Configuration Automation - Terraform

June 11, 2025

CONFIGURATION AUTOMATION - TERRAFORM

Table of Contents

Terraform at Ping Identity

We develop and provide Terraform providers that are used to integrate product administration APIs with the Hashicorp Terraform ecosystem. With Terraform providers, developers can create Configuration as Code packages of Ping Identity product configurations (alongside supported third-party vendor product configurations) that represent a declaration of the intended end-state configuration for an environment.

Terraform enables use cases such as automated configuration promotion through GitOps pipelines, while being able to detect and correct accidental or malicious configuration changes to managed configuration in a change-controlled environment.

Learn more about Terraform at Hashicorp Terraform, or learn more about the benefits of using Terraform at Ping Identity.

Published Terraform providers

View our published Terraform providers on the Terraform registry. The Terraform registry documentation provides a full reference of resources, data sources, and functions.

- PingOne
- PingFederate/PingOne Advanced Services
- PingDirectory

Get started

Begin development with our Terraform providers by first configuring your products to allow Terraform access.

- PingOne
- PingFederate
- PingDirectory

Discover Terraform

Learn what Terraform is, what Terraform providers are, how it's used, and how it can accelerate configuration promotion between environments and protect against accidental or malicious configuration changes.

- What is Terraform
- Benefits of Terraform with Ping Identity deployments

What is Terraform

HashiCorp Terraform is an open-source Infrastructure as Code (IaC) and Configuration as Code (CaC) tool that allows developers to define and provision infrastructure using a declarative configuration language named Hashcorp Configuration Langugage (HCL). It enables developers to manage infrastructure and configuration as code, which can be managed similarly to code a developer writes for an application. This includes using industry standard GitOps methodologies for version control, multi-team collaboration, and automation benefits.

Key use cases

- **Multi-product configuration management**: Provision and manage interdependent configuration across multiple Ping Identity products and third-party service providers.
- **Multi-cloud infrastructure management**: Provision and manage infrastructure across multiple cloud providers, on-premises data centers, and SaaS platforms.
- **Configuration as Code**: Define and manage Ping Identity product configurations in a human-readable and machine-executable format.
- **Use Case Deployment**: Provision the necessary configuration for use case deployment, ensuring consistency and repeatability between environments.
- **Templated Configuration**: Create reusable and sharable configuration templates for product configuration, including policies, application integrations, and use cases.
- **Platform Orchestration:** Manage complex infrastructure and configuration dependencies between Ping Identity products and customer applications and services.

Terraform providers

Providers are plugins that allow Terraform to interact with Ping Identity's products and services. They define the API interactions necessary to create, read, update, and delete resources on those platforms according to the HCL source code that the user defines. For example, the PingOne Terraform provider allows Terraform to manage PingOne configuration resources such as applications, schema attributes, policy configuration, branding, localised translations, and more.

Learn more in Hashicorp Terraform providers.

Terraform state

A key feature of Terraform is the Terraform state. The Terraform state represents a stored record of the configuration successfully applied at the last run of Terraform, where configuration changes were made by Terraform. During execution, Terraform compares the configuration stored in Terraform state with the actual configuration of the live service to determine whether configuration has unexpectedly changed.

If the configuration of the live service has changed relative to the stored state, Terraform can issue configuration changes to realign the live service back to the stored state, effectively correcting any unauthorized or unexpected configuration changes that could lead to service outages or vulnerabilities.

Terraform state can be managed in different ways and should always be kept secure.

Learn more in Hashicorp Terraform state.

Terraform resources

Resources represent the individual components of Ping Identity product configurations. They are the building blocks of a fully configured environment and manage the lifecycle of configuration resources through create, update, and delete actions. Resources typically align with a single API endpoint, and each resource is defined with a schema of fields that reflects that API's request and response payload and is used to specify the desired state. For configuration that a resource manages, the resource will retrieve the current configuration in the live service so that the configuration can be compared against the Terraform state and the developer's HCL. If changes are detected, Terraform will create a plan of action to reconcile and complete the changes.

For example, in the PingOne Terraform provider, the Terraform developer can define a single instance of an application using the `pingone_application` resource. Terraform can then manage the ongoing lifecycle of the application's configuration, including taking corrective actions if the configuration unexpectedly changes.

Learn more in Hashicorp Terraform resources.

Terraform data sources

Data sources allow Terraform to retrieve information about existing, unmanaged configuration or data from external systems. This enables developers to dynamically configure resources based on existing state or external inputs from data sources. Data sources typically align with a single API endpoint, and each data source is typically defined with a schema of fields that reflects that API's response payload and is used to retrieve some configuration state.

Terraform doesn't manage configuration using a data source, as data sources are read only.

For example, in the PingOne Terraform provider, the Terraform developer can read a single instance of an application using the `pingone_application` data source.

Learn more in Hashicorp Terraform data sources.

Using providers, resources, and data sources to meet key use cases

- The Terraform developer selects the appropriate provider for their target platform (for example, PingOne provider for PingOne).
- The Terraform developer then defines resources within their Terraform configuration files, specifying the desired state of each configuration item. A resource's Terraform code can be defined either:
    - Manually using the Terraform Registry documentation as a guide.
    - Export/Generation using Ping Identity's developer tools. Learn more in Exporting configuration.
- Data sources are used to retrieve dynamic information, enabling flexible and adaptable configurations based on outside conditions.
- Terraform uses the selected provider to interact with the target platform's API to create, modify, or delete the defined resources.

Benefits of Terraform with Ping Identity deployments

The following sections describe the benefits of using Terraform with Ping Identity deployments.

Accelerating configuration promotion

Terraform enables developers to define configuration once during development, create a Configuration as Code package, and then deploy that configuration consistently across different environments (development to test, test to production, and so on). By using Terraform’s variables and modules, developers can easily parameterize configuration (use cases, solutions, and applications) and adapt them to each environment’s specific requirements. This approach has the following benefits:

- Reduces the risk of errors and inconsistencies during deployment promotion and accelerates time to value for new features and end user experiences.
- Reduces the time needed to align configuration between environments.
- Enables quick rollback of configuration.

To get best value from accelerating configuration promotion, consider:

- Using GitOps methodology to store Terraform configuration as code in source control.
- Version controlling logical releases to accurately report the version of a solution that has been applied to any environment.
- Define a strong code review and change audit process to allow configuration owners to review and comment on potential changes before they enter (or during) the deployment pipeline.
- Apply testing automation to quickly alert to potential regression issues and gain confidence that configuration changes will be successful when deployed to production.

Accelerating development cycles

Terraform accelerates multi-team development cycles by providing a standardized, code-driven approach to infrastructure and configuration management. Using Configuration as Code, teams can use industry standard GitOps tools, such as a common source control repository or central deployment and testing pipeline, to rapidly merge changes with full audit and approval control. This approach can allow:

- Ephemeral, on-demand environments to prevent multiple teams from conflicting in the same development environment.
- Rapid refreshes of development environments to avoid conflict with experimental or sandbox configurations.
- Clear definitions of ownership and change review, where different teams can be responsible for their own configuration and be notified for review and approval if a team’s configuration is altered by a different team.
- Use cases, applications, and solutions can be modularized to help prevent configuration sprawl.

Protecting against accidental or malicious configuration changes

As an out-of-the-box feature, Terraform compares the configuration that it's stored in Terraform state (from the last successful Terraform operation), the configuration the developer defines in the Configuration as Code HCL, and the configuration at the point in time that is active on the live service. When comparing these three sources, Terraform can determine whether configuration has been modified in the HCL by the developer (the live service should be updated to align with the developer's intention), or whether configuration has been modified in the live service (and not in Terraform HCL or state, meaning the live service has changed unexpectedly) and the live service should be corrected.

If the live service configuration has changed unexpectedly, this change might indicate an accidental or malicious configuration change. In Ping Identity systems, such a change might lead to an issue in the user experience, a weakened IAM security policy, or an untested or unapproved use case.

Consider running Terraform on a regular schedule to alert administrators to potential drift in configuration so that the cause can be investigated. Optionally, as part of the scheduled run, Terraform can be configured to be run to automatically remediate changed configuration to return the environment back to the originally tested state.

Consistent and homogenous Configuration as Code syntax

Terraform provides two styles of Configuration as Code: Hashicorp Configuration Language (HCL) and JSON. For the Terraform community, both HCL and JSON are well understood, and the syntax can be applied to Ping Identity or any other supported third-party system. Both styles of code provide a consistent look and feel and have rich support for well-known code editors.

While both HCL and JSON are human and machine readable, the HCL style can be better organised by the developer to improve readability by:

- Adding lines of comments to add additional context to readers.
- Separating large code files into smaller, discrete files where file names provide clear direction as to the purpose of the contained HCL.
- Separating groups of files or repeated HCL code into clearly defined, well documented modules.
- Use of new line separators to create natural spacing between resources and data sources.

JSON can be used where machine readability (programatic manipulation) might be required over human readability.

Auditability and readability of changes

When using Terraform to perform changes to a live service, Terraform provides a full list of proposed changes during the plan generation phase using the `terraform plan` command.

The full list of changes allows the developer to track:

- What new configuration resources are to be created in the live service (new resources have been added to the code by the developer).
- What existing configuration resources need to be modified (the developer has requested changes in the code, or the live service configuration needs correcting).
- What existing configuration resources should be deleted (resources have been removed from the code by the developer).

This `plan` output provides Ping Identity configuration administrators with an ability to review all the changes to an environment before they are made, allowing teams to understand the impact of proposed changes and potentially stop any accidental changes from being applied to an environment.

The `plan` output also provides a way to record, in audit, what configuration changes have been requested to be made to an environment. When reviewed alongside the `apply` output, auditors can trace configuration changes that have been made to an environment over time.

Best practices

The following provides a set of best practices to apply when writing Terraform with Ping Identity providers and modules in general. This guide is intended to be used alongside provider and service-specific best practices.

These guidelines are not intended to educate on the use of Terraform and are not a getting started guide. You can find more information about Terraform in Hashicorp's Online Documentation. Learn how to get started with the Ping Identity Terraform providers in the Getting Started guides.

General use

`plan` first before `apply`

Running `terraform plan` before `terraform apply` is a crucial practice for Terraform users, as it provides a proactive approach to infrastructure management. The `plan` command generates an execution plan, detailing the changes that Terraform intends to make to the infrastructure. By reviewing this plan, administrators will gain insight into the potential modifications, additions, or deletions of configured resources.

This preview allows administrators to assess the impact of the proposed changes, identify any unexpected alterations, and verify that the configuration aligns with their intentions. This preventive step helps in avoiding unintended consequences and costly mistakes, ensuring a smoother and more controlled deployment process. Skipping the `plan` phase and directly executing `apply` can lead to inadvertent alterations, risking the stability and integrity of the infrastructure. Therefore, incorporating `terraform plan` as an integral part of the workflow, potentially as an automation in the "Pull Request" stage of a GitOps process, promotes responsible and informed infrastructure management practices.

Use `--auto-approve` with caution

Use of the `--auto-approve` feature in Terraform can lead to unintended and potentially destructive changes in your infrastructure. When running Terraform commands, such as `terraform apply`, without the `--auto-approve` flag, Terraform will provide a plan of the changes it intends to make and ask for confirmation before applying those changes.

By using `--auto-approve`, the process of reviewing planned changes to configuration and infrastructure is skipped, and Terraform immediately applies the changes. Using this flag is risky for several reasons:

- **Accidental Changes**: Without reviewing the plan, unintended changes could inadvertently be applied to the environment. The lack of review is particularly dangerous in production environments where mistakes can have significant consequences, such as causing breaking changes creating an outage or use case failure.

- **Destructive Changes**: Similar to accidental changes, Terraform could also destroy resources as part of the update. Without manual confirmation, unintentional removal of critical configuration or infrastructure can occur. This removal of resources applies to both `terraform apply` and `terraform destroy` commands.

- **Security Implications**: Auto-approving changes without verification increases the risk of security vulnerabilities. For example, sensitive data could be exposed unintentionally, or access policies could be negated or weakened.

To minimize the risks associated with `--auto-approve`, Ping Identity recommends you review the Terraform plan before applying changes. This review ensures that admins have a clear understanding of what modifications Terraform intends to make to live service configuration and infrastructure.

Store state securely

When operating production infrastructure with Terraform, the secure storage of Terraform state files is critical. These files serve as the foundational blueprint of your infrastructure, containing detailed configurations, credentials, and the current state of resources. As they contain sensitive information, exposure of these files could be used to gain unauthorized access to user data and allow manipulation of deployed infrastructure.

To safeguard against these threats, it is vital that robust security measures are implemented around state file storage. Such measures include: * Encrypting the state files to protect their contents during transit and at rest. * Employing stringent access controls to ensure only authorized personnel can retrieve or alter the state. If cloud blob storage is used (such as AWS S3), ensure public access is disabled. * Leveraging secure storage solutions that offer features such as versioning and backups.

One option is to use Terraform Cloud from Hashicorp, which provides secure state storage as part of the service.

Regardless of your method of storing the state information, these practices are crucial in maintaining the confidentiality, integrity, and availability of your infrastructure.

Learn more about state management when using Terraform in Hashicorp's online documentation.

Don't modify state directly

Directly modifying Terraform state files is strongly discouraged due to the critical role they play in Terraform's management of infrastructure resources.

The state file acts as a single source of truth for both the configuration and the real-world resources it manages, which is critical when Terraform calculates the differences between "intended" and "actual" configuration when running `terraform plan`.

Manual edits can lead to inconsistencies between the actual state of your infrastructure and Terraform's record, potentially causing unresolvable conflicts in the plan phase, or could result in errors when the provider is reading/writing the state of a resource. Such actions undermine the integrity of your infrastructure management, leading to difficult-to-diagnose issues, resource drift, and potentially the loss or corruption of critical infrastructure.

Instead of directly editing state files, it is best practice to use Terraform's built-in commands, such as `terraform state rm` or `terraform import`, to safely make changes. This approach ensures that Terraform can accurately track and manage the state of provisioned infrastructure, maintaining the reliability and predictability of your Infrastructure as Code environment.

Learn more about state management when using Terraform in Hashicorp's online documentation.

Ensure provider warnings are captured and reviewed

Terraform providers can produce warnings as a result of operations such as `terraform validate`, `terraform plan`, and `terraform apply`. When these operations are run using the CLI, the warnings are directed to the command line output, and when these operations are run in cloud services such as Terraform Cloud, these warnings are shown in the UI.

Ping Identity's Terraform providers can show warnings that need to be captured and reviewed. For example, the PingOne Terraform provider will produce warnings when specific configuration is used that remove guardrails to prevent accidental deletion of data.

It is highly recommended that warnings shown on the `terraform plan` stage especially are captured and reviewed before the `terraform apply` stage is run, as the messages might alert the administrator to potential undesired results of the `terraform apply` stage.

HCL writing recommendations

Use Terraform formatting tools

When writing Terraform HCL, using `terraform fmt` is a straightforward yet powerful practice. `terraform fmt` and equivalent formatting tools adjust the Terraform code to a standard style, keeping the codebase tidy and consistent. Typically, this formatting deals with maintaining consistent indentation, spacing, and alignment of code. If developing in Visual Studio Code, the "Hashicorp Terraform" extension can be set to run `terraform fmt` automatically as you write and save configuration.

This consistency makes your code easier to read and understand for anyone who might work on the project. Having consistent formatting reduces confusion and makes it easier to spot mistakes.

It is recommended to include `terraform fmt` into the development workflows as it has a big impact on the maintainability and clarity of your infrastructure code. The benefits in code quality and collaboration outweigh the minimal effort required to format everything consistently.

Additionally, it is recommended to include `terraform fmt` as a CI/CD validation check to ensure developers are applying consistent development practices when committing Configuration as Code to a common CI/CD pipeline code repository.

Validate Terraform HCL before `plan` and `apply`

When writing Terraform HCL, it is recommended to use `terraform validate` before running `terraform plan` and `terraform apply`.

This command serves as a preliminary check, verifying that Terraform HCL configurations are syntactically valid and internally consistent without actually applying any changes. There are some resources in Ping Identity's Terraform providers that have specific validation logic to ensure that configuration is valid before any platform API is called, which reduces the "time-to-error", if an error exists.

As a specific example, `davinci_flow` resources validate the `flow_json` input and the specified `connection_link` blocks to make sure remapped connections are valid.

Using `count` and `for_each` with resource iteration

When writing Terraform HCL, there are considerations around when to use `count` and when to use `for_each`, especially when iterating over resources. Using the incorrect iteration method could result in accidental or unnecessary destruction or recreation of resources as the data to iterate over changes.

Consider the following example, where a number of populations are being created from an array variable:

```
locals {
  populations = [
    "Retail Customers",
    "Business Customers",
    "Business Partners",
  ]
}

resource "pingone_population" "my_populations" {
  count = length(local.populations)

  environment_id = pingone_environment.my_environment.id
  name           = local.populations[count.index]
}
```

The HCL will create the populations successfully, but you will run into problems when the order of the array changes (for example, if it is sorted alphabetically in the code):

```
locals {
  populations = [
    "Business Customers",
    "Business Partners",
    "Retail Customers",
  ]
}

resource "pingone_population" "my_populations" {
  count = length(local.populations)

  environment_id = pingone_environment.my_environment.id
  name           = local.populations[count.index]
}
```

```
Terraform will perform the following actions:

  # pingone_population.my_populations[0] will be updated in-place
  ~ resource "pingone_population" "my_populations" {
        id             = "91ffa912-e24e-4fa7-a0f3-7fb48539f756"
      ~ name           = "Retail Customers" -> "Business Customers"
        # (1 unchanged attribute hidden)
    }

  # pingone_population.my_populations[1] will be updated in-place
  ~ resource "pingone_population" "my_populations" {
        id             = "f2df301c-c2a1-436b-afaf-33eb189fe7d6"
      ~ name           = "Business Customers" -> "Business Partners"
        # (1 unchanged attribute hidden)
    }

  # pingone_population.my_populations[2] will be updated in-place
  ~ resource "pingone_population" "my_populations" {
        id             = "f2df828e-cfd6-4ecb-815d-5bd33c566fa8"
      ~ name           = "Business Partners" -> "Retail Customers"
        # (1 unchanged attribute hidden)
    }
```

In this situation, users are inadvertently being moved from one population to another based on the names of the populations. Any downstream application that requires a hardcoded UUID for "Retail Customers" (for example) will instead return "Business Partners" identities.

The problem is compounded when adding and removing elements to and from the array. This scenario is an example of when to use `for_each` instead of `count`, as `for_each` will identify and store each resource with a unique key. Including guidance from the Use maps with static keys when using `for_each` on resources best practice, the following HCL is the recommended way to perform the same iteration:

```
locals {
  populations = {
    "business_customers" = "Business Customers",
    "retail_customers"   = "Retail Customers",
    "business_partners"  = "Business Partners",
  }
}

resource "pingone_population" "my_populations" {
  for_each = local.populations

  environment_id = pingone_environment.my_environment.id
  name           = each.value
}
```

Use maps with static keys when using `for_each` on resources

When writing Terraform HCL, there are considerations around the use of `for_each` when iterating over objects or maps to manage resources. Using a variable key could result in accidental or unnecessary destruction or recreation of resources as the data to iterate over changes. Ping Identity recommends using static keys and maps of objects when using `for_each` rather than a list or array of objects to control resource creation.

When Terraform creates and stores resources in state, iterated resources must be stored with a defined "key" value that uniquely identifies the resource against others.

Best practice

It is a best practice to use a map of objects, where there is a static key. Notice that `first_population` and `second_population` are both static keys for the objects you want to create:

```
variable "populations" {
  type = map(object({
    name        = string
    description = optional(string)
  }))

  default = {
    "first_population" = {
      name        = "My awesome population"
      description = "My awesome population for awesome people"
    },
    "second_population" = {
      name = "My awesome second population"
    }
  }
}

resource "pingone_population" "my_awesome_population_map_of_objects" {
  environment_id = pingone_environment.my_environment.id

  for_each = var.populations

  name        = each.value.name
  description = each.value.description
}
```

This results in creation of two unique resources:

- `pingone_population.my_awesome_population_map_of_objects["first_population"]`
- `pingone_population.my_awesome_population_map_of_objects["second_population"]`

In this case, if the `name` or `description` of any population changes, Terraform will correctly update the impacted resource, rather than potentially forcing a recreation.

Additionally, if the order of the key-object pairs changes in the map, Terraform correctly calculates that there are no changes to the data with the objects themselves because the relation of object to map key hasn't changed. This has similar advantages to using `for_each` over `count`, where changing the order of items does impact the plan that Terraform calculates because the counted index related to the data has changed.

Not best practice

The following example does not follow best practice where creation of multiple populations might use `for_each` over a *list* of objects. In this example, you might be inclined to use the `name` as the population's key, which can introduce functional issues and introduce security vulnerabilities for integrated production environments:

```
variable "populations" {
  type = list(object({
    name        = string
    description = optional(string)
  }))

  default = [
    {
      name        = "My awesome population"
      description = "My awesome population for awesome people"
    },
    {
      name = "My awesome second population"
    }
  ]
}

resource "pingone_population" "my_awesome_population_list_of_objects" {
  environment_id = pingone_environment.my_environment.id

  for_each = { for population in var.populations : population.name => population }

  name        = each.key
  description = each.value.description
}
```

The above results in creation of two unique resources:

- `pingone_population.my_awesome_population_list_of_objects["My awesome population"]`
- `pingone_population.my_awesome_population_list_of_objects["My awesome second population"]`

However, in this case, if the name of `My awesome population` is changed to `My awesome first population` in the variable, Terraform will destroy that population and recreate it with a new index value. This an unnecessary and dangerous way to change the population name, as destruction of populations will put user data at risk.

Write and publish reusable modules

When writing Terraform HCL, there are many cases when collections of resources and data sources are commonly used together or with the same or a similar structure. These collections of resources and data sources can be grouped together into a Terraform module. Writing and publishing Terraform modules embodies a best practice within Infrastructure as Code paradigms for several reasons:

- **Abstract complexity**: Modules encapsulate and abstract complex sets of resources and configurations, promoting reusability and reducing redundancy across your infrastructure setups. This modular approach enables teams to define standardized and vetted building blocks, ensuring consistency, compliance, and reliability across deployments.

- **Foster collaboration**: Publishing these modules, either internally within an organization or publicly in the Terraform Registry, fosters collaboration and knowledge sharing. It allows others to benefit from proven infrastructure patterns, contribute improvements, and stay aligned with the latest best practices. This culture of sharing and collaboration not only accelerates development cycles but also elevates the quality of infrastructure provisioning by leveraging the collective expertise and experience of the Terraform community.

Versioning

Use Terraform version control

Terraform releases change over time, which can include new features and bug fixes. Major version changes can introduce breaking changes to written code.

To ensure that Terraform HCL is run with consistent results between runs, it is recommended to restrict the version of Terraform in the `terraform {}` block with a lower version limit (in case the HCL includes syntax introduced in a specific version) and an upper version limit to protect against breaking changes.

For example:

```
terraform {
  required_version = ">= 1.3.0, < 2.0.0"

  # ... other configuration parameters
}
```

Another example that limits to a specific minor version:

```
terraform {
  required_version = "~> 1.6"

  # ... other configuration parameters
}
```

Terraform Documentation Reference

Use provider version control

Ping Identity and other vendors release changes to providers on a regular basis that can include new features and bug fixes. Major version changes can introduce breaking changes to written code as older deprecated resources, data sources, parameters, and attributes are removed. Provider versions that are less than 1.0.0 could also include breaking changes to written code. Ping Identity follows guidance issued by Hashicorp on deprecations, removals, and renames.

To ensure consistent results between iterations, it is recommended to restrict the version of each provider in the `terraform.required_providers` parameter with a lower version limit (in case the HCL includes syntax introduced in a specific version) and an upper version limit to protect against breaking changes.

For example, the following syntax for the `hashicorp/kubernetes` and `pingidentity/pingdirectory` providers is recommended for provider versions `>= 1.0.0`:

```
terraform {
  required_version = ">= 1.3.0, < 2.0.0"

  required_providers {
    kubernetes = {
      source  = "hashicorp/kubernetes"
      version = ">= 2.25.2, < 3.0.0"
    }
    pingdirectory = {
      source  = "pingidentity/pingdirectory"
      version = ">= 1.0.2, < 2.0.0"
    }
  }
}
```

The following example syntax for the `pingidentity/pingone` and `hashicorp/time` providers shows the recommended version pinning for provider versions `< 1.0.0` that could incur breaking changes during initial development, though it can also be used for provider versions `>= 1.0.0`:

```
terraform {
  required_version = "~> 1.6"

  required_providers {
    pingone = {
      source  = "pingidentity/pingone"
      version = "~> 1.0"
    }
    time = {
      source  = "hashicorp/time"
      version = "~> 0.9"
    }
  }
}
```

Terraform Documentation Reference

Use module version control

Ping Identity and other vendors release changes to modules on a regular basis that can include new features and bug fixes. Major version changes can introduce breaking changes to written code as older deprecated resources, data sources, parameters, and attributes are removed.

To ensure consistent results between iterations, it is recommended to restrict the version of each module with a lower version limit (in case the HCL includes syntax introduced in a specific version) and an upper version limit to protect against breaking changes.

For example, the following syntax for the `terraform-aws-modules/vpc/aws` module is recommended for module versions `>= 1.0.0`:

```
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = ">= 5.5.1, < 6.0.0"

  # ... other configuration parameters
}
```

The following example syntax for the `pingidentity/utils/pingone` module shows the recommended version pinning for module versions `< 1.0.0` that could incur breaking changes during initial development, though it can also be used for module versions `>= 1.0.0`:

```
module "utils" {
  source  = "pingidentity/utils/pingone"
  version = "~> 0.1"

  # ... other configuration parameters
}
```

Terraform Documentation Reference

Protect service configuration and data

Protect configuration and data with the `lifecycle.prevent_destroy` meta argument

While some resources are safe to remove and replace, there are some resources that, if removed, can result in data loss.

It is recommended to use the `lifecycle.prevent_destroy` meta argument to protect against accidental destroy plans that might cause data to be lost. You might also want to use the meta argument to prevent accidental removal of access policies and applications if dependent applications cannot be updated with Terraform in case of replacement.

For example:

```
resource "pingone_schema_attribute" "my_attribute" {
  environment_id = pingone_environment.my_environment.id

  name = "myAttribute"

  # ... other configuration parameters

  lifecycle {
    prevent_destroy = true
  }
}
```

Terraform Prevent Destroy Documentation

Secrets management

Use of Terraform variables and secrets management

When writing Terraform HCL, it might be tempting to write values that are sensitive (such as passwords, API keys, tokens, OpenID Connect Client Secrets, or TLS private key data) directly into the code. There is a significant risk that these secrets are then committed to source control, where they can be viewed by anyone who can access that code. This risk is elevated when the source control is a public Git repository hosted on sites such as GitHub or GitLab. After secrets are committed to a repository, removing them requires extensive effort and does not guarantee that they have not been copied or logged elsewhere.

In addition, version control systems are designed to track and preserve history, making it challenging to completely erase secrets after they are committed. This persistence in history means that even if the secrets are later removed from the codebase, they remain accessible in the commit history. Additionally repositories are often cloned, forked, or integrated with third-party services, further increasing the exposure of secrets.

In the end, if credentials have been leaked in this manner, the safest way to recover is to rotate them in the source systems, an activity which can have broad impact across systems and individuals.

To mitigate these risks, it is recommended to use secure secrets management tools and practices. Terraform supports various mechanisms for securely managing secrets, including environment variables, encrypted state files, and integration with dedicated secrets management systems like AWS Secrets Manager, Azure Key Vault, or HashiCorp Vault. These tools provide controlled access to secrets, audit trails, and the ability to rotate secrets periodically or in response to a breach.

By keeping secrets out of source control and employing robust secrets management strategies, users can significantly enhance the security posture of their infrastructure deployments. This approach not only protects sensitive information but also aligns with compliance requirements and best practices for secure infrastructure management.

Multi-team development

Use on-demand development environments

When using a GitOps CI/CD promotion process across multiple teams or individuals, a recommended approach is to spin up on-demand development and test environments where possible. These environments can be specific to new features or to individual teams to allow for development and integration testing that doesn't conflict with other team's activities. The Terraform provider allows administrators to use CI/CD automation to provision new environments as required and remove them after the project activity no longer needs them.

In a GitOps CI/CD promotion pipeline, configuration can be translated to Terraform Configuration as Code and then merged (with Pull Requests) with common test environments, where automated tests can be run. Doing so allows the activities in the on-demand environments to be merged into a common promotion pipeline to production environments.

In some cases, some integrated systems are not easily available for integration testing. For example, integrated HR systems or systems are installed on bare metal infrastructure. In these cases, where possible, these systems can be stubbed into the process and tested during the integration testing phase of the project when changes have been merged into a common promotion pipeline.

In some cases, it is not practical or possible to use on-demand environments. In these situations, one option is to create static development environments that are shared between teams or projects, along with processes to mitigate conflicts. Ideally these development environments will have their configuration periodically refreshed and aligned with that of common test environments further down the CI/CD promotion pipeline. When using a shared environment, ensure the update activity is appropriately scheduled with the project teams involved to avoid wiping configuration that is still in active development.

Continuous Integration/Continuous Delivery (CI/CD)

Use Terraform linting tools

Ping Identity recommends using linting tools in the development process, as these tools significantly enhance code quality, maintainability, and consistency across projects.

Linters are static code analysis tools designed to inspect code for potential errors, stylistic discrepancies, and deviations from established coding standards and best practices. By integrating linting tools into the development workflow, developers are proactively alerted to issues such as syntax errors, potential bugs, and security vulnerabilities before the code is even executed or deployed. This immediate feedback loop not only saves time and resources by catching issues early but also facilitates a learning environment where developers can gradually adopt best coding practices and improve their skills.

Moreover, linting tools play a pivotal role in maintaining codebase consistency, especially in collaborative environments where multiple developers contribute to the same project. They enforce a uniform coding style and standards, reducing the cognitive load on developers who need to understand and work with each other’s code. This standardization is vital for code readability, reducing the complexity of code reviews and easing the onboarding of new team members.

Furthermore, integrating linting tools into CI/CD pipelines automates the process of code quality checks, ensuring that only code that meets the defined quality criteria is advanced through the stages of development, testing, and deployment. This automation not only streamlines the development process but also aligns with agile practices and DevOps methodologies, promoting faster, more reliable, and higher-quality software releases.

One of the most common and full-featured linting tools is TFLint.

Use Terraform security scanning tools

Ping Identity recommends that users incorporate Terraform security scanning tools into the development and deployment workflow to help with security and compliance of Infrastructure as Code configurations.

Terraform manages highly sensitive and critical components of cloud infrastructure, making any misconfigurations or vulnerabilities potentially disastrous in terms of security breaches, data leaks, and compliance violations. Security scanning tools are designed to automatically inspect Terraform code for such issues before the infrastructure is provisioned or updated, highlighting practices that could lead to security weaknesses, such as overly permissive access controls, unencrypted data storage, or exposure of sensitive information.

By leveraging these tools, developers can preemptively identify and rectify security vulnerabilities within their infrastructure code, significantly reducing the risk of attacks and breaches. This proactive approach to security is aligned with the principles of DevSecOps, which advocates for "shifting left" on security, in other words, integrating security practices early in the software development lifecycle. It ensures that security considerations are embedded in the development process.

Furthermore, Terraform security scanning tools often provide compliance checks against common regulatory standards and best practices, such as the CIS benchmarks, making it easier for organizations to adhere to industry regulations and avoid penalties. These tools also promote a culture of security awareness among developers, educating them on secure coding practices and the importance of infrastructure security.

Overall, the use of Terraform security scanning tools enhances the security posture of cloud environments, protects against the financial and reputational damage associated with security incidents, and ensures continuous compliance with evolving regulatory requirements. This makes them an indispensable asset in the toolkit of any team working with Terraform and cloud infrastructure.

Example tools for security scanning include Trivy, Terrascan and checkov.

Check the `.terraform.lock.hcl` file into source control

Including the `.terraform.lock.hcl` file in source control is a recommended best practice for Terraform users, providing several benefits to the Infrastructure as Code workflow.

This file serves as a version lock file that records the specific versions of the provider plugins and modules (and their hashes) used in a Terraform configuration. By checking it into source control, teams ensure consistent and reproducible deployments across different environments. The lock file acts as a snapshot of the dependencies, guaranteeing that everyone working on the project has the same set of provider and module versions. This practice enhances collaboration, reduces the likelihood of version mismatches, and mitigates the risk of unexpected changes or disruptions during deployments. It also facilitates version tracking and simplifies the process of recreating the infrastructure at a later time. Overall, checking the `.terraform.lock.hcl` file into source control contributes to the reliability and maintainability of Terraform configurations within a collaborative development environment.

When used with a GitOps process that includes dependency scanning tools (such as GitHub’s Dependabot or Renovate), automations can be configured to generate automatic pull requests of provider or module version updates that might include bug fixes, enhancements, and security patches. The automated pull requests and associated checks can help streamline a CI/CD workflow, leading to higher productivity and reduced human error.

Develop with Terraform

Importing to Terraform state

Learn how to take a preconfigured product environment and import to Terraform state. Importing to Terraform state allows Terraform to manage a product environment without needing to recreate any configuration for that environment. This is useful when bringing a production environment under Terraform control retrospectively.

Importing to Terraform state

Exporting and generating Terraform HCL

Learn how to generate Terraform HCL configuration for a preconfigured environment. This is useful when exporting configuration from a development environment to store in source control, or promote to test or production.

Exporting Terraform configuration

Interface stability

Learn how Ping Identity's development practices provide stability and predictability when using Ping Identity's Terraform providers.

Interface stability

Getting support

Learn where and how to get support on Ping Identity's Terraform providers.

Getting support

Importing to Terraform state

Terraform maintains a storage of the last known configuration of a product environment. This is to allow Terraform to compare the **intended** configuration of an environment (declared using the Terraform code language, HCL), with the actual configuration of the environment (which the provider will do when reading the current configuration of an environment using the service API), with what Terraform believes is the last known configuration of an environment. Terraform compares these three sources to reconcile configuration and produce an action plan to fix configuration drift in an environment, while simultaneously handling configuration additions and deletions.

Terraform state is a required component of Terraform management of an environment and must be handled securely. The storage mechanism of the Terraform state is left to the customer to decide. You can find more information and HashiCorp's own best practices when handling Terraform state in State in the HashiCorp documentation.

You should not manage Terraform state configuration manually using a text editor. Terraform manages the state file itself in two ways:

1. By declaring end-state configuration in Terraform HCL and running the `terraform plan` and `terraform apply` commands. This method is most common when adding net-new configuration to a product, meaning that Terraform manages the full lifecycle of a configuration item including its initial creation.
2. By importing predefined configuration in an environment, using `import {}` block HCL code or using the `terraform import` command. This method is most common when bringing existing product configuration under Terraform control, where Terraform cannot manage the initial creation of that configuration.

Importing configuration is most suited to production environments that have been previously built and managed without using Terraform. Importing configuration to Terraform state is non-destructive (meaning that configuration is not expected to be removed or re-added). After configuration for an environment is imported to Terraform state, Terraform can manage the lifecycle of the imported configuration in the normal way.

Using Ping CLI to generate import blocks

Terraform's out-of-the-box import capability requires the developer to import each configuration item individually by its unique ID. When working with Ping Identity products and Ping Identity Terraform providers, the required ID used for importing a specific configuration item could be a single ID or a compound ID (where two or more IDs are concatenated together with a `/` forward slash separator). Depending on the Ping Identity product, the ID could be customer developer defined or could be a platform generated UUID. In both cases, the customer developer might have difficulty in retrieving the required IDs for all configuration for a product environment.

The Ping CLI command-line tool has features to simplify importing configuration to Terraform state. The `platform export` command is designed to connect to a supported Ping Identity product, read the live configuration of the service, and generate the required Terraform HCL files with clearly labelled `import {}` blocks, complete with all necessary IDs. The developer can then import all configuration for an environment or choose which configuration items to include in import.

Exporting Terraform configuration

When working with Terraform, you should configure use cases in the Ping Identity administration console (typically in the development environment) and then export that configuration to Terraform HCL to promote it to your test and production environments.

Terraform includes experimental out-of-the-box functionality to export or generate configuration from a configured environment. You can find more details in the Terraform documentation.

Using Ping CLI to generate Terraform configuration

Terraform's out-of-the-box capability depends on the `import {}` block capability described in Importing to Terraform state.

Ping CLI has features to simplify generating Terraform HCL code for an environment. The `platform export` command is designed to connect to a supported Ping Identity product, read the live configuration of the service, and generate the required Terraform HCL files with clearly labelled `import {}` blocks, complete with all necessary IDs. The developer can then use the Generate Terraform Configuration feature to generate the full HCL for an environment. The developer can then choose which configuration items to include their Terraform HCL project.

Interface stability

To provide predictability and stability when developing Terraform HCL, Ping Identity's development practices align with development guidelines set out by Hashicorp and follow the expectations of the Hashicorp community.

When developing Terraform providers, Ping Identity follows the SemVer (Semantic Versioning) methodology set out at Semver.org.

This includes a regular cadence of major, minor and patch versions of each Terraform provider. Releases are issued on the respective GitHub code repositories using GitHub's Releases feature. Changelogs are compiled on each release that provide a list of issues resolved, enhancements, new features, general updates, and any breaking changes that might cause HCL developers to alter the Terraform code in order to use the update.

High-level descriptions of the release types are:

- Major releases: Typically once every year or couple of years. Major releases are issued when significant changes are made to the Terraform provider and typically require customers to change their HCL code, otherwise known as breaking changes.
- Minor releases: Typically on a planned schedule, such as every few weeks or aligned with product releases. Minor releases are issued when the providers are enhanced with additional, optional functionality and can also include planned bug fixes.
- Patch releases: Typically ad-hoc, patch releases are issued between minor releases where bug fixes or documentation updates must be released before the next minor release.

Learn more about major, minor, and patch releases at Semver.org

Breaking changes

Breaking changes to a Terraform provider typically require the customer Terraform developer to make changes to the written HCL before they can use the update.

Breaking changes are required periodically to ensure Ping Identity's Terraform providers remain aligned with the product API. Significant breaking changes are typically planned in advance to be released in-bulk in the major release cycle.

As part of Ping Identity's development practices, breaking changes are kept to a minimum and are planned as technical debt in the major release cycle. It's uncommon for breaking changes to occur in minor and patch releases.

Occasionally, breaking changes cannot be included in the major release cycle and must be included in a minor release to ensure that the provider functions correctly. These breaking changes are marked clearly in the release notes.

Some examples of breaking changes are: * Scheduled removal of previously deprecated functionality. * Renaming resources, data sources, or functions without following a deprecation path. * Renaming or removing schema fields without following a deprecation path. * Changing an Optional field to be Required in a resource or data source schema.

When breaking changes are made to a provider, these are highlighted in the release notes. If the change requires significant rework of HCL code, or the corrective action cannot be included in the release note, or if there are many breaking changes to be handled, then a specific guide might be created and published on the Terraform registry to assist with the conversion process. Upgrade and migration guides are typically created when major releases are issued.

Getting support

Ping Identity's Terraform providers are delivered as open-source projects and can be found on Ping Identity's GitHub account according to Hashicorp's Terraform provider publishing guidelines. There are multiple ways to get support when using Ping Identity's Terraform providers:

- Community Forum: The DevOps developer forum provides a quick and simple way to interact with an active community of subject matter experts, including Ping Identity technical staff, customer, and partner specialists. The community forum is ideal for specific questions on how best to configure Ping Identity products using Terraform HCL to meet use cases.
- GitHub project issues: Each Terraform provider has a GitHub repository where open source provider code is published, and each repository has the GitHub issues feature enabled. With a GitHub account, anyone can create issues on the relevant project by filling in the provided template. Using GitHub project issues is ideal when requesting new features, enhancements, or reporting unexpected errors or provider bugs. The GitHub issues are closely monitored and are a quick and easy way to interact directly with the project team.
- Ping Identity Support: For customers with an active product license, support tickets can be raised according to the normal Ping Identity Support process. Raising a support ticket is ideal when there is a potential issue with the underlying product, or the issue should be tracked with the account team.

Getting started

The following provides guidance on preparing a PingOne tenant for Terraform access.

Requirements

- Terraform CLI 1.4+
- A licensed or trial PingOne cloud subscription. Try Ping here.
- Administrator access to the PingOne admin console.

PingOne subscription or trial

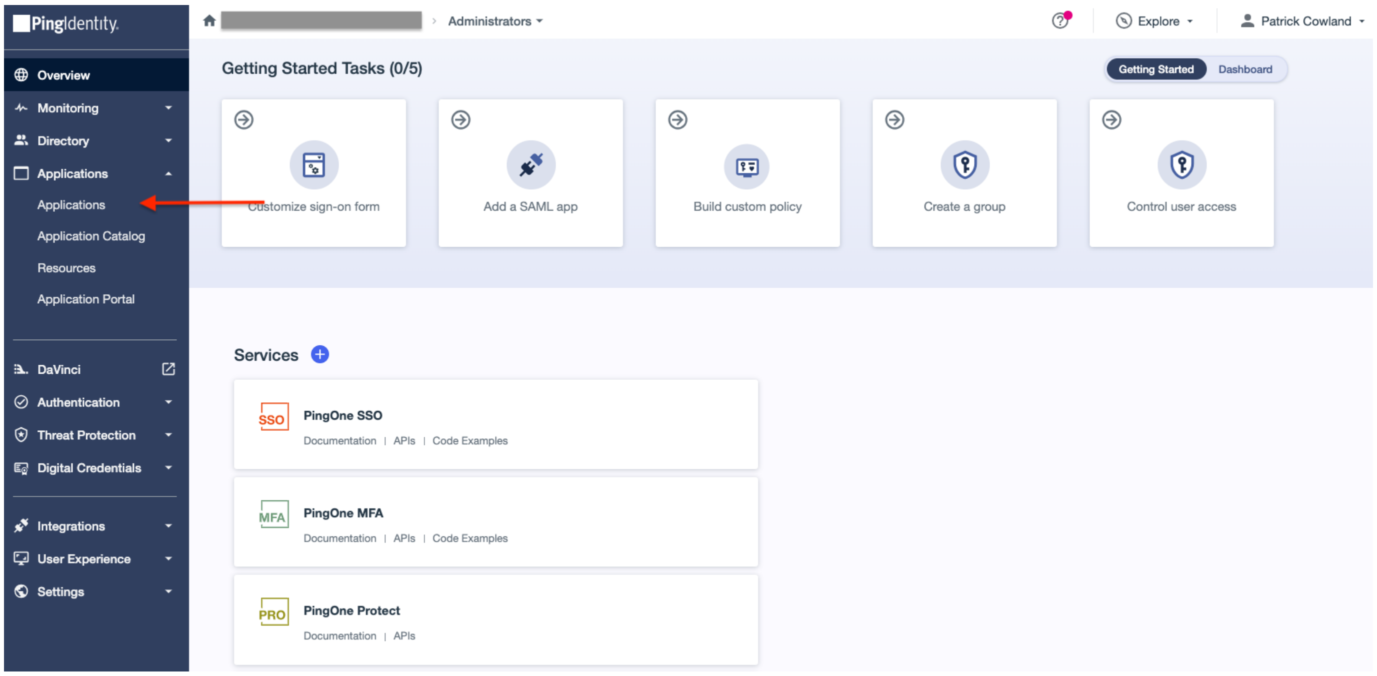
To get started using the PingOne Terraform provider, you must first have an active PingOne cloud subscription. Get instant access with a PingOne trial account or read more about Ping Identity at pingidentity.com

Configure PingOne for Terraform access

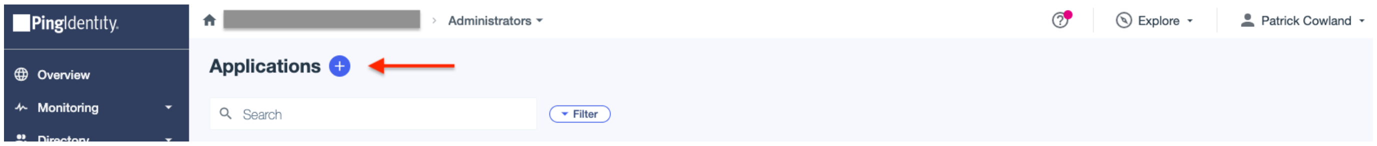
The PingOne Terraform provider requires the ability to connect to the PingOne Management APIs through the use of a worker application that has administrative roles assigned.

The following steps describe how to connect Terraform to your PingOne instance using a worker application:

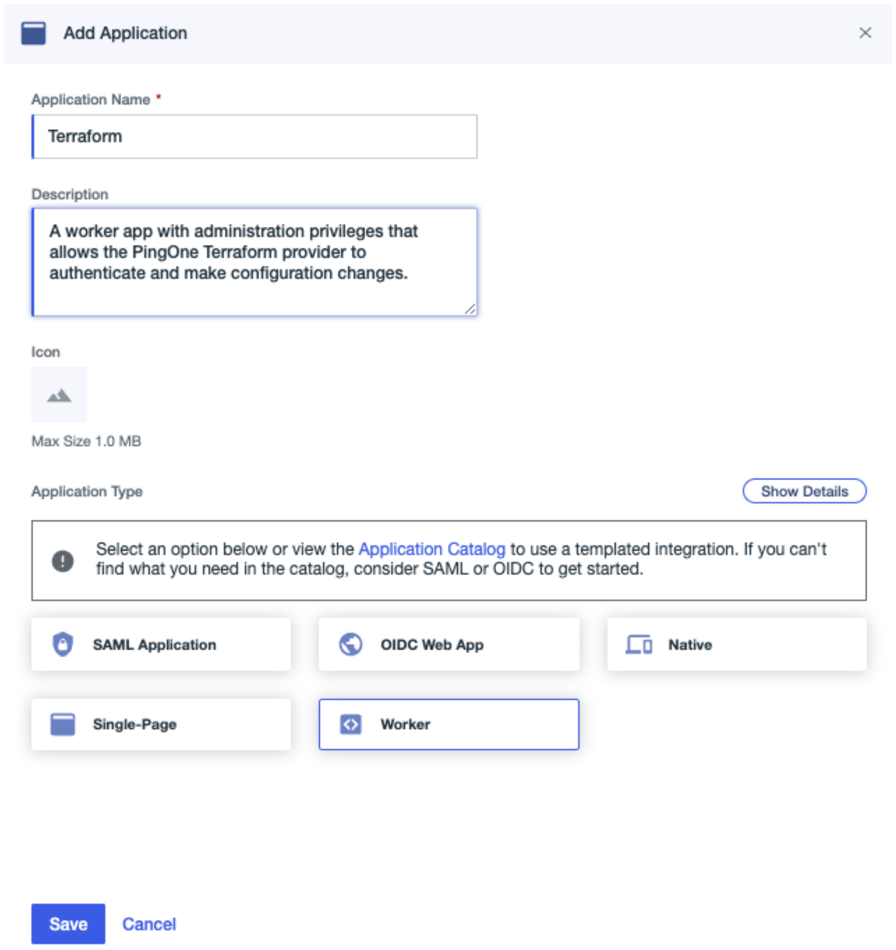
1. Sign on to your PingOne admin console. When you register for a trial, a link will be sent to your provided email address.
2. Open the **Administrators** environment. Note that any environment can be used.
3. Go to **Applications**.



4. Click the **+** icon to add a new application.



5. Enter a name and an optional description, and ensure that **Worker** is selected as the application type.



Application Name *

Terraform

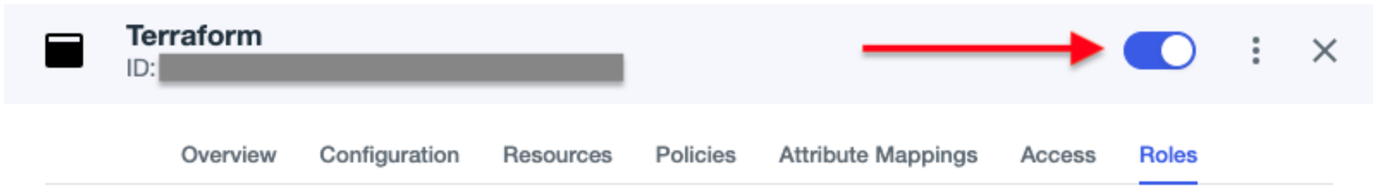
Description

A worker app with administration privileges that allows the PingOne Terraform provider to authenticate and make configuration changes.

Application Type

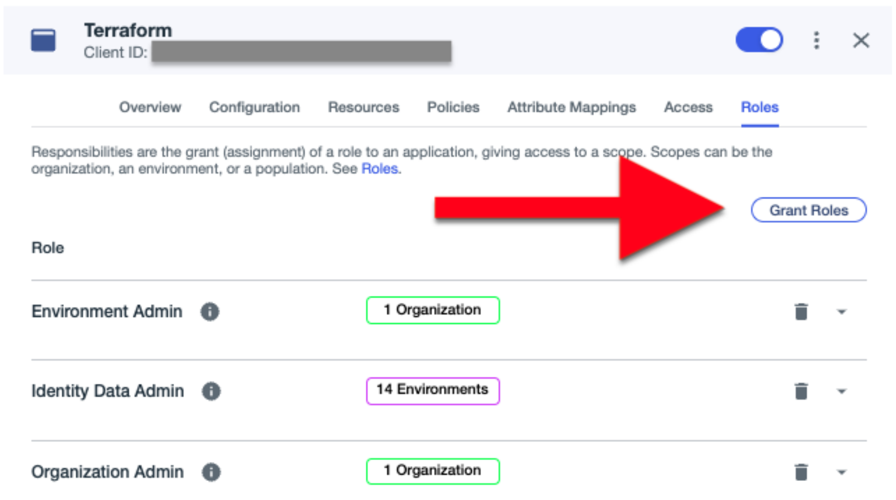
Select an option below or view the Application Catalog to use a templated integration. If you can't find what you need in the catalog, consider SAML or OIDC to get started.

6. Click the toggle to enable the application.



7. On the **Roles** tab, set the administrative roles accordingly.

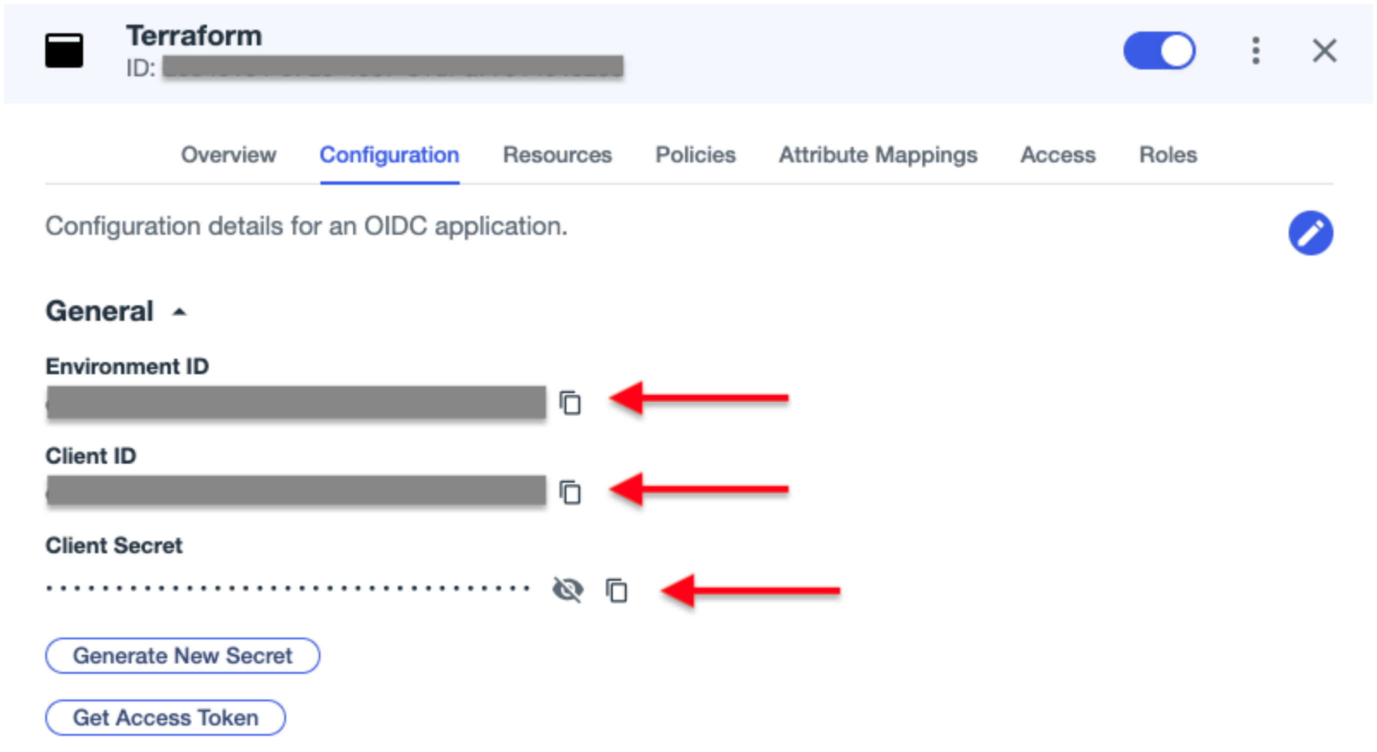
The following image shows example roles to be able to create and manage environments and their configurations. You can find more information about role permissions in Administrator Roles in the PingOne documentation.


Terraform
Client ID: [redacted]

Responsibilities are the grant (assignment) of a role to an application, giving access to a scope. Scopes can be the organization, an environment, or a population. See Roles.

Role	
Environment Admin	1 Organization
Identity Data Admin	14 Environments
Organization Admin	1 Organization

8. On the **Configuration** tab, expand the **General** section and copy the **Client ID**, **Client Secret**, and **Environment ID** values. These IDs are used to authenticate the provider to your PingOne tenant.



Configuration details for an OIDC application.

9. You can find the steps to configure the PingOne Terraform provider using these values in the Terraform Registry provider documentation.

Finding required IDs

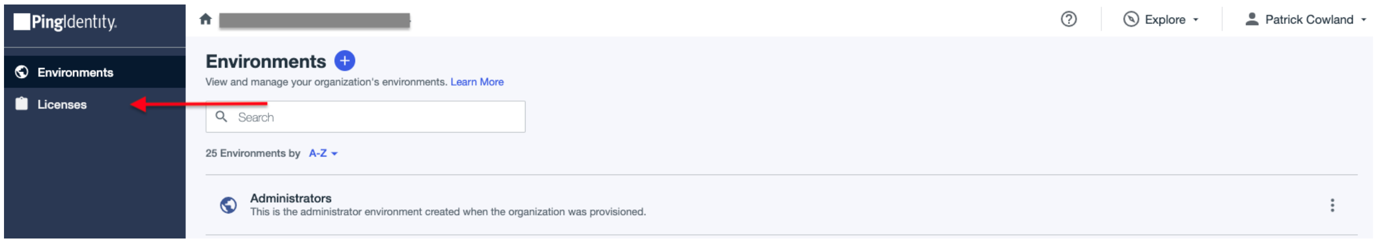
There are tenant specific, unique IDs and name values that are required for the provider to operate. The following sections show how to retrieve the relevant IDs.

License ID, organization ID, and organization name

The license ID is required when creating an environment using the `pingone_environment` resource.

The organization ID and organization name can be used with the `pingone_organization` data source. These values can be found with the following steps:

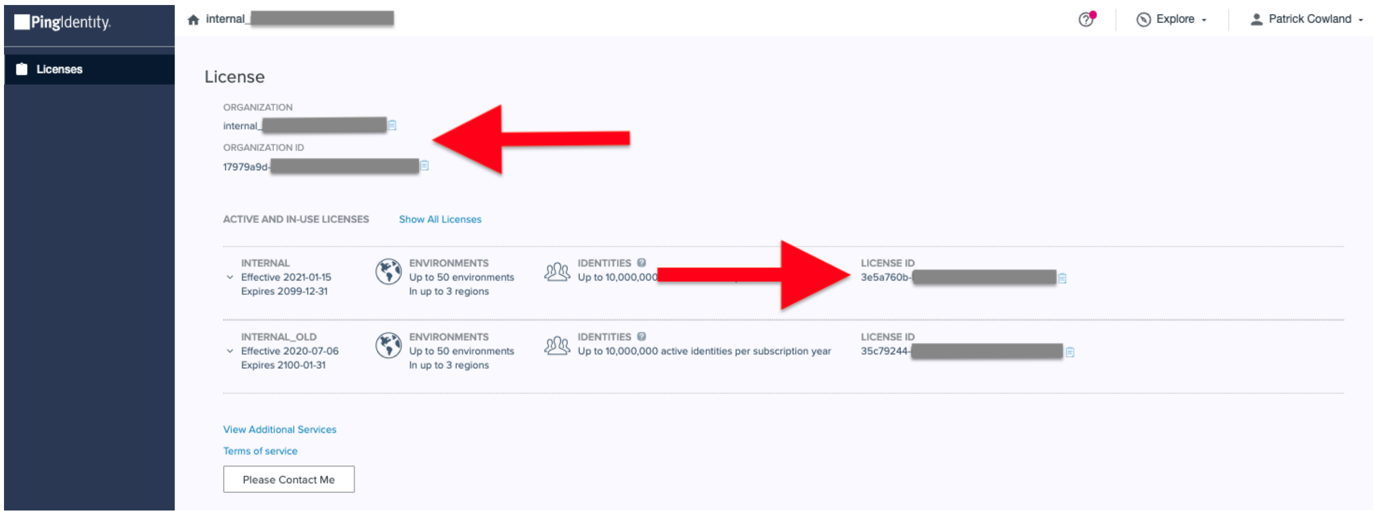
1. Sign on to the PingOne admin console using your unique console link.
2. Go to **Licenses**.



View and manage your organization's environments. Learn More

This is the administrator environment created when the organization was provisioned.

3. Look for the relevant license (that's not an Admin license) and click the **Copy link** icon to copy the ID. The organization name and organization ID are also shown and can be copied.



internal_[redacted]

License

ORGANIZATION

internal_[redacted]

ORGANIZATION ID

17979a9d-[redacted]

ACTIVE AND IN-USE LICENSES Show All Licenses

INTERNAL Effective 2021-01-15 Expires 2099-12-31	ENVIRONMENTS Up to 50 environments In up to 3 regions	IDENTITIES Up to 10,000,000	LICENSE ID 3e5a760b-[redacted]
INTERNAL_OLD Effective 2020-07-06 Expires 2100-01-31	ENVIRONMENTS Up to 50 environments In up to 3 regions	IDENTITIES Up to 10,000,000 active identities per subscription year	LICENSE ID 35c79244-[redacted]

View Additional Services

Terms of service

Please Contact Me

Best practices

The following sections provide a set of best practices to apply when writing Terraform with the PingOne Terraform provider and associated modules.

These guidelines are not intended to educate on the use of Terraform and are not a getting started guide. You can find more information about Terraform in Hashicorp's Online Documentation. Learn how to get started with the PingOne Terraform provider in the Getting started guide.

Develop in the admin console, promote using Configuration as Code

Ping Identity recommends performing use-case development activities in the PingOne admin console whenever possible. This recommendation is due to the complex nature of Workforce IAM and Customer IAM deployments that includes policy definition, user experience design, and associated testing and validation of designed use cases.

After you've developed a configuration in the PingOne admin console, you can extract it as Configuration as Code to be stored in source control (such as a Git code repository) and linked with CI/CD tooling to automate the delivery of use cases into test and production environments.

For professionals experienced in DevOps development, configuration can be created and altered outside of the PingOne admin console, but care must be taken when modifying complex configuration, such PingOne Authorize, PingOne MFA, PingOne Protect, or PingOne SSO sign-on policies.

Example or bootstrapped configuration dependencies

Deploy to "clean" environments, without example or bootstrapped configuration

Example or bootstrapped configuration is deployed automatically by the PingOne service when an environment is created or new services are provisioned to an existing environment. This is the default behavior of the PingOne admin console and the API.

Example or bootstrapped configuration can be a useful starting point when initially creating use cases with the service (in the development phase), but creates conflicts when migrating the configuration through to test and production environments.

The definition of the example or bootstrapped configuration for new environment can also change over time, as new features are released and use case configuration best practices are defined. An environment created today might not be the same as an environment created a year from now.

It's best practice to create a new environment as a "clean" (without example or bootstrapped configuration) environment for those environments outside of the initial development one. If environments cannot be recreated or are intended to be long-lasting (such as staging or production), it might be enough to remove bootstrapped configuration manually when an environment is created.

Notable examples of demo configuration include:

Platform

- The default branding theme
- Optional directory schema attributes (which can be disabled if not used)
    - `accountId`
    - `address`

  - `email`
  - `externalId`
  - `locale`
  - `mobilePhone`
  - `name`
  - `nickname`
  - `photo`
  - `preferredLanguage`
  - `primaryPhone`
  - `timezone`
  - `title`
  - `type`
- The default Keys and Certificates
- The default notification policies
- The default `Single_Factor` sign-on policy
- The example password policies
- The `PingOne Application Portal` (which can be disabled if not used)

DaVinci service

- Example Forms

MFA service

- The default MFA Device Policy
- The default FIDO2 policies
- The `Multi_Factor` sign-on policy

Verify service

- The default verify policy

Define all configuration dependencies in Terraform or elsewhere in the pipeline

Example or bootstrapped configuration is deployed automatically by the PingOne service when an environment is created or new services are provisioned to an existing environment. This is the default behavior of the PingOne admin console and the API.

Example or bootstrapped configuration can be a useful starting point when initially creating use cases with the service (in the development phase), but creates conflicts when migrating the configuration through to test and production environments.

The definition of the example or bootstrapped configuration for new environment can also change over time as new features are released and use case configuration best practices are defined. An environment created today might not be the same as an environment created a year from now.

It's best practice to explicitly define all configuration dependencies in Terraform (or as a prior step in the CI/CD pipeline) after developing flows for use cases. Most notably, this practice includes defining the policies (for example, sign-on, MFA Device, FIDO2, or Protect policies) that applications will use in HCL, rather than using the example or bootstrapped environment examples.

Not best practice

The following `pingone_population` definition doesn't follow best practice, as it depends on the "Passphrase" password policy that was deployed by default when the environment was created. This definition assumes that this password policy will always exist and have a consistent definition on every environment creation. However, the password policy could change over time, invalidating the configuration.

```
data "pingone_password_policy" "ootb_passphrase" {
  environment_id = pingone_environment.my_environment.id

  name = "Passphrase"
}

resource "pingone_population" "my_population" {
  environment_id = pingone_environment.my_environment.id

  name        = "My awesome population"
  description = "My new population for awesome people"

  password_policy_id = data.pingone_password_policy.ootb_passphrase.id

  lifecycle {
    # change the `prevent_destroy` parameter value to `true` to prevent this data carrying resource from being
destroyed
    prevent_destroy = false
  }
}
```

Best practice

The following `pingone_population` definition follows best practice, as the password policy that it depends on is explicitly defined using the `pingone_password_policy` resource. This explicit definition ensures that environments are built and configured consistently between development, test, and production.

```
resource "pingone_password_policy" "my_password_policy" {
  environment_id = pingone_environment.my_environment.id

  name        = "My awesome password policy"

  excludes_commonly_used_passwords = true
  excludes_profile_data            = true
  not_similar_to_current           = true

  history = {
    count          = 6
    retention_days = 365
  }

  # ... other configuration parameters
}

resource "pingone_population" "my_population" {
  environment_id = pingone_environment.my_environment.id

  name        = "My awesome population"
  description = "My new population for awesome people"

  password_policy_id = pingone_password_policy.my_password_policy.id

  lifecycle {
    # change the `prevent_destroy` parameter value to `true` to prevent this data carrying resource from being
destroyed
    prevent_destroy = false
  }
}
```

Protect service configuration and data

The following sections detail best practices to apply to ensure protection of production data beyond what is covered in Secrets management when using the PingOne Terraform provider.

Regularly rotate worker application secrets

In PingOne, administration management functions against the API can be performed by worker applications with admin roles assigned, as described in Configuring roles for a worker application in the PingOne documentation. To use these worker applications, you might need to generate an application secret and use that secret in downstream applications and services. You should rotate these secrets on a regular basis to help mitigate against unauthorized platform changes.

Rotation can be controlled by a secrets engine that can update with the relevant API, as described in the Update Application Secret in the PingOne developer documentation, but can also be rotated through the Terraform process.

For example, the following Terraform code will rotate an application secret for the application "My Awesome App" every 30 days:

```
resource "pingone_application" "my_application" {
  name = "My Awesome App"
  enabled        = true

  oidc_options = {
    type                        = "WORKER"
    grant_types                 = ["CLIENT_CREDENTIALS"]
    token_endpoint_auth_method  = "CLIENT_SECRET_BASIC"
  }

  # ... other configuration parameters
}

resource "time_rotating" "application_secret_rotation" {
  rotation_days = 30
}

resource "pingone_application_secret" "foo" {
  environment_id = pingone_environment.my_environment.id
  application_id = pingone_application.my_application.id

  regenerate_trigger_values = {
    "rotation_rfc3339" : time_rotating.application_secret_rotation.rotation_rfc3339,
  }
}
```

Review use of API force-delete provider overrides

The PingOne Terraform provider has a provider-level parameter named `global_options` that allows administrators to override API behaviors for development, test, and demo purposes. You can find details in the registry documentation of this parameter.

There are two parameters that allow force-deletion of configuration, which could result in loss of data if not used correctly.

`global_options.environment.production_type_force_delete`

WARNING:

The `global_options.environment.production_type_force_delete` global option was removed in the PingOne Terraform provider version v1.0. This section applies to prior provider versions (v0.29 or earlier).

Misuse of the parameter could cause unintended data loss, so it must be used with caution.

The purpose of the parameter is to override the API level restriction of not being able to destroy environments of type "PRODUCTION". The default value of this parameter is `false`, meaning that environments will not be force-deleted if a `pingone_environment` resource that has a `type` value of `PRODUCTION` has a destroy plan when run in the `terraform apply` phase. Use of this parameter is designed to help facilitate development, testing, or demonstration purposes and should be set to `false` or left undefined for environments that carry production data.

The implementation of this option is that the environment type will be changed from `PRODUCTION` to `SANDBOX` before a delete API request is issued. Instead of using this parameter, consider changing the type to `SANDBOX` manually before running a plan that destroys an environment.

`global_options.population.contains_users_force_delete`

Warning

Misuse of the parameter could cause unintended data loss, so it must be used with caution.

The purpose of the parameter is to override the API level restriction of not being able to destroy populations that contain user data. The default value of this parameter is `false`, meaning that populations that contain user data will not be force-deleted if a `pingone_population` resource has a destroy plan when run in the `terraform apply` phase. Use of this parameter is designed to help facilitate development, testing, or demonstration purposes where non-production user data is created and can be safely discarded. The parameter should strictly be set to `false` or left undefined for environments that carry production data.

The implementation of this option is that the provider will find and delete all users assigned to the population being destroyed before a delete API request is issued to the population. Instead of using this parameter, consider removing non-production data manually before running a plan that destroys a population.

Protect configuration and data with the `lifecycle.prevent_destroy` meta argument

While some resources are safe to remove and replace, there are some resources that, if removed, can result in data loss.

You should use the `lifecycle.prevent_destroy` meta argument to protect against accidental destroy plans that could cause data loss. You might also want to use the meta argument to prevent accidental removal of access policies and applications if dependent applications cannot be updated with Terraform in case of replacement.

For example:

```
resource "pingone_schema_attribute" "my_attribute" {
  environment_id = pingone_environment.my_environment.id

  name = "myAttribute"

  # ... other configuration parameters

  lifecycle {
    prevent_destroy = true
  }
}
```

The following resources, if destroyed, put data at risk within a PingOne environment:

- `pingone_schema_attribute`
  - If a custom schema attribute is created, a destroy of the schema attribute will erase that attribute's data for users.
- `pingone_population`
  - Users must belong to a population. If a population is removed, the users within that population could be at risk. There are platform controls to prevent accidental deletion of a population that contains users.

- `pingone_environment`
  - Users might belong to the environment's default population. If the environment is removed, the users within that population could be at risk. There are platform API-level controls to prevent accidental deletion of an environment where the environment's type is set to `PRODUCTION`. `SANDBOX` environments do not have such API restrictions.

Multi-team development

Use on-demand sandbox environments

PingOne customer tenants have a "tenant-in-tenant" architecture, whereby a PingOne tenant organisation can contain many individual environments. These individual environments can be purposed for development, test, pre-production, and production purposes. These separate environments allow for easy maintenance of multiple development and test instances.

The recommended approach for multi-team development, when using a GitOps CI/CD promotion process, is to spin up on-demand development and test environments, specific to new features or to individual teams, to allow for development and integration testing that doesn't conflict with other team's development and test activities. The Terraform provider allows administrators to use CI/CD automation to provision new environments as required and remove them after the project activity no longer needs them.

In a GitOps CI/CD promotion pipeline, configuration can be translated to Terraform Configuration as Code and then merged (with pull requests) with common test environments, where automated tests can be run. This flow allows the activities in the on-demand environments to be merged into a common promotion pipeline to production environments.

User administrator role assignment

Use group role assignments over Terraform-managed user role assignments

PingOne supports assigning administrator roles to groups, such that members of the group get the administrator roles assigned.

Ping Identity recommends that groups with admin role assignments are controlled by the Joiner/Mover/Leaver Identity Governance processes, separate from the Terraform CI/CD process that configures applications, policies, domain verification, and so on. It could be that the groups with their role assignments are initially seeded by Terraform. In this case, it should still be a separate Terraform process from the process that controls platform configuration, and the user group assignments should still happen in the Joiner/Mover/Leaver Identity Governance process.

You can use Terraform to assign administrator roles to individuals directly, but this is not recommended best practice except in development or non-production environments. Ping Identity recommends that role assignment processes in non-production environments align as closely as possible to role assignment processes in production environments.

Use custom roles to follow principles of least privilege

PingOne supports the creation of custom administrator roles, which allow an administrator to define their own admin role based on a collection of permissions that represent a specific purpose.

Ping Identity recommends that customers create custom administrator roles to follow least privilege principles. To mitigate accidental or malicious changes to environments, assign the minimum required permissions for users to be able to perform their roles.

You can use Terraform to manage custom administrator roles and also manage the assignment of custom roles to groups and users.

Develop with Terraform

Administrator role considerations

Learn about administrator role considerations when using Terraform to manage PingOne environments.

Admin Role Management Considerations

Importing a PingOne environment to Terraform state

Take a preconfigured PingOne environment and import to Terraform state. Importing to Terraform state allows Terraform to manage a product environment without needing to recreate any configuration for that environment. This is useful when bringing a production environment under Terraform control retrospectively.

Importing to Terraform state

Exporting or generating PingOne Terraform HCL

Generate Terraform HCL configuration for a preconfigured environment. This is useful when exporting configuration from a development environment to store in source control or promote to test or production.

Exporting Terraform configuration

Admin Role Management Considerations

When creating and managing environments using Terraform, admin role management must be considered to avoid unexpected errors or unexpected inability to manage platform resources. The following describes admin role management considerations that administrators must take when using the PingOne Terraform provider.

PingOne admin role model

An administrative role is a collection of permissions that you can assign to a user, group of users, application, or connection. Administrative roles give PingOne admins access to resources in the PingOne admin console, API, Terraform resources, and determine the actions they can take in PingOne.

You can find the list of available admin roles in PingOne roles in the PingOne documentation.

Each role is a collection of permissions. You can find the list of permissions that each role contains in PingOne role permissions in the PingOne developer documentation.

Considerations when using Terraform to create environments

When creating environments with Terraform using the `pingone_environment` resource, the worker application used to connect Terraform to the PingOne tenant should have the **Organization Admin** role assigned, as that role contains the permission **Create, promote, read, update, and delete environment**.

After the worker application has created the new environment, the worker application automatically inherits the following roles *scoped to the new environment*:

- **Environment Admin**.
- **Identity Data Admin**.
- **Client Application Developer**.

Any existing user, group of users, application, or connection that has an admin role assignment *scoped to the organization* also inherits the ability to manage the new environment with the permissions assigned to that role.

For example, if admin user Barbara Jensen has **Environment Admin** *scoped to the organization* and **Identity Data Admin** *scoped to individual environments*, either assigned directly or by being member of a group that has those admin roles assigned, Barbara will inherit the role permissions to be able to manage that new environment (inherited **Environment Admin** role permissions), but will *not* be able to manage user and group data of the environment (the **Identity Data Admin** role permissions have not been inherited). In this example, if Barbara needs to manage user identities and groups, she would need the **Identity Data Admin** role assigned *scoped to the new environment*, either directly or by being made a member of a group that has that role assigned. Learn more in Assigning admin roles.

Note

To prevent privilege escalation, admin users, worker applications, or connections that previously could view and manage the worker application's secret might no longer be able to do so after an environment has been created with Terraform. This change can lead to an error ***Actor does not have permissions to access worker application client secrets***. You can find more information in When admins cannot view a worker application secret.

Other than the birthright roles assigned to the worker application on environment creation and the inherited permissions on actors with roles scoped to the organization, no other role assignments are given implicitly.

It's now up to the customer tenant administrators to consider:

- Are the Terraform worker application's birthright roles sufficient to perform further configuration with Terraform?

  If not, further roles might need to be explicitly assigned to the worker application. Adding additional roles can be done in the PingOne admin console by an administrator, by API, or by Terraform. Learn more in Assigning admin roles.

- Are the worker application's birthright role permissions beyond what's required for the worker application to perform its configuration management purpose and contravene least privilege principles?

  In this case, roles might need to be revoked from the worker application. Role revoking can be done in the PingOne admin console by an administrator or by API. You cannot revoke birthright roles from a worker application used to create an environment using Terraform unless the birthright role assignments are first imported into Terraform state. Learn more in Importing role assignments to Terraform state._

- Should other users, worker applications, or connections be granted administrative roles that can manage the new environment or continue to manage the secret of the Terraform worker application? (Refer to When admins cannot view or manage a worker application secret).

  Roles can be explicitly assigned to any user, group of users, worker application, or connection in the PingOne admin console, by API, or by Terraform. Learn more in Assigning admin roles.

- Do existing users, worker applications, or connections that have roles *scoped to the organization* (and therefore implicitly gain permissions to manage the new environment) have the appropriate role permissions, or do those users (through direct assignment or through group membership), worker applications, or connections need to have their role scope reduced such that their roles should instead be *scoped to individual environments*.

  Reducing the scope of admin role assignments can be achieved in the PingOne admin console by an administrator, by API, or by Terraform (if those role assignments are managed using Terraform).

Assigning admin roles

When assigning admin roles to users, worker applications, or connections, the role assignments can be assigned with a scope to individual environments, populations, or to the entire organization (depending on the role). Users can be assigned roles directly or by being members of a group that's been assigned admin roles.

Admin role assignments can be managed in the PingOne admin console, by API, or by Terraform. When using Terraform, the following resources apply:

- Assigning admin roles directly to users: `pingone_user_role_assignment` resource. *Role conflicts might occur. Learn more in Role assignment scope conflicts.*
- Assigning admin roles to groups: `pingone_group_role_assignment` resource. *Role conflicts might occur on the group role assignment, but role conflicts on group user members are resolved automatically. Learn more in Role assignment scope conflicts.*
- Assigning admin roles to worker applications: `pingone_application_role_assignment` resource. *Role conflicts might occur. Learn more in Role assignment scope conflicts.*
- Assigning admin roles to connections: `pingone_gateway_role_assignment` resource. *Role conflicts might occur. Learn more in Role assignment scope conflicts.*

Learn how Terraform can be used to assign roles to administrative actors in the Role assignment with Terraform tutorial.

Ping Identity recommends that customers follow documented general best practices and PingOne-specific best practices for developing with Terraform, paying close attention to best practices around administrative role assignment.

Role assignment scope conflicts

When assigning roles to users, groups of users, applications, or connections using Terraform, there are situations where role assignment uniqueness conflicts can occur. Errors could look similar to the following:

```
PingOne Error Details:
ID: f21b****-****-****-****-********a7d0
Code: INVALID_DATA
Message: The request could not be completed. One or more validation errors were in the request.
Details:
  - Code:       UNIQUENESS_VIOLATION
    Message:    May not assign duplicate Role
    Target:     role
```

This error occurs when a user, group, worker application, or connection already has the role assignment at a scope that is greater than or equal to the scope being configured.

For example, Janet Smith has the **Environment Admin** role assigned *scoped to the organization*. The `terraform apply` run attempts to assign the **Environment Admin** to Janet using the `pingone_user_role_assignment` resource, *scoped to an individual environment*. Because Janet already has **Environment Admin** with organization-level permissions (and so can manage all environments), Terraform is attempting to add duplicate role permissions.

In this example, the user-level conflict can be resolved by instead managing user's role assignments using groups, using the `pingone_group_role_assignment` resource where needed. When using Terraform to manage role assignments, using groups to manage user's role assignments is a documented best practice.

In the case of worker applications and connections, the Terraform HCL must be adjusted to resolve the role assignment scope conflict. You can change the following:

- Change how role assignments are managed as an out-of-band control to avoid the possibility of conflict. In this way, admin-level controls outside of Terraform ensure that conflicts are unlikely to happen when Terraform needs to manage role conflicts.
- Manage roles with privileged access management tools to reduce the need for Terraform to manage role assignments.
- Use Terraform to calculate the possibility of conflicting role assignments.

Note

There are disadvantages to using Terraform to calculate the possibility of conflicting role assignments, as role assignments for a user, group, application, or connection must be fully known in order to calculate the potential conflicts. Because in Terraform, IDs are not known until the first `terraform apply`, this can lead to a situation where the Terraform HCL must be applied twice:

1. Applying the initial role assignments
2. Applying again to perform an accurate calculation to ensure any future role assignments are not in conflict

You can find an example HCL in GitHub issue 478

Importing role assignments to Terraform state

Role assignments to users, groups, worker applications, and connections might have been defined outside of Terraform's management control. This includes role assignments that:

- Have been defined by administrators in the PingOne admin console
- Have been defined by administrators or scripts using the PingOne platform management API

- Have been defined implicitly by the platform when creating new environments though the PingOne admin console, by API, or by Terraform (using the `pingone_environment` resource)

These role assignments can be brought under Terraform's management control by using the **Terraform import** functionality. Import is supported on the following resources:

- Admin role assignment to a user directly: `pingone_user_role_assignment` resource
- Admin role assignment to a group: `pingone_group_role_assignment` resource
- Admin role assignment to a worker application: `pingone_application_role_assignment` resource
- Admin role assignment to a connection: `pingone_gateway_role_assignment` resource

Import by Terraform CLI

Hashicorp Terraform provides a standard CLI command, `terraform import`, that can be used to import any supported resource into Terraform state. Learn more in Importing to Terraform state.

Each resource listed previously contains an example for importing the resource to Terraform state using the Terraform CLI.

Import by Terraform configuration language

Hashicorp Terraform provides a standard configuration language import declaration block, `import {}`, that you can use to import any supported resource into Terraform state and generate its resulting HCL. Learn more in Importing to Terraform state.

The following is an example of a role assignment import for `pingone_group_role_assignment`, where the ID is a composite ID of `<environment_id>/<group_id>/<role_assignment_id>`, as shown in the Terraform CLI import example in the registry documentation:

```
import {
  to = pingone_group_role_assignment.example
  id = "e16f****-****-****-****-********15be/80f9****-****-****-****-********e828/f936****-****-****-****-
********65c1"
}
```

When admins cannot view or manage a worker application secret

Admin actors (users, worker applications, and connections) might not be able to view or rotate a worker application's secret when they were able to previously as an unexpected change of behavior.

The issue can be observed in the PingOne admin console (manifesting as a lack of control over a worker application's secret), a `403` error response from the Read Application Secret API, or the following error within Terraform when attempting to use the `pingone_application_secret` resource or data source:

```
PingOne Error Details:
ID: f21b****-****-****-****-********a7d0
Code: ACCESS_FAILED
Message: The request could not be completed. You do not have access to this resource.
Details:
  - Code:       INSUFFICIENT_PERMISSIONS
    Message:    Actor does not have permissions to access worker application client secrets
```

The change in ability to manage a worker application's client secret typically occurs when the worker application is granted additional role permissions that the user, admin worker application, or connection doesn't have. The worker application whose secret cannot be managed has a higher level of privilege to manage configuration and data within the tenant. The ability to view and change the secret is therefore restricted to mitigate privilege escalation issues where admin actors could potentially use the higher privileged worker application to make changes they aren't authorized to make in the platform.

For example, the worker application Terraform Admin is used to create a new environment using the `pingone_environment` resource. The Terraform Admin worker application is implicitly granted birthright roles to be able to manage that environment (refer to Considerations when using Terraform to create environments), but other admin users, worker applications, and connections aren't provided the same birthright role permission assignments.

The Terraform Admin worker application that created the environment now has higher privileges than other administrators, so privilege escalation controls are applied to other platform administrators. Other platform administrators have now lost the ability to view and manage the Terraform Admin worker application secret.

The resolution is to apply the missing roles permissions by assigning roles to the users, worker applications, or connections that need to be able to manage the worker application's secret.

In the previous example, you would add a combination of **Environment Admin**, **Identity Data Admin**, and **Client Application Developer** roles *scoped to the newly created environment* to the users, worker applications, or connections that need to be able to manage the "Terraform Admin" worker application's secret.

Roles can be explicitly assigned to any user, group of users, worker application, or connection in the PingOne admin console, by API or by Terraform. Learn more in Assigning admin roles.

Tutorials

Role assignment using Terraform

Learn how to use Terraform to automatically grant roles to users, groups, worker applications, or gateways.

Role assignment with Terraform

Configure the end-user Self Service application

Learn how to use Terraform to enable and disable features in the default end-user Self Service application.

Configuring the PingOne Self Service application

Role assignment with Terraform

The following shows an example of environment creation using the PingOne Terraform provider, followed by role permission assignment to administration users that are members of a group we will create.

Note

PingOne supports assigning administrator roles groups, and members of those groups are assigned administrator roles. Although you can use Terraform to assign administrator roles to individuals directly, Ping Identity recommends that role assignments provisioned by Terraform are assigned to groups instead and that you manage group membership through Joiner/Mover/Leaver Identity Governance processes.

The example assumes that all relevant admins users will have a role strategy as follows:

- **Environment Admin**, scoped to individual environments (not scoped to the organization)
- **Identity Data Admin**, scoped to individual environments

Note

The example uses:

- The `pingone_admin_environment_id` variable that can be mapped directly or can be found from the environment name from the `pingone_environment` data source.
- The `license_id` variable that can be mapped directly or can be found from the license name from the `pingone_licenses` data source.

First, you'll create the group in PingOne to which you'll assign your administrator users. This example uses the `pingone_group` resource.

```
resource "pingone_group" "my_awesome_admins_group" {
  environment_id = var.pingone_admin_environment_id

  name        = "My awesome admins group"
  description = "My new awesome group for admins who are awesome"

  lifecycle {
    # change the `prevent_destroy` parameter value to `true` to prevent this data carrying resource from being
destroyed
    prevent_destroy = false
  }
}
```

Next, you'll fetch the required roles. You'll need to find the IDs of the **Identity Data Admin** and **Environment Admin** predefined admin roles, which are different between tenant organizations. You can use the `pingidentity/utils/pingone` helper module to retrieve the role IDs, so that you can use role IDs in role assignment to the group:

```
module "admin_utils" {
  source  = "pingidentity/utils/pingone"
  version = "0.1.0"

  region_code    = "EU" // Will be either NA, EU, CA, AU or AP depending on your tenant region.
  environment_id = var.pingone_admin_environment_id
}
```

Note

When including a new module in Terraform HCL, remember to re-run `terraform init` to initialize the module in the Terraform project.

You can then define the new sandbox environment using the PingOne Terraform provider with the `pingone_environment` resource, with the SSO service enabled. This is the environment to which you want to scope the administrator roles so that your users can manage configuration and data within this environment:

```
resource "pingone_environment" "my_environment" {
  name        = "Example PingOne Role Permission Assignment Environment"
  type        = "SANDBOX"
  license_id  = var.license_id

  services = [
    {
      type = "SSO"
    }
  ]
}
```

After you've created the new environment, you can assign the roles to the administration users with the `pingone_group_role_assignment` resource.

```
resource "pingone_group_role_assignment" "admin_sso_identity_admin" {
  environment_id = var.pingone_admin_environment_id
  group_id       = pingone_group.my_awesome_admins_group.id
  role_id        = module.admin_utils.pingone_role_id_identity_data_admin

  scope_environment_id = pingone_environment.my_environment.id
}

resource "pingone_group_role_assignment" "admin_sso_environment_admin" {
  environment_id = var.pingone_admin_environment_id
  group_id       = pingone_group.my_awesome_admins_group.id
  role_id        = module.admin_utils.pingone_role_id_environment_admin

  scope_environment_id = pingone_environment.my_environment.id
}
```

The group "My awesome admins group" has now been assigned the **Identity Data Admin** and **Environment Admin** roles. Any user who is made a member of the group will inherit these administrative roles and their associated permissions.

Configuring the PingOne Self Service application

The following shows an example of how to configure the PingOne Self Service system application.

You can configure the PingOne Self Service application in the PingOne admin console. It's a web application, and its capabilities are configured by assigning resource scopes to the application, rather than through a dedicated API or Terraform resource.

First, you'll need to ensure that the Self Service application itself is configured using the `pingone_system_application` resource.

```
resource "pingone_system_application" "pingone_self_service" {
  environment_id = pingone_environment.my_environment.id

  type    = "PING_ONE_SELF_SERVICE"
  enabled = true

  apply_default_theme         = true
  enable_default_theme_footer = true
}
```

You'll then select which self-service capabilities (the scopes) you want to apply to the Self Service application. The simplest way is to create a list and select the appropriate scope data using the `pingone_resource_scope` data source.

```
locals {
  pingone_api_scopes = [
    # Manage Profile
    "p1:read:user",
    "p1:update:user",

    # Manage Authentication
    "p1:create:device",
    "p1:create:pairingKey",
    "p1:delete:device",
    "p1:read:device",
    "p1:read:pairingKey",
    "p1:update:device",

    # Enable or Disable MFA
    "p1:update:userMfaEnabled",

    # Change Password
    "p1:read:userPassword",
    "p1:reset:userPassword",
    "p1:validate:userPassword",

    # Manage Linked Accounts
    "p1:delete:userLinkedAccounts",
    "p1:read:userLinkedAccounts",

    # Manage Sessions
    "p1:delete:sessions",
    "p1:read:sessions",

    # View Agreements
    "p1:read:userConsent",

    # Manage OAuth Consents
    "p1:read:oauthConsent",
    "p1:update:oauthConsent",
  ]
}

data "pingone_resource_scope" "pingone_api" {
  for_each = toset(local.pingone_api_scopes)

  environment_id = pingone_environment.my_environment.id
  resource_type  = "PINGONE_API"

  name = each.key
}
```

Next, you'll map the appropriate scopes to enable the specific self-service features you want using the `pingone_application_resource_grant` resource.

```
resource "pingone_application_resource_grant" "my_awesome_spa_pingone_api_resource_grants" {
  environment_id = pingone_environment.my_environment.id
  application_id = pingone_system_application.pingone_self_service.id

  resource_type = "PINGONE_API"

  scopes = [
    for scope in data.pingone_resource_scope.pingone_api : scope.id
  ]
}
```

The Self Service application is now configured with the required capabilities.

You can find the full runable example on GitHub.

Frequently Asked Questions

How do I export configuration from a previously configured environment?

You can export configuration from a PingOne environment using a combination of Ping CLI and Terraform CLI tools.

Learn more in Exporting Terraform configuration.

How do I bring a previously configured environment under Terraform management?

You can bring any environment that has been configured without using Terraform under Terraform management using a combination of Ping CLI and Terraform CLI tools.

Learn more in Importing to Terraform state.

I cannot create a workforce-enabled environment or where can I Terraform creation of a PingID-enabled environment?

The PingOne provider does not yet support creation of a PingID-enabled workforce environment. You can track the list of known issues and provider limitations on the project's GitHub.

I've created a new environment with Terraform, but my admins can't see it

Check the admin user's role permissions. The admin user must have any of the following roles to see it in the list of environments:

- **Organization Admin**
- **Environment Admin**
- **Identity Data Admin**
- **Client Application Developer**
- **Identity Data Read Only**
- **Configuration Read Only**

Refer to the Admin Role Management Considerations guide for details on role assignment and considerations for admin role management when using Terraform.

I've created a new environment or population with Terraform, but my admins can't view users, or manage group or population based configuration

Check the admin user's role permissions. The admin user must have any of the following roles to be able to view and manage identity data and configuration:

- **Identity Data Admin**

- **Identity Data Read Only**

Refer to the Admin Role Management Considerations guide for details on role assignment and considerations for admin role management when using Terraform.

I get an error "Actor does not have permissions to access worker application client secrets"

Admin actors (users, worker applications, connections) might not be able to view or rotate a worker application's secret when they could previously as an unexpected change of behavior.

The change in ability to manage a worker application's client secret typically occurs when the worker application is granted additional role permissions that the user, admin worker application, or connection doesn't have. The worker application whose secret cannot be managed has a higher level of privilege to manage configuration and data within the tenant. The ability to view and change the secret is therefore restricted to mitigate privilege escalation issues where admin actors could potentially use the higher privileged worker application to make changes they aren't authorized to make in the platform.

You can find more information and guidance on how to resolve this error in Admin Role Management Considerations.

Getting started

The following provides guidance on preparing a PingFederate deployment for Terraform access.

Requirements

- Terraform CLI 1.4+
- A running PingFederate server accessible over HTTPS, or Docker CLI to start one.
- If using Docker to start a PingFederate server, you must have a DevOps license. Register for the DevOps program here.

(Optional) Start a PingFederate Docker container

Note

If you already have a running PingFederate server that you can reach over HTTPS, you can skip this step. The provider can be used with any PingFederate server.

1. Start a PingFederate server. The following example shows how to start a single PingFederate server using Docker.

   Your DevOps credentials will be read from the `.pingidentity/config` file in the user's home directory. The HTTPS port (default `9999`) must be exposed. This example starts the product with the `getting-started/pingfederate` server profile, running the latest version of PingFederate from Docker Hub.

   ```
   docker run --name pingfederate_terraform_provider_container \
     -d -p 9031:9031 \
     -d -p 9999:9999 \
     --env-file "${HOME}/.pingidentity/config" \
     -e SERVER_PROFILE_URL=https://github.com/pingidentity/pingidentity-server-profiles.git \
     -e SERVER_PROFILE_PATH=getting-started/pingfederate
     pingidentity/pingfederate:latest
   ```

2. After starting the container, follow the logs until the server becomes available.

   ```
   docker logs -f pingfederate_terraform_provider_container
   ```

After you see the following message in the container logs, the server is ready to receive requests from the provider:

```
PingFederate is up
```

Accessing the PingFederate API

Gaining access to the API depends on where your PingFederate instance is running. In order to work with the API, you need to authenticate just as you would for performing administrative tasks. To interact directly with the API in the browser and view the documentation, you would typically go to the following URL:

https://<pf_host>:9999/pf-admin-api/api-docs/

where *<pf_host>* is the network address of your PingFederate server. This target can be an IP address, a host name, or a fully qualified domain name. It must be reachable from your computer.

If you're using OIDC, you can find information on how to connect in OIDC authentication in the PingFederate documentation.

When using a local container as shown above, the port mappings result in PingFederate being available at https://localhost:9999/pingfederate/app#/ by default. The provider supports an attribute `admin_api_path` which defaults to **/pf-admin-api/v1**. If your environment has a load balancer or other network configuration that requires a different path to the API, you can configure the provider to use it.

The PingFederate Terraform provider applies configuration at the API endpoints using the specified port (`9999`) over HTTPS when accessing the product locally under Docker.

Determine credentials that are able to configure the server

The provider supports basic authentication, OAuth2 client credentials flow authentication, and access token authentication for connection to the configuration API. In this example, basic authentication will be used.

When using basic authentication, the provider will need the `username` and `password` of a user with permission to manage server configuration.

Note

When using the Ping Identity Docker images, the default username and password can be used. Learn more in Deploy an Example Stack in the Ping Identity DevOps documenation.

When using OAuth2 client credentials, the `client_id` , `client_secret` , and `token_url` attributes are required in the provider configuration, with `scopes` being optional.

When using an access token, only `access_token` is required in the provider configuration.

You can find examples of configuring other authentication methods in the Terraform registry documentation.

Determine what version of PingFederate you're running

The provider requires that the version of PingFederate is specified.

You can do this one of two ways:

1. Using the `product_version` attribute when configuring the provider:

```
provider "pingfederate" {
  ...
  product_version = "12.1"
}
```

2. Setting the `PINGFEDERATE_PROVIDER_PRODUCT_VERSION` environment variable to the version of PingFederate you're running:

```
export PINGFEDERATE_PROVIDER_PRODUCT_VERSION=12.1
```

If using the container, you can view the product version using by running a shell in the container and searching for the appropriate environment variable:

```
docker container exec -it pingfederate_terraform_provider_container /bin/sh
```

Next, in the container shell, get the version:

```
env | grep PING_PRODUCT_VERSION
```

Note

The `product_version` provider configuration attribute takes precedence over the `PINGFEDERATE_PROVIDER_PRODUCT_VERSION` environment variable.

If neither is set, the provider will throw an error.

Trusting PingFederate certificates

PingFederate generates a self-signed certificate by default, which is presented by the server when connecting. The default self-signed certificate can be replaced with a custom certificate. The provider has a few ways of configuring trust for the HTTPS connection with the server.

By default, the provider will trust the host's default root certificate authority (CA) set when connecting to the server.

If you need to provide CA certificates for the provider to trust, you can use the `ca_certificate_pem_files` attribute. This attribute supports providing a set of paths to files containing PEM-encoded CA certificates to be trusted.

You can also use the `PINGFEDERATE_PROVIDER_CA_CERTIFICATE_PEM_FILES` environment variable, with commas to delimit multiple PEM file paths if necessary.

Finally, the provider supports an `insecure_trust_all_tls` boolean attribute that enables it to trust all certificates when connecting to the server.

Warning

The `insecure_trust_all_tls` provider flag, when set to `true`, is insecure and is intended for development and testing only. You should not enable this option for production use.

Use the provider to configure PingFederate

You are now ready to configure the PingFederate server using the provider.

You can find examples on configuring the Terraform provider to manage PingFederate configuration in the PingFederate Provider Registry documentation.

Frequently Asked Questions

What versions of PingFederate are supported?

You can find the list of supported PingFederate versions by provider version the latest provider documentation.

How do I configure trust for connecting over HTTPS?

You can find details on configuring TLS trust in the Getting started guide.

Getting started

The following provides guidance on preparing a PingDirectory deployment for Terraform access.

Requirements

- Terraform CLI 1.1+
- A running PingDirectory server accessible over HTTPS, or Docker CLI to start one.
- When using Docker to start a PingDirectory server, you must have a DevOps license. Register for the DevOps program here.

(Optional) Start a PingDirectory Docker container

Note

If you already have a running PingDirectory server that you can reach over HTTPS, you can skip this step. The provider can be used with any PingDirectory server.

1. Start a PingDirectory server. The following example shows how to start a single PingDirectory server using Docker.

   Your DevOps credentials will be read from the `.pingidentity/config` file in the user's home directory. The HTTPS port (default `1443`) must be exposed.

   ```
   docker run --name pingdirectory_terraform_provider_container \
                   -d -p 1443:1443 \
                   -d -p 1389:1389 \
                   -e TAIL_LOG_FILES= \
                   --env-file "${HOME}/.pingidentity/config" \
                   pingidentity/pingdirectory:latest
   ```

2. After starting the container, follow the logs until the server becomes available.

   ```
   docker logs -f pingdirectory_terraform_provider_container
   ```

After you see the following message in the container logs, the server is ready to receive requests from the provider:

```
Setting Server to Available
```

Ensure the Configuration HTTP Servlet extension is enabled

The PingDirectory Terraform provider applies configuration using the Configuration HTTP servlet extension, which must be enabled for the server's HTTPS connection handler.

This setting is already configured by default in PingDirectory, including when running in Docker.

If you've disabled the Configuration HTTP servlet extension on your server, you can re-enable it with dsconfig:

```
dsconfig set-connection-handler-prop --handler-name "HTTPS Connection Handler" --add http-servlet-
extension:Configuration
```

Determine what port the server is using for HTTPS connections

The PingDirectory Docker image uses port `1443` for HTTPS by default.

To determine what port you're using, use the `status` command and examine the output for a block containing the HTTPS port:

```
dsconfig status
```

```
          --- Connection Handlers ---
Address:Port : Protocol : State    : Name
-------------:----------:----------:-------------------------
0.0.0.0:1389 : LDAP     : Enabled  : LDAP Connection Handler
0.0.0.0:1443 : HTTPS    : Enabled  : HTTPS Connection Handler
0.0.0.0:1636 : LDAPS    : Enabled  : LDAPS Connection Handler
```

Determine credentials that are able to configure the server

The Configuration API used by the provider uses basic authentication. The provider will need the username and password of a user that has permissions to manage server configuration.

Note

When using the Ping Identity Docker images, the default username and password can be used. Learn more in Deploy an Example Stack.

Determine what version of PingDirectory you are running

The provider requires that the version of PingDirectory is specified through the `product_version` attribute or the `PINGDIRECTORY_PROVIDER_PRODUCT_VERSION` environment variable.

You can view the product version using the `status` command. Look for the Server Details section:

```
dsconfig status
```

```
          --- Server Details ---
Host Name:              ...
Instance Name:          ...
Administrative Users: cn=administrator
Installation Path:    /opt/out/instance
Server Version:       Ping Identity Directory Server 9.2.0.0
```

Trusting PingDirectory certificates

PingDirectory generates a self-signed certificate by default, which is presented by the server's HTTPS connection handler. You can replace the default self-signed certificate with a custom certificate. The provider has a few ways of configuring trust for the HTTPS connection with the server.

By default, the provider will trust the host's default root Certificate Authority (CA) set when connecting to the server.

The provider also supports an `insecure_trust_all_tls` boolean attribute (configurable with environment variable `PINGDIRECTORY_PROVIDER_INSECURE_TRUST_ALL_TLS`) that allows simply trusting all certificates when connecting to the server. This option is insecure and should not be used in production.

If you need to provide CA certificates for the provider to trust, you can use the `ca_certificate_pem_files` attribute. This attribute allows you to provide a set of paths to files containing PEM-encoded CA certificates to be trusted. The `PINGDIRECTORY_PROVIDER_CA_CERTIFICATE_PEM_FILES` environment variable can also be used, with commas to delimit multiple PEM file paths if necessary.

If you want to trust the default self-signed certificate of the PingDirectory server, you can export the certificate from the server's keystore using the `manage-certificates` command-line tool.

Write the output of that command to a file. Then you can include the path to that file in the `ca_certificate_pem_files` attribute when using the provider. The following example uses `cert.pem` as the filename:

```
manage-certificates export-certificate --keystore config/keystore --alias server-cert > cert.pem
```

Use the provider to configure PingDirectory

You are now ready to configure the PingDirectory server with the provider.

You can find examples on configuring the Terraform provider to manage PingDirectory configuration in the PingDirectory Provider Registry documentation.

Frequently Asked Questions

What versions of PingDirectory are supported?

You can find the list of supported PingDirectory versions by provider version the latest provider documentation.

How do I configure trust for connecting over HTTPS?

You can find details on configuring TLS trust in the Getting started guide.