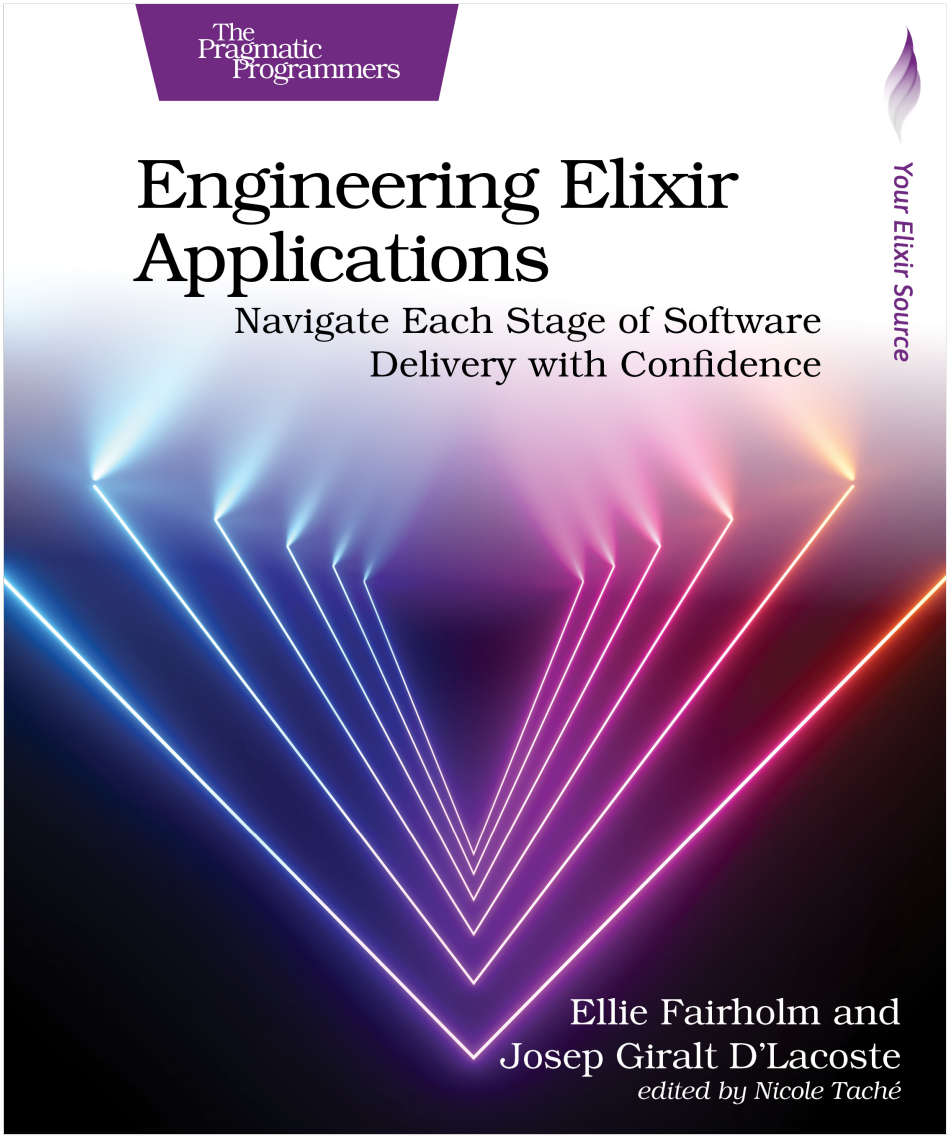


This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

Importing an Existing Infrastructure Resource with Terraform

You might think that it would be extremely time-consuming to replicate already created remote resources in Terraform. Well, it's not. You can use one simple Terraform block – an `import` block – and two simple commands – `terraform plan` and `terraform apply` – to do so.

When we introduced Terraform in chapter 2, we said that when dealing with an infrastructure resource, you should always go to the Terraform registry and find the documentation of the provider and its specific resources. We also said that each resource's docs give you import instructions. Keeping that in mind, the first resource you'll import is your EC2 instance. The Terraform resource related to EC2 instances is called `aws_instance`. Go to that resource's documentation[2] and you'll find the import instructions at the bottom of the page.

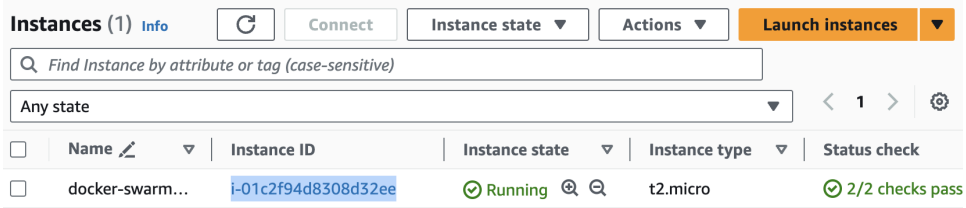
You'll see that there are two different ways of importing a resource: either with an `import` block or by executing the `terraform import` command. We prefer using an `import` block, as it allows you to see a plan of the import that you want to do before actually importing the resources. The `terraform import` command, on the other hand, imports the resource directly into your state without letting you see what it's importing first. Using an `import` block is safer and adheres to the Terraform work cycle that we mentioned in chapter 2, where you should plan your state modifications before you make them.

In the `aws_instance` resource documentation, you'll see that the `import` block has the following structure:

```
import {
  to = RESOURCE_ADDRESS
  id = ID
}
```

The `to` argument refers to the address that will be used in your Terraform configuration/state for this resource. It expects a `RESOURCE_ADDRESS` as a value . The `id` argument refers to ID of the remote resource that you're wanting to import. In this case, the ID will be the instance ID of your EC2 instance, which you can find in the "Instances" tab of the EC2 dashboard (shown in the following figure).

2. https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/instance#import



So, to import your EC2 resource into your Terraform configuration, you need to use an `import` block. To create this `import` block, you need a Terraform configuration file. You'll create one in the next section using Terraform modules.

Setting Up Terraform and Importing Your EC2 Instance

A Terraform module is a collection of `.tf` files kept together in a directory as a way to encapsulate and organize Terraform configurations into a self-contained and reusable unit. By defining your EC2 setup within a Terraform module, you can re-use that configuration to create the same resource across various different AWS environments. This is important in maintaining environment integrity as it ensures consistency when deploying resources. In this book, we'll only be showing you how to create your production environment, but in a real-world project you'd likely have a staging environment also. You've actually already had some practice with modules without knowing it. The project management Terraform code you created in chapter 2 is also a module.

To create your new AWS infrastructure module, create a new `cloud/aws/compute/swarm` nested directory (e.g. where `aws` is inside `cloud` and so on) inside your `modules` folder. In that module, create a new `main.tf` file. This is where your EC2 configuration will live.

The first step in adding your EC2 instance to your Terraform configuration is to add the `aws` provider to your new `main.tf` file. As we mentioned in chapter 2, there is a "Use Provider" button in the top right-hand corner of any resources's docs that gives you the installation instructions for that resource's provider. Copy the code that is shown when you click on this button and paste it into your `main.tf` file. You do not need to copy over the `provider` block, as you'll not be adding any configuration options. Your `main.tf` file should look something like this:

```
# in modules/cloud/aws/compute/swarm/main.tf

terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "5.13.1"
    }
```

```
  }
}
```

Great. Any time you create a new Terraform configuration or add any new providers to your configurations, you need to run the command terraform init. Think of it like a mix deps.get. It is important that you run terraform init in a folder that is specific to your production environment. If you were to initialize your production Terraform configuration inside your cloud/aws/compute/swarm module, you could not then create a staging configuration in the same directory. In other words, your module would not be re-usable. To encapsulate your production environment configuration in one, isolated place, create a new folder called environments in the root of your project and a subfolder called production inside environments. In this environments/production folder, create a new main.tf file and import your swarm module. You can do this using the module <BLOCK TYPE> and its source attribute. Your environments/production/main.tf file should look this this:

```
# in environments/production/main.tf

module "swarm" {
  source = "../../modules/cloud/aws/compute/swarm"
}
```

Now cd into your new environments/production folder and run terraform init. Your initial provider configuration is complete!

However, before you can try importing your EC2 instance, there is another crucial step to take: configuring your AWS access keys. Without configuring these, Terraform will throw an error because it will not know which AWS account to use when looking for the resource to import. To create your access keys, follow these instructions.[3] Don't worry about the warning of not using root keys. Ideally, you would use access keys that have IAM roles/permissions attached to them but that it not something that we will focus on in this book. Using the root access keys that do not restrict your permissions will simplify what we are doing in this chapter and throughout the book. You can find information on how to follow a more secure approach by assuming IAM roles.[4] Once you have created the keys, either add them to your shell profile or export them in your terminal like so:

```
$ export AWS_ACCESS_KEY_ID="YOUR_AWS_ACCESS_KEY_ID"
$ export AWS_SECRET_ACCESS_KEY="YOUR_AWS_SECRET_ACCESS_KEY"
$ export AWS_REGION=eu-west-1
```

3. https://docs.aws.amazon.com/accounts/latest/reference/root-user-access-key.html
4. https://developer.hashicorp.com/terraform/tutorials/aws/aws-assumerole

To validate that you have exported the keys properly, you can run `env | grep AWS`. You should see the following result:

```
$ env | grep AWS
AWS_ACCESS_KEY_ID=******************************
AWS_REGION=eu-west-1
AWS_SECRET_ACCESS_KEY=******************************
```

Okay, you've configured your initial Terraform provider and have set your AWS keys. Now you must configure the `import` block. `import` blocks must be defined in the root module. Copy and paste the example `import` block given in the `aws_instance` documentation so that it sits under your current `module` block in your `modules/environments/production/main.tf` file. Change the value of the `to` argument to `module.swarm.aws_instance.my_swarm` to tell Terraform where you will put the resource (inside your `swarm` module), and then change the value of the `id` argument to be the ID of your EC2 instance. Your `modules/environments/production/main.tf` file should now look something like this:

```
# in environments/production/main.tf

module "swarm" {
  source = "../../modules/cloud/aws/compute/swarm"
}

import {
  to = module.swarm.aws_instance.my_swarm
  id = "YOUR_EC2_INSTANCE_ID"
}
```

We mentioned that `import` blocks allow you to plan your import before you do it. So, let's do just that. Run `terraform plan` inside your `environments/production` folder as we have done here:

```
$  terraform plan
Planning failed. Terraform encountered an error while generating this plan.
╷
│ Error: Import block target does not exist
│
│   on main.tf line 7:
│    7: import {
│
│ The target for the given import block does not exist. The specified target
│ is within a module, and must be defined as a resource within that module
│ before anything can be imported.
```