Terraform - DAY-3

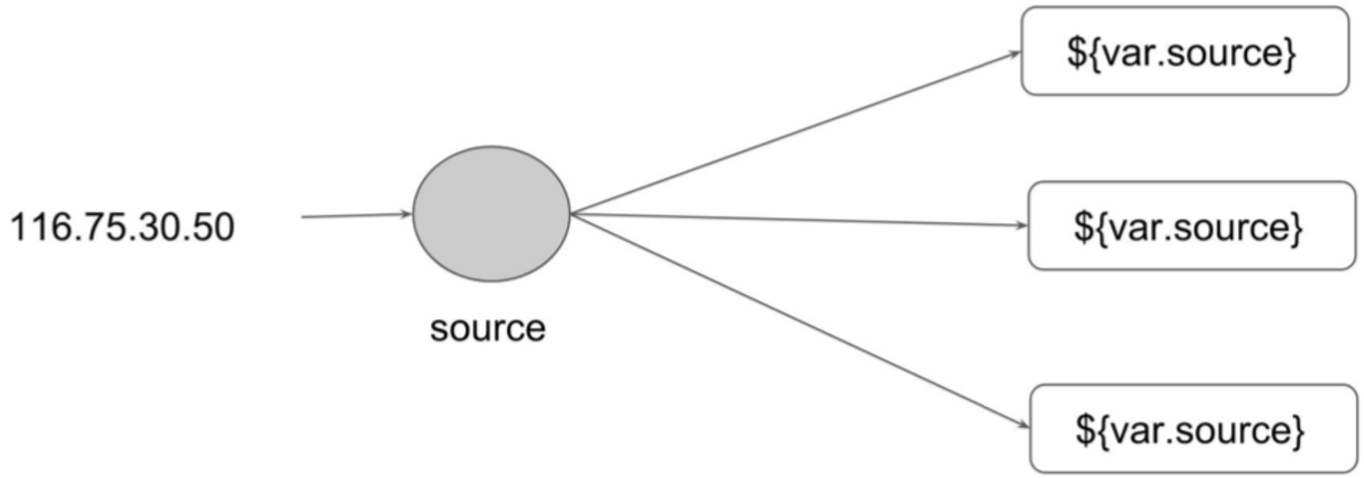
Presented By:
Rashi Rana

DRY Principle

Understanding DRY Approach

In software engineering, don't repeat yourself (DRY) is a principle of software development aimed at reducing the repetition of software patterns.

In the earlier lecture, we were making static content into variables so that there can be the single source of information.



We are repeating resource code

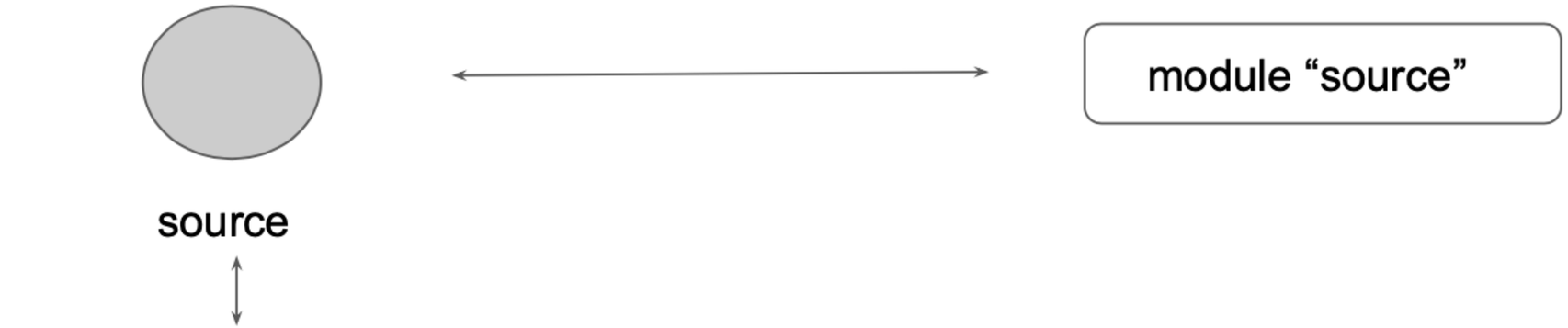
We do repeat multiple times various terraform resources for multiple projects.

Sample EC2 Resource

```
resource "aws_instance" "myweb" {
ami = "ami-bf5540df"
instance_type = "t2.micro"
security_groups = ["default"]
}
```

Centralized Structure

We can centralize the terraform resources and can call out from TF files whenever required.

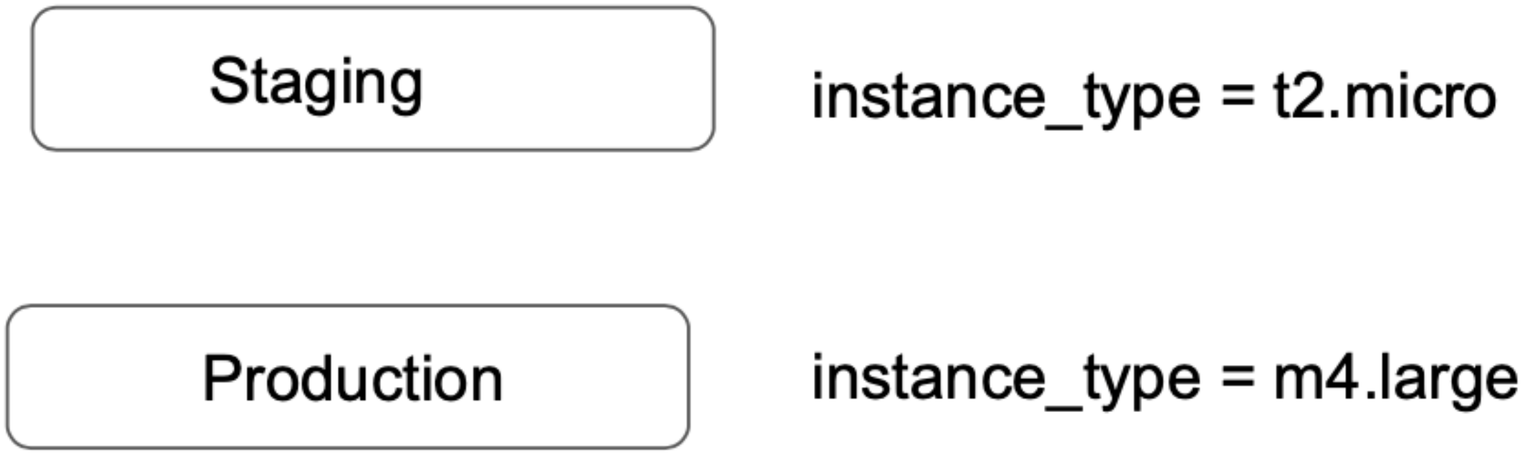



```
resource "aws_instance" "myweb" {

    ami = "ami-bf5540df"

   instance_type = "t2.micro"

   security_groups = ["default"]

}
```

Challenges with Modules

Challenges

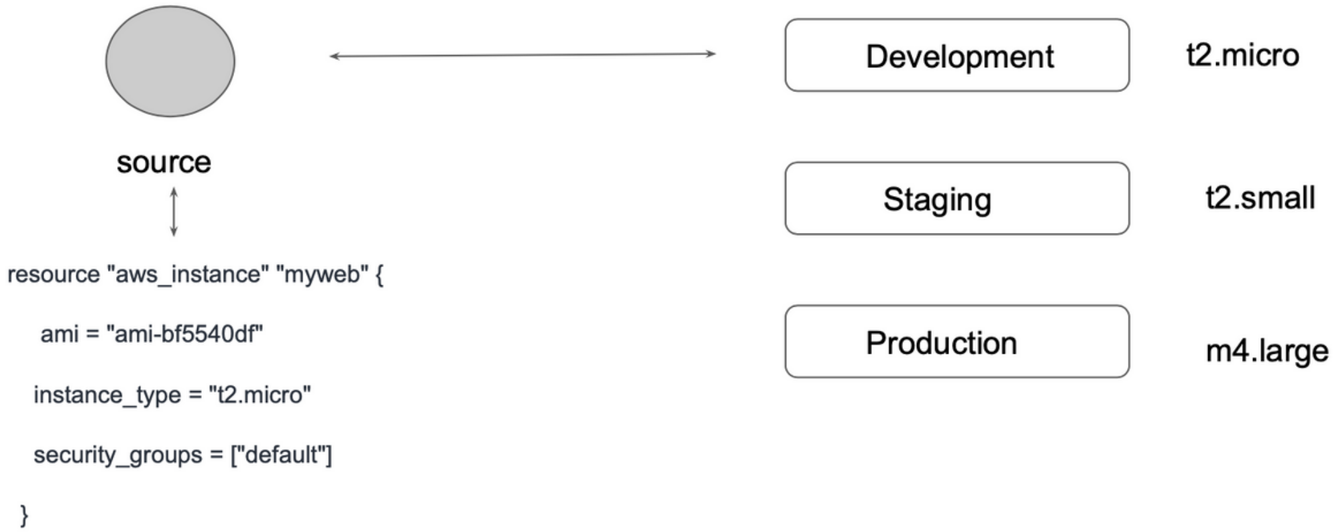
One common need on infrastructure management is to build environments like staging, production with similar setup but keeping environment variables different.



Challenges

One common need on infrastructure management is to build environments like staging, production with similar setup but keeping environment variables different.

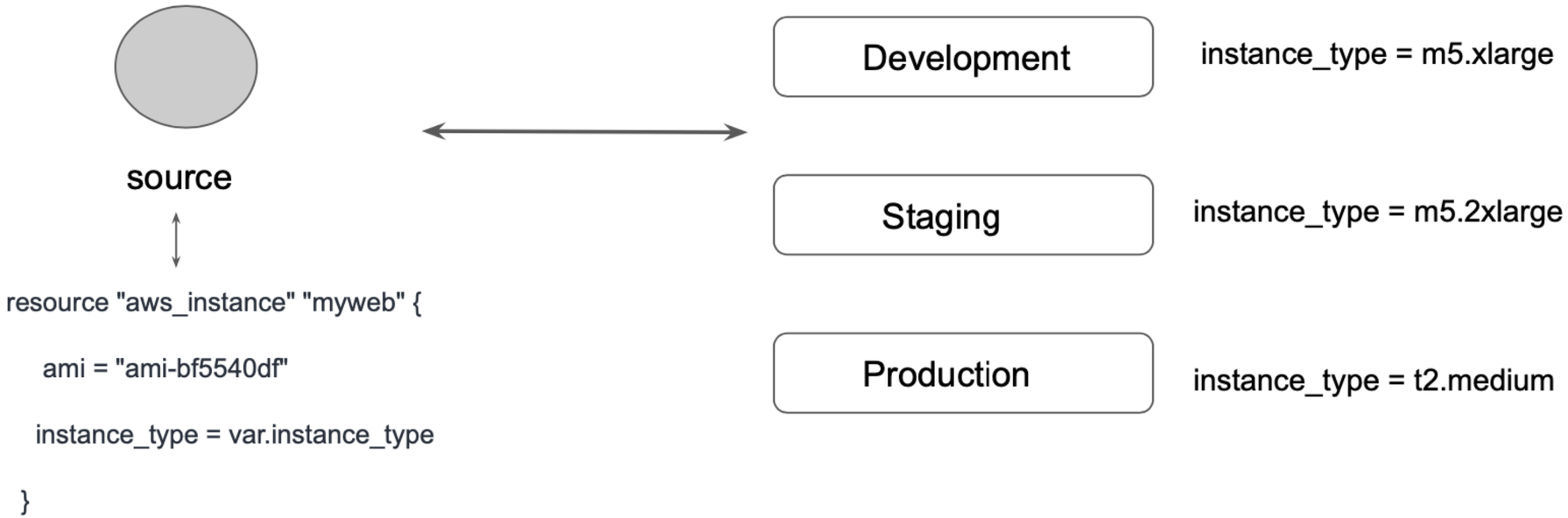
When we use modules directly, the resources will be replica of code in the module.



Using Locals with Modules

Understanding the challenge

Using variables in Modules can also allow users to override the values which you might not want.



Setting the Context

There can be many repetitive values in modules and this can make your code difficult to maintain.

You can centralize these using variables but users will be able to override it.

```
resource "aws_security_group" "elb-sg" {
  name        = "myelb-sg"


  ingress {
    description      = "Allow Inbound from Secret Application"
    from_port        = 8443
    to_port          = 8443
    protocol         = "tcp"
    cidr_blocks      = ["0.0.0.0/0"]
  }
```

Hardcoded Port


```
resource "aws_security_group" "elb-sg" {
  name        = "myelb-sg"


  ingress {
    description      = "Allow Inbound from Secret Application"
    from_port        = var.app_port
    to_port          = var.app_port
    protocol         = "tcp"
    cidr_blocks      = ["0.0.0.0/0"]
  }
```

Variable Port

Using Locals

Instead of variables, you can make use of locals to assign the values.

```
resource "aws_security_group" "ec2-sg" {
  name        = "myec2-sg"

  ingress {
    description      = "Allow Inbound from Secret Application"
    from_port        = local.app_port
    to_port          = local.app_port
    protocol         = "tcp"
    cidr_blocks      = ["0.0.0.0/0"]
  }

locals {
  app_port = 8443
}
```

Module Outputs

Revising Output Values

Output values make information about your infrastructure available on the command line, and can expose information for other Terraform configurations to use.

```
output "instance_ip_addr" {
  value = aws_instance.server.private_ip
}
```

Accessing Child Module Outputs

In a parent module, outputs of child modules are available in expressions as module.<MODULE NAME>.<OUTPUT NAME>

```
resource "aws_security_group" "ec2-sg" {
  name        = "myec2-sg"

  ingress {
    description      = "Allow Inbound from Secret Application"
    from_port        = 8433
    to_port          = 8433
    protocol         = "tcp"
    cidr_blocks      = ["0.0.0.0/0"]
  }

}

output "sg_id" {
  value = aws_security_group.ec2-sg.arn
}
```



```
module "sgmodule" {
  source = "../../modules/sg"
}

resource "aws_instance" "web" {
  ami           = "ami-0ca285d4c2cda3300"
  instance_type = "t3.micro"
  vpc_security_group_ids = [module.sgmodule.sg_id]
}
```

Terraform Registry

https://registry.terraform.io/

Module Location

If we intend to use a module, we need to define the path where the module files are present. The module files can be stored in multiple locations, some of these include:

- Local Path
- GitHub
- Terraform Registry
- S3 Bucket
- HTTP URLs

Using Registry Module in Terraform

To use Terraform Registry module within the code, we can make use of the source argument that contains the module path.

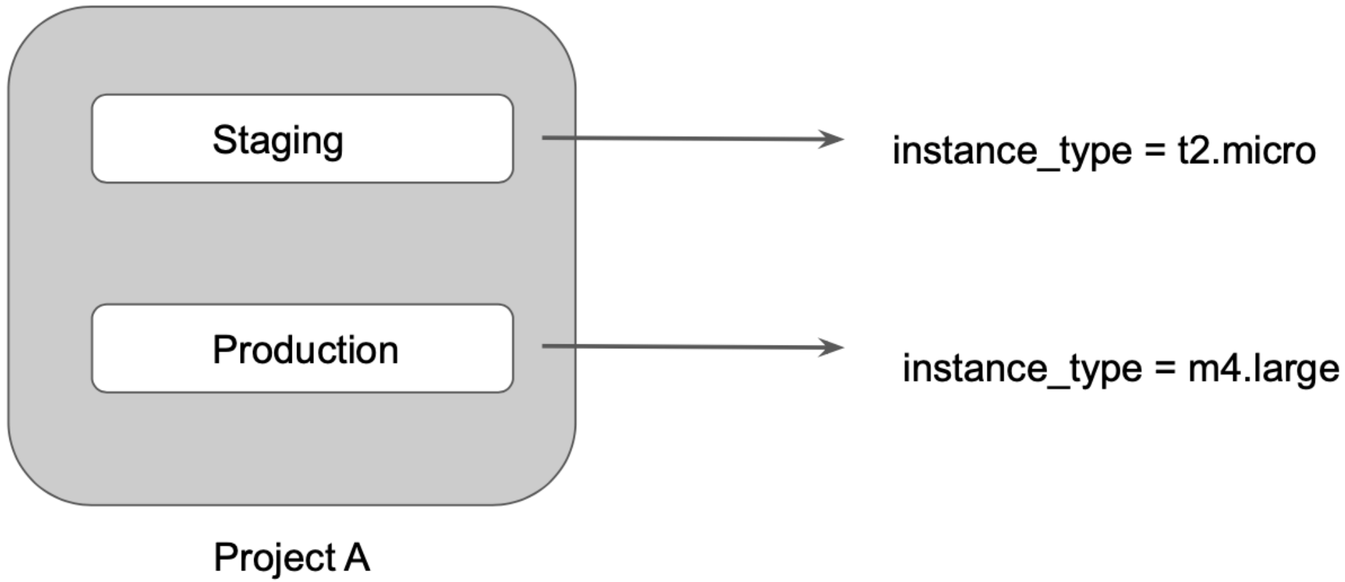
Below code references to the EC2 Instance module within terraform registry.

```
module "ec2-instance" {
  source  = "terraform-aws-modules/ec2-instance/aws"
  version = "2.13.0"
  # insert the 10 required variables here
}
```

Terraform Workspace

Understanding Workspaces

Terraform allows us to have multiple workspaces, with each of the workspace we can have different set of environment variables associated



Provisioners

Provisioners are interesting

Till now we have been working only on creation and destruction of infrastructure scenarios.

Let’s take an example:

We created a web-server EC2 instance with Terraform.

Problem: It is only an EC2 instance, it does not have any software installed.

What if we want a complete end to end solution ?

Welcome to Terraform Provisioners

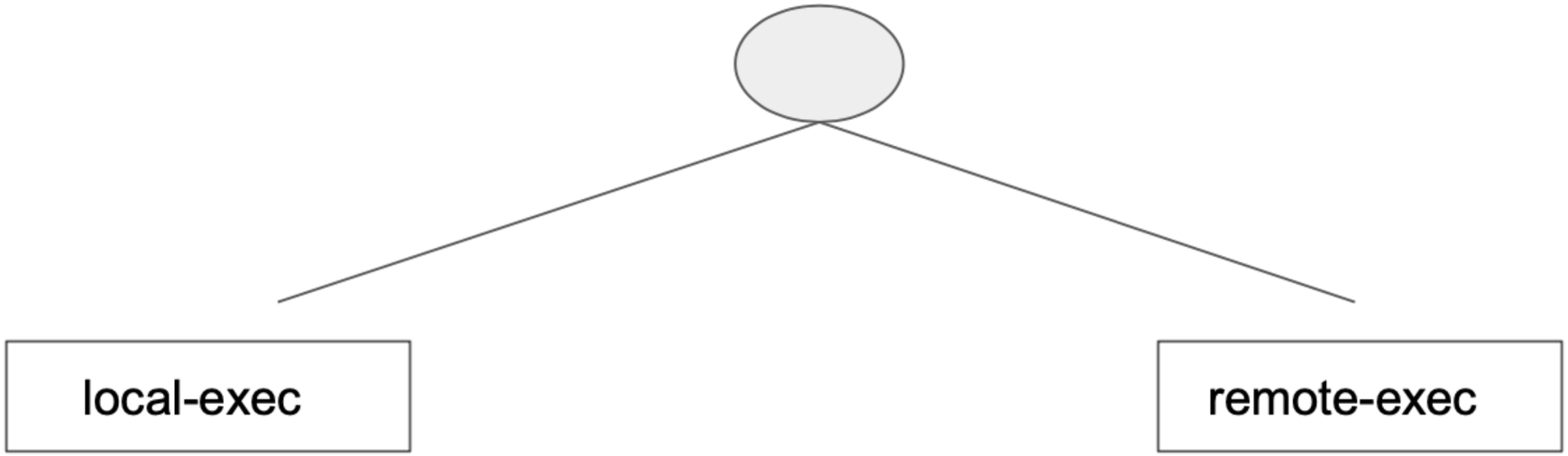
Provisioners are used to execute scripts on a local or remote machine as part of resource creation or destruction.

Let’s take an example:

On creation of Web-Server, execute a script which installs Nginx web-server.

Types of Provisioners

Types of Provisioners



Local Exec Provisioners

local-exec provisioners allow us to invoke local executable after the resource is created

Let’s take an example:

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


provisioner "local-exec" {
 command = "echo ${aws_instance.web.private_ip} >> private_ips.txt"
}

}
```

Remote Exec Provisioners

Remote-exec provisioners allow invoking scripts directly on the remote server.

Let’s take an example:

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


provisioner "remote-exec" {

.....................................
}
}
```