# Materials to learn from

* High level overview - [www.youtube.com](https://www.youtube.com/watch?v=l5k1ai_GBDE)
* Install and start using Terraform - [developer.hashicorp.com](https://developer.hashicorp.com/terraform/tutorials/azure-get-started/infrastructure-as-code)
* Detailed tutorial - [www.youtube.com](https://www.youtube.com/watch?v=V53AHWun17s)

# Terraform setup

* Terraform installation and basic tutorial: [developer.hashicorp.com](https://developer.hashicorp.com/terraform/tutorials/azure-get-started/infrastructure-as-code)
* When Terraform is creating Azure resources it is authenticating using a Service Principal. In order to allow Terraform to create other Service Principals, we need to create a Service Principal with proper permissions which will be used by Terraform for authentication.

In the ‘Authenticate using the Azure CLI > Create a Service Principal’ section in the instruction on developer.hashicorp.com we are creating a service principal with the ‘Contributor’ Azure role and we need to change it into ‘Owner’.

Also it is useful to add some name to the created service principal, for example ‘Terraform’. We can do this by using the ‘az ad sp create-for-rbac’ command with the ‘--name’ parameter.

Additionaly we need to assign the ‘Application Administrator’ Entra role to that service principal. It is described here how to do this: [docs.azure.cn](https://docs.azure.cn/en-us/entra/identity/role-based-access-control/manage-roles-portal?tabs=admin-center)

# Commands

* Init Terraform in a folder (we need to do this before we run a Terraform file located in that folder):
  + cd directory\_name
  + terraform init
* Create and destroy resources defined in the Terraform file:
  + terraform apply # create or modify resources
  + terraform destroy # destroy resources
* Create and destroy resources defined in the Terraform file with execution plan:
  + terraform plan -out main.tfplan # create execution plan
  + terraform apply # create or modify resources
  + terraform plan -destroy -out main.destroy.tfplan # create execution plan
  + terraform apply main.destroy.tfplan
* Get output value:
  + Terraform output -raw output\_name
  + Terraform output
  + Terraform output -json

# Creating resources

We need to prepare a set of .tf files in a repository. In those files we specify what resources we want to create. We can create multiple files creating different resources.

Then we need to create a plan about what resources will be built. For that we need to run:

* terraform plan -out main.tfplan

That will create a plan and save it in the main.tfplan file.

After that we run this command:

* terraform apply

In order to execute the plan and create all the resources specified in all the .tf files in our repository.

# Creating outputs

In terraform we can create outputs. In order to do that we need to write such a command in out .tf file:

A black screen with white text

AI-generated content may be incorrect.

Further sections of this document explains how we can use outputs.

# Using outputs in modules

In Terraform I can create modules. Every module is a directory containing multiple .tf files. For example we might have a structure like this:

terraform-modules/

├── main.tf # Parent module

├── variables.tf

├── outputs.tf

└── modules/

└── s3\_bucket/ # Child module

├── main.tf

├── variables.tf

└── outputs.tf

So here we are creating a child module called ‘s3\_bucket’.

This module can create output values. They are defined in the modules/s3\_bucket/outputs.tf file. This file can look for example like that:

modules/s3\_bucket/outputs.tf :

A black screen with white text

AI-generated content may be incorrect.

In order to access this module’s output in the parent module we need to define that module at first in the main.tf file in the parent module:

Main.tf file:

A screen shot of a computer code

AI-generated content may be incorrect.

So here we are creating a module called ‘my\_s3\_bucket’.

Now in the parent module we can use output from the child module by using command:

* module.my\_s3\_bucket.bucket\_arn.

We can use that output from the child module for example in output of the parent module. In that case we can create the output of the parent module in the file called output.tf:

Output.tf file:

A screenshot of a computer

AI-generated content may be incorrect.

# Configuration management

We can use Ansible for configuration management. For example it can configure VMs created by Terraform. It can interact with an operating system of that VM.

# State file

The state file describes what infrastructure we have created. Based on that file and our code Terraform will determine which resources need to be created or deleted.

# Terraform import

Get info about the existing infrastructure. It updated the state file but not configuration files (files with our code). We can use commands, like for example ‘terraform show’, to see the configuration for our current state file and then we can add it to our code files.

# Data block

It gets information about currently existing resources in cloud even if they have not been created by Terraform.

# Modules

Modules allow us to reuse code in multiple projects. There are different kinds of modules which differ in from where we are taking a code:

* Local modules
  + Source: A directory on a local filesystem
* Git based modules
  + Source: A git repository
* Terraform registry modules
  + Source: Terraform registry, public or private
* Private registry modules
  + Source: A private Terraform module registry (e.g., inside Terraform Cloud or Terraform Enterprise).

Each module has defined its own variables and outputs.

When creating a module, we need to provide all the necessary variables.

After creating a module, we can access its outputs.

For example, we can have local modules like that:

A screenshot of a computer

AI-generated content may be incorrect.

Here we have a separate module for creating different Azure resources, linux VM, networks, resource group, storage account and generating SSH keys.

Those modules can be then used in scripts in a repo like this:

A screenshot of a computer program

AI-generated content may be incorrect.

## Local modules

Local modules are saved in a directories on our local filesystem.

When creating a local module, as a source argument we are providing a path to a local directory containing this module.

For example if we have a repo like this:

A screenshot of a computer program

AI-generated content may be incorrect.

Then we specify a source for a module in the main.tf file like this:

A computer screen with white text

AI-generated content may be incorrect.

## Git based modules

We can have a Terraform module saved in a Git repository and access it directly in another repository:

A black background with white text

AI-generated content may be incorrect.

## Terraform registry modules

There are available modules on the Terraform Registry website [registry.terraform.io](https://registry.terraform.io/).

We create a module from a Terraform Registry like this:

A black screen with white text

AI-generated content may be incorrect.

## Private registry modules

We can create a private Terraform module registry for example inside Terraform Cloud or Terraform Enterprise.

We create a module from a private registry like this:

A black screen with white text

AI-generated content may be incorrect.

## Providers in modules

Sometimes in the module we need to specify providers even though we have those providers defined in the root module.

# Terraform console

We can use the ‘terraform console’ command which will create a console where we can test terraform commands, like different terraform functions.

# Drift

Drift is a situation when resources in cloud are different then those defined in Terraform configuration files. That when some resources have been created manually or by a different tool than Terraform. There are some tools in Terraform helping with that.

# Testing

There are tools helping with testing if our code is correct. Not only if that code will work without errors but if it is gonna work as excpected and we are not violating any common rules.

## TFLint

The TFLint is a tool for testing if Terraform code is correct. It knows what are the available values for different arguments and can detect errors, for example if we use the ‘banana’ as argument for VM size it will know it is not acceptable value.

## Checkov

The checkov is a tool for checking if we are exposing confidential data in our code or if we are following the recommended best practices for our code, like do we have all the recommended settings.

It also allows us to define our own rules (policies) which should be followed when writing a code and it can automatically validate them.

Another similar tools:

* Terrascan
* Trivy
* Sentinel
* Snyk
* Mondoo
* Terratest

## Terraform tools

### Check block

Check blocks are used to check if resources have been created correctly. They use the data block to get information about the created resources.

### Lifecycle block

Lifecycle blocks can be used to check if all requirements are met before creating resources, and like check blocks they can check if resources have been created properly.

### Assert block

Assert block can be used for testing our code.

### Mocking

There is the mocking option in Terraform for testing our code.

### Variables validation

We can use the validation argument in a variable in order to validate if the entered values for that variable is correct.

### Terraform test cli

There is the Terraform test cli tool which can be used for testing as well.

# Terraform docs

It is a tool which we can use for documenting our Terraform code.