

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 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 in the chapter 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. As shown in [Figure 3-4](#), if you take a look in

your S3 bucket, you'll find a *terraform.tfstate* file in the *workspaces-example* folder.

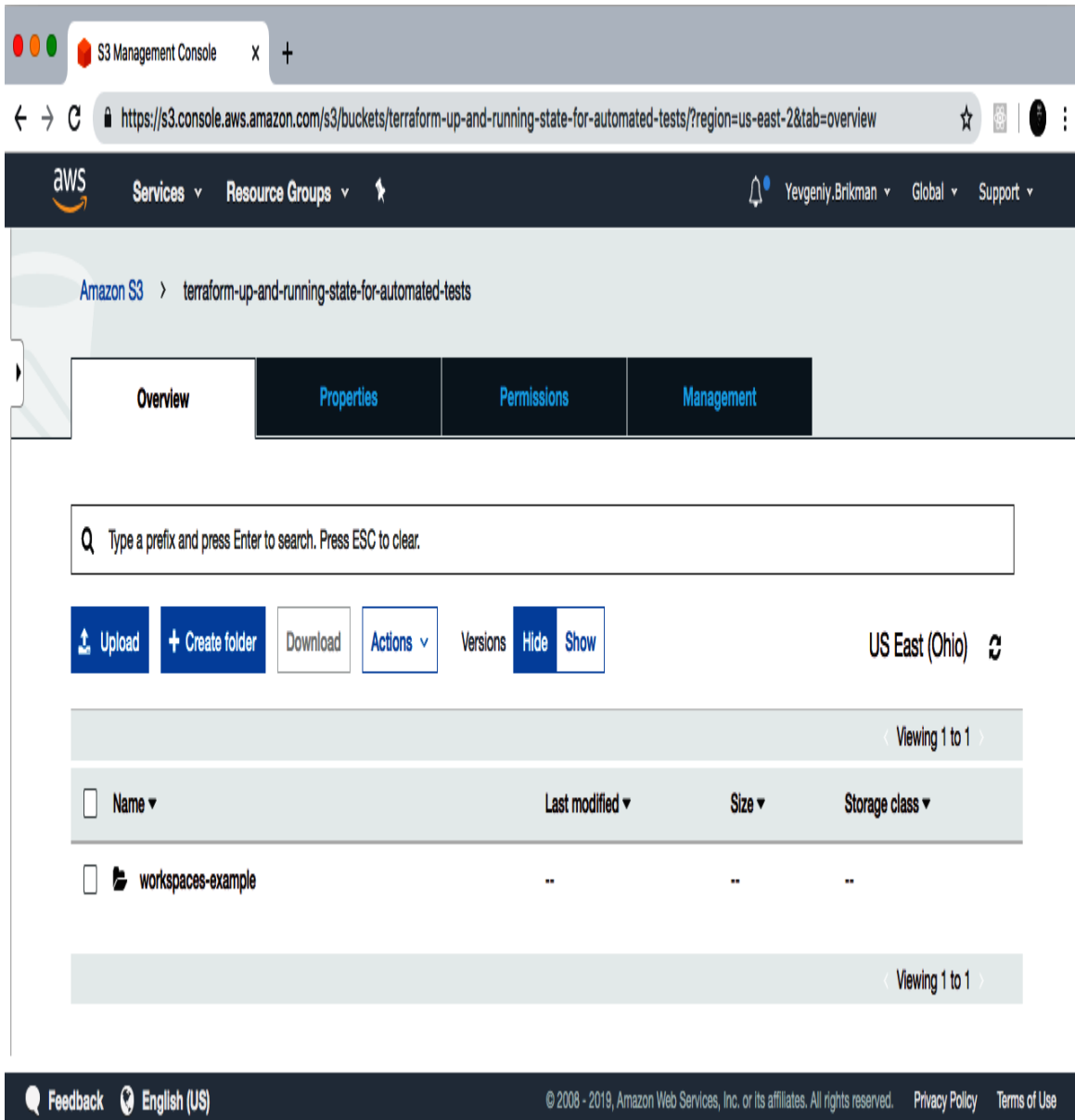


Figure 3-4. When using the default workspace, the S3 bucket will have just a single folder and state file in it.

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:*, as shown in **Figure 3-5**.

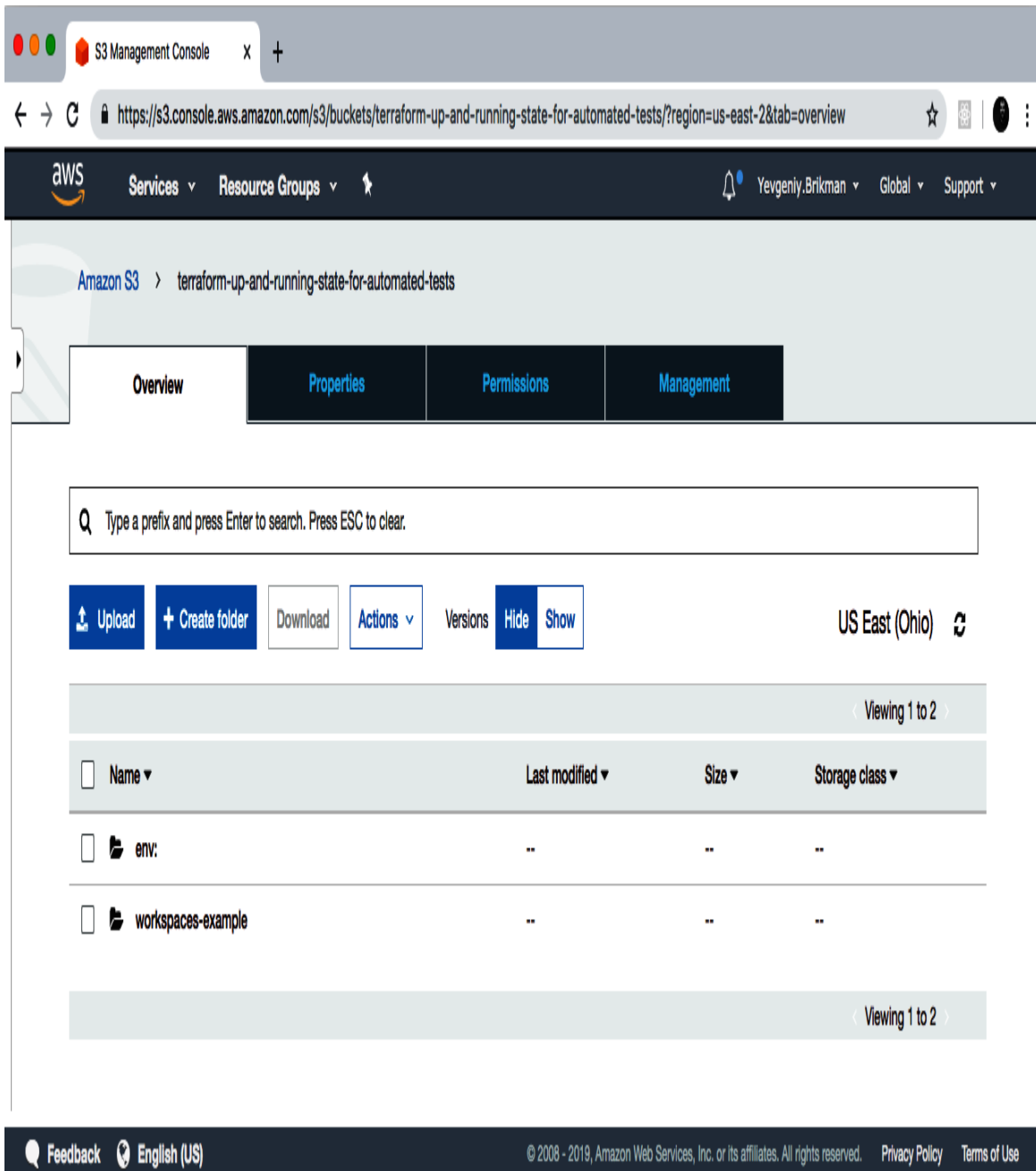


Figure 3-5. When using custom workspaces, the S3 bucket will have multiple folders and state files in it.

Inside the *env:* folder, you'll find one folder for each of your workspaces, as shown in [Figure 3-6](#).

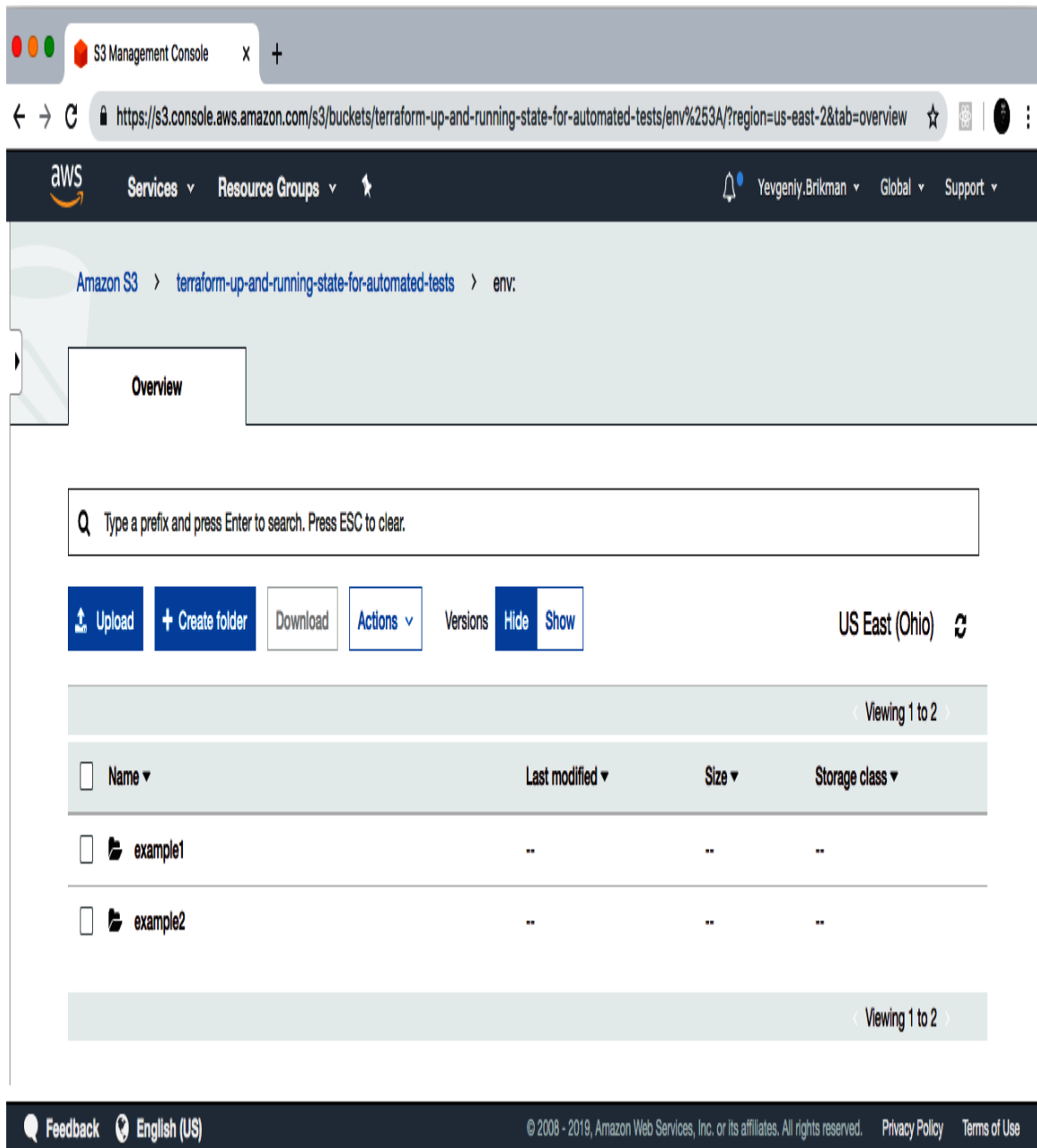


Figure 3-6. Terraform creates one folder per workspace.

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 Terraform in [Chapter 5](#).

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