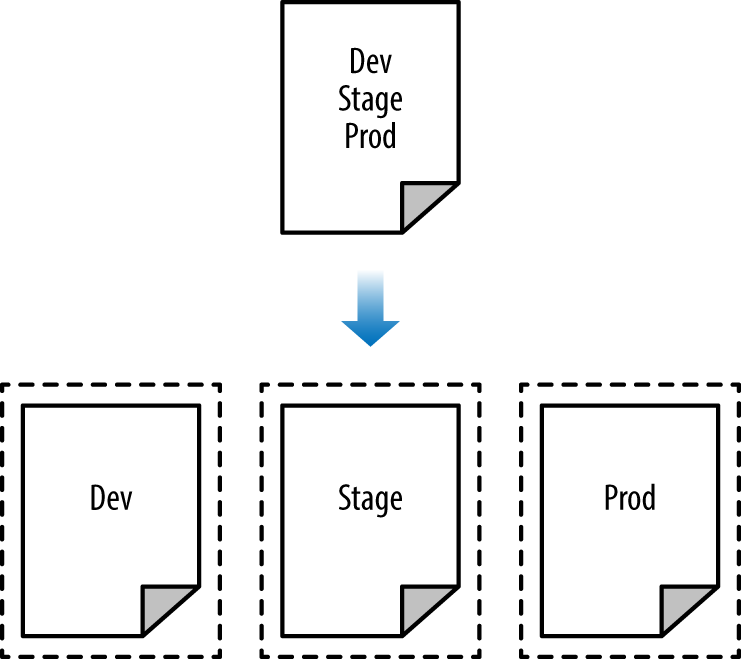
**Isolating state files**

With a remote backend and locking, collaboration is no longer a problem. However, there is still one more problem remaining: isolation. When you first start using Terraform, you might be tempted to define all of your infrastructure in a single Terraform file or a single set of Terraform files in one folder. The problem with this approach is that all of your Terraform state is now stored in a single file, too, and a mistake anywhere could break everything.

For example, while trying to deploy a new version of your app in staging, you might break the app in production. Or, worse yet, you might corrupt your entire state file, either because you didn’t use locking or due to a rare Terraform bug, and now all of your infrastructure in all environments is broken (here’s a [colorful example](https://charity.wtf/2016/03/30/terraform-vpc-and-why-you-want-a-tfstate-file-per-env/) of what happens when you don’t isolate Terraform state.)

The whole point of having separate environments is that they are isolated from one another, so if you are managing all the environments from a single set of Terraform configurations, you are breaking that isolation. Just as a ship has bulkheads that act as barriers to prevent a leak in one part of the ship from immediately flooding all the others, you should have “bulkheads” built into your Terraform design:



As the diagram above illustrates, instead of defining all your environments in a single set of Terraform configurations (top), you want to define each environment in a separate set of configurations (bottom), so a problem in one environment is completely isolated from the others. There are two ways you could isolate state files:

* **Isolation via workspaces:**Useful for quick, isolated tests on the same configuration
* **Isolation via file layout:**Useful for production use cases for which you need strong separation between environments

Let’s dive into each of these in the next two sections.

**Isolation via workspaces**

[*Terraform workspaces*](https://www.terraform.io/language/state/workspaces)allow you to store your Terraform state in multiple, separate, named workspaces. Terraform starts with a single workspace called “default,” and if you never explicitly specify a workspace, the default workspace is the one you’ll use the entire time. To create a new workspace or switch between workspaces, you use the terraform workspace commands. Let’s experiment with workspaces on some Terraform code that deploys a single EC2 Instance:

resource "aws\_instance" "example" {  
 ami = "ami-0fb653ca2d3203ac1"  
 instance\_type = "t2.micro"  
}

Configure a backend for this Instance using the S3 bucket and DynamoDB table you created earlier but with the key set to *workspaces-example/terraform.tfstate*:

terraform {  
 backend "s3" {  
 # Replace this with your bucket name!  
 bucket = "terraform-up-and-running-state"  
 key = "workspaces-example/terraform.tfstate"  
 region = "us-east-2"  
  
 # Replace this with your DynamoDB table name!  
 dynamodb\_table = "terraform-up-and-running-locks"  
 encrypt = true  
 }  
}

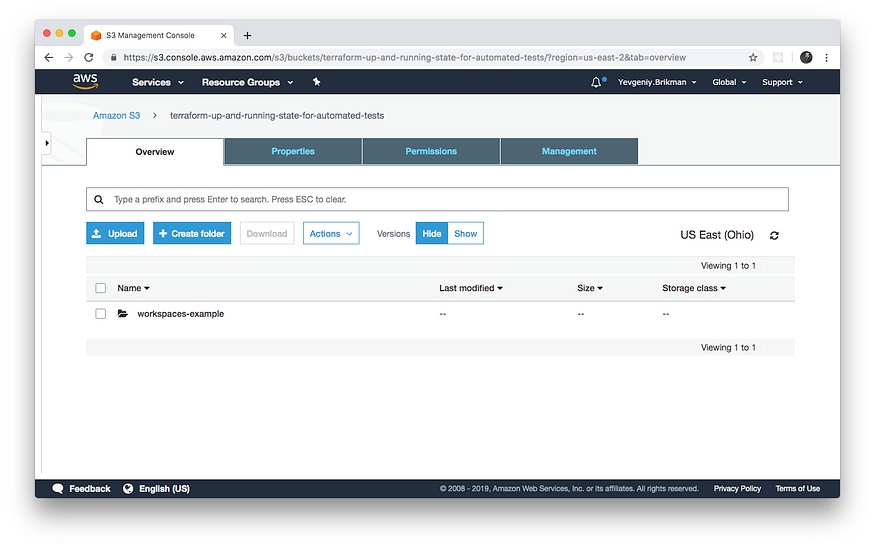
Run terraform init and terraform apply to deploy this code:

$ terraform init  
  
  
Initializing the backend...  
  
Successfully configured the backend "s3"! Terraform will automatically use this backend unless the backend configuration changes.  
  
Initializing provider plugins...  
  
(...)  
  
Terraform has been successfully initialized!  
  
  
  
$ terraform apply  
  
(...)  
  
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

The state for this deployment is stored in the default workspace. You can confirm this by running the terraform workspace show command, which will identify which workspace you’re currently in:

$ terraform workspace show  
default

The default workspace stores your state in exactly the location you specify via the key configuration. If you take a look in your S3 bucket, you’ll find a *terraform.tfstate*file in the *workspaces-example*folder:



Let’s create a new workspace called “example1” using the terraform workspace new command:

$ terraform workspace new example1  
Created and switched to workspace "example1"!  
  
You're now on a new, empty workspace. Workspaces isolate their state, so if you run "terraform plan" Terraform will not see any existing state for this configuration.

Now, note what happens if you try to run terraform plan:

$ terraform plan  
  
Terraform will perform the following actions:  
  
 # aws\_instance.example will be created  
 + resource "aws\_instance" "example" {  
 + ami = "ami-0fb653ca2d3203ac1"  
 + instance\_type = "t2.micro"  
 (...)  
 }  
  
Plan: 1 to add, 0 to change, 0 to destroy.

Terraform wants to create a totally new EC2 Instance from scratch! That’s because the state files in each workspace are isolated from one another, and because you’re now in the example1 workspace, Terraform isn’t using the state file from the default workspace and therefore doesn’t see the EC2 Instance was already created there.

Try running terraform apply to deploy this second EC2 Instance in the new workspace:

$ terraform apply  
  
(...)  
  
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

Repeat the exercise one more time and create another workspace called “example2”:

$ terraform workspace new example2  
Created and switched to workspace "example2"!  
  
You're now on a new, empty workspace. Workspaces isolate their state, so if you run "terraform plan" Terraform will not see any existing state for this configuration.

Run terraform apply again to deploy a third EC2 Instance:

$ terraform apply  
  
(...)  
  
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

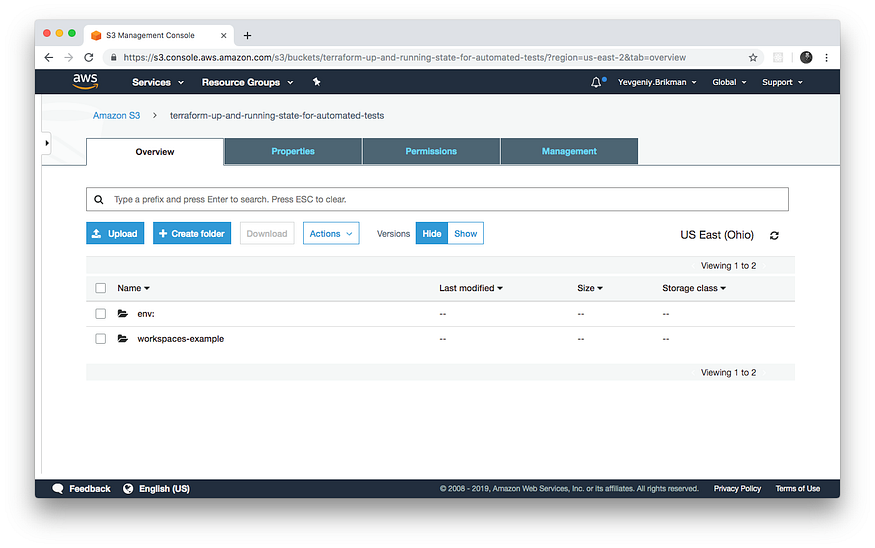
You now have three workspaces available, which you can see by using the terraform workspace list command:

$ terraform workspace list  
 default  
 example1  
\* example2

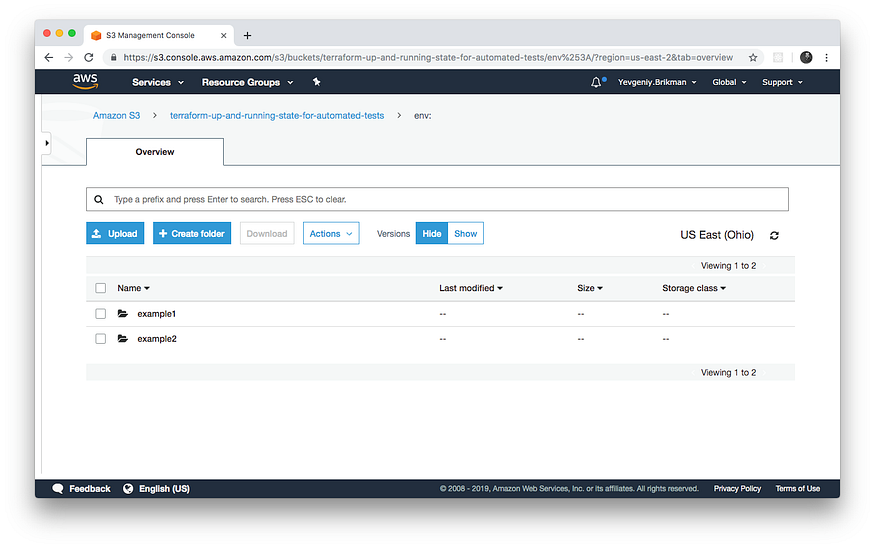
And you can switch between them at any time using the terraform workspace select command:

$ terraform workspace select example1  
Switched to workspace "example1".

To understand how this works under the hood, take a look again in your S3 bucket, you should now see a new folder called *env:*



Inside the *env:*folder, you’ll find one folder for each of your workspaces:



Inside each of those workspaces, Terraform uses the key you specified in your backend configuration, so you should find an *example1/workspaces-example/terraform.tfstate*and an *example2/workspaces-example/terraform.tfstate*. In other words, switching to a different workspace is equivalent to changing the path where your state file is stored.

This is handy when you already have a Terraform module deployed and you want to do some experiments with it (e.g., try to refactor the code) but you don’t want your experiments to affect the state of the already-deployed infrastructure. Terraform workspaces allow you to run terraform workspace new and deploy a new copy of the exact same infrastructure, but storing the state in a separate file.

In fact, you can even change how that module behaves based on the workspace you’re in by reading the workspace name using the expression terraform.workspace. For example, here’s how to set the Instance type to t2.medium in the default workspace and t2.micro in all other workspaces (e.g., to save money when experimenting):

resource "aws\_instance" "example" {  
 ami = "ami-0fb653ca2d3203ac1"  
 instance\_type = (  
 terraform.workspace == "default" ? "t2.medium" : "t2.micro"  
 )  
}

The preceding code uses *ternary syntax*to conditionally set instance\_type to either t2.medium or t2.micro, depending on the value of terraform.workspace. You’ll see the full details of ternary syntax and conditional logic in [Part 5 of this series](https://blog.gruntwork.io/terraform-tips-tricks-loops-if-statements-and-gotchas-f739bbae55f9).

Terraform workspaces can be a great way to quickly spin up and tear down different versions of your code, but they have a few drawbacks:

* The state files for all of your workspaces are stored in the same backend (e.g., the same S3 bucket). That means you use the same authentication and access controls for all the workspaces, which is one major reason workspaces are an unsuitable mechanism for isolating environments (e.g., isolating staging from production).
* Workspaces are not visible in the code or on the terminal unless you run terraform workspace commands. When browsing the code, a module that has been deployed in one workspace looks exactly the same as a module deployed in 10 workspaces. This makes maintenance more difficult, because you don’t have a good picture of your infrastructure.
* Putting the two previous items together, the result is that workspaces can be fairly error prone. The lack of visibility makes it easy to forget what workspace you’re in and accidentally deploy changes in the wrong one (e.g., accidentally running terraform destroy in a “production” workspace rather than a “staging” workspace), and because you must use the same authentication mechanism for all workspaces, you have no other layers of defense to protect against such errors.

Due to these drawbacks, workspaces are not a suitable mechanism for isolating one environment from another: e.g., isolating staging from production (the [workspaces documentation](https://www.terraform.io/language/state/workspaces) makes this same exact point, but it’s buried among several paragraphs of text, and as workspaces used to be called “environments,” I find many users are still confused about when and when not to use workspaces). To get proper isolation between environments, instead of workspaces, you’ll most likely want to use file layout, which is the topic of the next section.

Before moving on, make sure to clean up the three EC2 Instances you just deployed by running terraform workspace select <name> and terraform destroy in each of the three workspaces.

**Isolation via file layout**

To achieve full isolation between environments, you need to do the following:

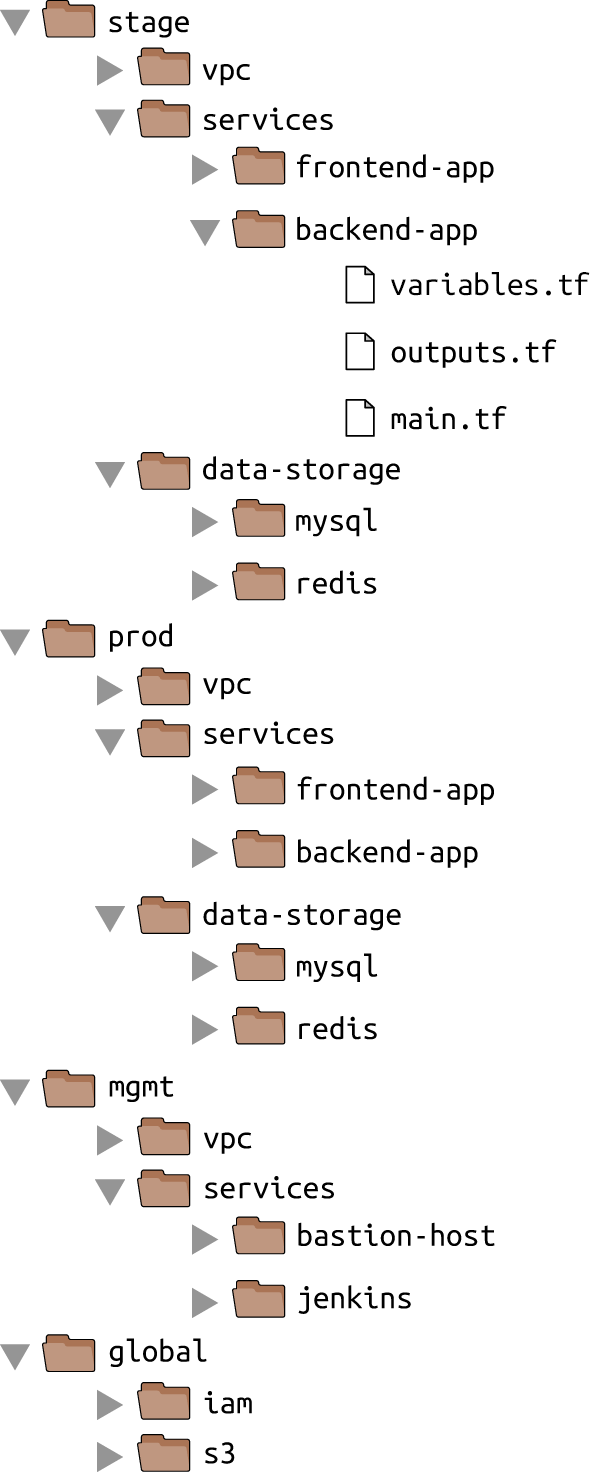
* Put the Terraform configuration files for each environment into a separate folder. For example, all of the configurations for the staging environment can be in a folder called *stage*and all the configurations for the production environment can be in a folder called *prod*.
* Configure a different backend for each environment, using different authentication mechanisms and access controls: e.g., each environment could live in a separate AWS account with a separate S3 bucket as a backend.

With this approach, the use of separate folders makes it much clearer which environments you’re deploying to, and the use of separate state files, with separate authentication mechanisms, makes it significantly less likely that a screw-up in one environment can have any impact on another.

In fact, you might want to take the isolation concept beyond environments and down to the “component” level, where a component is a coherent set of resources that you typically deploy together. For example, after you’ve set up the basic network topology for your infrastructure — in AWS lingo, your Virtual Private Cloud (VPC) and all the associated subnets, routing rules, VPNs, and network ACLs — you will probably change it only once every few months, at most. On the other hand, you might deploy a new version of a web server multiple times per day. If you manage the infrastructure for both the VPC component and the web server component in the same set of Terraform configurations, you are unnecessarily putting your entire network topology at risk of breakage (e.g., from a simple typo in the code or someone accidentally running the wrong command) multiple times per day.

Therefore, I recommend using separate Terraform folders (and therefore separate state files) for each environment (staging, production, etc.) and for each component (VPC, services, databases) within that environment. To see what this looks like in practice, let’s go through the recommended file layout for Terraform projects.

Here’s the file layout for my typical Terraform project:



At the top level, there are separate folders for each “environment.” The exact environments differ for every project, but the typical ones are as follows:

* **stage:**An environment for pre-production workloads (i.e., testing)
* **prod:**An environment for production workloads (i.e., user-facing apps)
* **mgmt:**An environment for DevOps tooling (e.g., bastion host, CI server)
* **global:**A place to put resources that are used across all environments (e.g., S3, IAM)

Within each environment, there are separate folders for each “component.” The components differ for every project, but here are the typical ones:

* **vpc:**The network topology for this environment.
* **services:**The apps or microservices to run in this environment, such as a Ruby on Rails frontend or a Scala backend. Each app could even live in its own folder to isolate it from all the other apps.
* **data-storage:**The data stores to run in this environment, such as MySQL or Redis. Each data store could even reside in its own folder to isolate it from all other data stores.

Within each component, there are the actual Terraform configuration files, which are organized according to the following naming conventions:

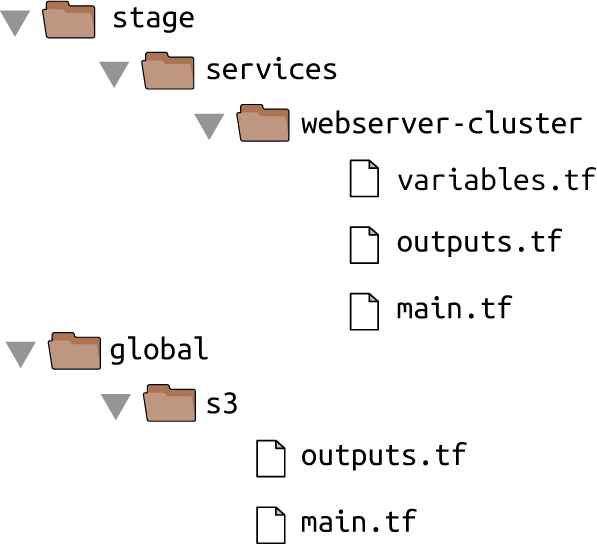
* ***variables.tf*:**Input variables
* ***outputs.tf:***Output variables
* ***main.tf:***Resources and data sources

When you run Terraform, it simply looks for files in the current directory with the *.tf*extension, so you can use whatever filenames you want. However, although Terraform may not care about filenames, your teammates probably do. Using a consistent, predictable naming convention makes your code easier to browse: e.g., you’ll always know where to look to find a variable, output, or resource.

Note that the preceding convention is the *minimum*convention you should follow, because in virtually all uses of Terraform, it’s useful to be able to jump to the input variables, output variables, and resources very quickly, but you may want to go beyond this convention. Here are just a few examples:

* ***dependencies.tf:***It’s common to put all your data sources in a *dependencies.tf*file to make it easier to see what external things the code depends on.
* ***providers.tf:***You may want to put your provider blocks into a *providers.tf*file so you can see, at a glance, what providers the code talks to and what authentication you’ll have to provide.
* ***main-xxx.tf:***If the *main.tf*file is getting really long because it contains a large number of resources, you could break it down into smaller files that group the resources in some logical way: e.g., *main-iam.tf*could contain all the IAM resources, *main-s3.tf*could contain all the S3 resources, and so on. Using the *main-*prefix makes it easier to scan the list of files in a folder when they are organized alphabetically, as all the resources will be grouped together. It’s also worth noting that if you find yourself managing a very large number of resources and struggling to break them down across many files, that might be a sign that you should break your code into smaller modules instead, which is a topic I’ll dive into in [Part 4 of this series](https://blog.gruntwork.io/how-to-create-reusable-infrastructure-with-terraform-modules-25526d65f73d).

Let’s take the web server cluster code you wrote in [Part 2 of the series](https://blog.gruntwork.io/an-introduction-to-terraform-f17df9c6d180), plus the Amazon S3 and DynamoDB code you wrote in this blog post, and rearrange it using the following folder structure:



The S3 bucket you created in this blog post should be moved into the *global/s3*folder. Move the output variables (s3\_bucket\_arn and dynamodb\_table\_name) into *outputs.tf*. When moving the folder, make sure that you don’t miss the (hidden) *.terraform*folder when copying files to the new location so you don’t need to reinitialize everything.

The web server cluster you created in [Part 2 of the series](https://blog.gruntwork.io/an-introduction-to-terraform-f17df9c6d180) should be moved into *stage/services/webserver-cluster*(think of this as the “testing” or “staging” version of that web server cluster; you’ll add a “production” version in the next part of this series). Again, make sure to copy over the *.terraform*folder, move input variables into *variables.tf*, and move output variables into *outputs.tf*.

You should also update the web server cluster to use S3 as a backend. You can copy and paste the backend config from *global/s3/main.tf*more or less verbatim, but make sure to change the key to the same folder path as the web server Terraform code: *stage/services/webserver-cluster/terraform.tfstate*. This gives you a 1:1 mapping between the layout of your Terraform code in version control and your Terraform state files in S3, so it’s obvious how the two are connected. The s3 module already sets the key using this convention.

This file layout has a number of advantages:

* **Clear code / environment layout:**It’s easy to browse the code and understand exactly what components are deployed in each environment.
* **Isolation:**This layout provides a good amount of isolation between environments and between components within an environment, ensuring that if something goes wrong, the damage is contained as much as possible to just one small part of your entire infrastructure.

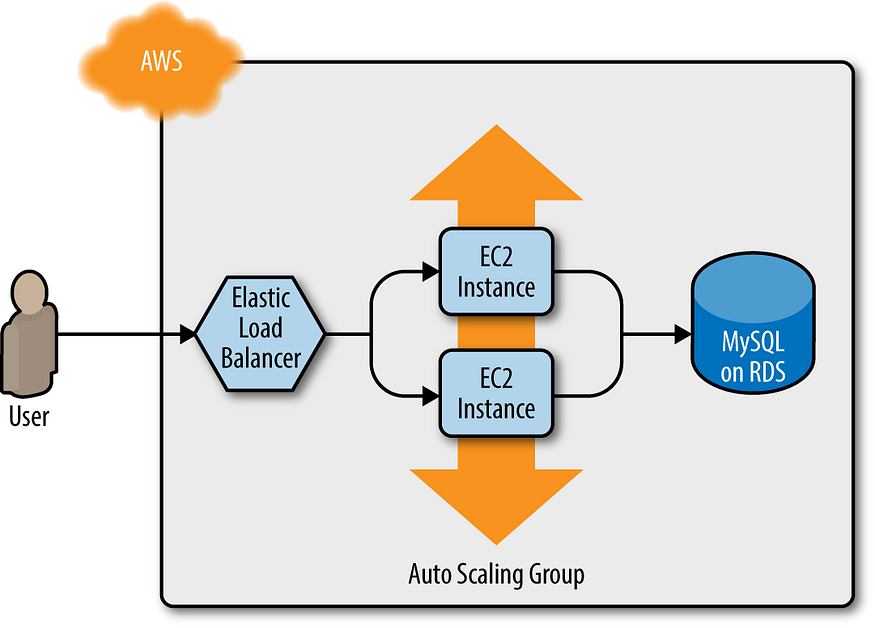
In some ways, these advantages are drawbacks, too:

* **Working with multiple folders:**Splitting components into separate folders prevents you from accidentally blowing up your entire infrastructure in one command, but it also prevents you from creating your entire infrastructure in one command. If all of the components for a single environment were defined in a single Terraform configuration, you could spin up an entire environment with a single call to terraform apply. But if all of the components are in separate folders, then you need to run terraform apply separately in each one. ***Solution***: If you use [Terragrunt](https://terragrunt.gruntwork.io/), you can run commands across multiple folders concurrently using the [run-all](https://terragrunt.gruntwork.io/docs/features/execute-terraform-commands-on-multiple-modules-at-once/) command.
* **Copy/paste:**The file layout described in this section has a lot of duplication. For example, the same frontend-app and backend-app live in both the *stage*and *prod*folders. ***Solution***: You won’t actually need to copy and paste all of that code! In [Part 4 of this series](https://blog.gruntwork.io/how-to-create-reusable-infrastructure-with-terraform-modules-25526d65f73d), you’ll see how to use Terraform modules to keep all of this code DRY.
* **Resource dependencies:**Breaking the code into multiple folders makes it more difficult to use resource dependencies. If your app code was defined in the same Terraform configuration files as the database code, that app code could directly access attributes of the database using an attribute reference (e.g., access the database address via aws\_db\_instance.foo.address). But if the app code and database code live in different folders, as I’ve recommended, you can no longer do that. ***Solution***: One option is to use [dependency](https://terragrunt.gruntwork.io/docs/reference/config-blocks-and-attributes/#dependency) blocks in Terragrunt, as you’ll see in [How to use Terraform as a Team](https://blog.gruntwork.io/how-to-use-terraform-as-a-team-251bc1104973). Another option is to use the terraform\_remote\_state data source, as described in the next section.

**The terraform\_remote\_state data source**

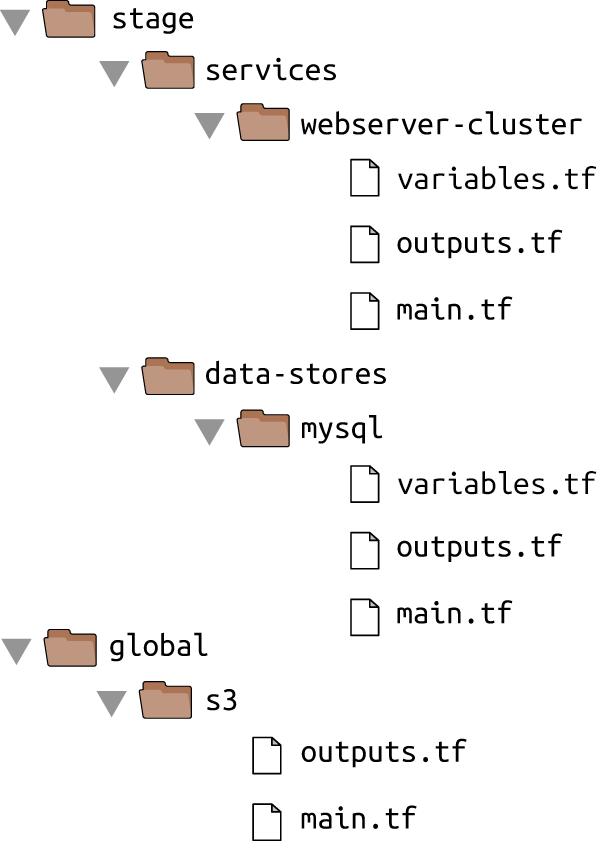
In [Part 2, An Introduction to Terraform](https://blog.gruntwork.io/an-introduction-to-terraform-f17df9c6d180#.j79allnik), you used data sources to fetch read-only information from AWS, such as the aws\_subnets data source, which returns a list of subnets in your VPC. There is another data source that is particularly useful when working with state: terraform\_remote\_state. You can use this data source to fetch the Terraform state file stored by another set of Terraform configurations.

Let’s go through an example. Imagine that your web server cluster needs to communicate with a MySQL database. Running a database that is scalable, secure, durable, and highly available is a lot of work. Again, you can let AWS take care of it for you, this time by using Amazon’s [*Relational Database Service*](https://aws.amazon.com/rds/)(RDS). RDS supports a variety of databases, including MySQL, PostgreSQL, SQL Server, and Oracle.



You might not want to define the MySQL database in the same set of configuration files as the web server cluster, because you’ll be deploying updates to the web server cluster far more frequently and don’t want to risk accidentally breaking the database each time you do so.

Therefore, your first step should be to create a new folder at *stage/data-stores/mysql*and create the basic Terraform files (*main.tf*, *variables.tf*, *outputs.tf*) within it:



Next, create the database resources in *stage/data-stores/mysql/main.tf*:

provider "aws" {  
 region = "us-east-2"  
}  
  
resource "aws\_db\_instance" "example" {  
 identifier\_prefix = "terraform-up-and-running"  
 engine = "mysql"  
 allocated\_storage = 10  
 instance\_class = "db.t2.micro"  
 skip\_final\_snapshot = true  
 db\_name = "example\_database" # How should we set the username and password?  
 username = "???"  
 password = "???"  
}

At the top of the file, you see the typical provider block, but just below that is a new resource: aws\_db\_instance. This resource creates a database in RDS with the following settings:

* MySQL as the database engine.
* 10 GB of storage.
* A db.t2.micro Instance, which has one virtual CPU, 1 GB of memory, and is part of the AWS Free Tier.
* The final snapshot is disabled, as this code is just for learning and testing (if you don’t disable the snapshot, or don’t provide a name for the snapshot via the final\_snapshot\_identifier parameter, destroy will fail).

Note that two of the parameters that you must pass to the aws\_db\_instance resource are the master username and master password. Because these are secrets, you should not put them directly into your code in plain text! In [A comprehensive guide to managing secrets in your Terraform code](https://blog.gruntwork.io/a-comprehensive-guide-to-managing-secrets-in-your-terraform-code-1d586955ace1), I discuss a variety of options for how to securely handle secrets with Terraform. For now, let’s use an option that avoids storing any secrets in plain text and is easy to use: you store your secrets, such as database passwords, outside of Terraform (e.g., in a password manager such as 1Password, LastPass, or macOS Keychain), and you pass those secrets into Terraform via environment variables.

To do that, declare variables called db\_username and db\_password in *stage/data-stores/mysql/variables.tf*:

variable "db\_username" {  
 description = "The username for the database"  
 type = string  
 sensitive = true  
}  
  
variable "db\_password" {  
 description = "The password for the database"  
 type = string  
 sensitive = true  
}

First, note that these variables are marked with sensitive = true to indicate they contain secrets. This ensures Terraform won’t log the values when you run plan or apply. Second, note that these variables do not have a default. This is intentional. You should not store your database credentials or any sensitive information in plain text. Instead, you’ll set these variables using environment variables.

Before doing that, let’s finish the code. First, pass the two new input variables through to the aws\_db\_instance resource:

resource "aws\_db\_instance" "example" {  
 identifier\_prefix = "terraform-up-and-running"  
 engine = "mysql"  
 allocated\_storage = 10  
 instance\_class = "db.t2.micro"  
 skip\_final\_snapshot = true  
 db\_name = "example\_database" username = var.db\_username  
 password = var.db\_password  
}

Next, configure this module to store its state in the S3 bucket you created earlier at the path *stage/data-stores/mysql/terraform.tfstate*:

terraform {  
 backend "s3" {  
 # Replace this with your bucket name!  
 bucket = "terraform-up-and-running-state"  
 key = "stage/data-stores/mysql/terraform.tfstate"  
 region = "us-east-2"  
  
 # Replace this with your DynamoDB table name!  
 dynamodb\_table = "terraform-up-and-running-locks"  
 encrypt = true  
 }  
}

Finally, add two output variables in *stage/data-stores/mysql/outputs.tf*to return the database’s address and port:

output "address" {  
 value = aws\_db\_instance.example.address  
 description = "Connect to the database at this endpoint"  
}  
  
output "port" {  
 value = aws\_db\_instance.example.port  
 description = "The port the database is listening on"  
}

You’re now ready to pass in the database username and password using environment variables. As a reminder, for each input variable foo defined in your Terraform configurations, you can provide Terraform the value of this variable using the environment variable TF\_VAR\_foo. For the db\_username and db\_password input variables, here is how you can set the TF\_VAR\_db\_username and TF\_VAR\_db\_password environment variables on Linux/Unix/macOS systems:

$ export TF\_VAR\_db\_username="(YOUR\_DB\_USERNAME)"  
$ export TF\_VAR\_db\_password="(YOUR\_DB\_PASSWORD)"

And here is how you do it on Windows systems:

$ set TF\_VAR\_db\_username="(YOUR\_DB\_USERNAME)"  
$ set TF\_VAR\_db\_password="(YOUR\_DB\_PASSWORD)"

Run terraform init and terraform apply to create the database. Note that Amazon RDS can take roughly 10 minutes to provision even a small database, so be patient. After apply completes, you should see the outputs in the terminal:

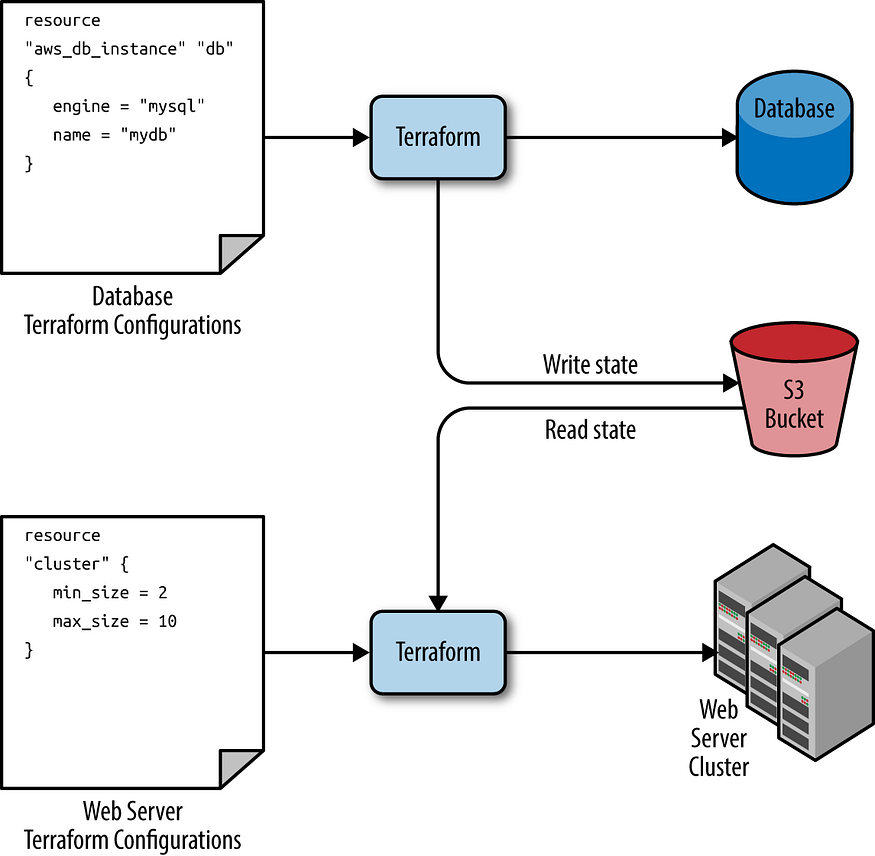
$ terraform apply  
  
(...)  
  
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.  
  
Outputs:  
  
address = "terraform-up-and-running.xxx.us-east-2.rds.amazonaws.com"  
port = 3306

These outputs are now also stored in the Terraform state for the database, which is in your S3 bucket at the path *stage/data-stores/mysql/terraform.tfstate*.

If you go back to your web server cluster code, you can get the web server to read those outputs from the database’s state file by adding the terraform\_remote\_state data source in *stage/services/webserver-cluster/main.tf*:

data "terraform\_remote\_state" "db" {  
 backend = "s3"  
  
 config = {  
 bucket = "(YOUR\_BUCKET\_NAME)"  
 key = "stage/data-stores/mysql/terraform.tfstate"  
 region = "us-east-2"  
 }  
}

This terraform\_remote\_state data source configures the web server cluster code to read the state file from the same S3 bucket and folder where the database stores its state:



It’s important to understand that, like all Terraform data sources, the data returned by terraform\_remote\_state is read-only. Nothing you do in your web server cluster Terraform code can modify that state, so you can pull in the database’s state data with no risk of causing any problems in the database itself.

All of the database’s output variables are stored in the state file, and you can read them from the terraform\_remote\_state data source using an attribute reference of the form:

data.terraform\_remote\_state.<NAME>.outputs.<ATTRIBUTE>

For example, here is how you can update the User Data of the web server cluster Instances to pull the database address and port out of the terraform\_remote\_state data source and expose that information in the HTTP response:

user\_data = <<EOF  
#!/bin/bash  
echo "Hello, World" >> index.html  
echo "${data.terraform\_remote\_state.db.outputs.address}">>index.html  
echo "${data.terraform\_remote\_state.db.outputs.port}">>index.html  
nohup busybox httpd -f -p ${var.server\_port} &  
EOF

As the User Data script is growing longer, defining it inline is becoming messier and messier. In general, embedding one programming language (Bash) inside another (Terraform) makes it more difficult to maintain each one, so let’s pause here for a moment to externalize the Bash script. To do that, you can use the templatefile built-in function.

Terraform includes a number of *built-in functions*that you can execute using an expression of the form:

function\_name(...)

For example, consider the format function:

format(<FMT>, <ARGS>, ...)

This function formats the arguments in ARGS according to the [sprintf syntax](https://pkg.go.dev/fmt) in the string FMT. A great way to experiment with built-in functions is to run the terraform console command to get an interactive console where you can try out Terraform syntax, query the state of your infrastructure, and see the results instantly:

$ terraform console  
  
> format("%.3f", 3.14159265359)  
3.142

Note that the Terraform console is read-only, so you don’t need to worry about accidentally changing infrastructure or state.

There are a number of other [built-in functions](https://www.terraform.io/language/functions) that you can use to manipulate strings, numbers, lists, and maps. One of them is the templatefile function:

templatefile(<PATH>, <VARS>)

This function reads the file at PATH, renders it as a template, and returns the result as a string. When I say “renders it as a template,” what I mean is that the file at PATH can use the string interpolation syntax in Terraform (${…}), and Terraform will render the contents of that file, filling variable references from VARS.

To see this in action, put the contents of the User Data script into the file *stage/services/webserver-cluster/user-data.sh*as follows:

#!/bin/bash  
  
cat > index.html <<EOF  
<h1>Hello, World</h1>  
<p>DB address: ${db\_address}</p>  
<p>DB port: ${db\_port}</p>  
EOF  
  
nohup busybox httpd -f -p ${server\_port} &

Note that this Bash script has a few changes from the original:

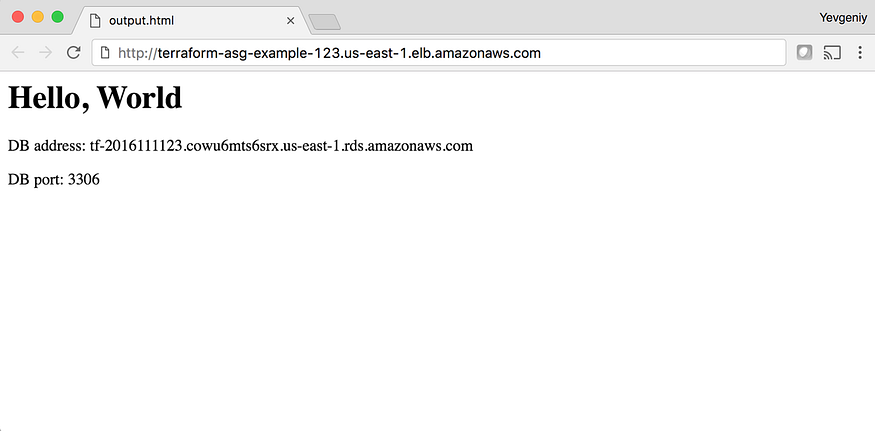
* It looks up variables using Terraform’s standard interpolation syntax, except the only variables it has access to are those you pass in via the second parameter to templatefile (as you’ll see shortly), so you don’t need any prefix to access them: for example, you should use ${server\_port} and not ${var.server\_port}.
* The script now includes some HTML syntax (e.g., <h1>) to make the output a bit more readable in a web browser.

The final step is to update the user\_data parameter of the aws\_launch\_configuration resource to call the templatefile function and pass in the variables it needs as a map:

resource "aws\_launch\_configuration" "example" {  
 image\_id = "ami-0fb653ca2d3203ac1"  
 instance\_type = "t2.micro"  
 security\_groups = [aws\_security\_group.instance.id]  
  
 # Render the User Data script as a template  
 user\_data = templatefile("user-data.sh", {  
 server\_port = var.server\_port  
 db\_address = data.terraform\_remote\_state.db.outputs.address  
 db\_port = data.terraform\_remote\_state.db.outputs.port  
 })  
  
 # Required when using a launch configuration with an ASG.  
 lifecycle {  
 create\_before\_destroy = true  
 }  
}

Ah, that’s much cleaner than writing Bash scripts inline!

If you deploy this cluster using terraform apply, wait for the Instances to register in the ALB, and open the ALB URL in a web browser, you’ll see something similar to this:



Congrats, your web server cluster can now programmatically access the database address and port via Terraform. If you were using a real web framework (e.g., Ruby on Rails), you could set the address and port as environment variables or write them to a config file so that they could be used by your database library (e.g., ActiveRecord) to communicate with the database.