Azure, Terraform, and GitHub

Ludovic Sauthier

June 2022

Contents

1 Introduction

1.1 What is the project about

I have created this training document to help you understand Azure and Terraform. You will go through the following steps:

In section 2 you will use Python and the framework Flask to create and run a webapplication locally on your machine. Then, you will create Terraform scripts to automate the deployment of your webapplication infrastructure in Azure, and finally you will use GitHub and its actions to deploy your code automatically into your Azure subscription, i.e. using CI/CD concepts.

In section 3 you will add a DB to your application locally but also in Azure using Terraform.

In section 4 you will improve the overall security using Azure Managed Identity and the Azure KeyVault which is also automatically deployed using Terraform.

1.2 Skills you need

You need to have some basic concepts of Python and Azure to understand what is going on. I don't explain all details in this document as I assume you will look for yourself, in my opinion, the best way to learn. I do provide all the source code and explain the key concepts. I also highlight the pitfalls to avoid and problems I had to manage during the creation of this training.

1.3 Setup you need

On your computer, your should create **two folders**, one for the source code I provide you (a.k.a the reference folder in the text) and one for your own

code (a.k.a your own folder in the text). Use my code as reference, copy and paste but don't use the same folder as you might have conflicts.

We use **git**[1] extensively, make sure to install it and understand the basic concepts. We provide the commands which are relevant for this training.

The webapplication runs on Python3.9 [2]. You must install it. All extensions are installed on a later stage, e.g. Flask.

You need an **Azure subscription** [3]. You can choose the free version but you will eventually have to upgrade for a "pay-as-you-go" version. I had some issues and I had to upgrade. Costs should be kept minimal as you will automatically destroy your full infrastructure every-time you are finished, most of the time, nothing will run on your subscription. You should also install Azure CLI [4]. Whenever possible we try to interact with Azure using the CLI and not the platform. Nonetheless, don't be afraid to use it in parallel to understand what is going on and to build additional skills.

You need a **GitHub account** which can be created for free [5]. You will use it to automate the deployment of your code into Azure.

Terraform and Postgres will be installed on a later stage.

If all those requirements are covered, I invite you to get the source code from my repository using the following commands:

```
> move into the reference folder (powershell)
> git clone https://github.com/lsauthie/canopus.git
```

You should see my source code which can be used as reference. There are three different branches corresponding to the three different steps of this project as described above. The branch "webappAzure" corresponds to the first step, the branch "dbAzure" to the second step, and "keyvaultAzure" to the third step. You can move between the branch using the following commands in your reference folder:

[1] https://git-scm.com/book/en/v2/Getting-Started-Installing-Git
[2] https://www.python.org/downloads/
[3] https://azure.microsoft.com/en-us/free/
[4] https://docs.microsoft.com/en-us/cli/azure/install-azure-cli
[5] https://github.com/

```
> move into the reference folder (powershell)
See all branches
> git branch -a
Move to a branch
> git checkout <name of the branch>
```

You can also consult the code using the following URL [6] as the repository is public. Use the ribbon "branches" to see the different branches and consult the code.

At this point, you should be all set to start with the first step of this training.

1.4 Donate if you are happy

If at the end of this training, you have the feeling you could learn something new, don't hesitate to buy me a beer using https://paypal.me/ludodole.

2 Canopus, a simple web application

In this section, we discover the webapplication we are using. First, we make sure the application can run on our local machine, second we create an Azure infrastructure using Terraform which can support our webapplication, and finally we use GitHub and CI/CD paradigms to automate the deployment of our application into Azure.

Make sure you have the first branch (webappAzure) active in your reference folder, see 1.3 for more details.

[6] https://github.com/lsauthie/canopus

2.1 Canopus running locally

I decided to develop a simple webapplication using Flask (which is a framework based on Python). This is in my opinion, the easiest way to create a proof of concept which is what we need for this learning project.

Basically, the application takes a list of strings as input. Those strings are sent over an http:GET request. Then, the application computes the similarity ratio between the strings and sends the result back in a JSON format. To compute the similarity ratio, it uses some natural language processing (NLP) techniques, empowered by the library SCAPY. To work, SCAPY needs a model which is usually downloaded and loaded by SCAPY itself. In our case and because we want to deploy our application onto Azure, we need to change this approach a little bit. We install the model (en_core_web_md-3.2.0) in the root folder and load it as an external file (see the special comment in the code).

Let's focus on the files contain in the reference folder. The following commands will install all required libraries and will start the Flask application.

```
> python -m pip install -r requirements.txt
> flask run
> open your browser and call: "localhost:5000"
```

You should see the landing page "Welcome on project Canopus". If you click on the link, you can see how the application processes inputs, note that you can change the strings in the URL as you like.

With this, we conclude the first part of our journey, which means, you have an application running locally. Feel free to review the code and make sure you understand how it works.

Security note: If you review the code, you will notice that we do check the input for html characters to avoid simple injections. You can try creating a string including some html strings, e.g. "<b>", you will get an error asking you not to use html patterns. Let's move on a create our infrastructure.

2.2 Webapplication and Azure

Now that we have a webapplication, let's see how to create an Azure infrastructure to host our webapplication. We will use Terraform [7] to deal with our infrastructure. Terraform is a solution to deliver infrastructure as code, which means that your infrastructure is defined as code, code which will create all Azure elements you need to run your solution, in our case, our webapplication. You can download Terraform which is a simple executable you can put in whatever folder you want. If you are new to Terraform and Azure, you can start with this small tutorial [8] to understand the basics.

If you go in your reference folder, you can find all files related to Terraform in "terraform" including the executable which you can reuse or download as explained before. Note that Terraform will process all files extended with ".tf" as one source. Currently, we have two ".tf" files: 1. main.tf which contains generic information, and 2. webservice.tf which contains all information required to create our webapplication resource on Azure. At this stage, there are a few pitfalls to watch for:

webservice.tf/appserviceplan Because we are planning to run a Flask application, we need a Linux service plan. Attributes "reserved" and "kind" must be defined.

webservice.tf/webapp If you want your application to start once deployed, you must pass the application setting SCM_DO_BUILD_[...] = "1". Those application settings can be created, modified, or deleted on the portal: Azure portal/webapplication/settings/configuration. Keep this in mind as we will reuse application settings later.

webservice.tf/webapp You must define which Python version you need through the site_config{} block.

As reminder, make sure to copy/paste or create your own files in your own folder to avoid conflicts. Do not use the reference folder. Once everything is set, you can try deploy the infrastructure using the following commands:

[7] https://www.terraform.io/

[8] https://learn.hashicorp.com/collections/terraform/azure-get-started

```
> move into your own folder
> az login
> terraform.exe init
> terraform.exe validate
> terraform.exe plan -out output.tfpl
> terraform.exe apply output.tfpl
```

If everything works as expected, which means without error, you can connect to your subscription on Azure and see how it looks online. If you should have some errors, try to find out the problem. You can always start from scratch by deleting files which are automatically generated in your terraform folder and by deleting your azure resource group using the following Azure CLI commands:

```
> azure group list
> azure group delete - -name canopus-rg
```

If everything works correctly you are ready to deploy your webapplication to your Azure infrastructure. To destroy your infrastructure, you can either proceed as we just explained with Azure CLI or you can use Terraform (preferred method):

```
> terraform.exe destroy
```

To keep costs to a minimum, we recommend your destroy your infrastructure every-time it is not needed, i.e. once you are done for the day.

2.3 Canopus deployment

At this point, we have a webapplication which works fine locally and we have an infrastructure running on Azure. Let's see how to move the webapplication from your local machine to Azure. We will do that using GitHub and its actions [9]. First, you need to create a GitHub account (which is free) and upload you code in it. You can use git to "push" your code on your remote

[9] https://github.com/features/actions

GitHub repository. Assuming git is already configured on your own folder, you can use the following commands:

```
> move into your folder (powershell)
> git status (see if configured)
> git remote add <your reference> <your GitHub>
> git push <your reference> main
```

Once completed, you should be able to see your code in your GitHub account. Now, the idea would be to trigger the deployment of the code once "push" is detected by GitHub, this is achieved using actions [10].

You need to deal with two concepts to automate the deployment of your application.

The pipeline is a YAML file which describes the different steps GitHub needs to complete to deploy your application. I won't explain all the details in this document but you can look at the file ".github/workflows/master_webapp.yml". Once this file has been deployed into your GitHub account, you can see an entry in the "Actions" ribbon (on Github). If you try execute this action, you will get an error because GitHub doesn't know your deployment information and credentials. The last line of the YAML file is referencing a publishing-profile using GitHub secrets. When the file is executed, it automatically retrieves the credentials from the GitHub vault. Let's see how to configure this.

The publishing-profile must be stored as an entry in the GitHub vault. The name of the entry must be the same as the reference in the YAML file, i.e. "AZUREAPPSERVICE_PUBLISHPROFILE". First, create the entry on your GitHub account under "GitHub/Settings/Secrets/Actions" - if you don't see the "settings" ribbon is because you are not logged-in. Then, retrieve the publishing-information from Azure using the following command line (your Azure infrastructure must be existing) and copy it into the entry you just created on GitHub. Note that this must be done every-time you create a new infrastructure on Azure, i.e. you run "terraform apply".

[10] https://docs.github.com/en/actions

```
> az webapp deployment list-publishing-profiles
- -name canopus-webapp -g canopus-rg - -xml
```

Once set, you should be able to trigger the action manually or through a "git push" command. The code is automatically deployed on your Azure infrastructure and the webapplication is started. The action returns the URL you can use to access the webapplication running on your Azure infrastructure. You can use the URL to test your application which should behave like your local instance. The section 5 contains some hints to help you debugging.

Let's move on and add some flesh.

3 Adding a Data Base

In this section we add a database to make things a bit more challenging. We use postgres locally but also on Azure where we use a PaaS solution [11]. Let's start to install and modify the code such that the application can work locally with a postgres instance.

In your reference folder, load the second branch "dbAzure".

3.1 Deploying the DB locally

From internet, install postgres [12]. Once installed, you should be able to administrate your DB with a tool called "pgAdmin". To access the DB, you need a user name, a password, and a server. Two things to keep in mind:

1. Those information will be different locally and on Azure.
2. Those information **must be kept secret**, i.e. should not be embedded in the source code.

[11] https://docs.microsoft.com/en-us/azure/postgresql/flexible-server/
[12] https://www.postgresql.org/

3.2 Using the DB

We use the library sqlAlchemy [13] to deal with the DB in the code.

First, we create a model of our table, this is done in the file "dbcanopus.py".

Second, we need to create our DB on top of the DB server. Use "pgAdmin" and create a DB named "canopus-db" or whatever you want (make sure to use the same name in the code).

Third, we create the code which helps us interact with the DB. This is done in the file "base.py". This is also the place where we deal with the two "things" we asked you to keep in mind. We use environment variables to store the critical information, i.e. **those information are not stored in the source code**. Locally, we use a helper function from python "load_dotenv()" where information are stored in a ".env" file. Note that this ".env" document is available on the reference folder (git) for training purpose, but in the real world, it should not be uploaded as it contains critical information. Such files can be excluded from git using the special ".gitignore" file, simply add an entry with ".env" and it will be ignored.

On Azure, the environment variables can be defined on the webapplication either manually on the platform under "webapplication/configuration/application settings" or using terraform as explained in the next section. Then, to differentiate between an instance running locally from an instance running on Azure, we use a simple "if/else" statement which checks the server-name.

The connection to the DB is established using a string which looks like "postgresql://<dbuser>:<dbpass>@<dbserver>/<dbname>?<arguments>" and this string differs if the code is running locally or on Azure. Note the argument "sslmode=require" for the Azure string which is best practise if you don't want your information, including the credentials, to be transmitted in clear text over the network.

Finally, we made some changes in the file "app.py" to achieve the following:

def dole_flush() delete all information from the DB.

[13] https://www.sqlalchemy.org/

def dole_all() get and return all information from the DB.

def canopus() populate the DB - we modify the function which was already existing.

Note that when the class "Base" is instantiated through "Base.metadata.create_all(engine)", the table is automatically created if not existing, i.e. you don't need to create the table beforehand (but you must create the DB on top of the server).

At this point, the application should work locally and the information should be stored in your local DB. In the next section, we will see how to use terraform to instantiate the DB in Azure.

3.3 Instantiating the DB in Azure

In this section, we see how to use terraform to deploy our DB into Azure (for more details see [14] [15] and [16]. We create a new file "terraform/postgres.tf". Remember that all files with the ".tf" extension are processed by terraform, which means we use this structure for more readability. Each block is explained within the source code but in a nutshell, what we need to do is:

1. Create a network as we need to access the DB server from the webapplication.
2. Create one subnet to host the DB, we create the subnet for the webapplication later.
3. Create a network security group to protect our DB. You need to specifically bind the NSG to the subnet.

[14] https://docs.microsoft.com/en-us/azure/developer/terraform/deploy-postgresql-flexible-server-database?tabs=azure-cli

[15] https://docs.microsoft.com/en-us/azure/postgresql/flexible-server/concepts-networking

[16] https://docs.microsoft.com/en-us/azure/postgresql/flexible-server/tutorial-webapp-server-vnet

4. We need a DNS entry for our postgres server, so we need to configure a DNS private zone and an entry.

5. Create the DB server which is configured with the DNS entry we just created. In this case, the admin password is hardcoded which is not good. We address this problem in the last section.

6. Create the DB on top of the server.

We must also modify the file "terraform/webservice.tf" to:

1. Create a second subnet to host the webapplication.

2. Bind the webapplication and the subnet.

3. We need to create additional application settings in the webapplication block. Those are environment variables which are leveraged by the code to access the DB. This is the same problem as mentioned before, the password is hard coded which is bad. This challenge is addressed in the last section.

At this point you can run terraform and the whole infrastructure including the DB should be deployed. Remember to update your publishing-profile information (your secret on GitHub). Then, you can commit and push your code into GitHub and the deployment into Azure should be completed automatically through the GitHub action we defined earlier.

Security note: in this case, we do publish some confidential information into GitHub like the credentials to access our DB. We will solve this issue in the next section but keep in mind that this should not be done in reality. Actually, you should receive a warning email from GitGuardian [17] which is scanning all public repository for credentials and is warning owners.

[17] https://www.gitguardian.com/

4 Adding Security

In this section, we see how to improve the security from our infrastructure. This is an overview and not a deep dive concept at this point.

4.1 Managed Identity

As you can see, our webapplication needs to access our postgres server on the Azure network. For this to happen, we need some sort of trust between the webapplication and the DB server. As we mentioned earlier, this is happening through credentials which are stored as environment variables (locally in the ".env" file and on Azure as application settings).

Azure has a concept of Managed Identity [18] which enables trust between services without you having to care about credentials. The whole process is automated and transparent to the user. Unfortunately, our postgres flexible server service does not support "managed identity" yet (at least at the point we wrote this document). An alternative solution is to use a keyvault and to store the credentials as secret in this keyvault. The keyvault service does support "managed identity" which solves our chicken-egg problem. What we want to achieve is the following: when the webapplication needs to access the DB, it retrieves the credentials from the keyvault. Through managed identity the webapplication is trusted by the keyvault. In this case, credentials are properly protected. Let's see how to implement that.

Managed Identity is controlled by the Azure AD. For the keyvault (which is a sensitive service), we have different layer of access controls. First, we have Azure AD which defines through RBAC [19] if the webapplication can access the keyvault service. The webapplication is identified through a "service principal id". Then, we have access policies on the keyvault which define what the service principal can do on the keyvault itself.

[18] https://docs.microsoft.com/en-us/azure/active-directory/managed-identities-azure-resources/overview

[19] https://docs.microsoft.com/en-us/azure/role-based-access-control/overview

4.1.1 Keyvault and Terraform

In your reference folder, load the last branch of the project (keyvaultAzure).

As usual, I have created a new file in "terraform/keyvault.tf" which is the base for our keyvault on Azure. But before we look at this file, let's make sure Azure set a service principal for our webapplication. In the file "terraform/webservice.tf", set the entry "identity" in the block "webapp". This tells Azure that he must create a service principal (i.e. an identity) for the webapplication. This identity can be used later on.

Then, we deal with the keyvault configuration in "terraform/keyvault.tf". In a nutshell:

1. Create the keyvault with standard information.
2. Tell Azure AD he must authorize the webapplication to access the keyvault through the definition of a RBAC block, in this case as "Reader".
3. Define the access policies on the keyvault: one for the admin, i.e. the current user who is logged in [20]; and one for our webapplicaion which has only "get" access rights. Note how objects are identified through the definition of "object_id". Access policies are the second layer of access controls we explained above.

Finally, we need to deal with the access credentials of the postgres DB. Because we don't want any credentials to be hardcoded, we will generate the access credentials on the flight, i.e. using a random function in terraform. This is also done in the file "terraform/keyvault.tf". We generate a random password and set the key "secret-sauce". This key is our access credential for our DB. For this to work, we need additional changes.

In the file "terraform/postgres.tf", we need to tell terraform to use the generated password instead of the hardcoded password when it generates the DB. Look at the line "administrator_password". Also, you can remove the entry "DBPASS" in the application settings in "terraform/webservice.tf" as

[20] Note that all actions conducted by terraform are done by the user which is logged in

the password is not retrieved from the enthronement variable any-more but from the keyvault. Let's see what changes need to be done in the code for our solution to work.

4.1.2 Keyvault and Code

Azure is providing libraries to deal with authentication [21]. The library is clever enough to understand what identity should be used depending on the context (locally or on Azure). For example, if the code is running locally and you are logged into Azure, it will use your identity. If the code is running on the webapplication in Azure, it will use the service principal we configured previously. This is very convenient as you have one function which can deal with different contexts. In "base.py" we need to change the code to tell python where to retrieve the credentials to access the DB. If the code is running on Azure, we want the code to use the credentials stored in the keyvault and not in the environment variable any-more.

1. Install the Azure libraries (see "requirements.txt").
2. Import the libraries in "base.py".
3. Change the if/else block. Note the line "credentials = ..." which is where the magic happens.

At this point, the application is running locally and on Azure and you should not find any credentials hardcoded in the code (except for the file ".env" which is an exception for the purpose of this training).

4.2 Other Security Considerations

Easyauth The webapplication is accessible from anybody in Internet. For example, you can run a scan to see what vulnerability you can find. If you

[21] https://docs.microsoft.com/en-us/azure/developer/python/sdk/authentication-overview

are designing a webapplication which should be used by a limited number of persons, you can shield it using Azure authentication process [22]. On the Azure platform, go on your webapplication service in "Settings/Authentication" and choose an identity provider. Once activated, people will have to authenticate before they can access your webapplication. Note that it acts as a proxy in front of your application, but once the person is authenticated, the webapplication is not protected.

Injection In the code, specifically in the file "app.py", we do implement some basic protection mechanisms against cross-site scripting. If you try to inject HTML tag in the URL, you will get an error message asking you to avoid using HTML tag. Note that for the sake of simplicity, we do process values which are sent through the URL, i.e. through GET requests. This is not a good practice as it allows injections.

Network We did not implement special protection on the network layer. We just add a network security rule with a "dummy" rule to illustrate how it works. This should be improved in a productive environment.

If you develop a productive instance, make sure to go through all Microsoft recommendations [23].

5 Annex: Debugging

Debugging might be a tricky exercise on Azure. Those are some hints I used when I set up this training.

The code should always works locally first, i.e. before you publish the code, solve all errors related to python or alike.

You can access the logs of the webapplication using the following command:

```
> az webapp log tail --name canopus-webapp
--resource-group canopus-rg
```

[22] https://docs.microsoft.com/en-us/azure/app-service/overview-authentication-authorization

[23] https://docs.microsoft.com/en-us/azure/app-service/security-recommendations

The logs can also be accessed online on the platform under "webapplication/-monitoring/log streams".

You have an SSH access to the webapplication through the platform under "webapplication/Development tools/SSH". This should help you see if all files have been properly transferred in case of issue. It also helps during network debugging issue, you can ping the DB for example or make sure the DNS name resolves properly using "dig".