SKILLS | DevOps and Cloud Computing

Terraform Configuration, State Management, and Modules

Objective

- Define and configure Terraform resources in main.tf.
- Use input variables and pass values via .tfvars for flexible configurations.
- Create output values in outputs.tf to reference important resource attributes.
- Understand Terraform state management, including local v remote backends.
- Learn how to create and use Terraform modules for reusab infrastructure code.

Explaining how Terraform defines resources in main.tf.

Let’s see

- Terraform defines resources in the **main.tf** file using resource blocks, which describe the infrastructure objects to be created or managed, such as virtual machines, networks, or storage buckets.
- Each resource block specifies the resource type, a unique name, and configuration arguments that define the resource's attributes.
- **main.tf** contains resource blocks that declaratively define the infrastructure components Terraform will provision, using variables and configuration arguments to customize those resources.

PW SKILLS

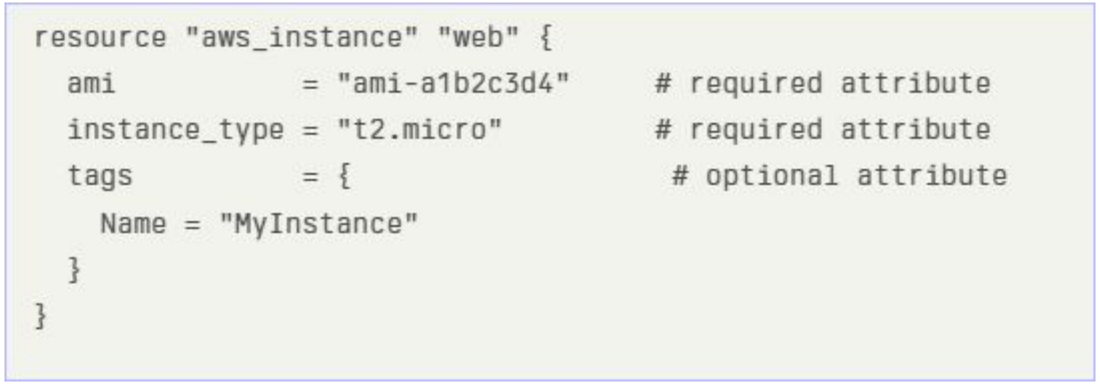
Discussing the structure of a resource block, including required and optional attributes.

Let’s discuss

A Terraform resource block has a defined structure that includes required and optional attributes:

Structure:

A resource block starts with the keyword resource followed by the resource type (e.g., "aws instance") and a unique local name (e.g., "web"). Inside curly braces {}, it contains configuration arguments specific to that resource type. For example:



```
resource "aws_instance" "web" {
  ami           = "ami-a1b2c3d4"     # required attribute
  instance_type = "t2.micro"         # required attribute
  tags          = {                   # optional attribute
    Name = "MyInstance"
  }
}
```

Let’s discuss

Required Attributes:

These depend on the resource type and must be provided for Terraform to create the resource. For example, ami and instance type are required for an AWS EC2 instance.

Optional Attributes:

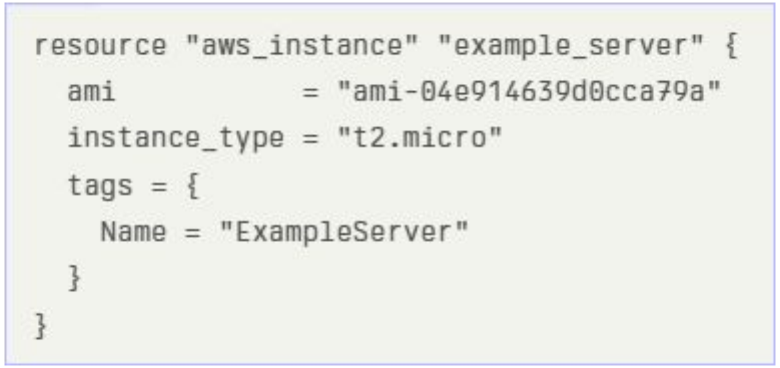
These can be omitted or set conditionally. Terraform treats attributes set to null as unset, so optional attributes can be dynamically included or excluded based on input variables. This is often done using conditional expressions or dynamic blocks to avoid including attributes when not needed.

Providing examples of commonly used resources

Examples

Compute Resources

- AWS EC2 Instance: Defines a virtual machine in AWS. Example snippet:

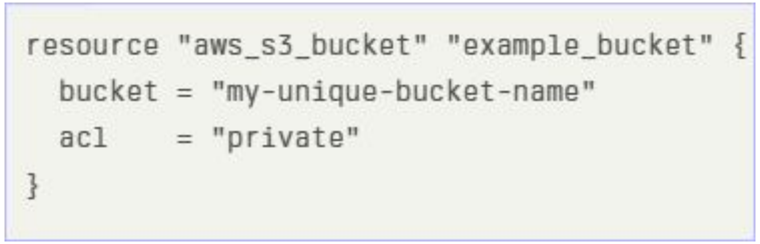


```
resource "aws_instance" "example_server" {
  ami           = "ami-04e914639d0cca79a"
  instance_type = "t2.micro"
  tags = {
    Name = "ExampleServer"
  }
}
```

Examples

Storage Resources

- AWS S3 Bucket: Defines an object storage bucket in AWS. Example snippet:



```
resource "aws_s3_bucket" "example_bucket" {
  bucket = "my-unique-bucket-name"
  acl    = "private"
}
```

Examples

Storage Resources

- AWS S3 Bucket: Defines an object storage bucket in AWS. Example snippet:

```
resource "aws_s3_bucket" "example_bucket" {
  bucket = "my-unique-bucket-name"
  acl    = "private"
}
```

- Google Cloud Storage Bucket: Used for storing Terraform state or other data in GCP. Example snippet:

```
resource "google_storage_bucket" "terraform_state" {
  name          = "my-terraform-state-bucket"
  location      = "US"
  force_destroy = true
}
```

Examples

Networking Resources

- AWS VPC (Virtual Private Cloud): Defines a virtual network in AWS. Example snippet:

```
resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0/16"
  tags = {
    Name = "ProjectVPC"
  }
}
```

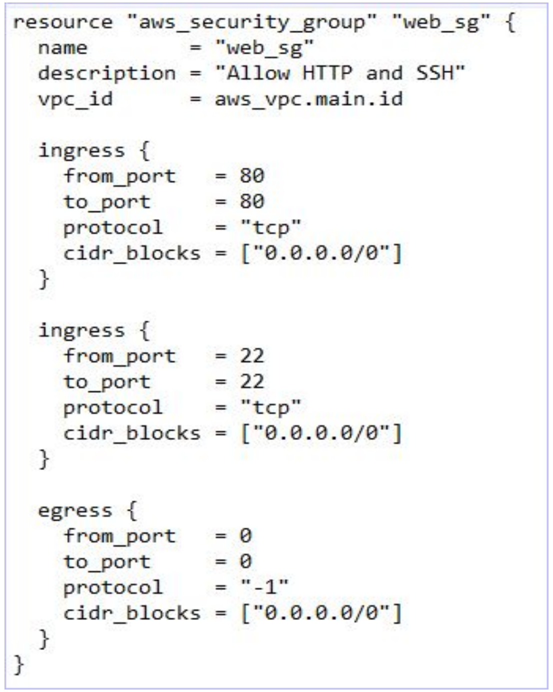
Examples

- AWS Subnet: Defines subnets within a VPC, specifying availability zones and CIDR blocks. Example snippet:

```
resource "aws_subnet" "public_subnets" {
  count             = 3
  vpc_id            = aws_vpc.main.id
  cidr_block        = element(var.public_subnet_cidrs, count.index)
  availability_zone = element(var.azs, count.index)
  tags = {
    Name = "Public Subnet ${count.index + 1}"
  }
}
```

Examples

- AWS Security Group: Defines firewall rules controlling inbound and outbound traffic. Example snippet:



```
resource "aws_security_group" "web_sg" {
  name        = "web_sg"
  description = "Allow HTTP and SSH"
  vpc_id      = aws_vpc.main.id

  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

PW SKILLS

Explaining the purpose of variables in Terraform for parameterizing infrastructure.

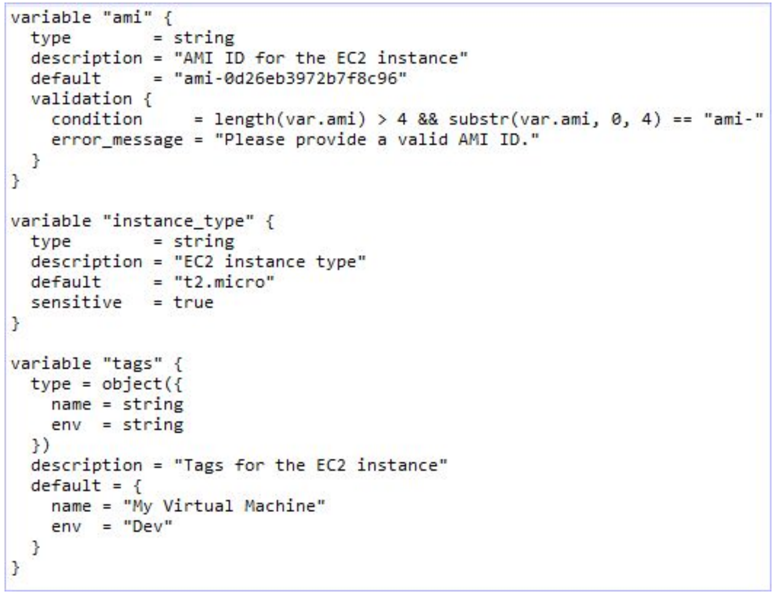
Let’s see

- Terraform variables serve as placeholders for values that parameterize infrastructure configurations, making them dynamic, reusable, and adaptable without changing the core code.
- They allow you to define inputs such as instance types, region names, or resource counts externally, enabling the same configuration to be deployed across different environments by simply changing variable values.
- This enhances scalability, maintainability, and flexibility in infrastructure management.
- Overall, variables enable efficient and customizable infrastructure provisioning by separating configuration logic from environment-specific data.

Demonstrating defining input variables in variables.tf.

Let’s do it

- In Terraform, input variables are defined in a variables.tf file using the variable block. Each variable can include attributes like type, description, default, and optional validation or sensitive flags. Here is a concise example:



```
variable "ami" {
  type        = string
  description = "AMI ID for the EC2 instance"
  default     = "ami-0d26eb3972b7f8c96"
  validation {
    condition     = length(var.ami) > 4 && substr(var.ami, 0, 4) == "ami-"
    error_message = "Please provide a valid AMI ID."
  }
}

variable "instance_type" {
  type        = string
  description = "EC2 instance type"
  default     = "t2.micro"
  sensitive   = true
}

variable "tags" {
  type = object({
    name = string
    env  = string
  })
  description = "Tags for the EC2 instance"
  default = {
    name = "My Virtual Machine"
    env  = "Dev"
  }
}
```

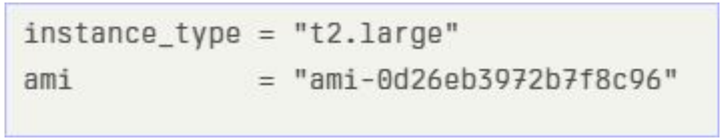
SKILLS

Explaining how to pass values dynamically using .tfvars files.

Let’s see

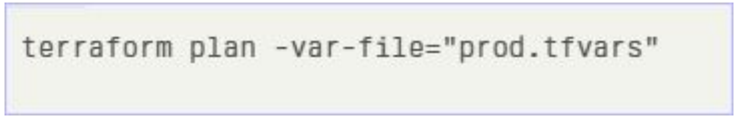
Terraform uses .tfvars files to pass variable values dynamically, allowing customization of infrastructure without modifying the main configuration files. To use them:

- Create a file with a .tfvars extension (e.g., terraform.tfvars, dev.tfvars, prod.tfvars) containing variable assignments in HCL or JSON format, for example:



```
instance_type = "t2.large"
ami           = "ami-0d26eb3972b7f8c96"
```

- By default, Terraform automatically loads terraform.tfvars or files ending with .auto.tfvars.
- For other .tfvars files, specify the file explicitly when running commands using the -var-file flag, e.g.:



```
terraform plan -var-file="prod.tfvars"
```

Let’s see

- This approach enables managing multiple environment configurations by switching .tfvars files without changing Terraform code.
- Additionally, values passed via CLI -var flags override .tfvars values, which in turn override default variable values.
- Using .tfvars files simplifies variable management, especially when dealing with many variables or multiple deployment environments.

PW SKILLS

Discussing default values and type constraints for input variables.

Let’s discuss

Terraform input variables can have default values and type constraints to control their usage:

Default Values:

- When a default value is specified in a variable block using the default argument, the variable becomes optional. If no value is provided during execution, Terraform uses this default.
- The default must be a literal value and cannot reference other variables or resources. This simplifies configuration by providing sensible fallbacks.

Let’s discuss

Type Constraints:

- The type argument restricts the kind of values a variable can accept, such as string, number, bool, list, map, object, or any. If no type is set, any value type is accepted.
- Specifying types helps catch errors early by validating inputs and improves documentation for users of the module. The default value must conform to the declared type if both are set.

PW SKILLS

Explaining why outputs are useful for retrieving information from deployed infrastructure.

Let’s see

Terraform outputs are useful because they export and expose important information about deployed infrastructure, such as resource attributes (e.g., IP addresses, IDs), making these values accessible outside the module or root configuration. This enables:

- Sharing data between modules and workspaces.
- Passing information to external automation tools or systems.
- Displaying key details after deployment for verification or use.
- Accessing outputs via remote state for cross-configuration references.

Demonstrating defining output values such as IP addresses, DNS names, and resource IDs.

Let’s do it

Here are concise examples of defining Terraform output values for common resource attributes like IP addresses, DNS names, and resource IDs:

```
output "instance_public_ip" {
  description = "Public IP address of the EC2 instance"
  value       = aws_instance.web_server.public_ip
}

output "instance_dns_name" {
  description = "Public DNS name of the EC2 instance"
  value       = aws_instance.web_server.public_dns
}

output "vpc_id" {
  description = "ID of the VPC"
  value       = aws_vpc.main.id
}
```

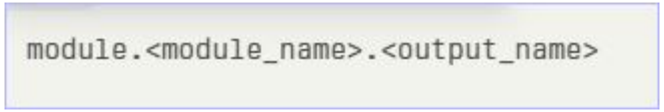
PW SKILLS

Discussing referencing outputs from one Terraform module in another.

Let’s discuss

To reference outputs from one Terraform module in another, you first define the output in the child module using an output block that exposes the desired value (e.g., resource ID, IP address).

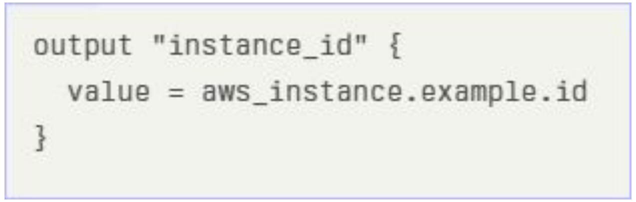
Then, in the parent or calling module, you access that output via the module's namespace using the syntax:



```
module.<module_name>.<output_name>
```

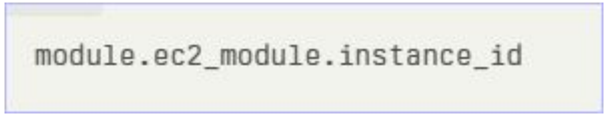
Let’s discuss

For example, if a child module defines:



```
output "instance_id" {
  value = aws_instance.example.id
}
```

You can reference it in the root module as:



```
module.ec2_module.instance_id
```

PW SKILLS

Take A 5-Minute **Break!**

- Stretch and relax
- Hydrate
- Clear your mind
- Be back in 5 minutes

SKILLS

Explaining how Terraform state tracks the current state of managed infrastructure.

Let’s see

- Terraform state tracks the current state of managed infrastructure by storing information about each resource in a state file (terraform.tfstate).
- This file acts as a source of truth for Terraform, mapping resources in your configuration to real infrastructure in the cloud or other providers.

Let’s see

Here’s how it works:

- Resource Mapping: It keeps track of resource IDs and metadata, linking them to the resources defined in your .tf files.
- Change Detection: When you run terraform plan, Terraform compares the state file with your current configuration and the actual infrastructure to detect changes.
- Efficient Updates: During terraform apply, it uses this comparison to create, update, or delete only what's necessary.

Pop Quiz

Q. What is the primary purpose of the Terraform state file (terraform.tfstate)?

A

To track the current state of managed infrastructure

B

To store the entire Terraform configuration

Pop Quiz

Q. What is the primary purpose of the Terraform state file (terraform.tfstate)?

A

To track the current state of managed infrastructure

B

To store the entire Terraform configuration

PW SKILLS

Discussing the importance of state files in planning and applying changes.

Let’s discuss

State files are crucial in Terraform because they:

- Track infrastructure: They store the current state of resources, enabling Terraform to know what exists.
- Plan accurately: Terraform compares the state file with the desired configuration to show precise changes.
- Apply efficiently: Only necessary changes are made, avoiding resource destruction or duplication.
- Enable collaboration: Remote state allows teams to work safely without overwriting each other's changes.

Demonstrating how to inspect the Terraform state to view existing resources.

Let’s do it

To inspect the Terraform state and view existing resources, use:

```
terraform state list
```

To view details of a specific resource:

```
terraform state show <resource_name>
```

Example:

```
terraform state show aws_instance.my_instance
```

Explaining state locking to prevent conflicts in collaborative environments.

Let’s see

- State locking in Terraform is used to prevent multiple users or processes from making simultaneous changes to the same state file. In collaborative environments, this avoids conflicts and potential corruption of infrastructure state.
- When a command like ‘terraform plan’ or ‘terraform apply’ is run, Terraform acquires a lock on the state file. Other operations are blocked until the lock is released, ensuring that only one user can modify the infrastructure at a time.

SKILLS

Discussing the differences between local state and remote state storage.

Let’s discuss

- **Local state** stores the terraform.tfstate file on your local machine, making it simple for single-user setups but risky for collaboration and backup.

Remote state stores the state file in a shared backend (like S3, Terraform Cloud), enabling:

- Collaboration: Shared access for teams
- State locking: Prevents concurrent changes
- Backups: Safer and more reliable

Explaining why remote state storage is essential for team collaboration.

Let’s discuss

Remote state storage is essential for team collaboration because it:

- Provides a shared source of truth for all team members
- Enables state locking to prevent conflicting changes
- Supports versioning and backups for recovery
- Improves security by storing state in a controlled, centralized location

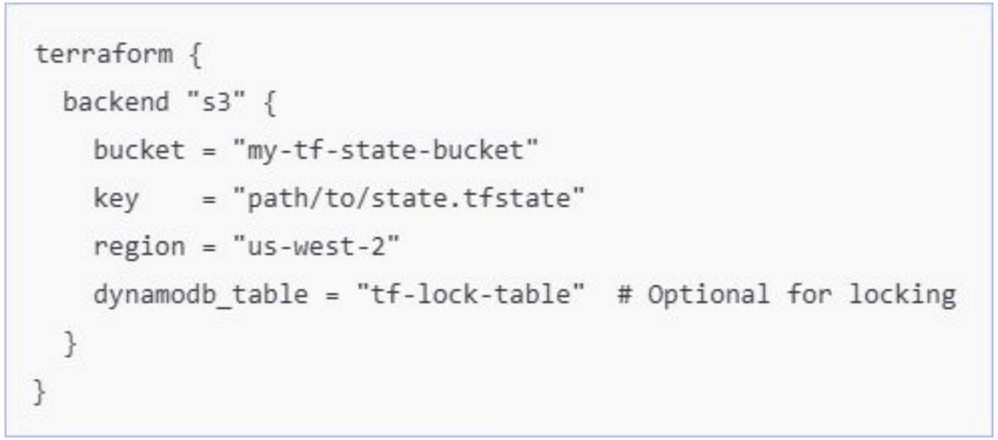
PW SKILLS

Demonstrating storing Terraform state in S3, Azure Blob Storage, or Google Cloud Storage.

Let’s do it

To store Terraform state remotely, configure the backend in your Terraform code. Here are short examples for each:

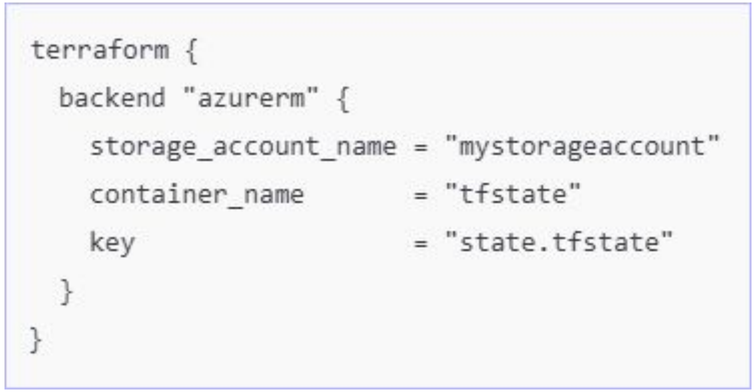
1. AWS S3:



```
terraform {
  backend "s3" {
    bucket = "my-tf-state-bucket"
    key    = "path/to/state.tfstate"
    region = "us-west-2"
    dynamodb_table = "tf-lock-table"  # Optional for locking
  }
}
```

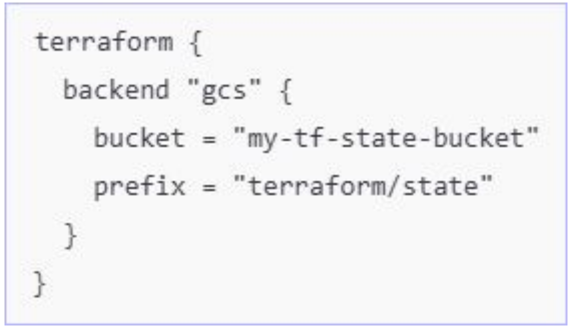
Let’s do it

2. Azure Blob Storage:



```
terraform {
  backend "azurerm" {
    storage_account_name = "mystorageaccount"
    container_name       = "tfstate"
    key                  = "state.tfstate"
  }
}
```

3. Google Cloud Storage (GCS):



```
terraform {
  backend "gcs" {
    bucket = "my-tf-state-bucket"
    prefix = "terraform/state"
  }
}
```

PW SKILLS

Explaining the benefits of remote backends, including state locking and shared access.

Let’s see

Remote backends offer key benefits:

- State locking: Prevents simultaneous changes, avoiding conflicts
- Shared access: Teams can safely work on the same infrastructure
- Centralized storage: Keeps state secure and consistent
- Backups & versioning: Enables recovery from errors
- Scalability: Supports team workflows and CI/CD pipelines

Discussing how Terraform integrates with Terraform Cloud and HashiCorp Remote Backends.

Let’s discuss

Terraform integrates with Terraform Cloud and HashiCorp Remote Backends to provide a collaborative, secure, and scalable infrastructure as code (IaC) provisioning workflow. This integration enables:

- Remote state storage: Centralized and secure state management.
- State locking & versioning: Prevents conflicts and supports rollback.
- Collaboration: Shared workspace for teams.
- Remote execution: Runs Terraform in the cloud, not locally.
- Access controls: Role-based permissions for secure operations.

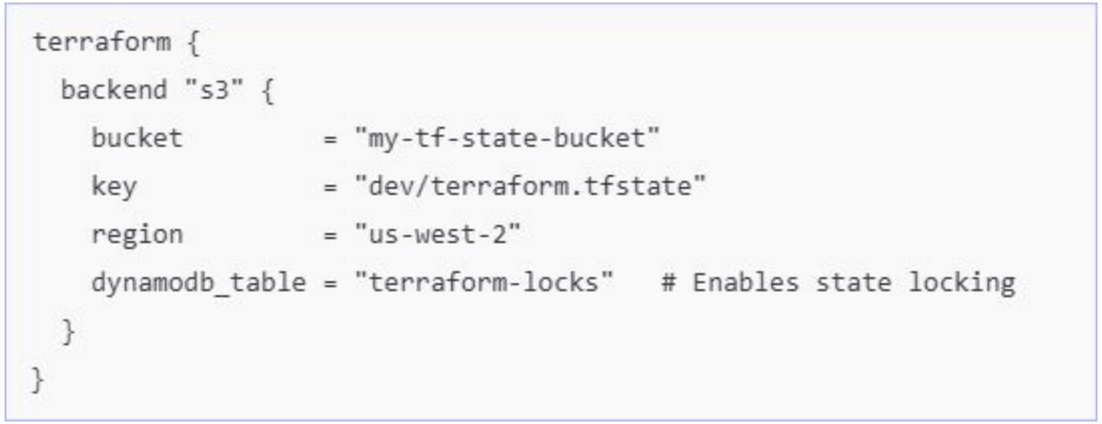
PW SKILLS

Demonstrating how multiple users can collaborate on Terraform configurations using remote state.

Let’s do it

To enable collaboration with multiple users in Terraform using remote state, follow these steps:

Step 1: Configure a Remote Backend (e.g., AWS S3 with DynamoDB)



```
terraform {
  backend "s3" {
    bucket         = "my-tf-state-bucket"
    key            = "dev/terraform.tfstate"
    region         = "us-west-2"
    dynamodb_table = "terraform-locks"   # Enables state locking
  }
}
```

Let’s do it

Step 2: Initialize Terraform

```
terraform init
```

This sets up the remote backend and migrates existing state if needed.

Step 3: Team Workflow

- Shared access: All users point to the same backend configuration.
- State locking: Only one user can modify infrastructure at a time.
- Versioning: Remote backends store state history for recovery.

PW SKILLS

Explaining the purpose of Terraform modules for organizing infrastructure.

Let’s see

Terraform modules help organize, reuse, and manage infrastructure code by grouping related resources into reusable blocks. They:

- Promote DRY principles (Don't Repeat Yourself)
- Improve readability and structure
- Enable scalability and consistency across environments
- Simplify complex infrastructure by breaking it into smaller parts

Pop Quiz

Q. What best describes a Terraform module?

A

A set of reusable Terraform configuration files

B

A cloud storage unit for Terraform state

Pop Quiz

Q. What best describes a Terraform module?

A

A set of reusable Terraform configuration files

B

A cloud storage unit for Terraform state

Discussing the structure of a Terraform module (main.tf, variables.tf, outputs.tf).

Let’s discuss

A Terraform module typically has this structure:

- main.tf – Defines the core resources and logic
- variables.tf – Declares input variables used in the module
- outputs.tf – Defines outputs to expose values from the module

This structure keeps code organized, reusable, and easy to understand.

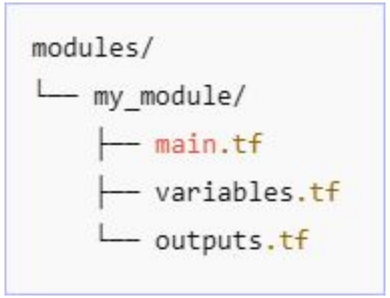
PW SKILLS

Demonstrating creating a custom module for reusable infrastructure components.

Let’s do it

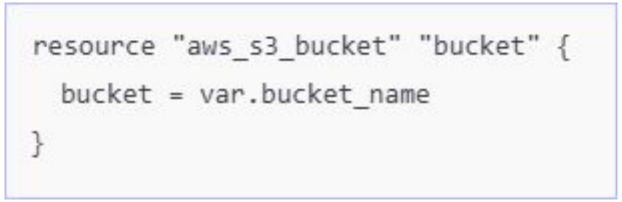
To create a custom Terraform module:

- Folder Structure



```
modules/
└── my_module/
    ├── main.tf
    ├── variables.tf
    └── outputs.tf
```

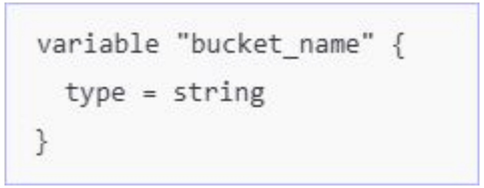
- Example: main.tf



```
resource "aws_s3_bucket" "bucket" {
  bucket = var.bucket_name
}
```

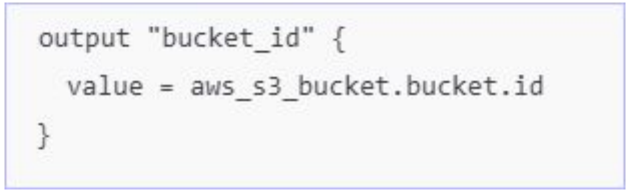
Let’s do it

- Example: variables.tf



```
variable "bucket_name" {
  type = string
}
```

- Example: outputs.tf



```
output "bucket_id" {
  value = aws_s3_bucket.bucket.id
}
```

Let’s do it

- Use the Module

In your root main.tf:

```
module "my_s3" {
  source      = "./modules/my_module"
  bucket_name = "my-reusable-bucket"
}
```

This allows you to reuse the ‘my module’ component across projects.

SKILLS

Introducing the Terraform Registry and how to find pre-built modules.

Terraform Registry

The Terraform Registry is an official online repository where you can find pre-built, reusable Terraform modules and providers for various cloud platforms.

How to Find Modules:

- Go to registry.terraform.io
- Use the search bar to find modules (e.g., "AWS VPC", "GCP storage")
- Browse module details, usage examples, inputs, and outputs
- Copy the module snippet into your Terraform config

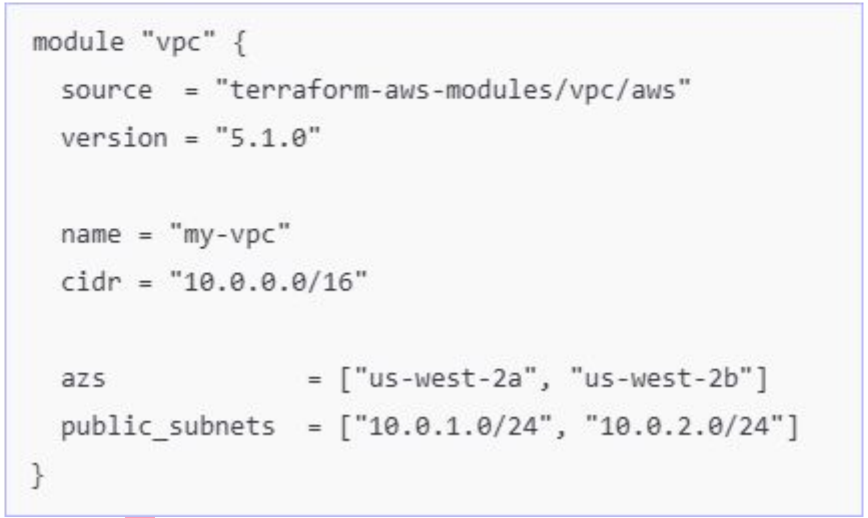
SKILLS

Demonstrating using a module from the registry for deploying common infrastructure.

Let’s do it

To use a module from the Terraform Registry:

Example: Deploy an AWS VPC



```
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "5.1.0"

  name = "my-vpc"
  cidr = "10.0.0.0/16"

  azs             = ["us-west-2a", "us-west-2b"]
  public_subnets  = ["10.0.1.0/24", "10.0.2.0/24"]
}
```

Let’s do it

Steps:

- Add the module block to your .tf file
- Run ‘terraform init’ to download the module
- Run ‘terraform apply’ to deploy the infrastructure

This reuses proven code and simplifies setup.

PW SKILLS

Explaining the benefits of custom modules for networking and cloud infrastructure.

Benefits

Custom modules for networking and cloud infrastructure offer key benefits:

- Reusability
- Consistency
- Simplification
- Maintainability
- Scalability

PW SKILLS

Discussing best practices for writing scalable and reusable modules.

Let’s discuss

Best practices for writing scalable and reusable Terraform modules:

- Use input variables
- Set defaults
- Output key values
- Keep modules focused
- Write clear documentation
- Avoid hardcoding
- Use version control

Demonstrating how to reference modules in Terraform configurations.

Let’s do it

To reference a module in Terraform:

Example:

```
module "vpc" {
  source   = "./modules/vpc"   # or from registry
  name     = "my-vpc"
  cidr     = "10.0.0.0/16"
}
```

Use Outputs from the Module:

```
output "vpc_id" {
  value = module.vpc.vpc_id
}
```

You call modules like functions, pass variables, and access their outputs with module.<name>.<output>.

Time for case study!

Important

- Complete the post-class assessment
- Complete assignments (if any)
- Practice the concepts and techniques taught in this sessio
- Review your lecture notes
- Note down questions and queries regarding this session ar
consult the teaching assistants

Thanks
!