HashiCorp
(https://www.hashicorp.com)

Workspaces

JUMP TO SECTION

Each Terraform configuration has an associated backend (/docs/language/settings/backends/index.html) that defines how operations are executed and where persistent data such as the Terraform state (https://www.terraform.io/docs/language/state/purpose.html) are stored.

The persistent data stored in the backend belongs to a *workspace*. Initially the backend has only one workspace, called "default", and thus there is only one Terraform state associated with that configuration.

Certain backends support *multiple* named workspaces, allowing multiple states to be associated with a single configuration. The configuration still has only one backend, but multiple distinct instances of that configuration to be deployed without configuring a new backend or changing authentication credentials.

Multiple workspaces are currently supported by the following backends:

- AzureRM (/docs/language/settings/backends/azurerm.html)
- Consul (/docs/language/settings/backends/consul.html)
- COS (/docs/language/settings/backends/cos.html)
- GCS (/docs/language/settings/backends/gcs.html)
- Kubernetes (/docs/language/settings/backends/kubernetes.html)
- Local (/docs/language/settings/backends/local.html)
- Manta (/docs/language/settings/backends/manta.html)
- Postgres (/docs/language/settings/backends/pg.html)
- Remote (/docs/language/settings/backends/remote.html)
- S3 (/docs/language/settings/backends/s3.html)

In the 0.9 line of Terraform releases, this concept was known as "environment". It was renamed in 0.10 based on feedback about confusion caused by the overloading of the word "environment" both within Terraform itself and within organizations that use Terraform.

> **Note**: The Terraform CLI workspace concept described in this document is different from but related to the Terraform Cloud workspace (/docs/cloud/workspaces/index.html) concept. If you use multiple Terraform CLI workspaces in a single Terraform configuration and are migrating that configuration to Terraform Cloud, see this migration document (/docs/cloud/migrate/workspaces.html).

Using Workspaces

Terraform starts with a single workspace named "default". This workspace is special both because it is the default and also because it cannot ever be deleted. If you've never explicitly used workspaces, then you've only ever worked on the "default" workspace.

Workspaces are managed with the `terraform workspace` set of commands. To create a new workspace and switch to it, you can use `terraform workspace new`; to switch workspaces you can use `terraform workspace select`; etc.

For example, creating a new workspace:

```
$ terraform workspace new bar
Created and switched to workspace "bar"!

You're now on a new, empty workspace. Workspaces isolate their state,
so if you run "terraform plan" Terraform will not see any existing state
for this configuration.
```

As the command says, if you run `terraform plan`, Terraform will not see any existing resources that existed on the default (or any other) workspace. **These resources still physically exist,** but are managed in another Terraform workspace.

Current Workspace Interpolation

Within your Terraform configuration, you may include the name of the current workspace using the `${terraform.workspace}` interpolation sequence. This can be used anywhere interpolations are allowed. However, it should **not** be used in remote operations against Terraform Cloud

workspaces. For an explanation, see the remote backend (/docs/language/settings/backends/remote.html#workspaces) document.

Referencing the current workspace is useful for changing behavior based on the workspace. For example, for non-default workspaces, it may be useful to spin up smaller cluster sizes. For example:

```
resource "aws_instance" "example" {
  count = "${terraform.workspace == "default" ? 5 : 1}"

  # ... other arguments
}
```

Another popular use case is using the workspace name as part of naming or tagging behavior:

```
resource "aws_instance" "example" {
  tags = {
    Name = "web - ${terraform.workspace}"
  }

  # ... other arguments
}
```

When to use Multiple Workspaces

Named workspaces allow conveniently switching between multiple instances of a *single* configuration within its *single* backend. They are convenient in a number of situations, but cannot solve all problems.

A common use for multiple workspaces is to create a parallel, distinct copy of a set of infrastructure in order to test a set of changes before modifying the main production infrastructure. For example, a developer working on a complex set of infrastructure changes might create a new temporary workspace in order to freely experiment with changes without affecting the default workspace.

Non-default workspaces are often related to feature branches in version control. The default workspace might correspond to the "main" or "trunk" branch, which describes the intended state of production infrastructure. When a feature branch is created to develop a change, the developer of that feature might create a corresponding workspace and deploy into it a temporary "copy" of

the main infrastructure so that changes can be tested without affecting the production infrastructure. Once the change is merged and deployed to the default workspace, the test infrastructure can be destroyed and the temporary workspace deleted.

When Terraform is used to manage larger systems, teams should use multiple separate Terraform configurations that correspond with suitable architectural boundaries within the system so that different components can be managed separately and, if appropriate, by distinct teams. Workspaces *alone* are not a suitable tool for system decomposition, because each subsystem should have its own separate configuration and backend, and will thus have its own distinct set of workspaces.

In particular, organizations commonly want to create a strong separation between multiple deployments of the same infrastructure serving different development stages (e.g. staging vs. production) or different internal teams. In this case, the backend used for each deployment often belongs to that deployment, with different credentials and access controls. Named workspaces are *not* a suitable isolation mechanism for this scenario.

Instead, use one or more re-usable modules (/docs/language/modules/develop/index.html) to represent the common elements, and then represent each instance as a separate configuration that instantiates those common elements in the context of a different backend. In that case, the root module of each configuration will consist only of a backend configuration and a small number of `module` blocks whose arguments describe any small differences between the deployments.

Where multiple configurations are representing distinct system components rather than multiple deployments, data can be passed from one component to another using paired resources types and data sources. For example:

- Where a shared Consul (https://consul.io/) cluster is available, use `consul_key_prefix` (https://registry.terraform.io/providers/hashicorp/consul/latest/docs/resources/key_prefix) to publish to the key/value store and `consul_keys` (https://registry.terraform.io/providers/hashicorp/consul/latest/docs/data-sources/keys) to retrieve those values in other configurations.
- In systems that support user-defined labels or tags, use a tagging convention to make resources automatically discoverable. For example, use the `aws_vpc` resource type (https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/vpc) to assign suitable tags and then the `aws_vpc` data source (https://registry.terraform.io/providers/hashicorp/aws/latest/docs/data-sources/vpc) to query by those tags in other configurations.

- For server addresses, use a provider-specific resource to create a DNS record with a predictable name and then either use that name directly or use the `dns` provider (https://registry.terraform.io/providers/hashicorp/dns/latest/docs) to retrieve the published addresses in other configurations.

- If a Terraform state for one configuration is stored in a remote backend that is accessible to other configurations then `terraform_remote_state` (/docs/language/state/remote-state-data.html) can be used to directly consume its root module outputs from those other configurations. This creates a tighter coupling between configurations, but avoids the need for the "producer" configuration to explicitly publish its results in a separate system.

Workspace Internals

Workspaces are technically equivalent to renaming your state file. They aren't any more complex than that. Terraform wraps this simple notion with a set of protections and support for remote state.

For local state, Terraform stores the workspace states in a directory called `terraform.tfstate.d`. This directory should be treated similarly to local-only `terraform.tfstate`; some teams commit these files to version control, although using a remote backend instead is recommended when there are multiple collaborators.

For remote state (/docs/language/state/remote.html), the workspaces are stored directly in the configured backend (/docs/language/settings/backends/index.html). For example, if you use Consul (/docs/language/settings/backends/consul.html), the workspaces are stored by appending the workspace name to the state path. To ensure that workspace names are stored correctly and safely in all backends, the name must be valid to use in a URL path segment without escaping.

The important thing about workspace internals is that workspaces are meant to be a shared resource. They aren't a private, local-only notion (unless you're using purely local state and not committing it).

The "current workspace" name is stored only locally in the ignored `.terraform` directory. This allows multiple team members to work on different workspaces concurrently. The "current workspace" name is **not** currently meaningful in Terraform Cloud workspaces since it will always have the value `default`.