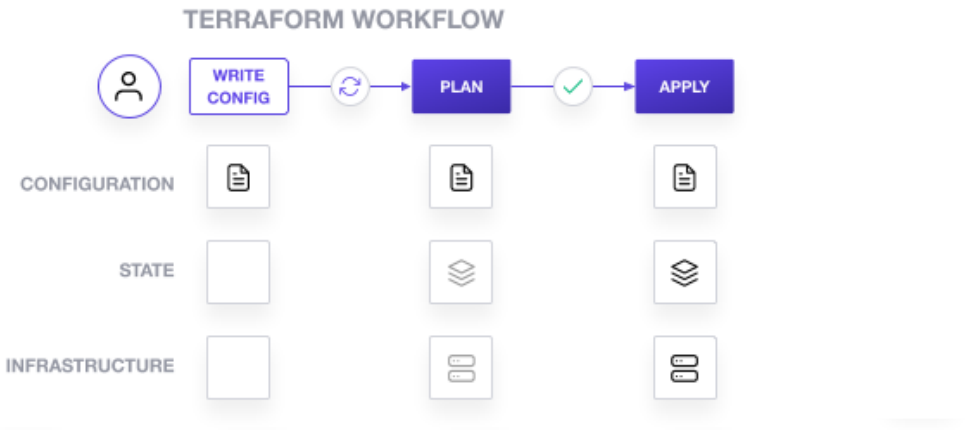
Import Terraform Configuration

developer.hashicorp.com/terraform/tutorials/certification-associate-tutorials/state-import

In this tutorial, you will import an existing Docker container and image into a Terraform project. By doing so, you will learn strategies and considerations for importing real-world infrastructure into Terraform.

When you create new infrastructure with Terraform, you will usually use the following workflow:

1. Write Terraform configuration that defines the infrastructure you want to create.
2. Review the Terraform plan to ensure the configuration will result in the expected infrastructure.
3. Apply the configuration to have Terraform create your infrastructure.



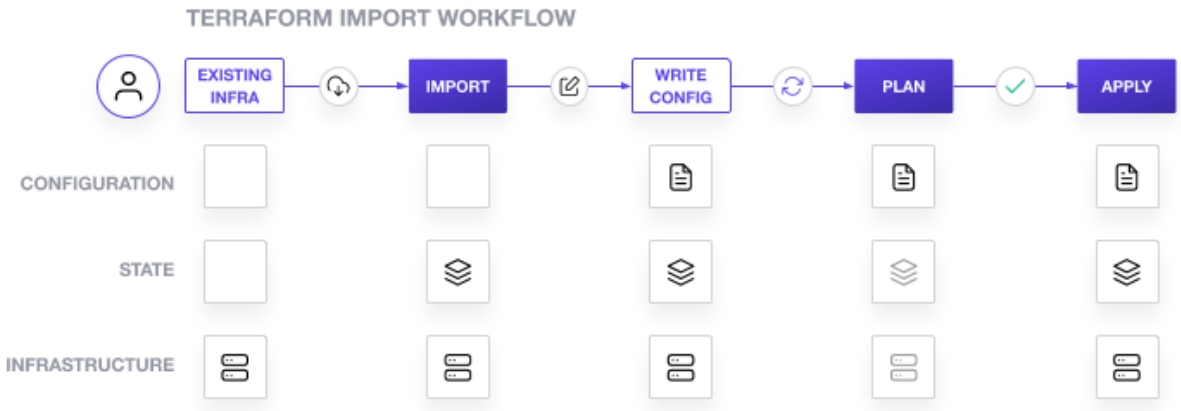
When you create infrastructure with Terraform, it stores information about your infrastructure in its state file. You can update your infrastructure by first changing your configuration, and then using Terraform to plan and apply the required changes. Terraform uses the information it stores in your state file to plan the changes it will make to your infrastructure.

Terraform also supports bringing existing infrastructure under its management. To do so, you can use the `import` command to migrate resources into your Terraform state file. The `import` command does not currently generate the configuration for the imported resource, so you must write the corresponding configuration block to map the imported resource to it.

Bringing existing infrastructure under Terraform's control involves five steps:

1. Identify the existing infrastructure you will import.
2. Import infrastructure into your Terraform state.
3. Write Terraform configuration that matches that infrastructure.

4. Review the Terraform plan to ensure the configuration matches the expected state and infrastructure.
5. Apply the configuration to update your Terraform state.



In this tutorial, first you will create a Docker container with the Docker CLI. Next, you will import it into a new Terraform project. Then you will update the container's configuration using Terraform before finally destroying it when you are done.

Warning: Importing infrastructure manipulates Terraform state in ways that could leave existing Terraform projects in an invalid state. Make a backup of your `terraform.tfstate` file and `.terraform` directory before using Terraform import on a real Terraform project, and store them securely.

Prerequisites

In order to follow this tutorial you will need the following.

1. The Terraform CLI.
2. Docker installed and running.
3. The git CLI.

Create a Docker container

Create a container named `hashicorp-learn` using the latest NGINX image from Docker Hub, and publish that container's port 80 (HTTP) to your local host system's port 8080. You will import this container in this tutorial.

```
$ docker run --name hashicorp-learn --detach --publish 8080:80 nginx:latest
```

Docker will output a message similar to the following as it downloads and runs the nginx image.

```
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
afb6ec6fdc1c: Pull complete
dd3ac8106a0b: Pull complete
8de28bdda69b: Pull complete
a2c431ac2669: Pull complete
e070d03fd1b5: Pull complete
Digest: sha256:883874c218a6c71640579ae54e6952398757ec65702f4c8ba7675655156fcca6
Status: Downloaded newer image for nginx:latest
e7ba41fd94e51c501533241e4cffd307fbda81c5b402c372d989c4578518d2e5
```

Verify that the container is running with `docker ps`.

```
$ docker ps --filter="name=hashicorp-learn"
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS                  NAMES
e7ba41fd94e5        nginx:latest        "/docker-entrypoint.…"   About a minute
ago   Up 59 seconds       0.0.0.0:8080->80/tcp   hashicorp-learn
```

Visit the address `0.0.0.0:8080` in your web browser to see the NGINX default index page.

Now you have a Docker image and container to import into your project and manage with Terraform.

Import the container into Terraform

Now, clone the example repository.

```
$ git clone https://github.com/hashicorp/learn-terraform-import.git
```

Next, change the directory.

```
$ cd learn-terraform-import
```

In this directory are three Terraform configuration files that you will use in this tutorial:

- `terraform.tf` configures Terraform and provider versions
- `main.tf` configures the Docker provider
- `docker.tf` will contain the configuration necessary to manage the Docker container you created in the previous step

Initialize your Terraform project with `terraform init`.

```
$ terraform init
```

Next, define an empty `docker_container` resource in your `docker.tf` file, which represents a Docker container with the Terraform resource ID `docker_container.web`.

docker.tf

```
resource "docker_container" "web" {}
```

Next, run `docker ps` to find the name of the container you want to import - in this case, the container you created in the previous step.

```
$ docker ps --filter "name=hashicorp-learn"
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS                  NAMES
d45091b71212        nginx:latest        "nginx -g 'daemon of…"   18 minutes ago
Up 18 minutes       0.0.0.0:8080->80/tcp   hashicorp-learn
```

Now run `terraform import` to attach the existing Docker container to the `docker_container.web` resource you just created. Terraform import requires this Terraform resource ID and the full Docker container ID. In the following example, the command `docker inspect --format="{{.ID}}" hashicorp-learn` returns the full SHA256 container ID.

```
$ terraform import docker_container.web $(docker inspect --format="{{.ID}}"
hashicorp-learn)
docker_container.web: Importing from ID
"d45091b7121266f0c0e69dd9985acdefd110a66bcedbd03797e3606fb0a7d7ee"...
docker_container.web: Import prepared!
  Prepared docker_container for import
docker_container.web: Refreshing state...
[id=d45091b7121266f0c0e69dd9985acdefd110a66bcedbd03797e3606fb0a7d7ee]

Import successful!

The resources that were imported are shown above. These resources are now in
your Terraform state and will henceforth be managed by Terraform.
```

Note: The ID accepted by `terraform import` varies by resource type and is documented in the provider documentation for any resource that can be imported to Terraform. For this example, consult the Docker provider documentation.

Now verify that the container has been imported into your Terraform state by running `terraform show`.

```
$ terraform show
# docker_container.web:
resource "docker_container" "web" {
    command           = [
      "nginx",
      "-g",
      "daemon off;",
    ]

    ## ...

    ports {
      external = 8080
      internal = 80
      ip       = "0.0.0.0"
      protocol = "tcp"
    }
}
```

This state contains everything that Terraform knows about the Docker container you just imported. However, Terraform import does **not** create the configuration for the resource.

Create configuration

You'll need to create Terraform configuration before you can use Terraform to manage this container.

Run `terraform plan`. Terraform will show errors for the missing required arguments `image` and `name`. Terraform cannot generate a plan for a resource that is missing required arguments.

```
$ terraform plan

Error: Missing required argument

  on docker.tf line 1, in resource "docker_container" "web":
   1: resource "docker_container" "web" { }

The argument "name" is required, but no definition was found.


Error: Missing required argument

  on docker.tf line 1, in resource "docker_container" "web":
   1: resource "docker_container" "web" { }

The argument "image" is required, but no definition was found.
```

There are two approaches to update the configuration in `docker.tf` to match the state you imported. You can either accept the entire current state of the resource into your configuration as-is or cherry-pick the required attributes into your configuration one at a time. You may find both of these approaches useful in different circumstances.

- Using the current state is often faster, but can result in an overly verbose configuration since every attribute is included in the state, whether it is necessary to include in your configuration or not.
- Cherry-picking the required attributes can lead to more manageable configuration, but requires you to understand which attributes need to be set in the configuration.

Try either or both of these approaches using the tabs below.

To use current state as configuration, you will:

1. After importing the resource, copy the Terraform state into a configuration file. You will base the configuration for the resource on its definition in state.
2. Run `terraform plan` to identify and remove read-only configuration arguments.
3. Re-run `terraform plan` to confirm the configuration is correct.
4. Run `terraform apply` to finish synchronizing your configuration, state, and infrastructure.

Use `terraform show` to copy your Terraform state into your `docker.tf` file.

```
$ terraform show -no-color > docker.tf
```

Warning: The `>` symbol will replace the entire contents of docker.tf with the output of the `terraform show` command. While this works for this example, importing a resource into a configuration that already manages resources will require you to edit the output of `terraform show` to remove existing resources whose configuration you do not want to replace wholesale, and merge the new resources into your existing configuration.

Inspect the `docker.tf` file to see that its contents have been replaced with the output of the `terraform show` command you just ran.

Now run `terraform plan`. Terraform will show warnings and errors about a deprecated argument ('links'), and several read-only arguments (`ip_address`, `network_data`, `gateway`, `ip_prefix_length`, `id`).

```
$ terraform plan

Warning: "links": [DEPRECATED] The --link flag is a legacy feature of Docker. It
may eventually be removed.

  on docker.tf line 2, in resource "docker_container" "web":
   2: resource "docker_container" "web" {



Error: "ip_prefix_length": this field cannot be set

  on docker.tf line 2, in resource "docker_container" "web":
   2: resource "docker_container" "web" {



Error: "ip_address": this field cannot be set

  on docker.tf line 2, in resource "docker_container" "web":
   2: resource "docker_container" "web" {



Error: "network_data": this field cannot be set

  on docker.tf line 2, in resource "docker_container" "web":
   2: resource "docker_container" "web" {



Error: "gateway": this field cannot be set

  on docker.tf line 2, in resource "docker_container" "web":
   2: resource "docker_container" "web" {



Error: : invalid or unknown key: id

  on docker.tf line 2, in resource "docker_container" "web":
   2: resource "docker_container" "web" {
```

These read-only arguments are values that Terraform stores in its state for Docker containers but that it cannot set via configuration since they are managed internally by Docker. Terraform can set the `links` argument with configuration, but still throws a warning because it is deprecated and may not be supported by future versions of the Docker provider.

Remove all six of these attributes from your `docker.tf` configuration file before continuing with the next step.

```
          "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
          "PKG_RELEASE=1~buster",
      ]
-     gateway           = "172.17.0.1"
      group_add         = []
      hostname          = "0cc3203b4634"
-     id                =
"0cc3203b46342f0adf7ed7a30d41311aed65e5e8632d29ca5e6107ee7be39f16"
      image             =
"sha256:2622e6cca7ebbb6e310743abce3fc47335393e79171b9d76ba9d4f446ce7b163"
-     ip_address        = "172.17.0.2"
-     ip_prefix_length  = 16
      ipc_mode          = "private"
-     links             = []
      log_driver        = "json-file"
      log_opts          = {}
      max_retry_count   = 0
      memory            = 0
      memory_swap       = 0
      name              = "hashicorp-learn"
-     network_data      = [
-         {
-             gateway          = "172.17.0.1"
-             ip_address       = "172.17.0.2"
-             ip_prefix_length = 16
-             network_name     = "bridge"
-         },
-     ]
      network_mode      = "default"
      privileged        = false
      publish_all_ports = false
```

When importing real infrastructure, consult the provider documentation to learn what each argument does. This will help you to determine how to handle any errors or warnings from the plan step. For instance, the documentation for the `links` argument is in the Docker provider documentation.

Now verify that the errors have been resolved by re-running `terraform plan`.

```
$ terraform plan
docker_container.web: Refreshing state...
[id=772ad3901a62667a28f4d7e6cc52a55fbadad13d544be811d4bc18bf455e1909]

Terraform used the selected providers to generate the following execution plan.
Resource actions are indicated with the following symbols:
-/+ destroy and then create replacement

Terraform will perform the following actions:

  # docker_container.web must be replaced
-/+ resource "docker_container" "web" {
      + attach            = false
      + bridge            = (known after apply)
      + container_logs    = (known after apply)
      - dns               = [] -> null
      - dns_opts          = [] -> null
      - dns_search        = [] -> null
      + env               = (known after apply) # forces replacement
      + exit_code         = (known after apply)
      ## ...
    }

Plan: 1 to add, 0 to change, 1 to destroy.
```

Notice that Terraform plans to destroy then recreate your Docker container because `env` is not defined.

Add `env` to your `docker_container.web` resource.

```
 resource "docker_container" "web" {
     command           = [
         "nginx",
         "-g",
         "daemon off;",
     ]
     cpu_shares        = 0
     dns               = []
     dns_opts          = []
     dns_search        = []
     entrypoint        = [
         "/docker-entrypoint.sh",
     ]
     group_add         = []
+    env               = []

     ## ...
 }
```

Now verify that the errors have been resolved by re-running `terraform plan`.

```
$ terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

docker_container.web: Refreshing state...
[id=d45091b7121266f0c0e69dd9985acdefd110a66bcedbd03797e3606fb0a7d7ee]

------------------------------------------------------------------------

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  ~ update in-place

Terraform will perform the following actions:

  ## docker_container.web will be updated in-place
  ~ resource "docker_container" "web" {
      + attach            = false
        command           = [
            "Nginx",
            "-g",
            "daemon off;",
        ]

        ## ...

        ports {
            external = 8080
            internal = 80
            ip       = "0.0.0.0"
            protocol = "tcp"
        }
}

Plan: 0 to add, 1 to change, 0 to destroy.

## ...
```

The plan should now execute successfully. Notice that the plan indicates that Terraform will update the container in place to add the `attach`, `logs`, `must_run`, and `start` attributes.

Terraform uses these attributes to create Docker containers, but Docker doesn't store them. As a result, `terraform import` didn't load their values into state. When you plan and apply your configuration, the Docker provider will assign the default values for these attributes and save them in state, but they won't affect the running container.

Note: It is not always clear when changes are safe just by reading the provider documentation. You must understand the lifecycle of the underlying resource in order to know if a given change is safe to apply.

Apply the changes and finish the process of syncing your Terraform configuration and state with the Docker container they represent. Remember to confirm the apply step with a `yes`.

```
$ terraform apply
docker_container.web: Refreshing state...
[id=d45091b7121266f0c0e69dd9985acdefd110a66bcedbd03797e3606fb0a7d7ee]

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  ~ update in-place

Terraform will perform the following actions:

  # docker_container.web will be updated in-place
  ~ resource "docker_container" "web" {
      + attach            = false
        command           = [
            "nginx",
            "-g",

        ## ...

        ports {
            external = 8080
            internal = 80
            ip       = "0.0.0.0"
            protocol = "tcp"
        }
    }

Plan: 0 to add, 1 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

docker_container.web: Modifying...
[id=d45091b7121266f0c0e69dd9985acdefd110a66bcedbd03797e3606fb0a7d7ee]
docker_container.web: Modifications complete after 0s
[id=d45091b7121266f0c0e69dd9985acdefd110a66bcedbd03797e3606fb0a7d7ee]

Apply complete! Resources: 0 added, 1 changed, 0 destroyed.
```

Now your configuration file, Terraform state, and the container are all in sync, and you can use Terraform to manage the Terraform container as you normally would. Because this apply step changed the container's state rather than destroying and recreating it, the container ID didn't change, and the container continued running normally during the process.

Since the approach shown here loads all of the attributes represented in Terraform state, your configuration includes optional attributes whose values are the same as their defaults. Which attributes are optional, and their default values, will vary from provider to provider, and can be found in the provider documentation.

Optionally, you can remove all of these attributes, keeping only the required attributes and those for whom your container differs from the default values. After removing these unnecessary attributes, your configuration should match the following.


docker.tf

```
resource "docker_container" "web" {
    image =
"sha256::602e111c06b6934013578ad80554a074049c59441d9bcd963cb4a7feccede7a5"
    name  = "hashicorp-learn"

    ports {
      external = 8080
      internal = 80
    }
}
```

Note: Your image ID may be different from the one shown here.

At this point, running `terraform plan` or `terraform apply` will show no changes, and your configuration only includes the minimum set of attributes needed to recreate the container as-is.

To cherry-pick the configuration for your Docker container, you will add the missing required attributes which caused the errors in your plan. Terraform can't generate a plan without all of the required attributes for your resource.

Run `terraform show` to find the correct values for the missing attributes 'image' and 'name'.

```
$ terraform show
# docker_container.web:
resource "docker_container" "web" {
  ## ...
  image             =
"sha256:4392e5dad77dbaf6a573650b0fe1e282b57c5fba6e6cea00a27c7d4b68539b81"
  ## ...
  name              = "hashicorp-learn"
  ## ...
}
```

Copy these values into the `"docker_container" "web"` block in `docker.tf`. Be sure to use the image ID from the output of `terraform show`, not the one shown here.

docker.tf

```
resource "docker_container" "web" {
  name  = "hashicorp-learn"
  image =
"sha256:4392e5dad77dbaf6a573650b0fe1e282b57c5fba6e6cea00a27c7d4b68539b81"
}
```

This will resolve the errors from missing required attributes, but your configuration still won't match the Terraform state or Docker container.

Run `terraform plan` again to see those differences. The plan will succeed, but applying it would destroy the existing container and add a new one with a different configuration, instead of bringing the existing container under Terraform's control.

```
$ terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

## ...

Terraform will perform the following actions:

  # docker_container.web must be replaced
-/+ resource "docker_container" "web" {
      + attach            = false
      + bridge            = (known after apply)
      ## ...

      + env               = (known after apply) # forces replacement

      ## ...

      - ports { # forces replacement
          - external = 8080 -> null
          - internal = 80 -> null
          - ip       = "0.0.0.0" -> null
          - protocol = "tcp" -> null
        }
    }

Plan: 1 to add, 0 to change, 1 to destroy.

## ...
```

Your configuration is missing values for `env` and `ports`. The values loaded from state tell us that Docker is exposing the container's port `80` as port `8080` in your host system. The output of Terraform plan shows that changes to the `env` and `port` attributes "forces replacement" of the container.

Resolve this by adding the `env` and `ports` attributes to the `docker_container.web` resource in `docker.tf`.

docker.tf

```
resource "docker_container" "web" {
  name  = "hashicorp-learn"
  image =
"sha256:602e111c06b6934013578ad80554a074049c59441d9bcd963cb4a7feccede7a5"

  env  = []

  ports {
    external = 8080
    internal = 80
  }
}
```

You do not need to include values for `ip` or `protocol` because these attributes are optional, and the current state is the same as their default values.

Note: Which attributes are optional, and their default values, will vary from provider to provider. Optional values for the docker_container resource type can be found in the Docker provider documentation.

Run `terraform plan` again to compare your new configuration to the state you imported earlier:

```
$ terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

## ...

Terraform will perform the following actions:

  # docker_container.web will be updated in-place
  ~ resource "docker_container" "web" {
      + attach            = false
        command           = [
            "nginx",

      ## ...

        log_driver        = "json-file"
        log_opts          = {}
      + logs              = false
        max_retry_count   = 0
        memory            = 0
        memory_swap       = 0
      + must_run          = true
        name              = "hashicorp-learn"

      ## ...
    }

Plan: 0 to add, 1 to change, 0 to destroy.

## ...
```

Notice that the plan indicates that Terraform will update the container in place to add the `attach`, `logs`, `must_run`, and `start` attributes.

Terraform uses these attributes to create Docker containers, but Docker doesn't store them, and so Terraform import didn't load values for them. When you plan and apply your configuration, the Docker provider assigns the default values for these attributes and saves them in state, but they don't affect the running container.

Note: It is not always clear when changes are safe just by reading the provider documentation. You must understand the lifecycle of the underlying resource in order to know if a given change is safe to apply.

Apply the changes to finish syncing your Terraform configuration and state with the Docker container they represent. Remember to confirm the apply step with a `yes`.

```
$ terraform apply
docker_container.web: Refreshing state...
[id=d45091b7121266f0c0e69dd9985acdefd110a66bcedbd03797e3606fb0a7d7ee]

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  ~ update in-place

Terraform will perform the following actions:

  # docker_container.web will be updated in-place
  ~ resource "docker_container" "web" {
      + attach            = false
        ## ...
        ports {
            external = 8080
            internal = 80
            ip       = "0.0.0.0"
            protocol = "tcp"
        }
    }

Plan: 0 to add, 1 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

docker_container.web: Modifying...
[id=d45091b7121266f0c0e69dd9985acdefd110a66bcedbd03797e3606fb0a7d7ee]
docker_container.web: Modifications complete after 0s
[id=d45091b7121266f0c0e69dd9985acdefd110a66bcedbd03797e3606fb0a7d7ee]

Apply complete! Resources: 0 added, 1 changed, 0 destroyed.
```

At this point, running `terraform plan` or `terraform apply` will show no further changes, and you can now manage the container with Terraform as you would any other resource.

If you want to try the other method for generating configuration before moving on, use the following steps to revert the changes you made in the previous section, then switch to the other tab.

1. Remove everything inside the `"docker_container" "web"` block in `docker.tf`, so that the file only contains `resource "docker_container" "web" { }`.
2. Remove the container from your Terraform project's state by running: `terraform state rm "docker_container.web"`.
3. Import the container to Terraform state again by running the command `terraform import docker_container.web $(docker inspect -f {{.ID}} hashicorp-learn)`.

Otherwise, proceed with the next step to verify your configuration.

Verify import

Regardless of which method you used, your Docker container is now managed by Terraform. Use the Docker CLI to inspect the container.

```
$ docker ps --filter "name=hashicorp-learn"
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS                  NAMES
fac6b3ddb49d        nginx:latest        "nginx -g 'daemon of…"   11 minutes ago
Up 11 minutes       0.0.0.0:8080->80/tcp   hashicorp-learn
```

Note the "Status" — the container has been up and running since it was created, so you know that it was not restarted when you imported it into Terraform. The ID has not changed either — this is the same container you created at the beginning of this tutorial.

Visit `0.0.0.0:8080` in your web browser to verify that the container is still working as intended.

Create image resource

In some cases, you can bring resources under Terraform's control without using the `terraform import` command. This is often the case for resources that are defined by a single unique ID or tag, such as Docker images.

In your `docker.tf` file, the `docker_container.web` resource specifies the SHA256 hash ID of the image used to create the container. This is how Docker stores the image ID internally, and so `terraform import` loaded the image ID directly into your state. However the image ID is not as human readable as the image tag or name, and it may not match your intent. For example, you might want to use the latest version of the "nginx" image.

Retrieve the image's tag name by running the following command. Replace the image ID with the image ID from `docker.tf`.

```
$ docker image inspect sha256:43example81 -f {{.RepoTags}}
[nginx:latest]
```

Then add the following configuration to your `docker.tf` file to represent this image as a resource.

docker.tf

```
resource "docker_image" "nginx" {
  name         = "nginx:latest"
}
```

Warning: Do not replace the `image` value in the `docker_container.web` resource yet, or Terraform will destroy and recreate your container. Since Terraform hasn't loaded the `docker_image.nginx` resource into state yet, it does not have an image ID to compare with the hardcoded one, which will cause Terraform to assume the container must be replaced. You can work around this situation by creating the image first, then updating the container to use it, as shown in this tutorial.

Run `terraform apply` to create an image resource in state. Remember to confirm the apply step with a `yes`.

```
$ terraform apply
docker_container.web: Refreshing state...
[id=023afc10768ab8eeaf646d6a3ac47b52a15af764367ded41702ef9cf5b91a976]

## ...

Terraform will perform the following actions:

  # docker_image.nginx will be created
  + resource "docker_image" "nginx" {
      + id     = (known after apply)
      + latest = (known after apply)
      + name   = "nginx:latest"
    }

## ...

Apply complete! Resources: 1 added, 0 changed, 0 destroyed!
```

Now that Terraform has created a resource for the image, you can reference it in your container's configuration. Change the `image` value for `docker_container.web` to reference the new image resource.

docker.tf

```
resource "docker_container" "web" {
  name  = "hashicorp-learn"
  image = docker_image.nginx.latest

  ## ...
}
```

Since `docker_image.nginx.latest` will match the hardcoded image ID you replaced. Running `terraform apply` at this point will show no changes.

```
$ terraform apply
docker_image.nginx: Refreshing state...
[id=sha256:4392e5dad77dbaf6a573650b0fe1e282b57c5fba6e6cea00a27c7d4b68539b81nginx:l
atest]
docker_container.web: Refreshing state...
[id=e7ba41fd94e51c501533241e4cffd307fbda81c5b402c372d989c4578518d2e5]

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
```

Note: If the image ID for the tag `nginx:latest` changed between the time you first created the Docker container and when you run this command, Docker will destroy the container and then recreate it with the new image.

Manage the container with Terraform

Now that Terraform manages the Docker container, use Terraform to change the its configuration.

In your `docker.tf` file, change the container's external port from `8080` to `8081`.

docker.tf

```
resource "docker_container" "web" {
  name  = "hashicorp-learn"
  image = docker_image.nginx.latest

  ports {
    external = 8081
    internal = 80
  }
}
```

Apply the change. This will cause Terraform to destroy and recreate the container with the new port configuration. Remember to confirm the apply step with a `yes`.

```
$ terraform apply
docker_container.web: Refreshing state...
[id=75278f99c53a6b39e94127d2c25f7dee13f97a4af89c52d74bff9dc783b3cce1]

## ...

Plan: 1 to add, 0 to change, 1 to destroy.

## ...

docker_container.web: Destroying...
[id=75278f99c53a6b39e94127d2c25f7dee13f97a4af89c52d74bff9dc783b3cce1]
docker_container.web: Destruction complete after 1s
docker_container.web: Creating...
docker_container.web: Creation complete after 1s
[id=023afc10768ab8eeaf646d6a3ac47b52a15af764367ded41702ef9cf5b91a976]

Apply complete! Resources: 1 added, 0 changed, 1 destroyed.
```

Now verify that the container has been replaced with a new one with the new configuration by running `docker ps` or visiting `0.0.0.0:8081` in your web browser.

```
$ docker ps --filter "name=hashicorp-learn"
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS                  NAMES
023afc10768a        4392e5dad77d        "nginx -g 'daemon of…"   3 minutes ago
Up 3 minutes        0.0.0.0:8081->80/tcp   hashicorp-learn
```

Notice that the container ID has changed. Because changing the port configuration required destroying and recreating it, this is a completely new container.

Destroy infrastructure

You have now imported your Docker container and the image used to create it into Terraform.

Destroy the container and image by running `terraform destroy`. Remember to confirm the destroy step by responding `yes` when prompted.

```
$ terraform destroy
docker_image.nginx: Refreshing state...
[id=sha256:9beeba249f3ee158d3e495a6ac25c5667ae2de8a43ac2a8bfd2bf687a58c06c9nginx:l
atest]
docker_container.web: Refreshing state...
[id=3fe1cb2e5326c31bac9250f6d09eade77945ee07ccea025d6424d91a89f98557]

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  - destroy

Terraform will perform the following actions:

  # docker_container.web will be destroyed
  - resource "docker_container" "web" {
      - attach            = false -> null

      ## ...
    }

Plan: 0 to add, 0 to change, 2 to destroy.

Do you really want to destroy all resources?
  Terraform will destroy all your managed infrastructure, as shown above.
  There is no undo. Only 'yes' will be accepted to confirm.

  Enter a value: yes

docker_container.web: Destroying...
[id=3fe1cb2e5326c31bac9250f6d09eade77945ee07ccea025d6424d91a89f98557]
docker_container.web: Destruction complete after 1s
docker_image.nginx: Destroying...
[id=sha256:9beeba249f3ee158d3e495a6ac25c5667ae2de8a43ac2a8bfd2bf687a58c06c9nginx:l
atest]
docker_image.nginx: Destruction complete after 0s

Destroy complete! Resources: 2 destroyed.
```

Finally, run `docker ps` to validate that the container was destroyed.

```
$ docker ps --filter "name=hashicorp-learn"
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
```

Tip: Since you added the image to your Terraform configuration as well as the container, the image will be removed from Docker as well as the container. If there were another container using the same image, the destroy step would fail. Remember that importing a resource into Terraform means that Terraform will manage the entire lifecycle of the resource, including destruction.

Limitations and other considerations

There are several important things to consider when importing resources into Terraform.

- Terraform import can only know the current state of infrastructure as reported by the Terraform provider. It does not know:
    - whether the infrastructure is working correctly
    - the intent of the infrastructure
    - changes you've made to the infrastructure that aren't controlled by Terraform — for example, the state of a Docker container's filesystem.
- Importing involves manual steps which can be error prone, especially if the person importing resources lacks the context of how and why those resources were created in the first place.
- Importing manipulates the Terraform state file, you may want to create a backup before importing new infrastructure.
- Terraform import doesn't detect or generate relationships between infrastructure.
- Terraform doesn't detect default attributes that don't need to be set in your configuration.
- Not all providers and resources support Terraform import.
- Just because infrastructure has been imported into Terraform does not mean that it can be destroyed and recreated by Terraform. For example, the imported infrastructure could rely on other unmanaged infrastructure or configuration.
- You may need to set local variables equivalent to the remote workspace variables to import to a remote backend. The `import` command always runs locally—unlike commands like `apply`, which run inside your Terraform Cloud environment. Because of this,`import` will not have access to information from the remote backend, such as workspace variables, unless you set them locally.

Following Infrastructure as Code (IaC) best practices such as immutable infrastructure can help prevent many of these problems, but infrastructure created by hand is unlikely to follow IaC best practices.

Tools such as Terraformer to automate some manual steps associated with importing infrastructure. However, these tools are not part of Terraform itself, and not endorsed or supported by HashiCorp.

Next steps

Now that you have imported infrastructure into Terraform, you may like to:

- Read the Terraform Import documentation.
- Migrate configuration to Terraform Cloud.