Module 6: Import and State Surgery

Learning Objectives

By the end of this module you will understand:

- How to use `import` blocks to bring existing resources under Terraform management
- How to use `moved` blocks to rename and reorganize resources in state
- How to use `removed` blocks to remove resources from Terraform management
- When to use each state operation and the trade-offs involved

---

Why Do We Need Import and State Surgery?

Imagine you're joining a team that has been running infrastructure in AWS for years. Some resources were created:

- Manually in the AWS Console by developers
- By scripts that have since been lost
- By a previous infrastructure tool that's been retired

Your organization wants to adopt Terraform, but recreating everything from scratch is risky and time-consuming. You need a way to bring existing resources under Terraform management without destroying and recreating them.

The Problems:

1. **Existing infrastructure**: You have an S3 bucket storing critical logs and an IAM role used by applications. These were created manually. If you write Terraform code and run `terraform apply`, Terraform will try to create new resources - it doesn't know about the existing ones.

2. **Resource renaming**: Over time, you realize your resource names don't follow a consistent pattern. You have `aws_s3_bucket.legacy_logs` but want to rename it to `aws_s3_bucket.logs`. If you just change the name in your configuration, Terraform thinks you want to destroy the old resource and create a new one.

3. **Transferring ownership**: A resource that was managed by your team is being handed off to another team who manages their own Terraform configuration. You need to remove it from your state without destroying the actual resource.

The Solutions:

- **Import blocks** let you bring existing resources into Terraform state without recreating them
- **Moved blocks** let you rename and reorganize resources in state without affecting the actual infrastructure
- **Removed blocks** let you remove resources from management without destroying them

---

Import Blocks: Bringing Existing Resources Under Management

The `import` block is the modern, declarative approach for bringing existing AWS resources under Terraform management. It was introduced in Terraform 1.5 and integrates with the standard plan/apply workflow.

The Format

```
import {
  to = resource_type.resource_name
  id = "the-resource-identifier"
}
```

Breaking this down:

- `to` specifies the Terraform resource address where this resource will be imported
- `id` is the identifier AWS uses for this resource (bucket name for S3, role name for IAM roles, instance ID for EC2, etc.)

Why Import Blocks Instead of terraform import Command?

Before import blocks, you had to use the `terraform import` command. Import blocks are better because:

- **Preview changes**: You can see what will be imported in `terraform plan` before it happens
- **CI/CD friendly**: Import blocks are declarative and can be reviewed in pull requests
- **Version controlled**: The import intention is captured in code, not a one-time command
- **Batch imports**: You can import multiple resources in a single apply

Example: Importing an S3 Bucket

Let's say you have an S3 bucket named `dev-logs-bucket` that was created manually. Here's how to import it:

Step 1: Write the resource configuration

First, you need a resource block that matches what the existing resource looks like:

```
# variables.tf
variable "project" {
  description = "Project name"
  type        = string
  default     = "dev"
}

locals {
  name_prefix = "${var.project}-logs"
}

# main.tf
resource "aws_s3_bucket" "logs" {
```

```
  bucket = "${local.name_prefix}-bucket"

  tags = {
    Name        = "${local.name_prefix}-bucket"
    Environment = "dev"
    Purpose     = "logs"
  }
}
```

Step 2: Create the import block

Create a separate file for imports (often named `imports.tf`):

```
# imports.tf
import {
  to = aws_s3_bucket.logs
  id = "dev-logs-bucket"
}
```

Step 3: Preview the import

Run `terraform plan` to see what will happen:

```
terraform plan
```

The plan output will show something like:

```
aws_s3_bucket.logs: Preparing import... [id=dev-logs-bucket]
aws_s3_bucket.logs: Refreshing state... [id=dev-logs-bucket]

Terraform will perform the following actions:

  # aws_s3_bucket.logs will be imported
    resource "aws_s3_bucket" "logs" {
        arn                         = "arn:aws:s3:::dev-logs-bucket"
        bucket                      = "dev-logs-bucket"
        ...
    }

Plan: 1 to import, 0 to add, 0 to change, 0 to destroy.
```

Step 4: Apply the import

```
terraform apply
```

After the apply completes, the S3 bucket is now in Terraform state and managed by Terraform.

Step 5: Clean up the import block

After a successful import, remove the `imports.tf` file. The import block is only needed once - to bring the resource into state. Keeping it around doesn't cause errors, but it's unnecessary clutter.

```
rm imports.tf
```

Handling Configuration Drift

After importing, run `terraform plan` again. If you see changes, your configuration doesn't exactly match the actual resource. This is called **configuration drift**.

For example, if the plan shows:

```
# aws_s3_bucket.logs will be updated in-place
  ~ resource "aws_s3_bucket" "logs" {
      ~ tags = {
          + "Owner" = "dev-team"
        }
    }
```

This means your configuration has an `Owner` tag that the actual bucket doesn't have. You can either:

1. Update your configuration to match reality (remove the tag from your code)
2. Let Terraform apply the change (add the tag to the actual bucket)

Moved Blocks: Reorganizing State

The `moved` block lets you rename resources or move them between modules without destroying and recreating them. It tells Terraform: "The resource that used to be at address A is now at address B."

The Format

```
moved {
  from = resource_type.old_name
  to   = resource_type.new_name
}
```

Breaking this down:

- `from` is the old resource address (where Terraform currently tracks this resource in state)
- `to` is the new resource address (where you want Terraform to track it going forward)

Why Use Moved Blocks?

Without moved blocks, if you rename a resource from `aws_s3_bucket.legacy_logs` to `aws_s3_bucket.logs`:

1. Terraform sees `legacy_logs` is gone - it plans to destroy it
2. Terraform sees `logs` is new - it plans to create it
3. You lose your S3 bucket and create a new empty one

With moved blocks, Terraform understands the rename and updates state without touching the actual resource.

Example: Renaming a Resource

You have a resource with a legacy name and want to clean it up:

Current configuration:

```
resource "aws_s3_bucket" "legacy_logs" {
  bucket = "dev-logs-bucket"

  tags = {
    Name = "dev-logs-bucket"
  }
}
```

Step 1: Create the new resource block with the better name

```
# organized_resources.tf
resource "aws_s3_bucket" "logs" {
  bucket = "dev-logs-bucket"

  tags = {
    Name = "dev-logs-bucket"
  }
}
```

Step 2: Create the moved block

```
# moves.tf
moved {
  from = aws_s3_bucket.legacy_logs
  to   = aws_s3_bucket.logs
}
```

Step 3: Remove the old resource block

Delete the old `aws_s3_bucket.legacy_logs` resource from your configuration.

Step 4: Preview and apply

```
terraform plan
```

The plan will show:

```
Terraform will perform the following actions:

  # aws_s3_bucket.legacy_logs has moved to aws_s3_bucket.logs
    resource "aws_s3_bucket" "logs" {
        id     = "dev-logs-bucket"
        bucket = "dev-logs-bucket"
        ...
    }

Plan: 0 to add, 0 to change, 0 to destroy.
```

Notice: **0 to destroy**. The bucket isn't being destroyed, just renamed in state.

```
terraform apply
```

Step 5: Optionally clean up moved blocks

Unlike import blocks, moved blocks can be left in your codebase. They serve as documentation of refactoring history and don't cause issues. However, you can remove them after the move is complete if you prefer a cleaner codebase.

Moving Resources Between Modules

Moved blocks also work when reorganizing resources into modules:

```
moved {
  from = aws_s3_bucket.logs
  to   = module.storage.aws_s3_bucket.logs
}
```

This is useful when you're refactoring a flat configuration into a modular structure.

Removed Blocks: Removing Resources from Management

Sometimes you need to stop managing a resource with Terraform without destroying it. The `removed` block (introduced in Terraform 1.7) is the modern, declarative approach for this. It integrates with the standard plan/apply workflow, just like `import` and `moved` blocks.

The Format

```
removed {
  from = resource_type.resource_name

  lifecycle {
    destroy = false  # Keep the resource in AWS
  }
}
```

Breaking this down:

- `from` specifies the resource address to remove from state
- `lifecycle { destroy = false }` tells Terraform to keep the actual resource in AWS (without this, Terraform would destroy the resource)

Why Use Removed Blocks?

Before removed blocks, you had to use the `terraform state rm` command. Removed blocks are better because:

- **Preview changes**: You can see what will be removed in `terraform plan` before it happens
- **CI/CD friendly**: Removed blocks are declarative and can be reviewed in pull requests
- **Version controlled**: The removal intention is captured in code
- **Consistent workflow**: Same plan/apply workflow as import and moved blocks

When to Use It

- **Transferring ownership**: Another team will manage this resource in their Terraform configuration
- **Moving to manual management**: The resource needs to be managed outside Terraform
- **The resource should continue to exist**: You want to stop tracking it, not delete it

Example: Removing an IAM Role from Management

Let's say another team will take over managing an IAM role:

Step 1: Create the removed block

Create a file named `removed.tf`:

```
# Remove IAM role from Terraform management
# The role will continue to exist in AWS
removed {
  from = aws_iam_role.application
```

```
  lifecycle {
    destroy = false  # Keep the resource in AWS
  }
}
```

Step 2: Remove the resource block from configuration

Delete the `aws_iam_role.application` resource block from your `.tf` files. You cannot have both the resource block and the removed block at the same time.

Step 3: Preview the removal

```
terraform plan
```

The plan will show the resource being removed from state:

```
Terraform will perform the following actions:

  # aws_iam_role.application will no longer be managed by Terraform, but
will not be destroyed
  # (destroy = false is set in the configuration)
  . resource "aws_iam_role" "application" {
        id   = "dev-app-role"
        name = "dev-app-role"
        ...
    }

Plan: 0 to add, 0 to change, 0 to destroy.
```

Step 4: Apply

```
terraform apply
```

Step 5: Clean up the removed block

After successful apply, delete the `removed.tf` file:

```
rm removed.tf
```

Important: The Resource Still Exists

After using a `removed` block with `destroy = false`, the resource continues to exist in AWS. Terraform simply stops tracking it. If you later run `terraform destroy`, this resource will **not** be destroyed because Terraform doesn't know about it anymore.

Legacy Alternative: terraform state rm

Before Terraform 1.7, you had to use the `terraform state rm` command:

```
terraform state rm aws_iam_role.application
```

This command still works but doesn't integrate with the plan/apply workflow - it immediately modifies state without a preview step. The `removed` block is the recommended approach for Terraform 1.7 and later.

Best Practices for State Surgery

Always Backup State First

Before any state operation, create a backup:

```
terraform state pull > backup.tfstate
```

If something goes wrong, you can restore:

```
terraform state push backup.tfstate
```

Test in Non-Production First

Practice import, move, and state removal operations in a dev or staging environment before touching production state.

Document State Operations

When performing state surgery, document:

- What operation you performed
- Why it was needed
- When it was done
- Who performed it

For teams still using Terraform versions before 1.7, document any `terraform state rm` operations since that command doesn't get version controlled.

Use Version Control

- Import blocks and moved blocks should be committed to version control
- Review them in pull requests before applying
- This creates an audit trail of state changes

Clean Up After Successful Operations

- Remove import blocks after successful import
- Moved blocks can be left for documentation or removed for cleanliness

---

Key Takeaways

1. **Import blocks** are the modern, declarative way to bring existing resources under Terraform management. They integrate with plan/apply and can be reviewed in pull requests.
2. **Moved blocks** let you rename and reorganize resources in state without destroying infrastructure. They tell Terraform that a resource has a new address.
3. **Removed blocks** let you remove resources from Terraform management without destroying them. The resource continues to exist in AWS but is no longer tracked by Terraform. Use `lifecycle { destroy = false }` to preserve the resource.
4. **Configuration drift** happens when your Terraform code doesn't match reality. After importing, always run `terraform plan` to check for drift.
5. **Always backup state** before performing state surgery operations with `terraform state pull > backup.tfstate`.
6. **All three block types are version controlled**: Import, moved, and removed blocks all integrate with the plan/apply workflow and can be reviewed in pull requests.

Knowledge Check

Question 1

What is the primary advantage of using import blocks over the `terraform import` command?

A) Import blocks are faster to execute B) Import blocks can import multiple resources simultaneously C) Import blocks integrate with plan/apply workflow and can be reviewed in pull requests D) Import blocks don't require you to write resource configuration first

---

Question 2

You use a `removed` block with `lifecycle { destroy = false }` to remove an S3 bucket from Terraform management. What happens to the bucket in AWS?

A) The bucket is deleted immediately B) The bucket is marked for deletion on the next apply C) The bucket continues to exist but is no longer managed by Terraform D) The bucket is moved to a different state file

---

Question 3

You want to rename `aws_instance.old_web` to `aws_instance.web` without destroying the EC2 instance. Which approach should you use?

A) Delete the old resource block and create a new one with the new name B) Use `terraform import` to import the instance under the new name C) Use a `moved` block to tell Terraform about the rename D) Use `terraform state rm` to remove the old name and re-import

---

Answers

Question 1: C - Import blocks integrate with the plan/apply workflow, allowing you to preview imports before they happen and review them in pull requests. The legacy `terraform import` command makes immediate changes to state without a preview step.

Question 2: C - When you use a `removed` block with `lifecycle { destroy = false }`, Terraform removes the resource from its state file but leaves the actual AWS resource unchanged. The bucket continues to exist in AWS but is no longer tracked or managed by Terraform.

Question 3: C - The `moved` block is specifically designed for renaming and reorganizing resources in state without affecting the actual infrastructure. It tells Terraform that the resource at the old address should now be tracked at the new address.

Dictionary of Terms

configuration drift - When the actual state of infrastructure differs from what's defined in Terraform configuration. Can occur after importing resources or when changes are made outside of Terraform.

import block - A declarative block in Terraform configuration that specifies an existing resource to bring under Terraform management. Integrates with the plan/apply workflow.

moved block - A declarative block that tells Terraform a resource has been renamed or moved to a new address. Allows reorganizing state without destroying and recreating resources.

removed block - A declarative block (introduced in Terraform 1.7) that removes a resource from Terraform state. Use `lifecycle { destroy = false }` to keep the actual resource in the cloud.

resource address - The full path to a resource in Terraform configuration, such as `aws_s3_bucket.logs` or `module.storage.aws_s3_bucket.logs`.

state surgery - The practice of directly manipulating Terraform state to import, move, or remove resources without affecting the actual infrastructure.

terraform state rm - A legacy Terraform command that removes a resource from state without destroying it. The `removed` block is the recommended approach for Terraform 1.7 and later.

Documentation Links

- Import Block
- Moved Block
- Removed Block
- State Command