Terraform

HashiCorp
Terraform

By: Theepana Govintharajah

Terraform

- **Open-source**

- **Infrastructure as Code (IaC) tool:** Allows you to describe your infrastructure resources and their relationships using a high-level configuration language (HashiCorp Configuration Language (HCL) or JSON).

- **Uses declarative language:** Defining only the desired state (final result) of your infrastructure rather than scripting the every steps to achieve that state.
  (In imperative approach you will define every step of execution)

- **Resource Dependencies and Ordering**: Terraform understands the dependencies between resources defined in your configuration. It automatically determines the correct order for provisioning and managing resources, ensuring that dependencies are resolved correctly.

By: Theepana Govintharajah

Terraform

- **State Management:** Terraform maintains a state file that tracks the state of your infrastructure. This state file keeps track of the resources you provisioned and their current state, allowing Terraform to make intelligent decisions about creating, updating, or destroying resources based on changes in your configuration.

- **Remote State Backend:** Terraform supports remote state storage, allowing you to store the state file in a remote location such as an object storage service. This facilitates collaboration among team members and helps maintain a single source of truth for the infrastructure state.

- **Resource Management:** Terraform provides a wide range of resource providers, allowing you to define and manage various infrastructure resources such as virtual machines, networks, storage, databases, and more. These resources are defined using Terraform configuration files

By: Theepana Govintharajah

Terraform

- **Multi-Cloud Provisioning:** Terraform is cloud-agnostic and supports multiple cloud providers, including AWS, Azure, Google Cloud, and many others. This allows you to manage infrastructure resources consistently across different cloud platforms using the same configuration syntax.

- **Cloud-Agnostic and On-Premises Support:** While Terraform supports multiple cloud providers, it is not limited to cloud environments. It can also be used to provision and manage infrastructure resources in on-premises data centers, allowing you to achieve consistent infrastructure management across hybrid or multi-cloud environments.

- **Version Control Integration:** Terraform configuration files can be stored in version control systems such as Git. This enables collaboration, history tracking, and the ability to roll back to previous versions of infrastructure configurations.

By: Theepana Govintharajah

Terraform Vs. Ansible

Terraform Vs. Ansible

- Ansible and Terraform are both popular tools for automating infrastructure provisioning and management.
- Both are open source tools.
- Both embrace the Infrastructure as Code (IaC) approach.

That's why, you might have some confusions, which tool should be used for which purpose?

Purpose of both tools are important in DevOps toolchain to automate and manage different aspects of your infrastructure and software deployments.

So let's see how they differ from each other.....

By: Theepana Govintharajah

Terraform Vs. Ansible

Ansible and Terraform are both popular DevOps tools but serve different purposes and have distinct features.

Terraform	Ansible
Main purpose of this tool is **Infrastructure Provisioning**	Main purpose of this tool is **Configuration Management**
Declarative approach: Define the desired state of your infrastructure, automatically manages the updates to make the actual state match the desired state.	**Procedural approach:** Specify the steps and actions to be executed in order to achieve a desired state.
More advanced in Orchestration	

By: Theepana Govintharajah

Terraform Architecture

Terraform Architecture

Terraform' s architecture is composed of two main components:

1. **Terraform Core**
2. **Terraform providers**

Terraform Core

Terraform Core is responsible for managing the provisioning and state management of infrastructure resources. Core uses two input sources in order to do its job. It takes the **Terraform configuration file**, which defines the desired infrastructure state, and compares it with the current state stored in the **Terraform state**.

Based on this comparison, Terraform Core determines the necessary actions to bring the actual state in line with the desired state. It generates an execution plan that outlines the steps for creating, modifying, or destroying resources to achieve the desired state.

By: Theepana Govintharajah

Terraform Architecture

Terraform Providers

Providers are plugins that interface with various infrastructure platforms or services. They provide Terraform with the necessary capabilities to interact with specific cloud providers, infrastructure-as-a-service platforms, or higher-level components like Kubernetes.

Terraform has over a hundred providers for these different technologies. Each provider enables Terraform to manage resources and perform actions specific to that platform.

Together, Terraform Core and Providers work in conjunction to automate the provisioning and management of infrastructure resources. Terraform Core generates execution plans and communicates with the appropriate providers to carry out the actions defined in the plan.

By: Theepana Govintharajah

Terraform Commands

Terraform Commands

terraform init: Initializes a Terraform working directory, downloads the necessary provider plugins, and sets up the backend configuration. By running terraform init before any other Terraform command, you ensure that your working directory is properly configured and ready for provisioning and managing your infrastructure resources. (prepares your environment for subsequent Terraform operations.)

terraform refresh: Bring the Terraform state file up to date with the actual state of the infrastructure resources, providing an accurate representation of the current state for future operations. When you run terraform refresh, Terraform will query the infrastructure provider (such as AWS, Azure, or Google Cloud) to fetch the current state of the resources. It compares this current state with the state recorded in the Terraform state file. The main purpose of terraform refresh is to update the state file with the latest information from the infrastructure provider. This can be useful in scenarios where the state file may have become out of sync due to external changes made to the infrastructure resources. (It performs a read-only operation and does not make any changes to the infrastructure.)

By: Theepana Govintharajah

Terraform Commands

terraform plan: This command is used to create an execution plan that outlines the actions Terraform will take to achieve the desired state defined in your configuration files. When you run terraform plan, Terraform compares the current state of your infrastructure (as recorded in the state file) with the desired state defined in your configuration files. It analyzes the differences between the two states and determines the actions required to bring the actual state in line with the desired state. Note that running terraform plan does not make any changes to your infrastructure. It is a read-only operation that provides you with an overview of the changes that Terraform will apply when you run terraform apply.

terraform apply: Applies the plan and makes the changes to your infrastructure. The terraform apply command combines the functionality of terraform refresh, terraform plan, and terraform apply into a single command. By using terraform apply, you can streamline the process and execute all the necessary steps in a single command. It saves time and simplifies the workflow, especially in cases where you are confident in the changes and want to proceed without explicitly running separate terraform refresh and terraform plan commands.

Terraform Commands

terraform destroy: Destroys all the resources created by Terraform, removing them from the infrastructure. Running terraform destroy does not remove the state file. The state file contains the information about the previously provisioned resources. If you want to completely clean up, you can manually delete the state file

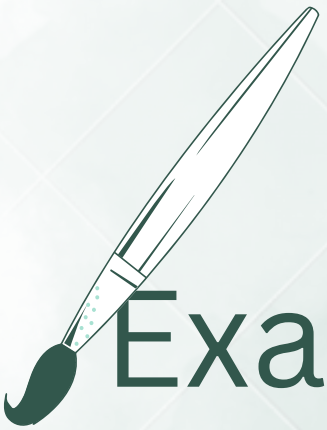
terraform state: This command is used to interact with and manage the state of your infrastructure. It allows you to view, modify, and perform operations on the resources recorded in the Terraform state file.

- **terraform state list:** Lists all the resources managed by Terraform in the state file. It provides a summary of the resources and their addresses.

- **terraform state show [resource_address]:** Displays the details of a specific resource in the state file. You need to provide the resource's address, which can be obtained from the terraform state list command.

- **terraform state rm [resource_address]:** Removes a resource from the state file. This is typically used when you want to delete a resource managed by Terraform.

By: Theepana Govintharajah



Example for Terraform Configuration file

Terraform Configuration file

Here's an example of a Terraform configuration file, often named main.tf, that provisions an AWS EC2 instance:

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

# Resource configuration
resource "aws_instance" "example" {
  ami           = "ami-0c94855ba95c71c99"  # Amazon Linux 2 AMI ID
  instance_type = "t2.micro"
  key_name      = "my-keypair"

  tags = {
    Name = "my-instance"
  }
}

# Output configuration
output "public_ip" {
  value = aws_instance.example.public_ip
}
```

By: Theepana Govintharajah

Best practices to follow in Terraform

Best Practices

1) Manipulate state only through TF commands

State file is the simple JSON file, so technically you could make adjustments to the state file directly by manually changing stuff inside. However best practise is only change the state file contents through Terraform commands execution (terraform apply, terraform state). Do not manually manipulate it, otherwise you may get some unexpected results.

2) Always set up a shared remote storage for the state file

Terraform creates state files automatically when you first execute "terraform apply" command, by default locally on your machine. But if we are working in a team, other team members also need to execute terraform commands and they will also need the latest state file before making their own updates. So best practise is to configure shared remote storage for the state file. In practice, remote storage backend for state file can be Amazon's S3 bucket, terrafrom Cloud, Azure, Google cloud, etc.

By: Theepana Govintharajah

Best Practices

3) Use State Locking (Locking the state file)

What if two team members execute Terraform commands at the same time? What happens to the state file when you have concurrent changes? You might get a conflict or your state file will be corrupted. To avoid changing terraform state at the same time, best practise is locking the state file until an update is fully completed and then unlock it for the next command. This way you can prevent concurrent edits to your state file. In practise, you will have this configured in your storage backend. For example, in S3 bucket: Dynamodb service is automatically used for state file locking. Note that, not all storage backends support this, so be aware of that when choosing remote storage for the state file. If supported, Terraform will lock your state file automatically.

4) Back up your state file

Something you may lost your state file accidentally. To avoid this, best practice is to back up your state file. In practice you can do this by enabling versioning for it. And many storage backends will have such a feature. This means that you have a history of state changes and you can reverse to any previous terraform state if you want to.

By: Theepana Govintharajah

Best Practices

5) Use one dedicated state file per environment

You have one state file for your infrastructure. But usually you will have multiple environments like development, testing, and production. Can you manage all the environments with one state file? Best practice is to use one dedicated state file per environment. Each file will have its own storage backend.

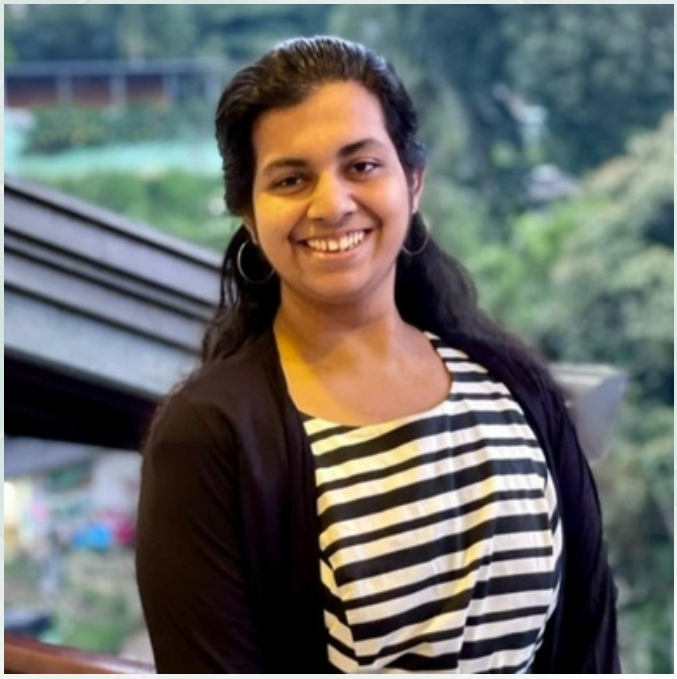
6) Host Terraform scripts in Git repository

When you are working as team, it's important to share the code in order to collaborate effectively, so as the best practice you should host terraform code in git repository just like the application code. This is not only beneficial for effective collaboration, but also version control for your Infrastructure as Code (IaC) changes.

7) Apply Infrastructure changes Only through CD pipeline

Eventually you want to update your infrastructure with changes. So best practice is applying Terraform commands to apply changes in a continuous deployment pipeline, instead of team members manually updating the infrastructure by executing Terraform commands from their own computers. It should happen only from an automated build. This way you have a single location from which all infrastructure changes happen.

Follow me for more similar posts



By: Theepana Govintharajah