

UNIVERSITY OF PADOVA

Department of Information Engineering
MASTER DEGREE IN COMPUTER ENGINEERING
HIGH PERFORMANCE AND BIG DATA COMPUTING
CURRICULA

**Design and development of a continuous
integration and continuous delivery
system for a cloud based web application**

Supervisor

PROF. CARLO FERRARI

Candidate

SANDY PIVATO

September 5, 2023

Academic Year 2022-2023

Abstract

The purpose of this work is to describe the process used in order to design and develop a continuous integration and continuous delivery system for a cloud based web application in a company scenario. Given the problem of automatising certain actions the developer had to take, this work show a solution to this task with different solutions for both testing and deploying automatically code to a cloud provider setting.

Contents

- Abstract i
- List of Figures vi

- 1 Background 1**

- 2 Introduction and context 3**

 - 2.1 Context 3
 - 2.2 The Devops methodology 4
 - 2.3 The tools used 5
 - 2.3.1 Gitlab CI 5
 - 2.3.2 Terraform 6
 - 2.3.3 Kubernetes 7
 - 2.3.4 Azure 8

- 3 The architecture 9**

 - 3.1 Summary 9
 - 3.2 The project scope and description 9
 - 3.3 The Architecture 12
 - 3.3.1 Front Office Architecture 12
 - The login system 13
 - 3.3.2 Back Office Architecture 13

4	Continuous Integration	17
4.1	Summary	17
4.2	The repository	17
4.3	The CI	19
4.3.1	Frontoffice CI	19
4.3.2	Backoffice CI	24
5	Continuous Delivery	27
5.1	Summary	27
5.2	The cloud architecture	27
5.3	Deployment configuration	30
5.3.1	Terraform structure	30
5.3.2	Configuration a microservice deployment	31
5.3.3	Configuration for other resources Deployment	34
5.3.4	Development and Production environment	35
5.4	CD for Infrastructure	36
5.5	Deployment of a microservice	38
6	Conclusions	41
	References	43
	Acknowledgements	45

List of Figures

2.1	DevOps chain diagram.	5
3.1	Manager view	10
3.2	Activity request	11
3.3	Back office interface	11
3.4	Front office architecture schema	12
3.5	Login system schema	13
3.6	Back office high level schema	14
3.7	Back office's core system schema	15
4.1	Frontend for frontoffice	17
4.2	Repository for backoffice	18
4.3	Monorepo for frontoffice backends	18
4.4	Stages of frontoffice's frontend CI	19
4.5	Test stage for frontoffice's frontend	20
4.6	Test stage for frontoffice's frontend in the pipeline	21
4.7	Build stage for frontoffice's frontend	22
4.8	Build stage for frontoffice's frontend for dev and prod	23
4.9	Build stage for frontoffice's backend	24
4.10	Test stage for backoffice's frontend	25
4.11	Build stage for backoffice's frontend	25

4.12	Test stage for backoffice's backend	26
4.13	Build stage for backoffice's backend	26
5.1	Cloud Architecture	29
5.2	Terraform Code structure	30
5.3	Backoffice backend deployments templates	33
5.4	Application.tf file for Backoffice backend	34
5.5	CD for Infrastructure	37
5.6	Pipeline for Infrastructure	38
5.7	CD for Backoffice backend	39
5.8	Full pipeline for Backoffice backend	39

1 | Background

Web application development and deployment in today's fast-paced digital landscape requires efficient and flexible design. The advent of cloud computing has changed the way applications are developed, deployed and managed. The rise of software development projects has necessitated the adoption of agile methodologies and the automation of development life cycle stages. Continuous Integration (CI) and Continuous Delivery (CD) have emerged as important practices in software development, and enables teams to quickly deliver high-quality applications

This thesis aims to describe the design and development of a system for continuous integration and continuous delivery of cloud-based web applications in a corporate environment, developed in an internship program

Continuous Integration tackles problem of having coherent and qualitative codebase by automating the process of merging code changes and running a series of tests to detect any issues at an early stage. By integrating code frequently, teams can identify and fix bugs quickly, resulting in more stable and reliable software.

On the other hand, Continuous Delivery takes the concept of Continuous Integration a step further by automating the deployment process. This ensures that the application is constantly updated with the latest codebase integration, allowing teams to deploy updates to production or other envi-

ronments with minimal effort. This eliminates the need for manual steps and reduces the risk of human error, making the deployment process more efficient and reliable. This also allows the application to be released in development and production environments, removing implementation responsibilities from developer. For cloud-based web applications, the benefits of adopting a CI/CD framework are essential to ensure a well-managed application release. In particular, manually managing deployments and infrastructure configurations can be time consuming and error prone. By implementing a CI/CD framework tailored to specific cloud environments, manufacturers can provide applications for cloud deployment robust and easy to operate and maintain.

2 | Introduction and context

2.1 Context

The work for this thesis is the result of an internship program I did during the time of six months inside a software development company. The goal of the program was to build an internal tool for employees management starting from scratch, using the latest and most modern technologies and methodologies for web development. The team was composed by 8 interns, supervised by a senior architect and a project manager. The whole project was developed in an agile environment, following the DevOps methodology. Every intern had his specific field of expertise, both functional and technical. The functional roles had the goal of understanding and develop functional solutions starting from the client functional requirements. The technical team, which I was part of, had the goal of understanding the functional solutions proposed by the functional team, and translate them into technical and working solutions. The technical team had several back-end and front-end developers. My role in the team was the architectural and devops engineering tasks. In particular, I was in charge of design and develop of the architecture and CICD for the project. The main tools used for this purpose were: Gitlab CI, Terraform and Azure cloud. Gitlab CI is the core tool for the project, since is the trigger for the whole CICD. In chapter 4, I'm going to explain

detail the continuous integration strategy which was used in the project. In Chapter 5, I am going to explain the continuous deployment part, which will trigger the Terraform tool to deploy on Azure cloud

2.2 The Devops methodology

As mentioned above, my role in the team was to apply the DevOps methodologies to the project. To better understand and present my role, it is useful to give a brief introduction to the role of the DevOps and the aforementioned methodologies. DevOps is a combination of software development (dev) and operations (ops). It is defined as a software engineering methodology which aims to integrate the work of development teams and operations teams by facilitating a culture of collaboration and shared responsibility. [1] There are different key principles of the DevOps culture and methodology, but the most important is the concept of automation: This includes automating testing, builds, releases, the provisioning of development environments, and other manual tasks that can slow down or introduce human error into the software delivery process. In figure 2.1, are shown the important steps of the DevOps methodology. The first part, namely the "Dev" part, is related to the integration of code to the codebase. The code must be build and tested properly. The release is the conjunction between the Dev and the Ops part: the validated code, must be put in production with a coherent release schedule. The second part, the "Ops" part, ensure the correct deployment of the code into the infrastructure, check if everything works fine during the life cycle of the release and monitor the performance. In the end, a new plan of features restarts the chain. This methodology is widely used in modern software development teams, and was therefore chosen a a methodology for

the internship I took part into.

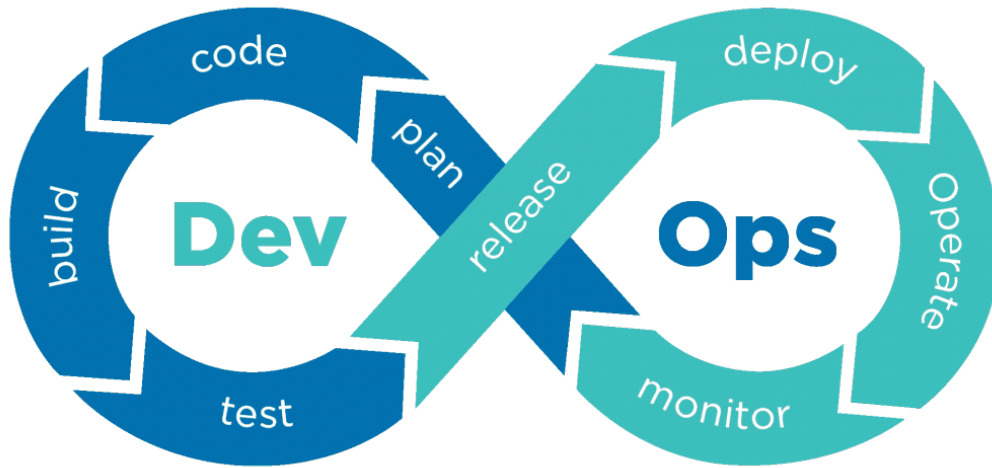


Figure 2.1: **DevOps chain diagram.**

2.3 The tools used

2.3.1 Gitlab CI

The whole project used Gitlab as versioning tool to store and manage the repositories of the code. The choice was made to leverage it's powerful integrated CICD tool: other versioning tools needs the use of third party tool like Jenkins or Chef, while Gitlab provides a strong native integration with the CICD. In particular, as I will describe in later chapters, Gitlab CI provides the ability to run the whole CICD within the codebase with ease and a lot of customization possibilities. Some notables examples, used in the project, are:

- The possibility to write the configuration file directly in the codebase, with automatic recognition of it, without the need of connecting it in complex ways.

- The possibility to modify the configuration directly from the Gitlab interface and to have a quick feedback about errors.
- The possibility to store variables useful directly in the repository and modify them.
- A great native kubernetes compatibility.

2.3.2 Terraform

Infrastructure as Code (IaC) is the managing and provisioning of infrastructure through code instead of through manual processes. [2] HashiCorp Terraform is an infrastructure as code tool that lets the user define cloud resources in human-readable configuration files that you can version, reuse, and share. [3] The tool was used to provision and manage the whole CD part of the CICD. As I am going to explain in more detail in chapter 5, the Gitlab CI triggers a Terraform application which has several roles:

- Check the current state of the infrastructure by retrieving information about it through APIs.
- If it detects changes at the cloud resource level, compare the current version to the newly created.
- Apply the incoming changes and assures the correct deployment.

A Terraform core concept are the "providers". Each cloud provider has it's own APIs and needs a tailor made IaC connection, with keywords and properties. Inside each provider, are defined the "resources", which are the possible piece of infrastructure which can be built through the use of the said provider. The resource management and creation is the main feature

of an IaC tool and is has been used extensively in the project. Terraform was chosen instead of other competitors because it has a wide community to support it, an official Azure support for the provider and it's open source design. The main drawback of using Terraform is it's steep learning curve: other competitors let the user use the IaC tool with know programming languages like Python and Go, while Terraform has it's own programming language (HCL), which can be cumbersome when learning it.

2.3.3 Kubernetes

Kubernetes (k8s for short) is an open-source container orchestration platform that automates the deployment, scaling, and management of container applications. [4] Originally developed by Google, it is now managed by the Cloud Native Computing Foundation (CNCF). Kubernetes provides a framework for managing the deployment and performance of application containers across hosts. It pulls out the underlying components and provides a unified API for container management, allowing developers and operations teams to focus on application logic rather than infrastructure issues

Applications in a Kubernetes environment are executed through containers (usually using Docker) with their dependencies and runtime environments. Kubernetes then takes care of distributing these containers across a cluster of machines, ensuring high availability, scalability and optimal resource utilization. It performs functions like container scheduling, load balancing, scaling, rolling updates and self-healing. In the project, kubernetes service is provided by Azure and it's responsible for managing the containers of our different microservices

2.3.4 Azure

A cloud service provider is an IT company that provides on-demand, scalable computing resources like computing power, data storage, or applications over the internet. [5] Microsoft Azure is a public cloud computing platform widely used in the industry which can provide all the needed tools for a development team to host applications without the need of building a architecture of servers and resources locally. [6] In the project, Azure was chosen as the cloud provider instead of other competitors because of internal company agreements for usage licenses.

3 | The architecture

3.1 Summary

In this chapter I will present the architecture of the project by showing different aspects and choices made to design the core feature of the project, to satisfy the needed requirements.

3.2 The project scope and description

As mentioned in the introduction, the project had the goal to create from scratch an internal employee management tool. I will now explain in more detail the structure of the project, to better understand the choices made during the design of the architecture part. In brief, the final product developed by team was divided in two parts: front office and back office. The front office is the part where the final user interacts. There are different kind of users, each with its own view and restrictions. In particular, is useful to describe some roles:

- The consultant: is the least powerful user. The said user does not have the visibility on other consultants and can only manage certain parts of the tool. The consultant can insert certain activities on the tool, each with a specific value, to the request some reward. This system was one

of the core requirements of the project and it took the name of "Alten grains".

- The manager: the manager has the visibility on all the consultants in its team. The manager can approve or deny the grain requests and gets notified with an email when a new request is created.
- The system administrator: has the same visibility and restriction as a normal consultant but has a special access to the back office part of the tool

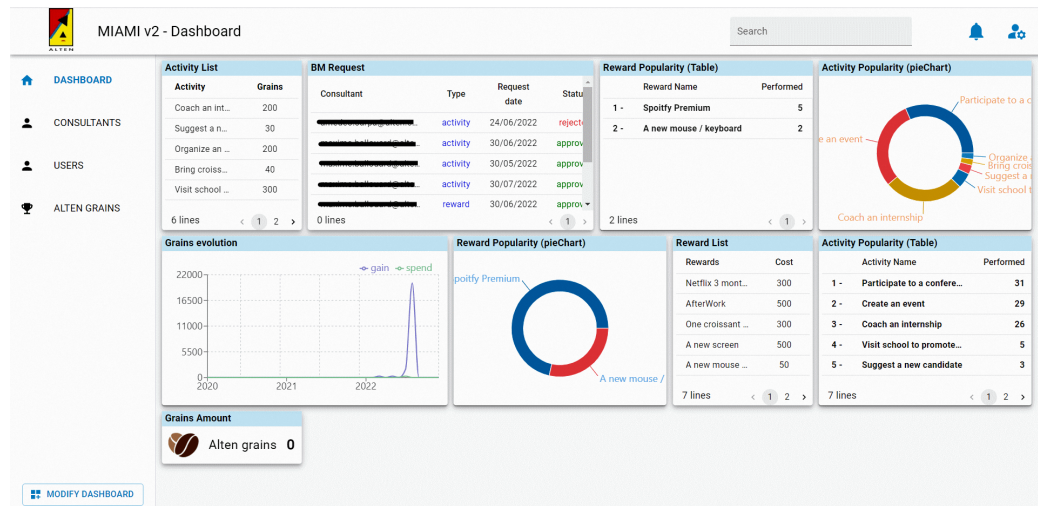


Figure 3.1: Manager view of the front office dashboard

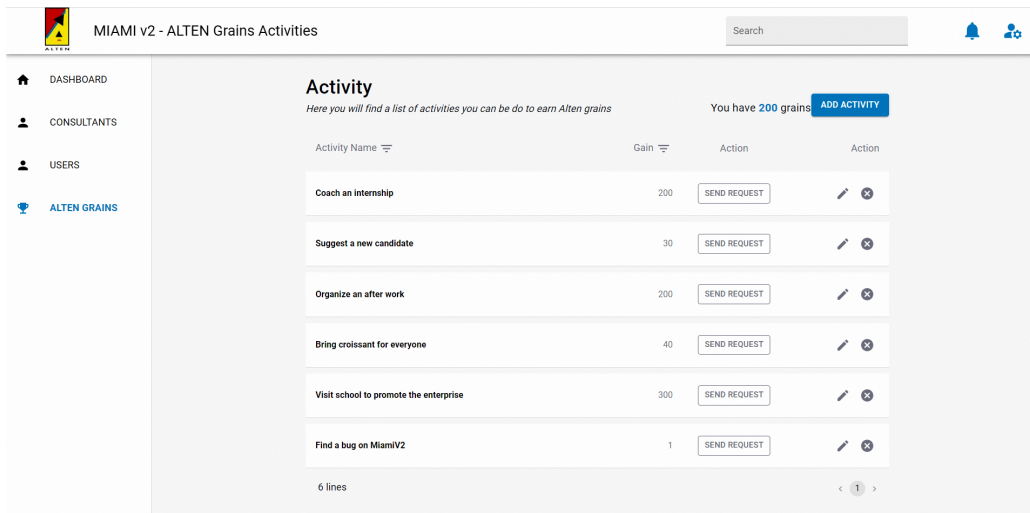


Figure 3.2: Request of an activity from a consultant

The back office on the other hand is only accessible and managed by the system administrator. The purpose of this part of the tool is to monitor the performances of the platform and to manage the different agencies of the company, "Tenants", by initializing and populating the users. The system administrator can also create and manage the roles of the users.

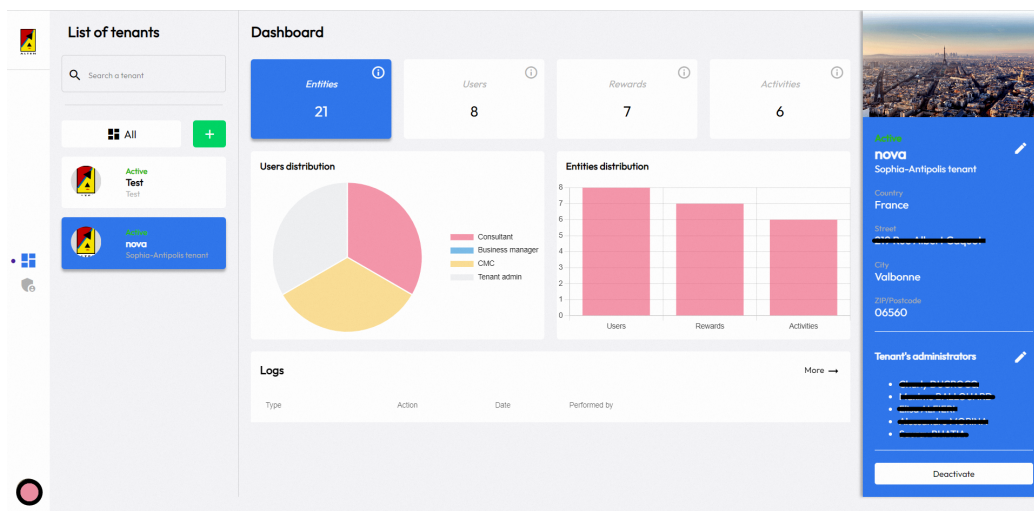


Figure 3.3: Back office interface

The login system

In the requirements for the project, it was required that the tool must be accessed with the company's credentials, which are stored in a Microsoft Active Directory. This has been proven as a challenging problem. Our login system was therefore based on MSAL (Microsoft Authentication Library) [7]. The protocol works as shown in the picture below. When performing the login, the application sends the authorization request with, username and password, to the Microsoft auth service which, if the credentials are present in the active directory, sends back an encrypted access key, named JWT token. This token then gets converted to a new format specific for the application and used by other micro services to perform their duties.

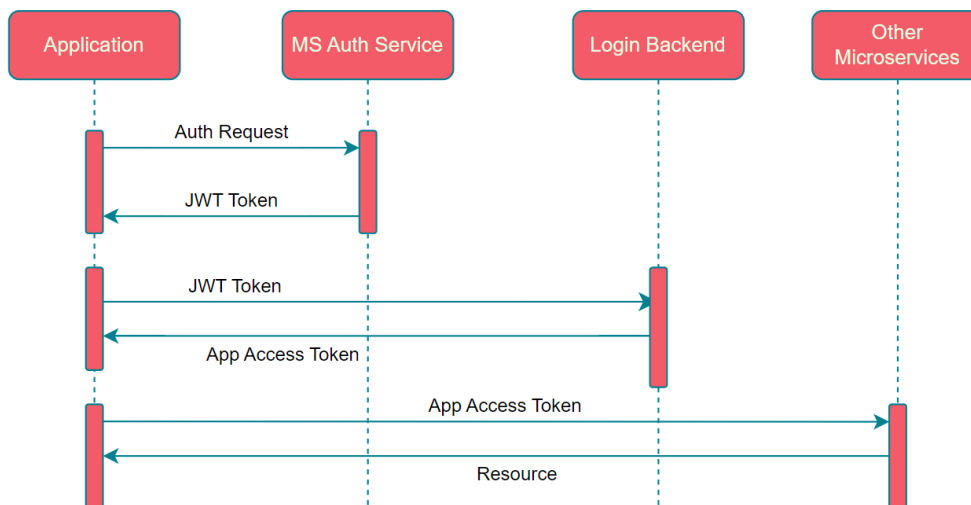


Figure 3.5: **Login System** schema

3.3.2 Back Office Architecture

Let's now focus on the architecture for the back office, which will be essential to understand better the steps behind the automation of the processes thanks to the CI/CD. To better comprehend the architecture, which schema is in

figure 3.7, let's describe before the high level schema of the back office.

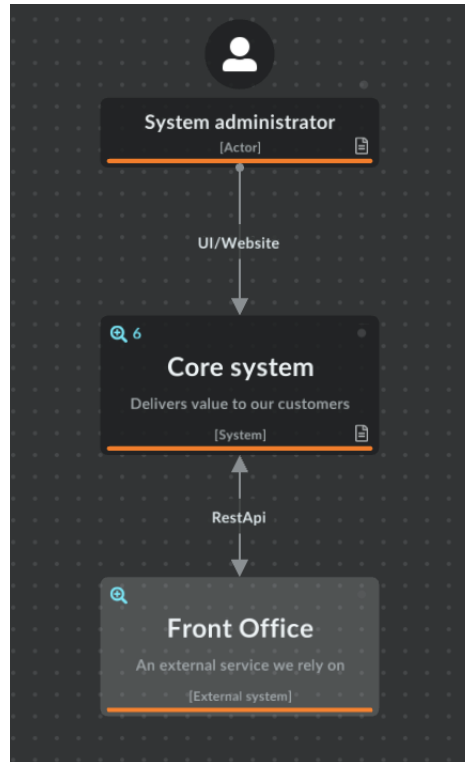


Figure 3.6: **Back office high level schema**

A system administrator, which gets its role defined manually in company's active directory, needs to interface, through the UI shown in figure 3.3, with the Core system I am going to describe in depth below. The Core System then, interacts through RestApi with the Front Office to provide the needed information.

Let's dive deeper into the Core System to better understand how the architecture is designed, as shown in figure 3.7

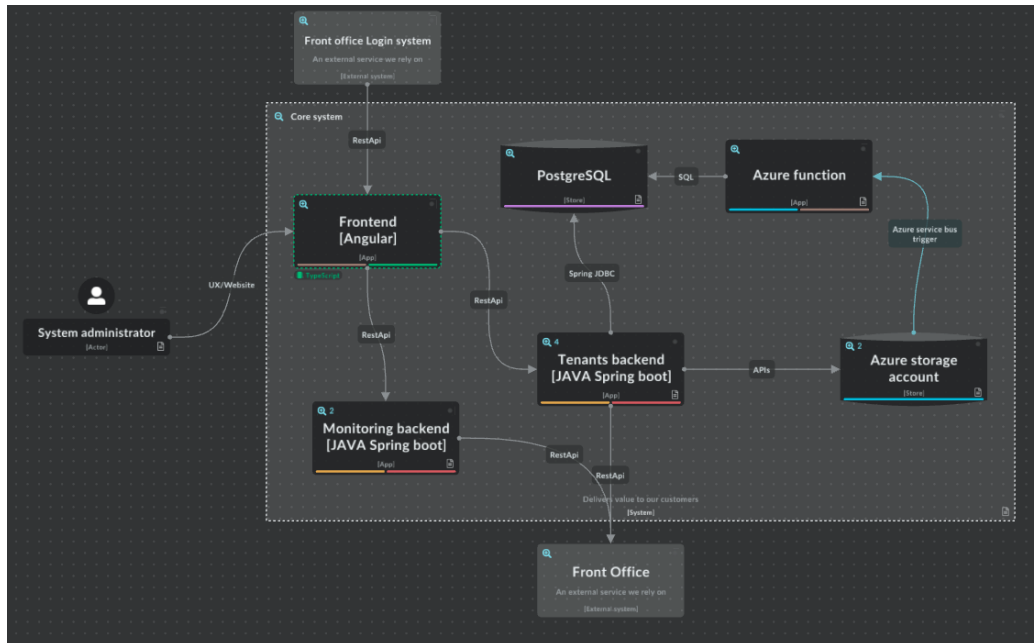


Figure 3.7: **Back office's core system schema**

The System administrator, through the UI, connects to the frontend, which was developed in Angular. This is the entry point of the schema. The frontend, communicates through RestApi with the Tenants backend, which is the backend responsible for the CRUD operations for the different tenants of the company, and the monitoring backend, which is responsible to retrieve and process statistics and for the whole system and produce the output shown in figure 3.3. The Tenant's backend is also connected to a PostgreSQL database, made to store the information provided by the backend, and the Azure storage account. The Azure storage account is a particular kind of database which stores "blob" files. This file are binaries, and its purpose is to let us store images for the tenants and the logs from the statistics. The storage is also connected to the Azure function, a particular code developed using Azure integrated tools, which let us read and manipulate the logs and inject

SQL to the database to save chunks of logs useful for the backend. This latest part will be described more in depth in chapter 5 while describing the cloud architecture for the project. This explanation concludes the description of the architecture designed for the project. In the next chapter, I am going to focus on the automatisation and the CICD, the core of this elaborate.

4 | Continuous Integration

4.1 Summary

The importance continuous integration has already been discussed in previous chapters of this work. In this chapter, after a brief description of the codebase structure, I will describe in depth how I design and integrate a fully functional continuous integration system and show some remarks of this task.

4.2 The repository

Before describing the continuous integration of the code, is important to take a quick look at the project structure to understand better how the code is actually integrated in the codebase. There are conceptually 3 projects: the front office project, the back office project and the infrastructure project. Being two different and separated websites, both front office and back office include a frontend and a backend each.

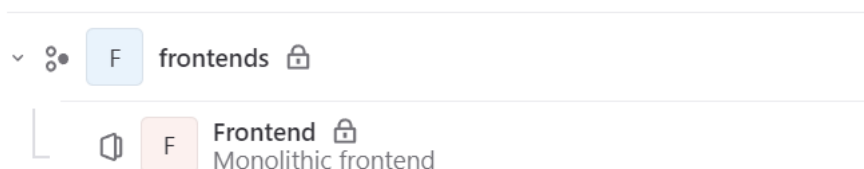


Figure 4.1: **Frontend for frontoffice**

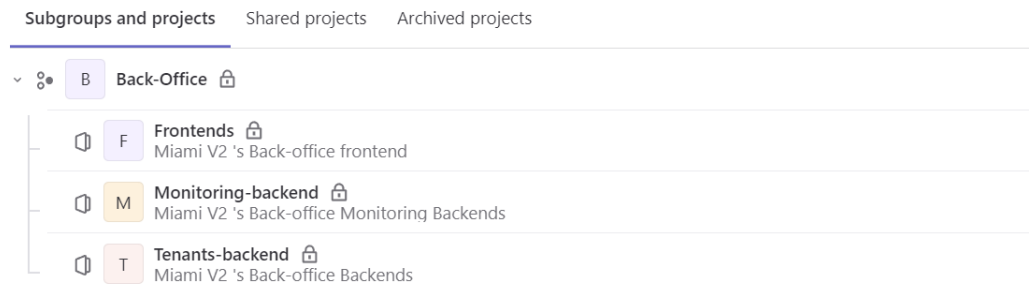


Figure 4.2: **Repository for backoffice**

There is an important difference between frontoffice and backoffice structure. Both projects are microservice based but, being a complex and big architecture project, frontoffice used a different approach for backend called **monorepo**. In the context of microservice based applications, a monorepo [8] is a repository which contains multiple microservices, sharing the same CICD. This opposes to the multirepo strategy, in which every microservice has its own repository, with its own CICD. The choice has been made to ease the collaboration between the team, to ease and standardise the implementation of unit and integration tests through a single CICD and to make easier to see the contribution of each member of the team to the project.

alten-grains-backend	snackbar_modifying
candidate-backend	reverse interviews order
common	added a new grpc function to get list of users
consultant-backend	feat/filterCandidates : filter queries done, need to do te...
login-backend	added a new grpc function to get list of users
notification-backend	feat: add tenant linked in notification + hard code forbid...
postman/test	ci: main file
proto-messages	added a new grpc function to get list of users
test	added tests (100% coverage on service and controller)
widget-backend	ci: make deploy depends from docker

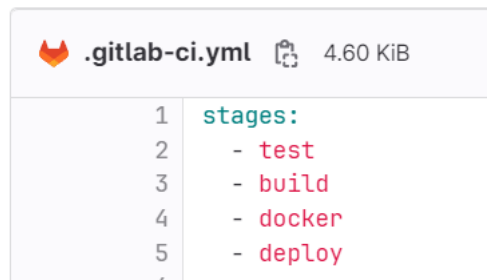
Figure 4.3: **Monorepo for frontoffice backends**

4.3 The CI

As already presented, Continuous integration (CI) is the practice of automating the integration of code changes from multiple contributors into a single software project. [9] In projects we have two different CIs, one for frontoffice and one for backoffice. Conceptually the two are similar, but they present some differences which will be explained in this chapter. The definition of CIs jobs are all contained in a `.gitlab-ci.yml` file.

4.3.1 Frontoffice CI

For frontoffice project, two pipelines needed to be built, one for frontend and one for backend. Let's start analyzing the CI for frontend. The CI is composed in different stages, shown in figure 4.4



```
.gitlab-ci.yml 4.60 KiB
1 stages:
2   - test
3   - build
4   - docker
5   - deploy
6
```

Figure 4.4: **Stages of frontoffice's frontend CI**

In this sections I will describe the test and build stages, while the others will be described in depth in the next chapter as the core of the CD.

The `test` stage, as shown in figure 4.5 is used to perform unit tests and produce a document in xml format which contains information about tests, reporting the percentage of code covered by unit tests. The higher this value, the better. For our intership, we decide to keep the code coverage higher than

85% but not to refuse a merge request if that would put the percentage lower than the threshold. This is risky and in production conditions should be avoided, but for the sake of simplicity we decided just to deliver a warning in the page and keep the 85% threshold as a good practise rule.

```
test:unittest:
  stage: test
  image: node:16-alpine
  coverage: /All files[^\|]*\| [^\|]*\s+([\d\.]+)/
  before_script:
    - npm install
    - npm i --save-dev @next/swc-linux-x64-gnu
  script:
    - npm run test:ci
  artifacts:
    when: always
    reports:
      junit:
        - junit.xml
    coverage_report:
      coverage_format: cobertura
      path: coverage/cobertura-coverage.xml
```

Figure 4.5: **Test stage for frontoffice’s frontend**

The core command to launch tests is the "npm run test:ci" command. This command, under the keyword "script" of the .gitlab-ci.yml file, is set by the frontend developers to do multiple actions,: the check of code linting, security tests, end to end tests and unit tests. Some of the test, in particular the end to end, are allowed to fail with a warning to the developer.

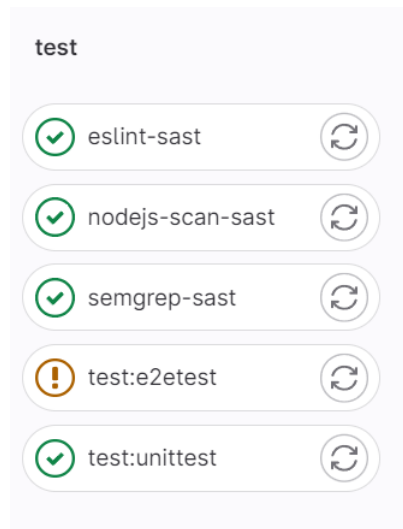


Figure 4.6: **Test stage for frontoffice's frontend in the pipeline**

Let's continue with the **build** stage. This stage has the goal of test the correct build of the application code. The main challenge is to ensure environment variables are correctly injected into the application, which is done directly through the pipeline itself. This stage also add a new challenge. In the project we set up two different environment: development and production. In this stage of the CI is important to test the integration of both environment. As a design choice, every development variable, link and URL needs to contain the `.dev` string. In this case, we just need to append the string for dev environment and omit it for production. From the CI point of view, that translates into using the `ENV_URL_TAG` where needed. The one shown in figure 4.7 is a template shared between development environment and production environment, while in figure 4.8 is shown the implementation of the difference between the two environments. In this stage, is also noteworthy to check the rule

- *if: \$CI_COMMIT_BRANCH == \$CI_DEFAULT_BRANCH*

This is the first example of a design choice that will be used multiple time in

the CI/CD of this elaborate. The rules uses some native environment variables of gitlab and the meaning of this rule is the following: "run this stage only if the branch where the code is pushed is the main branch". In other words, we want to run this stage only if the feature is fully developed and merged with the main branch to be deployed in production. This is a common practise to avoid developers to push their feature branch in production environment before passing the needed quality checks.

```
.build_template:
  allow_failure: false
  stage: build
  image: node:16-alpine
  before_script:
    - npm install
    - npm i --save-dev @next/swc-linux-x64-gnu
  script:
    - npm run build
  artifacts:
    expire_in: 1h
    paths:
      - .next
  variables:
    NEXT_PUBLIC_CONSULTANT_BACK_END_ENDPOINT: https://consultant-backend${ENV_URL_TAG}.miamiv2.alten-dcx.dev
    NEXT_PUBLIC_BACK_OFFICE_ENDPOINT: https://miamiv2.bo${ENV_URL_TAG}.miamiv2.alten-dcx.dev
    NEXT_PUBLIC_CANDIDATE_BACK_END_ENDPOINT: https://candidate-backend${ENV_URL_TAG}.miamiv2.alten-dcx.dev
    NEXT_PUBLIC_WIDGET_BACK_END_ENDPOINT: https://widget-system${ENV_URL_TAG}.miamiv2.alten-dcx.dev
    NEXT_PUBLIC_NOTIFICATION_BACK_END_ENDPOINT: https://notification${ENV_URL_TAG}.miamiv2.alten-dcx.dev
    NEXT_PUBLIC_LOGIN_SYSTEM_BACK_END_ENDPOINT: https://login-system${ENV_URL_TAG}.miamiv2.alten-dcx.dev
    NEXT_PUBLIC_ALTEN_GRAINS_BACK_END_ENDPOINT: https://alten-grains${ENV_URL_TAG}.miamiv2.alten-dcx.dev
    NEXT_PUBLIC_FRONT_END_ENDPOINT: https://miamiv2${ENV_URL_TAG}.miamiv2.alten-dcx.dev/blank.html
    NEXT_PUBLIC_MSAL_CLIENT_ID: d568aab9-4ce6-4f67-a100-4d9dd2c62be4
    NEXT_PUBLIC_MICROSOFT_AUTHORITY: https://login.microsoftonline.com/9bc3d1cd-55ca-4e13-b5a2-a9e9deaeba3f
  dependencies: [ ]
```

Figure 4.7: **Build stage for frontoffice's frontend**

```
build:dev:
  extends: .build_template
  variables:
    ENV_URL_TAG: .dev

build:prod:
  extends: .build_template
  when: manual
  rules:
    - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
  variables:
    ENV_URL_TAG: ""
```

Figure 4.8: **Build stage for frontoffice's frontend for dev and prod**

This concludes the CI for frontend application for the frontoffice. Conceptually all the other CIs are very similar so I'll focus on the differences between them. For the backend microservices of the frontoffice, as stated before, I develop have one single CI for all the different microservices. The test stage is very similar, just using different commands to comply with the language requirements. The build stage too is very similar, but needs a tweak in order to work with the monorepo approach. When merging a new branch to the main, the system detects if there are changes in the common folder for all microservices, which contains shared utilities between them. If there are changes, that will trigger a build for all the microservices. If there are no changes in that folder, it'll only build the microservices modified. The code is shown in figure 4.9

```
build:all:
  dependencies: [ ]
  stage: build
  image: node:16-alpine
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
      when: never
    - changes: !reference [ .common_files, changes ]
      if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
  before_script:
    - yarn install
  script:
    - yarn build
```

Figure 4.9: **Build stage for frontoffice's backend**

4.3.2 Backoffice CI

Being two similar project, I develop the CI for the backoffice is a similar fashion. The programming languages here are different so the tools used are different too, but the same concepts remains valid. There are two stages, the test and the build, which follow the same logic described before. One noteworthy change is that, in the build stage for frontend, there is no difference between development and production anymore. This is justified by the fact that, being a smaller project, there are no difference in terms of variables between development and production environments.4.11 Another noteworthy change is the addition of a separate code_coverage stage in backend pipeline. This change is due to the decision of putting a hard limit to the code coverage: using a tool named "jacoco" I decided to limit the coverage to 85% . The merge request will fail if the code coverage is below that threshold. Figure 4.12


```
test:unittest:
  stage: test
  before_script:
    - npm install -g @angular/cli
    - npm ci
    - apt-get update
    - wget https://dl.google.com/linux/direct/google-chrome-stable_current_amd64.deb
    - apt install -y ./google-chrome*.deb
    - export CHROME_BIN=/usr/bin/google-chrome
  allow_failure: false
  coverage: '/Statements\s*: \d+\.\?\d+/'
  script:
    - npm run test:ci
  artifacts:
    reports:
      coverage_report:
        coverage_format: cobertura
        path: ${CI_PROJECT_DIR}/coverage/frontends/cobertura-coverage.xml
    when: always

test:e2etest:
  image: cypress/browsers:node16.14.2-slim-chrome100-ff99-edge
  stage: test
  before_script:
    - npm install -g @angular/cli
    - npm ci
  script:
    - npm run start:e2e & npx wait-on http://localhost:3000
    - npx cypress run
```

Figure 4.10: Test stage for backoffice's frontend

```
build:
  stage: build
  dependencies: []
  before_script:
    - npm install -g @angular/cli
    - npm ci
  script:
    - npm run build
```

Figure 4.11: Build stage for backoffice's frontend

```

test:
  stage: test
  script:
    - mvn $MAVEN_CLI_OPTS clean org.jacoco:jacoco-maven-plugin:prepare-agent test jacoco:report
  coverage: '/Total.*?([0-9]{1,3})%/'
  artifacts:
    paths:
      - target/site/jacoco/jacoco.xml

#check and verify jacoco rule limits
code_coverage:
  stage: code_coverage
  image: registry.gitlab.com/haynes/jacoco2cobertura:1.0.7
  script:
    - python /opt/cover2cover.py target/site/jacoco/jacoco.xml $CI_PROJECT_DIR/src/main/java/ > target/site/cobertura.xml
  needs: ["test"]
  dependencies:
    - test
  artifacts:
    reports:
      coverage_report:
        coverage_format: cobertura
        path: target/site/cobertura.xml

```

Figure 4.12: **Test stage for backoffice's backend**

```

#build and package app
maven-build:dev:
  extends: .maven_build_template

maven-build:prod:
  extends: .maven_build_template
  rules:
    - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
      when: manual

```

Figure 4.13: **Build stage for backoffice's backend**

This concludes one of the two main parts of this elaborate. In the next chapter I will describe the second part, the continuous delivery system I design for this project.

5 | Continuous Delivery

5.1 Summary

In this chapter I will explain in details the continuous delivery system I implemented for the project. After explaining in depth the cloud architecture, I will explain how I used the gitlab CICD to automate the process of deploying new features in cloud provider.

5.2 The cloud architecture

In chapter 3 I described the architecture of the whole project, focusing more on the important microservices and communication between them. The goal of this paragraph is to describe the architectural decisions taken for the cloud part of the infrastructure. A schema of the architecture is shown in figure 5.1. The whole architecture is contained in Azure Cloud as stated in the introduction. All the microservices for both backoffice and frontoffice are managed through an orchestrator, Azure Kubernetes Service (AKS) provided by Azure itself to be fully compatible with the Cloud platform. AKS manages the workload, the network and the availability of the containers for application's microservices. The docker images for the microservices are stored in the Azure Container Registry (ACR). As I will describe later, the CICD

pushes the new code versions to the ACR directly, and the microservices fetch the new images from the registry. Frontoffice saves the data in a non relational database, CosmosDB. CosmosDB is a database managed by Azure platform which maintains and provides noSQL structure database to Azure services. [10]. CosmosDB has been chosen because of the need of a noSQL database for frontoffice. This need comes from having a lot of different kind of data which do not fit well the SQL structure. Instead, the noSQL structure gave us the possibility to manage in a cleaner and quicker way the different entities for the project. Having this need, CosmosDB was chosen over more known noSQL databases because of its native integration with Azure Cloud. Another core component of the architecture is the Azure Storage Account. An Azure storage account contains all Azure Storage data objects: blobs, files, queues, and tables. [11]. In particular, in this project, we used two feature of Storage Account: Azure Blob Storage and Azure Queue Storage. Azure Blob Storage is Microsoft's object storage solution for the cloud. Blob Storage is optimized for storing massive amounts of unstructured data. Unstructured data is data that doesn't adhere to a particular data model or definition, such as text or binary data [12]. The need for this particular way of storing data came from the amount of images present in the back office project. Each tenant has multiple images associated to it and therefore the need to store them in the most appropriate way. The images are stored in Blob Storage as binaries and retrieved by tenant's backend to associate them to each tenant. LogBuilder, as the name suggests, is the function responsible of managing the log of the application. This function is developed through a cloud native tool, Azure Function, which natively communicates with other Azure components to make easier to build, read and post messages to the message broker, which in this case is Azure Queue Storage. Messages are sent

to microservices by this message broker and saved to a PostgreSQL database, also managed by Azure Cloud.

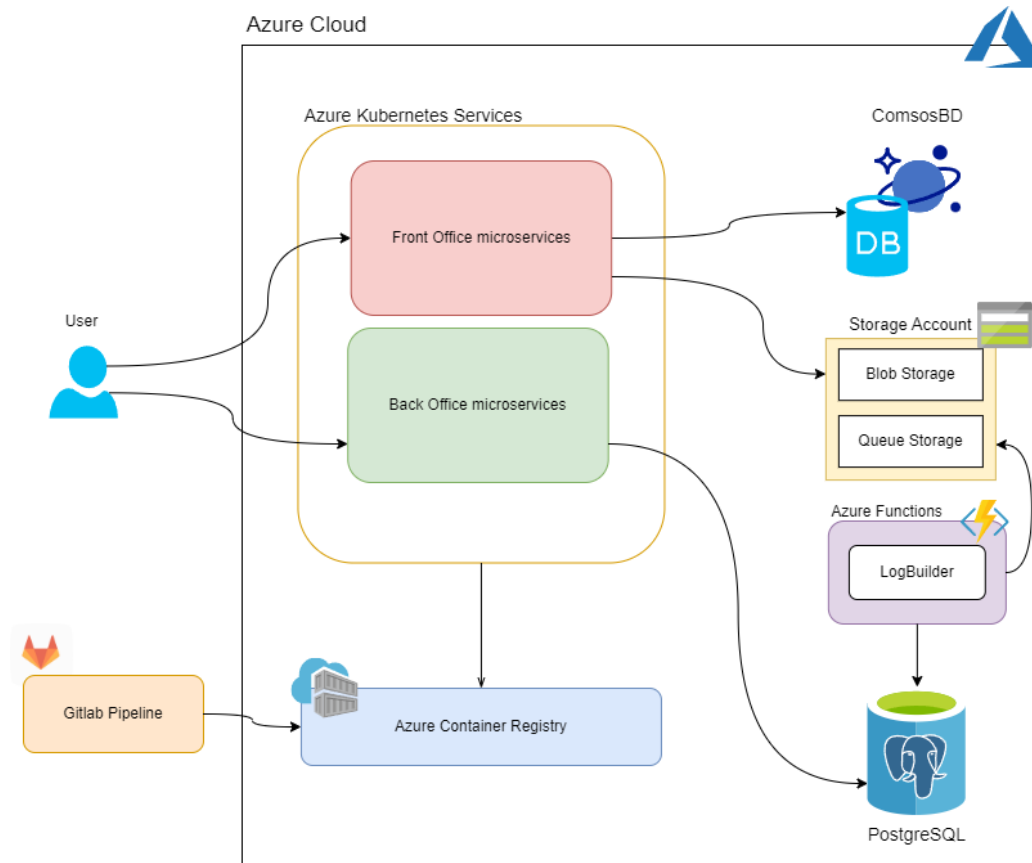


Figure 5.1: **Cloud Architecture**

This concludes the description of Cloud architecture for the project. In the next session I'll explain how this architecture is automatically managed with Terraform and how the code is finally made available to the cloud using the CICD.

5.3 Deployment configuration

5.3.1 Terraform structure

When deploying a cloud architecture, there are two choices. The first one, the more classical, is managing the architecture manually. This creates various problems, like synchronization problems, versioning and less control over the evolve of the infrastructure. The second choice, the one made for the project, is to use an infrastructure as a code (IaaS) tool. In this case, as described in chapter 2 , Terraform has been chosen as the core tool to manage our infrastructure.

To understand how the infrastructure is managed through Terraform let's take a look to its configuration. I will describe in depth one microservice to not stretch the dissertation, since the deployment of each microservice follows the same logical approach.

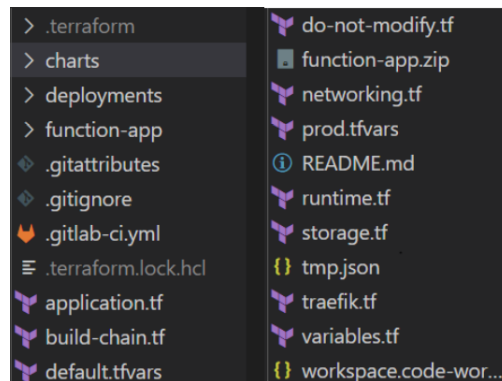


Figure 5.2: Terraform Code structure

I briefly explain each folder and file to focus more on the key ones.

- .terraform folder contains all the terraform files needed for the correct usage of the tool

- charts contains all the Helm Charts for the microservices. I'll explain in more details what a helm chart is in the following paragraph
- deployments contains the deployments value for the microservices
- function-app contains the binaries for the afore mentioned LogBuilder service
- .gitlab-ci.yaml is the CICD for infrastructure file, which will be describe in depth in the next session
- each .tf file contains the definition of different parts of the infrastructure suggested by the name.
- default.tfvars prod.tfvars files contain the definition f the different variables needed for production and development environment.

As previously said, in the next session I'll take a microservice as example and describe how its deployment works.

5.3.2 Configuration a microservice deployment

When a microservice is developed in local and needs to be put in the cloud, the first step is to create a Helm Chart for it. Helm is a tool that automates the creation, packaging, configuration, and deployment of Kubernetes applications by combining the configuration files into a single reusable package. [13].A chart is a collection of files that describe a related set of Kubernetes resources [14]. In particular, here I'm using helm Charts to create the resources templates needed for the microservice.

The helm chart contains the definition of the microservice used by other files

metadata to correctly deploy the application. Each microservice needs the following kubernetes resources to be deployed:

- A ConfigMap, which is where the needed values of the environment variables are set
- A Deployment which is the file setting the values of the image for the container and connecting the ConfigMap to it.
- An Ingress which specifies how to setup the external access to the cluster and the containers
- A Secret where the sensitive data are stored
- A Service which exposes needed network ports to the containers.

The Helm chart for this configuration contains the template for each of these resources. The template for the values is also present in the Helm chart. This is used to populate the chart with the right variables values, specified in the application.tf file. The following figure 5.3 contains an example of the files needed for the deployment of the backoffice backend microservice.


```

apiVersion: v1
kind: ConfigMap
metadata:
  labels:
    | {{ include "chart.labels" . | nindent 4 }}
  name: {{ include "chart.name" . }}
data:
  APPLICATION_ENVIRONMENT: {{ .Values.global.env }}
  APPLICATION_TIMEZONE: {{ .Values.global.timezone }}
  SPRING_DATASOURCE_URL: {{ .Values.postgresql.url }}
  DB_ID: {{ .Values.postgresql.id }}
  JWT_EXPIRATION: {{ .Values.jwt.expiration }}
  BO_FRONTEND_URL: {{ .Values.bo.frontend.url }}
  SPRING_DATASOURCE_USERNAME: {{ .Values.postgresql.login }}
  SPRING_PROFILES_ACTIVE: {{ .Values.postgresql.profile }}
  FE_LOGIN_BACKEND_URL: {{ .Values.login.backend.url }}
  FE_ALTERNATE_BACKEND_URL: {{ .Values.alternate.backend.url }}
  CONNECTION_STRING: {{ .Values.blob.connection.string }}
  SPRING_CLOUD_AZURE_STORAGE_BLOB_ENDPOINT: {{ .Values.blob.endpoint }}
  SPRING_CLOUD_AZURE_STORAGE_BLOB_ACCOUNT_NAME: {{ .Values.blob.name }}
  SPRING_CLOUD_AZURE_STORAGE_BLOB_ACCOUNT_KEY: {{ .Values.blob.key }}
  AZURE_CONTAINER_NAME: {{ .Values.blob.container.name }}
  QUEUE_ENDPOINT: {{ .Values.queue.endpoint }}

```

```

kind: Secret
apiVersion: v1

metadata:
  name: {{ include "chart.name" . }}
  labels:
    | {{ include "chart.labels" . | nindent 4 }}

stringData:
  SPRING_DATASOURCE_PASSWORD: {{ .Values.postgresql.password }}
  JWT_SECRET: {{ .Values.jwt.secret }}

```

```

apiVersion: v1
kind: Service
metadata:
  labels:
    | {{ include "chart.labels" . | nindent 4 }}
  name: {{ include "chart.name" . }}
spec:
  ports:
    - name: http
      port: 8080
      targetPort: "http"
  selector:
    | {{ include "chart.selector" . | nindent 4 }}

```

```

kind: Ingress
apiVersion: networking.k8s.io/v1

metadata:
  name: {{ include "chart.name" . }}
  labels:
    | {{ include "chart.labels" . | nindent 4 }}
  annotations:
    kubernetes.io/ingress.class: traefik
    traefik.ingress.kubernetes.io/router.entrypoints: websecure
    traefik.ingress.kubernetes.io/router.tls: 'true'
    traefik.ingress.kubernetes.io/router.tls.certresolver: le

spec:
  rules:
    - host: {{ .Values.ingress.host }}
      http:
        paths:
          - backend:
              service:
                name: {{ include "chart.name" . }}
                port:
                  name: http

            pathType: Prefix
            path: /

```

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ include "chart.name" . }}
  labels:
    | {{ include "chart.labels" . | nindent 4 }}
spec:
  selector:
    matchLabels:
      | {{ include "chart.selector" . | nindent 6 }}
  template:
    metadata:
      labels:
        | {{ include "chart.selector" . | nindent 8 }}
    spec:
      containers:
        - image: {{ .Values.image.name }}:{{ .Values.image.tag }}
          imagePullPolicy: Always
          envFrom:
            - configMapRef:
                name: {{ include "chart.name" . }}
            - secretRef:
                name: {{ include "chart.name" . }}
          name: {{ include "chart.name" . }}
          ports:
            - containerPort: 8080
              name: http
          livenessProbe:
            httpGet:
              port: http
              path: /api/v1/public/liveness
          restartPolicy: Always

```

Figure 5.3: Backoffice backend deployments templates

The application.tf file is a particular file that uses the integration between Terraform and Helm to provide the Helm chart the right values for the variables 5.4. To deploy different parts of the infrastructure, Terraform uses the "resource" type. The Helm_release resource is a resource which, by specifying

the templates, let's the developer manipulate helm variables by giving them Terraform provided values. This is very convenient because it creates an internal link between the dynamically allocated variables by Terraform and the templates

```
resource "helm_release" "bo-backend" {
  repository = "${path.module}/charts"
  chart      = "${path.module}/charts/bo-backend"
  name       = "bo-backend"
  atomic     = true
  timeout    = 300

  values = [
    templatefile("${path.module}/deployments/bo-backend-values.yaml", {
      psqLogin      = "${azurerm_postgresql_server.postgresql.administrator_login}@${azurerm_postgresql_server.postgresql.name}"
      psqPwd        = random_string.db_password.result
      springProfile = local.spring_profile
      environment   = var.environment
      timezone      = var.timezone
      image_name    = "${local.login_server}/${local.bo_slug}/backends"
      image_tag     = local.image_tag
      ingress_host  = local.bo_backend_url
      jwt_secret    = random_string.jwt_password.result
      postgresql_id = azurerm_postgresql_database.postgresql.name
      jwt_expiration = local.jwt_expiration
      bo_frontend_url = "https://${local.bo_frontend_url}"
      postgresql_url = "jdbc:postgresql://${azurerm_postgresql_database.postgresql.server_name}.postgres.database.azure.com:5432/postgresqldb"
      login_backend_url = "https://${local.login_backend_url}"
      alten_grains_backend_url = "https://${local.alten_grains_backend_url}"
      blob_endpoint   = azurerm_storage_account.blob-storage.primary_blob_endpoint
      blob_key        = azurerm_storage_account.blob-storage.primary_access_key
      blob_name       = azurerm_storage_account.blob-storage.name
      blob_connection_string = azurerm_storage_account.blob-storage.primary_connection_string
      queue_endpoint = azurerm_storage_account.blob-storage.primary_queue_endpoint
      blob_container_name = azurerm_storage_container.blob-container.name
    })
  ]
}
```

Figure 5.4: Application.tf file for Backoffice backend

The procedure is the same for each microservice created for the project.

5.3.3 Configuration for other resources Deployment

In the previous section I described how to prepare the configuration to deploy a microservice, but there are plenty of other resources needed for the cloud system to work. Terraform lets the developer manage all the resources through itself. Once connected to the Azure Cloud through the azurerm backend configuration, with the storage account and the key to access it, by creating resources I managed to create the architecture present in the previous chapter using the Azurerm provider [15]. The following resources are used to create the Azure Cloud environment:

- `azurerem_container_registry` to connect the ACR to Terraform and retrieve information for the microservice image
- `azurerem_public_ip`, `azurerem_dns_a_record`, `azurerem_virtual_network`, `azurerem_subnet`, `azurerem_private_dns_zone` are used to create and manage the network for the project.
- `azurerem_kubernetes_cluster` is used to create the kubernetes cluster on top of the requested physical machines.
- `azurerem_cosmosdb_account`, `azurerem_postgresql_server`, `azurerem_storage_container`, `azurerem_storage_queue` and `azurerem_function_app` to manage the databases, the blob storage, the queue storage the and the azure function deployment

Through the modification of these files and usage of these providers, is possible to create a fully functional cloud infrastructure by using Terraform.

5.3.4 Development and Production environment

While developing a web application is fundamental to have at least two different environments:

- A development environment where developers can see their features hosted directly in the cloud, instead than see them in their local machines. This step is crucial for testing purposes as it allows to test the new feature integration with the other parts of the application. This environment is prone to bug and can sometimes have some down time
- A production environment where the final users navigate and use the features developed. It is important to have a stable and bug less en-

environment so only carefully tested features can be deployed in this environment.

To achieve this goal, I used Terraform Workspaces. Workspaces in Terraform are simply independently managed state files. A workspace contains everything that Terraform needs to manage a given collection of infrastructure, and separate Workspaces function like completely separate working directories. It is possible to manage multiple environments with Workspaces. [16] In particular, I have implemented two different workspaces, the Default workspace and the production workspace. These two workspaces are essentially the same but use different variables, `default.tfvars` and `production.tfvars`.

In the next section, I'll explain how all the changes of the infrastructure are automatically managed by the infrastructure CICD.

5.4 CD for Infrastructure

Being conceptually very different from the other microservices, the Infrastructure repository is separated from the others described in chapter 4. For the same reason, the infrastructure needs a different pipeline for continuous deployment. Again, the behaviour of the pipeline is set in the `.gitlab-ci.yml` file. The pipeline has two steps: Terraform plan stage, Terraform apply stage. The Terraform plan stage is essential to setup the Terraform environment by starting the Terraform backend. In this stage, the Terraform Workspace is also chosen. The logic behind the workspace choice is the following:

- If the commit is made a code branch, the workspace is the default workspace
- If the commit is in the master branch, two different jobs are created,

the terraform:dev:plan and the terraform:prod:plan, each using the respective variables.

The Terraform plan stage goal is to retrieve information about the current state of the infrastructure and to show the user the future possible changes that the next stage would apply. By design choice, the terraform:dev:plan is always executed while being in the master branch, while the terraform:prod:plan must be executed manually from the Gitlab UI. The choice is made to make sure that a developer wants to push new code in production. An output file using the naming convention COMMIT_SHA.plan is produced to be reviewed and kept. A similar strategy is used for the other stage, the apply stage. The goal of this stage is to apply the new configuration to the infrastructure. Again, for terraform:dev:apply the procedure is automatic, while for terraform:prod:apply the developer must perform a manual action.

```

.job_template: &job_template
allow_failure: false
tags: [tooling]
image: hashicorp/terraform:$TERRAFORM_VERSION
before_script:
- terraform init
- terraform workspace select ${WORKSPACE_NAME}

.plan_template: &plan_template
<<: *job_template
resource_group: deployment
stage: plan
rules:
- if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
script:
- terraform plan -var-file ${VARFILE_NAME} -out ${CI_COMMIT_SHORT_SHA}.plan
artifacts:
  expire_in: 2h
  paths:
    - "${CI_COMMIT_SHORT_SHA}.plan"

.apply_template: &apply_template
<<: *job_template
resource_group: deployment
stage: apply
rules:
- if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
script:
- terraform apply -auto-approve ${CI_COMMIT_SHORT_SHA}.plan
- terraform output -raw kubeconfig > kubeconfig
artifacts:
  expire_in: 2h
  paths:
    - "kubeconfig"

stages:
- plan
- apply
- rollout
# You can use whatever you want as the name of
# and which does not contain spaces or weird ch
# Implicitly, this job will use the image `hash
terraform:dev:plan:
  variables:
    WORKSPACE_NAME: default
    VARFILE_NAME: default.tfvars
  <<: *plan_template
  when: always

terraform:prod:plan:
  variables:
    WORKSPACE_NAME: prod
    VARFILE_NAME: prod.tfvars
  <<: *plan_template
  when: manual

terraform:dev:apply:
  variables:
    WORKSPACE_NAME: default
  <<: *apply_template
  needs : ["terraform:dev:plan"]

terraform:prod:apply:
  variables:
    WORKSPACE_NAME: prod
  <<: *apply_template
  needs : ["terraform:prod:plan"]
  when: manual

```

Figure 5.5: CD for Infrastructure

There is also a third stage, the rollout stage. This stage, even if its technically part of the infrastructure pipeline, only gets triggered by the deployment of a new microservice, which I am going to describe in the next section.



Figure 5.6: **Pipeline for Infrastructure**

5.5 Deployment of a microservice

As anticipated in chapter 4, in this section I will describe how a microservice is deployed on the infrastructure. As stated before, there are two stages missing from image 4.4, the docker and the deploy stage. The **Docker** stage is the stage responsible for the docker actions needed to successfully deploy an image. In this stage, the code gets built into a docker image with a specific naming convention given by the name of the microservice and a tailor made image tag, which gets defined by the commit number to identify it. Then, after automatically login to the image registry, the image gets pushed to the aforementioned ACR, where it will be stored to be retrieved in the next stage. At this point, the image is successfully saved into the container registry, but the infrastructure is still not using the current image, since it needs to be updated from the previous one. To perform this action, the deploy stage performs an action which is called "kubernetes rollout". As stated before, this action is performed by the infrastructure pipeline, which, after being referenced by the deploy stage, triggers a restart of the newly pushed microservice and a fetch for the new image in the registry.

At the end, a check of the status is performed to confirm the success of the action.

```
.docker_build_template:
  dependencies: []
  stage: docker_build
  image: docker:20
  services:
    - name: docker:20-dind
      alias: docker
      command: ["--tls=false"]
  variables:
    DOCKER_HOST: tcp://docker:2375
    DOCKER_TLS_CERTDIR: ""
    DOCKER_DRIVER: overlay2
    IMAGE_BASE: ${REGISTRY_URL}/${CI_PROJECT_PATH}
  script:
    - docker build -t ${IMAGE_BASE}:${IMAGE_TAG} .
    - echo "${REGISTRY_PASSWORD}" | docker login --password-stdin -u ${REGISTRY_USER} ${REGISTRY_URL}
    - docker push ${IMAGE_BASE}:${IMAGE_TAG}
  rules:
    - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH

.deploy_template:
  dependencies: []
  image:
    name: bitnami/kubectl:latest
    entrypoint: [ '' ]
  stage: deploy
  before_script:
    - echo "$CURRENT_KUBECONFIG" > kubeconfig
  script:
    - kubectl -n default --kubeconfig kubeconfig rollout restart deploy/bo-monitoring-backend
    - kubectl -n default --kubeconfig kubeconfig rollout status deploy/bo-monitoring-backend
  rules:
    - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH

docker_build:dev:dockerimage:
  extends: .docker_build_template
  dependencies: ["maven-build:dev"]
  needs:
    - job: maven-build:dev
  variables:
    IMAGE_TAG: latest

docker_build:prod:dockerimage:
  extends: .docker_build_template
  dependencies: ["maven-build:prod"]
  needs:
    - job: maven-build:prod
  variables:
    IMAGE_TAG: prod

#####
##### DEPLOY #####
#####

deploy:dev:
  extends: .deploy_template
  needs:
    - job: docker_build:dev:dockerimage
  variables:
    CURRENT_KUBECONFIG: $DEV_KUBECONFIG

deploy:prod:
  extends: .deploy_template
  needs:
    - job: docker_build:prod:dockerimage
  variables:
    CURRENT_KUBECONFIG: $PROD_KUBECONFIG
```

Figure 5.7: CD for Backoffice backend

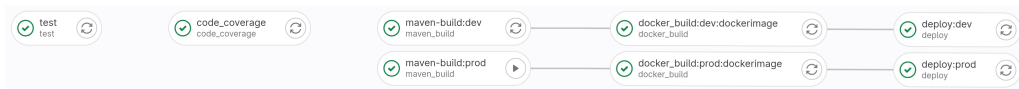


Figure 5.8: Full pipeline for Backoffice backend

6 | Conclusions

The goal of this project was to provide a solution for the task of developing an automated pipeline composed by an continuous integration and a continuous delivery parts in a enterprise environment. The proposed solution uses multiple industries well known methodologies and tools and uses them to develop a tailor made solution to follow the needs of a project developed from scratch. After studying and creating the design of the architecture, using the pipeline described in this work, the team was able to reach the goal of developing a fully functional web application hosted and deployed on Azure cloud. The application is currently used as a management tool by hundreds of employees every day, showing the solidity of the chosen architectural solutions. The solidity of solution adopted can be quickly and easily reproduced in future works expanding it to multiple and different clients, thanks to the success obtained by this project.

Bibliography

- [1] Atlassian, “Key devops principles.”
- [2] RedHat, “What is iac.”
- [3] HashiCorp, “What is terraform.”
- [4] RedHat, “What is kubernetes?.”
- [5] C. provider, “What is a cloud provider.”
- [6] Microsoft, “What is azure.”
- [7] Microsoft, “What is msal.”
- [8] T. Fernandez, “Release management for microservices.”
- [9] Atlassian, “What is continuous integration?.”
- [10] Microsoft, “Welcome to azure cosmos db.”
- [11] Microsoft, “Storage account overview.”
- [12] Microsoft, “Introduction to azure blob storage.”
- [13] CircleCI, “What is helm? a complete guide.”
- [14] Helm, “Helm charts.”

[15] Azure, “Azure provider.”

[16] A. Patel, “Terraform workspaces overview.”

Acknowledgements

With the conclusion of this work, my academic journey comes to an end. I would like to show appreciation towards my supervisor *Prof. Ferrari Carlo* for the help and the patience. I'd like to thank my company tutors *Massimo Gengarelli* and *Nicolas Launay* for the daily sustain both technical and personal they gave me during the internship program and *Virginie Prion* for giving me the opportunity to participate to this experience. A special thanks goes to my fellows interns: Alessandro, Amedeo, Charly, Elisa, Gregory, Maxime, Saurav, Sirine and Tommy. A special thanks to Alessandro, Sirine and Tommy, being the best team members one could desire.

I must thank my parents for the unconditional support and for belivieng in me during these sometimes rough years. Thanks Giorgia for being not only a partner but also a friend, family and a pillar during these long years we spent together in Padova. Thanks to my friends, Noemi, Alessandro, Memes group, life lasting friends and all the rest whom I am grateful to have had by my side all these years.

Thank you.

ITA

Voglio ringraziare i miei genitori per il supporto incondizionato e per aver creduto in me durante tutti questi, a volte complicati, anni. Grazie a Giorgia per essere non solo una partner, ma anche amica, famiglia e pilastro durante tutti questi anni che abbiamo passato assieme a Padova. Grazie ai miei amici, Noemi, Alessandro, gruppo Memes, amici di una vita e tutti gli altri. Vi sono grato per avervi avuti al mio fianco durante tutti questi anni. Grazie.

"A man's dream, will never end!"