QA Terraform in Practice

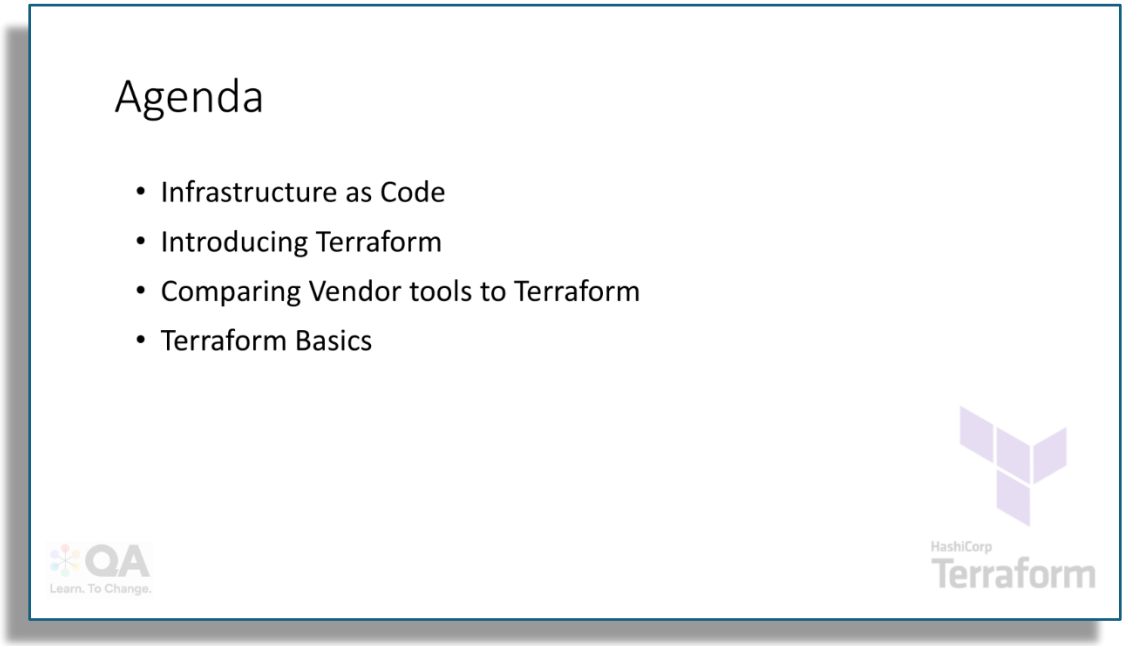
Terraforming Cloud Platforms

0. Hashicorp Terraform certification

Although not been developed specifically as an exam preparation course, the learning outcomes of this 3-day course loosely aligns with the Hashicorp Terraform certification exam “Terraform Associate (003)”.

Details of the exam can be found here

1. Introduction to Infrastructure as Code



Infrastructure as Code

In the traditional datacentre, infrastructure was built by physically visiting the location, installing the necessary hardware, and manually configuring the software. However, with the evolution of software defined datacenters and the rise of public service offerings, a novel approach has emerged. Now, the physical hardware is still present, but it is abstracted and managed as a software-defined service layer.

This software-defined layer, whether for networking, storage, or compute, allows us to treat IT infrastructure like any other software component. You can build, configure, and manipulate it programmatically, just as you work with other software applications. This shift has led to the emergence of "The Cloud," which can be private, residing in your own or someone else's datacentre, or public, available for general consumption.

To efficiently manage this infrastructure, you can embrace the concept of Infrastructure as Code (IaC). By manipulating the infrastructure through code, you can automate and streamline the provisioning, configuration, and management of your cloud-based or datacentre resources. This approach allows you to define your infrastructure as programmable code, bringing the benefits of automation, version control, and consistency to infrastructure management.

By adopting Infrastructure as Code (IaC), you gain the ability to define, test, and deploy your infrastructure in a consistent and repeatable manner. This eliminates the risk of human errors that can occur when manually installing and configuring servers. Instead, you can use a unified definition for your infrastructure, ensuring, for example, that ServerA and ServerB match each other precisely. This allows you to easily redeploy your infrastructure with a repeatable configuration, following secure and controlled practices.

One of the significant advantages of IaC is its disposability. Once a server has served its purpose and is no longer needed, you can dispose of it. However, this does not mean it goes to waste. With IaC, you can consistently recreate the server from the same code whenever you require it again, whether it is in a week or several months later.

It is important to note that the benefits of IaC apply not just to servers but to diverse types of infrastructure, including networking, servers, storage, containers, serverless functions, and other managed services. IaC enables you to bring automation, consistency, and repeatability to the deployment and management of diverse infrastructure components.

Introducing Terraform

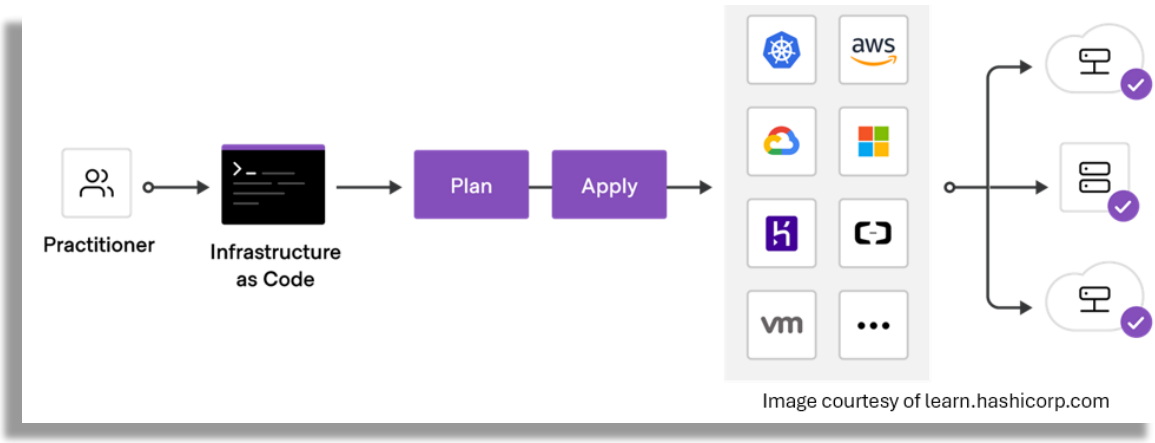


Image courtesy of learn.hashicorp.com

Terraform from Hashicorp is a leading vendor-independent IaC tool. It allows you to define infrastructure both on-premises and in the cloud, and it is not limited to a particular vendor's proprietary tool or language. This flexibility provides a significant advantage for organizations working in the cloud that adopt a multi-cloud strategy or wish to avoid vendor lock-in, making it easier to transition between different cloud platforms in the future.

Terraform provides configuration files in an easily readable format. While it can support JSON, it also has its own domain-specific language called HashiCorp Configuration Language (HCL), which offers a more human-readable syntax. This makes it simpler for teams to

understand and work with Terraform configurations, enhancing collaboration and reducing the learning curve.

The configuration files created with Terraform can be easily stored, shared, tested, deployed, and reused. This promotes efficient collaboration among team members, enables version control and code reuse, and simplifies the management of infrastructure configurations. As your enterprise grows, Terraform can scale alongside it, providing a consistent and scalable approach to managing infrastructure resources.

Comparing Vendor Tools to Terraform

When comparing vendor-specific IaC tools to Terraform, each cloud vendor offers their own set of capabilities, integrations, and services specific to their platform. These vendor-specific tools may have unique features and deep integrations with the vendor's services, allowing for fine-grained control and optimized utilization of their offerings. It is essential to consider the specific requirements of your project and the features provided by the respective vendor-specific IaC tools when evaluating them against Terraform.

AWS CloudFormation

AWS CloudFormation is a service that enables you to define and manage your AWS infrastructure using declarative templates written in JSON or YAML. By embracing the concept of Infrastructure as Code, CloudFormation allows you to model resources such as compute instances, networking, storage, and security as code. These templates describe the desired state of your infrastructure, and CloudFormation automates provisioning, updates, and deletions, simplifying the management of resources across multiple AWS services. Logical groupings of resources, called stacks, allow for centralized management and easy modifications, updates, or deletions as a single unit.

CloudFormation offers advanced features like change management and drift detection, ensuring that resources remain consistent with their defined state while allowing updates via template changes. It integrates seamlessly with AWS services like IAM, CloudTrail, and Service Catalog, providing extensibility through custom resource types and third-party extensions via AWS Lambda. To enhance reliability, CloudFormation includes automatic rollback capabilities, enabling safe and consistent deployments by reverting changes in case of errors during updates.

AWS CloudFormation manages state by maintaining a record of the current configuration of your infrastructure, known as the stack state. When you create, update, or delete a stack, CloudFormation tracks the status of each resource and ensures the actual state of your resources aligns with the desired state defined in the template. This state management enables automated updates, drift detection to identify changes made outside of CloudFormation, and rollback capabilities to revert to a known good state if an operation fails. By abstracting state management, CloudFormation simplifies infrastructure lifecycle management and ensures consistency across deployments.

However, CloudFormation comes with challenges. The service has a steep learning curve, requiring users to understand its template syntax and intricacies. As deployments scale, templates can become complex and harder to maintain, necessitating careful planning and organization. Additionally, while CloudFormation supports most AWS services, some newer or less common services may have limited or delayed support, requiring manual workarounds or alternative solutions.

Azure ARM Templates

Azure Resource Manager (ARM) templates are Microsoft's infrastructure-as-code solution for Azure, enabling users to define and deploy resources in a declarative manner. Written in JSON, ARM templates allow you to describe the desired state of your infrastructure, specifying resources, dependencies, and configurations. Azure Resource Manager orchestrates the provisioning and management of these resources based on the template, streamlining deployments, and ensuring consistency across environments.

ARM templates support advanced features like parameterization and variables, which make templates reusable and customizable for different scenarios. They also include template functions and expressions to enable dynamic values, calculations, and conditional logic. Resources are organized into logical resource groups, allowing for collective management, updates, or deletions of related resources. ARM templates integrate seamlessly with Azure services like Azure Policy for governance, Azure Key Vault for secure credential storage, and Azure DevOps for CI/CD pipelines. Pre-built templates for common use cases provide additional flexibility and a faster starting point.

While ARM templates offer significant benefits, they come with challenges. For complex deployments, the JSON structure can become cumbersome, requiring careful design to maintain readability and manage intricate resource relationships. ARM templates are Azure-specific, limiting portability across cloud providers. Additionally, there is a learning curve associated with mastering their syntax and features, particularly for inexperienced users.

Unlike Terraform, ARM templates do not maintain an explicit state file. Instead, Azure Resource Manager internally tracks the state of resources during provisioning, ensuring they align with the desired state defined in the template. Users can monitor deployment history and resource changes through the Azure portal, CLI, or PowerShell. For extended state tracking and configuration management, Azure offers complementary tools like Azure Automation State Configuration and Azure Policy to manage compliance and governance.

Google Cloud Deployment Manager

Google Cloud Deployment Manager (DM) is an infrastructure-as-code service from Google Cloud Platform (GCP) that allows you to define, configure, and deploy resources using YAML or Python templates. These templates enable you to describe the desired state of your infrastructure in a declarative and version-controlled manner, promoting automation and reproducibility. Deployment Manager

supports a wide range of GCP resources, such as compute instances, networks, storage, and databases, and handles the orchestration required to create, update, or delete these resources.

DM organizes resources into logical groups called deployments, which are managed as a single unit. These deployments simplify the collective management of resources, allowing for consistent updates, modifications, or deletions. The service integrates seamlessly with GCP features like IAM roles, Cloud Storage, and Cloud SQL, enabling efficient resource provisioning. Additionally, DM supports templating and parameterization using Jinja2, allowing for reusable and customizable deployments across different environments.

While DM offers benefits like versioning, auditing, and rollback mechanisms, it presents challenges. The service has a learning curve, requiring familiarity with GCP concepts and template syntax. Managing complex deployments can become intricate, requiring careful template design and organization. Furthermore, DM is specific to GCP, limiting portability to other cloud platforms without significant modifications.

State management in DM is handled internally, tracking the configuration and status of deployments and resources. It performs atomic and incremental updates, ensuring deployments are consistent and avoiding intermediate states during changes. Rollback capabilities enable DM to revert to previous states in case of failures, and synchronization ensures the actual state of resources aligns with the desired configuration. However, unlike Terraform, DM does not expose state files directly to users, as state tracking is entirely managed by the service.

Terraform Basics

Terraform simplifies infrastructure management by breaking it down into a clear lifecycle. The process starts with defining the **scope**, where users identify their goals, such as the type of infrastructure they want to build, and the resources needed. This clarity ensures that the subsequent steps align with organizational objectives.

Once the scope is clear, the next step is to **install** Terraform. As a lightweight tool, Terraform is easy to set up across various operating systems. After installation, users move to the **write** phase, where they define their infrastructure as workflow blocks using HashiCorp Configuration Language (HCL). This allows for modular, reusable, and easily version-controlled configurations.

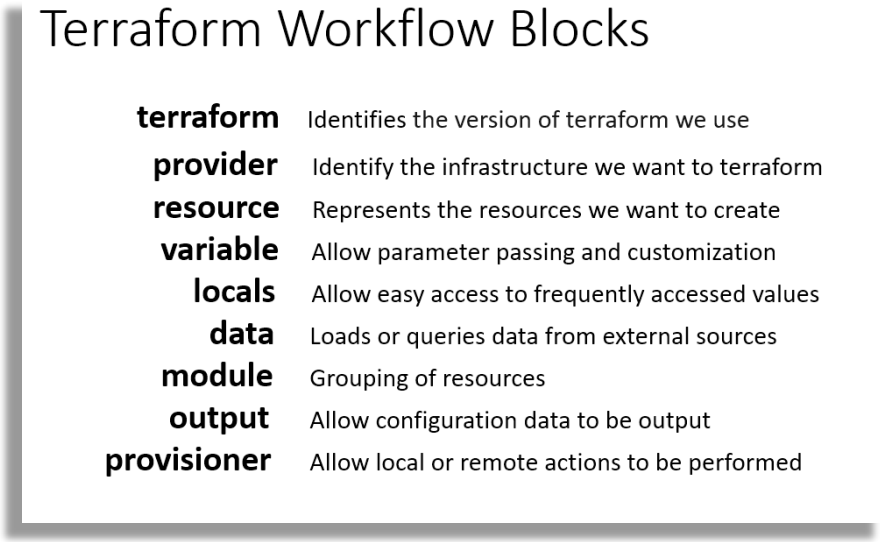
In the **plan** phase, Terraform generates an execution plan, showcasing the changes it will make to achieve the desired state. This step ensures transparency and provides an opportunity to review and approve changes before proceeding. Following this, the **apply** phase deploys the infrastructure, ensuring reproducibility across environments.

Terraform's **track** feature uses state files to record the deployed infrastructure, maintaining a source of truth that enables efficient updates and consistency. Finally, when resources are no longer needed, the **destroy** phase safely and comprehensively removes deployed infrastructure, minimizing resource wastage and costs. This lifecycle encapsulates Terraform's power to deliver a reliable, efficient infrastructure-as-code workflow.

Terraform is not a magic solution—it translates your desired state into actionable infrastructure, but only if you correctly define it. For example, if you are deploying virtual machines on AWS, you need to understand concepts like EC2 instance types, security groups, IAM roles, and networking components like VPCs and subnets. Similarly, if you are working with Kubernetes, familiarity with pods, services, and cluster configurations is essential.

A lack of foundational knowledge in the target technology can lead to misconfigured resources, security vulnerabilities, and inefficient deployments. Therefore, during the scoping phase, ensure that you or your team are familiar with the technology stack. This knowledge forms the backbone of writing accurate and effective Terraform configurations, ensuring a smooth transition into the subsequent phases of the Terraform workflow

Terraform Workflow Blocks



A workflow block is a discrete block of code used when defining how we want to use Terraform and what we want it to do. The format of a block includes a designation of the type of block, followed by zero or more labels and then braces { }, inside of which is the body of the block…

```
type “label1” “label2” {
  block body
}
```

Blocks can have sub-blocks and sub-sub blocks embedded within them

```
type “label1” “label2” {
  block body
  type “label1” “label2” {
    sub-block body
    type “label1” “label2” {
      sub-sub-block body
    }
  }
}
```

terraform block

The terraform block is used to configure settings related to Terraform itself, including the backend for storing state and required provider versions. This ensures consistency in deployments across different environments.

Azure example

```
terraform {
  backend "azurerm" {
    resource_group_name  = "myResourceGroup"
    storage_account_name = "myterraformstate"
    container_name = "tfstate"
    key = "terraform.tfstate"
  }
}
```

AWS example

```
terraform {
  backend "s3" {
    bucket = "my-terraform-state"
    key = "statefile.tfstate"
    region = "us-west-2"
  }
}
```

Google Cloud example

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

provider block

The provider block specifies the cloud provider that Terraform will use to deploy infrastructure. It defines authentication settings and configuration options.

Azure example

```
provider "azurerm" {
  features {}
  subscription_id = "your-subscription-id"
  tenant_id       = "your-tenant-id"
}
```

AWS example

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

Google example

```
provider "google" {
  project = "my-gcp-project"
  region  = "us-central1"
}
```

resource block

The resource block defines the actual cloud resources that Terraform will create, such as virtual machines, storage accounts, and networking components.

Azure example

```
resource "azurerm_virtual_network" "example" {
  name                = "myVNet"
  location            = "East US"
  resource_group_name = "myResourceGroup"
  address_space       = ["10.0.0.0/16"]
}
```

AWS example

```
resource "aws_instance" "example" {
  ami           = "ami-12345678"
  instance_type = "t2.micro"
}
```

Google example

```
resource "google_compute_instance" "example" {
  name         = "vm-instance"
  machine_type = "e2-medium"
  zone         = "us-central1-a"

  boot_disk {
    initialize_params {
      image = "debian-cloud/debian-11"
    }
  }

  network_interface {
    network = "default"
  }
}
```

variable block

The variable block allows for parameterization, making Terraform configurations reusable across multiple environments.

Azure example

```
variable "location" {
  type    = string
  default = "West Europe"
}
```

AWS example

```
variable "instance_type" {
  type    = string
  default = "t2.micro"
}
```

Google example

```
variable "project_id" {
  type    = string
  default = "my-gcp-project"
}
```

locals block

The locals block defines local values within a Terraform configuration, which are useful for simplifying repeated expressions.

Azure example

```
locals {
  default_tags = {
    Environment = "Dev"
    Owner       = "Admin"
  }
}
```

AWS example

```
locals {
  region = "us-west-2"
}
```

Google example

```
locals {
  zone = "us-central1-a"
}
```

data block

The data block allows Terraform to fetch information from existing cloud resources without creating them.

Azure example

```
data "azurerm_resource_group" "example" {
  name = "myResourceGroup"
}
```

AWS example

```
data "aws_ami" "latest" {
  most_recent = true
  owners      = ["amazon"]
}
```

Google example

```
data "google_compute_zones" "available" {}
```

module block

The module block is used to organize Terraform configurations by grouping resources into reusable components.

Azure example

```
module "network" {
  source              = "./modules/network"
  resource_group_name = "myResourceGroup"
}
```

AWS example

```
module "s3" {
  source  = "terraform-aws-modules/s3-bucket/aws"
  bucket  = "my-s3-bucket"
}
```

Google example

```
module "gke" {
  source  = "terraform-google-modules/kubernetes-engine/google"
  project = "my-gcp-project"
}
```

outputs block

The output block allows Terraform to display useful information about deployed resources.

Azure example

```
output "resource_group_name" {
  value = azurerm_resource_group.example.name
}
```

AWS example

```
output "instance_ip" {
  value = aws_instance.example.public_ip
}
```

Google example

```
output "vm_name" {
  value = google_compute_instance.example.name
}
```

provisioner block

The provisioner block is used to execute scripts or commands on a resource after creation.

Azure example

```
resource "azurerm_virtual_machine" "example" {
  name                = "vm-example"
  ....
  provisioner "remote-exec" {
    inline = ["echo 'Hello, world!'"]
  }
}
```

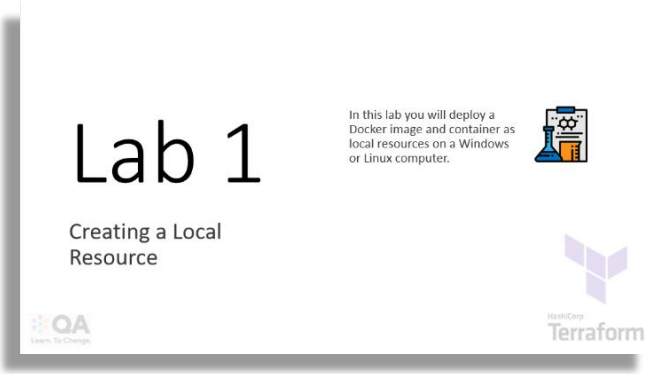
AWS example

```
resource "aws_instance" "example" {
  provisioner "file" {
    source      = "script.sh"
    destination = "/home/ubuntu/script.sh"
  }
}
```

Google example

```
resource "google_compute_instance" "example" {
  name         = "vm-instance"
  .....
  provisioner "remote-exec" {
    inline = [
      "sudo apt-get update",
      "sudo apt-get install -y nginx",
      "echo '<h1>Hello, GCP!</h1>' | sudo tee /var/www/html/index.html"
    ]
  }
  .....
}
```

Example Deployment



This lab introduces you to Terraform and guides you through using it to interact with locally installed Docker. It introduces providers, used to indicate the infrastructure that will be managed, and also the structure and composition of terraform files. You will review an example terraform file before initializing the client, planning, and then applying the deployment. You will then modify the deployment before eventually destroying it.

Let us explore a simple "Hello World" deployment using Terraform. This example uses Terraform to deploy a Docker container running the Apache HTTP server. This introduces the core concepts of Terraform, such as providers, resources, and the lifecycle workflow. The goal is to help you become familiar with Terraform basics before diving into more complex configurations. Aspects of this example deployment discussed here will be covered in greater detail later.

The Code

Below is the full Terraform configuration we will use for this example. This code will download a container image of the latest version of the Apache HTTP server and then create and run a container using it. Let us break down each section to understand how it works. Line numbers have been added for easy reference:

```
1  terraform {
2    required_providers {
3      docker = {
4        source = "kreuzwerker/docker"
5        version = "3.0.1"
6      }
7    }
8  }
9
10 provider "docker" {
11   # Configuration options as appropriate to platform
12   # on which Docker is running
13 }
14
15 # Pull the image
16 resource "docker_image" "apache_web" {
17   name  = "httpd:latest"
18 }
19
20 # Create a container
21 resource "docker_container" "web_server" {
22   image = docker_image.apache_web.image_id
23   name  = "web_server"
24   ports {
25     internal = 80
26     external = 88
27   }
28 }
```

Breaking It Down

Defining the Terraform Block

```
1  terraform {
2    required_providers {
3      docker = {
4        source = "kreuzwerker/docker"
5        version = "3.0.1"
6      }
7    }
8  }
```

This is the terraform code block. It specifies the provider(s) required for the configuration, here it is docker, indicating that we want to use Terraform to manage Docker resources. The source section, line 4, identifies the docker provider component to be downloaded during initialization, and the version, line 5, indicates which version is required.

Initializing the Provider

```
10  provider "docker" {
11    # Configuration options as appropriate to platform
12    # on which Docker is running
13  }
```

The provider block details parameters to be used when initializing the Docker provider. The configuration needed here is dependent upon the platform on which Docker is running. On Windows machines running Docker Desktop where the docker service is accessible using tcp port 2375, this would be..

```
10  provider "docker" {
11    host = "tcp://localhost:2375"
12  }
```

Whereas, on Unix machines running Docker, this might be..

```
10  provider "docker" {
11    host = "unix:///var/run/docker.sock"
12  }
```

Pulling the Docker Image

```
15  # Pull the image
16  resource "docker_image" "apache_web" {
17    name  = "httpd:latest"
18  }
```

This is one of two blocks of code that indicate the resources we want Terraform to create. Here we are defining a docker image that needs to be pulled from Docker hub. Line 17 identifies the image we want, the latest version of “httpd”, an Apache webserver.

Creating the Docker Container

```
20  # Create a container
21  resource "docker_container" "web_server" {
22    image = docker_image.apache_web.image_id
23    name  = "web_server"
24    ports {
25      internal = 80
26      external = 88
27    }
28  }
```

This is the second block of code that indicates the resources we want Terraform to create. Here we are defining that we want docker to create a container called “web-server” (line23) and we want it to be accessible using ports 80 and 88 (lines 24-27). A docker container is created from a docker image. Line 22 indicates that the image to be used to create this container is the one we pull from Docker hub as defined in lines 16-18. This is an example of one resource code block referencing another and it also indicates to terraform that it must download the image ***before*** it can create the container so that it can ascertain its id. This is known as implicit dependency; the container depends upon the image existing.

Running the Code

The file containing the above code should be saved with a file extension of .tf so that it is recognized by terraform. Any name can be used but conventionally the first terraform file created should be **main.tf** This file should be created and saved into an empty file directory which will become the root directory for the deployment. All terraform command should be run in this directory.

Deploy the Configuration

Having installed the Terraform client software, we would now switch to a command prompt (PowerShell, CMD, terminal etc) and navigate to the root directory containing the file. We would then initialize Terraform in this directory by running **terraform init.** A plan of the proposed deployment could now be generated by running **terraform plan** and after reviewing the plan we could apply the deployment by running **terraform apply** and entering **yes** when prompted.

Verify the deployment

To confirm the resources have been created we could now open a web browser and navigate to **http://localhost:88** to see the Apache welcome page. This would verify that the image was downloaded to Docker and a container was created and made available on Port 88.

Modify the deployment

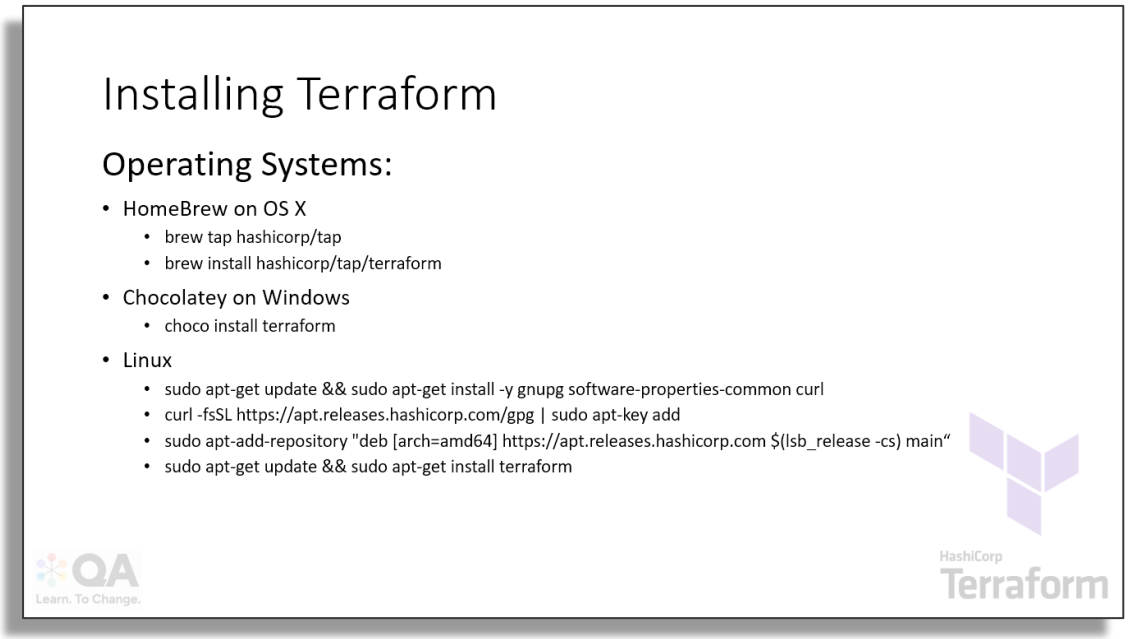
Should we need to, we could modify our deployed resources by editing the terraform file, changing the external port from 88 to **8080** for example, and then save the file changes. We could test these deployment changes by running **terraform plan** again. In this case we would be notified that the original container would be destroyed, and a new replacement created. Applying the new deployment by running **terraform apply** and entering **yes** when prompted would therefore delete the old container and deploy a replacement and this could then be verified by browsing to **http://localhost:8080**

Cleaning Up

To remove the resources and clean up the environment we would run **terraform destroy** and confirm with **yes**. This would remove the Docker container and any associated resources.

2. Terraform Foundations

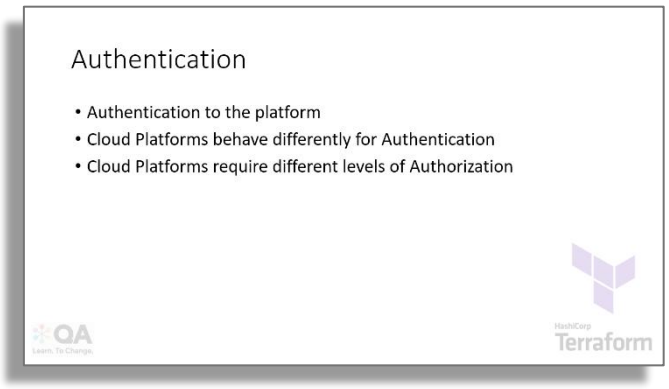
Installing Terraform



Terraform is a versatile tool that can be installed on Windows, macOS, and Linux. To install Terraform, start by visiting the official Terraform website and downloading the appropriate binary for your operating system. Once downloaded, extract the contents of the archive to a directory of your choice. For added convenience, consider adding this directory to your system's PATH environment variable, enabling you to run Terraform commands from any location in the command line. To confirm the installation, open a terminal or command prompt and run **terraform version**; if installed correctly, the version information will be displayed. While these steps provide a general approach, installation details can vary based on your operating system or preferred method. For more specific guidance, consult the official documentation. Additionally, package managers like Homebrew for macOS and Chocolatey for Windows offer a simplified way to install, and update Terraform. Refer to their documentation for further instructions.

Authentication

Generally, Terraform recommend using either a Service Principal or Managed Service Identity when running Terraform non-interactively (such as when running Terraform in a CI server) - and authenticating using the cloud vendor CLI when running Terraform locally.

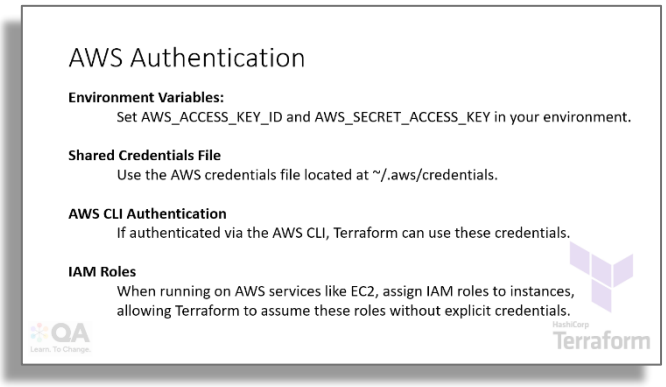


Credentials can be passed to locally run Terraform in numerous ways. These can include prompting for credentials, usage of locally cached credentials, hardcoding credentials within the Terraform file and referencing a file containing credentials.

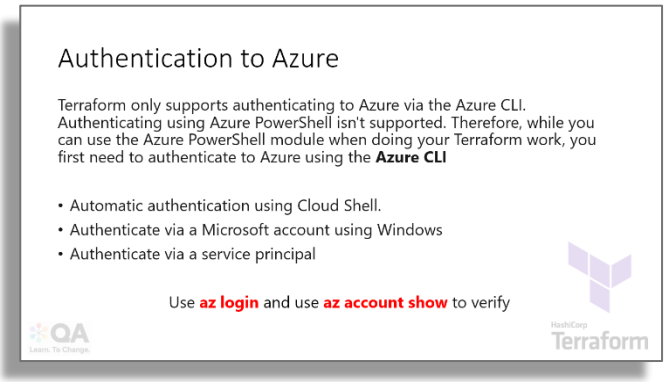
The specifics of these options will vary by provider and reference should be made to the appropriate guidance within the Terraform Registry.

AWS Authentication

AWS provides multiple ways to authenticate Terraform to interact with its services. The most straightforward method is through environment variables set directly in your environment. Another common approach is using the credentials file typically located in the .aws folder in your home directory. Alternatively, AWS CLI authentication can be used if a user has already authenticated via the CLI, enabling Terraform to leverage those credentials. Lastly, for enhanced security and scalability, IAM roles can be assigned to AWS resources like EC2 instances. These roles allow Terraform to assume permissions without explicitly providing access keys, streamlining authentication while adhering to best practices for security.



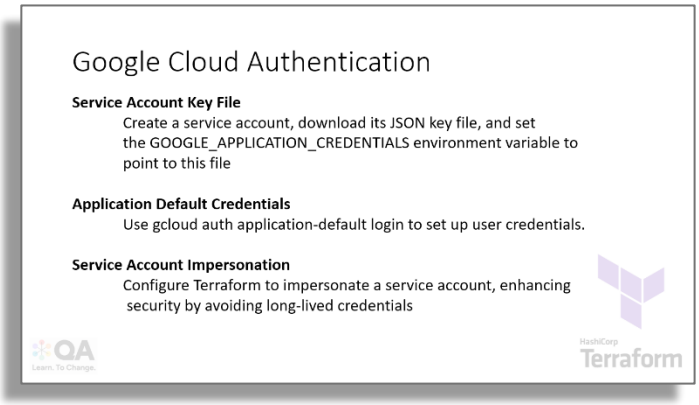
Azure Authentication



Terraform requires authentication to Azure through the **Azure CLI**, as other methods like Azure PowerShell are not supported for direct interaction. While Azure PowerShell can still be used for complementary tasks, the primary method to authenticate Terraform is via the Azure CLI. By running the az login command, you establish a session, which Terraform then uses to interact with Azure resources.

There are multiple ways to authenticate through Azure CLI to suit different use cases. For simplicity, you can use automatic authentication through Azure Cloud Shell, which provides a pre-authenticated environment. Alternatively, you can authenticate interactively using your Microsoft account on Windows. For automation or non-interactive setups, such as in CI/CD pipelines, service principal authentication is preferred. This method provides credentials for secure programmatic access. Once authenticated, you can verify your session using az account show, ensuring that Terraform has access to manage Azure resources effectively.

Google Cloud Authentication



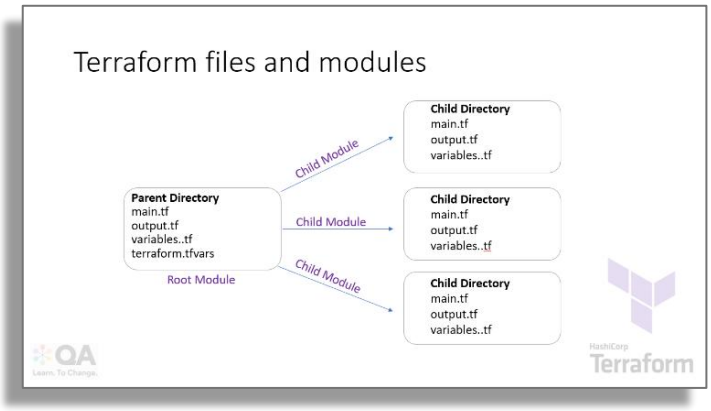
Google Cloud offers multiple methods for authenticating Terraform to manage resources securely and effectively. One of the most common approaches is using a **Service Account Key File**. In this method, you create a service account in Google Cloud, download its JSON key file, and set the GOOGLE_APPLICATION_CREDENTIALS environment variable to point to this file. This approach is widely used for automation and ensures that Terraform can authenticate with the necessary permissions to perform actions. Another option is leveraging Application Default Credentials (ADC). By running the gcloud auth application-default login command, you can authenticate Terraform using the credentials of your user account. This method is particularly useful for development or test environments, as it allows users to authenticate with their personal credentials without requiring a service account.

For enhanced security, Service Account Impersonation is a recommended practice. Instead of relying on long-lived service account key files, Terraform can be configured to impersonate a service account. This avoids the risks associated with managing key files and

ensures that permissions are granted only when necessary. By implementing these methods, Google Cloud ensures flexibility and security in Terraform deployments.

Terraform files

Terraform uses several files, plain text files generally with a .tf or .tfvars extension. It also supports JSON files with *.tf.json* extension. In this course we are using the Hashicorp Configuration Language (HCL) and therefore .tf files.



Some additional files are created automatically, those allowing communication between the local Terraform client and the CSP, and others recording our desired or current state.

Terraform fmt

The **terraform fmt** command in Terraform can be used to automatically format and standardize the structure and style of your Terraform code. It helps ensure consistent formatting across your configuration files, making them more readable and maintaining a standardized codebase.

```
terraform {
required_providers {
docker = {source = "kreuzwerker/docker"
version = "~> 2.21.0"}
}
}
provider "docker" {}
resource "docker_image" "Apache_web"{
name = "httpd:latest"
keep_locally = false
}
resource "docker_container" "web"{
image = docker_image.Apache_web.image_id
name  = "Web-demo"
ports {
internal = 80
external = 8080
}
}
```

```
terraform {
  required_providers {
    docker = { source = "kreuzwerker/docker"
    version = "~> 2.21.0" }
  }
}
provider "docker" {}
resource "docker_image" "Apache_web" {
  name         = "httpd:latest"
  keep_locally = false
}
resource "docker_container" "web" {
  image = docker_image.Apache_web.image_id
  name  = "Web-demo"
  ports {
    internal = 80
    external = 8080
  }
}
```

Code Formatting

`terraform fmt` reformats your Terraform code according to a predefined set of style conventions. It adjusts indentation, spacing, and alignment to ensure a consistent and uniform appearance of your configuration files. This improves code readability and makes it easier for team members to collaborate and review the code.

Configuration Files

`terraform fmt` applies formatting to Terraform configuration files (typically with the `.tf` extension) and supports both HCL (HashiCorp Configuration Language) and JSON formats. It automatically detects and reformats these files, making it a convenient way to maintain consistent style throughout your project.

In-place Modification

When you run `terraform fmt`, it modifies the files in place, updating them with the formatted code. The original file contents are replaced with the formatted version. It is recommended to have your code under version control before running `terraform fmt` to ensure you have a backup in case you need to revert any changes.

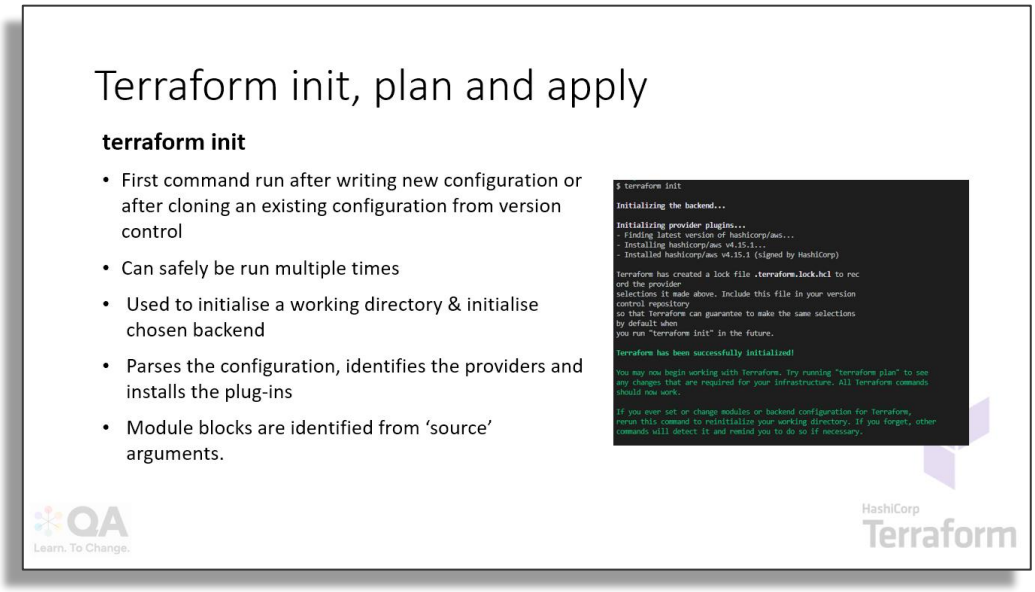
Integration with Editors and IDEs

Many popular code editors and integrated development environments (IDEs) have plugins or extensions that provide automatic formatting capabilities. These integrations can automatically run `terraform fmt` upon file save, ensuring your code is consistently formatted without needing to manually invoke the command.

By using `terraform fmt`, you can enforce a consistent coding style across your Terraform configuration files. This is particularly helpful when working with teams, as it eliminates inconsistencies and makes the codebase more maintainable and readable. Additionally, consistent formatting enhances the diff visibility in version control systems, allowing for clearer tracking of changes over time.

It is recommended to run `terraform fmt` as part of your code review process or as a pre-commit hook in your version control system to ensure that code formatting standards are enforced consistently across your Terraform project.

Terraform init



The 'terraform init' command initializes a working directory containing Terraform configuration files. This is the first command that should be run after writing a new Terraform configuration or cloning an existing one from version control. It is safe to run this command multiple times.

During init, the root configuration directory is consulted for backend configuration and the chosen backend is initialized using the given configuration settings. 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. This lets multiple people access the state data and work together on that collection of infrastructure resources. This topic is discussed later in the course.

Terraform plan

The 'terraform plan' command is used to generate an execution plan. It allows you to preview the changes that Terraform would make to your infrastructure without applying those changes.

When you run 'terraform plan', Terraform performs the following actions:

- Reads the Terraform configuration files.
- Validates and interprets the configuration, including provider settings and resource definitions.
- Compares the desired state defined in the configuration files with the current state stored in the state file.
- Analyses the differences between the desired state and the current state to determine what changes are required.
- Generates an execution plan that outlines the actions Terraform would take to reach the desired state.
- Outputs a detailed summary of the planned actions, including resource creation, modification, or deletion, as well as any other changes required.
- Provides information such as the estimated number of resources to be created, modified, or destroyed, and any potential errors or conflicts.

The 'terraform plan' command is useful for verifying the impact of changes to your infrastructure before applying them. It helps you catch potential issues or conflicts, understand the scope of the changes, and review the expected outcome. By reviewing the plan, you can ensure that the proposed changes align with your intentions and make any necessary adjustments to the configuration before proceeding with 'terraform apply' to execute the plan and make the changes to your infrastructure. Terraform does have an optional -out flag that is used in conjunction with the terraform plan and terraform apply commands. The -out flag allows you to specify a path to save the generated execution plan

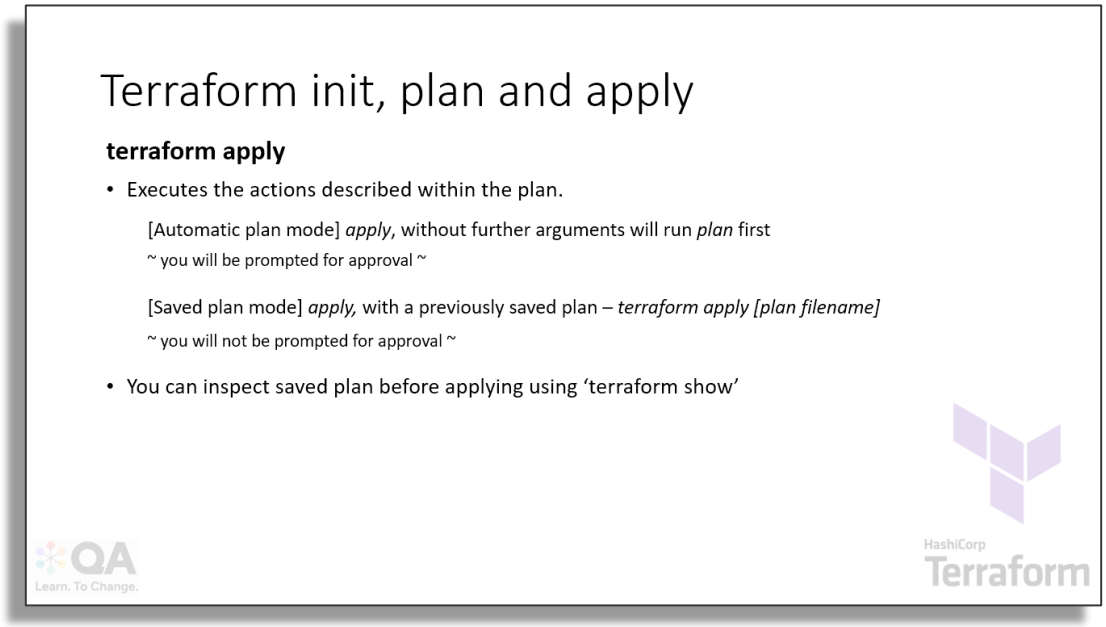
as a plan file. The plan file contains all the information about the planned changes, including resource creation, modification, and destruction.

When running the terraform plan command, you can use the -out flag to save the plan to a file: **terraform plan -out=tfplan** This command will generate an execution plan and save it to the file named "tfplan" (you can choose any desired filename).

Later, when you want to apply the plan and make the changes to your infrastructure, you can use the terraform apply command with the -input=false flag and provide the path to the saved plan file using the -out flag: **terraform apply -input=false tfplan**

By using the -out flag, you can separate the planning and execution steps, allowing you to save the plan for later use or share it with other team members. It ensures that the exact plan generated earlier is applied without recalculating it, providing consistency and predictability in your infrastructure changes.

Terraform apply



When you run the 'terraform apply' command, Terraform compares the current state of your infrastructure, as stored in its state file, with the desired state defined in your Terraform configuration files. It then determines the necessary changes to make the infrastructure match the desired state.

The 'terraform apply' command performs the following actions:

- Reads the Terraform configuration files.
- Validates and interprets the configuration, including provider settings and resource definitions.
- Compares the desired state to the current state by querying the state file.
- Determines the execution plan for making the infrastructure match the desired state.
- Presents the execution plan to the user, showing what changes Terraform will make.
- Asks for confirmation from the user to proceed with the execution plan.

- Executes the plan by creating, modifying, or destroying resources, as necessary.
- Updates the state file with the new state reflecting the changes made.
- Outputs any relevant information or metadata about the created or modified resources.

It is important to note that 'terraform apply' can create, modify, or delete infrastructure resources, depending on the changes required to reach the desired state. It is a powerful command that should be used with caution, especially in production environments, as it directly affects the state of your infrastructure. The 'terraform apply' command can use a saved plan generated by the 'terraform plan' command. The 'terraform plan' command generates an execution plan without making any changes to the infrastructure. It provides a preview of the actions that Terraform would take if you were to apply the plan. To use a saved plan with 'terraform apply', you can specify the path to the plan file using the '-out' flag. Here is the syntax:

terraform apply -input=false -auto-approve -state=<path-to-state-file> <path-to-plan-file>

-input=false This flag ensures that Terraform does not prompt for any input during the apply process, allowing it to be used in automated or non-interactive scenarios.

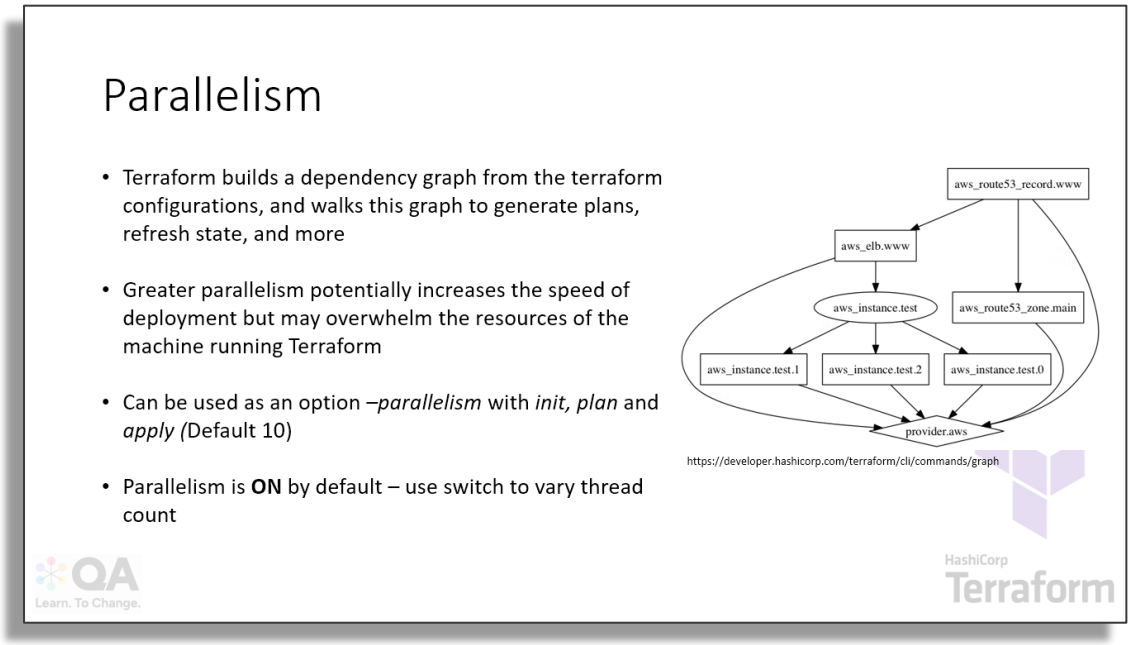
-auto-approve This flag instructs Terraform to automatically approve and apply the plan without asking for confirmation.

-state=<path-to-state-file>` This flag specifies the path to the state file associated with the infrastructure you want to apply changes to. It ensures Terraform uses the correct state file for tracking changes.

<path-to-plan-file> This is the path to the saved plan file generated by 'terraform plan'. Specify the path to the plan file you want to apply.

By providing the plan file generated by 'terraform plan' to the 'terraform apply' command, you can avoid recalculating the execution plan and directly apply the changes outlined in the saved plan. This can be useful when you want to ensure that the exact plan generated earlier is applied, without any potential changes due to infrastructure modifications or configuration updates.

Parallelism

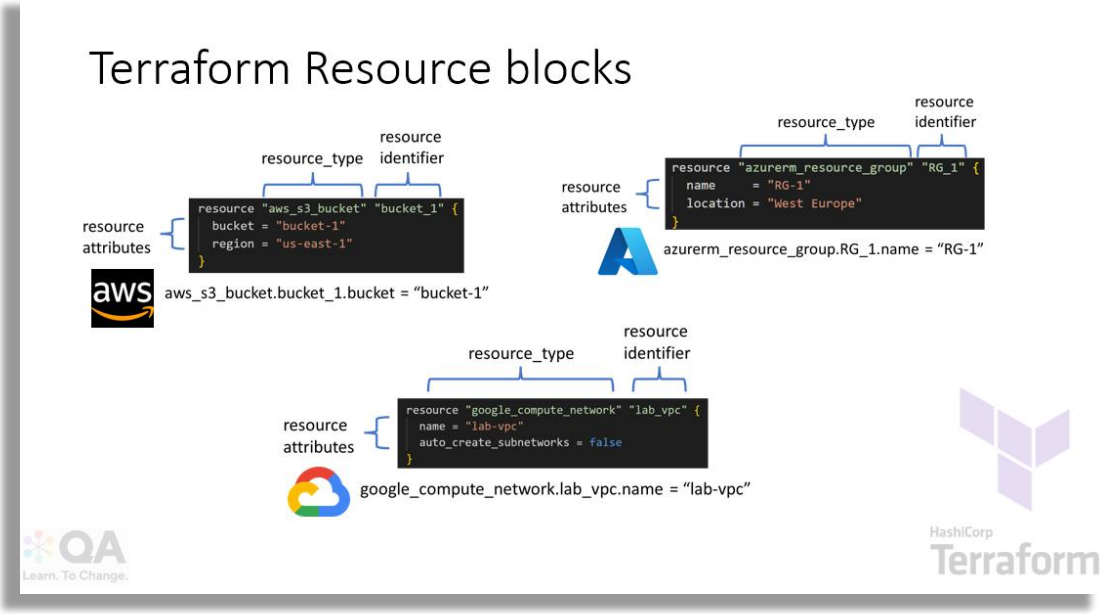


In Terraform, parallelism refers to the ability to perform multiple resource operations concurrently during the execution of Terraform commands such as terraform apply or terraform destroy. It allows Terraform to execute resource operations in parallel, which can significantly speed up the provisioning or destruction of resources in your infrastructure.

Without parallelism, Terraform executes operations sequentially, processing each resource one by one. However, when you have many resources, executing operations in parallel can improve the overall performance and reduce the time taken to apply or destroy changes. You can control the parallelism level in Terraform by using the -parallelism flag with the terraform apply, terraform destroy, or other relevant commands. The -parallelism flag allows you to specify the maximum number of resource operations to be executed concurrently.

Starting from Terraform 0.14, the default behaviour of dynamic parallelism configuration based on client CPU count changed and Terraform now uses a default static parallelism level of 10 if no explicit value of between 0 and 256 is specified for the -parallelism flag.

Terraform Resource Blocks



Terraform uses **resource blocks** to define the infrastructure components that it manages. Each resource block consists of a **resource type**, a **resource identifier**, and a set of **resource attributes**.

Resource Type: The first element, such as "aws_s3_bucket", specifies the type of resource being created. In this case, it defines an S3 bucket. The resource type determines what kind of infrastructure element Terraform will manage (e.g., virtual machines, storage accounts, etc.).

Resource Identifier: The second element, "bucket_1", acts as the logical name or identifier for the resource within your Terraform configuration. This name is used to reference the resource elsewhere in your Terraform code.

Resource Attributes: Attributes like *bucket = "bucket-1"* and *region = "us-east-1"* define the properties of the resource. These attributes specify how the resource should be configured. For example:

bucket: The name of the S3 bucket.

region: The geographical region where the bucket is deployed.

Once a resource is created, its attributes can be accessed using the format **resource_type.resource_identifier.attribute_name**. For example:

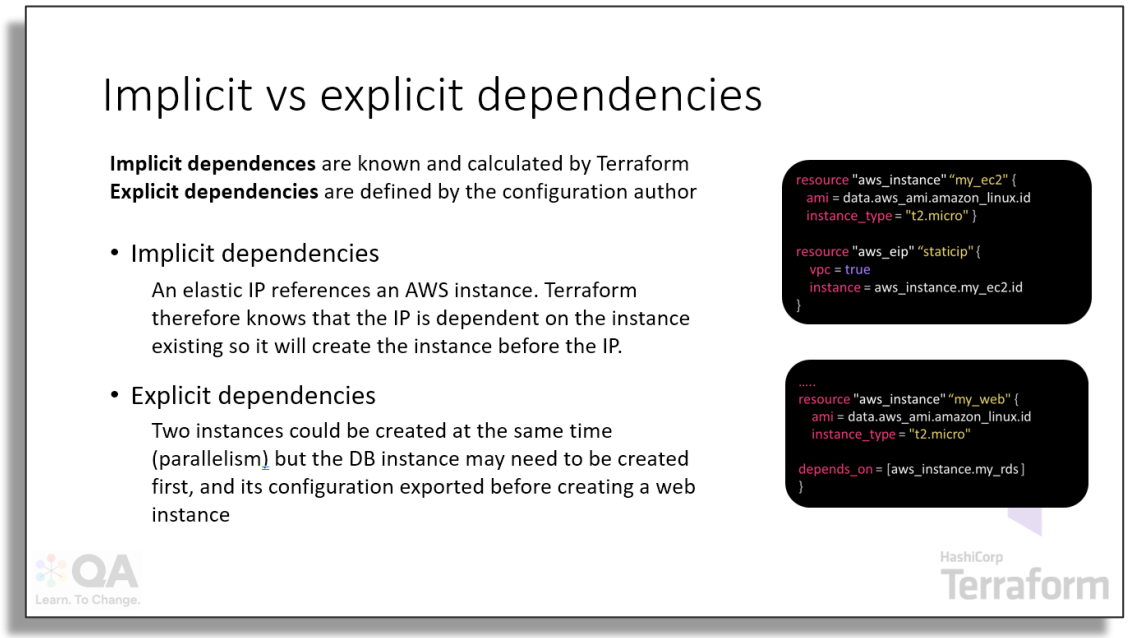
aws_s3_bucket.bucket_1.bucket returns "bucket-1".

Aws_s3_bucket.bucket_1.region returns "us-east-1".

This allows you to reference resource attributes dynamically in other parts of your configuration, enabling dependencies and modular infrastructure definitions.

By understanding this structure, you can effectively define, manage, and reference your infrastructure components using Terraform.

Implicit vs explicit dependencies



In Terraform, dependencies represent relationships between resources where the creation, modification, or deletion of one resource depends on the state of another resource. Dependencies ensure that resources are created or modified in the correct order to maintain the desired state of your infrastructure.

Terraform distinguishes between two types of dependencies: implicit and explicit.

Implicit Dependencies

Implicit dependencies are automatically inferred by Terraform based on resource references within your configuration. When Terraform detects a reference to another resource within a resource definition, it establishes an implicit dependency between them. Terraform analyses these references during the dependency graph construction phase to determine the correct order of resource creation, modification, or destruction.

In this example the location tag to be applied to the DynamoDB table references the region where an S3 bucket exists. Terraform will implicitly understand that the DynamoDB table creation depends on the existence of the S3 bucket. Terraform will ensure that the S3 bucket group is created or modified before attempting to create or modify the DynamoDB table.

```
resource "aws_s3_bucket" "bucket_1" {
  bucket = "bucket-1"
  region = "us-east-1"
}

resource "aws_dynamodb_table" "example_table" {
  name           = "example-table"
  hash_key       = "id"
  tags = {
    Location = aws_s3_bucket.bucket_1.region
  }
}
```

Implicit dependencies are determined by the resource relationships defined in your Terraform configuration and are typically based on input/output variables and references between resources.

Explicit Dependencies

Explicit dependencies are manually defined in your Terraform configuration using the `depends_on` argument. With explicit dependencies, you can explicitly specify the dependency relationship between resources that cannot be inferred by Terraform from the configuration alone. This can be useful when there are dependencies that cannot be determined through implicit means.

```
.....
resource "aws_instance" "my_web" {
  ami = data.aws_ami.amazon_linux.id
  instance_type = "t2.micro"

depends_on = [aws_instance.my_rds ]
}
```

The `depends_on` argument allows you to specify a list of resources that a particular resource depends on. This ensures that the specified resources are created or modified before the dependent resource.

Explicit dependencies give you more control over the ordering of resource operations and allow you to handle cases where the implicit dependencies are not sufficient or accurate.

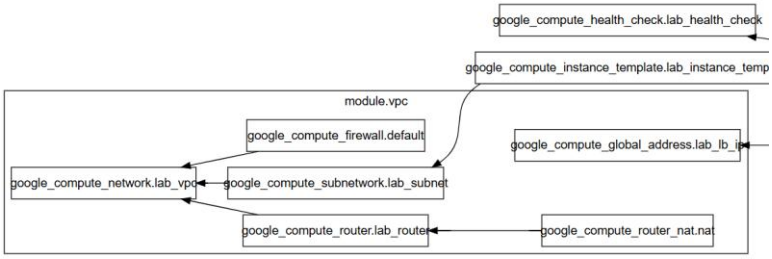
Both implicit and explicit dependencies play crucial roles in ensuring that Terraform executes resource operations in the correct order, maintaining the desired state of your infrastructure.

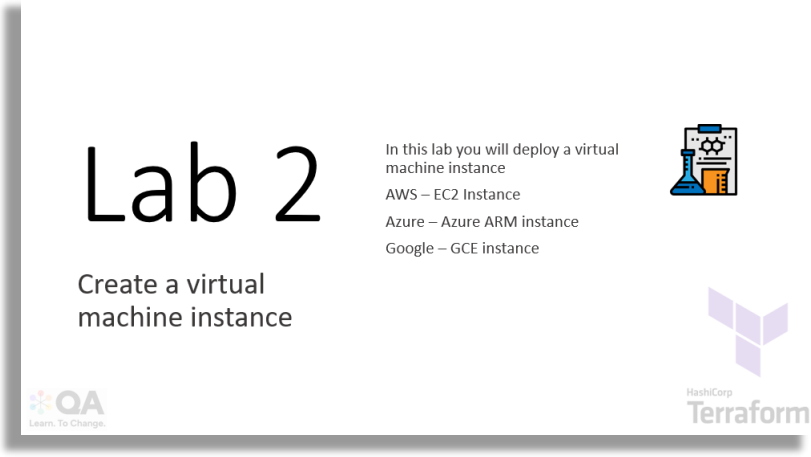
Terraform graph

The **terraform graph** command is used to generate a visual representation of the dependency graph for a Terraform configuration. This graph shows how resources and modules are linked together based on their dependencies, helping users understand the order in which Terraform will create or destroy resources. The output is generated in DOT format, which can be rendered into images using visualization tools like Graphviz. This can be especially helpful when troubleshooting or optimizing resource dependencies in complex infrastructures.

By default, terraform graph shows the relationships between all nodes, including resources, modules, data sources, and provisioners. It reflects both implicit dependencies (like a resource referencing another's output) and explicit ones defined using depends_on.

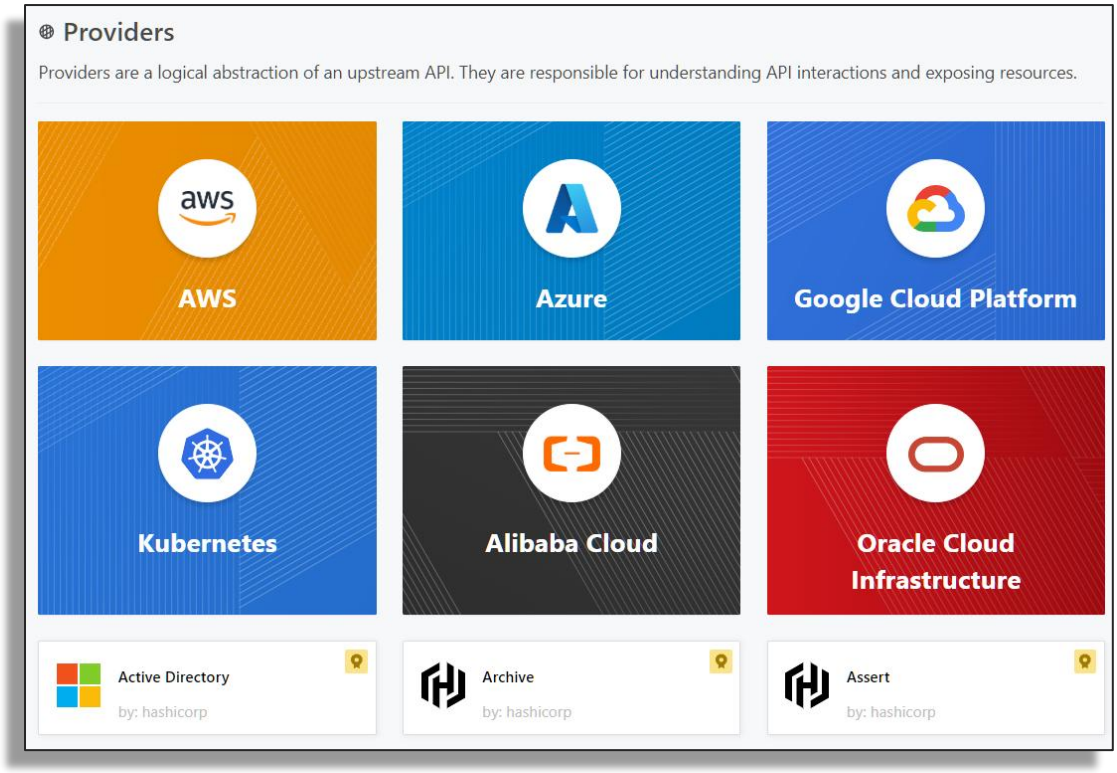
While the raw DOT output can be difficult to interpret directly, converting it into a diagram makes it easier to identify potential bottlenecks or unnecessary dependencies in the infrastructure plan.





This lab guides you through the creation of a virtual machine in your chosen cloud. It highlights the need for network support and re-enforces resource referencing and dependency

3. Terraform Providers



Introduction to Providers

In Terraform, providers are plugins that enable Terraform to interact with different infrastructure platforms, services, or providers. Providers serve as the bridge between Terraform and the target infrastructure, allowing Terraform to provision and manage resources within those platforms.

A provider in Terraform corresponds to a specific infrastructure platform or service. Examples of providers include AWS, Azure, Google Cloud, Kubernetes, Docker, and many others. Each provider has its own set of resources and data sources that can be managed using Terraform.

Provider Configuration

To use a provider in Terraform, you need to configure it within your Terraform configuration file. This involves specifying the provider block, which defines the provider and its specific configuration details. The configuration can include authentication credentials, region or location settings, access keys, endpoint URLs, and other parameters required to establish a connection to the infrastructure platform.

Resource and Data Source Provisioning

Providers in Terraform expose resources and data sources that can be managed using Terraform configuration. Resources represent infrastructure objects, such as virtual machines, networks, storage buckets, or databases. Data sources provide read-only access to existing infrastructure objects, allowing you to reference information from external systems within your Terraform configuration.

Provider Plugins

Providers in Terraform are implemented as plugins. When you run `terraform init` for a configuration that uses a particular provider, Terraform automatically downloads and installs the corresponding provider plugin. Provider plugins are versioned and managed separately from Terraform itself. You can view and manage installed provider plugins using the **terraform providers** command.

Terraform allows you to manage resources across multiple infrastructure platforms or services. Each Terraform configuration typically focuses on a single provider, meaning you would create separate configurations for each provider you want to manage. Each configuration would have its own provider block specific to the corresponding infrastructure platform or service.

By utilizing providers, Terraform offers a unified way to manage infrastructure across different platforms and services. It provides a consistent workflow for provisioning, modifying, and destroying resources, regardless of the underlying infrastructure provider.

Local Provider

Terraform's **local provider** allows you to manage resources and data sources that are confined to your local environment, rather than a cloud or external platform. This is particularly useful for workflows that involve generating files, configuring local systems, or reading data from local sources. The local provider simplifies these tasks by providing resources and data sources tailored for local file management.

```
resource "local_file" "foo" {
  content  = "foo!"
  filename = "${path.module}/foo.bar"
}

data "local_file" "foo" {
  filename = "${path.module}/foo.bar"
}

resource "aws_s3_object" "shared_zip" {
  bucket  = "my-bucket"
  key     = "my-key"
  content = data.local_file.foo.content
}
```

In the example here, the local provider is used to create and manage a local file (local_file) and to read its content using the data "local_file" data source. Here is how the process unfolds:

Creating a Local File with local_file: The local_file resource creates a file on the local filesystem. In this case, a file named foo.bar is created within the module's directory (${path.module}), and its content is set to the string "foo!". This demonstrates how Terraform can generate local files dynamically, which might be necessary for automation or as a precursor to uploading data elsewhere.

Reading the File with data "local_file": After the file is created, the data "local_file" block reads its content. This data source is helpful when you need to reference the file's content or metadata in other parts of your configuration. In the example, the file's content is extracted and used as input for an AWS S3 object.

Using the File's Content Elsewhere: Finally, the file's content is passed to an aws_s3_object resource, which uploads the content to an S3 bucket. The content argument of the S3 object directly references data.local_file.foo.content, showing how seamlessly local provider resources and data sources integrate with other Terraform providers.

Why Use the Local Provider?

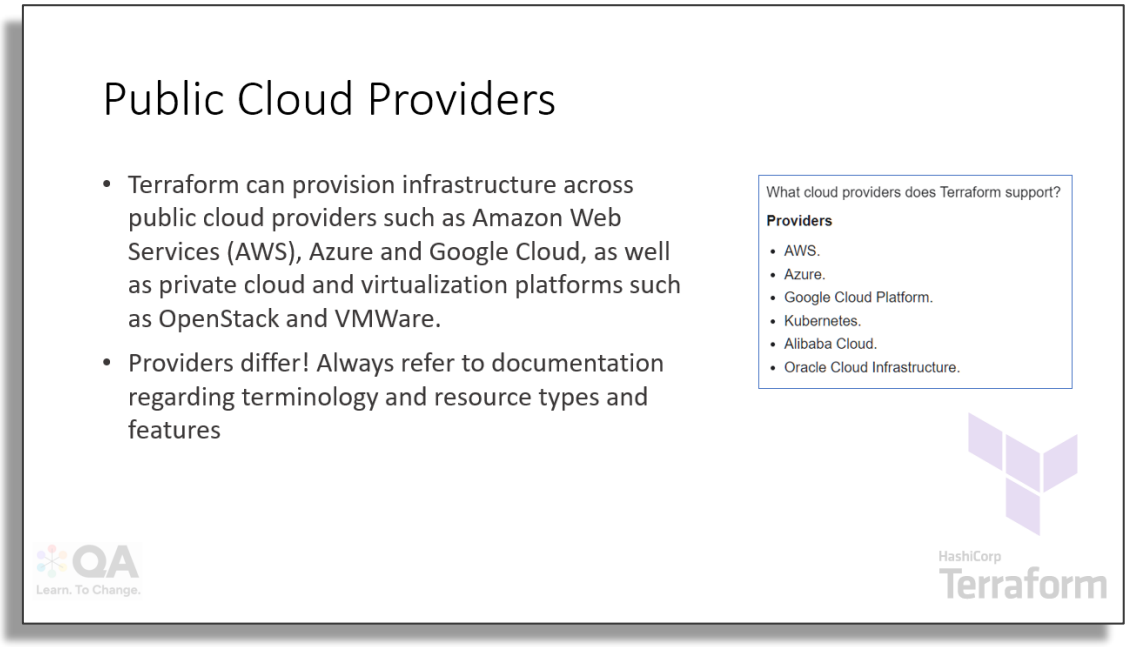
The local provider is valuable in scenarios where local resources, such as files or directories, are a critical part of your Terraform workflow. This might include:

- Dynamically generating configuration files to be used by other systems.
- Preparing data or logs for upload to a cloud service.
- Managing local resources for development or testing environments.

The example demonstrates the flexibility of the local provider. By first creating a local file, reading its content, and then integrating it with an AWS resource, it shows how Terraform can bridge local and remote resources seamlessly. This workflow is particularly relevant when local resources need to be manipulated or referenced as part of a larger Terraform deployment.

Note that the Local provider is limited to managing local resources and configurations and does not interact with remote infrastructure platforms or services. Its primary purpose is to incorporate local operations into your Terraform workflow.

Public Cloud Providers



Terraform integrates with various public cloud providers through provider-specific plugins. These plugins enable Terraform to interact with the APIs of different cloud platforms, allowing you to define and manage resources within those cloud environments using Terraform configuration files.

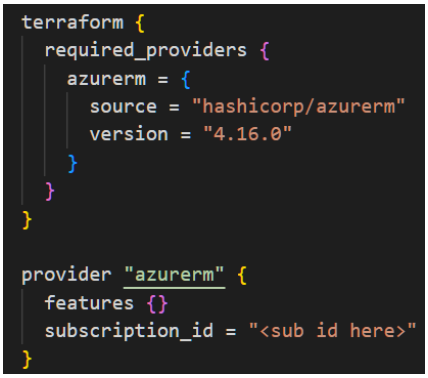
When working with Terraform, you typically include a provider block in your configuration to specify the cloud provider you are targeting. The provider block includes the necessary configuration details, such as authentication credentials, region, access keys, and other settings required to connect to the chosen provider.

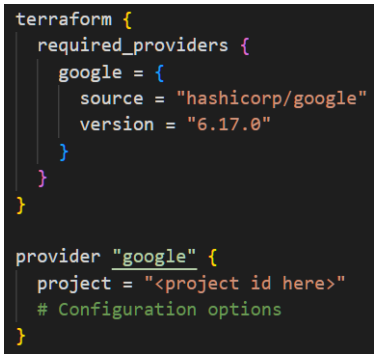
```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.53.0"
    }
  }
}

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

Here is an example of a provider block for AWS in a Terraform configuration. The provider block specifies use of the AWS provider which requires specifying a default region.

Here is an example of a provider block for Azure in a Terraform configuration. The provider block specifies use of the azurerm provider which requires specifying a subscription identifier.





Here is an example of a provider block for Google Cloud in a Terraform configuration. The provider block specifies use of the google provider. Specifying a project here is optional as it can be declared elsewhere.

Refer to the Terraform registry (https://registry.terraform.io/browse/providers) for provider information. Once you have configured the provider, you can use resource blocks within your Terraform configuration to define the desired infrastructure resources offered by the cloud provider. For example, you can create instances, storage buckets, virtual networks, load balancers, databases, and more using the resources provided by the public cloud provider. The Terraform registry provides example code to assist you, but this code may require modifying to suit your environment.

By leveraging the provider plugins in Terraform, you can harness the power of public cloud providers and define your infrastructure as code. Terraform allows you to provision, modify, and manage resources consistently across multiple cloud platforms, simplifying the management of complex infrastructure deployments.

Aliases

In Terraform, provider aliases allow you to define multiple configurations for the same provider and use them selectively in your code. This feature is particularly useful when working across:

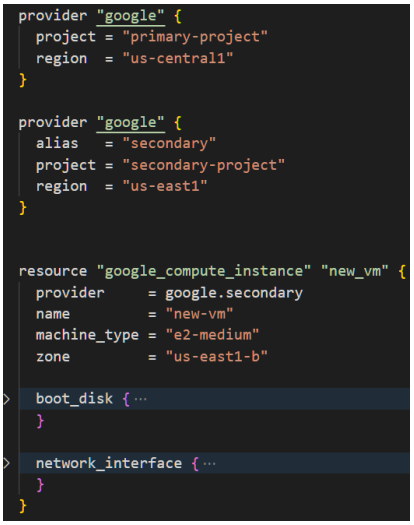
- **Multiple regions** in a cloud platform.
- **Separate accounts** or projects within the same platform.
- **Multiple instances of services**, such as Docker hosts or Consul clusters.

Why Use Aliases?

By default, Terraform assigns a single provider configuration to all resources. However, there are scenarios where you may need distinct configurations, such as:

- Deploying resources to different AWS regions.
- Using separate Azure subscriptions for development (dev) and production (prod) environments.
- Targeting multiple GCP projects for isolated resource management.

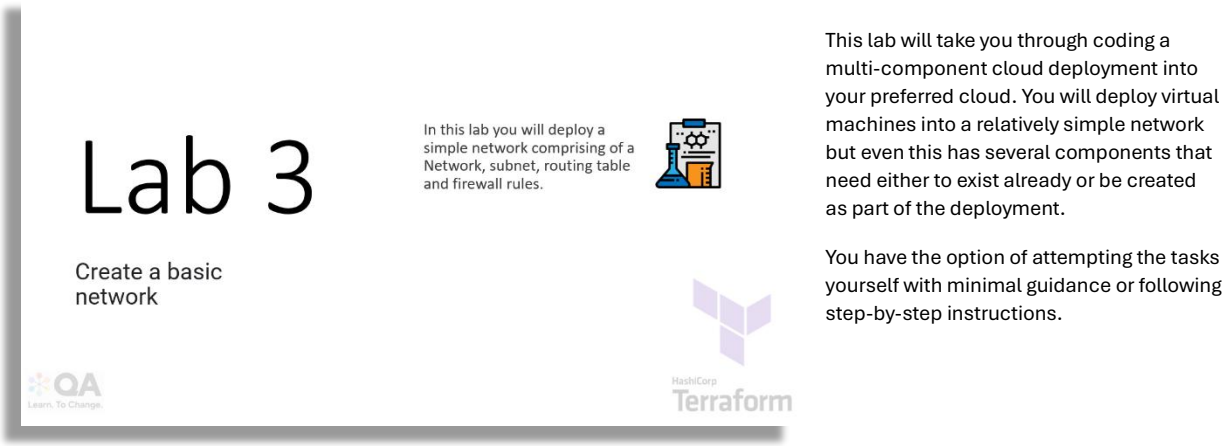
In this example, we define two provider blocks for AWS with different aliases and configurations. Each provider block represents a separate AWS region (us-east-1 and us-west-2) The alias's argument within the provider block allows you to assign alternative names to each provider configuration.



In this example, we define two provider blocks for Google with different aliases and configurations. Each provider block represents a separate Google Cloud project and different default locations (us-central1 and us-east1).

In this example, we define two provider aliases for Azure with different subscriptions being referenced through variables. The alias's argument within the provider block allows you to assign alternative names to each provider configuration, here 'dev' and 'prod'.

By using aliases within the provider block, you can differentiate between multiple instances of the same provider type and configure them with different settings. This can be helpful when you need to work with multiple regions, accounts, or environments within a single Terraform configuration.



This lab will take you through coding a multi-component cloud deployment into your preferred cloud. You will deploy virtual machines into a relatively simple network but even this has several components that need either to exist already or be created as part of the deployment.

You have the option of attempting the tasks yourself with minimal guidance or following step-by-step instructions.

4. Variables, Functions and Workspaces

Introduction to Variables

Variables allow you to define and use dynamic values within your configuration. They provide a way to parameterize your Terraform code, making it more flexible and reusable. Variables can be used to input values from external sources, pass values between modules, or simply make your configuration more configurable. In Terraform, there are two types of variables: input variables and output variables. They serve different purposes within your Terraform configuration.

Input Variables

Input variables allow you to define values that are provided as inputs to your Terraform configuration. They act as parameters that allow you to make your configuration more flexible and customizable. Input variables can be defined anywhere in your code, but conventionally, dedicated .tf files, typically named `**variables.tf**`, are used.

```
variable "region" {
  description = "AWS region"
  type        = string
  default     = "us-west-2"
}
```

Input variables can have type, descriptions, default values, and other attributes. In the example shown here, `aws_region` is an input variable that specifies the desired AWS region. It has a type of `string` and a default value of "us-west-2". If an alternative value is not provided when running Terraform, the

default value will be used whenever the variable is referenced in your code.

Having defined variables, they can then be referenced by your code as needed, thereby allowing values to be passed at runtime….

```
variable "inst_count" {
  description = "Number of instances"
  type        = number
  default     = 1
}

variable "inst_size" {
  description = "Instance size"
  type        = string
  default     = "t2.micro"
}

variable "inst_ami" {
  description = "Instance AMI"
  type        = string
  default     = "ami-0d08c3b92d0f4250a"
}
```

Here 3 variables are defined, **inst_count**, **inst_size** and **inst_ami**

So rather than writing our code with explicit values….

```
resource "aws_instance" "example" {
  ami           = "ami-0d08c3b92d0f4250a"
  instance_type = "t2.micro"
  count         = 1
}
```

We now reference the variables we have defined….

```
resource "aws_instance" "example" {
  ami           = var.inst_ami
  instance_type = var.inst_size
  count         = var.inst_count
}
```

As shown in the above example, variables are referenced by use of the **var.** prefix, followed by the variable name. A variable ***must*** be declared before it can be referenced. Referencing an undeclared variable will generate an error.

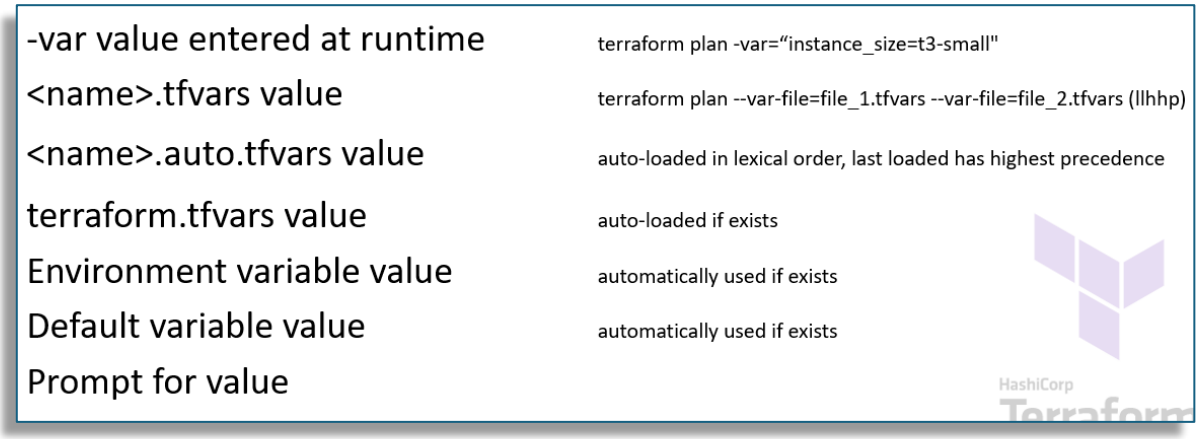
Whilst default values can be given to variables when they are declared, the actual value passed to the variable at runtime is configurable. If you do not assign a default value to the variable when it is declared and you do not supply a value by other means, then you will be prompted for the value.

Prompting for a variable value is Terraform's 'method of last resort' of obtaining a value. Input variables are more typically assigned values using command-line flags, variables files, or environment variables. These allow you to customize the configuration based on specific requirements. If all else fails though and Terraform still does not have a value to use, then you are prompted to supply one. If a variable value is declared in several places then there is a precedence order used that determines the value used.

By using input variables, you can make your Terraform configuration more dynamic and customizable, allowing users to provide values based on their specific needs. Output variables, on the other hand, enable you to export and share information or results from your Terraform infrastructure for further use or analysis.

Variable Precedence Order

Terraform follows a specific precedence order when evaluating variable values. This order determines which value is ultimately used for a variable if it is defined in multiple places. The precedence order, from highest to lowest, is as follows:



Command-line Flags:

Values provided via command-line flags using the `-var` option take the highest precedence. For example: **terraform apply -var="region=us-west-2"**

In this case, the value provided for the `region` variable via the command line will take precedence over any other value sources.

tfvars Files

Values specified in files ending **.tfvars** are the next level of precedence. Terraform automatically loads some of these files if present in the working directory, others must be specified...

<name>.tfvars Named tfvars files, if they exist, are not applied unless specified at the command line…

terraform plan/apply --var-file=<file_1>.tfvars --var-file=<file_2>.tfvars

If multiple named tfvars are included, then the last loaded has the highest priority if there are conflicting values.

<name>.auto.tfvars If these files exist then values in them are loaded automatically during plan and apply operations. If multiple of these files exist then they are loaded in lexical name order; **a.auto.tfvars** before **b.auto.tfvars** etc, the last loaded having highest priority if there are conflicting values.

terraform.tfvars If it exists, values in this file are loaded automatically during plan and apply operations.

If value conflicts exist, then **named.tfvars** take precedence over **auto.tfvars** and these take precedence over **terraform.tfvars**.

Environment Variables

Terraform automatically reads environment variables prefixed with `TF_VAR_` and matches them to corresponding variables in the configuration. For example, an environment variable `TF_VAR_region` can be used to set the value for the `region` variable.

Terraform Configuration defaults

Values specified directly within the Terraform configuration file (`*.tf`) come next in the precedence order. Variable values defined within the configuration itself can be set using the `default` attribute or without a default.

In this case, the default value specified within the configuration serves as a fallback if no other value is provided.

Prompt for value

If no value is assigned to the variable by any of the methods described, then terraform will generate a prompt, asking for a value to be input.

By following this precedence order, Terraform allows for flexibility in defining and overriding variable values based on specific use cases or environments. It provides multiple options to customize variable values and allows users to specify values at diverse levels, providing greater control over the configuration behaviour.

Output Variables

Output variables allow you to define values that are derived from the resources created or managed by Terraform. They provide a way to export information or results from your infrastructure to be used by other systems, modules, or external tools.

Output variables are defined using the `output` block within your Terraform configuration. You specify the name of the output variable and the value you want to expose. These values can be accessed after Terraform applies the configuration, and they can be used for further processing or for providing information to other systems.

In this example, `**priv_ip_add**` is an output variable that exposes the public IP address of an AWS EC2 instance. The value is derived from the `**private_ip**` attribute of the `**aws_instance.myec2**` resource

Output variables are useful for providing information to external systems or scripts, for passing values between modules, or for displaying results after applying Terraform configurations.

Understanding Functions in Terraform

Introduction

Functions in Terraform enable dynamic data manipulation, conditional logic, and efficient resource configuration. They allow you to transform values, construct complex configurations, and handle conditional scenarios without hardcoding values. Terraform evaluates these functions at runtime to produce the desired output.

Function Syntax

Terraform functions follow a simple and consistent syntax:

- **Structure**: A function name is followed by parentheses containing one or more arguments.
- **Usage**: Functions return values that can be:
    - Assigned to variables
    - Used directly in configurations
    - Nested within other functions

Example:

```
output "greeting" {
  value = concat("Hello", " ", "World!")
}
```

This example concatenates multiple string values into a single output.

How Functions Work in Terraform

Terraform functions accept one or more arguments, which can be:

- **Constants**: Directly provided static values.
- **Variables**: Dynamically assigned inputs.
- **Nested Functions**: Outputs of one function used as inputs for another.

Example:

```
variable "name" {
  default = "Terraform"
}

output "uppercase_name" {
  value = upper(var.name)
}
```

This example uses the upper function to convert the name variable's value to uppercase.

Variable Interpolation

You can use interpolation to insert the values of variables into strings or expressions within your configuration.

Interpolation is achieved using the **${...}** syntax. In this example, the ${var.name} interpolation is used to insert the value of the name variable into the instance tag value.

Example:

```
variable "name" {
  default = "Terraform"
}

resource "aws_instance" "example" {
  tags = {
    Name = "${var.name}-instance"
  }
}
```

This example dynamically inserts the name variable into the tags block of an AWS instance resource.

Note: Interpolation is now only required for expressions *inside* a string.

Function Return Values

Terraform functions return values that can be:

- Assigned to variables
- Used as inputs for other functions
- Embedded in resource definitions

Types of Terraform Functions

Terraform provides a variety of built-in functions for different operations.

String Functions

These functions manipulate string values:

- **concat**: Concatenates multiple strings or lists.
- **join**: Combines list elements into a single string with a delimiter.
- **format**: Formats strings with placeholders.

Example:

```
output "formatted_string" {
  value = format("Hello, %s!", "Terraform")
}
```

This example would produce output of “Hello, Terraform!”

List and Map Functions

These functions handle lists and maps:

- **element**: Retrieves an item from a list by index.
- **contains**: Checks if a list contains a specific value.
- **lookup**: Retrieves a value from a map by key.

Example:

```
variable "names" {
  default = ["Alice", "Bob", "Charlie"]
}
output "second_name" {
  value = element(var.names, 1)
}
```

This example would produce output of “Bob”

Regular Expression Functions

- **regex**: Matches a pattern in a string.

Example1:

```
output "matched_value" {
  value = regex("([a-z]+)", "terraform123")
}
```

The example applies the regular expression ([a-z]+) to the string "terraform123". This pattern captures one or more lowercase letters at the beginning of the string. Since "terraform123" starts with "terraform", the output is "terraform"

Example2:

```
output "matched_values" {
  value = regexall("[a-z]+", "terraform123code")
}
```

If multiple groups exist in the regex, Terraform only returns the first capture group. To extract multiple matches, consider using regexall. This example would return ["terraform", "code"]

Conditional Functions

Terraform includes functions for conditional evaluation:

- **coalesce**: Returns the first non-null value from a list of arguments.

- **Conditional Expressions**: Terraform does not have an explicit **if** function but uses conditional expressions.

Example:

```
variable "env" {
  default = "prod"
}

output "environment_type" {
  value = var.env == "prod" ? "Production" : "Development"
}
```

This example evaluates whether the environment is "prod" and returns the appropriate label.

Why Use Functions in Terraform?

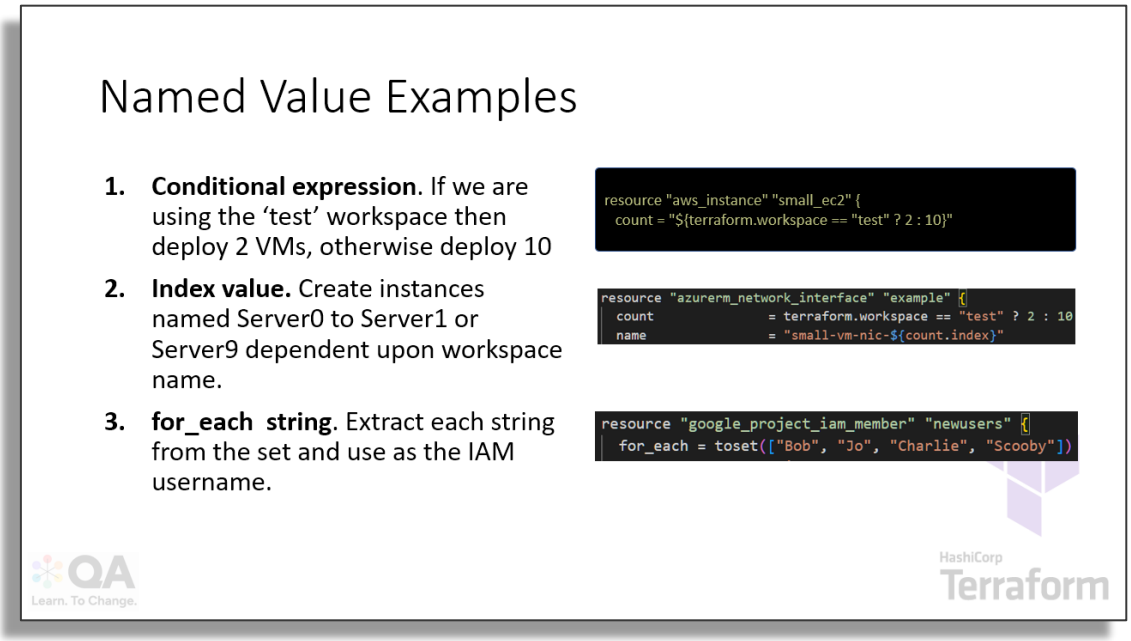
Functions enhance Terraform configurations by:

- **Reducing Hardcoding**: Generate values dynamically instead of manually specifying them.
- **Improving Maintainability**: Simplify complex configurations by computing values.
- **Enhancing Flexibility**: Adapt configurations based on inputs or conditions.

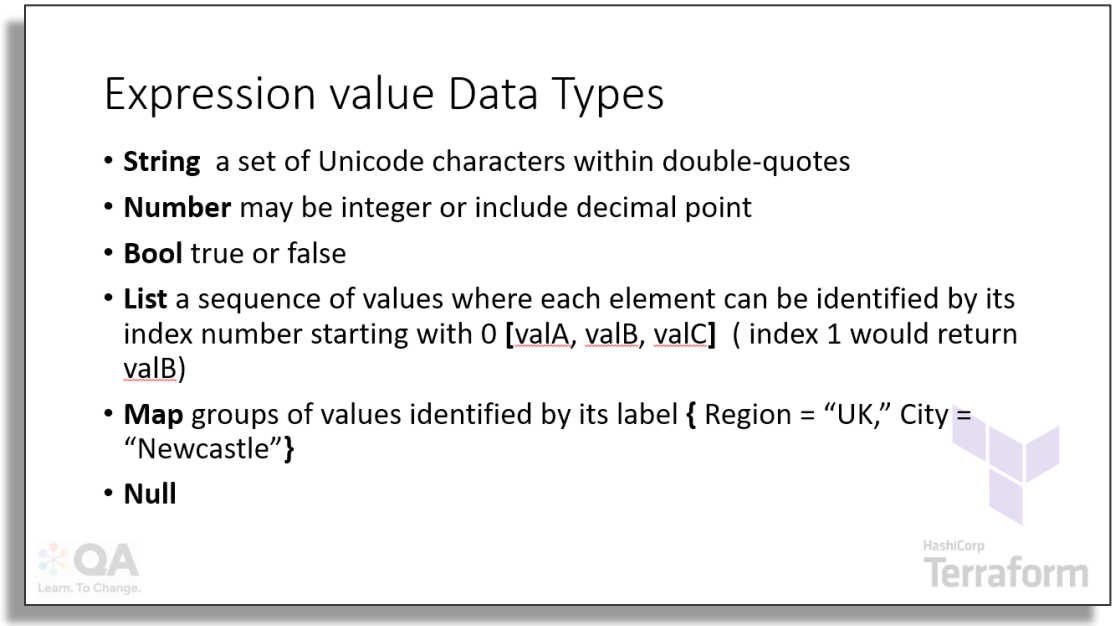
By leveraging Terraform functions, you can create more efficient, reusable, and scalable infrastructure configurations.

For a complete list of available functions, refer to the Terraform documentation.

Named Value Examples



These examples demonstrate how you can reference specific attributes or properties of named objects or resources within your Terraform configuration. By referencing these named values, you can establish relationships, dependencies, and dynamic behaviour between distinct parts of your infrastructure configuration.



In Terraform, expressions can evaluate to various data types based on the input values and operations involved. The data types available in Terraform include:

String: Represents a sequence of characters. Strings are typically used for textual data.

Number: Represents numerical values, including integers and floating-point numbers.

Boolean: Represents a logical value that can be either `true` or `false`. Boolean values are often used in conditional expressions.

List: Represents an ordered collection of values. Lists can contain elements of any data type, including other lists.

Map: Represents a collection of key-value pairs, where the keys and values can be of any data type.

Object (Struct): Represents a complex data structure that groups related values together. Objects are similar to maps but have a fixed structure defined by their attributes.

Tuple: Represents an ordered collection of values, similar to a list. However, unlike lists, tuples can contain elements of different data types.

Set: Represents an unordered collection of unique values. Sets can contain elements of any data type.

Null: Represents an absence of a value. It is used when a variable or expression has no assigned value.

Any: Represents a data type that can take any value. It is a flexible type used when the specific data type is unknown or can vary.

These data types allow Terraform to handle various kinds of values and perform operations based on their types. Terraform automatically infers the data types based on the context and the values used in expressions.

It is important to note that while Terraform has data types, it is a loosely typed language. This means that you do not need to explicitly declare or define the data types of variables. The types are inferred from the values assigned to them or used in expressions.

Understanding the data types in Terraform helps you work with variables, expressions, and resources more effectively, ensuring that the values are used appropriately in the context of your infrastructure configuration.

Iterations

Terraform supports **iteration** using constructs like **count** and **for_each**, which are particularly useful when managing multiples of a given resource. These constructs allow you to dynamically create multiple instances of a resource based on variables or data structures, making your configurations more flexible and scalable.

The count meta-argument creates a specified number of resource instances. For example, suppose you need three AWS EC2 instances. You can define a variable, instance_count, to hold the desired count and use count.index to create resources like example-instance-1, example-instance-2, and example-instance-3. A simple example configuration might look like this:

```hcl
variable "instance_count" {
  type    = number
  default = 3
}

resource "aws_instance" "example" {
  count         = var.instance_count
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"

  tags = {
    Name = "Example Instance ${count.index + 1}"
  }
}
```

In this example, the **count.index** dynamically increments with each iteration, allowing you to name and tag each instance uniquely. This approach is ideal when the number of resources is predetermined, and their differences can be expressed using sequential indices.

In contrast, the **for_each** meta-argument is used to iterate over a collection, such as a list or map, and create resources for each item. It's more flexible than count because it allows iteration over named collections. For instance, you can iterate through a map of S3 bucket names and their desired properties to create multiple S3 buckets with unique configurations. Using each.key and each.value, you can dynamically generate resource configurations.

```hcl
variable "buckets" {
  type    = map(string)
  default = {
    bucket1 = "us-west-1"
    bucket2 = "us-east-1"
    bucket3 = "eu-central-1"
  }
}

resource "aws_s3_bucket" "example" {
  for_each = var.buckets

  bucket = each.key
  acl    = "private"

  tags = {
    Name     = each.key
    Location = each.value
  }
}
```

This configuration creates three S3 buckets (bucket1, bucket2, bucket3) in different regions, as specified in the map. The for_each construct makes this possible by iterating over the provided map and dynamically setting the bucket names and tags based on each.key and each.value.

```hcl
variable "load_balancers" {
  type = list(string)
  default = ["web-elb", "app-elb", "db-elb"]
}

variable "tags" {
  type = map(string)
  default = {
    environment = "production"
    owner       = "devops-team"
  }
}

resource "aws_elb" "example" {
  for_each = toset(var.load_balancers)

  name               = each.key
  availability_zones = ["us-west-2a", "us-west-2b"]
  listener {
    instance_port     = 80
    instance_protocol = "HTTP"
    lb_port           = 80
    lb_protocol       = "HTTP"
  }

  tags = var.tags
}
```

For more advanced scenarios, you can combine these constructs for nested iterations. For instance, you might use for_each to iterate through a list of Elastic Load Balancers (ELBs) and apply tags from a map while associating each ELB with specific security groups. This ensures standardization while maintaining unique configurations for each resource. In this example, three Elastic Load Balancers (web-elb, app-elb, db-elb) are created with consistent tags, such as environment = "production", ensuring efficient resource management.

Workspaces

Terraform Community Edition, the locally run, free version of Terraform, has a workspace feature that allows you to manage multiple instances of your infrastructure within a single Terraform configuration. The `terraform workspace` command is used in this edition to interact with workspaces.

Workspaces allow you to manage multiple deployments or environments within the same configuration. Each workspace has its own separate state file and variable values, enabling you to isolate resources and manage configurations specific to each environment or deployment.

By using workspaces, you can switch between different deployments or environments easily, manage state separately, and maintain multiple instances of your infrastructure using a single Terraform configuration.

Switches or options available with the "terraform workspace" command:

terraform workspace new <name> Creates a new workspace with the specified name. For example, `**terraform workspace new staging**` creates a new workspace named "**staging**".

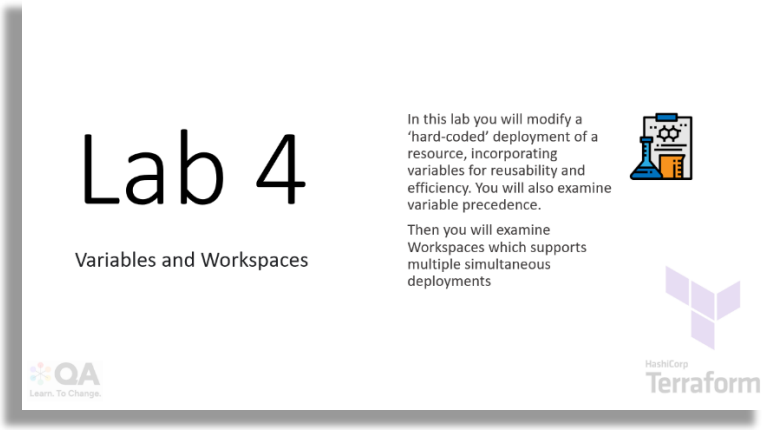
terraform workspace select <name> Switches to an existing workspace with the specified name. For example, `**terraform workspace select production**` switches to the "**production**" workspace.

terraform workspace list Lists all the available workspaces in the current Terraform configuration. It displays the names of the existing workspaces.

terraform workspace show Shows the name of the currently selected workspace.

terraform workspace delete <name> Deletes the specified workspace. It permanently removes the workspace and its associated state. Be cautious when using this command, as it cannot be undone. You cannot delete your currently workspace, nor the default workspace.

These workspace switches allow you to create switch between, list, show, and delete workspaces within your Terraform configuration. Each workspace maintains its own set of Terraform state files, allowing you to manage multiple environments or configurations independently. This separation enables you to deploy and manage infrastructure resources for distinct stages, such as development, staging, or production, using a single set of Terraform configuration files.



In this lab you will deploy a simple resource into your preferred cloud using a hard-coded terraform file. You will then modify the code, utilizing variables to it reuseable, and explore variable precedence.

Finally, you will utilize terraform workspaces to allow multiple simultaneous deployments.

5. State Management and Modules

Understanding State and the State File in Terraform

Terraform's state is a critical component that tracks and manages the resources it creates. Acting as a single source of truth, the state maintains a record of all resources in your infrastructure along with their configurations. Without the state, Terraform cannot effectively identify which resources have already been created, modified, or destroyed, making infrastructure management impossible.

The state file (terraform.tfstate) is a JSON file that stores this state information. It includes details about every managed resource, such as configurations, dependencies, and metadata. Terraform uses this file to compare the current state of resources with the desired configuration (defined in .tf files) and applies only the necessary updates. For instance, if a new resource is added, Terraform reads the state file to ensure only the new resource is created, leaving existing resources untouched.

State enables Terraform to efficiently manage resource metadata, such as IDs, properties, and relationships. This prevents duplicate resource creation and ensures changes are applied in the correct order by maintaining a dependency graph. By comparing the current state with the desired state, Terraform calculates precise updates, reducing errors and streamlining infrastructure management.

Managing the state file requires care due to its sensitive nature. It often contains confidential information like access keys, making secure storage essential. Remote backends with encryption and access controls are recommended for production environments. Directly editing the state file is discouraged as it can lead to inconsistencies or corruption. Instead, use Terraform commands such as terraform state for modifications. Additionally, while local backups are helpful, the state file should not be included in version control systems to protect sensitive data and avoid storage issues.

Local vs. Remote State Files

When managing infrastructure collaboratively, sharing the state file becomes critical. Terraform supports **remote backends** to centrally store state files, enabling features like state locking and team-wide consistency. Without a shared state, teams risk overwriting changes or creating conflicts. Below, we explore the key differences between local and remote state files.

Local State Files

By default, Terraform uses **local state files** to store infrastructure data. These files are created in the working directory as terraform.tfstate and are only accessible on the machine where Terraform runs. Local state files are suitable for single-person projects or environments where collaboration is unnecessary. The filename and location of the state file can also be customized to fit specific needs.

However, local state files have limitations in collaborative environments. They are not automatically synchronized or accessible to other team members, making them impractical for team-based projects. Additionally, local state files often contain sensitive information, such as credentials, making them unsuitable for inclusion in version control systems. Instead, for team or production use, remote backends should be considered.

Remote State Files

In collaborative or production environments, using a **remote backend** is strongly recommended. Remote backends store the state file in centralized locations such as AWS S3, Azure Blob Storage, Google Cloud Storage, or Terraform Cloud. This allows multiple users or systems to access and update the state file simultaneously, ensuring consistency across teams. Changes made by one team member are immediately reflected in the shared state, minimizing conflicts and errors.

Remote backends also support state locking mechanisms, preventing concurrent modifications to the state file. This ensures that only one user or process can update the state at a time, maintaining consistency and avoiding corruption. Additionally, many remote

backends offer versioning capabilities, enabling teams to track state changes and improve auditing.

Terraform supports several remote backend options tailored to specific infrastructure needs:

- **AWS S3 Backend**: Commonly used for AWS-based projects, the S3 backend stores the state file in an Amazon S3 bucket. It can integrate with DynamoDB for distributed state locking, ensuring only one user or process can modify the state file at a time.
- **Azure Blob Storage Backend**: Ideal for Azure environments, this backend provides centralized and secure state storage. It supports automatic locking to prevent concurrent updates.
- **Google Cloud Storage (GCS) Backend**: For teams using Google Cloud Platform, the GCS backend offers high availability and seamless integration with GCP services, along with built-in locking for consistent state management.

Choosing between local and remote state files depends on the complexity and collaboration requirements of your infrastructure. Local state files are sufficient for simple, single-user projects, offering an easy setup. However, for team-based or production environments, remote backends provide a robust solution with centralized storage, state locking, and enhanced collaboration. By selecting the appropriate backend and securing your state file, you can ensure efficient and reliable infrastructure management with Terraform.

Viewing the Terraform State File

The Terraform state file (terraform.tfstate) contains crucial information about the resources Terraform manages. Viewing the state file can help you understand the current state of your infrastructure, debug issues, or validate that changes were applied correctly. However, due to its sensitive nature, care must be taken when accessing it.

Direct File Access: The state file is stored in JSON format, making it readable with any text editor or JSON viewer. You can open the file directly to inspect its contents:

cat terraform.tfstate

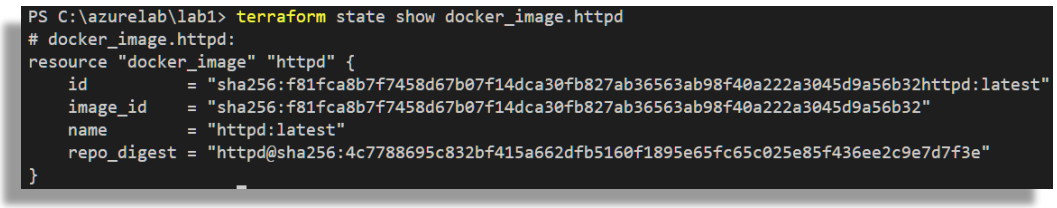
While this approach is straightforward, manually inspecting large state files can be cumbersome.

Using Terraform Commands: Terraform provides built-in commands to query and inspect the state file without opening it directly:

terraform state list: Lists all the resources tracked in the state file.

```
PS C:\azurelab\lab1> terraform state list
docker_container.webserver
docker_image.httpd
```

terraform state show <resource>: Displays detailed information about a specific resource in the state.



Remote Backends: If the state is stored in a remote backend, such as S3 or Azure Blob Storage, you cannot access the file directly. Instead, use Terraform commands like state list or state show to query the state. This ensures you interact with the latest state and avoid manual edits.

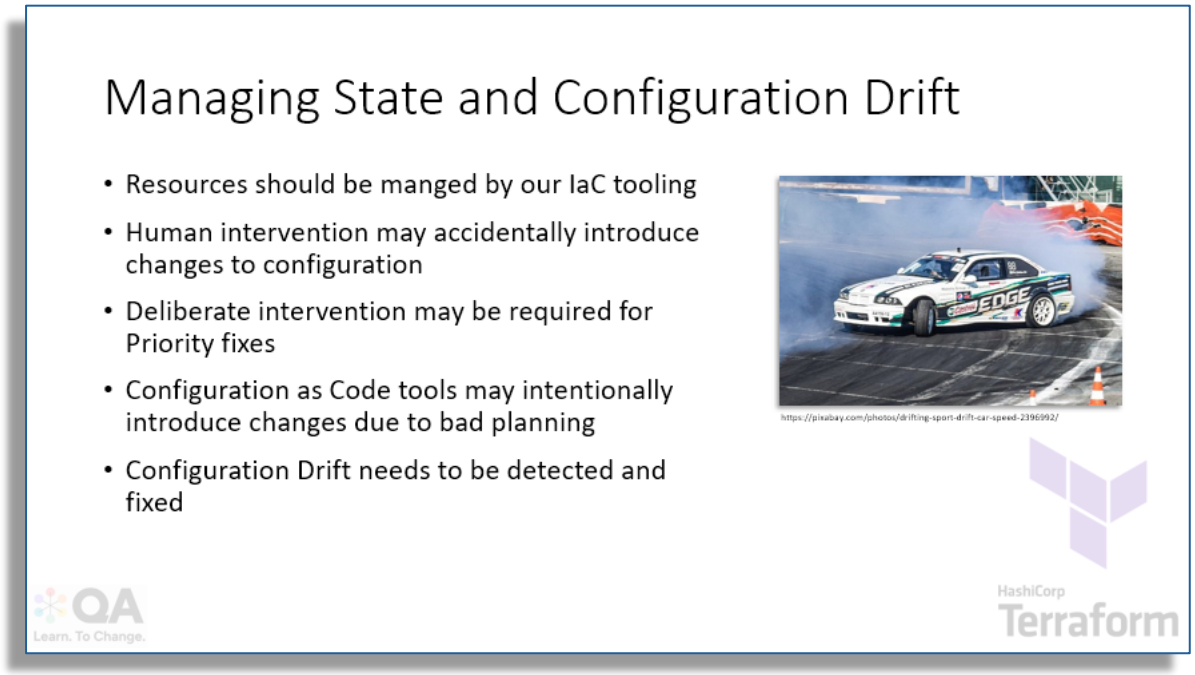
Backing Up and Restoring Terraform State

Terraform's state file is crucial for managing infrastructure, so regularly backing it up is essential to avoid disruptions. For local state files, backups can be created by simply copying the terraform.tfstate file to a secure location. A good practice is to use a timestamp in the backup filename, such as terraform.tfstate.backup.2025-01-10, to distinguish versions. Remote backends, such as AWS S3, Azure Blob Storage, or Google Cloud Storage, simplify the backup process by offering built-in versioning. Enabling versioning ensures previous versions of the state file are retained automatically, providing a reliable safety net. For teams using Terraform Cloud or Enterprise, backups are managed seamlessly with versioning and retention policies.

Restoring the state file involves replacing the current state with a backup when needed, such as in cases of corruption, accidental deletion, or misalignment. For local backends, this can be achieved by copying the backup file over the current state file. For remote backends, retrieve the desired backup version from the storage system and then overwrite the current state. Once restored, the state should be pushed back to the remote backend using Terraform commands like terraform state push. To ensure the restored state is correct, running terraform plan and terraform refresh will validate and reconcile the state with the actual infrastructure.

It is important to proceed with caution when restoring a state file. Ensure no Terraform operations, such as apply or plan, are running during the restore to avoid inconsistencies. Always verify the integrity of the backup before restoring to prevent applying incorrect or outdated changes. To minimize the likelihood of needing a restore, use automated backups, remote backends with locking and versioning, and secure storage for sensitive state files. By following these practices, you can confidently manage Terraform state and quickly recover from any unexpected issues.

Managing State and Configuration Drift



State drift, also known as configuration drift, refers to a situation where the actual state of deployed infrastructure resources diverges from the expected state defined in the Terraform configuration and tracked in the Terraform state file.

State drift can occur due to several reasons, such as manual changes made directly to resources outside of Terraform, changes made through other infrastructure management tools, or modifications made directly in the cloud provider's console.

State drift is a problem because it introduces inconsistencies between the desired state and the actual state of the infrastructure. This can lead to issues such as resource conflicts, unexpected behaviour, and difficulty in managing and maintaining the infrastructure.

Detecting State Drift

Terraform provides a `terraform plan` command that compares the desired state from the configuration with the actual state stored in the state file. Any differences or inconsistencies detected during this comparison indicate state drift.

Impact of State Drift

State drift can cause Terraform to incorrectly manage or operate on resources. It may lead to resource conflicts when attempting to modify or delete resources that have been changed externally. Additionally, Terraform may not be aware of the changes made outside its control, which can impact the stability and integrity of the infrastructure.

Managing State Drift

When state drift occurs, it is important to bring the actual state back in line with the desired state. This can be done by using the `terraform import` command to import the drifted

resources into Terraform's management and then modifying the configuration to reflect the current state.

Preventing State Drift

To minimize the occurrence of state drift, it is recommended to follow infrastructure-as-code practices consistently. Infrastructure changes should be made through Terraform and tracked in version control. Manual changes outside of Terraform should be avoided or minimized. Regular communication and coordination among team members can also help prevent unintentional drift.

State Locking

Terraform supports state locking mechanisms for remote state files to prevent concurrent modifications. State locking ensures that only one user or process can make changes to the state at a time, reducing the risk of conflicting changes and state drift.

It is important to address state drift promptly and resolve any inconsistencies between the desired and actual state. Regularly monitoring and auditing the infrastructure state can help identify and rectify drift, ensuring that Terraform remains the source of truth for your infrastructure configuration.

Terraform taint and replace

In Terraform, `taint` and `replace` are two commands used to manage and update resources in your infrastructure. Here is an overview of each command:

Taint

The `**terraform taint**` command is used to mark a resource as "tainted," indicating that

Terraform should consider it as needing replacement on the next `terraform apply` run. Tainting a resource helps trigger the recreation of that resource, ensuring that its state matches the desired configuration.

Usage: `terraform taint <resource_address>`

Example: `terraform taint aws_instance.example`

When you taint a resource, Terraform will plan to recreate that resource on the next

`terraform apply` run, regardless of any changes in the resource's configuration. This is useful in scenarios where you want to force the recreation of a resource due to external changes or to apply a new configuration.

Replace

The `**terraform apply -replace**` command is used in scenarios where a resource creation has failed or has been left in an incomplete state, and you want to reattempt its creation. It signals Terraform to treat the resource as if it needs replacement, even if no changes have been made to its configuration.

However, it is important to note that the `-replace` flag is not typically used for regular resource updates or modifications. Instead, Terraform determines whether a resource needs replacement based on changes in the resource configuration or other dependencies.

To use the `-replace` flag with `terraform apply`, you would run the following command: **terraform apply -replace=<resource_address>** where `<resource_address>` represents the address of the specific resource you want to replace.

Exercise caution when using the `-replace` flag, as it forces Terraform to reattempt the creation of the specified resource, potentially resulting in loss of data or unintended consequences. It is recommended to use this flag only in exceptional cases where a resource has failed to be created and needs to be retried without making any configuration changes.

Terraform Import

Overview

Terraform's import command allows you to bring existing infrastructure resources under Terraform's management. This is particularly useful when you have resources created outside Terraform and want to integrate them into your Terraform workflow. By importing, Terraform reads the current state of an existing resource and maps it to a corresponding resource configuration in your Terraform state file. However, it is important to note that importing only updates the state file—it does not automatically create a fully configured Terraform resource block.

Key Concepts of Terraform Import

Import Syntax

The syntax for the import command is:

> **terraform import [options] ADDRESS ID**
>
> ADDRESS: Specifies the Terraform resource address, typically in the format TYPE.NAME (e.g., aws_instance.my_vm).
>
> ID: Represents the unique identifier of the resource being imported, such as an instance ID or resource name.

Resource Discovery

Before importing, you must identify the correct resource address and unique identifier. This ensures Terraform maps the imported resource to the right configuration block.

State Management

Terraform updates the state file to include the imported resource. This ensures Terraform can track and manage the resource alongside other resources in your infrastructure.

Limitations and Considerations

Import Limitations

Not all types of resource support import operations. The ability to import depends on the provider and resource type. Complex resources with multiple dependencies may have restrictions or require additional steps.

Manual Configuration

Importing does not generate a complete resource configuration. After importing, you must manually define the resource block in your Terraform configuration files to reflect the resource's state accurately. This includes specifying attributes and parameters based on the imported resource.

Configuration Drift Risks

Incorrect imports can lead to drift between Terraform's state and the actual infrastructure. Always review and validate the imported resource's configuration to ensure alignment.

Testing

Perform import operations in non-production environments first to validate the process. Always take backups of your state file and infrastructure before importing into production.

Tools to Assist with Importing

While Terraform's import command is a powerful tool, there are additional tools and resources that can simplify and enhance the process:

Terraformer

An open-source tool developed by Google to automate resource imports across major cloud providers (e.g., AWS, Azure, GCP). It generates Terraform configuration files for existing resources, easing the integration into Terraform workflows.

Terraform Importer

A CLI tool developed by Gruntwork (now part of HashiCorp) to automate the generation of terraform import commands for specific resources. Designed to simplify and speed up the import process.

Provider Documentation

Cloud providers often provide specific guides for importing resources into Terraform, offering step-by-step instructions, examples, and important considerations.

Community Tools and Scripts

The Terraform community has developed various scripts, tools, and reusable modules available on platforms like GitHub and Terraform Registry. These resources can address complex scenarios or specific resource types.

Best Practices for Terraform Import

Backups and Validation

Always back up your Terraform state file and infrastructure before importing resources. Validate the import process in a staging or non-production environment before applying it to production.

Careful Review

After importing, thoroughly review and manually update the resource configuration to ensure it reflects the existing infrastructure accurately.

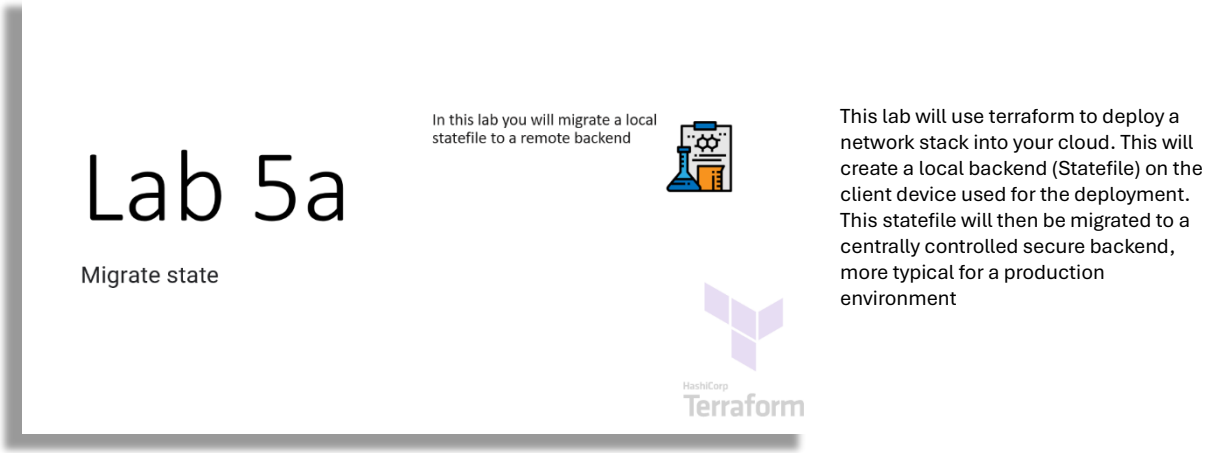
Caution with Third-Party Tools

When using tools like Terraformer or scripts from the community, validate their compatibility with your Terraform version, cloud provider, and infrastructure setup.

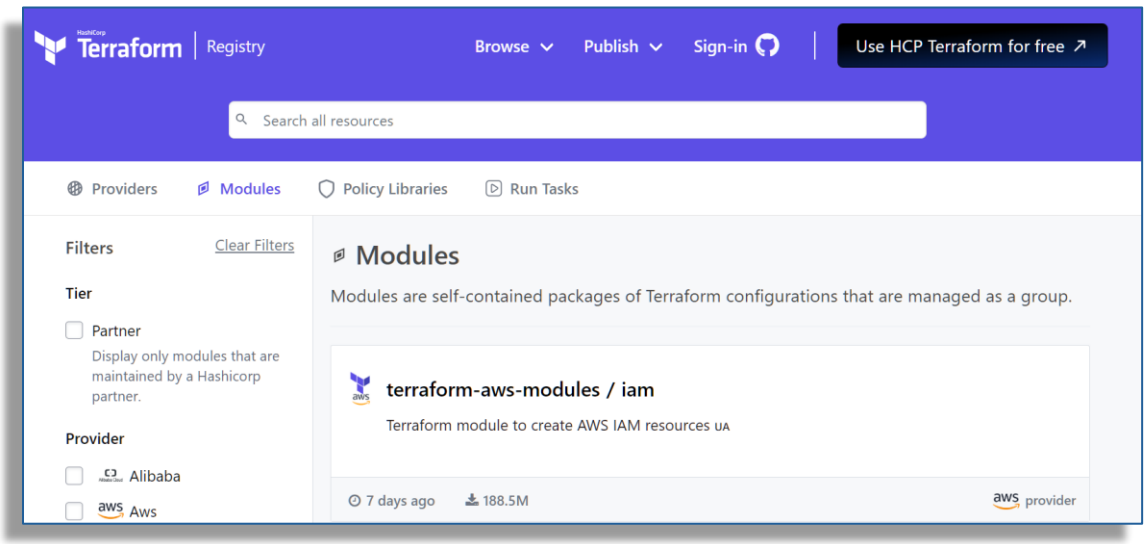
Testing

Test the import operation to minimize risks and avoid potential configuration drift.

Terraform import is a valuable feature for integrating existing resources into Terraform-managed infrastructure. While the process requires some manual intervention, tools like Terraformer and Terraform Importer can simplify the effort. By following best practices, such as taking backups, testing in non-production environments, and thoroughly reviewing configurations, you can ensure a smooth and reliable transition to Terraform-managed infrastructure. For further guidance, consult the official Terraform documentation on import.



Terraform Modules



Terraform modules are self-contained packages of configurations that enable you to organize, encapsulate, and reuse infrastructure code effectively. By breaking down your infrastructure into modular components, Terraform modules make your configurations easier to manage, maintain, and scale. Modules allow you to define reusable building blocks for consistent infrastructure deployment across projects and teams.

Root and Child Modules

Root Module

The root module is the entry point of your Terraform configuration, located at the top level of your project directory. It defines the overall infrastructure by declaring resources and calling other modules. Root modules typically include input variables, output values, provider configurations, and other top-level elements. They coordinate and manage the infrastructure by referencing child modules or directly defining resources.

Child Modules

Child modules represent reusable components of your infrastructure configuration, defined in separate directories. These modules encapsulate specific logical units, such as a VPC, database, or application. Child modules can be reused multiple times within the same root module or across different root modules, making your infrastructure code modular, maintainable, and scalable.

Child modules accept **input variables** to customize their behaviour and define **output values** to share information with the calling module. By abstracting complex infrastructure patterns into reusable components, child modules promote consistency, code reuse, and collaboration. For example, a team might build a library of reusable child modules for shared use across projects.

Modular Components and Benefits

Encapsulation and Abstraction

Terraform modules abstract implementation details by encapsulating a collection of resources into a single unit. Modules define **inputs**, **outputs**, and configurations, hiding the internal workings of the resources. This abstraction simplifies the usage of complex infrastructure components and makes configurations more reusable.

Input Variables

Modules accept input variables to customize their behaviour. These variables allow users to parameterize configurations, passing values from the calling module to the child module. Input variables make modules flexible, enabling them to adapt to different deployments or environments.

Output Values

Modules define output values to expose specific information or attributes about the resources they manage. These outputs provide a way to communicate resource details, such as IP addresses, connection strings, or IDs, back to the calling configuration. This makes it easier to integrate modules into complex configurations.

Composition and Nesting

Modules can be nested and composed to build higher-level abstractions. For example, a root module might call a child module for a VPC, which itself calls another module for subnets. This composition allows you to construct complex infrastructures by combining smaller, reusable modules.

Terraform Module Registry

Terraform provides a public module registry (https://registry.terraform.io/browse/modules) where users can discover and share pre-built modules created by the community. The registry promotes collaboration and best practices by offering reusable solutions for common infrastructure patterns. Examples include modules for setting up AWS VPCs, Azure Kubernetes clusters, or GCP storage buckets.

Organizations can create private modules to share reusable components internally. Private modules provide a controlled way to encapsulate and manage infrastructure configurations, ensuring consistency and security across teams.

Modules Best Practices and Benefits

Reusability

Modules enable the creation of reusable infrastructure components, reducing code duplication and ensuring consistency across deployments. For example, a module defining a load balancer can be reused across multiple environments with different configurations.

Abstraction

By abstracting the complexity of resource management, modules simplify configurations, making it easier to understand and work with your infrastructure.

Collaboration

Modules foster collaboration by providing shared components that teams can use across projects. This encourages adherence to best practices and standardization.

Maintainability

Modules encapsulate resources, making it easier to manage infrastructure configurations. They provide clear interfaces for inputs and outputs, reducing complexity and improving maintainability.

File Structure Example

A typical Terraform project structure with modules might look like this:

```
root
 ├ main.tf              # root module
 ├ variables.tf
 ├ outputs.tf
 ├/vpc                  # vpc module
   ├ main.tf
   ├ variables.tf
   └ outputs.tf
 ├/app                  # app module
   ├ main.tf
   ├ variables.tf
   └ outputs.tf
```

In this structure, the root module references child modules to organize the infrastructure into logical units.

Terraform modules are a powerful feature that enhances the scalability, maintainability, and reusability of your infrastructure code. By encapsulating resources into reusable components, modules simplify complex configurations, promote consistency, and enable efficient collaboration. Whether using public modules from the registry or creating private ones within your organization, modules help streamline infrastructure management and ensure reliable deployments.

Terraform Logging

Introduction

Logging is an essential feature in Terraform that helps you monitor execution, debug issues, and gain insights into the lifecycle of Terraform commands. Terraform provides several

logging options, enabling you to capture various levels of detail about its execution. These options range from simple console outputs to more advanced configurations, such as environment variables and debug flags.

Common Logging Options in Terraform

Standard Output (STDOUT)

By default, Terraform outputs log messages directly to the console. These messages include informational notices, warnings, and errors, providing immediate feedback during command execution. This is the simplest way to monitor Terraform's behaviour, as no additional configuration is required.

Log File

Terraform log messages can be redirected to a file for persistent storage. This is particularly useful for reviewing logs after command execution or when running Terraform in automated pipelines. For example, the following command stores log messages in a file named terraform.log:

terraform apply > terraform.log

The log file will contain the same messages typically displayed in the console, ensuring you have a permanent record of the execution.

Advanced Logging Configurations

Environment Variables

Terraform supports environment variables for fine-tuning its logging behaviour. Two key variables are:

TF_LOG: Controls the verbosity level of log messages. Accepted values are:

- TRACE: The most detailed level, including all execution details.
- DEBUG: Detailed information useful for debugging.
- INFO: General operational messages.
- WARN: Warnings that might need attention.
- ERROR: Error messages indicating issues in execution. For example, enabling debug-level logging can be done by setting: **export TF_LOG=DEBUG**

TF_LOG_PATH: Specifies a file path where Terraform should write its log messages. This provides an alternative to standard output redirection **export TF_LOG_PATH=/path/to/terraform.log**

Using these variables together allows detailed and persistent logging for Terraform operations.

Debug Output

Terraform commands include a -debug flag to provide additional execution details. This flag outputs verbose logs, including information like HTTP requests, responses, and resource interactions. For example: **terraform apply -debug**

Debug output is invaluable for diagnosing complex issues and understanding the internal workings of Terraform during execution.

Provider-Specific Logging

Some Terraform providers offer additional logging options tailored to their functionality. Provider documentation often details how to enable these options, allowing you to capture provider-specific debug information, such as API requests or internal errors.

Best Practices for Using Terraform Logs

Troubleshooting and Monitoring

Logs are a primary tool for diagnosing issues, such as failed resource creations or configuration errors. Use appropriate verbosity levels (e.g., DEBUG or TRACE) during debugging and revert to lower levels (e.g., INFO or WARN) for routine operations.

Log File Management

Store logs in organized directories and rotate them regularly to avoid excessive storage use. Consider using tools to analyze and aggregate logs for long-term monitoring.

Security Considerations

Terraform logs may contain sensitive information, such as credentials, resource IDs, or configuration details. Always secure log files with proper access controls and avoid sharing them publicly. Mask sensitive details if logs need to be shared for debugging.

Choosing the Right Logging Approach

The choice of logging configuration depends on your specific needs:

- Use **standard output** for quick, real-time monitoring during local execution.
- Leverage **log files** for post-execution analysis or when running Terraform in automated workflows.
- Configure **debug output** or TF_LOG=DEBUG for detailed troubleshooting of complex issues.
- Explore **provider-specific logging** for deeper insights into provider behaviour.

By using these options effectively, you can gain a comprehensive understanding of Terraform's execution and maintain a robust monitoring and debugging workflow.

Lab5b

Creating resources utilizing Modules

In this lab you utilize a pre-provisioned network module to deploy resources to distribute web traffic across virtual machines

HashiCorp Terraform

This lab aims to demonstrate how modules are used when deploying complex resource stacks using Terraform.

Attributes of resources created in modules are not exposed to the root module, so output files are used.

Root module resources that are dependent of module-level resources must be configured to reference their attributes as exposed in the output files.

6. Cloud Storage Management

Introduction

Object storage is designed to store large volumes of unstructured data, such as media files, documents, backups, and logs. It provides scalability, durability, and cost-effective storage, making it ideal for applications that require efficient handling of massive datasets. Most major cloud providers offer object storage solutions, including AWS S3, Azure Blob Storage, and Google Cloud Storage (GCS). While their implementations vary slightly, they share common concepts such as storage accounts, containers/buckets, objects (blobs), access tiers, and lifecycle policies.

Overview of Object Storage in AWS, Azure, and Google Cloud

Each cloud provider has its own implementation of object storage, but they follow a similar structure:

Concept	AWS S3	Azure Blob Storage	Google Cloud Storage (GCS)
Storage Account	S3 Bucket	Storage Account	GCS Bucket
Container	N/A (S3 uses flat storage)	Storage Container	N/A (GCS uses flat storage)
Object Types	Objects (Standard, Intelligent-Tiering, Glacier)	Blobs (Block, Append, Page)	Objects (Standard, Nearline, Coldline, Archive)
Access Tiers	Standard, IA, Glacier	Hot, Cool, Archive	Standard, Nearline, Coldline, Archive
Replication	Cross-Region, Same-Region	LRS, GRS, RA-GRS	Multi-Region, Dual-Region, Regional

Despite these differences, Terraform provides a consistent way to define and manage object storage across these providers.

Key Features of Cloud Object Storage

All cloud object storage services support:

- Scalability & Durability – Designed to handle petabytes of data with automatic redundancy.
- Access Tiers – Optimize storage costs based on how frequently data is accessed.
    - Frequent Access: AWS S3 Standard, Azure Hot, GCP Standard
    - Infrequent Access: AWS S3 IA, Azure Cool, GCP Nearline
    - Archival Storage: AWS Glacier, Azure Archive, GCP Archive
- Replication – Ensures high availability by copying data across regions.
    - AWS: Cross-Region Replication (CRR) and Same-Region Replication (SRR)
    - Azure: LRS (Local), GRS (Geo-Redundant), RA-GRS (Read-Access GRS)
    - GCP: Multi-Region, Dual-Region, Regional storage classes

Provisioning Object Storage with Terraform

Terraform enables cloud-agnostic infrastructure provisioning. Below are equivalent Terraform configurations for AWS S3, Azure Blob Storage, and GCP Cloud Storage.

Google Cloud Storage Example

```
resource "google_storage_bucket" "example" {
 name = "example-bucket"
 location = "US"
 storage_class = "STANDARD"
 versioning {
  enabled = true
 }
}
```

This defines a Google Cloud Storage bucket with versioning enabled.

Managing Object Storage Configurations Across Clouds

All providers offer additional configuration options, including versioning, access policies, and lifecycle management.

Versioning

Versioning helps track changes and prevent accidental deletions.

- **AWS S3**: Enabled using aws_s3_bucket_versioning
- **Azure Blob Storage**: Managed through azurerm_storage_management_policy

- **Google Cloud Storage**: Controlled via versioning { enabled = true } in the bucket configuration

Access Policies and Security

Securing object storage is critical. SAS tokens (Azure), IAM policies (AWS), and Signed URLs (GCP) are commonly used for controlled access.

Example: AWS S3 Bucket Policy for Restricted Access

```
resource "aws_s3_bucket_policy" "example" {
 bucket = aws_s3_bucket.example.id
 policy = <<EOF
{
 "Version": "2012-10-17",
 "Statement": [
  {
   "Effect": "Allow",
   "Principal": "*",
   "Action": "s3:GetObject",
   "Resource": "arn:aws:s3:::example-bucket/*"
  }
 ]
}
EOF
}
```

This policy allows public read access to objects in an S3 bucket.

Lifecycle Management

Object storage lifecycle policies help optimize costs by automatically transitioning objects between storage classes or deleting them after a set period.

GCP Cloud Storage Lifecycle Rule: Delete Objects After 30 Days

```
resource "google_storage_bucket" "example" {
 name        = "example-bucket"
 location    = "US"
 storage_class = "STANDARD"

 lifecycle_rule {
  action {
   type = "Delete"
  }
  condition {
```

```
    age = 30
  }
 }
}
```

Lab 6 Managing Object Storage

The lab will guide you through provisioning and managing cloud object storage.

7. Implementing Checks and Validations in Terraform

Introduction

Ensuring infrastructure as code (IaC) deployments adhere to organizational policies and best practices is crucial in cloud environments. Terraform provides mechanisms to enforce validation rules at different stages of the deployment process, preventing misconfigurations and ensuring compliance. This chapter explores input validation, precondition and postcondition checks, post-deployment validation, and external policy enforcement using HashiCorp Sentinel.

Input Validation in Terraform

Input validation ensures that variable values meet specific criteria before Terraform executes a plan or applies changes. It prevents invalid configurations from being deployed, reducing the risk of failure and misconfiguration.

Example: Validating Naming Conventions

Terraform allows validation rules to be applied to input variables using the validation block:

```
variable "resource_group_name" {
 description = "Name of the resource group."
 type        = string
 validation {
  condition     = can(regex("^RG-[1-6]$", var.resource_group_name))
  error_message = "Resource group name must be RG- followed by a number
from 1 to 6."
 }
}
```

This ensures that the resource group name follows the required naming convention (RG-1 to RG-6).

Example: Enforcing Region Selection

```
variable "location" {
 description = "Cloud region for resource deployment."
 type        = string
 validation {
  condition    = contains(["us-east-1", "us-west-2", "europe-west1"],
var.location)
  error_message = "Location must be one of the predefined regions."
 }
}
```

This condition restricts users to specific cloud regions, ensuring compliance with organizational policies.

Preconditions and Postconditions in Terraform

Terraform allows additional validation through precondition and postcondition blocks, enabling checks at runtime to enforce constraints before and after resource creation.

Example: Ensuring a Storage Account Follows Naming Standards

```
resource "azurerm_storage_account" "example" {
 name                = var.storage_account_name
 resource_group_name     = var.resource_group_name
 location            = var.location
 account_tier          = "Standard"
 account_replication_type = "LRS"

 lifecycle {
  precondition {
   condition    = startswith(var.storage_account_name, "stg")
   error_message = "Storage account name must start with 'stg'."
  }
  postcondition {
   condition    = length(self.name) <= 24
   error_message = "Storage account name must be 24 characters or fewer."
  }
 }
}
```

This configuration enforces naming policies and prevents Terraform from creating resources that violate those rules.

Post-Deployment Validation

Validation should not stop at Terraform's internal mechanisms. Some misconfigurations may only become evident after deployment. Cloud platforms provide policy enforcement tools that can be used in combination with Terraform.

Example: Using Policy for Post-Deployment Checks

Cloud policy mechanisms can be used to enforce resource compliance after deployment. For example, Azure Policy can check for compliance and, if a resource violates policy, the deployment is rejected or flagged for review.

```
{
 "properties": {
  "displayName": "Enforce Storage Account Naming Policy",
  "policyType": "Custom",
  "mode": "All",
  "parameters": {},
  "policyRule": {
   "if": {
    "allOf": [
     {
      "field": "name",
      "notLike": "stg*"
     },
     {
      "field": "type",
      "equals": "Microsoft.Storage/storageAccounts"
     }
    ]
   },
   "then": {
    "effect": "deny"
   }
  }
 }
}
```

This ensures that all storage accounts conform to naming conventions, even if Terraform validation was bypassed.

Enforcing Policy with HashiCorp Sentinel

HashiCorp Sentinel is a policy-as-code framework that provides fine-grained control over Terraform deployments. It allows organizations to define rules that must be met before Terraform applies any changes.

Example: Sentinel Policy to Restrict VM Naming

```
import "tfplan/v2" as tfplan
main = rule {
 all tfplan.resources.azurerm_virtual_machine as _, instances {
  all instances as _, r {
   r.name matches "^VM[1-6]$"
  }
 }
}
```

This policy ensures that virtual machines are named according to the VM1 to VM6 format.

Preventing Policy Bypass

One challenge of using validation within Terraform files is that developers can modify the rules. To prevent policy bypass, organizations should enforce governance through:

- **Terraform Cloud/Enterprise Sentinel Policies**: Ensuring policies run at the organization level.
- **Azure Policy, AWS Config Rules, or GCP Organization Policies**: Blocking non-compliant resources at the cloud provider level.
- **Code Reviews and Change Management**: Implementing peer reviews before merging infrastructure changes.

Best Practise

Implementing checks and validations in Terraform improves security, consistency, and governance of cloud deployments. By combining Terraform's built-in validation, cloud-native policy enforcement, and external policy frameworks like Sentinel, organizations can ensure that their infrastructure remains compliant and robust.

- Use Terraform variable validation to enforce input constraints.
- Apply precondition and postcondition checks to validate resources dynamically.
- Implement post-deployment checks using cloud-native policies (Azure Policy, AWS Config, GCP Organization Policies).

- If available, use Sentinel for policy-as-code enforcement at the organization level.
- Combine validation strategies to prevent misconfigurations and unauthorized changes.

8. Automating Terraform with Pipelines

There are several editions of Terraform.

Terraform Community Edition

Terraform Community Edition is the version of Terraform that we have been discussing thus far. It is available under the Business Source License (BSL, or BUSL) v1.1 and can be downloaded and used by anyone. Terraform provides the core functionality and features for infrastructure provisioning and configuration management.

Terraform Enterprise

Terraform Enterprise is the self-hosted, enterprise-grade version of Terraform. It offers the same features as Terraform Cloud but allows organizations to deploy and manage the Terraform infrastructure within their own private environment. Terraform Enterprise provides enhanced security, compliance, and customization options tailored for enterprise-scale infrastructure deployments.

Terraform Cloud and Terraform Enterprise are both commercial offerings provided by HashiCorp. They provide additional features, services, and support compared to Terraform OSS, making them suitable for larger teams, organizations, or those with specific compliance and security requirements.

Terraform Cloud

Terraform Cloud is the managed service offered by HashiCorp, the company behind Terraform. It provides a collaborative and cloud-based environment for managing infrastructure with Terraform. Terraform Cloud offers features such as remote state management, collaboration and team workflows, version control integration, and more. It provides additional functionality on top of Terraform Community Edition and simplifies certain aspects of infrastructure management.

Remote State Management

Terraform Cloud provides a centralized location to store and manage Terraform state files. Storing state remotely allows for easier collaboration, versioning, and sharing of

infrastructure state among team members. It also helps prevent issues related to managing and synchronising local state files.

Collaboration and Teamwork

Terraform Cloud enables multiple team members to work together on infrastructure projects. It provides features such as access controls, team management, and collaboration workflows to facilitate coordination and collaboration among team members. This allows teams to efficiently work together on infrastructure deployments.

Version Control Integration

Terraform Cloud integrates with popular version control systems like Git, allowing you to connect your infrastructure code repositories. This integration enables versioning, change tracking, and rollback capabilities for your infrastructure configurations. It also facilitates collaboration by allowing teams to work with familiar version control workflows.

Policy Enforcement and Governance

Terraform Cloud offers policy enforcement and governance features to ensure compliance and control over infrastructure deployments. It provides policy sets that allow you to define and enforce rules and best practices across your infrastructure. This helps maintain consistency, security, and compliance with organizational standards.

Workspaces and Environment Management

Workspaces in Terraform Cloud allow you to create and manage multiple instances of your infrastructure deployments. Workspaces provide isolation, environment separation, and variable management for different deployments or environments (such as development, staging, production). This helps streamline infrastructure management across distinct stages and environments.

Remote Operations and Runs

Terraform Cloud provides a web-based interface for executing Terraform operations, such as plan and apply, remotely. It offers visibility into the progress, logs, and results of these operations. This helps teams to centrally manage and monitor their infrastructure deployments.

Integration Ecosystem

Terraform Cloud integrates with various other tools and services, including CI/CD pipelines, notification systems, and HashiCorp's Consul for service discovery and dynamic configurations. These integrations enhance automation and enable end-to-end infrastructure management workflows.

Terraform Cloud simplifies infrastructure management, improves collaboration, and enhances governance capabilities for teams working with Terraform. It is designed to provide a scalable and secure platform for managing infrastructure deployments and can be

particularly beneficial for larger teams, organizations, and projects that require robust collaboration and governance features.

For more detailed information on Terraform Cloud, including pricing, features, and getting started guides, visiting the official Terraform Cloud website: https://www.terraform.io/cloud.

Terraform Cloud Workspaces

Terraform Cloud workspaces are a key feature that allows you to organize and manage your infrastructure deployments within the Terraform Cloud platform. Workspaces provide a way to isolate and control the lifecycle of your infrastructure configurations, variables, and state files.

Isolation and Environment Separation

Each workspace in Terraform Cloud represents a separate environment or deployment of your infrastructure. This could include different stages such as development, staging, and production, or any other logical separation you require. Workspaces allow you to keep your configurations, variables, and state files distinct and separate for each environment.

Variable Management

Workspaces provide a mechanism to manage variables specific to each environment. You can define input variables within a workspace and assign values to them. This allows you to customize the behaviour of your infrastructure configuration for each environment without modifying the underlying code.

State Management

Terraform Cloud stores the state files of each workspace in a secure and centralized manner. The state files contain the current state of the infrastructure, including resource IDs, metadata, and other details. By storing the state remotely, workspaces ensure easy access, versioning, and sharing of the state among team members.

Collaboration and Access Control

Workspaces facilitate collaboration by allowing multiple team members to work on the same infrastructure project simultaneously. Terraform Cloud provides access controls and team management features to define granular permissions for workspace access. This enables you to control who can view, modify, or manage the infrastructure configurations within each workspace.

Version Control Integration

Terraform Cloud integrates with popular version control systems, such as Git, allowing you to connect your infrastructure code repositories. This integration enables versioning, change tracking, and rollback capabilities for your infrastructure configurations within each

workspace. It also facilitates collaboration by allowing teams to work with familiar version control workflows.

Workspace Variables and Modules

Workspaces in Terraform Cloud can have their own set of variables and modules, allowing you to define workspace-specific inputs and reuse modular components across different environments. This enhances flexibility, maintainability, and reusability of your infrastructure code.

Run History and Logging

Terraform Cloud maintains a run history for each workspace, capturing details of executed Terraform commands, such as plan and apply. This includes logs, outputs, and results of each run, providing visibility into the changes made to the infrastructure over time.

By leveraging workspaces in Terraform Cloud, you can effectively manage and organize your infrastructure deployments, control variables and configurations per environment, collaborate with team members, and maintain a centralized and secure infrastructure state. Workspaces help streamline infrastructure management and provide a structured approach to managing different environments or deployments within your organization.

For more information on using workspaces in Terraform Cloud, refer to the official Terraform Cloud documentation: https://www.terraform.io/docs/cloud/workspaces/index.html

Terraform CI/CD Pipelines

Terraform CI/CD (Continuous Integration/Continuous Deployment) Pipelining refers to the process of automating the build, testing, and deployment of Terraform infrastructure using a CI/CD pipeline. It enables you to manage infrastructure changes efficiently and reliably throughout the development lifecycle, ensuring consistent and controlled deployments.

Continuous Integration (CI):

CI focuses on automating the build and testing phases of infrastructure changes. With Terraform, CI involves verifying the correctness of configuration files, checking for syntax errors, and running validation checks. CI tools, such as Jenkins or GitLab CI/CD can be used to trigger these automated checks whenever changes are pushed to version control.

Continuous Deployment (CD):

CD involves automating the deployment of infrastructure changes to various environments, such as development, staging, and production. CD pipelines allow you to define stages, such as plan, apply, and destroy, and control the execution of these stages based on triggers or manual approvals. CD tools, tools, such as Jenkins or GitLab CI/CD can help orchestrate the deployment process, manage infrastructure state, and handle environment specific configurations.

Infrastructure as Code (IaC):

Terraforms infrastructure-as-code approach fits well within a CI/CD pipeline. Infrastructure changes are defined in Terraform configuration files, which are versioned, tested, and deployed automatically. This ensures consistent, repeatable, and auditable infrastructure deployments.

GitOps

GitOps is a common approach to managing infrastructure with version control systems like Git. In the context of Terraform CI/CD, GitOps involves storing the Terraform configuration files in a Git repository and using Git-based workflows to trigger CI/CD pipelines. Changes made to the infrastructure configuration are reviewed, tested, and merged through pull requests, triggering the automated CI/CD pipeline.

Pipeline Orchestration and Integration:

CI/CD tools provide a range of capabilities for pipeline orchestration and integration with other services. They allow you to define build stages, execute tests, manage secrets and variables, integrate with version control, trigger deployments based on events, and notify stakeholders about pipeline status.

By incorporating Terraform into your CI/CD pipeline, you can achieve benefits such as:

- Faster and more reliable infrastructure changes
- Improved collaboration among development and operations teams
- Consistent and auditable deployments across environments
- Versioning and tracking of infrastructure changes
- Automated testing and validation of Terraform configurations
- Controlled and secure handling of infrastructure secrets and credentials

The specific implementation of Terraform CI/CD pipelines can vary based on the CI/CD tooling you choose, your infrastructure requirements, and the desired workflows. CI/CD tools like Jenkins or GitLab CI/CD, others provide plugins, integrations, and documentation to help you configure and customize your Terraform CI/CD pipelines.

It is important to consider best practices, security measures, and testing strategies while setting up and managing Terraform CI/CD pipelines to ensure efficient and reliable infrastructure deployments.

Jenkins Pipelines

Jenkins is an open-source automation server that has become a cornerstone of DevOps workflows, particularly for continuous integration and delivery—often referred to as CI/CD. It is designed to automate repetitive tasks, such as building, testing, and deploying software, making it a powerful tool for streamlining workflows.

One of Jenkins' greatest strengths is its extensibility. With thousands of plugins available, it can integrate with a wide range of tools and platforms, including Terraform, AWS, GitHub, Docker, and more. This makes Jenkins highly adaptable to various project requirements and environments.

By automating processes, Jenkins ensures consistency and repeatability, which are crucial in modern infrastructure management. Instead of manually running scripts or deploying code, Jenkins allows you to define workflows in a simple, structured way using Jenkinsfile. This customizability also supports scalability, enabling teams to handle large projects with distributed builds across multiple agents.

When we apply Jenkins to Terraform workflows, the benefits are immediate. Jenkins can automate the three key steps in Terraform deployment: initializing the environment with **terraform init**, previewing changes with **terraform plan**, and deploying resources with **terraform apply**. This automation reduces the manual effort involved in managing infrastructure as code, ensuring that deployments are faster, more consistent, and less prone to human error.

Jenkins not only enhances efficiency but also provides seamless integration with Terraform and AWS, making it an ideal choice for automating infrastructure deployments. Whether you are managing a single resource or a complex multi-tier architecture, Jenkins ensures that your Terraform workflows are reliable, repeatable, and easy to manage.

In essence, Jenkins transforms how we approach Terraform automation by bringing speed, precision, and scalability to the table. It is the glue that binds together your code, your infrastructure, and your deployment pipelines, enabling you to focus on building and innovating, rather than getting bogged down in manual tasks.

Jenkins Basics

To effectively use Jenkins, it is important to understand a few key concepts that form the foundation of its functionality. Let us start with **Jobs**, which are the basic building blocks of Jenkins. A job represents a task, or a series of tasks Jenkins will execute, such as running a script, building an application, or deploying infrastructure. These jobs can be simple, standalone tasks or part of a larger automated workflow.

That brings us to **Pipelines**, one of Jenkins' most powerful features. Pipelines are scripts that define a sequence of stages and steps, outlining exactly what Jenkins should do and in what order. They allow you to automate entire workflows—from pulling code from a repository to

deploying changes to a production environment. These pipelines are written in a file called a Jenkinsfile, which serves as a blueprint for your automation.

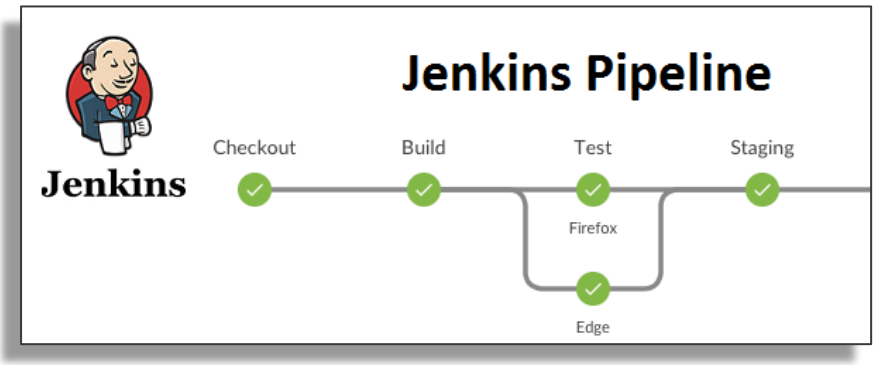
Within a pipeline, you will encounter **Stages**, which help break down workflows into logical steps. For example, in a Terraform pipeline, you might have stages for initialization, planning, and applying infrastructure changes. This structure ensures that each step of the process is clear, manageable, and easy to troubleshoot.

Plugins are another essential part of Jenkins. They extend Jenkins' capabilities, enabling it to integrate seamlessly with external tools. For our purposes, the **Terraform plugin** is critical, as it allows Jenkins to execute Terraform commands directly. Similarly, the **AWS plugin** is used to securely manage AWS credentials, ensuring Jenkins can interact with AWS services safely and efficiently.

Setting up Jenkins involves a few straightforward steps. Typically, you will start by installing Jenkins on an instance, such as an Ubuntu EC2 server in AWS. Once installed, you can configure Jenkins to work with your specific environment. This includes adding credentials for AWS, so Jenkins can authenticate securely, and installing plugins like Terraform to enable seamless execution of infrastructure workflows.

By understanding these core concepts—Jobs, Pipelines, Stages, and Plugins—you will have the foundation needed to use Jenkins effectively. In the next steps, we will see how these elements come together to create a simple Terraform pipeline that automates infrastructure deployment.

Building a Simple Terraform Pipeline



Now that we understand the basics of Jenkins, let us dive into building a simple Terraform pipeline. At the heart of this pipeline is the Jenkinsfile, which serves as a roadmap for automating your Terraform workflow. The Jenkinsfile defines a series of stages, each corresponding to a specific task in the deployment process.

The first stage is **Initialization**. This is where we run the terraform init command to set up the Terraform working directory. This step ensures that Terraform has access to the necessary backend configurations, such as remote state in an S3 bucket, and downloads any required provider plugins.

The second stage is **Planning**. Here, Jenkins executes terraform plan, which generates an execution plan. This step provides a detailed preview of what changes Terraform will make to your infrastructure. It is a critical checkpoint to validate that the changes align with your expectations before moving forward.

Finally, we have the **Applying** stage. This is where Jenkins runs terraform apply to implement the changes in your AWS environment. To add a layer of control, we include an approval step in the pipeline. This ensures that a human must review and confirm the plan before any changes are deployed, providing an extra safeguard for your infrastructure.

Let us look at an example Jenkinsfile to see how these stages are implemented. The file is straightforward:

```
pipeline {
  agent any
  stages {
    stage('Init') {
      steps {
        sh 'terraform init'
      }
    }
    stage('Plan') {
      steps {
        sh 'terraform plan'
      }
    }
    stage('Apply') {
      steps {
        input "Approve deployment?"
        sh 'terraform apply -auto-approve'
      }
    }
  }
}
```

The agent any line allows the pipeline to run on any available Jenkins agent.

Each stage—Init, Plan, and Apply—contains steps that execute the corresponding Terraform commands.

The Apply stage includes an input step to pause the pipeline and wait for user approval.

This structure not only simplifies Terraform workflows but also ensures that deployments are consistent and repeatable. With a few lines of configuration in a Jenkinsfile, you can automate complex infrastructure processes, reduce errors, and save time.

Lab 8 Checks and Validations

This final lab aims to demonstrate how businesses can control terraform deployments through the use of change control and release mechanisms. In this case Github will be used for change control combined with Jenkins which will deploy changes once approved.