Corndel DevOps Engineering Programme

in association with Softwire

Module 12: Project Exercise

Corndel Digital.

DB5I

Module 12

Project Exercise Brief

In this exercise we will look into using Infrastructure-as-Code (IaC) to declaratively describe our desired Azure infrastructure, and use that to deploy our todo-app with the same arrangement of Azure resources as in Module 11.

There are various different IaC tools and languages, for this we'll be using Terraform, which is a platform agnostic industry standard, so applicable to a wider variety of scenarios than a cloud-specific alternative. Also it is capable of not only configuring, but also provisioning services, and the declarative over procedural style lends itself well to immutable infrastructure.

Part 1 - Terraform running locally

Install

You also need to be logged into the Azure CLI locally, so if you didn't do that in the previous exercise then install & log into that now.

Make sure your Azure CLI has the correct subscription set as its default:

- Run `az login` then `az account list` to see the available subscriptions.
- If that command listed more than one Subscription, tell the azure cli which to use by running `az account set --subscription="SUBSCRIPTION_ID"`

Step 1 - Link Terraform to your project exercise resource group

- Make a new file called `main.tf` in the root of your Git repo.
- You can store your terraform configuration files in sub-directories if you wish, they will form a module per directory.
- You would need to `cd` into the applicable sub-folder to run terraform commands, so it is simplest for now to just keep them in the project root.
- Add the azurerm provider to allow linking with Azure, along with a basic resource group:

```
terraform {
  required_providers {
    azurerm = {
      source = "hashicorp/azurerm"
      version = ">= 2.49"
    }
  }
}

provider "azurerm" {
  features {}
}

data "azurerm_resource_group" "main" {
  name     = "<RESOURCE_GROUP_NAME>"
}
```

By using the `data` command here instead of `resource`, we tell terraform that the resource group is a pre-existing component that we can get read-only access to.

- First run `terraform init` to initialise the directory, set up the backend and link the `azurerm`. > You should add any files and directories generated to `.gitignore`, see recommended patterns here. > It is also worth adding them to your `.dockerignore` file so they don't get used in your docker `COPY` step.
- Running `terraform plan` will show what actions would be performed.

- Once we start adding resources you can run `terraform apply`, which will make a new plan, ask for approval, then provision the resources.
  - You can then go to the Azure portal and confirm whether any new resources have appeared on the dashboard.
  - Running `terraform destroy` will remove any resources you've created.
  - You can save the output of the `plan` command and pass that file into the `apply` command to ensure it does the same thing and skip approval.

Step 2 - Check state

- When you `apply` your configuration, Terraform stores the IDs and properties of the resources created so that it can manage or destroy those resources going forward.
- The default is a local file called `terraform.tfstate`, and as it could also contain sensitive values in plaintext, make sure to `gitignore` it, along with the `.terraform/` directory.
- We will look at alternatives to this local state in Step 3.
- You can run `terraform show` and `terraform state list` to view the current state and confirm it is as expected.

Step 3 - Add a App Service and Web App

Describing resources in Terraform requires similar parameters to the CLI commands we used in the last Exercise, however you can refer to resources provisioned higher in the file when describing later resources, allowing the easy re-use of attributes such as name, location and id.

Add the below declarations, to add an [app service] resource to your terraform configuration.

```
resource "azurerm_app_service_plan" "main" {
  name                = "terraformed-asp"
  location            = data.azurerm_resource_group.main.location
  resource_group_name = data.azurerm_resource_group.main.name
  kind                = "Linux"
  reserved            = true

  sku {
    tier = "Basic"
    size = "B1"
  }
}

resource "azurerm_app_service" "main" {
  name                = "<APP_NAME>"
  location            = data.azurerm_resource_group.main.location
  resource_group_name = data.azurerm_resource_group.main.name
  app_service_plan_id = azurerm_app_service_plan.main.id

  site_config {
    app_command_line = ""
    linux_fx_version = "DOCKER|appsvcsample/python-
helloworld:latest"
  }

  app_settings = {
    "DOCKER_REGISTRY_SERVER_URL" = "https://index.docker.io"
  }
}
```

You will need to update `<APP_NAME>` to a globally-unique name as this forms part of the app's URL, as in the previous exercise.

Make sure you use different names for the resources in the exercise, so you don't accidentally destroy or mis-configure any resources you used in your solution to the previous exercise.

- Run `terraform apply`
- Browse to `https://<app-name>.azurewebsites.net/` and check to see the HelloWorld page is visible.

Step 4 - Add the CosmosDB resource

Now go through and add into `main.tf` all the other resources we provisioned manually last Exercise.

1. Look at the `cosmosdb_mongo_database`docs, you will need both an account and database, as listed in the Example Usage section.

   To save costs, you should enable the Serverless capability:
   `capabilities { name = "EnableServerless" }`

2. You will need to use various output attributes from your `azurerm_cosmosdb_account` resource to construct a connection string to pass into your `azurerm_app_service`'s `app_settings`, which will be given to the app as an environment variable at runtime:
3. `"MONGODB_CONNECTION_STRING" = "mongodb://${azurerm_cosmosdb_account.main.name}:${azurerm_cosmosdb_account.main.primary_key}@${azurerm_cosmosdb_account.main.name}.mongo.cosmos.azure.com:10255/DefaultDatabase?ssl=true&replicaSet=globaldb&retrywrites=false&maxIdleTimeMS=120000"`

   Also remember to update the `azurerm_app_service`'s `site_config` to point at your docker image.

Step 5 - Check local Terraform is working

- Run `terraform apply` and browse to your app to check the functionality.

Step 6 - Prevent database destruction

- Run `terraform apply` a few more times, if you check the output you will notice that using the configuration suggested by the documentation will cause the CosmosDB database to be destroyed and recreated each time.
- In the outputted plan, it states that a "EnableMongo" capability change is what "forces replacement", so add `capabilities { name = "EnableMongo" }` to prevent this.
- Add the `lifecycle { prevent_destroy = true }` meta-argument to the CosmosDB configuration to make Terraform error when it detects a plan that would cause the database to be destroyed.

Part 2 - Parametrise to enable different environments

We now have a working Terraform configuration, however much of our configuration is hard-coded, and we don't have support for different environments.

To solve both of these issues we are now going to add Variables to our Terraform setup.

Step 1 - Input Variables

- Create a new file called `variables.tf`, with the following content:

```
variable "prefix" {
  description = "The prefix used for all resources in this
environment"
}

variable "location" {
  description = "The Azure location where all resources in this
deployment should be created"
  default     = "uksouth"
}
```

1. Also add terraform variables for each of your environment variables that vary between environments, such as your `GITHUB_CLIENT_ID`.
2. Now you can rename each resource to make use of this prefix, eg:

```
resource "azurerm_app_service_plan" "main" {
 name     = "${var.prefix}-terraformed-asp"
 location = var.location
...
```

1. Now when you run `terraform apply` it will prompt you for values of any variables without a default value.
2. To persist variables, you can either add them as defaults, or list them in another file named `terraform.tfvars`.
3. To pass in variables on the command line, use the `-var` flag: `terraform apply -var 'prefix=test' -var 'github_client_id='xxxxxx'`

Step 2 - Output variables

Terraform also allows you to define Output Variables, to expose information on resources that have been created.

Add a new file called `outputs.tf`, with the below contents:

```
output "webapp_url" {
  value = "https://$
{azurerm_app_service.main.default_site_hostname}"
}
```

Now when you run `terraform apply`, it will output a link to the app.

We are also going to use this functionality to export the CD webhook URL that we used in the last exercise, so Travis can call it during deployment.

Make another output variable that uses the attributes available in the app service to form the same webhook url you obtained manually in the previous exercise:

```
https://${azurerm_app_service.main.site_credential[0].username}:$
{azurerm_app_service.main.site_credential[0].password}@$
{azurerm_app_service.main.name}.scm.azurewebsites.net/docker/hook
```

After another `terraform apply`, you can then run `curl -dH -X POST "$(terraform output -raw cd_webhook)"` to test it works.

Part 3 - Use Azure Blob Storage as Remote State

Terraform stores state about which real-world infrastructure objects correspond to the resources in a configuration, along with associated metadata, in order to update or delete resources in response to changes.

The default `local` backend stores state as a json file on disk, `terraform.tfstate`, and also performs operations locally.

There are a variety of other backends available. Some simply store the state files on a remote storage disk; others support locking the state while operations are being performed, which helps prevent conflicts and inconsistencies; and some also perform the provisioning and updating operations on a remote server.

We'll be using the `azurerm` backend, which stores the state as an encrypted Blob with the given Key within a Blob Container in a Blob Storage Account, and supports state locking and consistency checking via native capabilities of Azure Blob Storage.

Run `terraform destroy` to deprovision your resources, delete the local state files and terraform directories, then follow this tutorial to set up a storage account and blobs and link your terraform configuration to it.

Part 4 - Link in with Travis

We want to now get Travis to run `terraform apply` to update the resources before deploy, and then use the terraform output to call the Azure CD webhook.

Step 1 - Service Principal Authentication

So that Travis can access and alter your azure resources, we are going to set up [Service Principal Authentication].

1. First run `az account list`, and make note of the `id` field - this is your subscription ID.
2. If that command listed more than one Subscription, tell the azure cli which to use by running `az account set --subscription="SUBSCRIPTION_ID"`
3. Now create a new Service Principle by running

```
az ad sp create-for-rbac --name "<SERVICE PRINCIPAL NAME>" --role
Contributor --scopes /subscriptions/<SUBSCRIPTION ID>/
resourceGroups/<RESOURCE GROUP NAME>
```

   where the resource group is your project exercise resource group.
4. In the Azure portal this will create an app registration with you as the owner.
5. Add the given values to your Travis config file as environment variables with these names: `ARM_CLIENT_ID`, `ARM_TENANT_ID`, `ARM_SUBSCRIPTION_ID` and `ARM_CLIENT_SECRET`

Step 2 - Installing the Terraform CLI in Travis

1. Add the environment variable `TF_VERSION=0.14.7` to allow easier upgrading of the terraform version.
2. Add the following block into your Travis config in an `install` step before the `before_deploy` step to install the terraform CLI:

```
install:
- wget https://releases.hashicorp.com/terraform/"$TF_VERSION"/
terraform_"$TF_VERSION"_linux_amd64.zip
- unzip terraform_"$TF_VERSION"_linux_amd64.zip
- sudo mv terraform /usr/local/bin/
- rm terraform_"$TF_VERSION"_linux_amd64.zip
- terraform init
```

Step 3 - Update commands

1. Add your `terraform apply` command with `-var` tags and the `-auto-approve` parameter to the `before_deploy` step in Travis.
2. Update your `deploy` script to POST with curl to the url given by the terraform output variable.

Part 5 (Stretch) - Run e2e tests on Terraformed Infrastructure

- Best practice is to run tests against dedicated production-like test infrastructure.
- Change how Travis runs E2E testing so it:
- Uses Terraform to create a test environment
- Runs the E2E tests against it
- Tears down the test environment
- You may also have to change the python e2e test code to take in a connection string instead of using a `localhost` mongo database.