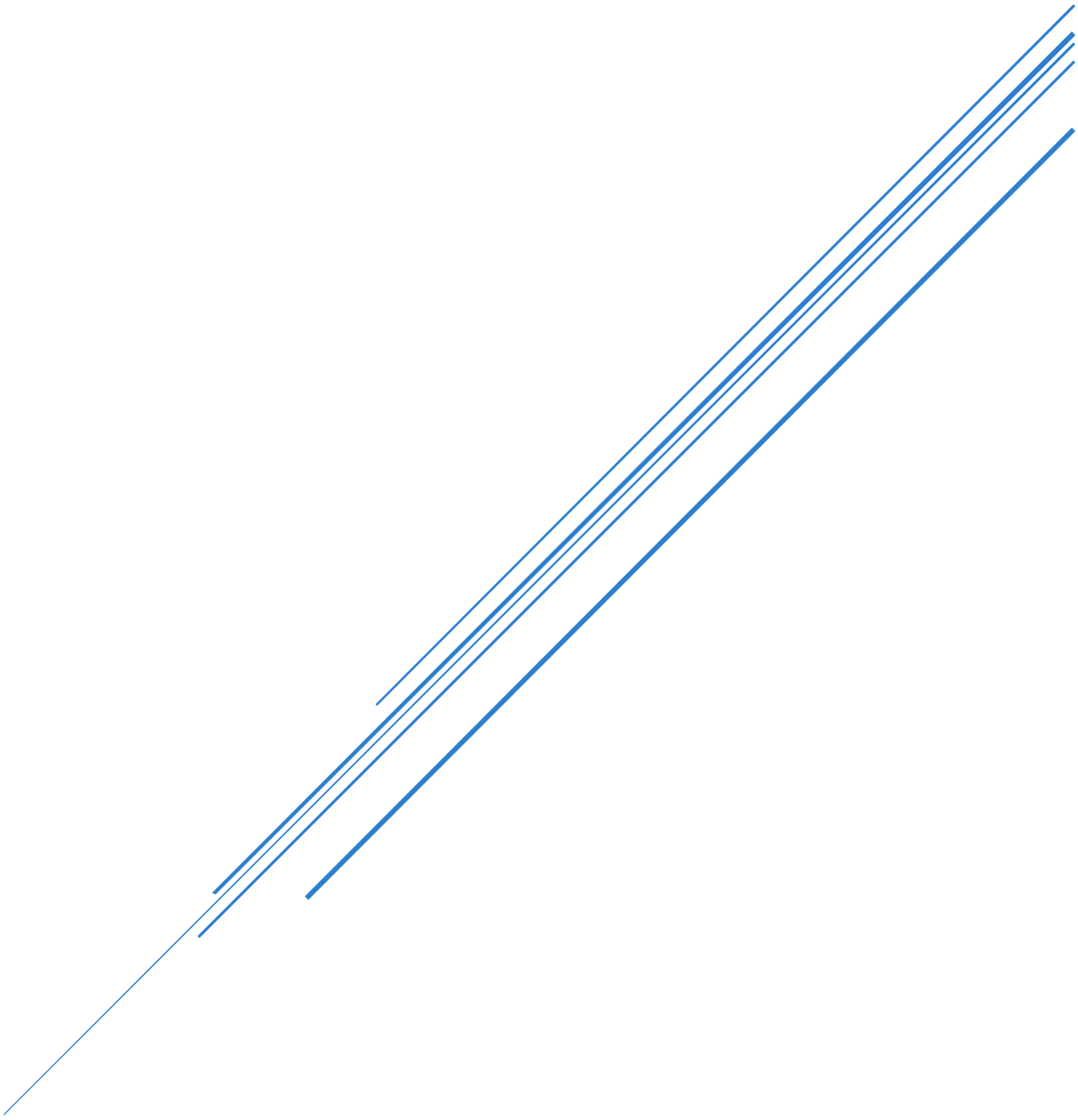
TERRAFORM MORE TOPICS

AWS IAC



Contents

Using Variables

Terraform tfvars File

In Terraform, .tfvars files are used to define values for variables declared in your configuration (variables.tf). They allow you to separate configuration values from your Terraform code, making it easier to manage and reuse.

1. Why Use tfvars Files?
   - ✓ Simplifies variable management: Instead of passing variables through the CLI or hardcoding them, you can define them in a .tfvars file.
   - ✓ Keeps code clean: Separates infrastructure logic (.tf files) from variable values (.tfvars files).
   - ✓ Reusable configurations: Easily swap configurations by using different .tfvars files.

2. Structure of a tfvars File
   - A tfvars file contains key-value pairs for variables.
   - The file extension is .tfvars or .auto.tfvars.
   - Terraform automatically loads .auto.tfvars files.

3. Step-by-Step: Using tfvars

Declare Variables in variables.tf

Define the variables in your variables.tf file:

```
variable "region" {
  description = "AWS region"
  default     = "us-east-1"
}

variable "instance_type" {
  description = "Type of EC2 instance"
  default     = "t2.micro"
}

variable "environment" {
  description = "Environment (e.g., dev, staging, production)"
}
```

Create a terraform.tfvars File

Create a file named terraform.tfvars and specify the variable values:

```
region      = "us-west-2"
instance_type = "t3.micro"
environment   = "dev"
```

Use tfvars in Terraform Configuration

Use the variables in your Terraform configuration (main.tf):

```
provider "aws" {
  region = var.region
}

resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0" # Replace with a valid AMI ID
```

```
  instance_type = var.instance_type

  tags = {
    Name        = "ExampleInstance"
    Environment = var.environment
  }
}
```

Apply the Configuration

```
terraform init
terraform apply
```

*Terraform will automatically load the terraform.tfvars file.

Specifying a Custom tfvars File

If you want to use a custom .tfvars file, specify it using the -var-file option:

Create a file named dev.tfvars:

```
region        = "us-west-2"
instance_type = "t3.micro"
environment   = "dev"
```

Use it during terraform apply:

```
terraform apply -var-file="dev.tfvars"
```

Automatic Loading of .auto.tfvars Files

Files ending in .auto.tfvars are automatically loaded by Terraform without explicitly specifying them.

Example:

File: variables.auto.tfvars

```
region        = "us-west-2"
instance_type = "t3.micro"
environment   = "staging"
```

Command:

```
terraform apply
```

Overriding Variable Values

Command-Line Variables: Values passed via the CLI take precedence:

```
terraform apply -var="region=eu-central-1"
```

*Default Variables: If a variable has a default value, it will be used unless overridden by tfvars or CLI.

Example Workflow

File: variables.tf

```
variable "region" {
  description = "AWS region"
  default     = "us-east-1"
}
```

```
variable "instance_type" {
  description = "Type of EC2 instance"
  default     = "t2.micro"
}

variable "environment" {
  description = "Environment (e.g., dev, staging, production)"
}
```

File: main.tf

```
provider "aws" {
  region = var.region
}

resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = var.instance_type

  tags = {
    Name        = "ExampleInstance"
    Environment = var.environment
  }
}
```

File: terraform.tfvars

```
region      = "us-west-2"
instance_type = "t3.micro"
environment   = "dev"
```

Commands

```
terraform init
terraform plan
terraform apply
```

Terraform Data Sources

A data source in Terraform allows you to fetch information from existing infrastructure or external services. This is useful for referencing existing resources instead of creating new ones.

Key Concepts

- ✓ Purpose: Data sources let you read information from external resources (e.g., existing AWS VPCs, AMIs, or security groups) and use that information in your configuration.
- ✓ Usage: Data sources are defined using the data block in Terraform.
- ✓ Read-Only: Data sources don't create or modify resources; they only retrieve information.

1. Basic Syntax

```
data "provider_type_resource" "name" {
  argument1 = "value"
  argument2 = "value"
}
```

2. Common Use Cases

- Fetching an existing resource like an AWS VPC or AMI.
- Referencing external services for dynamic configurations.
- Avoiding hardcoding by dynamically fetching information.

3. Step-by-Step Examples

Example 1: Fetch the Latest AWS AMI

Fetch the latest Amazon Linux 2 AMI to use for an EC2 instance.

```
# Data Source: Fetch the Latest AMI
data "aws_ami" "latest" {
  most_recent = true
  owners      = ["amazon"]

  filter {
    name   = "name"
    values = ["amzn2-ami-hvm-*-x86_64-gp2"]
  }
}

# Resource: Create an EC2 Instance Using the AMI
resource "aws_instance" "example" {
  ami           = data.aws_ami.latest.id
  instance_type = "t2.micro"

  tags = {
    Name = "ExampleInstance"
  }
}
```

Steps:

- Terraform fetches the latest AMI using the data block.
- The AMI ID is dynamically assigned to the aws_instance resource.

Example 2: Fetch an Existing VPC

Fetch the ID of an existing VPC based on its tag.

```
# Data Source: Fetch Existing VPC by Tag
data "aws_vpc" "example" {
  filter {
    name   = "tag:Name"
    values = ["MyVPC"]
  }
}
```

```
# Resource: Create a Subnet in the Existing VPC
resource "aws_subnet" "example" {
  vpc_id     = data.aws_vpc.example.id
  cidr_block = "10.0.1.0/24"

  tags = {
    Name = "MySubnet"
  }
}
```

Steps:

Terraform uses the aws_vpc data source to retrieve the VPC ID.
The subnet is created in the fetched VPC.

Example 3: Fetch an Existing Security Group

Reference an existing security group to attach to a new EC2 instance.

```
# Data Source: Fetch Security Group by Name
data "aws_security_group" "example" {
  filter {
    name   = "group-name"
    values = ["my-security-group"]
  }
}

# Resource: Launch EC2 Instance with the Security Group
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0" # Replace with a valid AMI ID
  instance_type = "t2.micro"
  vpc_security_group_ids = [data.aws_security_group.example.id]

  tags = {
    Name = "InstanceWithExistingSG"
  }
}
```

Steps:

- The data source fetches the security group ID based on its name.
- Terraform uses the ID when creating the EC2 instance.

Example 4: Fetch AWS Regions

Fetch all available AWS regions dynamically.

```
# Data Source: Fetch AWS Regions
data "aws_regions" "all" {}

# Output: List of Regions
output "available_regions" {
  value = data.aws_regions.all.names
}
```

Steps:

- The aws_regions data source retrieves all available regions.
- The output lists the regions.

Combining Data Sources with Variables

You can use data sources with variables to make configurations more dynamic.

Example: Fetch a VPC by Environment

```
# Variable: Define the Environment
variable "environment" {
  default = "dev"
}

# Data Source: Fetch VPC by Environment Tag
data "aws_vpc" "example" {
  filter {
    name   = "tag:Environment"
    values = [var.environment]
  }
}

# Resource: Create a Subnet in the Fetched VPC
resource "aws_subnet" "example" {
  vpc_id     = data.aws_vpc.example.id
  cidr_block = "10.0.1.0/24"

  tags = {
    Name = "${var.environment}-subnet"
  }
}
```

Debugging Data Sources

Plan the Configuration: Use terraform plan to verify that the data source fetches the correct resource.

Enable Debug Logging: Add the TF_LOG environment variable to debug:

```
export TF_LOG=DEBUG
terraform apply
```

Output Data Source Values: Use outputs to inspect values fetched by a data source:

```
output "vpc_id" {
  value = data.aws_vpc.example.id
}
```

Terraform Modules

A module in Terraform is a container for multiple resources that are used together. Modules allow you to organize, reuse, and share your Terraform configuration effectively.

Why Use Modules?

- ✓ Reusability: Write your configuration once and reuse it across projects.
- ✓ Maintainability: Simplify your main Terraform configuration by organizing it into logical components.
- ✓ Collaboration: Share modules with other teams or across environments.
- ✓ Consistency: Enforce standards and best practices.

1. Anatomy of a Module

A module typically consists of:

- main.tf: Contains the main resource definitions.
- variables.tf: Defines input variables.
- outputs.tf: Exposes outputs from the module.
- Optional files like providers.tf to define providers.

2. Module Directory Structure

A module can be structured as follows:

```
modules/
├── vpc/
│   ├── main.tf       # VPC and related resources
│   ├── variables.tf   # Input variables
│   ├── outputs.tf     # Outputs for the module
├── ec2/
│   ├── main.tf       # EC2 instance and related resources
│   ├── variables.tf
│   ├── outputs.tf
```

3. Creating and Using a Module

<u>Create a Simple Module</u>

Create a modules/vpc directory with the following files:

File: modules/vpc/main.tf

```
resource "aws_vpc" "example" {
  cidr_block         = var.cidr_block
  enable_dns_support   = true
  enable_dns_hostnames = true

  tags = {
    Name = var.name
  }
}
```

File: modules/vpc/variables.tf

```
variable "cidr_block" {
  description = "The CIDR block for the VPC"
}

variable "name" {
  description = "The name of the VPC"
}
```

File: modules/vpc/outputs.tf

```
output "vpc_id" {
  value = aws_vpc.example.id
}
```

Use the Module in Your Root Module

Create a main.tf file in your root directory to use the module:

File: main.tf

```
provider "aws" {
  region = "us-east-1"
}

module "vpc" {
  source    = "./modules/vpc"
  cidr_block = "10.0.0.0/16"
  name      = "MyVPC"
}

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

Apply the Configuration

```
terraform init
terraform plan
terraform apply
```

Module Inputs and Outputs

Modules accept inputs (via variables) and return outputs.

Input Variables

- Defined in variables.tf.
- Passed using the module block.

```
module "vpc" {
  source    = "./modules/vpc"
  cidr_block = "10.0.0.0/16"
  name      = "ProductionVPC"
}
```

Outputs

- Defined in outputs.tf.
- Accessed using module.<module_name>.<output_name>.

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

Best Practices for Modules

- ✓ Use Descriptive Names:
  Name your variables, outputs, and modules descriptively for clarity.
- ✓ Default Values for Variables:
  Provide defaults for frequently used variables.

```
variable "cidr_block" {
  default = "10.0.0.0/16"
}
```

- ✓ Document Your Module:
  Add a README.md to explain how to use the module.

- ✓ Separate Environments:
  Use modules for different environments like dev, staging, and prod.

```
environments/
├── dev/
│   └── main.tf
├── staging/
│   └── main.tf
├── prod/
│   └── main.tf
```

- ✓ Use Version Control:
  Store reusable modules in a Git repository.

Advanced Module Usage(Nested Modules)

Modules can call other modules to create a hierarchy.

Example:

```
module "vpc" {
  source = "./modules/vpc"
  cidr_block = "10.0.0.0/16"
  name       = "NestedVPC"
}

module "subnets" {
  source  = "./modules/subnets"
  vpc_id  = module.vpc.vpc_id
  subnets = ["10.0.1.0/24", "10.0.2.0/24"]
}
```

Advanced Module Usage (Use Remote Modules)
Use modules from the Terraform Registry or Git repositories.

Terraform Registry:

```
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "3.5.0"

  name = "MyVPC"
  cidr = "10.0.0.0/16"
}
```

Git Repository:

```
module "vpc" {
  source = "git::https://github.com/my-org/terraform-modules.git//vpc"
}
```

Understanding Terraform State

Terraform state is a key part of how Terraform manages and tracks your infrastructure. It serves as a record of the resources Terraform has created, updated, or destroyed.

1. What is Terraform State?
Definition: A state file (terraform.tfstate) is a JSON file that stores the current state of your infrastructure managed by Terraform.

Purpose:

- ✓ Tracks resources so Terraform knows which resources it manages.
- ✓ Maps Terraform configuration to real-world resources.
- ✓ Optimizes performance by storing metadata about resources.

2. Why is State Important?

- Sync Configuration and Infrastructure: State ensures Terraform knows what exists in the infrastructure.
- Dependency Tracking: Helps Terraform understand relationships between resources.
- Performance Optimization: Speeds up planning and applying configurations.

3. Types of State

- Local State: Stored in the working directory as a terraform.tfstate file (default).
- Remote State: Stored in a remote backend like S3, Consul, or Terraform Cloud. Useful for collaboration.

4. State File Location

- By default, Terraform creates a terraform.tfstate file in the same directory where the terraform apply command is run.

Example:

```
my-terraform-project/
├── main.tf
├── terraform.tfstate
```

5. Managing State

Inspect the State

You can view the current state of your infrastructure with:

```
terraform show
```

List State Resources

List all the resources tracked in the state file:

```
terraform state list
```

Example Output:

```
aws_instance.example
aws_vpc.my_vpc
```

Remove a Resource from State

If a resource is deleted manually (outside of Terraform), you can remove it from the state file without deleting the actual infrastructure:

```
terraform state rm resource_type.resource_name
```

Example:

```
terraform state rm aws_instance.example
```

Import an Existing Resource into State

If a resource exists outside of Terraform but you want Terraform to manage it:

Use the import command:

```
terraform import resource_type.resource_name resource_id
```

Example:

```
terraform import aws_instance.example i-0abcd1234efgh5678
```

Move a Resource Between Modules

If a resource needs to be moved to a different module:

```
terraform state mv source_resource target_resource
```

Example:

```
terraform state mv aws_instance.example module.new_module.aws_instance.example
```

6. Remote State

Remote state allows collaboration and state file storage in a secure, shared location.

Configure Remote State in S3

Example Configuration (main.tf):

```
terraform {
  backend "s3" {
    bucket       = "my-terraform-state"
```

```
  key          = "state/terraform.tfstate"
  region       = "us-east-1"
  encrypt      = true
  dynamodb_table = "terraform-locking"
 }
}
```

Initialize Remote Backend

Run the following command to migrate local state to the remote backend:

```
terraform init
```

*Terraform prompts you to confirm the migration.

7. Common State Commands

View the State:

```
terraform show
```

List Resources:

```
terraform state list
```

Remove a Resource from State:

```
terraform state rm resource_type.resource_name
```

Move Resources:

```
terraform state mv source_resource target_resource
```

Pull Remote State Locally:

```
terraform state pull
```

8. State Locking

State locking ensures that only one user or process can modify the state at a time. This is especially important for remote state backends.

DynamoDB for State Locking: If using S3 as a remote backend, enable state locking with DynamoDB:

```
terraform {
 backend "s3" {
  bucket       = "my-terraform-state"
  key          = "state/terraform.tfstate"
  region       = "us-east-1"
  dynamodb_table = "terraform-locking"
 }
}
```

Best Practices for Managing State

- ✓ Use Remote State:

Store state files in a backend like S3 for collaboration and security.

- ✓ Enable State Locking:
  Prevent concurrent modifications using locking mechanisms like DynamoDB.

- ✓ Secure State Files:
  Encrypt state files to protect sensitive information.

- ✓ Backup State Files:
  Keep backups of local state files for disaster recovery.

- ✓ Avoid Manual Edits:
  Do not manually edit terraform.tfstate files as it can corrupt the state.

- ✓ Use State Commands:
  Use Terraform's built-in state commands for modifications.

Modifying Resources Using Terraform

Terraform provides a structured workflow to modify resources in your infrastructure while ensuring safety and consistency.

1. General Workflow for Modifications
   a) Edit the Configuration: Update the resource definition in the .tf file.
   b) Preview Changes: Use terraform plan to review the changes Terraform will make.
   c) Apply the Changes: Use terraform apply to execute the modifications.

2. Step-by-Step Guide

Edit the Resource

For example, if you have an EC2 instance defined like this:

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

You can modify the instance_type from t2.micro to t3.micro:

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t3.micro"
}
```

Preview the Changes

Run the following command to preview the changes Terraform will make:

```
terraform plan
```

Output Example:

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

Apply the Changes

Run the following command to apply the changes:

```
terraform apply
```

Terraform will prompt you to confirm:

```
Do you want to perform these actions?
  Terraform will perform the following actions:
    ~ aws_instance.example
       instance_type: "t2.micro" => "t3.micro"

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

Type "yes" to continue.
```

*Type yes to proceed.

3. Advanced Scenarios

Changing a Resource That Requires Replacement

Some changes require replacing a resource (e.g., changing a VPC's CIDR block). Terraform indicates such changes with a -/+ symbol.

Example: Changing the ami of an EC2 instance:

```
resource "aws_instance" "example" {
  ami         = "ami-1234567890abcdef0" # New AMI
  instance_type = "t2.micro"
}
```

Plan Output:

```
-/+ aws_instance.example
    ami: "ami-0c55b159cbfafe1f0" => "ami-1234567890abcdef0"
```

Here:

```
-/+ means the resource will be destroyed and recreated.
```

Modifying Only Specific Resources

Use the -target flag to apply changes to specific resources:

```
terraform apply -target=aws_instance.example
```

Conditional Modifications

Use conditional expressions to modify resources based on variables.

```
variable "environment" {
  default = "dev"
}

resource "aws_instance" "example" {
  ami         = "ami-0c55b159cbfafe1f0"
  instance_type = var.environment == "prod" ? "t3.large" : "t2.micro"
}
```

Updating Resources Manually Imported into State

If you manually imported a resource using terraform import and want to modify it:

a) Edit the configuration file to reflect the resource's desired state.
b) Run terraform apply to reconcile the configuration with the actual resource.

4. Debugging Modifications

Inspect Current State: Use terraform state show resource_type.resource_name to inspect the resource's current state:

```
terraform state show aws_instance.example
```

Refresh State: If the resource has been modified outside Terraform, refresh its state:

```
terraform refresh
```

5. Important Considerations

- ✓ Plan Before Applying: Always run terraform plan before applying changes to ensure you're aware of the impact.
- ✓ State Locking: If you're using remote state, ensure state locking is enabled to avoid concurrency issues.
- ✓ Backup State: Always back up your terraform.tfstate file before making major changes.
- ✓ Dependencies: Be mindful of resource dependencies. Terraform automatically determines dependencies, but manual changes can introduce errors.

6. Example: Modifying a Security Group

Original Configuration:

```
resource "aws_security_group" "example" {
 name        = "example-sg"
 description = "Allow SSH inbound"

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

Modified Configuration:

Update the security group to allow HTTP traffic:

```
resource "aws_security_group" "example" {
 name        = "example-sg"
 description = "Allow SSH and HTTP inbound"

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

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

Steps:

```
terraform plan
terraform apply
```

Expected Output:

```
~ aws_security_group.example
    ingress.#: "1" => "2"
```

7. Summary

Modify Resources:

- Edit the .tf file and run terraform plan to preview changes.
- Use terraform apply to implement the changes.

Advanced Techniques:

- Use -target for selective resource updates.
- Use conditionals for dynamic modifications.

Important Commands:

- terraform state show: View the current state of a resource.
- terraform refresh: Sync the state file with actual infrastructure.