

# Terraform Import Refresh Drift



explain Terraform import –  
Terraform refresh –  
Terraform drift

## Terraform Import

`terraform import` is used to bring existing infrastructure resources under Terraform's management without having to recreate them. This allows Terraform to track the resource in its state file while leaving the actual infrastructure unchanged.

### Example Usage:

```
sh

terraform import aws_instance.my_instance i-1234567890abcdef0
```

- This imports the AWS EC2 instance `i-1234567890abcdef0` into Terraform as `aws_instance.my_instance`.
- After importing, you should run `terraform plan` to review the configuration and ensure it matches the actual resource.

## Terraform Refresh

`terraform refresh` updates the Terraform state file with the actual current state of the infrastructure. This is useful when changes have been made outside of Terraform (manually or via other tools).

### Usage:

```
sh

terraform refresh
```

- Terraform will fetch the latest state of all resources from the cloud provider.
- It does **not** modify infrastructure, only updates the local state file.
- Since Terraform 0.15, `terraform refresh` is deprecated, and `terraform apply -refresh-only` should be used instead.

## Terraform Drift

Terraform drift refers to unintended changes in infrastructure that cause a mismatch between Terraform's state file and the actual infrastructure. This can happen due to:

- Manual changes made to resources outside Terraform.
- Changes made by other automation tools or scripts.

- Provider-side changes (e.g., AWS updating an instance type).

To detect drift:

```
sh

terraform plan
```

- If Terraform detects changes that were not applied via Terraform, they will be shown in the plan.
- To fix drift, either:
  - Modify Terraform configuration to match the actual state.
  - Reapply the Terraform plan (`terraform apply`) to enforce the desired state.

Would you like a real-world example of handling drift? 🚀



yes

## Real-World Example of Handling Terraform Drift

### Scenario:

Imagine your team is managing an AWS EC2 instance using Terraform. The original Terraform configuration looks like this:

### Terraform Configuration (`main.tf`)

```
hcl

resource "aws_instance" "web_server" {
  ami           = "ami-0abcd1234efgh5678" # Amazon Linux AMI
  instance_type = "t2.micro"
  tags = {
    Name = "Terraform-Managed-Instance"
  }
}
```

### Drift Occurs:

A team member (or an automated process) manually changes the instance type from `t2.micro` to `t3.micro` using the AWS Console. Terraform is unaware of this change.

### Detecting the Drift:

Running `terraform plan` will show:

```
sh

~ aws_instance.web_server
  instance_type: "t2.micro" => "t3.micro"
```

This indicates that Terraform's state file still thinks the instance type is `t2.micro`, but the actual instance type is now `t3.micro`.

## Handling the Drift:

### 1. Option 1: Accept the Change

- If you want to keep the manually made change (`t3.micro`), update `main.tf`:

```
hcl

instance_type = "t3.micro"
```

- Then run:

```
sh

terraform apply
```

- This updates the Terraform state to match reality.

### 2. Option 2: Revert to the Desired State

- If you want to enforce `t2.micro`, run:

```
sh

terraform apply
```

- Terraform will update the instance type back to `t2.micro`, enforcing the original configuration.

## Preventing Future Drift:

- Use **Terraform Cloud** or **Terraform State Locking** to prevent unauthorized changes.
- Use **AWS IAM Policies** to restrict manual modifications.
- Schedule periodic `terraform plan` runs to detect and alert on drift.

This ensures that all infrastructure changes are tracked and controlled via Terraform. 🚀