Report for the Systems Engineering course's project

Goal

The goal of our project was to successfully establish a Continuous Deployment pipeline that automatically updates a cloud instance of an application every time a commit is made in the source code repository.
As a basis for the assignment, we chose to use Mealie, an application for managing recipes which is composed of a frontend *(Vue.Js)*, a backend *(FastAPI)*, and a database *(Postgres)*.
We managed the deployment on Azure with the avail of Terraform.

Technologies we used

- GitHub/GitLab for hosting the repository and running the pipeline
- Docker for image building
- Docker Hub for storing the built images
- Terraform for deploying on Azure
- Terraform Cloud/GitLab for saving the terraform state

Generally, upper-mentioned technologies were chosen by considering the following criterion:

- popularity and documentation availability;
- flexibility;
- ease of use;
- costs.

In particular:

- GitHub and GitLab are the widest used code hosting websites and both provide ways to set up pipelines *(more on that in the following sections);*
- Docker is the most popular system to run containerized applications, independently from the execution environment (local machine, cloud provider, kubernetes cluster, etc.);
- Docker Hub was chosen in spite of e.g. Azure Container Registry (ACR) so to be more flexible and not to tie to any specific cloud provider;
- Terraform is the leading Infrastructure-as-Code software and most importantly has support for all major cloud providers (AWS, Azure and Google Cloud);
- Azure offers a free Student Pack whose registration does not require specifying credit card details. Its services are quite easy to set up thanks to the extensive documentation and overall interface intuitivity.

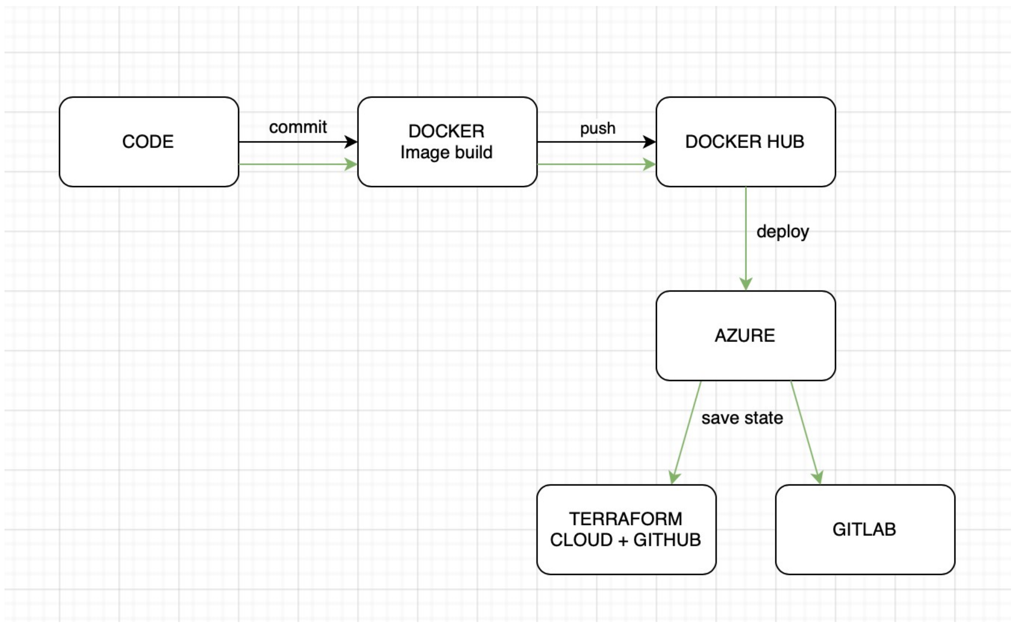
Implementation of the pipeline

Our pipeline is structured so to build an image on Docker every time there is a commit on the repository (for each branch and also pull requests). To simplify the further steps, we decided to bundle together frontend and backend in a single image (see cd/image folder). Once the image is built, it is pushed to Docker Hub with an appropriate version tag (particularly, the branch name or the release version).

The project includes the docker-compose.yml file, which contains all the instructions for connecting together the image and the database. Environment variables are defined along volumes for both image data and database.

The actual deployment is then carried out through Terraform, but it is actually executed only in the case where commits are made on the *main* branch. We then have a main.tf file to manage this stage. In particular, we create some entities specifically required by Azure: a resource group *(global container for all the other elements)*, a storage account and two storage shares *(one for the database and one for the image data)*. The two stage shares are then linked in the so-called *App Service*, where the docker compose file is also uploaded.

One thing to mention is that we initially intended to include a test phase. However, it turns out that the upstream application we are using has failures in tests that would definitely compromise the pipeline execution. These issues were not fixed within the project's span, and therefore we could not implement this phase properly.



Pipeline Diagram
Green: flow of the main branch
Black: flow of all other branches

Saving Terraform status

Ideally, we would like to keep track of the created Azure resources as soon as the deployment is finished. However, GitHub does not have such a feature. Consequently, we decided to use Terraform Cloud to make up for the lacking service: this website keeps track of the deployed resources, also allowing to modify or to destroy them.
Another alternative we tried was to use GitLab, which is indeed capable of saving the state, as a substitute of both GitHub and Terraform Cloud. The goal was to avoid having to rely on too many services, and the results were quite satisfactory, since everything worked as intended and with the clear advantage of having all the configuration stored in a single place. The only issue was that setting up the pipeline took more time as there was a lack of ready-to-use templates, unlike with GitHub Actions.

Change cloud provider

If we wanted to change the cloud provider, for example to switch from Azure to AWS, we would need to completely rewrite the Terraform configuration file, as it uses provider-specific concepts. The pipeline and the docker configuration would instead not change; the same would be for the technologies.

Encountered Issues

Overall, the biggest problems we faced was to make sense out of the different reference websites so to have a fully running pipeline. Configuring terraform was definitely the hardest part, as we needed to consult documentation from Azure, Terraform and Docker websites, which sometimes were using different approaches or were not completely up-to-date. Probably, the issue is that commands and options are changing quite often, so it is quite easy to find examples that do not work anymore. Nonetheless, after spending some time trying different solutions, we eventually found a way through.

Addendum — Running the pipeline

To allow the pipeline to successfully deploy on Azure, it is required to specify some repository “secrets” in GitHub/GitLab configuration. Particularly, one needs to have a valid Azure, Docker Hub and Terraform Cloud account.

For Azure, install the CLI on the local machine and run `az login`. After inserting username and password, run `az ad sp create-for-rbac --role="Contributor" -scopes="/subscriptions/[user_id]"` to create a set of credentials specifically dedicated for the deployment. The console outputs a JSON object with much information. This needs to be mapped to the environment variables that Terraform is expecting:

- *id* maps to ARM_SUBSCRIPTION_ID
- *tenant* maps to ARM_TENANT_ID
- *appId* maps to ARM_CLIENT_ID
- *password* maps to ARM_CLIENT_SECRET

For Docker Hub in GitHub, it is sufficient to provide username and password in the DOCKER_USERNAME and DOCKER_PASSWORD secrets. GitLab instead requires DOCKER_USERNAME and DOCKER_AUTH_TOKEN, which can be obtained by logging in with the command line *(*`docker login`*)* and then looking at the content of the file `.docker/config.json`.

For Terraform Cloud (therefore only in GitHub), one needs to create a User Token in the web interface and consequently use its value in TF_API_TOKEN.