```
  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
}
```

After v0.0.2 has been thoroughly tested and proven in staging, you can then update production, too. But if there turns out to be a bug in v0.0.2, no big deal, because it has no effect on the real users of your production environment. Fix the bug, release a new version, and repeat the entire process again until you have something stable enough for production.

DEVELOPING MODULES

Versioned modules are great when you're deploying to a shared environment (e.g., staging or production), but when you're just testing on your own computer, you'll want to use local file paths. This allows you to iterate faster, because you'll be able to make a change in the module folders and rerun the `plan` or `apply` command in the live folders immediately, rather than having to commit your code, publish a new version, and rerun `init` each time.

Since the goal of this book is to help you learn and experiment with Terraform as quickly as possible, the rest of the code examples will use local file paths for modules.

Conclusion

By defining infrastructure as code in modules, you can apply a variety of software engineering best practices to your infrastructure. You can validate each change to a module through code reviews and automated tests, you can create semantically versioned releases of each module, and you can safely try out different versions of a module in different environments and roll back to previous versions if you hit a problem.

All of this can dramatically increase your ability to build infrastructure quickly and reliably because developers will be able to reuse entire pieces of proven, tested, and documented infrastructure. For example, you could create a canonical module that defines how to deploy a single microservice —including how to run a cluster, how to scale the cluster in response to load, and how to distribute traffic requests across the cluster—and each team could use this module to manage their own microservices with just a few lines of code.

To make such a module work for multiple teams, the Terraform code in that module must be flexible and configurable. For example, one team might want to use your module to deploy a single Instance of their microservice with no load balancer, whereas another might want a dozen Instances of their microservice with a load balancer to distribute traffic between those Instances. How do you do conditional statements in Terraform? Is there a way to do a for-loop? Is there a way to use Terraform to roll out changes to this microservice without downtime? These advanced aspects of Terraform syntax are the topic of Chapter 5.

---

1 For the full details on source URLs, see the Terraform website.

2 See the GitHub documentation for a nice guide on working with SSH keys.