Table of Contents

Understanding Code Reusability

- Currently all environments are having same .tf files as code.

```
├── dev
│   ├── backends.tf
│   ├── compute.tf
│   ├── iam.tf
│   ├── dev.tfvars
│   ├── networking.tf
│   ├── providers.tf
│   └── variables.tf
├── qa
│   ├── backends.tf
│   ├── compute.tf
│   ├── iam.tf
│   ├── qa.tfvars
│   ├── networking.tf
│   ├── providers.tf
│   └── variables.tf
├── prod
│   ├── backends.tf
│   ├── compute.tf
│   ├── iam.tf
│   ├── prod.tfvars
│   ├── networking.tf
```

```
|   ├── providers.tf
|   └── variables.tf
```

- Here, the code files containing AWS Resources blocks are de-duplicated.
- Same code file is available in different folders.

Terraform Modules

Overview

- A **Terraform module** is any set of Terraform configuration files in a folder.
- All of the configurations we have written so far have technically been modules,since you deployed them directly, if you run apply directly on a module, it's referred to as a **root module**.
- Modules are the main way to package and reuse resource configurations with Terraform.
- Modules are code structure for multiple resources that are used together.
- A module consists of a collection of **.tf** files kept together in a directory.

The Root Module

- Every Terraform configuration has at least one module, known as its **root module**, which consists of the resources defined in the **.tf** files in the main working directory.

--

Child Modules

- A Terraform module (usually the root module of a configuration) can call other modules to include their resources into the configuration.
- A module that has been called by another module is often referred to as a **child module**.
- **Child modules** can be called multiple times within the same configuration, and multiple configurations can use the same child module.

--

Using Modules

- **Module Blocks** documents the syntax for calling a child module from a parent module.
- **Module Sources** documents what kinds of paths, addresses, and URIs can be used in the source argument of a module block.
- The Meta-Arguments section documents special arguments that can be used with every module, including providers, **depends_on**, **count**, and **for_each**.

Module Blocks

- A module can call other modules, which lets you include the child module's resources into the configuration.

- Modules can also be called multiple times, either within the same configuration or in separate configurations, allowing resource configurations to be packaged and re-used.

Calling a Child Module

- To call a module means to include the contents of that module into the configuration with specific values for its **input variables**.
- Modules are called from within other modules using module blocks:

```
module "servers" {
  source = "./app-cluster"
  servers = 5
}
```

--

- A module that includes a **module** block like above is the calling module of the child module.
- The label immediately after the **module** keyword is a local name, which the calling module can use to refer to this instance of the module.
- Within the block body (between **{ and }**) are the arguments for the module. Module calls use the following kinds of arguments:
    - The **source** argument is mandatory for all modules, Its value is the path to a local directory containing the module's configuration files.
    - The **version** argument is recommended for modules from a registry.
        - Use the **version** argument in the **module** block to specify versions.
    - Most other arguments correspond to **input variables** defined by the module.
        - In above code block, the **servers** argument in the example above is one of these.)

--

```
module "consul" {
  source  = "hashicorp/consul/aws"
  version = "0.0.5"

  servers = 3
}
```

Module Sources

- The source argument in a **module block** tells Terraform where to find the source code for the desired child module.
- For more information on possible values for this argument, see Module Sources

Module Development

Creating Modules

- The **.tf** files in your working directory when you run **terraform plan** or **terraform apply** together form the **root** module.
- That module may **call other modules** and connect them together by passing output values from one to input values of another.

Module structure

- Re-usable modules are defined using all of the same configuration language concepts we use in **root modules**. Most commonly, modules use:
  - **Input variables** to accept values from the calling module.
  - **Output values** to return results to the calling module, which it can then use to populate arguments elsewhere.
  - **Resources** to define one or more infrastructure objects that the module will manage.
- To define a module, **create a new directory** for it and place one or more .tf files inside just as you would do for a root module.
- Terraform can load modules from local relative paths.
- Modules can also call other modules using a module block, but it is recommended keeping the module tree relatively flat and using module composition as an alternative to a deeply-nested tree of modules.

--

When to write a module

- In principle any combination of resources and other constructs can be factored out into a module, but over-using modules can make your overall Terraform configuration harder to understand and maintain, so we recommend moderation.

Standard Module Structure

- The standard module structure is a file and directory layout that is recommended for reusable modules as shown below:
- **Root module**
  - This is the only required element for the standard module structure.
  - Terraform files must exist in the root directory of the repository.
  - This should be the primary entrypoint for the module.
- **main.tf, variables.tf, outputs.tf**
  - These are the recommended filenames for a minimal module, even if they're empty.
  - **main.tf** should be the primary entrypoint. For a simple module, this may be where all the resources are created.
  - **variables.tf and outputs.tf** should contain the declarations for variables and outputs, respectively.
- **Variables and outputs should have descriptions**
- **Nested modules**
  - Nested modules should exist under the **modules/** subdirectory.

--

- When writing terraform code creating one environment, in an ideal scenario you typically need at least two environments: one for your team's internal testing ("staging") and one that real users can access ("production").
- Ideally, the two environments are nearly identical, though you might run slightly different sets of server configuration in staging to save cost.
- To add/create a new environment without having to copy and paste all of the terraform code from staging, a Terraform module can be used and the code files can be reused by the module in multiple places.
- Instead of having the same code copied and pasted in the **development/testing/production** environments, we can have all senvironments reuse code from the same module.
- Modules are the key ingredient to writing reusable, maintainable, and testable Terraform code.

--

- To Create a module of your existing terraform code structure, create a new top-level folder called **modules**
- Open up the **main.tf** file in **modules/infra-services**, and remove the provider definition file.
- Providers should be configured only in root modules and not in reusable modules.
- You can now make use of this module in the qa environment. Here's the syntax for using a module:

```
module "<NAME>" {
  source = "<SOURCE>"

  [CONFIG ...]
}
```

- where **NAME** is an identifier you can use throughout the Terraform code to refer to this module (e.g., **infra_services**)
- **SOURCE** is the path where the module code can be found (i.e **modules/services/infra_services**)
- **CONFIG** consists of arguments that are specific to that module.

--

- For example, you can create a new file in **dev/services/infra_services/main.tf** and use the **infra_services** module in it as follows:

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

module "infra_services" {
  source = "../modules/services/infra_services"
}
```

- You can then reuse the exact same module in the qa environment by creating a new **qa/services/infra_services/main.tf** file with the following contents:

```
provider "aws" {
  region = "us-east-2"
}

module "infra_services" {
  source = "../modules/services/infra_services"
}
```

- Here, the same code is reused in multiple environments with minimal duplication.

> Note that whenever you add a module to your Terraform configurations or modify the source parameter of a module, you need to run the **init** command before you run **plan** or **apply**:

--

```
[ec2-user@ip-172-31-0-125 dev]$ terraform init
Initializing modules...
- webserver_cluster in ../modules/services/infra_services

Initializing the backend...

Initializing provider plugins...

Terraform has been successfully initialized!
```

- Once modules are initialized, any hardcoding of resource names should be avoided in module code, here we need to add configurable inputs to the **infra_services** module so that it can behave differently in different environments.
- **Module inputs**
    - In Terraform, modules can have input parameters.
    - To define them, we can use input variables.
    - Make sure **modules/services/infra_services/variables.tf** has variable definitions for input variables.
    - Also, the Terraform **.tf** files under **modules/services/infra_services/** should use **${var.vpc_cidr}** instead of the hardcoded names.

--

- Now in the stage environment, in **stage/main.tf**, you can set the input variables accordingly:

```
module "infra_services" {
  source = "../modules/services/infra_services"
```

```
  cloud_env = "stage"
  vpc_tag_name = "stage_vpc"
  instance_count = "2"
  instance_type = "t4g.small"
  vpc_cidr = "172.31.0.0/16"
  public_cidr = "172.31.1.0/24"
  private_cidr = "172.31.2.0/24"
}
```

- Similar to above stage environment, in **testing** environment **testing/main.tf**, set the values corresponding to that environment.

```
module "infra_services" {
  source = "../modules/services/infra_services"
  cloud_env = "testing"
  vpc_tag_name = "testing_vpc"
  instance_count = "2"
  instance_type = "t4g.small"
  vpc_cidr = "172.31.0.0/16"
  public_cidr = "172.31.1.0/24"
  private_cidr = "172.31.2.0/24"
}
```

- Here, input variables for a module are set by using the same syntax as setting arguments for a resource.
- The input variables are the API of the module, controlling how it will behave in different environments.

--

- Below is the structure of the code files:

```
[ec2-user@ip-172-31-0-125 stage]$ tree ../modules/
../modules/
└── services
    └── infra_services
        ├── compute.tf
        ├── networking.tf
        ├── outputs.tf
        └── variables.tf

2 directories, 4 files
```

You should also set these variables in the production environment in prod/services/webserver-cluster/main.tf but to different values that correspond to that environment:

Terraform Best Practices

- **Identify what should be declared in variable and resource block?**

    - Code should be generic (reused across environments, regions, accounts)
    - Any variable that will change across environments, regions, accounts should be declared in variables.tf file.
        - Static reference : definition of variables.
        - Dynamic : Resource reference : resource_type.resource_logical_name.id

- **Terraform configurations files separation**

    - **compute.tf** - define data sources to create all compute resources.
    - **networking.tf** - define data sources to create all networking resources.
    - **variables.tf** - contains declarations of variables used in other terraform files.
    - **terraform.tfvars** - contains variables values and should not be used anywhere and set by default.

--

- **Use separate directories for each environment:**

    - Use separate directory for each environment (dev, qa, prod).
    - Each environment directory corresponds to a default Terraform workspace and deploys a version of the service to that environment.

- **Variables Conventions**

    - Declare all variables in **variables.tf**.
    - Provide meaningful description for all variables.
    - Order keys in a variable block like this: **description , type, default**.

--

- **Better Security practices**

    - Use remote state:
        - Never to store the state file on your local machine or version control.
        - With remote state, Terraform writes the state data to a remote data store, which can be shared between all team members. This approach locks the state to allow for collaboration as a team.
        - Configure Terraform backend using remote state (shared locations) services such as Terraform Cloud, Amazon S3, Azure Blob Storage, GCP Cloud Storage.

- **Use Terraform Modules**

    - Use terraform modules for code-reusability

--

Terraform Assignment

- Provision a VPC Network Resources having 2 public subnets and 2 private subnets, IGW attached to VPC, VPC Gateway Endpoint for S3 Service.
- Create an S3 Bucket with sdlc name as prefix.

- Provision RDS Instance in VPC private subnet launched in the previous step (network resources)
- Create IAM Role, Policy and Provision a EC2 instances having this IAM Role attached, that contains IAM Permissions to read and write data to S3 buckets.
    - Validate the data copy from ec2 instance to/from S3 bucket.
    - Validate network to connect with RDS instance.
- Validate the connection to RDS Instance from EC2 instance by executing mysql commands
- Document all steps with AWS Service Screenshots into a Word File.

> Code structure should be re-usable for multiple environment setup

--

Notes

- Use RDS Free tier instance type to avoid cost
- Use Terraform to create above resources.
- Terraform Code used to create above resources should be generic to create multiple environments in Multiple Region/Accounts.
- Use **terraform destroy** once resources are created and tested to avoid cost.