Module 4: Lifecycle Configuration

Learning Objectives

By the end of this module you will understand:

- How to use `create_before_destroy` for zero-downtime updates
- How to use `prevent_destroy` to protect critical resources from accidental deletion
- How to use `ignore_changes` to handle attributes modified by external tools
- When to use each lifecycle option and how to combine them

Why Do We Need Lifecycle Configuration?

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

- A security group attached to running EC2 instances
- An S3 bucket storing critical business data
- EC2 instances where AWS adds tags automatically for cost tracking

The Problems:

1. **Replacement order issues**: When Terraform needs to replace a security group (for example, when the name changes), it normally tries to destroy the old one first. But AWS blocks deletion of security groups that are still attached to instances, causing Terraform to fail with dependency errors. You need a way to tell Terraform: "create the new one first, update the instances, then remove the old one."

2. **Accidental deletion**: A team member runs `terraform destroy` on what they think is a test environment. The RDS database - containing customer data - is deleted. There's no per-resource confirmation, and the data is gone. (Note: empty S3 buckets and RDS instances can be deleted immediately; S3 buckets with objects are protected by AWS unless `force_destroy = true` is set.)

3. **Fighting with external tools**: Your security scanning tools add tags like `LastScannedBy` or `ComplianceStatus` to instances. Every time you run Terraform, it tries to remove those tags - an endless battle you don't need.

The Solution: The `lifecycle` block gives you control over how Terraform creates, updates, and destroys resources. You can tell Terraform to create replacements before destroying originals, prevent certain resources from being deleted, or ignore changes to specific attributes.

The lifecycle Block

The `lifecycle` block is a **meta-argument** - a special configuration block that works with any resource type. You place it inside a resource block.

The Format

```
resource "aws_instance" "web" {
  ami           = "ami-12345678"
  instance_type = "t3.micro"

  # Lifecycle settings go inside the resource block
  lifecycle {
    # One or more lifecycle settings
    create_before_destroy = true
  }
}
```

The `lifecycle` block accepts these settings:

- `create_before_destroy` - Create the replacement before destroying the original
- `prevent_destroy` - Block any attempt to destroy this resource
- `ignore_changes` - Ignore changes to specific attributes

You can combine multiple settings in one `lifecycle` block.

---

create_before_destroy: Zero-Downtime Updates

By default, when Terraform needs to replace a resource (because a change requires recreation), it follows this order:

1. Destroy the existing resource
2. Create the new resource

This causes a gap where the resource doesn't exist - which can mean downtime.

The Format

```
lifecycle {
  create_before_destroy = true
}
```

How It Works

With `create_before_destroy = true`, Terraform reverses the order:

1. Create the new resource first
2. Update any references to point to the new resource
3. Destroy the old resource

When to Use It

Security groups attached to running instances are the classic example:

```
resource "aws_security_group" "web" {
  name        = "web-server-sg-${random_id.suffix.hex}"
  description = "Security group for web servers"
  vpc_id      = data.aws_vpc.default.id

  # Without this, updating the security group could disconnect your
instances
  lifecycle {
    create_before_destroy = true
  }
}
```

Why does this matter? Security groups can't be modified in place for certain changes (like changing the name). Without `create_before_destroy`:

1. Terraform destroys the old security group
2. Your EC2 instances lose their security group
3. Connections fail during this gap
4. Terraform creates the new security group
5. Terraform attaches it to the instances

With `create_before_destroy`:

1. Terraform creates the new security group
2. Terraform updates instances to use the new security group
3. Terraform destroys the old security group
4. No gap - your instances always have a security group

Other good use cases:

- Security groups attached to running instances
- IAM roles and policies attached to running services

Important Consideration

When using `create_before_destroy`, you may need unique naming. If two resources can't have the same name simultaneously, add a random suffix:

```
resource "random_id" "suffix" {
  byte_length = 4
}

resource "aws_security_group" "web" {
  # Random suffix ensures unique name even when old and new exist
simultaneously
  name = "web-sg-${random_id.suffix.hex}"

  lifecycle {
    create_before_destroy = true
  }
}
```

---

prevent_destroy: Protecting Critical Resources

Some resources should never be accidentally deleted - databases, S3 buckets with important data, or any resource where data loss would be catastrophic.

The Format

```
lifecycle {
  prevent_destroy = true
}
```

How It Works

When `prevent_destroy = true`, Terraform refuses to destroy the resource. If a plan includes destroying this resource - whether from `terraform destroy` or a configuration change that requires replacement - Terraform stops with an error.

Example: Protecting an S3 Bucket

```
resource "aws_s3_bucket" "critical_data" {
  bucket = "company-critical-data-${random_id.suffix.hex}"

  tags = {
    Name     = "Critical Business Data"
    Critical = "true"
  }

  # Prevent accidental deletion
  lifecycle {
    prevent_destroy = true
  }
}
```

What Happens When You Try to Destroy

If someone runs `terraform destroy`, they'll see:

```
Error: Instance cannot be destroyed

  on main.tf line 12:
  12: resource "aws_s3_bucket" "critical_data" {
```

```
Resource aws_s3_bucket.critical_data has lifecycle.prevent_destroy set,
but the plan calls for this resource to be destroyed.
```

Terraform stops completely - it won't destroy other resources either. This gives you a chance to review what's happening.

How to Destroy When You Really Need To

When you genuinely need to remove a protected resource (like decommissioning a system), you have two options:

Option 1: Remove the setting temporarily

```
lifecycle {
  # prevent_destroy = true  # Comment out this line
}
```

Then run `terraform apply` to update Terraform's understanding, then `terraform destroy`.

Option 2: Set to false

```
lifecycle {
  prevent_destroy = false
}
```

Apply, then destroy.

When to Use It

- S3 buckets containing important data
- RDS databases
- DynamoDB tables
- Any resource where accidental deletion would cause significant problems

Note: `prevent_destroy` only prevents Terraform from destroying the resource. It doesn't prevent someone from deleting it directly through the AWS Console or CLI.

---

ignore_changes: Handling External Modifications

Sometimes attributes of your resources get modified outside of Terraform - by AWS itself, by other automation tools, or by manual changes. By default, Terraform tries to "fix" these differences on the next apply, which can cause unnecessary changes or conflicts.

The Format

```
lifecycle {
  ignore_changes = [
    attribute_name,
    another_attribute,
  ]
}
```

How It Works

Terraform tracks the state of your resources. When you run `terraform plan`, it compares:

- What the configuration says the resource should look like
- What the resource actually looks like in AWS

Normally, any differences become changes in the plan. With `ignore_changes`, Terraform skips the comparison for listed attributes - changes to those attributes won't appear in plans.

Example: Ignoring External Tag Changes

AWS and other tools often add tags to resources. For example, Auto Scaling adds group membership tags, cost allocation adds tracking tags:

```
resource "aws_instance" "web" {
  ami           = data.aws_ami.amazon_linux.id
  instance_type = "t3.micro"

  tags = {
    Name        = "web-server"
    Environment = "dev"
  }

  lifecycle {
    # Ignore tags that security/compliance tools might add
    ignore_changes = [
      tags["LastScannedBy"],
      tags["ComplianceStatus"],
    ]
  }
}
```

This says: "Don't try to remove the `LastScannedBy` or `ComplianceStatus` tags if they appear on the instance."

Specifying What to Ignore

You can ignore:

Specific attributes:

```
ignore_changes = [
  instance_type,
  ami,
]
```

Nested attributes using dot notation:

```
ignore_changes = [
  tags["ExternalTag"],
  root_block_device[0].volume_size,
]
```

All attributes (use sparingly):

```
ignore_changes = all
```

Using `all` means Terraform will never update the resource after creation - only create or destroy it. This is rarely what you want.

When to Use It

- Tags added by AWS services (Auto Scaling, cost allocation)
- Attributes modified by external automation
- Settings that drift intentionally and shouldn't be "corrected"
- Attributes that are set once at creation and should never change

Caution

Don't use `ignore_changes` to hide configuration problems. If an attribute keeps changing unexpectedly, investigate why rather than just ignoring it.

Combining Lifecycle Settings

You can use multiple lifecycle settings together:

```
resource "aws_instance" "web" {
  ami           = data.aws_ami.amazon_linux.id
  instance_type = "t3.micro"

  vpc_security_group_ids = [aws_security_group.web.id]

  tags = {
    Name        = "web-server"
    Environment = "dev"
  }

  lifecycle {
    # Zero-downtime updates
    create_before_destroy = true

    # Ignore tags added by security/compliance tools
    ignore_changes = [
      tags["LastScannedBy"],
      tags["ComplianceStatus"],
    ]
  }
}
```

Note: You cannot combine `prevent_destroy = true` with `create_before_destroy = true` on the same resource in a meaningful way - if Terraform can't destroy the resource, it can't replace it either.

Best Practices

Security Groups

Always use `create_before_destroy` when security groups are attached to running instances:

```
resource "aws_security_group" "web" {
  name   = "web-sg-${random_id.suffix.hex}"
  vpc_id = data.aws_vpc.default.id

  lifecycle {
    create_before_destroy = true
  }
}
```

Critical Data Stores

Use `prevent_destroy` for S3 buckets and databases that contain important data:

```
resource "aws_s3_bucket" "data" {
  bucket = "critical-data-bucket"

  lifecycle {
    prevent_destroy = true
  }
}
```

External Modifications

Use `ignore_changes` specifically for attributes you know will be modified externally:

```
resource "aws_instance" "web" {
  # ... configuration ...

  lifecycle {
    ignore_changes = [
      tags["LastScannedBy"],
    ]
  }
}
```

Documentation

Comment why each lifecycle option is used - your future self (and teammates) will thank you:

```
lifecycle {
  # Create new instance before destroying old for zero-downtime deploys
  create_before_destroy = true

  # Security scanner adds this tag; don't fight it
  ignore_changes = [tags["LastScannedBy"]]
}
```

Key Takeaways

1. **create_before_destroy** reverses the normal destroy-then-create order, eliminating downtime during resource replacement
2. **prevent_destroy** stops Terraform from destroying a resource, protecting critical data from accidental deletion
3. **ignore_changes** tells Terraform to skip certain attributes when comparing state, useful for externally modified attributes
4. **Multiple settings** can be combined in one lifecycle block to address different concerns
5. **Security groups** attached to running instances should almost always have `create_before_destroy = true`
6. **Critical data stores** like S3 buckets and databases benefit from `prevent_destroy = true`

Knowledge Check

Question 1

What is the default order Terraform uses when a resource needs to be replaced?

A) Create the new resource, then destroy the old one B) Destroy the old resource, then create the new one C) Update the resource in place D) Terraform prompts you to choose

---

Question 2

You have an S3 bucket with `prevent_destroy = true`. What happens when you run `terraform destroy`?

A) The bucket is deleted after a confirmation prompt B) Terraform exits with an error and deletes nothing C) The bucket is skipped but other resources are deleted D) Terraform removes the bucket from state but doesn't delete it

---

Question 3

When would you use `ignore_changes` on a resource?

A) When you want to prevent the resource from being destroyed B) When external tools or AWS add attributes that Terraform shouldn't try to remove C) When you want Terraform to create the resource before destroying the old one D) When you want to force the resource to be recreated

---

Answers

Question 1: B - By default, Terraform destroys the existing resource first, then creates the new one. This can cause a gap where the resource doesn't exist. The `create_before_destroy` setting reverses this order.

Question 2: B - When `prevent_destroy = true`, Terraform refuses to proceed with any plan that would destroy the resource. It exits with an error, and no resources are destroyed. You must remove or disable the `prevent_destroy` setting first.

Question 3: B - `ignore_changes` is used when external tools, AWS services, or other processes modify resource attributes that Terraform shouldn't try to "correct." For example, tags added by AWS Auto Scaling or cost allocation tools.

Dictionary of Terms

create_before_destroy - A lifecycle setting that tells Terraform to create the replacement resource before destroying the original. Eliminates the gap where a resource doesn't exist during replacement.

ignore_changes - A lifecycle setting that tells Terraform to skip specified attributes when comparing desired state to actual state. Changes to ignored attributes won't appear in plans.

lifecycle block - A special configuration block inside a resource that controls how Terraform creates, updates, and destroys that resource.

meta-argument - A special argument like `lifecycle`, `count`, or `for_each` that works with any resource type and changes how Terraform handles the resource.

prevent_destroy - A lifecycle setting that causes Terraform to exit with an error if a plan would destroy the resource. Protects critical resources from accidental deletion.

zero-downtime update - An update strategy where the new version of a resource is created and made active before the old version is removed, ensuring continuous availability.

Documentation Links

- Lifecycle Meta-Arguments
- create_before_destroy
- prevent_destroy
- ignore_changes