POLITECNICO DI TORINO

Master's Degree in Computer Science

Master's Degree Thesis

Web App Development in Azure Cloud Environment and DevOps

Supervisors

Prof. Maurizio MORISIO

Prof. Louis JACHIET

Candidate

Alessandro MORINA

October 2022

Summary

Cloud-based technologies gained a lot of ground in the last decade. Major companies like Amadeus are migrating legacy codebases and architectures to an all-Cloud environment. A long and complicated operation for sure, but why are they doing that? Why is it so cost-effective to develop a cloud-based application? The answer is straight forward: companies prefer a cloud-based approach because they break away from all the tasks related to maintaining an online service at all time. They delegate these tasks to the companies that manages the cloud service. The benefits of these services, that include the assurance that your resource(s) will always be reachable, granted backup, and hardware properly maintained, usually outweigh the costs.

Figure 1: Cloud benefits [1]

In parallel, the DevOps (Development + Operations) methodology has also been introduced into the world of software development. It is a methodology, or philosophy for some, adopted by development teams to ensure excellent product quality, frequent releases and constant feedback from clients. The key word in DevOps methodology is automation. In later sections we will talk in more detail about the meaning of the word automation in our context, the tools needed, and the overall process (figure 2).

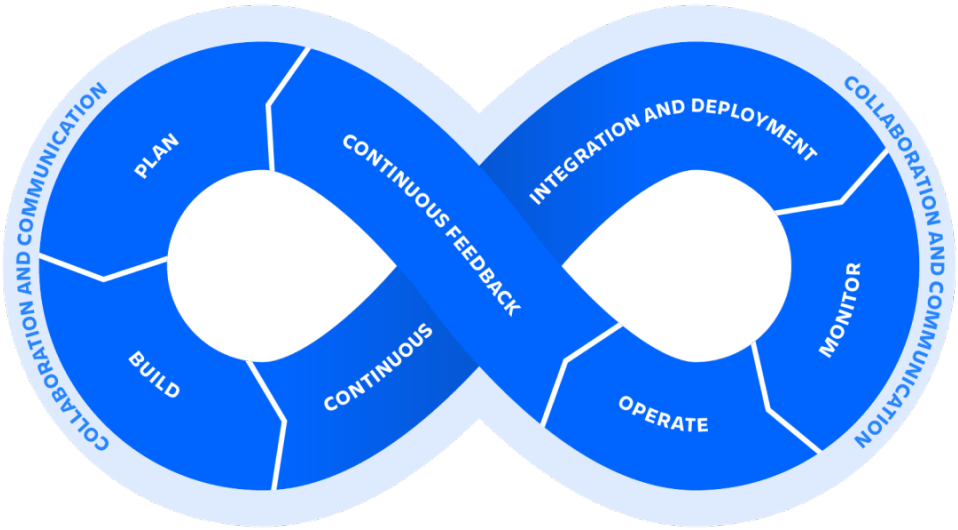

Figure 2: DevOps development cycle [2]

We will see how to get the best out of these two worlds, Cloud development and DevOps, through the study of a complete web application development.

Acknowledgements

My first thank you goes to Alten, for making me a part of this project, and in particular to Virginie, Massimo and Nicolas, who have accompanied and guided me all along the way.

Another big thank you goes to my fellow interns and friends.

Amedeo Sarpa
Charly Ducroq
Elisa Alfieri
Grégory Nardi
Maxime Balouard
Sandy Pivato
Saurav Bhatia
Sirine Kraiem
Tommy Lecourt

A special thanks to Tommy who helped me in dark times.

Table of Contents

List of Figures

Chapter 1

Context & Objectives

1.1 Internship context

Developing a web app from scratch is not an easy task.
The process starts with understanding what the client is asking for, analyzing the technical possibilities to make the product, and discussing with the client about them and, in some cases, it is needed to find compromise, or points of agreement, to actually build the requested product with the economic and time resources available.

During my internship at Alten GROUP [3], in the Sophia-Antipolis office, I worked on an internal project, in a team composed solely of interns.
The purpose of the internship was as much to learn as it was to build a web app that would replace an existing (old) one, but in a modern key and with a better design in some poorly developed aspect.

Being all interns (10 in total), with the most varied skills, the undertaking was not easy. It all started with a "cahier des charges", a document containing all the functional specifications of the desired application. It should be pointed out that it was forbidden for us, for the duration of the internship period, to peek at or take cues from the already existing application.

The journey began with an in-depth study of the cahier des charges, in parallel with the drafting of the C4 model, analyzing customer journey and discussing the technologies we would need to best meet the requirements.

We always had technical experts and senior developers to ask for advice or for suggestions about blocking points, but every final design and development decision was made by us, as the only ones responsible for the final product.

The app in question, as I mentioned earlier, is intended to replace an old one internal to the Alten's division of Sophia-Antipolis, but there are also plans to be able to export it (with a subscription) to other Alten divisions around the world.

Our internship lasted 6 months, but the total development time planned is at least 2 years. We were the first round of interns, the ones who laid the foundation (and more) of the application that from now on, we will call Miami v2.

1.2 Practical and theoretical objectives

As mentioned earlier, the internship goal is dual: to learn and to develop a real and functioning product while doing so.

On the learning side I mean about the technical aspects, that is learning about a range of technologies and development environments, such as front-end, back-end and cloud technologies; as well as on the team workflow aspect, such as Agile Scrum and DevOps deployment methodology. I will talk in detail about the benefits and impact of these methodologies on software development in the next chapters. As a side note it is worth saying that during the stage period we all tried out each role in a Scrum team, such as: developer, QA engineer, Scrum Master and Product Owner.

And of course it was expected that at the end of it, the team would be able to present to the customer an MVP (Minimum Valuable Product),

which means a working product with a set of minimum functionalities required.

1.3 Scrum

Before talking about our application, a few words about the Scrum [4] framework, with which our team worked, must be said.
Scrum is a framework that helps teams and organizations to handle complex problems and deliver high-quality products.

In other words, Scrum is a set of practices or rules that are made to allow the team to self-organize, in order to deal with unexpected and complex problems. On scrum.org [4], it is described as a heuristic algorithm, with respect for people.

1.3.1 Scrum Team

Talking about people, let's spend a few words on the principal roles in a Scrum Team.

- The **Product Owner** is the main interface to the client. They understand and then transfer their knowledge to the team in the form of User Stories (behaviour of a User in relation to the product. Each User Story could include several tasks, created by the Product Owner (PO), or by the developers them-selves), in a set called Product Backlog. Usually the PO is a full-time role.
- The **Scrum Master** is the point of reference in the Scrum Team. When there is a problem of any kind, a member to the team has to talk with the Scrum Master first. They are also responsible for directing the Scrum Ceremonies and for making sure that all team members have understood and are following the rules and principles of Scrum. Sometimes the Scrum Master could also be a Developer.
- The **Developers** is a general name that could refer to various kinds of developer. In a Scrum team it is whoever is taking part

to the development process of the product (Software engineer, QA engineer, ecc...).

1.3.2 Scrum Ceremonies

Scrum Ceremonies are a fundamental parts of the Scrum Team workflow. They are needed to create regularity and minimize meeting times.
They are key to keep the team efficient, motivated, and with a clear vision toward the goal.
The following Ceremonies live around the concept of Sprint, which is nothing more than a definition of time in which the Scrum Team makes a commitment to fulfill a predetermined set of tasks. A Sprint duration may vary between 1 and 4 weeks. After a Sprint ends, a new one begins.

- The **Daily Meeting** is a very short meeting done every morning or every evening, in order to flatten out the knowledge of the progresses of each member and more in general of all the team during the current Sprint, It is also the moment to talk about problems and seek for suggestions. If the problems seem too big, it will be necessary a dedicated meeting.
- During the **Sprint Planning** the Product Owner explains the User Stories (US) to the members of the team. Together they decide a set of US that the team can complete for the next Sprint.
- The **Sprint Review** is the presentation of the achievement the team accomplished during the Sprint, to the Client. In other words, the features they have been able to implement or problems to fix. During the Sprint Review they can receive immediate feed-back from the Client. This is one of the strong points of Scrum, that works very well with DevOps methodology.
- During the **Sprint Retrospective** the team members are free to talk about what they think went well and what could be improved. It's a meeting dedicated to discuss solutions to increase efficiency, effectiveness and overall quality.

1.3.3 Scrum Workflow

The following scheme (figure 1.1), represent the workflow in a Scrum team.

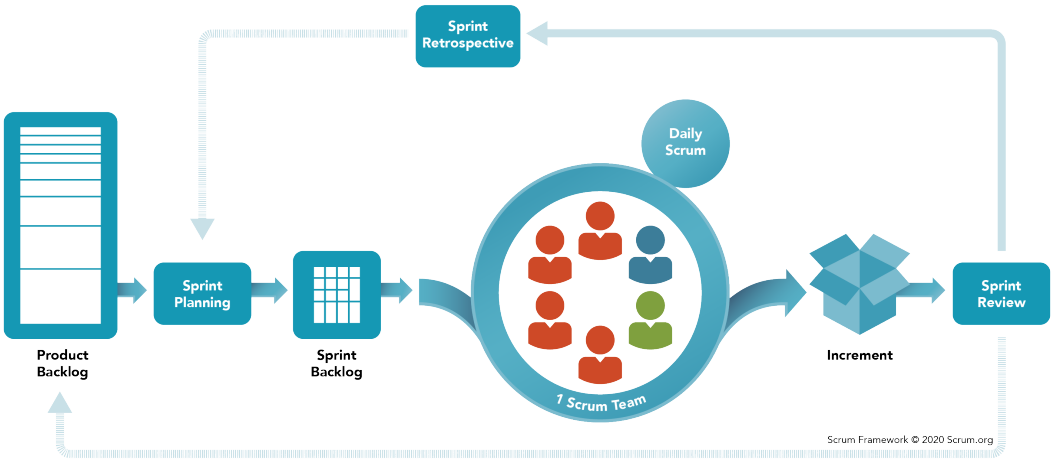

Figure 1.1: Scrum in action

From left to right we begin with the Product Backlog, that is the set of requirements (User Stories) created by the Product Owner. Then we proceed to the right with the Spring Planning in which, as i mentioned above, the team gathers to discuss about which User Story (taken from the Product Backlog) they will be able to complete during the next Sprint, and so forming the Sprint Backlog.
Everyday the team will do the Daily Meeting, referring to the Sprint Backlog as a point of reference of the progress of the whole team. Each team member, with the help of tools like Jira, will be able to consult the Product and Sprint Backlog, in order to know which tasks the other members are doing and which task is free to be taken.
After the duration of the whole Sprint, the team developed a certain number of features, that can be called Increment. The Increments are presented to the Client during the Sprint Review.
Usually right after the review, the team does the Sprint Retrospective to discuss about general improvement that could be made in the team

to increase quality of work environment and work delivered.
Sometimes new information arise from the Sprint Review (for example sometimes Clients change their mind), and so the Product Backlog has to be updated.
Finally a new cycle begins with the Sprint Planning.

1.3.4 Scrum in our Internship

As I said previously, one of the goal of our internship was to make us truly understand what it meant to work in such an environment. At the beginning of the internship we had a 2 days course lectured by an internal expert of Alten, and then, to make the concept stick, each one of us had to *play* each role. Of course the environment was not like in the real world. We could make mistake and ask questions, but it really made us ready to work in a real company, in a similar environment.

1.4 Miami v2

Alten is an engineering consulting firm. It makes the expertise of its Consultants available to various Clients, whether for existing projects, or for planning and design of new ones. Without getting lost in the details, the link between Consultants and Clients is managed by Business Managers (BMs).
Every user is part of a Tenant, that is a physical division (building) of Alten GROUP.

Our application will be an internal web portal with multiple features in function of the role of the user logged in.
In the future version of Miami v2 this could change, but so far we considered the following:

- **Consultants** have access to their own profile and to Alten Grains functionalities.
- **BMs** will have access to Consultants details, Clients details, and can manage requests for Alten Grains or Rewards of a Consultant. They can also see statistics related to their Tenant.
- **CMC** same as the BM, but they manage the kind of Activity a Consultant can do to get Alten Grains, and the kind of Rewards a Consultant can buy with Alten Grains.
- **Tenant Admin** have the same feature as a CMC, and can also create users and add them to their Tenant.
- **HR** can create/modify/delete Consultants profiles.
- **System Admin** is the main role of the Back-Office portal.

1.4.1 Front-Office vs Back-Office

As it is clear from the description of the System Administrator role, the team had to develop in practice 2 web portals. A Back-Office one, accessible by System Admin and Tenant Admin, and a Front-Office one, accessible by everyone else, including the Tenant Admin.

The Back-Office web app is a portal in which a System Administrator can manage the creation, deletion and modification of the Tenant entities; He or she can assign to other users the System Administrator and Tenant Administrator role, and can monitor and observe statistics about tenants. A Tenant Admin have access to the Back-Office platform but in read-only mode, in order to monitor and consult tenant statistics.

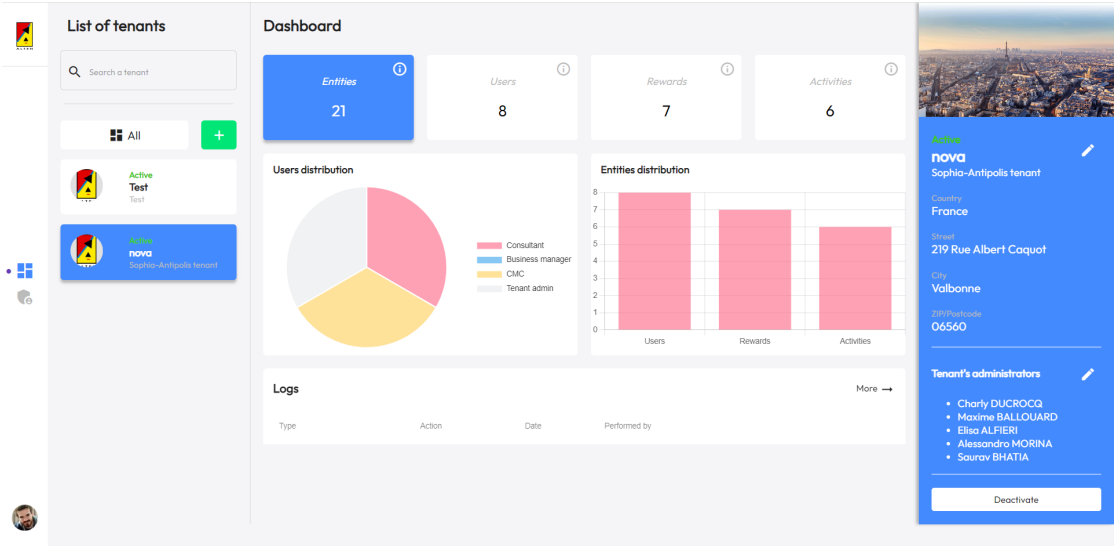

Figure 1.2: Back-Office homepage

The Front-Office app is instead, as I described briefly earlier, a suit of features, in function of the role.
Broadly:

- Customizable dashboard with widgets
- Users/Clients management
- Alten Grains System
- Statistics
- Notification System

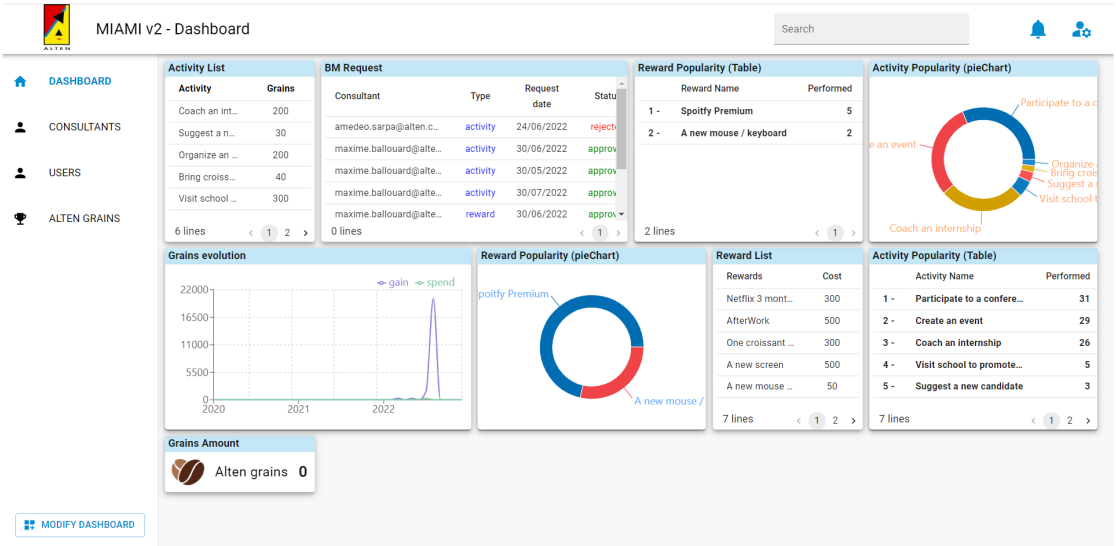

Figure 1.3: Front-Office dashboard

1.4.2 Alten Grains

It is worth to have a subsection dedicated to the Alten Grains feature, since it is a whole new system that was not present in the old version of Miami.
It is an activity-reward sub-portal, where Consultants will be able to declare to have done/organized a particular kind of Activity, and consequently, after verification by a BM or CMC, they would gain a certain amount of a virtual currency: the Alten Grains.

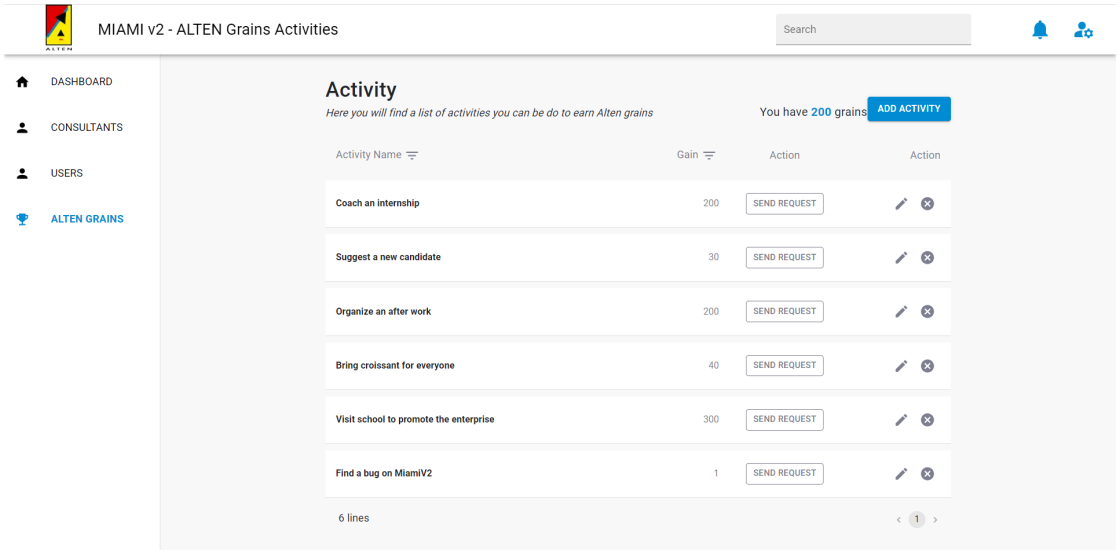

Figure 1.4: Alten Grains Activities page

The Consultants will then be able to spend Alten Grains in order to buy Rewards in the designated area (like an online shop).

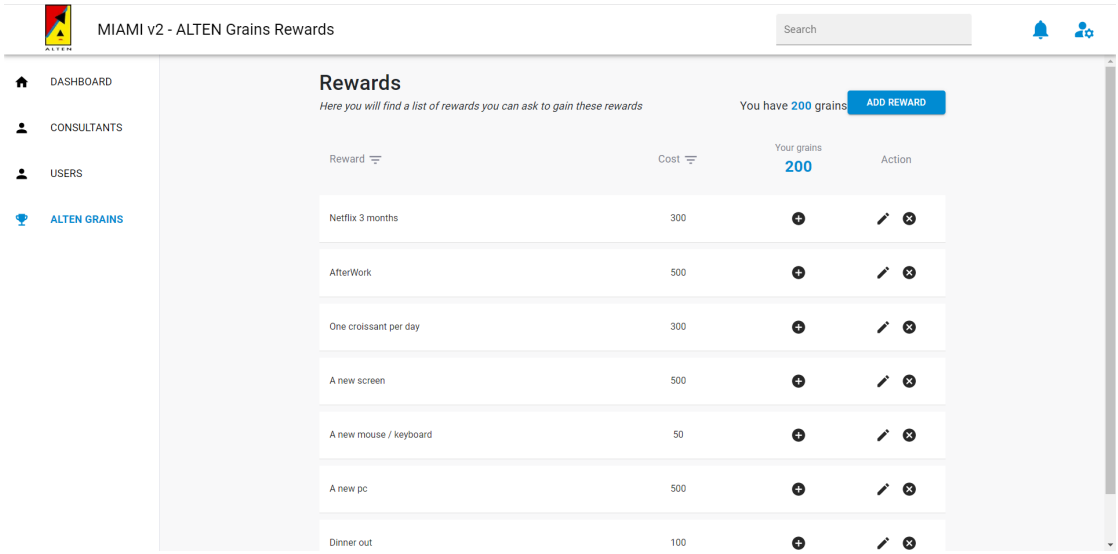

Figure 1.5: Alten Grains Rewards page

The elements in the pictures are mocked. The pictures are taken from the development environment.

From the point of view of a BM/CMC/Tenant Admin instead, we can also accept or deny the requests for Alten Grains or rewards, or we can directly send to them Alten Grains (this operation may seem dangerous, but all the transactions are tracked and monitored).

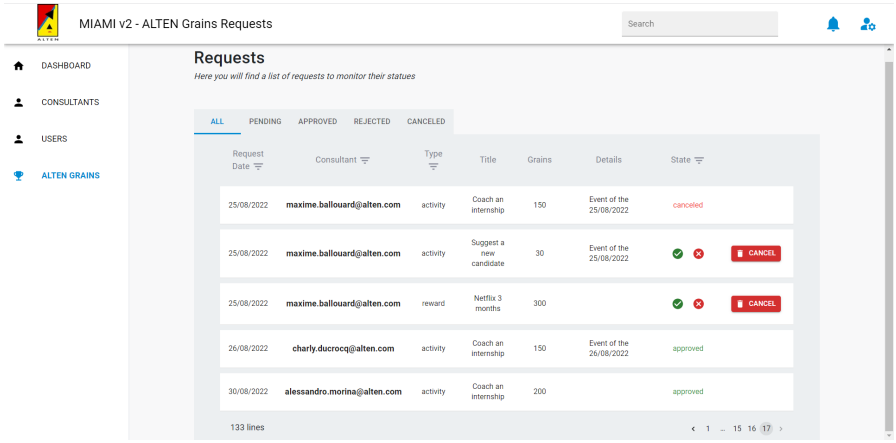

Figure 1.6: Alten Grains Requesrs page

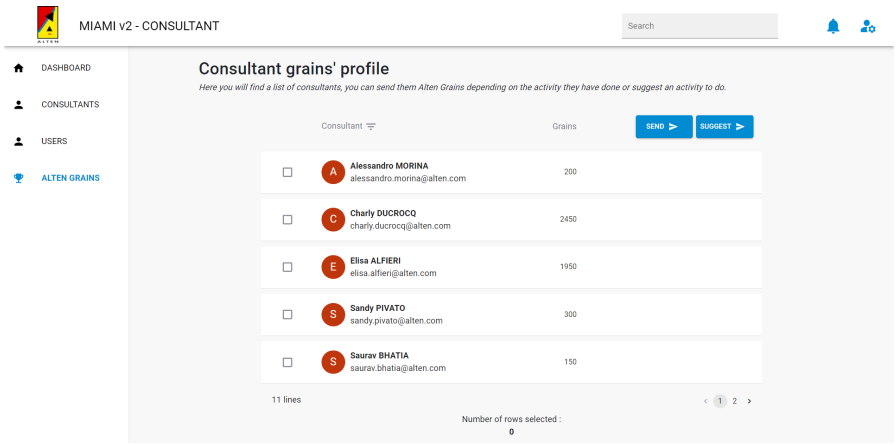

Figure 1.7: Send AG to Consultants or suggest them activities

1.4.3 MVP

The MVP requirements for our internship were:

- **Widget dashboard on Front-Office**
  Each user is provided with a customizable main dashboard, on the Front-Office portal.
- **Alten Grains on Front-Office**
  Alten Grains portal with all features present and working. Ability to send requests to BMs as a Consultant, to receive Alten Grains, or to spend them and get Rewards. Ability for BMs and CMCs to edit possible Activities to be performed and Rewards to be obtained, and acceptance or rejection of Requests.
- **Feature filtered by role**
  The monolithic portals (FO, BO) offer different functionalities depending on the role of the logged-in user.
- **Back-Office portal entirely working**
  The BO portal provides all functionality regarding Tenant management, System Admin and Tenant Adimn roles.
- **Deployment of production environment**
  The prod environment must be configured and accessible by all concerned users.

Chapter 2

DevOps methodology

2.1 Key concept: Automation

As it is probably clear from the title of this section, the key concept about the DevOps methodology is Automation.
But Automation of what exactly? Let's take a step back.
Taking again a look at the DevOps cycle we can notice how it is a fusion between two worlds.

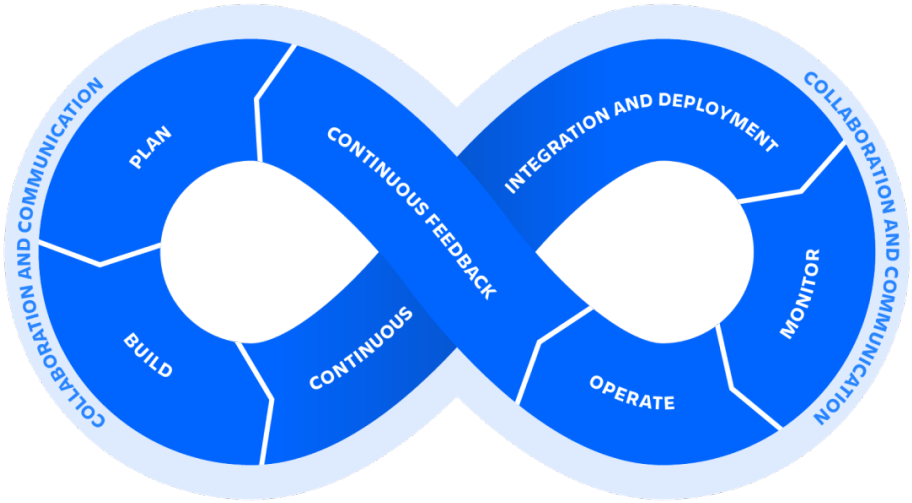

Figure 2.1: DevOps development cycle [2]

The left side of the figure 2.1 represents the Development world, that is concerned about the design of the software architecture and the actual coding, testing and building of the product. While the other side (the Operation side) is concerned about the deployment on the machines (as servers); their main tasks are to assure that everything works fine on the deployment environment and then monitor how the new version of the software performs.
In an old-school development environment these would have been two different teams, and the communication between these two was usually a critical point.

In DevOps we no longer have this division; developers are responsible for the steps of the development process, up to and including deployment. This is achieved through tools, which allow the automation of tasks such as Verification (Tests, Coverage), Build, Deployment, Operation and Monitoring.
This is why automation is the main concept of DevOps.
We can speak of DevOps only when the entire process is managed by an orchestrator tool that manages, like a pipeline, all the steps of the cycle.
This cycle is called Continuous Integration / Continuous Development.

Now, why there was the need of that revolution?
The benefits are many:

- **Frequency of updates:** thanks to the automated nature of deployment, the modification and updates to the code can be delivered with much more frequency in small portions.
- **Rapidity:** due to the feed-back after the frequent updates, it is easier and faster to fix eventual bugs or to apply modifications. This leads to the next advantage.
- **Quality:** the code benefits of an increased overall quality, in terms of performance, and code quality.
- **Scalability:** thanks to some Operation step tools, as Kubernetes, it is easy to scale the physical resources in function of the needs.

2.2 DevOps steps & technologies

In this section I will talk about the tools and technologies that have enabled us to apply the DevOps methodology, for each step of the cycle. I will spend a few words introducing the meaning of each step, and how we handled it in our internship.
I will also use some portions of code taken from our project to give concrete examples.

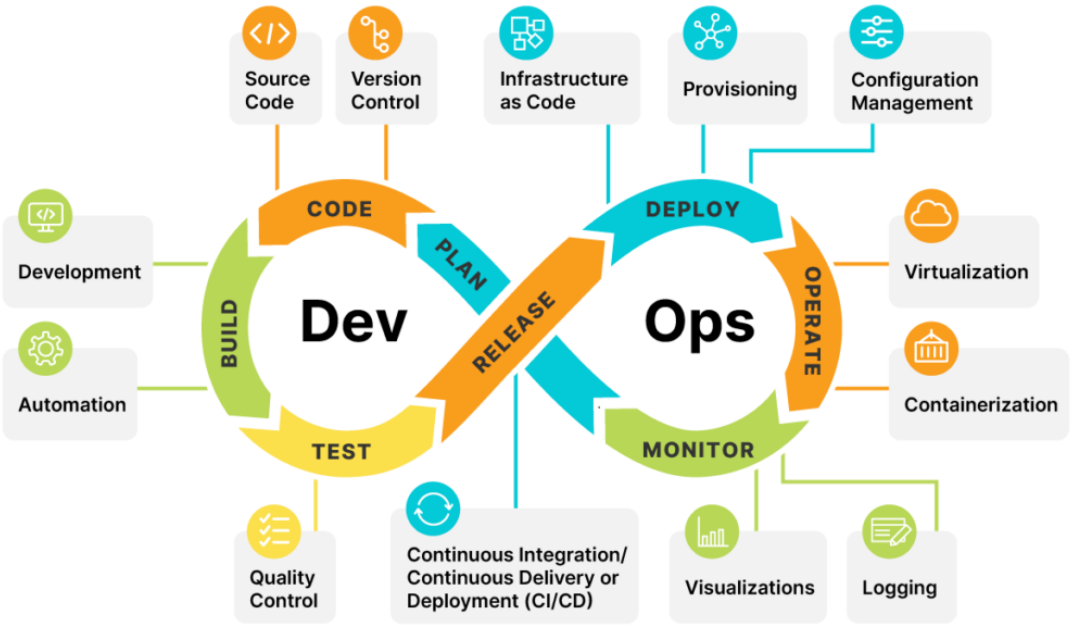

Figure 2.2: DevOps tools description [5]

2.2.1 Plan

The first phase is *planning*.
No specific technologies are needed in this phase, although it is now almost essential to use tools such as Jira [6], which help team members keep track of what everyone else (and themselves) is doing.

Depending on the point of view, various roles are responsible for this phase. From a macro perspective, it is the product owner who is

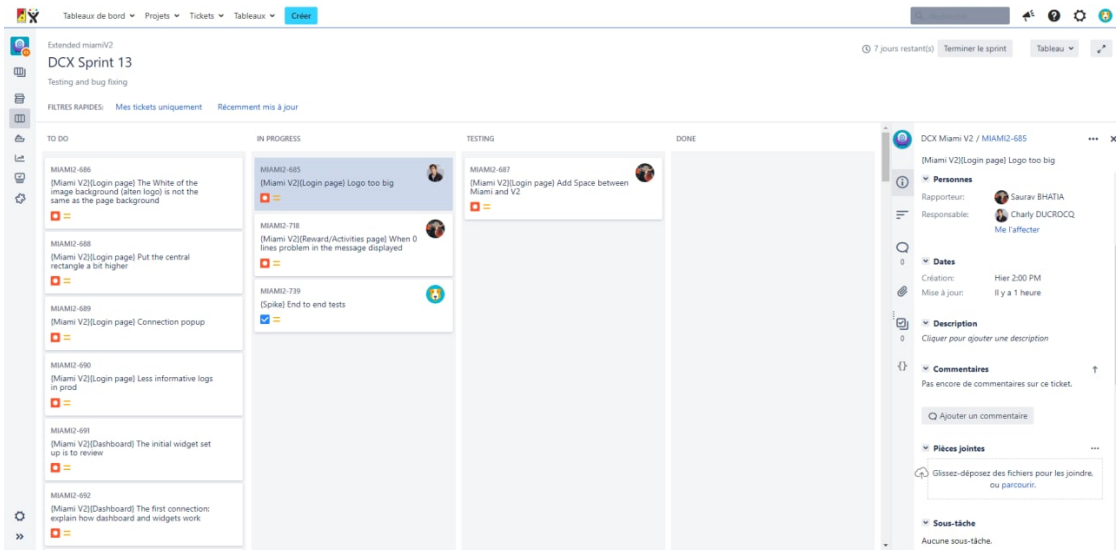

Figure 2.3: Jira board Miami v2 Front-Office

responsible in understanding the specifications from the clients, and organizing them into a set of User Stories (also called Product Backlog, as explained in this section). In doing so, a sort of road-map to specific goals can be created by sorting the US in order of priority.

At the same time, the developers have a say in discussing what technologies/frameworks/data structures/architecture would be optimal to use in order to best meet the requirements described by the OP. All of this needs to be done in an initial planning session, but the possibility of changes during future planning phases is not entirely ruled out, although one must be aware that making changes on the technical side at a later stage of development can be extremely costly, which is why devoting time and energy to the initial planning phase is essential.

On the micro level, however, the planning phase is represented by the daily meetings that, in a scrum team, are under responsibility of the scrum master (described previously here), in which he or she must ensure that all elements of the team know what to do, and do it as efficiently as possible.

2.2.2 Code

The *code* step is about the classic operation of coding, which includes not only writing code on an IDE, but also maintaining it, organizing it, and structuring it in a way that allows a whole team of developers to get their hands on the same codebase at the same time, and be able to make it easily accessible to any new entry to said team.

The softwares involved in this process are development environments (IDEs), extensions to enforce a common code style for developers and others to enforce the avoidance of poor programming practices and, finally, a versioning control system, to allow all developers to program easily on the same software, to keep track of old versions and thus always having the ability to roll back.

During our internship we used different languages: TypeScript, Java, and C#, but our main IDE has always been VSCode with the appropriate extensions and prettier (for code style consistency). We also used IntelliJ for the Java language.
As for versioning control, we used Git, offered through the DevOps platform GitLab (central tool in our whole DevOps cycle).

2.2.3 Build

The *build* operation is a fairly simple concept: a software build transforms some source code into binaries, ready to be used by the end user or, as in our case, to be deployed in a cloud environment, in order to allow access to users via a Web portal.

The *build* phase, along with the *test* phase, are considered one after the other in our pipeline, implemented through the GitLab CI/CD system, which I will explain in detail in the *release* phase.

In the *build* step we used the Apache Maven tool for Java and the Node Package Management (npm) for all the TypeScript codebases (Front-Office frontend/backend and Back-Office frontend).

2.2.4 Test

One of the most important phases in our DevOps cycle, but also in the software development world in general, is the *testing* phase. This phase is responsible for verifying the proper behavior of software features, to the identification of problems (bugs) and also as an assurance to the clients that the product is reliable and of high quality. It is worth noting that testing is not always done correctly, thus undermining the assurances that this phase is supposed to bring to the team and customers; therefore, it is important that the tests are at least checked by senior developers to ensure that they are correct and robust.

This phase, depending on the point of view, is performed either manually by developers and external users, ensuring that features are working properly through experimental use of the software, or automatically, by software that perform a series of coded tests (e.g. Unit Tests), and can thus be executed during the pipeline. In our case, each code-base has a Testing step during the release process, and if even one of the tests fails, the pipeline is stopped.
This indicates that something unexpected occurred and/or a feature has been compromised through a change, and that it no longer meets the required specification (or in simple terms no longer works properly).

Benefits

I have said previously that testing is important, but I have not yet explained why.
Let's take a look at the most important benefits that effective testing can bring to the team, to the product, and even to the customers.

- **It helps saving money**
  As previously announced, correcting errors is expensive, depending on the error, even very expensive. Spending valuable time to create and perform tests is never a waste.
  Testing is a long-term investment; it helps identify problems at an early stage of development, thus avoiding spending money to correct errors, or spending even more money to compensate third

parties who have been "victims" of these errors.

- **Quality**
  The product is considered to be of quality when it meets all the customer's requirements, and does so correctly and without presenting problems.
  Testing helps to deliver fully functioning software to the clients.
- **Customer satisfaction**
  The practice of testing makes the customer confident that the product is working properly, making them sure that the product will offer an optimal user experience.
- **Security**
  Testing can be oriented toward identifying vulnerabilities. Software security is now a critical issue in the modern world, and software that is well tested against vulnerabilities is a major goal of many teams.
- **Speed up the development process**
  Automated testing can help speed up the development process incredibly.
  With automated tests, it is much easier to identify the portions of code that are guilty of unexpected behavior, thus making the debugging process almost immediate. They also help develop new features without the risk of compromising old ones, because good tests always verify that what exists, continues to work as usual.

Types of tests

So far I have only talked about the importance and the benefits, and I recognize I was a little confusing, because some benefits are consequences of certain types of tests, and some from other types.

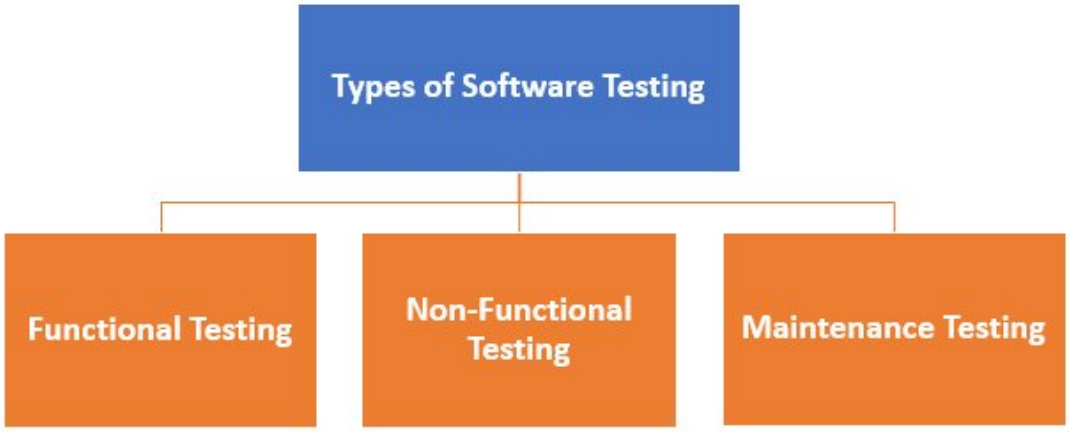

Figure 2.4: Test types [7]

In the figure 2.4, we can distinguish 3 main categories:

- **Functional**
  Functional tests are those responsible for the actual functionality of the product. The tests verify that there are no bugs and that all behaviors are consistent with the specification.
  E.g., Unit Test, Integration Test, User Acceptance Testing
- **Non-Functional**
  Non-Functional tests are responsible for features such as product portability, reliability, and security. Features that, during our internship period, were taken into consideration but were not tested.
- **Maintenance**
  This type of testing is done after the release of the product, to add a level of assurance that everything is working properly after updates or migration of the product. E.g, Confirmation Test, Regression Test

During our internship, we focused more on functional testing, in particular:

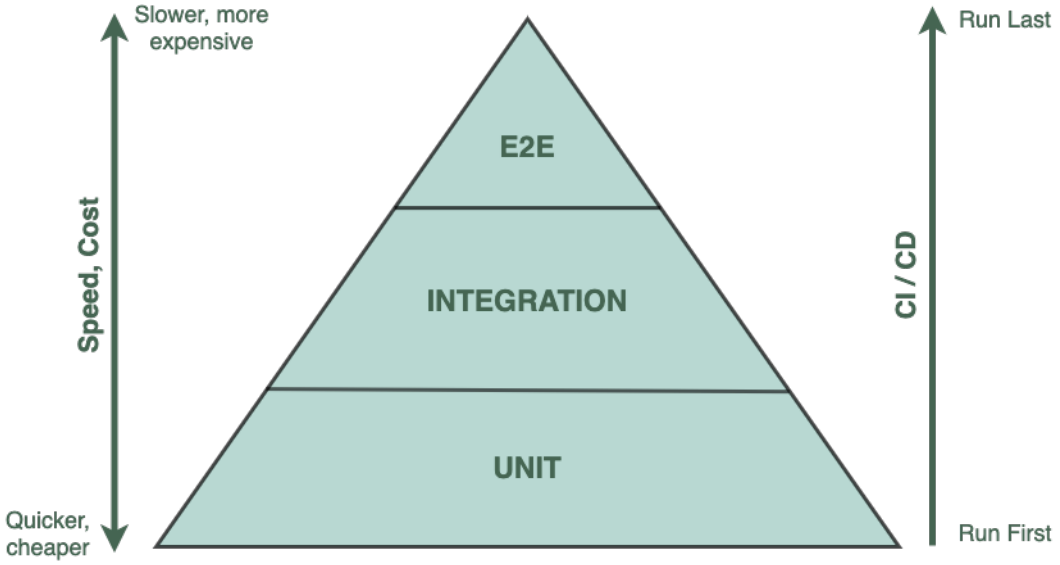

Figure 2.5: Main functional tests [8]

- **Unit Test**
  The Unit Test (UT) is the most basic of the tests.
  The task of a UT is to verify the correctness of a basic component. By basic component I mean a portion of code that does not depend on external systems, for example a function that performs a simple calculation or that calls different functions in function of the parameter. I just described the two general cases of UT: output-based testing and behaviour-based testing. In the former you test a particular expected output of a function, while in the latter you check that other functions (unit tested with the former scenario), get called under specific conditions.

  They are the easiest and fastest to code, and it is good practice to create unit tests for most of the components, even for those that at first glance may seem "too simple not to work as expected".

- **Integration Test**
  Integration Testing is a bit more complicated than UTs, as they are

responsible for verifying the correct operation of multiple modules or external components, which communicate with each other. E.g., When a particular backend end-point is called, it creates a new element in a database.
Integration Tests usually interacts with the real instances of the external systems, and not stubs or mocks of them.

- **End-to-end Test**
  End-to-end (E2E) tests are the most complete, but also the most complex and slow to perform. They are a simulation of a real user behavior, which then triggers the functionalities of the entire stack of technologies that may be involved in the product.
  For example, a simple E2E test could be used in the following scenario: in a site where there is a section in which a user can create elements, that then are stored in a database and consequently shown on the UI, we can simulate the click on the "create" button, and then check if the elements appears in the designated UI zone. In this example there is a communication between the frontend system and the backend, between the backend and the database, and then the other way around.
  This guarantees that the whole communication works correctly, in an environment similar (or identical) to the real one.

Tests in our internship

In our project we used various frameworks for the 2 main languages (TypeScript and Java).

- **Jest [9] & Karma [10]**
  Jest is one of the most popular testing frameworks for TypeScript, being easy to use, offering a smart engine for parallel test execution and a great mocking system.

  Consider the following example code taken from our Back-Office frontend.

  This function is used to update a value of an internal list in a

```
74    applyFilter() {
75      // if filter for tenants is empty, just assign the whole array
76      if (!this.tenantsFilter) {
77        this.tenantsFiltered = this.tenants.slice();
78        return;
79      }
80
81      // else, apply filter (letter case is ignored)
82      this.tenantsFiltered = this.tenants.filter((e) =>
83        e.name.toLowerCase().includes(this.tenantsFilter.toLowerCase())
84      );
85    }
```

Figure 2.6: applyFilter function BO frontend

component, in function of a filter inside another variable.

The following piece of code is a test for the applyFilter function.

```
78  it('tenantsFiltered should change when applyFilter is called and tenantsFilter is not falsy', () => {
79    const mockTenants: Tenant[] = [
80      { id: 0, name: 'Sophia', description: 'A', inactive: false, logo: '' },
81      { id: 1, name: 'Paris', description: 'B', inactive: false, logo: '' },
82      { id: 2, name: 'Padova', description: 'C', inactive: false, logo: '' },
83      { id: 3, name: 'Rapallo', description: 'D', inactive: false, logo: '' },
84    ];
85
86    const mockTenantsExpected: Tenant[] = [
87      { id: 1, name: 'Paris', description: 'B', inactive: false, logo: '' },
88      { id: 2, name: 'Padova', description: 'C', inactive: false, logo: '' },
89      { id: 3, name: 'Rapallo', description: 'D', inactive: false, logo: '' },
90    ];
91
92    component.tenants = mockTenants;
93
94    component.tenantsFilter = 'pa';
95    component.applyFilter();
96    expect(component.tenantsFiltered).toEqual(mockTenantsExpected);
97
98    component.tenantsFilter = 'PA';
99    component.applyFilter();
100   expect(component.tenantsFiltered).toEqual(mockTenantsExpected);
101 });
```

Figure 2.7: Test for the applyFilter function

Notice how this is an output-based unit test, because we are manually assigning a value to evaluate [line 92 in image 2.7], along with

filter values [lines 94 and 98]. Consequently, we call the component's internal function [lines 95 and 99], which is the one we want to test, and finally we verify, thanks to the Jest **expect** syntax, that the value computed by the applyFilter function (which we know is assigned to the tenantsFiltered variable), is exactly what we expect (comparing it with mockedTenantsExpected, through the toEqual method).

Karma instead is a framework that executes test in relation to web browsers in real-time. It is used to verify that all tests pass successfully, regardless of the browser on which the code is run.

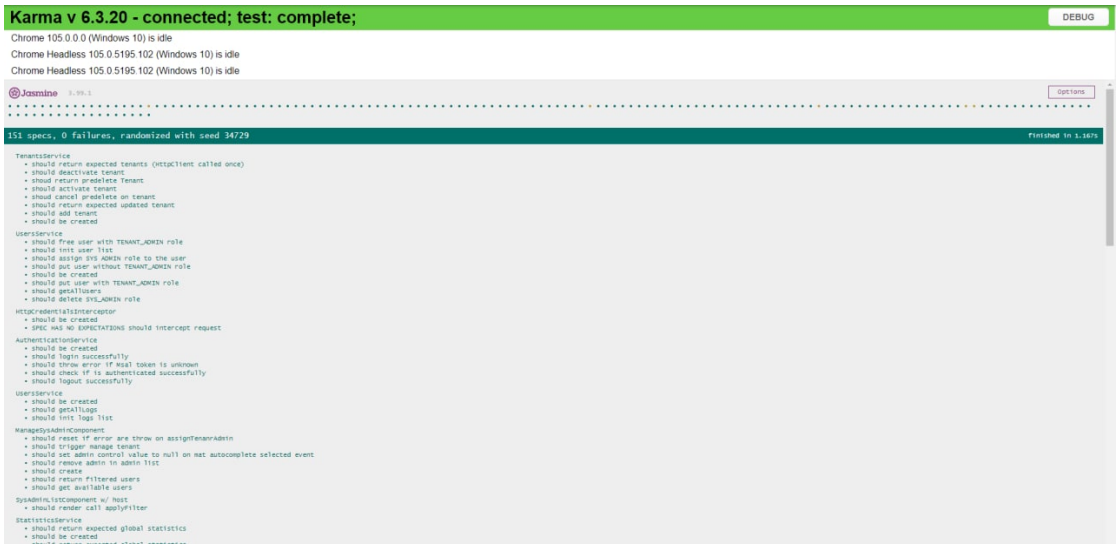

Figure 2.8: Karma interface

- **JUnit [11] & Mockito [12]**
  JUnit in turn is one of the most popular frameworks for unit testing in the Java environment. We also leveraged on the Mockito framework in parallel, to mock external elements that would be difficult to run in a UT environment.

```java
/**
 * POST request who add and return a new tenant.
 *
 * @param newTenant the new tenant to add according to this model: {@link com.alten.havana.model.Tenant}
 * @param file the logo associated to the tenant
 * @param authentication the current user's information
 * @return the new added tenant
 * @throws IOException the exception throw if a problem occurred with the blob storage
 */
@PostMapping
public Tenant newTenant(@RequestPart("tenant") @Valid Tenant newTenant,
                        @RequestPart(value = "file", required = false) MultipartFile file,
                        Authentication authentication) throws IOException {
    if (file != null) {
        String blobUrl = blobService.uploadBlob(newTenant.getName() + "logo", file);
        newTenant.setLogo(blobUrl);
    }

    String description = String.format("Creation of a new tenant %s", newTenant.getName());
    createLog(method: "POST", type: "action", description, authentication,newTenant.getName());
    return repository.save(newTenant);
}
```

Figure 2.9: Create new Tenant endpoint BO backend

The above code represents an endpoint. It communicates with several external systems (Azure Queue Storage, Blob Storage, Database), and yet we need to create a UT for it. The answer lies in Mockito.

```java
@MockBean
TenantRepository tenantRepository;
@MockBean
RestTemplate restTemplate;
@MockBean
BlobContainerClient container;
@MockBean
BlobService blobService;
@MockBean
Authentication authentication;
@MockBean
QueueClient queueClient;
@MockBean
QueueService queueService;
```

Figure 2.10: Mocks for BO backend

Without getting lost in the technical structure of the BO backend,

it is worth noticing that all the tests are grouped in relation to a specific XYZController, in a class called XYZControllerTest.

A Controller, in a backend context, is a set of endpoints (also called API), which can be accessed by an external request, like a frontend.

In each Controller test class are defined, through Mockito, mocked object of external system (figure 2.10), as the *tenantRepository* which represents the Database connection.

```
89      @Test
90      public void addNewTenant_success() throws Exception {
91
92          Tenant tenant = new Tenant(name: "Milan", description: "test");
93
94          when(tenantRepository.save(tenant)).thenReturn(tenant);
95          MockMultipartFile jsonFile = new MockMultipartFile(name: "tenant", originalFilename: "", conte
96          MockMultipartFile logoFile = new MockMultipartFile(name: "file", originalFilename: "logo.png",
97
98          when(blobService.uploadBlob(tenant.getName() + "logo", logoFile)).thenReturn(value: "url");
99          Mockito.doNothing().when(queueService).addQueueMessage(messageText: "test log message ");
100
101         mockMvc.perform(MockMvcRequestBuilders.multipart(urlTemplate: "/api/v1/tenants")
102                         .file(jsonFile)
103                         .file(logoFile)
104                         .principal(testingAuthenticationToken))
105                         .andExpect(status().isOk());
106     }
```

Figure 2.11: Test for addNewTenant_success BO backend

With the keyword **when** [line 95 of figure 2.11], we can tell the program to not execute the real behaviour of the external system, but instead return a mocked response, that we define with the method **thenReturn**.
We can also force some components to do nothing when they are called [line 99], because their execution does not affect the behavior of the portion of code we intend to test.
Finally, we simulate the call to the endpoint [line 101], and we then check the expected value (status only in this case).

- **Selenium [13]**
  Selenium is a software that we unfortunately only saw in theory during a little course taught by an experienced QA Engineer from Alten at the beginning of our internship, however, it will be used

to implement E2E testing in a future cycle of the internship. Selenium is a software for task automation in web-browsers, and one of its uses is to simulate behaviors similar to those of a real user, and then verify a browser state. Doing so we can indeed implement E2E testing. Selenium can also be integrated into a pipeline, since it creates a virtual instance of a browser to perform its operations.

Coverage

The last topic regarding the world of testing is that of coverage. Coverage is a percentage value, which indicates the amount of code that is "covered" by testing. Usually a development team will set a minimum coverage value, which indicates a kind of quality assurance, as it is usually around 95 percent.

In our case for instance, we have 100% coverage on the frontend, and 76% in the backend, regarding the Back-Office

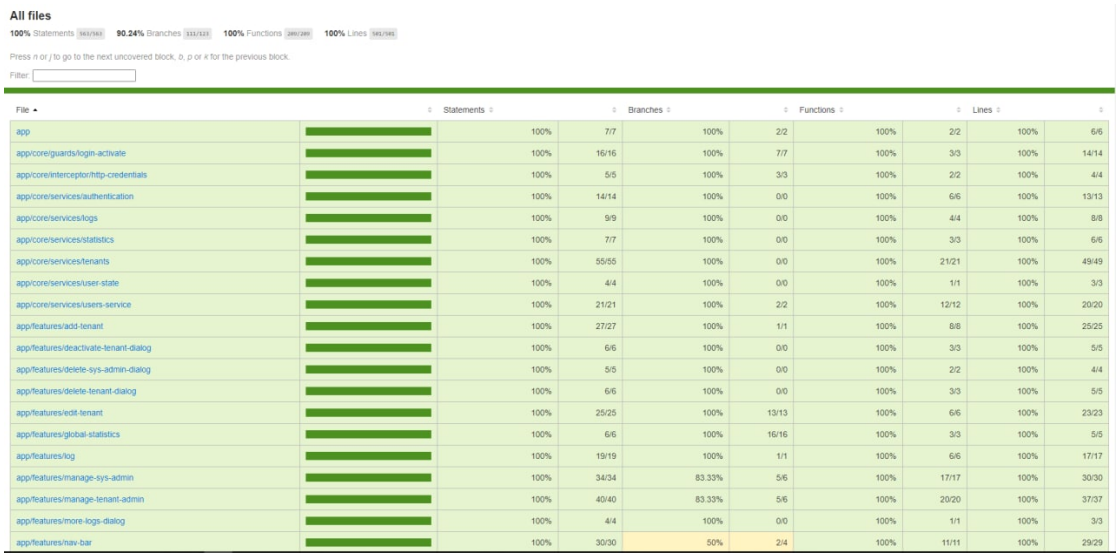

Figure 2.12: Coverage 100% BO frontend

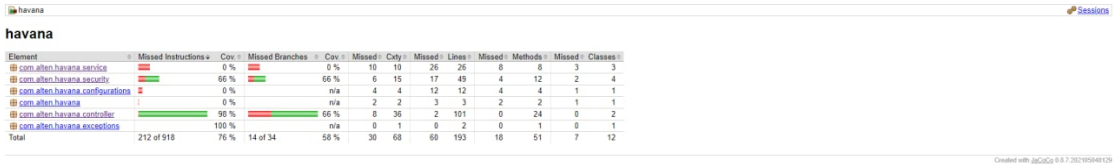

Figure 2.13: Coverage 76% BO backend

2.2.5 Release

The *release* stage can be regarded as the last of the development world, or first of the operations world. The release phase is responsible in producing a product release, which is a compiled version of the software. In order to do this, it is necessary for all tests to be successfully executed, as well as the compilation phase, and in the end, a release becomes available.
Available means that a release exists (e.g., as binaries), and is ready to be deployed. We will address the *deployment* concept in the next section.
For now, let's concentrate on how all of this is automated through the GitLab CI/CD feature, through a simple file called `gitlab-ci.yaml`.

GitLab CI/CD

GitLab CI/CD [14] is an incredibly powerful, feature-rich tool for dealing with the requirements of Continuous Integration/Delivery/Depolyment methodologies. During our internship, we used it to create a pipeline that would be triggered whenever something was pushed (or merged) to the repository (or branch specifically), also within the GitLab portal.

A push operation to a development branch triggered a "reduced" pipeline, verifying only the test and build steps, while a merge from one of the development branches, to the main branch, caused the entirety of the pipeline to execute.

The pipeline, in its primordial version, was defined as shown in the following diagram:

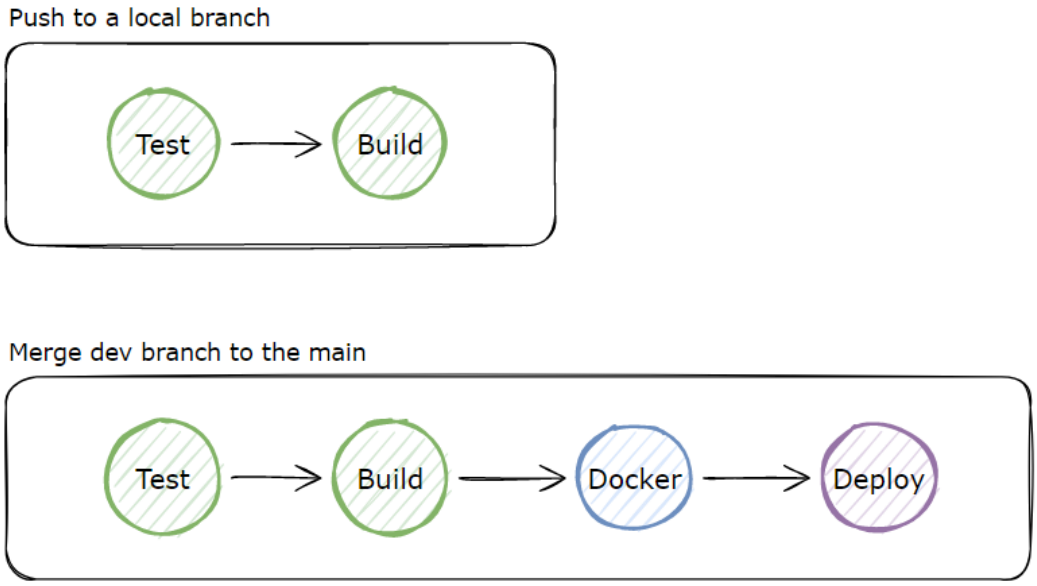

Figure 2.14: First version of pipeline

From the diagram shown above we can observe the reduced pipeline first, and then the full pipeline with all its steps.

Before talking about the Docker and Deploy steps, here is how the pipelines were defined through the `gitlab-cd.yaml` file.

The first thing to define in a `gitlab-ci.yaml` file, is the list of steps (or stages) of the pipeline.

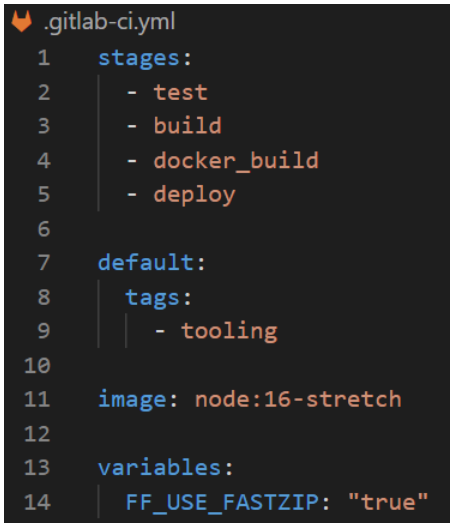

Figure 2.15: Stages in pipeline in `gitlab-ci.yaml`

We can also specify other parameters like tags and environment variables.

After that, each individual stage must be defined. Taking as an example a simple stage, such as the build stage (figure 2.16), we can see how only commands are specified to be executed, some of them are about configuration (`before_script`), while others are dedicated to the real task of the stage (`script`).

```
88  build:
89    stage: build
90    dependencies: []
91    before_script:
92      - npm install -g @angular/cli
93      - npm ci
94    script:
95      - npm run build
```

Figure 2.16: Stage definition in `gitlab-ci.yaml`

More complex stages might be dependent on results from other steps in the pipeline, might generate artifacts (temporary files), and might

have rules and conditions for their execution (such as executing the Docker and Deploy steps only if it is a merge on the main branch).

Docker

Docker is a platform that allows an application to be containerized, packaging the application itself along with all its dependencies from libraries and OS. This allows the Docker container to run via a software called Docker Engine, on any platform, removing the need for configuration of any sort.

Docker is an application that is usually considered in the deployment phase, but in our case, our release is actually a Docker Image.
A Docker Image is the immutable file that contains all the dependencies, from which a Docker Container can then be created and further configured.
In our case, Docker Containers are automatically created in Kubernetes Pods, which we will see in the next sections.

The following images represent the stage and the templates it extends.

```
101  docker_build:dev:dockerimage:
102    extends: .docker_build_template
103    variables:
104      IMAGE_TAG: latest
105      BUILD_ENV: build_test
```

Figure 2.17: Docker stage definition in `gitlab-ci.yaml`

```
.docker_build_template:
  stage: docker_build
  dependencies: []
  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} --build-arg build_environment=${BUILD_ENV} .
    - echo "${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
```

Figure 2.18: Docker template definition in `gitlab-ci.yaml`

In a nutshell, the template uses a docker image so that we can build our new image, with the appropriate parameters; and then it is uploaded to our azure cloud.

From there, the deployment phase can begin.

2.2.6 Deploy

Finally, the *deployment* phase is the one that, when completed, will make the new release available for external access.

What it means is that an outside user will be able to type a URL into his or her browser, and thus be able to access our site.
There are many steps and many middle infrastructure and configurations that I am glossing over, like firewalls, virtual hosts configuration and application servers, but mostly of that has been auto configured by the Terraform template we used (more on Terraform on the appropriate section).
Not to mention that the deployment phase also considers the deployment of supporting infrastructure, such as databases (which will be covered in the Cloud infrastructure section).

Now, the deployment itself is done rather simply:

```
47    script:
48        - kubectl -n default --kubeconfig kubeconfig rollout restart deploy/bo-frontend
49        - kubectl -n default --kubeconfig kubeconfig rollout status deploy/bo-frontend
```

Figure 2.19: Deployment script in `gitlab-ci.yaml`

These two commands in figure 2.19, are all we need in order to deploy our new release through our Kubernetes Cluster.

But what exactly is a Kubernetes Cluster? Why do we need it? How is it created? In the next two sections I broadly answer these questions.

Kubernetes

Kubernetes is an open-source container orchestrator, which allows to automatically manage the deployment and operation process of a set of containers.
The use of Kubernetes is almost necessary when dealing with non-monolithic systems, as in our case. I will discuss our multi-service architecture in a separate section.

Kubernetes features are many [15]:

- **Service discovery and load balancing** Kubernetes is able to expose containers externally through DNS or IP addresses. It is also able to autonomously distribute requests, in case there is a service running on several servers, so as not to overload just one and to keep stability.
- **Storage orchestration** Kubernetes allows you to automatically mount a storage system of your choice, such as local storages, public cloud providers, and more.
- **Automatic bin packing** You provide Kubernetes with a cluster of nodes that it can use to run containerized tasks. You tell Kubernetes how much CPU and memory (RAM) each container needs. Kubernetes can put containers into specific nodes to make the best use of resources.

- **Self-healing** Kubernetes notices when something is not right. When a container crashes, Kubernetes immediately tries to replace it with a new one.
- **Secret and configuration management** Sensitive data can be handled independently of container images, so that secrets can be updated without the need of rebuilding.

We understood that the key elements of kubernetes are clusters, nodes and pods. But how are these elements exactly interconnected?

The following might be a rough representation of a generic Kubernetes Cluster:

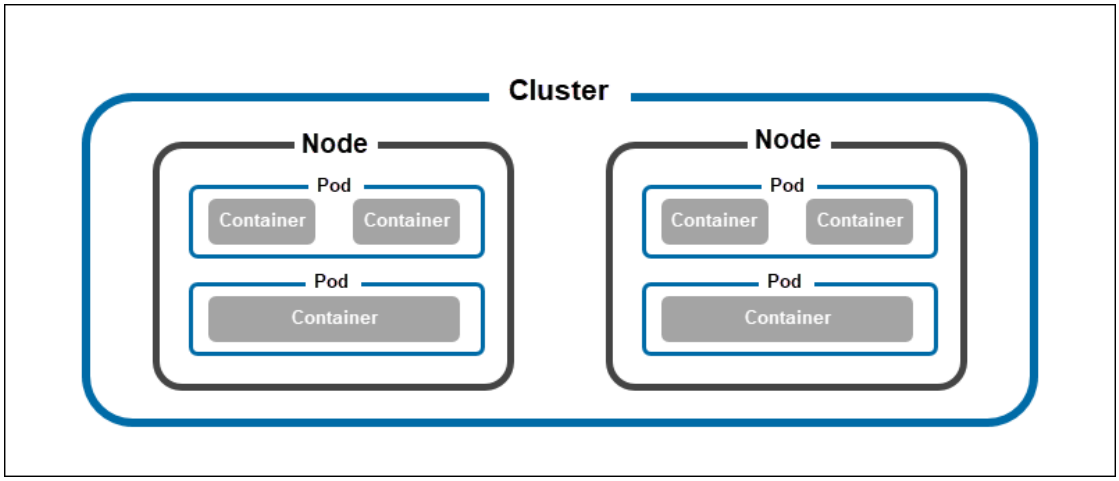

Figure 2.20: Kubernetes Cluster schema [16]

Let's precisely describe each component [17]:

- **Cluster**
  We can observe that a Kubernetes Cluster is the enclosing entity. Indeed the Kubernetes Cluster is just a set of Nodes that may run on several machines. All the nodes are managed by a Master Node, which are the responsible for the behaviours of the entirety of the cluster.
- **Nodes**
  Nodes are computing units. They could be physical machines or

virtual partitions. Each Node has a fixed amount of CPU power and RAM that can share among its Pods.

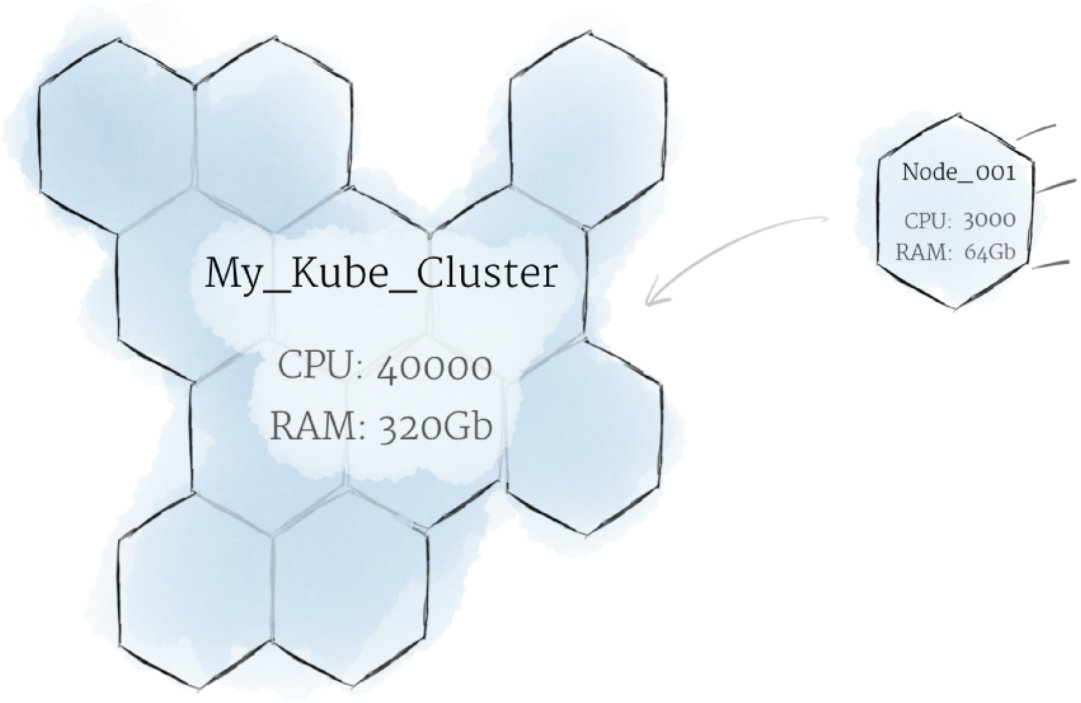

Figure 2.21: Kubernetes Cluster + Node [17]

- **Pods**
  Pods run inside Nodes and are a high-level structure that wraps containers. A Pod can request for a specific amount of resources (CPU and/or RAM) from its Node. Containers inside the same Pod will share resources and network properties (like IP).
  We can have as many containers as we want inside the same Pod, but since Pods are the single unit that could be potentially replicated, or replaced in case of failure, it is wise to make them with the least number of container inside as possible.

  This is because in the case where a service offered by a container inside a Pod has a large load of requests, the Pod containing this service is automatically duplicated N times, as much as needed

(Node resources permitting).
However, if there are other (more or less important) containers within the same Pod, they too will be multiplied, thus wasting unnecessary computational resources and memory.

- **Containers**
  Containers on Kubernetes are packaged as Linux containers. They can contain multiple programs even though once again it is better to isolate processes for the sake of efficiency, when possible.
  In our case Containers are generated automatically from Docker Images by the Helm Charts through Terraform (we will see all of that in the next sections).

- **Deployments**
  Actually what our Helm Chart does is define an entity called Deployment. It is a kind of declaration of parameters to be applied to a Pod (or more if we decide we want the same Pod duplicated N times by default).

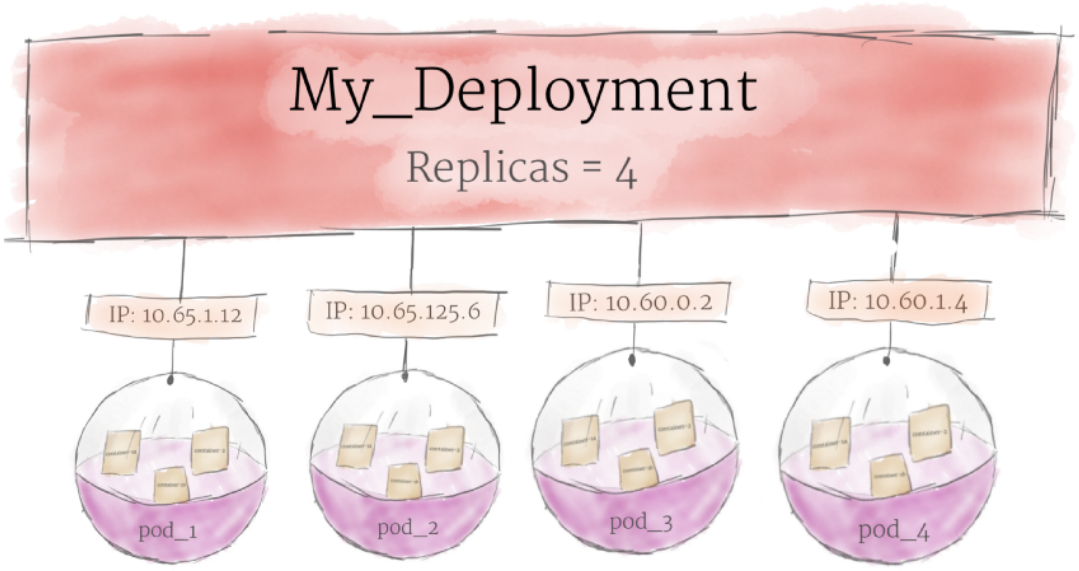

Figure 2.22: Pods Deployment [17]

- **Ingress**
  Finally, it is necessary to define for each pod offering a service

outside the Kubernetes Cluster, what is called Ingress.
The Ingress Controller then allows you to define whether and how to access the service inside the Pod (e.g., an URL).

In our case, the Kubernetes cluster is defined within our cloud, specifically Azure Cloud, which provides native Microsoft functionality to facilitate the use of a Kubernetes cluster, through the Azure Kubernetes Service (AKS).

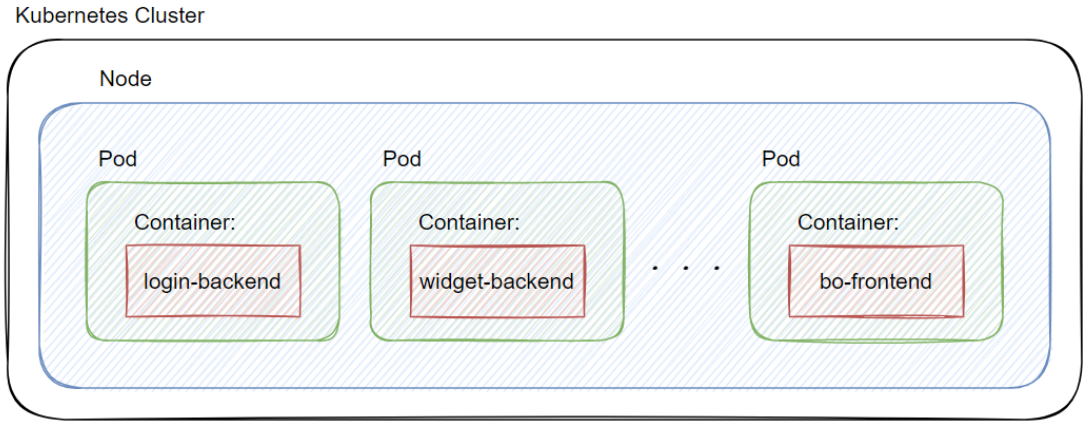

Figure 2.23: Our Kubernets Cluster architecture

The architecture is very simple: we have one node, and many pods, each containing one container (or service) inside. We will review the details in the section explaining the Cloud architecture.

Pods are launched through images that the AKS takes from the Azure Container Registry, which is a storage service inside the Azure Cloud that contains the latest versions of our releases.

Helm

Helm [18] is the answer to the question: "how do we define the Kubernetes architecture?".

There are several ways to configure a Kubernetes Cluster, but in a DevOps environment, where everything is automated, the creation of

the Kubernetes cluster must be as well. To do this we used Helm. Helm is referred to as the Kubernetes Manager. With Helm we can define, install and update any Kubernetes application.

There are only 3 key concepts about Helm:

- **Chart**
  A Chart is nothing more than a package of files with a well-defined structure, in which we can define all the specifications of a resource within a Kubernetes Cluster.

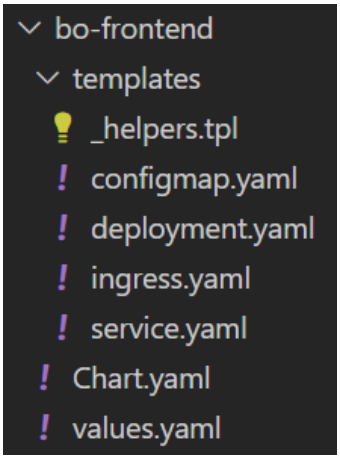

Figure 2.24: Helm directory structure

  Exploring how each individual file works is beyond our scope. However, it can be said that the Chart.yaml file contains basic information such as name, description, and Release version, and that the real "magic" is described in the other self-generated (and properly edited) template files, which are taken into consideration together with the Chart file.

- **Repository**
  A repository is a place where charts are saved or shared. There are public repositories accessible directly from the Helm client from which charts can be downloaded.
  Or you can have a private repository where you can create and save your own charts.

- **Release**
  Finally, a Release is an instance of the Helm chart deployed in the Kubernetes Cluster.
  Once the chart is installed via the Helm client (or via Terraform in our case), the Release with a certain version is created, and the resources in the Kubernetes Cluster are deployed.

As already mentioned, in our case we do not directly use a Helm client locally.
Everything is managed through the powerful Terraform tool, which automatically deploys helm releases.

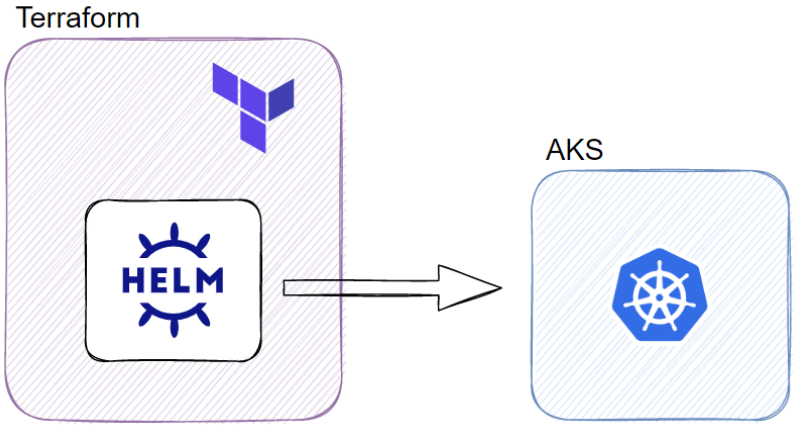

Figure 2.25: Helm lives within Terraform, to deploy resources in the AKS

The previous scheme will be further developed in the appropriate section about Terraform.

2.2.7 Operate

The *operate* phase is about keeping the application up and running. Fortunately, in a cloud environment, it is not our team's responsibility to worry about the hardware being correctly running all the time.

In addition, as discussed earlier, Kubernetes is able to handle the load

of requests itself and restart any pods with an application that has crashed.

2.2.8 Monitor

The last phase of the cycle is devoted to observing errors or problems, which can then be analyzed and considered for the next cycle so that any issues can be improved and corrected promptly.

During our internship we did not establish communication channels with users, mainly because our app was developed in a **dev** environment (we deployed in a **prod** environment at the end of our internship, as it was an MVP requirement).

The development (dev) environment is reserved exclusively for developers to test site functionalities in an environment equal to a real one, which is usually called the production (prod) environment.

This mainly means that the only feedback we had was from stakeholders during sprint reviews, and rarely close colleagues. The only monitoring channel was the logs generated by Kubernetes pods, which we accessed through the Lens software [19].

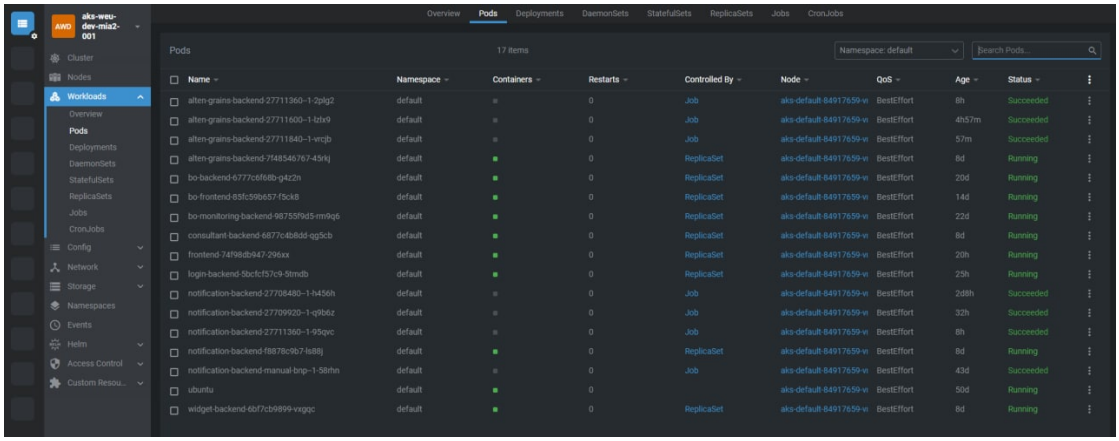

Figure 2.26: Lens interface

Chapter 3

Technical challenges & Solutions

After introducing the DevOps methodology, and the technologies we used to be able to apply it, in this chapter I will explain the most important milestones that characterized the actual development of our application, covering both theoretical and practical aspects.

3.1 Architecture & C4 model

At the beginning of our intenrship, as I mentioned in Chapter 1, we were given what is called a "cahier des charges" (in French). It is nothing more than a document containing the functional specifications of the project (or at least what had been thought of by the client up to that moment).

The document was 21 pages long, and we spent several days discussing the various implementation possibilities and what technologies would support us for the best.

The project was complex. Many users would have had access to our portal. Each of them would have had access to different functionality

depending on their role. The portal offered many different services and data of various kinds and relationships had to be stored.

The path to take seemed clear to us right from the beginning: microservices-oriented architecture.

The idea then was to have a microservices back-end system, with a monolithic front-end and a database (I will address microservices architecture and database choice in more detail in later chapters).

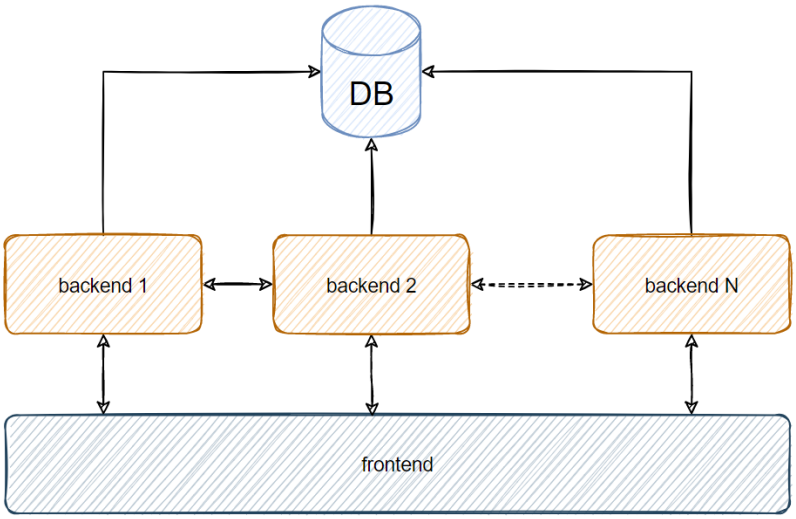

Figure 3.1: General idea of software achitecture

But the precise definition of the architecture is not a task that can be done with words alone, nor is it sufficient to take scattered notes or to draw poorly labeled squares and lines on a white-board (like in the picture 3.1).
It was essential to use a conceptual model with a well-defined structure of representation: the C4 model.

3.1.1 C4 Model

The C4 model [20] consists of an approach to schematization divided into 4 levels of abstraction. It is a way to annotate, describe, discuss,

and communicate the structure of software with clients or colleagues, regardless of their level of technical expertise.

The 4 levels are Context, Container, Components and Code; hence the name C4 model. We have to think of the C4 model as a concept map, in which you can zoom in for more detail, and you can do this from the Contex level (most abstract level), down to Code (most practical and technical level).

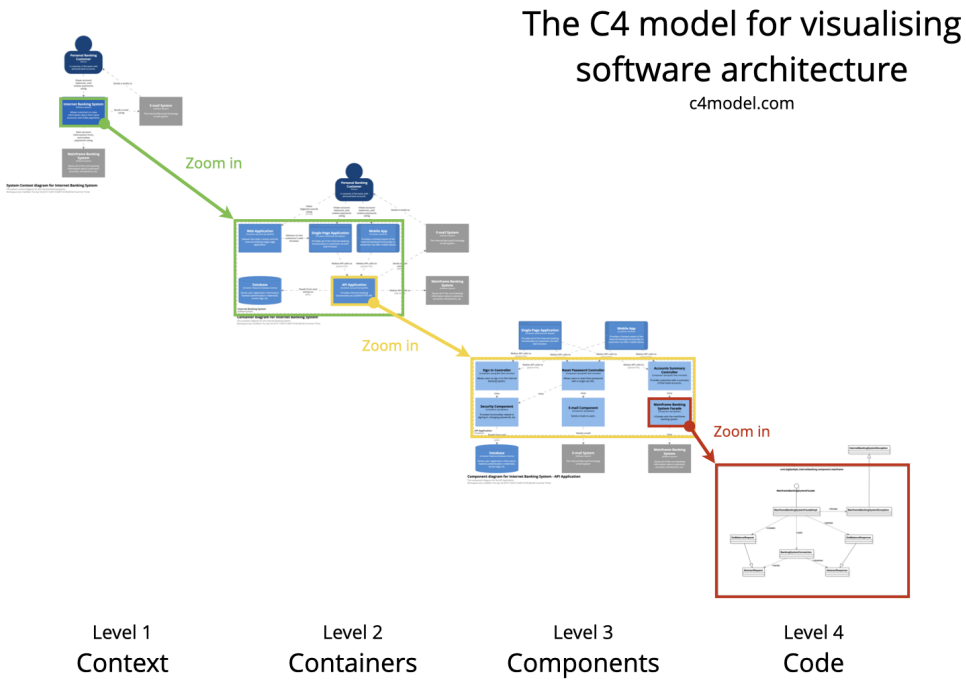

Figure 3.2: C4 model zooming concept

Broadly speaking, each level represents:

- **Context**
  The level of context is the most general. It is not important to specify technologies or technical details. This level focuses on people and what systems they will be dealing with. High-level links between external systems can also be represented. As an example, take the primordial version of the C4 model we created for our Back-Office portal (slightly outdated, many specifications

have been added over the months).

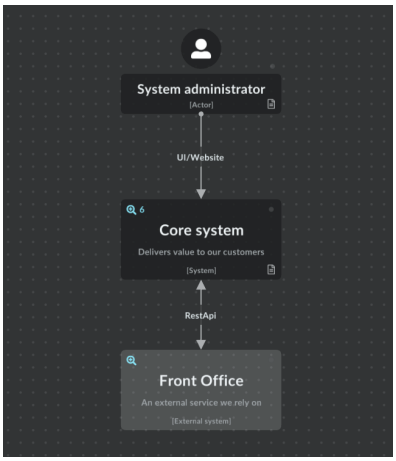

Figure 3.3: C4 Context Back-Office

You can see how only one user would have access to our main system, which in turn would have interactions with the Front-Office system.

- **Container**
  The container layer is not to be confused with elements like Docker Containers, although they *could* be Docker Containers. A container is a unit that could run or be deployed by itself, like for example server-side app, a Docker container, a databases, ecc.
  At this level are shown technologies and protocols, and one can begin to observe how the system actually interacts with the outside (and internally) in a technical way. (figure 3.4)

- **Component**
  When zooming inside a container, we can see the components. Components are an abstraction of a group of code within a container. Such as controllers or services, but it could be even classes or interfaces. (figure 3.5)

- **Code**
  Finally, the last level is very similar to a UML diagram, related to the code of a single component. This layer is rarely needed, and in

fact we have not considered it.

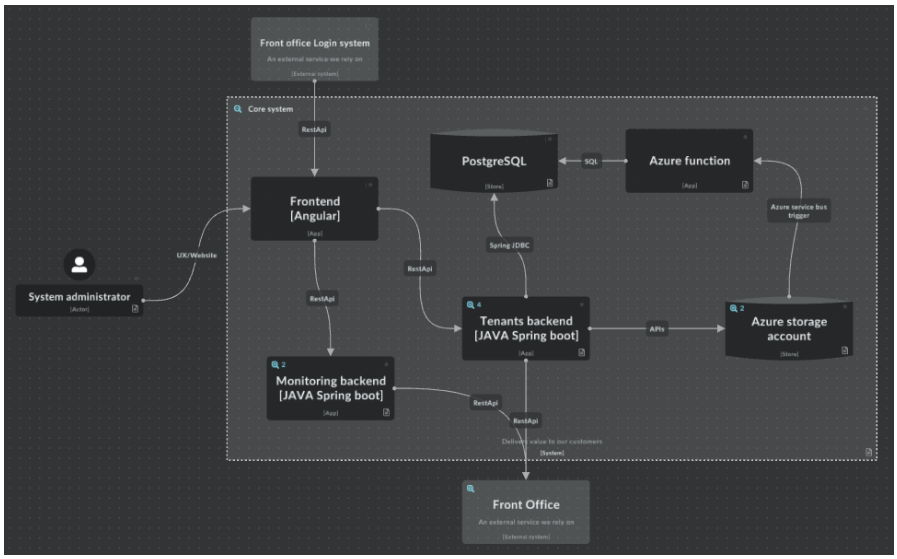

Figure 3.4: C4 Containers Back-Office

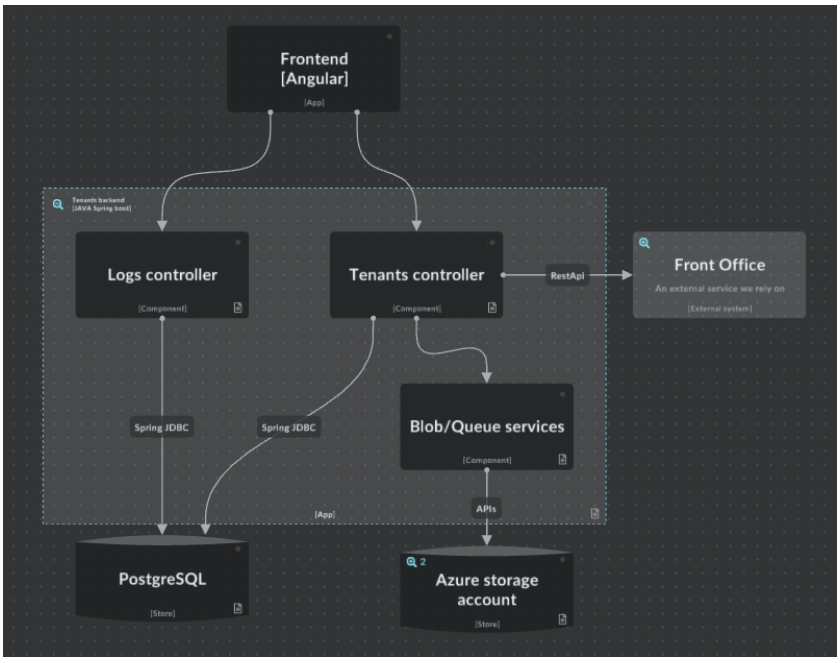

Figure 3.5: C4 Components Back-Office

The examples just shown depicted our back-office, for which in my

opinion it would not have been necessary to use the C4 model. For our front-office, however, it was essential.

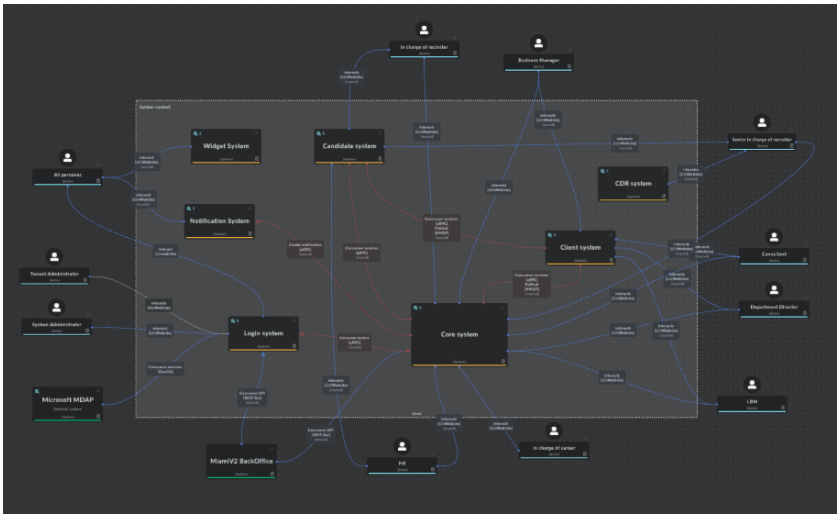

Figure 3.6: C4 Context Front-Office

3.1.2 Customer Journey

In parallel, 4 elements of the team dedicated themselves to the Customer Journey. It is a diagram representing the actions that an "actor" could do. In this example, the Sys Admin in the Back-Office.

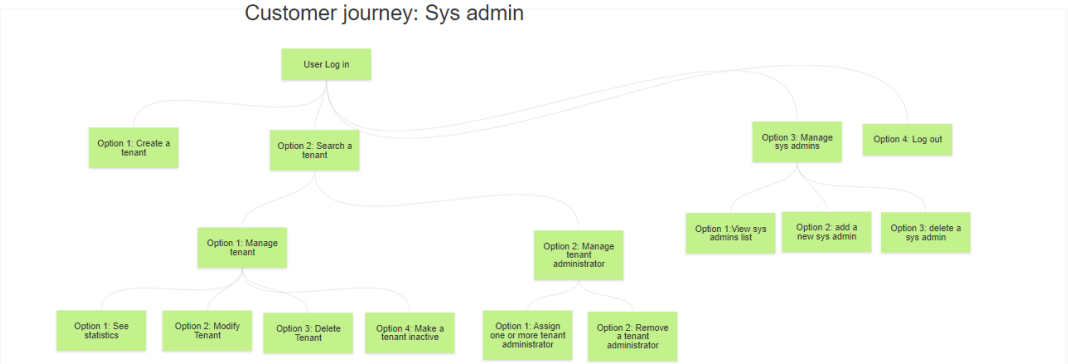

Figure 3.7: Customer Journey of Sys Admin in Back-Office

The purpose of this parallelization is **comparison**.

Regularly during the initial design days the people dedicated to Customer Journeys and those dedicated to the C4 model would discuss, to check the consistency of their respective schemes. It was a way to have an overlap of two different point of views with the same end goal.

3.1.3 Technology Stack

At the end of the discussions and after also consulting with experienced developers, these are the technologies we chose to adopt.

For the Front-Office team:

- **React**
  We had initially opted for Svelte, but since it is still in its primordial stage, we changed our minds quickly, and then chose to use React. We consider React easier than Angular, and it made sense for us to pick it since most of us had never touched web development.
- **NextJs**
  NextJs is a library that facilitates many operations like server-side rendering, routing and others, and it was strongly recommended by a senior frontend developer.
- **MUI**
  One of the most popular graphic libraries for importing ready and well-made components. Ideal for keeping visual consistency throughout the site.
- **NestJs**
  Based on Node.js, so lighter than Java alternatives. Also easy to learn and therefore suitable for our inexperienced team. Framework widely used in our work context (so useful for a future after the internship).

For the Back-Office team:

- **Angular**
  It should be noted that the back-office development started a few weeks late, and so, since our main goal during the internship was to

learn, we decided to step outside the comfort zone of React and opt for Angular: framework that is also widely used and could come in handy during our future in the industry.

- **Angular MUI**
  To keep graphic consistency (as much as possible) with the Front-Office application.
- **Spring-Boot**
  Again, this choice was dictated by the popularity of the framework and the desire to try new technologies, plus the fact that we would never need the speed and performance of NestJs in the BO.

3.2 Databases data structure

As was mentioned earlier, our project is deployed on Azure Cloud (I will discuss why in the next section).
Working in the Azure Cloud environment has offered us many advantages, but it has also influenced us about certain decisions. One of these concerns the choice of databases, such as CosmosDB.

Before explaining further the reasons for the choice and internal structure of our databases, it is good to take a step back and understand the difference between relational, and non-relational databases.

3.2.1 SQL vs NoSQL

The main characteristic of **relational** databases is that they contain highly structured data. Data are usually organized in tables, which are in turn divided by columns, which identify the information, and by rows, which identify the whole information record.
Each row has an element that is unique to the entire table: the *key*. This element allows us to create *relationships* between records in different tables.

The schema in Figure 3.9 represents an Employee table with a number of information about each employee (Name, Title, HireData) and the

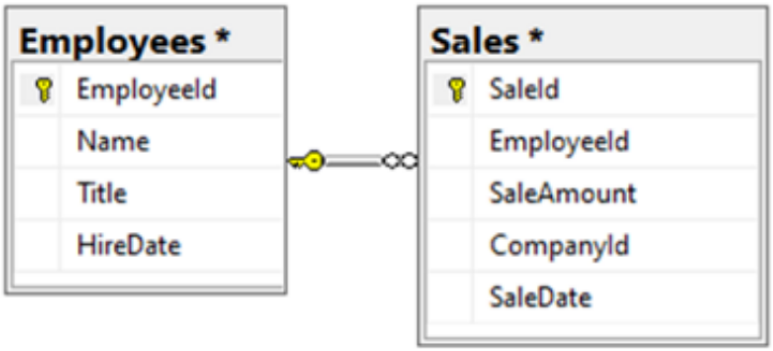

Figure 3.8: Relationship in relational database [21]

EmployeeId, which represents the key. On the right, on the other hand there is a table representing Sales, which contains, in addition to all the necessary information and its own key (SaleId), what are called foreign keys (EmployeeId and CompanyId), which reference keys to external tables (on the schema it's graphically represented only the relation between the table Employees). This is the property that defines the name of relational databases.
The ability to create, in a structured way, relationships between entities. The way relational databases are created has a number of advantages and disadvantages:

SQL Pros

- Relational databases are optimal to work with structured data with a lot of relations.
- The Structured Query Language (SQL) is an extraordinary powerful tool to extrapolate the right information in an efficient way.
- It exists a well defined normalization process [22] thanks to normal forms, which are rules that indicate how data inside relational databases should be stored.
  Having a normalized database implies having anomalies (redundancy, lack of integrity, ...) reduced or eliminated.

SQL Cons

- The schema of the data should be defined up-front, as it is hard to change structure.
- In function of the complexity of the structure, SQL retrieval and manipulation may be slow.
- In order to improve performance, vertical scaling is the only solution, which could be very expensive.

On the other hand, **non-relational** databases can contain unstructured data. It is up to the developers to define how the data will be stored within them.

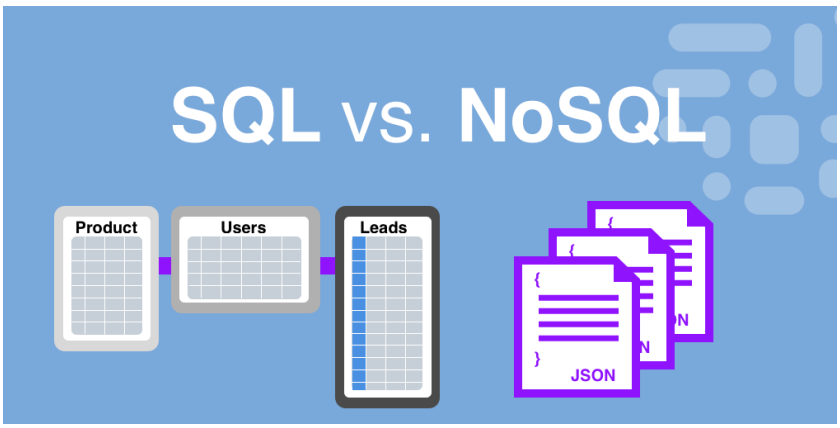

Figure 3.9: Structure SQL vs NoSQL [23]

There are many different types of NoSQL databases, such as Key-Value based, Graph based, and others, but the most common is Document based, which is what we used too.

Documents are usually organized following a specific format to make them easy for software to read, such as JSON or XML. In Documents, there are no strict rules like tables; data can be stored without a specific pattern. It is also possible to create references such as foreign keys. It is all in how this data will be read.

Non-relational databases are relatively new compared to relational ones, but there are still pros and cons to analyze when it comes to making a choice.

NoSQL Pros

- It is much easier to make changes to the structure of data even at a later stage of development.
- Non-relational databases are optimized to work with a huge quantity of data.
- It is Cloud oriented. It can easily scale horizontally.

NoSQL Cons

- Manual querying on unstructured database could be hard.
- Difficult to check for integrity and consistency.

3.2.2 Our choices

After explaining the difference between relational and non-relational databases, it is easy to motivate our choices.

For the Front-Office application, we chose to use CosmosDB (NoSQL). Not only because it is easy to use and configure in the Azure Cloud environment, but also because we knew that the data structure was very dynamic, and could change very often. Plus we knew that potentially our application could handle a significant amount of data because of Alten Grains (basically an e-commerce within the HR portal).
We also knew that our data would be highly dependent on relationships with other data, which made us hesitant. But I think the difficulty in manually managing the relationships was offset by the flexibility we had over the months in changing the data structure multiple times.

On the Back-Office side, we opted for PostgreSQL, due to the fact that the data structure was much simpler, and therefore easy to define

before development even began. Again, the data had relationships, and the volume would never reach that of the Front-Office.

3.2.3 Front-Office CosmosDB Document Example


```
{
    "dateRequest": "2022-07-18T08:36:21.777Z",
    "state": "approved",
    "type": "GrainsRequest",
    "requestType": "activity",
    "requestedObjId": "be77d353-6b9e-4707-af82-ffa7fcf052a2",
    "consultantId": "tommy.lecourt@alten.com",
    "details": "",
    "tenant": "nova",
    "dateEvent": "2022-07-17T22:00:00.000Z",
    "counterResentNotifications": 2,
    "id": "a8dc6cf7-7f95-45e2-9ec5-f93405581b80",
    "_rid": "KAZAAPOMnt+qAAAAAAAAAA==",
    "_self": "dbs/KAZAAA==/colls/KAZAAPOMnt8=/docs/KAZAAPOMnt+qAAAAAAAAAA==/",
    "_etag": "\"1400f0b3-0000-0d00-0000-62f6021f0000\"",
    "_attachments": "attachments/",
    "requestedObj": {
        "description": "Create an event",
        "gain": 100,
        "tenant": "nova",
        "type": "Activity",
        "id": "be77d353-6b9e-4707-af82-ffa7fcf052a2",
        "_rid": "KAZAAPOMnt9iAAAAAAAAAA==",
        "_self": "dbs/KAZAAA==/colls/KAZAAPOMnt8=/docs/KAZAAPOMnt9iAAAAAAAAAA==/",
        "_etag": "\"5e0038a1-0000-0d00-0000-62d53e0c0000\"",
        "_attachments": "attachments/",
        "_ts": 1658142220
    },
    "_ts": 1660289567
}
```


Figure 3.10: Example of data in a CosmosDB Document

As you can see from the image 3.10, the document is organized with the JSON format. Every information is represented as a pair key-value, and the latter can represent any type of data: numeric, string, date, array, dictionaries, etc...

There are also values that represent foreign keys and primary keys, as in a relational database, and it is worth noticing that CosmosDB integrates mechanisms as checking the uniqueness of id by itself. CosmosDB is indeed considered a NoSQL semi-structured database.

3.3 Cloud architecture & Terraform

A whole other thesis could be written on Cloud Computing, but it is not my intention to do so.
Instead, it is my intention to present a schematic of the Cloud architecture we have adopted and briefly explain each component, to make sense of it all.

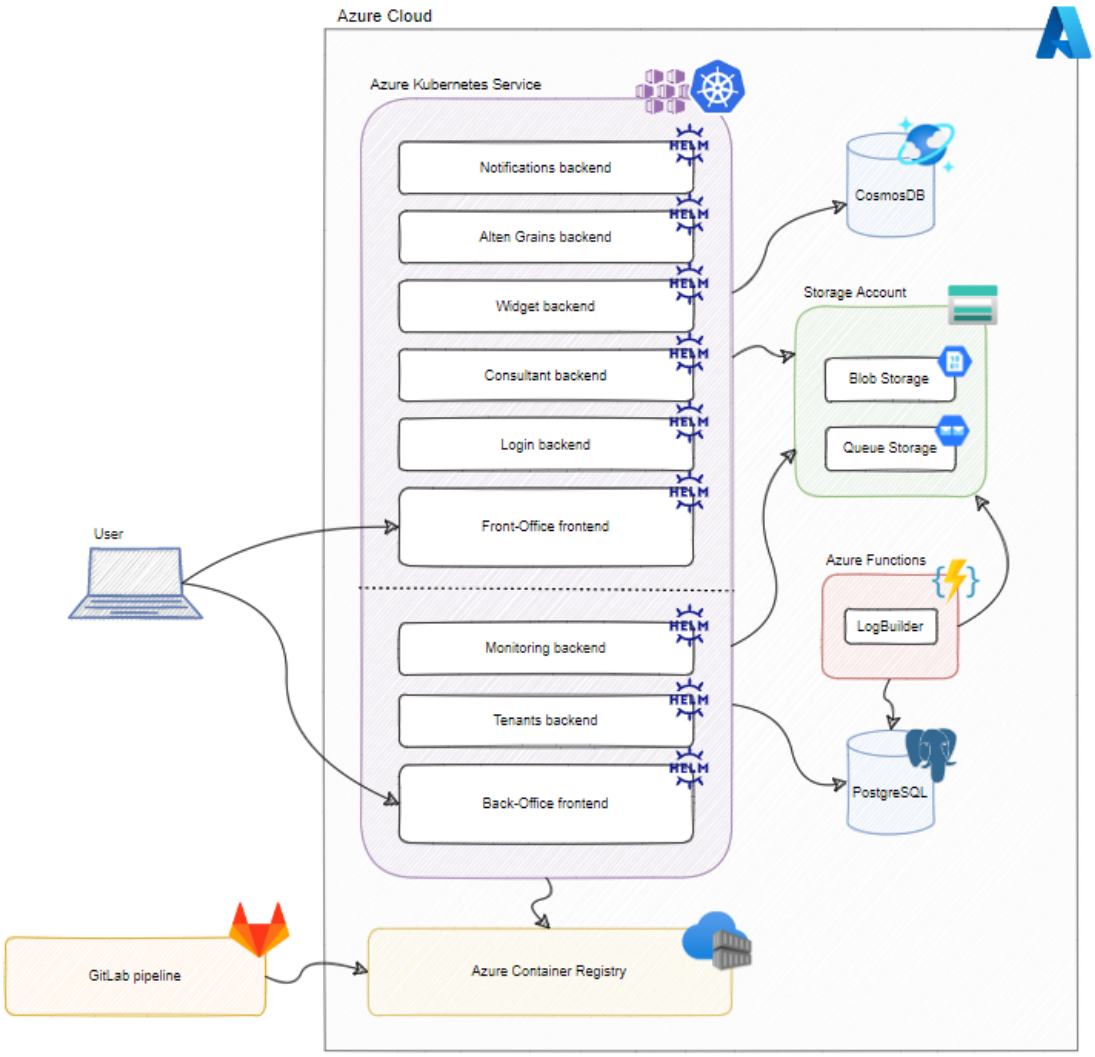

Figure 3.11: Schema of Cloud architecture

Seemingly meaningless relationships are not an oversight. Arrows

represent if an entity has a communication channel with another entity. The direction of the arrow does not represent the data-flow, rather from whom the initial request starts and to whom it is directed; after that often the data-flow is bidirectional.

3.3.1 Entities in Cloud schema

- **Azure Cloud**
  Azure Cloud is of course the most important component of this architecture. Many of the elements shown in the diagram are Microsoft proprietary in the Azure environment. Azure Cloud is a heterogeneous Cloud service, since it can be considered SaaS, PaaS, and Iaas, although we use it mostly as PaaS, since we ignore everything related to OS management and runtime management, and we let Azure handle it.

- **User**
  User represents the end-user that will have access to our web application through the frontends. They will probably be just Alten employees who have the right level of permissions to access our services.

- **GitLab and Azure Container Registry**
  These two are strictly related. The GitLab repository is where all our code is stored, and as we discussed in the Release section of the DevOps methodology, at the end of the Release phase in the pipeline, the Docker Image is uploaded (or "pushed") in the Azure Cloud. The Azure Container Registry (ACR) is where the latest version of each image is stored.

- **Azure Kubernetes Service**
  The task of the Deploy phase is indeed to make the Azure Kubernetes Service (AKS) to "pull" the latest image from the ACR, and with that replace the Pod containing the old image. The AKS contains instances of Helm Releases running. I omitted the arrows inside the ASK to avoid confusion, but the frontends communicates

with their corresponding backends (the dotted line conceptually separates the Front-Office (FO) and Back-Office (BO), but the pods all live in the same node), and in some cases there are communications even between FO backends and BO backends.

- **CosmosDB and PostgreSQL**
  We can notice how CosmosDB is connected only with the conceptual part of the FO. That's because only the FO backends communicates with the CosmosDB. Respectively, BO backends communicate only with PosgreSQL.

- **Storage Account, Blob Storage and Queue Storage**
  Both conceptual divisions of the Kubernetes Cluster communicate with the Storage Account. The Account Storage is a cloud storage service in Azure, which offers a Blob Storage service, that is used to store images (logos, profile pictures, ecc...) and a Queue Storage, about which I will talk more in the Asynchronous logs queue section.

- **Azure Functions**
  Azure Functions (AFs) are pieces of code that live inside the cloud and can be triggered on particular conditions. I will talk more about AFs in the asynchronous logs queue section.

All of these elements can be defined through the Azure Portal GUI or through the Azure Command-Line Interface, but since the elements are highly interconnected, a change to one element could trigger a chain reaction for several others, making a change operation time-consuming and tedious.
Fortunately, we are working in a DevOps environment, thus, automation.
This is where the Terraform tool, named several times now, comes into play.

3.3.2 Terraform

Terraform is a tool defined as infrastructure as a code.
Terraform is basically a set of files that contain a structured description

of the resources to be deployed on a cloud infrastructure.
Each time we run Terraform, it handles the creation, replacement or destruction of the resources automatically. This is a process easy to integrate in an automatized pipeline and has other important advantages:

- It's easy to share infrastructure, since it's all defined inside some files. A lot of infrastructure are ready to take and easy to modify.
- Ease of collaboration, using Terraform files together with a Version Control System as GitLab.
- It is enough to declare all the properties of the resources in Terraform, and it handles the creation process automatically.
- In case of dependencies, Terraform updates all the resources affected by a change.

But how does it work exactly?

Terraform's functionality is based on the APIs of the individual services with which it can interact. But between the APIs of the services and Terraform there is another element in order to standardize the way all elements are defined through Terraform: the *providers*.

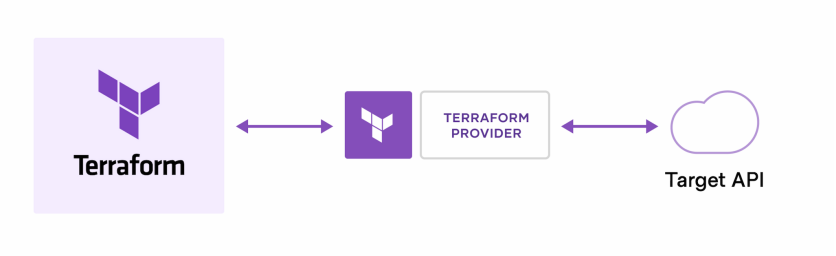

Figure 3.12: Terraform providers schema [24]

Providers are plugins we can include in our Terraform files (like in the figure 3.13) allowing us to communicate with the target API through Terraform code.

```
terraform {
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "~> 2.99"
    }
  }
```

Figure 3.13: Include providers in Terraform

In the following image it's shown the definition of the PostgreSQL database in our Azure environment.

```
resource "azurerm_postgresql_server" "postgresql" {
  name                = "psql-weu-${var.workspace}-mia2-001"
  location            = data.azurerm_resource_group.main-rg.location
  resource_group_name = data.azurerm_resource_group.main-rg.name

  administrator_login          = "fake_admin"
  administrator_login_password = random_string.db_password.result

  sku_name   = "GP_Gen5_2"
  version    = "11"
  storage_mb = 5120

  ssl_enforcement_enabled      = true
  backup_retention_days        = 7
  geo_redundant_backup_enabled = false
  auto_grow_enabled            = true

  tags = var.alten-tags
}

resource "azurerm_postgresql_database" "postgresql" {
  name                = "postgresqldb"
  resource_group_name = data.azurerm_resource_group.main-rg.name
  server_name         = azurerm_postgresql_server.postgresql.name
  charset             = "UTF8"
  collation           = "English_United States.1252"
}
```

Figure 3.14: Definition of PostgreSQL in Terraform

We can see how a resource is defined with the keyword *resource* followed by the type (given through the provider) and then the name, that could be used as a reference inside the Terraform files.

For instance at line 95 of figure 3.14 the `server_name` property is extrapolated from the attribute `name` from the resource called `postgresql` of type `azurerm_postgresql_server`.

We can see how in order to define a PostgreSQL, two resource are needed: a server and the database itself. If we were creating a database through the Azure Portal, we would have been guided through a step-by-step process that automatically included both resources. Fortunately, all Terraform providers are exhaustively documented, and they often include all necessary resources dependencies (see [25] as an example).

We are left only to understand how Terraform is actually executed. In our case, we used a Terraform linked to a GitLab template so that we could include the resource deployment process automatically in the pipeline. In a primordial stage, the Deployment phase of the general pipeline actually triggered the execution of a second pipeline, the one related to Terraform. But we realized that this was not necessary, so we ended to keep the code pipeline and the Terraform pipeline independent from each other.

The stages of the terraform pipeline are three:

- **Init**
  The execution of `terraform init` performs a number of steps in order to initialize and configure the working directory such as Backend Initialization, Plugin Installation and so on. [26]
- **Plan**
  With the command `terraform plan`, Terraform generates what's called an *execution plan*, which describes which resources Terraform has identified that need to be created, updated, and destroyed, and in which order it is going to perform these operations.
- **Apply**
  Finally, the `terraform apply` command uses a saved execution plan to apply the changes. Otherwise it runs the `terraform plan` command again, to generate a new execution plan and apply the

planned changes to the actual infrastructure.

3.4 Micro-services architecture and Login-System

Manage user login, receive requests for Alten Grains and Rewards, manage notifications, and so on.
These are just some of the services our application offers.
When an application contains many functionalities, it may be the case to consider a microservices-oriented architecture.

A microservice is defined as an independent deployable element that manages a particular functionality and, when needed, communicates with other microservices to send or obtain additional information.

A microservices-oriented architecture is a type of architecture that includes multiple microservices, each of which is "loosely coupled" to the others in order to allow communication while maintaining independence. Containers favor a microservices architecture, and communication channels are realized through rest APIs or specific protocols such as gRPC.

This independence between microservices (or services from now on) has numerous advantages:

- **Ease of development**
  It is much easier to develop the individual small components of a larger whole. Teams or individual team members can easily divide up the development of individual services, agreeing on a communication system should the need arise.
  It is also much easier to test the entirety of the component without incurring the risk of an unexpected behavior coming from another feature in the same monolithic code.
- **Deployment and Maintenance**
  Services can be updated and deployed without creating a general down-time. It is also easier to identify and localize any problems within a microservice.

- **Independence over technology**
  Independence gives the opportunity to develop each service with a different technology. This means always using the optimal technology for a certain service.
  Communication channels are never dependent on technology.

- **Overall reliability**
  One of the strengths of microservices architecture is its strong resilience and reliability. In the event that one service has problems, all the others are usually not impacted (those that do not need to communicate with the service in failure).
  It is fair to note, however, that we are not completely exempt from the risk of a total crash, as it is still possible the presence of a Single Point of Failure (SPOF). The SPOF, usually present in a physical or logical network, is an element on which many, if not all of the other elements are dependent, thus causing in the event of a crash, a forced stop of all the other components. In a microservices architecture there is usually some service more important than others.

- **Scalability**
  With this type of architecture, it is also very easy to duplicate heavily loaded services so as to keep up with the high volume of requests.

All the services that live in our Kubernetes cluster are independent Containers, which are depicted in Figure 3.11. I am going to briefly introduce each one of them, but there is one, however, that is worth describing in detail, and that will be the Login backend.

Frontends

Frontends are not usually thought of as microservices, rather microservices provide APIs for frontends.
In any case, Front-Office and Back-Office frontends are the elements that an end user can access directly through their device's browser, and that directly communicates to most of the microservices to provide

functionalities.

Alten Grains Backend

The Alten Grains Backend exposes APIs to perform CRUD (Create-Read-Update-Delete) operations for possible Activities and Rewards, as well as for Requests that Consultants can make to BMs (indeed to obtain compensation after an Activity has been made or completed or to spend Alten Grains to obtain real Rewards).

Notification Backend

The Notification Backend is the core element of the Notification System. It is a support service for various operations that occur in other backends.
For example, when a Request of a Consultant is accepted by a BM, the Alten Grains Backend sends a message to the Notification Backend via the gRPC protocol, which then adds a Notification item to the database, and sends to the user their notifications that have not yet been visualized (the Notification Backend is queried on page refresh).

Widget Backend

The Widget Backend exposes the API for widget management on the user dashboard. The widget Backend is the interface to the database that stores which and where widgets are placed on a dashboard. One database element for each user, containing their unique identifier and the list of widgets present on the dashboard, and their location.

Consultant Backend

Also the Consultant Backend allows just CRUD operations to Consultant profiles. We will see that the Login Backend deals with a similar concept, but the difference lies in the fact that the user-related objects managed by the Login Backend serve only for the purpose of access permissions, while those managed by the Consultant Backend contain

all the personal, career, and curriculum data related to each (and only to) consultant.

Tenant Backend

This is one of the two Back-Office backends, and it is used for CRUD management of Tenant objects and Logs, which I will discuss in more detail in the section on asynchronous logs queues.

Monitoring Backend

The other backend of the Back-Office, is responsible in querying the Front-Office backends (currently only the Login Backend) and the Tenant Backend to process and provide statistics regarding Alten GROUP users and tenants.

3.4.1 Login Backend

The Login Backend deserves a separate section because the Login System is the most complex system of them all. The complexity is due to an initial problem we had to overcome, related to the fact that all Alten employees would be able to log-in directly into our application with the already present Alten credentials, which are managed by a Microsoft Active Directory [27].

Note that the Login Backend is the microservice that exposes the APIs and functionalities for managing the login, while the Login System includes everything related to the login process, which I will explain in a moment.

To enable users to use the Alten credentials, it was necessary to use a library called MSAL (Microsoft Authentication Library), which, once logged-in through a Microsoft form, returns a Microsoft signed JWT token, containing some basic information (name and email of the user).

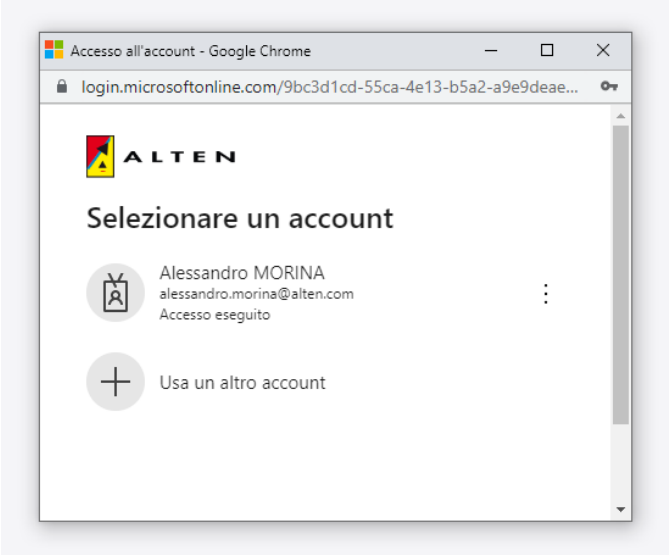

Figure 3.15: MSAL form

The first problem we faced was related to the fact that this basic information contained in the Microsoft token was actually the only one we could have access to, due to security restrictions.

The Active Directory contained a lot of other useful information, such as the role of the users, but we were forced to re-implement another database, containing the additional information necessary to guarantee each user the right level of access. Here comes the Login Backend, which receives the Microsoft token from the frontend, modifies it, adding the additional information taken from our internal database, and then sends it back to the user, who will then use it to access the resources (which are protected).
The image 3.16 schematizes the just explained token exchange.

As we can see, looking at image 3.16, the first thing that happens once the user successfully logs-in in the Microsoft login popup (figure 3.15), is receiving the Microsoft JWT token. Immediately afterwards, the token is sent to the Login Backend, which extracts the information from it, and verifies the existence of the user's additional data through the uniqueness of the email.

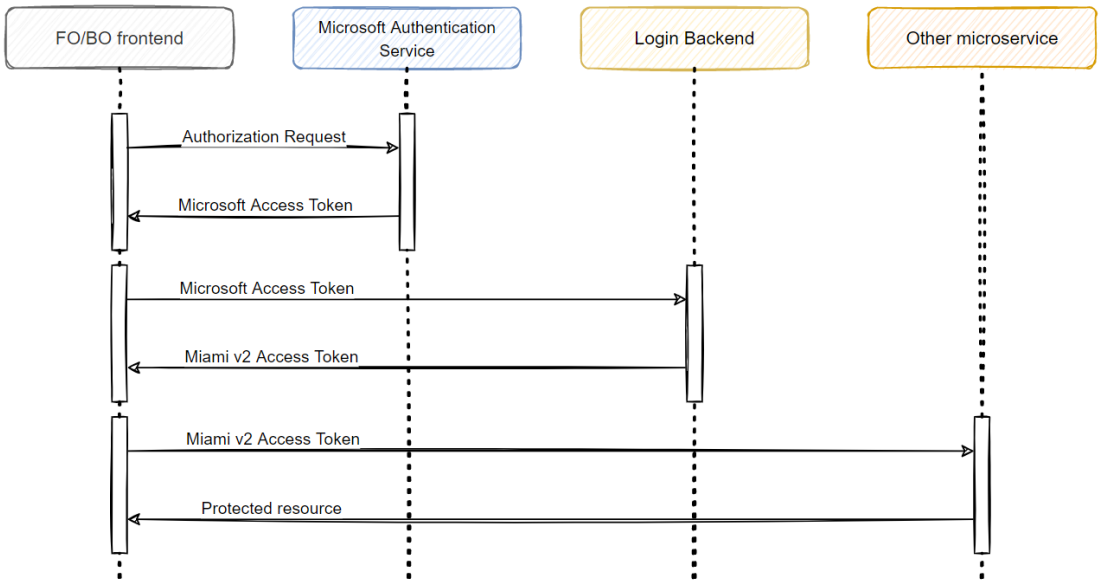

Figure 3.16: Schematization of Login System

If so, the additional data is added to a new token created and signed by our Login Backend, to be then sent back to the frontend. The final token will be finally used together with all the requests coming from the frontend, so as to be able to verify the roles, and consequently the access level, of the user in question.
In case the user does not exist yet, the user object in our database is created by the Login Backend, but since we cannot know which roles the user holds, he will not be able to fully log in yet. A high-level user (such as HR or Tenant Adrministrator) can subsequently assign roles to the user. From that moment on, the new user will have access to our application.

3.5 Asynchronous logs queue

This section is dedicated to explain the last peculiarity related to our cloud architecture scheme (figure 3.11): the Azure Functions.

This particular configuration was suggested to us by the technical supervisor of the internship, who is also a Software Architectures

Expert, in relation to a feature demanded by the clients in a later stage of development.

The request was for the capability to keep meticulous track of all the operations that were performed within the Back-Office portal, thus creating logs containing author, action, date, time, and tenant.
The functionality was apparently not complex, but there were several problematic factors to consider.

First and foremost, *time*. We did not have time to create a new backend to manage the logs independently, since when the new requirement came in there was only 1 month left before the MVP delivery, and still many things had to be done. Creating a new backend could take at least two team members occupied for a whole week, considering configurations in terms of testing and coverage, the implementation of the pipeline, the addition of the new backend in Terraform, and the creation of the Controller itself.
Then there was the *reliability* factor. We suggested adding, for each operation worth of being logged, a communication step with the database, so as to send the log of the operation just performed. However, there is a risk in this solution, as a communication with an external component could always cause some problem, it is better to minimize them. In this scenario, a failed communication with the database for logging could cause a whole, potentially much more important operation, to fail.
We could have created a handling system for these errors, but since logging logs could be very frequent, there was a possibility of affecting the performance of our backends.
After an analysis of the possibilities, the solution was: **asynchrony**.

The following diagram, represents the set of components involved in the final solution, which is an implementation of a message broker, through the Azure Storage Queue, and an Azure Function.

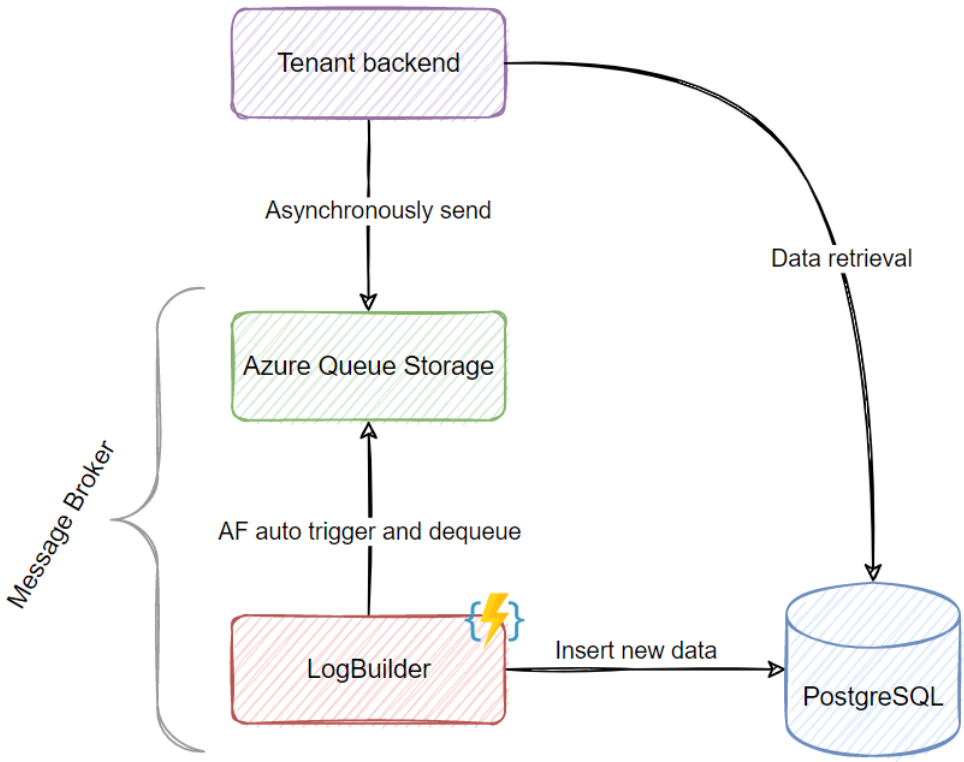

Figure 3.17: Message broker with AQS and AF

First of all, what is a message broker (MB)?
A MB is an architectural component in a communication network, whose job is to act as an intermediary between two or more systems in order to make them independent and to manage error handling itself, to ensure receipt of the message by the receiver and, in certain cases, to balance the workload.

To explain the life cycle of a log object, we can start at the Tenant backend, where a call has just been made to an endpoint.
A message composed as a string with the necessary information separated by a semicolon is sent to the Azure Queue Storage (AQS) [28].
The AQS is a service to which a large amount of data can be sent, which is then stored in the form of a queue. The queue can later be processed asynchronously.
At this point Azure Functions (AF) [29] come into action. As mentioned

earlier, Azure Functions are functions that live independently within the cloud. These functions can be triggered by a variety of conditions (e.g., at periodic time intervals, by a specific HTTP request, at the update of a database, at the receipt of a new message in an AQS, and many others) and, like normal functions, they can receive parameters and return something.

```
12      [FunctionName("LogBuilder")]
13      public async Task Run([QueueTrigger("storagequeue", Connection = "AzureWebJobsStorage")]string myQueueItem, ILogger log)
14      {
15          try {
16              log.LogInformation($"C# Queue trigger function processed: {myQueueItem}");
17
18              string cs = GetEnvironmentVariable("PostgresqlConnectionString");
19              using var con = new NpgsqlConnection(cs);
20
21              con.Open();
22
23              String[] logInfos = myQueueItem.Split(";");
24
25              String dateTime = logInfos[0];
26              String performer = logInfos[1];
27              String type = logInfos[2];
28              String method = logInfos[3];
29              String description = logInfos[4];
30              String tenantName = logInfos[5];
31
32              String sql = String.Format("INSERT INTO log (date_time, performer, type, method, description, tenant_name) "+
33                                         "VALUES ('{0}', '{1}', '{2}', '{3}', '{4}', '{5}');",
34                                          dateTime, performer, type, method, description, tenantName);
35              using var cmd = new NpgsqlCommand(sql, con);
36              await cmd.ExecuteNonQueryAsync();
37
38              log.LogInformation("Queue Trigger executed successfully.");
39          }
40          catch(Exception ex) {
41              log.LogInformation(ex.ToString());
42          }
43      }
```

Figure 3.18: LogBuilder AF

In our case, we can observe in Figure 3.18, the LogBuilder function, written in C#.
When a message is added to the AQS, the function is executed thanks to the predefined QueueTrigger. The message is removed from the queue and passed by parameter to the LogBuilder. Internally, a connection is established with the PostgreSQL database, the data extrapolated from the message, and a simple query executed. The function returns no value.

At a later time, the Tenant backend will be able to access the logs directly from the database.

Chapter 4

Conclusion

The purpose of this internship was dual. Alten needed a rework of one of its core management software programs, and to experiment with the "learn as you work" training model in a setting that simulated that of a real work environment.

During the first period, we had the opportunity to do several workshops with experts in topics we needed in our work, as Agile and DevOps methodology, Cloud Architectures and Testing.
These workshops and the sporadic help from technical leaders enabled us, even though we were the only ones responsible for all technical and functional decisions in the project, to be on the right track right from the start.

As I have explained in this paper, the team was responsible for creating from scratch an application, called Miami v2, whose purpose was to replace the outdated version of the software that permitted the HR department and Business Managers to manage Consultants, Clients and their association, along with other new features such as the Alten Grains section, and other planned but not yet implemented features, such as to the management of hiring Candidates.

The project began with the design of the program architecture at the software level and then at the cloud level.

Thereafter, each of us had the opportunity to experiment with new technologies in each domain in our development context, and then having the chance to focus on his or her favorite field.

In addition, by rotating and impersonating the 3 core roles in a DevOps team (developer, scrum master, product owner) throughout the duration of our project, we were able to better apply and assimilate the concepts of this methodology.

This way of learning is, in my opinion, extremely effective. Living in an environment that was not completely relaxed gave us the motivation to always want to do better. To present a quality product to our managers and senior devs. This led us to collectively improve in the areas where we were most lacking.
Being the only ones responsible for our work, without a leader, or someone more responsible than the others, every problem was addressed as a team, and solved together.
This created a strong feeling of belonging and unity, and it is this feeling of "moving forward together" that made our internship and project, in my opinion, a success.

Bibliography

[1] https://bluemogulenterprise.com/advantages-of-cloud-storages/ (cit. on p. ii).

[2] https://www.atlassian.com/devops (cit. on pp. iii, 13).

[3] https://www.alten.com/ (cit. on p. 1).

[4] https://www.scrum.org/resources/what-is-scrum (cit. on p. 3).

[5] https://orangematter.solarwinds.com/2022/03/21/what-is-devops/ (cit. on p. 15).

[6] https://www.atlassian.com/software/jira (cit. on p. 15).

[7] https://www.guru99.com/software-testing-introduction-importance.html (cit. on p. 20).

[8] https://medium.com/serverless-transformation/bridge-integrity-integration-testing-strategy-for-eventbridge-based-serverless-architectures-b73529397251 (cit. on p. 21).

[9] https://jestjs.io/ (cit. on p. 22).

[10] https://karma-runner.github.io/latest/index.html (cit. on p. 22).

[11] https://junit.org/junit5/ (cit. on p. 24).

[12] https://site.mockito.org/ (cit. on p. 24).

[13] https://www.selenium.dev/ (cit. on p. 26).

[14] https://docs.gitlab.com/ee/ci/ (cit. on p. 28).

[15] https://kubernetes.io/docs/concepts/overview/ (cit. on p. 33).

[16] https://phoenixnap.com/kb/kubernetes-objects (cit. on p. 34).

[17] https://medium.com/google-cloud/kubernetes-101-pods-nodes-containers-and-clusters-c1509e409e16 (cit. on pp. 34–36).

[18] https://helm.sh/ (cit. on p. 37).

[19] https://k8slens.dev/ (cit. on p. 40).

[20] https://c4model.com/ (cit. on p. 42).

[21] https://www.pluralsight.com/blog/software-development/relational-vs-non-relational-databases (cit. on p. 49).

[22] https://en.wikipedia.org/wiki/Database_normalization (cit. on p. 49).

[23] https://medium.com/hackernoon/sql-vs-nosql-what-is-better-for-you-cc9b73ab1215 (cit. on p. 50).

[24] https://www.terraform.io/intro (cit. on p. 56).

[25] https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/resources/postgresql_database (cit. on p. 58).

[26] https://www.terraform.io/cli/commands/init (cit. on p. 58).

[27] https://learn.microsoft.com/en-us/windows-server/identity/ad-ds/get-started/virtual-dc/active-directory-domain-services-overview (cit. on p. 62).

[28] https://learn.microsoft.com/en-us/azure/storage/queues/storage-queues-introduction (cit. on p. 66).

[29] https://learn.microsoft.com/en-us/azure/azure-functions/functions-overview (cit. on p. 66).