



QA Terraform in Practice

Terraforming Cloud Platforms

Course Introduction



Module 1. Introduction to Infrastructure as Code.....	5
Infrastructure as Code.....	5
Introducing Terraform.....	7
Comparing Vendor Tools to Terraform.....	8
Terraform Basics	15
Module 2 - Terraform Foundations	18
Introduction	18
Installing Terraform.....	18
Authentication	19
Terraform files	21
Terraform init.....	22
Terraform plan.....	23
Terraform apply	24
Parallelism.....	26
Implicit vs explicit dependencies	27
Module 3. Terraform Providers.....	28
Introduction to Providers.....	28
Local Provider.....	30
Public Cloud Providers.....	31
Module 4. Variables and Functions.....	33
Introduction to Variables	33
Input Variables	34
Output Variables	36
Functions.....	37
References to Named Values.....	39
Module 5. Templates, State and Drift	43
Introduction	43
Importing, Recreating and Terminating Resources.....	44
OSS Workspaces	47
Local vs. Remote State files.....	48
Templates.....	52
Managing State and Configuration Drift.....	55
Logging	58
Module6. Terraform Pipelining.....	60

Introduction to Terraform Cloud.....	61
Terraform CI/CD	64
GitOps	65
Jenkins Pipelines	66

Agenda

- Introductions
- The Learning environment
- Courseware and lab access
- Course Modules

Courseware and Labs

- Your instructor will take you through the process of accessing your course material and your AWS, Azure or Google Cloud lab environment.

Course Modules

1. Introducing Infrastructure as Code and Terraform
2. Terraform Foundations
3. Terraform Providers
4. Variables & Functions
5. Templates, State and Drift
6. Terraform Pipelining



Module 1. Introduction to Infrastructure as Code



Module 1

Introducing Infrastructure as Code and Terraform



Agenda

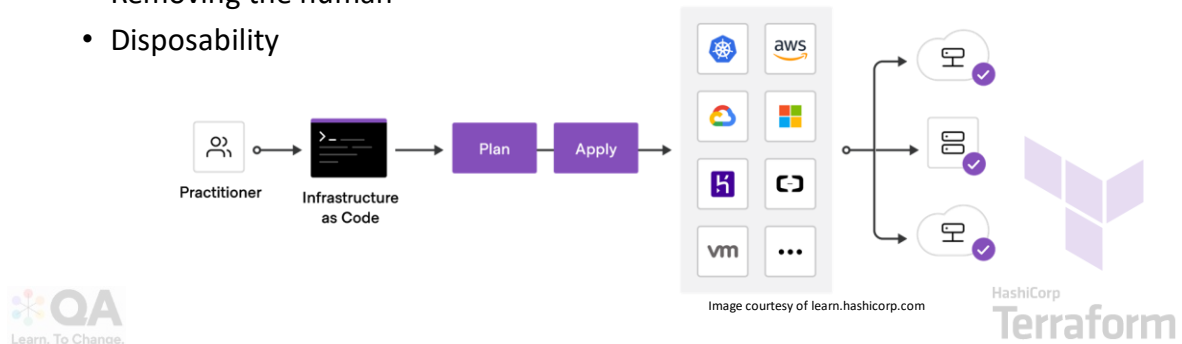
- Infrastructure as Code
- Introducing Terraform
- Comparing Vendor tools to Terraform
- Terraform Basics
- Lab 1 Creating a simple resource using Terraform



Infrastructure as Code

Infrastructure as Code

- Traditional Datacentre
- Software defined X
- Removing the human
- Disposability



In the traditional datacentre approach, infrastructure was built by physically visiting the location, installing the necessary hardware, and manually configuring the software. However, with the evolution of 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 represented as a software-defined service layer.

This software-defined layer, whether for networking, storage, or compute, allows us to treat it like any other software component. You can build, configure, and manipulate it programmatically, like how 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 managed services. IaC enables you to bring automation, consistency, and repeatability to the deployment and management of diverse infrastructure components.

Introducing Terraform

Introducing Terraform

- Infrastructure as Code tool
- Can manage on-premise and cloud resources
- Is cloud agnostic
- Supports both JSON and HashiCorp Configuration Language (HCL)



There are various Infrastructure as Code (IaC) tools available, and they can be categorized into vendor-specific and vendor-independent options. Vendor-specific tools are provided by cloud vendors like AWS, Azure, and GCP, while vendor-independent tools, such as HashiCorp Terraform, are not tied to any particular cloud provider.

Terraform from Hashicorp is a leading vendor-independent IaC tool, meaning it is not bound to a specific vendor. It allows you to define infrastructure both on-premises and in the cloud, and it is not limited to a particular cloud vendor's proprietary tool or language. This flexibility provides a significant advantage for organizations 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.

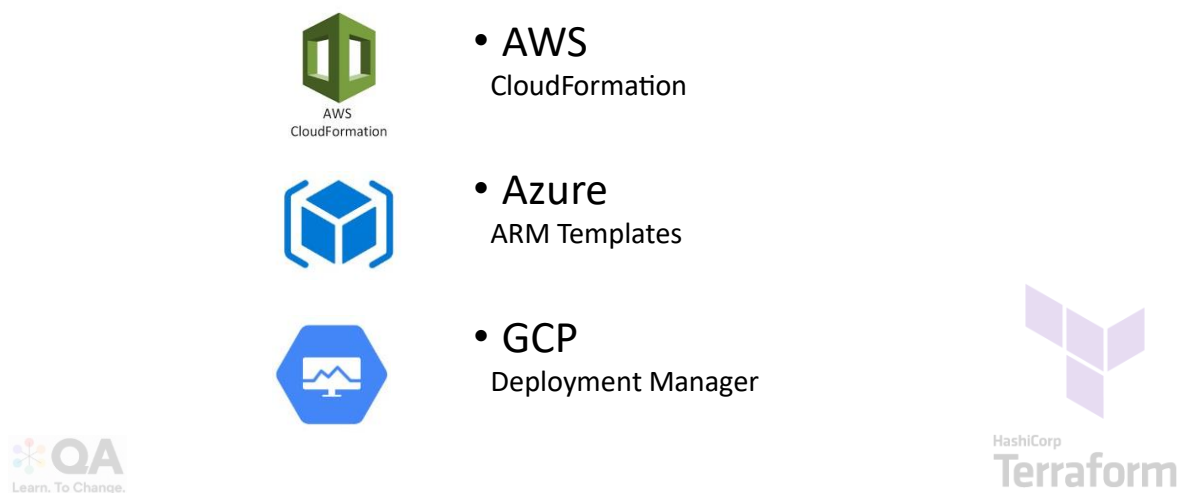
The configuration files created with Terraform can be easily stored, shared, tested, deployed, and reused. This promotes efficient collaboration among team members, enables version

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.

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.

Comparing Vendor Tools to Terraform

Comparing Vendor tools to Terraform



AWS CloudFormation

AWS CloudFormation is a service provided by Amazon Web Services (AWS) that enables you to define and provision infrastructure resources in a declarative manner using templates. It allows you to model your entire infrastructure stack as code, called a CloudFormation template, and then create, update, or delete the stack as a single unit. Here is an introduction to AWS CloudFormation:

Infrastructure as Code (IaC)

CloudFormation embraces the concept of Infrastructure as Code, where you define your infrastructure resources, configurations, and relationships using a JSON or YAML template file. This approach enables versioning, reuse, and consistency in managing your infrastructure deployments.

Declarative Templates

CloudFormation templates describe the desired state of your infrastructure, rather than specifying a sequence of steps to achieve that state. You define resources, properties, dependencies, and parameters in the template, and CloudFormation takes care of provisioning and managing those resources.

Resource Provisioning and Management

CloudFormation automates the provisioning and management of resources across multiple AWS services, including compute instances, networking, storage, databases, security groups, load balancers, and more. It handles the underlying orchestration and configuration required to create, update, or delete these resources.

Stack Management

CloudFormation organizes your infrastructure resources into logical groups called stacks. You create a stack by providing a CloudFormation template, and CloudFormation handles the creation and management of the resources defined in the template. Stacks can be easily managed, updated, or deleted as a single unit.

Change Management and Drift Detection

CloudFormation provides change management capabilities, allowing you to make updates to your infrastructure by modifying the template and applying changes through stack updates. It also helps detect drift, which refers to manual or unintentional changes made to resources outside of CloudFormation and provides options to bring the stack back to the desired state.

Integration and Extensibility

CloudFormation integrates with other AWS services, such as AWS Identity and Access Management (IAM), AWS CloudTrail for logging and auditing, and AWS Service Catalog for creating and managing portfolios of approved resources. It also supports custom resource types and third-party extensions through AWS Lambda.

Rollback and Recovery

CloudFormation enables safe and reliable deployments by providing automatic rollback in case of failures during stack updates. It maintains a record of previous stack configurations, making it easier to roll back to a known good state if needed.

Pros of AWS CloudFormation:

Infrastructure as Code

CloudFormation allows you to manage infrastructure using code, promoting versioning, automation, and collaboration.

Automation and Consistency

CloudFormation automates resource provisioning, leading to consistent deployments across environments.

Scalability

CloudFormation supports the creation and management of complex infrastructure setups with ease. CloudFormation integrates well with other AWS services and has a rich ecosystem of community-contributed templates.

Cons of AWS CloudFormation

Learning Curve

Understanding the CloudFormation template syntax and mastering the service can have a learning curve.

Template Complexity

CloudFormation templates can become complex for larger infrastructure deployments, requiring careful design and organization.

Lack of Support for Some Services: While CloudFormation supports a wide range of AWS services, some new or niche services might have limited or delayed CloudFormation support.

Azure ARM Templates

Azure Resource Manager (ARM) templates are the infrastructure-as-code solution provided by Microsoft Azure. They enable you to define and deploy Azure resources and configurations in a declarative manner. Here is an introduction to Azure ARM templates:

Infrastructure as Code (IaC): ARM templates enable you to describe and provision Azure resources using code. They follow a declarative approach where you define the desired state of your infrastructure, and Azure Resource Manager handles the provisioning and management of those resources.

Declarative Templates: ARM templates are JSON files that define the resources, properties, dependencies, and configurations required for your Azure infrastructure. These templates allow you to express the infrastructure requirements and relationships in a structured and version-controlled manner.

Resource Provisioning and Management: ARM templates facilitate the automated provisioning and management of Azure resources. They can describe a wide range of resources, including virtual machines, storage accounts, virtual networks, databases, load balancers, and more. ARM handles the orchestration and sequencing of resource deployment based on the template specifications.

Resource Group Management: ARM templates help organize resources into logical groups called resource groups. A resource group acts as a container for related resources,

allowing you to manage, update, or delete multiple resources collectively. Resource groups provide a level of isolation and organization for your Azure deployments.

Template Functions and Expressions: ARM templates support functions and expressions that enable dynamic values, calculations, and conditional logic within the template. These functions allow you to define parameterized templates, reference resource properties, and perform operations based on input values or conditions.

Parameterization and Variable Usage: ARM templates allow you to parameterize different aspects of the template, such as resource names, sizes, and configurations. This parameterization provides flexibility and reusability, allowing you to customize deployments for different environments or scenarios. Variables can also be used to define reusable values within the template.

Integration and Extensibility: ARM templates integrate with various Azure services and features. They can utilize features like Azure Policy for governance, Azure Key Vault for secure credential storage, Azure Functions for custom operations, and Azure DevOps for CI/CD pipelines. Additionally, Azure offers various pre-built ARM templates for popular services and scenarios, which can be used as a starting point.

Pros of Azure ARM templates:

Infrastructure as Code: ARM templates allow you to define and manage Azure resources using code, promoting automation, version control, and collaboration.

Azure Integration: ARM templates are specifically designed for Azure and offer seamless integration with Azure services, providing a comprehensive solution for managing Azure infrastructure.

Scalability: ARM templates can handle complex deployments with multiple resources and dependencies, making them suitable for scaling infrastructure deployments.

Azure Marketplace: ARM templates can be used to define solutions that can be published on the Azure Marketplace, allowing others to deploy and use your solution.

Cons of Azure ARM templates:

Template Complexity: ARM templates can become complex for large deployments or intricate resource relationships, requiring careful design, and understanding of the JSON syntax.

Limited Cross-Cloud Portability: ARM templates are Azure-specific and may not be easily portable to other cloud providers without modifications.

Learning Curve: Familiarizing yourself with the ARM template structure and functions may require some initial learning and understanding.

State Tracking

Azure Resource Manager (ARM) templates do not directly track state like Terraform does. Terraform has an explicit concept of state, which is a record of the resources managed by Terraform and their current state. In the case of Azure ARM templates, they are primarily focused on describing the desired state of your Azure infrastructure. They define the resources and their configurations but do not manage the state of those resources directly. ARM templates are essentially a set of instructions for Azure Resource Manager to provision and manage resources based on the template specifications.

The state tracking and management of resources provisioned by ARM templates are handled by Azure Resource Manager itself. When you deploy an ARM template, Azure Resource Manager takes care of creating, updating, or deleting the resources specified in the template. It maintains the state of those resources internally and ensures that the deployed resources align with the desired state defined in the ARM template.

Azure Resource Manager keeps track of the resource provisioning and management operations, allowing you to view the deployment history, monitor the status of deployments, and track any errors or changes made to the resources. You can access this information through Azure portal, Azure CLI, or Azure PowerShell.

If you need to manage the state of Azure resources and track changes over time, Azure provides additional services and features like Azure Resource Manager templates deployments, Azure Automation State Configuration, and Azure Policy, which can help with configuration management, compliance, and governance.

Google Cloud Deployment Manager

Google Cloud Deployment Manager is a service provided by Google Cloud Platform (GCP) that allows you to define, configure, and deploy your infrastructure resources using configuration files called Deployment Manager templates. Here is an introduction to Google Deployment Manager:

Infrastructure as Code (IaC): Google Deployment Manager enables you to define and manage your infrastructure resources as code. Using YAML or Python templates, you can express the desired state of your infrastructure and manage it in a version-controlled manner. This approach promotes automation, reproducibility, and collaboration.

Declarative Templates: Deployment Manager templates describe the resources, properties, dependencies, and configurations required for your GCP infrastructure. They allow you to express the infrastructure requirements and relationships using a structured and human readable format. Templates can be parameterized for flexibility across environments.

Resource Provisioning and Management: With Google Deployment Manager, you can automate the provisioning and management of GCP resources. It supports a wide range of

resources, including compute instances, networks, storage, databases, load balancers, and more. Deployment Manager handles the orchestration and configuration necessary to create, update, or delete these resources.

Stack Management: Deployment Manager organizes resources into logical groups called deployments or stacks. A deployment represents a collection of resources that are managed together as a unit. Stacks enable you to manage and operate multiple resources collectively, making it easier to create, update, or delete sets of resources in a consistent manner.

Integration with GCP Services: Deployment Manager integrates seamlessly with various Google Cloud services and features. It can leverage service accounts, IAM roles, Cloud Storage, Cloud DNS, Cloud SQL, and other GCP services to provision and manage your infrastructure. You can define resources specific to GCP services in your Deployment Manager templates.

Templating and Parameterization: Deployment Manager templates support templating and parameterization, allowing you to define reusable components and customize deployments for different environments or use cases. You can use Jinja2 templating in YAML or Python templates to dynamically generate resource configurations and set values based on input parameters.

Versioning and Auditing: Deployment Manager tracks and manages the configuration of your infrastructure deployments. It provides versioning capabilities, allowing you to keep track of changes made to your Deployment Manager templates over time. This auditability helps with troubleshooting, compliance, and change management.

Pros of Google Deployment Manager:

Infrastructure as Code: Deployment Manager enables infrastructure management through code, promoting automation, version control, and collaboration.

Native Integration: Deployment Manager is tightly integrated with other Google Cloud services, providing seamless management of GCP resources.

Scalability: Deployment Manager supports complex deployments and scaling of resources within the GCP ecosystem.

Google Cloud Marketplace: Deployment Manager templates can be published on the Google Cloud Marketplace, enabling easy sharing and deployment of solutions.

Cons of Google Deployment Manager:

Learning Curve: Understanding the Deployment Manager template syntax and mastering the service may require initial learning and familiarity with GCP concepts.

Limited Cross-Cloud Portability: Deployment Manager is specific to GCP and may require modifications for deployments on other cloud providers.

Template Complexity: Managing large and intricate infrastructure deployments with Deployment Manager templates may require careful design and organization.

State Tracking

Google Deployment Manager (DM) manages the state of your deployments and resources internally. When you create or update a deployment using DM, it automatically tracks the state and configuration of the deployed resources. Here's how DM deals with state:

Deployment State: DM maintains the state of your deployments, which represents the current configuration and status of the resources within a deployment. The deployment state includes information about the resources, their properties, relationships, and metadata.

Resource Tracking: DM keeps track of the state of individual resources within a deployment. It knows which resources were created, modified, or deleted as part of a deployment. DM uses this tracking information to determine the actions required during updates or deletions.

Atomic Updates: DM ensures that deployments are updated atomically, meaning that either the entire update succeeds, or it is rolled back to the previous state. This atomicity ensures that the deployment remains consistent and avoids leaving resources in an intermediate or inconsistent state.

Rollback and Recovery: If an update fails or encounters errors, DM can automatically roll back the deployment to its previous state, undoing the changes made during the update. This rollback mechanism helps prevent issues caused by failed updates and maintains the integrity of the deployment.

Resource Diffing and Updates: During an update, DM compares the desired state defined in the deployment configuration with the current state of the resources. It identifies differences or changes between the desired and current states and performs the necessary actions to bring the resources to the desired state.

Incremental Updates: DM performs incremental updates, meaning it only modifies or creates the necessary resources and properties that have changed. This minimizes the impact on existing resources and reduces the time taken for updates.

Synchronization and Consistency: DM synchronizes the desired state specified in the deployment configuration with the actual state of the resources. It ensures that the resources maintain the desired configuration and properties and applies any necessary changes to align them with the desired state.

It is important to note that while DM manages the state of your deployments and resources, it does not expose the state in the same way as tools like Terraform. The state is maintained internally by DM and is not directly accessible or managed by the user. However, you can use DM to query the state of a deployment and obtain information about the resources and their current configurations.

By handling the state management internally, Google Deployment Manager simplifies the process of managing the state and configuration of your resources. It abstracts away the complexities of state handling, allowing you to focus on defining and deploying your infrastructure using DM templates.

Terraform Basics

Terraform Basics

The Terraform Workflow:

- Scope** - What are you trying to achieve?
- Write** - Author infrastructure as code.
- Init** - Initialize the code stack.
- Plan** - Preview changes before applying.
- Apply** - Provision reproducible infrastructure.



```
1 terraform {
2   required_providers {
3     docker = {
4       source = "kreuzwerker/docker"
5       version = "~> 2.21.0"
6     }
7   }
8 }
9
10 provider "docker" {}
11
12 resource "docker_image" "Apache_web" {
13   name = "httpd:latest"
14   keep_locally = false
15 }
16
17 resource "docker_container" "web" {
18   image = docker_image.Apache_web.image_id
19   name = "Web-demo"
20   ports {
21     internal = 80
22     external = 8080
23   }
24 }
```

`terraform init`, **`terraform plan`**, and **`terraform apply`** are key commands in Terraform that help you initialize, plan, and apply your infrastructure changes. Here is an overview of each command:

terraform init

The **`terraform init`** command is used to initialize a new or existing Terraform configuration. It downloads the necessary provider plugins and sets up the backend configuration. This command is typically run once in a Terraform project's root directory to prepare the environment for Terraform operations.

During initialization, Terraform retrieves the required provider plugins based on the providers specified in your configuration. It also initializes the backend, which defines where Terraform stores the state file. The state file keeps track of the resources managed by Terraform and their current state.

terraform plan

The `terraform plan` command creates an execution plan for applying changes to your infrastructure. It analyses your Terraform configuration, compares it with the current state, and generates a detailed plan of what actions Terraform will take.

Running `terraform plan` does not make any actual changes to your infrastructure. Instead, it provides an overview of the changes that Terraform will apply when you run `terraform apply`. The plan includes additions, modifications, and deletions of resources, along with any required dependencies or actions.

The plan output displays information such as the resource creation or destruction, resource attributes that will be modified, and any variables or data sources used in the configuration. This allows you to review and validate the planned changes before applying them.

terraform apply

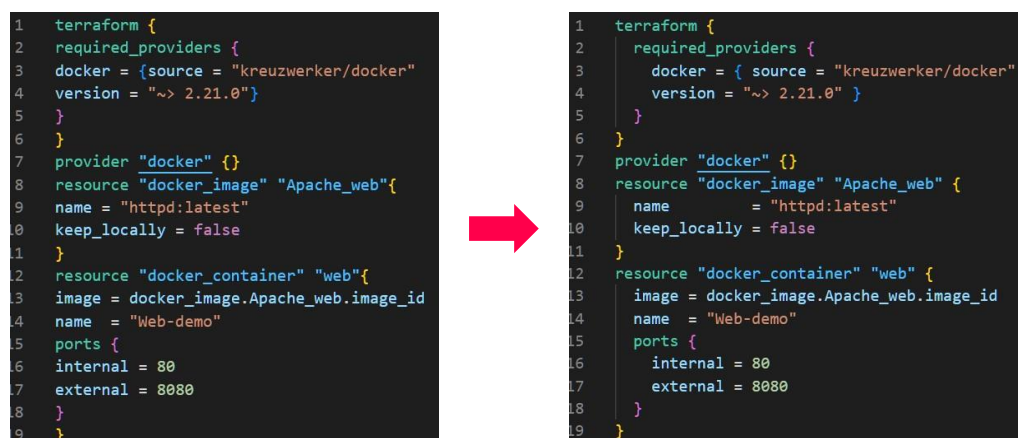
The `terraform apply` command is used to apply the planned changes to your infrastructure. It executes the changes defined in the Terraform configuration, creating, modifying, or deleting resources, as necessary.

When you run `terraform apply`, Terraform prompts you to confirm the changes based on the execution plan generated by `terraform plan`. Once you confirm the changes, Terraform starts applying them in a controlled and sequential manner.

`terraform apply` also updates the state file to reflect the new state of your infrastructure after the changes have been applied. This updated state file becomes the basis for future Terraform operations, enabling you to manage and track the state of your infrastructure over time.

These commands will be discussed more later.

Terraform fmt



The diagram illustrates the output of the `terraform fmt` command. It shows two versions of a Terraform configuration file side-by-side, separated by a red arrow pointing from left to right. The left version shows the code before formatting, with inconsistent indentation (using tabs and spaces) and no trailing commas. The right version shows the code after formatting, where all indentation is standardized to two spaces, trailing commas are added to the end of each line in a block, and the overall structure is more consistent and readable.

```
1 terraform {
2   required_providers {
3     docker = { source = "kreuzwerker/docker"
4       version = "~> 2.21.0" }
5   }
6 }
7 provider "docker" {}
8 resource "docker_image" "Apache_web" {
9   name = "httpd:latest"
10  keep_locally = false
11 }
12 resource "docker_container" "web" {
13   image = docker_image.Apache_web.image_id
14   name = "Web-demo"
15   ports {
16     internal = 80
17     external = 8080
18   }
19 }
```


The ``terraform fmt`` command in Terraform is 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. Here's some more information about ``terraform fmt``:

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.

Module 2 - Terraform Foundations

Introduction

Agenda

- Installing Terraform
- Authentication
- Terraform files with introduction to Modules
- Terraform init, plan and apply
- Parallelism
- Implicit vs explicit dependencies
- Lab 2 Create an EC2 instance



Installing Terraform

Installing Terraform

Operating Systems:

- HomeBrew on OS X
 - `brew tap hashicorp/tap`
 - `brew install hashicorp/tap/terraform`
- Chocolatey on Windows
 - `choco install terraform`
- Linux
 - `sudo apt-get update && sudo apt-get install -y gnupg software-properties-common curl`
 - `curl -fsSL https://apt.releases.hashicorp.com/gpg | sudo apt-key add`
 - `sudo apt-add-repository "deb [arch=amd64] https://apt.releases.hashicorp.com $(lsb_release -cs) main"`
 - `sudo apt-get update && sudo apt-get install terraform`



Terraform can be installed on various operating systems, including Windows, macOS, and Linux. Here are the general steps to install Terraform:

Download the Terraform binary: Visit the official Terraform website (<https://www.terraform.io/downloads.html>) and download the appropriate Terraform binary for your operating system.

Extract the downloaded archive: After downloading the Terraform binary, extract the contents of the archive to a directory of your choice.

Set up the PATH environment variable (optional): To run Terraform from any directory in your command-line interface (CLI), you can add the directory containing the Terraform binary to your system's PATH environment variable. This step is optional but recommended for convenience.

Verify the installation: Open a new terminal window or command prompt and run the following command to verify that Terraform is installed and accessible:

terraform version

If Terraform is installed correctly, you will see the version information displayed in the output.

Note: The exact installation steps may vary slightly depending on your operating system and preferred installation method. For more detailed installation instructions and OS-specific guidance, you can refer to the official Terraform documentation (<https://learn.hashicorp.com/tutorials/terraform/install-cli>).

It is worth mentioning that there are also package managers, such as Homebrew (macOS) and Chocolatey (Windows), that offer simplified installation and updating of Terraform. You can check their respective documentation for installation instructions using package managers.

Authentication

Authentication

- Authentication to the platform
- Cloud Platforms behave differently for Authentication
- Cloud Platforms require different levels of Authorization



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 Authentication

Environment Variables:

Set `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` in your environment.

Shared Credentials File

Use the AWS credentials file located at `~/.aws/credentials`.

AWS CLI Authentication

If authenticated via the AWS CLI, Terraform can use these credentials.

IAM Roles

When running on AWS services like EC2, assign IAM roles to instances, allowing Terraform to assume these roles without explicit credentials.



Azure Authentication

Azure Authentication

Azure CLI

Authenticate using `az login`. Terraform will use the active Azure CLI session.

Service Principal

Create a service principal and provide its credentials via environment variables or directly in the provider configuration.

Managed Identity

When running on Azure services, use managed identities for authentication.



Google Cloud Authentication

Google Cloud Authentication

Service Account Key File

Create a service account, download its JSON key file, and set the `GOOGLE_APPLICATION_CREDENTIALS` environment variable to point to this file

Application Default Credentials

Use `gcloud auth application-default login` to set up user credentials.

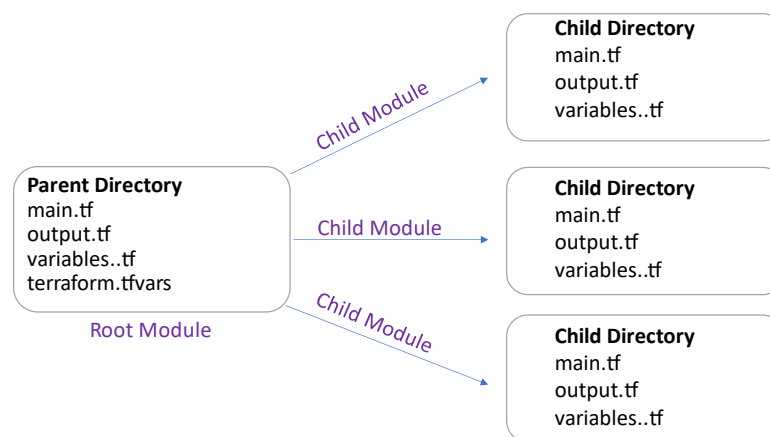
Service Account Impersonation

Configure Terraform to impersonate a service account, enhancing security by avoiding long-lived credentials



Terraform files

Terraform files and modules



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 init

Terraform init, plan and apply

terraform init

- First command run after writing new configuration or after cloning an existing configuration from version control
- Can safely be run multiple times
- Used to initialise a working directory & initialise chosen backend
- Parses the configuration, identifies the providers and installs the plug-ins
- Module blocks are identified from 'source' arguments.

```
$ terraform init
Initializing the backend...

Initializing provider plugins...
- Finding latest version of hashicorp/aws...
- Installing hashicorp/aws v4.15.1...
- Installed hashicorp/aws v4.15.1 (signed by HashiCorp)

Terraform has created a lock file .terraform.lock.hcl to record
the provider
selections it made above. Include this file in your version
control repository
so that Terraform can guarantee to make the same selections
by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```



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

Terraform init, plan and apply

terraform plan

- Inspects the current configuration proposed
- Inspects the current state of existing objects
- Proposes a set of changes to be made

```
listener.3238714211.lb_port:      "" => "443"
listener.3238714211.lb_protocol:  "" => "https"
listener.3238714211.ssl_certificate_id: "" => "arn:aws:acm:us-east-1:123456789012:certificate/12345678-9012-3456-7890-123456789012"

~ aws_iam_policy.user-my-test
  policy:  {}
          {
            "Effect": "Allow",
            "Resource": [
              ~ "arn:aws:s3::my-test-development",
              ~ "arn:aws:s3::my-test-development/*"
              + "arn:aws:s3::my-test-development"
            ],
            "Action": [
              "s3:*)"
            ]
          }
    }
```



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

Terraform init, plan and apply

terraform apply

- Executes the actions described within the plan.
 - [Automatic plan mode] *apply*, without further arguments will run *plan* first
~ you will be prompted for approval ~
 - [Saved plan mode] *apply*, with a previously saved plan – *terraform apply [plan filename]*
~ you will not be prompted for approval ~
- You can inspect saved plan before applying using 'terraform show'

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>
```

Let us break down these optional flags:

`-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.

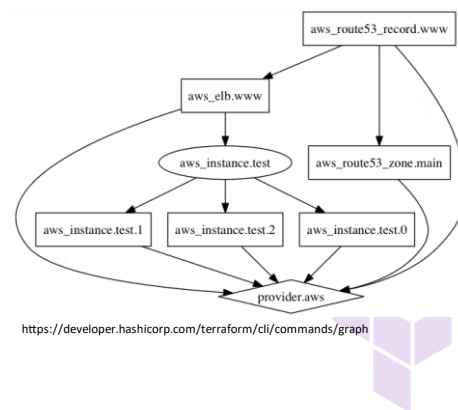
`<path-to-plan-file>`: This is the path to the saved plan file generated by 'terraform plan'. Specify the path to the plan file you want to apply.

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

Parallelism

- Terraform builds a dependency graph from the terraform configurations, and walks this graph to generate plans, refresh state, and more
- Greater parallelism potentially increases the speed of deployment but may overwhelm the resources of the machine running Terraform
- Can be used as an option `-parallelism` with `init`, `plan` and `apply` (Default 10)
- Parallelism is **ON** by default – use switch to vary thread count



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.

Implicit vs explicit dependencies

Implicit vs explicit dependencies

Implicit dependencies are known and calculated by Terraform
Explicit dependencies are defined by the configuration author

- **Implicit dependencies**
An elastic IP references an AWS instance. Terraform therefore knows that the IP is dependent on the instance existing so it will create the instance before the IP.
- **Explicit dependencies**
Two instances could be created at the same time (parallelism) but the DB instance may need to be created first, and its configuration exported before creating a web instance

```
resource "aws_instance" "my_ec2" {  
  ami = data.aws_ami.amazon_linux.id  
  instance_type = "t2.micro"  
}  
  
resource "aws_eip" "staticip" {  
  vpc = true  
  instance = aws_instance.my_ec2.id  
}
```

```
.....  
resource "aws_instance" "my_web" {  
  ami = data.aws_ami.amazon_linux.id  
  instance_type = "t2.micro"  
  
  depends_on = [aws_instance.my_rds]  
}
```



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 dependencies.

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.

For example, if you define a network interface resource that references a virtual network resource, Terraform will implicitly understand that the network interface depends on the existence of the virtual network. Terraform will ensure that the virtual network is created or modified before attempting to create or modify the network interface.

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.

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.

Module 3. Terraform Providers

Introduction to Providers

Agenda

- Introduction to Providers
- Local Provider
- Public Cloud Providers
- AWS resource examples
- Azure resource examples
- Google Cloud resource examples
- Lab 3 Create a basic network



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.

Introduction to Providers

- A provider in Terraform is a plugin that enables interaction with an API. The providers are specified in the Terraform configuration code. They tell Terraform which services it needs to interact with.
- Terraform currently has 1700+ providers
- Each provider adds a set of resource types and/or data sources that Terraform can manage.



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

Introduction to Providers

- A provider in Terraform is a plugin that enables interaction with an API. The providers are specified in the Terraform configuration code. They tell Terraform which services it needs to interact with.
- Terraform currently has 170+ providers
- Each provider adds a set of resource types and/or data sources that Terraform can manage.



The Local provider allows you to define and manage local resources and configurations within your Terraform code. It is a special provider that allows you to interact with local data and perform actions on the machine where Terraform is running.

The Local provider is useful for scenarios where you need to generate files, execute scripts, run commands, or perform other operations on the machine running Terraform. It provides a way to incorporate local operations into your Terraform configuration.

Here is an example of using the Local provider to generate a local file:

```
provider "local" {}

resource "local_file" "example_file" {
  filename = "/path/to/example.txt"
  content  = "This is an example file."
}
```

In this example, we have defined a Local provider without any specific configuration. Then, we have a `local_file` resource that creates a file at the specified path with the given content. This file will be generated on the machine where Terraform is executed.

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

Public Cloud Providers

- Terraform can provision infrastructure across public cloud providers such as Amazon Web Services (AWS), Azure and Google Cloud, as well as private cloud and virtualization platforms such as OpenStack and VMWare.
- Providers differ! Always refer to documentation regarding terminology and resource types and features

What cloud providers does Terraform support?

Providers

- AWS.
- Azure.
- Google Cloud Platform.
- Kubernetes.
- Alibaba Cloud.
- Oracle Cloud Infrastructure.



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.

Here is an example of a provider block for AWS in a Terraform configuration:

```
provider "aws" {  
  region = "us-west-2"  
  access_key = "YOUR_ACCESS_KEY"  
  secret_key = "YOUR_SECRET_KEY"  
}
```

In this example, the provider block specifies the AWS provider and includes the region and authentication details.

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.

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

Aliases

- You can optionally define multiple configurations for the same provider and select which one to use on a per-resource or per-module basis. The primary reason for this is to support multiple regions for a cloud platform; other examples include targeting multiple Docker hosts, multiple Consul hosts, etc.

```
terraform {
  required_providers {
    azure = {
      source = "hashicorp/azure"
      version = "~>3.0.0"
    }
  }
}

provider "azure" {
  alias = "dev"
  subscription_id = var.de
  features {}
}

provider "azure" {
  alias = "prod"
  subscription_id = var.pr
  features {}
}

# Default provider configuration for the primary GCP
provider "google" {
  project = "my-primary-project-id"
  region = "us-central1"
}

# Additional provider configuration for another region or project
provider "google" {
  alias = "secondary"
  project = "my-secondary-project-id"
  region = "europe-west1"
}
```



Aliases can be used within the provider block to specify alternative names for the provider configuration. This can be useful when you need to define multiple instances of the same provider with different configurations, such as when working with multiple regions or environments.

Here is an example of using aliases within the provider block:

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

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

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` argument within the provider block allows you to assign alternative names to each provider configuration.

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.

Module 4. Variables and Functions

Agenda

- Introduction to Variables
- Input Variables
- Output Values
- Functions
- References to Named Values
- Lab 4 Network deployment with variables and iterations



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.

Introduction to Variables

- Input variable act as parameters for Modules– “input parameters”

Input variables are like function arguments

- Output Values return values from a module– “outputs”

Output values are like function return values



Input Variables

Input Variables

Input variables for a module are declared in a **variable block**

label unique name for variable within the module block

type defines the value types accepted

default apply this value if no other value is passed

```
variable "az_names" {  
  type = list(string)  
  default = ["eu-west-1a"]  
}
```



In Terraform, there are two types of variables: input variables and output variables. They serve different purposes within your Terraform configuration.

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 are defined in a variables file, typically named `variables.tf` or `terraform.tfvars`, and they are used to accept input from the user or from external systems.

Input variables can have types, default values, descriptions, and other attributes. They are typically defined at the top of your configuration to provide a clear overview of the variables required by your infrastructure.

In the example shown above, `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 a value is not provided when running Terraform, the default value will be used.

Input variables are typically assigned values using command-line flags, variables files, or environment variables. They allow users to customize the configuration based on their specific requirements.

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 value precedence order

```
instance_type = var.ec2_instance_size
```

- -var value entered at runtime
- <name>.auto.tfvar value
- <name>.tfvars value
- Terraform.tfvars value
- Default variable value
- Environment variable value
- Prompt for value

```
terraform plan -var="instance_size=t3-small"
```

auto-loaded in lexical order, last loaded has highest precedence

```
terraform plan --var-file=file_1.tfvars --var-file=file_2.tfvars (llhhp)
```

auto-loaded if exists

automatically used if exists

automatically used if exists



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.`

Variable Files

Values specified in a variables file, such as `terraform.tfvars` or terraform.auto.tfvars`, are the next level of precedence. Terraform automatically loads some of these files if present in the working directory, others must be specified.`

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.`

```
variable "aws_region" {  
    description = "The AWS region for resources "
```

```

type = string
default = "us-west-2"
}

```

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

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.

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

- An output block is used to declare a value to be exported
- Can be used to output values to the command line
Root module may return information back to the CLI
- Can be used to output values to other terraform configurations.
A child module can pass some of its attributes back to the parent module where they can be referenced.
Syntax: `module.<module name>.<outputname>`
Example: `module.instance_create.priv_ip_add`
- Outputs are typically defined in `outputs.tf` file stored in module directory
- Query outputs with `terraform output` command

```

output "priv_ip_add" {
  value = az_instance.vm.private_ip
}

```

```

$ terraform output
priv_ip_add = "10.1.0.123"

```

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.

Example of defining an output variable:

```
output "instance_ip" {  
  description = "The public IP address of the instance"  
  value = aws_instance.example.public_ip  
}
```

In this example, `instance_ip` is an output variable that exposes the public IP address of an AWS EC2 instance. The value is derived from the `public_ip` attribute of the `aws_instance.example` 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.

Functions

Functions

- Terraform provide a number of functions that can be called within your expression

- Function syntax is `<name_of_function> (arguments)`

- Some of the function types are:

Numeric Functions	String Functions
Collection Functions	Encoding Functions
Filesystem Functions	Date and Time Functions
Hash and Crypto Functions	IP Network Functions
Type Conversion Functions	

- The Terraform language does not support user -defined functions

```
format("Your Region is, %s!", var.name)  
Your Region is, eu-west-2!
```

```
Timestamp()  
2022-05-13T07:08:45Z
```



In Terraform, functions are built-in operations that allow you to manipulate and transform values within your configuration. Functions can be used to perform various tasks, such as

manipulating strings, working with lists and maps, performing mathematical calculations, generating random values, and more.

Functions in Terraform follow a declarative approach, where you define the desired outcome and let Terraform evaluate and execute the function during the configuration's runtime.

Here are some key points to understand about functions in Terraform:

Function Syntax

Functions are invoked using a function call syntax, similar to many programming languages. The function name is followed by arguments within parentheses. The return value of a function can be assigned to a variable or used directly within the configuration.

Function Arguments

Functions accept one or more arguments, which can be constant values, variables, or other function calls. Arguments provide input values that the function operates on. The arguments can be of several types depending on the function's purpose.

Function Return Values

Functions in Terraform return values that can be used within the configuration. The return value can be assigned to a variable, passed as an argument to another function, or used directly in resource or variable definitions.

Common Functions

Terraform provides a wide range of built-in functions. Some common functions include ``concat`` (for concatenating strings), ``join`` (for joining list elements into a string), ``element`` (for accessing list elements), ``map`` (for creating a map), ``contains`` (for checking if a value is present in a list or map), ``regex`` (for matching regular expressions), ``format`` (for formatting strings), and many more.

Conditional Functions

Terraform also offers conditional functions like ``if`` and ``coalesce``, allowing you to conditionally evaluate expressions or provide fallback values.

To use a function, you include the function call within your Terraform configuration, providing the necessary arguments. Here is an example of using the ``concat`` function to concatenate strings:

```
variable "name" {  
  type = string  
  default = "John"  
}
```

```
variable "last_name" {
```

```
type = string
default = "Doe"
}

output "full_name" {
value = concat(var.name, " ", var.last_name)
}
```

In this example, the `concat` function is used within the `output` block to concatenate the `name` and `last_name` variables with a space in between. The resulting full name is then displayed as the output.

By using functions in Terraform, you can manipulate values, generate dynamic configurations, handle conditional logic, and enhance the flexibility and expressiveness of your infrastructure-as-code configurations. The Terraform documentation provides an extensive list of built-in functions and their usage.

References to Named Values

Reference to Named Values

- A name is an expression that references a value.
Can be a single expression or combined with other expressions
- They can be used as:
 - Resources
 - Input Variables
 - Local Values
 - Child Module outputs
 - Data Sources
 - Filesystem & Workspace info



In Terraform, a reference to a named value is a way to access or use the value of a named object or resource within your Terraform configuration. It allows you to retrieve specific attributes or properties of a resource and use them in other parts of your configuration.

When you declare a resource in Terraform, it has various attributes that represent specific characteristics or properties of that resource. For example, an AWS EC2 instance resource may have attributes such as its ID, IP address, availability zone, etc.

To reference a named value, you typically use the interpolation syntax `\${}` or ``data.<TYPE>.<NAME>.<ATTRIBUTE>``. The specific syntax depends on the context in which the reference is used.

Here are an example to illustrate referencing named values in Terraform:

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

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

In this example, the ``aws_instance.example.id`` reference is used to access the ``id`` attribute of the ``aws_instance.example`` resource. The value of the ``id`` attribute is then outputted as ``instance_id``.

Data Attribute Reference:

```
data "aws_ami" "example" {
  most_recent = true
  owners = ["self"]
  filters = {
    name = "ubuntu/images/*ubuntu-xenial-16.04-amd64-server-*"
  }
}

resource "aws_instance" "example" {
  instance_type = "t2.micro"
  ami = data.aws_ami.example.id
}
```

In this example, the ``data.aws_ami.example.id`` reference is used to access the ``id`` attribute of the ``aws_ami`` data source. The value of the ``id`` attribute is then used as the value for the ``ami`` parameter of the ``aws_instance`` resource.

Named Value Examples

Named Value Examples

1. **Conditional expression.** If we are using the 'test' workspace then deploy 2 VMs, otherwise deploy 10
2. **Index value.** Create instances named Server0 to Server1 or Server9 dependent upon workspace name.
3. **for_each string** . Extract each string from the set and use as the IAM user name.

```
resource "azurerm_linux_virtual_machine" "small_vm" {  
  count = terraform.workspace == "test" ? 2 : 10  
}
```

```
resource "azurerm_network_interface" "example" {  
  count = terraform.workspace == "test" ? 2 : 10  
  name = "small-vm-nic-${count.index}"  
}
```

```
resource "azurerm_role_assignment" "newusers" {  
  for_each = toset(["Bob", "Jo", "Charlie", "Scooby"])  
}
```



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, interpolation refers to the process of dynamically substituting or inserting values into strings or expressions within your Terraform configuration. It allows you to combine static text with dynamically generated values, such as variables, resource attributes, or function calls, to create dynamic and flexible configurations.

Variable Interpolation

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

```
variable "name" {  
  type = string  
  default = "John"  
}  
  
resource "aws_instance" "example" {  
  instance_type = "t2.micro"  
  tags = {  
    Name = "Instance for ${var.name}"  
  }  
}
```

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 tag value.

Expression value Data Types

- **String** a set of Unicode characters within doublequotes
- **Number** may be integer or include decimal point
- **Bool** true or false
- **List** a sequence of values where each element can be identified by its index number starting with 0 [valA, valB, valC] (index 1 would return valB)
- **Map** groups of values identified by its label { Region = "UK," City = "Newcastle" }
- **Null**



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.

Module 5. Templates, State and Drift

Introduction

Agenda

- Importing, Recreating and Terminating Resources
- Templates
- OSS Workspaces
- Local vs. Remote State files
- Managing State and Configuration Drift
- Logging
- Lab 5 Migrate state and create an Application Gateway

Importing, Recreating and Terminating Resources

Importing, Recreating and Terminating Resources

- Resources may already have been created and configured outside of terraform
- Identify Resources which remain independent of Terraform and those which need to be brought under Terraform control
- Consider importing resources into terraform



Terraform Import

- Terraform can import existing infrastructure resources. This functionality lets you bring existing resources under Terraform management.
- Terraform import can only import resources into the state. Importing does not generate configuration.
- Before you run terraform import you must manually write a resource configuration block for the resource. The resource block describes where Terraform should map the imported object.
- Must run locally, not supported in Terraform Cloud
- <https://developer.hashicorp.com/terraform/cli/import>



Terraform import is a command that allows you to import existing infrastructure resources into your Terraform state file. It is useful when you have infrastructure resources that were created outside of Terraform, and you want to manage them using Terraform going forward.

When you import a resource, Terraform reads the current state of that resource and creates a corresponding resource configuration in your Terraform configuration files. This enables Terraform to track and manage the imported resource alongside other resources defined in your configuration.

Here are some key points to understand about Terraform import:

1. Syntax: The syntax for the import command is as follows:

...

```
terraform import [options] ADDRESS ID
```

...

- ``ADDRESS``: Specifies the Terraform resource address for the resource being imported. It is typically in the format ``TYPE.NAME``, where ``TYPE`` is the resource type and ``NAME`` is the name or identifier of the resource.
- ``ID``: Represents the unique identifier or name of the existing resource to be imported.

Resource Discovery: Before performing the import, it is important to identify the appropriate resource address and the corresponding unique identifier for the resource. Terraform uses this information to map the imported resource to the corresponding configuration block.

State Management: Terraform import updates the Terraform state file to include the imported resource. The state file keeps track of the desired state and the current state of your infrastructure.

Import Limitations: Not all resources can be imported into Terraform. The ability to import resources depends on the provider and the resource type. Some resources may have limitations or restrictions on import due to their complexity or dependencies.

Manual Configuration: After importing a resource, you may need to manually update your Terraform configuration files to reflect the existing state of the imported resource. This involves specifying the necessary attributes and configuration options based on the imported resource.

It is important to note that importing resources does not automatically generate a complete and accurate Terraform configuration. You will need to carefully review and update the imported resource's configuration to ensure it aligns with the existing state of the resource.

Importing resources should be done with caution, as incorrect import operations can lead to configuration drift and inconsistencies between Terraform and the actual infrastructure. It is recommended to test and validate the import process on non-production environments before performing it in production.

The Terraform documentation provides more detailed information on the import command, including examples and specific considerations for each provider:

<https://www.terraform.io/docs/cli/import/index.html>

There are additional tools and resources available to assist with importing existing infrastructure into Terraform. These tools can provide automation, enhance the import process, or offer guidance for complex scenarios. Here are a few notable ones:

Terraformer: Terraformer is an open-source tool developed by Google that helps import existing infrastructure resources from various cloud providers into Terraform. It supports popular providers like AWS, Azure, GCP, and more. Terraformer generates Terraform

configuration files based on the existing resources, making it easier to integrate them into your Terraform workflow.

Terraform Importer: Terraform Importer is an open-source tool developed by Gruntwork (now part of HashiCorp). It helps automate the process of importing resources by providing a CLI tool that generates the Terraform import command for a given resource. It aims to simplify the import process and increase efficiency.

Provider Documentation: Many cloud providers offer documentation and guides specific to importing resources into Terraform. These resources often provide step-by-step instructions, examples, and considerations for importing diverse types of resources. Check the official documentation of your cloud provider for specific guidance.

Community Tools and Scripts: The Terraform community has developed various scripts, tools, and modules to assist with importing resources into Terraform. These community contributions can be found on platforms like GitHub, Terraform Registry, or other community forums. They can provide specific import scripts for complex resources or offer reusable modules to aid in the import process.

When using any third-party tools or community resources, it is important to review and validate them against your specific use case and requirements. Consider factors like compatibility with your Terraform version, support for your cloud provider, and community maintenance and support.

It is worth noting that while these tools and resources can assist in the import process, it is still necessary to carefully review and update the imported resource's configuration to ensure it accurately reflects the existing state of the resource.

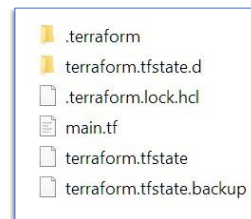
Always exercise caution when performing import operations, especially in production environments. Take backups and test on non-production environments to minimize the risk of configuration drift and ensure a smooth transition to managing your infrastructure with Terraform.

OSS Workspaces

Terraform CLI Workspaces

- Terraform CLI workspaces differ from Terraform Cloud Workspaces
- Terraform CLI workspaces – one by default called 'default'
- The 'default' cannot be deleted
- Each workspace holds our 'state'
- To create a new workspace called Test:
`terraform workspace new test`
- To switch to a workspace called Test: `terraform workspace select test`

```
resource "azurerm_linux_virtual_machine" "small_vm" {  
  count  
    = terraform.workspace == "test" ? 2 : 10  
}
```



HashiCorp
Terraform

Terraform Open Source (OSS) 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 Terraform OSS to interact with workspaces.

Workspaces in Terraform OSS 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.

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.

Local vs. Remote State files

Local vs Remote State files

- State may be recorded locally or remote

Local by default

May be migrated

May be 'moved' without data migration

```
terraform {  
  backend "local" {  
    path = "relative/path/to/terraform.tfstate"  
  }  
}
```

- Locations

Local: may use default location or you may specify path

Remote: many options available



Local State Files

Storage Location: Local state files are stored on the local machine where Terraform is executed. By default, Terraform creates a ``terraform.tfstate`` file in the working directory. However, the filename and location can be customized.

Accessibility: Local state files are accessible only on the machine where Terraform is executed. They are not shared or synchronized automatically with other team members or remote systems.

Collaboration: Local state files are typically used in single-person or single-environment scenarios where collaboration is not a concern. Each user or environment manages its own separate local state file.

Version Control: Local state files are not intended to be directly committed to version control systems. They contain sensitive information and can be quite large, making it impractical to store them in version control repositories.

Remote State Files

Storage Location: Remote state files are stored in a remote backend, such as Terraform Cloud, AWS S3, Azure Blob Storage, or HashiCorp Consul. The state file is uploaded to and stored in the chosen remote backend.

Accessibility: Remote state files can be accessed and shared by multiple team members or systems. They provide a centralized source of truth for the infrastructure state, allowing collaboration and coordination across the team.

Collaboration: Remote state files enable collaboration between team members, allowing them to work on the same infrastructure configuration. Changes made by one team member can be reflected in the remote state file, making it visible to others.

State Locking: Remote state files often support state locking mechanisms to prevent concurrent modifications. This ensures that only one user or process can make changes to the state at a time, preventing conflicts.

Version Control: Remote state files can be versioned and stored in version control systems alongside the Terraform configuration. This provides a history of changes and allows for easier management and auditing of infrastructure changes over time.

Using remote state files is generally recommended for production environments and collaborative team setups. They offer benefits such as centralized state management, collaboration, state locking, and better integration with version control systems.

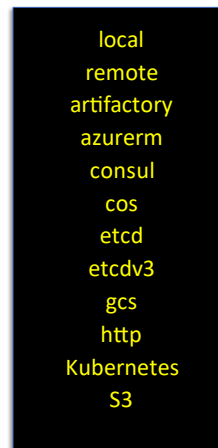
However, local state files can be sufficient for simple or personal projects, where collaboration and infrastructure complexity are minimal. They are easier to set up and work within such scenarios.

It is important to carefully consider the requirements, team structure, and complexity of your infrastructure project to decide whether to use local or remote state files.

Backends

- Not all backends provide the same functionality, some research may be required
- Even a particular backend may provide different features depending on its setup. S3 may simply store the state or use DynamoDB to perform state locking and consistency checking.

<https://developer.hashicorp.com/terraform/language/settings/backends/configuration>



HashiCorp
Terraform

Remote Backends

[AWS](#)

[Google](#)

[Azure](#)

```
terraform {  
  backend "s3" {  
    bucket = "mybucket"  
    key = "path/to/my/key"  
    region = "us-east-1"  
  }  
}
```

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

```
terraform {  
  backend "azurearm" {  
    resource_group_name = "StorageAccount -ResourceGroup"  
    storage_account_name = "abcd1234"  
    container_name = "tfstate"  
    key = "prod.terraform.tfstate"  
  }  
}
```



HashiCorp
Terraform

Terraform provides a variety of backends that allow you to store and manage your Terraform state remotely. Some commonly used backends include:

Local Backend: The local backend is the default backend for Terraform. It stores the state file locally on the machine where Terraform is executed. This backend is suitable for individual or small-scale projects where remote state storage is not required.

Amazon S3 Backend: The Amazon S3 backend allows you to store the state file in an Amazon S3 bucket. It provides durable and scalable storage for your state files. The S3 backend is commonly used for projects deployed on AWS.

State Locking example – AWS S3

When using the S3 backend in Terraform, you have the option to enable state locking by configuring an associated DynamoDB table. The DynamoDB table acts as a distributed lock to coordinate access to the state file stored in S3. Here is how it works:

S3 Backend Configuration: In your Terraform backend configuration, you specify the S3 bucket where the state file is stored. Additionally, you configure the ``dynamodb_table`` attribute with the name of the DynamoDB table to be used for state locking.

State Locking: When Terraform initializes or performs state-modifying operations, such as ``terraform apply``, it acquires a lock on the specified DynamoDB table before accessing the state file in S3. This lock ensures that only one user or process can modify the state at any given time.

Concurrent Access: If another user or process attempts to modify the state while a lock is in place, Terraform will wait until the lock is released. It retries acquiring the lock until it succeeds or times out. This ensures that conflicting modifications are prevented and that the state remains consistent.

Release of Lock: After completing the state-modifying operation, Terraform releases the lock, allowing other users or processes to acquire it for their own state modifications.

By leveraging Amazon S3 for storing the state file and DynamoDB for distributed locking, Terraform ensures that the state remains consistent and avoids conflicts when multiple users or processes are working with the same infrastructure. The combination of S3 and DynamoDB provides a scalable and reliable solution for managing state locking in Terraform.

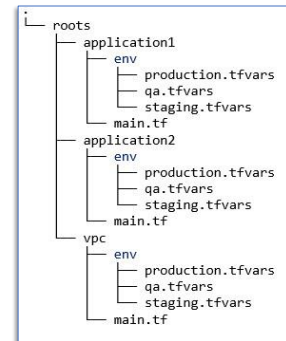
It is important to properly configure and secure both the S3 bucket and DynamoDB table to ensure the integrity and confidentiality of your Terraform state.

These are just a few examples of common backends available in Terraform. Each backend has its own advantages and considerations, and the choice of backend depends on your specific requirements, infrastructure setup, and cloud provider preferences.

Templates

Templates

- Questions to consider
 - How large is your main.tf?
 - How complex is your architecture that is being deployed and maintained by Terraform?
 - How complex is your team?
 - Do you require separation of concerns?
 - Can you re-use your code segments in other projects?
- Pros/Cons of one large configuration (single Root Module) vs. Parent / Child modules to define your configuration ?



In Terraform, modules are self-contained packages of Terraform configurations that can be used to organize and encapsulate reusable infrastructure code. They allow you to break down your configuration into smaller, modular components, making it easier to manage, reuse, and collaborate on infrastructure definitions. When working with modules, there is a concept of root modules and child modules.

Root & Child Modules

Child Modules use the same configuration language as the Root

Input variables
output values
Resources

Location

local – create a new directory and create one or more .tf files
remote - (public vs private registry)

Referencing the child modules

local – by relative path
remote – by path to registry / module

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



Root Module

A root module is the top-level module in your Terraform configuration. It represents the main entry point of your infrastructure configuration and typically defines the resources and

modules that compose your infrastructure. The root module is usually located in the root directory of your Terraform project.

Root modules often include input variables, output values, provider configurations, and other top-level configuration elements. They serve as the crucial point for coordinating and managing the overall infrastructure.

Child Modules:

Child modules are reusable components of your infrastructure configuration that are defined and encapsulated within separate directories. They are designed to represent specific logical units or components of your infrastructure.

Child modules can be used multiple times within a root module or across different root modules. They provide a way to abstract and reuse common infrastructure patterns, making your configurations more modular, maintainable, and scalable.

Child modules can have their own input variables, output values, and resource configurations. They can accept input values from the calling module using variable bindings and provide output values that can be used by the calling module or other modules.

By using child modules, you can build a library of reusable infrastructure components that can be shared across different projects or teams. This promotes code reuse, consistency, and collaboration.

Here is an example file structure illustrating the relationship between root and child modules:

```
├── main.tf          (Root module)
├── variables.tf     (Root module variables)
├── outputs.tf       (Root module outputs)
├── ── vpc
│   ├── main.tf     (Child module for VPC)
│   ├── variables.tf (Child module variables)
│   └── outputs.tf   (Child module outputs)
├── ── app
│   ├── main.tf     (Child module for application resources)
│   ├── variables.tf (Child module variables)
│   └── outputs.tf   (Child module outputs)
└── ...
```

In this example, the root module is located in the root directory and defines the main infrastructure resources. The `modules` directory contains separate directories for the VPC

module and the application module, each with their own set of files defining the respective resources, variables, and outputs.

To use a child module within a root module, you would declare a module block in the root module's configuration, providing the module's source and any input variable values needed.

Child modules offer a way to organize, reuse, and modularize your infrastructure configurations, making them more maintainable and scalable. They provide a higher level of abstraction and encapsulation, enabling you to create reusable building blocks for your infrastructure deployments.

Terraform modules are a way to encapsulate, and reuse Terraform configurations and resources. They allow you to define and package reusable components that can be shared across different infrastructure deployments. Modules promote code organization, modularity, and reusability, making it easier to manage and maintain infrastructure configurations.

Modular Components: A module in Terraform represents a collection of related resources and configurations that can be managed as a single unit. It acts as a building block for creating infrastructure components, such as a web server, database, or network subnet. Modules help organize and encapsulate resources, making it easier to manage complex infrastructure deployments.

Encapsulation and Abstraction: Modules provide encapsulation by abstracting the implementation details of a set of resources. They allow you to define inputs, outputs, and configuration parameters, hiding the internal details of how the resources are created or managed. This abstraction promotes simplicity and reusability.

Input Variables: Modules can accept input variables, which allow you to pass values or configurations from the calling module or root configuration. Input variables provide flexibility, enabling you to customize module behaviour for different deployments. They can be used to parameterize module configurations and allow users to provide their desired values.

Output Values: Modules can define output values, which represent information or data that can be used by other parts of the configuration. Output values can be used to expose resource attributes, connection details, or calculated results. They provide a way to communicate information from the module back to the calling configuration.

Module Composition and Nesting: Modules can be composed and nested, allowing you to create higher-level abstractions and composite infrastructure components. A module can call other modules, incorporating their resources and configurations. This composition capability allows you to build complex infrastructures by combining reusable modules.

Registry and Community Modules: Terraform offers a module registry where you can publish and share modules with the Terraform community. The registry provides a centralized repository for discovering and using pre-built modules created by others. It helps promote collaboration, sharing of best practices, and leveraging community expertise.

Private Modules: In addition to the public module registry, Terraform allows you to create and use private modules within your organization. Private modules offer a way to encapsulate and share infrastructure configurations within your team or company. They provide a controlled and reusable approach for managing internal infrastructure components.

By using Terraform modules, you can achieve several benefits:

Reusability: Modules enable you to create reusable components, reducing duplication and promoting consistency across infrastructure deployments.

Abstraction: Modules abstract the complexity of resource creation and management, making it easier to work with and understand infrastructure configurations.

Collaboration: Modules facilitate collaboration by allowing teams to share and reuse infrastructure components, promoting consistency and best practices.

Maintainability: Modules simplify the management of infrastructure configurations by encapsulating resources and providing clear interfaces for configuration inputs and outputs.

Terraform modules are a powerful feature that enhances the scalability, maintainability, and reusability of your infrastructure code. They allow you to create modular, composable, and shareable components, enabling efficient infrastructure management and promoting consistent and reliable deployments.

Managing State and Configuration Drift

Managing State and Configuration Drift

- Resources should be managed by our IaC tooling
- Human intervention may accidentally introduce changes to configuration
- Deliberate intervention may be required for Priority fixes
- Configuration as Code tools may intentionally introduce changes due to bad planning
- Configuration Drift needs to be detected and fixed



<https://pixabay.com/photos/drift-sport-drift-car-speed-2396992/>

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.

Some important points to understand about state drift:

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.

Taint/Replace an existing resource

- Taint/Replace is used to inform Terraform that a resource has become damaged or degraded and should be replaced at next `apply`

Taint < v0.15.2

```
terraform taint [options] aws_instance.vm1
```

Replace => v0.15.2

```
terraform apply -replace="aws_instance.vm1"
```



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.

Please 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.

Logging

Logging

- Enable logging using `TF_LOG` environment variable. This will direct any logging output to `stderr`
- You can set `TF_LOG` to `TRACE`, `DEBUG`, `INFO`, `WARN` or `ERROR` to change the verbosity of the logs. If you set `TF_LOG` to `JSON`, the output logs at the `TRACE` level in a `json` format.
- If you wish to persist the output to your own log, then set `TF_LOG_PATH` in order to append to a specific file when logging is enabled with `TF_LOG`
- Logging can be enabled separately for terraform itself and the provider plugins using the `TF_LOG_CORE` or `TF_LOG_PROVIDER` environment variables. These take the same level arguments as `TF_LOG`, but only activate a subset of the logs.

<https://developer.hashicorp.com/terraform/internals/debugging>



Some common logging options available in Terraform:

Standard Output (STDOUT)

Terraform outputs log messages to the standard output by default. This includes informational messages, warnings, and errors. By running Terraform commands without any additional configuration, you will see log messages printed to the console.

Log File

You can redirect Terraform log messages to a file by using standard output redirection. For example, you can use the following command to redirect the output to a file: **`terraform apply > terraform.log`** This will store the log messages in the `terraform.log` file instead of displaying them on the console.

Environment Variables

Terraform allows you to configure logging behaviour through environment variables. Two commonly used environment variables are:

`TF_LOG`: This variable controls the verbosity level of the log messages. It accepts values like ``TRACE``, ``DEBUG``, ``INFO``, ``WARN``, or ``ERROR``. For example, you can set ``TF_LOG=DEBUG`` to enable more detailed log messages.

`TF_LOG_PATH`: This variable specifies the path to a log file where Terraform should write log messages. For example, you can set ``TF_LOG_PATH=/path/to/terraform.log`` to store log messages in a specific file.

Debug Output

Terraform commands have a ``-debug`` flag that can be used to enable more detailed debug output. For example: **terraform apply -debug**

This flag provides additional information during command execution, such as HTTP requests and responses, which can be helpful for debugging.

Provider Logging

Some Terraform providers may offer additional logging options specific to their functionality. You can refer to the documentation of individual providers for information on how to enable provider-specific logging if available.

These logging options can help you troubleshoot issues, monitor Terraform execution, and gain insights into the Terraform command lifecycle. Choose the appropriate logging method based on your requirements and the level of detail needed for analysis.

Remember to use caution when storing log files or sharing log information, as they may contain sensitive information about your infrastructure or environment.

Module6. Terraform Pipelining

Agenda

- Terraform Cloud
- Github Actions pipelines
- Jenkins pipelines
- Lab 6 Create a Terraform pipeline

Terraform Cloud

- Terraform Editions
- Introduction to Terraform Cloud
- Terraform Cloud Workspaces

Terraform Editions

- Terraform Open Source

Terraform open source is a free, downloadable tool that you interact with on the command line. It lets you provision infrastructure on any cloud provider and manages configuration, plugins, infrastructure, and state.

- Terraform Cloud

SaaS application that runs Terraform in a stable, remote environment and securely stores state and secrets

- Terraform Enterprise

Terraform Enterprise allows you to set up a self-hosted distribution of Terraform Cloud. It offers customizable resource limits and is ideal for organizations with strict security and compliance requirements.



There are three main editions of Terraform:

Terraform Open Source (OSS)

Terraform Open Source, often referred to as Terraform OSS or simply Terraform, is the free and open-source version of Terraform. It is available under the Mozilla Public License 2.0 (MPL-2.0) and can be downloaded and used by anyone. Terraform OSS provides the core functionality and features for infrastructure provisioning and configuration management.

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 OSS and simplifies certain aspects of infrastructure 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.

Introduction to Terraform Cloud

Introduction to Terraform Cloud

- Run Terraform from the local CLI or in a remote environment, trigger operations through your version control system, or use an API to integrate Terraform Cloud into your existing workflows.
- Ensure that only approved teams can access, edit, and provision infrastructure with Terraform Cloud workspaces, single sign-on, and role-based access controls.
- Securely store and version Terraform state remotely, with encryption at rest. Versioned state files allow you to access state file history.
- Publish configuration modules in the Terraform Cloud private registry that define approved infrastructure patterns.
- Enforce best practices and security rules with the Sentinel embedded policy as code framework. For example, policies may restrict regions for production deployments.

Terraform Cloud is a managed service that offers collaboration, governance, and automation capabilities for managing infrastructure with Terraform. It provides a centralized platform for teams to work together, share infrastructure code, and securely manage infrastructure deployments.

Some key features and benefits of Terraform Cloud:

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

- Not the same as workspaces on your local machine
- Terraform Cloud manages infrastructure collections with *workspaces* instead of directories. A workspace contains everything Terraform needs to manage a given collection of infrastructure, and separate workspaces function like completely separate working directories.

Component	Local Terraform	Terraform Cloud
Terraform configuration	On disk	In linked version control repository, or periodically uploaded via API/CLI
Variable values	As <code>.tfvars</code> files, as CLI arguments, or in shell environment	In workspace
State	On disk or in remote backend	In workspace
Credentials and secrets	In shell environment or entered at prompts	In workspace, stored as sensitive variables

<https://developer.hashicorp.com/terraform/cloud-docs/overview>

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.

Key aspects of Terraform Cloud workspaces:

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

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, GitLab CI/CD, or Azure DevOps, 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, like Jenkins, GitLab CI/CD, or Azure DevOps, can help orchestrate the deployment process, manage infrastructure state, and handle environment specific configurations.

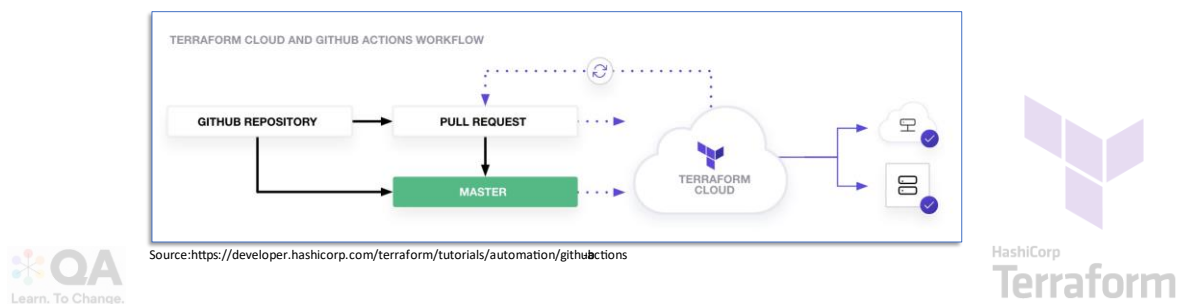
Infrastructure as Code (IaC):

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

Introduction

GitHub Actions add continuous integration to GitHub repositories to automate your software builds, tests, and deployments. Automating Terraform with CI/CD enforces configuration best practices, promotes collaboration and automates the Terraform workflow



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

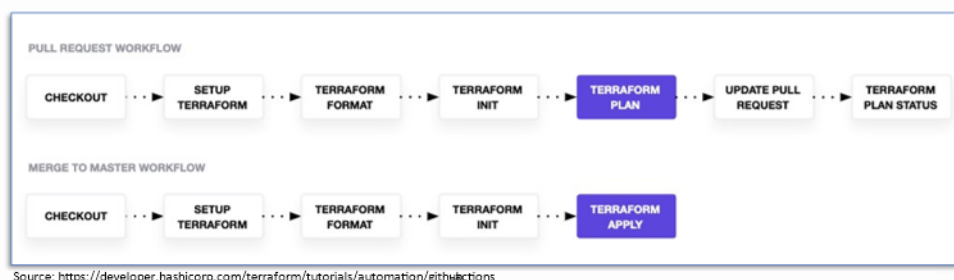
The specific implementation of Terraform CI/CD pipelines can vary based on the CI/CD tooling you choose, your infrastructure requirements, and the desired workflows. CI/CD tools like Jenkins, GitLab CI/CD, Azure DevOps, and others provide plugins, integrations, and documentation to help you configure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

Workflow

- Create TF files
- Create GitHub Actions workflow YAML file
 - Triggers on push or pull_request targeting main branch and terraform path changes
 - Environment variable for backend and possibly credential – secrets
- Job definition
 - Pull Request received – generate plan of incoming changes, add this as comment to the Pull Request
 - If Pull request approved – merge changes to 'main', triggering `terraform apply`

Pull and Merge Workflows



Jenkins Pipelines

Jenkins pipelines

- Understand the role of Jenkins in CI/CD workflows.
- Learn the basics of setting up Jenkins for Terraform pipelines.
- Configure and execute a simple Terraform deployment using Jenkins.

Introduction to Jenkins

- What is Jenkins? (Definition and Key Benefits)
- Role of Jenkins in automating Terraform workflows.
- Overview of tasks Jenkins can automate (terraform init, plan, apply)

What is Jenkins ?

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.

Understanding Jenkins Basics

- Key Concepts
 - Jobs, Pipelines, and Stages.
 - Plugins and integrations.
- Jenkins Setup
 - Jenkins installation on an EC2 instance.
 - Configuring AWS credentials and Terraform plugins.



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.

Jenkins Authentication Options

- Jenkins Built-In User Database.
- LDAP / Entra ID
- OAuth and Single Sign-On (SSO)
- SAML (Security Assertion Markup Language)
- Custom Authentication via Plugins
 - OpenID Connect Plugin
 - Keycloak Plugin



Jenkins Authorization Plugin

- Allows for RBAC.
- Typical roles created include
 - Administrator - Full access
 - Project Manager – Manage folders and jobs
 - Developer – Create and modify jobs for their project
 - QA/Tester – Build and test jobs
 - Viewer/Read Only – View configs, jobs and logs
 - Build Executor – Trigger jobs for CI/CD



Building a Simple Terraform Pipeline

Building a Simple Terraform Pipeline

- Pipeline Overview
 - Jenkinsfile
 - Stages for a basic Terraform pipeline
 - Initialization
 - Planning
 - Applying.

```
jenkins-jobs / Jenkinsfile
jenkins2087 update Jenkinsfile
Code | Blame | 10 Lines (52 SLOC) | 2.4K, 10 | Code 55% faster with GitHub Copilot
1 pipeline {
2   agent any
3   stages {
4     stage('Clone Repository') {
5       steps {
6         git branch: 'main', url: 'https://github.com/terraform-docs/terraform-docs.git'
7       }
8     }
9     stage('Checkout Terraform') {
10      steps {
11        checkout 'main'
12      }
13    }
14    stage('Terraform Init') {
15      steps {
16        terraform {
17          source './'
18        }
19        terraform init
20      }
21    }
22    stage('Terraform Plan') {
23      steps {
24        terraform {
25          source './'
26        }
27        terraform plan
28      }
29    }
30  }
31}
```



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.