# 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)
* In order to allow terraform to create service principals we need to create a service principal with the ‘owner’ role which will be used by the Terraform.

Creating such a service principal and assigning it to Terraform to use it is described here: [developer.hashicorp.com](https://developer.hashicorp.com/terraform/tutorials/azure-get-started/azure-build) in the ‘Authenticate using the Azure CLI’ section.

But in that section we are creating a service principal with the ‘Contributor’ 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 Microsoft Entra ‘Application Administrator’ 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

# Creating outputs

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

output "output\_name" {

value = ‘output\_value’

}

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 :

output "bucket\_arn" {

value = aws\_s3\_bucket.this.arn

}

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:

module "my\_s3\_bucket" {

source = "./modules/s3\_bucket"

bucket\_name = "my-unique-bucket-name-1234"

environment = "dev"

}

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:

output "s3\_bucket\_arn" {

value = module.my\_s3\_bucket.bucket\_arn

}

# Accessing outputs in terminal

I can access terraform outputs in terminal. At first we create output in the output.tf file:

output "output\_name" {

value = ‘output\_value’

}

And we can access it in terminal by using command:

Terraform output -raw output\_name

# Saving outputs in a file on local computer

In order to save an output in a file on local computer where we are runnin terraform commands we can use the following command:

#save all the outputs in the text file.

terraform output > outputs.txt

Or to save in JSON:

terraform output -json > outputs.json

Save only a specific output value:

terraform output -raw my\_output\_name > my\_output.txt

# Execute bash script on the VM created using Terraform

When we are creating a VM using Terraform we can also execute a bash script on that VM right after creating it also using Terraform.

We use for that the azurerm\_virtual\_machine\_extension resource. It uses a virtual machines extension to run a bash script. More info about how to run a bash script using the virtual machine extension here: [learn.microsoft.com](https://learn.microsoft.com/en-us/azure/virtual-machines/extensions/custom-script-linux).

We can use the templatefile function in order to insert variables into a bash script and then run that script on a VM using the azurerm\_virtual\_machine\_extension resource:

A screenshot of a computer program

AI-generated content may be incorrect.

## Troubleshooting

Logs from executing a bash script on VM can be found on that VM in the /var/lib/waagent/custom-script/download/0 folder. There are the 'stdout' and 'stderr' files with logs. More info about that here: [learn.microsoft.com](https://learn.microsoft.com/en-us/azure/virtual-machines/extensions/troubleshoot).

# 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

## Public modules on Terraform website

There are available modules on the terraform website. They are prepared for creating specific resources. They can be used as an example how to create resources.

## 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.