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

Command: init

JUMP TO SECTION

Hands-on: Try the Terraform: Get Started (https://learn.hashicorp.com/collections/terraform/aws-get-started?utm_source=WEBSITE&utm_medium=WEB_IO&utm_offer=ARTICLE_PAGE&utm_content=DOCS) collection on HashiCorp Learn.

The `terraform init` command is used to initialize a working directory containing Terraform configuration files. This is the first command that should be run after writing a new Terraform configuration or cloning an existing one from version control. It is safe to run this command multiple times.

Usage

Usage: `terraform init [options]`

This command performs several different initialization steps in order to prepare the current working directory for use with Terraform. More details on these are in the sections below, but in most cases it is not necessary to worry about these individual steps.

This command is always safe to run multiple times, to bring the working directory up to date with changes in the configuration. Though subsequent runs may give errors, this command will never delete your existing configuration or state.

General Options

The following options apply to all of (or several of) the initialization steps:

- `-input=true` Ask for input if necessary. If false, will error if input was required.
- `-lock=false` Disable locking of state files during state-related operations.
- `-lock-timeout=<duration>` Override the time Terraform will wait to acquire a state lock. The default is `0s` (zero seconds), which causes immediate failure if the lock is already held by another process.
- `-no-color` Disable color codes in the command output.
- `-upgrade` Opt to upgrade modules and plugins as part of their respective installation steps. See the sections below for more details.

Copy a Source Module

By default, `terraform init` assumes that the working directory already contains a configuration and will attempt to initialize that configuration.

Optionally, init can be run against an empty directory with the `-from-module=MODULE-SOURCE` option, in which case the given module will be copied into the target directory before any other initialization steps are run.

This special mode of operation supports two use-cases:

- Given a version control source, it can serve as a shorthand for checking out a configuration from version control and then initializing the working directory for it.
- If the source refers to an *example* configuration, it can be copied into a local directory to be used as a basis for a new configuration.

For routine use it is recommended to check out configuration from version control separately, using the version control system's own commands. This way it is possible to pass extra flags to the version control system when necessary, and to perform other preparation steps (such as configuration generation, or activating credentials) before running `terraform init`.

Backend Initialization

During init, the root configuration directory is consulted for backend configuration (/docs/language/settings/backends/configuration.html) and the chosen backend is initialized using the given configuration settings.

Re-running init with an already-initialized backend will update the working directory to use the new backend settings. Either `-reconfigure` or `-migrate-state` must be supplied to update the backend configuration.

The `-migrate-state` option will attempt to copy existing state to the new backend, and depending on what changed, may result in interactive prompts to confirm migration of workspace states. The `-force-copy` option suppresses these prompts and answers "yes" to the migration questions. This implies `-migrate-state`.

The `-reconfigure` option disregards any existing configuration, preventing migration of any existing state.

To skip backend configuration, use `-backend=false`. Note that some other init steps require an initialized backend, so it is recommended to use this flag only when the working directory was already previously initialized for a particular backend.

The `-backend-config=...` option can be used for partial backend configuration (/docs/language/settings/backends/configuration.html#partial-configuration), in situations where the backend settings are dynamic or sensitive and so cannot be statically specified in the configuration file.

Child Module Installation

During init, the configuration is searched for `module` blocks, and the source code for referenced modules (/docs/language/modules/develop/index.html) is retrieved from the locations given in their `source` arguments.

Re-running init with modules already installed will install the sources for any modules that were added to configuration since the last init, but will not change any already-installed modules. Use `-upgrade` to override this behavior, updating all modules to the latest available source code.

To skip child module installation, use `-get=false`. Note that some other init steps can complete only when the module tree is complete, so it's recommended to use this flag only when the working directory was already previously initialized with its child modules.

Plugin Installation

Most Terraform providers are published separately from Terraform as plugins. During init, Terraform searches the configuration for both direct and indirect references to providers and attempts to install the plugins for those providers.

For providers that are published in either the public Terraform Registry (https://registry.terraform.io/) or in a third-party provider registry, `terraform init` will automatically find, download, and install the necessary provider plugins. If you cannot or do not wish to install providers from their origin registries, you can customize how Terraform installs providers using the provider installation settings in the CLI configuration (/docs/cli/config/config-file.html#provider-installation).

For more information about specifying which providers are required for each of your modules, see Provider Requirements (/docs/language/providers/requirements.html).

After successful installation, Terraform writes information about the selected providers to the dependency lock file (/docs/language/dependency-lock.html). You should commit this file to your version control system to ensure that when you run `terraform init` again in future Terraform will select exactly the same provider versions. Use the `-upgrade` option if you want Terraform to ignore the dependency lock file and consider installing newer versions.

You can modify `terraform init`'s plugin behavior with the following options:

- `-upgrade` Upgrade all previously-selected plugins to the newest version that complies with the configuration's version constraints. This will cause Terraform to ignore any selections recorded in the dependency lock file, and to take the newest available version matching the configured version constraints.
- `-get-plugins=false` — Skip plugin installation.

  > Note: Since Terraform 0.13, this option has been superseded by the `provider_installation` (/docs/cli/config/config-file.html#provider-installation) and `plugin_cache_dir` (/docs/cli/config/config-file.html#plugin_cache_dir) settings. It should not be used in Terraform versions 0.13+, and this option was removed in Terraform 0.15.

- `-plugin-dir=PATH` — Force plugin installation to read plugins *only* from the specified directory, as if it had been configured as a `filesystem_mirror` in the CLI configuration. If you intend to routinely use a particular filesystem mirror then we recommend configuring Terraform's installation methods globally (/docs/cli/config/config-file.html#provider-installation). You can use `-plugin-dir` as a one-time override for exceptional situations, such as if you are testing a local build of a provider plugin you are currently developing.
- `-lockfile=MODE` Set a dependency lockfile mode.

The valid values for the lockfile mode are as follows:

- readonly: suppress the lockfile changes, but verify checksums against the information already recorded. It conflicts with the `-upgrade` flag. If you update the lockfile with third-party dependency management tools, it would be useful to control when it changes explicitly.

Running `terraform init` in automation

For teams that use Terraform as a key part of a change management and deployment pipeline, it can be desirable to orchestrate Terraform runs in some sort of automation in order to ensure consistency between runs, and provide other interesting features such as integration with version control hooks.

There are some special concerns when running `init` in such an environment, including optionally making plugins available locally to avoid repeated re-installation. For more information, see the Running Terraform in Automation (https://learn.hashicorp.com/tutorials/terraform/automate-terraform?in=terraform/automation&utm_source=WEBSITE&utm_medium=WEB_IO&utm_offer=ARTICLE_PAGE&utm_content=DOCS) tutorial on HashiCorp Learn.

Passing a Different Configuration Directory

Terraform v0.13 and earlier also accepted a directory path in place of the plan file argument to `terraform apply`, in which case Terraform would use that directory as the root module instead of the current working directory.

That usage is still supported in Terraform v0.14, but is now deprecated and we plan to remove it in Terraform v0.15. If your workflow relies on overriding the root module directory, use the `-chdir` global option (./#switching-working-directory-with-chdir) instead, which works across all commands and makes Terraform consistently look in the given directory for all files it would normaly read or write in the current working directory.

If your previous use of this legacy pattern was also relying on Terraform writing the `.terraform` subdirectory into the current working directory even though the root module directory was overridden, use the `TF_DATA_DIR` environment variable (/docs/cli/config/environment-variables.html#tf_data_dir) to direct Terraform to write the `.terraform` directory to a location other than the current working directory.