

Managing Terraform State

The Plan

- ◆ What is Terraform state?
- ◆ Shared storage for state files
- ◆ Limitations with Terraform's back-ends
- ◆ Isolating state files
 - Isolation via workspaces
 - Isolation via file layout
- ◆ The `terraform_remote_state` data source

What Is Terraform State?

- ◆ Every time you run Terraform, it records information about what infrastructure it created in a Terraform state file
- ◆ By default, when you run Terraform in the folder /foo/bar, Terraform creates the file /foo/bar/terraform.tfstate.
- ◆ Say, your Terraform configuration shows below

```
1 resource "aws_instance" "example" {  
2   ami           = "ami-0c55b159cbfaffe1f0"  
3   instance_type = "t2.micro"  
4 }
```

Example of "terraform.tfstate"

- ◆ After you run `terraform apply` you will see the output in `terraform.tfstate`

```
{
  "version": 4,
  "terraform_version": "0.12.0",
  "serial": 1,
  "lineage": "1f2087f9-4b3c-1b66-65db-8b78faafc6fb",
  "outputs": {},
  "resources": [
    {
      "mode": "managed",
      "type": "aws_instance",
      "name": "example",
      "provider": "provider.aws",
      "instances": [
        {
          "schema_version": 1,
          "attributes": {
            "ami": "ami-0c55b159cbfafa1f0",
            "availability_zone": "us-east-2c",
            "id": "i-00d689a0acc43af0f",
            "instance_state": "running",
            "instance_type": "t2.micro",
            "(...)": "(truncated)"
          }
        }
      ]
    }
  ]
}
```

Meaning of "terraform.tfstate"

- ◆ Resource with `type` `aws_instance` and `name` `example` corresponds to an EC2 Instance in your AWS account with ID `i-00d689a0acc43af0f`
- ◆ Every time you run Terraform
 - it can fetch the latest status of this EC2 Instance from AWS
 - compare that to what's in your Terraform configurations
 - determine what changes need to be applied
- ◆ Thus, the output of the `terraform plan` command is a diff
 - between the code on your computer and
 - the infrastructure deployed in the real world, as discovered via IDs in the state file

Managing Terraform States

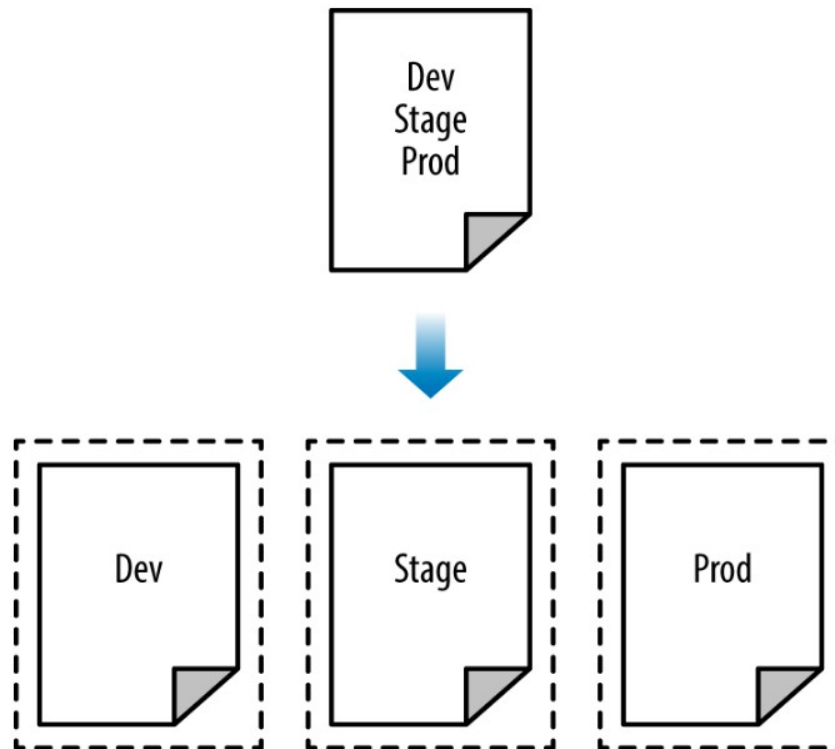
- ◆ State file tracks managed resources created by Terraform
- ◆ Other resources are ignored unless there is conflict
- ◆ Existing resources can be added to a Terraform state
- ◆ Resources can be removed from a Terraform state
- ◆ Multiple state files can be used
- ◆ Each state file manages a only its set of resources

The

- ◆ The state command has multiple options (not all are listed)
 - `terraform state list` : lists the resources being managed
 - `terraform state show <resource>` : displays state data for a resource
 - `terraform state rm <resource>` : stops managing the AWS object linked to <resource>
- ◆ `terraform import <resource> <AWS ID>` : links the Terraform resource with a terraform resource

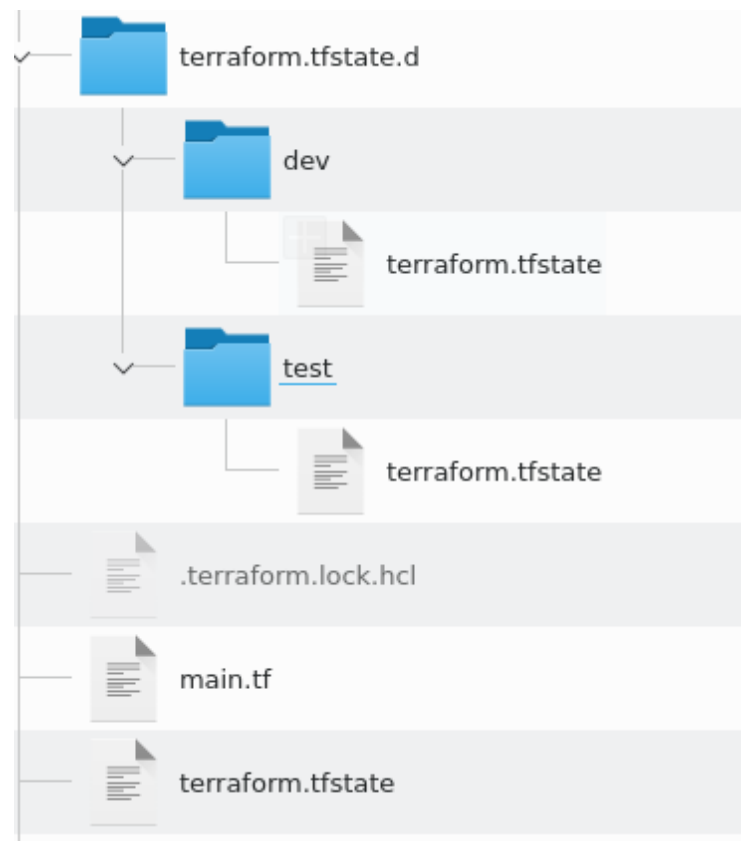
Separate Environments

- ◆ We often need multiple copies of a deployment for different purposes
- ◆ Common environments are: development, test, stage and production



Terraform Workspaces

- ◆ Terraform supports a separate configuration for each deployment
 - Each deployment is called a workspace
 - There is always a default workspace
- ◆ We can create additional workspaces as we need them
 - For example, we could have defined dev and test workspaces



- ◆ terraform workspace has several options:
 - list : lists all workspaces marking the current one with "*"
 - show : lists the currently active workspace
 - new <name> : creates and switches to the newly created workspace
 - select <name> : switches to the named workspace
 - delete <name> : deletes the named workspace
 - The "default" workspace can never be deleted
 - Deleting a workspace does **not** destroy the resources, it just leaves them not managed

Remote Back-ends

- ◆ Each Terraform configuration specifies where the state files are kept
 - This is called the "back-end"
 - The default is to use files in the local directory
 - This is what we have been using so far
- ◆ Terraform can also support "remote" back-ends
 - For example, we can keep state files in an S3 bucket on AWS
 - Not all providers can host remote back ends

Problem with Local Backends

- ◆ Shared storage for state files
 - Files need to be in common shared area so everyone on the team can access them
 - Without file locking, race conditions when concurrent updates to the state files take place
 - This can lead to conflicts, data loss, and state file corruption
- ◆ Isolation
 - It's difficult to isolate the code used in different environments
 - Lack of isolation makes it easy to accidentally overwrite environments
- ◆ Secrets
 - Confidential information is stored in the clear (i.e. AWS Keys)

Remote AWS Back-end

- ◆ Using S3 as a back-end resolves many of these issues
 - S3 manages the updating and access independently, and supports versions
 - S3 supports encryption
 - S3 supports locking for multiple access
 - S3 allows a common repository we can control access to
- ◆ S3 is also managed so that we don't have to manage it
 - S3 has high levels of availability and durability
 - S3 also means we have reduced the risk of "loosing" configurations

Setting up the S3 Bucket

- ◆ For back-end state

```
1 resource "aws_s3_bucket" "terraform_state" {
2     bucket = "terraform-up-and-running-state"
3
4     # Prevent accidental deletion of this S3 bucket
5     lifecycle {
6         prevent_destroy = true
7     }
8
9     # Enable versioning so we can see the full revision history of our
10    # state files
11    versioning {
12        enabled = true
13    }
14
15    # Enable server-side encryption by default
16    server_side_encryption_configuration {
17        rule {
18            apply_server_side_encryption_by_default {
19                sse_algorithm = "AES256"
20            }
21        }
22    }
23 }
```

Setting up the Locking Table

- ◆ Next, a DynamoDB table to use for locking
 - DynamoDB is Amazon's distributed key-value store
 - It supports strongly consistent reads and conditional writes

```
1 resource "aws_dynamodb_table" "terraform_locks" {  
2     name           = "terraform-up-and-running-locks"  
3     billing_mode    = "PAY_PER_REQUEST"  
4     hash_key       = "LockID"  
5  
6     attribute {  
7         name = "LockID"  
8         type  = "S"  
9     }  
10 }
```

Setting Up the Back-end

- ◆ We have to tell Terraform the back-end is now remote
 - We do this in the `terraform` directive

- ◆ More specifically:

```
1 terraform {  
2     backend "< BACKEND_NAME >" {  
3         [CONFIG...]  
4     }  
5 }
```

```
1 terraform {  
2     backend "s3" {  
3         # Replace this with your bucket name!  
4         bucket      = "terraform-up-and-running-state"  
5         key         = "global/s3/terraform.tfstate"  
6         region      = "us-east-2"  
7  
8         # Replace this with your DynamoDB table name!  
9         dynamodb_table = "terraform-up-and-running-locks"  
10        encrypt      = true  
11    }  
12 }
```


Moving State File Locations

- ◆ To move local state to a remote back-end
 - Create the remote back-end resources and define the back-end configuration
 - Run `terraform init` and the local config is copied to the remote back-end
- ◆ To move from remote back-end to a local back-end
 - Remove the back-end configuration
 - Run `terraform init` and the remote config is copied to the local back-end

Moving Back-ends Summary

- ◆ To make this work, you need to use a two-step process:
 - Create the S3 bucket and DynamoDB table and deploy that code with a local back-end
 - Add a remote back-end configuration to it to use the S3 bucket and DynamoDB table
 - Run terraform init to copy your local state to S3
- ◆ To revert to a local state back-end
 - Remove the back-end configuration
 - Rerun terraform init to copy the Terraform state to the local disk
 - Run terraform destroy to delete the S3 bucket and DynamoDB table

Remote Back-end Advantage

A single S3 bucket and DynamoDB table can be shared across all your Terraform code

- ◆ You'll probably only need to do it once per AWS account
- ◆ After the S3 bucket exists, in the rest of your Terraform code, you can specify the back-end configuration right from the start without any extra steps

Back-end Limitation

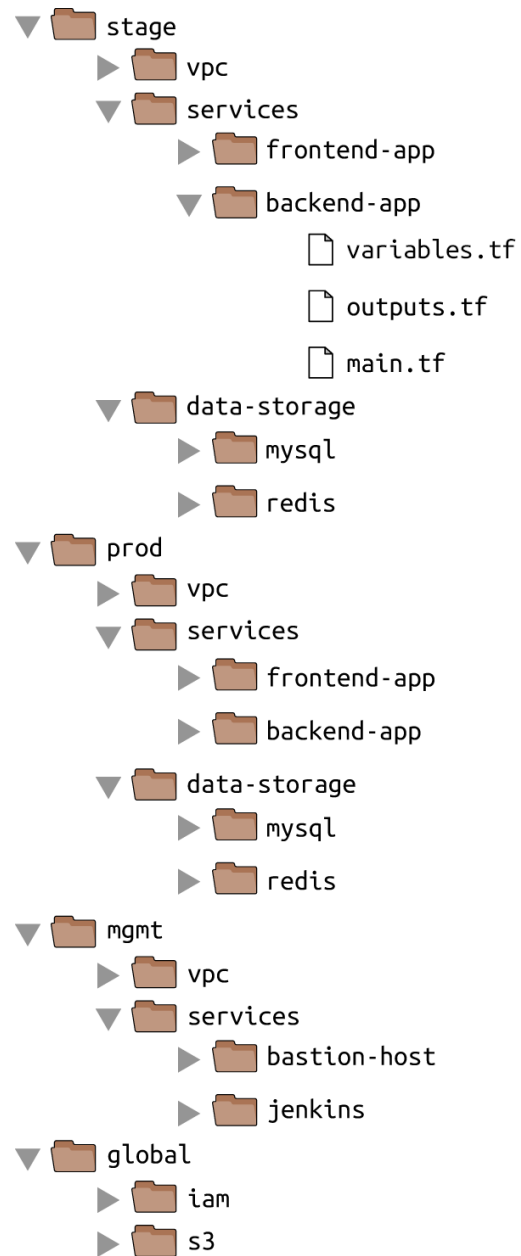
- ◆ Variables and references cannot be used in the back end block
- ◆ The following will **not** work

```
1 # This will NOT work. Variables aren't allowed in a backend configuration.
2 terraform {
3     backend "s3" {
4         bucket      = var.bucket
5         region      = var.region
6         dynamodb_table = var.dynamodb_table
7         key         = "example/terraform.tfstate"
8         encrypt      = true
9     }
10 }
```

File Isolation

- ◆ Most secure approach is to have a folder for each configuration
- ◆ Each deployment has its own back-end, local or remote.
 - This allows for isolation of all files
 - Allows for different access and authentication mechanisms
 - Eg. Different S3 buckets used as back-ends can have different policies

File Isolation Example



Workspaces Use Case

- ◆ If you already have a Terraform module deployed
 - you want to do some experiments with it
 - but you don't want your experiments to affect the state of the already deployed infrastructure
- ◆ Run `terraform workspace new` to deploy a new copy of the exact same infrastructure, but storing the state in a separate file

Workspace Specific Configurations

- ◆ 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`

```
1 resource "aws_instance" "example" {  
2     ami          = "ami-0c55b159cbfafa1f0"  
3     instance_type = terraform.workspace == "default" ? "t2.medium" : "t2.micro"  
4 }
```

- ◆ Workspaces allow a fast and easy way to quickly spin up and tear down different versions of your code

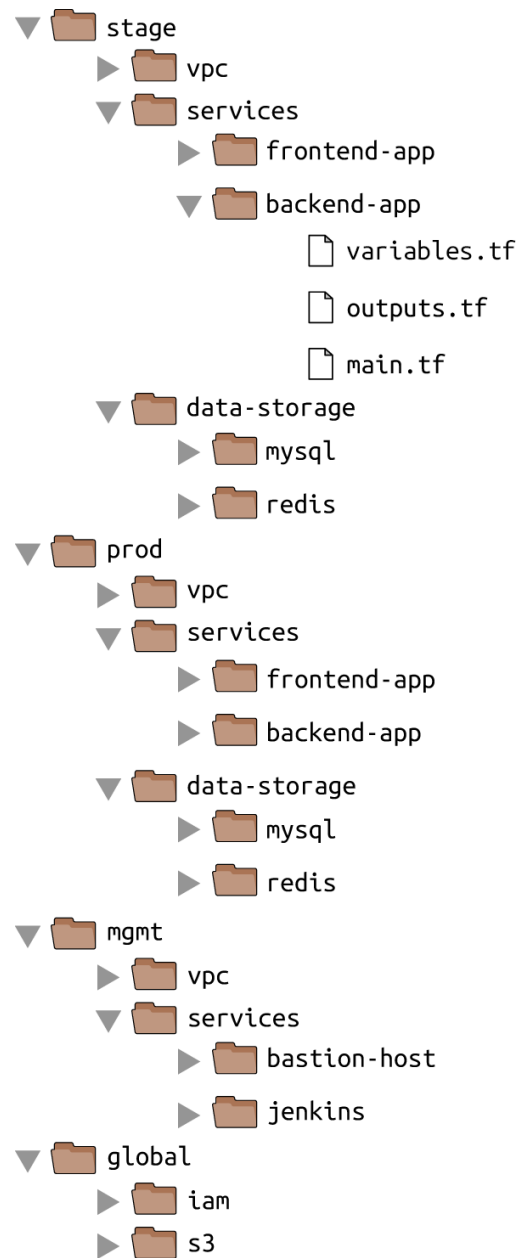
Workspace Drawbacks

- ◆ All workspace state files are stored in the same back-end
 - They share same authentication and access controls which means they are not good for isolating
- ◆ Workspaces are not visible in the code or on the terminal unless you run terraform workspace commands
 - A module 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
- ◆ Workspaces can be fairly error prone
 - The lack of visibility makes it easy to forget what workspace you're in and accidentally make changes in the wrong one

Isolation via File Layout

- ◆ To achieve full isolation between environments:
 - 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
 - All the configurations for the production environment can be in a folder called prod
- ◆ Configure a different back-end for each environment, using different authentication mechanisms and access controls
 - Each environment could live in a separate AWS account with a separate S3 bucket as a back-end

Typical Project File Layout



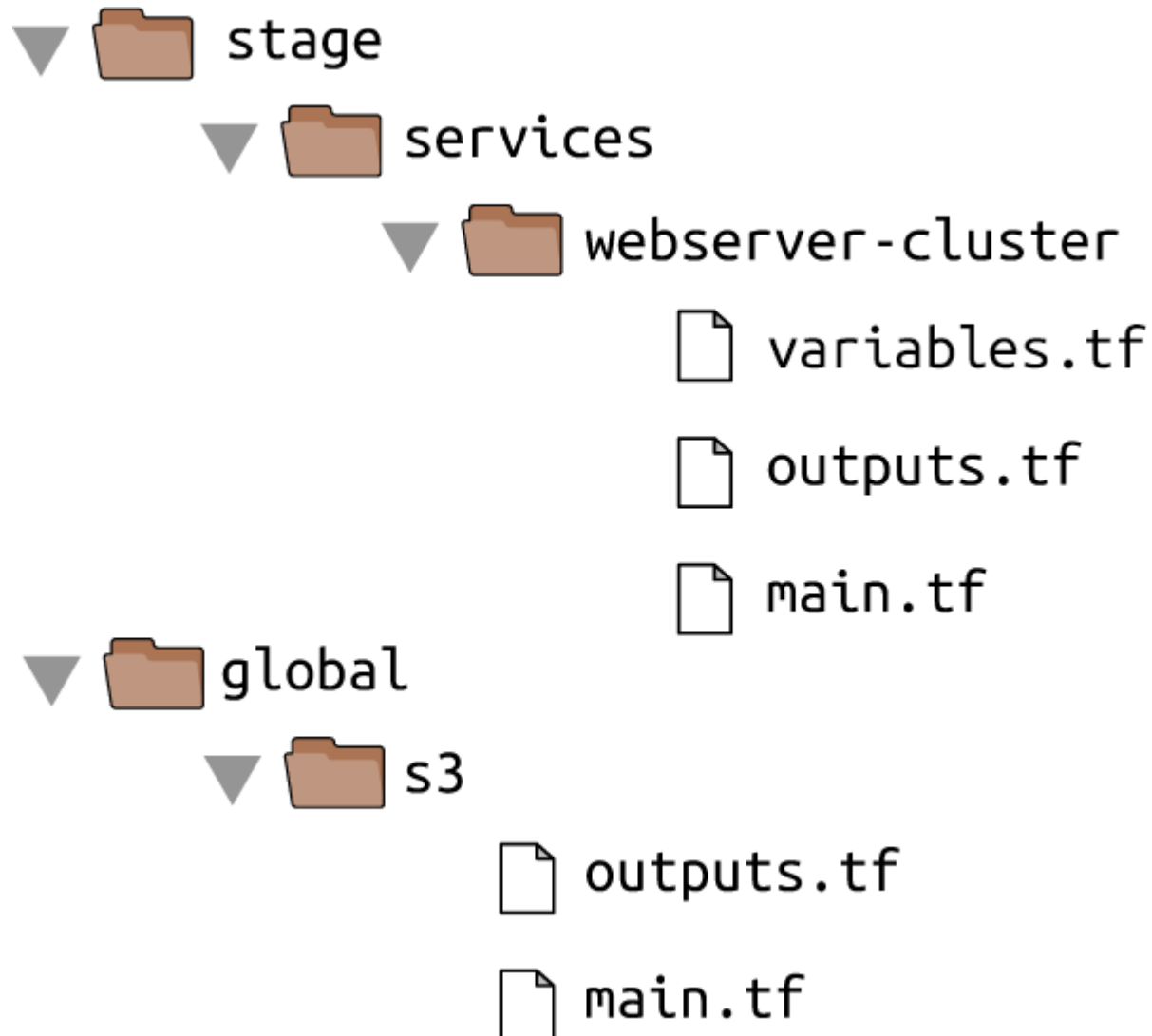
Isolation via File Layout

- ◆ At the top level, there are separate folders for each “environment”
 - **stage** : An environment for preproduction workloads (testing)
 - **prod** : An environment for production workloads (user facing apps)
 - **mgmt** : An environment for DevOps tooling (Jenkins etc.)
 - **global** : Resources that are used across all environments (S3, IAM)
- ◆ Within each environment, there are separate folders for each “component”:
 - **vpc** : Network topology for this environment
 - **services** : Apps or microservices to run in this environment - each app could have its own folder to isolate it
 - **data-storage** : The data stores to run in this environment, such as MySQL or Redis

Isolation via File Layout

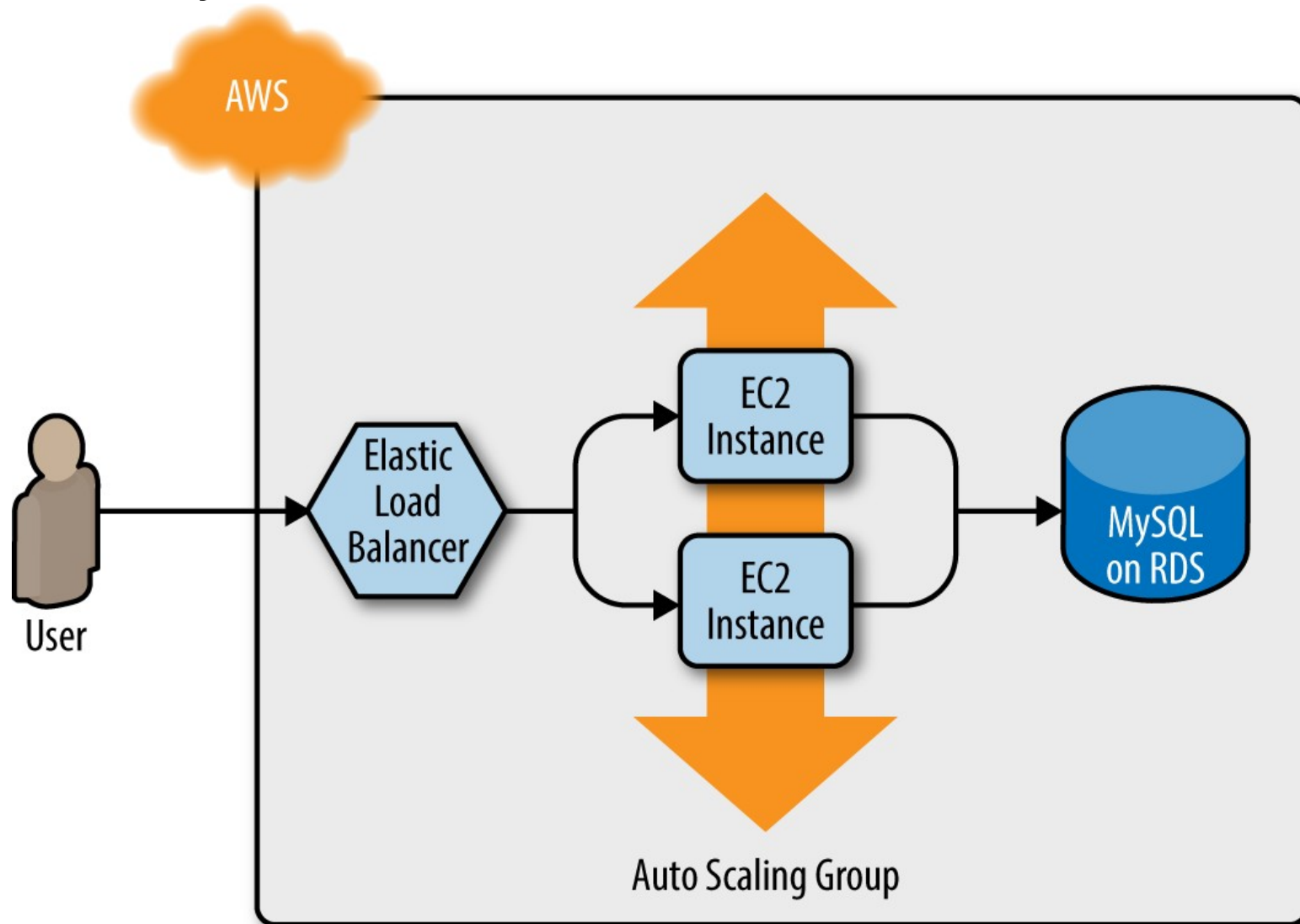
- ◆ Within each component are the actual Terraform configuration files with the following naming conventions:
 - **variables.tf** : Input variables
 - **outputs.tf** : Output variables
 - **main.tf** : The resources
- ◆ Terraform looks for files in the current directory with the .tf extension
 - Using a consistent, predictable naming convention makes code easier to browse
 - Then you always know where to look to find a variable, output, or resource. If individual Terraform files are

Rearranged Sample Code



The terraform_remote_state Data Source

- ◆ Assume that the web server cluster needs to communicate with a MySQL database

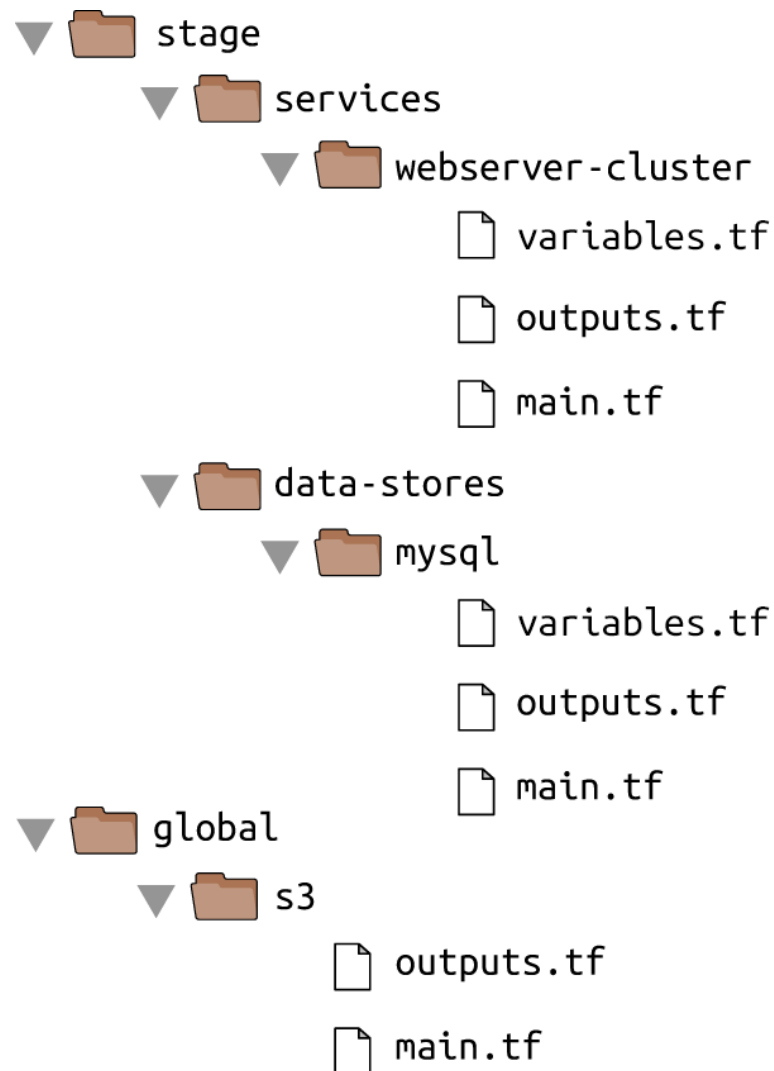


Deployment Consideration

- ◆ The MySQL database should probably be managed with a different set of configuration files as the web server cluster
 - Updates will probably be deployed to the web server cluster frequently
 - Want to avoid accidentally breaking the database when doing an update

Deployment Consideration

- ◆ Isolate the MySQL configurations in a data-stores folder



Keeping Secrets

- ◆ One of the parameters that you must pass to the `aws_db_instance` resource is the master password to use for the database
 - This should not be in the code in plain text
 - There are two other options
- ◆ Read the secret from a secret store - there are multiple secrets managers
 - AWS Secrets Manager and the `aws_secretsmanager_secret_version` data source (shown in the example code)
 - AWS Systems Manager Parameter Store and the `aws_ssm_parameter` data source
 - AWS Key Management Service (AWS KMS) and the `aws_kms_secrets` data source
 - Google Cloud KMS and the `google_kms_secret` data source
 - Azure Key Vault and the `azurerm_key_vault_secret` data source
 - HashiCorp Vault and the `vault_generic_secret` data source

Using AWS Secrets Manager

```
1 resource "aws_db_instance" "example" {
2     identifier_prefix    = "terraform-up-and-running"
3     engine               = "mysql"
4     allocated_storage    = 10
5     instance_class       = "db.t2.micro"
6     name                 = "example_database"
7     username             = "admin"
8
9     password =
10     data.aws_secretsmanager_secret_version.db_password.secret_string
11 }
12
13 data "aws_secretsmanager_secret_version" "db_password" {
14     secret_id = "mysql-master-password-stage"
15 }
```

Keeping Secrets II

- ◆ Other option is to manage them completely outside of Terraform
 - Then pass the secret into Terraform via an environment variable.
 - In the code below, there is no default since it's a secret
- ◆ A known weakness of Terraform:
 - The secret will be stored in the Terraform state file in plain text
 - The only solution is to lock down and encrypt the state files

```
1 variable "db_password" {
2     description = "The password for the database"
3     type        = string
4 }
5
6 export TF_VAR_db_password="(YOUR_DB_PASSWORD)"
7 $ terraform apply
```

Creating the Database

- ◆ Using the back-end to create the state repository:

```
1 terraform {  
2     backend "s3" {  
3         # Replace this with your bucket name!  
4         bucket      = "terraform-up-and-running-state"  
5         key          = "stage/data-stores/mysql/terraform.tfstate"  
6         region      = "us-east-2"  
7     }  
8     # Replace this with your DynamoDB table name!  
9     dynamodb_table = "terraform-up-and-running-locks"  
10    encrypt          = true  
11 }  
12 }
```

- ◆ Run the Terraform `init` and `apply` commands to create the database
 - Provide the database ports to the webserver cluster

```
1 output "address" {  
2     value      = aws_db_instance.example.address  
3     description = "Connect to the database at this endpoint"  
4 }  
5  
6 output "port" {  
7     value      = aws_db_instance.example.port  
8     description = "The port the database is listening on"  
9 }
```

Integrating the Database

- ◆ Running `init` again produces:

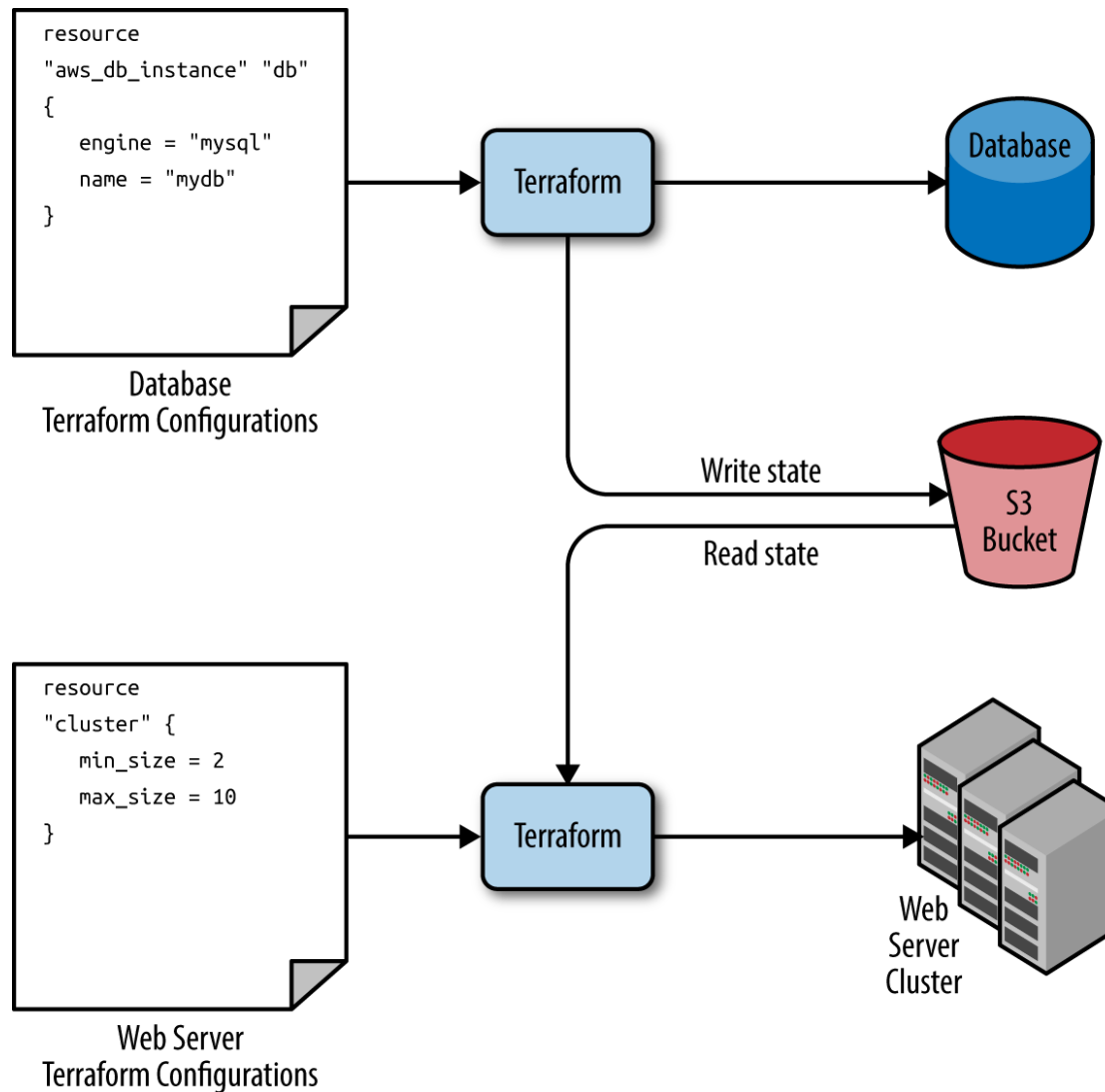
```
1 $ terraform apply
2
3 (...)
4
5 Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
6
7 Outputs:
8
9 address = tf-2016111123.cowu6mts6srx.us-east-2.rds.amazonaws.com
10 port = 3306
```

- ◆ These outputs are now stored in the Terraform state for the database
 - The web server cluster code can read the data from this state file by adding the `terraform_remote_state` data source in `stage/services/webserver-cluster/main.tf`:

```
1 data "terraform_remote_state" "db" {
2     backend = "s3"
3
4     config = {
5         bucket = "(YOUR_BUCKET_NAME)"
6         key     = "stage/data-stores/mysql/terraform.tfstate"
7         region  = "us-east-2"
8     }
```

Reading the DB State

- ◆ The webserver cluster reads the DB configuration from the state file in the S3 bucket



Reading the DB State

- ◆ All Terraform data sources, like the data returned by `terraform_remote_state`, is read-only
 - Nothing in the webserver code can modify the state
 - The database's state data can be read with no risk to the database
- ◆ The output variables are stored in the state file can be read using an attribute reference of the form:

```
1 data.terraform_remote_state.< NAME >.outputs.< ATTRIBUTE >
```

- ◆ This code updates the User Data of the web server cluster Instances to pull the database address and port out of the `terraform_remote_state` data source:

```
1 user_data = << EOF # space added for formatting
2 #!/bin/bash
3 echo "Hello, World" >> index.html
4 echo "${data.terraform_remote_state.db.outputs.address}" >> index.html
5 echo "${data.terraform_remote_state.db.outputs.port}" >> index.html
6 nohup busybox httpd -f -p ${var.server_port} &
7 EOF
```


Template Files

- ◆ Hard-coding the script in the previous slide is not effective
 - Instead we can read the contents from a file using the Terraform `file()` function which reads the contents of a file and returns it as a string
 - However, we have to dynamically insert Terraform data
- ◆ To provide this facility, Terraform has a `template_file` data source that has two arguments: A template, which is a string to render and a map of variables to use while rendering as well as one output attribute called `rendered`, which is the result of rendering template
 - For example, this can be added to the `webserver-cluster`

```
1 data "template_file" "user_data" {  
2     template = file("user-data.sh")  
3  
4     vars = {  
5         server_port = var.server_port  
6         db_address  = data.terraform_remote_state.db.outputs.address  
7         db_port      = data.terraform_remote_state.db.outputs.port  
8     }  
9 }
```

Template Files

- ◆ We also have to provide the "slots" in the template code for insertion of the variables
 - We use standard Terraform string interpolation, and we don't need the `var` prefix

```
1 #!/bin/bash
2
3 cat > index.html << EOF # space added for formatting
4 <h1>Hello, World</h1>
5 <p>DB address: ${db_address}</p>
6 <p>DB port: ${db_port}</p>
7 EOF
8
9 nohup busybox httpd -f -p ${server_port} &
```

```
1 resource "aws_launch_configuration" "example" {
2     image_id      = "ami-0c55b159cbfafa1f0"
3     instance_type = "t2.micro"
4     security_groups = [aws_security_group.instance.id]
5     user_data      = data.template_file.user_data.rendered
6
7     lifecycle {
8         create_before_destroy = true
9     }
10 }
```

Final Notes

- ◆ Correct isolation, locking and state must be a priority
 - Bugs in a program only break a part of an app
 - Bugs in infrastructure can have catastrophic effects and result in whole systems crashing and becoming unworkable
- ◆ Infrastructure has to be planned and incrementally tested
 - We never code infrastructure "on the fly"
- ◆ We never experiment with infrastructure in a production environment
 - Always work in a sandbox
 - With IaaS, this is easily done