To enable remote state storage with Amazon S3, the first step is to create an S3 bucket. Create a *main.tf*file in a new folder (it should be a different folder from where you store the configurations from Part 1 of this series), and at the top of the file, specify AWS as the provider:

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

Next, create an S3 bucket by using the aws\_s3\_bucket resource:

resource "aws\_s3\_bucket" "terraform\_state" {  
 bucket = "terraform-up-and-running-state"  
   
 # Prevent accidental deletion of this S3 bucket  
 lifecycle {  
 prevent\_destroy = true  
 }  
}

This code sets the following arguments:

* **bucket**: This is the name of the S3 bucket. Note that S3 bucket names must be *globally*unique among all AWS customers. Therefore, you will need to change the bucket parameter from “terraform-up-and-running-state” (which I already created) to your own name. Make sure to remember this name and take note of what AWS region you’re using because you’ll need both pieces of information again a little later on.
* **prevent\_destroy**: prevent\_destroy is the second lifecycle setting you’ve seen (the first was create\_before\_destroy in [Part 2 of this series](https://blog.gruntwork.io/an-introduction-to-terraform-f17df9c6d180)). When you set prevent\_destroy to true on a resource, any attempt to delete that resource (e.g., by running terraform destroy) will cause Terraform to exit with an error. This is a good way to prevent accidental deletion of an important resource, such as this S3 bucket, which will store all of your Terraform state. Of course, if you really mean to delete it, you can just comment that setting out.

Let’s now add several extra layers of protection to this S3 bucket.

First, use the aws\_s3\_bucket\_versioning resource to enable versioning on the S3 bucket so that every update to a file in the bucket actually creates a new version of that file. This allows you to see older versions of the file and revert to those older versions at any time, which can be a useful fallback mechanism if something goes wrong:

resource "aws\_s3\_bucket\_versioning" "enabled" {  
 bucket = aws\_s3\_bucket.terraform\_state.id  
 versioning\_configuration {  
 status = "Enabled"  
 }  
}

Second, use the aws\_s3\_bucket\_server\_side\_encryption\_configuration resource to turn server-side encryption on by default for all data written to this S3 bucket. This ensures that your state files, and any secrets they might contain, are always encrypted on disk when stored in S3:

resource "aws\_s3\_bucket\_server\_side\_encryption\_configuration" "default" {  
 bucket = aws\_s3\_bucket.terraform\_state.id  
  
 rule {  
 apply\_server\_side\_encryption\_by\_default {  
 sse\_algorithm = "AES256"  
 }  
 }  
}

Third, use the aws\_s3\_bucket\_public\_access\_block resource to block all public access to the S3 bucket. S3 buckets are private by default, but as they are often used to serve static content — e.g., images, fonts, CSS, JS, HTML — it is possible, even easy, to make the buckets public. Since your Terraform state files may contain sensitive data and secrets, it’s worth adding this extra layer of protection to ensure no one on your team can ever accidentally make this S3 bucket public:

resource "aws\_s3\_bucket\_public\_access\_block" "public\_access" {  
 bucket = aws\_s3\_bucket.terraform\_state.id  
 block\_public\_acls = true  
 block\_public\_policy = true  
 ignore\_public\_acls = true  
 restrict\_public\_buckets = true  
}

Next, you need to create a DynamoDB table to use for locking. [DynamoDB](https://aws.amazon.com/dynamodb/) is Amazon’s distributed key-value store. It supports strongly consistent reads and conditional writes, which are all the ingredients you need for a distributed lock system. Moreover, it’s completely managed, so you don’t have any infrastructure to run yourself, and it’s inexpensive, with most Terraform usage easily fitting into the AWS Free Tier.

To use DynamoDB for locking with Terraform, you must create a DynamoDB table that has a primary key called LockID (with this *exact*spelling and capitalization). You can create such a table using the aws\_dynamodb\_table resource:

resource "aws\_dynamodb\_table" "terraform\_locks" {  
 name = "terraform-up-and-running-locks"  
 billing\_mode = "PAY\_PER\_REQUEST"  
 hash\_key = "LockID"  
  
 attribute {  
 name = "LockID"  
 type = "S"  
 }  
}

Run terraform init to download the provider code, and then run terraform apply to deploy. After everything is deployed, you will have an S3 bucket and DynamoDB table, but your Terraform state will still be stored locally. To configure Terraform to store the state in your S3 bucket (with encryption and locking), you need to add a backend configuration to your Terraform code. This is configuration for Terraform itself, so it resides within a terraform block and has the following syntax:

terraform {  
 backend "<BACKEND\_NAME>" {  
 [CONFIG...]  
 }  
}

where BACKEND\_NAME is the name of the backend you want to use (e.g., “s3”) and CONFIG consists of one or more arguments that are specific to that backend (e.g., the name of the S3 bucket to use). Here’s what the backend configuration looks like for an S3 bucket:

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

Let’s go through these settings one at a time:

* **bucket:**The name of the S3 bucket to use. Make sure to replace this with the name of the S3 bucket you created earlier.
* **key**: The filepath within the S3 bucket where the Terraform state file should be written. You’ll see a little later on why the preceding example code sets this to *global/s3/terraform.tfstate*.
* **region**: The AWS region where the S3 bucket lives. Make sure to replace this with the region of the S3 bucket you created earlier.
* **dynamodb\_table**: The DynamoDB table to use for locking. Make sure to replace this with the name of the DynamoDB table you created earlier.
* **encrypt**: Setting this to true ensures that your Terraform state will be encrypted on disk when stored in S3. We already enabled default encryption in the S3 bucket itself, so this is here as a second layer to ensure that the data is always encrypted.

To instruct Terraform to store your state file in this S3 bucket, you’re going to use the terraform init -backend-config="access\_key=AKIAUJU24ZR3ZBGGXT4E" -backend-config="secret\_key=D2Oq3iFeZurL/OULXSPZozoC1OZUHF7GC54tCNuc" command again. This command not only can download provider code, but also configure your Terraform backend (and you’ll see yet another use later on, too). Moreover, the init command is idempotent, so it’s safe to run it multiple times:

$ terraform init -backend-config="access\_key=AKIAUJU24ZR3ZBGGXT4E" -backend-config="secret\_key=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"  
  
Initializing the backend...  
Acquiring state lock. This may take a few moments...  
Do you want to copy existing state to the new backend?  
 Pre-existing state was found while migrating the previous "local"   
 backend to the newly configured "s3" backend. No existing state   
 was found in the newly configured "s3" backend. Do you want to   
 copy this state to the new "s3" backend? Enter "yes" to copy and   
 "no" to start with an empty state.  
  
 Enter a value:

Terraform will automatically detect that you already have a state file locally and prompt you to copy it to the new S3 backend. If you type **yes**, you should see the following:

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

After running this command, your Terraform state will be stored in the S3 bucket. You can check this by heading over to the [S3 Management Console](https://console.aws.amazon.com/s3/) in your browser and clicking your bucket.

**Limitations with Terraform’s Backends**

Terraform’s backends have a few limitations and gotchas that you need to be aware of. The first limitation is the chicken-and-egg situation of using Terraform to create the S3 bucket where you want to store your Terraform state. To make this work, you had to use a two-step process:

1. Write Terraform code to create the S3 bucket and DynamoDB table, and deploy that code with a local backend.
2. Go back to the Terraform code, add a remote backend configuration to it to use the newly created S3 bucket and DynamoDB table, and run terraform init to copy your local state to S3.

If you ever wanted to delete the S3 bucket and DynamoDB table, you’d have to do this two-step process in reverse:

1. Go to the Terraform code, remove the backend configuration, and rerun terraform init to copy the Terraform state back to your local disk.
2. Run terraform destroy to delete the S3 bucket and DynamoDB table.

This two-step process is a bit awkward, but the good news is that you can share a single S3 bucket and DynamoDB table across all of your Terraform code, so you’ll probably only need to do it once (or once per AWS account if you have multiple accounts). After the S3 bucket exists, in the rest of your Terraform code, you can specify the backend configuration right from the start without any extra steps.

The second limitation is more painful: the backend block in Terraform does not allow you to use any variables or references. The following code will *not*work:

# This will NOT work. Variables aren't allowed in a backend configuration.  
terraform {  
 backend "s3" {  
 bucket = var.bucket  
 region = var.region  
 dynamodb\_table = var.dynamodb\_table  
 key = "example/terraform.tfstate"  
 encrypt = true  
 }  
}

This means that you need to manually copy and paste the S3 bucket name, region, DynamoDB table name, etc., OR we can use modules; modules are a way to organize and reuse Terraform code and that real-world Terraform code typically consists of many small modules). Even worse, you must very carefully *not*copy and paste the key value but ensure a unique key for every Terraform module you deploy so that you don’t accidentally overwrite the state of some other module! Having to do lots of copy-and-pastes *and*lots of manual changes is error prone, especially if you need to deploy and manage many Terraform modules across many environments.

One option for reducing copy-and-paste is to use *partial configurations*, where you omit certain parameters from the backend configuration in your Terraform code and instead pass those in via -backend-config command-line arguments when calling terraform init. For example, you could extract the repeated *backend*arguments, such as bucket and region, into a separate file called *backend.hcl*:

# backend.hcl  
bucket = "terraform-up-and-running-state"  
region = "us-east-2"  
dynamodb\_table = "terraform-up-and-running-locks"  
encrypt = true

Only the key parameter remains in the Terraform code, since you still need to set a different key value for each module:

# Partial configuration. The other settings (e.g., bucket, region)   
# will be passed in from a file via -backend-config arguments to   
# 'terraform init'  
terraform {  
 backend "s3" {  
 key = "example/terraform.tfstate"  
 }  
}

To put all your partial configurations together, run terraform init with the -backend-config argument:

$ terraform init -backend-config=backend.hcl

Terraform merges the partial configuration in *backend.hcl*with the partial configuration in your Terraform code to produce the full configuration used by your module. You can use the same *backend.hcl*file with all of your modules, which reduces duplication considerably; however, you’ll still need to manually set a unique key value in every module.

Another option for reducing copy-and-paste is to use [Terragrunt](https://terragrunt.gruntwork.io/" \t "_blank), an open source tool that tries to fill in a few gaps in Terraform. Terragrunt can help you keep your entire backend configuration DRY (Don’t Repeat Yourself) by defining all the basic backend settings (bucket name, region, DynamoDB table name) in one file and automatically setting the key argument to the relative folder path of the module.