Welcome to Terraform

Introduction

- Terraform is a tool for building, changing, and versioning infrastructure safely and efficiently.
- Terraform can manage existing and popular service providers as well as custom in-house solutions.
- Configuration files describe to Terraform the components needed to run a single application or your entire datacenter.
- Terraform generates an execution plan describing what it will do to reach the desired state, and then executes it to build the described infrastructure.
- As the configuration changes, Terraform is able to determine what changed and create incremental execution plans which can be applied.
- The infrastructure Terraform can manage includes low-level components such as compute instances, storage, and networking, as well as high-level components such as DNS entries, SaaS features, etc.

Why Terraform?

- **Terraform is an "orchestration" tool, not an "automation tool:"** Automation is a task completed without human intervention, while Orchestration means, taking a task and creating a workflow, or running several automated tasks called as processes.
- For example, orchestration is a way of combining multiple automation tasks to create IP or creating a security group
- **Terraform is "Declarative" not "Procedural/Imperative:"** These are contrasting programming patterns.
- Declarative programming does not control the flow of the program, you just say what you want and not say how to do it.
- Procedural programming, on the other hand, you define the whole process and provide the steps how to do it.

Why Terraform?

- **Terraform follows Client Only Architecture, not Client/Server Architecture**: Chef, Ansible all follow client/server architecture. You issue commands on a client system and it executes your commands and stores the state of your system.
- The server talks to agents, which must installed on every instance you want to configure. With this architecture, moving parts coming as a perk and may cause new failure modes. While Terraform uses the cloud provider API's to configure your infrastructure.
- **Terraform has Multi-Provider Support**: Terraform provides convenience to switch between different cloud providers like Google, AWS, Open Stack, Azure. Terraform allows you to write code specific to each provider.

Key Features

- **Infrastructure as Code**
  - Infrastructure is described using a high-level configuration syntax. This allows a blueprint of your datacenter to be versioned and treated as you would any other code. Additionally, infrastructure can be shared and re-used.
- **Execution Plans**
  - Terraform has a "planning" step where it generates an execution plan. The execution plan shows what Terraform will do when you call apply. This lets you avoid any surprises when Terraform manipulates infrastructure.
- **Resource Graph**
  - Terraform builds a graph of all your resources, and parallelizes the creation and modification of any non-dependent resources. Because of this, Terraform builds infrastructure as efficiently as possible, and operators get insight into dependencies in their infrastructure.

Key Features

- **Change Automation**
  - Complex changesets can be applied to your infrastructure with minimal human interaction. With the previously mentioned execution plan and resource graph, you know exactly what Terraform will change and in what order, avoiding many possible human errors.

Terraform Building Blocks

- **Providers**

A provider is responsible for understanding API interactions and exposing resources. Providers generally are an IaaS (e.g. Alibaba Cloud, AWS, GCP, Microsoft Azure, OpenStack), PaaS (e.g. Heroku), or SaaS services (e.g. Terraform Cloud, DNSimple, CloudFlare).

- **Resources -** Resource is part of the infrastructure create on the provider. Every Provider provides resources.
- **Outputs**
- **Variables -** To give options for the user to enter different values to resources.

Terraform Building Blocks

Datasources

- A data source is accessed via a special kind of resource known as a data resource, declared using a data block:

```
data "aws_ami" "example" {
  most_recent = true

  owners = ["self"]
  tags = {
    Name   = "app-server"
    Tested = "true"
  }
}
```

- A data block requests that Terraform read from a given data source ("aws_ami") and export the result under the given local name ("example"). The name is used to refer to this resource from elsewhere in the same Terraform module, but has no significance outside of the scope of a module.

Terraform Building Blocks

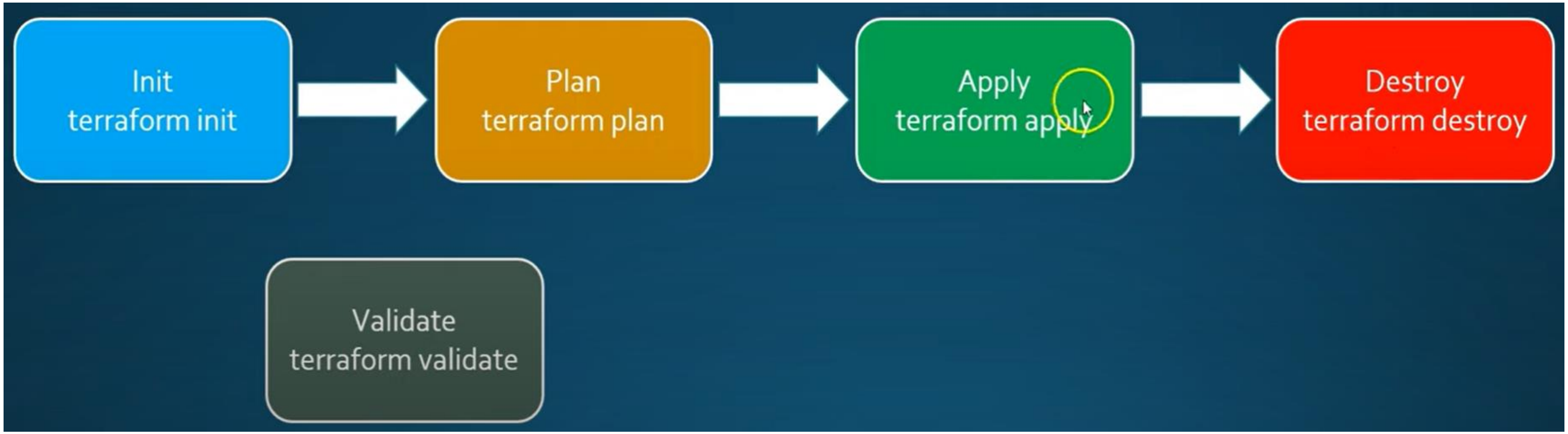
- Terraform templates are written in Custom DSL. This DSL mostly looks like JSON.
- Terraform templates end with .tf extension

Resource Dependencies

- When Resource A requires resource B to be present (or already existing) this is called as dependency
- In Terraform terms you have to create resource B before resource A
- To Demonstrate this lets add subnets to VPC. To create subnet resource vpc id is required (vpc has to be existing)
- To create resource dependencies use the following expression

```
"${<resource-type>.<resource-name>.<attribute-name>}"
"${aws_vpc.myvpc.id}"
```

Terraform Workflow



Terraform Commands

- **terraform init**
  - Use **terraform init**, a command to initialize download provider plugins to your local system.
- **terraform plan**
  - To see what are you going to deploy
  - The plan command tells you what TF is about to do; a few will be filled in and some will be computed.
- **terraform apply**
  - Execute the terraform plan
- **terraform destroy**

The State of Terraform

- How does TF know what needs to be changed or destroyed when we issue the plan/apply command?
- Usually, Terraform records the state of everything it has done, so it stores the state locally as .**tfstate** files.
- Usually, it's not advised to store the state in local hard disk, as it may contain sensitive data, so it would be good to use a backend like S3 to store Terraform state information.
- JSON
- Updated when Terraform runs
- Refreshed before an operation takes place
- Backwards compatible between TF versions

Terraform Graph

- The terraform graph command is used to generate a visual representation of either a configuration or execution plan.
- The output is in the DOT format, which can be used by GraphViz to generate charts.

```
terraform graph | dot -Tsvg > graph.svg
```

```
choco install graphviz
```

OR

Download from

[Download | Graphviz]

Q & A