goal is to catch as many errors as early as you can, before those errors can do any damage.

Use `postcondition` blocks for enforcing basic guarantees

Use `postcondition` blocks in all of your production-grade modules to check guarantees about how your module behaves *after* changes have been deployed. The goal is to give users of your module confidence that your module will either do what it says when they run `apply` or exit with an error. It also gives maintainers of that module a clearer signal of what behaviors you want this module to enforce, so those aren't accidentally lost during a refactor.

Use automated testing tools for enforcing more advanced assumptions and guarantees

`validation`, `precondition`, and `postcondition` blocks are all useful tools, but they can only do basic checks. This is because you can only use data sources, resources, and language constructs built into Terraform to do these checks, and those are often not enough for more advanced behavior. For example, if you built a module to deploy a web service, you might want to add a check after deployment that the web service is able to respond to HTTP requests. You could try to do this in a `postcondition` block by making HTTP requests to the service using Terraform's http provider, but most deployments happen asynchronously, so you may need to retry the HTTP request multiple times, and there is no retry mechanism built into that provider. Moreover, if you deployed an internal web service, it might not be accessible over the public internet, so you'd need to connect to some internal network or VPN first, which is also tricky to do in pure Terraform code. Therefore, to do more robust checks, you'll want to use automated testing tools such as OPA and Terratest, both of which you'll see in Chapter 9.

Versioned Modules

There are two types of versioning you'll want to think through with modules:

- Versioning of the module's dependencies
- Versioning of the module itself

Let's start with versioning of the module's dependencies. Your Terraform code has three types of dependencies:

Terraform core
: The version of the `terraform` binary you depend on

Providers
: The version of each provider your code depends on, such as the `aws` provider

Modules
: The version of each module you depend on that are pulled in via `module` blocks

As a general rule, you'll want to practice *versioning pinning* with all of your dependencies. That means that you pin each of these three types of dependencies to a specific, fixed, known version. Deployments should be predictable and repeatable: if the code didn't change, then running `apply` should always produce the same result, whether you run it today or three months from now or three years from now. To make that happen, you need to avoid pulling in new versions of dependencies accidentally. Instead, version upgrades should always be an explicit, deliberate action that is visible in the code you check into version control.

Let's go through how to do version pinning for the three types of Terraform dependencies.

To pin the version of the first type of dependency, your Terraform core version, you can use the `required_version` argument in your code. At a bare minimum, you should require a specific major version of Terraform:

```
terraform {
  # Require any 1.x version of Terraform
  required_version = ">= 1.0.0, < 2.0.0"
}
```

This is critical, because each major release of Terraform is backward incompatible: e.g., the upgrade from 1.0.0 to 2.0.0 will likely include breaking changes, so you don't want to do it by accident. The preceding code will allow you to use only 1.x.x versions of Terraform with that module, so 1.0.0 and 1.2.3 will work, but if you try to use, perhaps accidentally, 0.14.3 or 2.0.0, and run `terraform apply`, you immediately get an error:

```
$ terraform apply

Error: Unsupported Terraform Core version

This configuration does not support Terraform version 0.14.3. To
proceed,
either choose another supported Terraform version or update the
root module's
version constraint. Version constraints are normally set for good
reason, so
updating the constraint may lead to other errors or unexpected
behavior.
```

For production-grade code, you may want to pin not only the major version but the minor and patch version too:

```
terraform {
  # Require Terraform at exactly version 1.2.3
  required_version = "1.2.3"
}
```

In the past, before the Terraform 1.0.0 release, this was absolutely required, as every release of Terraform potentially included backward-incompatible changes, including to the state file format: e.g., a state file written by Terraform version 0.12.1 could not be read by Terraform version 0.12.0.

Fortunately, after the 1.0.0 release, this is no longer the case: as per the officially published Terraform v1.0 Compatibility Promises, upgrades between v1.x releases should require no changes to your code or workflows.

That said, you might still not want to upgrade to a new version of Terraform *accidentally*. New versions introduce new features, and if some of your computers (developer workstations and CI servers) start using those features but others are still on the old versions, you'll run into issues. Moreover, new versions of Terraform may have bugs, and you'll want to test that out in pre-production environments before trying it in production. Therefore, while pinning the major version is the bare minimum, I also recommend pinning the minor and patch version and applying Terraform upgrades intentionally, carefully, and consistently throughout each environment.

Note that, occasionally, you may have to use different versions of Terraform within a single codebase. For example, perhaps you are testing out Terraform 1.2.3 in the stage environment, while the prod environment is still on Terraform 1.0.0. Having to deal with multiple Terraform versions, whether on your own computer or on your CI servers, can be tricky. Fortunately, the open source tool tfenv, the Terraform version manager, makes this much easier.

At its most basic level, you can use `tfenv` to install and switch between multiple versions of Terraform. For example, you can use the `tfenv install` command to install a specific version of Terraform:

```
$ tfenv install 1.2.3
Installing Terraform v1.2.3
Downloading release tarball from
https://releases.hashicorp.com/terraform/1.2.3/terraform_1.2.3_da
rwin_amd64.zip
Archive:  tfenv_download.ZUS3Qn/terraform_1.2.3_darwin_amd64.zip
  inflating:
/opt/homebrew/Cellar/tfenv/2.2.2/versions/1.2.3/terraform
Installation of terraform v1.2.3 successful.
```

TFENV ON APPLE SILICON (M1, M2)

As of June 2022, `tfenv` did not install the proper version of Terraform on Apple Silicon, such as Macs running M1 or M2 processors (see this open issue for details). The workaround is to set the `TFENV_ARCH` environment variable to `arm64`:

```
$ export TFENV_ARCH=arm64
$ tfenv install 1.2.3
```

You can list the versions you have installed using the `list` command:

```
$ tfenv list
  1.2.3
  1.1.4
  1.1.0
* 1.0.0 (set by /opt/homebrew/Cellar/tfenv/2.2.2/version)
```

And you can select the version of Terraform to use from that list using the `use` command:

```
$ tfenv use 1.2.3
Switching default version to v1.2.3
Switching completed
```

These commands are all handy for working with multiple versions of Terraform, but the real power of `tfenv` is its support for *.terraform-version* files. `tfenv` will automatically look for a *.terraform-version* file in the current folder, as well as all the parent folders, all the way up to the project root—that is, the version control root (e.g., the folder with a *.git* folder in it)—and if it finds that file, any `terraform` command you run will automatically use the version defined in that file.

For example, if you wanted to try out Terraform 1.2.3 in the stage environment, while sticking with Terraform 1.0.0 in the prod environment, you could use the following folder structure:

```
live
 └ stage
   └ vpc
   └ mysql
   └ frontend-app
   └ .terraform-version
 └ prod
   └ vpc
   └ mysql
   └ frontend-app
   └ .terraform-version
```

Inside of *live/stage/.terraform-version*, you would have the following:

```
1.2.3
```

And inside of *live/prod/.terraform-version*, you would have the following:

```
1.0.0
```

Now, any `terraform` command you run in *stage* or any subfolder will automatically use Terraform 1.2.3. You can check this by running the `terraform version` command:

```
$ cd stage/vpc
$ terraform version
Terraform v1.2.3
```

And similarly, any `terraform` command you run in *prod* will automatically use Terraform 1.0.0:

```
$ cd prod/vpc
$ terraform version
Terraform v1.0.0
```

This works automatically on any developer workstation and in your CI server so long as everyone has `tfenv` installed. If you're a Terragrunt user, tgswitch offers similar functionality to automatically pick the Terragrunt version based on a *.terragrunt-version* file.

Let’s now turn our attention to the second type of dependency in your Terraform code: providers. As you saw in Chapter 7, to pin provider versions, you can use the `required_providers` block :

```
terraform {
  required_version = ">= 1.0.0, < 2.0.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}
```

This code pins the AWS Provider code to any 4.x version (the `~> 4.0` syntax is equivalent to `>= 4.0, < 5.0`). Again, the bare minimum is to pin to a specific major version number to avoid accidentally pulling in backward-incompatible changes. With Terraform 0.14.0 and above, you don’t need to pin minor or patch versions for providers, as this happens automatically due to the *lock file*. The first time you run `terraform init`, Terraform creates a *.terraform.lock.hcl* file, which records the following information:

The exact version of each provider you used

> If you check the *.terraform.lock.hcl* file into version control (which you should!), then in the future, if you run `terraform init` again, on this computer or any other, Terraform will download the *exact* same version of each provider. That’s why you don’t need to pin the minor and patch version number in the `required_providers` block, as that’s the default behavior anyway. If you want to explicitly upgrade a provider version, you can update the version constraint in the `required_providers` block and run `terraform init -upgrade`. Terraform will download new providers that match your version constraints and update the *.terraform.lock.hcl* file; you should review those updates and commit them to version control.

The checksums for each provider

Terraform records the checksum of each provider it downloads, and on subsequent runs of `terraform init`, it will show an error if the checksum changed. This is a security measure to ensure someone can't swap out provider code with malicious code in the future. If the provider is cryptographically signed (most official HashiCorp providers are), Terraform will also validate the signature as an additional check that the code can be trusted.

LOCK FILES WITH MULTIPLE OPERATING SYSTEMS

By default, Terraform only records checksums for the platform you ran `init` on: for example, if you ran `init` on Linux, then Terraform will only record the checksums for Linux provider binaries in *.terraform.lock.hcl*. If you check that file in and, later on, you run `init` on that code on a Mac, you'll get an error, as the Mac checksums won't be in the *.terraform.lock.hcl* file. If your team works across multiple operating systems, you'll need to run the `terraform providers lock` command to record the checksums for every platform you use:

```
terraform providers lock \
  -platform=windows_amd64 \ # 64-bit Windows
  -platform=darwin_amd64 \  # 64-bit macOS
  -platform=darwin_arm64 \  # 64-bit macOS (ARM)
  -platform=linux_amd64     # 64-bit Linux
```

Finally, let's now look at the third type of dependencies: modules. As discussed in "Module Versioning", I strongly recommend pinning module versions by using `source` URLs (rather than local file paths) with the `ref` parameter set to a Git tag:

```
  source = "git@github.com:foo/modules.git//services/hello-world-
app?ref=v0.0.5"
```

If you use these sorts of URLs, Terraform will always download the exact same code for the module every time you run `terraform init`.

Now that you've seen how to version your code's dependencies, let's talk about how to version the code itself. As you saw in "Module Versioning", you can version your code by using Git tags with semantic versioning:

```
$ git tag -a "v0.0.5" -m "Create new hello-world-app module"
$ git push --follow-tags
```

For example, to deploy version `v0.0.5` of your `hello-world-app` module in the staging environment, put the following code into *live/stage/services/hello-world-app/main.tf*:

```
provider "aws" {
  region = "us-east-2"
}

module "hello_world_app" {
  # TODO: replace this with your own module URL and version!!
  source = "git@github.com:foo/modules.git//services/hello-world-
app?ref=v0.0.5"

  server_text            = "New server text"
  environment            = "stage"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "stage/data-
stores/mysql/terraform.tfstate"

  instance_type      = "t2.micro"
  min_size           = 2
  max_size           = 2
  enable_autoscaling = false
  ami                = data.aws_ami.ubuntu.id
}

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

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-
server-*"]
  }
}
```

Next, pass through the ALB DNS name as an output in *live/stage/services/hello-world-app/outputs.tf*:

```
output "alb_dns_name" {
  value       = module.hello_world_app.alb_dns_name
  description = "The domain name of the load balancer"
}
```

Now you can deploy your versioned module by running `terraform init` and `terraform apply`:

```
$ terraform apply

(...)

Apply complete! Resources: 13 added, 0 changed, 0 destroyed.

Outputs:

alb_dns_name = "hello-world-stage-477699288.us-east-
2.elb.amazonaws.com"
```

If that works well, you can then deploy the exact same version—and therefore the exact same code—to other environments, including production. If you ever encounter an issue, versioning also gives you the option to roll back by deploying an older version.

Another option for releasing modules is to publish them in the Terraform Registry. The Public Terraform Registry includes hundreds of reusable, community-maintained, open source modules for AWS, Google Cloud, Azure, and many other providers. There are a few requirements to publish a module to the Public Terraform Registry:[6]

- The module must live in a public GitHub repo.
- The repo must be named `terraform-<PROVIDER>-<NAME>`, where `PROVIDER` is the provider the module is targeting (e.g., `aws`) and `NAME` is the name of the module (e.g., `rds`).

- The module must follow a specific file structure, including defining Terraform code in the root of the repo, providing a *README.md*, and using the convention of *main.tf*, *variables.tf*, and *outputs.tf* as filenames.
- The repo must use Git tags with semantic versioning (`x.y.z`) for releases.

If your module meets those requirements, you can share it with the world by logging in to the Terraform Registry with your GitHub account and using the web UI to publish the module. Once your modules are in the Registry, your team can use a web UI to discover modules and learn how to use them.

Terraform even supports a special syntax for consuming modules from the Terraform Registry. Instead of long Git URLs with hard-to-spot `ref` parameters, you can use a special shorter registry URL in the `source` argument and specify the version via a separate `version` argument using the following syntax:

```
module "<NAME>" {
  source  = "<OWNER>/<REPO>/<PROVIDER>"
  version = "<VERSION>"

  # (...)
}
```

where `NAME` is the identifier to use for the module in your Terraform code, `OWNER` is the owner of the GitHub repo (e.g., in `github.com/foo/bar`, the owner is `foo`), `REPO` is the name of the GitHub repo (e.g., in `github.com/foo/bar`, the repo is `bar`), `PROVIDER` is the provider you're targeting (e.g., `aws`), and `VERSION` is the version of the module to use. Here's an example of how to use an open source RDS module from the Terraform Registry:

```
module "rds" {
  source  = "terraform-aws-modules/rds/aws"
  version = "4.4.0"
```