Self-Service REST API Reference

/ ForgeRock Identity Management 6.0

Latest update: 6.0.0.7

Lana Frost

,
, ,

Abstract

Reference documentation for the ForgeRock® Identity Management Self-Service REST API.

Table of Contents

Preface

ForgeRock Identity Platform™ serves as the basis for our simple and comprehensive Identity and Access Management solution. We help our customers deepen their relationships with their customers, and improve the productivity and connectivity of their employees and partners. For more information about ForgeRock and about the platform, see https://www.forgerock.com.

1. Using This Guide

This guide is intended for anyone developing a self-service application that acts as a client of ForgeRock Identity Management (IDM).

This guide is written with the expectation that you already have basic familiarity with the following topics:

- REST APIs
- JavaScript Object Notation (JSON) and basic IDM configuration

2. Accessing Documentation Online

ForgeRock publishes comprehensive documentation online:

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

  While many articles are visible to community members, ForgeRock customers have access to much more, including advanced information for customers using ForgeRock software in a mission-critical capacity.

- ForgeRock 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.

3. Using the ForgeRock.org Site

The ForgeRock.org site has links to source code for ForgeRock open source software, as well as links to the ForgeRock forums and technical blogs.

If you are a *ForgeRock customer*, raise a support ticket instead of using the forums. ForgeRock support professionals will get in touch to help you.

Chapter 1

Understanding Self-Service Processes

IDM provides a sample Self-Service UI that implements a number of processes, such as self-registration and password reset, based on a Self-Service REST API.

Self-service processes are configured in files named `selfservice-process-name.json` in your project's `conf` directory. Every self-service process steps through a series of *stages*, each with its own requirements, until the end of the process is reached or until the process exits with an exception. The flow through the stages differs, depending on how you have configured the process.

You can customize the default processes, or write your own custom processes by implementing the stages described in "*Self-Service Stages*". For information about the default Self-Service UI, see "*Configuring User Self-Service*" in the *Integrator's Guide*.

The Self-Service REST API supports only two HTTP requests:

- `GET` which obtains the requirements for that stage
- `POST` with `_action=submitRequirements`

The response to the `POST` request instructs the client how to proceed. The response can have one of two outcomes:

- Success—all requirements have been submitted and the process advances to the next stage.
- Failure—the behavior here differs by stage. Certain stages will exit with an exception, others will convert the exception into an error that the client must handle, others will simply return the requirements again.

1.1. The Self-Service Process Flow

Each self-service process advances through the stages in the order in which they are listed in the `stageConfigs` array in the process configuration file. The password reset process, for example, might include the following stages:

```
{
    "stageConfigs" : [
        {
            "name": "parameters",
            ...
        },
        {
            "name" : "userQuery",
            ...
        },
        {
            "name" : "validateActiveAccount",
            ...
        },
        {
            "name" : "emailValidation",
            ...
        },
        {
            "name" : "kbaSecurityAnswerVerificationStage",
            ...
        },
        {
            "name" : "resetStage",
            ..
        }
    ],
    ...
}
```

A process definition also includes an optional `snapshotToken` and `storage` parameter, for example:

```
{
    "stageConfigs" : [
    ],
    "snapshotToken" : {
        "type" : "jwt",
        "jweAlgorithm" : "RSAES_PKCS1_V1_5",
        "encryptionMethod" : "A128CBC_HS256",
        "jwsAlgorithm" : "HS256",
        "tokenExpiry" : 300
    },
    "storage" : "stateless"
}
```

The `snapshotToken` specifies the format of the token that is passed between the client and the server with each request. By default, this is a JWT token, stored statelessly, which means that the state is stored in the client, rather than on the server side. Because some legacy clients cannot handle the long URLs provided in a JWT token, you can store the snapshot token locally, as a `uuid` with the following configuration:

```
{
...
    "snapshotToken" : {
        "type" : "uuid"
    },
    "storage" : "local"
}
```

In this case, the 16-character token is stored in the local JSON store. For more information, see "Tokens and User Self-Service" in the *Integrator's Guide*.

If you do not include the `snapshotToken` and `storage` in the configuration, the default stateless configuration applies.

When a stage advances, it can optionally insert parameters into the process context or *state* for consumption by stages that occur later in the process. The snapshot token is essentially the state of the stage. It is the container in which `state`, `successAdditions` and other data are stored, and then returned to the client at the end of the process, as an encrypted blob named `token`.

Sample configurations for each default self-service process are available in the `/path/to/openidm/samples/example-configurations/self-service` directory.

Each self service process has a specific endpoint under `openidm/selfservice` with the name of the process; for example `openidm/selfservice/reset` for the Password Reset process. If you create a custom self-service process with a configuration file such as `selfservice-myprocess.json`, you produce an endpoint such as `http://localhost:8080/openidm/selfservice/myprocess`.

All REST actions occur against that endpoint. For example, the following initial GET request against the password reset endpoint returns the requirements for the following stage:

```
$ curl \
 --header "X-OpenIDM-Username: anonymous" \
 --header "X-OpenIDM-Password: anonymous" \
 --request GET \
 "http://localhost:8080/openidm/selfservice/reset"
{
  "_id": "1",
  "_rev": "-852427048",
  "type": "captcha",
  "tag": "initial",
  "requirements": {
    "$schema": "http://json-schema.org/draft-04/schema#",
    "description": "Captcha stage",
    "type": "object",
    "required": [
      "response"
    ],
    "properties": {
      "response": {
        "recaptchaSiteKey": "6LcvE1IUAAAAAA5AI1SZzZJl-AlGvHM_dzUg-0_S",
        "description": "Captcha response",
        "type": "string"
      }
    }
  }
}
```

The default Self-Service UI implements the following processes:

- Self-registration (under the endpoint `selfservice/registration`)
- Password reset (under the endpoint `selfservice/reset`)
- Forgotten username retrieval (under the endpoint `selfservice/username`)
- Social registration (under the endpoint `selfservice/socialUserClaim`)
- Progressive profile completion (under `selfservice/profile`)
- Security question updates (under `selfservice/kbaUpdate`)
- Terms and conditions (under `selfservice/termsAndConditions`)

The remainder of this guide describes each stage, its requirements, and expected responses. It also describes the logical flow for each default self-service process.

Chapter 2

Self-Service Stages

This chapter describes the individual stages that can be called by a self-service process, the purpose of the stage, any required parameters, dependencies on preceding or following stages, and the expected stage output.

The stages are listed in alphabetical order, for ease of reference but they cannot be configured in random order. For example, some stages require input from the process `state` that has been populated by a preceding stage.

The `identityServiceURL` is a required parameter for most self-service stages. The self-service stages operate on a managed object. The `identityServiceURL` indicates the object type, for example, `managed/user`.

2.1. All-In-One Registration

A registration process that consists of more than one stage can include an optional "*super stage*" named `allInOneRegistration`, that is set outside of the `stageConfigs` array as follows:

```
"allInOneRegistration" : true
```

All-in-one registration covers a number of registration stages. If this property is `true`, in the registration process configuration, IDM scans the configuration for any of the following stages:

- `parameters`
- `captcha`
- `termsAndConditions`
- `kbaSecurityAnswerDefinitionStage`
- `consent`
- `idmUserDetails`

If any of these stages are found, the individual stages are effectively removed from the configuration and a new configuration is generated that accumulates all the found stages.

The purpose of all-in-one registration is to obtain a set of initial requirements, then to advance to the end of all six stages simultaneously. This enables self-registration to be completed on a single

registration form. As the process advances, it gathers any output, errors, and so on from all six stages (or however many stages have been configured). The process then returns whatever was gathered from the cumulative stages, including any outstanding requirements. Depending on the output, the process might be required to go through the stages more than once, as the outstanding requirements are provided.

Important

All-in-one registration requires multiple registration stages. If your registration process includes only one stage, for example, `consent`, `allInOneRegistration` must be set to `false`, to preserve the registration flow.

If all-in-one registration is `false`, any additional stages listed in the registration process (`selfservice-registration.json`) must be listed *after* the parameters and `idmUserDetails` stages. If a stage occurs before the `idmUserDetails` stage without all-in-one registration, both social and regular registration will not work.

2.2. OpenAM Auto-Login Stage

This stage is used to perform auto-login with ForgeRock Access Management (AM). The stage is similar to the local auto-login stage but also requires the `returnParams` in `state` (populated in the Parameters Stage).

The stage obtains the `user` object but instead of creating a JWT, creates an AM authentication request. If authentication fails, the server generates a bad request exception. If authentication is successful, AM responds with a URL that is the `successURL`. The `successURL` is added to the `successAdditions` and the process moves on to the next stage.

Example configuration

```
{
    "name" : "openAmAutoLogin",
    "authenticationEndpoint" : "https://openam.example.com:8443/openam/oauth2",
    "openAMBaseUrl" : "http://openam.example.com:8080",
    "identityUsernameField": "userName",
    "identityPasswordField": "password"
}
```

Dependencies

This stage should appear towards the end of a process—it cannot be the first stage in a process.

Required Parameters

- `authenticationEndpoint` - the AM Authentication Endpoint URL.
- `openAMBaseUrl` - the URL of the AM server.
- `identityUsernameField` - the managed object property that contains the username.

- `identityPasswordField` - the managed object property that contains the user password.

2.3. Attribute Collection Stage

The purpose of this stage is to collect managed object properties to insert into the user profile. The list of properties to be collected is defined as part of the configuration.

This stage updates the managed object directly, and checks whether attributes are *required*. If required attributes are not provided, the stage returns the list of requirements again. This stage can throw an exception if there is an error attempting to save the updated attributes.

Example configuration

```
{
    "name" : "attributecollection",
    "identityServiceUrl" : "managed/user",
    "uiConfig" : {
        "displayName" : "Add your telephone number",
        "purpose" : "Help us verify your identity",
        "buttonText" : "Save"
    },
    "attributes" : [
        {
            "name" : "telephoneNumber",
            "isRequired" : true
        }
    ]
}
```

Dependencies

No dependencies on previous or following stages. This stage can occur anywhere in a process.

Required Parameters

- `identityServiceUrl` - the managed object type on which this stage acts
- `uiConfig` - how the requirements list is conveyed to an end user
- `attributes` - the array of attributes to be collected. For each attribute, the `isRequired` parameter indicates whether the attribute is mandatory for the stage to proceed.

2.4. Captcha Stage

This stage verifies a `response` variable populated in `state` by the reCaptcha mechanism. If the response is missing, or if validation fails (typically if the configuration does not include the required reCaptcha configuration parameters) the stage throws a bad request exception. If validation succeeds, the process advances to the next stage.

Example configuration

```
{
    "name" : "captcha",
    "recaptchaSiteKey" : "6LdahVIUAAAAAJcwGTWdl4OsG9tpdgFIyZKUSzyU",
    "recaptchaSecretKey" : "6LdahVIUAAAAANF-O17E-b8PyBqLrhLaOHUX8ch-",
    "recaptchaUri" : "https://www.google.com/recaptcha/api/siteverify"
},
```

Dependencies

No dependencies on previous or following stages. This stage can occur anywhere in a process.

Required Parameters

- `recaptchaSiteKey` - invokes the reCAPTCHA service
- `recaptchaSecretKey` - authorizes communication between IDM and the reCAPTCHA server to verify the user's response
- `recaptchaUri` - the reCaptcha verification API

2.5. Conditional User Stage

Defines a condition, that results in a boolean (`true` or `false`). The outcome of the condition determines which stage should be executed next.

Example configuration

```
{
    "name": "conditionaluser",
    "identityServiceUrl": "managed/user",
    "condition": {
        "type": "kbaQuestions"
    },
    "evaluateConditionOnField": "user",
    "onConditionFalse": {
        "name": "kbaUpdateStage",
        "kbaConfig": null,
        "identityServiceUrl" : "managed/user",
        "uiConfig" : {
            "displayName" : "Update your security questions",
            "purpose" : "Please review and update your security questions",
            "buttonText" : "Update"
        }
    }
}
```

Dependencies

No dependencies on previous or following stages. This stage can occur anywhere in a process. If the condition evaluates to `true`, the process moves on to the next stage.

Required Parameters

- `identityServiceUrl` - the managed object type on which this stage acts
- `condition` - the condition type, which can be one of the following:
  - `kbaQuestions` - a boolean (`true` or `false`) that indicates whether configured security questions have been answered
  - `queryFilter` - a common filter expression such as `"filter" : "/co eq \"US\""`
  - `script` - enables you to configure a custom scripted condition
  - `loginCount` - a condition based on the number of password or social authentication-based login requests
  - `terms` - a boolean (`true` or `false`) that indicates whether configured Terms and Confitions have been accepted
  - `timesincelogin` - sets a condition based on the period of time since the last login, in years, months, weeks, days, hours, and minutes
- `evaluateConditionOnField` - the property on which the condition should be evaluated
- `onConditionFalse` - the details of the stage to be called if the condition evaluates to false

2.6. Consent Stage

This stage evaluates a boolean `consentGiven` (`true` or `false`). If consent is required but not given, the stage fails with an exception. It is up to the client to handle that exception, for example, to prevent registration if the user does not provide consent.

Example configuration

This stage is not configured in the same way as the other self-service stages (in the `stageConfigs` array). The configuration is in a `consent.json` file in the project `conf` directory and includes only one parameter:

```
{
    "enabled" : true
}
```

Dependencies

No dependencies on previous or following stages. This stage can occur anywhere in a process.

Required Parameters

- None, other than for consent to be enabled with `"enabled" : true` in `conf/consent.json`.

2.7. Email Validation Stage

This stage retrieves the email address from `state` (or in response to initial requirements) then verifies the validity of the email address with the user who submitted the requirements through an email process.

Example configuration

```
{
    "name" : "emailValidation",
    "identityEmailField" : "mail",
    "emailServiceUrl" : "external/email",
    "emailServiceParameters" : {
        "waitForCompletion" : false
    },
    "from" : "info@admin.org",
    "subject" : "Reset password email",
    "mimeType" : "text/html",
    "subjectTranslations" : {
        "en" : "Reset your password",
        "fr" : "Réinitialisez votre mot de passe"
    },
    "messageTranslations" : {
        "en" : "Click to reset your password <a href=\"%link%\">Password reset link</a>",
        "fr" : "Cliquez pour réinitialiser votre mot de passe<a href=\"%link%\">Mot de passe lien de
 réinitialisation</a>"
    },
    "verificationLinkToken" : "%link%",
    "verificationLink" : "https://localhost:8443/#passwordReset/"
},
```

Dependencies

This stage expects a preceding stage to populate the user email address in `state`. The stage has no downstream dependencies.

Required Parameters

- Email configuration. For more information, see "Configuring Emails for Self-Service Registration" in the *Integrator's Guide*.

2.8. IDM User Details Stage

This stage collects new user data and stores it in `state`. This is the only stage that sets up a user from nothing. The stage does not *create* a managed object directly—it simply gathers and stores the data. The Self-Registration Stage consumes the stored user data and creates the managed object from it.

The IDM User Details stage executes multiple times, requesting additional requirements each time. There are different ways for the stage to advance, depending on how the user create request is initiated.

If the user completes a self-service registration form, the input contains a `user` object, collected from the form, and populates that user in `state`. If the user registers through social authentication, the stage reads the profile from the remote identity provider, normalizes it, then maps it to a user object. That user object is then put into `state`.

If the new user object in `state` is incomplete or does not meet policy requirements, the stage returns a new set of requirements, indicating the collected data and the missing data. The registering user is requested to submit the additional data, then the stage revalidates the object in `state`. When all of the required data to register a user is present, the process advances to the next stage.

Important

The user data remains in `state`—no managed user object is created.

Example configuration

```
{
    "name" : "idmUserDetails",
    "identityEmailField" : "mail",
    "socialRegistrationEnabled" : true,
    "identityServiceUrl" : "managed/user",
    "registrationProperties" : [
        "userName",
        "givenName",
        "sn",
        "mail"
    ],
    "registrationPreferences": ["marketing", "updates"]
},
```

Dependencies

This stage *must* occur in any registration process. It has no dependencies on previous stages but must have the Self-Registration Stage somewhere downstream in the process, to create the managed user object.

Required Parameters

- `identityEmailField` - the attribute on the managed user object that contains the user email.
- `identityServiceUrl` - the managed object type on which this stage acts.
- `socialRegistrationEnabled` - optional, `false` if not specified. Indicates whether the stage must read the user profile from a remote identity provider and normalize it.
- `registrationProperties` - an array of properties that must be provided by a registering user in order for the stage to progress.
- `registrationPreferences` - optional, an array of properties that can be requested after the user has provided the required properties.

2.9. KBA Security Answer Definition Stage

In the context of registration, this stage supplies security questions to the user and captures the answers provided by the user.

The stage validates any answers against the user object. If the requirement is not met (incorrect number of questions answered correctly) the stage throws a bad request exception and increments the failure count of the managed user. If the requirement is met (correct number of questions answered correctly) the process advances to the next stage.

This stage also disallows users from entering custom questions that duplicate any questions defined by the administrator, regardless of the locale. It does this comparison by removing any special characters and making a lower case comparision. For example, `What Is YoUr FaVorite COLOR????` would be evaluated as the same question as `what is your favorite color?`.

Example configuration

```
{
    "name" : "kbaSecurityAnswerDefinitionStage",
    "kbaConfig" : null
},
```

Dependencies

The stage depends on a previous stage to populate the user ID in `state`. It has no dependencies on following stages.

Required Parameters

- `kbaConfig` - reads the KBA configuration from the corresponding `selfservice.kba.json` file

2.10. KBA Security Answer Verification Stage

This stage verifies security answers and validates user lockout. The stage requires a user ID in `state`.

The stage reads the user object and validates that the user has not already failed to answer the security questions. The stage then obtains the configured security questions, and returns the minimum number of randomly selected questions as a requirement.

The stage validates any answers against the user object. If the requirement is not met (incorrect number of questions answered correctly) the stage throws a bad request exception and increments the failure count of the managed user. If the requirement is met (correct number of questions answered correctly) the process advances to the next stage.

Example configuration

```
{
    "name" : "kbaSecurityAnswerDefinitionStage",
    "kbaConfig" : null
},
```

Dependencies

The stage depends on a previous stage to populate the user ID in `state`. It has no dependencies on following stages.

Required Parameters

- `kbaConfig` - reads the KBA configuration from the corresponding `selfservice.kba.json` file

2.11. KBA Update stage

The KBA Update stage is used as part of progressive profile completion to enable users to update their existing security questions and to add any additional questions that are needed. This stage updates the user object directly. If a user fails to provide sufficient questions, the stage returns the requirements again. If the object cannot be updated, the stage throws an exception. The stage outputs nothing to the `state` and has no downstream dependencies.

Example configuration

```
{
    "name": "kbaUpdateStage",
    "kbaConfig": null,
    "identityServiceUrl" : "managed/user",
    "uiConfig" : {
        "displayName" : "Update your security questions",
        "purpose" : "Please review and update your security questions",
        "buttonText" : "Update"
    }
}
```

Dependencies

No dependencies on previous or following stages. This stage can occur anywhere in a process. If the condition evaluates to `true`, the process moves on to the next stage.

Required Parameters

- `kbaConfig` - returns the minimum number of security questions that must be provided
- `identityServiceUrl` - the managed object type on which this stage acts
- `uiConfig` - how the requirements are conveyed to an end user

2.12. Local Auto-Login Stage

This stage is used to perform auto-login with IDM. The stage obtains the `OAuth Login` from `state` and populates the `user` object (`username` and `password`) in `state`.

The stage adds the OAuth login to the `successAdditions` (with a value of `true`) and adds the `successURL` from its own configuration. If IDM can obtain all those details from `state`, it takes the user object,

locates the `username` and `password`, and generates a `CREDENTIAL_JWT`. That JWT is then placed in the `successAdditions` parameter.

If IDM is unable to generate the `CREDENTIAL_JWT` it generates an internal server error (500).

Example configuration

```
{
    "name" : "localAutoLogin",
    "successUrl" : "",
    "identityUsernameField": "userName",
    "identityPasswordField": "password"
}
```

Dependencies

This stage should appear towards the end of a process—it cannot be the first stage in a process.

Required Parameters

- `successURL` - the URL to which an end-user should be redirected following successful registration.
- `identityUsernameField` - the managed object property that contains the username.
- `identityPasswordField` - the managed object property that contains the user password.

2.13. Parameters Stage

This stage captures parameters in the original request. To advance, the stage assesses the input body. Any values that have been passed in and are listed in the configuration are put into `state`. The stage ignores any values that are not listed in the configuration. The self-service mechanism passes the parameters back to the client at the end of the process.

By default, this stage is required *only* if you are integrating IDM with ForgeRock Access Management. The stage is added automatically if you use the UI to configure a self-service process, but can generally be ignored unless a custom client or UI requires it.

Example configuration

```
{
    "name" : "parameters",
    "parameterNames" : [
        "returnParams"
    ]
}
```

Dependencies

In all of the default IDM self-service processes, this must be the first stage in the process. In a custom process, the stage has no order dependencies, and can occur anywhere in a process. All this stage does is to copy named parameters into `successAdditions` for the process to output at `tag:end`.

Required Parameters

- `parameterNames` - a list of parameters the stage supports. These parameters are returned in the requirements.

2.14. Patch Object Stage

Currently, this stage is used *only* to patch the managed object with the terms and conditions acceptance obtained from `state`. If the terms and conditions state is not present, the stage simply advances to the next stage in the process.

Example configuration

```
{
    "name" : "patchObject",
    "identityServiceUrl" : "managed/user"
}
```

Dependencies

This stage requires the Terms and Conditions Stage to have preceded it. It can be followed by any stage and can occur anywhere in a process.

Requirements

- `identityServiceUrl` - the managed object type on which this stage acts

2.15. Password Reset Stage

This stage updates the managed object directly, changing the value of the configured `identityPasswordField`. To gather the initial requirements the stage reads the managed user object, and checks that the `email` and `userID` of the object match what is in `state`. If they do not match, the stage exits with a `Bad request exception`.

If they do match, the stage returns with its requirements (the new `password` value). When the requirements are submitted, the stage advances, locates the `userId` again, the new `password`. If the password is empty, the stage throws an exception. If the password is valid, the stage patches the managed user object directly to update the password. If the patch fails, the stage returns the requirements again, along with an error message (for example, a password policy requirement).

Example configuration

```
{
    "name" : "resetStage",
    "identityServiceUrl" : "managed/user",
    "identityPasswordField" : "password"
}
```

Dependencies

This stage cannot be the first stage in a process. It expects a previous stage to populate the `userId` and `mail` attributes of the `user` in `state`.

Required Parameters

- `identityServiceUrl` - the managed object type on which this stage acts
- `identityPasswordField` - the managed object property that contains the user password.

2.16. Self-Registration Stage

This is currently the final stage in the default user registration process. The stage obtains all the user details from `state`. When the stage advances, it checks `state` for any `idpdata`, combines that with the user data, and creates the managed user object. This stage *must* occur in any registration process.

Note

If you are integrating IDM with AM, the OpenAM Auto-Login Stage can follow this stage.

Example configuration

```
{
    "name" : "selfRegistration",
    "identityServiceUrl" : "managed/user"
},
```

Dependencies

This stage *must* come after a stage that has populated the user in `state`. If the user is absent, the stage exits with an illegal argument exception.

Required Parameters

- `identityServiceUrl` - the managed object type that the stage creates.

2.17. Social User Claim Stage

This stage enables an existing managed user to claim a social identity. The stage obtains a `CLIENT_TOKEN` from some social identity provider. That token includes the following data:

- OAuth token
- Identity provider name
- Renewal token

- Expiration date

Using the `CLIENT_TOKEN`, the stage retrieves the user profile from the social identity provider and normalizes the profile into a user object (using the regular normalization mapping for social identity providers). For more information on this mapping, see "Many Social Identity Providers, One Schema" in the *Integrator's Guide*.

If the stage is unable to retrieve the user profile, or unable to normalize it using the mapping, it exits with an exception. It does not return any missing requirements.

When the user profile has been normalized, the stage attempts to identify any existing managed users that match the profile. If there are no matches, it simply advances to the next stage in the process. If it finds a match, it extracts the existing managed object and returns that as a new set of requirements.

The new requirement is that the user must provide their `password`, either their managed/user password, or the password to another social identity provider if they registered through a separate identity provider.

The stage then does the following:

- Verifies the login
- Creates a `managed/idp` object for the user
- Establishes a relationship between the managed object and the idp object
- Puts `OAUTH_LOGIN:true` into `state`
- Puts a `claimedProfile` containing the URL of the managed object that was claimed into `successAdditions`

Example configuration

```
{
    "name" : "socialUserClaim",
    "identityServiceUrl" : "managed/user",
    "claimQueryFilter" : "/mail eq \"{{mail}}\""
},
```

Dependencies

This stage has no dependencies on previous or subsequent stages and can occur anywhere in a process.

Required Parameters

- `identityServiceUrl` - the managed object type against which the stage verifies the profile.
- `claimQueryFilter` - the query filter that is used to locate the managed object from the social identity provider profile.

Notice the double-brace notation in preceding example `"claimQueryFilter" : "/mail eq \"{{mail}}\""`. This notation indicates that the named property from the user object in `state` is substituted for the double-braced value. In this example, `{{mail}}` would become the value of the `mail` property of the user in `state`, such as `bjensen@example.com` if that was in the user in `state`. You can use this notation with any user property.

2.18. Terms and Conditions Stage

This stage evaluates a boolean `accepted` (`true` or `false`).

Example configuration

This stage is configured in a `selfservice.terms.json` file in the project `conf` directory and includes the following parameters:

```
{
    "versions" : [
        {
            "version" : "1",
            "termsTranslations" : {
                "en" : "Sample terms and conditions"
            },
            "createDate" : "2018-04-10T09:52:25.478Z"
        }
    ],
    "uiConfig" : {
        "displayName" : "We have updated our terms",
        "purpose" : "To proceed, accept these terms",
        "buttonText" : "Accept"
    },
    "active" : "1"
}
```

The stage can stand on its own (as it does in the default registration configuration) or be called from the Conditional User Stage with a configuration similar to the following:

```
{
    "name" : "conditionaluser",
    "identityServiceUrl" : "managed/user",
    "condition" : {
        "type" : "terms"
    },
    "evaluateConditionOnField" : "user",
    "onConditionTrue" : {
        "name" : "termsAndConditions"
    }
},
```

Dependencies

Configured as part of the Conditional User Stage. *Must* have the Patch Object Stage somewhere downstream. This stage can occur anywhere in a process.

Requirements

Requires Terms and Conditions to be accepted before continuing to the next stage:

- If `accept` is absent, the stage returns the requirements again.
- If `accept` is present but `false`, the stage generates an exception. It is up to the client to handle that exception.
- If `accept` is `true`, this stage puts all the outputs into `state` and advances to the next stage.

Outputs

`TERMS_ACCEPTED`, `TERMS_DATE`, and `TERMS_VERSION`

2.19. User Query Stage

This stage queries the managed user repository for a user, based on the supplied query fields. If the stage identifies a user, it populates the `mail`, `userId`, `userName`, and `accountStatus` fields in `state`.

Example configuration

```
{
    "name" : "userQuery",
    "validQueryFields" : [
        "userName",
        "mail",
        "givenName",
        "sn"
    ],
    "identityIdField" : "_id",
    "identityEmailField" : "mail",
    "identityUsernameField" : "userName",
    "identityServiceUrl" : "managed/user",
    "identityAccountStatusField" : "accountStatus"
},
```

Dependencies

This stage has no dependencies on preceding or following stages but cannot be the only stage in a process.

Required Parameters

- `validQueryFields` - an array of fields on which the query can be based.
- `identityIdField` - the managed object property that contains the user ID to be provided to `state`.
- `identityEmailField` - the managed object property that contains the user mail to be provided to `state`.

- `identityUsernameField` - the managed object property that contains the username to be provided to `state`.
- `identityAccountStatusField` - the managed object property that contains the user account status to be provided to `state`.
- `identityServiceUrl` - the managed object type on which this stage acts

Chapter 3

Password Reset Process

Password reset enables registered users to reset their own passwords. The following stages can be included in a password reset process:

- Captcha Stage (optional)
- User Query Stage (mandatory)
- Email Validation Stage (optional)
- KBA Security Answer Verification Stage (optional)
- Password Reset Stage (mandatory)

If all of these stages are configured, the password reset configuration (in `conf/selfservice-profile.json` looks similar to the following:

```
{
    "stageConfigs" : [
        {
            "name" : "captcha",
            "recaptchaSiteKey" : "...",
            "recaptchaSecretKey" : "...",
            "recaptchaUri" : "https://www.google.com/recaptcha/api/siteverify"
        },
        {
            "name" : "userQuery",
            "validQueryFields" : [
                "userName",
                "mail",
                "givenName",
                "sn"
            ],
            "identityIdField" : "_id",
            "identityEmailField" : "mail",
            "identityUsernameField" : "userName",
            "identityServiceUrl" : "managed/user"
        },
        {
            "name" : "emailValidation",
            "identityEmailField" : "mail",
            "emailServiceUrl" : "external/email",
            "emailServiceParameters" : {
                "waitForCompletion" : false
            },
            "from" : "info@example.com",
```

```
            "subject" : "Reset password email",
            "mimeType" : "text/html",
            "subjectTranslations" : {
                "en" : "Reset your password",
                "fr" : "Réinitialisez votre mot de passe"
            },
            "messageTranslations" : {
                "en" : "...Click to reset your password...",
                "fr" : "...Cliquez pour réinitialiser votre mot de passe..."
            },
            "verificationLinkToken" : "%link%",
            "verificationLink" : "https://localhost:8443/#passwordReset/"
        },
        {
            "name" : "kbaSecurityAnswerVerificationStage",
            "kbaPropertyName" : "kbaInfo",
            "identityServiceUrl" : "managed/user",
            "kbaConfig" : null
        },
        {
            "name" : "resetStage",
            "identityServiceUrl" : "managed/user",
            "identityPasswordField" : "password"
        }
    ],
    "snapshotToken" : {
        "type" : "jwt",
        "jweAlgorithm" : "RSAES_PKCS1_V1_5",
        "encryptionMethod" : "A128CBC_HS256",
        "jwsAlgorithm" : "HS256",
        "tokenExpiry" : "300"
    },
    "storage" : "stateless"
}
```

3.1. REST Requests in a Password Reset Process

The following REST requests and responses demonstrate the flow through a simple password reset process. To keep the process simple, this flow does not include the Google ReCAPTCHA stage, or the Security Answer Verification stage:

1. Client initiates the password reset, server returns the `initial` tag:

```
curl \
 --request GET \
 "https://localhost:8443/openidm/selfservice/reset"
{
 "type": "parameters",
 "tag": "initial",
 "requirements": {
  "$schema": "http://json-schema.org/draft-04/schema#",
  "description": "Parameters",
  "type": "object",
  "properties": {
   "returnParams": {
    "description": "Parameter named 'returnParams'",
    "type": "string"
   }
  }
 }
}
```

2. Initial requirements submission with an empty payload, server returns requirements for the `userQuery` stage, and the JWT:

```
curl \
 --header "X-OpenIDM-Username: anonymous" \
 --header "X-OpenIDM-Password: anonymous" \
 --request POST \
 --data '{
     "input":{}
 }' \
 "https://localhost:8443/openidm/selfservice/reset?_action=submitRequirements"
{
  "type": "userQuery",
  "tag": "initial",
  "requirements": {
    "$schema": "http:\/\/json-schema.org\/draft-04\/schema#",
    "description": "Find your account",
    "type": "object",
    "required": [
      "queryFilter"
    ],
    "properties": {
      "queryFilter": {
        "description": "filter string to find account",
        "type": "string"
      }
    }
  },
  "token": "eyJ0eXAiOiJKV1QiLCJjdHkiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.ZXlKMGVY...W5ywOcr8"
}
```

3. The client provides the requirements for the `userQuery` stage, along with the JWT. The process progresses to the `emailValidation` stage:

```
curl \
 --header "X-OpenIDM-Username: anonymous" \
 --header "X-OpenIDM-Password: anonymous" \
 --request POST \
 --data {
  "token": "eyJ0eXAiOiJKV1QiLCJjdHkiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.ZXlKMGVY...W5ywOcr8",
  "input": {
    "queryFilter": "userName eq \"bjensen\""
  }
} \
 "https://localhost:8443/openidm/selfservice/reset?_action=submitRequirements"
{
  "type": "emailValidation",
  "tag": "validateCode",
  "requirements": {
    "$schema": "http:\/\/json-schema.org\/draft-04\/schema#",
    "description": "Verify emailed code",
    "type": "object",
    "required": [
      "code"
    ],
    "properties": {
      "code": {
        "description": "Enter code emailed",
        "type": "string"
      }
    }
  },
  "token": "eyJ0eXAiOiJKV1QiLCJjdHkiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.ZXlKMGVY...SD7J6dO4"
}
```

The server converts that requirement and token to a URL that is emailed.

4. Clicking the email link sends another POST request to the `emailValidation` stage, along with the token. The process advances to the reset stage and returns its requirements.

5. After email validation, the client submits the new password. The process advances to the reset stage, updates the managed object, and exits:

```
curl \
 --header "X-OpenIDM-Username: anonymous" \
 --header "X-OpenIDM-Password: anonymous" \
 --request POST \
 --data {
  "token": "eyJ0eXAiOiJKV1QiLCJjdHkiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.ZXlKMGVY...rufKZXTVc",
  "input": {
    "password": "Passw0rd"
  }
} \
 "https://localhost:8443/openidm/selfservice/reset?_action=submitRequirements"
{
  "type": "resetStage",
  "tag": "end",
  "status": {
    "success": true
  },
  "additions": {
  }
}
```