Managing Terraform State

The Plan

- What is Terraform state?
- Shared storage for state files
- Limitations with Terraform's backends
- 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

```
resource "aws_instance" "example" {
 ami           = "ami-0c55b159cbfafe1f0"
instance_type = "t2.micro"
}
```

Example of "terraform.tfstate"

- After you run `terraform apply` you will see the output in t

```json
{
  "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-0c55b159cbfafe1f0",
            "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

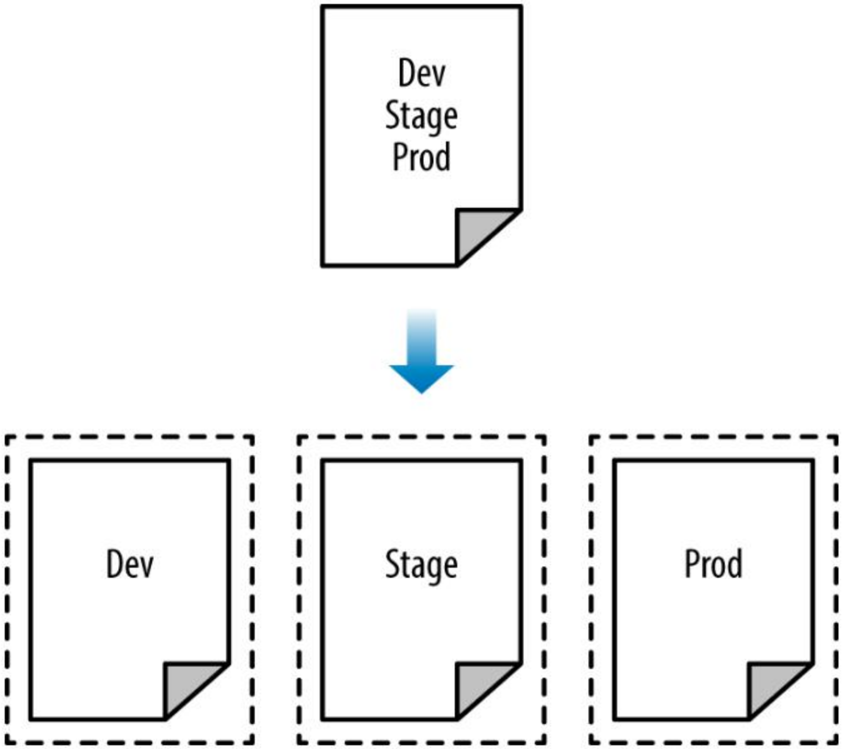
- State file tracks managed resources created by Terraform
- Other resources are ignored unless there is conflict
- Existing resources can be added to a Terraform state
- Resources can be removed from a Terraform state
- Multiple state files can be used
- Each state file manages a only its set of resources

The "state" Command

- 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 terrafrom 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
- For local state files, each new workspace's state file is in its own folder

The "workspace" Command

- `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 unmanaged

Remote Backends

- Each Terraform configuration specifies where the state files are kept
  - This is called the "backend"
  - The default is to use files in the local directory
  - This is what we have been using so far
- Terraform can also support "remote" backends
  - 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 Backend

- Using S3 as a backend 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

```
resource "aws_s3_bucket" "terraform_state" {
        bucket = "terraform-up-and-running-state"

        # Prevent accidental deletion of this S3 bucket
        lifecycle {
            prevent_destroy = true
        }

        # Enable versioning so we can see the full revision history of our
        # state files
        versioning {
            enabled = true
        }

        # Enable server-side encryption by default
        server_side_encryption_configuration {
            rule {
                apply_server_side_encryption_by_default {
                    sse_algorithm = "AES256"
                }
            }
        }
    }
```

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

```
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"
    }
}
```

Setting Up the Backend

- We have to tell Terraform the backend in now remote
  - We do this in the `terraform` directive

```
terraform {
  backend "< BACKEND_NAME >" {
   [CONFIG...]
      }
  }
```

```
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
        }
}
```

Moving State File Locations

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

Moving Backends 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 backend
  - Add a remote backend 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 backend
  - Remove the backend 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 Backend 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 backend configuration right from the start without any extra steps

Backend Limitation

- Variables and references cannot be used in the `backend` block
- The following 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
     }
}
```

File Isolation

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

File Isolation Example

- stage
  - vpc
  - services
    - frontend-app
    - backend-app
      - variables.tf
      - outputs.tf
      - main.tf
  - data-storage
    - mysql
    - redis
- prod
  - vpc
  - services
    - frontend-app
    - backend-app
  - data-storage
    - mysql
    - redis
- mgmt
  - vpc
  - services
    - bastion-host
    - jenkins
- global
  - iam
  - s3

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`

```
resource "aws_instance" "example" {
    ami           = "ami-0c55b159cbfafe1f0"
    instance_type = terraform.workspace == "default" ? "t2.medium" : "t2.micro"
}
```

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

Workspace Drawbacks

- All workspace state fiels are stored in the same backend
  - 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 backend 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 backend

Typical Project File Layout

- stage
  - vpc
  - services
    - frontend-app
    - backend-app
      - variables.tf
      - outputs.tf
      - main.tf
  - data-storage
    - mysql
    - redis
- prod
  - vpc
  - services
    - frontend-app
    - backend-app
  - data-storage
    - mysql
    - redis
- mgmt
  - vpc
  - services
    - bastion-host
    - jenkins
- global
  - iam
  - s3

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

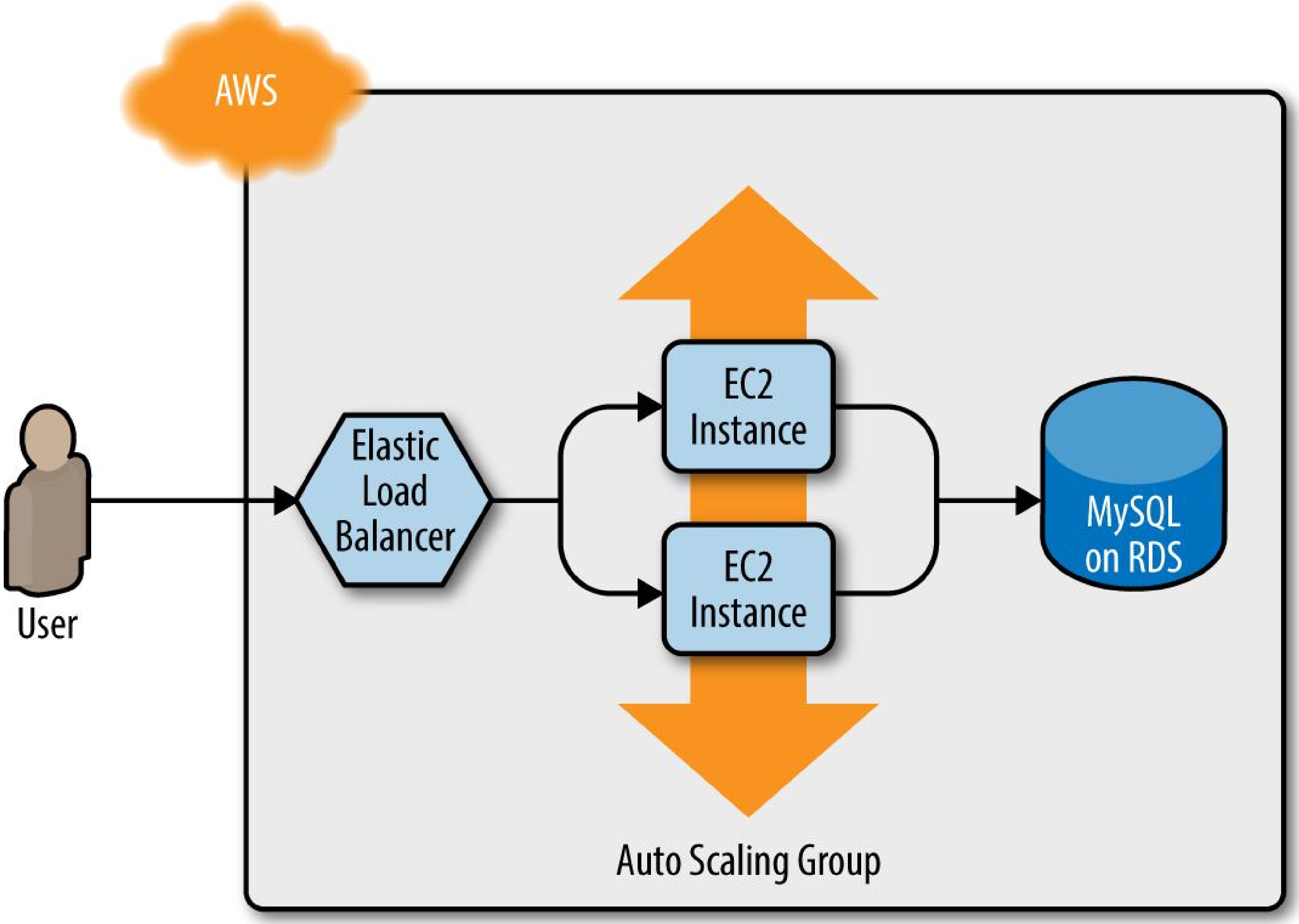
- Within each component are the actual Terraform configuration files with the following naming conventions:
  - **variables.tf** : Input variables
  - **outputs.tf** : Output variables
  - **main.tf** : The resources
- Terraform looks for files in the current directory with the .tf extension
  - Using 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

Rarranged Sample Code

- stage
  - services
    - webserver-cluster
      - variables.tf
      - outputs.tf
      - main.tf
- global
  - s3
    - outputs.tf
    - main.tf

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
  - Wnat to avoid accidentally breaking the database when doing an update

Deployment Consideration

- Isolate the MySql configurations in a data-stores folder

```
stage
    services
        webserver-cluster
            variables.tf
            outputs.tf
            main.tf
    data-stores
        mysql
            variables.tf
            outputs.tf
            main.tf
global
    s3
        outputs.tf
        main.tf
```

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

```
resource "aws_db_instance" "example" {
     identifier_prefix   = "terraform-up-and-running"
    engine                  = "mysql"
    allocated_storage   = 10
    instance_class          = "db.t2.micro"
    name                    = "example_database"
    username                = "admin"

    password =
     data.aws_secretsmanager_secret_version.db_password.secret_string
}

data "aws_secretsmanager_secret_version" "db_password" {
     secret_id = "mysql-master-password-stage"
}
```

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

```
variable "db_password" {
    description = "The password for the database"
     type         = string
}

export TF_VAR_db_password="(YOUR_DB_PASSWORD)"
$ terraform apply
```

- 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 ecrypt the state files

Creating the Database

- Using the backend to create the state repository:

```
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
    }
}
```

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

```
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"
}
```

Integrating the Database

- Running `init` again produces:

```
$ terraform apply

(...)

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

address = tf-2016111123.cowu6mts6srx.us-east-2.rds.amazonaws.com
port = 3306
```

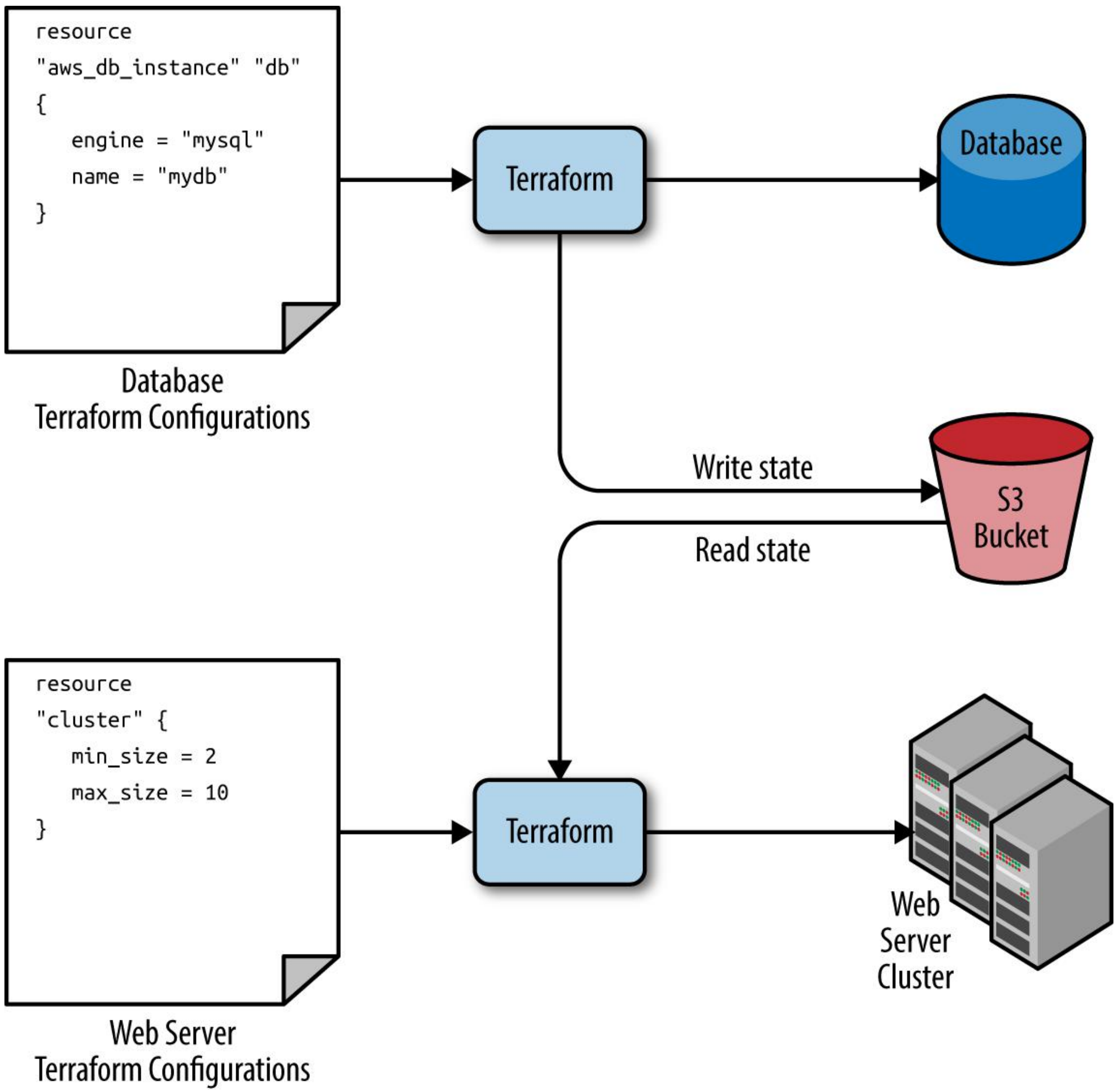
- These outputs are now stored in the Terraform state for the database
  - The web server cluster code can read the data from this state file by adding the terraform_remote_state data source in stage/services/webserver-cluster/main.tf:

```
data "terraform_remote_state" "db" {
    backend = "s3"

    config = {
    bucket = "(YOUR_BUCKET_NAME)"
    key    = "stage/data-stores/mysql/terraform.tfstate"
    region = "us-east-2"
    }
```

Reading the DB State

- The webservice 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:

```
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:

```
user_data = << EOF # space added for formatting
#!/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
```

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

```
data "template_file" "user_data" {
    template = file("user-data.sh")

    vars = {
        server_port = var.server_port
        db_address  = data.terraform_remote_state.db.outputs.address
        db_port     = data.terraform_remote_state.db.outputs.port
    }
}
```

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

```
#!/bin/bash

cat > index.html << EOF # space added for formatting
<h1>Hello, World</h1>
<p>DB address: ${db_address}</p>
<p>DB port: ${db_port}</p>
EOF

nohup busybox httpd -f -p ${server_port} &
```

```
resource "aws_launch_configuration" "example" {
    image_id        = "ami-0c55b159cbfafe1f0"
    instance_type   = "t2.micro"
    security_groups = [aws_security_group.instance.id]
    user_data       = data.template_file.user_data.rendered

     lifecycle {
        create_before_destroy = true
     }
}
```

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