Module 5: Layered Architecture

Terraform-Intermediate-Day2-Module-5

Learning Objectives

By the end of this module you will understand:

- How to split Terraform configurations into separate layers with independent state files
- How to use terraform_remote_state to share data between layers
- How outputs serve as the interface between layers
- How to manage deployment order and dependencies between layers

Why Do We Need Layered Architecture?

Imagine you're managing infrastructure for a growing web application. Your team has:

- Network infrastructure (VPCs, subnets, security groups)
- Application resources (EC2 instances, load balancers)
- Multiple teams working on different parts

Why Do We Need Layered Architecture?

The Problems:

1. **Blast radius:**
   - When all infrastructure is in one Terraform configuration, a mistake anywhere can affect everything.
   - A typo in your EC2 configuration could trigger changes to your networking during a plan.
2. **Team conflicts:**
   - The network team and application team are constantly stepping on each other.
   - Every pull request touches the same state file, causing merge conflicts and blocking deployments.
3. **Slow operations:**
   - With 200+ resources in one state file, terraform plan takes 10 minutes.
   - Every small change requires waiting for Terraform to check every resource.
4. **Tight coupling:**
   - You cannot deploy a new application version without also running infrastructure changes.
   - Everything is locked together.

Why Do We Need Layered Architecture?

The Solutions:

- Split your infrastructure into **layers** - separate Terraform configurations with their own state files.
- Each layer manages a specific concern and communicates with other layers through a well-defined interface.

What is Layered Architecture?

Layered architecture divides your infrastructure into separate Terraform configurations based on:

- Change frequency: Infrastructure that changes rarely (VPCs, subnets) vs. infrastructure that
- changes often (application servers)
- Team ownership: Resources managed by different teams
- Blast radius: Isolating critical resources from frequently changed ones

Common Layer Pattern

Common Layer Pattern

A typical two-layer setup:

Infrastructure Layer - Foundation resources that change infrequently:

- VPCs and subnets
- Security groups
- IAM roles and policies
- Shared resources

Application Layer - Resources that change with deployments:

- EC2 instances
- Auto Scaling groups
- Application-specific resources

Each layer has its own:

- **Directory with Terraform files**
- **State file (stored in S3 with different keys)**
- **Deployment lifecycle**

Separate State Files Per Layer

The key to layered architecture is that each layer has its own state file. You achieve this by using the same S3 bucket but different keys.

Infrastructure Layer Backend

```
# infrastructure/terraform.tf
terraform {
    backend "s3" {
    bucket = "company-terraform-state"
    key = "project/infrastructure/terraform.tfstate" # Infrastructure state
     ......
    }
}
```

Application Layer Backend

```
# application/terraform.tf
terraform {
    backend "s3" {
    bucket = "company-terraform-state"
    key = "project/application/terraform.tfstate" # Application state (different key!)
     .....
    }
}
```

Separate State Files Per Layer

Notice both use the same bucket (company-terraform-state) but different keys. This means:

- Each layer can be applied independently
- Changes to application don't lock infrastructure
- Different teams can work in parallel
- State files stay small and operations stay fast

The terraform_remote_state Data Source

The Format

```
data "terraform_remote_state" "layer_name" {
	backend = "s3"
		config = {
		bucket = "bucket-name"
		key = "path/to/terraform.tfstate"
		region = "us-west-1"
	}
}
```

How It Works

1. The infrastructure layer defines outputs for values other layers need
2. These outputs are stored in the infrastructure layer's state file
3. The application layer uses terraform_remote_state to read that state file
4. The application layer accesses outputs via data.terraform_remote_state.layer_name.outputs.output_name

Example: Reading Infrastructure Outputs

```
# application/data.tf
# Read the infrastructure layer's state
data "terraform_remote_state" "infrastructure" {
    backend = "s3"
        config = {
        bucket = "company-terraform-state"
        key = "project/infrastructure/terraform.tfstate"
        region = "us-west-1"
    }
}
```

Example: Reading Infrastructure Outputs

```
# application/main.tf
locals {
    # Pull values from infrastructure layer
    subnet_id =  data.terraform_remote_state.infrastructure.outputs.subnet_id
    security_group_id =
    data.terraform_remote_state.infrastructure.outputs.security_group_id
    environment =
    data.terraform_remote_state.infrastructure.outputs.environment
}
```

Example: Reading Infrastructure Outputs

```
# application/main.tf
resource "aws_instance" "web" {
      ami = data.aws_ami.amazon_linux.id
      instance_type = var.instance_type
      subnet_id = local.subnet_id # From infrastructure
      layer
      vpc_security_group_ids = [local.security_group_id] # From infrastructure
      layer
      tags = {
            Name = "web-${local.environment}"
      }
}
```

Outputs as the Interface Between Layers

Outputs define what information a layer exposes to other layers. Think of outputs as the public API of your layer - they define the contract between layers.

```
# infrastructure/outputs.tf
# Network resources for application layer
output "subnet_id" {
	description = "Subnet ID for application resources"
	value = aws_subnet.app.id
}
output "security_group_id" {
	description = "Security group ID for web servers"
	value = aws_security_group.web.id
}
# Context information
output "environment" {
	description = "Environment name (dev, staging)"
	value = var.environment
}
```

Outputs as the Interface Between Layers

Why This Matters

When we change the infrastructure layer, we must consider:

- **Adding outputs:** Safe - application layer can start using them
- **Changing output values:** Usually safe - underlying resource IDs typically don't change
- **Removing outputs:** Breaking change - application layer will fail if it references removed outputs

This is why outputs form a contract.
Treat them like a public API - be careful about breaking changes.

Deployment Order

With layered architecture, deployment order matters:

Deploy Order (Create)

- Infrastructure layer first - Creates foundation resources
- Application layer second - Depends on infrastructure outputs

Example Flow:

```
# Initial deployment
    cd infrastructure/
        terraform init
        terraform apply
    cd ../application/
        terraform init
        terraform apply
```

If we try to deploy the application layer before infrastructure, ***terraform_remote_state*** will fail because the infrastructure state file doesn't exist yet.

Deployment Order .....

Destroy Order (Delete)

- Application layer first - Remove resources that depend on infrastructure
- Infrastructure layer second - Now safe to remove foundation

Example Flow:

```
# Cleanup (reverse order!)
    cd application/
        terraform destroy
    cd ../infrastructure/
        terraform destroy
```

If we destroy infrastructure first,
AWS may block deletion because resources still depend on them (for example, you cannot delete a security group attached to running instances).

Complete Example: Two-Layer Setup

Directory Structure

```
project/
    infrastructure/
        terraform.tf
        main.tf
        variables.tf
        outputs.tf
    application/
        terraform.tf
        main.tf
        data.tf
        variables.tf
        outputs.tf
```

DEMO

Benefits of Layered Architecture

- **Reduced blast radius** - Changes to application cannot accidentally affect networking. A mistake in one layer doesn't propagate to others.
- **Team independence** - The network team manages infrastructure layer. The application team manages application layer. No more merge conflicts.
- **Faster operations** - Each layer has fewer resources. terraform plan completes in seconds instead of minutes.
- **Independent deployment** - Deploy a new application version without touching infrastructure. Update infrastructure without redeploying applications.
- **Clear ownership** - Each layer has defined boundaries and interfaces. Teams know exactly what they’re responsible for.

Key Takeaways

- **Layered architecture** splits infrastructure into separate Terraform configurations with independent state files, reducing blast radius and enabling team independence.
- **Same bucket, different keys** - Layers share an S3 bucket but use different keys, giving each layer its own state file.
- **terraform_remote_state** reads another layer's state file and makes its outputs available, enabling cross-layer communication.
- **Outputs are the interface** - Design outputs carefully as they form the contract between layers. Adding outputs is safe; removing them is a breaking change.
- **Deployment order matters** - Deploy infrastructure before application; destroy application before infrastructure.
- **Start simple** - Two layers (infrastructure and application) handles most use cases. Add more layers only when you have clear separation of concerns.

Knowledge Check

Question 1

- You have an infrastructure layer and an application layer.
- The application layer uses terraform_remote_state to read infrastructure outputs.
- What happens if you try to run terraform plan in the application layer before deploying the infrastructure layer?

A) Terraform uses default values for the missing outputs
B) Terraform creates placeholder resources
C) Terraform fails because it cannot read the infrastructure state file
D) Terraform skips resources that depend on infrastructure outputs

Solution:
C - Terraform fails because the infrastructure state file doesn't exist yet.
The terraform_remote_state data source tries to read the state file at the specified location, and if it doesn't exist, Terraform cannot proceed. This is why deployment order matters: infrastructure must be deployed before application.

Knowledge Check

Question 2

- Your infrastructure layer outputs subnet_id. You want to rename it to app_subnet_id. What is the safest approach?

A. Rename the output and deploy both layers
B. Add the new output name, update the application layer to use it, then remove the old output
C. Delete the output and let Terraform handle the migration
D. Outputs cannot be renamed once created

Solution:
B - The safest approach is to add the new output name alongside the old one, update all consumers (the application layer) to use the new name, then remove the old output. This avoids any moment where the application layer references a non-existent output. Directly renaming would break the application layer until it's updated.

Knowledge Check

Question 3

- What is the correct order to destroy resources in a two-layer architecture?

A. Infrastructure layer first, then application layer
B. Application layer first, then infrastructure layer
C. Both layers can be destroyed in any order
D. Use terraform destroy -all to handle both layers

Solution:
B - Destroy application layer first, then infrastructure. The application layer depends on infrastructure resources (like subnets and security groups). If you destroy infrastructure first, AWS may block deletion because resources still reference them. Destroying in reverse dependency order ensures clean removal.