**Cloud Guru - Managing MS Azure app and infra with Terraform**

Instructor: Russ Kraff

Azure sandbox: (open in incognito window)

Username: [cloud\_user\_p\_738639a4@azurelabs.linuxacademy.com](mailto:cloud_user_p_738639a4@azurelabs.linuxacademy.com)

Password: $khq1N8NUoyP8aCA8eQk

url: <https://portal.azure.com/#@azurelabs.linuxacademy.com/resource/subscriptions/964df7ca-3ba4-48b6-a695-1ed9db5723f8/resourceGroups/1-2a7cd257-playground-sandbox/overview>

**### 2.1- Basic Terraform installation**

Terraform is **idempotent,** support a large number of cloud vendors.

Each vendor may have their specific **resource name** and **config parameters.**

Terraform download: <https://www.terraform.io/downloads.html>

Terraform is supported natively within **Azure CLI.**

Open **azure Shell** (after login to portal.azure.com using user acct), and use **azure CLI**

**### 2.2 - Terraform State Storage: Local vs. Remote**

**Terraform state files** - with all the parameters and things that you deploy in your env,

stored either locally or remote (code version control system, like **github**)

**### 2.3 - Terraform Authorization Methods in Azure**

**Azure authorization** – via **Azure CLI** (user acct login), or via a **Service principal** and a client certificate

**main.tf** - initial config data,

technically you can be put all the resources in this file, for organization purpose and other reasons, recommended use the file to **handle authorization**.

Using **user acct for authenication**

provider

subscription\_id (if not specify, will use the default sub id, but if you want to limit the subid that terraform has accessed to, you can list them here)

for the lab, we use user acct login as authorization

**best practice:** use **service principal** or **managed service identity** authorization(acct unrelated to your user login id)

supports **remote state storage**, and backend features, like **state locking** (prevent multiple update occurs simultaneously, and allow terraform to do large deployment independent of the workstation you used to initiate the deployment)

Using **managed service identity for authenication**

Provider

**use\_msi = true**

subscription\_id

tenant\_id

Using **service prinicipal for authentication** (allow key exchange, hence can use **key vault**)

provider

subscription\_id

**client\_id**

**client\_certificate\_path** (defined with var, previously exported)

**client\_certificate\_password** (defined with var, previously exported)

tenant\_id

\* care must be taken to whether file with **sensitive info** will be check into VCS, like **github**

**### 2.4 - Shared State Files: Security and Encryption**

Using **azure cli for authentication**, and doing so, technically qualify as local state storage. But, if we config terraform backend in a config file, we can push terraform state files into a **storage acct** and container of our choosing.

Doing so give us access to the great feature of **state file security** - Blob storage is **encrypted** at rest, **snapshot** of state file (extra backup), apply delete lock on state file (prevent accidental delete), key value or password or privilege info in state file, using **role access** you can restrict the access to storage acct (hence limit access to the state file), running terraform in VMs, you can use **network restriction** to allow only vnet/subnet that contains the VMs to have access to the storage acct.

Add to **main.tf**:

terraform

backend “azurerm”

**resource\_group\_name = “TFResourceGroup”**

**storage\_account\_name = “storage4terraform”**

**container\_name = “statefile”**

**key = “terraform.tfstate”** (name of the state file .tfstate)

**terraform init** - read the changes in main.tf file, and prompt us to copy the terraform state to the storage (storage acct) we determined

in the portal, home -> all resources -> TFResourceGroup -> storage4terraform -> container (mid screen) -> statefile -> terraform.tfstate (in now **azure storage acct**)

The rest of the terraform files should check into VCS, like **github**

**### 3.1 - Deploy Resource Groups**

**1rgcreate.tf**

**terraform plan** – you always want to run a terraform plan before you apply, it will show what changes terraform going to make

**terraform apply** - prompt to proceed

**### 3.2 – Tagging**

Update **1rgcreate.tf** with tag

tags = {

environment = "Terraform"

deployedby = "Admin"

}

**terraform plan**

**terraform apply**

**### 4.1 - Deploy Azure Storage Accounts**

Typical terraform file (the lab will use this type) – **storage\_acct\_deploy.tf**

More generic terraform file, with var statement, which will prompt for value during terraform run – storage\_acct\_deploy\_generic.tf

variable “region” {}

variable “ResourceGroup” {}

**variable** “Storage\_Account\_Name” {}

resource “azurerm\_storage\_account” “sa” {

name = **var.** Storage\_Account\_Name

**terraform plan**

**terraform apply**

**### 4.2 - Deploying Recovery Service Vaults**

recy\_svc\_vault\_deploy.tf

**### 4.3 - Deploy Azure File Share and Blob Storage**

For file share, define file share size

For blob storage, you need to define a container and name the blob storage

Resource “azurerm\_storage\_account” “lab” {

name = “unique storage account name”

- “azurerm\_storage\_account” is azure object name that you are creating, “lab” is terraform name, “name” is azure name

- later, you can reference it via, **azurerm\_storage\_account.lab.name**