**Terraform**

**IaC tool Specifications:**

AWS Architecture

Version 1.1.1

**BMV SYSTEM INTEGRATION PRIVATE LIMITED**

**Idea... Implementation... Innovation...**

**:: CORPORATE HEAD OFFICE::**

**A503, The First,**

**Behind The ITC Narmada Hotel & Keshavbaug Party Plot,**

**Off 132 ft Road, Vastrapur, Ahmedabad.**

**Gujarat- 380015**

**Phone: +91 (79) 40 30 53 02**

**Website:** [**www.systemintegration.in**](http://www.systemintegration.in)

**Mail:** [**info@systemintegration.in**](mailto:info@systemintegration.in)

Revision History

| **Date** | **Doc Version** | **Author** | **Status Created/Updated/Reviewed/Approved/Verified)**  **In case of updates- Mention what has been updated. give change/issue/req id as ref.** |
| --- | --- | --- | --- |
| 15-12-2023 | 1.1.1 | [Hardik Kalkani](mailto:hardikbk@systemintegration.in) | Terraform |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Table of Content

1. Introduction

1.1 Terraform Cloud

1.2 Terraform Enterprise

1. Why Terraform?
2. Use cases of Terraform
3. Getting Started

4.1 Installation

4.2 Configuration

4.3 Initializing

1. IaC(Infrastructure as Code)

5.1 Variables

5.2 Providers

5.3 Resources

5.4 Arguments and Meta-Arguments

5.5 Modules

1. Commands

6.1 Terraform init

6.2 Terraform plan

6.3 Terraform apply

6.4 Terraform destroy

6.5 Terraform validate

1. Terraform backend

7.1 Backend Configuration

7.2 Available Backends

1. Terraform State

8.1 Purpose

8.2 Locking

8.3 Workspaces

8.4 Remote state

8.5 Sensitive data

1. Best Practices

7.1 File structure

7.2 Workflow

1. Examples

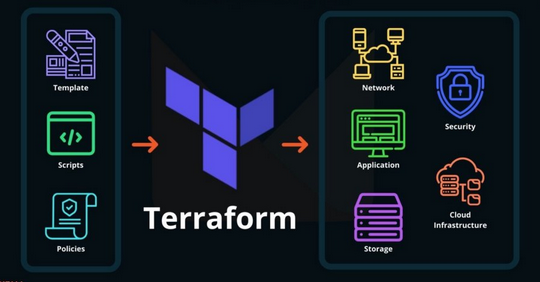
10.1 Create AWS-VPC

1. References

**Terraform**

1. **Introduction**

HashiCorp Terraform is an infrastructure as a code tool that lets you define both cloud and on-prem resources in human-readable configuration files that you can version, reuse, and share. You can then use a consistent workflow to provision and manage all of your infrastructure throughout its lifecycle. Terraform can manage low-level components like computing, storage, and networking resources, as well as high-level components like DNS entries and SaaS features.



**1.1 Terraform Cloud**

Terraform Cloud is a SaaS that we support—instead, when you run Terraform you still could run it on your local machine, but now it saves and retrieves the state file from Terraform Cloud—which is running over here.

**1.2 Terraform Enterprise**

Terraform Enterprise is our self-hosted distribution of [Terraform Cloud](https://developer.hashicorp.com/terraform/cloud-docs). It offers enterprises a private instance of the Terraform Cloud application, with no resource limits and with additional enterprise-grade architectural features like audit logging and SAML single sign-on.

1. **Why Terraform?**

Terraform makes it easy to set up and manage computer systems by writing down what you want, letting you do it the same way every time, and allowing your team to work together smoothly.

There are multiple reasons why Terraform is used over the other IaC tools but below are the main reasons.

* **Multi-Cloud Support:** Terraform is known for its multi-cloud support. It allows you to define infrastructure in a cloud-agnostic way, meaning you can use the same configuration code to provision resources on various cloud providers (AWS, Azure, Google Cloud, etc.) and even on-premises infrastructure. This flexibility can be beneficial if your organization uses multiple cloud providers or plans to migrate between them.
* **Large Ecosystem:** Terraform has a vast ecosystem of providers and modules contributed by both HashiCorp (the company behind Terraform) and the community. This means you can find pre-built modules and configurations for a wide range of services and infrastructure components, saving you time and effort in writing custom configurations.
* **Declarative Syntax:** Terraform uses a declarative syntax, allowing you to specify the desired end-state of your infrastructure. This makes it easier to understand and maintain your code compared to imperative scripting languages.
* **State Management:** Terraform maintains a state file that tracks the current state of your infrastructure. This state file helps Terraform understand the differences between the desired and actual states of your infrastructure, enabling it to make informed decisions when you apply changes.
* **Plan and Apply:** Terraform's "plan" and "apply" workflow allows you to preview changes before applying them. This helps prevent unexpected modifications to your infrastructure and provides an opportunity to review and approve changes before they are implemented.
* **Community Support:** Terraform has a large and active user community, which means you can find answers to common questions, troubleshooting tips, and a wealth of documentation and tutorials online.
* **Integration with Other Tools:** Terraform can be integrated with other DevOps and automation tools, such as Docker, Kubernetes, Ansible, and Jenkins, allowing you to create comprehensive automation pipelines.
* **HCL Language:** Terraform uses HashiCorp Configuration Language (HCL), which is designed specifically for defining infrastructure. It's human-readable and expressive, making it easier for both developers and operators to work with.

1. **Use cases**

Terraform helps set up and manage computer stuff in the cloud. Imagine you want to create a virtual space to store your files. Instead of clicking around on the cloud provider's website every time, you write down the steps in Terraform. Now, every time you need that space, you just tell Terraform, and it creates it for you, saving time and ensuring it's always the same.

Here are some best use cases of Terraform

* **Cloud Infrastructure Management:**
  + Easily create and manage resources on different cloud platforms.
* **Multi-Environment Setup:**
  + Define and manage various environments like development, testing, and production.
* **Reusable Components:**
  + Create reusable infrastructure components for efficient code organization.
* **Application Deployment:**
  + Deploy software applications alongside necessary infrastructure.
* **Environment Reproducibility:**
  + Recreate environments easily for testing, debugging, or scaling purposes.
* **Infrastructure as Code (IaC):**
  + Implement IaC best practices for version control and collaboration.
* **Collaborative Development:**
  + Work collaboratively with teams using Terraform Cloud's shared workspace.
* **Automated Compliance:**
  + Enforce access controls, policies, and audit logs with Terraform Enterprise.
* **Hybrid Cloud Deployments:**
  + Manage both on-premises and cloud infrastructure seamlessly.
* **Cloud Migration:**
  + Facilitate the migration of applications and infrastructure to the cloud.

1. **Getting Started**

**4.1 Installation**

Installing Terraform is the first step to start using it on your computer. Follow these simple steps:

1. **Download:**
   * Download the appropriate version for your operating system, you can help with the official terraform site according to your OS system. (Windows, macOS, or Linux).
2. **Installation:**
   * For Windows, extract the downloaded ZIP file and move the executable to a directory in your system's PATH.
   * For macOS and Linux, extract the downloaded ZIP file and move the executable to a directory in your system's PATH.
3. **Verify Installation:**
   * Open a new terminal or command prompt.
   * Type terraform --version and press Enter.
   * If installed correctly, you'll see the Terraform version information.

**4.2 Configuration**

Once Terraform is installed, you need to configure it with your credentials and settings:

1. **Create Configuration Files:**
   * Start by creating a new directory for your Terraform project.
   * Inside the directory, create a file named main.tf (Terraform configuration file).
2. **Define Provider:**
   * In main.tf, specify the cloud provider you're using (e.g., AWS, Azure) and configure the necessary provider settings.
3. **Credentials:**
   * If required, set up credentials or authentication information. This might involve environment variables, configuration files, or other methods depending on the provider.

**4.3 Initializing**

After configuring your Terraform project, the next step is to initialize it:

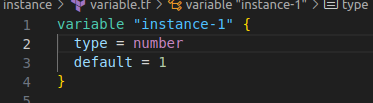
1. **Open Terminal/Command Prompt:**
   * Navigate to the directory where your Terraform configuration files are located.
2. **Run Initialization:**
   * Type terraform init and press Enter.
   * Terraform initializes the project, downloading necessary plugins and setting up the working directory.
3. **Review Output:**
   * Check the initialization output for any errors or warnings.
   * If successful, you're ready to move on to other Terraform commands like Terraform apply to create or update infrastructure.
4. **IaC(Infrastructure as Code)**

Managing and provisioning IT infrastructure through code, treating infrastructure configurations as software. Terraform, as an IaC tool, simplifies this process with its declarative syntax, version control integration, modular design, multi-cloud support, and effective state management.

In Terraform we can show many components, which are explained below

**5.1 Variables**

A Terraform variable is a parameter that allows dynamic input in configurations, providing flexibility and reusability. Variables help customize infrastructure configurations by allowing users to input values at runtime. They enhance the adaptability of Terraform code, making it easier to manage various environments or adjust configurations without modifying the underlying code.

* ****

There are 6 types of variables

* + **String:** A string variable is a parameter that holds text or characters, enabling the dynamic configuration of string values in infrastructure code.
  + **Number:** A number variable is a parameter used to represent numeric values in infrastructure configurations, allowing for dynamic and adaptable numeric input.
  + **List:** A list variable is a parameter that stores an ordered collection of values, enabling the dynamic configuration of lists or arrays in infrastructure code.
  + **Bool:** A boolean variable is a parameter that holds a true or false value, allowing for dynamic configuration of Boolean conditions in infrastructure code.
  + **Object:** An object variable is a parameter that encapsulates a set of related attributes or key-value pairs, providing a structured way to define and manage complex configurations in infrastructure code.
  + **Map:** a map variable is a parameter that represents an associative array or dictionary, allowing the dynamic configuration of key-value pairs for versatile and structured data input in infrastructure code.

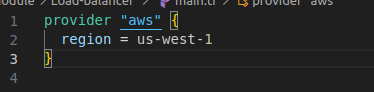
**5.2 Providers**

Providers are a logical abstraction of an upstream API. They are responsible for understanding API interactions and exposing resources. Providers allow Terraform to interact with cloud providers, SaaS providers, and other APIs.

**Examples -** AWS, Azure, Google Cloud Platform, Kubernetes, etc..

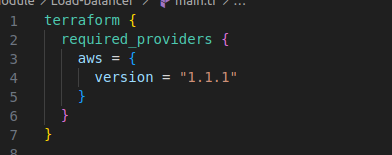
**5.2.1 Provider Configuration**

* Terraform configurations must declare which providers they require so that Terraform can install and use them.



**5.2.2 Requiring Providers**

* Each Terraform module must declare which providers it requires so that Terraform can install and use them.



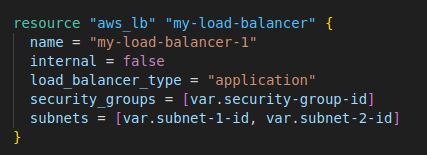
**5.3 Resources**

Resources are the most important element in the Terraform language. Each resource block describes one or more infrastructure objects, such as virtual networks, compute instances, or higher-level components such as DNS records.

**Examples-** AWS\_ec2\_instance, AWS\_s3\_bucket, AWS\_vpc, etc..

**5.3.1 Resources blocks**

* A "resource" block declares a resource of a specific type with a specific local name. The name refers to this resource in the same Terraform module but has no meaning outside that module's scope.
* Here “Aws\_lb” is a resource of AWS providers, and in this block, we declared arguments



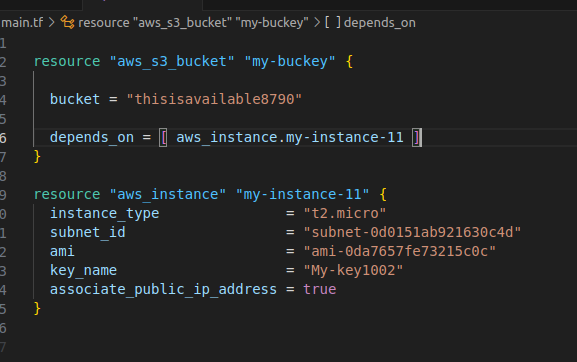
**5.4 Arguments and Meta-Arguments**

**5.4.1 Arguments**

* Arguments in Terraform: Arguments in Terraform are parameters or configuration options used within the Terraform code to customize the behaviour of resources or modules.
* **Examples-** name, internal, load\_balancer\_type, etc.

**5.4.2 Meta-Arguments**

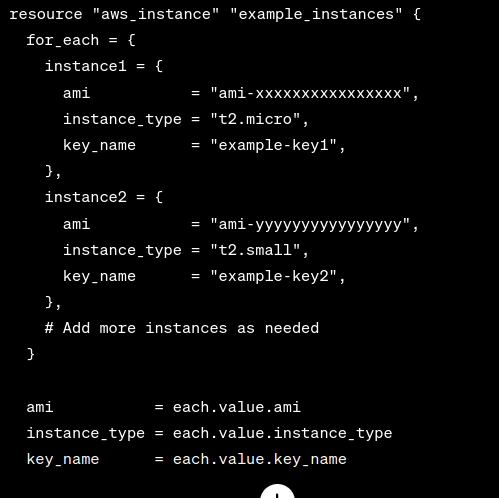
1. **Depends\_on**

* Use the depends\_on meta-argument to handle hidden resource or module dependencies that Terraform cannot automatically infer. You only need to explicitly specify a dependency when a resource or module relies on another resource's behavior but does not access any of that resource's data in its arguments.
* It helps when a resource depends on another resource when you need to create the first dependable resource before the current resource
* 
  + In this code, instance creates first then bucket creates.

1. **Count**

* count is a meta-argument defined by the Terraform language. It can be used with modules and with every resource type.
* When you need the same resource, multiple times in a structure
  + Ex. When you write “count = 5” in aws\_ec2\_instance it means terraform creates the same instance 5 times
  + You can use count in multiple ways
  + 
  + 
  + 

1. **For\_each**

* The for\_each meta-argument accepts a map or a set of strings and creates an instance for each item in that map or set. Each instance has a distinct infrastructure object associated with it, and each is separately created, updated, or destroyed when the configuration is applied.
* it helps when you create the same resource multiple times with the help of objects
  + Firstly create an object then using for\_each, you can create multiple instances, in which data already exist in an object
  + 

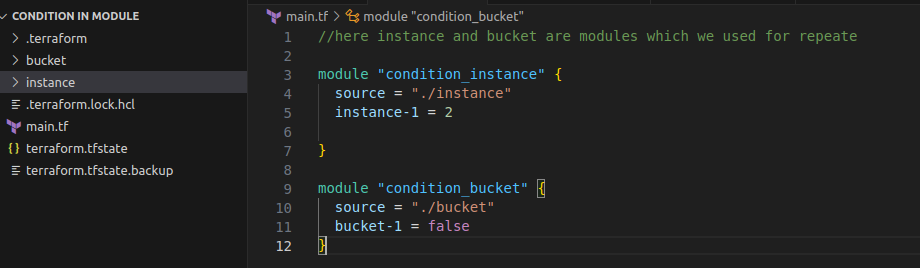
1. **Life cycle**

* **Create\_before\_destroy** - The create\_before\_destroy meta-argument changes this behavior so that the new replacement object is created first, and the prior object is destroyed after the replacement is created.
* **Prevent\_destroy** - This can be used as a measure of safety against the accidental replacement of objects that may be costly to reproduce, such as database instances. However, it will make certain configuration changes impossible to apply and will prevent the use of the terraform destroy command once such objects are created, and so this option should be used sparingly.
* **ignore\_changes** - The ignore\_changes feature is intended to be used when a resource is created with references to data that may change in the future but should not affect said resource after its creation.
* **Replace\_triggered\_by** - replace\_triggered\_by allows only resource addresses because the decision is based on the planned actions for all of the given resources. Plain values such as local values or input variables do not have planned actions of their own, but you can treat them with a resource-like lifecycle by using them with the terraform\_data resource type

**5.5 Modules**

Modules are containers for multiple resources that are used together. A module consists of a collection of .tf and/or .tf.json files kept together in a directory.

It helps us to easily understand big projects and helps when we need the same code repeated times

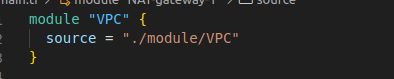


In this code, you can create 2 instances on a single apply, and bucket and instance are other modules

* **Source type -** Local files, Terraform registry(You can take or upload already created module files), Github, Bitbucket, S3 bucket

**5.4.1 Root Module**

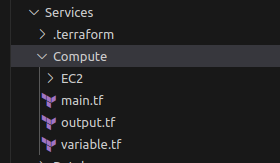
Every Terraform configuration has at least one module, known as its *root module*, which consists of the resources defined in the .tf files in the main working directory.

* **

**5.4.2 Child Module**

A module that has been called another module is often referred to as a child module

We can show this in the picture, we have an EC2 module which is the child module of the Compute module, and the compute module is the child module of the Services module

* 

1. **Commands**

**6.1 Terraform init [options]**

This command performs several different initialization steps to prepare the current working directory for use with Terraform.

During init, the root configuration directory is consulted for backend configuration and the chosen backend is initialized using the given configuration settings.

* 

**6.2 Terraform plan [options]**

The Terraform plan command creates an execution plan, which lets you preview the changes that Terraform plans to make to your infrastructure.

* [**P**](https://developer.hashicorp.com/terraform/cli/commands/plan#planning-modes)**lanning Modes**: There are some special alternative planning modes that you can use for some special situations where your goal is not just to change the remote system to match your configuration.
* **Planning Options**: Alongside the special planning modes, there are also some options you can set to customize the planning process for unusual needs.
  + **Resource Targeting** is one particular special planning option that has some important caveats associated with it.
* **Other Options**: These change the behavior of the planning command itself, rather than customizing the content of the generated plan.
* 

**6.3 Terraform apply [options] [plan file]**

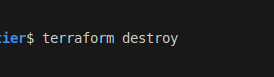
This command executes the actions proposed in a Terraform plan.

When you run without passing a saved plan file, Terraform automatically creates a new execution plan as if you had runthe Terraform plan, prompts you to approve that plan, and takes the indicated actions. You can use all of the planning modes and planning options to customize how Terraform will create the plan.



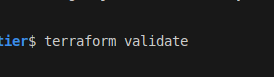
**6.4 Terraform destroy**

The Terraform destroy command terminates resources managed by your Terraform project. This command is the inverse of Terraform apply in that it terminates all the resources specified in your Terraform state. It does not destroy resources running elsewhere that are not managed by the current Terraform project.



**6.5 Terraform validate**

Validate runs checks that verify whether a configuration is syntactically valid and internally consistent, regardless of any provided variables or existing state. It is thus primarily useful for the general verification of reusable modules, including the correctness of attribute names and value types.

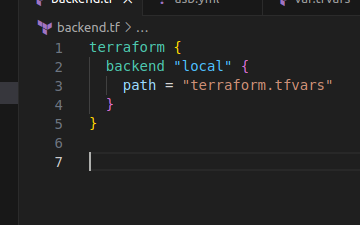


1. **Terraform Backends**

**7.1 Backend Configuration**

* A backend defines where Terraform stores its state data files.
* Terraform uses persisted state data to keep track of the resources it manages. Most non-trivial Terraform configurations either integrate with Terraform Cloud or use a backend to store state remotely.

You do not need to configure a backend when using Terraform Cloud because Terraform Cloud automatically manages the state in the workspaces associated with your configuration. If your configuration includes a cloud block, it cannot include a backend block.



* A configuration can only provide one backend block.

When you change a backend's configuration, you must run terraform init again to validate and configure the backend before you can perform any plans, applies, or state operations.

You can change your backend configuration at any time. You can change both the configuration itself as well as the type of backend.

Terraform will automatically detect any changes in your configuration and request a reinitialization. As part of the reinitialization process, Terraform will ask if you'd like to migrate your existing state to the new configuration. This allows you to easily switch from one backend to another.

If you no longer want to use any backend, you can simply remove the configuration from the file. Terraform will detect this like any other change and prompt you to reinitialize.

**7.2 Available Backends**

* **Local-** The local backend stores state on the local filesystem, locks that state using system APIs, and performs operations locally.
* **Consul**- Stores the state in the Hashicorp Consul KV store at a given path.
* **Http-** Stores the state using a simple REST client.
  + State will be fetched via GET, updated via POST, and purged with DELETE. The method used for updating is configurable.
* **Kubernetes-** Stores the state in a Kubernetes secret.
* **PG-** Stores the state in a Postgres database version 10 or newer.
* **S3/ azurerm/ cos/ gcs/ oss-** Store the state in a different Clouds.

1. **Terraform State**

**8.1 Purpose**

* **Mapping to the real world**
* **Metadata and Dependencies** - Alongside the mappings between resources and remote objects, Terraform must also track metadata such as resource dependencies.
* **Performance -** Terraform stores a cache of the attribute values for all resources in the state. This is the most optional feature of Terraform state and is done only as a performance improvement.
* **Syncing -** Terraform stores the state in a file in the current working directory where Terraform was run. This is okay for getting started, but when using Terraform in a team it is important for everyone to be working with the same state so that operations will be applied to the same remote objects.

**8.2 Locking**

* State locking happens automatically on all operations that could write state. You won't see any message that it is happening. If state locking fails, Terraform will not continue. You can disable state locking for most commands with the -lock flag but it is not recommended.
* If you unlock the state when someone else is holding the lock it could cause multiple writers. Force unlock should only be used to unlock your own lock in a situation where automatic unlocking fails.

**8.3 Workspaces**

* The persistent data stored in the backend belongs to a workspace. The backend initially has only one workspace containing one Terraform state associated with that configuration. Some backends support multiple named workspaces, allowing multiple states to be associated with a single configuration. The configuration still has only one backend, but you can deploy multiple distinct instances of that configuration without configuring a new backend or changing authentication credentials.

**8.4 Remote state**

* With *remote* state, Terraform writes the state data to a remote data store, which can then be shared between all members of a team. Terraform supports storing state in Terraform Cloud, HashiCorp Consul, Amazon S3, Azure Blob Storage, Google Cloud Storage, Alibaba Cloud OSS, and more.
* Remote state is implemented by a backend or by Terraform Cloud, both of which you can configure in your configuration's root module.

**8.5 Sensitive data**

* Terraform state can contain sensitive data, depending on the resources in use and your definition of "sensitive." The state contains resource IDs and all resource attributes. For resources such as databases, this may contain initial passwords.
* Storing the state remotely can provide better security. As of Terraform 0.9, Terraform does not persist state to the local disk when the remote state is in use, and some backends can be configured to encrypt the state data at rest.

**9. Best Practices**

**9.1 File structure**

**Main.tf** - will contain the main set of configurations for your module. You can also create other configuration files and organize them however makes sense for your project.

**variable.tf** - will contain the variable definitions for your module. When your module is used by others, the variables will be configured as arguments in the module block. Since all Terraform values must be defined, any variables that are not given a default value will become required arguments. Variables with default values can also be provided as module arguments, overriding the default value.

**output.tf** - will contain the output definitions for your module. Module outputs are made available to the configuration using the module, so they are often used to pass information about the parts of your infrastructure defined by the module to other parts of your configuration.

**.terraform.lock.hcl** - this is the lock file for our project when multiple users are working on the same project, this file gives permissions to only one user at a time, others are waiting.

**.terraform.tfstate** - It shows the current condition of infrastructure (after the condition of the last use of terraform applied)

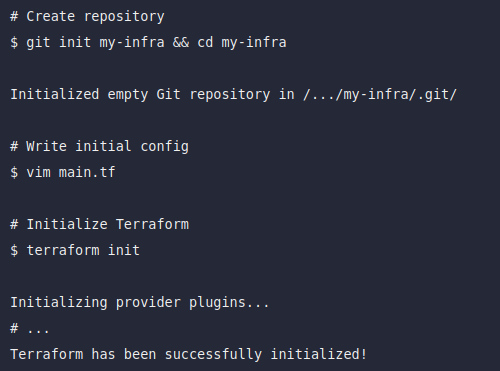
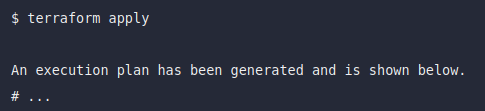
**.terraform.tfstate.backup** - It is a backup file, it helps when the current file is lost or corrupt

* terraform.tfstate and terraform.tfstate.backup: These files contain your Terraform state, and are how Terraform keeps track of the relationship between your configuration and the infrastructure provisioned by it.

**\*.tfvars** - Since module input variables are set via arguments to the module block in your configuration, you don't need to distribute any \*.tfvars files with your module, unless you are also using it as a standalone Terraform configuration.

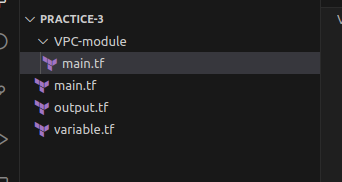
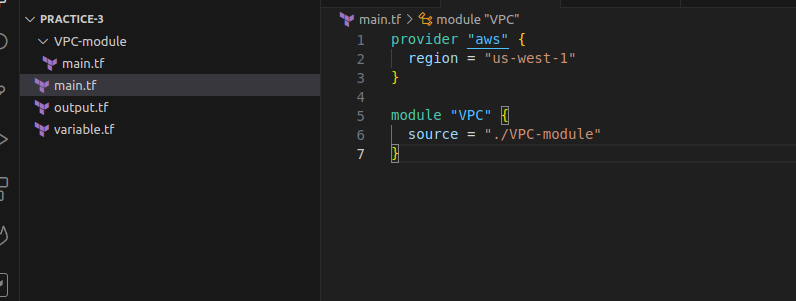
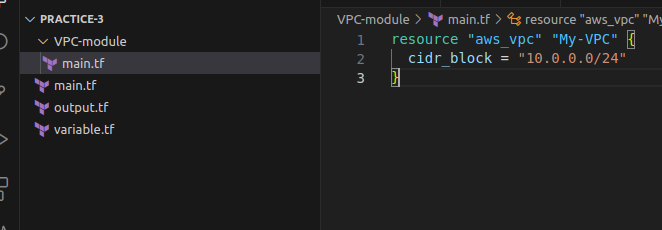
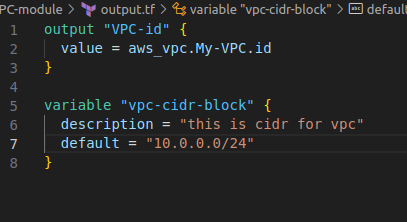
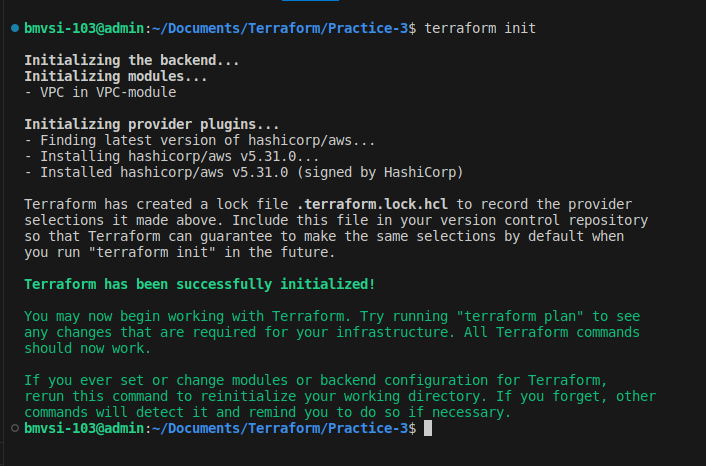
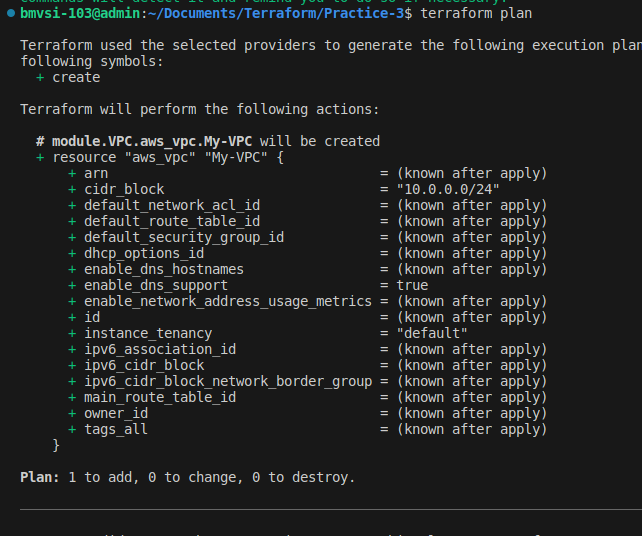
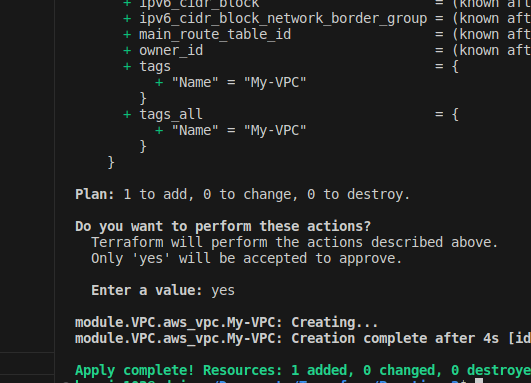
**9.2 Workflow**

The core Terraform workflow has three steps:

1. **Write** - Author infrastructure as code.
   * + - 1. You write Terraform configuration just like you write code: in your editor of choice. It's common practice to store your work in a version-controlled repository even when you're just operating as an individual.
         2. 
2. **Plan** - Preview changes before applying.
   * + - 1. When the feedback loop of the Write step has yielded a change that looks good, it's time to commit your work and review the final plan.
         2. Because Terraform Apply will display a plan for confirmation before proceeding to change any infrastructure, that's the command you run for final review.
         3. 
3. **Apply** - Provision of reproducible infrastructure.
   * + - 1. After one last check, you are ready to tell Terraform to provision real infrastructure.

**10. Examples**

**10.1 Create AWS-VPC**

* **Write a code**
  + Create module structure for vpc
    - 
    - In this picture we can show, that we created the main.tf, variable.tf, and output.tf global files, and create a local main.tf in VPC-module
  + Write code for provider and modules in the global main.tf
    - 
  + Create VPC in local main.tf file, using module we call this file to main global main.tf
    - 
  + Variables and outputs
    - 
* **Planning** 
  + Go to the terminal and write “terraform init”
    - 
  + After successfully initialization, write “terraform plan”
    - 
  + After the right planning, we are ready to apply this code to our AWS-cloud
* **Apply**
  + For the apply code to the cloud, write “terraform apply”
    - Accept “yes”, and it gives us the VPC structure in AWS
    - 

1. **References** 
   1. **Terraform hashicorp -** [developer.hashicorp.com](https://developer.hashicorp.com/)