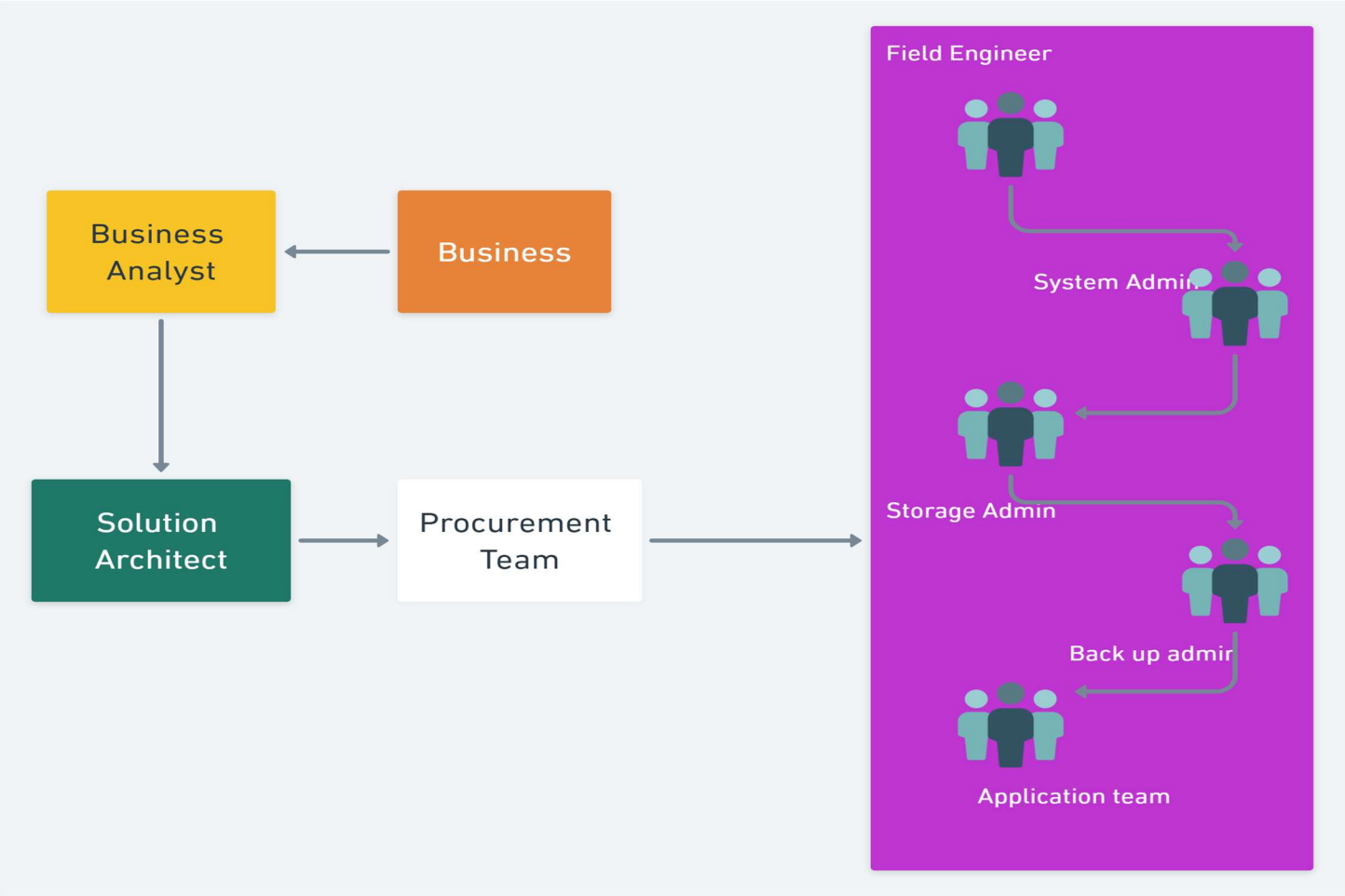
Terraform Documentation



Challenges with traditional IT infrastructure

- Slow deployment
- Expensive
- Limited automation
- Human Error
- Inconsistency

Example:
Let's say you are working in large organization, and you need to setup this list of infrastructure on cloud and here the list goes on

Linux instance with auto scaling groups + ELB

1. AWS aurora
2. AWS EKS cluster (3 nodes)
3. AWS Route53
4. AWS CloudFront

So, it will be difficult to provision such a large infra and then deprovision it as it required and involve human error a lot, so from here we require some kind of automation and managed, **the process of managing and automating the whole infrastructure provisioning is known as Infrastructure as Code (IaC).**

Infrastructure as a code (IaC) is the managing and provisioning of infrastructure through code instead of manual process.

Definitions of Terraform:
Def1: Terraform is an infrastructure as code tool that allows you to build, change and version infrastructure safely and efficiently.
Def2: Terraform is a tool for building, changing and versioning infrastructure safely and efficiently which enables you to create, change and improve infrastructure securely and predictably.

How to install, Verify and Configure AWS CLI:

Install:

- Download https://awscli.amazonaws.com/AWSCLIV2.msi
- Open with administrator and proceed to install.

Verify:

- Open PowerShell and type 'AWS', if it returns then AWS CLI installed successfully.

Configure:

- Create AWS free tier account if no AWS account.
- IAM Users->Users->Create new user->create Access ID and secret key, and download/save Access key and secret.
- Open PowerShell and type 'aws configure'
- Provide Access key ID and Secret key

Terraform Installation:

Process1:

- Got to https://developer.hashicorp.com/terraform/downloads
- Download the file according to system OS and install

Process2:

- Open PowerShell with administrator.
- Copy the link and paste in PowerShell to install chocolatey
- Link: *Set-ExecutionPolicy Bypass -Scope Process -Force; [System.Net.ServicePointManager]::SecurityProtocol = [System.Net.ServicePointManager]::SecurityProtocol -bor 3072; iex ((New-Object System.Net.WebClient).DownloadString('https://community.chocolatey.org/install.ps1'))*
- Type Choco install terraform (it will download and install terraform in windows)
- To verify, type terraform in PowerShell it will return list of commands in terraform.

Basic commands of terraform:

Usage: terraform [global options] <subcommand> [args]

1. .tf is the extension for terraform files.
2. The available commands for execution are listed below.
3. The primary workflow commands are given first, followed by less common or more advanced commands.

Main commands:

init	Prepare your working directory for other commands
validate	Check whether the configuration is valid
plan	Show changes required by the current configuration
apply	Create or update infrastructure
destroy	Destroy previously created infrastructure

All other commands:

console	Try Terraform expressions at an interactive command prompt
fmt	Reformat your configuration in the standard style
force-unlock	Release a stuck lock on the current workspace
get	Install or upgrade remote Terraform modules
graph	Generate a Graphviz graph of the steps in an operation
import	Associate existing infrastructure with a Terraform resource
login	Obtain and save credentials for a remote host
logout	Remove locally stored credentials for a remote host
output	Show output values from your root module
providers	Show the providers required for this configuration
refresh	Update the state to match remote systems
show	Show the current state or a saved plan
state	Advanced state management
taint	Mark a resource instance as not fully functional
untaint	Remove the 'tainted' state from a resource instance
version	Show the current Terraform version
workspace	Workspace management

Explanation:

Terraform Get

The terraform get command is used to download and update modules mentioned in the root module

Important points:

1. The modules are downloaded into a .terraform sub directory of the current working. Don't commit ths directory to your version.
2. The get command supports the following options
    - -update -If specified, nodule that are already downloaded will be checked for updates and the updates with be downloaded if present.
    - -no-color Disable text coloring in the output.

Terraform get downloads and update modules mentioned in the root module Terraform get -update=true modules already downloaded will be checked for updates and

Terraform Validate

1. This command check whether the execution plan for a configuration matches your expectations before provisioning or changing infrastructure.
2. The terraform validate command validates the configuration files in a directory, referring only to the configuration and not accessing any remote services such as remote stare, provider APIs, etc.
3. Validate run checks that verify whether a configuration is syntactically valid and internally consistent, regardless of any provided variable or existing state. It is thus primarily useful for general verification of reusable modules, including correctness of attribute names and value types.

Terraform validate validate configuration files for syntax Terraform validate -backend=false validate code skip backend validation

Terraform init

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

Important points:

1. This command is always safe to run multiple times, to bring the working directory up to date with changes in the configuration.
2. Through subsequent runs may give errors, this command will never delete your existing configuration or state.
3. It initializes a working directory containing terraform configuration files.
4. Performs backend initialization, storage for terraform state file, modules installation.
5. It download from terraform registry to local path, provider(s) plugins installation, the plugins are downloaded in the sub directory of the present working directory at the path of .terraform/plugins
6. Supports -upgrade to update all previously installed plugins to the newest version that complies with the configuration version constraints, does not delete the existing configuration or state.

Terraform init initialize directory, pull down providers Terraform init -get-plugins=false initialize directory, do not download plugins Terraform init -verify-plugins=false initialize directory, do not verofy plugins for Hashicorp signature Terraform init -lock=false Disable locking of state files during state-related operations

Terraform plan

Run terraform plan to check whether the execution plan for a configuration matches your expectations before provisioning or changing infrastructure.

The terraform plan command creates an execution plan. By default, creating plan consists of:

- Reading the current stare of any already existing remote objects to make sure that the terraform stare is up to date.
- Comparing the current configuration to the prior state and noting any differences.
- Proposing s set of change actions that should, if applied, make the remote objects match the configuration.

Terraform plan creates an execution plan (dry run) Terraform plan -out=path save generated plan output as a file Terraform plan -destroy outputs a destroy plan

Terraform apply

1. The terraform apply command executes the actions proposed in a terraform plan.
2. The most straightforward way to use terraform apply is to run it without any arguments at all, in which case it will automatically create a new execution plan (as if you had run terraform plan) and then prompt you to approve that plan, before taking the indicated actions.

Another way to use terraform apply is to pass it the filename of a saved plan file you created earlier with terraform plan -out=…., in which case terraform will apply the changes in the plan without any confirmation prompt. This tow step workflow is primarily intended for when running terraform in automation.

Terraform apply Executes changes to the actual environment Terraform apply -auto-approve Apply changes without being prompted to enter 'yes' Terraform apply -refresh=true Update the state for each resource prior to planning and applying Terraform apply -refresh=false Do not reconcile state file with real world resources Terraform apply -input=false Ask for input for variables if not directly set Terraform apply -var 'foo=bar' set a variable in the terraform configuration, can be used multiple times Terraform apply -var-file=foo Specify a file that contains key/value pairs for variable values. Terraform apply -target Only apply/deploy changes to the targeted infrastructure Terraform apply -parallelism=5 Number of simultaneous resource operations Terraform apply -lock=true lock the state file so it can't be modified by any other terraform apply or modification actions.

Terraform destroy

1. The terraform destroy command is a convenient way to destroy all remote objects managed by a particular terraform configuration.
2. While you will typically not want to destroy long lived objects in production environment, terraform is sometimes to manage ephemeral infrastructure for development purposes, in which case you can use terraform destroy to conveniently up all those temporary objects once you are finished with your work.

Terraform destroy Destroy all remote objects managed by a particular terraform configuration **Terraform destroy** -auto-approve Destroy/cleanup without being prompted to enter 'yes' **Terraform destroy** -target Only destroy the targeted resources and its dependencies,

Terraform Workspaces

In terraform CLI, workspace are separate instance of state data that can be used the same working directory. You can see workspaces to manage multiple non overlapping group of resources with the same configuration.

Important points:

- It helps manage multiple distint set of infrastructure resources or environments with the same code.
- Just need to create needed workspace and use them instead of creating a directory for each environment to manage.
- State files for each workspace are stored in the directory terraform.tfsate.d

'Terraform workspace new' creates new workspace and switches to it as well 'Terraform workspace select' helps to select workspace 'Terraform workspace list' the workspace lists and show the current active one with 'Terraform workspace show' show the current directory 'Terraform workspace delete' delete the workspace

Examples

- Terraform workspace new dev
- Terraform workspace list

- Terraform workspace select dev
- Terraform workspace delete dev

Terraform variables, Input & output:

Variable in terraform are the way to describe value in terraform configuration file. Depending upon the usage the variables are divided into two types.

1. Input
2. Output

Input variable:

- In order to tailor out deployment. We use terraform input variable as parameters to enter data at run time. Although input terraform variables can be provided in the main.tf configuration file. It is advisable to specify them in a separate variable.tf file for better readability and organization.

- A variable is defined by using a "variable block" with label. Then label is the name of the variable and must be unique among all the variables in the same configuration.

Example:

```
variable 'macchine_id' {
        Type = string
}
```

Output variable:

- Output variables are used to output the parameters written in the configuration file.

Example:

```
Variable "set_password" {
        Value = var.machine_id
}
```

Terraform Tfvars and Autovars

1. A terraform.tfvars file used to to 'set the actual values' of the variable.terraform.tfvars and .auto.tfvars are automatically loaded without any additional options, they behave similarly to defaults, but the intent of these is difficult.

2. When running terraform in automation, some users have their automation to generate a terraform.tfvars file or .auto.tfvars just before running terraform in order to pass in values the automation knows, such as what environment the automation is running for etc.

3. Create a terraform.tfvars file or files named .auto.tfvars, which are treated the same as -var-file arguments but are loaded automatically.

Example:

Input.tf

```
        Variable "ec2_instance_type" {

        }
```

Terraform.tfvars

```
        ec2_instance_type = "t2.nicro"
```

output.tf

```
        output "ec2_insta" {
        value = var.ec2_instance_type
        }
```

Terraform state and state files

State:

1. Terraform uses state data to remember which real world object corresponds to each resource in the configuration, this allows it to modify on existing object its resource declaration changes.
2. Terraform updates state data, usually to compensate for changes to the configuration or the real managed infrastructure.

Some important points:

- State helps keep track of the infrastructure terraform manages
- It maps real world resources to terraform configuration.
- It stored locally in the terraform.tfstate or can be stored in the remote server such as terraform cloud, hashicorp consul, azure blob storage, Google cloud storage, Alibaba cloud OS

Terraform state list List all resource in the state file Terraform state list resource_reference Only list resource with the give name Terraform state mv Move an item in the state file Terraform state rm Remove items from the state file Terraform state pull Pull current state and output to stdout Terraform state push Update remote state from a local state file

State file:

- Terraform must store state about managed infrastructure and configuration.
- This state is used by terraform to map real world resources to your organization, keep track of matadata and to improve performance for large infrastructure.
- This state is stored by default in a local fine named 'terraform.tfstate'
- Terraformused this local state to create plans and make changes to your infrastructure
- The primary purpose of terraform state is to store binding between objects in a remote system and resource instances declared in your configuration.

Terraform providers

Terraform configuration relies on plugins called "providers" to interact with cloud providers, SaaS providers, and other APIs. Terraform configurations must declare which providers they require so that terraform can install and use them. Additionally, some providers require configuration like endpoints URL or cloud regions) before they can be used.

```
terraform {
        required_providers {
        Aws = {
                Source = 'hashicorp/aws"
                Version = "~>3.0"
                }
        }
}

#configure the AWS provider

Provider "aws" {
        Region = "ap-south-1"
}
```

Launch EC2 instance using Terraform

Ec2.tf

```
Resource “aws_instance” “ec2_example” {
   ami             = “”
   instance_type = “${var.instance_type}”
  key_name = {
     Name = “Hello-world”
  }
}
```

Variables.tf

```
Variable “instance_type” {

}
```

Terraform.ftvars

Instance_type = “t2.micro”

Providers.tf

```
Provider “aws” {
   access_key = “”
   secret_key = “”
   region = “”
}
```

Output.tf

```
Output “public_ip” {
   Description = “List of public IP address assigned to ec2_example”
   Value = aws_instance.ec2_example.public_ip
}
Output “id” {
   Description = “List of IDs of instances”
   Value = aws_instance.ec2_example.id
}
Output “availability_zone” {
   Description = “List of availability zones of instances”
   Value = aws_instance.ec2_example.availability_zone
}
Output “public_dns” {
   Description = “List of public DNS names assigned to the instance”
   Value = aws_instance.ec2_example.public_dns
}
```