Terraform:

<https://learn.hashicorp.com/tutorials/terraform/infrastructure-as-code?in=terraform/azure-get-started>

<https://www.baeldung.com/ops/terraform-intro>

Terraform is an open-source infrastructure as code software tool that provides a consistent CLI workflow to manage hundreds of cloud services. Terraform codifies cloud APIs into declarative configuration files.

Terraform is the infrastructure as code tool from HashiCorp.  It is a tool for building, changing, and managing infrastructure in a safe, repeatable way.

Operators and Infrastructure teams can use Terraform to manage environments with a configuration language called the HashiCorp Configuration Language (HCL) for human-readable, automated deployments.

[**»**](https://learn.hashicorp.com/tutorials/terraform/infrastructure-as-code?in=terraform/azure-get-started#infrastructure-as-code)**Infrastructure as Code**

If you are new to infrastructure as code as a concept, it is the process of managing infrastructure in a file or files rather than manually configuring resources in a user interface. A resource in this instance is any piece of infrastructure in a given environment, such as a virtual machine, security group, network interface, etc

## Workflows

1. Scope - Confirm what resources need to be created for a given project.
2. Author - Create the configuration file in HCL based on the scoped parameters
3. Initialize - Run terraform init in the project directory with the configuration files. This will download the correct provider plug-ins for the project.
4. Plan & Apply - Run terraform plan to verify creation process and then terraform apply to create real resources as well as state file that compares future changes in your configuration files to what actually exists in your deployment environment.

**BASICS:**

Terraform is a tool for building and changing infrastructure safety [sic] and efficiently.

Hello, Terraform:

Terraform version check:

terraform –v

**Creating Our First Project:**

mkdir hello-terraform

cd hello-terraform

create main.tf file

provider "local" { version = "~> 1.4" }

resource "local\_file" "hello" {

content = "Hello, Terraform" filename = "hello.txt"

}

**init, plan, and apply:**

1. terraform init
2. terraform plan
3. terraform apply - will ask confirmation for creating the resources
4. terraform apply -auto-approve – will not ask confirmation
5. See the resources which we created - cat hello.txt
6. if you want to see the full list of all our resources and their

attributes you can run the terraform show

1. terraform output –json
2. terraform destroy –auto-approve

**Core Concepts:**

[**https://www.baeldung.com/ops/terraform-intro**](https://www.baeldung.com/ops/terraform-intro)

[**https://k21academy.com/terraform-iac/terraform-beginners-guide/#Read\_our\_blog\_to\_know\_why\_Terraform\_is\_preferred\_over\_other\_IaC\_tools\_Why\_Terraform\_Not\_Chef\_Ansible\_Puppet\_CloudFormation**](https://k21academy.com/terraform-iac/terraform-beginners-guide/#Read_our_blog_to_know_why_Terraform_is_preferred_over_other_IaC_tools_Why_Terraform_Not_Chef_Ansible_Puppet_CloudFormation)

[**https://learn.hashicorp.com/tutorials/terraform/aws-build**](https://learn.hashicorp.com/tutorials/terraform/aws-build)

1. **Providers:**

A provider works pretty much as an operating system's device driver. It exposes a set of resource types using a common abstraction, thus masking the details of how to create, modify, and destroy a resource pretty much transparent to users.

1. **Resources:**

In Terraform, a resource is anything that can be a target for CRUD operations in the context of a given provider. Some examples are an EC2 instance, an Azure MariaDB, or a DNS entry.

resource "aws\_instance" "web" {

ami = "some-ami-id"

instance\_type = "t2.micro"

}

Note: **A key point about resources is that once created, we can use expressions to query their attributes.** Also, and equally important, **we can use those attributes as arguments for other resources**.

**count and for\_each Meta Arguments:**

The count and for\_each meta arguments allow us to create multiple instances of any resource. The main difference between them is that count expects a non-negative number, whereas for\_each accepts a list or map of values.

**Ex:**

**resource "aws\_instance" "server" {**

**count = var.server\_count**

**ami = "ami-xxxxxxx"**

**instance\_type = "t2.micro"**

**tags = {**

**Name = "WebServer - ${count.index}"**

**}**

**}**

**Data Sources:**

Data sources work pretty much as “read-only” resources, in the sense that we can get information about existing ones but can't create or change them. They are usually used to fetch parameters needed to create other resources.

**State:**

The state of a Terraform project is a file that stores all details about resources that were created in the context of a given project.

For instance, if we declare an azure\_resourcegroup resource in our project and run Terraform, the state file will store its identifier.

The primary purpose of the state file is to provide information about already existing resources, so when we modify our resource definitions, Terraform can figure out what it needs to do.

**An important point about state files is that they may contain sensitive information**. Examples include initial passwords used to create a database, private keys, and so on.

EX:

terraform {

backend "s3" {

bucket = "some-bucket" key = "some-storage-key" region = "us-east-1" }

}

**Modules:**

Terraform modules are the main feature that allows us to reuse resource definitions across multiple projects or simply have a better organization in a single project. This is much like what we do in standard programming: instead of a single file containing all code, we organize our code across multiple files and packages.

A module is just a directory containing one or more resource definition files. In fact, even when we put all our code in a single file/directory, we're still using modules — in this case, just one. The important point is that sub-directories are not included as part of a module. Instead, the parent module must explicitly include them using the *module* declaration:

**Input Variables:**

A variable has a *type****,***which can be a *string*, *map*, or *set*, among others. It also *may* have a default value and description. For variables defined at the top-level module, Terraform will assign actual values to a variable using several sources:

* *-var* command-line option
* *.tfvar* files, using command-line options or scanning for well-known files/locations
* Environment variables starting with *TF\_VAR\_*
* The variable's *default*value, if present

**Output Values:**

By design, a module's consumer has no access to any resources created within the module. Sometimes, however, we need some of those attributes to use as input for another module or resource. To address those cases, a module can define output blocks that expose a subset of the created resources:

output "web\_addr" {

value = aws\_instance.web.private\_ip description = "Web server's private IP address"

}

**Local Variables:**

Local variables work like standard variables, but their scope is limited to the module where they're declared. The use of local variables tends to reduce code repetition, especially when dealing with output values from modules:

**Workspaces:**

Terraform workspaces allow us to keep multiple state files for the same project. When we run Terraform for the first time in a project, the generated state file will go into the default workspace. Later, we can create a new workspace with the terraform workspace new command, optionally supplying an existing state file as a parameter.

Variables:

Variable types

1. String

Ex:

variable "access\_key" {

description = "The AWS access key."

}

variable "region" {

description = "The AWS region."

default = "us-east-1"

}

1. Map:

variable "ami" {

type = "map"

default = {

us-east-1 = "ami-0d729a60"

us-west-1 = "ami-7c4b331c"

}

description = "The AMIs to use."

}

1. List

variable "security\_group\_ids" {

type = "list"

description = "List of security group IDs."

default = ["sg-4f713c35", "sg-4f713c35", "sg-4f713c35"]

}

Lists are zero-indexed. We can retrieve a single element of a list using the syntax:

${var.variable[element]}

**Populating variables:**

terraform plan -var 'access\_key = abc123' -var 'secret\_key = abc123'

COSMOSDB:

<https://github.com/terraform-providers/terraform-provider-azurerm/tree/master/examples/cosmos-db/basic>

<https://dev.to/willvelida/provisioning-azure-cosmos-db-resources-with-terraform-4i64>

Books:

<https://www.dropbox.com/sh/pfj9e1m5d7nqhp3/AAAqW1OI3oYPa4UXfYb4fLGaa?preview=aws.pdf>

**Azure COSMOS Database:**

Azure Cosmos DB is a NoSQL data store. It is different from the traditional relational database where we have a table, and the table will have a fixed number of columns, and each row in the table should adhere to the scheme of the table. In the NoSQL database, you don't define any schema at all for the table, and each item or row within the table can have different values, or different schema itself.

* **No Schema & Index management:** The Azure database engine is fully schema-agnostic. Therefore no schema and index management are required. We also don't have to worry about application downtime while migrating schemas.
* **Industry-leading comprehensive SLAs:** Cosmos DB is the first and only service to offer industry-leading full 99.99% high availability, read and write latency at the 99th percentile, guaranteed throughput, and consistency
* **The low total cost of Ownership:** Since Cosmos DB is a fully managed service, we no longer need to manage and operate complex multi-datacenter deployment, and upgrades of our database software pay for the support, licensing, or operations.
* **Developing application using NoSQL APIs:** Cosmos DB implements Cassandra, MongoDB, Gremlin, and Azure Table Storage wire protocol directly on the service.
* **Global distribution:** Cosmos DB allows us to add or remove any of the Azure regions to our Cosmos account at any time, with a click of a button.