

## Lab 2: Remote State

Duration: 10 minutes

This lab demonstrates how to read state from another Terraform project. It uses the simplest possible example: a project on the same local disk.

- Task 1: Create a Terraform configuration that defines an output
- Task 2: Read an output value from that project's state

### Prerequisites

This lab only requires a copy of [Terraform](#). It doesn't require any cloud provider credentials.

### Task 1: Create a Terraform configuration that defines an output

For this task, you'll create two Terraform configurations in two separate directories. One will read from the other (using state files).

#### Step 2.1.1

In this step, you'll create a Terraform project on disk that does nothing but emit an output. It should emit `public_ip` which can be a hard-coded value (for simplicity).

The project should consist of a single file which can be named something like `primary/main.tf`.

```
$ mkdir primary
$ touch primary/main.tf
$ cd primary
```

The contents of `main.tf` are a single output for `public_ip`. This is the entire contents of the file.

```
# primary/main.tf
output "public_ip" {
  value = "8.8.8.8"
}
```

#### Step 2.1.2

Generate a state file for the project. Within that project, run `terraform init` and `apply`. You should see a `terraform.tfstate` file after running these commands.

Run the standard terraform commands within the `primary` project.

```
$ terraform init
$ terraform apply
```

## Task 2: Read an output value from that project's state

### Step 2.2.1

Create a new Terraform configuration that uses a data source to read the configuration from the `primary` project.

Create a second directory named `secondary`.

```
$ cd ../
$ mkdir secondary
$ cd secondary
$ touch main.tf
```

Define a `terraform_remote_state` data source that uses a `local` backend which points to the `primary` project.

```
# secondary/main.tf
# Read state from another Terraform config's state
data "terraform_remote_state" "primary" {
  backend = "local"
  config {
    path = "../primary/terraform.tfstate"
  }
}
```

Initialize the `secondary` project with `init`.

```
$ terraform init
```

### Step 2.2.2

Declare the `public_ip` as an output.

Within `secondary/main.tf`, define an output whose value is the `public_ip` from the data source you just defined.

```
output "primary_public_ip" {  
  value = "${data.terraform_remote_state.primary.public_ip}"  
}
```

Finally, run `apply`. You should see the IP address you defined in the `primary` configuration.

```
$ terraform apply  
  
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.  
  
Outputs:  
  
primary_public_ip = 8.8.8.8
```

# Lab 3: Store State Remotely on S3

Duration: 20 minutes

This lab demonstrates how to store state on cloud storage and read from it. You'll bootstrap an existing project onto cloud storage and use a second project to read from it.

- Task 1: Create an S3 bucket for state storage
- Task 2: Bootstrap the configuration to store its own state on S3
- Task 3: Create another Terraform config that reads from that state on S3

## Prerequisites

For this lab, we'll assume that you've installed [Terraform](#) and that you have defined your AWS credentials in your environment, including a region. See [Configuring the AWS CLI](#) in the AWS docs for details.

```
export AWS_ACCESS_KEY_ID="AAAAAA"
export AWS_SECRET_ACCESS_KEY="XXXXXXXX"
export AWS_DEFAULT_REGION="us-west-2"
```

## Task 1: Create an S3 bucket for state storage

In order to store state remotely, we need a place in which to store it. Amazon S3 is a supported storage medium and works well with Terraform.

For this task, you'll use the Terraform configuration API to create an S3 bucket with the required features enabled: logging, versioning, and optionally locking and at-rest encryption.

We'll also emit an output value so we can read it from another project later.

### Step 3.1.1:

Create an S3 bucket with versioning and logging enabled.

Start by declaring the `aws` provider.

```
# create-s3-bucket/main.tf
provider "aws" {
  version = "~> 1.16"
}
```

Now, define a resource for the `aws_s3_bucket`. Give it a Terraform identifier such as `tfstate_store`.

The bucket needs a name within AWS, such as `acme-moose-engineering-prod-tfstate` (be sure to include your animal name for uniqueness). You'll also want to set it to `private` so only authorized accounts can read from it.

Terraform does a great job of storing partial state if something goes wrong, but for extra protection, turn on versioning as well by defining a versioning block and setting `enabled` to `true`.

```
resource "aws_s3_bucket" "tfstate_store" {
  bucket = "acme-moose-engineering-prod-tfstate"
  acl    = "private"

  versioning {
    enabled = true
  }
}
```

A few more attributes are needed within this same resource. If we ever want to destroy this bucket, it will be inconvenient to have to manually delete all the contents first. By setting `force_destroy=true`, we give ourselves an easy way to destroy this bucket in the future.

**NOTE** If this bucket is being used for production infrastructure, it may be a good idea to make it difficult to accidentally destroy the entire bucket and all its contents. For production systems (or heavily used development or staging systems), consider omitting this.

```
resource "aws_s3_bucket" "tfstate_store" {
  # ...
  force_destroy = true
}
```

### Step 3.1.2

Next, create a second S3 bucket for logging. Give it a Terraform identifier such as `logs` and a unique name within AWS such as `acme-moose-engineering-prod-tfstate-logs` (use the `namespace` variable here).

For security, set access control to `log-delivery-write` so only the logging engine can write to the bucket.

```
resource "aws_s3_bucket" "logs" {
  bucket = "${var.namespace}-engineering-prod-tfstate-logs"
  acl    = "log-delivery-write"
}
```

Configure the existing S3 bucket to perform logging to the new logs bucket. Add these lines to the `tfstate_store` resource from above.

```
resource "aws_s3_bucket" "tfstate_store" {  
  # ...  
  logging {  
    target_bucket = "${aws_s3_bucket.logs.id}"  
    target_prefix = "log/"  
  }  
}
```

The entire file should now look like this:

```
provider "aws" {  
  version = "~> 1.16"  
}  
  
resource "aws_s3_bucket" "tfstate_store" {  
  bucket = "${var.namespace}-engineering-prod-tfstate"  
  acl    = "private"  
  force_destroy = true  
  
  versioning {  
    enabled = true  
  }  
  
  logging {  
    target_bucket = "${aws_s3_bucket.logs.id}"  
    target_prefix = "log/"  
  }  
}  
  
resource "aws_s3_bucket" "logs" {  
  bucket = "${var.namespace}-engineering-prod-tfstate-logs"  
  acl    = "log-delivery-write"  
}
```

### Step 3.1.3

Provision the buckets with `terraform init` & `terraform apply -auto-approve`

```
$ terraform init  
$ terraform apply -auto-approve
```

You should see a success message from the command. If you have access to the AWS Console, you'll be able to see the new S3 buckets in the [AWS S3 Console](#).

## Task 2: Bootstrap the configuration to store its own state on S3

Now that we have created a bucket, any new Terraform projects can be configured to store their state in it. But for the purposes of this tutorial, let's store the bucket creation project's own state in S3. We'll also emit an output so another project can read it.

### Step 3.2.1

Define a `terraform` stanza at the top of the file to use the S3 bucket that was just created. Use `s3` as the backend. Define the name of the bucket and the name of the file in which state will be stored (as key).

**NOTE** Due to the sequence by which Terraform boots itself, these values must be hard-coded. We can save some trouble by using environment variables for the AWS credentials as recommended in this lab's prerequisites.

```
# create-s3-bucket/main.tf
terraform {
  backend "s3" {
    bucket = "acme-moose-engineering-prod-tfstate"
    key    = "root.tfstate"
    encrypt = true
  }
}
```

### Step 3.2.2

Bootstrap the project's configuration into its own bucket by running `terraform init`, which will copy over existing state file to the S3 bucket. You'll be prompted to make the transfer.

```
$ terraform init

Initializing the backend...
Do you want to copy existing state to the new backend?
Pre-existing state was found while migrating the previous "local" backend
to the newly configured "s3" backend. No existing state was found in the
newly configured "s3" backend. Do you want to copy this state to the new
"s3" backend? Enter "yes" to copy and "no" to start with an empty state.

Enter a value: yes

Successfully configured the backend "s3"! Terraform will automatically
use this backend unless the backend configuration changes.
```

### Step 3.2.3

In order to give us a value to read in the next step, define one output. Let's reuse the `public_ip` idea from the previous lab. The exact value is hard-coded as part of this lab and doesn't have any other meaning.

```
# create-s3-bucket/main.tf
output "public_ip" {
  value = "8.8.8.8"
}
```

Run `terraform apply` and the updated state will be silently read and written to S3 as configured.

```
$ terraform apply
```

Congratulations! You're now storing state remotely.

## Task 3: Create a second Terraform project that reads from the primary project's remote state

As the final task in this lab, let's create a new project in a new directory. It will read the stored state from the previous `create-s3-bucket` project and will print the value of that project's `public_ip` output to its own output.

### Step 3.3.1

Create a new directory and `main.tf` file for the new project.

```
$ cd ..
$ mkdir use-state
$ cd use-state
$ touch main.tf
```

Since the remote S3 bucket has already been provisioned, we can point this brand new project at the S3 bucket before any other work has been done.

Define a similar `terraform` stanza at the top of the file. Define an `s3` backend and the same bucket name as before. However, we want to store this project's state in its own file. So for `key`, use a value that identifies this project, such as `use-state.tfstate`.



```
# use-state/main.tf
terraform {
  backend "s3" {
    bucket = "acme-moose-engineering-prod-tfstate"
    key    = "use-state.tfstate"
    encrypt = true
  }
}
```

Now without any other code, we can `init` and `apply` the project and the new (but empty) state will be stored in S3 alongside the `root` project's state.

```
$ terraform init
$ terraform apply
```

Even if we didn't need to read from another project's state, we would benefit from versioning, logging, and minimal collaboration by storing this project's state in cloud-based storage.

### Step 3.3.2

We want to build an enterprise scale, robust collaboration system for not only storing state, but sharing output values between projects. For example, one Terraform configuration might create a VPC and emit the VPC ID in an output. Other projects could read that ID and create their resources inside the previously created VPC.

We know that we'll be using Amazon Web Services resources, so declare the `aws` provider.

```
# use-state/main.tf
provider "aws" {
  version = "~> 1.16"
}
```

Next, configure a data source that points at the `root` state file. We'll use this to read from that project's output values. The `terraform_remote_state` data source implements all the functionality you need. It supports several cloud providers, but we'll be using `s3` as the backend.

Credentials are provided with our previously mentioned environment variables. However, the name of the `bucket` and the `key` (filename) of the other project's state file are needed (`root.tfstate`).

```
# use-state/main.tf
data "terraform_remote_state" "root" {
  backend = "s3"

  config {
    bucket = "${var.namespace}-engineering-prod-tfstate"
    key    = "root.tfstate"
  }
}
```

Run `init` again to install the necessary supporting files.

```
$ terraform init
```

### Step 3.3.3

To verify that we've successfully retrieved the `public_ip` value from the `root` project, let's emit its value to our own output.

You'll use the standard Terraform syntax to find the value based on the names of resources specified so far: `terraform_remote_state`, `root`, and the other projects output, `public_ip`.

```
# use-state/main.tf
output "public_ip" {
  value = "${data.terraform_remote_state.root.public_ip}"
}
```

Run `apply` to see the output.

```
$ terraform apply

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

Outputs:

public_ip = 8.8.8.8
```

It worked! You won't find the value `8.8.8.8` in the `use-state` project. It was read from the `root` (`create-s3-bucket`) project.

### Step 3.3.4

Now that you have written and run the code, you can *destroy* the resources that you created (in both projects).

```
$ pwd
~/use-state
$ terraform destroy -force
$ cd ../create-s3-bucket
$ terraform destroy -force

Failed to save state: failed to upload state: NoSuchBucket: The specified bucket
does not exist status code: 404, request id: AAAAAAAAAA, host id: AAAAA

Error: Failed to persist state to backend.
```

**NOTE** An error is displayed because the bucket was destroyed before the `create-s3-bucket` project's state could be saved there. This is expected since we were storing the project's own state on an S3 bucket which no longer exists.

# Lab 5: Terraform Enterprise Basics

Duration: 45 minutes

This lab demonstrates how to connect Terraform Enterprise to a source code management system (GitHub) and create a workspace that can apply the Terraform configuration when changes are committed.

- Task 1: Connect GitHub to TFE and Fork a GitHub Repo
- Task 2: Create public\_key and private\_key Variables
- Task 3: Queue a Plan (Warning: it will fail)
- Task 4: Edit Code on GitHub to Use Variables instead of file
- Task 5: Confirm and Apply the Plan

## Prerequisites

For this lab, we'll assume that you've installed [Terraform](#) and that you have [signed up](#) for a free trial Terraform Enterprise account. You'll also need a [GitHub](#) account.

## Task 1: Connect GitHub to TFE and Fork a GitHub Repo

Using a GitHub repository will allow us to use source control best practices on our infrastructure configs.

Populating variables to Terraform Enterprise will give Terraform Enterprise our AWS credentials so it can run Terraform on our behalf.

Connecting Terraform Enterprise to GitHub will give us a continuous integration style of workflow for managing infrastructure.

### Step 5.1.1: Fork the repo

Visit this GitHub repository and fork it so you have a copy in your own GitHub account:

```
https://github.com/hashicorp/demo-terraform-101
```

Optionally, clone the repository to your local machine (if you prefer to edit code locally instead of in the browser).

```
$ git clone https://github.com/$USER/demo-terraform-101.git
```

We will work with the `after-tfe` branch. If you choose to work locally, check out this branch:

```
$ git checkout -t origin/after-tfe
```

### Step 5.1.2: Connect GitHub to TFE

Now go to <https://app.terraform.io>

We are going to connect our GitHub repository to Terraform Enterprise. If you don't already have an organization in Terraform Enterprise, create one. If you were invited to an existing TFE organization, you can access that as well.

You'll see an empty page where your workspaces will be. Click the "New Workspace" button in the top right.

We can't create a workspace yet because there is no source repository to connect to. Let's setup GitHub by creating and connecting an OAuth client.

Go to [GitHub](#) and find your Settings page, accessed from the menu on your avatar.

At the bottom of the settings page is "Developer Settings." Click that.

Now you'll see a list of OAuth Apps. Click the "New OAuth App" button.

This form is straightforward with one exception. None of these fields are critical except the one that we'll have to leave blank initially..."Authorization Callback URL".

The screenshot shows the 'Register a new OAuth application' form. It contains the following fields and text:

- Application name:** Terraform (Training)  
Something users will recognize and trust
- Homepage URL:** <https://atlas.hashicorp.com>  
The full URL to your application homepage
- Application description:** Training application  
This is displayed to all users of your application
- Authorization callback URL:** (Empty field)  
Your application's callback URL. Read our [OAuth documentation](#) for more information.

At the bottom are two buttons: 'Register application' (green) and 'Cancel' (blue). A large blue arrow points to the 'Authorization callback URL' field with the text 'Leave Blank'.

Type in any name for the "Application Name." Use "<https://app.terraform.io>" as the URL.

Leave the final "Authorization Callback URL" field blank for now. Click "Register Application."

You'll need the Client ID and Client Secret from the resulting page. Leave the page open and copy the "Client ID" to your clipboard.

OAuth Apps

GitHub Apps

Personal access tokens

## Terraform (Training)



topfunky owns this application.

You can list your application in the [GitHub Marketplace](#) so that other users can discover it.

0 users

Client ID

7c9cc681ca0ccc4577a3

Client Secret

bf441eea7d2b982719e2aa768e9389b2f8cfda59

Revoke all user tokens

Reset client secret

Now go back to Terraform Enterprise.

Go to your OAuth Configuration by clicking the downward facing arrow by your organization's name and choosing "Organization Settings." You'll See "OAuth Configuration."

Paste in the copied Client ID and Client Secret from GitHub. Scroll down and click the button to "Create OAuth Client."

Now we finally have a "GitHub Callback URL" to use! Copy it from Terraform Enterprise and we'll take it back to GitHub.

## OAuth Clients



GitHub

Callback URL

<https://atlas.hashicorp.com/auth/94dc3315-5f10-42af-8e04-7d95b17dcd8a/callback>

HTTP URL

<https://github.com>

API URL

<https://api.github.com>

Created

Dec 12, 2017 17:22:54PM

Connect to GitHub

Connecting to GitHub will take your GitHub user through the OAuth flow to create an authorization token for access to all repositories for this organization. This means that your currently logged in GitHub user token will be used for all GitHub API interactions by any Terraform Enterprise user anywhere within the scope of **topfunky**.

Connect organization topfunky

Back at GitHub, scroll down and paste the URL into "Authorization Callback URL." Save the form.

#### Homepage URL

`https://atlas.hashicorp.com`

The full URL to your application homepage

#### Application description

Training application

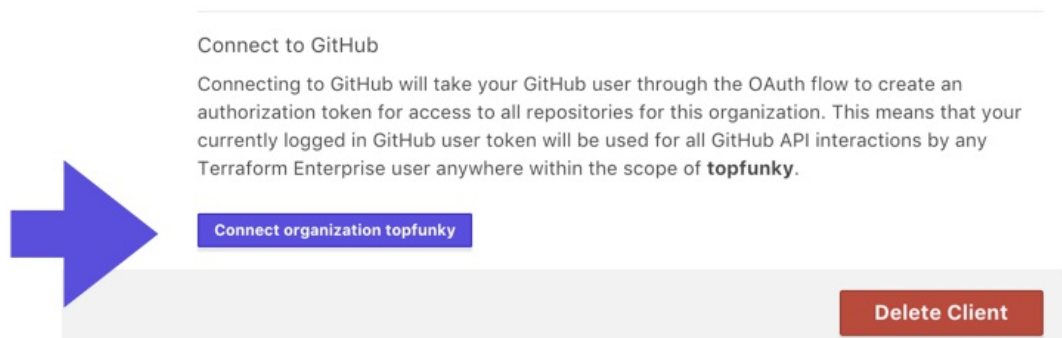
This is displayed to all users of your application

#### Authorization callback URL

`https://atlas.hashicorp.com/auth/94dc3315-5f10-42af-8e04-7d`

Your application's callback URL. Read our [OAuth documentation](#) for more information.

We're almost done. Back at Terraform Enterprise, click the purple "Connect Organization" button. You'll see an authorization screen at GitHub. Click to approve.



Back at Terraform Enterprise, you'll see that it's connected.

**NOTE:** For full capabilities, you can add your private SSH key which will be used to clone repositories and submodules. This is especially important if you use submodules and those submodules are in private repositories. That isn't the case for us so I'll leave that up to you.

### Step 5.1.3: Create a workspace in TFE

Finally, we're ready to fully create a Terraform Enterprise workspace. Go to <https://app.terraform.io> and click the "New Workspace" button at the top right.

Give it a name such as "training-demo".

GitHub is our only VCS connection. Click the "Repository" field and you'll see a list of available repositories in an auto-complete menu. Find the `demo-terraform-101` repo. If yours isn't here, refresh the page.

## Create a new Workspace

This workspace will be created under the current organization, **topfunky**.

### WORKSPACE NAME

training

The name of your workspace is unique and used in tools, routing, and UI. Dashes and alphanumeric characters are permitted. Learn more about [naming workspaces](#).

### VCS CONNECTION



Don't see the VCS connection you're looking for? [Configure a different VCS connection](#).

### REPOSITORY



☐ topfunky/training

Terraform Enterprise can deploy from any branch. We'll use the `after-tfe` branch which has been minimally modified to work with Terraform Enterprise.

### VCS BRANCH

after

The branch from which to import new versions. This defaults to the value your version control provides this repository.

Click the "More Options" link and scroll down to "VCS Branch." Type `after-tfe` to use that branch.

You'll see a screen showing that a Terraform Enterprise workspace is connected to your GitHub repository. But we still need to provide Terraform with our secret key, access key, and other variables defined in the Terraform code as variables.

### Step 5.1.4: Configure variables

Go to the "Variables" tab. On the variables page, you'll see there are two kinds of variables:

- Terraform variables: these are fed into Terraform, similar to `terraform.tfvars`
- Environment variables: these are populated in the runtime environment where Terraform executes

In the top "Terraform Variables" section, click "Edit" and add keys and values for all the variables in the project's `variables.tf` file. The only one you'll need initially is `identity` which is your unique animal name.



## Task 2: Create `public_key` and `private_key` Variables in Terraform Enterprise

### Step 5.2.1: Configure SSH keys

Terraform Enterprise runs in an environment with limited privileges. It does not have access to your instance's SSH keys that were read from local storage when this code was originally written.

Start by copying the contents of the instance's public and private SSH keys to Terraform Enterprise as Terraform variables.

**NOTE:** These keys are temporary and are used only for this class. However, you should be careful about ever sending private keys to any website, API, individual, or machine. You could avoid this scenario by building the instance in Packer or by other means.

Here is one way to display them in your terminal in a way that can be copied to your local clipboard.

Use `cat` these in the terminal, copy them to the clipboard, and create variables on Terraform Enterprise named `public_key` and `private_key`. Be sure to click "Add" after entering each key and value.

```
$ cat ~/.ssh/id_rsa.pub  
  
ssh-rsa AAAA...
```

You can mark values as sensitive, which makes them "write-only" (they will not be displayed in the UI after they are saved). Be sure to do that at least with your private key.

```
$ cat ~/.ssh/id_rsa  
  
-----BEGIN RSA PRIVATE KEY-----  
MII...
```

You MUST hit "Save" (or "Save and Plan"). Variables will not be saved otherwise.

### Step 5.2.2: Enter AWS Credentials

There is also a section for environment variables. We'll use these to store AWS credentials.

Click "Edit" and add variables for your AWS credentials.

```
AWS_ACCESS_KEY_ID="AAAA"  
AWS_SECRET_ACCESS_KEY="AAAA"  
AWS_DEFAULT_REGION="us-west-2"
```

Click the "Save" button.

## Task 3: Queue a Plan

For this task, you'll queue a terraform plan.

### Step 5.3.1: Queue a plan and read the output

Click the "Queue Plan" button at the top right.

Go to the "Runs" tab, or "Latest Run". Find the most recent one (there will probably be only one).

Scroll down to where it shows the plan. Click the button to "View Plan." You'll see the same kind of output that you are used to seeing on the command line.

We'll make another change from GitHub before running this plan, so click "Discard Plan."

## Task 4: Edit Code on GitHub to Upgrade the AWS Provider Version

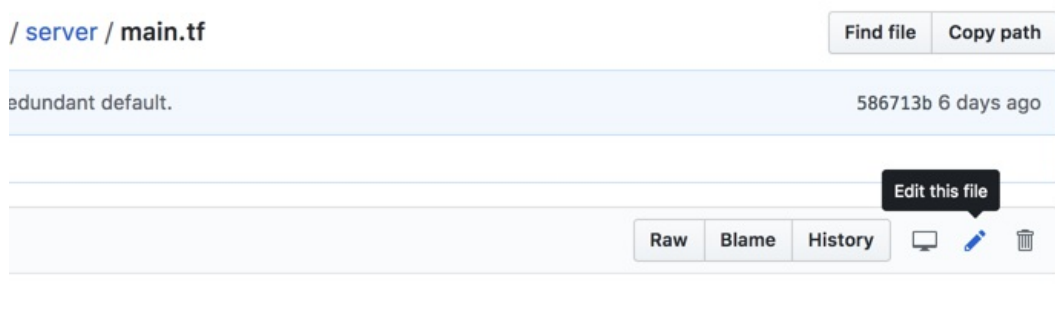
Edit code on GitHub to upgrade the AWS provider version to `>= 1.20.0`.

You'll make a pull request with these changes and observe the status of the pull request on GitHub.

### Step 5.4.1

On GitHub, find the "Branch" pulldown and switch to the `after-tfe` branch.

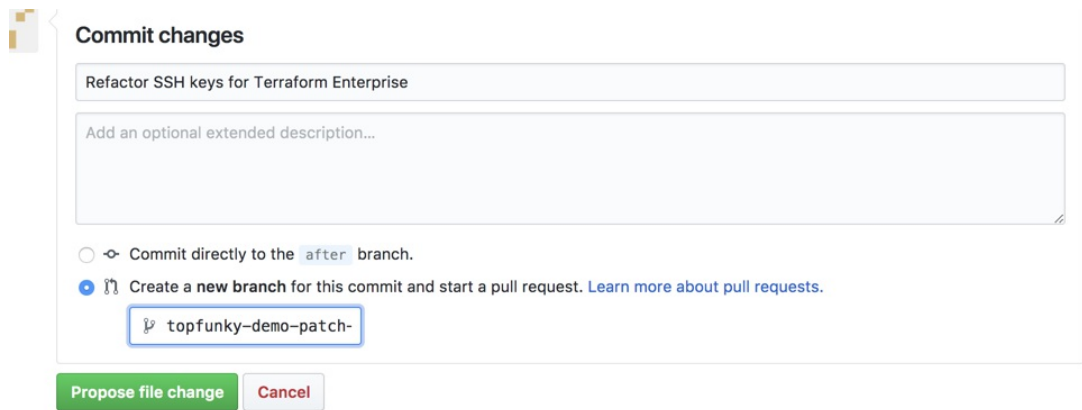
Navigate to `main.tf`. Find the pencil icon. Click to edit this file directly in the browser.



Edit the code to match the lines below.

```
provider "aws" {  
  # MODIFY this line to look for 1.20.0 or greater  
  version = ">= 1.20.0"  
}
```

Scroll to the bottom and select the option to "Create a new branch and start a pull request."



**Commit changes**

Refactor SSH keys for Terraform Enterprise

Add an optional extended description...

☐ Commit directly to the `after` branch.

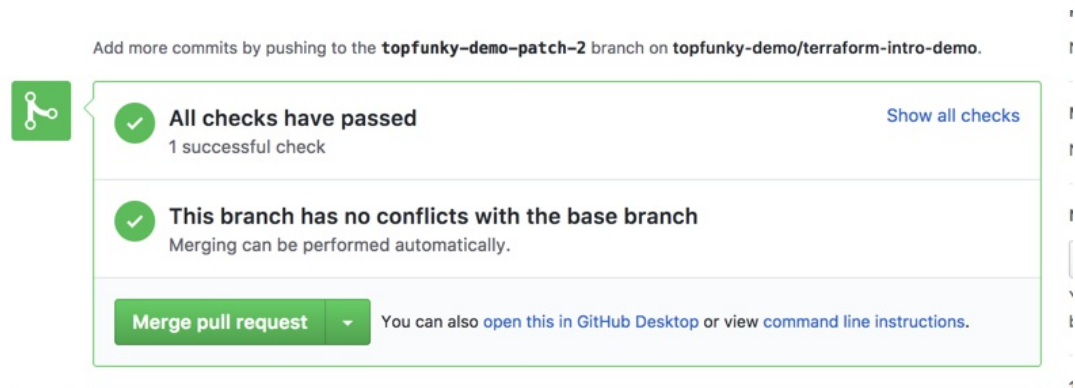
☒ Create a new branch for this commit and start a pull request. [Learn more about pull requests.](#)

`topfunky-demo-patch-`


**Propose file change** **Cancel**


You'll be taken to a screen to create a pull request. Click the green "Propose file change" button. The page will be pre-populated with your commit message. Click "Create pull request."

After a few seconds, you'll see that Terraform Enterprise checked the plan and that it passed.



Add more commits by pushing to the `topfunky-demo-patch-2` branch on `topfunky-demo/terraform-intro-demo`.

 **All checks have passed** [Show all checks](#)  
1 successful check

 **This branch has no conflicts with the base branch**  
Merging can be performed automatically.

**Merge pull request** You can also [open this in GitHub Desktop](#) or view [command line instructions](#).

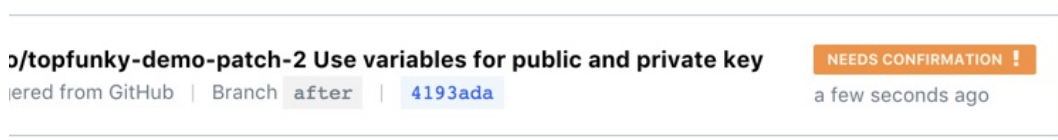
To see what was checked, click "Show all checks" and click "Details" next to the line that says "Terraform plan has changes."

Merge the pull request to the `after-tfe` branch with the big green "Merge pull request" button. Click the "Confirm merge" button.

## Task 5: Confirm and Apply the Plan

### Step 5.5.1: Confirm and `apply`


Back at Terraform Enterprise, find the "Current Run" tab. Click it and you'll see the merge commit has triggered a plan and it needs confirmation.



**o/topfunky-demo-patch-2 Use variables for public and private key** **NEEDS CONFIRMATION !**

Inherited from GitHub | Branch `after` | `4193ada` a few seconds ago

Scroll to the bottom of the run and confirm the plan. At the bottom of the page you'll see a place to comment (optional) and click "Confirm & Apply."




**Executed and saved successfully**  
a minute ago

**State versions output with this plan**  
No state versions output

[Confirm & Apply](#) [Discard Plan](#)

This will queue a terraform apply.

Examine the output of `apply` and find the IP address of the new instance. The output looks like what you've previously seen in the terminal. Copy the `public_ip` address and paste it into your browser. You'll see the running web application.



**Executed successfully with changes**  
a few seconds ago

[View raw log](#)

```
module.server.aws_instance.web (remote-exec): Private key: true
module.server.aws_instance.web (remote-exec): SSH Agent: false
module.server.aws_instance.web (remote-exec): Connected!
module.server.aws_instance.web: Creation complete after 27s (ID: i-0936d03b68874f0b6)

Apply complete! Resources: 2 added, 0 changed, 0 destroyed.

Outputs:

public_dns = [
  ec2-34-212-49-75.us-west-2.compute.amazonaws.com
]
public_ip = [
  34.212.49.75
]
```

## Task 6: Destroy

To clean up, destroy the infrastructure you've just created.


### Step 5.6.1: Configure `CONFIRM_DESTROY` variable





Go to the "Settings" tab in Terraform Enterprise and scroll to the bottom. Note the instructions under "Workspace Delete." We want to destroy the infrastructure but not necessarily the workspace.

You'll need to create an environment variable (not a Terraform variable) named `CONFIRM_DESTROY` and set it to 1.

Go to the "Variables" tab and do that.

#### Environment Variables

These variables are set using `export` in the environment running the plan and apply Editing 

|  |   |   |   |
|--|---|---|---|
| <input type="text" value="AWS_ACCESS_KEY_ID"/>     | <input type="text" value="sensitive - write only"/> | <input checked="" type="checkbox"/> Sensitive |  |
| <input type="text" value="AWS_DEFAULT_REGION"/>    | <input type="text" value="us-west-2"/>              | <input type="checkbox"/> Sensitive            |  |
| <input type="text" value="AWS_SECRET_ACCESS_KEY"/> | <input type="text" value="sensitive - write only"/> | <input checked="" type="checkbox"/> Sensitive |  |
| <input type="text" value="CONFIRM_DESTROY"/>       | <input type="text" value="1"/>                      | <input type="checkbox"/> Sensitive            |  |
| <input type="text" value="key"/>                   | <input type="text" value="value"/>                  | <input type="checkbox"/> Sensitive            | <input type="button" value="Add"/>  |

Click "Add" and "Save".

### Step 5.6.2: Queue destroy plan

It's sometimes necessary to queue a normal plan and then queue the destroy plan.

At the top of the page, click the "Queue Plan" button. The plan will run and detect that no changes need to be provisioned.

Now go back to the "Settings" tab. Scroll to the bottom and click "Queue Destroy Plan." Note the messages under "Plan" that indicate that it will destroy several resources.

Click "Confirm and Apply." After a few seconds, your infrastructure will be destroyed as requested.

# Lab 7: Template Provider

Duration: 20 minutes

This lab demonstrates how to define and render a text template with the `template_file` provider.

- Task 1: Create a Terraform configuration that contains a template to be rendered
- Task 2: Use `template_file` to render variables into the template
- Task 3: Create S3 bucket and attach policy (advanced/optional)

## Prerequisites

For this lab, we'll assume that you've installed [Terraform](#). This example is executed locally and doesn't require any other credentials or other authentication.

## Task 1: Create a Terraform configuration that contains a template to be rendered

You'll create an S3 bucket and then attach the necessary permissions to your user to list, get, and delete objects as well as create and destroy that bucket.

### Step 1.1: Create a Terraform configuration

Create a directory for the Terraform configuration. Create a sub-directory for templates. Create a template file.

```
$ mkdir -p demo-s3-policy/templates && cd $_  
$ touch iampolicy.tpl.json
```

The name of the template file is completely up to you. We like to include `tpl` to identify it as a template, but also use a suffix that corresponds to the file type of the contents (such as `json`).

### Step 1.2: Write contents of template with some placeholders for variables

In the file we just created, let's create a standard JSON AWS policy file that will allow objects in an S3 bucket to be modified. Our action statement defines the allowed operations and our resource statement defines where those operations can occur. We can define the resource through Terraform interpolation and define those variables outside the template and avoid hardcoding variables into our policies for added usability.

Terraform interpolation syntax is used (such as "\${bucket\_name}"). No prefix is needed (var. or data. or any other).

```
{
  "Id": "Policy1527877254663",
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Stmt1527877245190",
      "Action": [
        "s3:CreateBucket",
        "s3:DeleteBucket",
        "s3:DeleteObject",
        "s3:GetObject",
        "s3:ListObjects"
      ],
      "Effect": "Allow",
      "Resource": "arn:aws:s3:::${bucket_name}"
    }
  ]
}
```

## Task 2: Use `template_file` to render variable values into the template

We'll write HCL code to render the template. Let's create a file named `main.tf` in the project folder and reference the template we just created.

### Step 2.1: Define the `template_file` resource

The resource uses the `template_file` provider (created and published by HashiCorp). The second argument is a name you choose (such as `iam_policy`).

The only required argument is `template` which must contain the contents of the template (or use `file` to read from local storage).

```
data "template_file" "iam_policy" {
  template = "${file("${path.module}/templates/iam_policy.tpl.json")}"
}
```

We also use `path.module` here for robustness in any module or submodule.

### Step 2.2: Pass variables into the template

Create a variable with the value you want to pass into the template. This value could be provided externally, fetched from a data source, or even hard-coded.

Templates do not have access to all variables in a configuration. They must be explicitly passed with the `vars` block.

```
variable "bucket_name" {
  default = "anaconda-${uuid}"
}

data "template_file" "iam_policy" {
  template = "${file("${path.module}/templates/iam_policy.tpl.json")}"

  vars {
    bucket_name = "${var.bucket_name}"
  }
}
```

### Step 2.3: Render the template into an output

Rendered template content may be used anywhere: as arguments to a module or resource, or executed remotely as a script. In our case, we'll emit it as an output for easy viewing.

Use the `rendered` attribute of the template to access the merged template with the provided variable values.

```
output "iam_policy" {
  value = "${data.template_file.iam_policy.rendered}"
}
```

### Step 2.4: Run the code

Run the code with `terraform apply` and you will see the rendered template.



```

$ terraform init
$ terraform apply

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

Outputs:

iam_policy = {
  "Id": "Policy1527877254663",
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Stmt1527877245190",
      "Action": [
        "s3:CreateBucket",
        "s3:DeleteBucket",
        "s3:DeleteObject",
        "s3:GetObject",
        "s3:ListObjects"
      ],
      "Effect": "Allow",
      "Resource": "arn:aws:s3:::anaconda-327c0c36-2d6e-1df4-0b39-770780b99b9e"
    }
  ]
}

```

## Task 3: Create S3 Bucket and Attach Policy

If you have extra time and want to try a more complete, real-world example, you could create an S3 bucket, use the generated id in a policy template, and attach the rendered template as a policy.

### Step 3.1: Create an S3 bucket

Declare the `aws` provider and create an S3 bucket with your AWS owner ID.

```

provider "aws" {
}

variable "owner_id" {
  default = "anaconda"
}

resource "aws_s3_bucket" "bucket1" {
  bucket = "${var.owner_id}-${uuid()}"
  acl    = "private"
}

```

### Step 3.2: Render the template into a resource policy

The policy must be created using a provider. Create the `aws_iam_policy` and the `aws_iam_policy_attachment`.

```
resource "aws_iam_policy" "bucket1"{
  name = "${aws_s3_bucket.bucket1.id}-policy"
  policy = "${data.template_file.iam_policy.rendered}"
}

resource "aws_iam_user_policy_attachment" "attach-policy" {
  user      = "${var.owner_id}"
  policy_arn = "${aws_iam_policy.bucket1.arn}"
}
```

Use the dynamically generated `aws_s3_bucket.bucket1.id` as the `vars` value to render to the template.

```
data "template_file" "iam_policy" {
  template = "${file("${path.module}/templates/iam_policy.tpl.json")}"

  vars {
    bucket_name = "${aws_s3_bucket.bucket1.id}"
  }
}
```

### Step 3.3: Run the code

You should see four operations upon applying this configuration: the policy template data being read, the bucket creation, the policy creation, and the action of attaching the policy to your user.

```
$ terraform init
$ terraform apply
```

An execution plan has been generated and is shown below.  
Resource actions are indicated with the following symbols:  
+ create  
<= read (data resources)

Terraform will perform the following actions:

```
<= data.template_file.iam_policy
...

+ aws_iam_policy.bucket1
...

+ aws_iam_user_policy_attachment.attach-policy
```

```

...

+ aws_s3_bucket.bucket1
...

Plan: 3 to add, 0 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

aws_s3_bucket.bucket1: Creating...
...
aws_s3_bucket.bucket1: Creation complete after 3s (ID: rachel-327c0c36-2d6e-1df4-0b39-770780b99b9e)
data.template_file.iam_policy: Refreshing state...
aws_iam_policy.bucket1: Creating...
...
aws_iam_policy.bucket1: Creation complete after 0s (ID:
arn:aws:iam::130490850807:policy/rachel...c36-2d6e-1df4-0b39-770780b99b9e-policy)
aws_iam_user_policy_attachment.attach-policy: Creating...
...
aws_iam_user_policy_attachment.attach-policy: Creation complete after 1s (ID:
rachel-20180601200550527900000001)

Apply complete! Resources: 3 added, 0 changed, 0 destroyed.

Outputs:

iam_policy = {
  "Id": "Policy1527877254663",
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Stmt1527877245190",
      "Action": [
        "s3:CreateBucket",
        "s3>DeleteBucket",
        "s3>DeleteObject",
        "s3:GetObject",
        "s3:ListObjects"
      ],
      "Effect": "Allow",
      "Resource": "arn:aws:s3:::anaconda-327c0c36-2d6e-1df4-0b39-770780b99b9e"
    }
  ]
}

```

# Lab 8: Packer

Duration: 15 minutes

This lab demonstrates how to use Packer to build machine images.

- Task 1: Build an image with Packer
- Task 2: Simulate an error to prove that incorrect images will fail
- Task 3: Edit `demo-terraform-101` to use the Packer-generated AMI

## Prerequisites

For this lab, we'll assume that you've installed [Packer](#) and that you have defined your AWS credentials in your environment. See [Configuring the AWS CLI](#) in the AWS docs for details.

```
export AWS_ACCESS_KEY_ID="AAAAAA"  
export AWS_SECRET_ACCESS_KEY="XXXXXXX"  
export AWS_DEFAULT_REGION="us-west-2"
```

## Task 1: Build an image with Packer

You'll create an AWS AMI that enhances an existing Ubuntu image with a Go web application that displays a list of live visitors to the page. It will start on boot and be served on port 80.

### Step 8.1.1: Write a Packer JSON file

Copy the following into a json file. The exact name is irrelevant. Try something like `web-visitors.json`

```
{
  "variables": {
    "aws_source_ami": "ami-bd8f33c5"
  },
  "builders": [
    {
      "type": "amazon-ebs",
      "region": "us-west-2",
      "source_ami": "{{user `aws_source_ami`}}",
      "instance_type": "t1.micro",
      "ssh_username": "ubuntu",
      "ssh_pty": "true",
      "ami_name": "web-visitors-{{timestamp}}",
      "tags": {
        "Created-by": "Packer",
        "OS_Version": "Ubuntu",
        "Release": "Latest"
      }
    }
  ],
  "provisioners": [
    {
      "type": "shell",
      "inline": [
        "mkdir ~/src",
        "cd ~/src",
        "git clone https://github.com/hashicorp/demo-terraform-101.git",
        "cp -R ~/src/demo-terraform-beginner/assets /tmp",
        "sudo sh /tmp/assets/setup-web.sh"
      ]
    }
  ]
}
```

### Step 8.1.2: Validate the file

Verify that the file is correct by running the `validate` command on it.

```
$ packer validate web-visitors.json
```

You should see that it was successful.

```
Template validated successfully.
```

### Step 8.1.3: Build the image with Packer

Build the image with the `build` command. This may take a few minutes (possibly even 10 minutes).

```
$ packer build web-visitors.json
```

When it builds successfully, you will see the AMI ID of the new image.

As an advanced exercise, you could provision the Terraform configuration from the `lifecycles` chapter, but with the AMI you just created.

## Task 2: Simulate an error to prove that incorrect images will fail

In this step, you'll intentionally cause an error in the build process so you can see how failures affect the build process.

### Step 8.2.1: Add `false` to simulate an error

Make a copy of the existing Packer configuration file.

```
$ cp web-visitors.json web-visitors-fail.json
```

At the end of the `provisioners` section in the new document, replace all shell commands with the single value `false`. This will simulate an error and will cause the build to fail.

```
{
  "provisioners": [
    {
      "type": "shell",
      "inline": [
        "false"
      ]
    }
  ]
}
```

### Step 8.2.2: Build with error

Run `build` as before. You should see an error.

```
$ terraform build web-visitors-fail.json
```

```
Build 'amazon-eks' errored: Script exited with non-zero exit status: 1
```

This demonstrates that Packer will enforce an extra level of validation and will not proceed if the image did not build cleanly.

## Task 3: Edit demo-terraform-101 to use the Packer-generated AMI

The existing demo-terraform-101 that you forked a few chapters ago can be rewritten to use the Packer-generated AMI instead of the provisioner code it was originally written with.

### Step 8.3.1: Edit the ami variable in TFE

When a run is triggered from your VCS, Terraform Enterprise will use a snapshot of the variables as defined at the time that the run is triggered.

So we must first edit the ami variable before we change the code at GitHub.

Go to the "Variables" tab for your workspace. Click the "Edit" toggle. Find the ami variable and enter the ID of the AMI you created with Packer.

Click the "Save" button.

### Step 8.3.2: Update the code to use the AMI

The existing code already accepts an ami variable and feeds that through to the code in server/main.tf. Since the Packer-generated AMI is already configured with the Go web application, we can delete all the provisioner code in Terraform that accomplishes the same task.

Go to GitHub and find your fork of demo-terraform-101. Find the server/main.tf file. Edit the file to remove the following code.

```
# Delete the following 15 lines from `aws_instance`
connection {
  user      = "ubuntu"
  private_key = "${var.private_key}"
}

provisioner "file" {
  source      = "assets"
  destination = "/tmp/"
}

provisioner "remote-exec" {
  inline = [
    "sudo sh /tmp/assets/setup-web.sh",
  ]
}
```

Create a pull request and merge the pull request as you did a few chapters ago.

This will trigger a plan in Terraform Enterprise.

### Step 8.3.3: Run apply

Go to Terraform Enterprise and find the most recent run. Scroll to the bottom and approve the plan.

This will recreate the instance, but with the new AMI.

Visit the emitted IP address in the browser. The web application should work as before, but is now being provisioned directly from code baked into the AMI.

**NOTE:** Provisioning directly from an AMI usually completes more quickly than similar code provisioned with `provisioner`. Using an AMI eliminates the need to establish an SSH connection to the instance. Subsequent `provisioner` shell commands are no longer needed if the applications have been preconfigured inside the AMI.



# Lab 9: Multi-provider

Duration: 15 minutes

This lab demonstrates how several providers can be used together.

- Task 1: Provision an image with open access
- Task 2: Use the GitHub provider to constrain access to only GitHub Pages IPs

## Prerequisites

For this lab, we'll assume that you've generated a [Packer](#) image from a previous lab, or that you have access to a machine image that allows SSH access. We also assume that you have defined your AWS credentials in your environment. See [Configuring the AWS CLI](#) in the AWS docs for details. Finally, you'll need a [GitHub account](#) and an [API token](#).

```
export AWS_ACCESS_KEY_ID="AAAAAA"
export AWS_SECRET_ACCESS_KEY="XXXXXXX"
export AWS_DEFAULT_REGION="us-west-2"
```

## Task 1: Provision an image with open access

For this task, you'll generate a GitHub access token, clone demo code for this lab, and provision infrastructure without any constraints. This will show that the security group has no restrictions to start with (you'll lock it down in the next step).

### Step 9.1.1: Create a GitHub access token

Go to your [GitHub Settings Page](#) and find "Developer Settings." Click the "Personal access tokens" menu item. Click the "Generate new token" button.

Personal access tokens

Generate new token

Revoke all

Tokens you have generated that can be used to access the [GitHub API](#).

For the "Token Description", enter a name such as "Terraform 201 Token". You don't need to select any scopes. Scroll to the bottom and click the green "Generate Token" button.

Copy the resulting token and save it to a secure location.

### Step 9.1.2: Clone the demo code

Clone the course demo code to your local machine:

```
$ git clone https://github.com/hashicorp/demo-terraform-201.git
```

The code for this lab is in the 9-multi-provider directory.

```
$ cd 9-multi-provider
```

### Step 9.1.3: Configure the initial code

Configure two variables so the code can run. You'll need to copy your `github_token` and your `ami` ID from a previous lab to `terraform.tfvars`. You may need to create this file from scratch.

```
# terraform.tfvars
github_token="abcdef"
ami="ami-abcdef"
# optional
identity="badger"
namespace="tf201-demo"
```

### Step 9.1.4: Note the configuration of the GitHub provider

Open `main.tf` and look for provider `"github"` within the first 15 lines of the file.

Just as we have configured the `aws` provider, we can provide credentials for other APIs.

See the [GitHub Provider](#) for details on required configuration settings and available data sources and other resources that can be created.

```
# Configure the GitHub Provider
provider "github" {
  token      = "${var.github_token}"
  organization = "placeholder"
}
```

In order to simplify this demo, we use the word `placeholder` as the organization name. If you are creating resources which apply to a specific organization, you must configure it correctly here.

Also note the line where we use a data source to query `github_ip_ranges`. This gives us access to values such as all the IP address ranges of [GitHub Pages](#) instances or other parts of the GitHub infrastructure.

```
data "github_ip_ranges" "test" {}
```

Later on, you'll use values from this data source to configure an AWS EC2 security group.

### Step 9.1.5: Initialize and apply the configuration

Initialize and apply the configuration.

```
$ terraform init
$ terraform apply -auto-approve
```

You'll see that a provisioner step successfully pings both a GitHub IP address and hashicorp.com.

```
aws_instance.example (remote-exec): PING 192.30.252.153 (192.30.252.153) 56(84)
bytes of data.
aws_instance.example (remote-exec): 64 bytes from 192.30.252.153: icmp_seq=1
ttl=43 time=75.0 ms

aws_instance.example (remote-exec): PING hashicorp.com (151.101.193.183) 56(84)
bytes of data.
aws_instance.example (remote-exec): 64 bytes from 151.101.193.183: icmp_seq=1
ttl=46 time=19.6 ms
```

In the next task, you'll lock down the instance so it can only communicate to the GitHub IP address, not the hashicorp.com IP address.

But first, destroy the instance so we can start fresh in the next task.

```
$ terraform destroy -force
```

## Task 2: Use the GitHub provider to constrain access to only GitHub Pages IPs

In this task, you'll use data collected from the GitHub API to configure the EC2 security group. You'll restrict egress (outgoing) to only allow access to IP addresses that represent GitHub Pages resources.

### Step 9.2.1: Reconfigure the security group

The original code defined a `cidr_blocks` argument that allowed all IP address ranges (`0.0.0.0/0`).

Delete the `0.0.0.0/0` line and uncomment the line that restricts egress to only the values retrieved from the GitHub API (`data.github_ip_ranges.test.pages`).

```

egress {
  from_port    = 0
  to_port      = 0
  protocol     = "-1"
  # REMOVE the following line
  cidr_blocks  = ["0.0.0.0/0"]
  # ADD the following line
  cidr_blocks  = ["${data.github_ip_ranges.test.pages}"]
}

```

This will restrict outgoing traffic to only allow communication to IP addresses associated with the GitHub Pages service.

### Step 9.2.2: Provision and observe (intended) failure

Now provision the infrastructure as before. The ping to the first IP address will succeed since it is a GitHub Pages IP address. The second ping to hashicorp.com will fail since it is no longer allowed by the security group setting.

```

$ terraform apply -auto-approve

...
aws_instance.example (remote-exec): PING 192.30.252.153 (192.30.252.153) 56(84)
bytes of data.
aws_instance.example (remote-exec): 64 bytes from 192.30.252.153: icmp_seq=1
ttl=43 time=75.4 ms

aws_instance.example (remote-exec): PING hashicorp.com (151.101.1.183) 56(84)
bytes of data.
aws_instance.example: Still creating... (40s elapsed)

Error: Error applying plan:

1 error(s) occurred:

* aws_instance.example: error executing "/tmp/terraform_614419066.sh": Process
exited with status 1

```

In order to understand this further, read the lines in the `aws_instance` resource which define a `remote-exec` provisioner which sends a ping to both GitHub and HashiCorp.

### Step 9.2.3: Destroy

Clean up after yourself by destroying the infrastructure created in this lab.

```

$ terraform destroy -force

```

# Lab 10: Lifecycles

Duration: 15 minutes

This lab demonstrates how to use lifecycle directives to control the order in which Terraform creates and destroys resources.

- Task 1: Use `create_before_destroy` with a simple AWS security group and instance
- Task 2: Use `prevent_destroy` with an instance

## Prerequisites

For this lab, we'll assume that you've installed [Terraform](#) and that you have defined your AWS credentials in your environment, including a region. See [Configuring the AWS CLI](#) in the AWS docs for details.

```
export AWS_ACCESS_KEY_ID="AAAAAA"  
export AWS_SECRET_ACCESS_KEY="XXXXXXX"  
export AWS_DEFAULT_REGION="us-west-2"
```

## Task 1: Use `create_before_destroy` with a simple AWS security group and instance

You'll create a simple AWS configuration with a security group and an associated EC2 instance. Provision them with `terraform`, then make a change to the security group. Observe that `apply` fails because the security group can not be destroyed and recreated while the instance lives.

You'll solve this situation by using `create_before_destroy` to create the new security group before destroying the original one.

### Step 10.1.1: Create a security group and an instance

Create an S3 security group and an instance that uses it.

```

provider "aws" {}

resource "aws_security_group" "training" {
  name_prefix = "demo"

  ingress {
    from_port = 0
    to_port   = 0
    protocol  = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    from_port = 0
    to_port   = 0
    protocol  = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

resource "aws_instance" "web" {
  ami           = "ami-e474db9c"
  instance_type = "t2.micro"
  vpc_security_group_ids = ["${aws_security_group.training.id}"]

  tags {
    Name = "demo-simple-instance"
  }
}

```

Provision these resources.

```

$ terraform init
$ terraform apply

```

The commands should succeed without error.

### Step 10.1.2: Change the name of the security group

In order to see how some resources cannot be recreated under the default lifecycle settings, change the name of the security group from `demo` to something like `demo-modified`.

```

resource "aws_security_group" "training" {
  name_prefix = "demo-modified"
  # ...
}

```

Apply this change.

```
$ terraform apply
```

You'll note that it takes many minutes and eventually shows an error. You may choose to terminate the `apply` action with `^C` before the 15 minutes elapses.

```
$ terraform apply

aws_security_group: Still destroying... (14m00s elapsed)
aws_security_group: Still destroying... (14m10s elapsed)

Error: Error applying plan:

1 error(s) occurred:

* aws_security_group.training (destroy): 1 error(s) occurred:

* aws_security_group.training: DependencyViolation: resource sg-6d36fc1c has a
dependent object
    status code: 400, request id: 4665247f-165b-46fc-b8ea-9c01a23dd4e9
```

In the next step, we'll solve this problem with a `lifecycle` directive.

### Step 10.1.3: Use `create_before_destroy`

Add a `lifecycle` configuration to the `aws_security_group` resource. Specify that this resource should be created before the existing security group is destroyed.

```
resource "aws_security_group" "training" {
  name_prefix = "demo-modified"

  # ...

  lifecycle {
    create_before_destroy = true
  }
}
```

Now provision the new resources with the improved `lifecycle` configuration.

```
$ terraform apply
```

It should succeed within a short amount of time.

## Task 2: Use `prevent_destroy` with an instance

We'll demonstrate how `prevent_destroy` can be used to guard an instance from being destroyed.

### Step 10.2.1: Use `prevent_destroy`

Add `prevent_destroy = true` to the same `lifecycle` stanza where you added `create_before_destroy`.

```
resource "aws_security_group" "training" {
  name_prefix = "demo-modified"

  # ...

  lifecycle {
    create_before_destroy = true
    prevent_destroy = true
  }
}
```

Attempt to destroy the existing infrastructure. You should see the error that follows.

```
$ terraform destroy -force

Error: Error running plan: 1 error(s) occurred:

* aws_security_group.training: aws_security_group.training: the plan would
destroy this resource, but it currently has lifecycle.prevent_destroy set to
true. To avoid this error and continue with the plan, either disable
lifecycle.prevent_destroy or adjust the scope of the plan using the -target flag.
```

### Step 10.2.2: Destroy cleanly

Now that you have finished the steps in this lab, destroy the infrastructure you have created.

Remove the `prevent_destroy` attribute.

```
resource "aws_security_group" "training" {
  name_prefix = "demo-modified"

  # ...

  lifecycle {
    create_before_destroy = true
    # Comment out or delete this line
    # prevent_destroy = true
  }
}
```

Finally, run `destroy`.



```
$ terraform destroy -force
```

The command should succeed and you should see a message confirming Destroy complete! Resources: 2 destroyed.

# Lab 11: Sentinel

Duration: 25 minutes

This lab demonstrates how to write, test, and deploy simple Sentinel policies.

- Task 1: Define a local policy and test it
- Task 2: Define a policy with data structures
- Task 3: Run a policy on Terraform Enterprise

## Prerequisites

For this lab, we'll assume that you've installed [Terraform](#), that you have a [Terraform Enterprise](#) account, and that you've created a Terraform Enterprise workspace for the [TF101](#) demo application (after branch).

You'll also need the [Sentinel Simulator](#).

## Task 1: Define a local policy and test it

In this task, you'll define the simplest possible Sentinel policy in order to understand how rules work and how tests are defined.

### Step 1.1: Create a simple policy

Create a directory for your Sentinel project named `simple`. Create a file inside named `simple.sentinel`

The file will contain the bare minimum needed to satisfy a passing rule.

```
main = rule {  
  true  
}
```

By default, Sentinel's test runner starts in the PASS state. Prove this by running `sentinel test` against this policy (even though no tests have been defined).

```
$ sentinel test  
PASS - simple.sentinel
```

### Step 1.2: Write a passing test

Now create a corresponding test file that defines the passing condition. Since our rule is hard-coded to `true`, this will not require any additional variables, global state, or mocking.

Create a file named `pass.json` in a test directory within a subdirectory that matches the name of the policy (`simple`).

```
$ mkdir -p test/simple
$ touch test/simple/pass.json
```

Within `pass.json`, we can define a data structure that describes test rules, global variables, and mock data.

Type these contents into `pass.json`:

```
{
  "test": {
    "main": true
  }
}
```

This declares a `main` rule and an expectation that the return value from `main` should be `true`.

### Step 1.3: Run the test

From the `simple` root directory, run `sentinel test`.

```
$ sentinel test
PASS - simple.sentinel
PASS - test/simple/pass.json
```

You should see that the main policy passed as well as the JSON test file that we defined.

## Task 2: Define a policy with global values and more complex data structures

For this task, you'll define a more complex policy that uses string values, a map value, and a more complex value that is similar to what you'll see in production.

### Step 2.1: Create a new policy that enforces an exact `instance_type`

Create a directory for a slightly more complex policy named `with-data`.

```
$ mkdir with-data
$ touch with-data/with-data.sentinel
```

Define a rule that expects a global `instance_type` variable to be equal to `t2.medium`.

```
main = rule {
  instance_type_is_medium
}

instance_type_is_medium = rule {
  instance_type is "t2.medium"
}
```

Rules can be composed of other rules. When debugging or testing a policy, it is much easier to work with several small rules rather than one rule that has many conditions in it.

### Step 2.2: Implement a test that passes

Implement a test that defines global values expected by the policy.

In this case, we'll define `instance_type` as a global value. As before, we expect the `main` rule to be `true`. You could also list the `instance_type_is_medium` rule and specify that it is expected to be `true`.

```
{
  "global": {
    "instance_type": "t2.medium"
  },
  "test": {
    "main": true,
    "instance_type_is_medium": true
  }
}
```

### Step 2.3: Run tests

Run the tests. They should pass.

```
$ sentinel test
PASS - with-data.sentinel
PASS - test/with-data/pass.json
```

Try running the test suite again with the `-verbose` flag.

### Step 2.4: Use a map data structure

Let's use a data structure that approaches what we might see in a real-world use case. Define a rule that looks for a value in a map data structure, such as `ami.id`. Later, you'll learn to write a test that defines this.

```
main = rule {
  instance_type_is_medium and
  ami_is_present
}
```

Implement the `ami_is_present` rule using `ami.id` and the `is` comparison keyword.

```
ami_is_present = rule {
  ami.id is "ami-e474db9c"
}
```

### Step 2.5: Define a map structure in the passing test data

Add a nested `ami` with `id` value to your JSON data structure in `pass.json`.

```
{
  "global": {
    "instance_type": "t2.medium",
    "ami": {
      "id": "ami-e474db9c"
    }
  },
  "test": {
    "main": true
  }
}
```

### Step 2.6: Run tests

Run the test suite again. It should pass as before.

### Step 2.7: Add a more complex data structure

As the final implementation code for this policy, let's add a more complex data structure that requires iteration.

Add the `has_id` rule onto the `main` rule.

```
main = rule {
  instance_type_is_medium and
  ami_is_present and
  has_id
}
```

Implement the `has_id` rule which specifies that at least one server must have an `id` equal to `deciding-pegasus`. The list of servers is under the `tfplan.random_pet` namespace.

NOTE: The iterator uses Go language syntax. The `_` ignores a part of the iterator since we are only concerned with the `servers` data structure. It also analyzes the `applied` segment of the data structure since we want to examine the pending state of the server.

```
has_id = rule {  
  any tfplan.random_pet.server as _, servers {  
    servers.applied.id is "deciding-pegasus"  
  }  
}
```

## Step 2.8: Test the server applied ID

Within the `global` values, add the following `tfplan` data. This roughly matches what we might see in a production scenario.

NOTE: The `test` section has been omitted but should be left intact in your file.

```
{  
  "global": {  
    "instance_type": "t2.medium",  
    "ami": {  
      "id": "ami-e474db9c"  
    },  
    "tfplan": {  
      "random_pet": {  
        "server": {  
          "0": {  
            "applied": {  
              "id": "deciding-pegasus",  
              "length": "2",  
              "separator": "-"  
            },  
            "diff": {}  
          },  
          "1": {}  
        },  
        "2": {}  
      },  
      "diff": {}  
    },  
    "test": {}  
  },  
  "test": {}  
}
```

Run test and you should see success.

```

$ sentinel test -verbose
PASS - with-data.sentinel
  PASS - test/with-data/pass.json
    trace:
      TRUE - with-data.sentinel:1:1 - Rule "main"
        TRUE - with-data.sentinel:2:3 - instance_type_is_medium and
          ami_is_present
            TRUE - with-data.sentinel:2:3 - instance_type_is_medium
              TRUE - with-data.sentinel:8:3 - instance_type is "t2.medium"
            TRUE - with-data.sentinel:3:5 - ami_is_present
              TRUE - with-data.sentinel:12:3 - ami.id is "ami-e474db9c"
            TRUE - with-data.sentinel:4:5 - has_id
              TRUE - with-data.sentinel:16:3 - any tfplan.random_pet.server as _,
servers {
  servers.applied.id is "deciding-pegasus"
}
      TRUE - with-data.sentinel:11:1 - Rule "ami_is_present"
      TRUE - with-data.sentinel:15:1 - Rule "has_id"
      TRUE - with-data.sentinel:7:1 - Rule "instance_type_is_medium"

```

## Step 2.9: Add failing condition tests

Write a few tests that describe incorrect data that should cause specific rules (or main) to fail.

Create a JSON file in test/with-data named expected-failure.json. Paste the following contents:

```

{
  "global": {
    "instance_type": "t2.nano",
    "ami": {
      "id": "ami-11111111"
    }
  },
  "test": {
    "main": false,
    "instance_type_is_medium": false,
    "ami_is_present": false,
    "has_id": false
  }
}

```

This data specifies an instance\_type that should cause the instance\_type\_is\_medium rule to fail. That rule is also specified under test as expecting a false result.

It also specifies an incorrect ami.id.

Run test and it should pass. We specified that the provided data should result in a false value for the rules. Since this scenario was correctly achieved, the result is PASS.

```
$ sentinel test
PASS - with-data.sentinel
    PASS - test/with-data/expected-failure.json  PASS - test/with-data/pass.json
```

NOTE: For more advanced examples, see the [HashiCorp terraform-guides repo](#).

## Task 3: Run a policy on Terraform Enterprise

In this task, you'll add a Sentinel policy to your existing Terraform Enterprise workspace that is setup with the `demo-terraform-101` code (branch `after-tfe`).

### Step 3.1: Implement a Sentinel policy

Navigate to your Terraform Enterprise organization settings either by clicking on your organization name and clicking "Organization Settings", or by visiting a URL like:

```
https://app.terraform.io/app/$TFE_ORGANIZATION/settings/profile
```

Go to the "Sentinel Policy" menu item. Click "Create new policy."

### Sentinel Policy

Sentinel Policies are rules which are enforced on every workspace run to validate the terraform plan and corresponding resources are in compliance with company policies.

You do not currently have any Sentinel policies configured for your organization.

Create new policy

Mark the policy as `soft-mandatory`.

#### ENFORCEMENT MODE

- ✓ hard-mandatory (cannot override)
- soft-mandatory (can override)
- advisory (logging only)

Organization can override **soft-mandatory** checks.

Paste the following code and click "Save policy."



```

import "tfplan"

allowed_sizes = ["t2.medium"]

# Get all AWS instances contained in all modules being used
get_aws_instances = func() {
    instances = []
    for tfplan.module_paths as path {
        instances += values(tfplan.module(path).resources.aws_instance) else []
    }
    return instances
}

aws_instances = get_aws_instances()

instance_types = rule {
    all aws_instances as _, instances {
        all instances as index, r {
            all allowed_sizes as t {
                r.applied.instance_type contains t
            }
        }
    }
}

main = rule {
    (instance_types) else true
}

```

NOTE: Terraform Enterprise is not an IDE and will not report syntax errors on save. To check syntax, paste the code into a file on your local storage and run `sentinel fmt $FILENAME` against it. Any syntax errors will be reported before formatting the code with proper indents and newlines.

### Step 3.2: Make a change that queues a plan

Go to your `demo-terraform-101` repository on GitHub. Visit the `after-tfe` branch. Make an edit to the `server/main.tf` file, such as changing the `instance_type` to `t2.nano`.

Commit the change to the `after-tfe` branch. This will queue a run on Terraform Enterprise.

Visit the workspace for your `demo-terraform-101` project, or go to a URL such as:

```
https://app.terraform.io/app/$TFE_ORGANIZATION/demo-terraform-101/current
```

Scroll to the bottom and view the output of the plan. It should show failure.

## POLICY CHECK

Queued 16 minutes ago



### Organization policies hard failed

16 minutes ago

Click "View check" to see the full output, which should look something like this.

Sentinel Result: **false**

Sentinel evaluated to **false** because one or more Sentinel policies evaluated to **false**. This **false** was not due to an undefined value or runtime error.

1 policies evaluated.

## Policy 1: instance\_type\_is\_medium.sentinel (hard-mandatory)

Result: **false**

```
FALSE - instance_type_is_medium.sentinel:26:1 - Rule "main"
  FALSE - instance_type_is_medium.sentinel:17:2 - all aws_instances as _,
instances {
  all instances as index, r {
    all allowed_sizes as t {
      r.applied.instance_type contains t
    }
  }
}
```

```
FALSE - instance_type_is_medium.sentinel:16:1 - Rule "instance_types"
```

### Step 3.3: Restore code or remove policy

In order to return the configuration to a successful state, either edit your code on GitHub to restore the `t2.medium` instance type, or delete the Sentinel policy from your organization.

# Lab 12: Private Module Registry

Duration: 20 minutes

This lab demonstrates how to publish modules to the private module registry.

- Task 1: Fork a repository on GitHub and add it to your private module repository
- Task 2: Configure the module with the web UI

## Prerequisites

For this lab, we'll assume that you've installed [Terraform](#) and that you have a [GitHub](#) account.

## Task 1: Fork a repository on GitHub and add it to your private module repository

For this task, you'll fork an existing GitHub repository that is pre-built to work as a module. You'll import it to your private module repository.

### Step 1.1: Fork on GitHub

Go to GitHub and fork the `terraform-demo-animal` repository to your own account.

```
https://github.com/hashicorp/terraform-demo-animal/
```

### Step 1.2: Import to Terraform Enterprise

Go to Terraform Enterprise and click the "Modules" button in the top menu.

Click the "+ Add Module" button on the right. You'll see a form where you can choose your VCS provider (use GitHub) and a field to find your fork of the `terraform-demo-animal` repository.

Click "Publish Module" to add it to your Terraform Enterprise instance.

You can now view the details of the module.

## Task 2: Configure the module with the web UI

The Terraform Enterprise module configuration designer supports a producer/consumer pattern where some teams create modules and other teams use them to create infrastructure. You'll use the configuration designer to generate code that can be copy-and-pasted into a new Terraform project.

## Step 2.1: Launch the configuration designer

Start by either clicking the "Open in Configuration Designer" button under the right hand code snippet, or go back to the organization dashboard and click the "+ Design Configuration" button.



In either case, you'll see a screen with a list of modules. Click the "Add Module" button on the `animal` module that you imported.

## Step 2.2: Configure variables

Click the green "Next" button to proceed to the configuration screen. You'll see a list of variables, a description of each, and an input field where you can type a value for the variable.

### CONFIGURE VARIABLES

A screenshot of the 'CONFIGURE VARIABLES' form in Terraform Enterprise. The form has a light purple header with the text 'name REQUIRED'. Below this, there is a text input field containing 'a name'. At the bottom of the form, there is a search bar with the placeholder text 'enter value or type \${ to search variables' and a button labeled 'Deferred' with a checkbox icon.

Type any name into the name field, such as "web".

Click the large green "Next" button on the top right.

You'll be taken to a screen where you can preview the generated code, or download it as a file (it will be named `main.tf`).

In a production scenario, you would save this file to a new or existing Terraform project, add it to a repository in your source code control system, and connect the repository to Terraform Enterprise for provisioning.