**IaC using Terraform Templates**

* Overview of Terraform
* Terraform Files Structure
* Terraform Commands
* Writing HCL with the Azure Provider
* Working with modules
* Working with Terraform Enterprise (user/developer personal)

**Infrastructure as Code (IaC)**

Infrastructure as code is provisioning infrastructure through **software (**It's not a manual process ) to achieve **consistent** and **predictable** environments

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 using Source Control system like GitHub or Bitbucket.

Your Infrasturcture is going to look exactly like the configuration files say it should look. That's very important, especially when you have multiple environments that will be running the same version of an application.

You should be storing that code somewhere in source control. The code is versioned, and multiple developers can work on it simultaneously.

When it comes to the actual code itself, there are two different approaches to implementing infrastructure as code.

1. Declarative
2. Imperative

**Benefits of IaC**

* Automated deployment
* Repeatable process
* Consistent environments
* Reusable components
* Documented architecture

**Cloud specific Native IaC solutions**

1) Azure - ARM

2) AWS - CloudFormation

3) GCP - Cloud Deployment Manager

### **Overview of Terraform**

* Terraform by HashiCorp, is an open-source tool for building, changing, and versioning infrastructure safely and efficiently. Terraform can manage existing and popular cloud service providers as well as custom in-house solutions.
* Terraform's command-line interface provides you with a simple way to provision Azure resources such as virtual machines, containers, and networks.

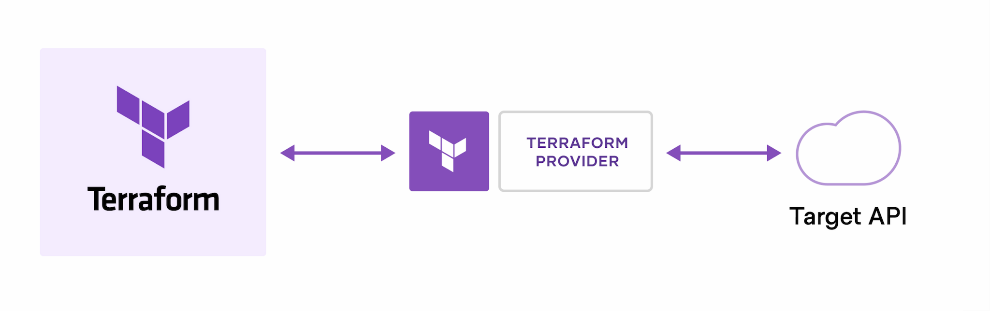
How Azure Works:

Diagram

Description automatically generated

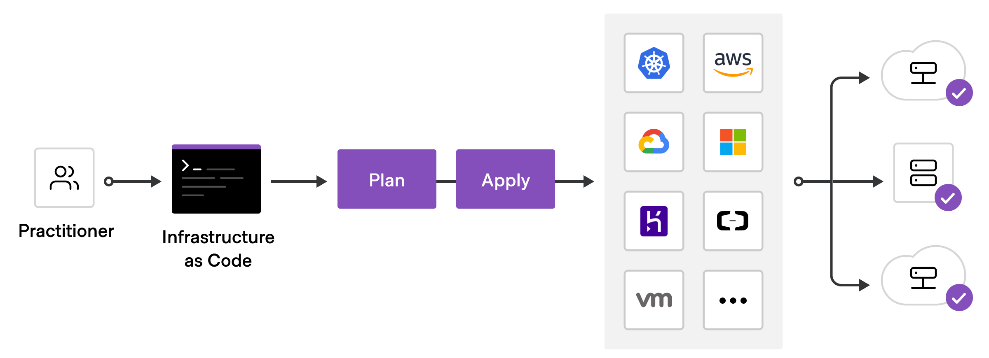
**How does Terraform work?**

* Terraform creates and manages resources on cloud platforms and other services through their application programming interfaces (APIs). Providers enable Terraform to work with virtually any platform or service with an accessible API.
* HashiCorp and the Terraform community have written more than 1700 providers to manage thousands of different types of resources and services (this number continues to grow). We can find all publicly available providers on the Terraform Registry, including Amazon Web Services (AWS), Azure, Google Cloud Platform (GCP), Kubernetes, Helm, GitHub, Splunk and many more.



Steps to follow in order to deploy infrastructure with Terraform:

* **Scope** - Identify the infrastructure for your project.
* **Author** - Write the configuration for your infrastructure.
* **Initialize** - Install the plugins Terraform needs to manage the infrastructure.
* **Plan** - Preview the changes Terraform will make to match your configuration.
* **Apply** - Make the planned changes.



Configuration files (use declarative syntax) describe to Terraform the components needed to run a single application or your entire datacenter.

Block Syntax.

block\_type "label" "name\_label" {

  key = "value"

  nested\_block {

    key = "value"

  }

}

Terraform Features:

* **Declarative**: Terraform uses Hashicorp Configuration Langauge which provides a declarative syntax to develop infrastructure as code. The HCL configuration language helps declare the target state of cloud resources to be provisioned.
* **Cloud** **Agnostic**: Terraform is a great tool to automate [multi-cloud deployments](https://spacelift.io/blog/multi-cloud-infrastructure-strategy). Its modular architecture enables working with multiple well-known cloud vendors simultaneously.
* **Ecosystem**: The provider and module ecosystem of Terraform is well established. Certified modules and [providers](https://spacelift.io/blog/terraform-providers) are available on **Terraform Registry** to be readily used. Customers can create and publish their own modules both publicly and privately.
* **Extendible**: Terraform can be extended to support lesser-known or private data centers.
* **Agentless**: Terraform works with the programmatic access provided by cloud provider APIs. Thus there is no need to install agents.

Benefits of Using Terraform

* Consistency: With infrastructure being managed via code, it becomes very easy to version and track changes. Since cloud resource provision happens logically, we can rely on its consistency to create a scaled set of infrastructure.
* Automation: Terraform workflow manages [**the lifecycle**](https://spacelift.io/blog/terraform-resource-lifecycle) of cloud resources – from their creation till they are destroyed or decommissioned. This provides an opportunity to enable end-to-end automation right from the infrastructure layer. Automation workflows also assist in strengthening deployment strategies.
* **Less Risk**: Using Terraform to develop infrastructure as code provides validation beforehand. I also isolates manual efforts and errors associated with it. Thus once developed, infrastructure provisioning and de-provisioning cycles execute identically.
* **Modular and DRY (Don't Repeat Yourself):** Infrastructure can be developed in a modular way so that it can be reused across multiple projects. This approach also enables organizations to ingrain their security and governance practices around infrastructure resources, thus offsetting the initial efforts spent every time a new project kicks off.

Core Components of Terraform

1. Executable
2. Configuration Files
3. Provider Plugins
4. State data

Terraform Components

1. **Configuration** **files**: Text-based configuration files allow you to define infrastructure and application configuration.
2. [**Resource**](https://www.terraform.io/docs/configuration/resources.html) is an entity of a cloud service declared in Terraform code that is created according to specified and inferred properties. Multiple resources form infrastructure with their mutual connections.
3. **Terraform state** is the mechanism via which it keeps track of resources that are actually deployed in the cloud. State is stored in backends (locally on disk or remotely on a file storage cloud service or specialized state management software) for optimal redundancy and reliability.
4. **Modules** in Terraform (akin to libraries in other programming languages) are parametrized code containers enclosing multiple resource declarations. They allow you to abstract away a common part of your infrastructure and reuse it later with different inputs. A module consists of a collection of .tf and/or .tf.json files kept together in a directory.

Directory Structure:

* Root Module
  + Configuration Files (.tf)
  + Variable (terraform.tfvars / random.auto.tfvars / random.tfvars)
  + State File (terraform.tfstate)
  + .terraform
    - providers
    - modules
  + Child Module

**Installation and Setup**

The Terraform binary can be downloaded from the <https://www.terraform.io/downloads>

The installation steps generally consist of downloading the appropriate binary and setting up the path variable.

**For Linux:**

wget -O- https://apt.releases.hashicorp.com/gpg | sudo gpg --dearmor -o /usr/share/keyrings/hashicorp-archive-keyring.gpg

echo "deb [signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg] https://apt.releases.hashicorp.com $(lsb\_release -cs) main" | sudo tee /etc/apt/sources.list.d/hashicorp.list

sudo apt update && sudo apt install terraform

**If you download Terraform for the Windows operating system:**

1. Find the install package, which is bundled as a zip file.
2. Copy files from the zip to a local directory such as **d:\terraform** and extract the same.
3. Windows 🡪 Search 🡪 Edit the System Environment Variables 🡪 Environment Variables
4. Select Variable = Path 🡪 Edit 🡪 Add **c:\terraform** 🡪 OK

A screenshot of a computer

Description automatically generated

Once installed successfully, test if the installation works by checking the version below.

terraform -v

**Terraform Commands**

### **Format**

The **terraform fmt** command helps us format the Terraform code for all .tf files in the given directory.

### **Initialize**

The **terraform init** command initializes your Terraform environment. This command downloads the plug-ins that you need. It also verifies that Terraform can access your plan's state file. If the version of the provider is changed, it will report an error. To resolve execute the command terraform init -upgrade

1. **Validate**

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 state, provider APIs, etc.

### **Plan**

The **terraform** **plan** -out=tfplan command produces an execution plan that's based on your configuration. This command doesn't modify any infrastructure. It's just a way for a human to review what changes will be made if the plan is applied.

### **Apply**

The **terraform** **apply** -auto-approve tfplan command runs your execution plan. Think of it as a way to apply the proposed changes that you get from the terraform plan command.

The terraform apply command is an idempotent operation.

### **Destroy**

The **terraform** destroy command destroys all infrastructure resources that are defined in your plan.

1. **Verify the results:**

**terraform output** if for obtaining either the full set of root module output values or a specific named output value from the latest state snapshot.

Azure Infrastructure

**Create a Service Principal In Azure AD:**

1. Azure Portal 🡪 Azure Active Directory 🡪 App registration 🡪 New registration 🡪 Name=TerraformServicePrincipal 🡪 Click on Register
2. After your application is created, you'll be taken to the "Overview" page. Note down the "**Application (client) ID**" as you'll need it in the next steps.
3. Click on the "Certificates & secrets" link from the left-hand menu.
4. Click on "**New client secret**" button.
5. In the "**Add a client secret**".
6. **To grant Service Principal the Contributor role over the Subscription**

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**Windows with PowerShell**

In Powershell terminal, set the following environment variables. Update the variable values with the values Azure returned in the previous command.

$Env:ARM\_CLIENT\_ID="625a9b41-3942-4da7-81c7-8bc04359c292"

$Env:ARM\_CLIENT\_SECRET="T.I8Q~mv7mXtdqtvmsvoiCJ4yWJBVyMMF8oO-ckz"

$Env:ARM\_TENANT\_ID="9a2b4fd4-c9d2-4e05-82d5-63405d8e2a1f"

$Env:ARM\_SUBSCRIPTION\_ID="8bc36a19-2229-4109-a3ec-8faabec66bc4"

**macOS/Linux Terminal**

export ARM\_CLIENT\_ID="625a9b41-3942-4da7-81c7-8bc04359c292"

export ARM\_CLIENT\_SECRET="T.I8Q~mv7mXtdqtvmsvoiCJ4yWJBVyMMF8oO-ckz"

export ARM\_TENANT\_ID="9a2b4fd4-c9d2-4e05-82d5-63405d8e2a1f"

export ARM\_SUBSCRIPTION\_ID="8bc36a19-2229-4109-a3ec-8faabec66bc4" # **"<SUBSCRIPTION\_ID>"**

**Azure Provider Configuration:**

<https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/guides/service_principal_client_secret>

**How are Terraform files structured?**

* ***main.tf:*** This file is often called a **Terraform plan***.* This file holds your Terraform **configuration code**. Your Terraform plan specifies the infrastructure resources that you need.
* ***terraform.tfvars:*** A .tfvars file is a good way to maintain larger sets of variables.
* ***terraform.tfstate:*** It's called the state file. It's a JSON file that Terraform manages. It helps map the resources that you define in your plan to the running resources that your plan produces.

**Walkthrough**

1. Create a folder and save the below file in that folder.
   1. mkdir terraformdemos
   2. cd terraformdemos
   3. code .
2. Create c:\terraformdemos\main.tf and edit content as below:
3. Also, replace the SubscriptionId, client\_id, client\_secret and tenant\_id with values we copied earlier.

**main.tf**

terraform {

   required\_providers {

**azurerm** = {

      source = "hashicorp/azurerm"

      version = "3.11.0"

    }

  }

}

provider "azurerm" {

    subscription\_id = "51081bf2-da0d-4998-9462-b59b512f8690"

    client\_id = "73eabec2-9978-4e0d-a27f-56d5c2beb191"

    client\_secret = "5Ys8Q~qyqEn06NaHCZLLoUo3h.BSSgOsPRKGaIp"

    tenant\_id = "82d8af3b-d3f9-465-c724-0fb186cc28c7"

    features {}

}

#Creating a Resource Group

resource "azurerm\_resource\_group" "my" {

  name     = "Sandeep-Terraform-rg"

  location = "east us"

}

**Execute the following commands from the same folder:**

C:\terraformdemos>**terraform init**

C:\terraformdemos>**terraform plan -out=tfplan**

C:\terraformdemos>**terraform apply tfplan**

**Observation**: Go to Azure Portal and note that Resource Group, **Sandeep-Terraform-rg** is created

C:\terraformdemos>**terraform destroy** -auto-apply

**Observation**: Go to Azure Portal and note that Resource Group, **Sandeep-Terraform-rg** is deleted

**How Terraform Applies a Configuration**

When Terraform creates a new infrastructure object represented by a resource block, the identifier for that real object is saved in Terraform's state, allowing it to be updated and destroyed in response to future changes. For resource blocks that already have an associated infrastructure object in the state, Terraform compares the actual configuration of the object with the arguments given in the configuration and, if necessary, updates the object to match the configuration.

**In summary, applying a Terraform configuration will:**

1. **Create** resources that exist in the configuration but are not associated with a real infrastructure object in the state.
2. **Destroy** resources that exist in the state but no longer exist in the configuration.
3. **Update** in-place resources whose arguments have changed.
4. **Destroy** **and re-create** resources whose arguments have changed but which cannot be updated in-place due to remote API limitations.