







Terraform: Up and Running

 PREV  
5. Terraform Tips and Tricks: Loops, If-Statements, D



NEXT   
A. Recommended Reading

---

## Chapter 6. How to Use Terraform as a Team

If your team is used to managing all of its infrastructure by hand, switching to infrastructure as code (IAC) requires more than just introducing a new tool or technology. It also requires changing the culture and processes of the team. In particular, your team will need to shift from a mindset of making changes directly to infrastructure (e.g., by SSHing to a server and running commands) to a mindset where you make those changes indirectly (e.g., by updating Terraform code) and allowing automated processes to do all the actual work.

Making this transition can be a bit uncomfortable. Initially, team members may complain, as learning new languages, new technologies, and new processes will feel slower and more complicated than just jumping onto a server manually like they've been doing for years. But this up-front investment in learning has a massive payoff. Doing things by hand may feel simpler and faster for a few servers, but once you have tens, hundreds, or thousands of servers, proper IAC processes are the only options that work.

In this chapter, I'll dive into the key processes you need to put in place to make IAC work for your team, including:

- Version control
- Automated testing
- Coding guidelines
- Workflow

Let's go through these topics one at a time.

**EXAMPLE CODE**

As a reminder, all of the code examples in the book can be found at the following URL:  
<https://github.com/brikis98/terraform-up-and-running-code>.

### Version Control

All of your code should be in version control. No exceptions. It was the #1 item on the classic [Joel Test](http://bit.ly/2meqAb7) (<http://bit.ly/2meqAb7>) when Joel Spolsky created it 16+ years ago, and the only things that have changed since then are that (a) with tools like GitHub, it's easier than ever to use version control and (b) you can represent more and more things as code. This includes documentation (e.g., a README written in Markdown), application configuration (e.g., a config file written in YAML), specifications (e.g., test code written with RSpec), tests (e.g., automated tests written with JUnit), databases (e.g., schema migrations written in Active Record), and of course, infrastructure.

Not only should the code that defines your infrastructure be stored in version control, but you may want to have at least two separate version control repositories: one for modules, and one for live infrastructure. Let's look at these one at a time.

Find answers on the fly, or master something new. Subscribe today. [See pricing options.](#)

As discussed in [Chapter 4](#), your team should have one or more separate repositories where you define versioned, reusable modules. Think of each module as a “blueprint” that defines a specific part of your infrastructure.

The beauty of this arrangement is that you could have an infrastructure team that specializes in creating reusable, best-practices definitions of pieces of infrastructure within the *modules* repo. For example, the infrastructure team could take the `webserver-cluster` module you’ve developed in this book and turn it into a canonical module everyone at your company can use to run their microservices. This module handles all the details of deployment, scaling, load balancing, monitoring, alerting, and so on, so all of the other dev teams can grab this module and create and manage their own microservices independently, without being bottlenecked by the infrastructure team.

A Repository for Live Infrastructure

There should be a separate repository that defines the live infrastructure you’re running in each environment (stage, prod, mgmt, etc). Think of this as the “houses” you build from the “blueprints” in the *modules* repository. For example, here is how a dev team might use the microservice module to deploy a search and a profile microservice with different settings for each one:

```
module "search_service" {
  source = "../../../../../modules/services/webserver-cluster"

  ami           = "${data.aws_ami.ubuntu.id}"
  server_text   = "Hello from search"

  cluster_name   = "search-service-prod"
  db_remote_state_bucket = "${YOUR_BUCKET_NAME}"
  db_remote_state_key   = "prod/data-stores/mysql/terraform.tfstate"

  instance_type = "x1.16xlarge"
  min_size      = 4
  max_size      = 4

  enable_autoscaling = false
}

module "profile_service" {
  source = "../../../../../modules/services/webserver-cluster"

  ami           = "${data.aws_ami.ubuntu.id}"
  server_text   = "Hello from profile"

  cluster_name   = "profile-service-prod"
  db_remote_state_bucket = "${YOUR_BUCKET_NAME}"
  db_remote_state_key   = "prod/data-stores/mysql/terraform.tfstate"

  instance_type = "m4.large"
  min_size      = 12
  max_size      = 40

  enable_autoscaling = true
}
```

Since everyone is using the same “canonical” modules under the hood, and since those modules are versioned (which is possible because those modules are defined in a separate repository), the infrastructure team can ensure that even as the company and the number of microservices grows, everything remains consistent and maintainable. Or to be more accurate, it will remain maintainable as long as you follow the golden rule of Terraform.

The Golden Rule of Terraform

You should be able to reason about your infrastructure just by looking at the live repository. If you can scan the code of that repository and get an accurate understanding of what’s deployed, then you’ll find it easy to maintain your infrastructure. If you have to resort to checking a web console, or worse yet, relying on your developers to remember what they did or why they did it, you will find maintenance much more difficult.

This idea can be captured in a single sentence that I shall dub *The Golden Rule of Terraform*:

*The master branch of the live repository should be a 1:1 representation of what’s actually deployed in production.*

Let’s break this sentence down, starting at the end and working our way back:

“...what’s actually deployed”

The only way to ensure that the Terraform code in the live repository is an up-to-date representation of what’s actually deployed is to *never make out-of-band changes*. Once you start using Terraform, do not make changes via a web UI, or manual API calls, or any other mechanism. As you saw in [Chapter 5](#), out-of-band changes not only lead to complicated bugs, but they also void many of the benefits you get from using infrastructure as code in the first place.

“...a 1:1 representation...”

Every resource you have deployed should have a corresponding line of code in your live repository. This may seem obvious, but as a Terraform newbie, you may be tempted to “reuse” the same set of Terraform configurations to deploy many resources. For example, you might define a single set of configuration files to deploy a server, and then try to create 10 servers by running `terraform apply` 10 times on this same set of configuration files, configuring it to use a different state file and passing in different parameters via the `-var` options each time. If you do this, then after reading through the Terraform code, you’ll still have no idea what’s actually deployed, as the code gives no indication whether you ran `terraform apply` once or 10 times. The better way to get this kind of reuse is to create a module, write explicit code that uses that module 10 times, and run `terraform apply` once. Alternatively, you can use the same set of configurations over and over again, but you should store the unique variables and remote state configuration in files (you’ll see an example of this in “[Larger Teams May Need to Use a Development Pipeline](#)”).

“The master branch...”

You should only have to look at a single branch to understand what’s actually deployed in production. Typically, that branch will be master. That means all changes that affect the production environment should go directly into master (you can create a separate branch, but only to create a pull request with the intention of merging that branch into master) and you should only run `terraform apply` for the production environment against the master branch. I’ll discuss the process of making changes in master and in production in “[Workflow](#)”.

### Automated Tests

When you manage infrastructure manually, every time you go to make a change, there is an element of fear and uncertainty. You’re not sure exactly what that change will affect or if it’ll work the way you expect. You’re not sure if you applied the change the same way on all servers in all environments. You’re not sure if there will be another outage, and if there is, how late into the night you’ll have to work to fix it. As companies grow, there is more and more at stake, which makes the manual deployment process even scarier, and even more error prone. Many companies try to mitigate this risk by doing deployments less frequently, but the result is that each deployment is larger, and actually more prone to breakage.

If you manage your infrastructure through code, you have a better way to mitigate risk: automated tests. The idea is to write code that verifies that your infrastructure code works as expected. You should run these tests after every commit and revert any commits that fail. This way, every change that makes it into your codebase is proven to work and most issues will be found at build time rather than during a nervewracking deployment.

How do you write automated tests for Terraform configurations? Here are the steps:

- Prepare your code
- Write the test code
- Use multiple types of automated tests

### Prepare Your Code

One of the challenges with testing Terraform code is that, under the hood, Terraform configurations are just a convenient language for making API calls to a provider (e.g., making API calls to AWS). With automated tests for general-purpose programming languages, you’d often replace complicated dependencies, such as AWS, with *test doubles*, which implement the API of the dependency, but return hard-coded data that’s convenient for testing. With automated tests for Terraform, this technique isn’t as useful, as the interaction with that complicated dependency (such as AWS) is precisely what you want to test!

Therefore, most automated tests for Terraform simply run `terraform apply` and then try to verify that the deployed resources behave as expected. That means that automated tests for infrastructure are a bit slower to run and a bit more fragile than other types of automated tests. However, this is a small price to pay for the ability to validate all your infrastructure changes before those changes can cause problems in production. Let’s go through an example.

At the end of [Chapter 5](#), you were deploying a web server cluster in the production environment as follows:

```
provider "aws" {
  region = "us-east-1"
}

module "webserver_cluster" {
  source = "../../modules/services/webserver-cluster"

  ami           = "ami-40d28157"
  server_text   = "Hello, World"

  cluster_name   = "webservers-prod"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
}
```

```

db_remote_state_key = "prod/data-stores/mysql/terraform.tfstate"

instance_type = "m4.large"
min_size = 2
max_size = 10
enable_autoscaling = true
}

```

How can you write an automated test that verifies if this cluster works? Ideally, you would deploy the cluster into an isolated environment and test that when you hit the ELB URL, it returns the text you expect. However, if you wrote an automated test that ran `terraform apply` directly on this code, you'd run into a problem: the code is designed for deployment into the production environment, so your automated test would run in production, too, which could cause problems!

To avoid these problems, you need to make it possible to deploy your Terraform configurations into an isolated test environment by making the following parameters configurable:

#### region

The `aws` provider is hard-coded to the `us-east-1` region. It can be risky to run arbitrary test code (which could have its own bugs!) in the same region as you run your production code, so you'll need a way to specify a custom region that is dedicated for testing and has no chance of affecting production.

#### cluster\_name

The `cluster_name` parameter is hard-coded to `"webservers-prod"`. This `cluster_name` parameter is used as the name of all the resources created by the `webserver-cluster` module, including the names of the ELB and security groups. The problem is that these names must be unique, so if someone else has deployed the same web server cluster in the same region (e.g., another developer on your team running the same test at the same time), you'll get an error.

#### db\_remote\_state\_key

The `db_remote_state_key` is hard-coded to `"prod/data-stores/mysql/terraform.tfstate"`, which is the production database. You definitely don't want to run any automated tests against it, so you need a way to change this value at test time.

Therefore, the first step to making Terraform code testable is to make the various aspects of the environment pluggable. You already know how to do this: use input variables! First, you need to update the `webserver-cluster` module. Open up `modules/services/webserver-cluster/vars.tf` and add a new input variable for the AWS region:

```

variable "aws_region" {
  description = "The AWS region to use"
}

```

Make use of the new `aws_region` input variable in `modules/services/webserver-cluster/main.tf` to configure the `region` parameter of the `terraform_remote_state` data source:

```

data "terraform_remote_state" "db" {
  backend = "s3"

  config {
    bucket = "${var.db_remote_state_bucket}"
    key    = "${var.db_remote_state_key}"
    region = "${var.aws_region}"
  }
}

```

Next, head back to the production configurations and add four new input variables in `live/prod/services/webserver-cluster/vars.tf` (you should make analogous changes in staging, too):

```

variable "aws_region" {
  description = "The AWS region to use"
  default     = "us-east-1"
}

variable "cluster_name" {
  description = "The name to use for all the cluster resources"
  default     = "webservers-prod"
}

variable "db_remote_state_bucket" {
  description = "The S3 bucket used for the database's remote state"
  default     = "(YOUR_BUCKET_NAME)"
}

variable "db_remote_state_key" {
  description = "The path for the database's remote state in S3"
  default     = "prod/data-stores/mysql/terraform.tfstate"
}

```

Note that since the AWS region is now configurable, you'll have to update how you specify Amazon Machine Images (AMIs), as the AMI IDs are different in each region. Currently, the `ami` parameter is hard-coded to the ID of an Ubuntu AMI in `us-east-1`, but the ID of that AMI in another region will be completely different. To figure out the right AMI for each region, you can use the `aws_ami` data source, which allows you to search and filter the AWS Marketplace for a specific AMI. For example, here is how you can find the most recent Ubuntu 16.04 AMI from Canonical:

```
data "aws_ami" "ubuntu" {
  most_recent = true
  owners      = ["099720109477"] # Canonical

  filter {
    name = "virtualization-type"
    values = ["hvm"]
  }

  filter {
    name = "architecture"
    values = ["x86_64"]
  }

  filter {
    name = "image-type"
    values = ["machine"]
  }

  filter {
    name = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-xenial-16.04-amd64-server-*"]
  }
}
```

Pass the `id` output attribute of the `aws_ami` data source, as well as the new input variables, as parameters to the `webserver-cluster` module in `live/prod/services/webserver-cluster/main.tf`:

```
module "webserver_cluster" {
  source = "../../modules/services/webserver-cluster"

  ami          = "${data.aws_ami.ubuntu.id}"
  server_text  = "Hello, World"

  aws_region   = "${var.aws_region}"
  cluster_name = "${var.cluster_name}"
  db_remote_state_bucket = "${var.db_remote_state_bucket}"
  db_remote_state_key   = "${var.db_remote_state_key}"

  instance_type = "m4.large"
  min_size      = 2
  max_size      = 10
  enable_autoscaling = true
}
```

Also, don't forget to update the `provider` in the same file so it uses the `aws_region` input variable:

```
provider "aws" {
  region = "${var.aws_region}"
}
```

Note how the new input variables set their `default` parameters to the original, hard-coded values so that the behavior of this cluster is exactly the same as it was before. However, by exposing these as variables, you have the *option* to override these values when necessary, such as at test time. In fact, most of these changes have made the code more flexible and powerful. For example, you can now deploy a web server cluster in any AWS region and not just `us-east-1` and it will always use the latest Ubuntu 16.04 release in that region, with all the latest security patches, rather than being fixed to one version forever. This is a common pattern with automated tests: making the code more testable often improves the design of the code overall.

#### Write the Test Code

Now that you've prepared your Terraform code, you can finally write the actual test code. The test code will need to execute the following steps:

1. Run `terraform apply` on these configurations, passing in values for `cluster_name`, `region`, and `db_remote_state_key` that allow the test to run in complete isolation. For the `cluster_name` variable, you can generate a unique identifier each time the test runs. <sup>2</sup> For the `region` variable, you could pick a region that you don't normally use in production and dedicate it to testing. For the `db_remote_state_key`, you could have a dedicated database for testing or even deploy a mock database each time you run the test.
2. Run the `terraform output` command to retrieve the `elb_dns_name` output.
3. Use an HTTP client to test the URL `http://<elb_dns_name>` and check that the value you get back is what you expect (e.g., "Hello, World"). You may have to retry the HTTP request several times, as the EC2 Instances may take a minute or two to boot up and register in the ELB.

4. Run `terraform destroy` to clean up all the resources once the test is done.

Here's a simple Ruby script that implements all of these steps:

```
require 'net/http'

if ARGV.length != 3
  raise 'Invalid args. Usage: terraform-test.rb REGION DB_BUCKET DB_KEY'
end

vars = {
  # A unique (ish) 6-char string: http://stachoverflow.com/a/88341/48
  :cluster_name => (0...6).map { (65 + rand(26)).chr }.join,
  :aws_region => ARGV[0],
  :db_remote_state_bucket => ARGV[1],
  :db_remote_state_key => ARGV[2],
}

vars_string = vars.map{|key, value| "-var '#{key}' = \"#{value}\""}.join

def test_url(url, expected_text, retries)
  retries.times do
    begin
      output = Net::HTTP.get(URI.parse(url))
      puts "Output from #{url}: #{output}"
      return 'Success!' if output.include? expected_text
    rescue => e
      puts "Error from #{url}: #{e}"
    end

    puts 'Sleeping for 30 seconds and trying again'
    sleep 30
  end

  raise "Response didn't contain '#{expected_text}' after #{retries} re"
end

begin
  puts "Deploying code in #{Dir.pwd}"
  puts 'terraform get 2>&1'
  puts 'terraform apply #{vars_string} 2>&1'

  elb_dns_name = 'terraform output -no-color elb_dns_name'
  puts test_url("http://#{elb_dns_name.strip}/", 'Hello, World!', 10)
ensure
  puts "Undeploying code in #{Dir.pwd}"
  puts 'terraform destroy -force #{vars_string} 2>&1'
end
```

To run the script, you would pick a region for testing (e.g., `eu-west-1`), deploy some sort of mock database, taking care to remember the S3 bucket name and key used for the database's remote state (e.g., `my-terraform-state` and `test/data-stores/mysql/terraform.tfstate`), and then run:

```
> cd live/prod/services/webserver-cluster
> ruby terraform-test.rb \
  eu-west-1 \
  my-terraform-state \
  test/data-stores/mysql/terraform.tfstate
```

The Ruby script is just a starting point and needs a fair amount of work to make it robust. As an exercise for the reader, I recommend picking your favorite programming language and creating a simple *Domain Specific Language* (DSL) on top of it that gives you reusable primitives for writing automated tests. This DSL could include helper functions for common test tasks, such as running Terraform commands (e.g., `apply`, `output`, `destroy`), verifying HTTP endpoints, and connecting to servers over SSH. You may find some of the existing infrastructure testing tools handy, such as `kitchen-terraform` and `serverspec` (<http://serverspec.org/>).

Use Multiple Types of Automated Tests

There are several different types of automated tests you may write for your Terraform code, including unit tests, integration tests, and smoke tests. Most teams should use a combination of all three types of tests, as each type can help prevent different types of bugs.

Unit tests

Unit tests verify the functionality of a single, small unit of code. The definition of *unit* varies, but in a general-purpose programming language, it's typically a single function or class. The equivalent in Terraform is to test a single module. For example, if you had a module that deployed a database, then you may want to add tests that run each time someone modifies this module to verify that you can successfully run `terraform apply` on it, that the database boots successfully after running `terraform apply`, that you can communicate with the database and store data in it, and so on.

Integration tests

Integration tests verify that multiple units work together correctly. In a general-purpose programming language, you might test that several functions or classes work together correctly. The equivalent in Terraform is to test that several modules work together. For example, let's say you have code in your "live" repo that combines one module that creates a database, another module that creates a cluster of web servers, and a third module that deploys a load

balancer. You may want to add integration tests to this repo that run after every commit to verify that `terraform apply` completes without errors, that the database, web server cluster, and load balancer all boot correctly, that you can talk to the web servers via the load balancer, and that the data that comes back is coming from the database.

#### Smoke tests

Smoke tests run as part of the deployment process, rather than after each commit. You typically have a set of smoke tests that run each time you deploy to staging and production that do a sanity check that the code is working as expected. For example, when an app is booting, the app might run a quick smoke test to ensure it can talk to the database and that it is able to receive HTTP requests. If either of these checks fails, the app can abort the entire deployment before it causes any problems.

### Coding Guidelines

Whenever you're writing code as a team, regardless of what type of code you're writing, you should define guidelines for everyone to follow. One of my favorite definitions of "clean code" comes from an interview with Nick Dellamaggiore in the book *Hello, Startup* (<http://www.hello-startup.net/>):

*If I look at a single file and it's written by 10 different engineers, it should be almost indistinguishable which part was written by which person. To me, that is clean code.*

*The way you do that is through code reviews and publishing your style guide, your patterns, and your language idioms. Once you learn them, everybody is way more productive because you all know how to write code the same way. At that point, it's more about what you're writing and not how you write it.*

—Nick Dellamaggiore, Infrastructure Lead at Coursera

The coding guidelines that make sense for each team will be different, so here, I'll list a few of the key guidelines to consider and some examples of what you can do for each one:

- Documentation
- File layout
- Style guide

#### Documentation

In some sense, Terraform code is, in and of itself, a form of documentation. It describes in a simple language exactly what infrastructure you deployed and how that infrastructure is configured. However, there is no such thing as self-documenting code. While well-written code can tell you *what* it does, no programming language I'm aware of (including Terraform) can tell you *why* it does it.

This is why all software, including IAC, needs documentation beyond the code itself. There are several types of documentation you can consider:

#### Written documentation

Most Terraform modules should have a README that explains what the module does, why it exists, how to use it, and how to modify it. In fact, you may want to write the README first, before any of the actual Terraform code, as that will force you to consider *what* you're building and *why* you're building it before you dive into the code and get lost in the details of *how* to build it.<sup>3</sup> Spending 20 minutes writing a README can often save you hours of writing code that solves the wrong problem. Beyond the basics of a README, you may also want to have tutorials, API documentation, wiki pages, and design documents that go deeper into how the code works and why it was built this way.

#### Code documentation

Within the code itself, you can use comments as a form of documentation. Terraform treats any text that starts with a hash (`#`) as a comment. Don't use comments to explain what the code does; the code should do that itself. Only include comments to offer information that can't be expressed in code, such as how the code is meant to be used or why the code uses a particular design choice. Terraform also allows every input variable to declare a `description` parameter, which is a great place to describe how that variable should be used.

#### Example code

When creating Terraform modules, you may also want to create example code that shows how that module is meant to be used. This is a great way to highlight proper usage patterns as well as a way for users to try your module without having to write code. Example code can also be a great place to add automated tests.

### File Layout

Your team should define conventions for where Terraform code is stored and the file layout you use. Since the file layout for Terraform also determines the way Terraform state is stored, you should be especially mindful of how file layout impacts your ability to provide isolation guarantees, such as ensuring changes in a staging environment cannot accidentally cause problems in production.

Take a look at “[File Layout](#)” for a recommended file layout that provides isolation between different environments (e.g., stage and prod) and different components (e.g., a network topology for the entire environment and a single app within that environment). For larger teams, you may prefer the file layout described later in this chapter in “[Larger Teams May Need to Use a Development Pipeline](#)”.

Note that the file layout for modules is more flexible, since the modules represent reusable code and are not deployed directly. [Figure 6-1](#) shows an example file layout for a module that includes documentation, examples, and test code.

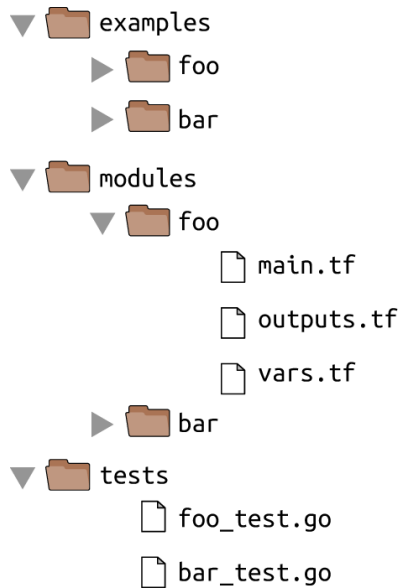


Figure 6-1. Example file layout for a Terraform module

### Style Guide

Every team should enforce a set of conventions about code style, including the use of whitespace, newlines, indentation, curly braces, variable naming, and so on. Although programmers love to debate spaces versus tabs and where the curly brace should go, the actual choice isn't that important. What really matters is that you are consistent throughout your codebase. Formatting tools are available for most text editors and IDEs, and as commit hooks for version control systems to help you enforce a common code layout.

Terraform even has a built-in `fmt` command that can reformat code to a consistent style automatically:

```
> terraform fmt
```

You could run this command as part of a commit hook to ensure that all code committed to version control automatically gets a consistent style.

### Workflow

Now that you have guidelines for how to write, test, and version Terraform code, the final thing you need is a workflow for making changes to that code. The workflow I recommend for most teams consists of the following:

1. Plan
2. Staging
3. Code review
4. Production

#### Plan



The `terraform plan` command allows you to see what changes Terraform will make before you actually `apply` those changes. The rule here is simple:

*Always run `plan` before `apply`.*

You'd be amazed at the type of errors you can catch by taking 30 seconds to run this command and scanning the "diff" you get as an output. The `plan` command even allows you to store that diff output in a file:

```
> terraform plan -out=example.plan
```

You can then run the `apply` command on this saved plan file to ensure that it applies *exactly* the changes you saw originally:

```
> terraform apply example.plan
```

Note that, just like Terraform state, the saved plan files may contain secrets. For example, if you're deploying a database with Terraform, the plan file may contain the database password. Since the plan files are not encrypted, if you want to store them for any length of time, you'll have to provide your own encryption.

Once you have a plan that looks good, the next step is to try it out in staging.

### Staging

I recommend that every team maintains at least two environments:

#### Production

An environment for production workloads (i.e., user-facing apps).

#### Staging

An environment for nonproduction workloads (i.e., testing).

Generally, staging should be an exact mirror of production, except perhaps scaled down to save money (i.e., a smaller number of servers and a smaller size for each server).

The rule here is also simple:

*Always test Terraform changes in staging before production.*

Since everything is automated with Terraform anyway, it doesn't cost you much extra effort to try a change in staging before production, but it will catch a huge number of errors. Note: you have to take into consideration what happens if multiple team members are testing uncommitted changes in staging simultaneously, something I'll come back to later in this chapter.

Testing in staging is especially important because *Terraform does not roll back changes in the case of errors*. If you run `terraform apply` and something goes wrong, you have to fix it yourself. This is easier and less stressful to do if you catch the error in staging rather than production.

If the changes work well in staging, make the same changes to the production code (but do not apply them yet!), and submit a code review.

### Code Review

If you're using GitHub, you can submit your changes for a code review using a *pull request*. For other version control systems, you may need to use other code review tools such as Phabricator or ReviewBoard. The code review should include:

#### Diff output

Just about every code review tool automatically includes the code diff so everyone can review it. Often, the mere knowledge that someone else will be looking at your code is enough motivation to get you to clean up your code, add more documentation, and write more tests. Even reviewing your own code before committing will often reveal bugs, so it's worth doing a code review even if no one but you actually looks at it!

#### Plan output

Run the `plan` command against the production environment and copy/paste the diff it returns into the code review so your teammates can review that, too, and call out anything that looks suspicious.

#### Automated test output

If you have automated tests for your Terraform configurations, run them, and paste the output into the code review. If you're using continuous integration tools such as CircleCI, TravisCI, or Jenkins, you can configure them to run after every commit and show the test results in the code review automatically.

The other members on your team should review the code to make sure it follows the coding guidelines from earlier in this chapter. Is the code formatted correctly? Is there example code? Are there automated tests? Has the documentation been updated? Does the plan indicate any possible downtime?

If everything looks good, merge the code into the master branch and prepare for deployment to production.

### Production

Once the code has been merged into master, it is safe to deploy to production. As always, run the `plan` command before `apply`, and make sure the plan matches up with what you saw in staging. If everything looks good, run `apply`, and your changes will be live.

This workflow should work for most use cases, but there are three caveats to be aware of that can significantly affect the workflow:

- Some types of Terraform changes can be automated
- Some types of Terraform changes can cause conflicts
- Larger teams may need to use a deployment pipeline

Let's discuss each of these caveats next.

### Some Types of Terraform Changes Can Be Automated

The workflow I've outlined works well for infrequent changes, such as adding new infrastructure or reconfiguring existing infrastructure. However, if you're using Terraform to apply the same type of change over and over again, you may want to completely automate the process.

A common use case is to use Terraform to deploy a new version of your app. For example, let's say you have a simple Terraform configuration that deploys an EC2 Instance:

```
resource "aws_instance" "example" {  
  ami           = "${var.version}"  
  instance_type = "t2.micro"  
}
```

The ID of the AMI to deploy is configured via an input variable called `version`:

```
variable "version" {  
  description = "The version of the app to deploy"  
  default     = "ami-40d28157"  
}
```

Every time you deploy a new version of this app, you update the `default` parameter of `var.version` to the new AMI ID, and run `terraform apply`. If you deploy new versions of your app often (e.g., some teams deploy dozens of times per day), doing a code review each time for such a trivial change is too much overhead.

For these sorts of repetitive, mechanical changes, you can write a deployment script that automatically performs the following steps:

1. Check out the live Terraform repository.
2. Update the version number for the app in the corresponding Terraform configurations.
3. Commit the changes back to version control.
4. Run `terraform apply`.

To make it easy to update the version number automatically, instead of setting the `default` parameter of `var.version`, you can put the value into a `terraform.tfvars` file in the same directory as the Terraform configuration files. It uses the same HCL syntax as Terraform, so to specify values for your variables, you just provide a bunch of key-value pairs:

```
version = "ami-40d28157"
```

Whenever you run the `plan` or `apply` command, Terraform automatically looks for a `terraform.tfvars` file, and if it finds one, it uses any variables defined within it to set the variables in your configurations. The `.tfvars` file format is fairly easy to generate from an automated deployment script, although if you don't want to deal with HCL syntax, Terraform also allows you to use JSON in a `terraform.tfvars.json` file:

```
{  
  "version": "ami-40d28157"  
}
```

You can set up commit hooks to run your automated deployment script after every commit to the master branch of an application's repository. For example, the commit hook could deploy the app to staging if the commit message contains the text "release-stage" and to production if it contains the text "release-prod". That way, deployments are triggered by commits and the history of all of those deployments ends up in your commit log. If something breaks in production, the commit log may be the first thing you check, as it now contains the information for both what code changed and what code was deployed.

### Some Types of Terraform Changes Lead to Conflicts

What happens to this workflow if multiple team members are making changes at the same time? In Chapter 3, you saw that you can use a locking mechanism such as Terragrunt or Terraform Pro/Enterprise to ensure that if two team members are running `terraform apply` at the same time on the same set of Terraform configurations, their changes do not overwrite each other. Unfortunately, this only solves part of the problem. While Terragrunt and Terraform Pro/Enterprise provide locking for Terraform state, they cannot help you with locking at the level of the Terraform configurations themselves.

For example, let's say one of your team members, Anna, makes some changes to the Terraform configurations for an app called "foo" that consists of a single EC2 Instance:

```
resource "aws_instance" "foo" {
  ami           = "ami-40d28157"
  instance_type = "t2.micro"
}
```

The app is getting a lot of traffic, so Anna decides to change the `instance_type` from `t2.micro` to `t2.medium`:

```
resource "aws_instance" "foo" {
  ami           = "ami-40d28157"
  instance_type = "t2.medium"
}
```

Here's what Anna sees when she runs `terraform plan`:

```
> terraform plan

aws_instance.foo: Refreshing state... (ID: i-6a7c545b)
(...)

~ aws_instance.foo
  instance_type: "t2.micro" => "t2.medium"

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

Those changes look good, so she runs `terraform apply` to deploy to staging.

In the meantime, Bill comes along and also starts making changes to the Terraform configurations for the same app. All Bill wants to do is to add a tag to the app:

```
resource "aws_instance" "foo" {
  ami           = "ami-40d28157"
  instance_type = "t2.micro"

  tags {
    Name = "foo"
  }
}
```

Note that Anna's changes are already deployed in staging, but as they have not been merged into master yet, Bill's code still has the old `t2.micro` `instance_type`. Here's what Bill sees when he runs the `plan` command:

```
> terraform plan

aws_instance.foo: Refreshing state... (ID: i-6a7c545b)
(...)

~ aws_instance.foo
  instance_type: "t2.medium" => "t2.micro"
  tags.%:      "0" => "1"
  tags.Name:   "" => "foo"

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

Uh oh, he's about to undo Anna's `instance_type` change! If Anna is still testing in staging, she'll be very confused when the server suddenly redeploys and starts behaving differently.

The good news is that if Bill diligently runs the `plan` command and scans its output, he will realize something is wrong, and won't cause any problems for Anna. Moreover, such a problem could never occur in production, since this workflow requires all changes to be merged to master before they can be applied to the production environment.

Nevertheless, the point of the example is to highlight what happens when developers deploy changes to a shared environment before committing those

changes to version control. You might be tempted to change the workflow to require submitting a pull request and merging to master *before* deploying to staging, but that approach has its own problems:

1. You now have no way to test your changes (other than the `plan` command) before submitting a code review. That means you'll be merging untested code into master, only to uncover bugs when you try to deploy that code to staging, so then you'll have to submit the fix for another code review, which is also untested, which means you'll be merging untested code into master...
2. Even if Anna's changes had been merged into master before Bill started working, but Bill simply forgot to update his local copy of the source code, the exact same problem would've happened anyway.

Let's take a step back and consider the typical process for updating code that is not infrastructure as code. For example, if you were updating a Ruby on Rails app, you would:

1. Make changes to your local copy of the code.
2. Test those changes locally, both through manual testing (running the Rails app on localhost) and automated testing (running the app's unit tests).
3. If everything works well locally, submit a pull request.
4. Once that pull request is merged, deploy your changes to staging.
5. If everything works well in staging, deploy your changes to prod.

The crucial difference between this workflow and the one outlined for Terraform is that with "normal" (noninfrastructure) code, you have the ability to test your changes locally before you submit a pull request and deploy them into a shared environment like staging. Since most Terraform code defines how to deploy infrastructure in a cloud environment, you can't test most Terraform changes "locally." That leaves you with two options:

#### Deal with conflicts

Use staging as a shared testing environment for all your developers and deal with conflicts as they happen. The conflicts will be relatively rare (until your teams get large), will only affect staging, and, as long as your developers are aware of the possibility of conflict and are diligent about running the `plan` command, most conflicts will be caught before they cause any problems.

#### More environments

If you have a large team that is making lots of Terraform changes and running into frequent conflicts in the staging environment, then you could create multiple staging environments instead of just one. With fewer developers in each environment, the chances of a conflict are lower. In fact, the gold standard is to allow a developer to spin up their own personal testing environment on demand whenever they are making infrastructure changes, and to tear those environments down when they are done. If all of your infrastructure is defined as code, this entire process can be automated. This is as close to doing "local testing" as you can get with infrastructure code, and it reduces the chances of a conflict to nearly zero.

If you go with the more environments option, you may find that managing a large number of environments by hand can become tedious and error prone. Once you're at this stage, you may want to start using a deployment pipeline.

#### Larger Teams May Need to Use a Development Pipeline

Let's say that you've decided that every member of your team (Ann, Bill, Cindy, etc.) can spin up an isolated environment for their own testing whenever they need it. Moreover, your company has expanded internationally, and now your code is deployed across AWS data centers around the world, including `us-east-1`, `us-west-1`, `eu-west-1`, and so on. You may end up with a folder structure that looks like [Figure 6-2](#).

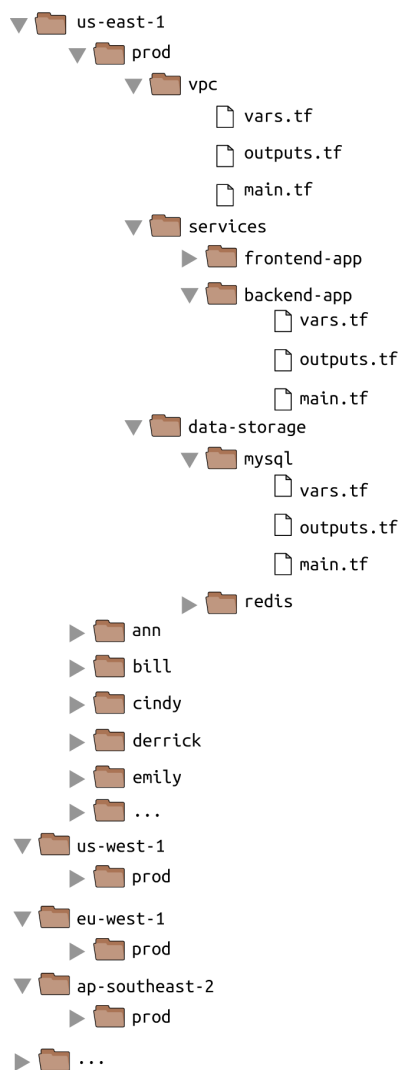


Figure 6-2. File layout with a large number of environments across many developers and many regions

Even if you are using Terraform modules, if each of those environments has a full copy of your entire stack—the VPC, the services, the data stores, and so on—you will still end up with a lot of copy/pasted Terraform code. This will make maintenance difficult and lead to errors where you make an update in one environment (e.g., `us-east-1/prod`) but forget to make the same update in other environments (e.g., `us-west-1/prod`).

One way to solve this problem is to use a *deployment pipeline*.<sup>4</sup> The general idea is to define the Terraform code in a single place and to create a pipeline that allows you to promote a single, immutable version of that definition through each of your environments.

Here's one way to implement this idea: in your *modules* repository, define all of your Terraform code for a single environment just as if you were defining it in the live repo (see Figure 6-3).

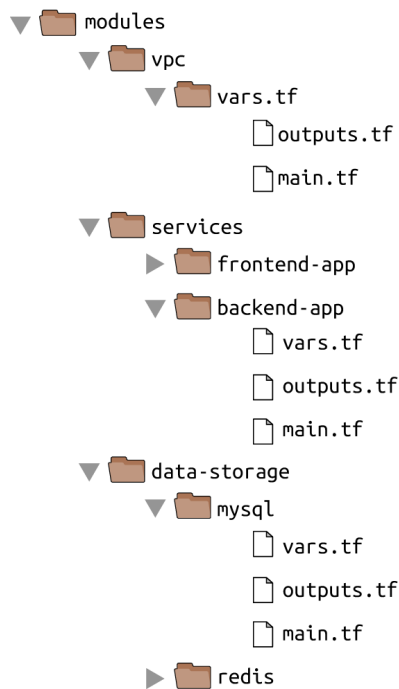


Figure 6-3. Define all the Terraform code for a single environment in the *modules* repo

Each of the components in the *modules* repo (e.g., *vpc*, *frontend-app*, *mysql*) contains standard Terraform code, ready to be deployed with a call to `terraform apply`, except for one thing: anything that needs to vary between environments is exposed as an input variable. For example, the *frontend-app* module may expose the following variables:

```

variable "aws_region" {
  description = "The AWS region to deploy into (e.g. us-east-1)"
}

variable "environment_name" {
  description = "The name of the environment (e.g. stage, prod)"
}

variable "frontend_app_instance_type" {
  description = "The instance type to run (e.g. t2.micro)"
}

variable "frontend_app_instance_count" {
  description = "The number of instances to run"
}

```

In your live repository, you can deploy each component by creating a *.tfvars* file that sets those input variables to the appropriate values for each environment. Your folder structure in the live repo would look something like [Figure 6-4](#).

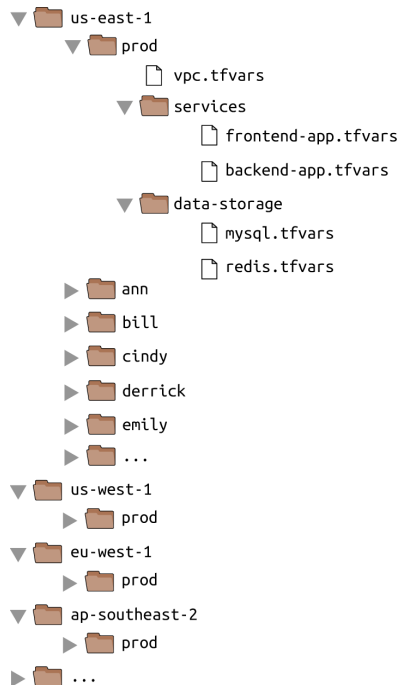


Figure 6-4. The file layout of the live repo when used with a deployment pipeline

Notice how there are no Terraform configurations (\*.tf files) in the live repo. Instead, each *.tfvars* file specifies where its Terraform configurations live using a special parameter. For example, to deploy the frontend-app module in the production environment, you might have the following settings in *us-east-1/prod/frontend-app.tfvars*:

```

source = "git::git@github.com:foo/modules.git//frontend-app?ref=v0.0.3"

aws_region = "us-east-1"
environment_name = "prod"
frontend_app_instance_type = "m4.large"
frontend_app_instance_count = 10

```

To deploy a different version of the frontend-app module in staging, you could have the following settings in *us-east-1/stage/frontend-app.tfvars*:

```

source = "git::git@github.com:foo/modules.git//frontend-app?ref=v0.0.7"

aws_region = "us-east-1"
environment_name = "stage"
frontend_app_instance_type = "t2.micro"
frontend_app_instance_count = 2

```

Both *.tfvars* files specify the location of their Terraform configurations using the *source* parameter, which can specify either a local file path or a versioned Git URL. The *.tfvars* files also define values for every variable in those Terraform configurations.

To do a deployment, you can create a script that takes the path to a *.tfvars* file as an input and does the following:<sup>5</sup>

1. Run `terraform init` to check out the *modules* repo from the URL specified in the *source* parameter of the *.tfvars* file.
2. Run `terraform apply -var-file <TF_VARS_PATH>`, where *TF\_VARS\_PATH* is the path to the *.tfvars* file.

You could run this script to promote a single version of each component through each environment. For example, if Ann just released *v0.0.7* of the frontend-app module, she could update the *source* URL to this new version in *us-east-1/ann/frontend-app.tfvars* and run the script to deploy the new version into her isolated testing environment. If the deployment succeeded and all the automated and manual tests passed, Ann could use the script to deploy the exact same version of frontend-app into the staging environment. Once again, if all tests passed, Ann could use the script one more time to promote *v0.0.7* to the production environment(s). Of course, if the automated testing is thorough enough, Ann could even configure a build server (e.g., Jenkins or Circle CI) to do all the promotions automatically so long as the automated tests were passing.

The benefit of this approach is that the code it takes to define an environment is reduced to just a handful of *tfvars* files, each of which specifies solely the variables that are different for each environment. This is about as DRY as you can get, which helps reduce the maintenance overhead and copy/paste errors of maintaining multiple environments. As a result, developers will find it easier to spin up an isolated environment for testing, and tear those environments back down when they are done. Moreover, the pipeline idea allows you to treat your infrastructure code as an immutable artifact that you promote through each environment, as shown in Figure 6-5.

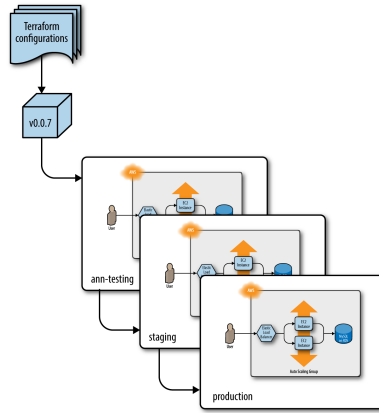


Figure 6-5. Promoting a specific version of the Terraform code from environment to environment

In Chapter 1, you saw how you can package an application as a versioned, immutable artifact (e.g., a VM image), and how that made it easier to test your application and ensure that it ran the exact same way in all environments. And now here, in the final chapter of the book, you are seeing how to package your entire infrastructure as a versioned, immutable artifact, and how that makes it easier to test your infrastructure and ensure that it runs the exact same way in all environments.

### Conclusion

If you've made it to this point in the book, you now know just about everything you need to use Terraform in the real world, including how to write Terraform code, how to manage Terraform state, how to create reusable modules with Terraform, how to do loops, if-statements, and deployments, and how to use Terraform as a team. You've worked through examples of deploying and managing servers, clusters of servers, load balancers, databases, auto scaling schedules, CloudWatch alarms, IAM users, reusable modules, zero-downtime deployment, automated tests, and more. Phew! Just don't forget to run `terraform destroy` in each module when you're all done!

The power of Terraform, and more generally, infrastructure as code, is that you can manage all the operational concerns around an application using the same coding principles as the application itself. This allows you to apply the full power of software engineering to your infrastructure, including modules, code reviews, version control, and automated testing.

If you use Terraform correctly, your team will be able to deploy faster and respond to changes more quickly. Hopefully, deployments will become routine and boring—and in the world of operations, boring is a very good thing. And if you really do your job right, rather than spending all your time managing infrastructure by hand, your team will be able to spend more and more time improving that infrastructure, allowing you to go even faster.

This is the end of the book, but just the beginning of your journey with Terraform. To learn more about Terraform, infrastructure as code, and DevOps, head over to [Appendix A](#) for a list of recommended reading. And if you've got feedback or questions, I'd love to hear from you at [jim@yrikman.com](mailto:jim@yrikman.com). Thank you for reading!

<sup>1</sup> You may want to run tests not only in an isolated region, but also an isolated Virtual Private Cloud (VPC) within that region. All the code in this book uses the Default VPC to keep the examples simple, but in real-world use cases, you should explicitly specify a VPC for staging, production, and test, and ensure that all three are completely isolated from each other.

<sup>2</sup> For an example of how to generate alphanumeric strings that are unique enough to avoid conflicts and short enough to use as AWS identifiers, see: <http://stackoverflow.com/a/9543797/483528> (<http://stackoverflow.com/a/9543797/483528>).

<sup>3</sup> Writing the README first is often called README Driven Development, as described here: <http://bit.ly/1p8QBor> (<http://bit.ly/1p8QBor>).



<sup>4</sup> Credit for this idea goes to Kief Morris: [Using Pipelines to Manage Environments with Infrastructure as Code \(http://bit.ly/2Hmsu8J\)](http://bit.ly/2Hmsu8J).

<sup>5</sup> Terragrunt has support built-in for this workflow.

[Recommended](#) / [Queue](#) / [History](#) / [Topics](#) / [Tutorials](#) / [Settings](#) / [Get the App](#) / [Sign Out](#)

 [PREV](#)  
[5. Terraform Tips and Tricks: Loops, If-Statements, Deployment...](#)

[NEXT](#)   
[A. Recommended Reading](#)