract the contents of the downloaded ZIP file to a suitable location.

Use a Git-compatible tool to clone the repo locally

1. Click **Code**, and then copy the HTTPS URL.
2. Use the URL to clone the repository to a suitable location.

   For example, from the command-line you could run:

```
git clone https://github.com/ForgeRock/sdk-sample-apps.git
```

The result of these steps is a local folder named `sdk-sample-apps`.

Step 2. Install the Ping SDK

Prepare › Download › **Install** › Configure › Run

In the following procedure, you install the Ping SDK for JavaScript.

1. In a terminal window, navigate to the `sdk-sample-apps` folder.
2. To install the required packages, enter the following:

```
npm install
```

   The `npm` tool downloads the required packages, and places them inside a `node_modules` folder.

More information

- API reference: `Ping SDK for JavaScript`
- Cloning a repository | GitHub Help
- Get npm | npm

Step 3. Configure connection properties

Prepare › Download › Install › **Configure** › Run

In this step, you configure the sample app to connect to the OAuth 2.0 application you created in PingOne Advanced Identity Cloud.

1. In the IDE of your choice, open the `sdk-sample-apps` folder you cloned in the previous step.
2. Make a copy of the `/javascript/central-login-oidc/.env.example` file, and name it `.env`.

   The `.env` file provides the values used by the `forgerock.Config.setAsync()` method in `javascript/central-login-oidc/src/main.js`.
3. Update the `.env` file with the details of your PingAM server.

   ```
   SCOPE="$SCOPE"
   TIMEOUT=$TIMEOUT
   WEB_OAUTH_CLIENT="$WEB_OAUTH_CLIENT"
   WELL_KNOWN="$WELL_KNOWN"
   SERVER_TYPE="$SERVER_TYPE"
   ```

   Replace the following strings with the values you obtained when preparing your environment.

   $SCOPE

   The scopes you added to your OAuth 2.0 application in PingOne Advanced Identity Cloud.

   For example, `address email openid phone profile`

   $TIMEOUT

   How long to wait for OAuth 2.0 timeouts, in milliseconds.

   For example, `3000`

   $WEB_OAUTH_CLIENT

   The client ID from your OAuth 2.0 application in PingAM.

   For example, `sdkPublicClient`

   $WELL_KNOWN

   The `.well-known` endpoint from your PingAM tenant.

   For example, `https://openam.example.com:8443/openam/oauth2/.well-known/openid-configuration`

   $SERVER_TYPE

   Ensures the sample app uses the correct behavior for the different servers it supports, for example what logout parameters to use.

For PingOne Advanced Identity Cloud and PingAM servers, specify `AIC`.

The result resembles the following:

```
.env

SCOPE="address email openid phone profile"
TIMEOUT=3000
WEB_OAUTH_CLIENT="sdkPublicClient"
WELL_KNOWN="https://openam.example.com:8443/openam/oauth2/.well-known/openid-configuration"
SERVER_TYPE="AIC"
```

Step 4. Test the app

Prepare › Download › Install › Configure › **Run**

In the following procedure, you run the sample app that you configured in the previous step.

The sample connects to your PingAM server to obtain the correct URIs to authenticate the user, and redirects the browser to your PingAM server.

After authentication, PingAM redirects the browser back to your application, which then obtains an OAuth 2.0 access token and displays the related user information.

Run the sample

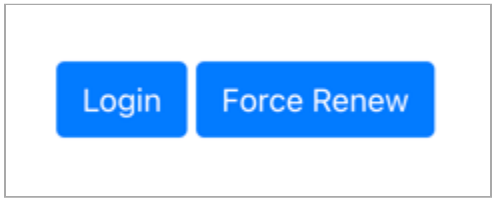
1. In a terminal window, navigate to `sdk-sample-apps/javascript` in your project.
2. To run the embedded login sample, enter the following:

   ```
   npm run start:central-login-oidc
   ```

3. In a web browser, navigate to the following URL:

   `https://localhost:8443`

   The sample displays a page with two buttons:

4. Click **Login**.

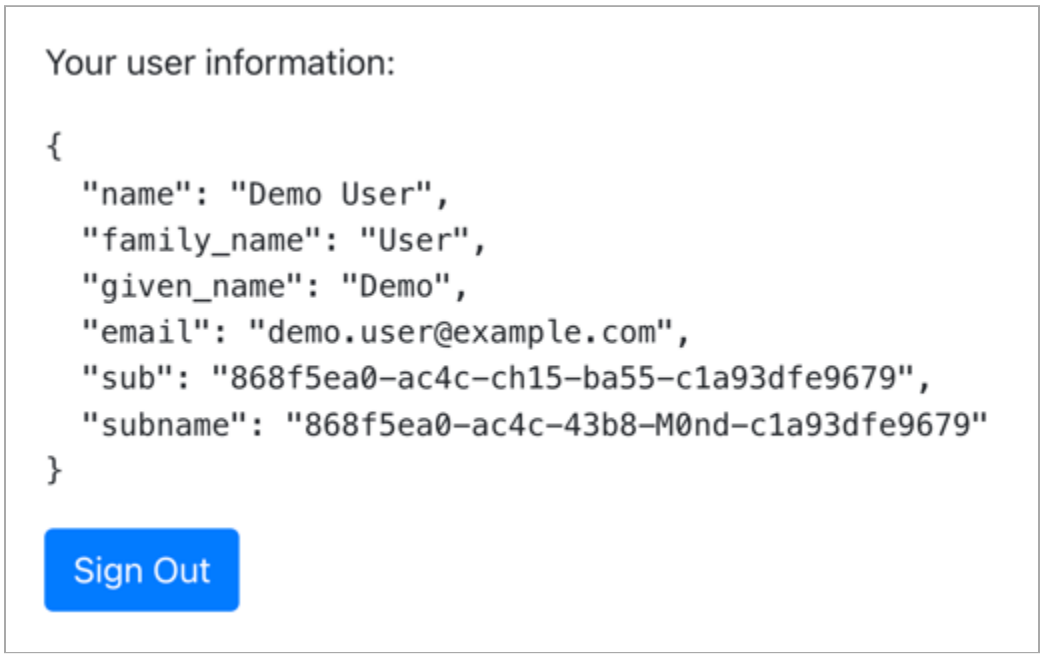
   The sample app redirects the browser to your PingAM instance.

> **Tip**
>
> To see the application calling the `authorize` endpoint, and the redirect back from PingAM with the `code` and `state` OAuth 2.0 parameters, open the **Network** tab of your browser's developer tools.

5. Authenticate as a known user in your PingAM system.

   After successful authentication, PingAM redirects the browser to the client application.

   If the app displays the user information, authentication was successful:



6. To revoke the OAuth 2.0 token, click the **Sign Out** button.

   In this tutorial, PingAM redirects users back to the client application, ready to authenticate again.

Recap

Congratulations!

You have now used the Ping SDK for JavaScript to obtain an OAuth 2.0 access token on behalf of a user from your PingAM server.

You have seen how to obtain OAuth 2.0 tokens, and view the related user information.

More information

- API reference: `TokenManager`
- API reference: `UserManager`

OIDC login to PingFederate tutorial for JavaScript

Prepare › Download › Install › Configure › Run

In this tutorial you update a sample app that uses OIDC-based login to obtain tokens by redirecting to the PingFederate UI for authentication.

The sample connects to the `.well-known` endpoint of your PingFederate server to obtain the correct URIs to authenticate the user, and redirects to your PingFederate server's login UI.

After authentication, PingFederate redirects the browser back to your application, which then obtains an OAuth 2.0 access token and displays the related user information.

Before you begin

Before you begin this tutorial ensure you have set up your PingFederate server with the required configuration.

For example, you will need to configure CORS, have an OAuth 2.0 client application set up.

Complete prerequisites »

Step 1. Download the samples

To start this tutorial, you need to download the SDK sample apps repo, which contains the projects you will use.

Start step 1 »

Step 2. Install the Ping SDK

The sample projects need a number of dependencies that you can install by using the `npm` command.

For example, the Ping SDK for JavaScript itself is one of the dependencies.

Start step 2 »

Step 3. Configure connection properties

In this step, you configure the sample app to connect to the OAuth 2.0 application you created in PingFederate.

Start step 3 »

Step 4. Test the app

In the following procedure, you run the sample app that you configured in the previous step.

The sample connects to your PingFederate server to obtain the correct URIs to authenticate the user, and redirects the browser to your PingFederate server.

After authentication, PingFederate redirects the browser back to your application, which then obtains an OAuth 2.0 access token and displays the related user information.

Test app »

Before you begin

Prepare › Download › Install › Configure › Run

To successfully complete this tutorial refer to the prerequisites in this section.

The tutorial also requires a configured PingFederate server.

Prerequisites

Node and NPM

This sample requires a minimum Node.js version of `18`, and is tested on versions `18` and `20`. To get a supported version of Node.js, refer to the Node.js download page.

You will also need `npm` to build the code and run the samples.

Server configuration

This tutorial requires you to configure your PingFederate server as follows:

OAuth 2.0 client application profiles define how applications connect to PingFederate and obtain OAuth 2.0 tokens.

To allow the Ping SDKs to connect to PingFederate and obtain OAuth 2.0 tokens, you must register an OAuth 2.0 client application:

1. Log in to the PingFederate administration console as an administrator.
2. Navigate to **Applications** > **OAuth** > **Clients**.
3. Click **Add Client**.

   PingFederate displays the **Clients** | **Client** page.
4. In **Client ID** and **Name**, enter a name for the profile, for example `sdkPublicClient`

   Make a note of the **Client ID** value, you will need it when you configure the sample code.

5. In **Client Authentication**, select `None`.
6. In **Redirect URIs**, add the following values:

   `https://localhost:8443`

   > **Important**
   >
   > Also add any other URLs where you host SDK applications.
   > Failure to add redirect URLs that exactly match your client app's values can cause PingFederate to display an error message such as `Redirect URI mismatch` when attempting to end a session by redirecting from the SDK.

7. In **Allowed Grant Types**, select the following values:

   `Authorization Code`

   `Refresh Token`

8. In the **OpenID Connect** section:

   1. In **Logout Mode**, select **Ping Front-Channel**
   2. In **Front-Channel Logout URIs**, add the following values:

      `https://localhost:8443`

      > **Important**
      >
      > Also add any other URLs that redirect users to PingFederate to end their session.
      > Failure to add sign off URLs that exactly match your client app's values can cause PingFederate to display an error message such as `invalid post logout redirect URI` when attempting to end a session by redirecting from the SDK.

   3. In **Post-Logout Redirect URIs**, add the following values:

      `https://localhost:8443`

9. Click **Save**.

   > **Important**
   >
   > After changing PingFederate configuration using the administration console, you must replicate the changes to each server node in the cluster before they take effect.
   > In the PingFederate administration console, navigate to **System > Server > Cluster Management**, and click **Replicate**.

   The application is now configured to accept client connections from and issue OAuth 2.0 tokens to the Ping SDK PingFederate example applications and tutorials covered by this documentation.

Cross-origin resource sharing (CORS) lets user agents make cross-domain server requests. In PingFederate, you can configure CORS to allow browsers or apps from trusted domains to access protected resources.

To configure CORS in PingFederate follow these steps:

1. Log in to the PingFederate administration console as an administrator.
2. Navigate to **System > OAuth Settings > Authorization Server Settings**.
3. In the **Cross-Origin Resource Sharing Settings** section, in the **Allowed Origin** field, enter any DNS aliases you use for your SDK apps.

   This documentation assumes the following configuration:

Property	Values
`Allowed Origin`	`https://localhost:8443`

4. Click **Save**.

> **Important**
>
> After changing PingFederate configuration using the administration console, you must replicate the changes to each server node in the cluster before they take effect.
> In the PingFederate administration console, navigate to **System > Server > Cluster Management**, and click **Replicate**.

Your PingFederate server is now able to accept connections from origins hosting apps built with the Ping SDKs.

Step 1. Download the samples

Prepare › Download › Install › Configure › Run

To start this tutorial, you need to download the Ping SDK sample apps repo, which contains the projects you will use.

1. In a web browser, navigate to the Ping SDK sample apps repository.
2. Download the source code using one of the following methods:

Download a ZIP file

1. Click **Code**, and then click **Download ZIP**.
2. Extract the contents of the downloaded ZIP file to a suitable location.

Use a Git-compatible tool to clone the repo locally

1. Click **Code**, and then copy the HTTPS URL.
2. Use the URL to clone the repository to a suitable location.

   For example, from the command-line you could run:

```
git clone https://github.com/ForgeRock/sdk-sample-apps.git
```

The result of these steps is a local folder named `sdk-sample-apps`.

Step 2. Install the Ping SDK

Prepare › Download › **Install** › Configure › Run

In the following procedure, you install the Ping SDK for JavaScript.

1. In a terminal window, navigate to the `sdk-sample-apps` folder.
2. To install the required packages, enter the following:

   ```
   npm install
   ```

   The `npm` tool downloads the required packages, and places them inside a `node_modules` folder.

More information

- API reference: `Ping SDK for JavaScript`
- Cloning a repository | GitHub Help
- Get npm | npm

Step 3. Configure connection properties

Prepare › Download › Install › **Configure** › Run

In this step, you configure the sample app to connect to the OAuth 2.0 application you created in PingFederate.

1. In the IDE of your choice, open the `sdk-sample-apps` folder you cloned in the previous step.
2. Open the `/javascript/central-login-oidc/src/main.js` file.
3. Locate the `forgerock.Config.setAsync()` method and update the properties to match your PingFederate environment:

```
await forgerock.Config.setAsync({
  clientId: process.env.WEB_OAUTH_CLIENT, // e.g. 'ForgeRockSDKClient' or PingOne Services Client GUID
  redirectUri: `${window.location.origin}`, // Redirect back to your app, e.g. 'https://localhost:8443' or the
domain your app is served.
  scope: process.env.SCOPE, // e.g. 'openid profile email address phone revoke' When using PingOne services
revoke scope is required
  serverConfig: {
    wellknown: process.env.WELL_KNOWN,
    timeout: process.env.TIMEOUT, // Any value between 3000 to 5000 is good, this impacts the redirect time to
login. Change that according to your needs.
  },
});
```

Replace the following strings with the values you obtained when you registered an OAuth 2.0 application in PingFederate:

process.env.WEB_OAUTH_CLIENT

The client ID from your OAuth 2.0 application in PingFederate.

For example, `sdkPublicClient`

process.env.SCOPE

The scopes you added to your OAuth 2.0 application in PingFederate.

For example, `openid profile email phone`

process.env.WELL_KNOWN

The `.well-known` endpoint from your OAuth 2.0 application in PingFederate.

To form the `.well-known` endpoint for a PingFederate server:

1. Log in to your PingFederate administration console.
2. Navigate to **System** > **Server** > **Protocol Settings**.
3. Make a note of the **Base URL** value.

   For example, `https://pingfed.example.com`

   > **Note**
   >
   > Do not use the admin console URL.

4. Append `/.well-known/openid-configuration` after the base URL value to form the `.well-known` endpoint of your server.

   For example, `https://pingfed.example.com/.well-known/openid-configuration`.

   The SDK reads the OAuth 2.0 paths it requires from this endpoint.

For example, `https://pingfed.example.com/.well-known/openid-configuration`

process.env.TIMEOUT

Enter how many milliseconds to wait before timing out the OAuth 2.0 flow.

For example, `3000`

The result resembles the following:

```
await forgerock.Config.setAsync({
  clientId: "sdkPublicClient",
  redirectUri: `${window.location.origin}`,
  scope: "openid profile email phone",
  serverConfig: {
    wellknown: "https://auth.pingone.com/3072206d-c6ce-ch15-m0nd-f87e972c7cc3/as/.well-known/openid-configuration",
    timeout: 3000
  },
});
```

Step 4. Test the app

Prepare › Download › Install › Configure › **Run**

In the following procedure, you run the sample app that you configured in the previous step.

The sample connects to your PingFederate server to obtain the correct URIs to authenticate the user, and redirects the browser to your PingFederate server.

After authentication, PingFederate redirects the browser back to your application, which then obtains an OAuth 2.0 access token and displays the related user information.

Run the sample

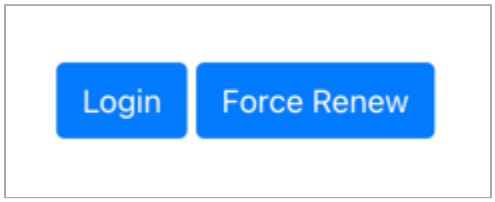
1. In a terminal window, navigate to the `/javascript` folder in your `sdk-sample-apps` project.
2. To run the embedded login sample, enter the following:

```
npm run start:central-login-oidc
```

3. In a web browser, navigate to the following URL:

   `https://localhost:8443`

   The sample displays a page with two buttons:



4. Click **Login**.

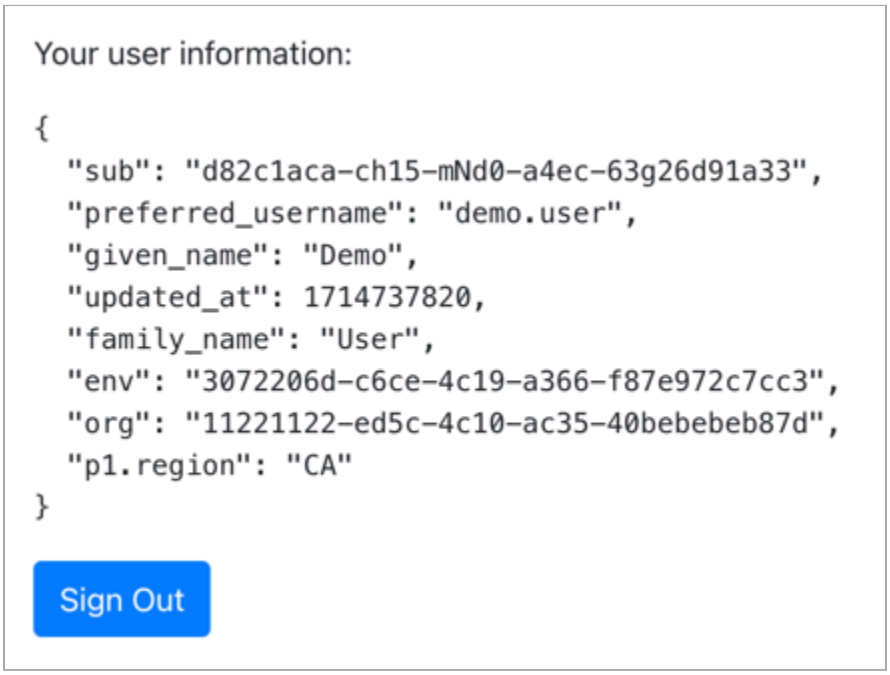
   The sample app redirects the browser to your PingFederate instance.

   > **Tip**
   >
   > To see the application calling the `authorize` endpoint, and the redirect back from PingFederate with the `code` and `state` OAuth 2.0 parameters, open the **Network** tab of your browser's developer tools.

5. Authenticate as a known user in your PingFederate system.

   After successful authentication, PingFederate redirects the browser to the client application.

   If the app displays the user information, authentication was successful:



6. To revoke the OAuth 2.0 token, click the **Sign Out** button.

   In this tutorial, PingFederate redirects users back to the client application, ready to authenticate again.

Recap

Congratulations!

You have now used the Ping SDK for JavaScript to obtain an OAuth 2.0 access token on behalf of a user from your PingFederate server.

You have seen how to obtain OAuth 2.0 tokens, and view the related user information.

More information

- API reference: `TokenManager`
- API reference: `UserManager`

Implement your use cases with the Ping SDKs

The SDKs enable you to implement many authentication, registration, and self-service use cases into your mobile and web apps.

Visit the following pages for more information on implementing different OIDC login use cases using the Ping SDKs:

Creating a custom UI to share across OIDC apps

Applies to: Android | iOS | JavaScript

Learn how to replace the default PingAM or PingOne Advanced Identity Cloud user interface for authentication with your own custom user interface.

You'll use an existing JavaScript sample application to act as your custom UI. This app will step through your authentication journeys, and act as the central UI for one or more sample client apps.

Read more »

Creating a custom UI app to share across OIDC apps

Server support:

- ✕ PingOne
- ✓ PingOne Advanced Identity Cloud
- ✓ PingAM
- ✕ PingFederate

SDK support:

- ✓ Ping SDK for Android
- ✓ Ping SDK for iOS
- ✓ Ping SDK for JavaScript

In this tutorial you replace the default PingAM or PingOne Advanced Identity Cloud user interface for authentication with your own custom user interface.

You'll use an existing JavaScript sample application to act as your custom UI. This app will step through your authentication journeys, and act as the central UI for one or more sample client apps.

Understanding the custom UI flow

When the Ping SDKs perform OIDC login they initiate the OAuth 2.0 Authorization Code flow on your PingOne Advanced Identity Cloud tenant or PingAM server.

Your server would usually use its built-in user interface to authenticate the user, before returning to the client app with the authorization code.

For this tutorial you'll configure your server to use your custom UI app to authenticate users instead.

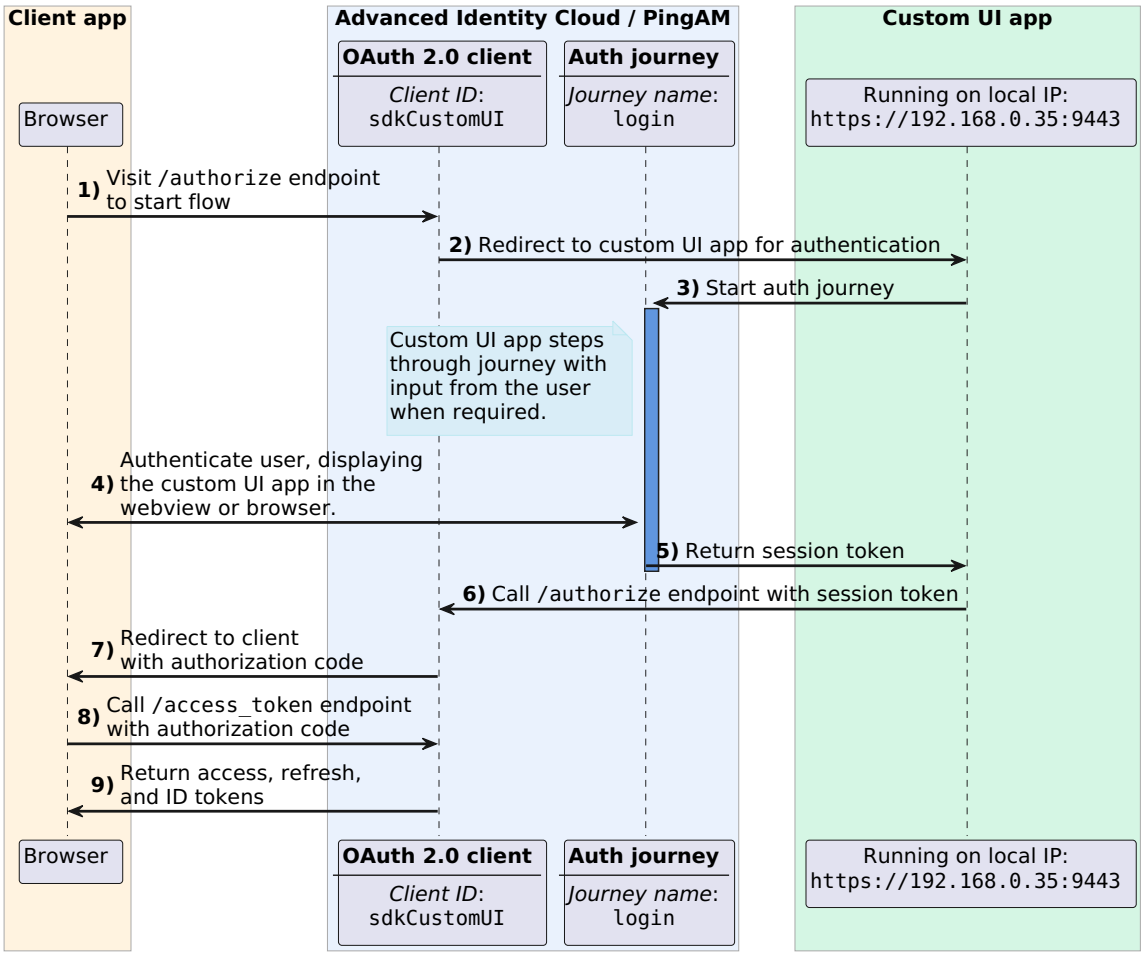


Figure 1. Custom UI app in OAuth 2.0 authorization code flow

The overall authentication flow is as follows:

1. In the client app, when the user initiates sign on the client app calls the `/authorize` endpoint to start the OAuth 2.0 flow.
2. The OAuth 2.0 client you configure in the server needs to authenticate the user, and redirects users to your custom UI.

   It appends a `goto` query parameter to the URL, so that the custom UI app can redirect to the server after successful authentication.
3. The custom UI starts the authentication journey, and steps through each node, handling the callbacks as necessary.

   Note that the custom UI app does not perform any OAuth 2.0 operations itself. Its only role is to authenticate the user by stepping through the journey, and then returning back to the server with the session token, so that the server can continue the OAuth 2.0 flow.

4. The custom UI app renders the UI to handle any interactive callbacks in the journey. For example, to capture the username and password credentials.

5. When authentication is successful, the server issues a session token on behalf of the user to the custom UI app.

   The following snippet shows how the custom sample app handles the successful authentication, and uses the `goto` parameter:

   Excerpt from `/javascript/embedded-login/src/main.js`

   ```
   // Check URL for query parameters
   const url = new URL(document.location);
   const params = url.searchParams;
   const goto = params.get('goto');

   const handleStep = async (step) => {
     switch (step.type) {
       case 'LoginSuccess': {
         if (goto != null) {
           // Journey complete
           // Return to server to issue auth code
           window.location.replace(goto);
           return;
         }
       ...
     }
   ...
   }
   ```

   Important

   For security reasons your custom UI must validate that the URL in the `goto` parameter matches a domain you trust before redirecting to it.
   Failure to validate the domain of the `goto` URL before redirecting could result in link hijacking or other security-related problems.

6. Due to the presence of the `goto` parameter, the custom UI app can now redirect to the OAuth 2.0 `/authorize` endpoint, using the session token to authenticate the request.

> **Note**
>
> To authenticate the user the browser attaches the session token as a cookie in the request to the `/authorize` endpoint.
> Your server issues session token cookies for use on the same domain you have assigned to it. That domain will be one of the following:
> - The original URL for the server, for example:
>   - `openam-forgerock-sdks.forgeblocks.com` (PingOne Advanced Identity Cloud tenants)
>   - `openam.example.com` (PingAM servers)
> - A custom domain assigned to the server, for example:
>   - `id.mycompany.com`
>
> If the custom UI app is running on a different domain than your server then browsers consider the cookie to be from a third party.
> Some browsers, such as Safari block access to third-party cookies, so the requests that use them appear to not be authenticated.
> For this tutorial you can disable third-party cookie checks in the browser, but for production you must ensure your custom UI and your server are sharing the same domain.

7. The OAuth 2.0 client accepts the request, and returns the authorization code to the client application.

   The URL of the client app must match one of the redirect URIs listed in the OAuth 2.0 client configuration.

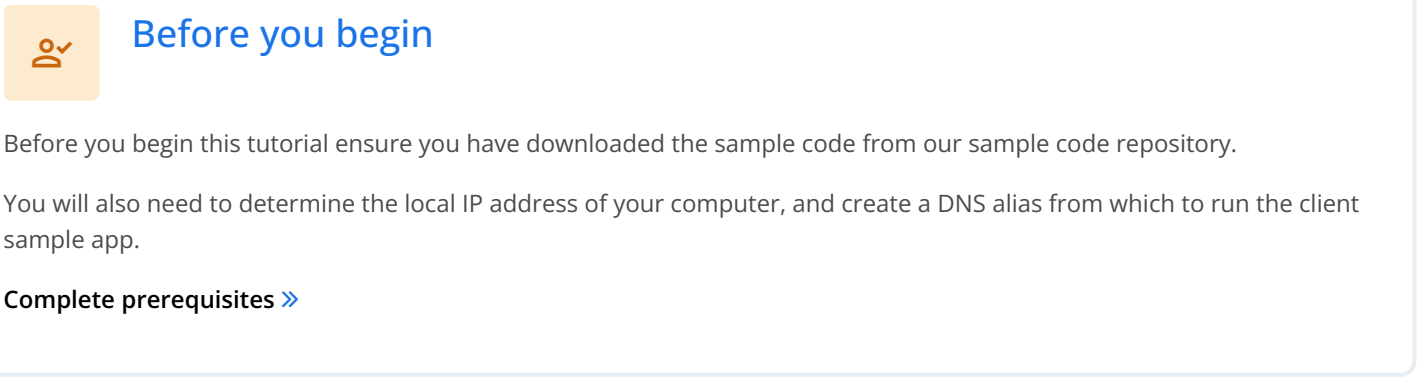
8. The client application recognizes that an authorization code is present and that authentication was a success, and can now exchange the authorization code by calling the `/access_token` OAuth 2.0 endpoint.

9. The server validates the authorization code and issues the access token, an ID token, and if enabled, refresh tokens.

   The client app can now call the `userinfo` endpoint, using the access token as a bearer token for authentication, and retrieve information about the user.

Tutorial steps

Complete the following tasks to try out this tutorial:



Part 1. Configuring your PingAM server or PingOne Advanced Identity Cloud tenant

In this section you configure your PingAM server or PingOne Advanced Identity Cloud tenant with an OAuth 2.0 client to accept connections from the sample client apps.

The OAuth 2.0 client also contains the configuration to redirect your users to authenticate using the custom UI sample app running on your local IP address.

Lastly, you configure Cross-origin Resource Sharing (CORS) to allow the sample UI app, and the client JavaScript app to connect to protected endpoints.

Start step 1 »

Part 2. Running the JavaScript custom UI sample app

In this section you configure the **embedded login** sample JavaScript app to act as your custom UI.

Start step 2 »

Part 3. Running a client sample app

The final step is to configure and run client OIDC sample apps.

These sample apps start the OAuth 2.0 flow on your PingAM server or PingOne Advanced Identity Cloud tenant, which redirects the user to your custom UI sample app, running locally on your computer.

After authentication the server redirects users back to the client sample apps and appends the `code` parameter, which your client app uses to complete the OAuth 2.0 flow.

Test app »

Before you begin

Step 1. Downloading the samples

You need to download the SDK sample apps repo, which contains the projects you will use for this tutorial.

1. In a web browser, navigate to the SDK Sample Apps repository.

2. Download the source code using one of the following methods:

Download a ZIP file

1. Click **Code**, and then click **Download ZIP**.
2. Extract the contents of the downloaded ZIP file to a suitable location.

Use a Git-compatible tool to clone the repo locally

1. Click **Code**, and then copy the HTTPS URL.
2. Use the URL to clone the repository to a suitable location.

   For example, from the command-line you could run:

```
git clone https://github.com/ForgeRock/sdk-sample-apps.git
```

The result of these steps is a local folder named `sdk-sample-apps`.

Step 2. Installing the dependencies

In the following procedure, you install the required modules and dependencies, including the Ping SDK for JavaScript.

1. In a terminal window, navigate to the `sdk-sample-apps/javascript` folder.
2. To install the required packages, enter the following:

```
npm install
```

   The `npm` tool downloads the required packages, and places them inside a `node_modules` folder.

Step 3. Hosting the sample apps

In a production scenario your custom login UI app would have its own fully-qualified domain name that your Android, iOS, and JavaScript clients could all connect to.

For simplicity, in this tutorial you will serve your custom login UI app from the local IP address of your host computer.

Using the local IP of your host computer means Android and iOS apps running on a simulator can resolve the address, and also JavaScript apps running locally.

Obtaining your local IP address

Complete the following steps to obtain your local IP address:

Windows

1. In a command prompt, enter `ipconfig /all`

   Windows displays information about the network adapters in your computer.

```
Windows IP Configuration
   Host Name . . . . . . . . . . . . : Windows
   Primary Dns Suffix  . . . . . . . :
   Node Type . . . . . . . . . . . . : Hybrid
   IP Routing Enabled. . . . . . . . : No
   WINS Proxy Enabled. . . . . . . . : No

Ethernet adapter Ethernet:
   Media State . . . . . . . . . . . : Media disconnected
   Description . . . . . . . . . . . : E3100G 2.5 Gigabit Ethernet Controller
   Physical Address. . . . . . . . . : 74-34-E2-2b-30-44
   DHCP Enabled. . . . . . . . . . . : Yes
   Autoconfiguration Enabled . . . . : Yes

Wireless LAN adapter Local Area Connection* 1:
   Media State . . . . . . . . . . . : Media disconnected
   Description . . . . . . . . . . . : Microsoft Wi-Fi Direct Virtual Adapter
   Physical Address. . . . . . . . . : 67-6C-EB-B3-46-82
   DHCP Enabled. . . . . . . . . . . : Yes
   Autoconfiguration Enabled . . . . : Yes

Wireless LAN adapter Wi-Fi:
   Description . . . . . . . . . . . : Wireless Network Adapter (210NGW)
   Physical Address. . . . . . . . . : 87-6C-DF-C9-17-90
   DHCP Enabled. . . . . . . . . . . : Yes
   Autoconfiguration Enabled . . . . : Yes
   IPv6 Address. . . . . . . . . . . : 2406:3d08:2f61:1400::2d47
   Lease Obtained. . . . . . . . . . : January 27, 2025 11:09:26 AM
   Lease Expires . . . . . . . . . . : January 28, 2025 6:09:26 AM
   IPv6 Address. . . . . . . . . . . : 2406:3d08:2f61:1400::2d47
   Temporary IPv6 Address. . . . . . : 2604:2b08:2f93:2600:b479:b5b4:25ff:acc8
   Link-local IPv6 Address . . . . . : fe54::d9e5:16ff:d9d4:e22%10
   IPv4 Address. . . . . . . . . . . : 192.168.0.35
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Lease Obtained. . . . . . . . . . : January 27, 2025 11:09:24 AM
   Lease Expires . . . . . . . . . . : January 29, 2025 11:09:26 AM
   Default Gateway . . . . . . . . . : fe80::bb8:c0ee:fea5:8c58%10
                                       192.168.0.1
   DHCP Server . . . . . . . . . . . : 192.168.0.1
   DHCPv6 IAID . . . . . . . . . . . : 893252287
   DHCPv6 Client DUID. . . . . . . . : 00-01-00-01-1b-87-59-2D-74-86-C4-3C-30-88
   DNS Servers . . . . . . . . . . . : 2025:4e8:0:230b::11
                                       2025:4e8:0:230c::11
                                       8.8.8.8
   NetBIOS over Tcpip. . . . . . . . : Enabled
```

2. Ignoring adapters where the **Media State** property is listed as `Media Disconnected`, locate the ethernet or wireless adapter that connects to your router.

3. Make a note of the **IPv4 Address** field.

> **Tip**
>
> The address will often start with `192.168.`, `10.0.`, or `172.16.`, which are the first digits of the commonly used reserved private IPv4 addresses.

In this case, the local IPv4 IP address is `192.168.0.35`.

You will use this address to access your custom UI app for this tutorial.

macOS

1. In a terminal window, enter `ifconfig`

macOS displays information about the network interfaces in your computer.

```
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 16384
        options=1203<RXCSUM,TXCSUM,TXSTATUS,SW_TIMESTAMP>
        inet 127.0.0.1 netmask 0xff000000
        inet6 ::1 prefixlen 128
        inet6 fe80::1%lo0 prefixlen 64 scopeid 0x1
        nd6 options=201<PERFORMNUD,DAD>
gif0: flags=8010<POINTOPOINT,MULTICAST> mtu 1280
stf0: flags=0<> mtu 1280
anpi0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        options=400<CHANNEL_IO>
        ether 22:d0:cb:e5:fd:09
        media: none
        status: inactive
en3: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        options=404<VLAN_MTU,CHANNEL_IO>
        ether f8:e4:3b:ad:67:c5
        inet6 fe80::ca4:9a6c:f835:80c9%en8 prefixlen 64 secured scopeid 0x7
        inet6 fd84:bb80:dd60:23b3:855:171f:3651:b7de prefixlen 64 autoconf secured
        inet 192.168.0.35 netmask 0xffffff00 broadcast 192.168.0.255
        nd6 options=201<PERFORMNUD,DAD>
        media: autoselect (1000baseT <full-duplex>)
        status: active
en1: flags=8963<UP,BROADCAST,SMART,RUNNING,PROMISC,SIMPLEX,MULTICAST> mtu 1500
        options=460<TSO4,TSO6,CHANNEL_IO>
        ether 36:e5:80:6e:d1:40
        media: autoselect <full-duplex>
        status: inactive
en2: flags=8963<UP,BROADCAST,SMART,RUNNING,PROMISC,SIMPLEX,MULTICAST> mtu 1500
        options=460<TSO4,TSO6,CHANNEL_IO>
        ether 36:e5:80:6e:d1:44
        media: autoselect <full-duplex>
        status: inactive
bridge0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        options=63<RXCSUM,TXCSUM,TSO4,TSO6>
        ether 36:e5:80:6e:d1:40
        Configuration:
                id 0:0:0:0:0:0 priority 0 hellotime 0 fwddelay 0
                maxage 0 holdcnt 0 proto stp maxaddr 100 timeout 1200
                root id 0:0:0:0:0:0 priority 0 ifcost 0 port 0
                ipfilter disabled flags 0x0
        member: en1 flags=3<LEARNING,DISCOVER>
                ifmaxaddr 0 port 11 priority 0 path cost 0
        member: en2 flags=3<LEARNING,DISCOVER>
                ifmaxaddr 0 port 12 priority 0 path cost 0
        member: en3 flags=3<LEARNING,DISCOVER>
                ifmaxaddr 0 port 13 priority 0 path cost 0
        media: <unknown type>
        status: inactive
ap1: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        options=6460<TSO4,TSO6,CHANNEL_IO,PARTIAL_CSUM,ZEROINVERT_CSUM>
        ether d6:0f:2c:90:e9:b6
        nd6 options=201<PERFORMNUD,DAD>
        media: autoselect (none)
        status: inactive
```

```
en0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        options=6460<TSO4,TSO6,CHANNEL_IO,PARTIAL_CSUM,ZEROINVERT_CSUM>
        ether c6:2a:06:29:ee:28
        nd6 options=201<PERFORMNUD,DAD>
        media: autoselect
        status: inactive
utun0: flags=8051<UP,POINTOPOINT,RUNNING,MULTICAST> mtu 1500
        inet6 fe80::a19f:5de6:a4ca:fd90%utun0 prefixlen 64 scopeid 0x13
        nd6 options=201<PERFORMNUD,DAD>
```

2. Looking at interfaces where the **status** property is listed as `active`, locate the ethernet or wireless interface that connects to your router.

   Often the prefix of the interface is `en`.

3. Make a note of the IPv4 address in the **inet** field.

   > **Tip**
   >
   > The address will often start with `192.168.`, `10.0.`, or `172.16.`, which are the first digits of the commonly used reserved private IPv4 addresses.

   In this case, the local IPv4 IP address is `192.168.0.35`.

   You will use this address to access your custom UI app for this tutorial.

Creating a DNS alias for the JavaScript client application

You should assign a DNS alias to your localhost address to help differentiate the client application from the custom UI application during this tutorial.

You can choose whatever host name you prefer for your client application. This tutorial uses `sdkapp.example.com`.

Complete the following steps to configure a DNS alias for your local IP address:

Windows

1. As an administrator, in a text editor open the `%SystemRoot%\system32\drivers\etc\hosts` file.
2. Add the following:

   ```
   127.0.0.1 sdkapp.example.com
   ```

3. Close and save the file.

macOS

1. As an administrator, in a text editor open the `/etc/hosts` file.
2. Add the following:

   ```
   127.0.0.1 sdkapp.example.com
   ```

3. Close and save the file.

Part 1. Configuring your PingAM server or PingOne Advanced Identity Cloud tenant

In this section you configure your PingAM server or PingOne Advanced Identity Cloud tenant with an OAuth 2.0 client to accept connections from the sample client apps.

The OAuth 2.0 client also contains the configuration to redirect your users to authenticate using the custom UI sample app running on your local IP address.

Lastly, you configure Cross-origin Resource Sharing (CORS) to allow the sample UI app, and the client JavaScript app to connect to protected endpoints.

Step 1. Configure an OAuth 2.0 client

The initial request to authenticate a user from your client application requires an OAuth 2.0 client is setup.

This client also contains the configuration for your custom UI application. This means that you can have different UI applications for different clients, allowing you to apply different branding to each.

In this step you configure a suitable OAuth 2.0 client in either your PingAM server of PingOne Advanced Identity Cloud tenant, by using the **Access Management** native console.

If you are using an PingOne Advanced Identity Cloud tenant, follow these steps to open the **Access Management** native console:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. Navigate to **Native Consoles** > **Access Management**.

In the **Access Management** native console, complete these steps to register an OAuth 2.0 client for this tutorial:

1. Navigate to **Applications** > **OAuth 2.0** > **Clients**, and then click **+ Add Client**
2. On the **New OAuth 2.0 Client** page:
   1. In **Client ID**, enter an identifier for the client.

      For example, `sdkCustomUI`

   2. In **Redirection URIs**, enter the URIs that will be hosting your client applications.
      - Add the DNS alias you created earlier for the JavaScript sample client app:

`https://sdkapp.example.com:8443`

- If you want to test your setup with the Android or iOS sample apps, also add the following:

  `org.forgerock.demo://oauth2redirect`

3. In **Scopes**, enter each of the following:
   - `openid`
   - `profile`
   - `email`
   - `phone`
   - `address`
4. Click **Create**.

   The authorization server creates the client and navigates to the edit page for it.
5. On the **Core** tab:
   1. In **Client type**, select `Public`.
   2. Click **Save Changes**.

      > **Important**
      >
      > You must click **Save Changes** before changing tabs otherwise the page discards any changes you made on the tab.
6. On the **Advanced** tab:
   1. In **Token Endpoint Authentication Method**, select `none`.
   2. Set **Implied Consent** to **Enabled**.
   3. Click **Save Changes**.
7. On the **OAuth2 Provider Overrides** tab:
   1. Set **Enable OAuth2 Provider Overrides** to **Enabled**.
   2. In **Custom Login URL Template**, enter the URL of the custom UI application, followed by the query parameters required to navigate between the client, custom UI, and authorization server.

      For this tutorial, use the local IP address of your computer that you obtained earlier, and the port number you configured for the custom UI sample app (`9443`):

      ```
      https://192.168.0.35:9443?goto=${goto}<#if acrValues??>&acr_values=${acrValues}</#if><#if realm??>&realm=${realm}</#if><#if module??>&module=${module}</#if><#if service??>&service=${service}</#if><#if locale??>&locale=${locale}</#if>
      ```
   3. Set **Use Client-Side Access & Refresh Tokens** to **Enabled**.
   4. Set **Allow Clients to Skip Consent** to **Enabled**

5. Click **Save Changes**.

You have now configured the OAuth 2.0 client to issue tokens to your client applications, and redirect login requests to your custom UI application.

Step 2. Configure CORS

Cross-origin resource sharing (CORS) lets user agents make cross-domain server requests. Configure CORS to allow browsers from trusted addresses to access your protected resources.

For this tutorial you configure CORS to allow the client JavaScript application to access OAuth 2.0 endpoints, and the custom UI application to access your authentication journeys.

To configure CORS, select your authorization server:

PingOne Advanced Identity Cloud

To update the CORS configuration in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. At the top right of the screen, click your name, and then select **Tenant settings**.
3. On the **Global Settings** tab, click **Cross-Origin Resource Sharing (CORS)**.
4. Perform one of the following actions:
   - If available, click **ForgeRockSDK**.
   - If you haven't added any CORS configurations to the tenant, click **+ Add a CORS Configuration**, select **Ping SDK**, and then click **Next**.
5. Add the IP addresses and port numbers of where you are hosting the custom UI app and the client sample app to the **Accepted Origins** property.
6. Complete the remaining fields to suit your environment.

   An example configuration for this tutorial is as follows:

Property	Values
	`https://sdkapp.example.com:8443` `https://192.168.0.35:9443`
`Accepted Methods`	`GET` `POST`

`Accepted Headers`	`accept-api-version`
	`x-requested-with`
	`content-type`
	`authorization`
	`if-match`
	`x-requested-platform`
	`iPlanetDirectoryPro` [1]
	`ch15fefc5407912` [2]

`Exposed Headers`	`authorization` `content-type`
`Enable Caching`	`True`
`Max Age`	`600`
`Allow Credentials`	`True`

> **Tip**
>
> Click **Show advanced settings** to be able to edit all available fields.

7. Click **Save CORS Configuration**.

PingAM

To enable CORS in PingAM, and create a CORS filter to allow requests from your configured domain names, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. Navigate to **Configure** > **Global Services** > **CORS Service** > **Configuration**, and set the **Enable the CORS filter** property to `true`.

   > **Important**
   >
   > If this property is not enabled, CORS headers are not added to responses from PingAM, and CORS is disabled entirely.

3. On the **Secondary Configurations** tab, click **Click Add a Secondary Configuration**.
4. In the **Name** field, enter `ForgeRockSDK`.
5. In the **Accepted Origins** field, enter the IP addresses and port numbers of where you are hosting the custom UI app and the client sample app.

   An example configuration for this tutorial is as follows:

Property	Values
	`https://sdkapp.example.com:8443` `https://192.168.0.35:9443`
`Accepted Methods`	`GET` `POST`

`Accepted Headers`	`accept-api-version`
	`x-requested-with`
	`content-type`
	`authorization`
	`if-match`
	`x-requested-platform`
	`iPlanetDirectoryPro` [1]
	`ch15fefc5407912` [2]

`Exposed Headers`	`authorization` `content-type`
`Enable Caching`	`True`
`Max Age`	`600`
`Allow Credentials`	`True`

6. Click **Create**.

   PingAM displays the configuration of your new CORS filter.

7. On the CORS filter configuration page:
    1. Ensure **Enable the CORS filter** is enabled.
    2. Set the **Max Age** property to `600`
    3. Ensure **Allow Credentials** is enabled.
8. Click **Save Changes**.

You are now ready to configure and run the sample JavaScript app to act as the custom UI.

1. Cookie name value in PingAM servers.
2. In PingOne Advanced Identity Cloud tenants, go to **Tenant Settings > Global Settings > Cookie** to find this dynamic cookie name value.

Part 2. Running the JavaScript custom UI sample app

In this section you configure the **embedded login** sample JavaScript app to act as your custom UI.

This app walks your users through the authentication tree to obtain a session, which it returns to your client app via the authentication server.

1. In a JavaScript-capable IDE, open the `sdk-sample-apps` folder you downloaded earlier.
2. Navigate to the `/javascript/embedded-login` folder, and open the `.env.example` file.
3. Edit the values in the file to match your environment:
    1. In **SERVER_URL**, enter the base URL of the PingAM component of your deployment, including the deployment path.

        PingAM example:

        `https://openam.example.com:8443/openam`

        PingOne Advanced Identity Cloud example:

        `https://openam-forgerock-sdks.forgeblocks.com/am`

    2. In **REALM_PATH**, enter the realm that contains the authentication journey you will use.

        PingAM example:

        `root`

        PingOne Advanced Identity Cloud example:

        `alpha`

    3. In **TREE**, enter the name of the authentication journey to sign on end users.

        Note that the sample custom UI app only supports a limited number of callbacks by default, so choose a simple authentication tree that authenticates with only username and password.

For example, you can use the default `Login` authentication tree.

> **Tip**
>
> As the custom UI does not perform any OAuth 2.0 interactions you can leave the `SCOPE` and `WEB_OAUTH_CLIENT` properties blank.

The result will resemble the following:

Example `.env.example` file

```
SERVER_URL=https://openam-docs-regular.forgeblocks.com/am
REALM_PATH=alpha
SCOPE=
TIMEOUT=$TIMEOUT
TREE=Login
WEB_OAUTH_CLIENT=
```

4. Save the file as `.env` in the same folder.
5. Update the `webpack.config.js` file:
    1. Change the `port` value to `9443` so that it does not clash with the client sample app.
    2. Change the `host` value to `0.0.0.0`, so that the app is made available on your local IP address, rather than just `localhost`.

    The result resembles the following:

    ```
    devServer: {
      port: 9443,
      host: '0.0.0.0',
      ...
    }
    ```

6. From the `/javascript` folder, run the embedded login custom UI app as follows:

    ```
    cd javascript
    npm run start:embedded-login
    ```

    Webpack compiles the code and serves it on the local IP address of your computer.

7. In a browser, open the local IP address of your computer, with the port number you edited earlier.

    For example, `https://192.168.0.35:9443`

> **Tip**
>
> Webpack outputs links to the locations it is serving in the console, that you can click or copy and paste into a browser.
> Look for the line that includes the text `On Your Network (IPv4)`:
>
> ```
> [webpack-dev-server] Project is running at:
> [webpack-dev-server] Loopback: https://localhost:9443/
> [webpack-dev-server] On Your Network (IPv4): https://192.168.0.35:9443/
> [webpack-dev-server] On Your Network (IPv6): https://[fe80::1]:9443/
> ```

8. As the custom UI sample app is running on a self-signed SSL certificate on your local IP address, your browser might display a warning message.

   You can ignore this warning for this tutorial:

   - In **Chrome**, click **Advanced** and then click **Proceed to 192.168.0.35 (unsafe)**.
   - In **Firefox**, click **Advanced** and then click **Accept the risk and continue**.
   - In **Safari**, click **Show Details** and then click **visit this website**.

   The custom UI app displays the first interactive node of the configured authentication journey:

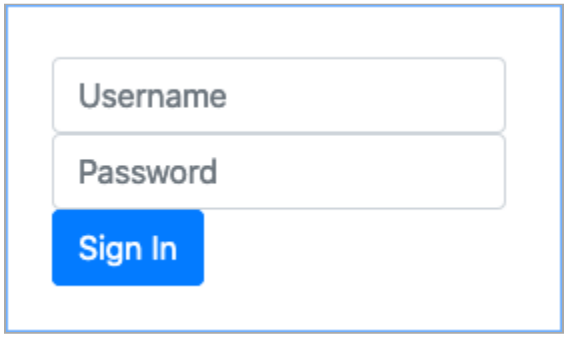


Figure 1. Custom UI app showing the first interactive node of the configured journey.

With the custom UI sample app running, and your server configured, you can now proceed to test the setup by using one of the OIDC sample apps as a client.

Part 3. Running a client sample app

In this section you configure one of the OIDC (centralized login) sample apps to test out your custom UI.

Running the JavaScript sample OIDC client app

1. In a JavaScript-capable IDE, open the `sdk-sample-apps` folder you downloaded earlier.
2. Open the `/javascript/central-login-oidc` sample.
3. Edit the values in the `.env.sample` file to match your environment:
   1. In **SCOPE**, enter the list of scopes to request are present in the issued OAuth 2.0 token.

For example, `"openid profile email phone address"`

2. In **WEB_OAUTH_CLIENT**, enter the client ID of the OAuth 2.0 client you configured earlier.

   For example, `sdkCustomUI`

3. In **WELL_KNOWN**, enter the `.well-known` URL of the realm in which you created the OAuth 2.0 client.

   You can view the `.well-known` endpoint for an OAuth 2.0 client in the PingOne Advanced Identity Cloud admin console:

   1. Log in to your PingOne Advanced Identity Cloud administration console.
   2. Click **Applications**, and then select the OAuth 2.0 client you created earlier. For example, sdkPublicClient.
   3. On the **Sign On** tab, in the **Client Credentials** section, copy the **Discovery URI** value.

   To form the `.well-known` URL for an PingAM server, concatenate the following information into a single URL:

   1. The base URL of the PingAM component of your deployment, including the port number and deployment path.

      For example, `https://openam.example.com:8443/openam`

   2. The string `/oauth2`
   3. The hierarchy of the realm that contains the OAuth 2.0 client.

      You must specify the entire hierarchy of the realm, starting at the Top Level Realm. Prefix each realm in the hierarchy with the `realms/` keyword.

      For example, `/realms/root/realms/customers`

      > **Tip**
      >
      > If you omit the realm hierarchy, the top level `ROOT` realm is used by default.

   4. The string `/.well-known/openid-configuration`

   For example, `https://openam.example.com:8443/openam/oauth2/realms/root/.well-known/openid-configuration`

The result will resemble the following:

Example `.env.example` file

```
SCOPE="openid profile email phone address"
TIMEOUT=1000
WEB_OAUTH_CLIENT=sdkCustomUI
WELL_KNOWN=https://openam-docs-regular.forgeblocks.com/am/oauth2/alpha/.well-known/openid-configuration
SERVER_TYPE=AIC
```

4. Save the file as `.env` in the same folder.

5. Edit the `webpack.config.js` file.

    1. In the `devServer` section, add the DNS alias you created earlier, for example `sdkapp.example.com` in a new `allowedHosts` property:

    ```
    devServer: {
        port: 8443,
        host: 'localhost',
        allowedHosts: 'sdkapp.example.com',
        …
    }
    ```

6. From the `/javascript` folder, run the central login OIDC sample app as follows:

    ```
    cd javascript
    npm run start:central-login-oidc
    ```

    Webpack compiles the code and serves it on the local IP address of your computer.

Try it out

To test that your custom UI app is acting as the centralized login pages, perform the following steps.

1. In a browser, open the local IP address of your computer, with the port number you edited earlier.

    For example, `https://sdkapp.example.com:8443`

2. As the custom UI sample app is running on a self-signed SSL certificate on your local IP address, your browser might display a warning message.

    You can ignore this warning for this tutorial:

    - In **Chrome**, click **Advanced** and then click **Proceed to sdkapp.example.com (unsafe)**.
    - In **Firefox**, click **Advanced** and then click **Accept the risk and continue**.
    - In **Safari**, click **Show Details** and then click **visit this website**.

    The sample OIDC login app displays **Login** and **Force Renew** buttons:

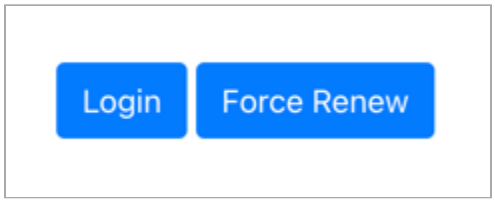
    

    Figure 1. JavaScript OIDC sample app showing Login and Force Renew buttons.

3. Click the **Login** button.

    The client app connects to the OAuth 2.0 client you created earlier, and is redirected to the custom UI sample running on your local computer.

The URL contains a `goto` parameter that contains the URL that the UI sample app redirects to after successful authentication. That URL will be the `authorize` endpoint of your authorization server.

4. Enter the credentials of a known user, and then click **Sign In**.

    1. If authentication is successful, the custom UI app redirects the browser to the `goto` URL, which points to your authorization server. The authorization server redirects back to your client app with the necessary `code` parameter.

    2. The sample client app uses the `code` parameter to contact the authorization server `access_token` endpoint to obtain the OAuth 2.0 access and ID tokens.

    3. Using the access token as a bearer token, the sample app calls the `userinfo` endpoint and displays them on the page:

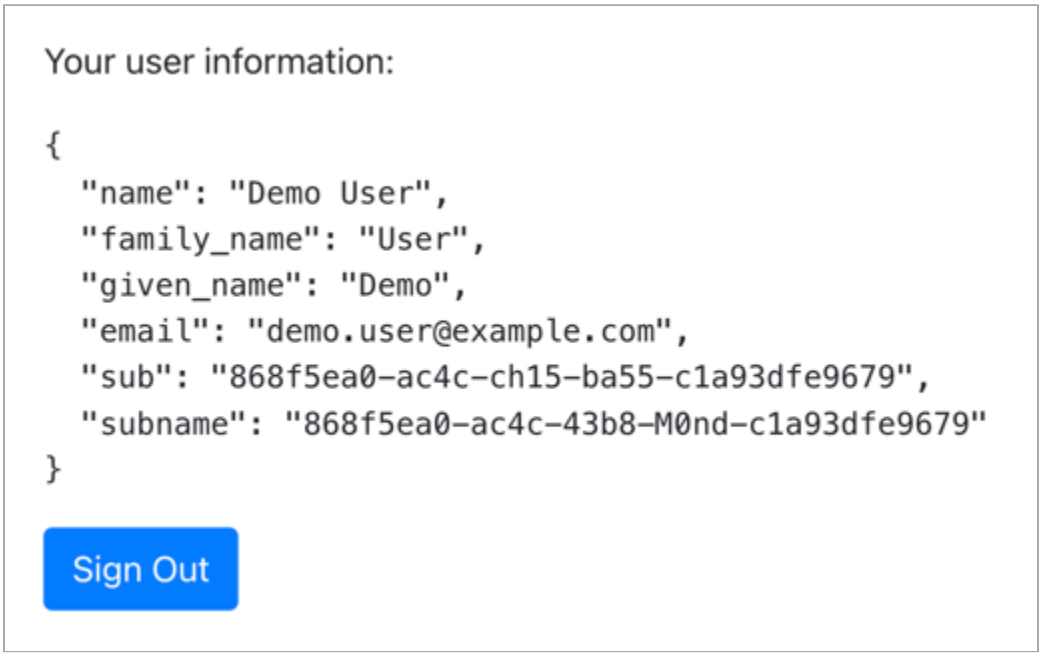


Figure 2. Userinfo of the authenticated user.

Running the Android sample OIDC client app

1. In **Android Studio**, open the `/android/kotlin-central-login-oidc` sample from the repo you downloaded earlier.

2. Edit the **app > kotlin+java >** `Config.kt` file to match your environment:

    1. In **discoveryEndpoint**, enter the `.well-known` URI of the realm in which you created the OAuth 2.0 client.

    You can view the `.well-known` endpoint for an OAuth 2.0 client in the PingOne Advanced Identity Cloud admin console:

        1. Log in to your PingAM administration console.

        2. Click **Applications**, and then select the OAuth 2.0 client you created earlier. For example, sdkPublicClient.

        3. On the **Sign On** tab, in the **Client Credentials** section, copy the **Discovery URI** value.

To form the `.well-known` URL for an PingAM server, concatenate the following information into a single URL:

1. The base URL of the PingAM component of your deployment, including the port number and deployment path.

   For example, `https://openam.example.com:8443/openam`

2. The string `/oauth2`

3. The hierarchy of the realm that contains the OAuth 2.0 client.

   You must specify the entire hierarchy of the realm, starting at the Top Level Realm. Prefix each realm in the hierarchy with the `realms/` keyword.

   For example, `/realms/root/realms/customers`

   > **Tip**
   >
   > If you omit the realm hierarchy, the top level `ROOT` realm is used by default.

4. The string `/.well-known/openid-configuration`

For example, `https://openam.example.com:8443/openam/oauth2/realms/root/.well-known/openid-configuration`

2. In **oauthClientId**, enter the client ID of the OAuth 2.0 client you configured earlier.

   For example, `sdkCustomUI`

3. In **oauthRedirectUri**, enter the redirect URI you configured in the OAuth 2.0 client.

   For example, `org.forgerock.demo://oauth2redirect`

4. In **cookieName**, enter the name of the cookie your authorization server uses to store session tokens on the client.

   For example, PingAM servers use `iPlanetDirectoryPro`

   To locate the cookie name in an PingOne Advanced Identity Cloud tenant:

   1. Navigate to **Tenant settings** > **Global Settings**

   2. Copy the value of the **Cookie** property.

5. In **oauthScope**, enter the list of scopes to request are present in the issued OAuth 2.0 token.

   For example, `openid profile email phone address`

The result will resemble the following:

```
data class PingConfig(
    var discoveryEndpoint: String = "https://openam-forgerock-sdks.forgeblocks.com/am/oauth2/realms/
alpha/.well-known/openid-configuration",
    var oauthClientId: String = "sdkCustomUI",
    var oauthRedirectUri: String = "org.forgerock.demo://oauth2redirect",
    var oauthSignOutRedirectUri: String = "",
    var cookieName: String = "ch15fefc5407912",
    var oauthScope: String = "openid profile email phone address"
)
```

3. Save your changes.

Try it out

To test that your custom UI app is acting as the centralized login pages, perform the following steps.

1. In Android Studio, on the **Run** menu, select **Run 'ping-oidc.app'**.

   Android Studio compiles and launches the app in either your connected device or a simulator, and displays the configuration you entered earlier:

Figure 3. Android sample app showing the configuration.

> **Tip**
>
> You can alter the values in the app if required.

2. Click **Centralized Login**.
3. As the custom UI sample app is running on a self-signed SSL certificate on your local IP address, your browser might display a warning message.

   You can ignore this warning for this tutorial:

   - In **Chrome**, click **Advanced** and then click **Proceed to 192.168.0.35 (unsafe)**.
   - In **Firefox**, click **Advanced** and then click **Accept the risk and continue**.

   The client app connects to the OAuth 2.0 client you created earlier, and is redirected to the custom UI sample running on your local computer.

   The URL contains a `goto` parameter that contains the URL that the UI sample app redirects to after successful authentication. That URL will be the `authorize` endpoint of your authorization server.
4. Enter the credentials of a known user, and then click **Sign In**.

   1. If authentication is successful, the custom UI app redirects the browser to the `goto` URL, which points to your authorization server. The authorization server redirects back to your client app with the necessary `code` parameter.
   2. The sample client app uses the `code` parameter to contact the authorization server `access_token` endpoint to obtain the OAuth 2.0 access and ID tokens:

Figure 4. Android OIDC sample app showing the access and ID tokens.

3. Tap the menu icon (☰), and then tap **User Profile**:

   The OIDC sample app uses the access token as a bearer token and calls the `userinfo` endpoint, displaying the result:

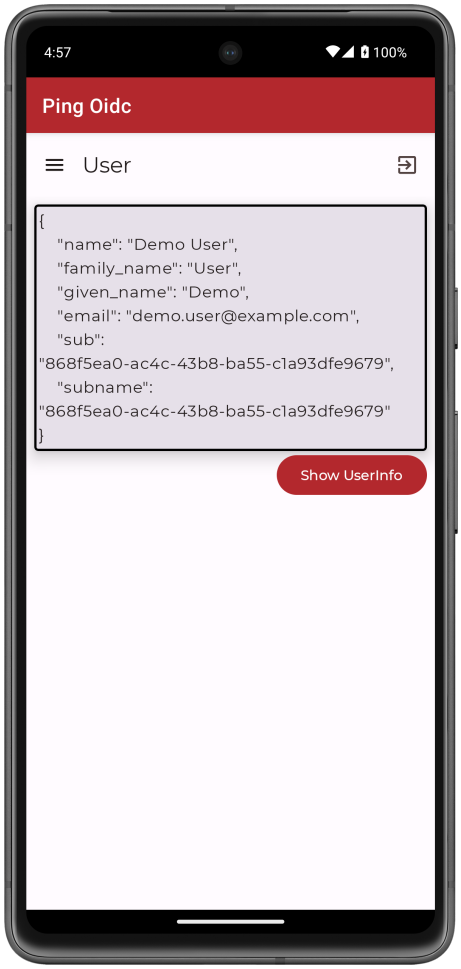


Figure 5. Userinfo of the authenticated user.

Running the iOS sample OIDC client app

1. In **Xcode**, in the `sdk-sample-apps` folder you cloned in the previous step, open the `iOS` > `swiftui-oidc` > `PingExample` > `PingExample.xcodeproj` file.
2. Locate the `ConfigurationViewModel` function which contains placeholder configuration properties.

> **Tip**
>
> The function is commented with `//TODO:` in the source to make it easier to locate.

```
return ConfigurationViewModel(
    clientId: "[CLIENT ID]",
    scopes: ["openid", "email", "address", "phone", "profile"],
    redirectUri: "[REDIRECT URI]",
    signOutUri: "[SIGN OUT URI]",
    discoveryEndpoint: "[DISCOVERY ENDPOINT URL]",
    environment: "[ENVIRONMENT - EITHER AIC OR PingOne]",
    cookieName: "[COOKIE NAME - OPTIONAL (Applicable for AIC only)]",
    browserSeletorType: .authSession
)
```

1. In **clientId**, enter the client ID of the OAuth 2.0 client you configured earlier.

   For example, `sdkCustomUI`

2. In **scopes**, enter an array of scopes to request are present in the issued OAuth 2.0 token.

   For example, `["openid","profile","email","address","phone"]`

3. In **redirectUri** and **signOutUri**, enter the redirect URI you configured in the OAuth 2.0 client.

   For example, `org.forgerock.demo://oauth2redirect`

4. In **discoveryEndpoint**, enter the `.well-known` URI of the realm in which you created the OAuth 2.0 client.

   You can view the `.well-known` endpoint for an OAuth 2.0 client in the PingOne Advanced Identity Cloud admin console:

   1. Log in to your PingAM administration console.
   2. Click **Applications**, and then select the OAuth 2.0 client you created earlier. For example, sdkPublicClient.
   3. On the **Sign On** tab, in the **Client Credentials** section, copy the **Discovery URI** value.

   To form the `.well-known` URL for an PingAM server, concatenate the following information into a single URL:

   1. The base URL of the PingAM component of your deployment, including the port number and deployment path.

      For example, `https://openam.example.com:8443/openam`

   2. The string `/oauth2`
   3. The hierarchy of the realm that contains the OAuth 2.0 client.

      You must specify the entire hierarchy of the realm, starting at the Top Level Realm. Prefix each realm in the hierarchy with the `realms/` keyword.

      For example, `/realms/root/realms/customers`

      > **Tip**
      >
      > If you omit the realm hierarchy, the top level `ROOT` realm is used by default.

   4. The string `/.well-known/openid-configuration`

For example, `https://openam.example.com:8443/openam/oauth2/realms/root/.well-known/openid-configuration`

5. In **environment**, enter `AIC`.
6. In **cookieName**, enter the name of the cookie your authorization server uses to store session tokens on the client.

   For example, PingAM servers use `iPlanetDirectoryPro`

   To locate the cookie name in an PingOne Advanced Identity Cloud tenant:

   1. Navigate to **Tenant settings** > **Global Settings**
   2. Copy the value of the **Cookie** property.
7. In **browserSeletorType**, enter `.authSession`.

   You can specify what type of browser the client iOS device opens to handle centralized login.

   Each browser has slightly different characteristics, which make them suitable to different scenarios, as outlined in this table:

Browser type	Characteristics
`.authSession`	Opens a web authentication session browser. Designed specifically for authentication sessions, however it prompts the user before opening the browser with a modal that asks them to confirm the domain is allowed to authenticate them. This is the default option in the Ping SDK for iOS.
`.ephemeralAuthSession`	Opens a web authentication session browser, but enables the `prefersEphemeralWebBrowserSession` parameter. This browser type *does not* prompt the user before opening the browser with a modal. The difference between this and `.authSession` is that the browser does not include any existing data such as cookies in the request, and also discards any data obtained during the browser session, including any session tokens. When is `ephemeralAuthSession` suitable: ■ ✕ `ephemeralAuthSession` is *not* suitable when you require single sign-on (SSO) between your iOS apps, as the browser will not maintain session tokens. ■ ✕ `ephemeralAuthSession` is *not* suitable when you require a session token to log a user out of the server, for example for logging out of PingOne, as the browser will not maintain session tokens. ■ ✓ Use `ephemeralAuthSession` when you do not want the user's existing sessions to affect the authentication.

Browser type	Characteristics
`.nativeBrowserApp`	Opens the installed browser that is marked as the default by the user. Often Safari. The browser opens without any interaction from the user. However, the browser does display a modal when returning to your application.
`.sfViewController`	Opens a Safari view controller browser. Your client app is ***not*** able to interact with the pages in the `sfViewController` or access the data or browsing history. The view controller opens within your app without any interaction from the user. As the user does not leave your app, the view controller does not need to display a warning modal when authentication is complete and control returns to your application.

The result resembles the following:

```
return ConfigurationViewModel(
    clientId: "sdkCustomUI",
    scopes: ["openid", "email", "address", "phone", "profile"],
    redirectUri: "org.forgerock.demo://oauth2redirect",
    signOutUri: "org.forgerock.demo://oauth2redirect",
    discoveryEndpoint: "https://openam-forgerock-sdks.forgeblocks.com/am/oauth2/realms/alpha/.well-known/
openid-configuration",
    environment: "AIC",
    cookieName: "ch15fefc5407912",
    browserSeletorType: .authSession
)
```

3. Save your changes.

Try it out

To test that your custom UI app is acting as the centralized login pages, perform the following steps.

1. In Xcode, select **Product > Run**.

   Xcode launches the sample app in the iPhone simulator.

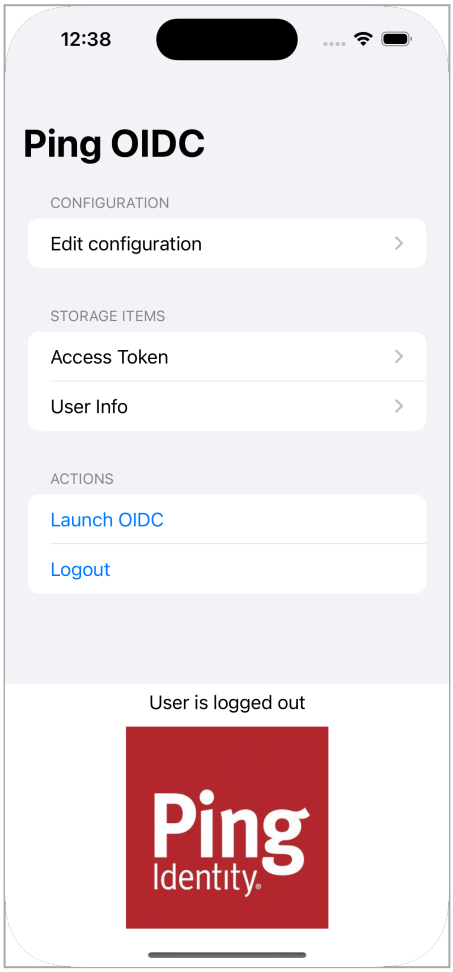


Figure 6. iOS OIDC login sample home screen

Important

Third-party cookies in the Safari browser

Safari blocks third-party cookies by default, so the custom UI app will not be able to authenticate calls to the `/authorize` endpoint if it is running on a different domain than the server.

In production, ensure your custom UI app and server share the same domain.

For this tutorial disable third-party cookie checks in Safari as follows:

1. In the iPhone simulator, in the toolbar click **Home**. On the screen, use your mouse to swipe to the right, and then tap **Settings**.
2. Tap **Safari**, scroll to the **Privacy & Security** section, and disable **Prevent Cross-Site Tracking**.

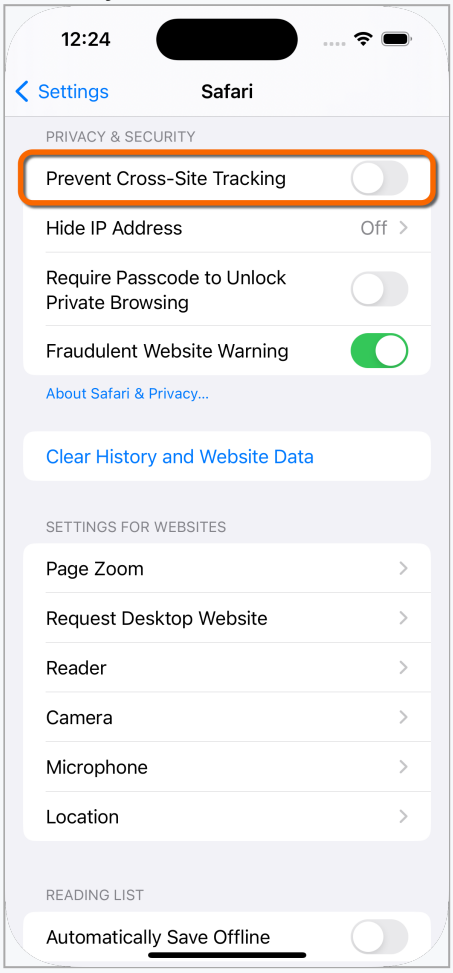


Figure 7. Allow third-party cookies in Safari

3. To return to the sample app, in the toolbar double-click **Home** to show the open apps, and then tap the **PingOIDC** app.

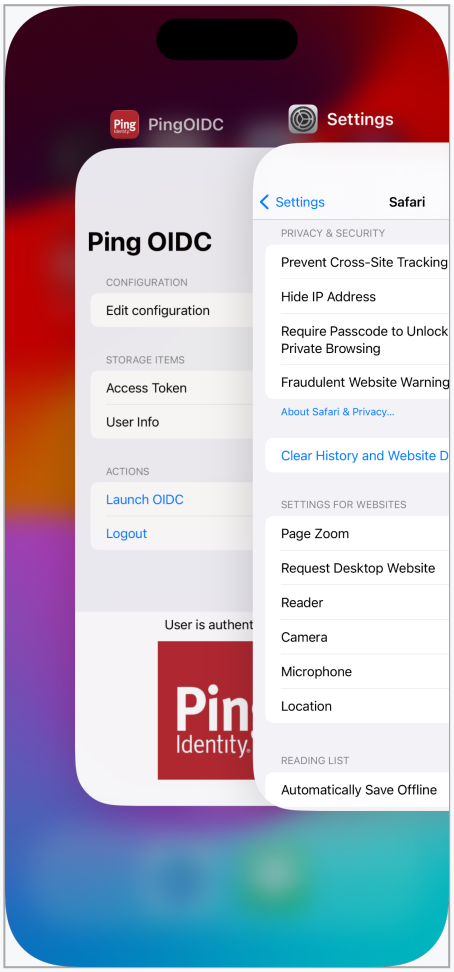


Figure 8. Return to the OIDC sample app by double-clicking the home icon.

2. In the sample app on the iPhone simulator, tap **Edit configuration**, and verify or edit the configuration you entered in the previous step.

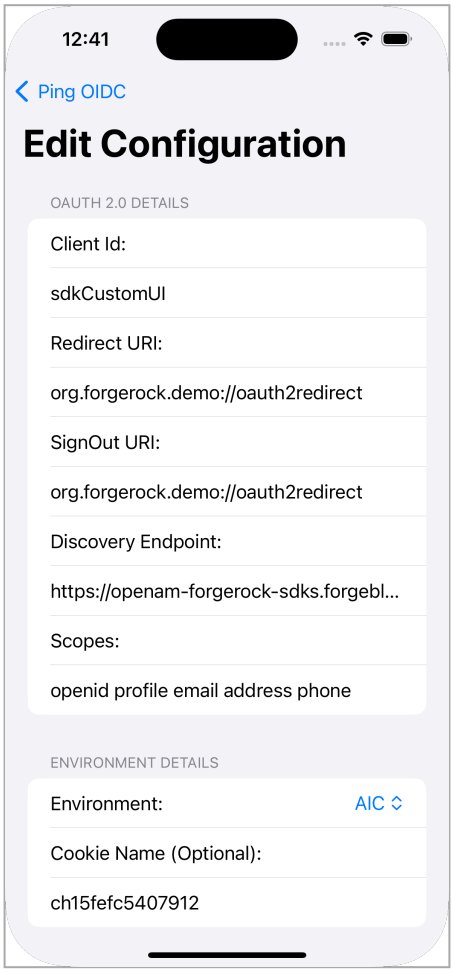


Figure 9. Verify the configuration settings

3. Tap < **Ping OIDC** to go back to the main menu, and then tap **Launch OIDC**.

> **Note**
>
> You might see a dialog asking if you want to open a browser. If you do, tap **Continue**.

The app launches a web browser and connects to your authorization server, which then redirects to the custom UI app running locally on your computer:

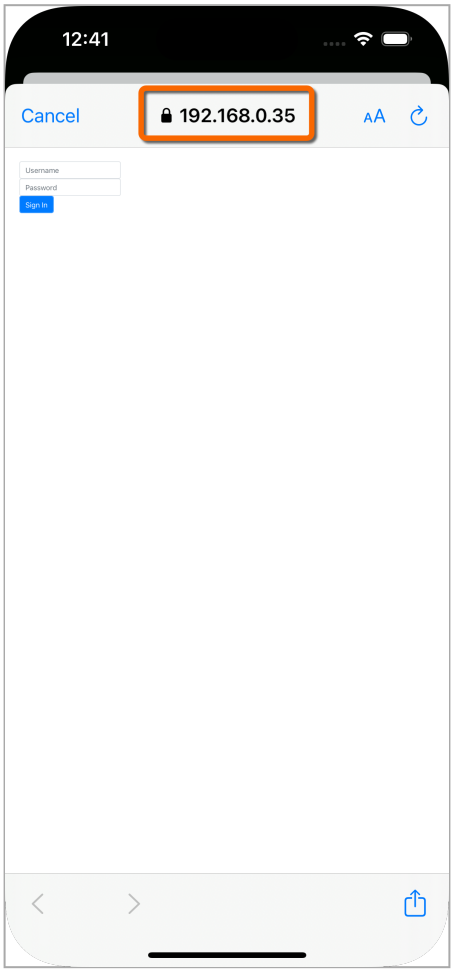


Figure 10. Browser launched and redirected to local custom UI app.

4. Sign on as a demo user:
    - **Name**: `demo`
    - **Password**: `Ch4ng3it!`

If authentication is successful, the application displays the access token issued by PingAM.

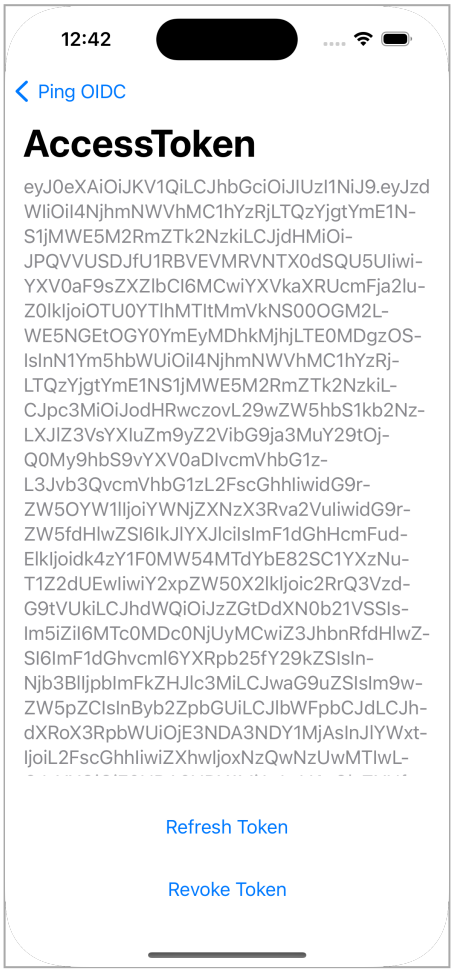


Figure 11. Access token after successful authentication

5. Tap < **Ping OIDC** to go back to the main menu, and then tap **User Info**.

   The app displays the information relating to the access token:

Figure 12. User info relating to the access token

6. Tap < **Ping OIDC** to go back to the main menu, and then tap **Logout**.

   The app logs the user out of the authorization server and prints a message to the Xcode console:

```
[FRCore][4.8.0] [🌐 - Network] Response | [☑ 204] :
  https://openam.example.com:443/am/oauth2/connect/endSession?
id_token_hint=eyJ0...sbrA&client_id=sdkPublicClient in 34 ms
[FRAuth][4.8.0] [FRUser.swift:211 : logout()] [Verbose]
  Invalidating OIDC Session successful
```

Ping (ForgeRock) Login Widget

Server support:

- ✕ PingOne
- ✓ PingOne Advanced Identity Cloud
- ✓ PingAM
- ✕ PingFederate

SDK support:

- ✕ Ping SDK for Android
- ✕ Ping SDK for iOS
- ✓ Ping SDK for JavaScript

The Ping (ForgeRock) Login Widget for PingOne Advanced Identity Cloud and PingAM is an all-inclusive UI component to help you add authentication, user registration, and other self-service journeys into your web applications.

You can use the Ping (ForgeRock) Login Widget within React, Vue, Angular and a number of other modern JavaScript frameworks, as well as vanilla JavaScript.

It does not currently support server-side rendering (SSR), including Node.js.

The Ping (ForgeRock) Login Widget uses the Ping SDK for JavaScript internally, and adds a user interface and state management. This rendering layer helps eliminate the need to develop and maintain the UI components for providing complex authentication experiences.

This rendering layer uses Svelte and Tailwind, but these are "compiled away" resulting in no runtime dependencies.

The resulting Ping (ForgeRock) Login Widget is both library- and framework-agnostic.

Topics

Get started with the Ping (ForgeRock) Login Widget in the following sections:

Tutorial

Learn how to install the Ping (ForgeRock) Login Widget, add it to your applications and manage user authentication and self-service journeys.

Themes

Discover how to reconfigure the Ping (ForgeRock) Login Widget to use different colors, fonts, or sizing, or select between the light and dark modes.

Use cases

Find out how to achieve some common use case scenarios using the Ping (ForgeRock) Login Widget.

Integrations

Integrate the Ping (ForgeRock) Login Widget into various different frameworks.

API

Access a list of the modules included in the Ping (ForgeRock) Login Widget and the API they offer.

Functionality

The Ping (ForgeRock) Login Widget supports the following PingOne Advanced Identity Cloud and PingAM features:

✓ Supported	✕ Unsupported

• Page node
• Username
• Password
• QR codes
• Push authentication and registration
• One-time passwords and registration
• WebAuthn
• Device profiles
• Social login providers:
  ◦ Apple
  ◦ Facebook
  ◦ Google
• Email suspend, or "magic links"
• CAPTCHA display
  ◦ hCaptcha
  ◦ reCAPTCHA v2
  ◦ reCAPTCHA v3
• PingOne Protect
	• Centralized login
	• `TextOutputCallback` callbacks containing scripts
	• SAML federation

Requirements

The Ping (ForgeRock) Login Widget is designed to work with the following:

- An ECMAScript module or CommonJS enabled client-side JavaScript app
- A "modern", fully-featured browser such as Chrome, Firefox, Safari, or Chromium Edge

The Ping (ForgeRock) Login Widget supports vanilla JavaScript and many frameworks. It is tested against the following:

✓ Tested	✕ Unsupported
• Angular • React • Vue • Svelte	• Server-side rendering (SSR), including Node.js

The Ping (ForgeRock) Login Widget is ***not designed or tested*** for use with the following:

- Internet Explorer
- Legacy Edge
- WebView
- Electron

- Modified, browser-like environments

Tutorial

This tutorial guides you through adding the Ping (ForgeRock) Login Widget to your application in the *modal* or *inline* form factor.

Prerequisites

You need to set up your PingOne Advanced Identity Cloud or PingAM instance with an authentication journey, and a demo user. To obtain access tokens, you also need to create an OAuth 2.0 client.

You may need to edit the CORS configuration on your server.

Server configuration

This tutorial requires you to configure one of the following servers:

PingOne Advanced Identity Cloud

PingOne Advanced Identity Cloud

PingAM

PingAM

PingOne Advanced Identity Cloud

Cross-origin resource sharing (CORS) lets user agents make cross-domain server requests. In PingOne Advanced Identity Cloud, you can configure CORS to allow browsers from trusted domains to access PingOne Advanced Identity Cloud protected resources. For example, you might want a custom web application running on your own domain to get an end-user's profile information using the PingOne Advanced Identity Cloud REST API.

The Ping SDK for JavaScript samples and tutorials use `https://localhost:8443` as the host domain, which you should add to your CORS configuration.

If you are using a different domain for hosting SDK applications, ensure you add them to the CORS configuration as accepted origin domains.

To update the CORS configuration in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. At the top right of the screen, click your name, and then select **Tenant settings**.
3. On the **Global Settings** tab, click **Cross-Origin Resource Sharing (CORS)**.
4. Perform one of the following actions:
    - If available, click **ForgeRockSDK**.
    - If you haven't added any CORS configurations to the tenant, click **+ Add a CORS Configuration**, select **Ping SDK**, and then click **Next**.
5. Add `https://localhost:8443` and any DNS aliases you use to host your Ping SDK for JavaScript applications to the **Accepted Origins** property.
6. Complete the remaining fields to suit your environment.

This documentation assumes the following configuration, required for the tutorials and sample applications:

Property	Values
`Accepted Origins`	`https://localhost:8443`
`Accepted Methods`	`GET` `POST`

`Accepted Headers`	`accept-api-version`
	`x-requested-with`
	`content-type`
	`authorization`
	`if-match`
	`x-requested-platform`
	`iPlanetDirectoryPro` [1]
	`ch15fefc5407912` [2]

`Exposed Headers`	`authorization` `content-type`
`Enable Caching`	`True`
`Max Age`	`600`
`Allow Credentials`	`True`

> **Tip**
>
> Click **Show advanced settings** to be able to edit all available fields.

7. Click **Save CORS Configuration**.

The samples and tutorials in this documentation often require that you have an identity set up so that you can test authentication.

To create a demo user in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. In the left panel, click **Identities** > **Manage**.
3. Click **+ New Alpha realm - User**.
4. Enter the following details:
    - **Username** = `demo`
    - **First Name** = `Demo`
    - **Last Name** = `User`
    - **Email Address** = `demo.user@example.com`

- **Password** = `Ch4ng3it!`

5. Click **Save**.

Authentication journeys provide fine-grained authentication by allowing multiple paths and decision points throughout the flow. Authentication journeys are made up of nodes that define actions taken during authentication.

Each node performs a single task, such as collecting a username or making a simple decision. Nodes can have multiple outcomes rather than just success or failure. For details, see the Authentication nodes configuration reference in the PingAM documentation.

To create a simple journey for use when testing the Ping SDKs, follow these steps:

1. In your PingOne Advanced Identity Cloud tenant, navigate to **Journeys**, and click **+ New Journey**.
2. Enter a name, such as `sdkUsernamePasswordJourney` and click **Save**.

   The authentication journey designer appears.
3. Drag the following nodes into the designer area:
   - **Page Node**
   - **Platform Username**
   - **Platform Password**
   - **Data Store Decision**
4. Drag and drop the **Platform Username** and **Platform Password** nodes onto the **Page Node**, so that they both appear on the same page when logging in.
5. Connect the nodes as follows:

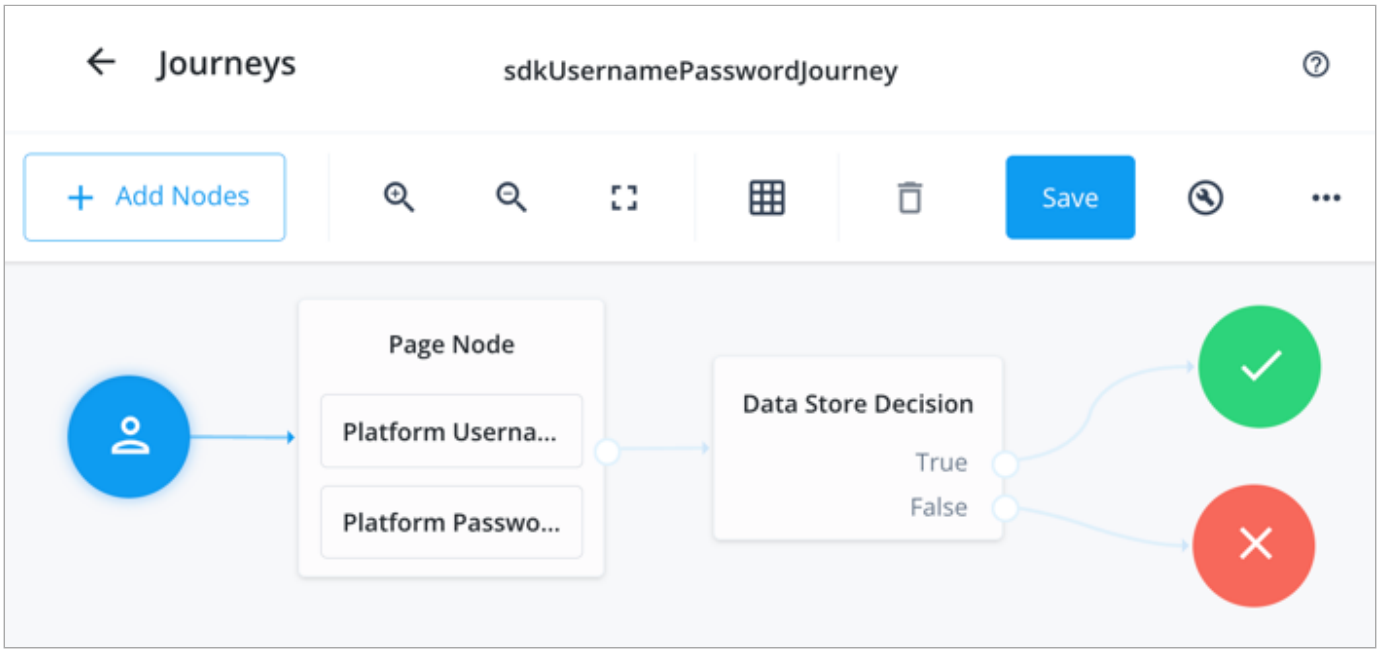


Figure 1. Example username and password authentication journey

6. Click **Save**.

Public clients do not use a client secret to obtain tokens because they are unable to keep them hidden. The Ping SDKs commonly use this type of client to obtain tokens, as they cannot guarantee safekeeping of the client credentials in a browser or on a mobile device.

To register a *public* OAuth 2.0 client application for use with the SDKs in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. In the left panel, click **Applications**.
3. Click **+ Custom Application**.
4. Select **OIDC - OpenId Connect** as the sign-in method, and then click **Next**.
5. Select **Native / SPA** as the application type, and then click **Next**.
6. In **Name**, enter a name for the application, such as `Public SDK Client`.
7. In **Owners**, select a user that is responsible for maintaining the application, and then click **Next**.

   > **Tip**
   >
   > When trying out the SDKs, you could select the `demo` user you created previously.

8. In **Client ID**, enter `sdkPublicClient`, and then click **Create Application**.

   PingOne Advanced Identity Cloud creates the application and displays the details screen.
9. On the **Sign On** tab:
   1. In **Sign-In URLs**, enter the following values:

      > **Important**
      >
      > Also add any other domains where you host SDK applications.

   2. In **Grant Types**, enter the following values:

      `Authorization Code`

      `Refresh Token`

   3. In **Scopes**, enter the following values:

      `openid profile email address`
10. Click Show advanced settings, and on the **Authentication** tab:
    1. In **Token Endpoint Authentication Method**, select `none`.
    2. In **Client Type**, select `Public`.
    3. Enable the **Implied Consent** property.
11. Click **Save**.

The application is now configured to accept client connections from and issue OAuth 2.0 tokens to the example applications and tutorials covered by this documentation.

The provider specifies the supported OAuth 2.0 configuration options for a realm.

To ensure the PingOne Advanced Identity Cloud OAuth 2.0 provider service is configured for use with the Ping SDKs, follow these steps:

1. In your PingOne Advanced Identity Cloud tenant, navigate to **Native Consoles** > **Access Management**.
2. In the left panel, click **Services**.
3. In the list of services, click **OAuth2 Provider**.
4. On the **Core** tab, ensure **Issue Refresh Tokens** is enabled.
5. On the **Consent** tab, ensure **Allow Clients to Skip Consent** is enabled.
6. Click **Save Changes**.

PingAM

Cross-origin resource sharing (CORS) lets user agents make cross-domain server requests. In PingAM, you can configure CORS to allow browsers from trusted domains to access PingAM protected resources. For example, you might want a custom web application running on your own domain to get an end-user's profile information using the PingAM REST API.

The Ping SDK for JavaScript samples and tutorials all use `https://localhost:8443` as the host domain, which you should add to your CORS configuration.

If you are using a different URL for hosting SDK applications, ensure you add them to the CORS configuration as accepted origin domains.

To enable CORS in PingAM, and create a CORS filter to allow requests from your configured domain names, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. Navigate to **Configure** > **Global Services** > **CORS Service** > **Configuration**, and set the **Enable the CORS filter** property to `true`.

   > **Important**
   >
   > If this property is not enabled, CORS headers are not added to responses from PingAM, and CORS is disabled entirely.

3. On the **Secondary Configurations** tab, click **Click Add a Secondary Configuration**.
4. In the **Name** field, enter `ForgeRockSDK`.
5. in the **Accepted Origins** field, enter any DNS aliases you use for your SDK apps.

   This documentation assumes the following configuration:

Property	Values
`Accepted Origins`	`https://localhost:8443`

Property	Values
`Accepted Methods`	`GET` `POST`

`Accepted Headers`	`accept-api-version`
	`x-requested-with`
	`content-type`
	`authorization`
	`if-match`
	`x-requested-platform`
	`iPlanetDirectoryPro` [1]
	`ch15fefc5407912` [2]

`Exposed Headers`	`authorization` `content-type`

6. Click **Create**.

   PingAM displays the configuration of your new CORS filter.

7. On the CORS filter configuration page:

   1. Ensure **Enable the CORS filter** is enabled.
   2. Set the **Max Age** property to `600`
   3. Ensure **Allow Credentials** is enabled.

8. Click **Save Changes**.

The samples and tutorials in this documentation often require that you have an identity set up so that you can test authentication.

To create a demo user in PingAM, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. Navigate to **Identities**, and then click **+ Add Identity**.
3. Enter the following details:
   - **User ID** = `demo`
   - **Password** = `Ch4ng3it!`
   - **Email Address** = `demo.user@example.com`
4. Click **Create**.

Authentication trees provide fine-grained authentication by allowing multiple paths and decision points throughout the authentication flow. Authentication trees are made up of nodes that define actions taken during authentication.

Each node performs a single task, such as collecting a username or making a simple decision. Nodes can have multiple outcomes rather than just success or failure. For details, see the Authentication nodes configuration reference in the PingAM documentation.

To create a simple tree for use when testing the Ping SDKs, follow these steps:

1. Under **Realm Overview**, click **Authentication Trees**, then click **Create Tree**.
2. Enter a tree name, for example `sdkUsernamePasswordJourney`, and then click **Create**.

   The authentication tree designer appears, showing the **Start** entry point connected to the **Failure** exit point.
3. Drag the following nodes from the **Components** panel on the left side into the designer area:
   - **Page Node**
   - **Username Collector**
   - **Password Collector**
   - **Data Store Decision**
4. Drag and drop the **Username Collector** and **Password Collector** nodes onto the **Page Node**, so that they both appear on the same page when logging in.
5. Connect the nodes as follows:

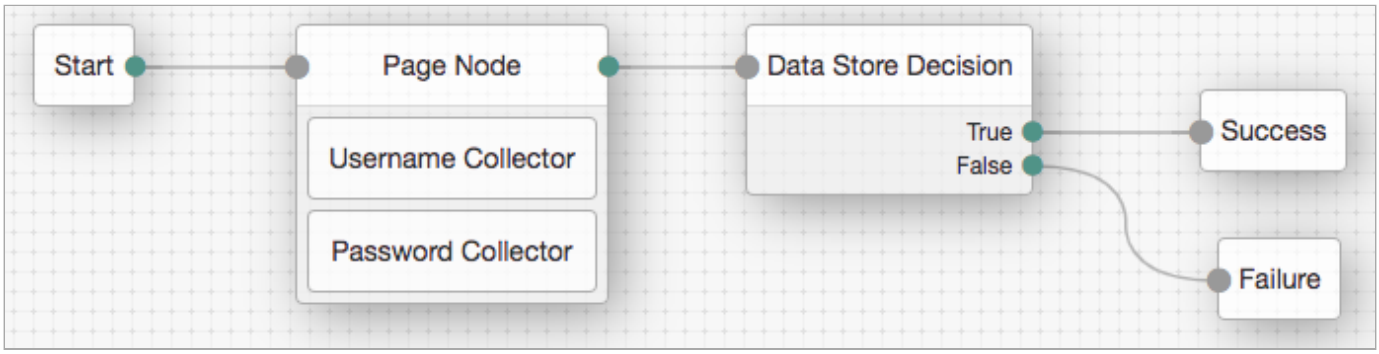


Figure 2. Example username and password authentication tree

6. Select the **Page Node**, and in the **Properties** pane, set the **Stage** property to `UsernamePassword`.

   > **Tip**
   >
   > You can configure the node properties by selecting a node and altering properties in the right-hand panel.

   One of the samples uses this specific value to determine the custom UI to display.
7. Click **Save**.

Public clients do not use a client secret to obtain tokens because they are unable to keep them hidden. The Ping SDKs commonly use this type of client to obtain tokens, as they cannot guarantee safekeeping of the client credentials in a browser or on a mobile device.

To register a *public* OAuth 2.0 client application for use with the SDKs in AM, follow these steps:

1. Log in to the PingAM admin UI as an administrator.

2. Navigate to **Applications** > **OAuth 2.0** > **Clients**, and then click **+ Add Client**.
3. In **Client ID**, enter `sdkPublicClient`.
4. Leave **Client secret** empty.
5. In **Redirection URIs**, enter the following values:

   > **Important**
   >
   > Also add any other domains where you will be hosting SDK applications.

6. In **Scopes**, enter the following values:

   `openid profile email address`

7. Click **Create**.

   PingAM creates the new OAuth 2.0 client, and displays the properties for further configuration.

8. On the **Core** tab:
   1. In **Client type**, select `Public`.
   2. Disable **Allow wildcard ports in redirect URIs**.
   3. Click **Save Changes**.
9. On the **Advanced** tab:
   1. In **Grant Types**, enter the following values:

      ```
      Authorization Code
      Refresh Token
      ```

   2. In **Token Endpoint Authentication Method**, select `None`.
   3. Enable the **Implied consent** property.
10. Click **Save Changes**.

The provider specifies the supported OAuth 2.0 configuration options for a realm.

To ensure the PingAM OAuth 2.0 provider service is configured for use with the Ping SDKs, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. In the left panel, click **Services**.
3. In the list of services, click **OAuth2 Provider**.
4. On the **Core** tab, ensure **Issue Refresh Tokens** is enabled.
5. On the **Consent** tab, ensure **Allow Clients to Skip Consent** is enabled.
6. Click **Save Changes**.

Steps

Step 1. Install the widget

In this step, you use `npm` to add the Ping (ForgeRock) Login Widget to your project. It also covers how to download and build the Ping (ForgeRock) Login Widget to support custom requirements.

Step 2. Configure the CSS

In this step, you add the default CSS to your app, and learn how to use layers to control the CSS cascade.

Step 3. Import the widget

In this step, you import the modules from the Ping (ForgeRock) Login Widget you want to use in your app.

Step 4. Configure the SDK

In this step, you provide the configuration necessary for the Ping (ForgeRock) Login Widget to contact your server, such as which realm to use, and the server URL.

Step 5. Instantiate the widget

In this step, you choose where in your app to mount the Ping (ForgeRock) Login Widget, and then instantiate an instance, choosing either the inline or modal form factor.

Step 6. Start a journey

In ths step, you start a journey so that the Ping (ForgeRock) Login Widget can display the UI for the first callback.

Step 7. Subscribe to events

In this step, you subscribe to *observables* to capture and react to events that occur during use of the Ping (ForgeRock) Login Widget.

---

1. Cookie name value in PingAM servers.
2. In PingOne Advanced Identity Cloud tenants, go to **Tenant Settings > Global Settings > Cookie** to find this dynamic cookie name value.

Step 1. Install the widget

You can add the Ping (ForgeRock) Login Widget to your app by using Node Package Manager (npm), or you can download it from GitHub and build it yourself, adding results to your project directly.

Install the Ping (ForgeRock) Login Widget with npm

The easiest way to add the Ping (ForgeRock) Login Widget to your project.

Build a customized Ping (ForgeRock) Login Widget

If you want to customize the themes included in the Ping (ForgeRock) Login Widget, you need to download the Ping (ForgeRock) Web Login Framework source, make your modifications, and build a customized package.

Install the Ping (ForgeRock) Login Widget with npm

Add the Ping (ForgeRock) Login Widget to your project using npm as follows:

```
npm install @forgerock/login-widget
```

Next, you can Step 2. Configure the CSS.

Build a customized Ping (ForgeRock) Login Widget

The following steps show how to download the Ping (ForgeRock) Web Login Framework and build the Ping (ForgeRock) Login Widget:

1. Download the Ping (ForgeRock) Web Login Framework from the Git repository:

   ```
   git clone https://github.com/ForgeRock/forgerock-web-login-framework.git
   ```

2. In a terminal window, navigate to the root of the Ping (ForgeRock) Web Login Framework:

   ```
   cd forgerock-web-login-framework
   ```

3. Run `npm` to download and install the required packages and modules:

   ```
   npm install
   ```

4. Build the Ping (ForgeRock) Login Widget with `npm`:

   ```
   npm run build:widget
   ```

5. Copy the built `package/` directory into your app project

6. Import the `Widget` component into your app:

   ```
   import Widget from '../path/to/package/index.js';
   ```

   > **Note**
   >
   > The exact syntax for importing the widget into your app varies depending on the technologies your app uses.

Next

Next, you can Step 2. Configure the CSS.

Step 2. Configure the CSS

You can use any of the following methods to add the default Ping (ForgeRock) Login Widget styles to your app:

1. Import it into your JavaScript project as a module.

2. Import it using a CSS preprocessor, like Sass, Less, or PostCSS.

If you decide to import the CSS into your JavaScript, make sure your bundler is able to import and process the CSS as a module. If using a CSS preprocessor, configure your preprocessor to access files from within your package or from a directory.

Examples

Import the CSS into your JavaScript:

npm

```
// app.js
import '@forgerock/login-widget/widget.css';
```

Local

```
// app.js
import '../path/to/widget.css';
```

Import the CSS into your CSS:

npm

```
/* style.css */
@import '@forgerock/login-widget/widget.css';
```

Local

```
/* style.css */
@import '../path/to/widget.css';
```

Controlling the CSS cascade

How the browser applies styles to an app can depend on the order you import or declare CSS into it, referred to as the *cascade*.

You can use the `@layer` CSS rule to declare a cascade layer, ensuring Ping (ForgeRock) Login Widget styles apply separately from your own.

For more information, refer to `@layer` in the MDN docs.

> **Note**
>
> The Ping (ForgeRock) Login Widget styles will not overwrite any of your CSS.
> They are namespaced to help prevent collisions and use a CSS selector prefix of `tw_` .

To create a cascade layer for the Ping (ForgeRock) Login Widget styles:

1. Wrap your existing styles in a new layer, for example, "app":

```
@layer app {
    /* Your app's existing CSS */
}
```

2. Declare the order of layers in your index HTML file before loading any CSS.

> **Note**
>
> The Widget has multiple `@layer` declarations in its CSS files

```
<style type="text/css">
  /* Your existing "app" CSS layer first */
  @layer app;

  /* List the Widget layers after your own styles */
  @layer 'fr-widget.base';
  @layer 'fr-widget.utilities';
  @layer 'fr-widget.components';
  @layer 'fr-widget.variants';
</style>
```

> **Note**
>
> The CSS imported for the Widget will not overwrite any of your app's CSS. It's all namespaced to ensure there are no collisions. To help achieve this, the Ping (ForgeRock) Login Widget uses a selector naming convention with a `tw_` prefix.

Next

Next, you can Step 3. Import the widget

Step 3. Import the widget

To use the Ping (ForgeRock) Login Widget, import the modules you want to use into your app:

```
// Import the Login Widget
import Widget, { configuration } from '@forgerock/login-widget';
```

The exact syntax for importing the widget depends on the module system you are using.

The Ping (ForgeRock) Login Widget exports a number of different modules, each providing different functionality.

Ping (ForgeRock) Login Widget modules

Module	Description	API reference
`Widget`	Use this main class to instantiate the Ping (ForgeRock) Login Widget, mount it into the DOM, and set up event listeners.	Widget API reference
`configuration`	Use this module to configure the Ping (ForgeRock) Login Widget. You can configure the settings it needs to contact the authorization server, styles, layout, and override content.	Configuration API reference
`journey`	Use this module to configure and start an authentication journey.	Journey API reference
`component`	Use this module to subscribe to events triggered by the Ping (ForgeRock) Login Widget and for controlling the modal form factor.	Component API reference
`user`	Use this module for managing users in the Ping (ForgeRock) Login Widget, such as obtaining user or token information, and logging users out.	User API reference

Next

Next, you can Step 4. Configure the SDK.

Step 4. Configure the SDK

The Ping (ForgeRock) Login Widget requires information about the server instance it connects to, as well as OAuth 2.0 client configuration and other settings.

To provide these settings, import and use the `configuration` module and its `set()` method.

The Ping (ForgeRock) Login Widget uses the same underlying configuration properties as the main SDK. Add your configuration under the `forgerock` property:

Example Ping (ForgeRock) Login Widget configuration

```
// Import the modules
import Widget, { configuration } from '@forgerock/login-widget';

// Create a configuration instance
const myConfig = configuration();

// Set the configuration properties
myConfig.set({
  forgerock: {
    // Minimum required configuration:
    serverConfig: {
        baseUrl: 'https://openam-forgerock-sdks.forgeblocks.com/am',
        timeout: 3000,
    },
    // Optional configuration:
    clientId: 'sdkPublicClient', // The default is `WebLoginWidgetClient`
    realmPath: 'alpha',  // This is the default if not specified
    redirectUri: window.location.href,  // This is the default if not specified
    scope: 'openid profile email address', // The default is `openid profile` if not specified
  },
});
```

Tip

Set your Ping (ForgeRock) Login Widget configuration at the top level of your application, such as its `index.js` or `app.js` file.
This ensures the Ping (ForgeRock) Login Widget has the configuration needed to call out to your PingOne Advanced Identity Cloud or PingAM server whenever and wherever you use its APIs in your app.
For example, you must set the configuration before starting a journey with `journeyEvents.start()` or calling either `userEvents.get()` or `tokenEvents.get()`.

SDK configuration properties

The configuration properties available in both the SDK and the Ping (ForgeRock) Login Widget are as follows:

Server

Properties

Property	Description
`serverConfig`	An interface for configuring how the SDK contacts the PingAM instance. Contains `baseUrl` and `timeout`.

Property	Description

`serverConfig: {baseUrl}`	The base URL of the server to connect to, including port and deployment path.
	Identity Cloud example:
	`https://openam-forgerock-sdks.forgeblocks.com/am`
	Self-hosted example:
	`https://openam.example.com:8443/openam`

`serverConfig: {wellknown}`	A URL to the server's `.well-known/openid-configuration` endpoint.
	Use the `Config.setAsync()` method to set SDK configuration using values derived from those provided at the URL.
	Example:
	`https://openam-forgerock-sdks.forgeblocks.com/am/oauth2/realms/root/realms/alpha/.well-known/openid-configuration`
	Self-hosted example:
	`https://openam.example.com:8443/openam/oauth2/realms/root/.well-known/openid-configuration`

`serverConfig: {timeout}`	A timeout, in milliseconds, for each request that communicates with your server. For example, for 30 seconds specify `30000`. Defaults to `5000` (5 seconds).
`realmPath`	The realm in which the OAuth 2.0 client profile and authentication journeys are configured. For example, `alpha`. Defaults to the self-hosted top-level realm `root`.
`tree`	The name of the user ***authentication*** tree configured in your server. For example, `sdkUsernamePasswordJourney`.

OAuth 2.0

Properties

Property	Description
`clientId`	The `client_id` of the OAuth 2.0 client profile to use.

Property	Description
`redirectUri`	The `redirect_uri` as configured in the OAuth 2.0 client profile. **Tip** The Ping SDK for JavaScript attempts to load the redirect page to capture the OAuth 2.0 `code` and `state` query parameters that the server appended to the redirect URL. If the page you redirect to does not exist, takes a long time to load, or runs any JavaScript you might get a timeout, delayed authentication, or unexpected errors. To ensure the best user experience, we ***highly recommend*** that you redirect to a static HTML page with minimal HTML and no JavaScript when obtaining OAuth 2.0 tokens. For example, `https://localhost:8443/callback.html`.
`scope`	A list of scopes to request when performing an OAuth 2.0 authorization flow, separated by spaces. For example, `openid profile email address`.
`oauthThreshold`	A threshold, in seconds, to refresh an OAuth 2.0 token before the `access_token` expires. Defaults to `30` seconds.

Storage

Properties

Property	Description
`tokenStore`	The API to use for storing tokens on the client: **`sessionStorage`** Store tokens using the `sessionStorage` API. The browser clears session storage when a page session ends. **`localStorage`** Store tokens using the `localStorage` API. The browser saves local storage data across browser sessions. This is the default setting, as it provides the highest browser compatibility. **`{{custom}}`** Specify a custom implementation that has functions that can set, retrieve, and remove, items from a custom storage scheme. Learn more in Customize storage on JavaScript.

Property	Description
`prefix`	Override the default `fr` prefix string applied to the keys used for storing data on the client, such as tokens, device IDs, and information about the steps in a journey. For example, the key used for storing tokens consists of the `prefix`, followed by the ID of the OAuth 2.0 client: `fr-sdkPublicClient`.

Logging

Properties

Property	Description
`logLevel`	Specify whether the SDK should output its log messages in the console and the level of messages to display. One of: • `none` (default) • `info` • `warn` • `error` • `debug`
`logger`	Specify a function to override the default logging behavior. Refer to Customize the Ping SDK for JavaScript logger.

General

Properties

Property	Description
`platformHeader`	Specify whether to include an `X-Requested-Platform` header in outgoing requests. The server can use the value of this header to alter the logic of an authentication flow. For example, if the value indicates a JavaScript web app, the journey could avoid device binding nodes, as they are only supported by Android and iOS apps. Defaults to `false`.

Endpoints

Properties

Property	Description
`serverConfig: { paths: { authenticate }}`	Override the path to the authorization server's `authenticate` endpoint. *Default*: `json/{realmPath}/authenticate`
`serverConfig: { paths: { authorize }}`	Override the path to the authorization server's `authorize` endpoint. *Default*: `oauth2/{realmPath}/authorize`
`serverConfig: { paths: { accessToken }}`	Override the path to the authorization server's `access_token` endpoint. *Default*: `oauth2/{realmPath}/access_token`
`serverConfig: { paths: { revoke }}`	Override the path to the authorization server's `revoke` endpoint. *Default*: `oauth2/{realmPath}/token/revoke`
`serverConfig: { paths: { userInfo }}`	Override the path to the authorization server's `userinfo` endpoint. *Default*: `oauth2/{realmPath}/userinfo`
`serverConfig: { paths: { sessions }}`	Override the path to the authorization server's `sessions` endpoint. *Default*: `json/{realmPath}/sessions`
`serverConfig: { paths: { endSession }}`	Override the path to the authorization server's `endSession` endpoint. *Default*: `oauth2/{realmPath}/connect/endSession`

Next

Next, you can Step 5. Instantiate the widget.

Step 5. Instantiate the widget

To use the Ping (ForgeRock) Login Widget in your app you must choose an appropriate place to mount it. Then, you need to choose which form factor to implement, either inline, or modal.

With those decisions made, you can instantiate the Ping (ForgeRock) Login Widget in your app, ready for your users to start their authentication or self-service journey.

Choose where to mount the Ping (ForgeRock) Login Widget

To implement the Ping (ForgeRock) Login Widget, we recommend you add a new element into your HTML file.

For most single page applications (SPA) this is your `index.html` file.

This new element should be a direct child element of `<body>` and not within the element where you mount your SPA.

Example HTML structure

```
<!DOCTYPE html>
<html lang="en">
<head>
<!-- ... -->
</head>
<body>
    <!-- Root element for main app -->
    <div id="root"></div>

    <!-- Root element for Widget -->
    <div id="widget-root"></div>

    <!-- scripts ... -->
  </body>
</html>
```

Tip

We recommend that you do not inject the element into which you mount the modal form factor in your app. This can cause virtual DOM issues.
Instead, manually hard-code the element in your HTML file.

Instantiate the modal form factor

To use the default Ping (ForgeRock) Login Widget modal form factor, import the modules into your app and instantiate the widget as follows:

Instantiate the modal form factor

```
// Import the Login Widget
import Widget, { configuration } from '@forgerock/login-widget';

// Configure SDK options
const myConfig = configuration();

myConfig.set({
  forgerock: {
    serverConfig: {
        baseUrl: 'https://openam-forgerock-sdks.forgeblocks.com/am',
        timeout: 3000,
    },
    // Optional but recommended configuration:
    realmPath: 'alpha',
    clientId: 'sdkPublicClient',
    redirectUri: window.location.href,
    scope: 'openid profile email address'
    },
  },
});

// Get the element in your HTML into which you will mount the widget
const widgetRootEl = document.getElementById('widget-root');

// Instantiate Widget with the `new` keyword
new Widget({
  target: widgetRootEl,
});
```

This mounts the Ping (ForgeRock) Login Widget into the DOM. The modal form factor is the default and is hidden when first instantiated.

Top open the modal, import the `component` module, assign the function, and call its `open()` method:

Open the modal

```
// Import the Login Widget
import Widget, { configuration, component } from '@forgerock/login-widget';

// Configure SDK options
const myConfig = configuration();

myConfig.set({
  forgerock: {
    serverConfig: {
        baseUrl: 'https://openam-forgerock-sdks.forgeblocks.com/am',
        timeout: 3000,
    },
    // Optional but recommended configuration:
    realmPath: 'alpha',
    clientId: 'sdkPublicClient',
    redirectUri: window.location.href,
    scope: 'openid profile email address'
    },
  },
});

// Get the element in your HTML into which you will mount the widget
const widgetRootEl = document.getElementById('widget-root');

// Instantiate Widget with the `new` keyword
new Widget({
  target: widgetRootEl, // Any existing element from static HTML file
});

// Assign the component function
const componentEvents = component();

// Call the open() method, for example after a button click
const loginButton = document.getElementById('loginButton');

loginButton.addEventListener('click', () => {
  componentEvents.open();
});
```

The modal form factor opens and displays a spinner graphic until you start a journey.

Tip

The modal form factor closes itself when a journey completes successfully.
You can also close it by calling `componentEvents.close();`

Instantiate the inline form factor

You override the default Ping (ForgeRock) Login Widget modal form factor and instead use the inline form factor. The inline form factor mounts within your application's controlled DOM, so it is important to consider how your framework mounts elements to the DOM.

For example, the inline form factor cannot mount into a virtual DOM element, such as those used by React. In this scenario, you must wait until the element has been property mounted to the real DOM before instantiating the Ping (ForgeRock) Login Widget.

To use the inline form factor, instantiate the widget with a `type: 'inline'` property, as follows:

Instantiate the inline form factor

```
// Import the Ping (ForgeRock) Login Widget
import Widget, { configuration } from '@forgerock/login-widget';

import { useRef } from 'react';

// Configure SDK options
const myConfig = configuration();

myConfig.set({
  forgerock: {
    serverConfig: {
        baseUrl: 'https://openam-forgerock-sdks.forgeblocks.com/am',
        timeout: 3000,
    },
    // Optional but recommended configuration:
    realmPath: 'alpha',
    clientId: 'sdkPublicClient',
    redirectUri: window.location.href,
    scope: 'openid profile email address'
    },
  },
});

// Target needs to be an actual DOM element, so ref is needed with inline type
const widgetElement = useRef(null);

// Instantiate Widget with the `new` keyword
new Widget({
  target: widgetElement.current,
  props: {
    type: 'inline', // Override the default 'modal' form factor
  },
});
```

The inline form factor loads into the specified DOM element and displays a spinner graphic until you start a journey.

Next

Next, you can Step 6. Start a journey.

Step 6. Start a journey

The Ping (ForgeRock) Login Widget displays a loading spinner graphic if it does not yet have a callback from the server to render.

You must specify and start a journey to make the initial call to the server and obtain the first callback.

To start a journey, import the `journey` function and execute it to receive a `journeyEvents` object. After you have this `journeyEvents` object, you can call the `journeyEvents.start()` method, which starts making requests to the server for the initial form fields.

You can call the `journeyEvents.start()` method anywhere in your application, or anytime, as long as it is after calling the configuration's `set()` method and after instantiating the Widget.

Start the default journey

```
// Import the Login Widget
import Widget, { configuration, journey } from '@forgerock/login-widget';

// Configure SDK options
const myConfig = configuration();

myConfig.set({
  forgerock: {
    serverConfig: {
        baseUrl: 'https://openam-forgerock-sdks.forgeblocks.com/am',
        timeout: 3000,
    },
    // Optional but recommended configuration:
    realmPath: 'alpha',
    clientId: 'sdkPublicClient',
    redirectUri: window.location.href,
    scope: 'openid profile email address'
    },
  },
});

// Get the element in your HTML into which you will mount the widget
const widgetRootEl = document.getElementById('widget-root');

// Instantiate Widget with the `new` keyword
new Widget({
  target: widgetRootEl,
});

// Assign the journey function
const journeyEvents = journey();

// Ensure you call `.start` *AFTER* instantiating the Widget
journeyEvents.start();
```

This starts the journey configured as the default in your server and renders the initial callback.

To specify which journey to use and other parameters, refer to Configure start() parameters.

Configure start() parameters

If you do not pass any parameters when calling the `start()` method the Ping (ForgeRock) Login Widget will use whichever journey is marked as the default in your server.

The Ping (ForgeRock) Login Widget will also use the values configured in the last invocation of the configuration's `set()` method.

You can override both of these behaviors by passing in JSON parameters:

Optional `start()` *parameters*

Parameter	Description
`journey`	Specify the name of the journey to use. If not specified, the Ping (ForgeRock) Login Widget uses whichever journey is marked as the default
`forgerock`	Override the current SDK configuration with any new or altered settings. For more information, refer to Step 4. Configure the SDK.
`resumeUrl`	Specify the full URL to visit if resuming a suspended journey. The server uses this to return your users to your application after clicking a "magic link" in an email, for example. The default is `window.location.href`.

Example of specifying the journey to use:

```
// Specify a different journey
journeyEvents.start({
  journey: 'sdkRegistrationJourney',
});
```

For more information, refer to journey in the API reference.

Configure journey() parameters

If you do not pass any parameters when calling the `journey()` function the Ping (ForgeRock) Login Widget will attempt to retrieve OAuth 2.0 tokens and user information by default.

You can override this behavior by passing in the following JSON parameters:

Optional `journey()` *parameters*

Parameter	Description
`oauth`	Set to `false` to prevent the Ping (ForgeRock) Login Widget attempting to obtain OAuth 2.0 tokens after successfully completing a journey. The default is `true`.
`user`	Set to `user` to prevent the Ping (ForgeRock) Login Widget attempting to obtain user information by calling the `/oauth2/userinfo` endpoint after successfully completing a journey. The default is `true`.

Example - obtaining only a user session token:

```
const journeyEvents = journey({
    oauth: false,
    user: false,
});
```

For more information, refer to journey API reference

Listen for journey completion

Use the `journeyEvents.subscribe` method to know when a user has completed their journey.

A summary of the events for a journey and their order is as follow:

1. Journey is loading
2. Journey is complete
3. Tokens are loading
4. Tokens are complete
5. Userinfo is loading
6. Userinfo is complete

Pass a callback function into this method to run on journey related events, of which there will be many, and each event object you receive contains a lot of information about the event.

You conditionally check for the events you are interested in and ignore what you do not need.

Example - subscribe to journey events

```
journeyEvents.subscribe((event) => {
  // Called multiple times, filtering by event data is recommended
  if (event.journey.successful) {
    // Only output successfull journey log entries
    console.log(event);
  }
});
```

Next

Next, you can learn more information about observables and how to Step 7. Subscribe to events.

Step 7. Subscribe to events

The Ping (ForgeRock) Login Widget has a number of asynchronous APIs, which are designed around an event-centric *observable* pattern. It uses Svelte's simplified, standard observable implementation called a "store".

> **Note**
>
> These Svelte stores are embedded into the Ping (ForgeRock) Login Widget itself.
> They are not a dependency that your app layer needs to import or manage.

For more information on Svelte stores, refer to the Svelte documentation.

This observable pattern is optimal for UI development as it allows for a dynamic user experience. You can update your application in response to the events occurring within the Ping (ForgeRock) Login Widget. For example, the Ping (ForgeRock) Login Widget has events such as "loading", "completed", "success", and, "failure".

Assign an observable

You can create a variable and assign the observable to it:

Assign an observable

```
const userInfoEvents = user.info();
```

Subscribe to observable events

An observable is a stream of events over time. The Ping (ForgeRock) Login Widget invokes the callback for each and every event from the observable, until you unsubscribe from it.

Use the `subscribe()` method on your variable to observe the event stream:

Example userInfoEvents observable

```
userInfoEvents.subscribe((event) => {
  if (event.loading) {
    console.log('User info is being requested from server');
  } else if (event.successful) {
    console.log('User info request was successful');
    console.log(event.response);
  } else if (event.error) {
    console.error('User info request failed');
    console.error(event.error.message);
  }
});
```

For information on the events each observable returns, refer to the API reference.

Unsubscribe from an observable

Unlike a JavaScript *promise*, an observable does not resolve and then get cleaned up after completion.

You need to unsubscribe from an observable if it is no longer needed. This is especially important if you are subscribing to observables in a component that gets created and destroyed many times over. Subscribing to an observable over and over without unsubscribing creates a memory leak.

To unsubscribe, assign the function that is returned from calling the `subscribe()` method to a variable. Call this variable at a later time to unsubscribe from the obeservable.

Example unsubscribe from an observable

```
const unsubUserInfoEvents = userInfoEvents.subscribe((event) => console.log(event));

// ...

// Unsubscribe when no longer needed
unsubUserInfoEvents();
```

You *do not need* to unsubscribe from observables if you subscribe to observables in a top-level component of your app that is only initiated once, and is retained over the lifetime of your application.

A good location in which to subscribe to observables might be the central state management component or module of your application.

Get current local values

The Ping (ForgeRock) Login Widget stores a number of important values internally.

You can get the current values stored within the Ping (ForgeRock) Login Widget without subscribing to any future events or their resulting state changes by calling `subscribe()` and then immediately calling its unsubscribe method:

Get current stored values and unsubscribe

```
// Create variable for user info
let userInfo;

// Call subscribe, get the current local value, and then immediately call the returned function
userInfoEvents.subscribe((event) => (userinfo = event.response))(); // <-- notice the second pair of parentheses
```

Get updated values

You can ask the Ping (ForgeRock) Login Widget to request new, fresh values from the server, rather than just what it has stored locally, by calling the observable action methods, such as `get`.

Get latest values from the server

```
userInfoEvents.get();
```

When using the observable pattern, you can call this method and forget about it. The get causes any `subscribe` callback functions you have for the observable to receive the events and new state.

The `subscribe` can exist before or after this `get` call, and it will still capture the resulting events.

Use promises rather than observables

We recommend observables, but the choice is up to you.

All of the Ping (ForgeRock) Login Widget APIs that involve network calls have an alternative promise implementation that you can use.

The following example again shows `userInfoEvents` but converted to use promises:

Using promises rather than observables

```
// async-await
let userInfo;
async function example() {
  try {
    userInfo = await userInfoEvents.get();
  } catch (err) {
    console.log(err);
  }
}

// Promise
let userInfo;
userInfoEvents
  .get()
  .then((data) => (userInfo = data))
  .catch((err) => console.log(err));
```

Theme the widget

The Ping (ForgeRock) Login Widget provides a default theme with both light and dark modes.

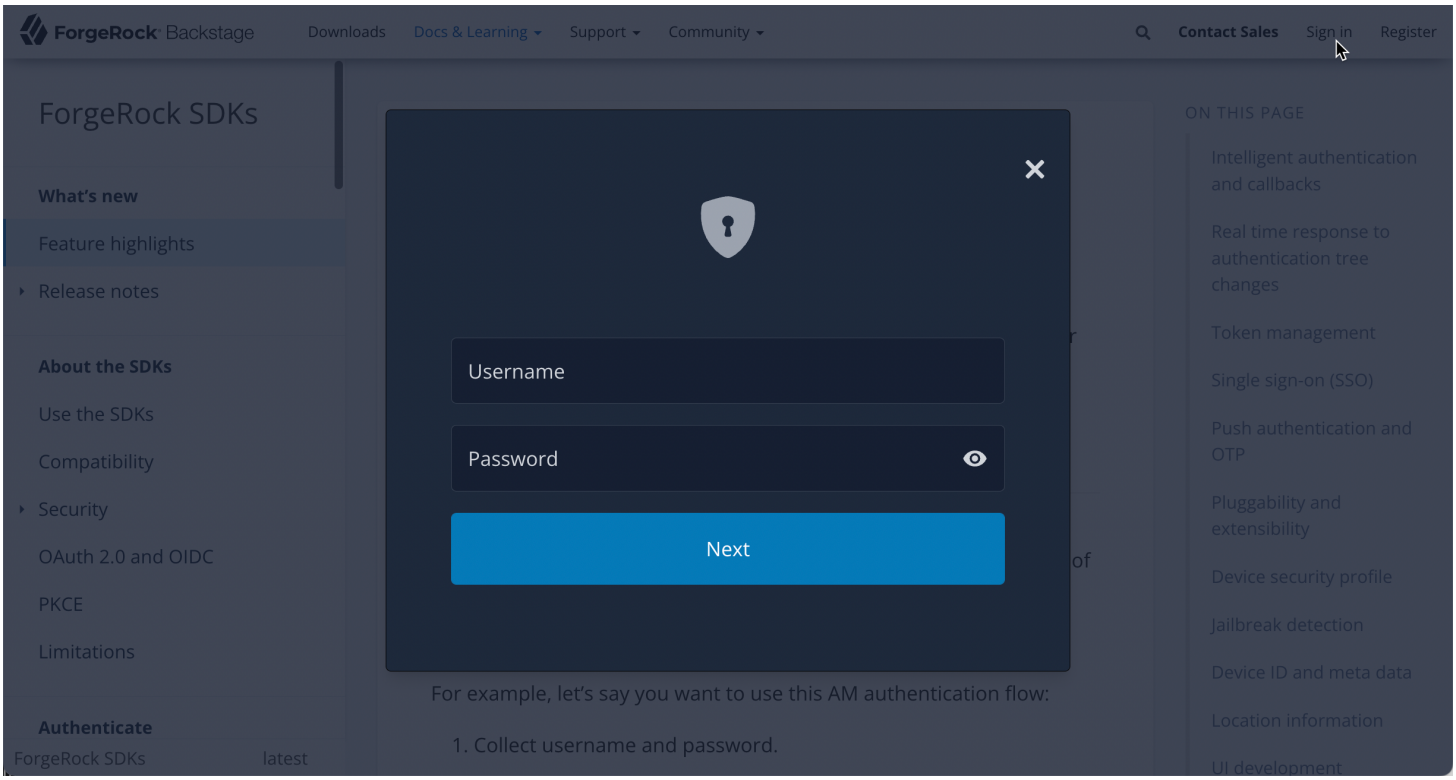


Figure 1. Example of the modal component in dark mode

You can alter these themes by using the Tailwind configuration file.

> **Note**
>
> If your organization's UX or brand requires pixel-perfect UIs developed in accordance to Photoshop, Figma or other "design mocks", Ping (ForgeRock) Login Widget might not be able to support such requirements.

Switch between light and dark themes

To switch to the dark mode, you can manually add `tw_dark` to the `<body>` element:

```
<body class="tw_dark"></body>
```

The Ping (ForgeRock) Login Widget defaults to the light mode if the class is not present.

You can programmatically apply the class if required:

```
const prefersDarkTheme = window.matchMedia('(prefers-color-scheme: dark)').matches;
if (prefersDarkTheme) {
    document.body.classList.add('tw_dark');
}
```

Customize the theme

To reconfigure the theme to use different colors, fonts, or sizing you can provide new values to the Tailwind configuration file and rebuild the Ping (ForgeRock) Login Widget, as follows:

1. Download the Ping (ForgeRock) Web Login Framework from the GitHub repository:

   ```
   git clone https://github.com/ForgeRock/forgerock-web-login-framework.git
   ```

2. In a terminal window, navigate to the root of the Ping (ForgeRock) Web Login Framework:

   ```
   cd forgerock-web-login-framework
   ```

3. Run `npm` to download and install the required packages and modules:

   ```
   npm install
   ```

4. Run the development script:

   ```
   npm run dev
   ```

5. Run Storybook:

   ```
   npm run storybook
   ```

   Make a note of the URLs to the Storybook UI listed in the terminal output:

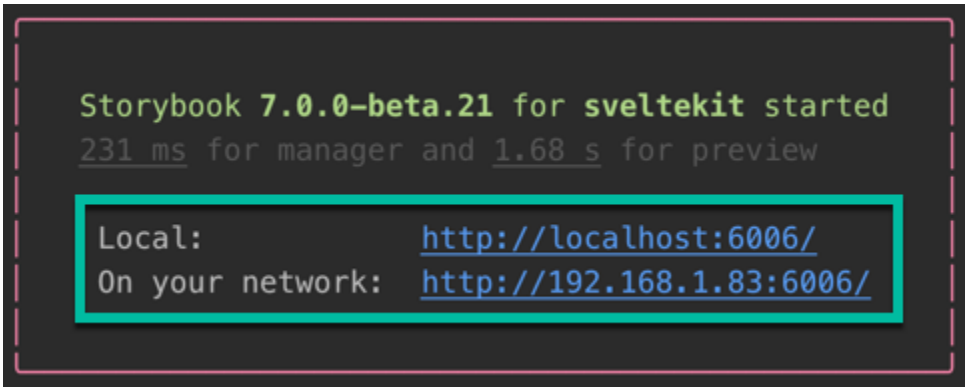


Figure 2. URLs to the Storybook UI

6. Open the `tailwind.config.cjs` file in the root of the Ping (ForgeRock) Web Login Framework and adjust your theme by adding them under the `extend` property:

```
// tailwind.config.cjs
module.exports = {
  content: ['./src/**/*.{html,js,svelte,ts}', './.storybook/preview-head.html'],
  darkMode: 'class',
  presets: [require('./themes/default/config.cjs')],
  theme: {
    extend: {
      // Add your customizations here
      colors: {
        body: {
          light: 'darkblue',
        },
        primary: {
          dark: 'darkorange',
        },
        background: {
          light: "gainsboro",
        },
      },
      fontFamily: {
        sans: ['Impact'],
      },
    },
  },
};
```

7. Navigate to the Storybook UI provided in the terminal output earlier to view your changes:

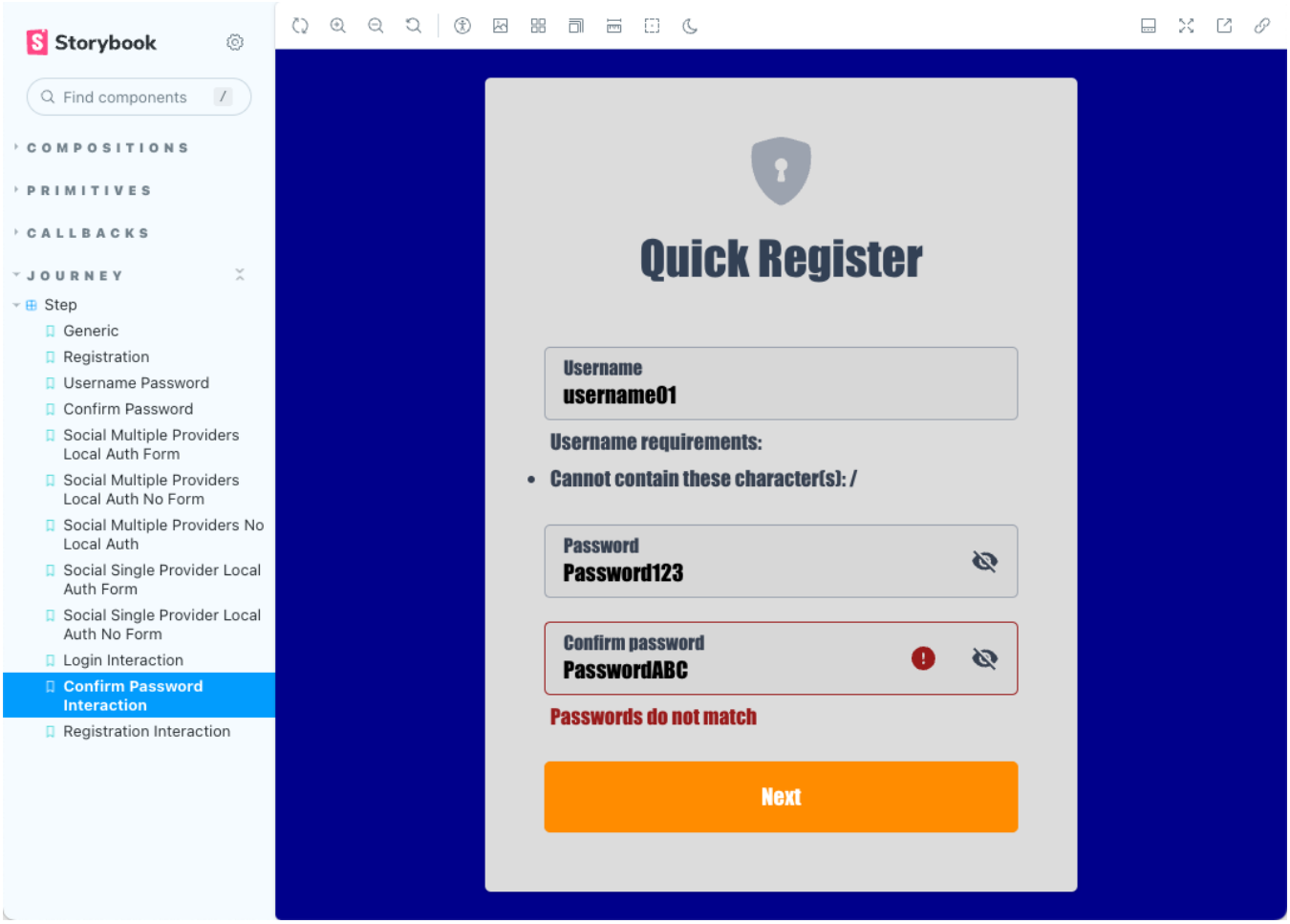


Figure 3. Customized registration modal in Storybook

Changes you make to the `tailwind.config.cjs` file are automatically reflected in the Storybook UI when you save them to disk.

Recommendations

Anything configurable in Tailwind is also configurable in the theme. The custom properties the default theme uses are stored in the `/themes/default/tokens.cjs` file.

We recommend you do not directly modify these default theme files. Only modify the root `tailwind.config.cjs`.

Supported customization

Using the methods on this page, you can customize:

- Colors
- Fonts
- Type sizes
- Spacing

- Breakpoints

Implement your use cases with the Ping (ForgeRock) Login Widget

Find out how to achieve some common use case scenarios using the Ping (ForgeRock) Login Widget.

Log in with social authentication

Social authentication provides your users with a choice of ways to sign in that suits them.

Select from supported social providers in a journey to initiate an OAuth 2.0 flow to authenticate with the social provider, before returning to the original journey.

Log in with OATH one-time passwords

If your users have registered the ForgeRock Authenticator for one-time passwords using a browser for example, then an app using the Ping (ForgeRock) Login Widget will be able to accept the one-time password from the authenticator app.

Implement a CAPTCHA

Help to prevent automated scripts from attempting to authenticate to your servers by implementing a CAPTCHA in your Ping (ForgeRock) Login Widget app.

Suspend journeys with "magic links"

You can use the Email Suspend Node within your journeys to support a number of authentication experiences, including verifying a user's email address, building a "forgot password" flow, or using an email address for multifactor authentication.

Log in with social authentication

Social authentication provides your users with a choice of ways to sign in that suits them.

The Ping SDK for JavaScript supports social authentication with the following providers:

- Apple
- Facebook
- Google

Selecting one of these providers in a journey initiates an OAuth 2.0 flow allowing them to authenticate themselves with the social provider before returning to the original journey.

To enable this flow you need to:

1. Offer a choice of social identity providers using the **Select Identity Provider** node.
    - Optionally, you can allow users to skip social authentication and enter their credentials in the same form, provided nodes such as a username collector are also present.
2. Handle the OAuth 2.0 flow for your users using the **Social Provider Handler Node**.
3. Determine if the user signed in to the social provider maps to a known user by adding the **Identify Existing User Node**.

The following is an example journey for social authentication:

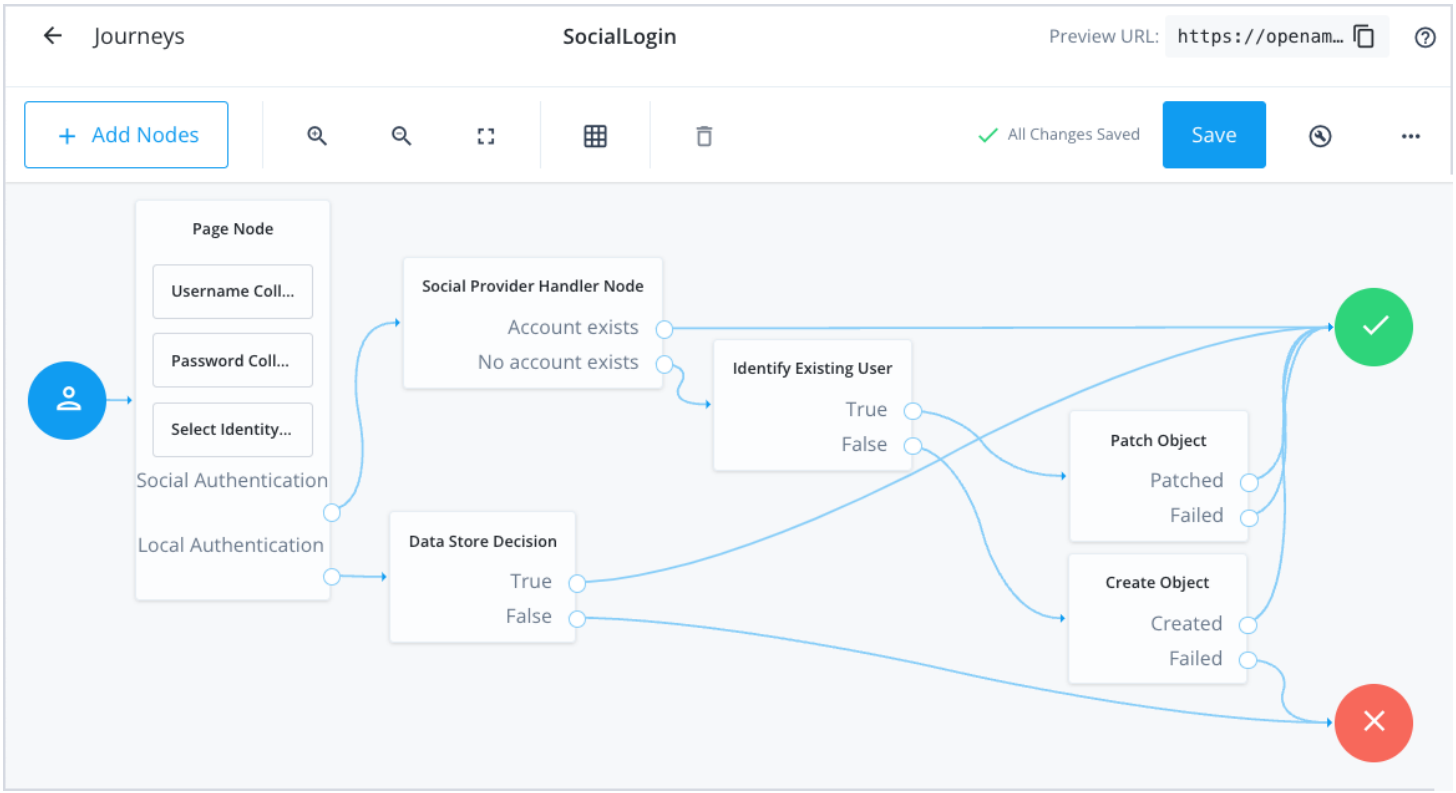


Figure 1. Example social login journey

Tip

For a detailed guide covering the creation of social authentication journeys, refer to How do I create end user journeys for social registration and login in PingOne Advanced Identity Cloud? in the Backstage Knowledge Base.

On the client side, the Ping (ForgeRock) Login Widget handles the selection of the identity provider and redirection to the provider.

You need to ensure your app manages the return back from the provider. To handle the return from a social provider, detect `code`, `state` and `form_post_entry` query parameters, as these instruct the Ping (ForgeRock) Login Widget to resume authentication using the current URL:

Detect social authentication query parameters and continue the journey

```
import { journey } from '@forgerock/login-widget';

const journeyEvents = journey();

const url = new URL(location.href);
const codeParam = url.searchParams.get('code');
const stateParam = url.searchParams.get('state');
const formPostEntryParam = url.searchParams.get('form_post_entry');

if (formPostEntryParam || (codeParam && stateParam)) {
    journey.start({ resumeUrl: location.href });
}
```

The `location.href` value includes any query parameters returned from the social provider. Without all the query parameters, the Ping (ForgeRock) Login Widget might not be able to rehydrate the journey and continue as needed.

Log in with OATH one-time passwords

The Ping (ForgeRock) Login Widget provides UI elements for the **OATH Token Verifier** node but not currently the **OATH Registration**.

If your users have registered the ForgeRock Authenticator for one-time passwords using a browser, for example, then an app using the Ping (ForgeRock) Login Widget will be able to accept the one-time password from the authenticator app.

The Ping (ForgeRock) Login Widget requires that the **OATH Token Verifier** node is contained within a **Page node** configured with a specific **Stage** property.

In the containing **Page Node**, set the **Stage** property to `OneTimePassword`:

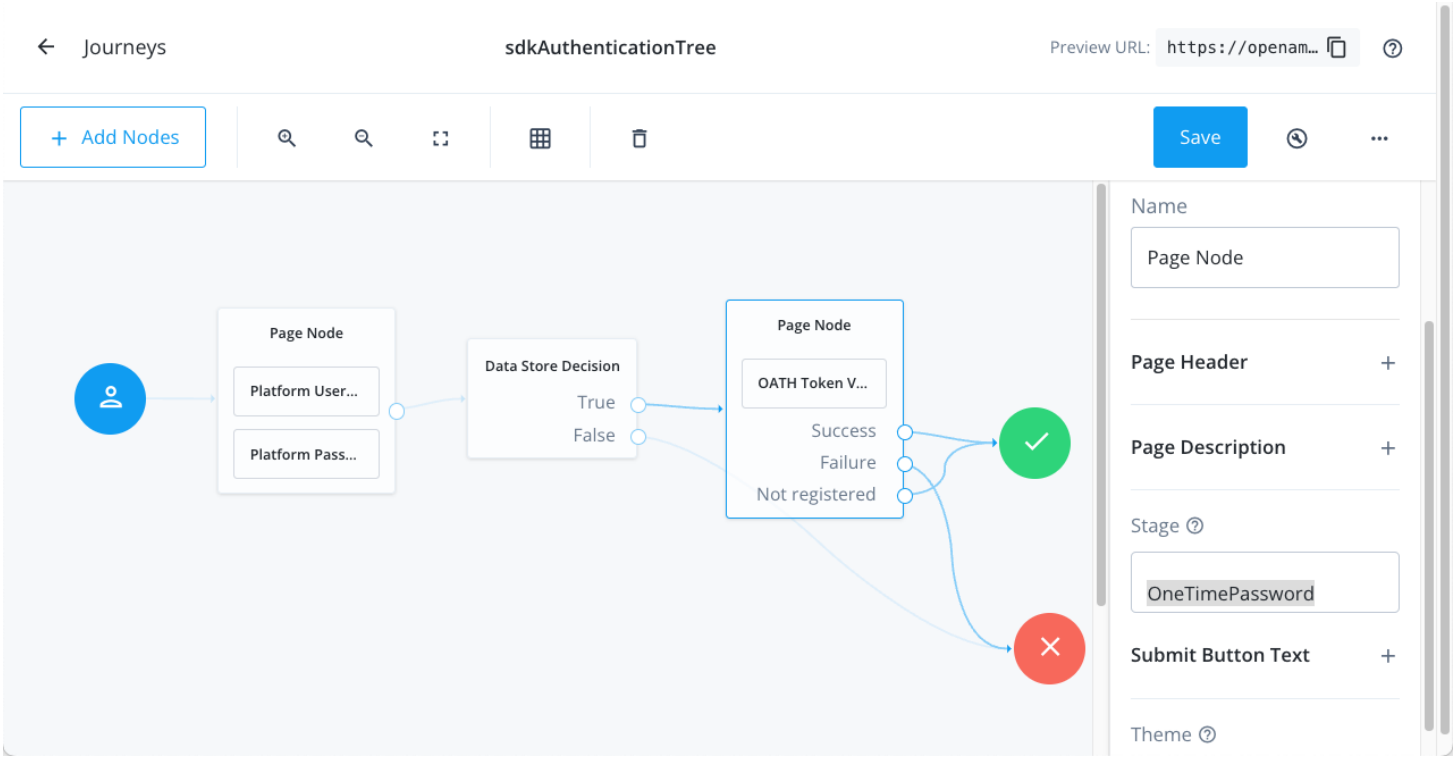


Figure 1. OATH journey example

The Ping (ForgeRock) Login Widget detects that stage value as a special case and renders the appropriate UI:

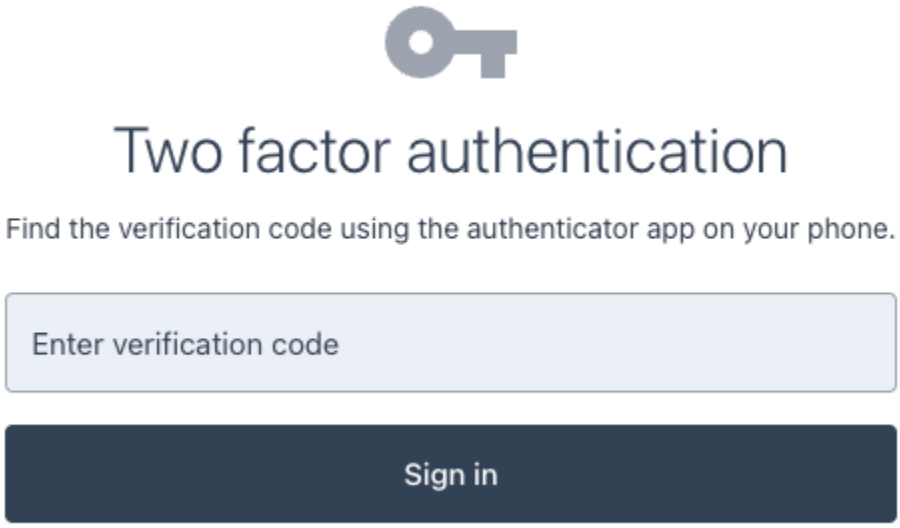


Figure 2. Rendering with the `OneTimePassword` stage property

If you do not put the **OATH Token Verifier** node within a **Page node**, the Ping (ForgeRock) Login Widget will not render the UI correctly:

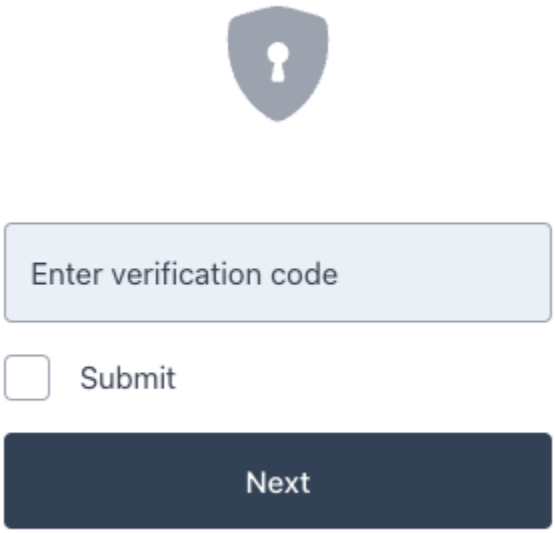


Figure 3. Rendering a lone OATH Token Verifier node

Implement a CAPTCHA

The Ping (ForgeRock) Login Widget supports the use of a CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart), which helps to prevent automated scripts from attempting to authenticate to your servers.

To use a CAPTCHA in the Ping (ForgeRock) Login Widget, add the CAPTCHA node to your authentication journey.

Supported CAPTCHA variants

Ping (ForgeRock) Login Widget supports the following CAPTCHA variants:

CAPTCHA variant	Support
hCaptcha	✓ Active ✕ Passive / 99% Passive ✕ Invisible
reCAPTCHA v2	– Full, except Invisible reCAPTCHA.
reCAPTCHA v3	✓ Full
reCAPTCHA Enterprise	✕ Not currently supported

Configure your app

The Ping (ForgeRock) Login Widget cannot inject the scripts necessary to use a CAPTCHA in your app.

You must add the relevant scripts yourself, usually to the `<head>` of the page:

hCaptcha

If you are using hCaptcha with the Ping (ForgeRock) Login Widget, you must first have the JavaScript loaded into your app's DOM:

```
<script src="https://js.hcaptcha.com/1/api.js" async defer></script>
```

reCAPTCHA v2

If you are using Google reCAPTCHA v2 with the Ping (ForgeRock) Login Widget, you must first have the JavaScript loaded into your app's DOM:

```
<script async src="https://www.google.com/recaptcha/api.js"></script>
```

reCAPTCHA v3

If you are using Google reCAPTCHA v3, you must append your site key in a query string parameter named `render`:

```
<script async src="https://www.google.com/recaptcha/api.js?render={reCAPTCHA_site_key}"></script>
```

When calling a journey that uses reCAPTCHA v3 you can add a `recaptchaAction` property with a custom value. That value tags the event in the reCAPTCHA console so that you can track different usage:

```
journey.start({
    journey: 'reCAPTCHAv3journey',
    // ….any other journey config required….
    recaptchaAction: 'loginVIPArea', // reCAPTCHA v3 only, falls back to journey name
});
```

If you do not provide a `recaptchaAction` value, the SDK attempts to use the name of the journey instead, if available.

Test a CAPTCHA

With your app configured and the necessary scripts in place, you can visit any journey that contains a CAPTCHA node to test a CAPTCHA with the Ping (ForgeRock) Login Widget.

For example, the following image shows how the Ping (ForgeRock) Login Widget handles a CAPTCHA node alongside a platform username and platform password nodes, all within a single page node:

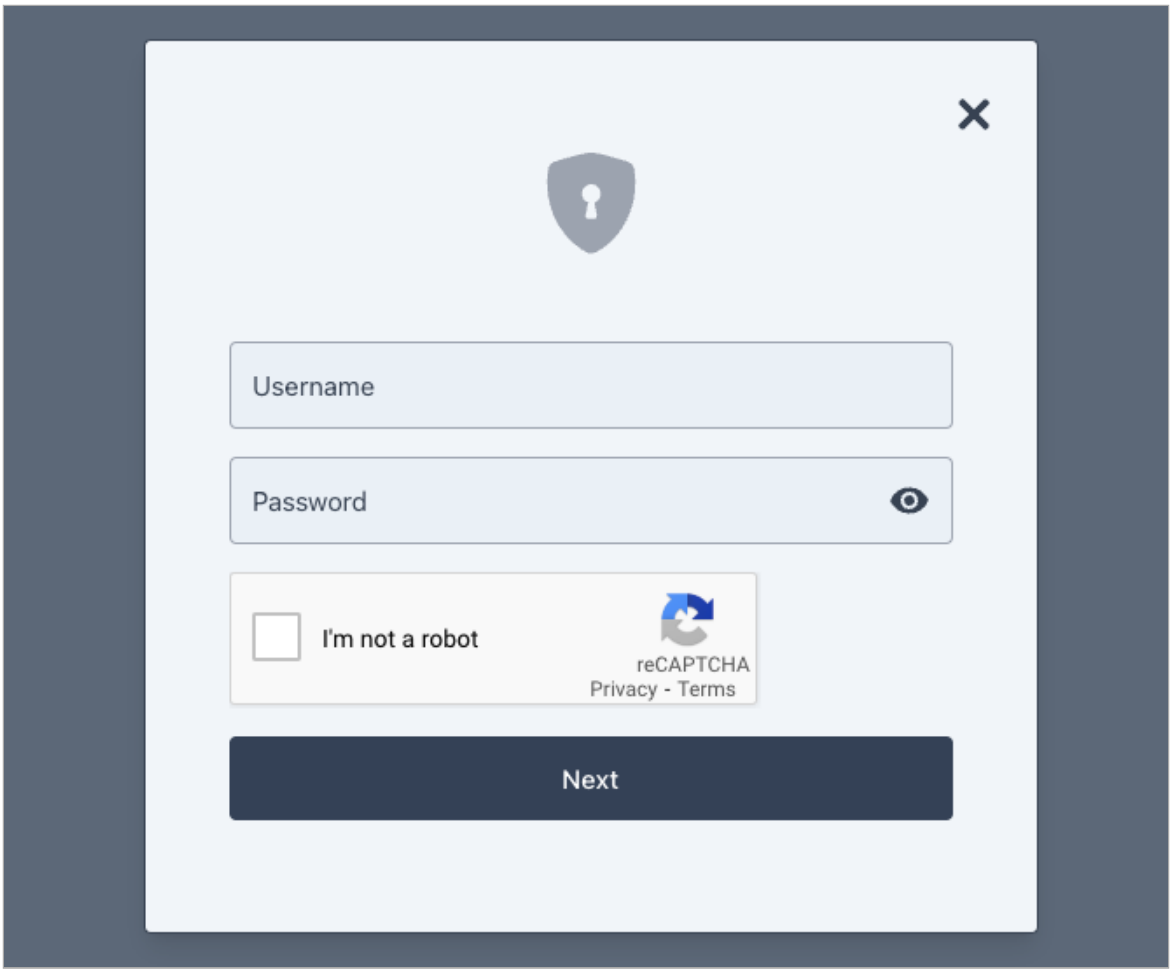


Figure 1. Login Widget handling reCAPTCHA v2 in a page node

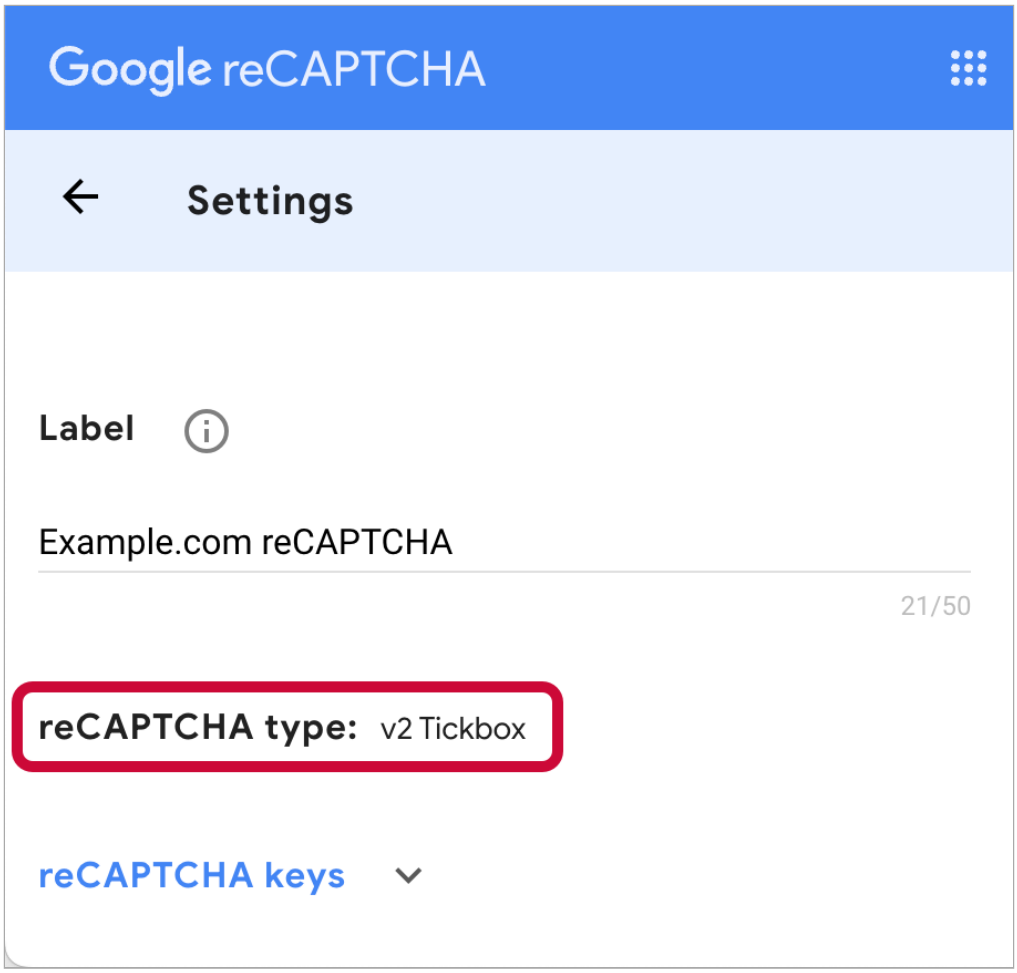
Troubleshooting

This section contains information on how to diagnose issues when using a CAPTCHA with the Ping (ForgeRock) Login Widget.

Why does the reCAPTCHA display "ERROR for site owner: Invalid key type"?

Ensure the site key you have specified in the CAPTCHA node is configured for the version of reCAPTCHA type you want to use.

For example, the following image shows a configuration for **v2 Tickbox**:



When using a v2 site key, do not select **ReCaptcha V3 Node** in the CAPTCHA node configuration.

Why does the browser console display "Error: Invalid site key or not loaded in api.js"?

Ensure you have added the correct site key value as a `render` query parameter of the Google `api.js` script.

For example:

```
<script async src="https://www.google.com/recaptcha/api.js?render=1249672216234"></script>
```

Why does the reCAPTCHA display "Localhost is not in the list of supported domains for this site key."?

The `localhost` domain is blocked from working with reCAPTCHA by default.

You can add the domain to the site key configuration for testing purposes if required.

> **Note**
>
> It can take several minutes for changes to the allowed domains to take effect.

For more information, refer to Google's reCAPTCHA documentation.

Suspend journeys with "magic links"

You can use the Email Suspend Node within your journeys to support a number of experiences, including verifying a user's email address, building a "forgot password" flow or using an email address for multifactor authentication.

The node suspends the journey until the user clicks a link—referred to as a *magic link*--in their email. This link contains a generated unique code that can continue the suspended journey.

This page shows how to configure the Ping (ForgeRock) Login Widget to take advantage of this feature.

Configure the authentication server

1. Add the Email Suspend Node to the journey to suspend it until the user continues the journey from the link found in their email.

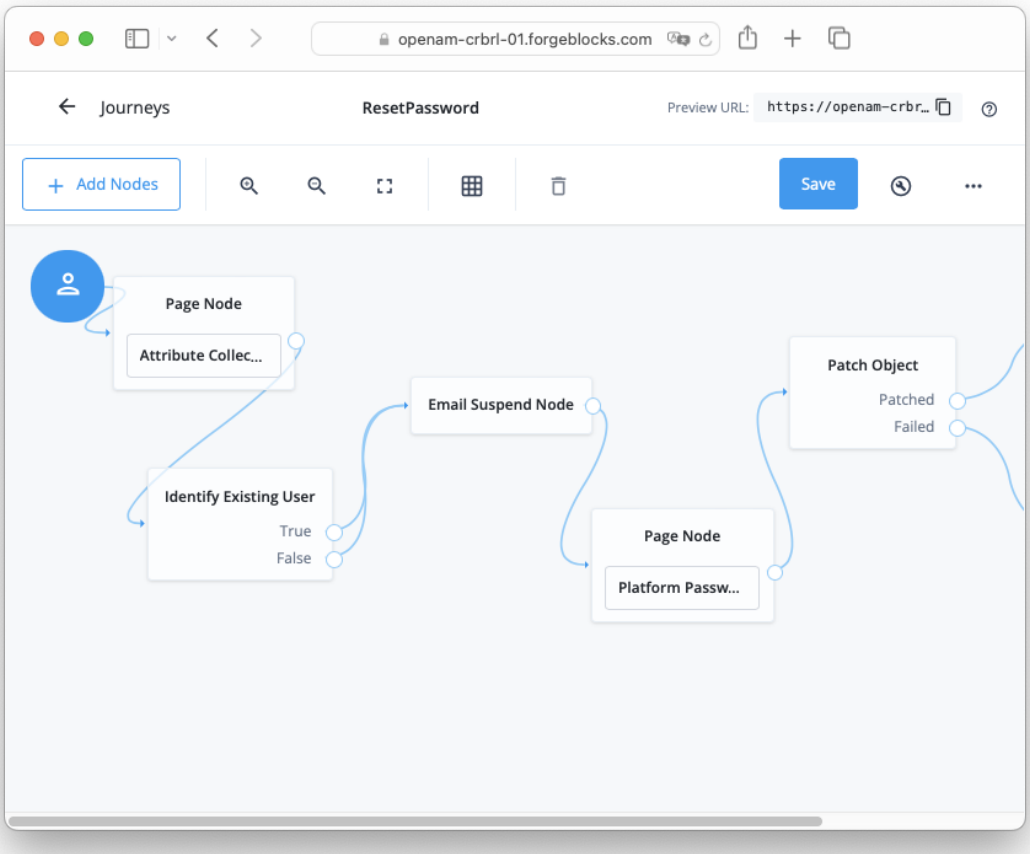


Figure 1. Insert the Email Suspend Node into your journey

2. Configure the **External Login Page URL** property in the Access Management native console to match your custom app's URL. This ensures the magic links are able to redirect users to your app to resume the journey. If not specified, the default behavior is to route users to the login page.

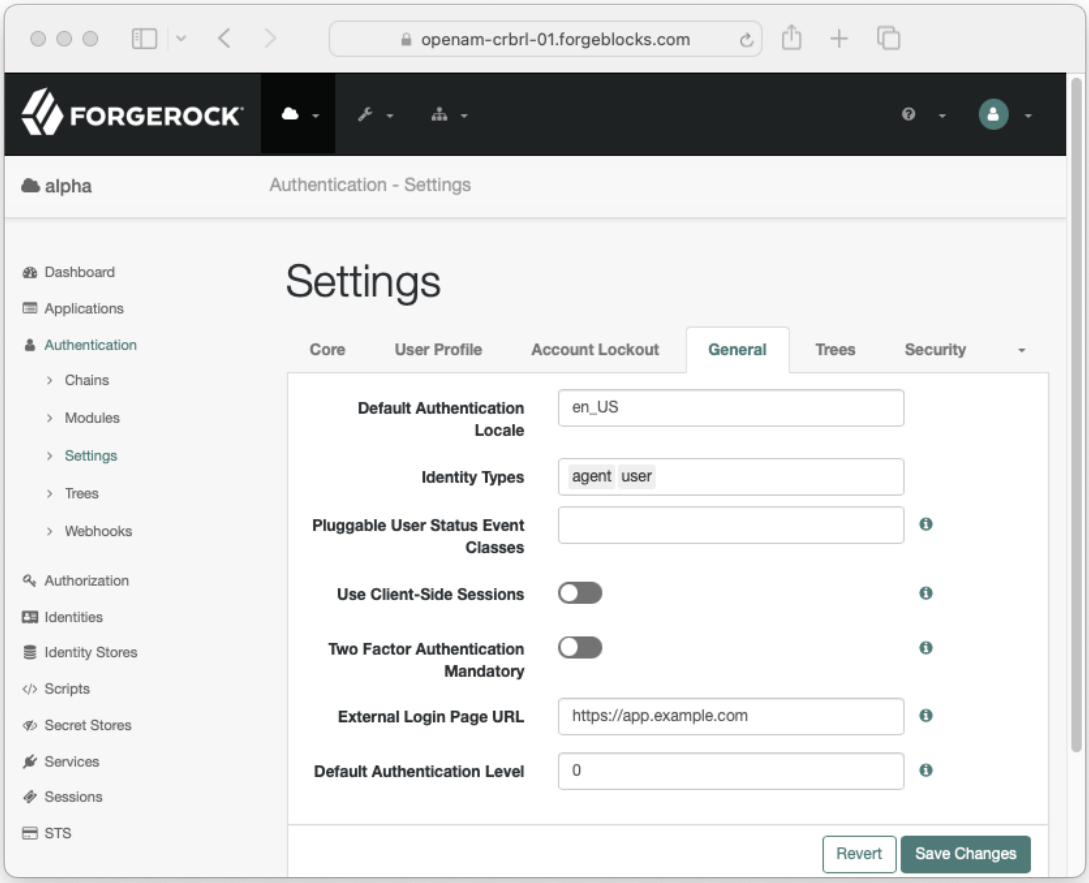


Figure 2. Configure external login URL in the PingAM native console

3. When the Ping (ForgeRock) Login Widget encounters the Email Suspend Node, it renders the string configured in the **Email Suspend Message** property configured in the node. The user is not able to continue the journey until they click the link emailed to them.

Handle suspend IDs in your app

When your app handles a magic link, it needs to recognize it as a special condition and provide the Ping (ForgeRock) Login Widget with the full URL that the user clicked in their email.

Return this URL, including all query parameters, to the server as the value of the `resumeUrl` parameter:

```
import { journey } from '@forgerock/login-widget';

const journeyEvents = journey();

const url = new URL(location.href);
const suspendedId = url.searchParams.get('suspendedId');

if (suspendedId) {
  journeyEvents.start({ resumeUrl: location.href });
}
```

The `location.href` value includes all query parameters included in the magic link. Without all the query parameters, the Ping (ForgeRock) Login Widget might not be able to rehydrate the journey and continue as needed.

Integrating the Ping (ForgeRock) Login Widget

Find out how you can integrate the Ping (ForgeRock) Login Widget with different frameworks and libraries.

Integrate with PingOne Protect for risk evaluations

The Ping (ForgeRock) Login Widget can integrate with PingOne Protect to evaluate the risk involved in a transaction.

Use PingOne Protect in your authentication journeys to help prevent identity fraud by incorporating advanced features and real-time detection.

Integrate Login Widget into a React app

Learn how to integrate the Ping (ForgeRock) Login Widget into a simple React app that you scaffold using Vite.

Integrate with PingOne Protect for risk evaluations

The Ping (ForgeRock) Login Widget can integrate with PingOne Protect to evaluate the risk involved in a transaction.

Important

PingOne Protect is supported in the following servers:

Advanced Identity Cloud

Use the official PingOne Protect nodes

PingAM 7.5 and later

Use the official PingOne Protect nodes

PingAM 7.2 - 7.4

Use the marketplace PingOne Protect nodes

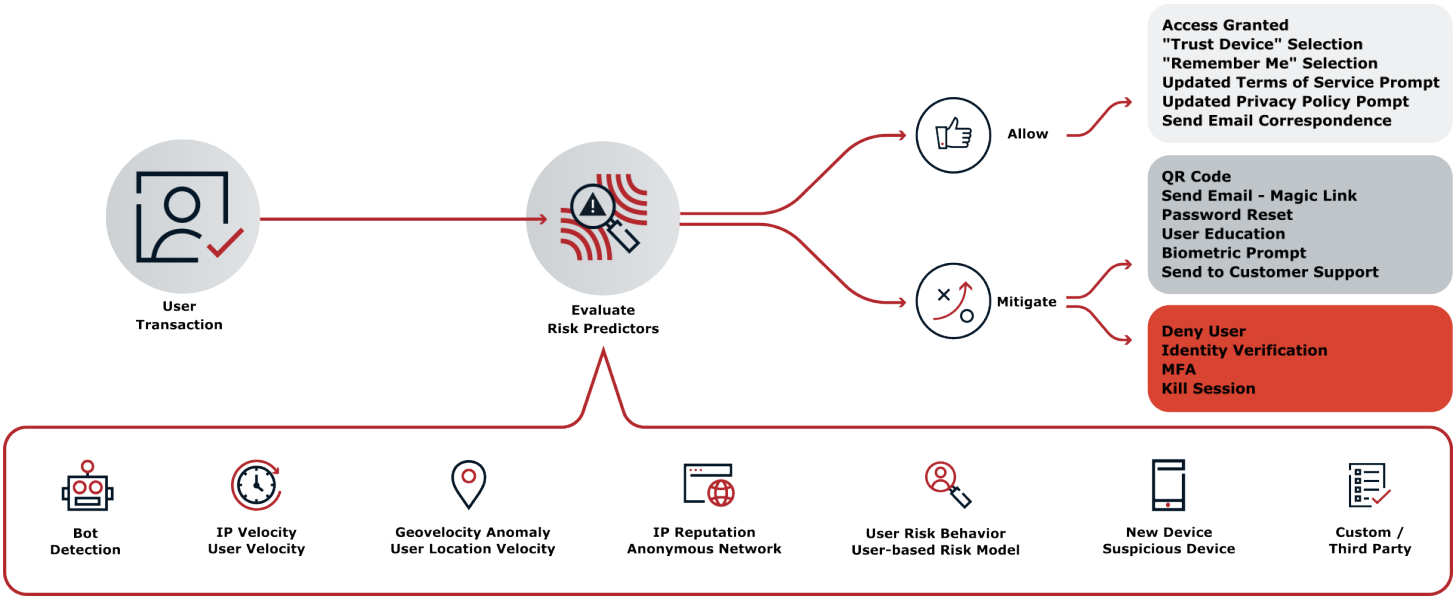


Figure 1. A flowchart illustrating how risk predictors evaluate many different data points.

You can instruct the Ping (ForgeRock) Login Widget to use the embedded PingOne Signals SDK to gather information during a transaction. Your authentication journeys can then gather this information together and request a risk evaluation from PingOne.

Based on the response, you can choose whether to allow or deny the transaction or perform additional mitigation, such as bot detection measures.

You can use the audit functionality in PingOne to view the risk evaluations:

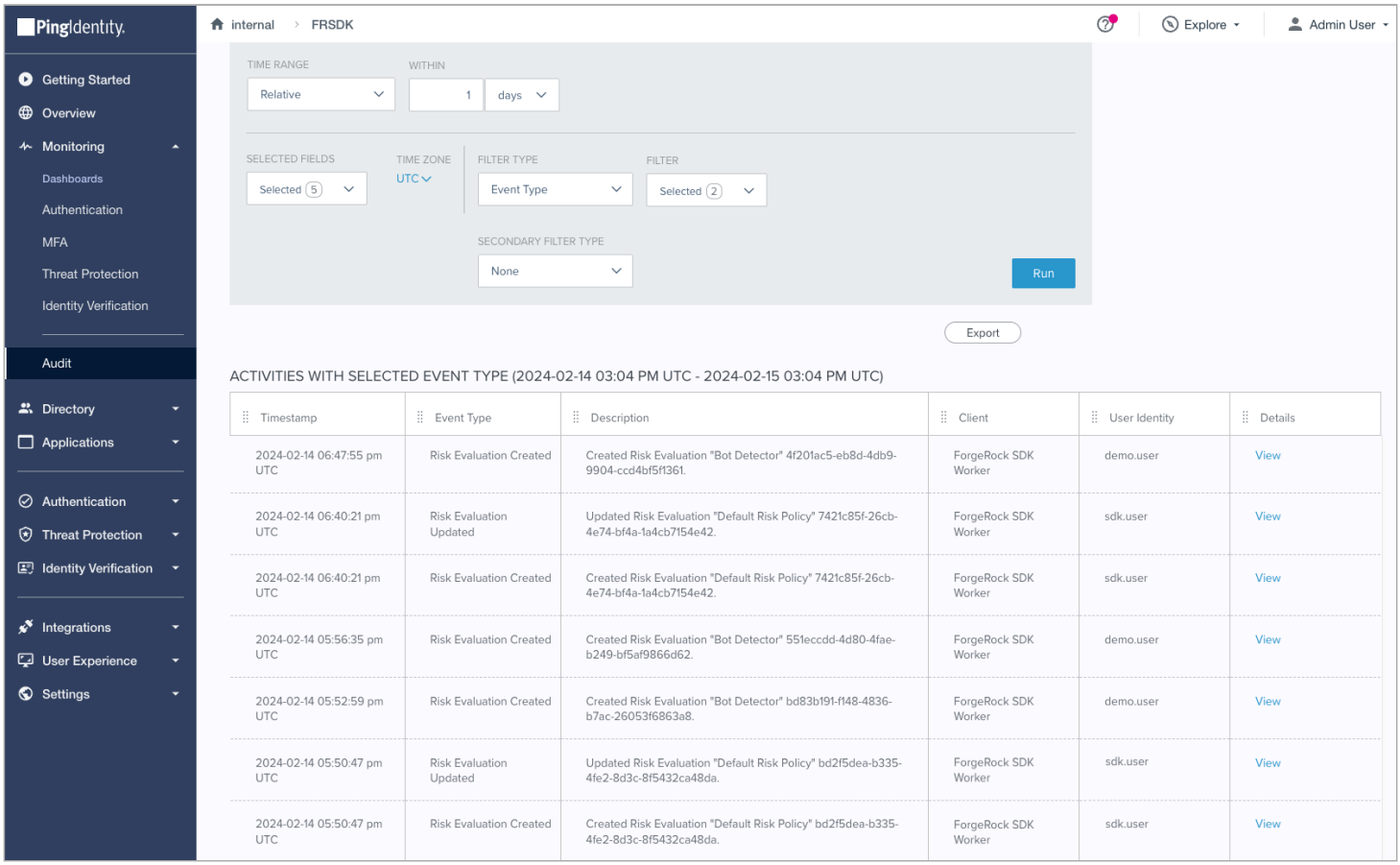


Figure 2. Risk evaluation records in the PingOne audit viewer.

Steps

Step 1. Set up the servers

In this step, you set up your PingOne Advanced Identity Cloud or PingAM server, and your PingOne instance to perform risk evaluations.

For example, you create a worker application in PingOne and configure your server to access it. You also create an authentication journey that uses the relevant nodes.

Step 2. Configure the Ping (ForgeRock) Login Widget for PingOne Protect

With everything prepared, you can now configure the Ping (ForgeRock) Login Widget to evaluate risk by using PingOne Protect.

Step 1. Set up the servers

In this step, you set up your PingOne Advanced Identity Cloud or PingAM server, and your PingOne instance to perform risk evaluations.

1. Create a worker application in PingOne
2. Configure the PingOne Worker service in your server
3. Configure a journey to perform PingOne Protect risk evaluations

Create a worker application in PingOne

To allow your server to access the PingOne administration API you must create a worker application in PingOne.

The worker application provides the client credentials your server uses to communicate with the PingOne admin APIs using the OpenID Connect protocol.

To create a worker application in PingOne:

1. In the PingOne administration console, navigate to **Applications** > **Applications**, and then click **Add** (+).
2. In the **Add Application** panel:
    1. In **Application name**, enter a unique identifier for the worker application.

       For example, `Ping SDK Worker`.
    2. Optionally, enter a **Description** for the application and select an **Icon**.

       These do not affect the operation of the worker application but do help you identify it in the list.
    3. In **Application Type**, select **Worker**.
    4. Click **Save**.
3. In the application properties panel for the worker application you created:
    1. On the **Roles** tab, click **Grant Roles**.
    2. On the **Available responsibilities** tab, select the **Identity Data Admin** row, and ensure the environment is correct.
    3. Click **Save**.
    4. On the **Overview** tab, ensure your worker application resembles the following image, and then enable it by using the toggle (1):

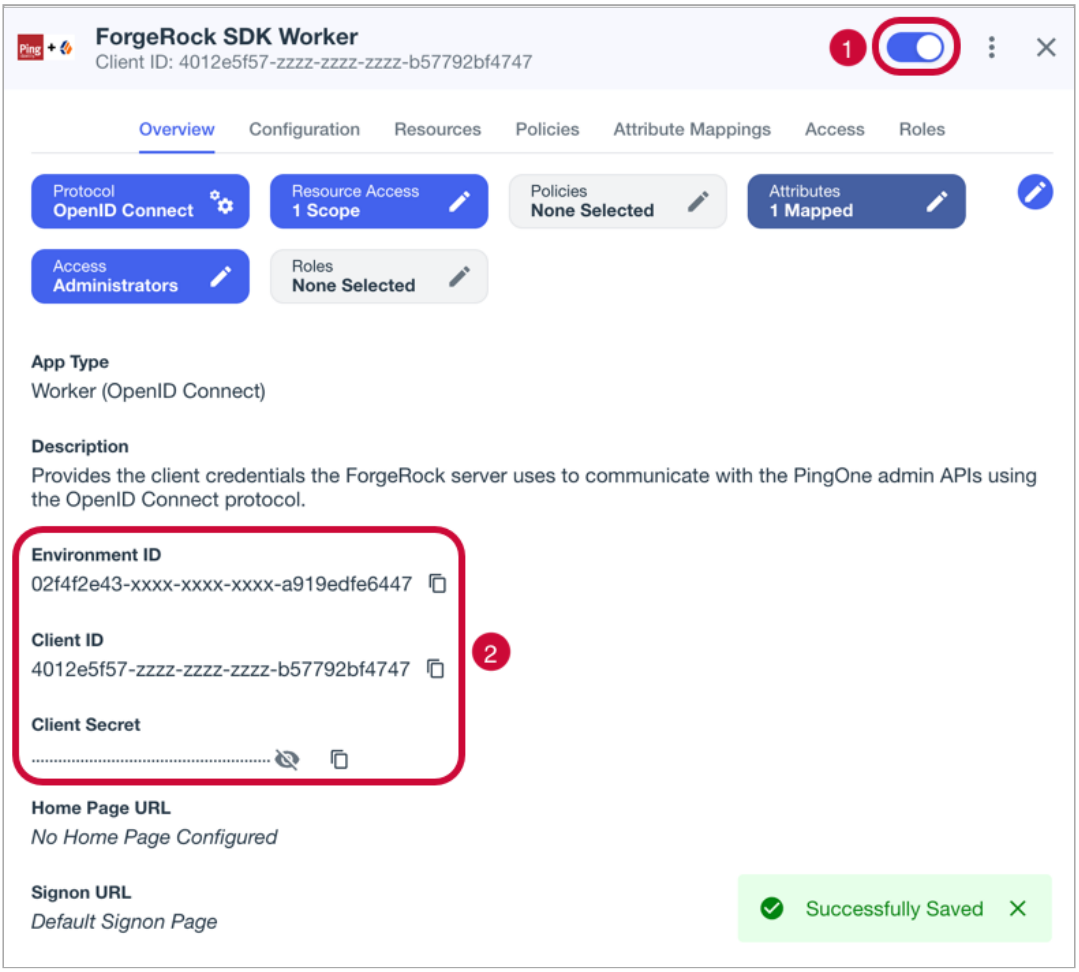


Figure 1. Example worker application in PingOne

5. Make a note of the **Environment ID**, **Client ID**, and **Client Secret** values (2).

   You need these values in the next step when you Configure the PingOne Worker service in your server.

Configure the PingOne Worker service in your server

After you create a worker application in PingOne, you must configure the PingOne Worker service in your server with the credentials.

Prerequisites

You need the following values from the PingOne Worker application you created in PingOne:

Client ID

Client ID of the worker application in PingOne.

Example: `6c7eb89a-66e9-ab12-cd34-eeaf795650b2`

Client Secret

Client secret of the worker application in PingOne.

> **Tip**
>
> Use the **Secret Mask** (👁) or **Copy to Clipboard** (📋) buttons to obtain the value in the PingOne administration console.

Example: `Ch15~o5Hm8N4_eS_m8~ARrV0KQAIQS6d.sJWe8TMXurEb~KWexY_p0gelR`

Environment ID

Identifier of the environment that contains the worker application in PingOne.

Example: `3072206d-c6ce-ch15-m0nd-f87e972c7cc3`

> **Important**
>
> The PingOne Worker Service requires a configured OAuth2 provider service in your server.
>
> - If you are using a self-managed AM server, you must configure the OAuth2 Provider service in a realm to expose the OAuth 2.0 endpoints and OAuth 2.0 administration REST endpoints.
> - The **OAuth2 provider service** is preconfigured in Advanced Identity Cloud.

Register the client secret in the server

You need to make the client secret of the worker application in PingOne available for use in the PingOne worker service.

Advanced Identity Cloud

If you are using Advanced Identity Cloud you will need to create an environment secret to hold the client secret value, as follows:

1. In the Advanced Identity Cloud admin UI, go to ⚙ **Tenant Settings** > **Global Settings** > **Environment Secrets & Variables**.
2. Click the **Secrets** tab.
3. Click **+ Add Secret**.
4. In the **Add a Secret** modal window, enter the following information:

Name	Enter a secret name. For example, `ping-protect-client-secret`. **Note** Secret names cannot be modified after the secret has been created.
Description	(optional) Enter a description of the purpose of the secret.
Value	Enter the **Client Secret** value you obtained when creating the worker application in PingOne. For example, `Ch15~o5Hm8N4_eS_m8~ARrV0KQAIQS6d.sJWe8TMXurEb~KWexY_p0gelR`. The field obscures the secret value by default. You can optionally click the visibility toggle (👁) to view the secret value as you enter it.

5. Click **Save** to create the variable.
6. Click **View Update**, check the details of the new secret, and then click **Apply Update**.

   Advanced Identity Cloud displays a final confirmation page.

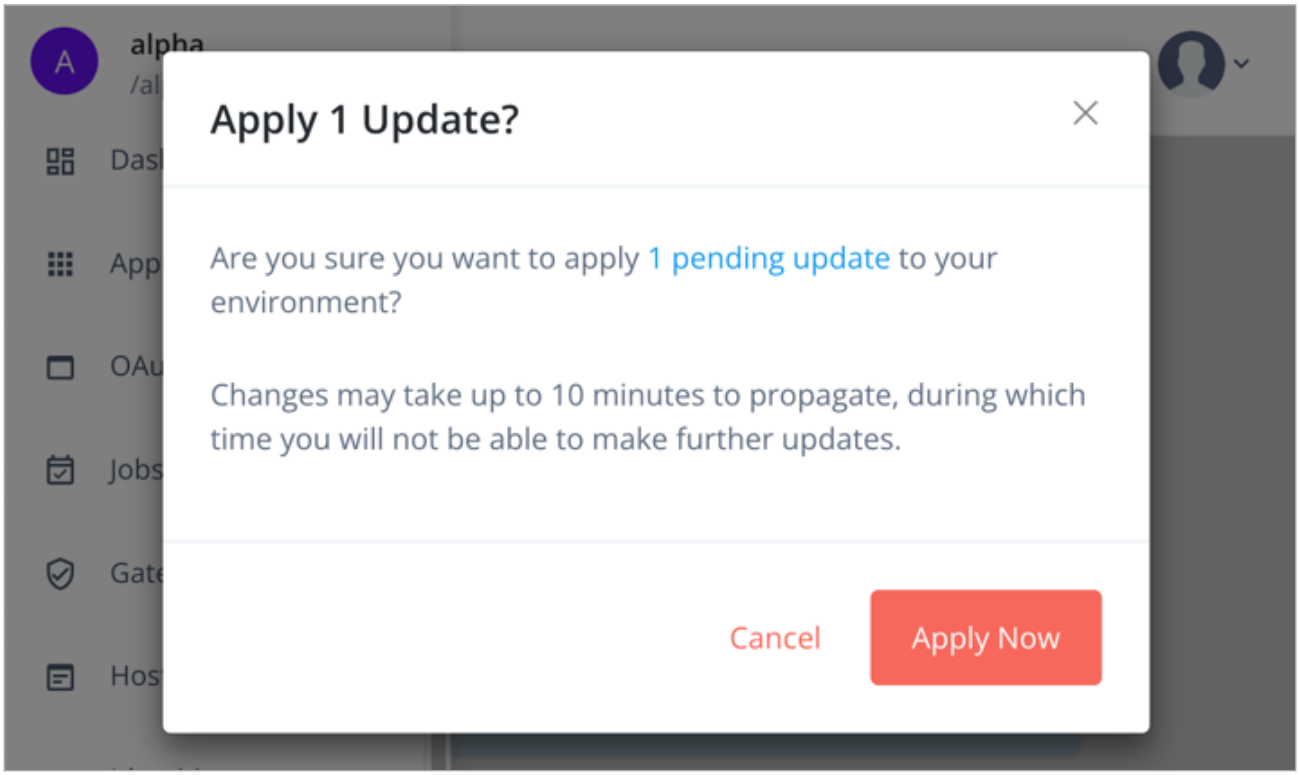


Figure 2. Apply updated secrets in Advanced Identity Cloud

7. Click **Apply Now**.

   Advanced Identity Cloud propagates the new secret and its value to all servers. You **must** wait until the secrets have propagated throughout the environment before attempting to use the secret.

   The **Environment Secrets & Variables** page displays the following message while the update is in progress:

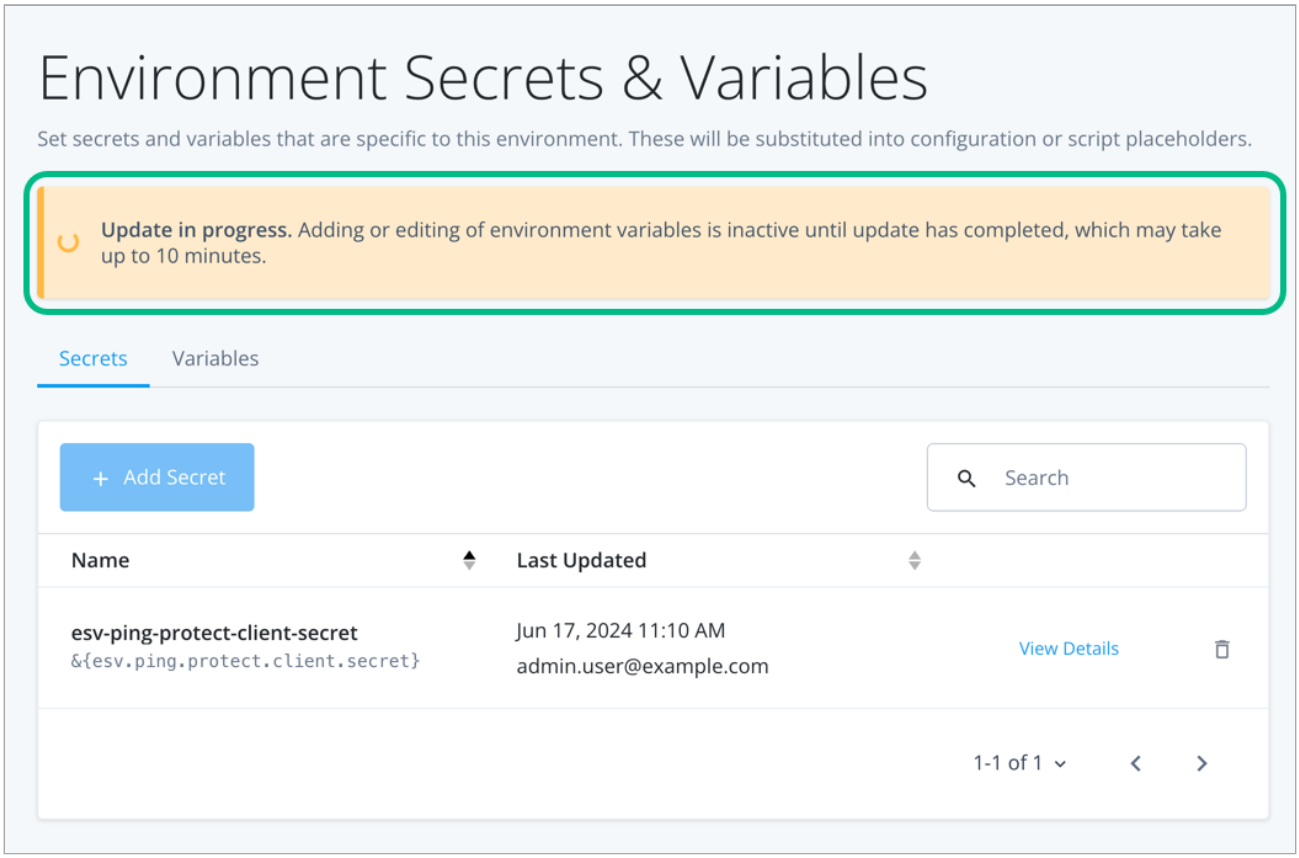


Figure 3. Propagating secrets in progress in Advanced Identity Cloud.

Self-managed AM

For information on adding secret values for use in services in a self-managed AM instance, refer to Create key aliases in the AM documentation.

Configure the PingOne worker service

To configure the PingOne worker service:

1. If you are using PingOne Advanced Identity Cloud, in the administration console navigate to **Native Consoles** > **Access Management**.
2. In the AM admin UI, click **Services**.
3. If the **PingOne Worker Service** is in the list of services, select it.
4. If you do not yet have a **PingOne Worker Service**:
    1. Click **+ Add a Service**.
    2. In **Choose a service type**, select `PingOne Worker Service`, and then click **Create**.
5. On the **Secondary Configurations** tab, click **+ Add a Secondary Configuration**.
6. On the **New workers configuration** page:
    1. Enter a **Name** for the configuration.

For example, `SDK PingOne Worker`.

You use this value when you configure an authentication journey that performs risk evaluations.

2. In **Client ID**, enter the client ID of the PingOne Worker application you created earlier.
3. In **Client Secret Label Identifier**, enter an identifier to create a specific secret label to represent the client secret of the worker application.

   For example, `workerAppClientSecret`.

   The secret label uses the template `am.services.pingone.worker.identifier.clientsecret` where identifier is the **Client Secret Label Identifier** value.

   This field can only contain characters `a-z`, `A-Z`, `0-9`, and `.` and can't start or end with a period.
4. In **Environment ID**, enter the environment ID containing the PingOne Worker application you created earlier.
5. Click **Create**

7. On the **Workers Configuration** page, ensure that the **PingOne API Server URL** and **PingOne Authorization Server URL** are correct for the region of your PingOne servers:

PingOne URLs by region

Region	Authorization URL	API URL
North America (Excluding Canada)	`https://auth.pingone.com`	`https://api.pingone.com/v1`
Canada	`https://auth.pingone.ca`	`https://api.pingone.ca/v1`
Europe	`https://auth.pingone.eu`	`https://api.pingone.eu/v1`
Asia-Pacific	`https://auth.pingone.asia`	`https://api.pingone.asia/v1`

8. Confirm your configuration resembles the image below, and then click **Save changes**.

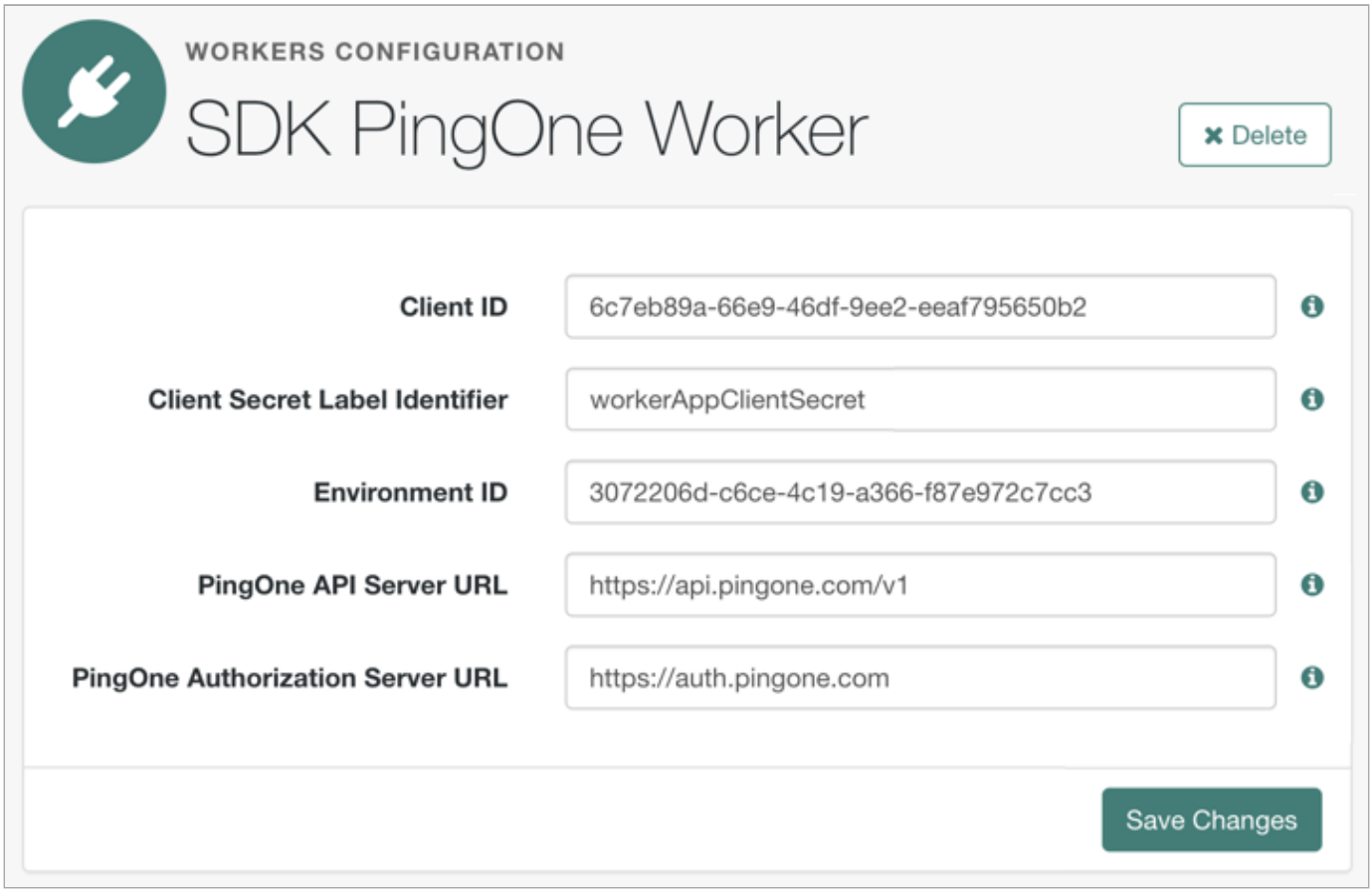


Figure 4. Example worker application in PingOne

Map the Client Secret Label Identifier to a secret

To make the client secret available to the PingOne Worker Service, you must map the secret to the ID created.

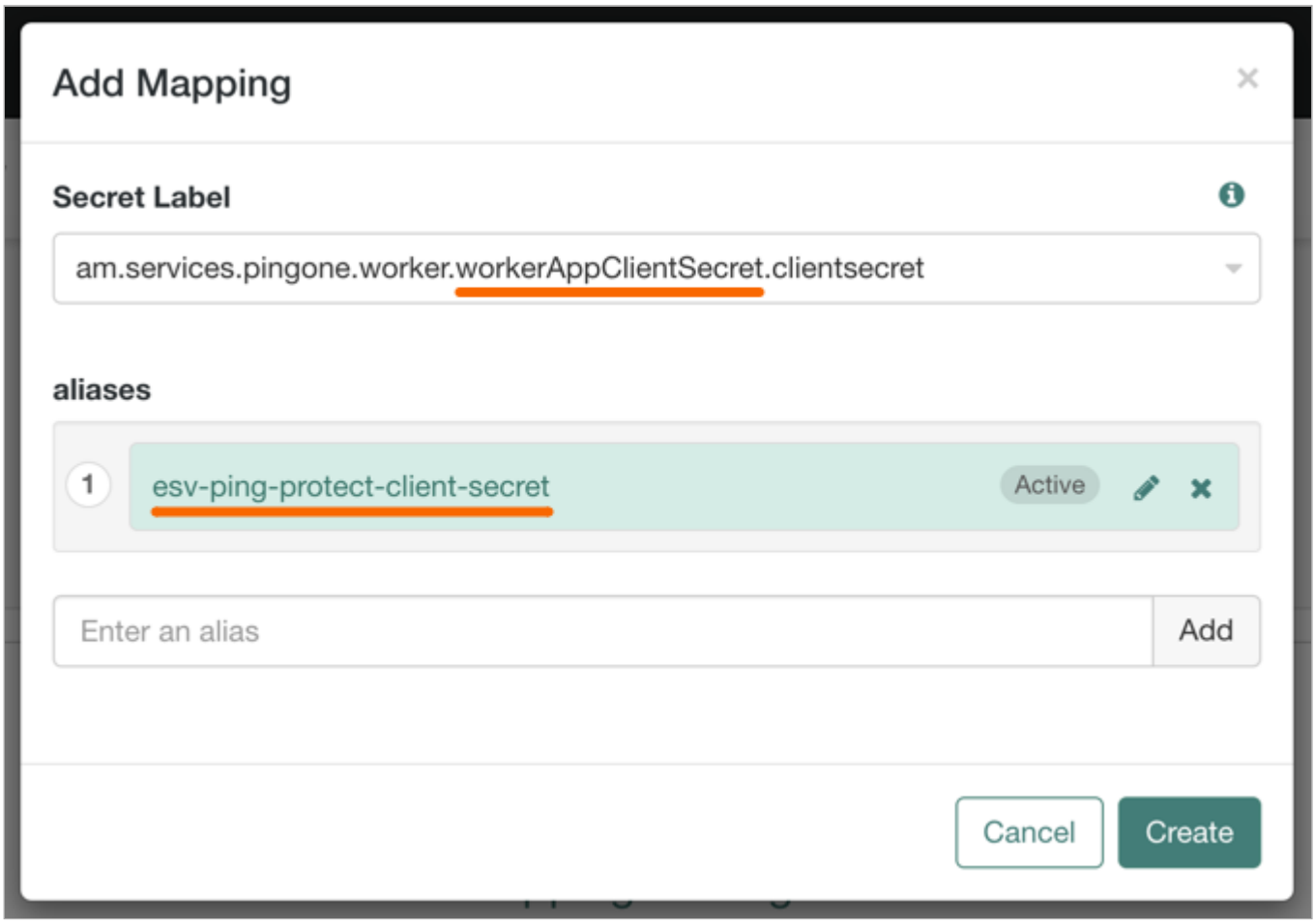
Map secrets in Advanced Identity Cloud

1. In the Advanced Identity Cloud admin UI, click **Native Consoles** > **Access Management**.
2. In the AM admin UI (native console), go to **Realm** > **Secret Stores**.
3. Click the **ESV** secret store, then click **Mappings**.
4. Click **+ Add Mapping**.
    1. In **Secret Label**, select the label generated when you entered the **Client Secret Label Identifier** previously.

       For example, `am.services.pingone.worker.workerAppClientSecret.clientsecret`.
    2. In **aliases**, enter the name of the ESV secret you created earlier, including the `esv-` prefix, and then click **Add**.

       For example, `esv-ping-protect-client-secret`

The result resembles the following:



5. Click **Create**.

To learn more about mapping secrets and label identifiers in Advanced Identity Cloud, refer to Secret labels.

Map secrets in self-managed AM

To learn about mapping secrets in self-managed AM, refer to Map and rotate secrets.

You have now configured the PingOne Worker service in your server. You can now Configure a journey to perform PingOne Protect risk evaluations.

Configure a journey to perform PingOne Protect risk evaluations

To make risk evaluations in PingOne, you must configure an authentication journey in your server.

The following table covers the authentication nodes and callbacks for integrating your authentication journeys with PingOne Protect.

Node	Callback	Description
PingOne Protect Initialization node	PingOneProtectInitiateCallback	Instruct the embedded PingOne Signals SDK to start gathering contextual information.
PingOne Protect Evaluation node	PingOneProtectEvaluationCallback	Returns contextual information that the server can send to your PingOne Protect instance to perform a risk evaluation.
PingOne Protect Result node	Non-interactive	Inform the PingOne Protect instance about the status of the transaction.

In your server, log in as an administrator and create a new authentication journey similar to the following example:

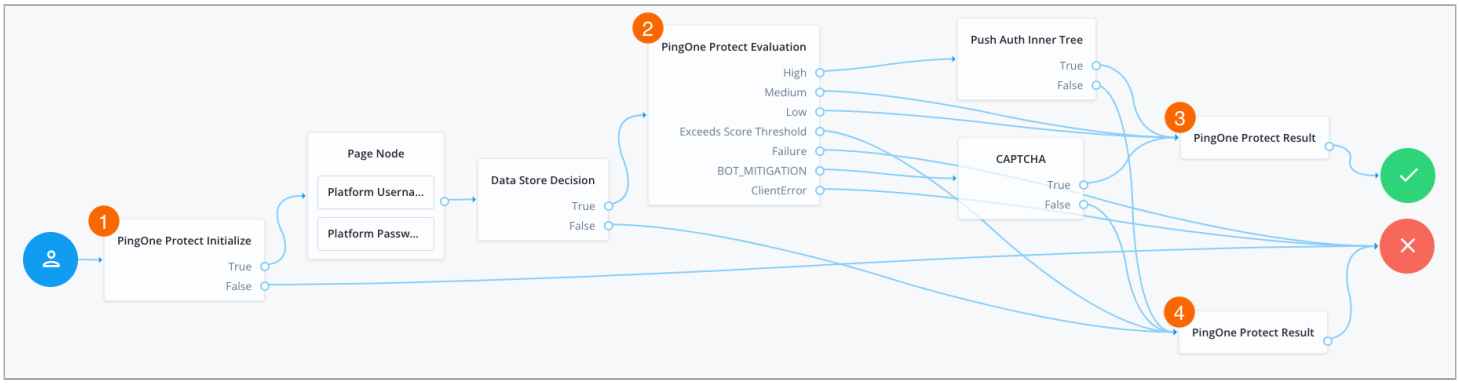


Figure 5. An example PingOne Protect journey

- The PingOne Protect Initialize node 1 instructs the SDK to initialize the PingOne Protect Signals API with the configured properties.

  Initialize the PingOne Protect Signals API as early in the journey as possible, before any user interaction.

  This enables it to gather sufficient contextual data to make an informed risk evaluation.

  > **Tip**
  >
  > You can initialize the PingOne Protect Signals API whenever you want to start collecting data. This could be at application startup, or when a particular page or view is visited.
  > Learn more at initializing data collection.

- The user enters their credentials, which are verified against the identity store.
- The PingOne Protect Evaluation node 2 performs a risk evaluation against a risk policy in PingOne.

  The example journey continues depending on the outcome:

  `High`

  The journey requests that the user respond to a push notification.

`Medium` *or* `Low`

The risk is not significant, so no further authentication factors are required.

`Exceeds Score Threshold`

The score returned is higher than the configured threshold and is considered too risky to complete successfully.

`Failure`

The risk evaluation could not be completed, so the authentication attempt continues to the **Failure** node.

`BOT_MITIGATION`

The risk evaluation returned a recommended action to check for the presence of a human, so the journey continues to a CAPTCHA node.

`ClientError`

The client returned an error when attempting to capture the data to perform a risk evaluation, so the authentication attempt continues to the **Failure** node.

- An instance of the PingOne Protect Result node 3 returns the `Success` result to PingOne, which can be viewed in the audit console to help with analysis and risk policy tuning.
- A second instance of the PingOne Protect Result node 4 returns the `Failed` result to PingOne, which can be viewed in the audit console to help with analysis and risk policy tuning.

You have now configured a suitable authentication journey in your server. You can now proceed to Step 2. Configure the Ping (ForgeRock) Login Widget for PingOne Protect.

Step 2. Configure the Ping (ForgeRock) Login Widget for PingOne Protect

Integrating the Ping (ForgeRock) Login Widget with PingOne Protect enables you to perform risk evaluations during your customer's journey.

Complete the following tasks to fully integrate with PingOne Protect:

1. Initialize data collection
2. Pause and resume behavioral data capture
3. Return collected data for a risk evaluation

Initialize data collection

You must initialize the PingOne Signals SDK so that it collects the data needed to evaluate risk.

The earlier you can initialize the PingOne Signals SDK, the more data it can collect to make a risk evaluation.

There are two options for initializing the PingOne Signals SDK in the Ping (ForgeRock) Login Widget:

1. The Ping (ForgeRock) Login Widget automatically initializes the PingOne Signals SDK on receipt of a `PingOneProtectInitializeCallback` callback from a journey you have started.

2. Manually initialize the PingOne Signals SDK, import the module and pass in any configuration parameters you need, as follows:

```
import Widget, { configuration, journey, protect } from '@forgerock/login-widget';

new Widget({ target: widgetEl });

// Start PingOne Protect Signals SDK
await protect.start({
  envId: 3072206d-c6ce-ch15-m0nd-f87e972c7cc3,
  behavioralDataCollection: true,
  consoleLogEnabled: true,
});
```

The PingOne Signals SDK supports a number of parameters which you can supply yourself, or are contained in the `PingOneProtectInitializeCallback` callback.

Parameter			Description
Android	*iOS*	*JavaScript*	
`envID`			Required. Your PingOne environment identifier.
`deviceAttributesToIgnore`			Optional. A list of device attributes to ignore when collecting device signals. For example, `AUDIO_OUTPUT_DEVICES` or `IS_ACCEPT_COOKIES`.
`isBehavioralDataCollection`	`behavioralDataCollection`		When `true`, collect behavioral data. Default is `true`.
`isConsoleLogEnabled`	`consoleLogEnabled`		When `true`, output SDK log messages in the developer console. Default is `false`.
`isLazyMetadata`	`lazyMetadata`		When `true`, calculate metadata on demand rather than automatically after calling `start`. Default is `false`.
N/A		`deviceKeyRsynIntervals`	Number of days that device attestation can rely upon the device fallback key. Default: `14`
N/A		`disableHub`	When `true`, the client stores device data in the browser's `localStorage` only. When `false` the client uses an iframe. Default is `false`.

N/A	`disableTags`	When `true`, the client does not collect tag data. Tags are used to record the pages the user visited, forming a browsing history. Default is `false`.
N/A	`enableTrust`	When `true`, tie the device payload to a non-extractable crypto key stored in the browser for content authenticity verification. Default is `false`.
N/A	`externalIdentifiers`	Optional. A list of custom identifiers that are associated with the device entity in PingOne Protect.
N/A	`hubUrl`	Optional. The iframe URL to use for cross-storage device IDs.
N/A	`waitForWindowLoad`	When `true`, initialize the SDK on the `load` event, instead of the `DOMContentLoaded` event. Default is `true`.

Return collected data for a risk evaluation

To perform risk evaluations, the PingOne server requires the captured data.

There are two options for returning data in the Ping (ForgeRock) Login Widget:

1. On receipt of a `PingOneProtectEvaluationCallback` callback within a journey, the Ping (ForgeRock) Login Widget automatically returns the captured data.
2. Use the `getData()` method to manually return the captured data:

```
import Widget, { configuration, journey, protect } from '@forgerock/login-widget';

new Widget({ target: widgetEl });

// Start PingOne Protect Signals SDK
await protect.start({
  envId: 3072206d-c6ce-ch15-m0nd-f87e972c7cc3,
  behavioralDataCollection: true,
  consoleLogEnabled: true,
});

// Return gathered data to the server
await protect.getData();
```

Pause and resume behavioral data capture

The PingOne Protect Signals SDK can capture behavioral data, such as how the user interacts with the app, to help when performing evaluations.

There are scenarios where you might want to pause the collection of behavioral data. For example, the user might not be interacting with the app, or you only want to use device attribute data to be considered when performing PingOne Protect evaluations. You can then resume behavioral data collection when required.

There are two options for pausing and resuming behavioral data capture in the Ping (ForgeRock) Login Widget:

1. The `PingOneProtectEvaluationCallback` callback can include a flag to pause or resume behavioral capture, which the Ping (ForgeRock) Login Widget automatically responds to.
2. Use the `pauseBehavioralData()` and `resumeBehavioralData()` methods to manually pause or resume the capture of behavioral data:

```
import Widget, { configuration, journey, protect } from '@forgerock/login-widget';

new Widget({ target: widgetEl });

// Start PingOne Protect Signals SDK
await protect.start({
  envId: 3072206d-c6ce-ch15-m0nd-f87e972c7cc3,
  behavioralDataCollection: true,
  consoleLogEnabled: true,
});

// Return gathered data to the server
await protect.getData();

// Pause behavioral data collection
protect.pauseBehavioralData();

// Resume behavioral data collection
protect.resumeBehavioralData();
```

Integrate the Ping (ForgeRock) Login Widget into a React app

In this tutorial, you will learn how to integrate the Ping (ForgeRock) Login Widget into a simple React app that you scaffold using Vite.

You install the Ping (ForgeRock) Login Widget using `npm`, add an element to the HTML file for mounting the modal form factor, and wrap the app's CSS in a layer.

With the app prepared, you then import and instantiate various components of the Ping (ForgeRock) Login Widget to start a journey. You subscribe to the events the Ping (ForgeRock) Login Widget emits so that the app can respond and display the appropriate UI.

When you have successfully authenticated a user, you add code to log the user out and invalidate their tokens, as well as update the UI to alter the button state.

Requirements

1. Node 18+
2. NPM 8+

Configure your server

Configure your PingOne Advanced Identity Cloud or self-managed PingAM server by following the steps in the Ping (ForgeRock) Login Widget Tutorial.

- When creating the OAuth 2.0 client, add the URL that you are using to host the app to the **Sign-In URLs** property.

> **Tip**
>
> The URL is output to the console when you run the `npm run dev` command, and defaults to `http://localhost:5173/`

- If your instance has the default **Login** journey, you can use that instead of creating a new journey as described in the tutorial.

Create a Vite app

1. In a terminal window, create a Vite app with React as the template:

```
npm create vite@latest login-widget-react-demo -- --template react
```

   For more information, refer to Scaffolding Your First Vite Project in the Vite developer documentation.

2. When completed, change to the new directory, for example `login-widget-react-demo`, and then install dependencies with `npm`:

```
npm install
```

3. Run the app in developer mode:

```
npm run dev
```

4. In a web browser, open the URL output by the previous command to render the app. The URL is usually `http://localhost:5173`

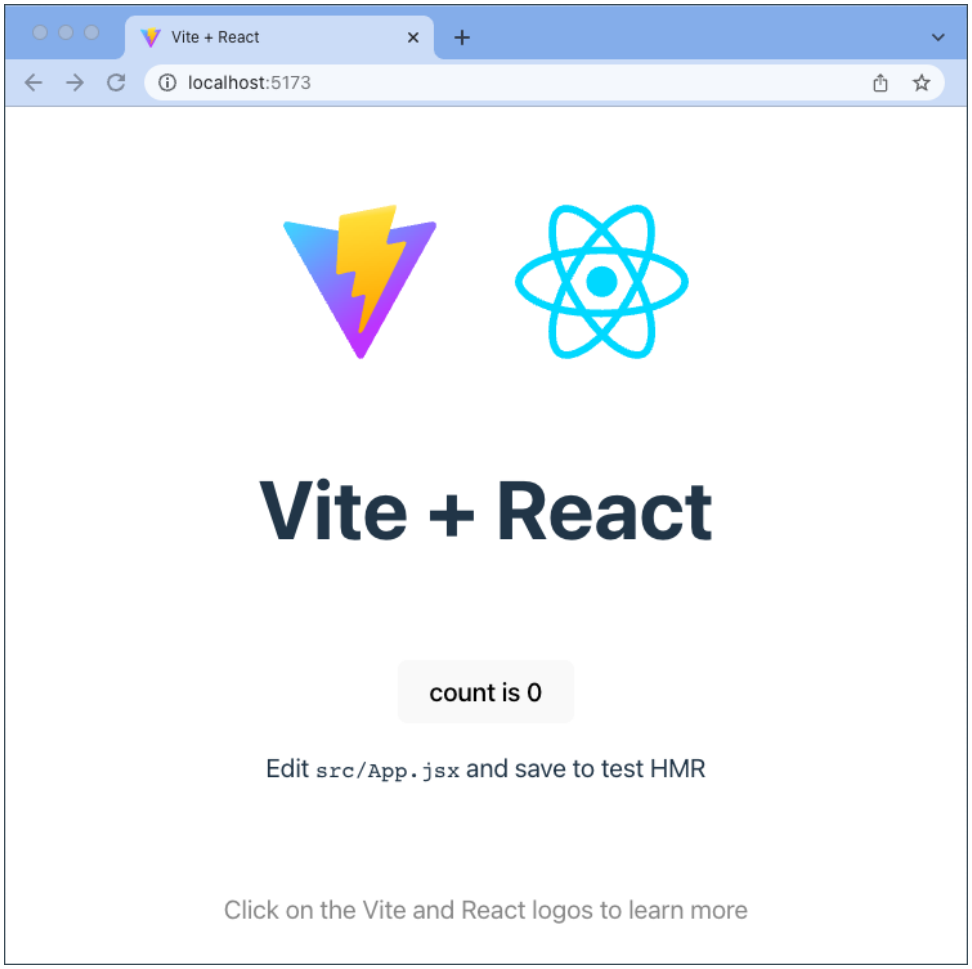


Figure 1. Example Vite + React app

> **Tip**
>
> Use a different browser for development testing than the one you use to log into PingOne Advanced Identity Cloud or PingAM.
> This prevents admin user and test user sessions colliding and causing unexpected authentication failures.

Install the Ping (ForgeRock) Login Widget

In a new terminal window, install the Ping (ForgeRock) Login Widget using `npm`:

```
npm install @forgerock/login-widget
```

Prepare the HTML

In your preferred IDE, open the directory where you created the Vite app, and then open the `index.html` file.

To implement the modal form factor, create a root element to contain the Ping (ForgeRock) Login Widget.

Add `<div id="widget-root"></div>` toward the bottom of the `<body>` element but before the `<script>` tag:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Vite + React</title>
  </head>
  <body>
    <div id="root"></div>
    <!-- Widget mount point -->
    <div id="widget-root"></div>
    <script type="module" src="/src/main.jsx"></script>
  </body>
</html>
```

Prepare the CSS

You should wrap the app's CSS using `@layer`. This helps control the CSS cascade.

To wrap the app's CSS, in your IDE, open `src/index.css` and `src/App.css` and wrap them both with the following code:

```
@layer app {
    /* existing styles */
    #root {
        max-width: 1280px;
        margin: 0 auto;
        padding: 2rem;
        text-align: center;
    }

    .logo {
        height: 6em;
        padding: 1.5em;
        will-change: filter;
        transition: filter 300ms;
    }
    /* ... */
}
```

You can then specify the order of the various layers as follows:

```
<style>
  @layer app;
  /* List the Widget layers last */
  @layer 'fr-widget.base';
  @layer 'fr-widget.utilities';
  @layer 'fr-widget.components';
  @layer 'fr-widget.variants';
</style>
```

Import and configure the Ping (ForgeRock) Login Widget

In your IDE, open the top-level application file, often called `App.jsx`.

Import the `Widget` class, the `configuration` module, and the CSS:

```
import Widget, { configuration } from '@forgerock/login-widget';
import '@forgerock/login-widget/widget.css';
```

Add a call to the `configuration` method within your `App` function component and save off the return value to a `config` variable for later use.

This internally prepares the `Widget` for use.

```
function App() {
  const [count, setCount] = useState(0);

  // Initiate all the Widget modules
  const config = configuration();

  // ...
```

Instantiate and mount the Ping (ForgeRock) Login Widget

To continue, you need to import `useEffect` from the React library. This is to control the execution of a number of statements you are going to write.

After importing `useEffect`, write it into the component with an empty dependency array:

```
import React, { useEffect, useState } from 'react';

// ...

function App() {

  // ...

  useEffect(() => {}, []);

  // ...
```

> **Note**
>
> The empty dependency array is to tell React this has no dependencies at this point and should only run once.

Now that you have the `useEffect` written, follow these steps:

1. Instantiate the `Widget` class within `useEffect`
2. In the arguments, pass an object with a `target` property that specifies the DOM element you created in an earlier step

3. Assign its return value to a `widget` variable
4. Return a function that calls `widget.$destroy()`

```
useEffect(() => {
  const widget = new Widget({ target: document.getElementById('widget-root') });

  return () => {
    widget.$destroy();
  };
}, []);
```

> **Note**
>
> The reason for the returned function is for proper clean up when the React component unmounts.
> If it remounts, you do not get two widgets added to the DOM.

In your browser, the app doesn't look any different. This is because the Widget, by default, is invisible at startup.

To ensure it is working as expected, inspect the DOM in the browser developer tools.

Open the `<div id="widget-root">` element in the DOM, and you should see the Ping (ForgeRock) Login Widget mounted within it:

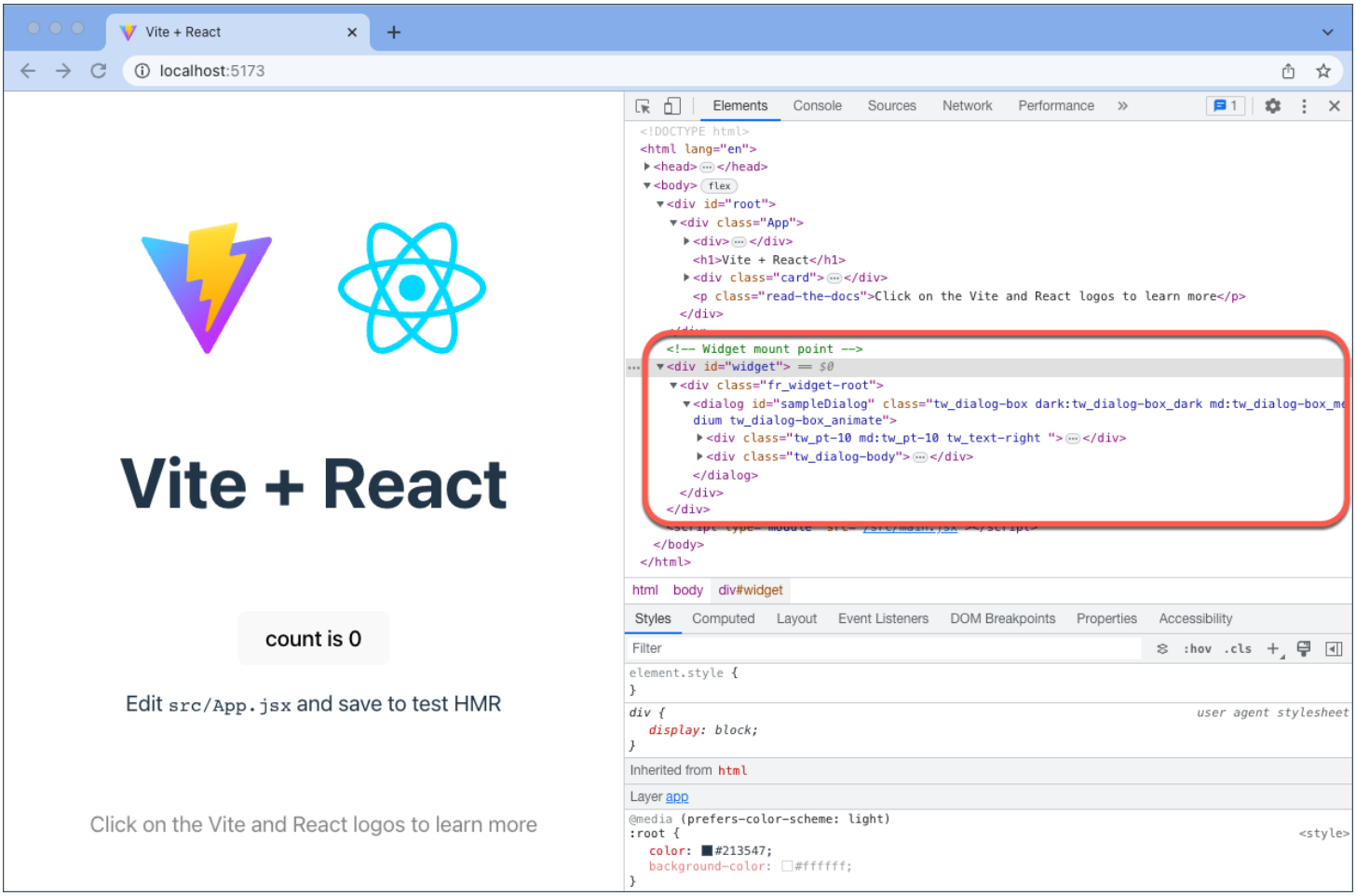


Figure 2. Instantiated and mounted modal form factor

Controlling the component

An invisible Ping (ForgeRock) Login Widget is not all that useful, so your next task is to pull in the `component` module to manage the component's events.

1. Add the `component` module to the list of imports from the `@forgerock/login-widget`
2. Call the `component` function just under the `configuration` function
3. Assign its return value to a `componentEvents` variable:

```
import Widget, { component, configuration } from '@forgerock/login-widget';

// ...

function App() {
  // ...

  const config = configuration();
  const componentEvents = component();

  // ...
```

Now that you have a reference to the `component` events observable, you can trigger an event such as `open`, and you can also listen for events.

Before calling the `open` method, repurpose the existing **count is 0** button within the `App` component.

1. Within the button's `onClick` handler, change the `setCount` function to `componentEvents.open`
2. Change the button text to read "Login"

The result resembles the following:

```
<button
  onClick={() => {
    componentEvents.open();
  }}>
  Login
</button>
```

You can now revisit your test browser and click the **Login** button. The modal opens and displays a "spinner" animating on repeat.

This is expected, because the Ping (ForgeRock) Login Widget does not yet have any information to render.

Click the button in the top-right to close the modal. The modal should be dismissed as expected.

Now that you have the modal mounted and functional, move on to the next step which configures the Ping (ForgeRock) Login Widget to be able to call the authorization server to get authentication data.

Calling the authorization server

Before the Ping (ForgeRock) Login Widget can connect you need to use the `config` variable you created earlier.

Call its `set` method within the exiting `useEffect`, and provide the configuration values for your server:

```
useEffect(() => {

  config.set({
    forgerock: {
      serverConfig: {
        baseUrl: 'https://openam-forgerock-sdks.forgeblocks.com/am',
        timeout: 3000,
      },
    },
  });

  const widget = new Widget({ target: document.getElementById('widget-root')});

  // ...
```

Now that you have the Ping (ForgeRock) Login Widget configured, add the `journey` module to the list of imports so that you can start the authentication flow:

```
import Widget, {
  component,
  configuration,
  journey,
} from '@forgerock/login-widget';
```

Execute the `journey` function and assign its returned value to a `journeyEvents` variable. Do this underneath the other existing "event" variables:

```
import Widget, { component, configuration, journey } from '@forgerock/login-widget';

// ...

function App() {
  // ...

  const config = configuration();
  const componentEvents = component();
  const journeyEvents = journey();

  // ...
```

This new observable provides access to journey events. Within the **Login** button's `onClick` handler add the `start` method.

Now, when you open the modal, you also call `start` to request the first authentication step from the server.

```
<button onClick={() => {
    journeyEvents.start();
    componentEvents.open();
  }>
  Login
</button>
```

You are now capable of authenticating a user. With an existing user, authenticate as that user and see what happens.

If successful, you'll notice the modal dismisses itself but your app is not capturing anything from this action. Proceed to the next step to capture user data.

Authenticating a user

There are multiple ways to capture the event of a successful login and access the user information.

In this guide, you use the `journeyEvents` observable created previously.

Within the existing `useEffect` function:

1. Call the `subscribe` method and assign its return value to an unsubscribe variable
2. Pass in a function that logs the emitted events to the console
3. Call the unsubscribe function within the return function of `useEffect`

```
// ...

useEffect(() => {
  // ...

  const widget = new Widget({ target: document.getElementById('widget-root') });

  const journeyEventsUnsub = journeyEvents.subscribe((event) => {
    console.log(event);
  });

  return () => {
    widget.$destroy();
    journeyEventsUnsub();
  };
}, []);
```

Note

Unsubscribing from the observable is important to avoid memory leaks if the component mounts and unmounts frequently.

Revisit your app in the test browser, but remove all of the browser's cookies and Web Storage to ensure you have a fresh start.

Tip

In Chromium browsers, you can find it under the "Application" tab of the developer tools.
In Firefox and Safari, you can find it under the "Storage" tab.

Once you have deleted all the cookies and storage, refresh the browser and try to log in your test user.

You will notice in the developer tools console that a lot of events are emitted.

Initially, you may not have much need for all this data, but over time, this information might become more valuable to you.

To narrow down all of this information, capture just one piece of the event: the user response after successfully logging in.

To do that, you can add a conditional, as follows:

- Add an `if` condition within the `subscribe` callback function that tests for the existence of the user response.

```
const journeyEventsUnsub = journeyEvents.subscribe((event) => {
  if (event.user.response) {
    console.log(event.user.response);
  }
});
```

With the above condition, the Ping (ForgeRock) Login Widget only writes out the user information when it's *truthy*. This helps narrow down the information to what is useful right now.

Remove all the cookies and Web Storage again and refresh the page. Try logging in again, and you should see only one log of the user information when it's available:

Example `user.event.response` output

```
{
    email: 'sdk.demo-user@example.com',
    sub: '54c77653-dc88-48fb-ac6b-d5078ebe9fb0',
    subname: '54c77653-dc88-48fb-ac6b-d5078ebe9fb0'
}
```

Next, repurpose the `useState` hook that's already used in the component to save the user information.

1. Change the zeroth index of the returned value from `count` to `userInfo`
2. Change the first index of the returned value from `setCount` to `setUserInfo`
3. Change the default value passed into the `useState` from `0` to `null`
4. Change the condition from just truthy to `userInfo !== event.user.response`
5. Replace the `console.log` with the `setUserInfo` function
6. Add the `userInfo` variable in the dependency array of the `useEffect`

The top part of your `App` function component should resemble the following:

```
function App() {
  const [userInfo, setUserInfo] = useState(null);

  // Initiate all the Widget modules
  const config = configuration();
  const componentEvents = component();
  const journeyEvents = journey();

  useEffect(() => {
    // Set the Widget's configuration
    config.set({
      forgerock: {
        serverConfig: {
          baseUrl: 'https://openam-forgerock-sdks.forgeblocks.com/am',
          timeout: 3000,
        }
      }
    });

    // Instantiate the Widget and assign it to a variable
    const widget = new Widget({ target: document.getElementById('widget-root') });

    // Subscribe to journey observable and assign unsubscribe function to variable
    const journeyEventsUnsub = journeyEvents.subscribe((event) => {
      if (userInfo !== event.user.response) {
        setUserInfo(event.user.response);
      }
    });

    // Return a function that destroys the Widget and unsubscribes from the journey observable
    return () => {
      widget.$destroy();
      journeyEventsUnsub();
    };
  }, [userInfo]);

  // ...
```

Note

The condition comparing `userInfo` to `event.user.response` reduces the number of times the `setUserInfo` is called as it will now only be called if what is set in the hook is different than what is emitted from the Ping (ForgeRock) Login Widget.

Now that you have the user data set into the React component, print it out into the DOM.

1. Replace the paragraph tag containing the text `Edit <code>src/App.jsx</code> and save to test HMR` with a `<pre>` tag
2. Within the `<pre>` tag, write a pair of braces: `{}`
3. Within these braces, use the `JSON.stringify` method to serialize the `userInfo` value

Your JSX should now look like this:

```
<pre>{JSON.stringify(userInfo, null, ' ')}</pre>
```

Tip

The `null` and `' '` (literal space character) help format the JSON to be more reader-friendly.

After clearing the browser data, try logging the user in and observe the user info get rendered onto the page after success.

Figure 3. User info displaying after successful log in

Logging a user out

The final step is to log the user out, clearing all the user-related cookies, storage, and cache.

To do this, add the `user` module to the list of imports from the Ping (ForgeRock) Login Widget:

```
import Widget, {
  configuration,
  component,
  journey,
  user,
} from '@forgerock/login-widget';
```

Next, configure the app to display the button as a **Login** button when the user has not yet authenticated and a **Logout** button when the user has already logged in:

1. Wrap the button element with braces containing a ternary, using the *falsiness* of the `userInfo` as the condition
2. When no `userInfo` exists—the user is logged out—render the **Login** button
3. Write a **Logout** button with an `onClick` handler to run the `user.logout` function

The resulting JSX should resemble this:

```
import Widget, { user, component, configuration, journey } from '@forgerock/login-widget';

// ...

<h1>Vite + React</h1>
<div className="card">
  {
    !userInfo ? (
      <button
        onClick={() => {
          journeyEvents.start();
          componentEvents.open();
        }}>
        Login
      </button>
    ) : (
      <button
        onClick={() => {
          user.logout();
        }}>
        Logout
      </button>
    )
  }
  <pre>{JSON.stringify(userInfo, null, ' ')}</pre>
</div>
// ...
```

Tip

You do not have to add code to reset the `userInfo` with the `setUserInfo` function, because you are already "listening" to events emitted from the `user` observable nested within the `journeyEvents` subscribe.

If your app is already reacting to the presence of user info, it should be rendering the **Logout** button already. Click it and observe the application reacting.

You should now be able to log a user in and out, with the app reacting to the changes in state.

API reference

This page lists the modules that the Ping (ForgeRock) Login Widget provides for use in your apps.

Widget

This is a compiled Svelte class. This is what instantiates the component, mounts it to the DOM, and sets up all the event listeners.

```
import Widget from '@forgerock/login-widget';

// Instantiate Widget
const widget = new Widget({
  target: widgetRootEl, // REQUIRED; Element mounted in DOM
  props: {
    type: 'modal', // OPTIONAL; "modal" or "inline"; "modal" is default
  },
});

// OPTIONAL; Remove widget from DOM and destroy component listeners
widget.$destroy();
```

Important

Call `$destroy()` if you instantiate the Ping (ForgeRock) Login Widget within a part of your application frequently created and destroyed.
We strongly encourage you to instantiate the modal form factor of the Ping (ForgeRock) Login Widget high up in your application code. Instantiate it close to the top-level file in a component that is created once and preserved.

Configuration

The Ping (ForgeRock) Login Widget requires information about the server instance it connects to, as well as OAuth 2.0 client configuration and other settings.

For information on setting up your server for use with the Ping (ForgeRock) Login Widget, refer to Prerequisites.

To provide these settings, import and use the `configuration` module and its `set()` method.

The Ping (ForgeRock) Login Widget uses the same underlying configuration properties as the main SDK.

```
import { configuration } from '@forgerock/login-widget';

const myConfig = configuration();
myConfig.set({
  forgerock: {
    /**
     * REQUIRED; SDK configuration object
     */
    serverConfig: {
      baseUrl: 'https://openam-forgerock-sdks.forgeblocks.com/am',
      timeout: 3000, // Number (in milliseconds); 3 to 5 seconds should be fine
    },
    /**
     * OPTIONAL, *BUT ENCOURAGED*, CONFIGURATION
     * Remaining config is optional with fallback values shown
     */
    clientId: 'sdkPublicClient', // String; defaults to 'WebLoginWidgetClient'
    realmPath: 'alpha', // String; defaults to 'alpha'
    redirectUri: window.location.href, // URL string; defaults to `window.location.href`
    scope: 'openid profile email address', // String; defaults to 'openid email'
  },
  /**
   * OPTIONAL; Pass custom content
   */
  content: {},
  /**
   * OPTIONAL; Provide link for terms and conditions page
   */
  links: {},
  /**
   * OPTIONAL; Provide style configuration
   */
  style: {},
  /**
   * OPTIONAL; Map HREFs to journeys or trees
   */
  journeys: {},
});
```

Content configuration options

Use the `content` configuration element to pass custom text content to the Ping (ForgeRock) Login Widget, replacing its default values.

Example `content` configuration

```
const myConfig = configuration();

myConfig.set({
     content: {
        "userName": "Identifier",
        "passwordCallback": "Passphrase",
        "nextButton": "Let's go!",
    },
});
```

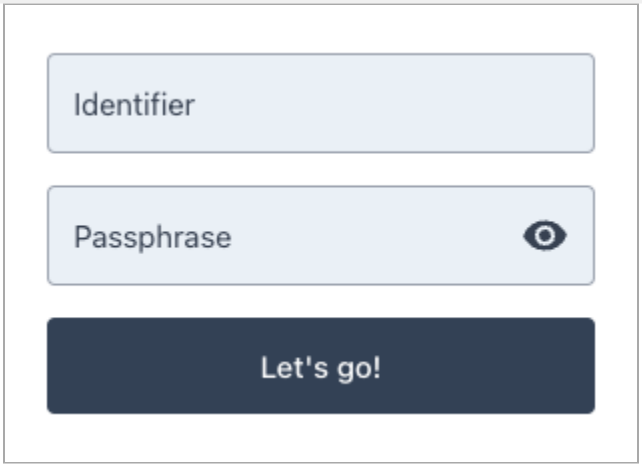


Figure 1. Result of example content configuration

For a list of the content you can override, refer to the en-us locale file in the Ping (ForgeRock) Web Login Framework repository.

Links configuration options

Use the `links` configuration element to set the full canonical URL to your terms and conditions page.

This should be a page hosted on your website or elsewhere within your app. Users are sent to this URL if they click to view the terms and conditions in the Ping (ForgeRock) Login Widget.

This supports the `TermsAndConditionsCallback` often used in registration journeys.

Example `links` configuration

```
const myConfig = configuration();

myConfig.set({
    links: {
        termsAndConditions: 'https://example.com/terms',
    },
});
```

Style configuration options

Use the `style` configuration element to configure the look and feel of the Ping (ForgeRock) Login Widget. This allows you to choose the type of labels used or provide a logo for the modal.

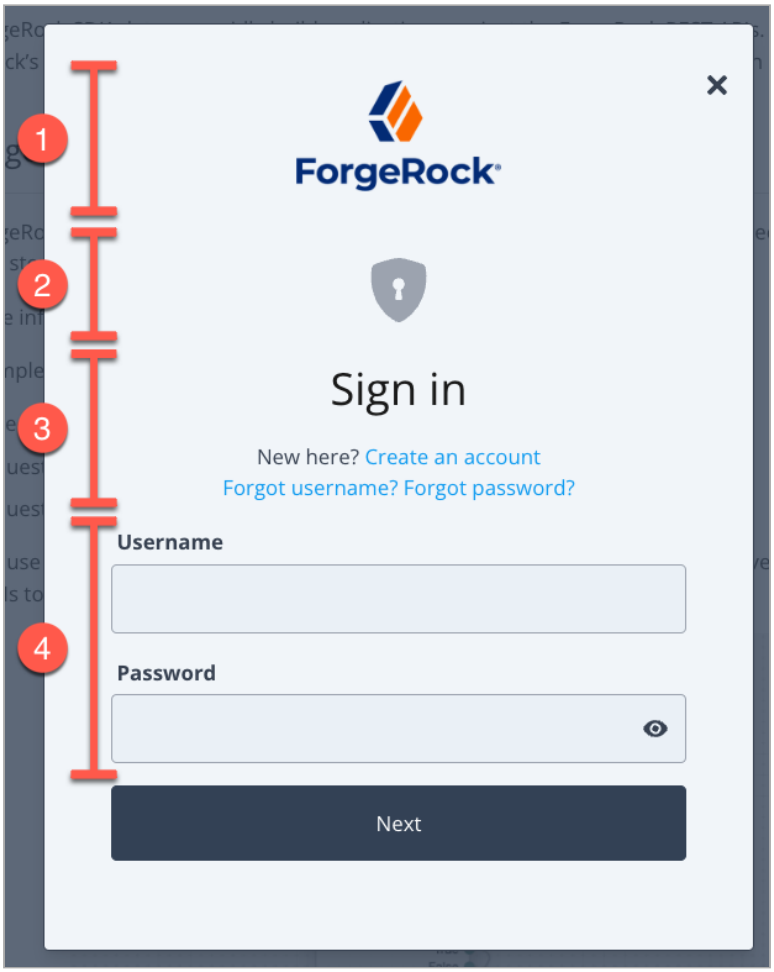


Figure 2. Use the `style` property to control aspects of the display

Key:

1. Use `style/logo` to add images for use in dark or light modes
2. Set `style/stage/icon` to `true` to render UI specific to certain **stage** parameter values. Supported **stage** values are:
    - `OneTimePassword` - enable the Ping (ForgeRock) Login Widget to display one-time password entry forms correctly.
    - `DefaultRegistration` - adds UI elements to the display most suitable for user self-registration forms.
    - `DefaultLogin` - adds UI elements to the display most suitable for user log in forms.
3. A section that displays the **Page Header** and **Page Description** fields from the page node configuration
4. To float labels above their respective fields, set `style/labels` to `floating`

Adding logos and enabling icons

```
const myConfig = configuration();

myConfig.set({
  style: {
    checksAndRadios: 'animated', // OPTIONAL; choices are 'animated' or 'standard'
    labels: 'floating', // OPTIONAL; choices are 'floating' or 'stacked'
    logo: {
      // OPTIONAL; only used with modal form factor
      dark: 'https://example.com/img/white-logo.png', // OPTIONAL; used if theme has a dark variant
      light: 'https://example.com/img/black-logo.png', // REQUIRED if logo property is provided; full URL
      height: 300, // OPTIONAL; number of pixels for providing additional controls to logo display
      width: 400, // OPTIONAL; number of pixels for providing additional controls to logo display
    },
    sections: {
      // OPTIONAL; only used with modal form factor
      header: false, // OPTIONAL; separate the logo section from the rest of the modal
    },
    stage: {
      icon: true, // OPTIONAL; displays generic icons for the provided stages
    },
  },
});
```

Note

The `logo` and `sections` properties only apply to the "modal" form factor and not the "inline".

Add a header section

Enabling the header section separates the logo or branding from the journey form.

If you set `header: true` within the `style/sections` property, the modal uses a section with a separating line, and extra space:

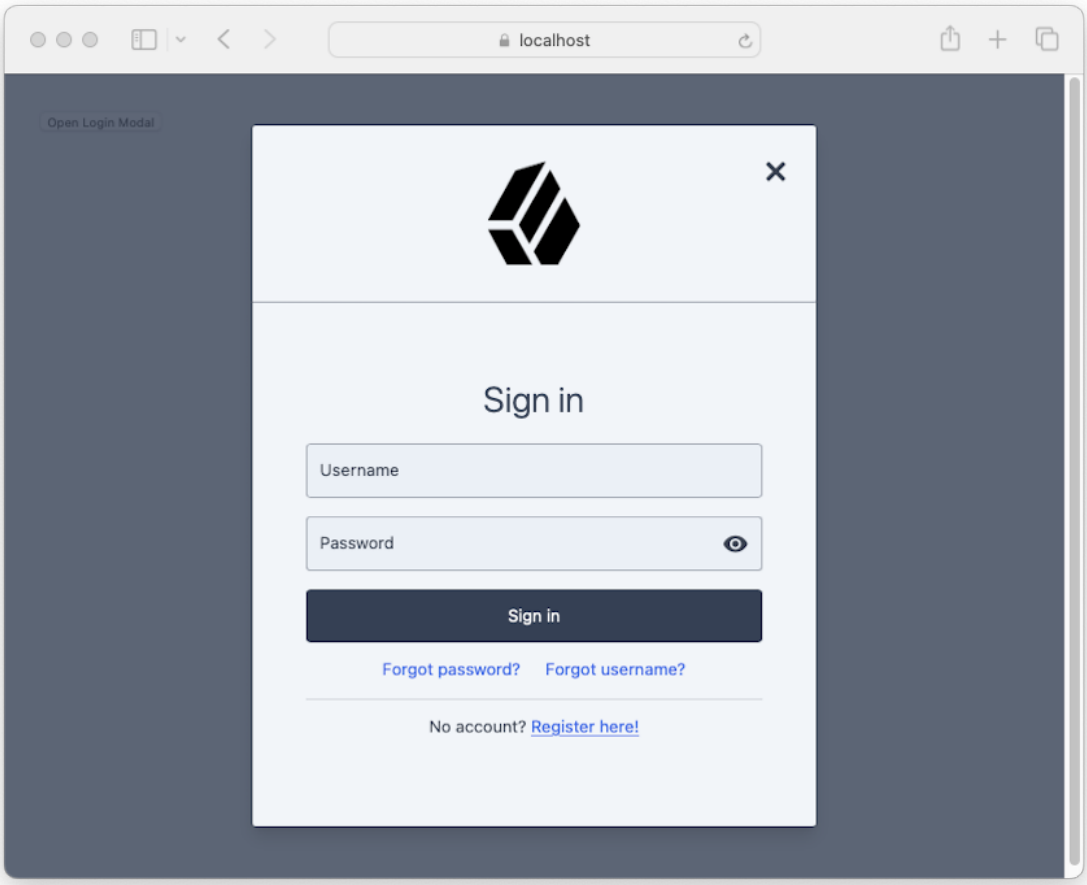


Figure 3. Modal form factor with header enabled

By default, the separating section is not enabled:

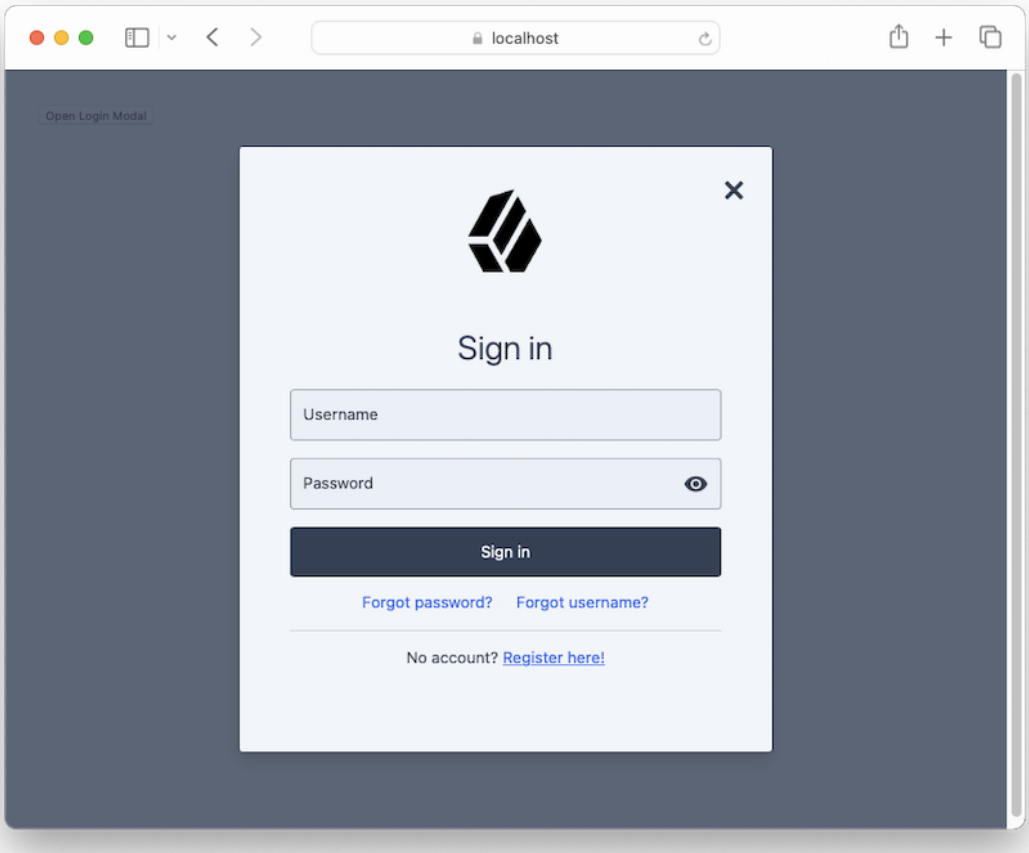


Figure 4. Default modal form factor with header disabled

Journeys configuration options

Use the `journeys` configuration element to map HREF values rendered within the Ping (ForgeRock) Login Widget to start a journey or authentication tree instead of visiting the URL.

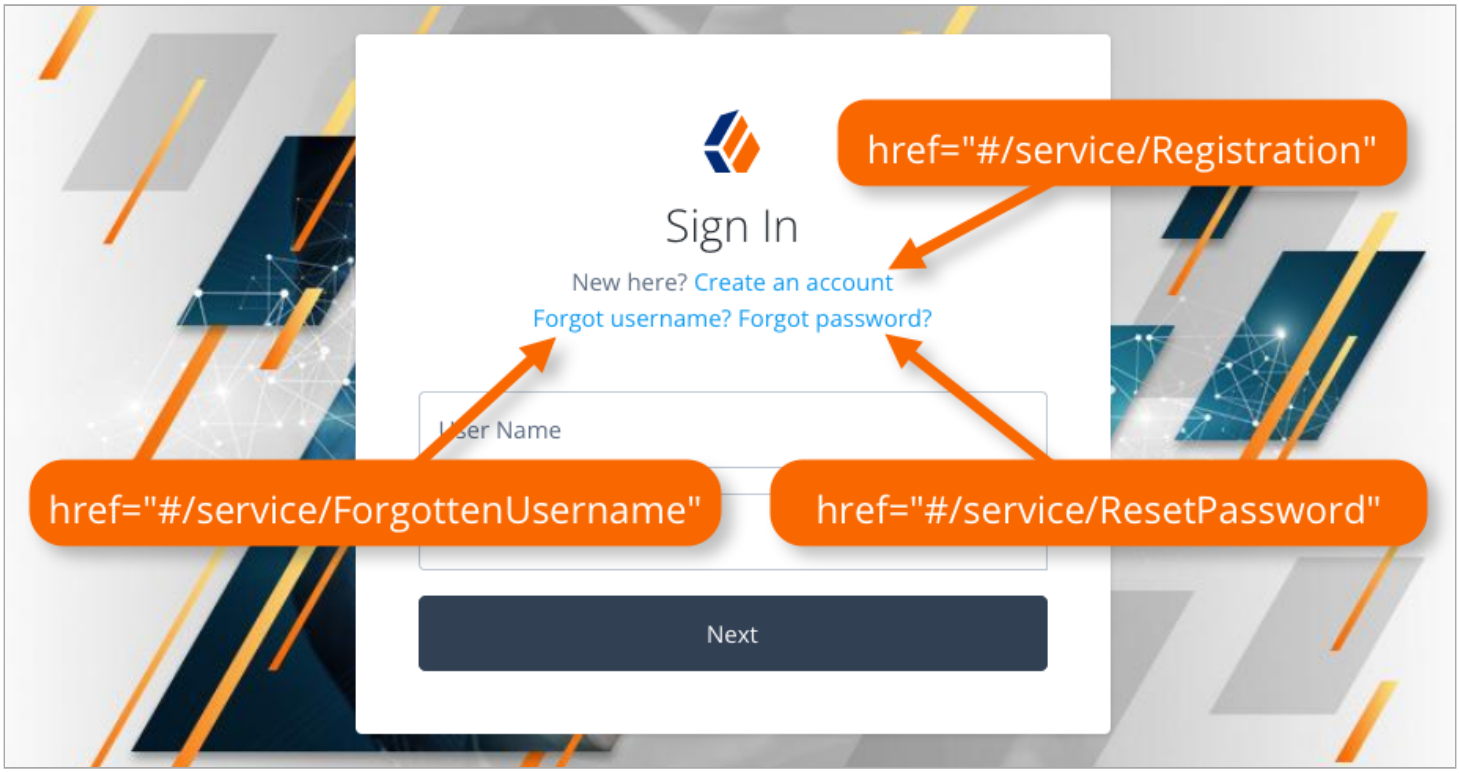


Figure 5. Example HREF values in a page node

The Ping (ForgeRock) Login Widget listens for click events on elements rendered within its container and compares the HREF to the configured mappings. If there is a match, it prevents the default action of visiting the URL and starts the journey configured in the mapping.

Mapping HREF strings to a journey

```
config.set({
  forgerock: {
    // SDK config
  },
  journeys: {
    forgetCookie: { // Any string, as long as it's not overriding a default mapping
      journey: 'ForgetCookie', // Must match actual journey name in the {fr_server}
      match: [ '#/service/ForgetCookie' ], // Array of strings that match `HREF` values (case-sensitive)
    }
  }
});
```

The Ping (ForgeRock) Login Widget has mappings configured internally to handle the links displayed in page nodes by default. These map the HREF values that are displayed by an out-of-the-box page node to corresponding journeys in an PingOne Advanced Identity Cloud tenant. You can override the mappings if required or add your own.

Default HREF strings to journey mappings

```
forgotPassword: {
    journey: 'ResetPassword',
    match: ['#/service/ResetPassword', '?journey=ResetPassword'],
},
forgotUsername: {
    journey: 'ForgottenUsername',
    match: ['#/service/ForgottenUsername', '?journey=ForgottenUsername'],
},
login: {
    journey: 'Login',
    match: ['#/service/Login', '?journey', '?journey=Login'],
},
register: {
    journey: 'Registration',
    match: ['#/service/Registration', '?journey=Registration'],
},
```

Component

Use the `component` module for subscribing to modal and inline form factor events and for opening and controlling the modal form factor.

Call the `component()` method and assign the result to a variable to receive the observable. Subscribe to the observable to listen and react to the state of the Ping (ForgeRock) Login Widget component.

```
import { component } from '@forgerock/login-widget';

// Initiate the component API
const componentEvents = component();

// Know when the component, both modal and inline has been mounted.
// When using the modal type, you will also receive open and close events.
// The property `reason` will be either "auto", "external", or "user"

const unsubComponentEvents = componentEvents.subscribe((event) => {
    /* Run anything you want */
});

// Open the modal
componentEvents.open();

// Close the modal
componentEvents.close();

// Recommended: call when your UI component is destroyed
unsubComponentEvents();
```

Schema for `component` events

The schema for `component` events is as follows:

Schema for `component` events

```
{
    lastAction: null, // null or the most recent action; one of `close`, `open`, or `mount`
    error: null, // null or object with `code`, `message`, and `step` that failed
    mounted: false, // boolean
    open: null, // boolean for the modal form factor, or null for inline form factor
    reason: null, // string to describe the reason for the event
    type: null, // 'modal' or 'inline'
}
```

Use the `reason` value to determine why the modal has closed.

The possible `reason` values are:

`user`

The user closed the dialog within the UI

`auto`

The modal was closed because the user successfully authenticated

`external`

The application called the `close()` function

Journey

Use the `journey` module to manage interaction with authentication and self-service journeys.

```
import { journey } from '@forgerock/login-widget';

// Call to start the journey
// Optional config can be passed in, see below for more details
const journeyEvents = journey({
  oauth: true, // OPTIONAL; defaults to true; uses OAuth flow for acquiring tokens
  user: true, // OPTIONAL; default to true; returns user information from `userinfo` endpoint
});

// Start a journey
journeyEvents.start({
  forgerock: {}, // OPTIONAL; configuration overrides
  journey: 'Login', // OPTIONAL; specify the journey or tree you want to use
  resumeUrl: window.location.href, // OPTIONAL; the full URL for resuming a tree
  recaptchaAction: 'myCaptchaTag', // OPTIONAL; tag v3 reCAPTCHAs. Fallback to journey name.
  pingProtect: { // Set manually, or obtain from `PingOneProtectInitializeCallback` callback.
    // REQUIRED; Your {p1_name} environment identifier.
    envId: '3072206d-c6ce-4c19-a366-f87e972c7cc3',
    // OPTIONAL; When `true`, collect behavioral data.
    behavioralDataCollection: true,
    // OPTIONAL; When `true`, output SDK log messages in the developer console.
    consoleLogEnabled: false,
  },
});

// Subscribe to journey events
const unsubJourneyEvents = journeyEvents.subscribe((event) => {
  /* Run anything you want */
});

// Recommended: call when your UI component is destroyed
unsubJourneyEvents();
```

Schema for `journey` events

The schema for `journey` events is as follows:

Schema for `journey` events

```
{
  journey: {
    completed: false, // boolean
    error: null, // null or object with `code`, `message`, and `step` that failed
    loading: false, // boolean
    step: null, // null or object with the last step object from the server
    successful: false, // boolean
    response: null, // null or object if successful containing the success response from the server
  },
  oauth: {
    completed: false, // boolean
    error: null, // null or object with `code` and `message` properties
    loading: false, // boolean
    successful: false, // boolean
    response: null, // null or object with OAuth/OIDC tokens
  },
  user: {
    completed: false, // boolean
    error: null, // null or object with `code` and `message` properties
    loading: false, // boolean
    successful: false, // boolean
    response: null, // null or object with user information driven by OAuth scope config
  },
}
```

User

Use the `user` module to access methods for managing users:

- `user.info`
- `user.tokens`
- `user.logout`

The `user.info` and `user.tokens` methods requires use of OAuth 2.0. The `user.info` method also requires a scope value of `openid`.

The Ping (ForgeRock) Login Widget is configured to use both requirements by default.

You can use the `user.logout` method with both OAuth 2.0 and session-based authentication.

```
import { user } from '@forgerock/login-widget';

/**
 * User info API
 */
const userEvents = user.info();

// Subscribe to user info changes
const unsubUserEvents = userEvents.subscribe((event) => {
  // Return current, *local*, user info and future state changes
  console.log(event);
});

// Fetch/get fresh user info from the server
userEvents.get(); // New state is returned in your `userEvents.subscribe` callback function

/**
 * User tokens API
 */
const tokenEvents = user.tokens();

// Subscribe to user token changes
const unsubTokenEvents = tokenEvents.subscribe((event) => {
  // Return current, *local*, user tokens and future state changes
  console.log(event);
});

// Return existing user tokens if available and not expired or about to expire
// Otherwise obtain fresh ones from the server
tokenEvents.get(); // State is returned in your `tokenEvents.subscribe` callback function

/**
 * Logout
 * Log user out and clear user data (info and tokens)
 */
user.logout(); // Resets user and emits event to your info and tokens' `.subscribe` callback function

// Recommended: call when your UI component is destroyed
unsubUserEvents();
unsubTokenEvents();
```

You can use `get()` with both `user.info` and `user.tokens` to obtain the user's profile or OAuth 2.0 tokens. The `get()` function maps to the following methods in the Ping SDK for JavaScript, and support the same parameters:

- `userEvents.get()` = `UserManager.getCurrentUser()`
- `tokenEvents.get()` = `TokenManager.getTokens()`

For example, when getting a user's tokens you can force the Ping (ForgeRock) Login Widget to obtain fresh tokens from the server as follows:

```
tokenEvents.get({forceRenew: true});
```

Refer to the Ping SDK for JavaScript API reference for more information.

Schema for `user.info` events

The schema for `user.info` events is as follows:

Schema for `user.info` events

```
{
    completed: false, // boolean
    error: null,  // null or object with `code`, `message`, and `step` that failed
    loading: false, // boolean
    successful: false, // boolean
    response: null, // object returned from the `/userinfo` endpoint
}
```

Schema for `user.tokens` events

The schema for `user.tokens` events is as follows:

Schema for `user.tokens` events

```
{
    completed: false, // boolean
    error: null,  // null or object with `code`, `message`, and `step` that failed
    loading: false, // boolean
    successful: false, // boolean
    response: null, // object returned from the `/access_token` endpoint
}
```

Request

The Ping (ForgeRock) Login Widget has an alias to the Ping SDK for JavaScript's `HttpClient.request` method, which is a convenience wrapper around the native `fetch`. This method will auto-inject the access token into the `Authorization` header and manage some of the lifecycle around the token.

```
import { request } from '@forgerock/login-widget';

const response = await request({ init: { method: 'GET' }, url: 'https://protected.resource.com' });
```

The full `options` object:

```
{
  bypassAuthentication: false, // Boolean; if true, the access token is not injected into the `Authorization` header
  init: {
    // Options object for `fetch` API: https://developer.mozilla.org/en-US/docs/Web/API/fetch
  },
  timeout: 3000, // Fetch timeout in milliseconds
  url: 'https://protected.resource.com', // String; the URL of the resource

  // Unsupported properties
  authorization: {},
  requiresNewToken: () => {},
}
```

For more information, refer to the HttpClient reference documentation.

ForgeRock Authenticator

Server support:

- ✕ PingOne
- ✓ PingOne Advanced Identity Cloud
- ✓ PingAM
- ✕ PingFederate

SDK support:

- ✓ Ping SDK for Android
- ✓ Ping SDK for iOS
- ✕ Ping SDK for JavaScript

The ForgeRock Authenticator is an official multi-factor authentication (MFA) application for customers.

The ForgeRock Authenticator supports the following MFA methods:

- Time-based one-time passwords (TOTP)
- HMAC-based one-time passwords (HOTP)
- Push notifications

You can also enable *authentication app policies* in the ForgeRock Authenticator that prevent operation on rooted devices, for example.

Download

Download the ForgeRock Authenticator from the following stores:

Android

ForgeRock Authenticator on Google Play

iOS

ForgeRock Authenticator on the Apple App Store

Develop

As a customer, you can use the Ping (ForgeRock) Authenticator module to integrate the functionality of the ForgeRock Authenticator into your own apps.

Implement your use cases with the ForgeRock Authenticator

Find out how to achieve some common use case scenarios using the ForgeRock Authenticator.

Implement MFA using push notifications

In this use case, you authenticate a user with MFA by setting up the ForgeRock Authenticator for push notification.

To receive push notifications when authenticating, end users must register an Android or iOS device running the ForgeRock Authenticator.

Read more »

Implement MFA using OATH one-time passwords

In this use case you configure your server to require a single-use, one-time password generated by the ForgeRock Authenticator when users authenticate.

The ForgeRock Authenticator supports time-based and HMAC-based one-time passwords.

Read more »

Secure the Authenticator app using policies

You can distribute the ForgeRock Authenticator to your users so that they can participate in multi-factor authentication journeys.

To help ensure the security of the app—and therefore your system—you can apply *Authenticator app policies*.

Read more »

Develop your own use case solutions

As a customer, you can use the Ping (ForgeRock) Authenticator module to integrate the functionality of the ForgeRock Authenticator into your own Android and iOS apps.

For more information, refer to Ping (ForgeRock) Authenticator module.

Implement MFA using push notifications

Applies to:

- ✓ Ping SDK for Android
- ✓ Ping SDK for iOS
- ✕ Ping SDK for JavaScript

In this use case, you authenticate a user with MFA by setting up the ForgeRock Authenticator for push notification.

To receive push notifications when authenticating, end users must register an Android or iOS device running the ForgeRock Authenticator.

The registered device can then be used as an additional factor when authenticating. Your server sends the device a push notification, which is handled by the ForgeRock Authenticator. In the app, you can allow or deny the request that generated the push notification and return the response to AM.

How push authentication works

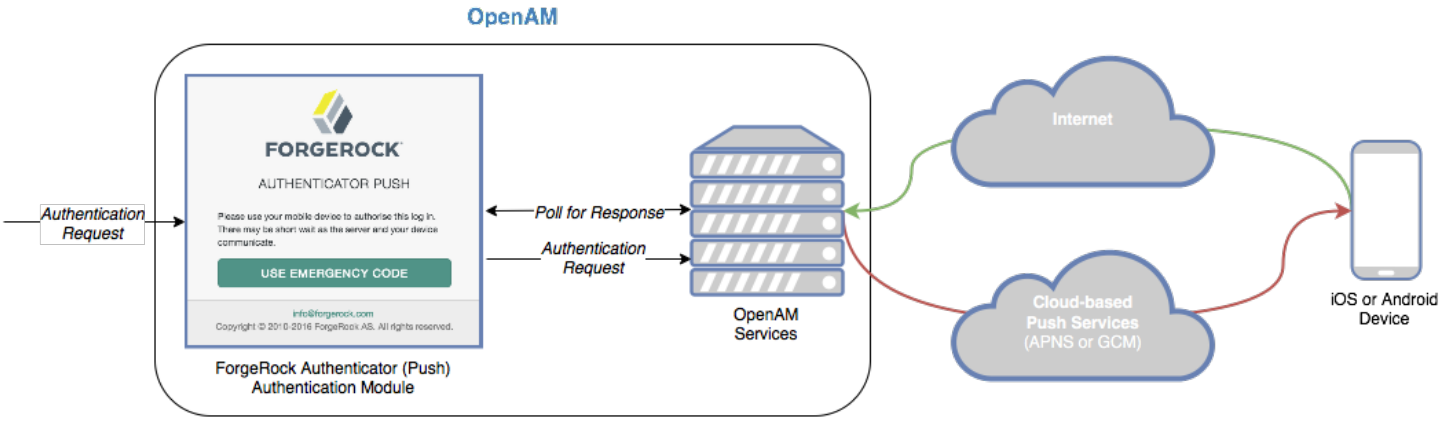


Figure 1. Overview of Push Authentication

The following steps occur as a user completes a push notification journey:

1. The user provides credentials to enable AM to locate the user profile and determine if they have a registered mobile device.
2. The journey prompts the user to register a mobile device if they have not done so already.

   The user registers their device by using the ForgeRock Authenticator. The application supports a variety of methods for responding to push notifications, such as tapping a button, or using biometrics.

   Registering a device stores device metadata in the user profile that is required for push notifications. AM uses the configured ForgeRock Authenticator (Push) service, which supports encrypting the metadata.
3. Once the user has a registered device, AM creates a push message specific to the device.

The message has a unique ID that AM stores while waiting for the response.

AM writes a pending record with the same message ID to the CTS store for redundancy should an individual server go offline during the authentication process.

4. AM sends the push message to the registered device.

   AM delivers the message through the configured push notification service.

   Depending on the registered device, AM uses either Apple Push Notification Services (APNS) or Google Cloud Messaging (GCM) to deliver the message.

   AM begins to poll the CTS for an accepted response from the registered device.

5. The user responds to the notification through the ForgeRock Authenticator application on the device, for example, approving or rejecting the notification.

   The application responds to the push notification message with the user's choice.

6. AM verifies the message is from the correct registered device and has not been tampered with, and marks the pending record as accepted if valid.

   AM detects the accepted record and redirects the user to their profile page, completing the authentication.

Step 1. Enable the ForgeRock Authenticator Push service

In this step you configure your server to operate with the ForgeRock Authenticator. We will use the default settings.

Advanced Identity Cloud

1. Log in to the Advanced Identity Cloud admin UI as an administrator.
2. In the left menu pane, select **Native Consoles** > **Access Management**.

   The realm overview for the Alpha realm displays.
3. Select **Services**, and then click **+ Add a Service**.
4. In **Choose a service type**, select **ForgeRock Authenticator (Push) Service**, and then click **Create**.
5. Click **Save Changes** to accept the default settings.

Self-managed PingAM server

1. Log in to the AM admin UI as an administrator.

   The realm overview for the Top Level Realm displays.

2. Select **Services**, and then click **+ Add a Service**.

3. In **Choose a service type**, select **ForgeRock Authenticator (Push) Service**, and then click **Create**.

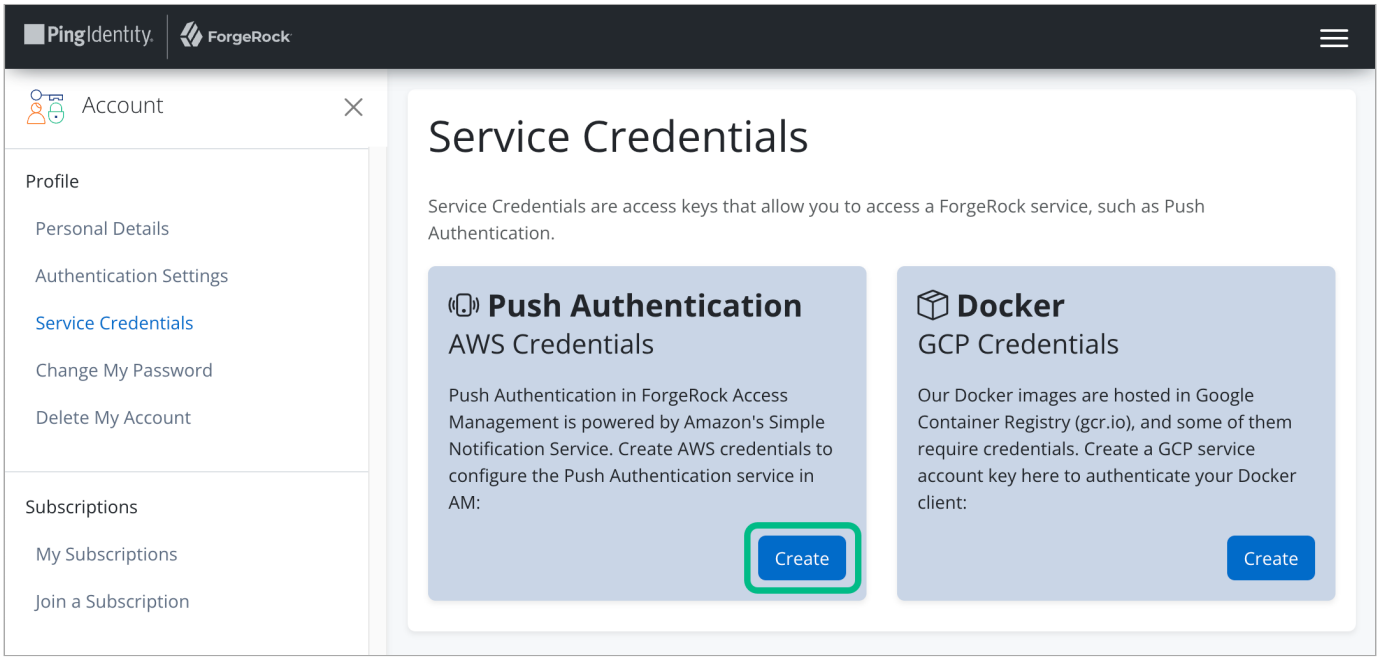
4. Click **Save Changes** to accept the default settings.

Step 2: Create Push service credentials in Backstage

Your server uses an external AWS service to send push notifications. Its configuration requires access keys and other metadata. As a customer, you have streamlined access to the required metadata:

1. In a web browser, log in to the Service Credentials page on Backstage.

2. Under **Push Authentication AWS Credentials**, click **Create**.



3. In **Description**, enter `Push credentials for MFA journey.`

4. In **Region**, select the location closest to the majority of your users.

5. Click **Create**.

   The **Create New AWS Credential** screen displays the settings you must add to your server.

6. Click **Download as JSON**.

   This downloads a file in JSON format containing all the values for you to store securely:

Example of a downloaded JSON file with push service credential values


```
{
  "id": "uzW8...rwJC",
  "provider": "AWS",
  "creationTimestamp": "2023-10-18T17:02:00.882Z",
  "createdBy": "admin.user",
  "updateTimestamp": null,
  "updatedBy": null,
  "description": "Push credentials for MFA journey",
  "supportKey": null,
  "validUntil": null,
  "writable": true,
  "region": "eu-west-1",
  "accessKeyId": "AKIA...6N74",
  "accessKeySecret": "uVjT...uu+k",
  "applicationArns": {
    "APNS": "arn:aws:sns:eu-west-1:1234:app/APNS/rgBO...n4RA",
    "GCM": "arn:aws:sns:eu-west-1:1234:app/GCM/rgBO...n4RA"
  }
}
```


Important

It is vital that you have a copy of these values, especially the `accessKeySecret` value, as it is not stored on Backstage.

7. Click **Close**.

Step 3: Configure the Push Notification service in your server

In this step you configure your server with the settings it needs to be able to send push notifications to mobile devices.

You will need the AWS service credentials obtained in the previous step.

Advanced Identity Cloud

1. Log in to the Advanced Identity Cloud admin UI as an administrator.
2. In the left menu pane, select **Native Consoles** > **Access Management**.

   The realm overview for the Alpha realm displays.
3. Select **Services**, and then click **+ Add a Service**.
4. In **Choose a service type**, select **Push Notification Service**.
5. Open the JSON file you obtained in the previous step:

```
{
  "id": "uzW8...rwJC",
  "provider": "AWS",
  "creationTimestamp": "2023-10-18T17:02:00.882Z",
  "createdBy": "admin.user",
  "updateTimestamp": null,
  "updatedBy": null,
  "description": "Push credentials for MFA journey",
  "supportKey": null,
  "validUntil": null,
  "writable": true,
  "region": "eu-west-1",
  "accessKeyId": "AKIA...6N74", (1)
  "accessKeySecret": "uVjT...uu+k", (2)
  "applicationArns": {
    "APNS": "arn:aws:sns:eu-west-1:1234:app/APNS/rBO...n4A", (3)
    "GCM": "arn:aws:sns:eu-west-1:1234:app/GCM/rBO...n4A" (4)
  }
}
```

6. Enter the fields from the JSON file into the fields that display:

Field in Advanced Identity Cloud admin UI	Field in JSON file	Description
SNS Access Key ID	① `accessKeyId`	The generated Key ID; the `"accessKeyId"` key in the JSON.
SNS Access Key Secret	② `accessKeySecret`	The generated Secret; the `"accessKeySecret"` key in the JSON.
SNS Endpoint for APNS	③ `APNS`	The generated APNS value; the `"applicationArns/APNS"` key in the JSON. The Apple Push Notification Service (APNS) endpoint that SNS uses to send push notifications to iOS devices.

Field in Advanced Identity Cloud admin UI	Field in JSON file	Description
SNS Endpoint for GCM	④ `GCM`	The generated GCM value; the `"applicationArns/GCM"` key in the JSON. The Google Cloud Messaging (GCM) endpoint that SNS uses to send push notifications to Android devices.

> **Note**
>
> Do not enter the quotes that surround the JSON values.

7. Click **Create**, and then click **Save Changes**.

Self-managed PingAM server

1. Log in to the AM admin UI as an administrator.

   The realm overview for the Top Level Realm displays.

2. Select **Services**, and then click **+ Add a Service**.

3. In **Choose a service type**, select **Push Notification Service**.

4. Open the JSON file you obtained in the previous step:

```
{
  "id": "uzW8...rwJC",
  "provider": "AWS",
  "creationTimestamp": "2023-10-18T17:02:00.882Z",
  "createdBy": "admin.user",
  "updateTimestamp": null,
  "updatedBy": null,
  "description": "Push credentials for MFA journey",
  "supportKey": null,
  "validUntil": null,
  "writable": true,
  "region": "eu-west-1",
  "accessKeyId": "AKIA...6N74", (1)
  "accessKeySecret": "uVjT...uu+k", (2)
  "applicationArns": {
    "APNS": "arn:aws:sns:eu-west-1:1234:app/APNS/rBO...n4A", (3)
    "GCM": "arn:aws:sns:eu-west-1:1234:app/GCM/rBO...n4A" (4)
  }
}
```

5. Enter the fields from the JSON file into the fields that display:

Field in AM admin UI	Field in JSON file	Description
SNS Access Key ID	① `accessKeyId`	The generated Key ID; the `"accessKeyId"` in the JSON.
SNS Access Key Secret	② `accessKeySecret`	The generated Secret; the `"accessKeySecret"` in the JSON.
SNS Endpoint for APNS	③ `APNS`	The generated APNS value; the `"applicationArns/APNS"` key in the JSON. The Apple Push Notification Service (APNS) endpoint that SNS uses to send push notifications to iOS devices.
SNS Endpoint for GCM	④ `GCM`	The generated GCM value; the `"applicationArns/GCM"` key in the JSON. The Google Cloud Messaging (GCM) endpoint that SNS uses to send push notifications to Android devices.

> **Note**
>
> Do not enter the quotes that surround the JSON values.

6. Click **Create**, and then click **Save Changes**.

Step 4: Create a push registration and authentication journey

In this step you create an authentication journey that registers a device running the ForgeRock Authenticator to the user's profile if they have not done so already, then send a push notification to that device.

The journey then polls until it receives a response or timeout from the device. It verifies the returned data and completes the authentication journey if valid.

Choose whether you are creating the journey in PingOne Advanced Identity Cloud or a self-managed PingAM server, and follow the instructions to create the required authentication journey:

Advanced Identity Cloud

1. In the Advanced Identity Cloud admin UI
    1. Select the realm that will contain the authentication journey.
    2. Select **Journeys**, and click **+ New Journey**.
    3. Enter a name for your tree in **Name** page; for example, `MFAwithPush`
    4. In **Identity Object**, select the identity type that will be authenticating, for example `Alpha realm - Users`.
    5. Click **Save**.

       The authentication journey designer page is displayed with the default Start, Failure, and Success nodes.
2. Add the following nodes to the designer area:
    - Page node
    - Password Collector node
    - Username Collector node
    - Data Store Decision node
    - Push Sender node
    - Push Registration node or Combined MFA Registration node [1]
    - Push Wait node
    - Push Result Verifier node
3. Connect the nodes as shown:

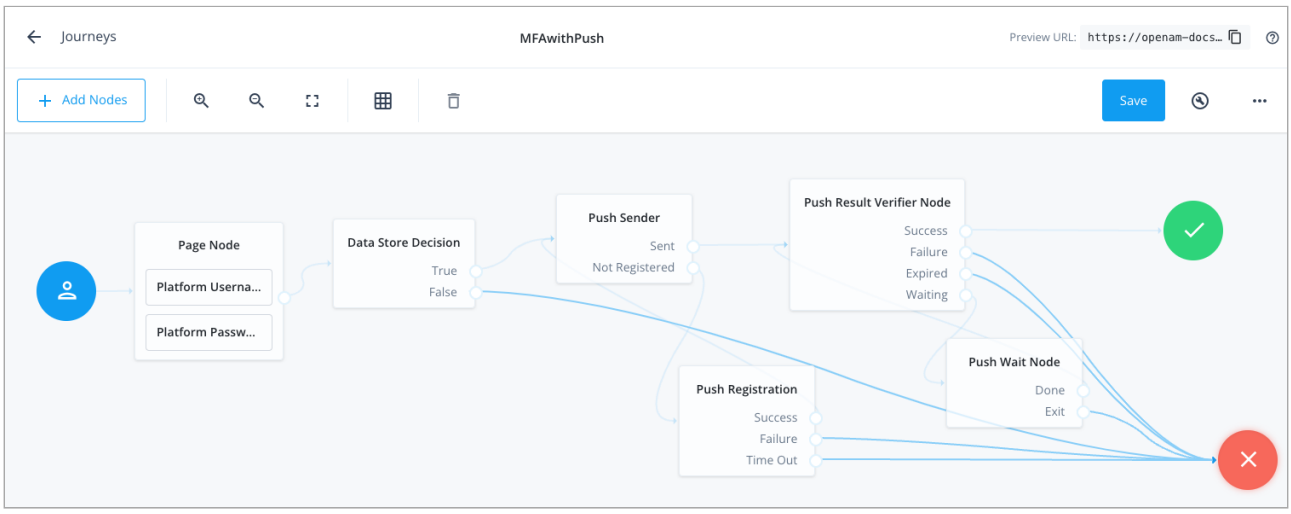


Figure 2. Connect the nodes to identify the user, send a push notification, and very the result.

4. In the Push Sender node, select the type of push notification the journey sends to the ForgeRock Authenticator:

Tap to Accept

Requires the user to tap to accept.

Display Challenge Code

Requires the user to select one of three numbers displayed on their device. This selected number must match the code displayed in the browser for the request to be verified.

Use Biometrics to Accept

Requires the user's biometric authentication to process the notification.

For information on how these options appear in the ForgeRock Authenticator, refer to Authenticate using a push notification.

5. Save your changes.

Self-managed PingAM server

1. In the AM admin UI:
    1. Select the realm that will contain the authentication tree.
    2. Select **Authentication > Trees**, and click **+ Create Tree**.
    3. Enter a name for your tree in the **New Tree** page; for example, `MFAwithPush`, and click **Create**.

       The authentication tree designer page is displayed with the default Start, Failure, and Success nodes.
2. Add the following nodes to the designer area:
    - Page node
    - Password Collector node
    - Username Collector node
    - Data Store Decision node
    - Push Sender node
    - Push Registration node or Combined MFA Registration node [1]
    - Push Wait node
    - Push Result Verifier node
3. Connect the nodes as shown:

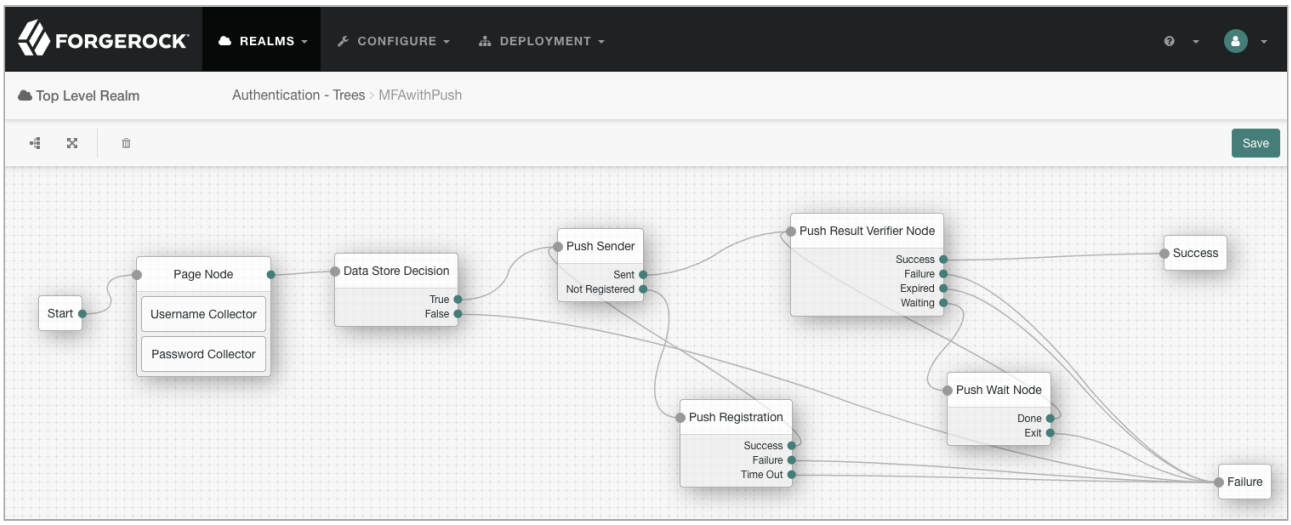


Figure 3. Connect the nodes to identify the user, send a push notification, and very the result.

4. In the Push Sender node, select the type of push notification the journey sends to the ForgeRock Authenticator:

Tap to Accept

Requires the user to tap to accept.

Display Challenge Code

Requires the user to select one of three numbers displayed on their device. This selected number must match the code displayed in the browser for the request to be verified.

Use Biometrics to Accept

Requires the user's biometric authentication to process the notification.

For information on how these options appear in the ForgeRock Authenticator, refer to Authenticate using a push notification.

5. Save your changes.

The tree you create is a simple example for the purposes of demonstrating a basic push authentication journey. In a production environment, you could include additional nodes, such as:

Get Authenticator App node

Provides links to download the ForgeRock Authenticator for Android and iOS.

MFA Registration Options node

Provides options for users to register a multi-factor authentication device, get the authenticator app, or skip the registration process.

Opt-out Multi-Factor Authentication node

Sets an attribute in the user's profile which lets them skip multi-factor authentication.

Recovery Code Display node

Lets a user view recovery codes to use in case they lose or damage the authenticator device they register.

Recovery Code Collector Decision node

Lets a user enter their recovery codes to authenticate in case they have lost or damaged their registered authenticator device.

Retry Limit Decision node

Lets a journey loop a specified number of times, for example, in case the user's device is experiencing connectivity issues, for example.

Step 5: Authenticate using a push notification

After creating the journey, you can register the ForgeRock Authenticator, and use it to respond to the push notificaiton message:

1. If you have not already done so, create a demo user in your server:

   The samples and tutorials in this documentation often require that you have an identity set up so that you can test authentication.

To create a demo user in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. In the left panel, click **Identities** > **Manage**.
3. Click **+ New Alpha realm - User**.
4. Enter the following details:
   - **Username** = `demo`
   - **First Name** = `Demo`
   - **Last Name** = `User`
   - **Email Address** = `demo.user@example.com`
   - **Password** = `Ch4ng3it!`
5. Click **Save**.

The samples and tutorials in this documentation often require that you have an identity set up so that you can test authentication.

To create a demo user in PingAM, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. Navigate to **Identities**, and then click **+ Add Identity**.
3. Enter the following details:
   - **User ID** = `demo`
   - **Password** = `Ch4ng3it!`
   - **Email Address** = `demo.user@example.com`
4. Click **Create**.

2. In an incognito browser window, browse to the journey you created in the previous step:

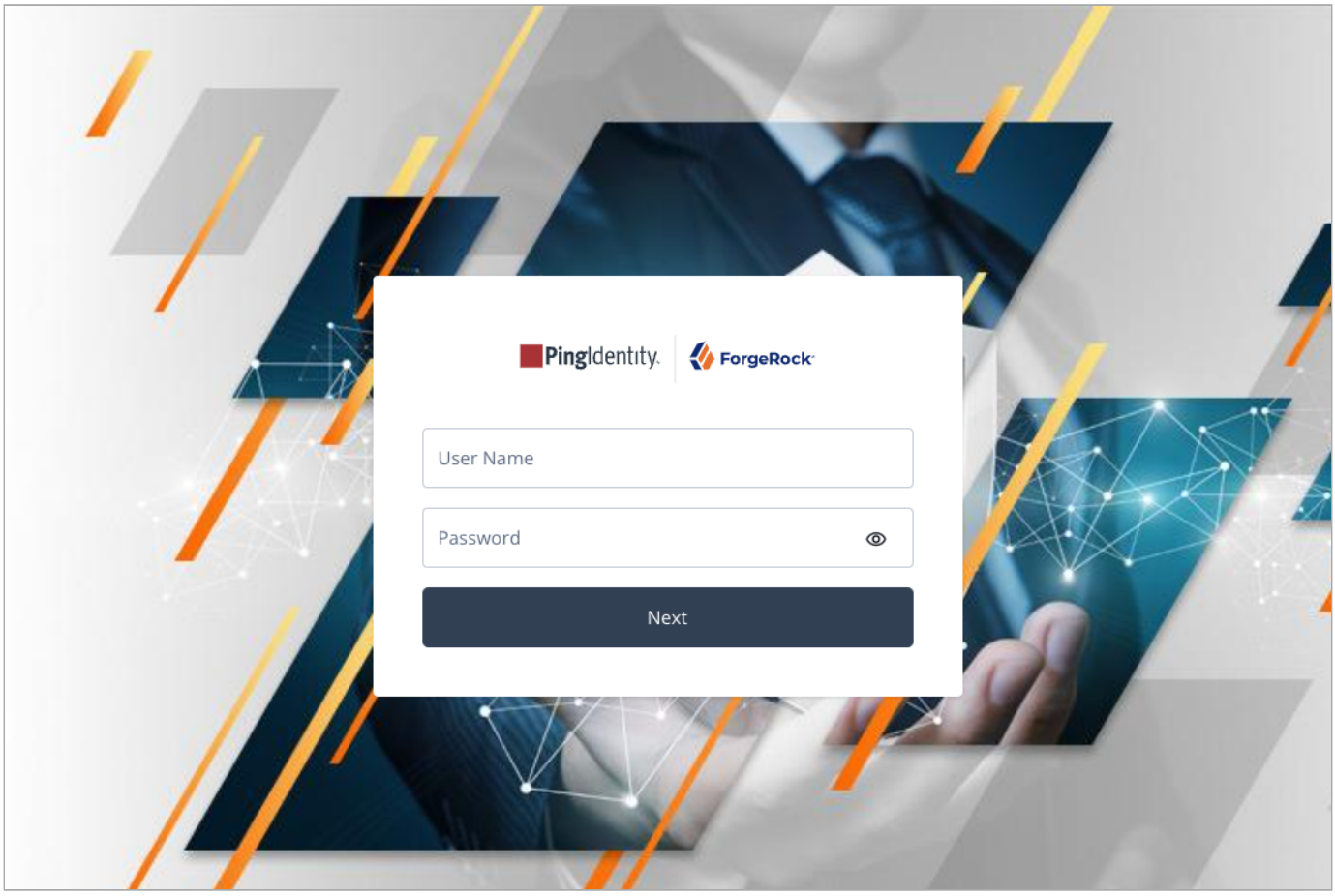
Advanced Identity Cloud

```
https://openam-forgerock-sdks.forgeblocks.com/am/XUI/?
realm=alpha&authIndexType=service&authIndexValue=MFAwithPush
```

Self-managed PingAM server

```
https://openam.example.com:8443/openam/XUI/?
realm=alpha&authIndexType=service&authIndexValue=MFAwithPush
```

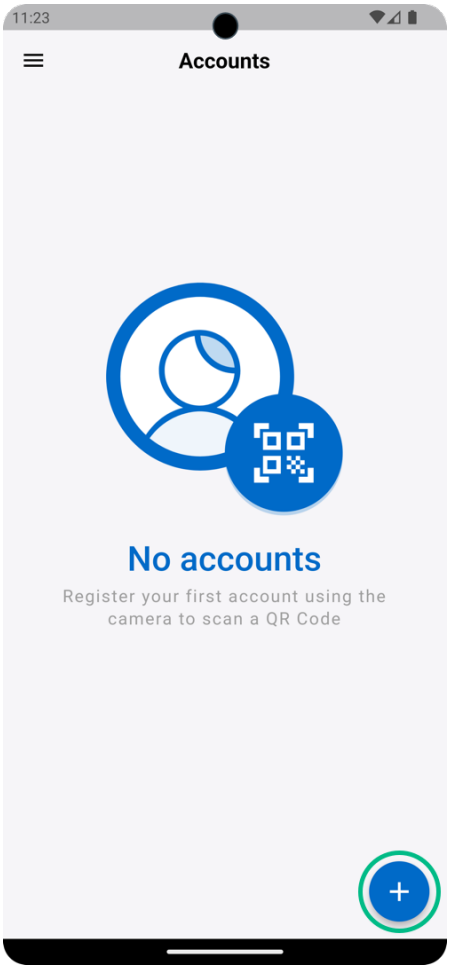
The journey asks for your credentials:



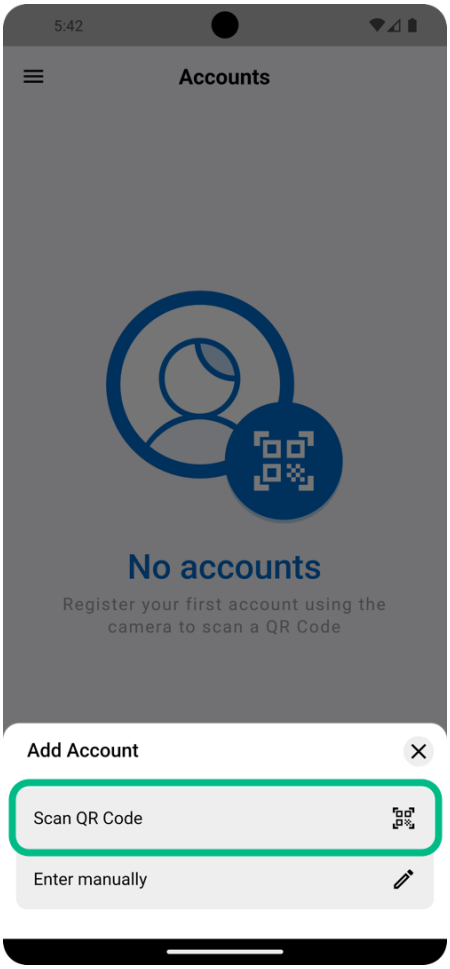
3. Sign in with the username and password of your demo user.

   If you have not yet registered the ForgeRock Authenticator, the journey displays a QR code:

4. In the ForgeRock Authenticator, click the blue plus icon to register the account on the device:



5. In the **Add Account** menu that appears, select **Scan QR Code**:



6. Scan the QR code on screen using the ForgeRock Authenticator:

When the ForgeRock Authenticator registers the account it notifies your server, which then initiates the configured push notification.

7. In the ForgeRock Authenticator, complete the authentication as requested:

    - If the journey is configured with the default **Push Type** setting, `Tap to Accept`, tap the **Accept** button:

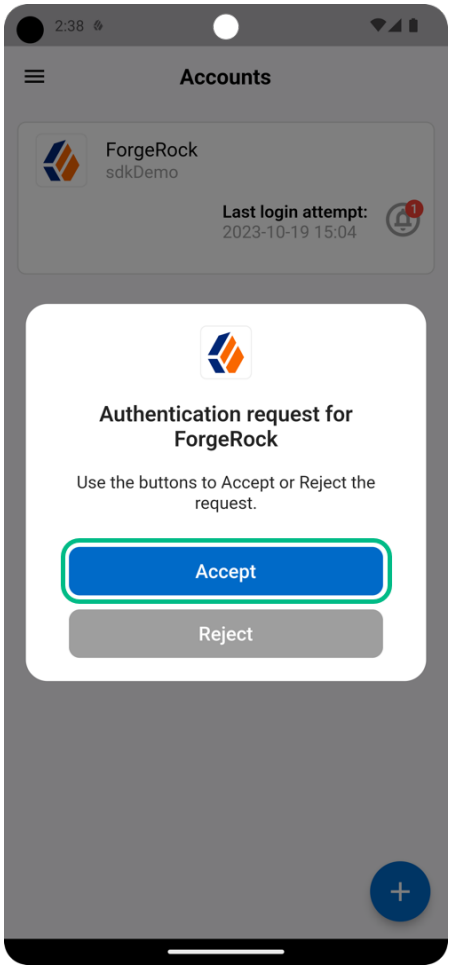


Figure 4. Tap the Accept button.

- If the journey is configured with a **Push Type** setting of `Display Challenge Code`, tap the number in the ForgeRock Authenticator that matches the number displayed by the journey:

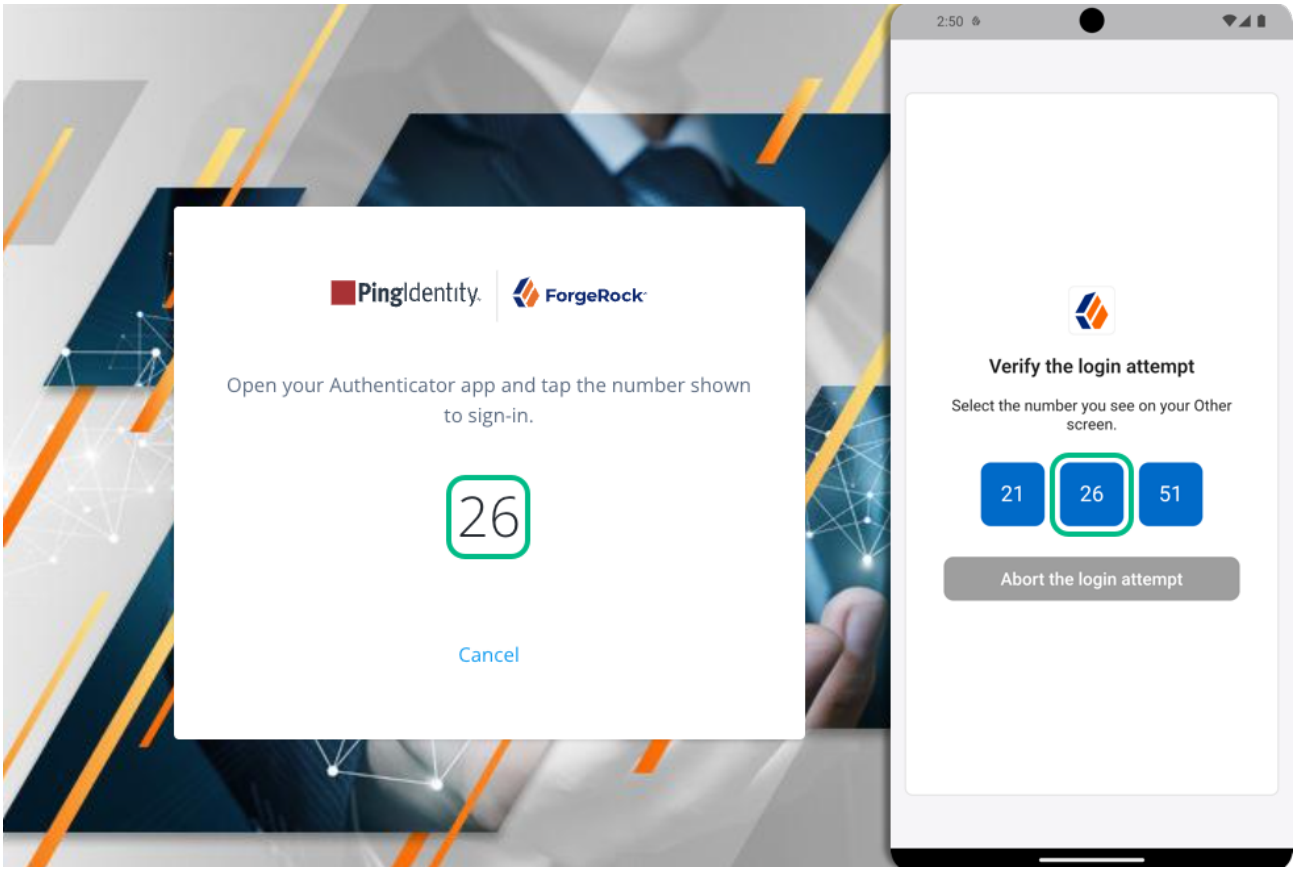


Figure 5. Tap the number in the app that matches the one displayed by the journey.

- If the journey is configured with a **Push Type** setting of `Use Biometrics to Accept`, tap the **Accept** button, and then use your devices biometric capabilities to complete authentication:

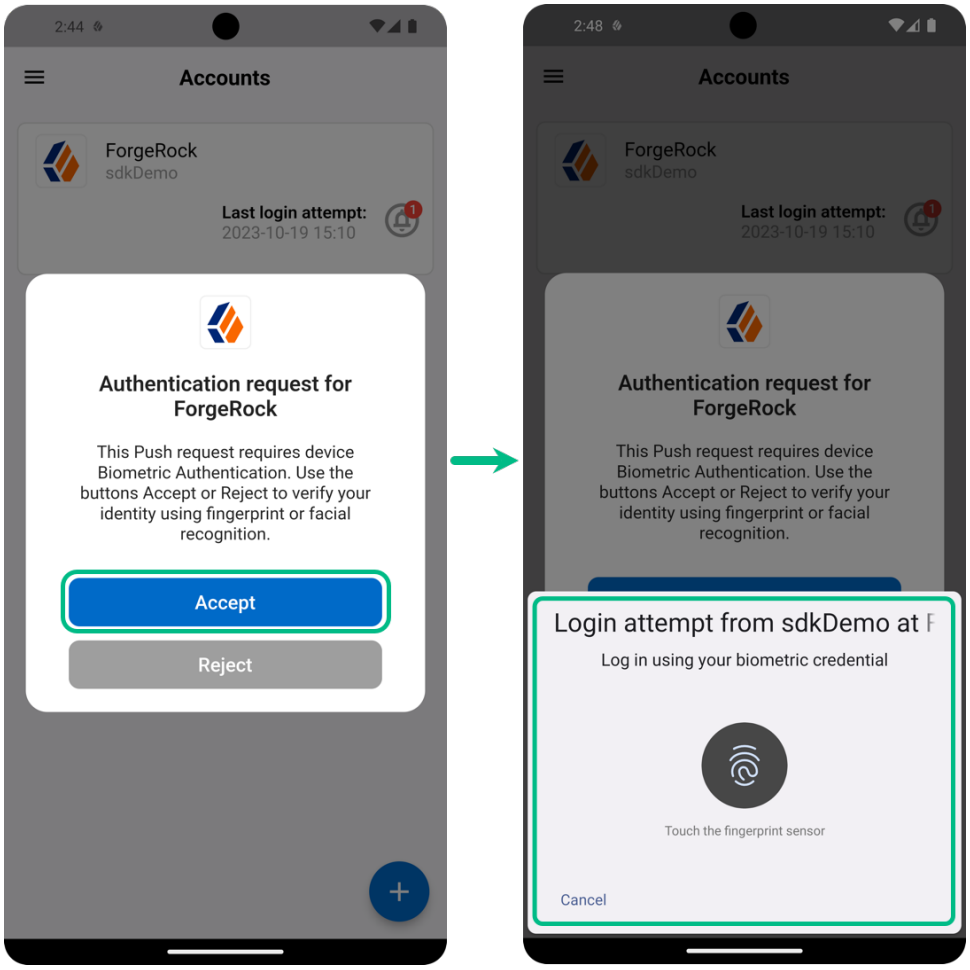


Figure 6. Tap Accept, then use a biometric method to authenticate.

After you successfully complete the required response, the browser displays the user's profile page.

Next steps

You can add support for MFA using push notifications to your own Android and iOS applications, by using the Ping (ForgeRock) Authenticator module.

For more information, refer to Integrate MFA using push notifications.

1. Use the combined MFA registration node if you intend to also add OATH one-time passwords as an MFA method.

Implement MFA using OATH one-time passwords

Applies to:

- ✓ Ping SDK for Android
- ✓ Ping SDK for iOS
- ✗ Ping SDK for JavaScript

In this use case you configure your server to require a single-use, one-time password generated by the ForgeRock Authenticator when users authenticate.

Overview

The ForgeRock Authenticator supports one-time password (OTP) authentication as defined in the OATH standard protocols.

The following methods for generating one-time passwords are supported:

HMAC-based one-time passwords (HOTP)

As described in RFC 4226, HOTP authentication generates the one-time password (OTP) every time the user requests a new password on their device.

The device tracks the number of times the user requests a new one-time password with a counter. The user may be further in the counter on their device than the server.

Your server resynchronizes the counter when the user finally logs in. To accommodate this, you set the number of passwords a user can generate before their device cannot be resynchronized.

For example, if you set the **HOTP Window Size** to `50` and someone presses the button 30 times in the ForgeRock Authenticator to generate a new password, the counter in your server will review the passwords until it reaches the one-time password entered by the user.

If, however, someone presses the button 51 times, you will need to reset the counter to match the number on the device's counter before the user can log in.

HOTP authentication does not check earlier passwords, so if the user attempts to reset the counter on their device, they will not be able to log in until you reset the counter on the server to match their device.

Time-based one-time passwords (TOTP)

As described in (RFC 6238), TOTP authentication constantly generates a new one-time password based on a time interval you specify.

The **TOTP Time Step Interval** setting configures how often a new password is generated by the ForgeRock Authenticator.

Use the **TOTP Time Steps** setting to provide a margin in case the time varies between your server and the device running the ForgeRock Authenticator. For example, set this to `1` to accept either the previous, the current, or the next password as valid.

Step 1. Create an OATH registration and authentication journey

In this step you create an authentication journey that registers a device running the ForgeRock Authenticator to the user's profile if they have not done so already, then requests a one-time password from the device.

Choose whether you are creating the journey in PingOne Advanced Identity Cloud or a self-managed PingAM server, and follow the instructions to create the required authentication journey:

Advanced Identity Cloud

1. In the Advanced Identity Cloud admin UI
    1. Select the realm that will contain the authentication journey.
    2. Select **Journeys**, and click **+ New Journey**.
    3. Enter a name for your tree in **Name** page; for example, `MFAwithOATH`
    4. In **Identity Object**, select the identity type that will be authenticating, for example `Alpha realm - Users`.
    5. Click **Save**.

       The authentication journey designer page is displayed with the default Start, Failure, and Success nodes.
2. Add the following nodes to the designer area:
    - Page node
    - Platform Password node
    - Platform Username node
    - Data Store Decision node
    - OATH Token Verifier node
    - OATH Registration node or Combined MFA Registration node [1]
3. Connect the nodes as shown:

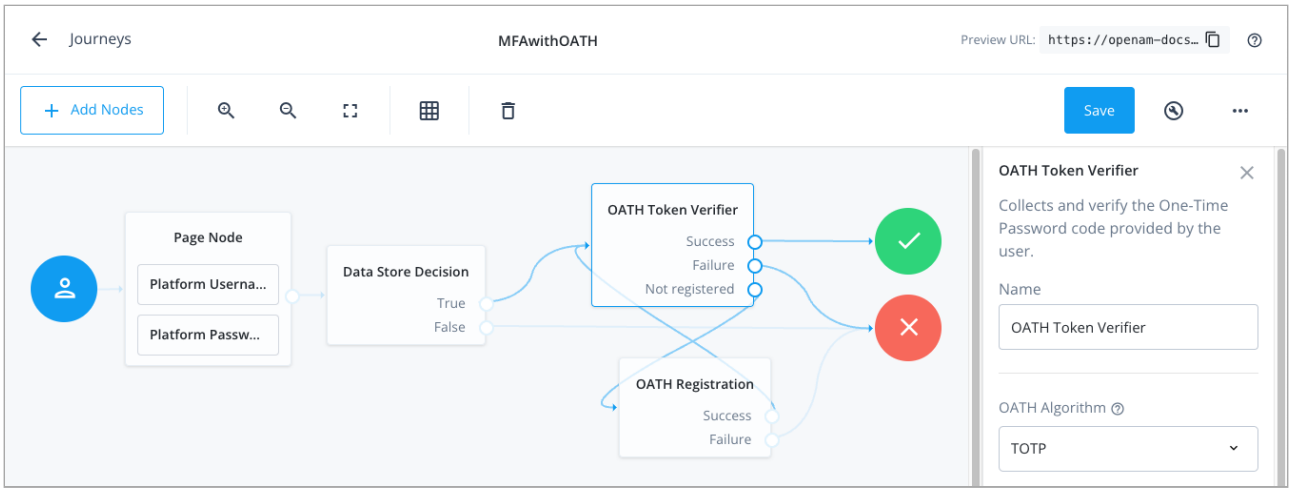


Figure 1. Connect the nodes to identify the user, then verify their OATH token.

4. Ensure that the OATH Token Verifier node and the OATH Registration node or Combined MFA Registration node are using the same value for **OATH Algorithm**.

   In this example, select `TOTP`.
5. Save your changes.

Self-managed PingAM server

1. In the AM admin UI:
    1. Select the realm that will contain the authentication tree.
    2. Select **Authentication > Trees**, and click **+ Create Tree**.
    3. Enter a name for your tree in the **New Tree** page; for example, `MFAwithOATH`, and click **Create**.

       The authentication tree designer page is displayed with the default Start, Failure, and Success nodes.
2. Add the following nodes to the designer area:
    - Page node
    - Password Collector node
    - Username Collector node
    - Data Store Decision node
    - OATH Token Verifier node
    - OATH Registration node or Combined MFA Registration node [1]
3. Connect the nodes as shown:

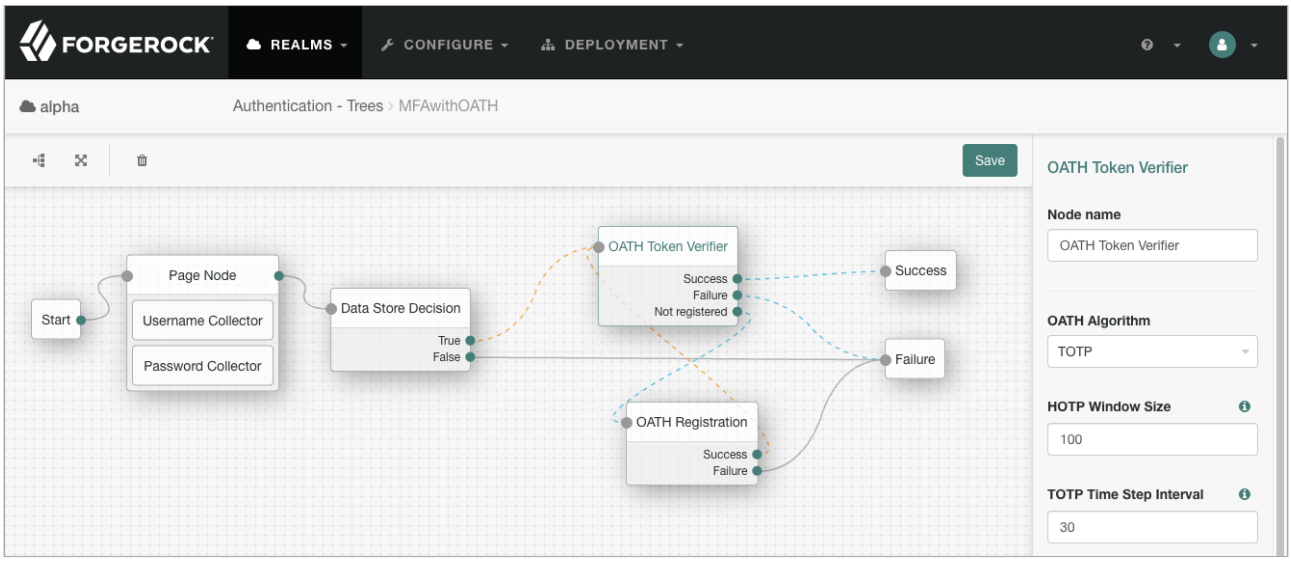


Figure 2. Connect the nodes to identify the user, then verify their OATH token.

4. Ensure that the OATH Token Verifier node and the OATH Registration node or Combined MFA Registration node are using the same value for **OATH Algorithm**.

   In this example, select `TOTP`.
5. Save your changes.

The tree you create is a simple example for the purposes of demonstrating a basic OATH authentication journey. In a production environment, you could include additional nodes, such as:

Get Authenticator App node

Provides links to download the ForgeRock Authenticator for Android and iOS.

MFA Registration Options node

Provides options for users to register a multi-factor authentication device, get the authenticator app, or skip the registration process.

Opt-out Multi-Factor Authentication node

Sets an attribute in the user's profile which lets them skip multi-factor authentication.

Recovery Code Display node

Lets a user view recovery codes to use in case they lose or damage the authenticator device they register.

Recovery Code Collector Decision node

Lets a user enter their recovery codes to authenticate in case they have lost or damaged their registered authenticator device.

Retry Limit Decision node

Lets a journey loop a specified number of times, for example, to allow a user to retry entering their OATH token.

Step 2. Authenticate using a one-time password

After creating the journey, you can register the ForgeRock Authenticator, and use it to generate and authenticate with a single-use, one-time password:

1. If you have not already done so, create a demo user in your server:

   The samples and tutorials in this documentation often require that you have an identity set up so that you can test authentication.

   To create a demo user in PingOne Advanced Identity Cloud, follow these steps:

   1. Log in to your PingOne Advanced Identity Cloud tenant.
   2. In the left panel, click **Identities** > **Manage**.
   3. Click **+ New Alpha realm - User**.
   4. Enter the following details:
      - **Username** = `demo`
      - **First Name** = `Demo`
      - **Last Name** = `User`
      - **Email Address** = `demo.user@example.com`

- **Password** = `Ch4ng3it!`

5. Click **Save**.

The samples and tutorials in this documentation often require that you have an identity set up so that you can test authentication.

To create a demo user in PingAM, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. Navigate to **Identities**, and then click **+ Add Identity**.
3. Enter the following details:
   - **User ID** = `demo`
   - **Password** = `Ch4ng3it!`
   - **Email Address** = `demo.user@example.com`
4. Click **Create**.

2. In an incognito browser window, browse to the journey you created in the previous step:

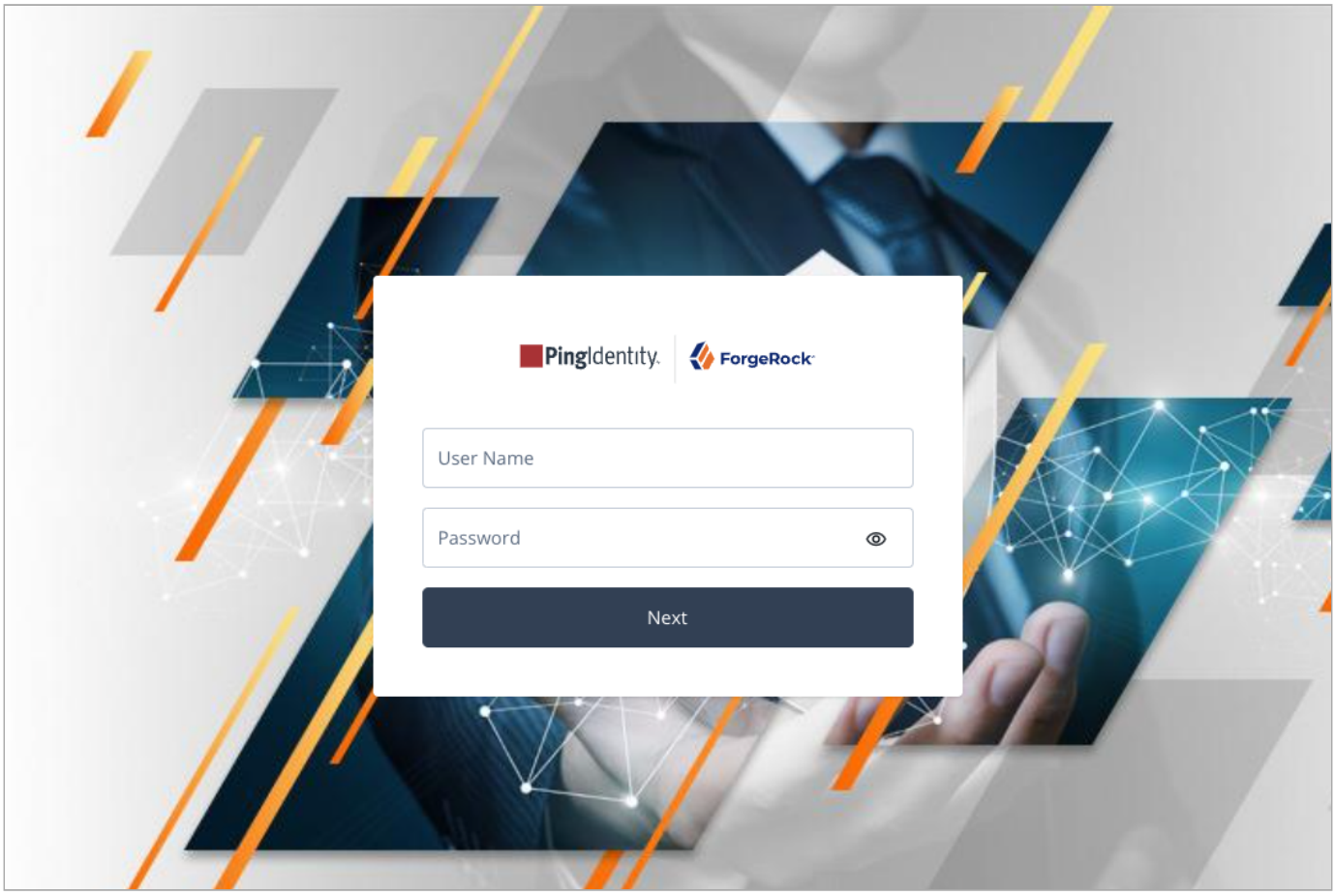
Advanced Identity Cloud

```
https://openam-forgerock-sdks.forgeblocks.com/am/XUI/?
realm=alpha&authIndexType=service&authIndexValue=MFAwithOATH
```

Self-managed PingAM server

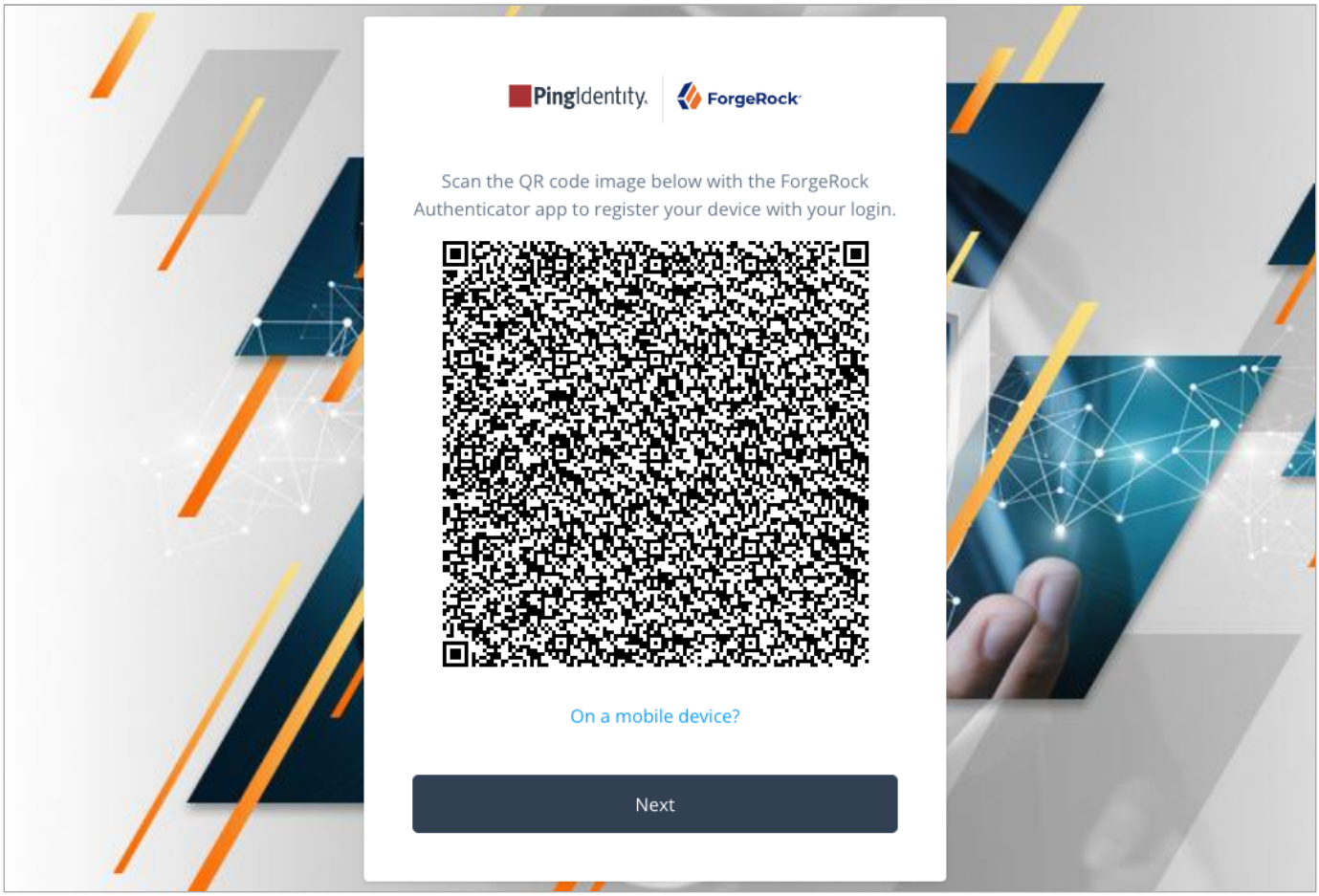
```
https://openam.example.com:8443/openam/XUI/?
realm=alpha&authIndexType=service&authIndexValue=MFAwithOATH
```

The journey asks for your credentials:

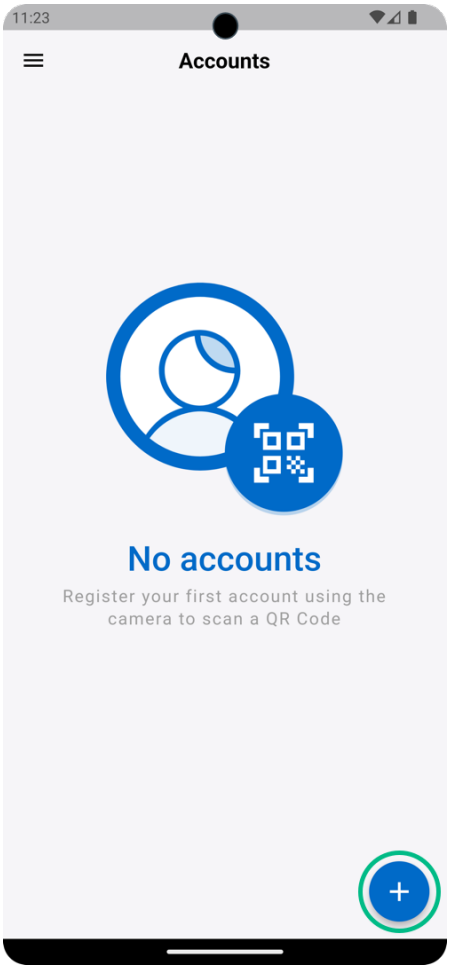


3. Sign in with the username and password of your demo user.

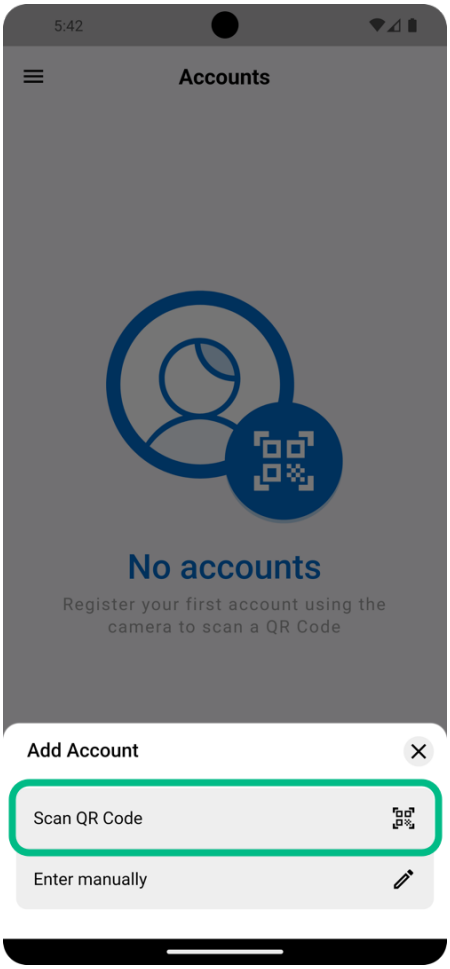
   If you have not yet registered the ForgeRock Authenticator, the journey displays a QR code:



4. In the ForgeRock Authenticator, click the blue plus icon to register the account on the device:



5. In the **Add Account** menu that appears, select **Scan QR Code**:

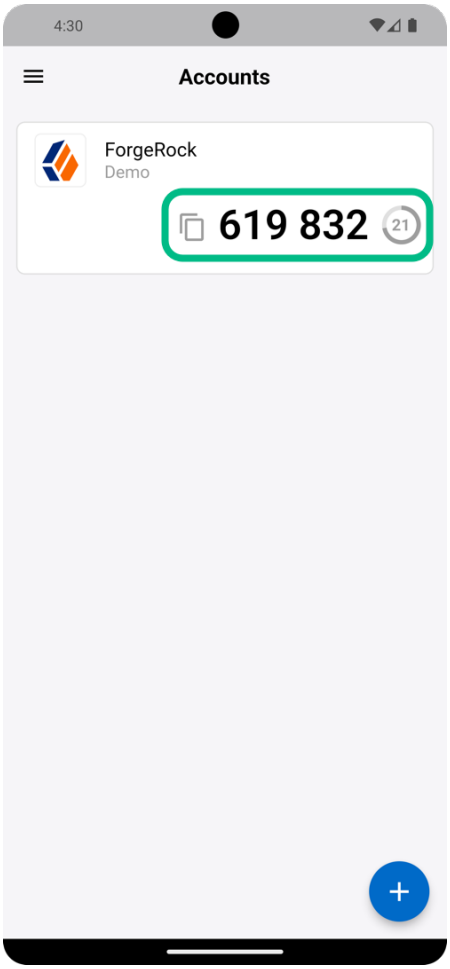


6. Scan the QR code on screen using the ForgeRock Authenticator:

7. When the ForgeRock Authenticator registers the account, in your browser, click **Next** to continue the journey.

   The journey requests the one-time password:

8. Enter the verification code from the ForgeRock Authenticator.



> **Tip**
>
> If the animated timer indicates the one-time password is close to expiry, wait for the app to generate a new one.

If you enter a valid one-time password, the browser displays the user's profile page.

Next steps

You can add support for MFA using one-time passwords to your own Android and iOS applications, by using the Ping (ForgeRock) Authenticator module.

For more information, refer to Integrate MFA using OATH one-time passwords.

1. Use the combined MFA registration node if you intend to add Push notifications as an MFA method.

Secure the Authenticator app using policies

Applies to:

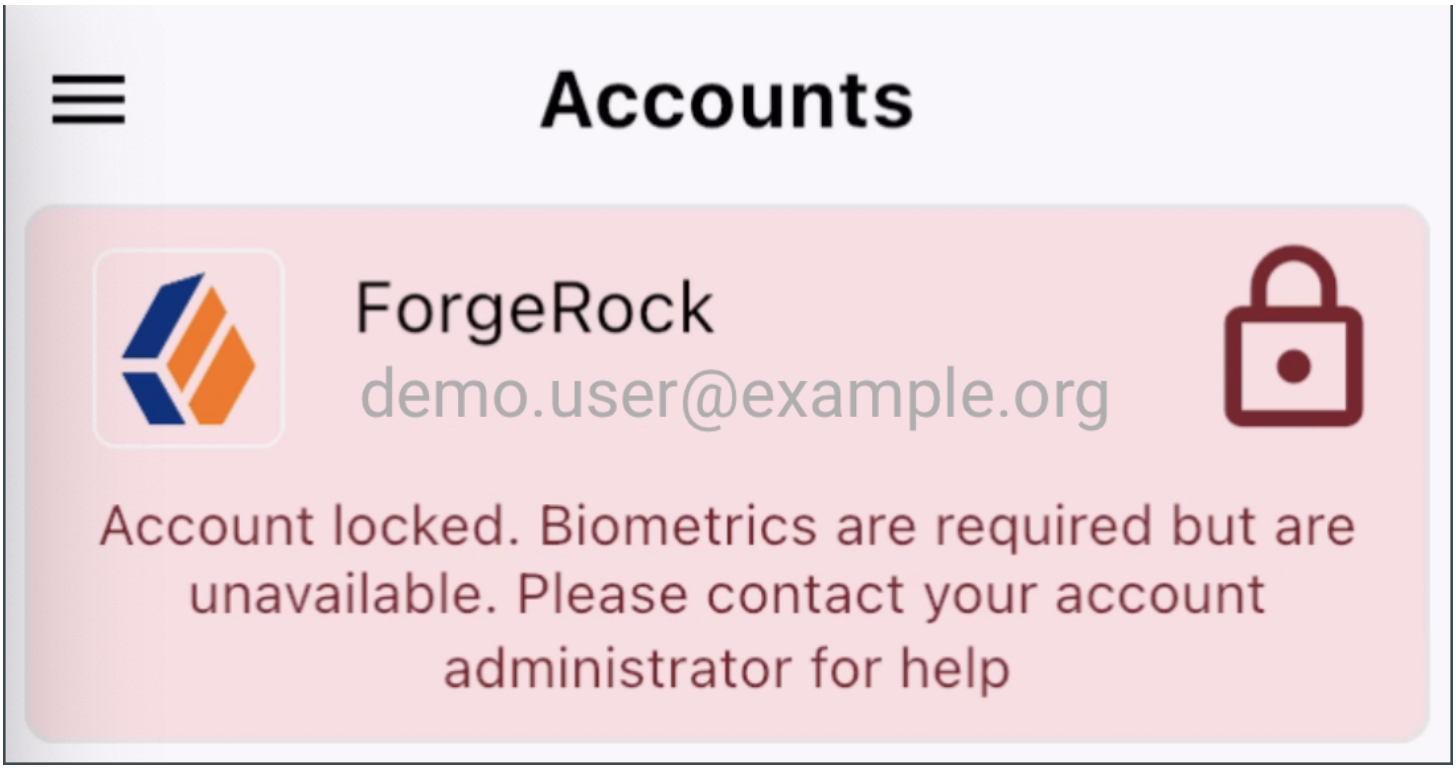
- ✓ Ping SDK for Android
- ✓ Ping SDK for iOS
- ✕ Ping SDK for JavaScript

You can distribute the ForgeRock Authenticator app to your users so that they can participate in multi-factor authentication journeys. To help ensure the security of the app—and therefore your system—you can enable *Authenticator app policies*.

The Combined MFA Registration node can apply authenticator app policies during registration of client devices.

These policies can perform checks on the client device. For example, that the device has not been rooted or jailbroken, or verify the use of biometrics on the device.

If the conditions of the policy are not met, the account cannot be registered in the Authenticator app. If the conditions of the policies applied to the account are breached anytime after successful registration, the account is locked, and MFA is blocked:



Available policies

The Authenticator app supports the following policies by default:

Require biometrics

Policy name: `biometricAvailable`

Require the device uses biometric sensors to unlock the operating system.

Device tampering detection

Policy name: `deviceTampering`

Require the device has not been tampered with, for example, if it has root access or is jailbroken.

This policy applies if the tampering likelihood score returned by the device to the Authenticator app exceeds the provided `score` parameter, which is a number between `0` and `1.0`. The higher the score, the more likely it is that the device has been tampered with.

Enable Authenticator app policies

Use the **JSON Authenticator Policies** property in the Combined MFA Registration node to enable policies.

Specify the policies and their parameters to apply to the device being registered in JSON format, as follows:

```
{
    "policyName" : { policyParameters | empty }
}
```

Example:

```
{
  "biometricAvailable": { },
  "deviceTampering": {
    "score": 0.8
  }
}
```

Next steps

You can add support for app policies in your own Android and iOS applications, by using the Ping (ForgeRock) Authenticator module.

For more information, refer to Integrate authenticator app policies.

Troubleshoot the ForgeRock Authenticator

Multi-factor authentication requires you to register a device, which is used as an additional factor when you log in to AM.

The following table summarizes different tasks related to devices used for multi-factor authentication:

Task	Resources
Recover user accounts Learn how to recover a user account when the user has lost their registered device, or when their device has become out of sync with AM.	• Recover after replacing a lost device • Recover after a device becomes out of sync
Reset registered devices In some scenarios, for example, when users are not able to access their recovery codes, you may need to reset their registered devices to allow them to register again.	• Reset registered devices over REST

Recover after replacing a lost device

If you register a device with AM and then lose it, you must authenticate to AM using a recovery code. After deleting the lost device, you can register a new device.

1. Access the list of recovery codes you saved when registering the lost device.

   If you did not save the recovery codes when you registered the device, contact your administrator to remove the device from your user profile instead of following these steps.

2. Begin to sign in as you normally would.

   When prompted to use a multi-factor option, click the **Use Recovery Code** link.

3. Enter the recovery code when prompted.

   Because recovery codes are valid for a single use only, remove the code you used from your list.

   AM lets you sign in to access your profile page.

4. Under **Dashboard** > **Authentication Devices**, click the context menu button for the lost device, and click **Delete**.

5. Register your new device by signing out, then accessing the protected resource that requires MFA.

Recover after a device becomes out of sync

A device that generates one-time passwords can get out of sync with the OATH authentication service in some cases. If you repeatedly enter valid one-time passwords that appear to be valid passwords, but AM rejects the passwords as unauthorized, your device is likely out of sync.

To resynchronize your device, you must authenticate with a recovery code, and register the device again. Follow the steps in Recover after replacing a lost device.

If you did not save the recovery codes when you registered the device, contact your administrator to remove the device from your user profile instead.

Reset registered devices over REST

As described in Recover after replacing a lost device, a user who has lost a mobile phone registered with AM can register a replacement device by authenticating using a recovery code, deleting their existing device, and then registering a new device.

Additional support is required for users who lose mobile devices but did not save their recovery codes when they initially registered the device, and for users who have used up all their recovery codes.

AM provides a REST API to reset a device profile by deleting information about a user's registered device. Both the user and administrator accounts can use the REST API to reset a device profile. Administrators can:

- Provide authenticated users with a self-service page that calls the REST API to reset their devices.
- Call the REST API themselves to reset a user's device profiles.
- Call the REST API themselves to reset a device that is out of sync, where the HOTP counter exceeds the HOTP threshold window and requires a reset.

Reset OATH devices

To reset a user's OATH device profile, perform an HTTP POST to `/users/{user}/devices/2fa/oath?_action=reset`.

When making a REST API call, specify the realm in the path component of the endpoint.

Authenticate the request with the SSO cookie token belonging to an administrator.

Advanced Identity Cloud

The following example resets the OATH devices for a user in the `alpha` realm.

In PingOne Advanced Identity Cloud, use the `_ID` property of the user, not their username. The `demo` users' `_id` in this example is `014c54bd-6078-4639-8316-8ce0e7746fa4`.

Reset OATH device

```
$ curl \
--request POST \
--header "Content-Type: application/json" \
--header "Accept-API-Version: resource=1.0" \
--header "<session-cookie-name>: AQIC5w…2NzEz*" \
--data '{}' \
'https://<tenant-env-fqdn>/am/json/realms/root/realms/alpha/users/014c54bd-6078-4639-8316-8ce0e7746fa4/
devices/2fa/oath?_action=reset'
```

Self-managed PingAM server

The following example resets the OATH device of a user named `demo` in a realm called `mySubrealm`:

Reset OATH device

```
$ curl \
--request POST \
--header "Content-Type: application/json" \
--header "Accept-API-Version: resource=1.0" \
--header "iplanetDirectoryPro: AQIC5w…2NzEz*" \
--data '{}' \
'https://openam.example.com:8443/openam/json/realms/root/realms/mySubrealm/users/demo/devices/2fa/oath?
_action=reset'
```

Result

```
{
    "result":true
}
```

The reset action deletes the OATH device profile, which by default has a limit of one profile per device, and sets the **Select to Enable Skip** option to its default value of `Not Set`.

Reset push devices

To reset push devices over REST, perform an HTTP POST to `/users/{user}/devices/2fa/push?_action=reset`.

When making a REST API call, specify the realm in the path component of the endpoint.

Authenticate the request with the SSO cookie token belonging an administrator.

Advanced Identity Cloud

The following example resets the push devices for a user in the `alpha` realm.

In PingOne Advanced Identity Cloud, use the `_ID` property of the user, not their username. The `demo` users' `_id` in this example is `014c54bd-6078-4639-8316-8ce0e7746fa4`.

Reset push device

```
$ curl \
--request POST \
--header "Content-Type: application/json" \
--header "Accept-API-Version: resource=1.0" \
--header "<session-cookie-name>: AQIC5w…2NzEz*" \
--data '{}' \
'https://<tenant-env-fqdn>/am/json/realms/root/realms/alpha/users/014c54bd-6078-4639-8316-8ce0e7746fa4/
devices/2fa/push?_action=reset'
```

Self-managed PingAM server

The following example resets the push device of a user named `demo` in a realm called `mySubrealm`:

Reset push device

```
$ curl \
--request POST \
--header "Content-Type: application/json" \
--header "Accept-API-Version: resource=1.0" \
--header "iplanetDirectoryPro: AQIC5w…2NzEz*" \
--data '{}' \
'https://openam.example.com:8443/openam/json/realms/root/realms/mySubrealm/users/demo/devices/2fa/push?
_action=reset'
```

Result

```
{
    "result":true
}
```

Ping (ForgeRock) Authenticator module

Server support:

- ✕ PingOne
- ✓ PingOne Advanced Identity Cloud
- ✓ PingAM
- ✕ PingFederate

SDK support:

- ✓ Ping SDK for Android
- ✓ Ping SDK for iOS
- ✕ Ping SDK for JavaScript

The Ping (ForgeRock) Authenticator module helps you build the functionality of the ForgeRock Authenticator application into your own Android and iOS apps.

The module supports:

- Time-based one-time passwords (TOTP)
- HMAC-based one-time passwords (HOTP)
- Push notifications

Topics

Getting started

Learn how to get started with the Ping (ForgeRock) Authenticator module in your Android and iOS app projects.

Use cases

Discover how to integrate some common use case scenarios into your applications by using the Ping (ForgeRock) Authenticator module.

API Reference

Browse API reference documentation for the Ping (ForgeRock) Authenticator module.

Getting started with the Ping (ForgeRock) Authenticator module

Refer to the following topics to set up and get started with the Ping (ForgeRock) Authenticator module in your Android and iOS app projects:

Set up your Ping (ForgeRock) Authenticator module project

Learn how to configure compile options, add module dependencies, and declare the permissions required to run the Ping (ForgeRock) Authenticator module in your Android and iOS apps.

Initialize the Ping (ForgeRock) Authenticator module

After setting up your project, find out how to start the Ping (ForgeRock) Authenticator module so you can begin implementing your MFA use cases.

Optional tasks

Customize the storage client

Optionally, implement and register your own `StorageClient` interface to override the default storage mechanisms.

For example, you could use an SQLite database or other storage destination.

Set up your Ping (ForgeRock) Authenticator module project

Android

Set compile options

The Ping SDK for Android requires at least Java 8 (v1.8). Configure the compile options in your project to use this version, or newer.

For example, to specify Java 17, in your `build.gradle` file, add the following code at the top level:

```
kotlin {
  jvmToolchain {
    languageVersion.set(JavaLanguageVersion.of(17))
  }
}
```

Add module dependencies

Add the following dependency to use the Ping (ForgeRock) Authenticator module in your Android applications:

```
dependencies {
    ...
    implementation 'org.forgerock:forgerock-authenticator:4.8.1'
}
```

Additional dependencies you may require:

Feature	Dependency
Push notifications	`com.google.firebase:firebase-messaging:20.2.0`

Request notification permissions

To process push notifications successfully on Android 13 (API level 33) and later, the app must request the new Notification runtime permission for sending non-exempt notifications from an app.

Declare the permission

To request the new notification permission from your app, update your app to target Android 13 (API level 33) and declare `POST_NOTIFICATIONS` in your app's manifest file, as in the following code snippet:

```
<manifest ...>
    <uses-permission android:name="android.permission.POST_NOTIFICATIONS"/>
    <application ...>
        ...
    </application>
</manifest>
```

iOS

Install the Ping (ForgeRock) Authenticator module using CocoaPods

CocoaPods is a dependency manager for iOS projects, and is a simple way to integrate the Ping (ForgeRock) Authenticator module into your application.

1. If you do not already have CocoaPods, install the latest version.
2. In a terminal window, run the following command to create a new Podfile:

   ```
   pod init
   ```

3. Add the following lines to your Podfile:

   ```
   pod 'FRAuthenticator'
   ```

4. Run the following command to install pods:

   ```
   pod install
   ```

Install the Ping (ForgeRock) Authenticator module using Swift Package Manager (SPM)

1. With your project open in **Xcode**, select **File** > **Add Package Dependencies**.
2. In the search bar, enter the Ping SDK for iOS repository URL: `https://github.com/ForgeRock/forgerock-ios-sdk`.
3. Select the `forgerock-ios-sdk` package, and then click **Add Package**.
4. In the **Choose Package Products** dialog, ensure that the `FRAuthenticator` library is added to your target project:

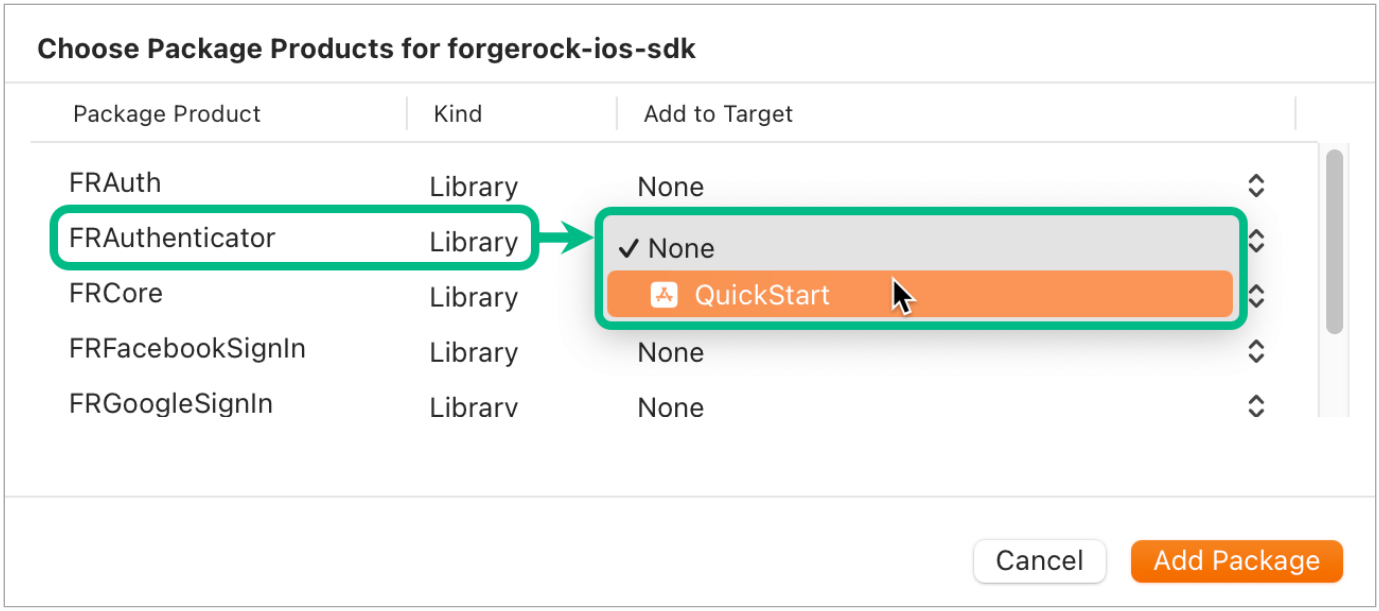


Figure 1. Adding the 'FRAuthenticator' module to an iOS project.

5. Click **Add Package**.
6. In your project, import the module:

```
// Import the Ping (ForgeRock) Authenticator module
import FRAuthenticator
```

Initialize the Ping (ForgeRock) Authenticator module

Android

Start the module

To use the features of the Ping (ForgeRock) Authenticator module, add code similar to the following to your application:

```
FRAClient fraClient = new FRAClient.FRAClientBuilder()
    .withContext(this)
    .start();
```

iOS

Start the module

To use the features of the Ping (ForgeRock) Authenticator module in your iOS app, first import the `FRAuthenticator`:

```
import FRAuthenticator
```

And secondly, add code similar to the following to your application:

```
FRAClient.start()
```

Customize the storage client

The Ping (ForgeRock) Authenticator module lets you customize the storage client and manages all data through that client.

Android

The Ping (ForgeRock) Authenticator module offers a default storage client that uses `SecuredSharedPreferences`, an encrypted storage mechanism built on Android SharedPreferences. It is available in the `forgerock-core` module.

`SecuredSharedPreferences` stores and manages all shared secret account information and notifications.

The Authenticator module lets you customize the `StorageClient`. You can implement the `StorageClient` interface, and register your own `StorageClient` in the module.

> **Important**
>
> For maximum compatibility with devices from different manufacturers we highly recommend that you implement your own custom storage client.
> The default client that is built on `SharedPreferences` can behave unpredictably on devices from certain manufacturers that customize the Android operating system.
> For example, you might not be able to access the registered PUSH or OATH accounts.

You can implement your custom storage client in any location you choose, for example you could use the SQLite-based EncryptedSharedPreferences.

The Ping (ForgeRock) Authenticator module uses your storage client and manages all data through that client.

To customize the `StorageClient`, implement the following interfaces:

```
public interface StorageClient {
    /**
     * Get the Account object with its id
     * @param accountId The account unique ID
     * @return The account object.
     */
    Account getAccount(String accountId);
    /**
     * Get all accounts stored in the system.
     * @return The complete list of accounts.
     */
    List<Account> getAllAccounts();
    /**
     * Delete the Account that was passed in.
     * @param account The account object to delete.
     * @return boolean as result of the operation
     */
    boolean removeAccount(Account account);
    /**
     * Add or Update the Account to the storage system.
     * @param account The Account to store or update.
     * @return boolean as result of the operation
     */
    boolean setAccount(Account account);
    /**
     * Get the mechanisms associated with an account.
     * @param account The Account object
     * @return The list of mechanisms for the account.
     */
    List<Mechanism> getMechanismsForAccount(Account account);
    /**
     * Get the mechanism by UUID.
     * @param mechanismUID The uniquely identifiable UUID for the mechanism
     * @return The mechanism object.
     */
    Mechanism getMechanismByUUID(String mechanismUID);
    /**
     * Delete the mechanism uniquely identified by an id.
     * @param mechanism The mechanism object to delete.
     * @return boolean as result of the operation
     */
    boolean removeMechanism(Mechanism mechanism);
    /**
     * Add or update the mechanism to the storage system.
     * If the owning Account is not yet stored, store that as well.
     * @param mechanism The mechanism to store or update.
     * @return boolean as result of the operation
     */
    boolean setMechanism(Mechanism mechanism);
    /**
     * Get all notifications for within the mechanism.
     * @param mechanism The mechanism object
     * @return The list of notifications for the mechanism.
     */
    List<PushNotification> getAllNotificationsForMechanism(Mechanism mechanism);
    /**
     * Delete the pushNotification uniquely identified by an id.
     * @param pushNotification The pushNotification object to delete.
     */
    boolean removeNotification(PushNotification pushNotification);
```

```
    /**
     * Add or update the pushNotification to the storage system.
     * @param pushNotification The pushNotification to store.
     * @return boolean as result of the operation
     */
    boolean setNotification(PushNotification pushNotification);
    /**
     * Whether the storage system currently contains any data.
     * @return True if the storage system is empty, false otherwise.
     */
    boolean isEmpty();
}
```

> **Tip**
>
> For each method of getting an `Account`, `Mechanism`, or `PushNotification` object, your `StorageClient` should only be responsible for retrieving the objects, and not any other object associated with it.
> For example, when retrieving `Account` objects, the `StorageClient` should not be responsible for retrieving `Mechanism` and `PushNotification` objects. All object mapping and associations are handled by the Ping (ForgeRock) Authenticator module itself.

After implementing your custom `StorageClient`, register it to `FRAClient` as follows:

```
// Initiate your custom StorageClient
StorageClient customStorageClient = CustomStorageClient()

// Register it to FRAClient
FRAClient fraClient = new FRAClient.FRAClientBuilder()
                .withContext(this)
                .withStorage(customStorageClient)
                .start();
```

> **Warning**
>
> You must register the `StorageClient` **before** you start the Ping SDK.
> The `StorageClient` used by `FRAClient` cannot be changed after the Ping SDK starts.

iOS

The ForgeRock Authenticator default storage client utilizes both Apple's Keychain Service, and Secure Enclave.

This means that the Ping (ForgeRock) Authenticator module safely stores all shared secrets, account information, and notifications.

You can also customize the `StorageClient`. You can implement the `StorageClient` protocol, and register your own `StorageClient` with the Ping (ForgeRock) Authenticator module.

For example, you could customize `StorageClient` to use SQLite, CoreData, or any other storage destination.

The Ping (ForgeRock) Authenticator module uses your storage client and manages all data through that client.

To customize `StorageClient` you must implement the following interfaces:

```
/// StorageClient protocol represents predefined interfaces and protocols for FRAuthenticator's storage method.
public protocol StorageClient {

    /// Stores Account object into Storage Client and returns discardable Boolean result of operation.
    /// - Parameter account: Account object to store.
    @discardableResult func setAccount(account: Account) -> Bool

    /// Removes Account object from Storage Client, and returns discardable Boolean result of operation.
    /// - Parameter account: Account object to remove.
    @discardableResult func removeAccount(account: Account) -> Bool

    /// Retrieves Account object with its unique identifier.
    /// - Parameter accountIdentifier: String value of Account's unique identifier.
    func getAccount(accountIdentifier: String) -> Account?

    /// Retrieves all Account objects stored in Storage Client.
    func getAllAccounts() -> [Account]

    /// Stores Mechanism object into Storage Client, and returns discardable Boolean result of operation.
    /// - Parameter mechanism: Mechanism object to store.
    @discardableResult func setMechanism(mechanism: Mechanism) -> Bool

    /// Removes Mechanism object from Storage Client, and returns discardable Boolean result of operation.
    /// - Parameter mechanism: Mechanism object to remove.
    @discardableResult func removeMechanism(mechanism: Mechanism) -> Bool

    /// Retrieves all Mechanism objects stored in Storage Client.
    /// - Parameter account: Account object that is associated with Mechanism(s).
    func getMechanismsForAccount(account: Account) -> [Mechanism]

    /// Retrieves Mechanism object with given Mechanism UUID.
    /// - Parameter uuid: UUID of Mechanism.
    func getMechanismForUUID(uuid: String) -> Mechanism?

    /// Stores PushNotification object into Storage Client, and returns discardable Boolean result of operation.
    /// - Parameter notification: PushNotification object to store.
    @discardableResult func setNotification(notification: PushNotification) -> Bool

    /// Removes PushNotification object from Storage Client, and returns discardable Boolean result of operation.
    /// - Parameter notification: PushNotification object to remove.
    @discardableResult func removeNotification(notification: PushNotification) -> Bool

    /// Retrieves all Notification objects from Storage Client with given Mechanism object.
    /// - Parameter mechanism: Mechanism object that is associated with Notification(s).
    func getAllNotificationsForMechanism(mechanism: Mechanism) -> [PushNotification]

    /// Returns whether or not StorageClient has any data stored.
    @discardableResult func isEmpty() -> Bool
}
```

Tip

For each method of getting an `Account`, `Mechanism`, or `PushNotification` object, your `StorageClient` should only be responsible for retrieving the objects, and not any other object associated with it.
For example, when retrieving `Account` objects, the `StorageClient` should not be responsible for retrieving `Mechanism` and `PushNotification` objects. All object mapping and associations are handled by the Ping (ForgeRock) Authenticator module itself.

After implementing your custom `StorageClient`, register it with your `FRAClient` as follows:

```
// Initiate your custom StorageClient
let customStorageClient = CustomStorageClient()
// Register it with your FRAClient
FRAClient.setStorage(storage: customStorageClient)
// Initiate the SDK
FRAClient.start()
```

> ⚠ **Warning**
>
> You must register the `StorageClient` **before** you start the Ping SDK.
> Once the SDK starts, the `StorageClient` used by `FRAClient` cannot be changed.

Implement your use cases with the Ping (ForgeRock) Authenticator module

Find out how to integrate some common use case scenarios into your applications by using the Ping (ForgeRock) Authenticator module.

Integrate MFA using push notifications

In this use case, you integrate the ability to authenticate a user with push notifications into your app.

To receive push notifications when authenticating, end users must register an Android or iOS device running your app built with the Ping (ForgeRock) Authenticator module.

Read more »

Integrate MFA using OATH one-time passwords

In this use case you integrate the ability to generate a single-use, one-time password into your application by using the Ping (ForgeRock) Authenticator module.

The Ping (ForgeRock) Authenticator module supports time-based and HMAC-based one-time passwords.

Read more »

Integrate authenticator app policies

You can build and distribute your own authenticator app to your users so that they can participate in multi-factor authentication journeys, by using the Ping (ForgeRock) Authenticator module.

To help ensure the security of your app—and therefore your system—you can enable *authenticator app policies*.

Read more »

Integrate MFA using push notifications

Applies to:

- ✓ Ping SDK for Android
- ✓ Ping SDK for iOS
- ✗ Ping SDK for JavaScript

This use case explains how to integrate support for push authentication into your Android or iOS projects that use the Ping (ForgeRock) Authenticator module.

Tasks

Step 1. Configure Push notifications for Android

In this step, you configure Google Firebase Cloud Messaging (FCM), which handles sending the push notifications to Android devices.

You create a service account that provides access to the service for third-parties.

Step 2. Configure Push notifications for iOS

In this step, you configure Apple Push Notification service (APNS), which handles sending the push notifications to iOS devices.

You create a key that provides access to the service for third-parties.

Step 3. Configure Push notifications in AWS

In this step, you use the service account and key created in the previous steps to set up Amazon Simple Notification Service (SNS) to be able to route push notification messages to Android and iOS devices.

You also create a service account and associated access token to provide access to the service to your server.

Configure a server for push notifications

In this step, you configure your server to connect to SNS so that it can send out push notifications.

You also create an authentication journey that will register your client application as an MFA device, and send out push notifications.

Step 5: Configure the app for push notifications

In this step, you configure your application projects to use either Firebase Cloud Messaging or the Apple Push Notification service.

Step 6. Configure the Ping (ForgeRock) Authenticator module for push notifications

In this final step, you add the code to your application that obtains the unique device code required to ensure push notifications reach their intended audience.

You also add code that leverages the Ping (ForgeRock) Authenticator module to handle the push registration and authentication journey you created earlier.

Step 1. Configure Push notifications for Android

In this step, you configure Google Firebase Cloud Messaging (FCM), which handles sending the push notifications to Android devices.

You create service account that provides access to the service for third-parties.

Prerequisites

- A Google cloud account, with access to the Firebase console.
- An Android application into which you want to integrate push notifications.

> **Tip**
>
> For development purposes, you can download the Ping (ForgeRock) Authenticator module sample app from GitHub.

Create a project in Google Firebase

1. Log in to the Google Firebase console.
2. Click **Add project**.
3. Enter a name for the project, for example `sdk-authenticator-push`, and then click **Continue**.
4. Disable Google Analytics for the project, and then click **Create project**.
5. When your Firebase project is ready, click **Continue**.

Add your Android app to the Firebase project

1. Log in to the Google Firebase console.
2. In the left menu, in **Project Overview**, click the gear icon (⚙) and then click **Project settings**.

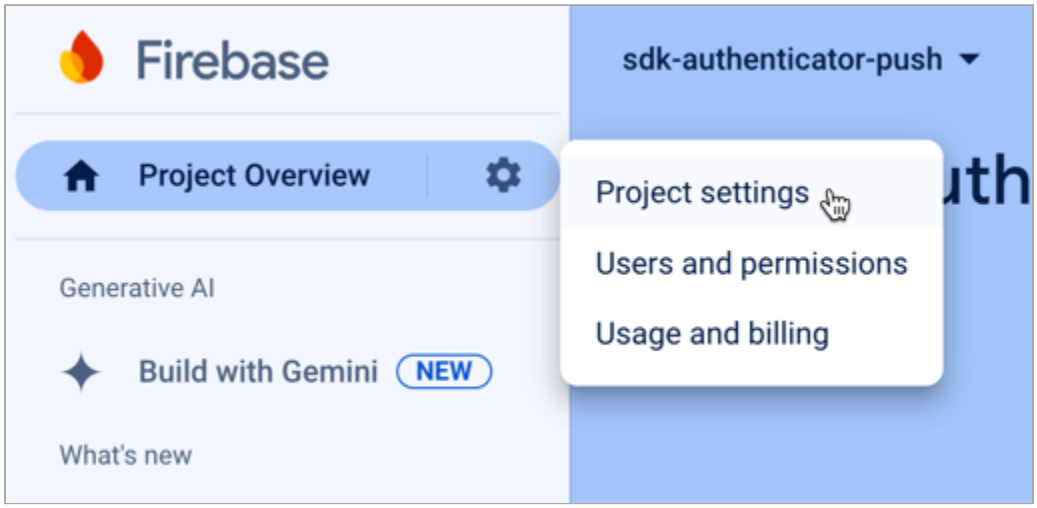


Figure 1. Opening the project settings panel in the Firebase console.

3. In **Your apps**, click **Add app**.
4. Click the Android icon (🤖).
5. In the **Register app** section:
    1. In Android package name, enter the package of the app into which you are integrating push notifications.

       For example, `org.forgerock.authenticator.sample`.
    2. In **App nickname**, enter a user-friendly name for the app.

       For example, `Authenticator Push Sample`.
    3. Click **Register app**.
6. In the **Download and then add config file** section:
    1. Click **Download google-services.json**, and keep the file somewhere safe.

       You will need the file when configuring your application to access Firebase Cloud Messaging in a later step.
    2. Click **Next**.
7. In the **Add Firebase SDK** section, click **Next**.

   You will add the Firebase SDKs to your application in a later step.
8. In the **Next steps** section, click **Continue to the console**.

Create a key for the Firebase service account

1. Log in to the Google Firebase console.
2. In the left menu, in **Project Overview**, click the gear icon (⚙) and then click **Project settings**.

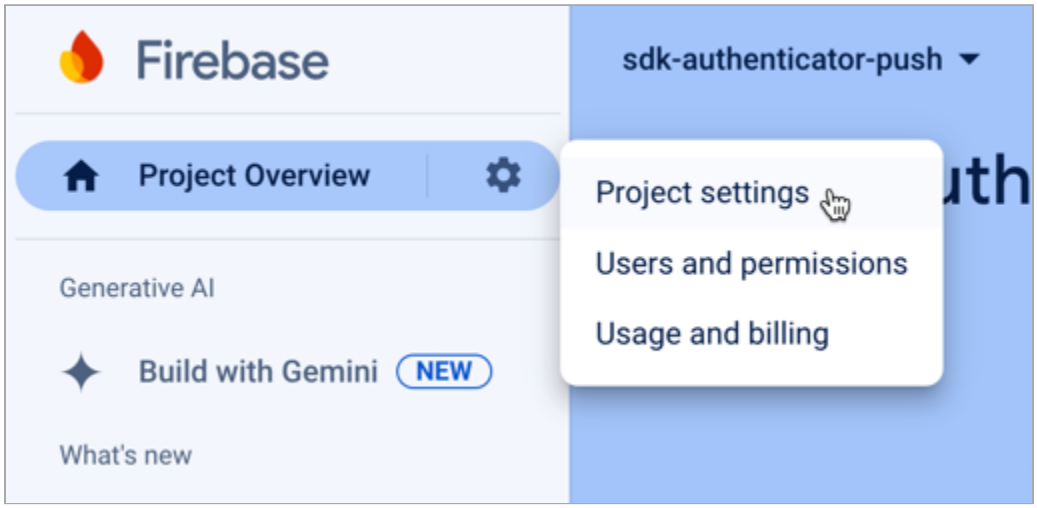


Figure 2. Opening the project settings panel in the Firebase console.

3. On the **Cloud Messaging** tab, click **Manage service accounts**.

   The link opens the Google Cloud **IAM & Admin** page, and displays the service accounts automatically generated when you created the Firebase project:

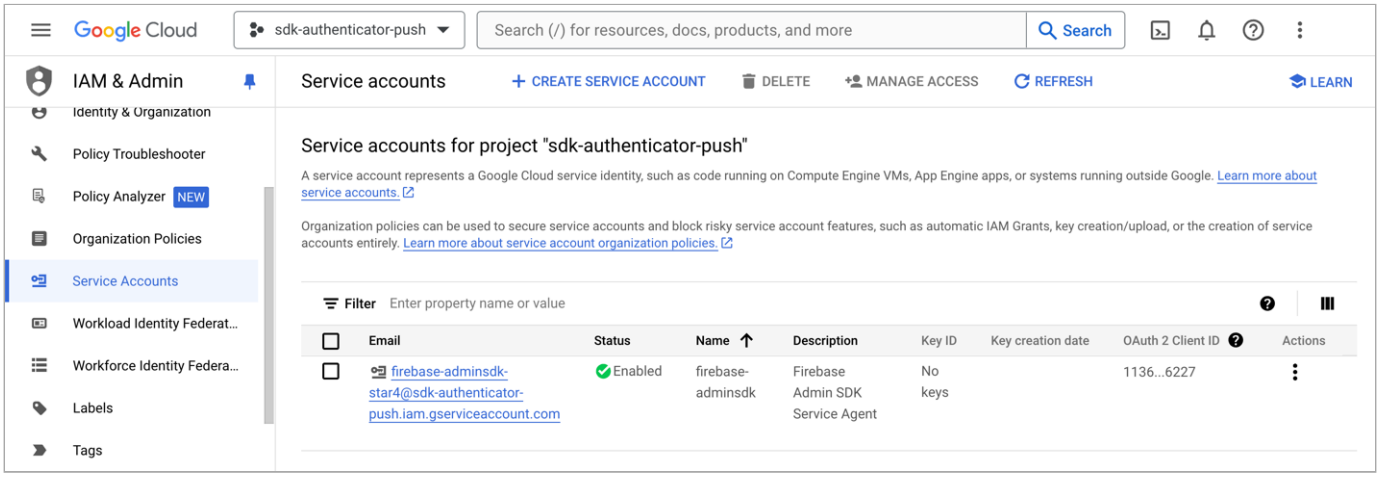


Figure 3. Google Cloud console showing the generated Firebase service account.

4. In the **Actions** column, click the vertical ellipsis icon (⋮) next to the service account, and then click **Manage keys**.
5. On the **Keys** page, click **Add key**, and then click **Create new key**.
6. Select **JSON**, and then click **Create**.

   The page generates a key for the Firebase service account, and downloads the JSON file.

   You will upload this file when you configure Push notifications in AWS.

Step 2. Configure Push notifications for iOS

In this step, you configure Apple Push Notification service (APNS), which handles sending the push notifications to iOS devices.

You create a key that provides access to the service for third-parties.

Prerequisites

- An Apple developer *Admin* account.

  Developer accounts cannot create the required keys.

- An iOS application in XCode configured with the **Push Notifications** capability.

  You can add capabilities when creating an iOS app project, or add them to an existing project. Refer to Adding capabilities to your app in the Apple developer documentation.

> **Tip**
>
> For development purposes, you can download the Ping (ForgeRock) Authenticator module sample app from GitHub.

Register a new key for APNs

1. With an admin account, log in to the Apple Developer console.
2. Navigate to **Program resources** > **Certificates, IDs & Profiles** > **Keys**.
3. Next to the **Keys** label, click the **Add** icon (⊕).
4. Enter a **Key Name**.

   For example, `APNs key for Push`.
5. Select **Apple Push Notifications service (APNs)**.
6. Click **Continue**.
7. Check the details of the key, and then click **Register**.
8. On the **Download Your Key** page:
   1. Make a note of the 10-character **Key ID**.

      For example, `YCH15BO820`.
   2. Click **Download** and keep a copy of the `.p8` file safe.

> **Important**
>
> You cannot download or view the key again, so ensure you have a local copy.

9. Click **Done**.

You will upload the `.p8` file and use the key ID when you configure Push notifications in AWS.

Step 3. Configure Push notifications in AWS

In this step, you use the service account and key created in the previous steps to set up Amazon Simple Notification Service (SNS) to be able to route push notification messages to Android and iOS devices.

You also create a service account and associated access token to provide access to the service to your server.

Set up AWS for Android push notifications

1. Log in to the AWS console: https://console.aws.amazon.com/console/home
2. In the search bar, enter `SNS`, and then select **Simple Notification Service** from the list of results.

   > **Tip**
   >
   > Click the star icon (☆) to pin the service to the toolbar in the AWS console.

3. In the left menu, navigate to **Mobile** > **Push notifications**.
4. In the **Platform applications** panel, click **Create platform application**.
5. On the **Create platform application** page:
   1. In **Application name**, enter a name for the platform application.

      For example, `Android_Push_Messaging`.

   2. In **Push notification platform**, select **Firebase Cloud Messaging (FCM)**.

      The page displays the **Firebase Cloud Messaging Credentials** section.

      1. In **Authentication method**, select **Token**.

         The page displays additional fields.

      2. In **Service JSON**, click **Choose file**, and navigate to the JSON file that you downloaded from Firebase when you created a Firebase key previously.

   3. Click **Create platform application**.

      The page creates the application and displays the details pane:

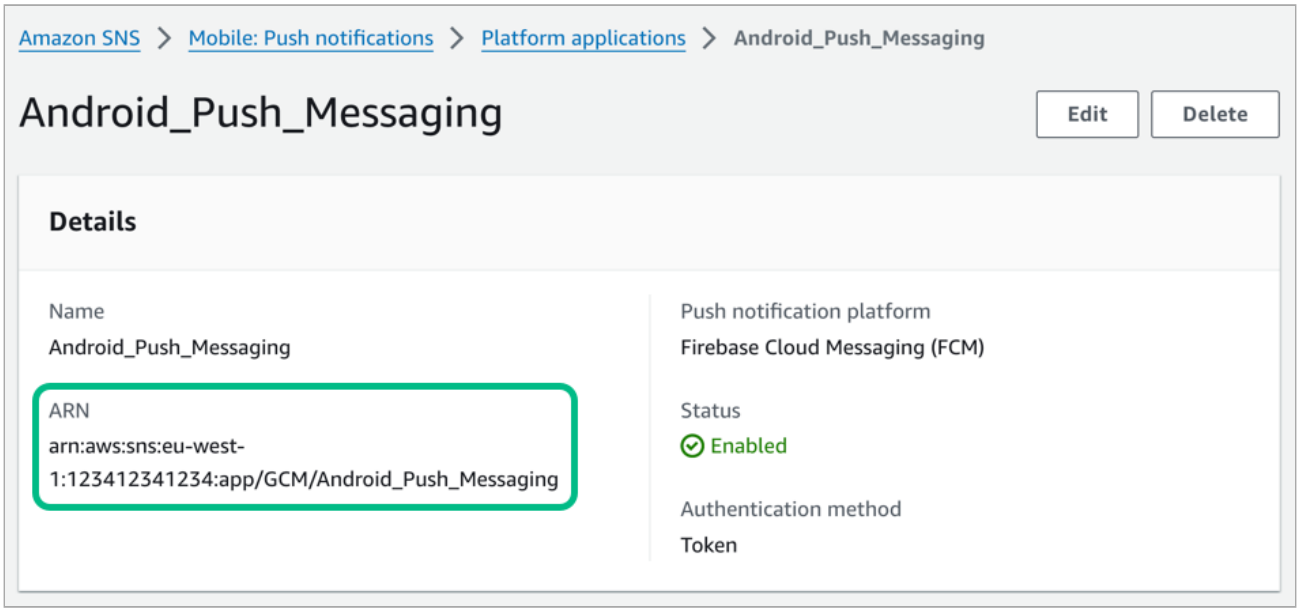


Figure 1. An Android platform application in AWS, showing the ARN.

6. Make a note of the **ARN** value. You need this value when you Configure a server for push notifications.

Set up AWS for iOS push notifications

1. Log in to the AWS console: https://console.aws.amazon.com/console/home
2. In the search bar, enter `SNS`, and then select **Simple Notification Service** from the list of results.

   > **Tip**
   >
   > Click the star icon (☆) to pin the service to the toolbar in the AWS console.

3. In the left menu, navigate to **Mobile** > **Push notifications**.
4. In the **Platform applications** panel, click **Create platform application**.
5. On the **Create platform application** page:
   1. In **Application name**, enter a name for the platform application.

      For example, `iOS_Push_Messaging`.
   2. In **Push notification platform**, select **Apple iOS/VoIP/MacOS**.

      The page displays the **Apple credentials** section.
   3. In the **Apple credentials** section:
      1. In **Push service**, select **iOS**.
      2. In **Authentication method**, select **Token**.

         The page displays additional fields.
      3. In **Signing key**, click **Choose file**, and navigate to the `.p8` file that you downloaded from Apple when you registered a new key for APNs.

         After selecting the file, the page populates the **Signing key** text field with the private key from the `.p8` file.
      4. In **Signing key ID**, enter the 10-digit ID of the key you created when you registered a new key for APNs.

         For example, `YUGX2BO820`.
      5. In **Team ID**, enter the ID of your team in the Apple Developer Program.

         > **Tip**
         >
         > You can view your Team ID on the Membership details page in the Apple developer console.

      6. In **Bundle ID**, enter the bundle ID of the iOS application you are adding push notifications to.

         For example, `com.forgerock.authenticator.sample`.
   4. Click **Create platform application**.

      The page creates the application and displays the details pane:

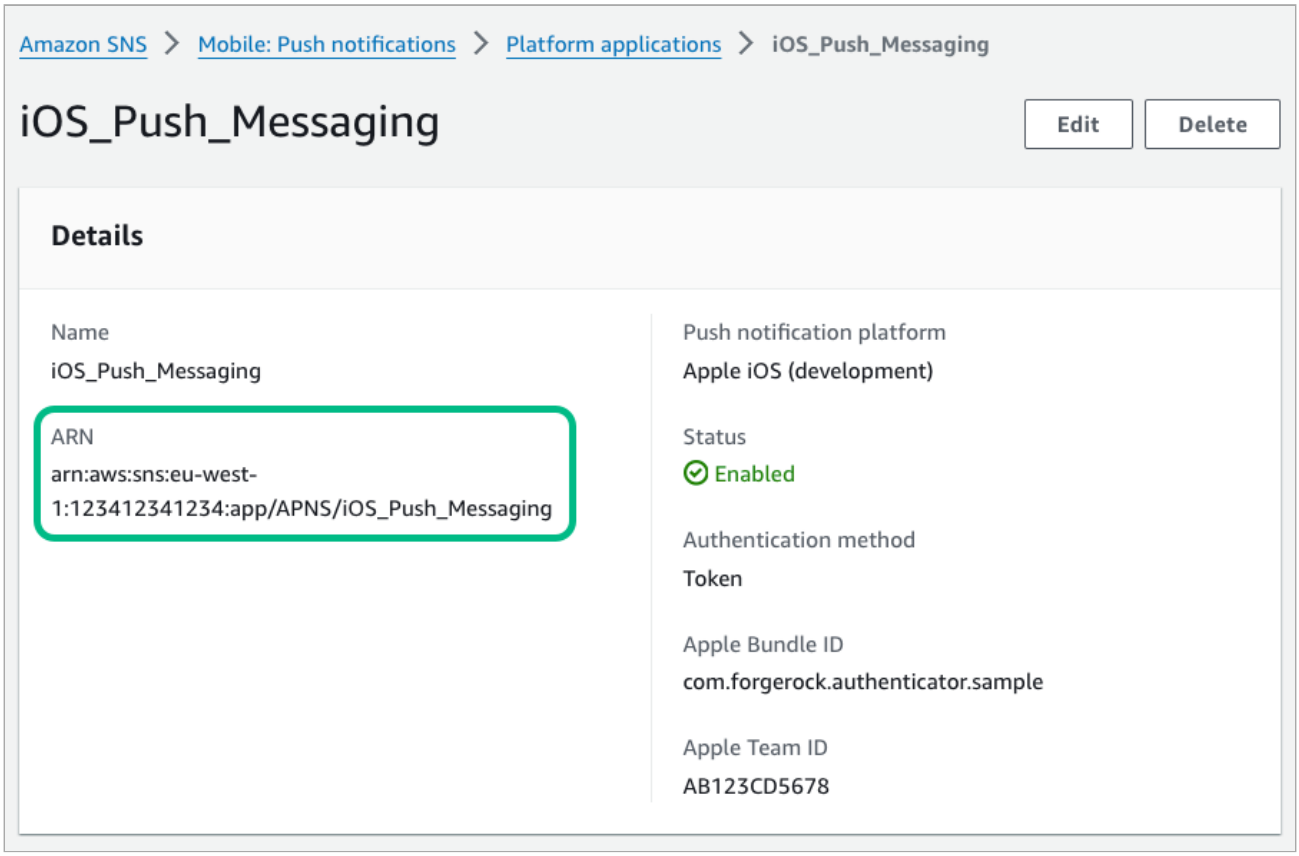


Figure 2. An iOS platform application in AWS, showing the ARN.

6. Make a note of the **ARN** value. You need this value when you Configure a server for push notifications.

Create a service account with access to the ARN endpoints

1. Log in to the AWS console: https://console.aws.amazon.com/console/home
2. In the search bar, enter `IAM`, and then select **IAM** from the list of results.

> **Tip**
>
> Click the star icon (☆) to pin the service to the toolbar in the AWS console.

3. In the left menu, navigate to **Access management** > **Users**.
4. Click **Create user**.
5. In **User name**, enter a name for the user account that the access key will represent.

   For example, `sns_arn_user`.
6. Click **Next**.
7. In **Permissions options**, select **Attach policies directly**.

   The page displays additional fields.
8. In **Permissions policies**, in the search bar, enter `SNSFull`, and then select the checkbox next to `AmazonSNSFullAccess`.

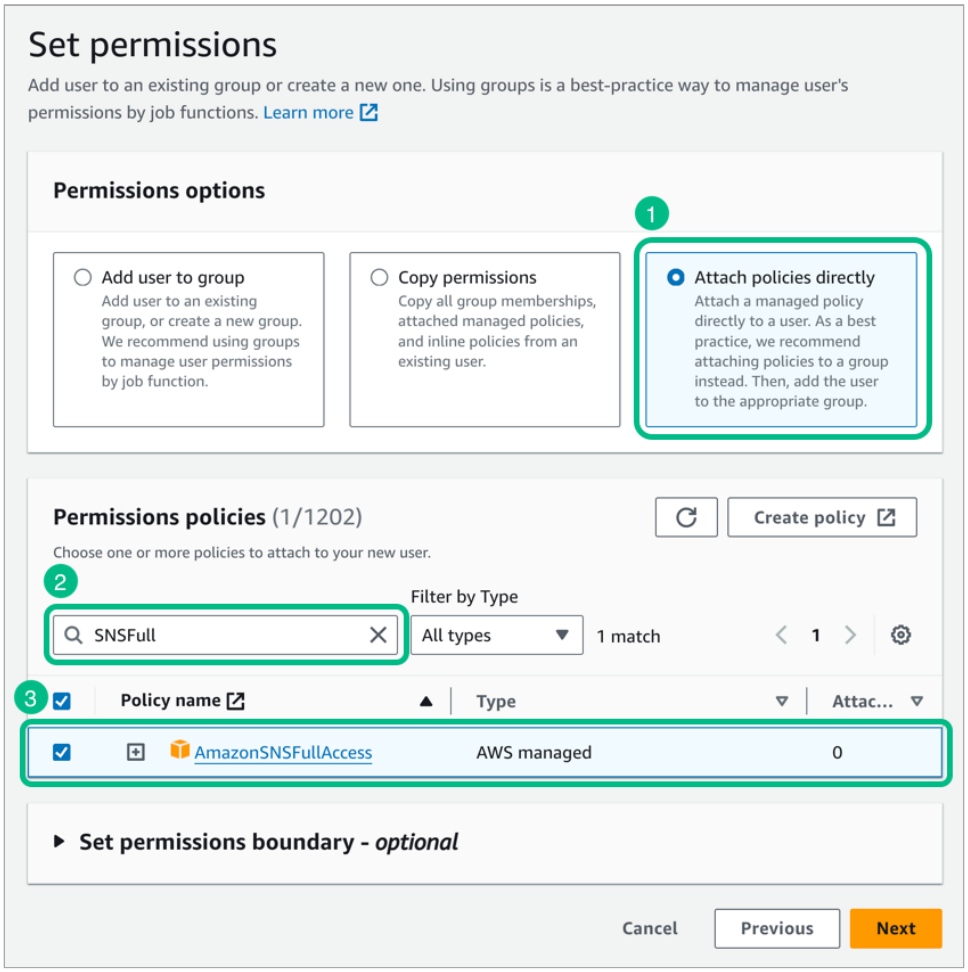


Figure 3. Adding the SNS permission policy to a user in AWS IAM.

9. Click **Next**, review the details of the account, and then click **Create user**.

Create an access token for the service account

1. Log in to the AWS console: https://console.aws.amazon.com/console/home
2. In the search bar, enter `IAM`, and then select **IAM** from the list of results.

> **Tip**
>
> Click the star icon (☆) to pin the service to the toolbar in the AWS console.

3. In the left menu, navigate to **Access management** > **Users**, and then click the service account you created previously.
4. In the **Summary** pane, click **Create access key**.
5. On the **Access key best practices & alternatives** page:
    1. In **Use case**, select **Third-party service**
    2. Under **Confirmation**, select the **I understand the above recommendation and want to proceed to create an access key** checkbox.
    3. Click **Next**.

6. In **Description tag value**, enter the purpose of the access key.

   For example, `server access to SNS ARN endpoints`

7. Click **Create access key**.

8. Make a note of the provided values:

   1. **Access Key ID**.

      For example, `AKIAXOSPRCH15LEES`

   2. **Secret access key**.

      For example, `9eF7EcWMZzChI51BBHkLeElXk8R3XHv7/n7QSiwoUFJ`

   > **Tip**
   >
   > Click **Download .csv file** to download a file containing the values for safe-keeping.

9. Click **Done**.

Step 4. Configure an PingOne Advanced Identity Cloud or PingAM server for push notifications

In this step, you configure your server to connect to SNS so that it can send out push notifications.

You also create an authentication journey that will register your client application as an MFA device, and send out push notifications.

Add the Authenticator (Push) service

In this step you configure your server to operate with the Ping (ForgeRock) Authenticator module.

Advanced Identity Cloud

1. Log in to the Advanced Identity Cloud admin UI as an administrator.
2. In the left menu pane, select **Native Consoles** > **Access Management**.

   The realm overview for the Alpha realm displays.
3. Select **Services**, and then click **+ Add a Service**.
4. In **Choose a service type**, select **ForgeRock Authenticator (Push) Service**, and then click **Create**.
5. Click **Save Changes** to accept the default settings.

Self-managed PingAM server

1. Log in to the AM admin UI as an administrator.

   The realm overview for the Top Level Realm displays.

2. Select **Services**, and then click **+ Add a Service**.

3. In **Choose a service type**, select **ForgeRock Authenticator (Push) Service**, and then click **Create**.

4. Click **Save Changes** to accept the default settings.

Connect your server to Amazon SNS

In this step you configure your server with the settings it needs to be able to contact Amazon SNS to send push notifications to mobile devices.

Advanced Identity Cloud

1. Log in to the Advanced Identity Cloud admin UI as an administrator.
2. In the left menu pane, select **Native Consoles** > **Access Management**.

   The realm overview for the Alpha realm displays.
3. Select **Services**, and then click **+ Add a Service**.
4. In **Choose a service type**, select **Push Notification Service**.
5. In **SNS Access Key ID**, enter the **Access key ID** value of the access token you created previously.

   For example, `AKIAXOSPRCH15LEES`.

   > **Tip**
   >
   > If you downloaded the CSV file when you created the access key, the first value in the file is the **Access Key ID**.
6. In **SNS Access Key Secret**, enter the **Access key** value from the access token you created previously.

   For example, `9eF7EcWMZzChI51BBHkLeElXk8R3XHv7/n7QSiwoUFJ`.

   > **Tip**
   >
   > If you downloaded the CSV file when you created the access key, the second value in the CSV file is the **Secret access key**.
7. In **SNS Endpoint for APNS**, enter the iOS ARN endpoint generated by Amazon SNS.

   For example, `arn:aws:sns:eu-west-1:123412341234:app/APNS/iOS_Push_Messaging`.
8. In **SNS Endpoint for GCM**, enter the Android ARN endpoint generated by Amazon SNS.

   For example, `arn:aws:sns:eu-west-1:123412341234:app/GCM/Android_Push_Messaging`.
9. Click **Create**, and then click **Save Changes**.

Self-managed PingAM server

1. Log in to the AM admin UI as an administrator.

   The realm overview for the Top Level Realm displays.

2. Select **Services**, and then click **+ Add a Service**.

3. In **Choose a service type**, select **Push Notification Service**.

4. In **SNS Access Key ID**, enter the **Access key ID** value of the access token you created previously.

   For example, `AKIAXOSPRCH15LEES`.

   > **Tip**
   >
   > If you downloaded the CSV file when you created the access key, the first value in the file is the **Access Key ID**.

5. In **SNS Access Key Secret**, enter the **Access key** value from the access token you created previously.

   For example, `9eF7EcWMZzChI51BBHkLeElXk8R3XHv7/n7QSiwoUFJ`.

   > **Tip**
   >
   > If you downloaded the CSV file when you created the access key, the second value in the CSV file is the **Secret access key**.

6. In **SNS Endpoint for APNS**, enter the iOS ARN endpoint generated by Amazon SNS.

   For example, `arn:aws:sns:eu-west-1:123412341234:app/APNS/iOS_Push_Messaging`.

7. In **SNS Endpoint for GCM**, enter the Android ARN endpoint generated by Amazon SNS.

   For example, `arn:aws:sns:eu-west-1:123412341234:app/GCM/Android_Push_Messaging`.

8. Click **Create**, and then click **Save Changes**.

Create a push registration and authentication journey

In this step you create an authentication journey that registers a device running an app built with the Ping (ForgeRock) Authenticator module to the user's profile if they have not done so already, then send a push notification to that device.

The journey then polls until it receives a response or timeout from the device. It verifies the returned data and completes the authentication journey if valid.

Choose whether you are creating the journey in PingOne Advanced Identity Cloud or a self-managed PingAM server, and follow the instructions to create the required authentication journey:

Advanced Identity Cloud

1. In the Advanced Identity Cloud admin UI
    1. Select the realm that will contain the authentication journey.
    2. Select **Journeys**, and click **+ New Journey**.
    3. Enter a name for your tree in **Name** page; for example, `MFAwithPush`
    4. In **Identity Object**, select the identity type that will be authenticating, for example `Alpha realm - Users`.
    5. Click **Save**.

       The authentication journey designer page is displayed with the default Start, Failure, and Success nodes.
2. Add the following nodes to the designer area:
    - Page node
    - Password Collector node
    - Username Collector node
    - Data Store Decision node
    - Push Sender node
    - Push Registration node or Combined MFA Registration node [1]
    - Push Wait node
    - Push Result Verifier node
3. Connect the nodes as shown:

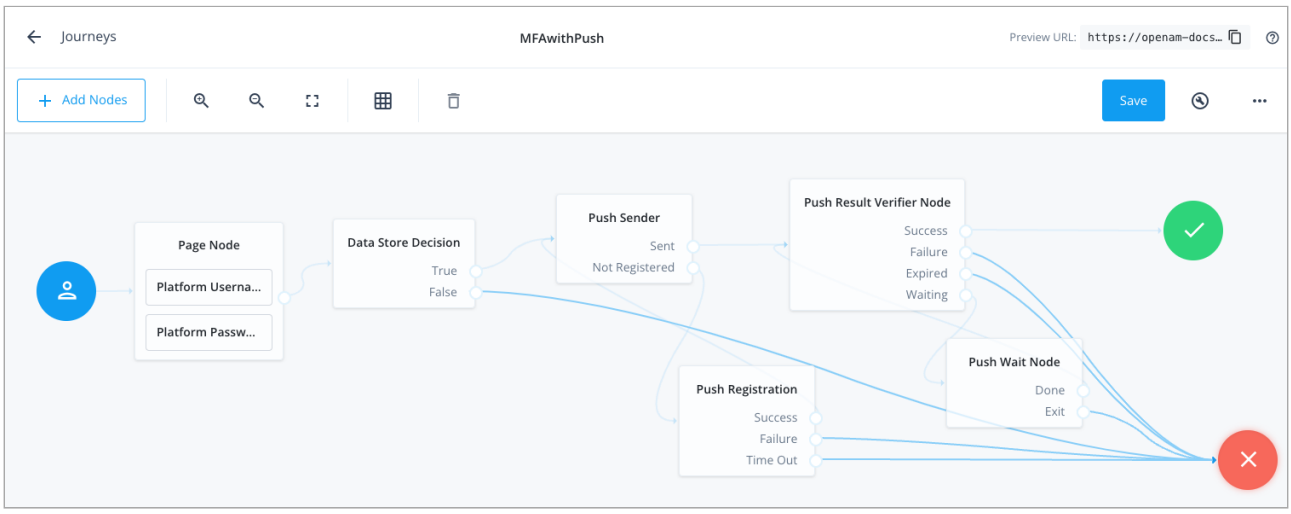


Figure 1. Connect the nodes to identify the user, send a push notification, and very the result.

4. In the Push Sender node, select the type of push notification the journey sends to the ForgeRock Authenticator:

Tap to Accept

Requires the user to tap to accept.

Display Challenge Code

Requires the user to select one of three numbers displayed on their device. This selected number must match the code displayed in the browser for the request to be verified.

Use Biometrics to Accept

Requires the user's biometric authentication to process the notification.

For information on how these options appear in the ForgeRock Authenticator, refer to Authenticate using a push notification.

5. Save your changes.

Self-managed PingAM server

1. In the AM admin UI:
    1. Select the realm that will contain the authentication tree.
    2. Select **Authentication > Trees**, and click **+ Create Tree**.
    3. Enter a name for your tree in the **New Tree** page; for example, `MFAwithPush`, and click **Create**.

       The authentication tree designer page is displayed with the default Start, Failure, and Success nodes.
2. Add the following nodes to the designer area:
    - Page node
    - Password Collector node
    - Username Collector node
    - Data Store Decision node
    - Push Sender node
    - Push Registration node or Combined MFA Registration node [1]
    - Push Wait node
    - Push Result Verifier node
3. Connect the nodes as shown:

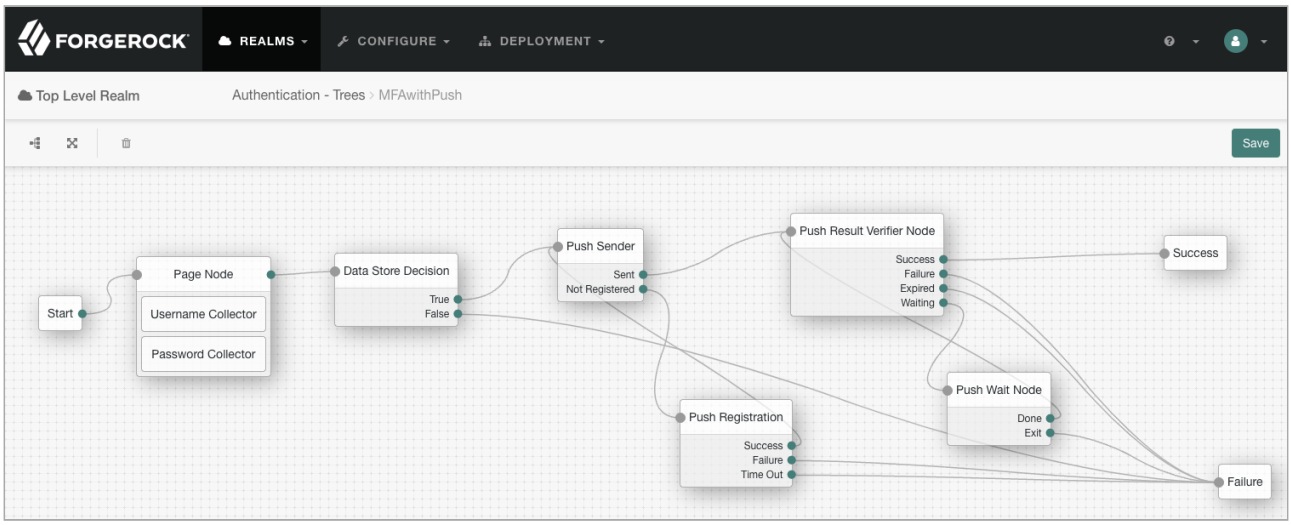


Figure 2. Connect the nodes to identify the user, send a push notification, and very the result.

4. In the Push Sender node, select the type of push notification the journey sends to the ForgeRock Authenticator:

Tap to Accept

Requires the user to tap to accept.

Display Challenge Code

Requires the user to select one of three numbers displayed on their device. This selected number must match the code displayed in the browser for the request to be verified.

Use Biometrics to Accept

Requires the user's biometric authentication to process the notification.

For information on how these options appear in the ForgeRock Authenticator, refer to Authenticate using a push notification.

5. Save your changes.

The tree you create is a simple example for the purposes of demonstrating a basic push authentication journey. In a production environment, you could include additional nodes, such as:

Get Authenticator App node

Provides links to download the ForgeRock Authenticator for Android and iOS.

MFA Registration Options node

Provides options for users to register a multi-factor authentication device, get the authenticator app, or skip the registration process.

Opt-out Multi-Factor Authentication node

Sets an attribute in the user's profile which lets them skip multi-factor authentication.

Recovery Code Display node

Lets a user view recovery codes to use in case they lose or damage the authenticator device they register.

Recovery Code Collector Decision node

Lets a user enter their recovery codes to authenticate in case they have lost or damaged their registered authenticator device.

Retry Limit Decision node

Lets a journey loop a specified number of times, for example, in case the user's device is experiencing connectivity issues, for example.

1. Use the combined MFA registration node if you intend to also add OATH one-time passwords as an MFA method.

Step 5: Configure the app for push notifications

In this step, you configure your application projects to use either Firebase Cloud Messaging or the Apple Push Notification service.

Enabling push notification support in an Android app

1. In Android Studio, open your authenticator app project and switch to the **Project** view.
2. Copy the `google-services.json` file that you downloaded from the Firebase console into the app-level root directory of your authenticator app:

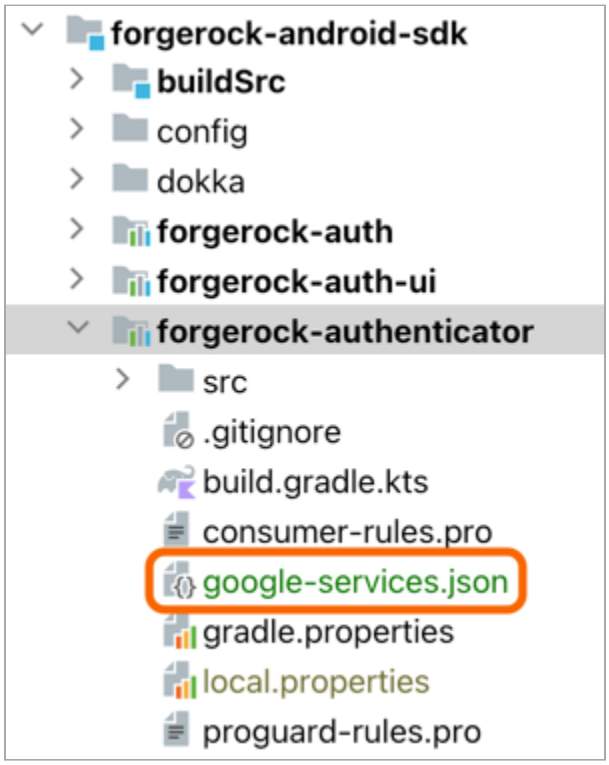


Figure 1. Adding the `google-services.json` to the app root in Android Studio.

3. To make the file available to the app, add the Google services Gradle plugin (`com.google.gms.google-services`) as a dependency to your project.
    1. In your ***root-level*** Gradle file:

Kotlin

build.gradle.kts

```
plugins {
  id("io.github.gradle-nexus.publish-plugin") version "1.1.0"
  id("org.sonatype.gradle.plugins.scan") version "2.4.0"
  id("org.jetbrains.dokka") version "1.9.10"
  id("com.android.application") version "8.3.2" apply false
  id("com.android.library") version "8.3.2" apply false
  id("org.jetbrains.kotlin.android") version "1.9.22" apply false
  // ...

  // Add the dependency for the Google services Gradle plugin
  id("com.google.gms.google-services") version "4.4.2" apply false
}
```

Groovy

build.gradle

```
plugins {
  id 'io.github.gradle-nexus.publish-plugin' version '1.1.0'
  id 'org.sonatype.gradle.plugins.scan' version '2.4.0'
  id 'org.jetbrains.dokka' version '1.9.10'
  id 'com.android.application' version '8.3.2' apply false
  id 'com.android.library' version '8.3.2' apply false
  id 'org.jetbrains.kotlin.android' version '1.9.22' apply false
  // ...

  // Add the dependency for the Google services Gradle plugin
  id 'com.google.gms.google-services' version '4.4.2' apply false
}
```

2. In your ***app-level*** Gradle file:

Kotlin

build.gradle.kts

```
plugins {
  id("com.android.library")
  id("com.adarshr.test-logger")
  id("maven-publish")
  id("signing")
  id("kotlin-android")
  // ...

  // Add the Google services Gradle plugin
  id("com.google.gms.google-services")
}
```

Groovy

build.gradle

```
plugins {
  id 'com.android.library'
  id 'com.adarshr.test-logger'
  id 'maven-publish'
  id 'signing'
  id 'kotlin-android'
  // ...

  // Add the Google services Gradle plugin
  id 'com.google.gms.google-services'
}
```

4. Switch to the Android view in Android Studio, and add the following code to the authenticator application manifest file.

   Insert the code inside the `<application>` tag.

AndroidManifest.xml

```
<service
    android:name=".controller.FcmService"
    android:exported="false">
    <intent-filter>
        <action android:name="com.google.firebase.MESSAGING_EVENT" />
    </intent-filter>
</service>
```

Enabling push notification support to an iOS app

1. In Xcode, open your authenticator app project.
2. In the left menu, select the project root folder.
3. In the project pane, under **Targets**, click your application, then click the **Signing & Capabilities** tab.
4. Confirm that the application has the **Push Notifications** capability:

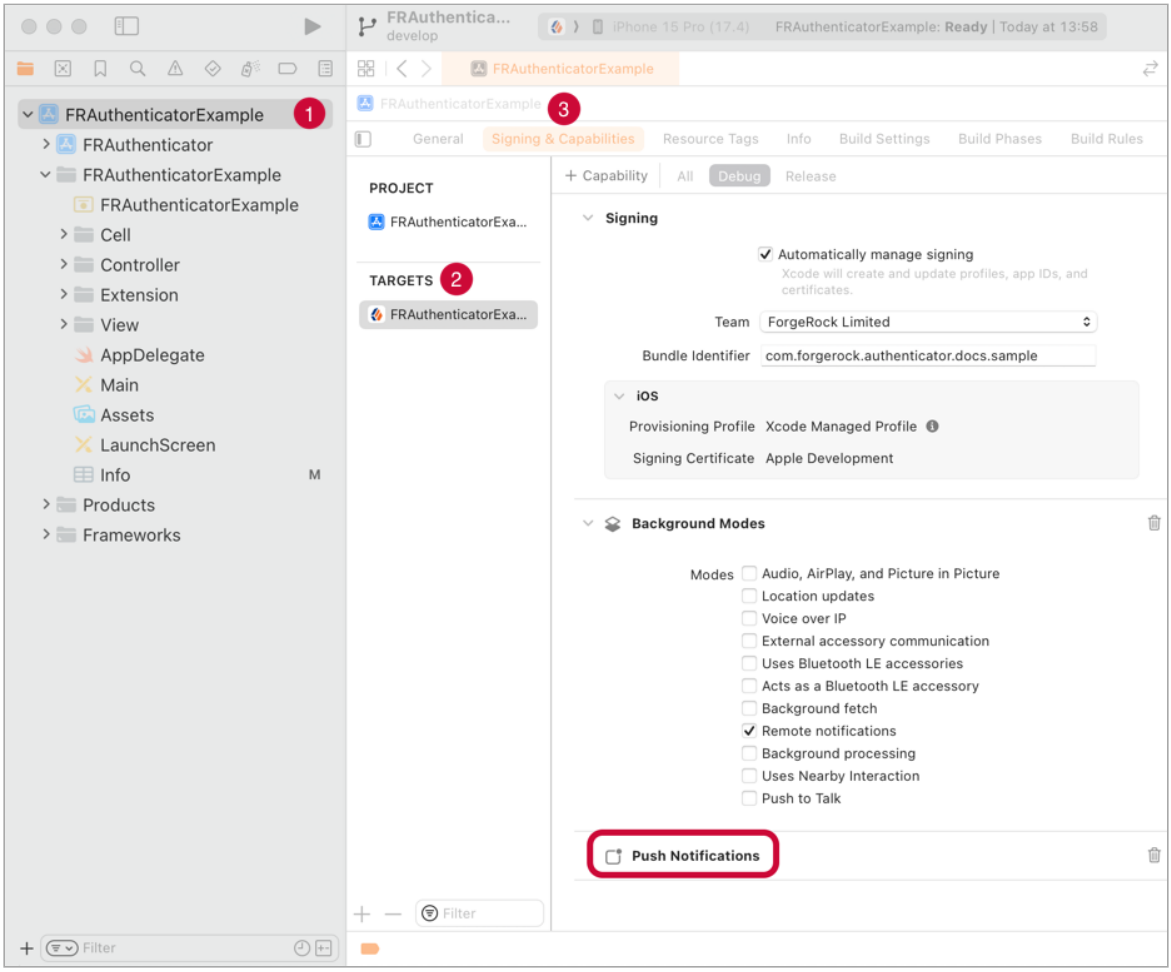


Figure 2. Checking for the Push Notifications capability in Xcode.

5. With an admin account, log in to the Apple Developer console.
6. Navigate to **Program resources** > **Certificates, IDs & Profiles** > **Identifiers**.
7. Click the name of the application to which you are adding push notification support.
8. On the **Capabilities** tab, ensure **Push Notifications** is selected.

Step 6. Configure the Ping (ForgeRock) Authenticator module for push notifications

In this step, you add the code to your application that obtains the unique device token required to ensure push notifications reach their intended audience.

You also add code that leverages the Ping (ForgeRock) Authenticator module to handle the push registration and authentication journey you created earlier.

Prerequisites

To complete the procedures on this page, you must set up your application to use the Ping (ForgeRock) Authenticator module:

- Configuring an Android app to use the Ping (ForgeRock) Authenticator module
- Configuring an iOS app to use the Ping (ForgeRock) Authenticator module

Register a device token to receive notifications

The Ping (ForgeRock) Authenticator module uses an Apple or Google service to receive push notifications sent from your server via Amazon SNS.

Each instance of your application requires a unique device registration token to receive these push notifications.

Use the `registerForRemoteNotifications()` method to register the token:

Android

```
// Retrieve the FCM token
FirebaseMessaging.getInstance().getToken()
        .addOnCompleteListener(task -> {
            if (!task.isSuccessful()) {
                Log.e(TAG, "getInstanceId failed", task.getException());
                return;
            }

            // Get new Instance ID token
            fcmToken = task.getResult();
            Log.v("FCM token:", fcmToken);

            // Register the token with the SDK to enable Push mechanisms
            try {
                fraClient.registerForRemoteNotifications(fcmToken);
            } catch (AuthenticatorException e) {
                Log.e(TAG,"Error registering FCM token: ", e);
            }
        });
```

iOS

```
func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions:
[UIApplication.LaunchOptionsKey: Any]?) -> Bool {
    // Register Push Notification for your app in AppDelegate.swift
    let center = UNUserNotificationCenter.current()
    center.requestAuthorization(options: [.alert, .sound, .badge]) { (granted, error) in }
    application.registerForRemoteNotifications()
    return true
}

func application(_ application: UIApplication, didRegisterForRemoteNotificationsWithDeviceToken deviceToken:
Data) {
    // Upon successful registration and receiving device token from APNs, register the token to the
Authenticator module
    FRAPushHandler.shared.application(application, didRegisterForRemoteNotificationsWithDeviceToken:
deviceToken)
}

func application(_ application: UIApplication, didFailToRegisterForRemoteNotificationsWithError error: Error)
{
    // Upon receiving an error from APNs, notify Authenticator module to properly update the status
    FRAPushHandler.shared.application(application, didFailToRegisterForRemoteNotificationsWithError: error)
}
```

Updating device tokens for existing accounts

Under certain circumstances the client operating system issues a new device token that your app should use for receiving push notifications.

The Ping SDKs for Android and iOS provide methods for updating the device token associated with accounts it has registered to receive Push notifications. These methods will also contact the PingAM server or PingOne Advanced Identity Cloud tenant that registered the device to update the device token it has stored in the user's profile.

Retrieving the existing device token

The Ping SDKs for Android and iOS automatically persist the device token, making it easier to update it when necessary.

Android

Use the `FRAClient.getPushDeviceToken()` method to retrieve the `PushDeviceToken` object, that contains the token and its issuance timestamp. If no token is available, it returns `null`.

iOS

Use the `FRAPushHandler.deviceToken` property that returns the current device token, or `nil` if it is not available.

Updating existing accounts with a new device token

If your app has existing accounts registered for push notifications and the operating system issues a new device token you must update both the accounts in the app and their respective servers with the updated token.

Android

Implement the `onNewToken` method within a custom `FirebaseMessagingService` subclass to receive the updated token.

The Ping SDK for Android provides the following methods to facilitate updating accounts and the server with the new device token:

`FRAClient.updateDeviceToken(String fcmToken, FRAListener<Void> listener)`

This method updates the device token for all accounts registered to receive push notifications.

Updating all accounts registered for push notifications

```
fraClient.updateDeviceToken(token, new FRAListener<Void>() {
    @Override
    public void onSuccess(Void result) {
      //DeviceToken is updated successfully
    }

    @Override
    public void onException(Exception e) {
      //Failed to update DeviceToken
    }
});
```

`FRAClient.updateDeviceTokenForMechanism(String deviceToken, PushMechanism pushMechanism, FRAListener<Void> listener)`

This method updates the device token for a single account registered to receive push notifications.

Updating a single account registered for push notifications

```
fraClient.updateDeviceTokenForMechanism(token, pushMechanism, new FRAListener<Void>() {
    @Override
    public void onSuccess(Void result) {
      //DeviceToken is updated successfully
    }

    @Override
    public void onException(Exception e) {
      //Failed to update DeviceToken
    }
});
```

iOS

If iOS issues a new device token it triggers the `didRegisterForRemoteNotificationsWithDeviceToken` application method.

The AppDelegate methods in the `FRAPushHandler` class stores issued device tokens locally, and automatically updates local registered accounts, and contacts the relevant server to update the user profiles with the new device token.

AppDelegate method automatically updates local and remote accounts

```
FRAPushHandler.application(_ application: UIApplication, didRegisterForRemoteNotificationsWithDeviceToken
deviceToken: Data)
```

The Ping SDK for iOS provides the following methods to manually update accounts and the server with the new device token:

`FRAPushHandler.updateDeviceToken(deviceToken: String, onSuccess: SuccessCallback, onFailure: ErrorCallback)`

This method updates the device token for all accounts registered to receive push notifications.

Updating all accounts registered for push notifications

```
FRAPushHandler.instance.updateDeviceToken(deviceToken: deviceTokenString, onSuccess: {
    // DeviceTokens updated successfully
}) { (error) in
    // Failed to update DeviceTokens with following error: \(error.localizedDescription)
}
```

`FRAPushHandler.updateDeviceToken(mechanism: PushMechanism, deviceToken: String, onSuccess: SuccessCallback, onFailure: ErrorCallback)`

This method updates the device token for a single account registered to receive push notifications.

Updating a single account registered for push notifications

```
FRAPushHandler.instance.updateDeviceToken(mechanism: pushMechanism, deviceToken: deviceTokenString,
onSuccess: {
    // DeviceToken updated successfully
}) { (error) in
    // Failed to update DeviceToken with following error: \(error.localizedDescription)
}
```

Handle registration of the app for push notifications

The first time you authenticate to your authentication tree, you are asked to register a device by scanning a QR code.

> **Important**
>
> Your application must implement a QR code scanning mechanism. The QR code contains the URI used for registering the device, although you could also offer a method for entering the URI manually.

After capturing the URI, register the authentication mechanism in your app:

Android

```
fraClient.createMechanismFromUri("qrcode_scan_result", new FRAListener<Mechanism>() {

    @Override
    public void onSuccess(Mechanism mechanism) {
        // called when device enrollment was successful.
    }

    @Override
    public void onFailure(final MechanismCreationException e) {
        // called when device enrollment has failed.
    }
});
```

iOS

```
guard let fraClient = FRAClient.shared else {
    print("FRAuthenticator SDK is not initialized")
    return
}

fraClient.createMechanismFromUri(uri: url, onSuccess: { (mechanism) in
    // Method call occurs when device enrollment is successful.
}, onError: { (error) in
    // Method call occurs when device enrollment fails.
})
```

Handle push notifications from the server

Your app that uses the Ping (ForgeRock) Authenticator module needs to respond to incoming push notifications, and ask the user to either accept or reject the authentication.

Android

Receive FCM Push notifications by using `FirebaseMessagingService#onMessageReceived`.

To handle `RemoteMessage`, use the `FRAClient.handleMessage()` method:

```
public void onMessageReceived(final RemoteMessage message) {
   PushNotification notification = fraClient.handleMessage(message);
}
```

iOS

Receive Apple push notifications by using the `application(_:didReceiveRemoteNotification:fetchCompletionHandler:)` method in `AppDelegate`.

To handle `RemoteNotification`, use the `FRAPushHandler.shared.application(:didReceiveRemoteNotification)` method.

The method returns a `PushNotification` object, which contains the `accept` and `deny` methods to handle the authentication request:

```
func application(_ application: UIApplication, didReceiveRemoteNotification userInfo: [AnyHashable : Any],
fetchCompletionHandler completionHandler: @escaping (UIBackgroundFetchResult) -> Void) {

    // Once you receive the remote notification, handle it with FRAPushHandler to get the PushNotification
object.
    // If RemoteNotification does not contain the expected payload structured from {am_name}, the
Authenticator module does not return the PushNotification object.
    if let notification = FRAPushHandler.shared.application(application, didReceiveRemoteNotification:
userInfo) {
        // With the PushNotification object, you can either accept or deny
        notification.accept(onSuccess: {

        }) { (error) in

        }
    }
}
```

Obtain values from the push notification payload

The `pushNotification` class provide the following methods for obtaining values from the payload received in the push notification:

Android method	iOS method	Description
`getCustomPayload()`	`customPayload`	Returns a JSON string containing the values specified in the **Custom Payload Attributes** property of the Push Sender node.
`getMessage()`	`message`	Returns the string specified in the **User Message** property of the Push Sender node, such as `Login attempt from Demo at ForgeRock`.
`getContextInfo()`	`contextInfo`	Returns a JSON string containing additional context information when the **Share Context info** property is enabled in the Push Sender node. Possible attributes in the JSON string are as follows: • `location` • `userAgent` • `remoteIp` Ensure you check these attributes for `null` values, as they depend on being able to be collected by the Device Profile Collector node. Example: `{ "location": { "latitude": 51.4517076, "longitude": -2.5950234 }, "userAgent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/100.0.4896.127 Safari/537.36", "remoteIp": "198.51.100.23" }`

Android method	iOS method	Description
`getPushType()`	`pushType`	Returns a PushType enum value that specifies the type of push notification to present to the user. This value is based on the configuration of the **Push Type** property in the Push Sender node. Possible values are: **`PushType.default`** Requires the user to tap to accept. **`PushType.challenge`** Requires the user to select one of three numbers displayed on their device. This selected number must match the code displayed in the browser for the request to be verified. **`PushType.biometric`** Requires the user's biometric authentication to process the notification.
`getNumbersChallenge()`	`numbersChallengeArray`	Returns an array of integers that matches those displayed on the login screen and populates the `numbersChallenge` attribute, if the **Push Type** property in the Push Sender node is set to **Display Challenge Code**.
`timeAdded`	`timeAdded`	Returns the timestamp of when the authentication server generated the push authentication payload.

Handle different push notification types

Use code similar to the following to determine which push type was requested in the payload:

Android

```
if (notification.getPushType() == PushType.CHALLENGE) {
    notification.accept(choice, listener);
} else if (notification.getPushType() == PushType.BIOMETRIC) {
    notification.accept(null, null, true, activity, listener);
} else {
    notification.accept(listener);
}
```

iOS

```
if notification.pushType == .challenge {
    notification.accept(
        challengeResponse: "34",
        onSuccess: successCallback,
        onError: errorCallback
    )
} else if notification.pushType == .biometric {
    notification.accept(
        title: "title",
        allowDeviceCredentials: true,
        onSuccess: successCallback,
        onError: errorCallback
    )
} else {
    notification.accept(
        onSuccess: successCallback,
        onError: errorCallback
    )
}
```

Handle the default push type

The `PushNotification` class or object provides an `accept` method for handling a `PushType.default` authentication request:

Android

```
pushNotification.accept(new FRAListener<Void>() {

    @Override
    public void onSuccess(Void result) {
        // called when accepting the push authentication request was successful.
    }

    @Override
    public void onFailure(final PushAuthenticationException e) {
        // called when denying the push authentication request, or it has failed.
    }
});
```

iOS

```
pushNotification.accept(onSuccess: {
    // called when accepting the push authentication request was successful.
}, onError: { (error) in
    // called when denying the push authentication request, or it has failed.
})
```

Handle the challenge push type

For `PushType.challenge` authentication requests, use the following `accept` method that receives the challenge as a parameter:

Android

```
public final void accept(
    @NonNull String challengeResponse,
    @NonNull FRAListener<Void> listener
) {}
```

iOS

```
public func accept(
    challengeResponse: String,
    onSuccess: @escaping SuccessCallback,
    onError: @escaping ErrorCallback
) {}
```

Handle the biometric push type

For `PushType.biometric` authentication requests, use the following `accept` method that processes the biometric authentication request:

Android

```
@RequiresApi(Build.VERSION_CODES.M)
public final void accept(
    String title,
    String subtitle,
    boolean allowDeviceCredentials,
    @NonNull AppCompatActivity activity,
    @NonNull FRAListener<Void> listener
) {}
```

iOS

```
public func accept(
    title: String,
    allowDeviceCredentials: Bool,
    onSuccess: @escaping SuccessCallback,
    onError: @escaping ErrorCallback
) {}
```

More information

Refer to the following links for information on some of the interfaces and objects used in this topic:

Android	iOS
PushNotification	PushNotification
PushMechanism	PushMechanism
PushType	PushType

Integrate MFA using OATH one-time passwords

Applies to:

- ✓ Ping SDK for Android
- ✓ Ping SDK for iOS
- ✕ Ping SDK for JavaScript

This topic explains how to integrate support for OATH one-time passwords into your projects that use the Ping (ForgeRock) Authenticator module.

Prerequisites

To integrate OATH one-time passwords into your application that uses the Ping (ForgeRock) Authenticator module, ensure you have completed the following tasks first:

1. Configure your server to request a one-time password during the authentication journey.

   Refer to Create an OATH registration and authentication journey.

2. Integrate the Ping (ForgeRock) Authenticator module into your app.

   Refer to Set up your Ping (ForgeRock) Authenticator module project.

3. Start the Ping (ForgeRock) Authenticator module in your app.

   Refer to Initialize the Ping (ForgeRock) Authenticator module.

Sample apps

You can find example source code for integrating one-time passwords in the sample authenticator application repositories on GitHub:

Android

ForgeRock Authenticator sample for Android.

iOS

ForgeRock Authenticator sample for iOS.

Step 1. Register your app

The first time you authenticate you are asked to register a device by scanning a QR code.

Your application must implement a QR code scanning mechanism. The QR code contains the URI used for registering the device, although you could also offer a method for entering the URI manually.

After obtaining the URI, register the authentication mechanism in your app:

Android

Register the OATH mechanism by implementing the `FRAClient.createMechanismFromUri()` method, and use `FRAListener` to receive the newly created mechanism:

```
fraClient.createMechanismFromUri("qrcode_scan_result", new FRAListener<Mechanism>() {

    @Override
    public void onSuccess(Mechanism mechanism) {
        // called when device enrollment was successful.
    }

    @Override
    public void onFailure(final MechanismCreationException e) {
        // called when device enrollment has failed.
    }
});
```

iOS

Implement `FRAClient.shared` in your `ViewController`, or `View` to receive the `Mechanism` object:

```
guard let fraClient = FRAClient.shared else {
    print("FRAuthenticator SDK is not initialized")
    return
}

fraClient.createMechanismFromUri(uri: url, onSuccess: { (mechanism) in
    // Method call occurs when device enrollment is successful.
}, onError: { (error) in
    // Method call occurs when device enrollment fails.
})
```

Step 2. Generate one-time passwords

With the OATH mechanisms now registered, your app can obtain the current, and next tokens, as an `OathTokenCode` object:

Android

```
OathTokenCode token = oath.getOathTokenCode();
String otp = token.getCurrentCode();
```

iOS

```
do {
    // Generate OathTokenCode
    let code = try mechanism.generateCode()
    // Update UI with generated code
    codeLabel?.text = code.code
} catch {
    // Handle errors for generating OATH code
}
```

More information

Refer to the following links for information on some of the interfaces and objects used in this topic:

Android	iOS
OathMechanism	OathMechanism
createMechanismFromUri	createMechanismFromUri

Integrate authenticator app policies

Applies to:

- ✓ Ping SDK for Android
- ✓ Ping SDK for iOS
- ✕ Ping SDK for JavaScript

You can build and distribute your own authenticator app to your users so that they can participate in multi-factor authentication journeys. To help ensure the security of your app—and therefore your system—you can enable *authenticator app policies*.

This topic explains how to integrate support for authenticator app policies into your projects that use the Ping (ForgeRock) Authenticator module.

Prerequisites

To integrate app policies into your application that uses the Ping (ForgeRock) Authenticator module, ensure you have completed the following tasks first:

1. Configure your server to apply app policies.

   Refer to Secure the Authenticator app using policies.

2. Integrate the Ping (ForgeRock) Authenticator module into your app.

   Refer to Set up your Ping (ForgeRock) Authenticator module project.

3. Start the Ping (ForgeRock) Authenticator module in your app.

   Refer to Initialize the Ping (ForgeRock) Authenticator module.

Step 1. Handle policies on the client

Policies are associated with an account registered in your authenticator app.

The `Account` class has the following attributes for handling app policies:

Attribute	Type	Visibility	Description
`lockingPolicy`	String	Public	The policy that caused the account to become locked. Only the first policy that was breached is listed.
`policies`	String	Public	A JSON string containing the policy names to apply, as configured in the combined MFA node.
`lock`	Boolean	Private [1]	Whether the account is currently locked or not.

[1] Use the public `isLocked` method to determine whether the account is currently locked or not

You can use the `lockAccount` and `unlockAccount` methods to manage registered accounts. To lock an account, you need to provide the policy that has been breached, as follows:

Android

```
// Reference to the authenticator object:
FRAClient fraClient = FRAClient.builder()
    .withContext(context)
    .start();

// Reference to the "Device tampering detection" policy:
FRAPolicy policy = new DeviceTamperingPolicy();

// Lock the account:
boolean result = fraClient.lockAccount(account, policy);
```

iOS

```
// Create the authenticator object:
FRAClient.start()

// Reference to the "Device tampering detection" policy:
let policy = DeviceTamperingPolicy()

// Lock the account:
let result = try FRAClient.lockAccount(account: account, policy: policy)
```

Step 2. Create custom policies

You can extend the new abstract class `FRAPolicy` to create new policies that you can attach to accounts.

In the class, implement the `evaluate` method which returns `true` when policy conditions are met or `false` if the conditions are breached. For example, if the tampered score exceeds the specified value, the evaluator would return `false`.

Android

```
static class AppIsUpToDatePolicy extends FRAPolicy {
    @Override
    public String getName() {
        return "appIsUpToDate";
    }

    @Override
    public boolean evaluate(Context context) {
        // Policy condition logic here
        return true; // policy conditions met
        // return false; // policy conditions breached - lock account
    }
}
```

iOS

```
class AppIsUpToDatePolicy: FRAPolicy {

    public var name: String = "appIsUpToDate"

    public var data: Any?

    public func evaluate() -> Bool {
        // Policy condition logic here
        return true // policy conditions met
        // return false // policy conditions breached - lock account
    }
}
```

To have the SDK evaluate your new policy, create a policy evaluator, as follows:

Android

Use `FRAPolicyEvaluator.FRAPolicyEvaluatorBuilder` and its methods `withPolicies` and `withPolicy` to pass policies to the evaluator:

```
FRAPolicyEvaluator policyEvaluator = new FRAPolicyEvaluator.FRAPolicyEvaluatorBuilder()
    .withPolicies(FRAPolicyEvaluator.DEFAULT_POLICIES)
    .withPolicy(new AppIsUpToDatePolicy())
    .build();
```

iOS

Use the `FRAPolicyEvaluator.registerPolicies()` method to pass policies to the evaluator.

Note that the default built-in policies are always evaluated.

To keep any existing registered policies on the account, specify the `shouldOverride: false` parameter:

```
let policyEvaluator = FRAPolicyEvaluator()
try policyEvaluator.registerPolicies(policies: [AppIsUpToDatePolicy()], shouldOverride: false)
```

> **Note**
>
> `FRAPolicyEvaluator.DEFAULT_POLICIES` includes both of the default built-in policies `BiometricAvailablePolicy` and `DeviceTamperingPolicy`.

Pass the policy evaluator when building your authenticator client:

Android

```
FRAClient.builder()
    .withContext(context.getApplicationContext())
    .withPolicyEvaluator(policyEvaluator)
    .start();
```

iOS

```
try FRAClient.setPolicyEvaluator(policyEvaluator: policyEvaluator)
FRAClient.start()
```

If the policy evaluator fails, the SDK automatically locks the account.

Locked accounts block certain methods, including `FRAClient.updateAccount`, `PushMechanism.accept` and `OATHMechanism.getNextOathToken`. Calling these methods on a locked account throws an `AccountLockException`.

API reference

Browse API reference documentation for the Ping (ForgeRock) Authenticator module:

Android

ForgeRock Authenticator API reference for Android.

iOS

ForgeRock Authenticator API reference for iOS.

Token Vault

Server support:

- ✓ PingOne
- ✓ PingOne Advanced Identity Cloud
- ✓ PingAM
- ✓ PingFederate

SDK support:

- ✗ Ping SDK for Android
- ✗ Ping SDK for iOS
- ✓ Ping SDK for JavaScript

Token Vault provides an additional layer of security for storing and using OAuth 2.0 and OpenID Connect 1.0 tokens in your JavaScript single-page applications (SPAs).

Caution

Intended audience
Token Vault is complex to set up.
It is designed for situations that demand the highest level of client-side security for OAuth 2.0 token management.
Token Vault might be suitable in these scenarios:

- Your industry has compliance or regulatory requirements, such as those for financial or government organizations
- You need to run untrusted, third-party code in your main application, such as from external advertisers, or other embedded applications

Due to the complexity of deployment, we recommend considering alternative solutions if your use case does not absolutely require the high level of client-side security that the Token Vault offers.
Alternative solutions include:

- Reduced Access Token lifetimes, without using refresh tokens
- Reduced idle timeouts on sessions with user-event-driven "keep-alive" requests
- Reduction or elimination of third-party code
- Usage of a server-side Backend For Frontend (BFF) approach for storing tokens

Implemented as a plugin for the Ping SDK for JavaScript, Token Vault provides a feature called *origin isolation*.

Web applications can only access data that gets stored within the matching origin - the unique combination of protocol (usually HTTPS), hostname, and port number.

By storing OAuth 2.0 and OpenID Connect 1.0 tokens under a *different origin* than your main application, you are *isolating* these tokens from malicious code.

Your main app uses the Ping SDK for JavaScript as usual to request tokens and access protected resources, however the Token Vault intercepts these requests and manages related tokens in the isolated origin.

As your main app does not get access to the contents of the isolated tokens, they are protected from being exposed or reused in attacks such as cross-site scripting.

Token Vault components

Token Vault consists of two main components:

Token Vault Proxy

The Token Vault Proxy is responsible for:

- Receiving and storing tokens in responses from your authorization server
- Redacting responses from your authorization server that contain token values, before passing the redacted response to your application
- Attaching stored tokens to requests that match the configured list of endpoints that require authorization

The Token Vault Proxy is embedded into your main application by using an inline frame, or *iframe*. An embedded iframe has a parent-child relationship with the main app, and the two can communicate with one another as long as they share the same parent domain, such as `example.com`.

To enable isolation, however, the *origins* must be different. For example, if your main app is served from `https://sdkapp.example.com`, the Token Vault Proxy could be served from `https://proxy.example.com`.

Token Vault Interceptor

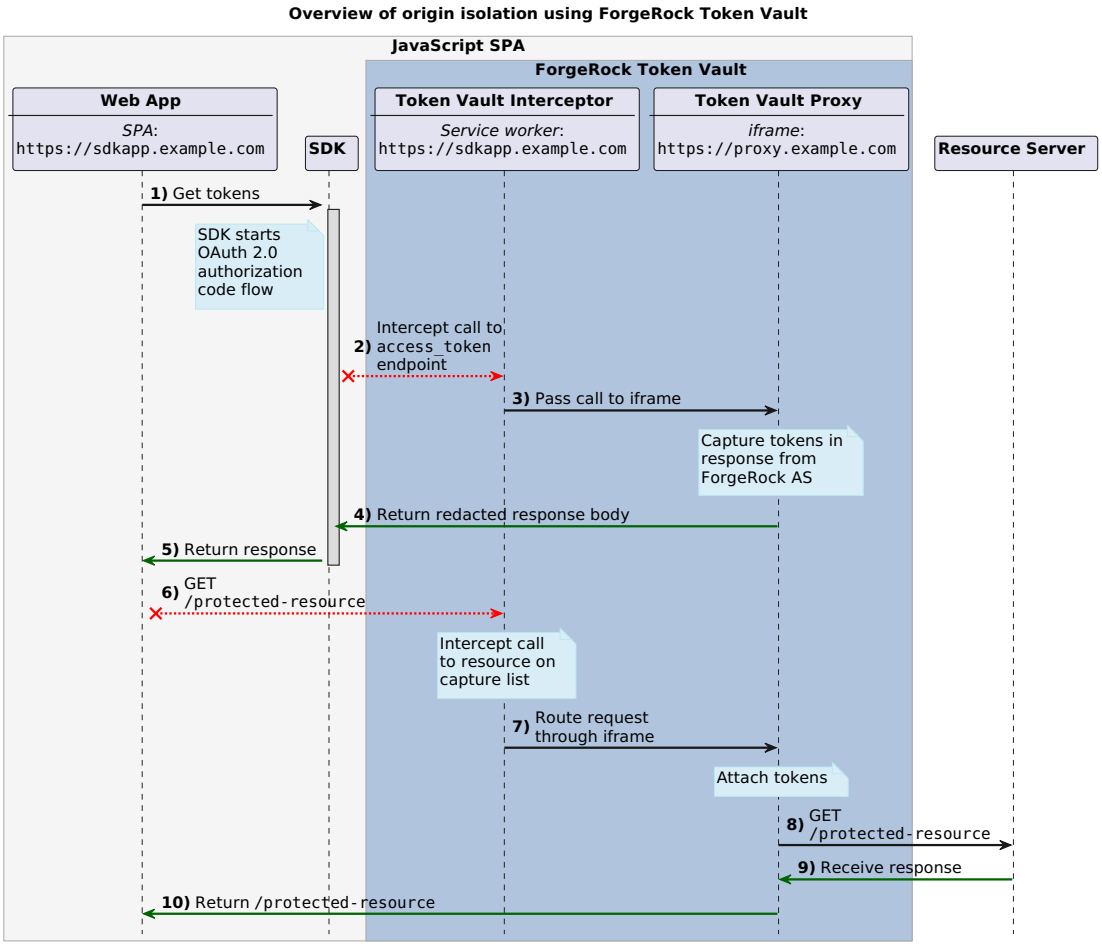
The Token Vault Interceptor is implemented as a service worker on your main app and intercepts requests to URLs that match a configured list.

These URLs are your protected resources and therefore require authorization to access. The Token Vault Interceptor captures these requests and passes them to the Token Vault Proxy iframe to add the relevant tokens.

The Token Vault Interceptor is also responsible for capturing OAuth 2.0 calls from the SDK to the authorization server and routing them through the Token Vault Proxy. The Token Vault Proxy forwards these requests to the authorization server and stores the returned tokens inside its own origin away from your main app.

Token Vault flow

The following diagram gives a simplified high-level overview of how the Token Vault isolates tokens away from your main application:



1. When your app uses the Ping SDK to request tokens from your authorization server it uses the Authorization Code Flow with PKCE. The last step in this flow is a call to the `/access_token` endpoint.
2. The Token Vault Interceptor captures this call to the `/access_token` endpoint.
3. The Token Vault Interceptor forwards the call to the Token Vault Proxy to handle instead of going directly to your authorization server.

   The Token Vault Proxy completes the authorization code flow and captures the tokens from the response. It stores the tokens securely in its origin, which is different from your main app but shares the parent domain. The Token Vault Proxy can attach these stored tokens to any future calls routed through it that require authorization.
4. The Token Vault Proxy then returns a redacted version of the response body. This ensures the main app never receives or stores the tokens.

Example of a redacted response body

```
{
  "accessToken": "REDACTED",
  "idToken": "eyJ0eXAiOiJKV1QiLCJra...7r8soMCk8A7QdQpg",
  "refreshToken": "REDACTED",
  "tokenExpiry": 1690712227226,
}
```

5. The SDK makes the redacted response available to your application.
6. Your app can now make requests for protected resources that require authorization.
7. If the protected resource matches a value on the configured list then the Token Vault Interceptor routes the request through the Token Vault Proxy.
8. The Token Vault Proxy attaches the tokens it has stored in its own origin to the request and sends the request to the resource server.
9. If the request has valid authorization bearer tokens attached, the resource server returns the protected content.
10. The protected resource is returned to the main app.

    If the tokens were invalid or expired in the previous step, the main app receives a 400 error instead. In this case your app must restart the authorization code flow.

Getting started

Get started with the Token Vault:

Configure the authorization server

Learn how to set up your PingOne Advanced Identity Cloud or PingAM instance with an OAuth 2.0 client and suitable CORS configuration.

Prepare for Token Vault

Discover how to integrate the Token Vault into your app and make some necessary changes to your environment.

Implement Token Vault code

Find out about the three main components to configure when implementing the Token Vault.

Access a protected resource via Token Vault

Implement origin isolation by accessing a resource through the Token Vault.

Configure your Authorization Server

You need to set up your PingOne Advanced Identity Cloud or PingAM instance with an OAuth 2.0 client and suitable CORS configuration.

Configure an OAuth 2.0 client

Follow the instructions below to create the public OAuth 2.0 client the Token Vault requires:

Public clients do not use a client secret to obtain tokens because they are unable to keep them hidden. The Ping SDKs commonly use this type of client to obtain tokens, as they cannot guarantee safekeeping of the client credentials in a browser or on a mobile device.

To register a *public* OAuth 2.0 client application for use with the SDKs in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.

2. In the left panel, click **Applications**.
3. Click **+ Custom Application**.
4. Select **OIDC - OpenId Connect** as the sign-in method, and then click **Next**.
5. Select **Native / SPA** as the application type, and then click **Next**.
6. In **Name**, enter a name for the application, such as `Public SDK Client`.
7. In **Owners**, select a user that is responsible for maintaining the application, and then click **Next**.

> **Tip**
>
> When trying out the SDKs, you could select the `demo` user you created previously.

8. In **Client ID**, enter `sdkPublicClient`, and then click **Create Application**.

   PingOne Advanced Identity Cloud creates the application and displays the details screen.
9. On the **Sign On** tab:
   1. In **Sign-In URLs**, enter the following values:

      > **Important**
      >
      > Also add any other domains where you host SDK applications.

   2. In **Grant Types**, enter the following values:

      `Authorization Code`

      `Refresh Token`

   3. In **Scopes**, enter the following values:

      `openid profile email address`
10. Click Show advanced settings, and on the **Authentication** tab:
    1. In **Token Endpoint Authentication Method**, select `none`.
    2. In **Client Type**, select `Public`.
    3. Enable the **Implied Consent** property.
11. Click **Save**.

The application is now configured to accept client connections from and issue OAuth 2.0 tokens to the example applications and tutorials covered by this documentation.

The provider specifies the supported OAuth 2.0 configuration options for a realm.

To ensure the PingAM OAuth 2.0 provider service is configured for use with the Ping SDKs, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. In the left panel, click **Services**.

3. In the list of services, click **OAuth2 Provider**.
4. On the **Core** tab, ensure **Issue Refresh Tokens** is enabled.
5. On the **Consent** tab, ensure **Allow Clients to Skip Consent** is enabled.
6. Click **Save Changes**.

In addition to the instructions above, perform the following steps:

1. Add the fully-qualified URL where you will host the Token Vault Proxy.

   For example, `https://proxy.example.com`.

   Add this value to either the **Redirection URIs** (self-managed PingAM) or the **Sign-in URLs** (PingOne Advanced Identity Cloud) property.

2. Enable refresh tokens in your authorization server:

   1. Add `refresh_token` to either the **Advanced** > **Response Types** (self-managed PingAM) or the **Access** > **Response Types** (PingOne Advanced Identity Cloud) property.
   2. Ensure **Refresh Token** is added to either the **Advanced** > **Grant Types** (self-managed PingAM) or the **Sign On** > **Grant Types** (PingOne Advanced Identity Cloud) property.

> **Note**
>
> Generally, we do not recommend the use of OAuth 2.0 refresh tokens with typical web-based applications, but using the Token Vault mitigates a number of the security concerns with using refresh tokens, so they can be enabled to allow refreshing the access tokens without user intervention.

Configure CORS

Follow the instructions below to configure CORS to allow the Token Vault to connect to your server:

Cross-origin resource sharing (CORS) lets user agents make cross-domain server requests. In PingOne Advanced Identity Cloud, you can configure CORS to allow browsers from trusted domains to access PingOne Advanced Identity Cloud protected resources. For example, you might want a custom web application running on your own domain to get an end-user's profile information using the PingOne Advanced Identity Cloud REST API.

The Ping SDK for JavaScript samples and tutorials use `https://localhost:8443` as the host domain, which you should add to your CORS configuration.

If you are using a different domain for hosting SDK applications, ensure you add them to the CORS configuration as accepted origin domains.

To update the CORS configuration in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. At the top right of the screen, click your name, and then select **Tenant settings**.
3. On the **Global Settings** tab, click **Cross-Origin Resource Sharing (CORS)**.

4. Perform one of the following actions:
   - If available, click **ForgeRockSDK**.
   - If you haven't added any CORS configurations to the tenant, click **+ Add a CORS Configuration**, select **Ping SDK**, and then click **Next**.
5. Add `https://localhost:8443` and any DNS aliases you use to host your Ping SDK for JavaScript applications to the **Accepted Origins** property.
6. Complete the remaining fields to suit your environment.

   This documentation assumes the following configuration, required for the tutorials and sample applications:

Property	Values
`Accepted Origins`	`https://localhost:8443`
`Accepted Methods`	`GET` `POST`

`Accepted Headers`	`accept-api-version`
	`x-requested-with`
	`content-type`
	`authorization`
	`if-match`
	`x-requested-platform`
	`iPlanetDirectoryPro` [1]
	`ch15fefc5407912` [2]

`Exposed Headers`	`authorization` `content-type`
`Enable Caching`	`True`
`Max Age`	`600`
`Allow Credentials`	`True`

> **Tip**
>
> Click **Show advanced settings** to be able to edit all available fields.

7. Click **Save CORS Configuration**.

Cross-origin resource sharing (CORS) lets user agents make cross-domain server requests. In PingAM, you can configure CORS to allow browsers from trusted domains to access PingAM protected resources. For example, you might want a custom web application running on your own domain to get an end-user's profile information using the PingAM REST API.

The Ping SDK for JavaScript samples and tutorials all use `https://localhost:8443` as the host domain, which you should add to your CORS configuration.

If you are using a different URL for hosting SDK applications, ensure you add them to the CORS configuration as accepted origin domains.

To enable CORS in PingAM, and create a CORS filter to allow requests from your configured domain names, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. Navigate to **Configure** > **Global Services** > **CORS Service** > **Configuration**, and set the **Enable the CORS filter** property to `true`.

   > **Important**
   >
   > If this property is not enabled, CORS headers are not added to responses from PingAM, and CORS is disabled entirely.

3. On the **Secondary Configurations** tab, click **Click Add a Secondary Configuration**.
4. In the **Name** field, enter `ForgeRockSDK`.
5. in the **Accepted Origins** field, enter any DNS aliases you use for your SDK apps.

   This documentation assumes the following configuration:

Property	Values
`Accepted Origins`	`https://localhost:8443`
`Accepted Methods`	`GET` `POST`

`Accepted Headers`	`accept-api-version`
	`x-requested-with`
	`content-type`
	`authorization`
	`if-match`
	`x-requested-platform`
	`iPlanetDirectoryPro` [1]
	`ch15fefc5407912` [2]

`Exposed Headers`	`authorization` `content-type`

6. Click **Create**.

   PingAM displays the configuration of your new CORS filter.

7. On the CORS filter configuration page:
   1. Ensure **Enable the CORS filter** is enabled.
   2. Set the **Max Age** property to `600`
   3. Ensure **Allow Credentials** is enabled.

8. Click **Save Changes**.

In addition to the instructions above, perform the following steps:

- Add the origins where you will host your main application and the Token Vault Proxy.

  For example, `https://sdkapp.example.com` and `https://proxy.example.com`, or when testing locally `http://localhost:5173` and `http://localhost:5174`.

  Add these values to the **Accepted origins** property.

1. Cookie name value in PingAM servers.
2. In PingOne Advanced Identity Cloud tenants, go to **Tenant Settings > Global Settings > Cookie** to find this dynamic cookie name value.

Prepare for Token Vault

To integrate the Token Vault into your app you need to make some changes to your environment.

Install the Token Vault

Install the Token Vault by using npm:

```
npm install @forgerock/token-vault
```

Configure your module bundler

You need to implement the Token Vault Interceptor component as a service worker in your main application.

To maximize cross-browser compatibility, we recommend that the Token Vault Interceptor is unified and *down-levelled* into a single output file.

To achieve this, you should create a separate bundler configuration dedicated to the Token Vault Interceptor. This configuration should produce a single file *without* using ES Module syntax. For example, Webpack version 5 and earlier can do this by default.

In addition, the Token Vault Proxy needs to have its own module bundler configuration as it must be separate from your main application.

Configure your web servers

You must serve the Token Vault Proxy from a different *origin* than your main application, but using the same *parent domain*.

To achieve this, we recommend that the Token Vault Proxy uses a dedicated web server. Your main application can then embed the Token Vault Proxy within an iframe to implement origin isolation for your tokens.

> **Tip**
>
> If you are testing locally, you can use different port numbers to ensure the origins differ.

In addition, you must configure the web server for the main application to avoid rewriting incoming URLs. The redirections back to your app from the authorization server contain query parameters that must be preserved and read by the SDK.

URL rewriting can cause timeout issues related to `/authorize` requests. For example, even though it succeeds and redirects back to your app with the `code` and `state` query parameters, the request to `/access_token` is not made. This can be caused by your web server rewriting the URL after receipt to only `/` and stripping the query parameters. This can cause the OAuth 2.0 flow to fail to resolve correctly leading to failures.

Structure your codebase

To help you integrate the Token Vault into your apps successfully, we recommend a codebase structure such as the following, which uses the *Vite* development environment:

```
root
├── .env (1)
├── package.json (2)
├── app/ (3)
|   ├── public/
|   |   └── <static files>
|   ├── src/
|   |   ├── main.js
|   |   └── <app files>
|   ├── interceptor/ (4)
|   |   └── interceptor.js
|   ├── index.html
|   ├── package.json
|   ├── vite.config.js (5)
|   └── vite.interceptor.config.js (6)
└── proxy/ (7)
  ├── src/
  |   └── proxy.js
  ├── index.html
  ├── package.json
  └── vite.config.js (8)
```

1 You could store shared configuration properties in an `.env` file
2 You could use npm workspaces
3 Main app folder
4 Separate Token Vault Interceptor code from your main app code
5 Vite configuration file to build your main app
6 Dedicated Vite configuration file to build the Token Vault Interceptor
7 Separate folder for the Token Vault Proxy
8 Separate Vite configuration to build the Token Vault Proxy

The structure of some of these files might resemble the following:

index.html

```
<!DOCTYPE html>
<html lang="en">
  <head></head>

  <body>
    <!-- Root div for mounting app -->
    <div id="root"></div>

    <!-- Root div for mounting Token Vault Proxy (iframe) -->
    <div id="token-vault"></div>

    <!-- Import main app -->
    <script type="module" src="/src/main.js"></script>
  </body>
</html>
```

main.js

```
import { Config, TokenManager } from '@forgerock/javascript-sdk';
import { client } from '@forgerock/token-vault';

const register = client({ /* global config */ });

register.interceptor();
register.proxy(document.getElementById('token-vault'));

const tokenVaultStore = register.store();

Config.set({ /* {sdk_name} config */});
```

interceptor.js

```
import { interceptor } from '@forgerock/token-vault';

interceptor({ /* config */ });
```

proxy.js

```
import { proxy } from '@forgerock/token-vault';

proxy({ /* config */ });
```

Next steps

When you have set up your project, you can proceed to Implement Token Vault code.

Implement Token Vault code

There are three main components to configure when implementing the Token Vault. It is important the configuration is consistent between these components. To achieve this, you could add your shared configuration to a `.env` file in the root of your project.

In the following examples, the configuration uses literal values to help understand the values required.

Implement main app code

This configuration should be within your app's `index` or `main` file.

Initialize the Token Vault client:

app/src/main.js

```
import { Config, TokenManager } from '@forgerock/javascript-sdk';
import { client } from '@forgerock/token-vault';

/**
 * This factory function takes in a config object and returns
 * the necessary methods to setup the iframe ("proxy"), the
 * service worker ("interceptor"), and the token store replacement
 * API ("store").
 */
const register = client({
    app: {
        origin: 'https://app.example.com',
    },
    interceptor: {
        file: '/interceptor.js',
    },
    proxy: {
        origin: 'https://proxy.example.com',
    },
});

/**
 * Sets up the service worker for intercepting fetch requests
 */
register.interceptor({
    /* optional interceptor worker config */
});

/**
 * Injects the iframe into the DOM to setup the proxy
 * Make sure to pass in the required, real DOM element as the zeroeth argument
 */
register.proxy(document.getElementById('token-vault'), {
    /* optional proxy config */
});

/**
 * Creates the store replacement for the SDK
 */
const tokenVaultStore = register.store({
    /* optional store config */
});
```

In the same file, configure the SDK:

app/src/main.js

```
Config.set({
  clientId: 'ForgeRockSDKClient',
  redirectUri: location.href,
  scope: 'openid profile email address',
  serverConfig: {
    baseUrl: 'https://openam-forgerock-sdks.forgeblocks.com/am',
    timeout: 5000,
  },
  realmPath: 'alpha',
  // Replace the default token store with Token Vault's store
  tokenStore: tokenVaultStore,
});
```

Implement Token Vault Interceptor code

This configuration should be within the Service Worker's entry file, which is separate from your main application code.

This is also the file to which your `client()` method config object property of `interceptor.file` method references.

Reference this file in your main application when calling the `client()` method, as the `interceptor.file` property.

Example configuration is as follows:

app/interceptor/interceptor.js

```
import { interceptor } from '@forgerock/token-vault';

interceptor({
  interceptor: {
    // Use either fully qualified URLs
    // Or end with a single asterisk as a wildcard
    urls: [/* Your protected endpoint URLs */],
  },
  forgerock: {
    // MUST match what you configured in your main app
    serverConfig: {
      baseUrl: 'https://openam-forgerock-sdks.forgeblocks.com/am',
      timeout: 5000,
    },
    realmPath: 'alpha',
  },
});
```

Tip

The `interceptor.urls` array accepts a `/*` ending to match any request from a particular root domain and path. This means you do not have to list each and every unique protected endpoint that your app might use.
For example, `https://backend.example.com/resources/protected/*`
This is not a full glob-pattern feature - just a single trailing wildcard.

Implement Token Vault Proxy code

This configuration should be within the Token Vault Proxy entry file.

Example configuration is as follows:

proxy/src/proxy.js

```
import { proxy } from '@forgerock/token-vault';

proxy({
  app: {
    // This MUST match the origin where your main app runs
    origin: 'https://app.example.com',
  },
  forgerock: {
    //  MUST match the config in your main app and interceptor
    clientId: 'ForgeRockSDKClient',
    redirectUri: location.href,
    scope: 'openid profile email address',
    serverConfig: {
      baseUrl: 'https://openam-forgerock-sdks.forgeblocks.com/am',
      timeout: 5000,
    },
    realmPath: 'alpha',
  },
});
```

Build the code

The Token Vault requires more complex building and bundling configuration than a regular JavaScript app that uses the SDK because it requires three different bundles:

1. The main application
2. The Token Vault Interceptor (Service Worker)
3. The Token Vault Proxy (iframe)

You can often use a default configuration for the main application and the Token Vault Proxy when using any of the popular bundlers, such as *Webpack* or *Vite*

The Token Vault Interceptor requires a specific build configuration to ensure maximum compatibility with various browsers.

Bundling the Interceptor

To provide the best cross-browser support, the Token Vault Interceptor requires a dedicated bundle configuration so that it results in a single-file output, down-leveled to at least ES2020 without *any* ES Module syntax.

We recommend using a separate `vite.interceptor.config.js` or `webpack.interceptor.config.js` for the Token Vault Interceptor, as well as a build separate command that consumes this separate configuration file.

> **Tip**
>
> If you are using Vite, you might achieve the best results with bundling into an Immediately Invoked Function Expression (IIFE).

If you are using Webpack, its defaults are good for bundling the Token Vault Interceptor.

Next steps

After you implement the code to enable the Token Vault, you can update your app to obtain tokens using origin isolation.

Access resources using Token Vault

After you complete the set up of the Token Vault successfully, you can use the Ping SDK for JavaScript or any HTTP or `fetch` library to request protected resources.

With the exception of refreshing tokens, and configuration of the token storage mechanism, using the Ping SDK for JavaScript with the Token Vault is almost entirely transparent.

The Token Vault manages token lifecycle automatically. If you enable refresh tokens in your OAuth 2.0 client, the Token Vault automatically refreshes access tokens.

Request tokens

Use the `TokenManager` class from the SDK as usual to request tokens and have them safely stored within the Token Vault Proxy:

```
import { TokenManager } from '@forgerock/javascript-sdk';

const tokens = TokenManager.getTokens();

console.log(tokens); // Refresh & Access Token values will be redacted
```

You can verify the tokens are stored under the origin of the Token Vault Proxy, not the origin of your main app, by using the developer tools in your browser.

The response your app and the SDK receive contains redacted values. This is expected behavior and increases security.

For example:

```
{
    "accessToken": "REDACTED",
    "idToken": "eyJ0eXAiOiJKV1QiLCJra...7r8soMCk8A7QdQpg",
    "refreshToken": "REDACTED",
    "tokenExpiry": 1690712227226,
}
```

Make requests

Use the native `fetch` API or any HTTP request library that emits a fetch event.

For example, you could use the `HttpClient` module provided in the Ping SDK for JavaScript.

The Token Vault Interceptor routes any of these requests that matches its configuration through the Token Vault Proxy so that the relevant tokens get attached before reaching your resource server.

Revoke tokens

To remove tokens and log the user out, use the `FRUser` class as usual:

```
import { FRUser } from '@forgerock/javascript-sdk';

FRUser.logout();
```

This destroys the user's session, revokes tokens on the server, and removes tokens from the Token Vault Proxy.

Use convenience methods

The `tokenVaultStore` object provides some convenience functions for use in your apps.

These methods are useful as your main app does not have any direct access to the tokens in the Token Vault.

The `has` method

Use the `has` method to determine whether the Token Vault has relevant tokens stored.

The method returns an object with a `hasTokens` property and a boolean value. It does not return the tokens.

```
const tokenVaultStore = register.store();

const { hasTokens } = tokenVaultStore.has();

console.log(hasTokens); // logs `true` or `false`
```

> **Note**
>
> This method reflects the presence of tokens but does not validate those tokens. They may have expired or were revoked by the server.
> To validate the tokens use the `UserManager.getCurrentUser` method. You can consider the tokens valid if the method returns user data.

The `refresh` method

Use the `refresh` method to manually request that the Token Vault refreshes its tokens.

The Token Vault attempts to refresh tokens automatically when required, but you can use this `refresh` method to force a refresh of the tokens, if needed.

The method returns an object with a `refreshTokens` property with a boolean value.

```
const tokenVaultStore = register.store();

const { refreshTokens } = tokenVaultStore.refresh();

console.log(refreshTokens); // logs `true` or `false`
```

Build advanced token security in a JavaScript single-page app

This tutorial covers the advanced development required for implementing the Token Vault with the Ping SDK for JavaScript.

First, why advanced token security?

In JavaScript Single Page Applications (or SPA), OAuth/OIDC Tokens (referred to from here on as *tokens*) are typically stored by using the browser's Web Storage API: `localStorage` or `sessionStorage`.

The security mechanism the browser uses to ensure data doesn't leak out to unintended actors is through the Same-Origin Policy. In short, only JavaScript running on the exact same origin, that is scheme, domain, and port, can access the stored data.

For example, if an SPA running on `https://auth.example.com/login` stores data, JavaScript running on the following ***will*** be able to access it:

- `https://auth.example.com/users`: origins match, regardless of path
- `https://auth.example.com?status=abc`: origins match, regardless of query parameters

The following ***will NOT*** be able to access the data:

- `http://auth.example.com`: uses a different scheme, `http` vs. `https`
- `https://auth.examples.com`: uses a different domain; notice the plurality
- `https://example.com`: does not include the sub-domain
- `https://auth.example.com:8000`: uses a different port

For the majority of web applications, this security model can be sufficient.

In most JavaScript applications, the code running on the app's origin can usually be trusted; hence, the browser's Web Storage API is sufficient as long as good security practices are in place.

However, in applications that are high-value targets, such as apps required to run untrusted, third-party code, or apps that have elevated scrutiny due to regulatory or compliance needs, the Same-Origin Policy may not be enough to protect the stored tokens.

Examples of situations where the Same-Origin Policy may not be sufficient include government agencies, financial organizations, or those that store sensitive data, such as medical records. The web applications of these entities may have enough inherent risk to offset the complexity of a more advanced token security implementation.

There are two solutions that can increase token security:

1. Backend for Frontend (BFF)
2. Origin Isolation

The Backend for Frontend (BFF) pattern

One solution that is quite common is to avoid storing high-value secrets within the browser in the first place. This can be done with a dedicated *Backend for Frontend*, or BFF. Yeah, it's a silly initialism.

This is increasingly becoming a common pattern for apps made with common meta-frameworks, such as Next, Nuxt, SvelteKit, and so on. The server component of these frameworks can store the tokens on the front end's behalf using in-memory stores or just writing the token into an *HTTPOnly* cookie, and requests that require authorization can be proxied through the accompanying server where the tokens are accessible. Therefore, no token needs to be stored on the front end, eliminating the risk.

You can read more about the arguments in favor of, and against, this design in the Hacker News discussion on the blog post titled "SPAs are Dead".

If, on the other hand, you are not in the position to develop, deploy and maintain a full backend, you have an alternative choice for securing your tokens: Origin Isolation.

Origin Isolation

Origin Isolation is a concept that introduces an alternative to BFF, and provides a more advanced mechanism for token security.

The concept is to store tokens in a *different* and dedicated origin related to the main application. To do this, two web servers are needed to respond to two different origins: one is your main app, and the second is an *iframed* app dedicated to managing tokens.

For more information, refer to the patent: Transparently using origin isolation to protect access tokens.

This particular design means that if your main application gets compromised, the malicious code running in your main app's origin still has no access to the tokens stored in the alternative origin.

You'll still need a web server of some kind to serve the files necessary to handle requests to this alternative origin, but being that only static files are served, the options are much simpler and lightweight.

This solution, which is implemented by using the Token Vault is the focus of this tutorial.

What is Token Vault?

Token Vault is a codified implementation of Origin Isolation. For more information, refer to Token Vault in the documentation.

It is a plugin available to customers that use the Ping SDK for JavaScript to enable OAuth 2.0 or OIDC token request and management in their apps. These apps can remain largely unmodified to take advantage of Token Vault.

> **Note**
>
> Even though your main app doesn't need much modification, additional build and server requirements are necessary, which introduces complexity and added maintenance to your system.
> We recommend that you only integrate Token Vault into an app if heightened security measures are a requirement of your system.

What you will learn

We will use an existing React JS, to-do sample application similar to what we built in another guide as a starting point for implementing Token Vault. This represents a realistic web application that has an existing implementation of the Ping SDK for JavaScript. Unlike the previous tutorial, we'll start with a fully working app and focus on adding Token Vault.

This tutorial focuses on OAuth 2.0 and OIDC authorization and token management. Authentication-related concerns, such as login and registration journeys are handled outside the app. We call this approach **OIDC login**.

> **Important**
>
> This is not a guide on how to build a React app
> How you architect or construct React apps is outside the scope of this guide.
> It's also worth noting that there are many React-based frameworks and generators for building web applications, such as Next.js, Remix, and Gatsby.
> What is *best* is highly contextual to the product and user requirements for any given project.

To demonstrate Token Vault integration, we will be using a simplified, non-production-ready, to-do app. This to-do app represents a Single Page Application (SPA) with nothing more than React Router added to the stack for routing and redirection.

Using this tutorial

This is a *hands-on* tutorial. We are providing the web app and resource server for you. You can find the repo on GitHub to follow along.

All you'll need is your own PingOne Advanced Identity Cloud or PingAM. If you don't have access to either, reach out to a representative today, and we'll be happy to get you started.

There are two ways to use this guide:

1. Follow along by building portions of the app yourself: continue by ensuring you can meet the requirements below.
2. Just curious about the code implementation details: skip to Implementing the Token Vault.

Requirements

Knowledge requirements

1. JavaScript and the npm ecosystem of modules
2. The command line interface, such as Shell or Bash
3. Core Git commands, including `clone` and `checkout`
4. React and basic React conventions
5. Context API: the concept for managing global state
6. Build systems/bundlers and development servers: We use basic Vite configuration and commands

Technical requirements

- Admin access to an instance of PingOne Advanced Identity Cloud or self-managed PingAM
- Node.js >= `16`

  Check your version with `node -v`
- npm >= `8`

  Check your version with `npm -v`

Authorization server setup

> **Tip**
>
> If you've already completed the previous tutorial for React JS or Angular, then you may already have most of this setup within your server. We'll call out the newly introduced data points to ensure you don't miss the configuration.

Step 1. Configure CORS (Cross-Origin Resource Sharing)

Due to the fact that pieces of our system will be running on different origins (scheme, domain, and port), we need to configure CORS in the server to allow our web app to make requests. Use the following values:

- **Allowed Origins**: `http://localhost:5173` `http://localhost:5175`
- **Allowed Methods**: `GET` `POST`
- **Allowed headers**: `accept-api-version` `authorization` `content-type` `x-requested-with`
- **Allow credentials**: enable

Rather than domain aliases and self-signed certificates, we will use `localhost` as that is a trusted domain by default. The main application will be on the `5173` port, and the proxy will be on the 5175 port. Because the proxy also make calls to the PingAM server, its origin must also be allowed.

> **Note**
>
> Service Workers are not compatible with self-signed certificates, regardless of tool or creation method. Certificates from a system-trusted, valid Certificate Authority (CA) are required or direct use of `localhost`. Self-signed certificates lead to a `fetch` error similar to the following:
>
> ```
> Failed to register a ServiceWorker for scope ('https://example.com:8000/') with script ('https://
> example.com:8000/sw.js'):
> An unknown error occurred when fetching the script
> ```

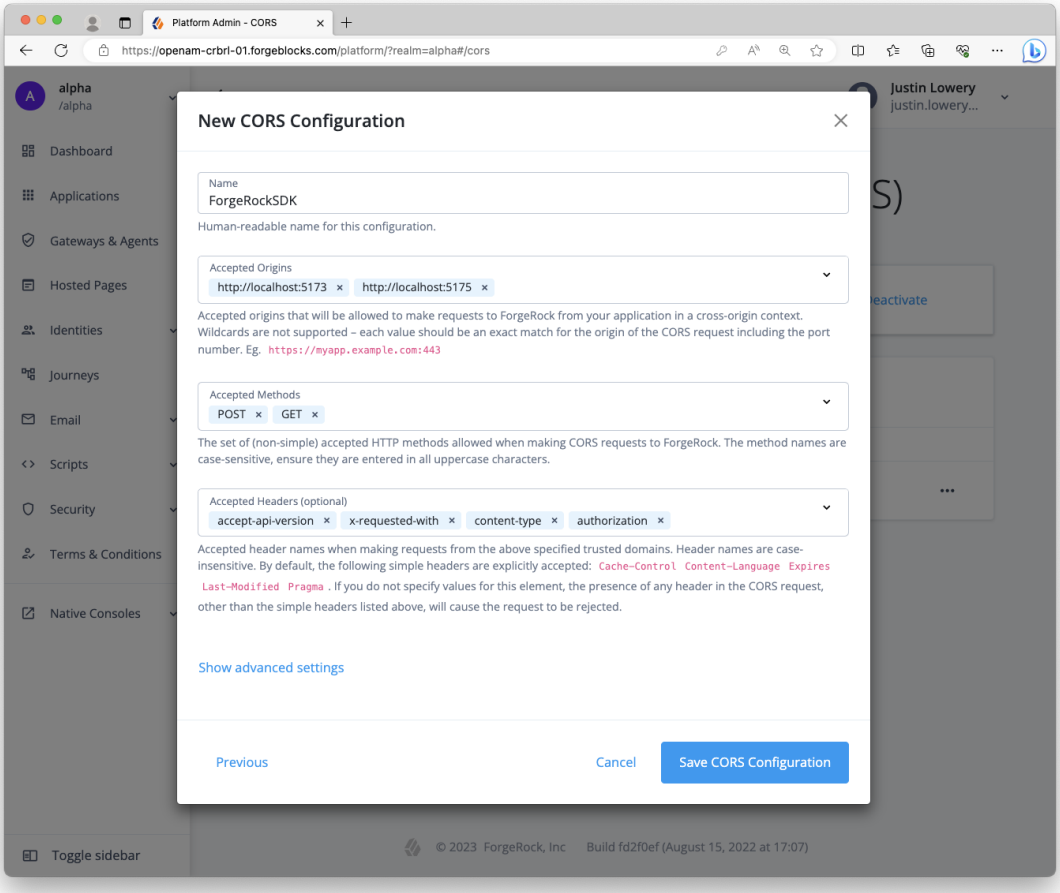


Figure 1. Example CORS configuration in PingOne Advanced Identity Cloud

For more information about CORS configuration, refer to the following:

- Configure CORS in PingAM
- Configure CORS in PingOne Advanced Identity Cloud.

Step 2. Create two OAuth 2.0 clients

Within the server, create two OAuth 2.0 clients: one for the React web app and one for the Node.js resource server.

Why two? It's conventional to have one OAuth 2.0 client per app in the system. For this case, a public OAuth 2.0 client for the React app provides our app with OAuth 2.0 or OIDC tokens. The Node.js server validates the user's Access Token shared via the React app using its own confidential OAuth 2.0 client.

Public OAuth 2.0 client settings

- **Client name/ID**: `CentralLoginOAuthClient`
- **Client type**: `Public`
- **Secret**: `<leave empty>`
- **Scopes**: `openid` `profile` `email`

- **Grant types**: `Authorization Code` `Refresh Token`
- **Implicit consent**: enabled
- **Redirection URLs/Sign-in URLs**: `http://localhost:5173/login`
- Response types: `code` `id_token` `refresh_token`
- **Token authentication endpoint method**: `none`

> **Note**
>
> The client name and redirection/sign-in URL values have changed from the previous tutorial, as well as the Refresh Token grant and response type values.

Confidential OAuth 2.0 client settings

- **Client name/ID**: `RestOAuthClient`
- **Client type**: `Confidential`
- **Secret**: `<alphanumeric string>` (treat it like a password)
- **Default scope**: `am-introspect-all-tokens`
- **Grant types**: `Authorization Code`
- **Token authentication endpoint method**: `client_secret_basic`

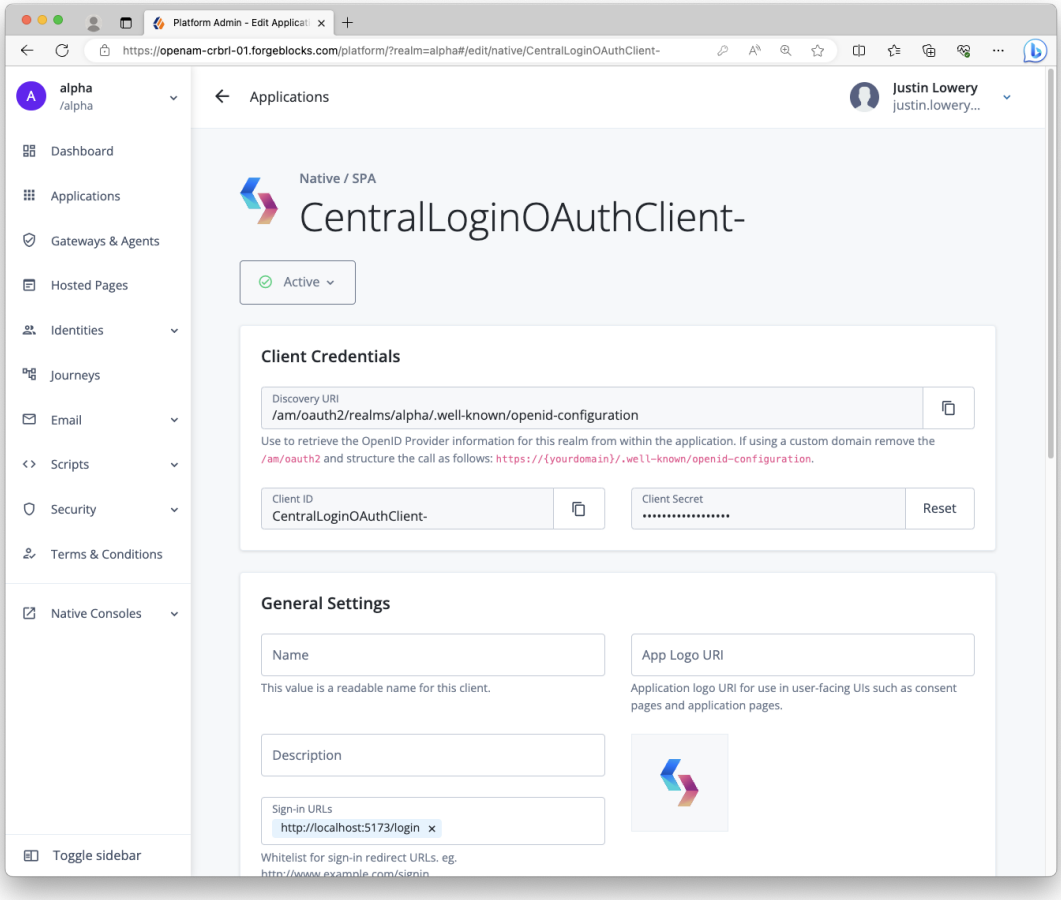


Figure 2. Example OAuth 2.0 client from PingOne Advanced Identity Cloud

Select the environment you are using for more information on configuring OAuth 2.0 clients:

Public clients do not use a client secret to obtain tokens because they are unable to keep them hidden. The Ping SDKs commonly use this type of client to obtain tokens, as they cannot guarantee safekeeping of the client credentials in a browser or on a mobile device.

To register a *public* OAuth 2.0 client application for use with the SDKs in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. In the left panel, click **Applications**.
3. Click **+ Custom Application**.
4. Select **OIDC - OpenId Connect** as the sign-in method, and then click **Next**.
5. Select **Native / SPA** as the application type, and then click **Next**.
6. In **Name**, enter a name for the application, such as `Public SDK Client`.
7. In **Owners**, select a user that is responsible for maintaining the application, and then click **Next**.

> **Tip**
>
> When trying out the SDKs, you could select the `demo` user you created previously.

8. In **Client ID**, enter `sdkPublicClient`, and then click **Create Application**.

   PingOne Advanced Identity Cloud creates the application and displays the details screen.

9. On the **Sign On** tab:

   1. In **Sign-In URLs**, enter the following values:

      > **Important**
      >
      > Also add any other domains where you host SDK applications.

   2. In **Grant Types**, enter the following values:

      `Authorization Code`

      `Refresh Token`

   3. In **Scopes**, enter the following values:

      `openid profile email address`

10. Click Show advanced settings, and on the **Authentication** tab:

    1. In **Token Endpoint Authentication Method**, select `none`.
    2. In **Client Type**, select `Public`.
    3. Enable the **Implied Consent** property.

11. Click **Save**.

The application is now configured to accept client connections from and issue OAuth 2.0 tokens to the example applications and tutorials covered by this documentation.

The provider specifies the supported OAuth 2.0 configuration options for a realm.

To ensure the PingAM OAuth 2.0 provider service is configured for use with the Ping SDKs, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. In the left panel, click **Services**.
3. In the list of services, click **OAuth2 Provider**.
4. On the **Core** tab, ensure **Issue Refresh Tokens** is enabled.
5. On the **Consent** tab, ensure **Allow Clients to Skip Consent** is enabled.
6. Click **Save Changes**.

Step 3. Create a test user

Create a test user, or *identity*, in your server within the realm you will be using.

Select the environment you are using for instructions on how to create a demo user:

The samples and tutorials in this documentation often require that you have an identity set up so that you can test authentication.

To create a demo user in PingOne Advanced Identity Cloud, follow these steps:

1. Log in to your PingOne Advanced Identity Cloud tenant.
2. In the left panel, click **Identities** > **Manage**.
3. Click **+ New Alpha realm - User**.
4. Enter the following details:
    - **Username** = `demo`
    - **First Name** = `Demo`
    - **Last Name** = `User`
    - **Email Address** = `demo.user@example.com`
    - **Password** = `Ch4ng3it!`
5. Click **Save**.

The samples and tutorials in this documentation often require that you have an identity set up so that you can test authentication.

To create a demo user in PingAM, follow these steps:

1. Log in to the PingAM admin UI as an administrator.
2. Navigate to **Identities**, and then click **+ Add Identity**.
3. Enter the following details:
    - **User ID** = `demo`
    - **Password** = `Ch4ng3it!`
    - **Email Address** = `demo.user@example.com`
4. Click **Create**.

Local project setup

Step 1. Installing the project

First, clone the forgerock-sample-web-react-ts repo to your local computer, `cd` (change directory) into the project folder, check out the branch for this guide, and install the needed dependencies:

```
git clone https://github.com/cerebrl/forgerock-sample-web-react-ts
cd forgerock-sample-web-react-ts
git checkout blog/token-vault-tutorial/start
npm install
```

> **Tip**
>
> There's also a branch that represents the completion of this guide. If you get stuck, you can check out the `blog/token-vault-tutorial/complete` branch from GitHub.

Step 2. Create an `.env` file

First, open the `.env.example` file in the root directory. Copy this file and rename it `.env`. Add your relevant values to this new file as it provides all the important configuration settings to your applications.

Here's a *hypothetical* example `.env` file:

Example of a populated `.env` file

```
# .env

# System settings
VITE_AM_URL=https://openam-forgerock-sdks.forgeblocks.com/am/ # Needs to be your {fr_server}
VITE_APP_URL=http://localhost:5173
VITE_API_URL=http://localhost:5174
VITE_PROXY_URL=http://localhost:5175 # This will be our Token Vault Proxy URL

# {am_name} settings
VITE_AM_JOURNEY_LOGIN=Login # Not used with Centralized Login
VITE_AM_JOURNEY_REGISTER=Registration # Not used with Centralized Login
VITE_AM_TIMEOUT=50000
VITE_AM_REALM_PATH=alpha
VITE_AM_WEB_OAUTH_CLIENT=CentralLoginOAuthClient
VITE_AM_WEB_OAUTH_SCOPE=openid email profile

# {am_name} settings for your API (todos) server
# (does not need VITE prefix)
AM_REST_OAUTH_CLIENT=RestOAuthClient
AM_REST_OAUTH_SECRET=6MWE8hs46k68g9s7fHOJd2LEfv # Don't use; this is just an example
```

We are using Vite for our client apps' build system and development server, so by using the `VITE_` prefix, Vite automatically includes these environment variables in our source code. Here are descriptions for some of the values:

- `VITE_AM_URL`: This should be your server and the URL almost always ends with `/am/`
- `VITE_APP_URL`, `VITE_API_URL`, and `VITE_PROXY_URL`: These will be the URLs you use for your locally running apps
- `VITE_AM_REALM_PATH`: The realm of your server. Likely, `alpha` if using PingOne Advanced Identity Cloud or `root` if using a self-managed PingAM server
- `VITE_REST_OAUTH_CLIENT` and `VITE_REST_OAUTH_SECRET`: This is the OAuth 2.0 client you configure in your server to support the REST API server

Build and run the project

Now that everything is set up, build and run the to-do app project. Open two terminal windows and use the following commands in the root directory of the SDK repo:

First terminal window

```
# Start the React app
npm run dev:react
```

This `dev:react` command uses Vite restarts on any change to a dependent file. This also applies to the `dev:proxy` we will build shortly.

Second terminal window

```
# Start the Rest API
npm run dev:api
```

This `dev:api` command runs a basic Node server with no "watchers." This should not be relevant as you won't have to modify any of its code. If a change is made within the `todo-api-server` workspace or an environment variable it relies on, a restart would be required.

> **Note**
>
> We use npm workspaces to manage our multiple sample apps but understanding how it works is not relevant to this tutorial.

Open the app in browser

In a *different* browser than the one you are using to administer the server, visit the following URL: `http://localhost:5173`. For example, you could use Edge for the app development and Chrome for the server administration.

A home page should be rendered explaining the purpose of the project. It should look like the example below, but it might be a dark variant if you have the dark theme/mode set in your OS:

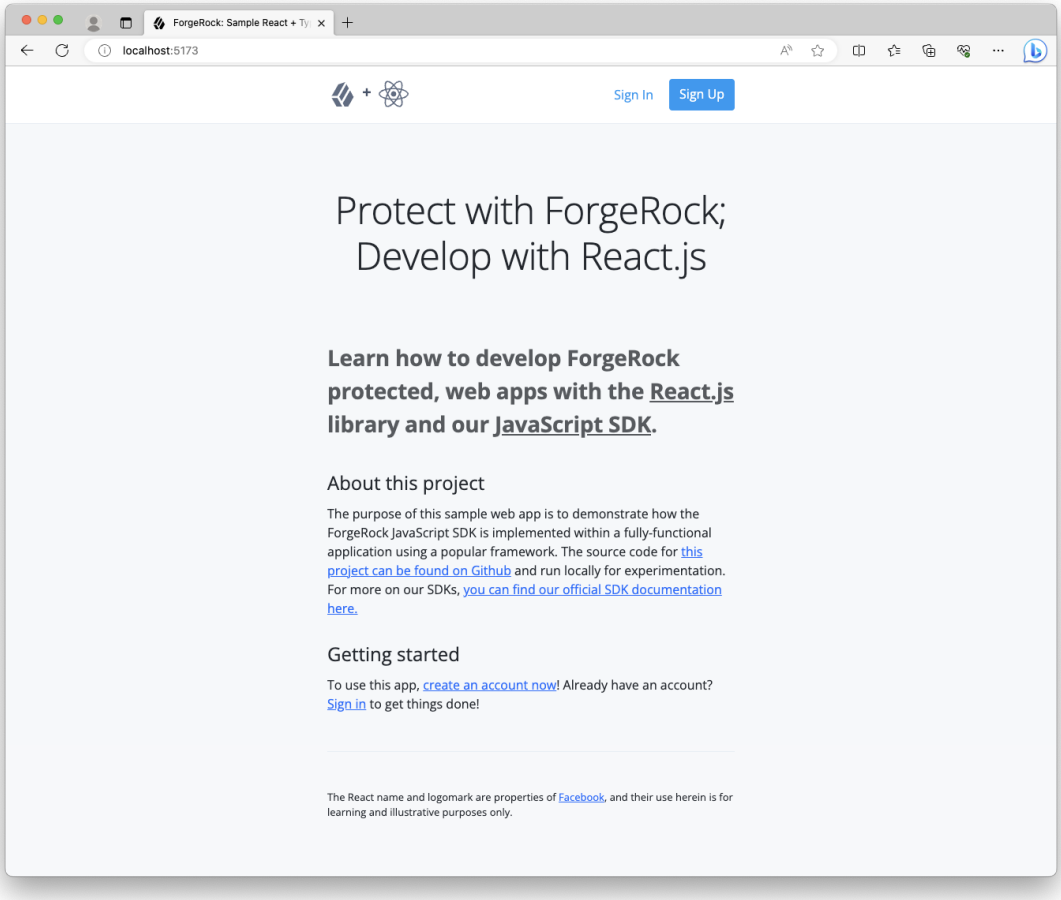


Figure 3. To-do app home page

If you encounter errors, here are a few tips:

- Visit `http://localhost:5174/healthcheck` in the same browser you use for the React app; ensure it responds with "OK"
- Check the terminal that has the `dev:react` command running for error output
- Ensure you are not logged into the server within the same browser as the sample app; log out if you are and use a different browser

Click the **Sign in** link in the header or in the Getting started section to sign in to the app with your test user. After successfully authenticating, you should see the app respond to the existence of the valid tokens.

Open your browser's developer tools to inspect its `localStorage`. You should see a single origin with an object containing tokens:

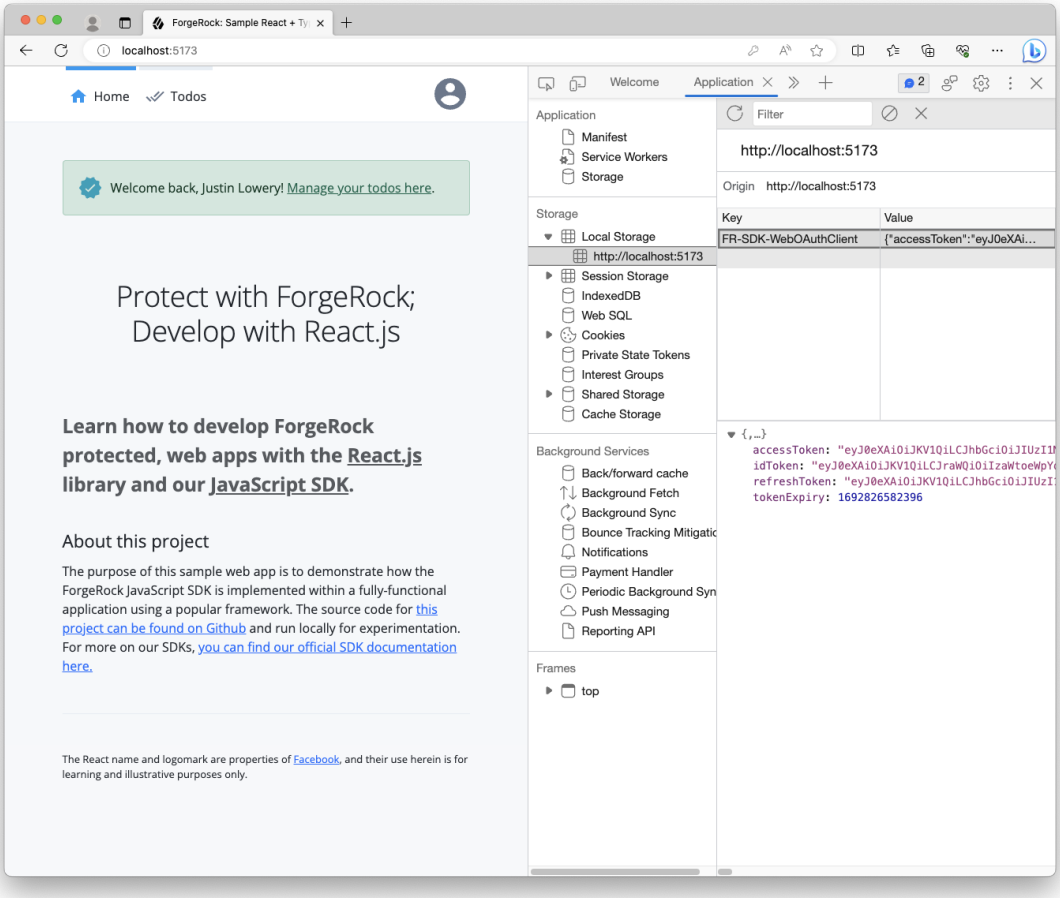


Figure 4. To-do app home page when logged in as a user

Install Token Vault module

Install the Token Vault npm module within the root of the app:

```
npm install @forgerock/token-vault
```

This npm module is used throughout multiple applications in our project, so installing it at the root rather than at the app or workspace level is a benefit.

Implement the Token Vault Proxy

Step 1. Scaffold the Proxy

Next, we'll need to create a third application, the Token Vault Proxy.

Follow this structure when creating the new directory and its files:

```
  root
 ├── todo-api-server/
 ├── todo-react-app/
+└─┬ token-vault-proxy/
+  ├─┬ src/
+  │ └── index.ts
+  ├── index.html
+  ├── package.json
+  └── vite.config.ts
```

Step 2. Add the npm workspace

To ease some of the dependency management and script running, add this new "workspace" to our root `package.json`, and then add a new script to our `scripts`:

Root `package.json` file

```
@@ package.json @@

@@ collapsed @@

  "scripts": {
    "clean": "git clean -fdX -e \"!.env\"",
    "build:api": "npm run build --workspace todo-api-server",
    "build:react": "npm run build --workspace todo-react-app",
+   "build:proxy": "npm run build --workspace token-vault-proxy",
    "dev:api": "npm run dev --workspace todo-api-server",
    "dev:react": "npm run dev --workspace todo-react-app",
+   "dev:proxy": "npm run dev --workspace token-vault-proxy",
    "dev:server": "npm run dev --workspace todo-api-server",
    "lint": "eslint --ext ts,tsx --report-unused-disable-directives --max-warnings 0",
    "serve:api": "npm run serve --workspace todo-api-server",
-   "serve:react": "npm run serve --workspace todo-react-app"
+   "serve:react": "npm run serve --workspace todo-react-app",
+   "serve:proxy": "npm run serve --workspace token-vault-proxy"
  },

@@ collapsed @@

  "workspaces": [
    "todo-api-server"
-   "todo-react-app"
+   "todo-react-app",
+   "token-vault-proxy",
  ]
```

Step 3. Setup the supporting files

Create a new directory at the root named `token-vault-proxy`, then create a `package.json` file:

Token Vault Proxy `package.json` file

```
@@ token-vault-proxy/package.json @@

+ {
+   "name": "token-vault-proxy",
+   "private": true,
+   "version": "1.0.0",
+   "description": "The proxy for Token Vault",
+   "main": "index.js",
+   "scripts": {
+     "dev": "vite",
+     "build": "vite build",
+     "serve": "vite preview --port 5175"
+   },
+   "dependencies": {},
+   "devDependencies": {},
+   "license": "MIT"
+ }
```

Now, create the Vite config file:

Token Vault Proxy Vite configuration file

```
@@ token-vault-proxy/vite.config.ts @@

+ import { defineConfig, loadEnv } from 'vite'; (1)
+
+ // https://vitejs.dev/config/
+ export default defineConfig(({ mode }) => { (2)
+   const env = loadEnv(mode, `${process.cwd()}/../`); (3)
+   const port = Number(new URL(env.VITE_APP_URL).port); (4)
+
+   return { (5)
+     envDir: '../', // Points to the `.env` created in the root dir
+     root: process.cwd(),
+     server: {
+       port,
+       strictPort: true,
+     },
+   };
+ });
```

What does the above do? Good question! Let's review it:

1 We import helper functions from `vite`
2 Using `defineConfig`, we pass in a function, as opposed to an object, because we want to calculate values at runtime
3 The parameter `mode` helps inform Vite how the config is being executed, useful when you need to calculate env variables
4 Then, extract the `port` out of our app's configured origin, which *should* be `5175`
5 Finally, use this data to construct the config object and return it

Now, create the `index.html` file. This file can be overly simple as all you need is the inclusion of the JavaScript file that will be our proxy:

Token Vault Proxy `index.html` file

```
@@ token-vault-proxy/index.html @@

+ <!DOCTYPE html>
+ <html>
+   <p>Proxy is OK</p>
+   <script type="module" src="src/index.ts"></script>
+ </html>
```

If you're not familiar with how Vite works, seeing the `.ts` extension may look a bit odd in an HTML file but don't worry. Vite uses this to find entry files, and it rewrites the actual `.js` reference for us.

Step 4. Create and configure the Proxy

Let's create and configure the Token Vault Proxy according to our needs. First, create the `src` directory and the `index.ts` file within it.

Token Vault Proxy `index.ts` file

```
@@ src/index.ts @@

+ import { proxy } from '@forgerock/token-vault';
+
+ // Initialize the token vault proxy
+ proxy({
+   app: {
+     origin: new URL(import.meta.env.VITE_APP_URL).origin, (1)
+   },
+   forgerock: { (2)
+     clientId: import.meta.env.VITE_AM_WEB_OAUTH_CLIENT,
+     scope: import.meta.env.VITE_AM_WEB_OAUTH_SCOPE,
+     serverConfig: {
+       baseUrl: import.meta.env.VITE_AM_URL,
+     },
+     realmPath: import.meta.env.VITE_AM_REALM_PATH,
+   },
+   proxy: { (3)
+     urls: [`${import.meta.env.VITE_API_URL}/*`],
+   }
+ });
```

The configuration above represents the minimum needed to create the Token Vault Proxy:

1 We need to declare the app's origin, as that's the only source to which the Proxy will respond.
2 We have the configuration in order for the Proxy to call out to the server effectively for token lifecycle management.
3

Finally, there's the Proxy's `urls` array that acts as an allow-list to ensure only valid URLs are proxied with the appropriate tokens.

Step 5. Build and verify the Proxy

With everything set up, build the proxy app and verify it's being served correctly.

```
npm run dev:proxy
```

Once the script finishes its initial build and runs the server, you can now check the app and ensure it's running. Go to `http://localhost:5175` in your browser. You should see "Proxy is OK" printed on the screen, and there should be no errors in the Console or Network tab of your browser's dev tools.

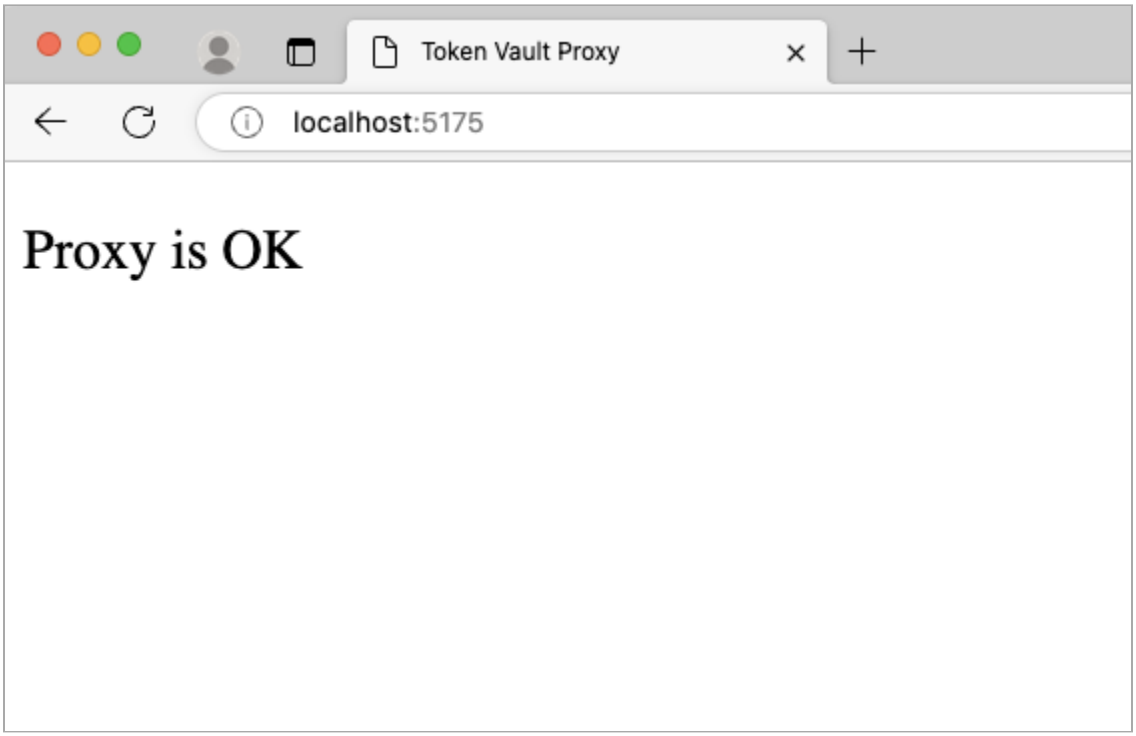


Figure 5. Proxy viewed directly in browser

Implement the Token Vault Interceptor

Step 1. Create the Token Vault Interceptor build config

Since the Token Vault Interceptor is a Service Worker, it needs to be bundled separately from your main application code. To do this, write a *new* Vite config file within the `todo-react-app` directory/workspace named `vite.interceptor.config.ts`.

We do not recommend trying to use the same configuration file for both your app and Interceptor.

```
 root
├── ...
├─┬ todo-react-app/
│ ├── ...
│ ├── vite.config.ts
+ │ └── vite.interceptor.config.ts
```

Now that you have the new Vite config for the Interceptor, import the `defineConfig` method and pass it the appropriate configuration.

Token Vault Interceptor `vite.interceptor.config.ts` file

```
@@ todo-react-app/vite.interceptor.config.ts @@

+ import { defineConfig } from 'vite';
+
+ // https://vitejs.dev/config/
+ export default defineConfig({
+   build: {
+     emptyOutDir: false,
+     rollupOptions: {
+       input: 'src/interceptor/index.ts',
+       output: {
+         dir: 'public', // Treating this like a static asset is important
+         entryFileNames: 'interceptor.js',
+         format: 'iife', // Very important for better browser support
+       },
+     },
+   },
+   envDir: '../', // Points to the `.env` created in the root dir
+ });
```

> **Note**
>
> Rather than passing a function into `defineConfig`, we are passing a plain config object.
> This is because we don't need any variables at runtime, like env values.

In the above, we provide the Token Vault Interceptor source file as the input, and then explicitly tell Vite to bundle it as an IIFE (Immediately Invoked Function Expression) and save the output to this app's `public` directory. This means the Token Vault Interceptor will be available as a static asset at the root of our web server.

It is important to know that bundling it as an IIFE and configuring the output to the public directory is intentional and important. Bundling as an IIFE removes any module system from the file, which is vital to supporting all major browsers within the Service Worker context. Outputting it to the public directory like a static asset is also important. It allows the `scope` of the Service Worker to also be available at the root.

For more information, refer to Service Worker scopes on MDN.

Step 2. Create the new Token Vault Interceptor file

Let's create the new Token Vault Interceptor source file that is expected as the entry file to our new Vite config.

```
  root
 ├─┬ todo-react-app
 │ ├─┬ src/
+│ │ ├─┬ interceptor/
+│ │ │ └── index.ts
 │ │ └── ...
 │ ├── ...
 │ └── vite.interceptor.config.ts
```

Step 3. Import and initialize the `interceptor` module

Configure your Token Vault Interceptor with the following variables from your `.env` file.

Token Vault Interceptor `index.ts` source file

```
@@ todo-react-app/src/interceptor/index.ts @@

+ import { interceptor } from "@forgerock/token-vault";
+
+ interceptor({
+   interceptor: {
+     urls: [`${import.meta.env.VITE_API_URL}/*`], (1)
+   },
+   forgerock: { (2)
+     serverConfig: {
+       baseUrl: import.meta.env.VITE_AM_URL,
+       timeout: import.meta.env.VITE_AM_TIMEOUT,
+     },
+     realmPath: import.meta.env.VITE_AM_REALM_PATH,
+   },
+ });
```

The above only covers the minimal configuration needed, but it's enough to get a basic Interceptor started.

1. The `urls` array represents all the URLs you'd like intercepted and proxied through the Token Vault Proxy in order for the Access Token to be added to the outbound request. This should only be for requesting your "protected resources." The wildcard (`*`) can be used if you want a catch-all for endpoints of a certain origin or root path. Full glob patterns are *not* supported, so a URL value can only end with `*`.
2. The configuration here must match the configuration in the main app. This is easily enforced by using the `.env` file

Step 4. Build the Interceptor

Now that we have the dedicated Vite config and the Token Vault Interceptor entry file created, add a dedicated build command to the `package.json` within the `todo-react-app` workspace.

React app `package.json` file

```
@@ todo-react-app/package.json @@

@@ collapsed @@

    "scripts": {
-     "dev": "vite",
+     "dev": "npm run build:interceptor && vite",
-     "build": "vite build",
+     "build": "npm run build:interceptor && vite build",
+     "build:interceptor": "vite build -c ./vite.interceptor.config.ts",
      "serve": "vite preview --port 5173"
    },

@@ collapsed @@
```

It's worth noting that the Token Vault Interceptor will only be rebuilt at the start of the command and not rebuilt after any change thereafter as there's no `watch` command used here for the Token Vault Interceptor itself. Once this portion of code is correctly set up, it should rarely change, so this should be fine.

Your main app will still be rebuilt and "hot-reloading" will take place.

Enter `npm run build:interceptor -w todo-react-app` to run the new command you just wrote above in the `todo-react-app` workspace. You can see the resulting `interceptor.js` built and placed into your `public` directory.

```
  root
 ├─┬ todo-react-app
 │ ├─┬ public/
 │ │ ├── ...
+│ │ ├── interceptor.js
```

Step 5. Ensure `interceptor.js` is accessible

Since we haven't implemented the Token Vault Interceptor yet in the main app, we can't really test it; however, we can at least make sure the file is accessible in the browser as we expect. To do this, run the following command:

```
npm run dev:react
```

After the command starts the server, using your browser, visit `http://localhost:5173/interceptor.js`.

You should plainly see the fully built JavaScript file. Ensure it does not have any `import` statements and looks complete. It should contain more code than just the original source file you wrote above.

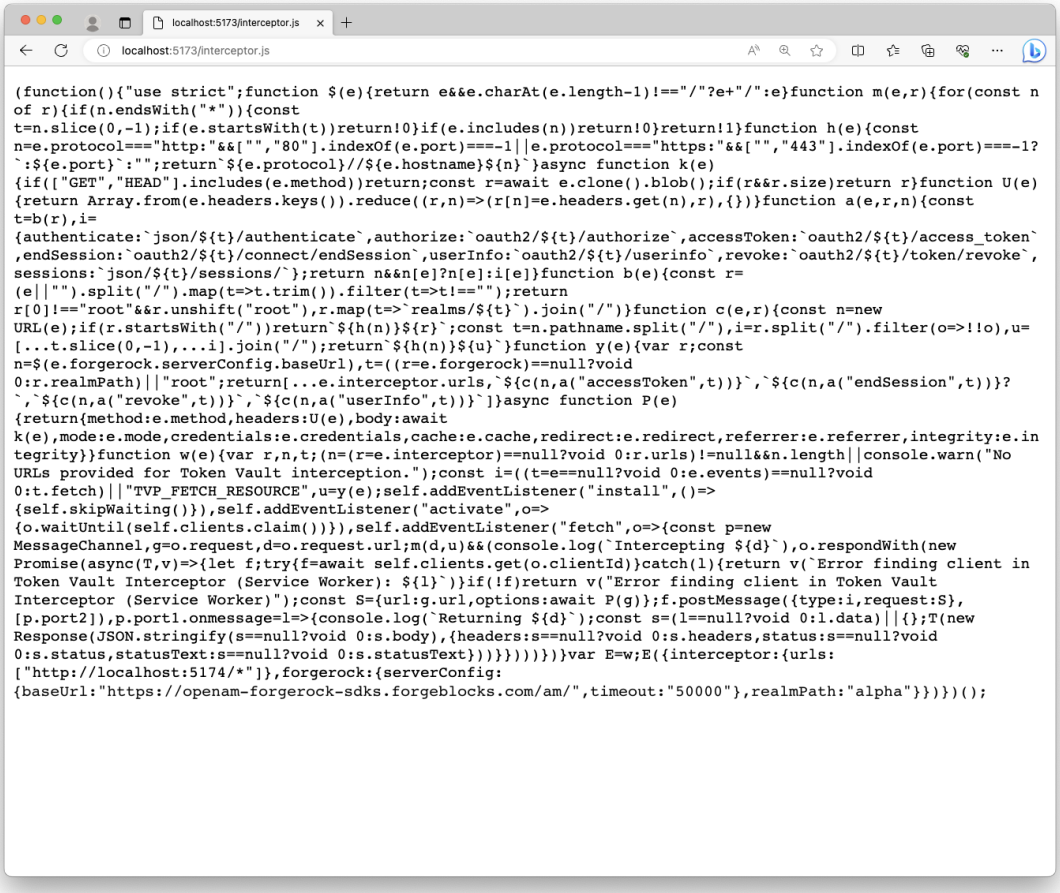


Figure 6. Raw JavaScript of Interceptor

Implement the Token Vault Client

Now that we have all the separate pieces set up, wire it all together with the Token Vault Client plugin.

Step 1. Add HTML element to `index.html`

When we initiate the Token Vault Proxy, it needs a real DOM element to mount to. The easiest way to ensure we have a proper element is to add it to the `index.html` directly.

React app `index.html` file

```
@@ todo-react-app/index.html @@

@@ collapsed @@

    <body>
      <!-- Root div for mounting React app -->
      <div id="root" class="cstm_root"></div>
+
+     <!-- Root div for mounting Token Vault Proxy (iframe) -->
+     <div id="token-vault"></div>

      <!-- Import React app -->
      <script type="module" src="/src/index.tsx"></script>
    </body>
  </html>
```

Step 2. Import and initialize the `client` module

First, import the `client` module and remove the `TokenStorage` module from the SDK import.

Second, call the `client` function with the below minimal configuration. This is how we "glue" the three entities together within your main app. This function returns an object that we use to register and instantiate each entity.

React app `index.tsx` source file

```
@@ todo-react-app/src/index.tsx @@

- import { Config, TokenStorage } from '@forgerock/javascript-sdk';
+ import { Config } from '@forgerock/javascript-sdk';
+ import { client } from '@forgerock/token-vault';
  import ReactDOM from 'react-dom/client';

@@ collapsed @@

+ const register = client({
+   app: {
+     origin: c.TOKEN_VAULT_APP_ORIGIN,
+   },
+   interceptor: {
+     file: '/interceptor.js', // references public/interceptor.js
+   },
+   proxy: {
+     origin: c.TOKEN_VAULT_PROXY_ORIGIN,
+   },
+ });

/**
   * Initialize the React application
   */
  (async function initAndHydrate() {

@@ collapsed @@
```

Remember, the `file` reference within the `interceptor` object needs to point to the *built* Token Vault Interceptor file, which will be located in the `public` directory as a static file but served from the root, not the *source file* itself.

This function ensures the app, Token Vault Interceptor and Token Vault Proxy are appropriately configured.

Step 2. Register the interceptor, proxy, and token store

Now that we've initialized and configured the client, we now register the Token Vault Interceptor, the Token Vault Proxy, and the token vault store just under the newly added code from above:

React app `index.tsx` source file

```
@@ todo-react-app/src/index.tsx @@

@@ collapsed @@

   proxy: {
      origin: c.TOKEN_VAULT_PROXY_ORIGIN,
    },
  });
+
+ // Register the Token Vault Interceptor
+ await register.interceptor();
+
+ // Register the Token Vault Proxy
+ await register.proxy(
+   // This must be a live DOM element; it cannot be a Virtual DOM element
+   // `token-vault` is the element added in Step 1 above to `todo-react-app/index.html`
+   document.getElementById('token-vault') as HTMLElement
+ );
+
+ // Register the Token Vault Store
+ const tokenStore = register.store();

/**
   * Initialize the React application
   */
  (async function initAndHydrate() {

@@ collapsed @@
```

Registering the Token Vault Interceptor is what requests and registers the Service Worker. Calling `register.interceptor` returns the `ServiceWorkerRegistration` object that can be used to unregister the Service Worker, as well as other functions, if that's needed. We won't be implementing that in this tutorial.

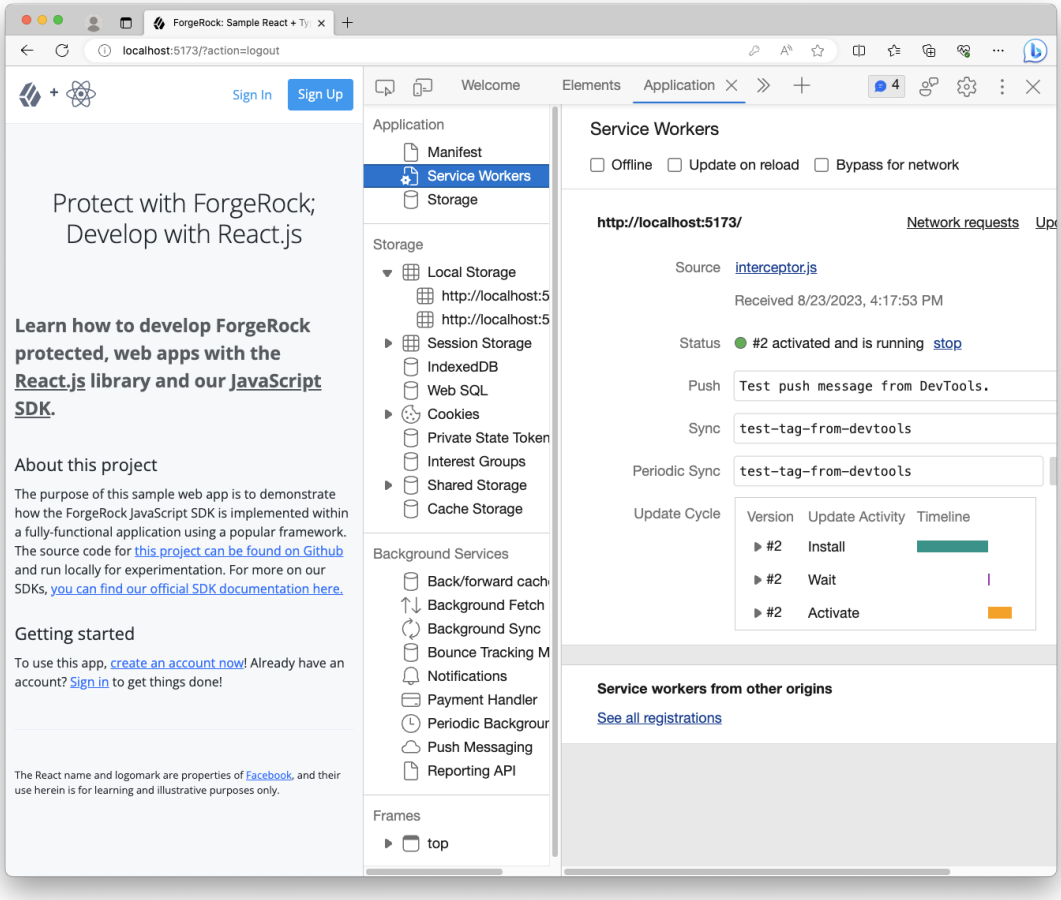


Figure 7. Sample app with Token Vault Interceptor active

Registering the Token Vault Proxy constructs the `iframe` component and mounts it do the DOM element passed into the method. It's important to note that this must be a real, available DOM element, not a Virtual DOM element. This results in the Token Vault Proxy being "registered" as a child frame and, therefore, accessible to your main app.

Calling `register.proxy` also returns an optional reference to the DOM element of the `iframe` that can be used to manually destroy the element and the Token Vault Proxy, if needed.

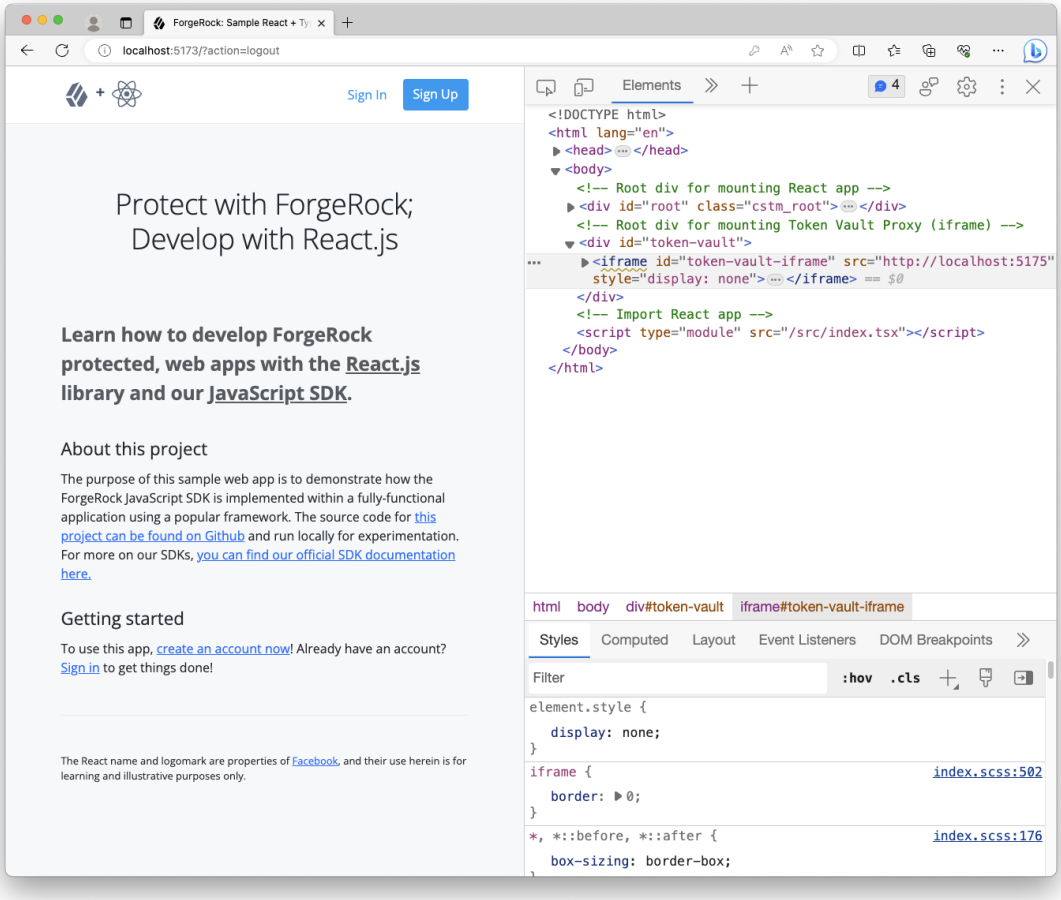


Figure 8. Sample app with Token Vault Proxy mounted in DOM

Finally, registering the store provides us with the object that replaces the default token store within the SDK. There are some additional convenience methods on this `store` object that we'll take advantage of later in the tutorial.

You will see a few errors in the console, but don't worry about those at the moment. The next steps will resolve them.

Step 3. Replace the SDK's default token store

Within the existing SDK configuration, pass the `tokenStore` object we created in the previous step to the `set` method to override the SDK's internal token store.

React app `index.tsx` source file

```
@@ todo-react-app/src/index.tsx @@

@@ collapsed @@

  // Configure the SDK
  Config.set({
    clientId: import.meta.env.WEB_OAUTH_CLIENT,
    redirectUri: import.meta.env.REDIRECT_URI,
    scope: import.meta.env.OAUTH_SCOPE,
    serverConfig: {
      baseUrl: import.meta.env.AM_URL,
      timeout: import.meta.env.TIMEOUT,
    },
    realmPath: import.meta.env.REALM_PATH,
+   tokenStore, // this references the Token Vault Store we created above
  });

@@ collapsed @@
```

This configures the SDK to use the Token Vault Store, which is within the Token Vault Proxy, and that it needs to manage the tokens internally.

Step 4. Check for existing tokens

Currently in our application, we check for the existence of stored tokens to provide a hint if our user is authorized. Now that the main app doesn't have access to the tokens, we have to ask the Token Vault Proxy if it has tokens.

To do this, replace the SDK method of `TokenStorage.get` with the Token Vault Proxy `has` method:

React app `index.tsx` source file

```
@@ todo-react-app/src/index.tsx @@

@@ collapsed @@

   let isAuthenticated = false;
    try {
-     isAuthenticated = !((await TokenStorage.get()) == null);
+     isAuthenticated = !!(await tokenStore?.has())?.hasTokens;
    } catch (err) {
      console.error(`Error: token retrieval for hydration; ${err}`);
    }

@@ collapsed @@
```

Note that this doesn't return the tokens as that would violate the security of keeping them in another origin, but the Token Vault Proxy will inform you of their existence. This is enough to hint to our UI that the user is likely authorized.

Build and run the apps

At this point, all the necessary entities are set up. We can now run all the needed servers and test out our new application with Token Vault enabled.

Open three different terminal windows, all from within the root of this project. Enter each command in its own window:

First terminal window

```
npm run dev:react
```

Second terminal window

```
npm run dev:api
```

Third terminal window

```
npm run dev:proxy
```

Allow all the commands to complete the build and start the development servers.

Then, visit `http://localhost:5173` in your browser of choice. The to-do application should look and behave no different from before.

Open the dev tools of your browser, and proceed to sign in to the app. You will be redirected to the login page, and then redirected back after successfully authenticating. You may notice some additional redirection within the React app itself, this is normal.

Once you land on the home page, you should see the "logged in experience" with your username in the success alert.

To test whether Token Vault is successfully implemented, go to the **Application** or **Storage** tab of your dev tools and inspect the `localStorage` section.

You should see two origins: `http://localhost:5173`, our main app, and `http://localhost:5175`, our Token Vault Proxy.

Your user's tokens should be stored under the Token Vault Proxy origin on port `5175`, not under the React app's origin on port `5173`.

If you observe that behavior, then you have successfully implemented Token Vault. Congratulations, your tokens are now more securely stored!

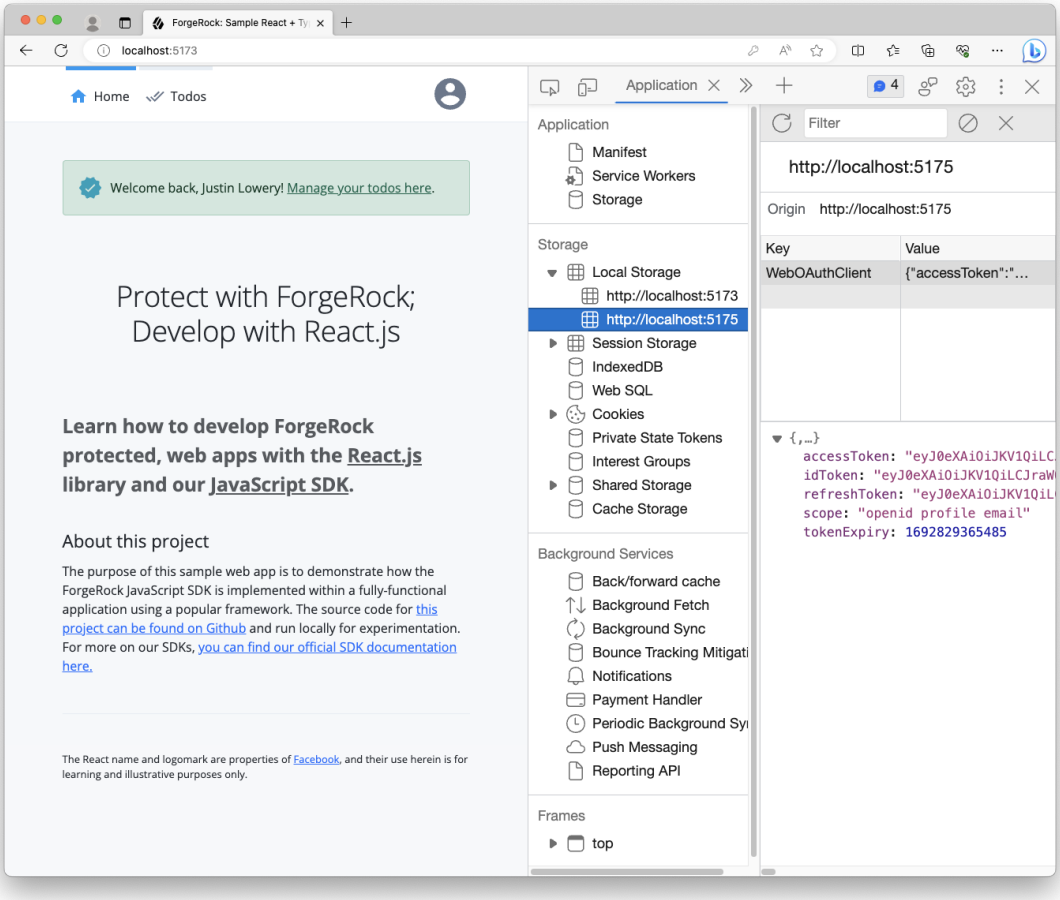


Figure 9. To-do app home page with Token Vault and logged in

If you don't see the tokens in the Token Vault Proxy origin's `localStorage`, then follow the troubleshooting section below:

Troubleshooting

Getting failures in the service worker registration

Make sure your dedicated Vite configuration is correct.

Also, check the actual file output for the `interceptor.js` file. If the built file has ES Module syntax in it, or it looks incomplete, then it can cause this issue — Service Workers in some browsers, even the latest versions, don't support the same ES version or features as the main browser context.

Tokens are not being saved under any origin

Open up your browser's dev tools and ensure the following:

1. Your Token Vault Interceptor is running under your main app's origin.
2. Do you have an `/access_token` request? It comes after the `/authorize` request and redirect.

3. Your Token Vault Interceptor is intercepting the `/access_token` request. If it is, you should see two outgoing requests: one for the main app and one for the Token Vault Proxy.
4. Your Token Vault Proxy is running within the `iframe` and forwarding the request.
5. There are no network errors.
6. There are no console errors.

> **Tip**
>
> Enabling **Preserve Log** for both the **Console** and the **Network** tabs is very helpful.

I'm getting a CORS failure

Make sure you have both origins listed in your CORS configuration.

Additionally, it's best if you use the Ping SDK template when creating a new CORS config in your server.

If both origins are listed, make sure you have no typos in the allowed methods. The methods like `GET` and `POST` are case-sensitive. Also, check the headers, which are NOT case-sensitive.

Allow Credentials must be enabled.

Troubleshoot the Token Vault

How do I fix CORS errors?

Make sure your CORS configuration in your authorization server allows and accepts origins from both the origin of your main app and also the origin of the Token Vault Proxy.

These two origins should be unique from one another.

What can cause iframe errors?

This is likely an error coming from the `/authorize` request to collect OAuth 2.0 or OIDC tokens.

Make sure you are using Ping SDK for JavaScript 4.0 or newer.

To diagnose the issue, copy the full `/authorize` request URL from the network tab in your dev tools and paste it into your browser's URL field to directly visit it.

A 400 error coming from the `/authorize` endpoint could be caused by a misconfiguration. For example, if a consent page is rendering ensure you enabled the implied consent property in both your OAuth 2.0 Provider and the OAuth 2.0 client.

Make sure you are allowing the use of third-party cookies. For example, the incognito or private modes in Chromium browsers *disable* third-party cookies by default, as do Webkit-based browsers.

Why are the tokens not being stored?

If you are receiving tokens from the `/access_token` endpoint but they are not getting stored, this is likely caused by the Token Vault Interceptor not routing the requests to the Token Vault Proxy configured in your main app.

Only the Token Vault Proxy can store tokens when the Token Vault is enabled.

To fix this, ensure your config is identical between your main app's SDK config found in `Config.set()` and the config found in your Token Vault Interceptor file.

We recommend using environment variables, rather than hard-coding the values directly in each of the modules.

Why does the Interceptor (Service Worker) not work or report errors in Firefox or Safari?

Your bundler is likely not bundling the Token Vault Interceptor into a single file, and language features are present in the bundle that these browsers do not support in a Service Worker context.

Ensure that your bundler configuration, such as Vite or Webpack, is creating a single file output and that it is down-leveled to `ES2020`.

We recommend a dedicated bundle configuration for the Token Vault Interceptor, separate from your application bundle.

What can cause “400 Proxy Error”?

These errors often occur when the Token Vault Proxy itself is encountering an error, and not actually an error response from your authorization server.

Inspect the network tab in your dev tools to find the specific error message in the response, which will help you debug the underlying issue.