POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Design and Implementation of a Microservices-Based Serialization System

Supervisors
Prof. Stefano Quer

Company
AROL Group

Candidate
Alessio Chessa

October 2024

Summary

Digital innovation is transforming tools and processes across all industries, and adapting to these changes is essential for companies that aim to stay competitive in their field. Integration of new technologies, tools and software can significantly improve communication between businesses and clients, speed up crucial processes, and increase overall productivity.

One critical internal process affected by this digital transformation is product serialization. An effective serialization system simplifies the tracking a product's lifecycle for company operators and provides customers with accessible product information. Therefore, enhancing this process can lead to improvements in production management and customer relationship management.

This thesis focuses on the development of a system to digitalize the serialization process and to support business-customer interactions through a customer portal for AROL company. Serialization allows operators to generate a unique QR code for each product, which can be scanned by customers or operators to obtain product-specific information, differentiated based on user permissions. To achieve these goals, the system consists of a frontend web application, discussed in a parallel thesis by another PoliTo student, and a backend system, which is the main focus of this project.

Following the current best practices in modern web application development, the backend system will be built using a microservices architecture. It will not only manage product serialization, but also provide functionalities to support a customer portal for client interactions. A crucial feature of this system is user verification and permission management, which determines what information users can access by scanning a QR code and what operations they are authorized to perform. This security layer is handled by an IAM (Identity and Access Management) microservice. Additionally, the system includes a machinery microservice that manages AROL product data, enabling users to fetch information about specific products. The ticket microservice, which constitutes the core of the serialization mechanism, translates unique string identifiers into internal serial codes and vice versa. These identifiers are then used to generate QR codes for the corresponding AROL products.

The thesis covers the design and implementation of these microservices and concludes with the release a proof-of-concept software that will be evaluated by the company to assess its feasibility and potential integration into the existing IT ecosystem.

Contents

List of Figures

List of Tables

Acronyms

ACID Atomicity, Consistency, Isolation, Durability.

ACR Azure Container Registry.

AMQP Advanced Message Queuing Protocol.

API Application Programming Interface.

AWS Amazon Web Services.

BC Bounded Context.

CI/CD Continuous Integration and Continuous Deployment.

CLI Command-Line Interface.

CRUD Create, Retrieve, Update, Delete.

DB Database.

DBMS Database Management System.

DDD Domain-Driven Design.

DevOps Development and Operations.

DI Dependency Injection.

DNS Domain Name System.

HPA Horizontal Pod Autoscaler.

HTTP HyperText Transfer Protocol.

IAM Identity and Access Management.

JSON JavaScript Object Notation.

JWT JSON Web Token.

KPI Key Performance Indicator.

ORM Object-Relational Mapper.

OS Operating System.

R&D Research and Development.

RBAC Role-Based Access Control.

REST Representational State Transfer.

SDK Software Development Kit.

SPA Single Page Application.

URI Uniform Resource Identifier.

URL Uniform Resource Locator.

UUID Universally Unique Identifier.

YAML Yet Another Markup Language.

Chapter 1

Introduction

In today's interconnected world, the industrial sector is making efforts towards digital transformation and companies must adapt in order to remain competitive. This shift towards digital integration is often referred to as "digital transition".

Digital tools not only can enhance interactions between customers and companies, but also they allow business processes to be more efficient and less time-consuming. Operations such as storing and retrieving product information within seconds, tracking product history, and quickly identifying products are just a few examples of how digital transformation can benefit companies.

These advancements can result in increased user satisfaction and higher throughput in company operations.

1.1 AROL

AROL Closure Systems is an Italian company specializing in capping solutions, founded in 1978 in Canelli (AT)[1]. Over the years, AROL has expanded globally and now has many branches all over the world to meet growing international market demands.

The company designs and manufactures capping solutions for its customers, with machinery produced and assembled in AROL plants. An example of capping machine built by AROL can be seen in Figure 1.1. These products are then delivered to customers, with AROL also providing installation and after-sales services[2].

AROL Closure Systems company is part of AROL Group, which also includes: MACA Engineering, specialized in machines for the production, assembly and cut of aluminium and plastic caps and closures; Tirelli, specialized in machines for the cosmetics and home and body care industry, and Unimac-Gherri, specialized in filling and capping of containers with twist-off caps for dense, semi dense and pasty products[3].

Figure 1.1. EURO PK: example of AROL capping machinery[4]

1.2 Project Aim: Enhancing Digital Integration

In line with its commitment to innovation, AROL aims to develop a web portal accessible to customers, AROL operators, and external stakeholders interested in AROL's products. A key feature of this application will be the ability to scan a QR code on an AROL machine or component and retrieve the corresponding information based on the user's role. Authorized AROL operators will have access to detailed and restricted data, while customers will access general product information. External users will be redirected to the company's landing page without accessing specific machine data.

This service will be implemented as a web application accessible via a dedicated URL. The aim of this thesis is to develop and deploy the back-end infrastructure necessary to support the front-end application and to expose an API for administrators to perform restricted management operations. The service must be globally reachable, guaranteeing both availability (i.e., the system is operational most of the time) and reliability (i.e., operations are performed without failures).

1.2.1 Digital Integration Benefits

The QR code project offers significant benefits from both a horizontal perspective, which is related to the primary business activities of the company, and a vertical perspective, which includes the activities that support those primary functions. This categorization can be traced back to Michael Porter with his famous **value chain** concept, which differentiates between these two types of activities in a business (see Figure 1.2).

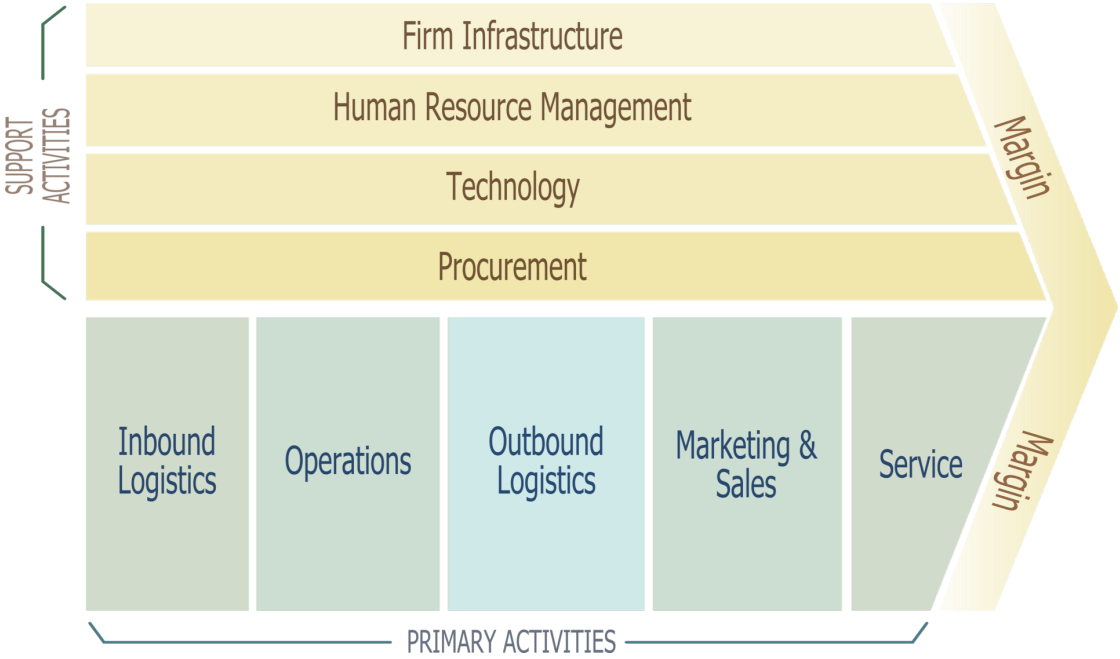

Figure 1.2. Porter value chain representation[5]

The vertical and horizontal axes are closely related, with both contributing to the creation of value for the company. The QR code serialization mechanism improves the efficiency of horizontal activities by supporting accurate tracking of AROL products. For example, warehouse operators can identify a product almost instantly by scanning its QR code, a process that is significantly quicker and more efficient than manually searching for shipping documents or internal codes to retrieve product information.

From a vertical perspective, providing a QR code to customers means offers them easier access to product data. Instead of manually typing a long identifier into their computer, users can simply scan the QR code to be quickly directed to the relevant product information, enhancing the user experience and saving time[6].

The vertical and horizontal axes intersect when the internal product-tracking system used for operational efficiency also supports the customer portal. The same serialization

mechanism used to track products within the company can be employed to provide customers with the information they are looking for. In this way, a single QR code can improve internal workflows and strengthen the customer relationship.

Chapter 2

Requirements

The first step in the project is to define the requirements for the system, producing the necessary artifacts that can be used to design the architecture later on. From the discussion with the stakeholders, some important features of the application came out and they are here described.

2.1 Informal description

The system must support a customer portal, serving as an interface between AROL and its customers, as well as a serialization mechanism used by both AROL operators and the portal. This serialization mechanism allows AROL operators to identify products during their work activities, and the portal leverages it to display product-specific information to users.

The serialization mechanism is based on QR codes. Authorized AROL operators can generate new QR codes for specific products by assigning a unique serial code to each product. Both AROL customers and operators can authenticate through the portal and scan QR codes within the application to retrieve the corresponding product information, if the QR code is valid. Operators are granted access to more detailed product information that is not visible to customers. Additionally, external users who are not authenticated in the system can also scan a QR code. In such cases, they are redirected to the AROL portal's main page, where they can explore available products. System administrators have the highest privileges, allowing them to manage users, products, and QR codes.

2.2 Stakeholders and Context Diagram

After collecting an informal description of the project, we started with analyzing the requirements to produce more useful artifacts that can support us in the next steps. One of the first things to do is identifying the actors (or stakeholders) that may interact with the system, without going into the details of what operations they perform. The stakeholders that we identified in the system are four:

- **AROL system administrator**: manages the system;
- **AROL operator**: works for AROL as operator that can use the system to manage QR codes and products;
- **Customer**: owns AROL products and can use the system to obtain information about those products;
- **External user**: unknown to AROL systems, wants to discover AROL products.

The context diagram of the system that is represented in Figure 2.1 helps to easily identify the actors at a glance.

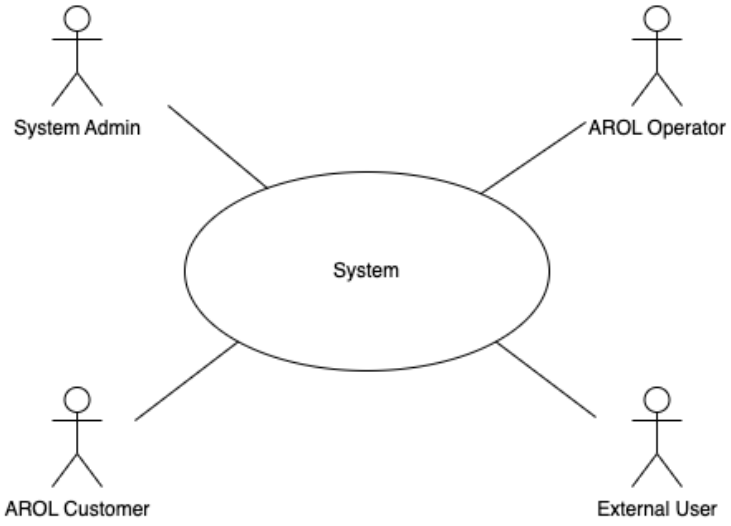

Figure 2.1. Context Diagram of the system

2.3 Functional Requirements

Functional requirements help defining the actual operations that the system should perform in a more formal way. We gathered the list of all the functional requirements in the Table 2.1. There are four macro functionalities (FR1, FR2, FR3, FR4) that group common functional requirements into a single one that represents the overall operations. For example, FR2 (User management) includes the operations on the user entities, such as creation, update, retrieval and deletion. This list of functional requirements is extremely useful during the phase of architecture design, because they can be delegated to specific parts of the system and they provide a guide to test their functionality.

Requirement	Description
FR1	**User authentication**
FR1.1	Login
FR1.2	Logout
FR1.3	Registration
FR2	**User management**
FR2.1	Create user
FR2.2	Retrieve user
FR2.3	Update user
FR2.4	Delete user
FR3	**Product management**
FR3.1	Create product
FR3.2	Retrieve product
FR3.3	Update product
FR3.4	Delete product
FR4	**QR code management**
FR4.1	Create QR code
FR4.2	Retrieve information from QR code
FR4.3	Update QR code
FR4.4	Invalidate QR code

Table 2.1. Functional requirements of the system

2.4 Use Cases

In order to understand how the system should behave under different circumstances, it is essential to outline the typical usage flows, also known as **use cases**. A use case is a structured description of an interaction between the actors and the system to achieve a specific goal or complete a task. Use cases are important in the requirements' analysis, because they help to clarify system behavior from an end-user perspective while focusing on the tasks that the system must support. Additionally, they guide developers and stakeholders in verifying that the system satisfies the intended requirements.

In this context, the system is treated as a single entity, without making distinctions among the various microservices that compose it. Each use case provides a high-level description of the interaction between external actors and the system to fulfill a particular function or operation.

Use cases can be further decomposed into more detailed **scenarios**, which describe specific paths through the system based on varying conditions or inputs. These scenarios allows to explore different flows, error-handling paths, and edge cases that ensure the system's robustness. Typically, a scenario includes: **pre-conditions**, conditions that must be satisfied for this scenario to be reachable, **post-conditions**, which represent the state

of the system after the scenario ends, and a **goal**, which is as brief description of what we expect to achieve.

All use cases are represented in a **use case diagram** that connects the actors with the use cases they are involved in[7]. The diagram provides an overall view of the system features and the various scenarios. The use case diagram of the system is depicted in Figure 2.2

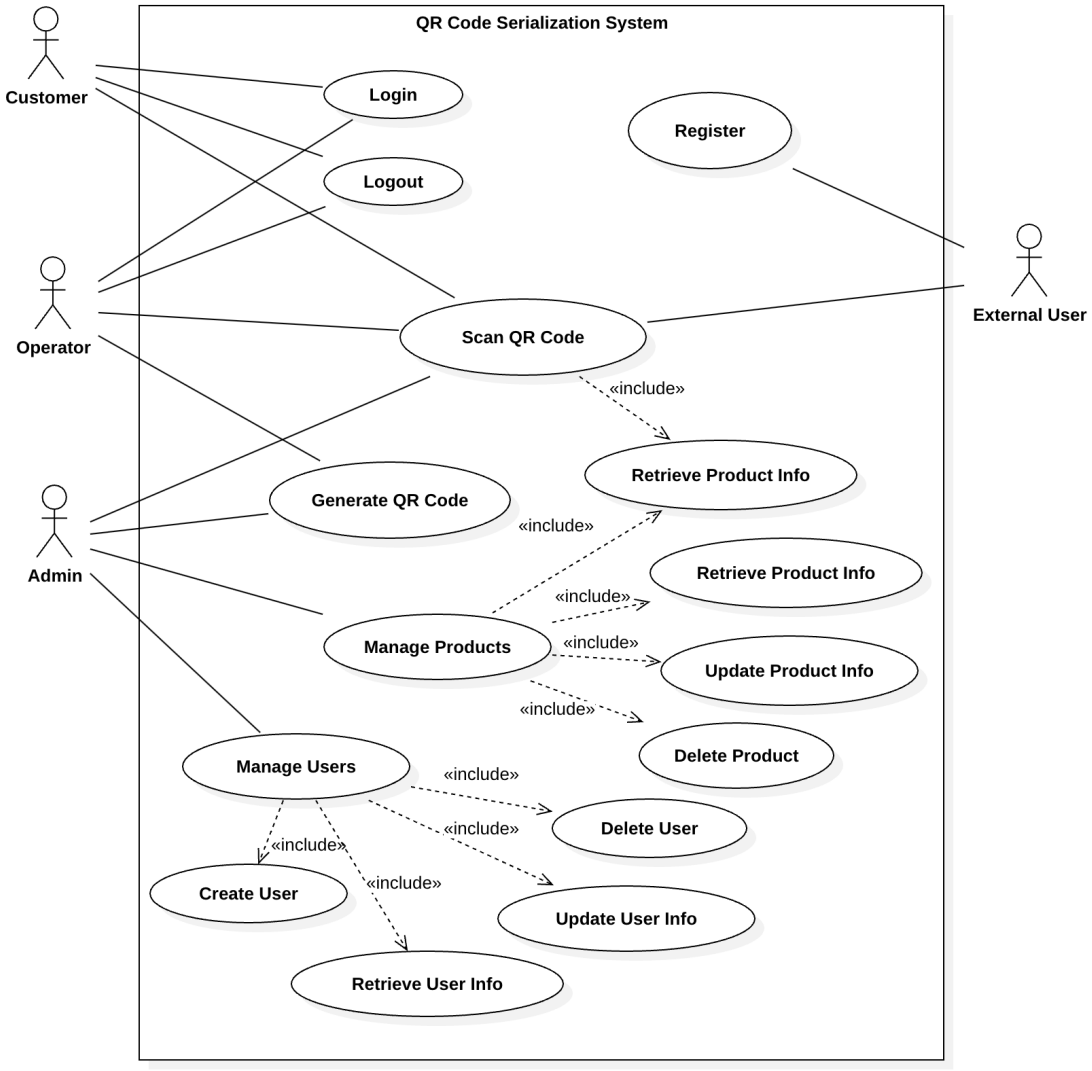

Figure 2.2. Use Case Diagram

2.4.1 Use Case 1: User Registration

This use case describes the steps required for an external user, not registered in the system yet, to create a new account in the AROL Customer Portal. The user is required to enter some basic information, a unique username, email, and a secure password. The system checks if the inserted data is valid and if there is no other user with the same username or email, then registers the new user into the system. More details are included in Table 2.2.

Actors	External User
Goal	Register a new user into the portal
Preconditions	The user must not have an existing registered account
Main Scenario	1. The user navigates to the registration page. 2. The user enters their new credentials and required data. 3. The system verifies the validity of the data. 4. The system creates a new user with the specified data. 5. The user can log in after successful registration.
Alternate Scenarios	Invalid Input Data, User Already Registered

Table 2.2. Use case 1 - User Registration

2.4.2 Use Case 2: User Login

This use case describes the process of logging an existing user (Customer, Operator, or Administrator) into the AROL Customer Portal. The user must provide valid credentials (email/username and password), and the system verifies this information to grant access with the appropriate permissions. If the user inserts wrong credentials or is not registered yet, the system returns an error. The Table 2.3 describes the steps in more detail.

2.4.3 Use Case 3: User Logout

This use case details the steps required for a logged-in user (Customer, Operator, or Administrator) to log out of the system. The user must be logged in first, otherwise the logout operation will not have any effect. More details are included in Table 2.4.

Actors	Customer, Operator, Administrator
Goal	Sign in an existing user into the portal
Preconditions	The user must have an existing registered account
Main Scenario	1. The user navigates to the login page. 2. The user enters their credentials. 3. The system verifies the validity of the credentials. 4. The system grants access with appropriate permissions. 5. The user can log out.
Alternate Scenarios	Invalid Credentials

Table 2.3. Use case 2 - User Login

Actors	Customer, Operator, Administrator
Goal	Sign out a logged-in user
Preconditions	The user must be logged in
Main Scenario	1. The user requests to be logged out of the system. 2. The system invalidates the current session. 3. The user can log in again.
Alternate Scenarios	Not Authenticated

Table 2.4. Use case 3 - User Logout

2.4.4 Use Case 4: QR Code Generation

This use case outlines how an Operator or Administrator can generate a QR code for a specific product in the system. The system links the generated QR code to the product's serial code and part number. More details are included in Table 2.5.

2.4.5 Use Case 5: Scan QR Code (Authenticated User)

This use case describes how authenticated users (Customer, Operator, or Administrator) can scan a QR code to retrieve product information. The system ensures that operators and administrators have access to more detailed information than customers. More details are included in Table 2.6.

Actors	Operator, Administrator
Goal	Generate a QR code for a specific product
Preconditions	The operator or administrator must be logged in
Main Scenario	1. The user navigates to the QR code management section. 2. The user selects to create a new QR code. 3. The user enters the product's serial code and part number. 4. The system generates a QR code and associates it with the product.
Alternate Scenarios	Invalid Serial Code, Duplicate QR Code

Table 2.5. Use case 4 - QR Code Generation

Actors	Customer, Operator, Administrator
Goal	Scan a QR code and retrieve product information
Preconditions	The user must be logged in
Main Scenario	1. The user scans a QR code using the portal. 2. The system verifies the validity of the QR code. 3. The system retrieves and returns product information. 4. Operators are shown additional details not visible to customers.
Alternate Scenarios	Invalid QR Code, Unauthorized Access

Table 2.6. Use case 5 - Scan QR Code (Authenticated User)

2.4.6 Use Case 6: Scan QR Code (Unauthenticated User)

This use case covers how external users (not logged into the portal) can scan a QR code to access basic information about products on the AROL portal. These users are limited to general browsing unless they register. More details are included in Table 2.7.

Actors	External User
Goal	Scan a QR code without authentication
Preconditions	The user is not logged in to the portal
Main Scenario	1. The user scans a QR code. 2. The system verifies the QR code's validity. 3. The system redirects the user to the main page of the AROL portal. 4. The user can explore available products but cannot access specific product details. 5. The user can register into the system.
Alternate Scenarios	Invalid QR Code

Table 2.7. Use case 6 - Scan QR Code (Unauthenticated User)

2.4.7 Use Case 7: Manage Products (Create, Retrieve, Update, Delete)

This use case describes how an administrator manages product information in the system. The administrator can add, retrieve, update, or delete products as needed. The system ensures data validity and prevents duplication. It returns an error if the user is not logged in or does not have permissions. More details are included in Table 2.8.

2.4.8 Use Case 8: Manage Users (Create, Retrieve, Update, Delete)

This use case describes how the administrator can create, retrieve, update, or delete user information. Administrators can assign different roles (Customer, Operator, Administrator) when creating users. It returns an error if the user is not logged in or does not have permissions. More details are included in Table 2.9.

Actors	Administrator
Goal	Retrieve, update, or delete products
Preconditions	The administrator must be logged in
Add New Product	1. The administrator enters required product data. 2. The system verifies the validity of the data and possible duplication. 3. The system saves the new product.
Retrieve Product	1. The administrator searches a product by its identifier. 2. The system shows the information about the product.
Update Product	1. The administrator searches a product by its identifier. 2. The system shows the information about the product. 3. The administrator enters the new product information. 4. The system verifies the validity of the data. 5. The system saves the changes to the product.
Delete Product	1. The administrator searches a product by its identifier. 2. The system shows the information about the product. 3. The administrator chooses to delete the product from the system. 4. The system removes the product.
Alternate Scenarios	Product Not Found, Unauthorized

Table 2.8. Use case 7 - Manage Products

Actors	Administrator
Goal	Retrieve, update, or delete user information
Preconditions	The administrator must be logged in
Create New User	1. The administrator enters required user's information and credentials, choosing a role. 2. The system verifies the validity of the data and possible duplication. 3. The system saves the new user.
Retrieve User Information	1. The administrator searches a user by its username or identifier. 2. The system shows the information about the user.
Update User Information	1. The administrator searches a user by its username or identifier. 2. The system shows the information about the user. 3. The administrator enters the new user information. 4. The system verifies the validity of the data. 5. The system saves the changes to the user.
Delete User	1. The administrator searches a user by its username or identifier. 2. The system shows the information about the user. 3. The administrator chooses to delete the user. 4. The system removes the user.
Alternate Scenarios	User Not Found, Unauthorized, Duplicated User

Table 2.9. Use case 8 - Manage Users

Chapter 3

Architecture

Designing a distributed system is a complex task[8], that needs to careful analyze the requirements, both functional and non-functional, to create an architecture that is not only robust but also scalable and maintainable. In the initial phase we identified the requirements of the system to have an idea of how it should work and what should be its main features. To summarize, we identified some high level functionalities based on the project requirements:

1. User authentication and authorization
2. User management
3. Machinery and parts management
4. QR code creation and information retrieval

These four points represent distinct concerns and could logically correspond to different modules of the architecture. The first two about users identification and management are tightly related, so we decided to handle them in the same module.

After having outlined the core modules, we evaluated the type of architecture to adopt in the project, considering the two most common options for this type of system: a monolithic approach and a microservice-based approach.

3.1 Monolithic architecture

The monolithic architecture represents the traditional approach to application design[9], where a single application performs all the tasks defined previously as a single process running on a machine. While this approach may simplify initial development and deployment, it often leads to significant challenges as the application grows.

In a monolithic system, modules are tightly coupled, thus any change to a module requires redeploying the entire application, which can be risky and expensive as the codebase expands. Furthermore, as teams working on different modules must coordinate to avoid conflicts, development and release speed can drastically slow down. This approach also limits the ability to use different programming languages or technologies for different modules, as a monolithic application is usually developed using a single technology stack.

3.2 Microservices architecture

A modern approach that has been increasingly adopted to build distributed systems is the microservices architecture[10]. Instead of a single, monolithic application, the system is broken down into smaller, independent sub-systems, each focusing on a specific business functionality, known as microservices[11]. As depicted in the Figure 3.1, in a monolithic system there is a single database for all the modules, and no independence between modules. In the microservices architecture instead, modules are independent and use their own data source.

Each of them operates as an independent process running on a machine, which can be developed, deployed and scaled independently. Despite the name, microservices can vary in size and complexity. This type of architecture requires careful design, ensuring that each service follows some general principles, such as separation of concerns, well defined interfaces, and single responsibility principle[12].

3.2.1 What is a microservice?

We can think of the entire system as a collection of smaller, autonomous services that interact with each other over well defined interfaces. Each microservice typically represents a specific business capability or domain, not strictly related to the others, allowing it to be developed, deployed, and scaled independently by different teams.

The concept of microservice is closely related to the idea of a **Bounded Context (BC)**[13] in the **Domain-Driven Design (DDD)**. According to Martin Fowler, DDD is "an approach to software development that centers the development on programming a domain model that has a rich understanding of the processes and rules of a domain"[14]. Applying DDD principles can often help define the boundaries of a microservice, as these boundaries generally align with the bounded contexts' ones. This approach also highlights the independence of each microservice and the points of contact between them, which might help defining the communication interfaces.

Another important aspect of a microservice is also the independence of data, meaning that the data stored and used by one microservice is not directly accessible by others but can only be accessed through the interfaces exposed by that microservice. This independence simplifies the process of updating the internal implementation of a service or even of replacing an entire service, provided that the interface contracts remain unchanged.

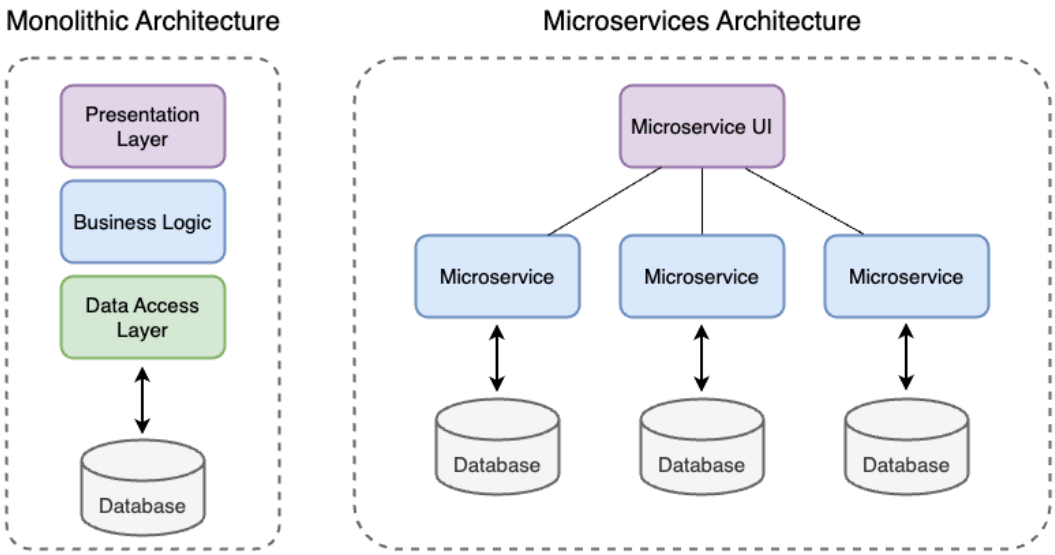

Figure 3.1. Monolithic architecture vs. microservice-based architecture

3.3 Monolithic vs. Microservices

The decision to adopt a monolithic or microservices architecture should be carefully evaluated from both a technical and an economic perspective. Developing and deploying a monolithic application usually requires fewer resources and less time to release, but it may present challenges in the long term. For example, if one module needs to scale, the entire application must be scaled, leading to increased resource usage and higher costs. Additionally, expanding or updating a monolithic application can be difficult due to tightly coupled modules and a change in one part of the code can have effects to other parts. This can slow down development and cause delays in meeting release deadlines.

On the other hand, adopting a microservice-based approach requires changes not only in the infrastructure but also in the company's working culture and organization. Each microservice should be assigned to a different team, with both the microservices and the teams operating independently. This does not mean that teams do not communicate, but rather that they can have independent meetings, processes and so on.

From a technical standpoint, building a microservices-based application requires an architecture that can support it. The infrastructure must allow for scalability, orchestration of microservices, monitoring, health checks, and Continuous Integration and Continuous Deployment (CI/CD) pipelines[15], often within a DevOps culture[16].

In summary, the monolithic approach requires less initial effort but may present scalability and maintainability issues in the long run. In contrast, the microservice-based approach may require more effort and resources initially but in the long term it offers better scalability, faster feature delivery, and fewer coordination problems among teams.

For the purposes of this thesis, after careful consideration with the company, we have decided to adopt a microservice-based approach. The goal is to build a scalable and maintainable infrastructure that can grow with the company's needs and to evaluate whether the benefits of this type of architecture justify the initial investment. Given that AROL has thousands of machines distributed globally, a microservices architecture is well-suited to meet these demands. It allows for easy scaling and distribution across multiple nodes, ensuring high performance and availability from all locations.

3.4 Proposed solution

To meet the requirements outlined by the company, we came up with a possible solution that is here presented. As explained in the previous section, the application is based on microservices architecture and each module handles a specific business area. We identified three main microservices that are able to cover all the functionalities defined during the requirements' collection phase.

The first one is the **Identity and Access Management (IAM)**, that handles the authentication and authorization in the system. As this is a cross-cutting concern, it is beneficial to delegate it to a dedicated microservice like the IAM. This service allows users to register, login and logout, but also manages their permissions for resource access. System administrators can use this microservice to manage users' information and much more. Without exploring some technical details that will be covered in the next chapter with the proper attention, we can say that this microservices follows the most common way to deal with authorization, that is by using objects called tokens. Those special objects are generated and signed by the IAM and are then used in all the microservices to claim and verify user identity.

The **Machinery** microservice has the responsibility to store the information about all the machines and their components, exposing functionalities to add, update, remove but also retrieve product-specific data. Based on users' permissions it will return the appropriate amount of information.

Finally, there is the **Ticket** microservice, that enables the actual serialization and identification of products through their QR codes. This service allows to create and manage **tickets**, entities that map a unique identifier to a machine or component stored in the Machinery microservice. The generated identifier is then used to generate the corresponding QR code in the frontend application.

As depicted in the Figure 3.2, the independence of microservices is guaranteed also

by data independence. This is achieved through the use of a separate database for each context, meaning that different services are connected to different databases. While this approach leads to more challenges with data consistency across all the system, it ensures that different microservices remain available even when a failure occurs in one data source.

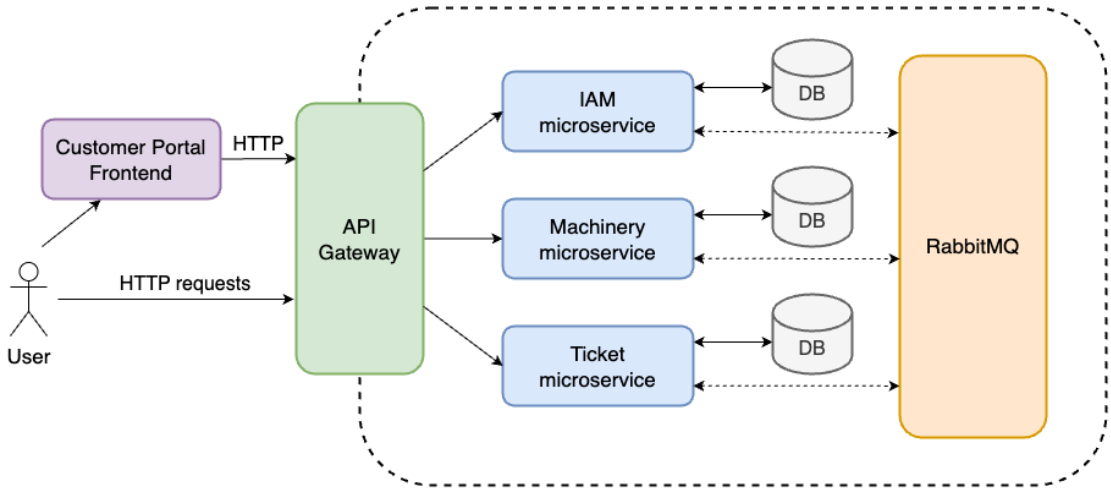

Figure 3.2. Architecture Overview Diagram

Each of these microservices exposes a REST API through which it can be accessed via HTTP requests. Microservices are not directly exposed to the external world, but all requests are routed through a single entry point, known as the **API Gateway**, which redirects them to the appropriate microservice. The requests can be performed by the Customer Portal frontend application, that provides a user friendly interface for interacting with the customer or the operator, or they can be directly sent using specific tools without using the web application. We decided to keep this last option open so that system administrators can perform management operations that are not supported in the Customer Portal. Additionally, it can also benefit new applications that may want to leverage on the system's functionalities through HTTP protocol. Internally, microservices communicate with each others using an asynchronous messaging system, rather than synchronous HTTP protocol as used for external requests. This approach contributes to reducing coupling and increasing the system's resilience.

Each microservice can be deployed independently and horizontally scaled as needed, depending on available resources and request volume. For example, the IAM service may need to be scaled more frequently than others due to high number of requests for user registration, login and logout. In such cases, only the IAM service would be scaled, rather than all services. For this reasons, a microservice is deployed as an independent process inside a **Docker container**, ensuring complete isolation between microservices, even if they run on the same machine. More details about Docker are covered in the next chapter.

Chapter 4

Technologies and Tools

There are many programming languages and tools available for developers, and each one of them has pros and cons. It is important for a developer to know which tools to use depending on the circumstances. The right choices can significantly impact the efficiency, scalability, and maintainability of the application. For this reason, in this chapter are described all the tools selected for this project, with some details about every one of them and the reason behind each choice.

4.1 .NET Platform and C# language

The first choice was about what language to use to develop each microservice. In this case, the most sensible approach was to use a single language for all of them, because there would be just one developer on the project. This led to the decision to use C# and the .NET platform, which is aligned with the existing technological stack within the company, guaranteeing easier integration with other existing systems and facilitating future improvements.

.NET is an open-source developer platform maintained by Microsoft for building a wide range of applications, from desktop and web applications to cloud services and microservices[17]. It supports multiple programming languages, with C# being the most used among them[18]. The platform offers several features for developers, such as garbage collection, asynchronous programming support and concurrency primitives. .NET includes a runtime to execute applications, additional libraries, compilers for the supported languages, and Software Development Kit (SDK) to help building applications.[19]. .NET comes in different flavors, such as .NET Core, .NET Framework, and Mono. For this project, **.NET Core** was selected due to its cross-platform capabilities, allowing development and deployment across Windows, Linux and MacOS. .NET Core is also optimized for performance and scalability, making it an ideal fit for microservice-based architectures.

As mentioned before, C# was chosen as the programming language, because it is well supported by .NET Core and it is widely used in various scenarios, from object-oriented

to functional programming[20]. The language offers powerful abstraction, making it easier for programmers to write clean and maintainable code. Moreover, C# has a robust type system and supports many modern programming paradigms, which makes it a flexible choice for building complex systems.

.NET Core also integrate ASP.NET Core framework[21], a high performance framework that provides support for building web applications and APIs. This framework, in conjunction with C# language, simplifies the development of HTTP-based services, offering built-in support for routing, middleware, and dependency injection. This allows the developers to focus on implementing the business logic part, while leveraging on ASP.NET Core's abstractions for handling HTTP requests and responses efficiently.

4.2 Docker and Docker Compose

As explained in the previous chapter, each microservice should be independent and somewhat isolated from the others. Running it as a separated process might be enough, but usually an higher level of isolation is required, and the solution is **containerization**[22]. The concept of containers is to create an environment on top of the host operating system, adding layers to provide additional functionalities and libraries, where the process can run without directly interfering with the host Operating System (see Figure 4.1). In this way, if something bad happens, no damage will be done to the machine where the process is run, but just inside its container. This allows to run multiple containers on the same machine (or also virtual machine), thus optimizing the server resources.

Docker is a platform that allows to develop, deploy and manage applications leveraging on the containerization technology. Docker Engine constitutes the core of the platform, a powerful engine that acts as a client-server application. The Figure 4.2 provides an overview of the main components of the Docker architecture and how it works. First of all, a daemon process called "dockerd" runs as a server and exposes an API which can be used to make requests to it. To provide an easier interaction with Docker for the user, there is also a Command-Line Interface (CLI) client called "docker".[24] Every container is instantiated from a Docker image, which is an object that contains the information of how the corresponding container environment should be built. An image can be stored in a Docker registry, for example Docker Hub, or it can be built locally following a configuration written in Dockerfiles. Basically, a container can be seen as an instance of a Docker image, in the same way as a process is an instance of a program.

Another Docker client called Docker Compose[25] was used to manage multiple containers simultaneously. Docker Compose can read the configuration of containers and how to build them from a YAML file, usually called compose.yml or docker-compose.yml. The compose file can specify what services should be started, how the containers should be built, the network and volume configuration required by the application. In this way, the project's microservices can be started, stopped, and scaled together, simplifying the management of the system's lifecycle.

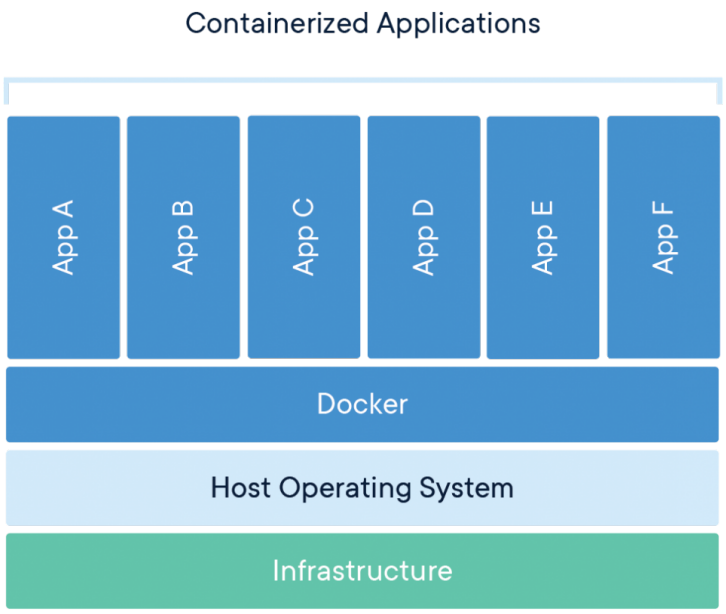

Figure 4.1. Multiple containers running on the same machine[23]

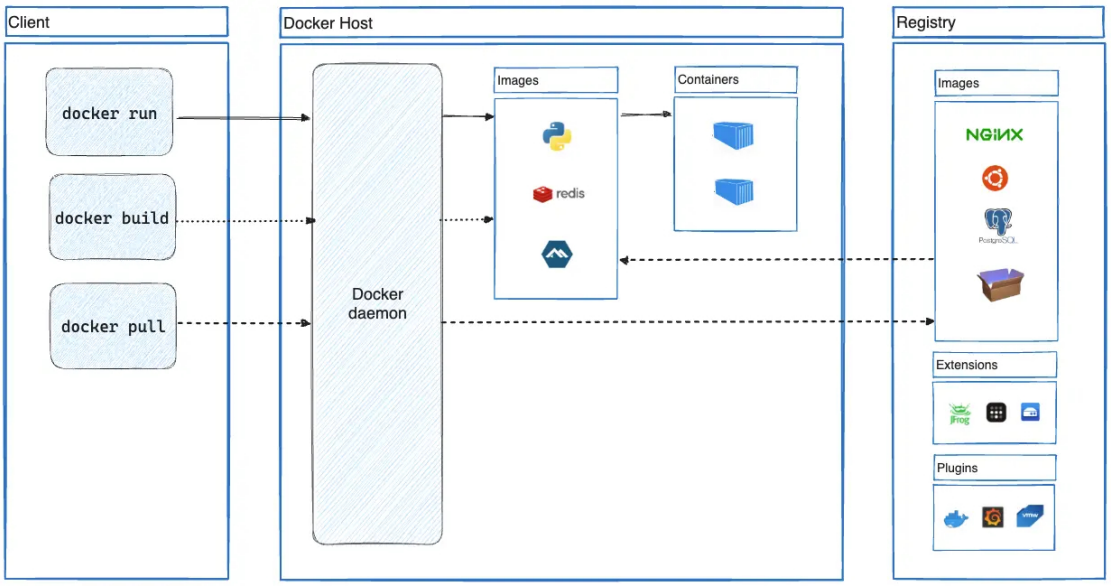

Figure 4.2. Docker architecture[26]

4.3 PostgreSQL

For persistent data storage, each microservice is connected to a PostgreSQL database. PostgreSQL is a powerful, open source relational Database Management System (DBMS)[27], known for being robust and flexible. It supports a wide range of data types, indexing techniques, and ACID transactions. In this project, each microservice is paired with its own PostgreSQL instance, deployed as a Docker container. This kind of setup allows each service to manage its own database independently, ensuring better scalability and fault isolation.

4.4 RabbitMQ

The internal communication among microservices is based on asynchronous messages, which is crucial to ensure loose coupling between them and consequently improve scalability[28]. The communication is mediated by an additional process called "message broker", that allow clients to send and receive messages.

For this project, RabbitMQ was chosen as the message broker[29]. RabbitMQ is a reliable, open source message broker that supports multiple protocols, including Advanced Message Queuing Protocol (AMQP). It is lightweight and easy to deploy, making it a perfect fit also for production. RabbitMQ is available via Docker container and microservices can communicate with it sending and receiving messages through message queues. This allows them to operate independently and handle messages at their own pace, which is especially beneficial under high loads.

4.5 Azure Container Registry

During development, Docker images are frequently updated as the corresponding microservices evolve. Those image are important because they are used to instantiate the microservices. Thus, they should be stored and managed securely and the safest choice is to use a private container registry. While public registries like Docker Hub are available, they do not offer the same level of security needed for enterprise applications.

For these reasons, **Azure Container Registry (ACR)** was chosen in this project. ACR is a private Docker registry service provided by Microsoft[30]. It offers geo-replication across different regions, Role-Based Access Control (RBAC) and other advanced features. It also supports workflows such as CI/CD.

4.6 Kubernetes

For orchestrating and managing containerized microservices, **Kubernetes** was chosen. Kubernetes is an open-source platform that automates deployment, scaling, and management of containerized applications[31]. It groups containers into units called Pods,

which can be distributed across multiple machines to ensure better resource utilization and high availability. Kubernetes also automates load balancing, scaling based on traffic, and provides self-healing capabilities, such as automatically restarting failed containers. This makes it an ideal solution for deploying and managing the microservices architecture used in this project.

4.7 Git and Bitbucket

As mentioned in the introduction, this project is developed in collaboration with another thesis project based on the corresponding front-end part. An important part in the development of an application is how to coordinate the work between team members, how to see and change the code written by the other members, and also how to check the previous versions of the same code, which sometimes might be useful in order to find where was a bug introduced and why.

This is why version control is important in software development, enabling teams to collaborate, manage code changes, and track project history[32]. Git is a distributed version control system and it is undoubtedly the most popular tool for this purpose. Git allows developers to create repositories where the code is hosted, commit changes, branch off new features or bug fixes, and merge them back into the main project, maintaining a detailed history of the project's evolution.

Bitbucket is a collaboration tool and Git-based hosting system provided by Atlassian[33]. Bitbucket offers integration with other Atlassian tools, such as Jira, facilitating seamless team collaboration and project management. The codebase for this project is hosted in a private Bitbucket repository, making sure that is is accessible only to authorized team members.

4.8 Jira

When working in a team and managing a project, it is critical to use tools that help us working efficiently. Jira is a project management tool created by Atlassian, that supports tracking issues, managing tasks, and integration with agile workflows[34].

In this project, we used Jira to track the progress, manage tasks, and document issues and bugs. The integration between Jira and Bitbucket turned out to be extremely useful to synchronize codebase changes with project's progress tracking. During regular meetings with the company, we created new tasks, assigned them to the appropriate member and monitored their status to ensure that the project progressed and remained on schedule.

Chapter 5

Microservices design and implementation

This chapter focuses on the description of the actual implementation of each one of the microservice. The requirements collection phase outlined some of the features that the system should provide and the most important use cases, that help to understand how the system should behave in different scenarios. The next step is to translate those specifications into code, building a robust code base that follows the best practices for creating web API services.

Here the most important design patterns used in the project are described, followed by a detailed explanation of the design choices behind each microservice and their internal structure.

5.1 Common patterns

Before starting, I want to talk about the common patterns used for all microservices. A **design pattern** is a possible solution for a common problem, providing a sort of guide or template to follow in order to solve that problem. Design patterns are really helpful for developers because they allow to write cleaner, more efficient, and more modular code. There are lots of patterns commonly used during development, but here only the most important ones are listed and described.

5.1.1 Controller-Service-Repository

The most important is the **Controller-Service-Repository** pattern. This simple pattern is about organizing the code in layers, and it is very effective to enhance modularity, because it enforces separation of concerns between layers[35]. Each layer calls the functions exposed by the layer below and returns some results to the layer above. This type of structure allows to change a layer without affecting to other ones, given that the interfaces between layers do not change.

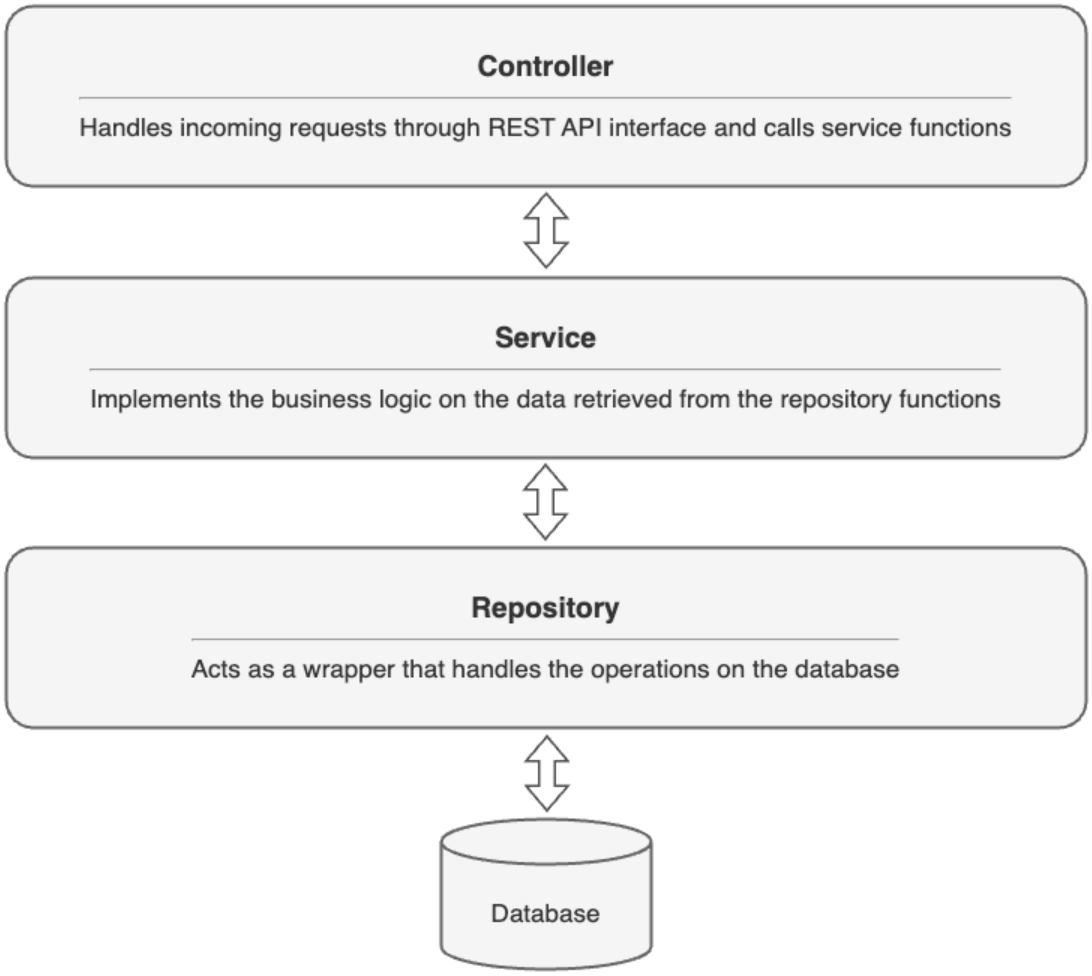

Figure 5.1. Controller-Service-Repository Pattern

The **Controller layer** handles the incoming HTTP requests and produces the corresponding HTTP response. A controller usually does not perform any significant operation, but it just calls the lower layer in order to obtain the results which are then sent in the response. Sometimes a controller can perform validation on the input or on the user authorization.

Below there is the **Service layer**, where all the business logic is contained. This is the core of the application, because it is here that the input data is managed in order to produce the output data. Service functions can store and retrieve data indirectly by calling the **Repository layer**.

A repository is an entity that exposes functions to manage the data in the database,

so that the service functions don't have to interact directly with it. To understand the importance of this layer, if the team wants to migrate from a DBMS to another one, this is the only layer that will change. In fact, if the interface of the repository does not change, we can modify the internal implementation as we want, but the upper layers won't need intervention. The Figure 5.1 summarizes this pattern highlighting the connections between the layers.

This pattern is really helpful when testing the application. Each layer can be tested in isolation from the other two by providing a mock implementation of the called functions, and allowing developers to easily write unit and integration tests for each part of the code base.

5.1.2 Dependency Injection

Dependency Injection (DI) is a design pattern that promotes loose coupling and flexibility in the code[36]. Typically, there is some class or function (often called a **consumer** or **client**) that relies on another entity to perform its tasks, known as a **service** or **dependency**. Using Dependency Injection, the consumer requests a service to be passed through a constructor or as function parameter, rather than instantiating it directly. This approach allows the consumer to depend on an interface that represents the "behavior" of the requested service, which the service must implement in order to be used by the consumer.

The framework, often called **injector**, stores the service instances, manages their lifecycle, and injects them as needed, in that way reducing the amount of code the developers must write and freeing them from the responsibility of managing object lifecycles. Dependency Injection enhances flexibility, because new implementations of an interface can be added without changing the classes that depend on it. Additionally, it benefits testability, because by using interfaces it is easy to swap out real implementations for mock or stub versions, which is essential for unit testing. This allows developers to test the behavior of a class in isolation, without needing the actual implementation, leading to faster and more reliable tests. .NET Core framework supports DI, that can be achieved by registering each dependency as a **service** in the program setup[37]. After that, the framework will instantiate the objects as needed and will inject them where they are requested at runtime.

5.1.3 RESTful API

A **RESTful API** (Representational State Transfer Application Programming Interface) is a web service architectural style that leverages standard HTTP methods to facilitate communication between systems over the internet. RESTful APIs adhere to a set of constraints and principles designed to create scalable, lightweight, and efficient services. These APIs use common HTTP methods such as GET, POST, PUT, DELETE, and PATCH to perform Create, Retrieve, Update, Delete (CRUD) operations on resources, which are

uniquely identified by an Uniform Resource Identifier (URI). One of the key characteristics of RESTful APIs is being **stateless**, which means that each request from a client must contain all the necessary information for the server to understand and process it, without relying on any previous interactions. This stateless nature simplifies the server's design, improves scalability, and allows different servers to handle requests independently, making RESTful APIs highly suitable for distributed systems like microservices.

In the context of microservices, RESTful APIs play a crucial role in enabling independent services to provide a communication interface for accessing their functionalities. Each microservice can expose its functionality via RESTful endpoints, using a common data format such as JavaScript Object Notation (JSON) for communication. JSON is lightweight, easy to read and write, and widely supported across different programming languages, making it an ideal choice for data exchange. RESTful APIs' ability to support stateless communication ensures that each microservice remains autonomous, reducing dependencies and allowing for more flexible and resilient system architectures. This approach enhances the overall reliability and performance of the microservices ecosystem, and also allows for better scalability of each microservice.

5.2 IAM microservice

The IAM service is essential for providing a centralized system for user registration, authentication, and authorization across all microservices. It not only handles user authentication but also provides system administrators with tools to manage user roles and permissions effectively. This document outlines the design and implementation of the IAM API, emphasizing its role in maintaining security and access control within the microservices ecosystem.

To provide its functionality, the IAM microservice is divided into two main parts:

- **Identity Management**: Manages user authentication and identity verification across the microservices ecosystem.
- **User Management**: Allows system administrators to manage user details and roles, ensuring proper access control.

5.2.1 Identity Management API

The Identity Management API focuses on enabling users to authenticate and verify their identity within the microservices environment. This includes functionalities such as user registration, login, logout, and session management.

Security is a primary concern and the API is designed to handle multiple scenarios:

- Unique user identification to avoid ambiguity.
- Role-based operation permissions to control access based on user roles.

- Time-limited sessions for authenticated users to manage access duration.
- Session extension through identity verification before expiration.
- Immediate session termination upon user request or if validity conditions change.

Each user is identified uniquely by an integer ID, automatically assigned during registration. Users also provide a unique username and email address, alongside personal information such as first and last names. To support organizational needs, a company attribute is included for associating users with specific companies.

User roles are integral to implementing RBAC, which grants permissions based on predefined roles. Role assignment is controlled by the system to prevent unauthorized role escalation, such as users assigning themselves an admin role.

5.2.2 Role Assignment Criteria

Access permissions are managed using Role-Based Access Control (RBAC). This technique assigns a role to each user and the role determines the user's permissions, specifying which actions they can perform and which resources they can access. In the system we used three roles: administrator, customer and AROL operator. The **administrator** role represents the users that can manage all the system's resources, such as CRUD operations on users, machines, parts and tickets. The **AROL operator** can generate and retrieve QR codes and obtain private information on AROL machines and components. The last role is the **customer**, that represents an AROL customer and can only scan QR codes and access the corresponding unrestricted data, such as name, description and other basic information.

5.2.3 User Management API

The User Management API provides functionalities for system administrators to manage user data securely. Each endpoint is protected by authorization mechanisms to prevent unauthorized access.

The key functionalities include:

- Adding new users.
- Removing users.
- Editing user details and attributes.
- Retrieving specific user information.
- Retrieving all users with optional filtering criteria.

These functions are generally restricted to administrators, ensuring that only authorized personnel can perform sensitive operations. Endpoints for user creation (e.g., POST /users) are admin-only, while general users can interact with their profiles through authenticated endpoints.

5.2.4 Entities

Each domain entity is represented by a class that maps directly to database tables, facilitated by Entity Framework Core, an Object-Relational Mapper (ORM).

Key entities include:

- **User**: Represents a system user, capturing attributes such as ID, username, email, company, password, and role.
- **Refresh Token**: Represents a refresh token associated with a user, allowing secure token rotation and session management.

The Refresh Token entity is linked to the User entity in a one-to-one relationship, ensuring that each user has a maximum of one active refresh token. On one hand, this setup simplifies token revocation and rotation, as tokens can be easily invalidated by deleting the corresponding database entry. On the other hand, this mechanism doesn't allow users to have multiple sessions opened at the same time, which constitutes a limitation.

5.2.5 Access tokens and refresh tokens

Until now, access and refresh tokens were mentioned, related to the user identification and permissions. Therefore, it is necessary to explain in more detail what are those tokens, their format, how they are generated and used.

An **access token**[38] is an entity that should contain the basic information about the user who is making the authenticated requests, but also the information about the user permissions. A **refresh token**[39] has a different purpose, because it does not contain all the information stored in the access token, but just some information that can be used to create another access token. For example, inside a refresh token can be stored the identifier of a user, so the user can be retrieve from the database and a new access token can be generated using that data. The refresh token can be used by the user to request a new access token, when its previous one expired, therefore the refresh token usually lasts much longer then the access one. These mechanism allows the user to maintain a valid authenticated session for a longer period of time, until the refresh token expires.

In this project, both the access and refresh token are formatted as base 64 strings of characters. The chosen token type is JSON Web Token (JWT), which is a very convenient and widespread way of representation[41]. A JWT token is formed by three sections:

- **Header**: Contains a field "alg" that specifies the algorithm used for signing the token and a field "typ" that specifies the type of token, in this case "JWT".
- **Payload**: This section contains the so called **claims**, that are important information about the user and the token itself. There are 7 registered claims:
    1. **iss**: issuer, specifies the **principal** that issued the token; typically it corresponds to the hostname.

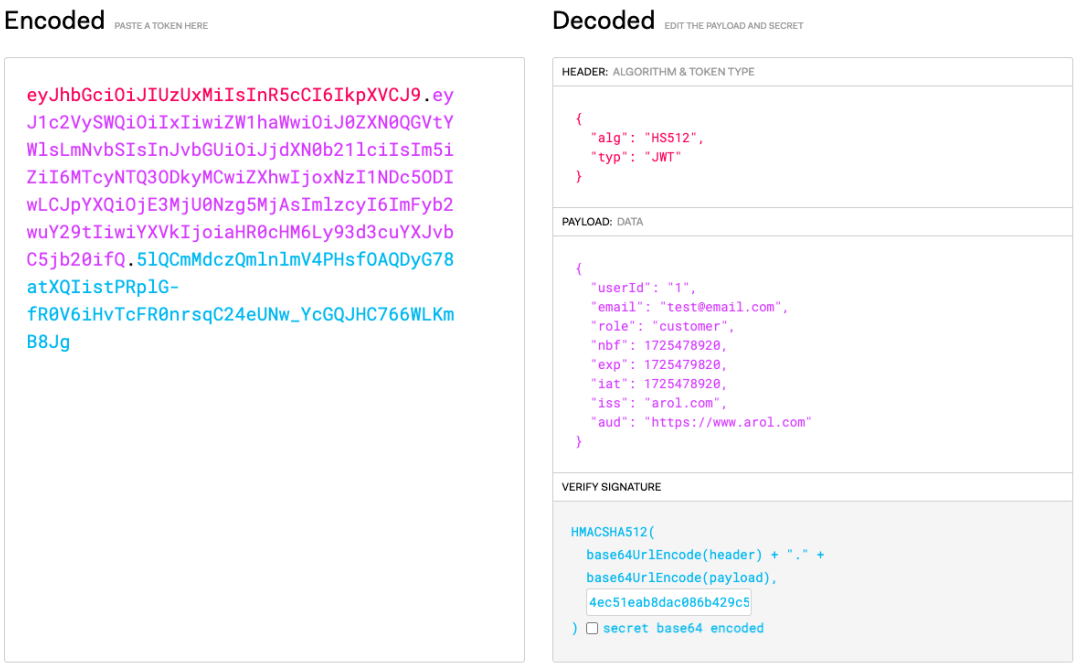

Figure 5.2. Example of a JWT token, on the left the encoded string and on the right the decoded content of the token[40]

2. **sub**: subject, identifies the subject of the JWT, usually the user.
3. **aud**: audience, the recipients of the token.
4. **exp**: expiration timestamp of the token.
5. **nbf**: not before, the date before which the token is not valid yet.
6. **iat**: issued at, the timestamp when the token was issued.
7. **jti**: JWT ID, a code that identifies the token.

Additional claims can be added on creation.

- **Signature**: the section validates the integrity and authenticity of the token. It is obtain by encoding the first two sections using Base64 encoding and concatenating them with a period as separator. Then the resulting string is hashed using the cryptographic algorithm specified in the header section (the "alg" field).

All three fields are encoded into Base64 strings and then concatenated with periods. The resulting string is the final JWT token.

5.2.6 Authentication flow

Users must be authenticated in order to perform restricted requests to the system. Microservices can identify the user and authorize them using the access token released by

the IAM microservice. JWT tokens are self-contained, which means that all the necessary information is included in the token, without the need to contact the IAM service each time a new request comes in. This stateless mechanism enhances scalability and performance, because it cuts off a major number of verification requests to the authentication microservice.

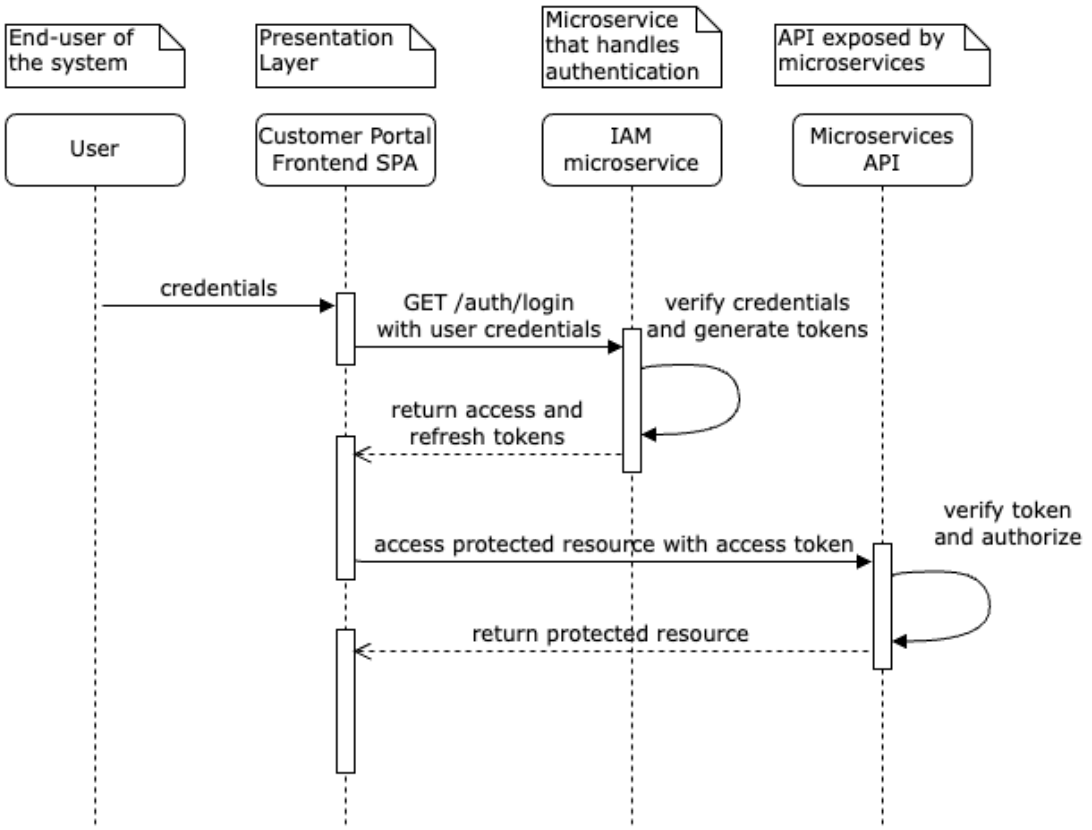

Figure 5.3. Login flow to access protected resources

The Figure 5.3 describes the flow to make an authenticated request to access a protected resource from one of the microservices. Making authenticated HTTP requests means adding the access token to an authentication header, using bearer token authentication. When a new request comes in, the corresponding microservice checks if the token is present, then it verifies the signature, to ensure integrity and authenticity of the token, and then it decodes it. After that, the token is checked to validate the issuer and other fields, among which there is also the expiration date. If some validation step fails, the token is considered invalid and the request fails with code 401 (i.e., not authenticated). If instead the token is valid, the next step is the role validation. In this step, the microservice checks that the role claim's value matches the role needed to access the requested

resource, otherwise code 403 (i.e., not authorized) is returned.


```
{
    "user": {
        "id": 602,
        "email": "admin@admin.com",
        "username": "admin",
        "firstName": "admin",
        "lastName": "default",
        "company": "AROL",
        "role": "admin",
        "creationDate": "2024-10-06T11:16:17.134404Z"
    },
    "accessToken": "eyJhbGciOiJIUzUxMiIsInR5cCI6IkpXVCJ9...",
    "accessTokenExpiresAt": "2024-10-15T14:07:28.1947457Z",
    "refreshToken": "eyJhbGciOiJIUzUxMiIsInR5cCI6IkpXVCJ9...",
    "refreshTokenExpiresAt": "2024-10-22T13:52:28.1949511Z"
}
```


Figure 5.4. Example of login response (tokens have been trimmed due to excessive length)

In order to access the protected resource, the user should have obtained beforehand an access token from the IAM, which can be put inside the request headers. To obtain the pair of access and refresh tokens, the user must sign in into the system, performing a POST request to */api/auth/login* and passing in the body their username and password. This assumes that the user has been registered before, either by an administrator that created their account or by themselves, doing a POST request to */api/auth/register* endpoint. The login request, if successful, returns an object containing the user information, an access token, a refresh token and the expiration timestamps of both the access token and refresh token (see Figure 5.4).

The user is responsible for the secure storage of the tokens, and also for setting the headers before sending the authenticated requests. In the front-end web application that uses this project's microservices, tokens are stored in the local storage of the browser. This isn't the most secure way for a Single Page Application (SPA), because a malicious attacker can steal them. However, for development purposes, this solution is enough to guarantee secure communication between front-end and back-end. To minimize the risk, it has been decided to set the lifespan of each access token to 15 minutes.

5.2.7 Refresh token revocation and rotation

The access token is short-lived, it lasts only some minutes, and it isn't stored anywhere in the server, meaning that it is stateless, and once it is generated, it cannot be revoked

in any way. For this reason, it is short-lived, usually shorter than an hour; in this project, it lasts 15 minutes as mentioned before. In this way, an attacker only has a short span of time to steal and use an access token, and it can't be refreshed without stealing also the refresh token. The refresh token should have a longer time to live, because it is used to renew an expired access token and it would not be useful otherwise. In the context of this project, the lifespan of a refresh token is 24 hours, which means that a user can request a new access token within a day from the moment the login is performed. This is a good compromise between security and convenience, because it allows the user to sign in no more than once a day, or even less thanks to the token **rotation** mechanism.

Token rotation is a simple technique that means changing the refresh token every time it is used[42]. Basically, every time the user calls the endpoint to refresh the access token and provides the refresh token, not only a new access token is released, but also a new refresh token, invalidating the previous one (see Figure 5.5). This mechanism enhances security, because a refresh token will be exposed for a shorter time than its lifespan, and also it prolongs the authenticated session of the user, meaning that the user does not have to perform login for a long time, if he keeps refreshing the token before its expiration.

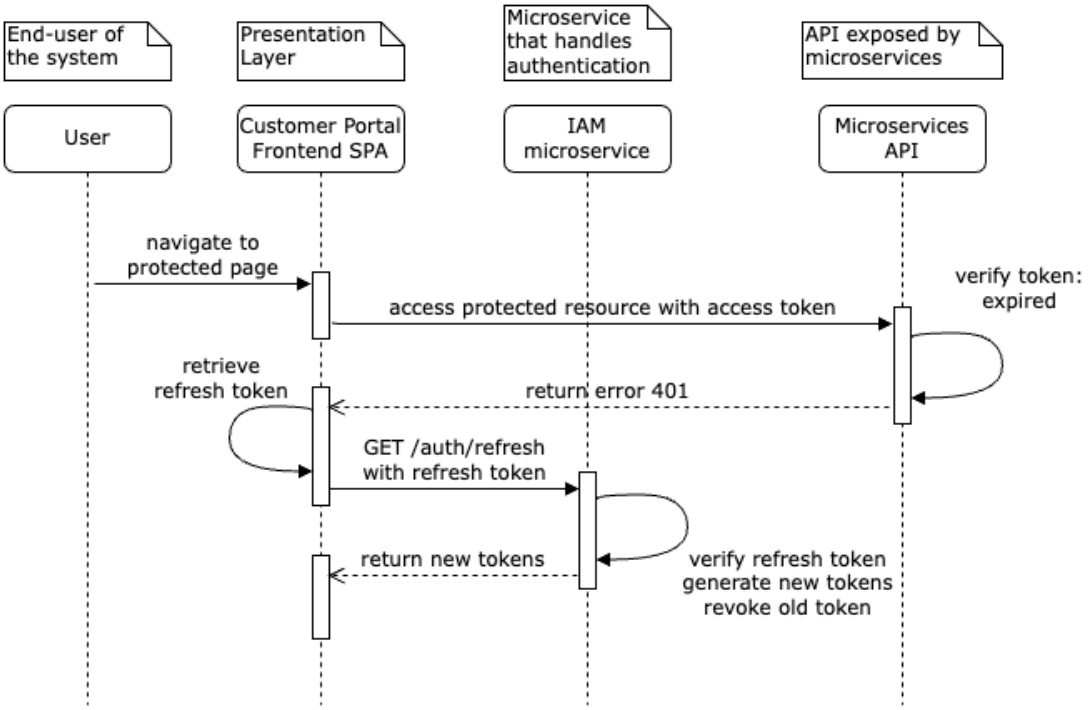

Figure 5.5. Flow of refresh token mechanism

The invalidation of a refresh token is also called **revocation**. As previously mentioned,

in the IAM microservice's Database are stored both the users and their refresh tokens, and there is at most one refresh token associated to each user. Using this stateful mechanism to manage the tokens, it is easy to verify the validity of a refresh token every time a user tries to call the refresh endpoint. These are the steps for the validation:

1. Verify signature and expiration date of the refresh token provided by the user in the request's headers
2. Retrieve the user Id claim from the token and use it to retrieve the corresponding user info and refresh token in the database
3. If no token was found in the database, the user session expired and the user must perform the login again, thus no new tokens are provided
4. Compare the two tokens: if they are the same, proceed to the creation of the new access and refresh tokens
5. Save the new refresh token in the database, replacing the old one
6. Return the tokens to the user in the HTTP response

Refresh tokens are revoked also when the user explicitly performs the logout operation. In that case, the refresh token associated to the current authenticated user is deleted from the database and, the next time the user wants to perform authenticated requests, he must sign in again. Being the access tokens stateless, they can't be revoked, which means that the logout is not a "full" logout. For this reason in the front-end application, it is the application itself that proceeds to erase both tokens from the local storage after the user logs out.

5.2.8 API endpoints

As explained in the previous paragraphs, the IAM microservice exposes API endpoints for user authentication and for user management. The Table 5.1 describes each endpoint with the corresponding HTTP method, the path and a brief description of the operation performed.

Method	Path	Description
POST	/auth/register	Register new user into the system
POST	/auth/login	Sign in with credentials
DELETE	/auth/logout	Sign out from current session
GET	/auth/current	Retrieve info about current logged in user
POST	/auth/refresh	Refresh tokens
GET	/auth/users	Retrieve all the registered users
POST	/auth/users	Add new user
GET	/auth/users/{userId}	Retrieve user by their identifier
PUT	/auth/users/{userId}	Update user information

Table 5.1. API endpoints exposed by the IAM microservice

5.3 Machinery microservice

The Machinery microservice is responsible for managing all requests related to AROL machinery and its components. Its primary function is to handle requests that are initiated when an AROL product's QR code is scanned, providing detailed information about the product based on the QR code's data. Upon scanning, the client application sends a request to the Machinery service, passing the unique identifier embedded in the QR code content as a parameter. The service processes this request by extracting the ticket identifier, which is then used to interact with the Ticket service to retrieve the relevant part number and, if available, the serial number associated with the ticket.

Using the part number and serial number, the Machinery service queries its database to find the corresponding part or machinery information. The data returned to the client is tailored based on the user's permissions, ensuring that sensitive information is protected and only accessible to authorized users. Additionally, the service maintains records of the link between customers and the machinery they have purchased. This linkage is crucial for determining ownership; if a component is linked to machinery purchased by a user, that user is considered the owner of the component, thus granting them access to information not available to non-owners.

5.3.1 Entities

The entities defined within the Machinery microservice are the following:

- **User**: Represents any individual interacting with the system, including AROL customers, AROL employees (with or without administrative privileges), or external users.
- **Machinery**: Represents an individual piece of AROL machinery, uniquely identified by both a serial number and a part number. Each piece of machinery is distinct and tracked separately within the system.

- **Machinery Part**: Refers to any component of AROL machinery. Parts may be serialized, meaning each instance is individually tracked with a unique identifier, or non-serialized, where instances are not individually tracked but are identified by a part number, name, and description.

There are several important aspects of the system's design and structure that need to be understood:

- **Hierarchical Structure**

  Machinery components are organized hierarchically, matching with a tree structure. This hierarchy captures the parent-child relationships among components, allowing complex assemblies to be represented accurately. For example, a main machinery unit may contain several sub-components, each of which could have further sub-components.

- **Serialized vs. Non-Serialized Parts**

  Components can either be serialized or non-serialized. Serialized parts have unique identifiers, allowing them to be tracked individually. This is crucial for parts where individual tracking is necessary, such as critical components or customized components. Conversely, non-serialized parts, like standard screws, are not tracked individually but are identified by their type. These parts share common attributes but are otherwise indistinguishable from one another.

- **Inheritance in Hierarchical Structure**

  The serialization status impacts the hierarchy. A serialized part may contain both serialized and non-serialized sub-parts (e.g., a main unit containing individually tracked sensors and indistinguishable screws). However, non-serialized parts do not contain serialized sub-parts, maintaining the rule that if a component is not uniquely tracked, none of its sub-components will be uniquely tracked either. This maintains consistency in the tracking logic, ensuring that the hierarchy aligns with the serialization strategy.

The Machinery microservice exposes a set of API endpoints, primarily dedicated to administrators, which allow for the management and retrieval of machinery and parts information. These endpoints include functionality for creating, updating, and deleting machinery and parts, as well as retrieving detailed information about specific entities.

When retrieving data, the API supports selective information retrieval to optimize performance and relevance. For instance, when a request is made to retrieve details about a particular machinery or part, the response may include only basic information. Then, if the user wants to obtain additional data about its sub-components, they can make subsequent requests to specialized endpoints. This approach ensures that the system is not overloaded with unnecessary data and that clients receive the most relevant information first, focusing on lightweight and quick responses.

5.3.2 Database Relationships and Hierarchical Retrieval

The database is structured to reflect the hierarchical nature of machinery and their components, where machinery items serve as the highest-level entities. These machinery items do not store references to any parent components, as they represent the root of the hierarchy. In contrast, child components are designed to maintain references to their parent components, allowing for navigation upwards through the hierarchy. However, parent components do not hold any references to their children, which means that moving downward through the hierarchy cannot be done in constant time.

Serialized parts are specifically linked to the root machinery item to ensure traceability back to the machine they belong to. Additionally, every serialized part also references another serialized part as its parent. Non-serialized parts, on the other hand, may refer to either serialized or non-serialized components as their parent, depending on their location within the hierarchical structure. These non-serialized parts do not have direct references to machinery items, as they could be part of multiple machines, which complicates direct tracing back to a single machine. To optimize retrieval operations, each part, whether serialized or not, contains a boolean indicator that flags whether it has any child components. This improves the efficiency of querying and hierarchical navigation within the database.

A typical sequence of HTTP requests to construct the full hierarchy from a piece of machinery might look like this:

1. **GET** /machinery/{serial} - Retrieves basic details about the specified machinery.
2. **GET** /machinery/{serial}/parts - Retrieves serialized parts associated with the specified machinery.
3. For each serialized part use **GET** /parts/serialized/{serial}/children to retrieve all its child parts.
4. For each non-serialized part, use **GET** /parts/not-serialized/{partNumber}/children to retrieve all its child parts.

This recursive querying continues until all parts of the tree are retrieved, including all leaf nodes, ensuring that the entire machinery component structure is captured.

The endpoints that facilitate these retrievals are accessible to authenticated users, with restrictions in place to protect sensitive information. Serialized part details are limited to AROL employees and verified customers who have purchased machinery containing those parts. This ensures that proprietary and sensitive information is not exposed to unauthorized users.

5.3.3 API endpoints

The Machinery microservice exposes a set of REST API endpoints that can be called via HTTP to perform the requested operations. The Table 5.2 lists all the available calls, among which there are CRUD operations for the machines and parts, but also the decoding functionality used when scanning a QR code from the application.

Method	Path	Description
GET	/machinery	Retrieve machinery list
POST	/machinery	Add machinery
GET	/machinery/{serial}	Retrieve machinery by its serial
GET	/machinery/{serial}/parts	Retrieve machinery parts by its serial
GET	/parts/serialized/{serial}	Retrieve serialized part by its serial
GET	/parts/not-serialized/{partNumber}	Retrieve not serialized part
POST	/parts/serialized	Add new serialized part
POST	/parts/not-serialized	Add new not serialized part
GET	/decode/{ticketId}	Obtain product info from ticket id
GET	/decode/{ticketId}/expanded	Obtain hierarchical product info

Table 5.2. API endpoints exposed by the Machinery microservice

5.4 Ticket microservice

The ticket microservice is the core of the serialization and deserialization processes between QR codes and corresponding AROL identifiers. This microservice works as a transcoding service from ticket identifier to a pair of serial number and part number, and vice versa. The unique identifier of each ticket can be used to create a unique QR code for an AROL product.

5.4.1 Entities

The only entity in this program is the **Ticket** entity. A ticket is uniquely identified by a **Ticket Id**, which is a Universally Unique Identifier (UUID), in form of string. A ticket also contains the **serial number** and the **part number** of the product which is referred by it. Despite their names, those two are not necessarily number, but strings of characters. The serial is associated with a serialized product, and it is unique inside AROL company. The part number describes the type of product, thus it is common to all the parts with the same characteristics and it is not unique. Non serialized parts don't have any serial number, therefore the system sets a default value for them, which implies that the QR code will not be unique for those parts. Another field associated to a ticket is the **scope**, which is a string of characters that indicates the scope of validity of the QR code related to that ticket. For example, when creating a new ticket, the operator can type "test" as scope, to specify that the corresponding QR code is valid just for test purposes and it

can't be used on actual shipped products.

5.4.2 Permissions

This microservices allows the creation of identifiers and, consequently, the generation of the corresponding QR codes. For this reason, it is not accessible by all users, but its usage is limited to administrators only. Therefore, all the incoming HTTP requests must carry a bearer token, signed by the IAM microservice, which contains the role "admin". In case the user is not authenticated, the service will return a 401 status, and if authenticated but not authorized, it will return 403.

5.4.3 API endpoints

The API exposed by the Ticket microservice is simple and straightforward. As represented in Table 5.3 the endpoints support basic operations to retrieve, add, update and delete tickets.

Method	Path	Description
GET	/tickets	Retrieve list of tickets
POST	/tickets	Create new ticket
GET	/tickets/{ticketId}	Retrieve ticket by its identifier
PUT	/tickets/{ticketId}	Update ticket fields
DELETE	/tickets/{ticketId}	Delete ticket

Table 5.3. API endpoints exposed by the Ticket microservice

5.5 API Gateway

All requests coming into the microservices ecosystem pass through a single point, called API Gateway. This service can be very useful in some cases, because it can take care of cross-cutting concerns among all the microservices, such as security, logging, redirecting requests, aggregating data from microservices, and so on.

In this project, the API gateway has only one task, that is taking care of all the incoming requests and redirecting them to the appropriate microservice to which the requests are destined. This functionality is called **reverse proxy**. Reverse proxying the incoming requests is important to make sure that the specific microservices can't be directly contacted from the external world, forcing the requests to pass through the gateway. As mentioned before, this system may be used to check the content and permissions on every single request, forwarding only the ones that pass the checks.

The API gateway pattern has some disadvantages. First of all, it constitutes a single point of failure for the entire system, thus if the gateway goes down, all the system becomes unreachable. Another problem is scalability: every request passes through the gateway, which means that when the traffic increases, it will require many resources to scale up this service. Furthermore, if the API gateway contains business logic related to some other service, it should change accordingly to the changes happening in the other service. For example, if the gateway makes a call the machinery service to aggregate some data but the API changes, also the gateway logic must be changed. This can be expensive in terms of time and it might be the source of bugs in the system.

5.5.1 NGINX

In order to provide reverse proxy functionality, the system uses a Docker container with NGINX, that is a widely used web server[43]. NGINX is very flexible and it can be use as proxy, reverse proxy, load balancer and much more. The service can be setup writing a configuration file and mounting it as a Docker volume.

```
events {
}
http {
  server {
    listen 80;
    location /auth/ {
      proxy_pass http://auth-service/;
    }
    location /machinery/ {
      proxy_pass http://machinery-service/;
    }
    location /ticket/ {
      proxy_pass http://ticket-service/;
    }
  }
}
```

Figure 5.6. nginx.conf file content

The configuration file used for the API Gateway in this project can be seen in Figure 5.6. In the configuration file, http server is the section where the nginx server is setup. It is instructed to listen on the port 80 and to perform redirection to microservices based on the path of the incoming requests. The syntax is "location <request path to match>" and "proxy_pass <redirection url>"[44]. For example, when a request comes on port 80 with path starting with */auth*, all the prefix part (including host and port) is stripped

and replaced with *http://auth-service/*, which redirects the request to the IAM service. The headers are kept, which means that the authentication tokens are forwarded together with the request. Docker or Kubernetes DNS functionality[45][46] allows us to write the name of the service (e.g., auth-service, machinery-service, ticket-service) instead of the host and port. This is also beneficial if we deploy more than one instance of the same microservice to perform load balancing.

Chapter 6

Logging, Metrics Collection and Monitoring

When the system is deployed and running, it is essential to regularly monitor its current state to ensure that all components are working correctly. This is especially important in microservice-based architectures, where multiple services work independently but are responsible for the system's overall performance. Effective monitoring not only helps identify and fix bugs but also provides valuable insights about the system's behavior, which can lead to performance improvements[47].

6.1 Logging

At runtime, the applications produce **logs**, pieces of textual information about events and actions occurring within the system. Logs are one of the most important sources of real-time information for understanding system operations and for diagnostic. Each log entry is associated with a *level*, which indicates the severity or importance of the event being recorded. The most common logging levels are:

- **Trace**: Extremely granular information about the system's behavior, providing precise details about code execution. This level is used primarily during development for debugging specific parts of the code.
- **Debug**: Provides more useful debugging information than the trace level, while being less verbose. It is typically used to understand how the system is functioning in real-world scenarios when the system is already deployed in production.
- **Info**: Represents general events that describe the normal operation of the system. These logs document successful operations and typical events in the system.
- **Warning**: Indicates that something unexpected occurred, but without affecting system functionality. Warnings often highlight potential problems that may require attention to prevent more critical errors.

- **Error**: Signals that an error has occurred and is causing certain operations to fail. Errors require immediate attention, but the system may continue to function.
- **Fatal**: Indicates a severe problem that prevents the system from functioning correctly, often causing the service to crash or become unusable.

Logs are a crucial tool for system monitoring, allowing administrators and developers to track both the history and real-time behavior of the system. In the context of microservices, logging can become challenging because each service produces its own logs, making it really complex to centralize and analyze them. For this reason, it is important to collect and aggregate logs in a centralized location, where logs from different services can be viewed in chronological order and filtered based on severity or origin. To ensure consistency, all microservices must follow a unified log format. This makes it easier to process and analyze logs across the system.

Serilog is an open-source logging library that produces structured logs in a predefined format[48]. In this project, Serilog is configured for each microservice to generate JSON-formatted logs, containing also additional metadata such as log level, application name, and host machine details. This structured approach facilitates easier searching and analysis of logs.

Loki is a tool developed by Grafana Labs used in the project for centralized log collection[49]. Loki collects and indexes logs from all microservices, improving log management and search. By using a **sink**, Loki can be integrated with Serilog, ensuring that all logs are streamed to Loki in real-time. Operators can then query logs using Loki's query language, **LogQL**, to filter, group, or aggregate log data based on specific criteria.


```
{
  "Message": "Executed endpoint '\"HTTP: POST /api/auth/login\"'",
  "MessageTemplate": "Executed endpoint '{EndpointName}'",
  "EndpointName": "HTTP: POST /api/auth/login",
  "EventId": {
    "Id": 1,
    "Name": "ExecutedEndpoint"
  },
  "SourceContext": "Microsoft.AspNetCore.Routing.EndpointMiddleware",
  "RequestId": "0HN7D1J18M833:00000001",
  "RequestPath": "/api/auth/login",
  "ConnectionId": "0HN7D1J18M833",
  "ThreadId": 24,
  "level": "info"
}
```


Figure 6.1. Example of a log event from the IAM microservice formatted in JSON and collected by Loki.

6.2 Metrics Collection

While logs provide insight into events that have occurred, they do not offer a complete view of the system's real-time performance. For that, we need **metrics**, that are numerical measurements collected over time that reflect various aspects of system health and performance. Metrics allow for a more quantitative view of how the system is functioning, often helping to identify issues or performance bottlenecks.

In the context of web API applications, the following metrics are particularly useful:

- **Number of incoming requests per second**: This indicates the overall load on the system and helps monitor traffic.
- **Average request latency**: This measures the time taken to process requests, indicating performance and responsiveness.
- **Error rate**: The percentage of failed requests, which is critical for monitoring system reliability.
- **Request distribution**: The number of requests handled by each microservice, helping detect load imbalances and thus helping system administrators to know when a microservices requires to be scaled.
- **Microservice health status**: Health checks of individual services to ensure that they are up and working correctly.

Each microservice in this project exposes its metrics through a dedicated endpoint. A monitoring tool, **Prometheus**, is used to periodically "scrape" (i.e., collect) metrics data from these endpoints. Prometheus is an open-source toolkit widely adopted for monitoring and alerting[50]. Metrics are stored as time-series data, allowing operators to track how these metrics change over time and therefore to analyze the evolution of the system. Prometheus also provides its own query language, **PromQL**, which allows users to filter, aggregate, and analyze metrics data. With PromQL, system administrators can query for specific metrics, set up custom alerting rules, and analyze the overall performance of microservices.

6.3 Monitoring with Grafana

While Prometheus and Loki are powerful tools for collecting logs and metrics, they do not provide a visualization layer. This is where **Grafana** comes in. Grafana, developed by the same team behind Loki (i.e., Grafana Labs), serves as a visualization platform that can connect to various data sources, including Prometheus and Loki[51].

Grafana allows users to build custom **dashboards** by creating panels that display logs and metrics in the form of charts, graphs, or tables. These dashboards provide an intuitive overview of the system, making it easy to track Key Performance Indicators (KPIs) and

spot potential issues early. For example, a dashboard can display metrics such as CPU usage, memory consumption, or request latency in real-time, alongside logs from Loki, showing a complete view of the system's health and performance status.

Dashboards can be customized to meet the specific needs of the operations team, allowing them to monitor microservices, set up alerts for critical events, and quickly respond to anomalies. With Grafana's capabilities and its integration with PromQL and LogQL query languages, users can explore metrics and logs to analyze specific problems, helping reducing downtime and improving overall system reliability.

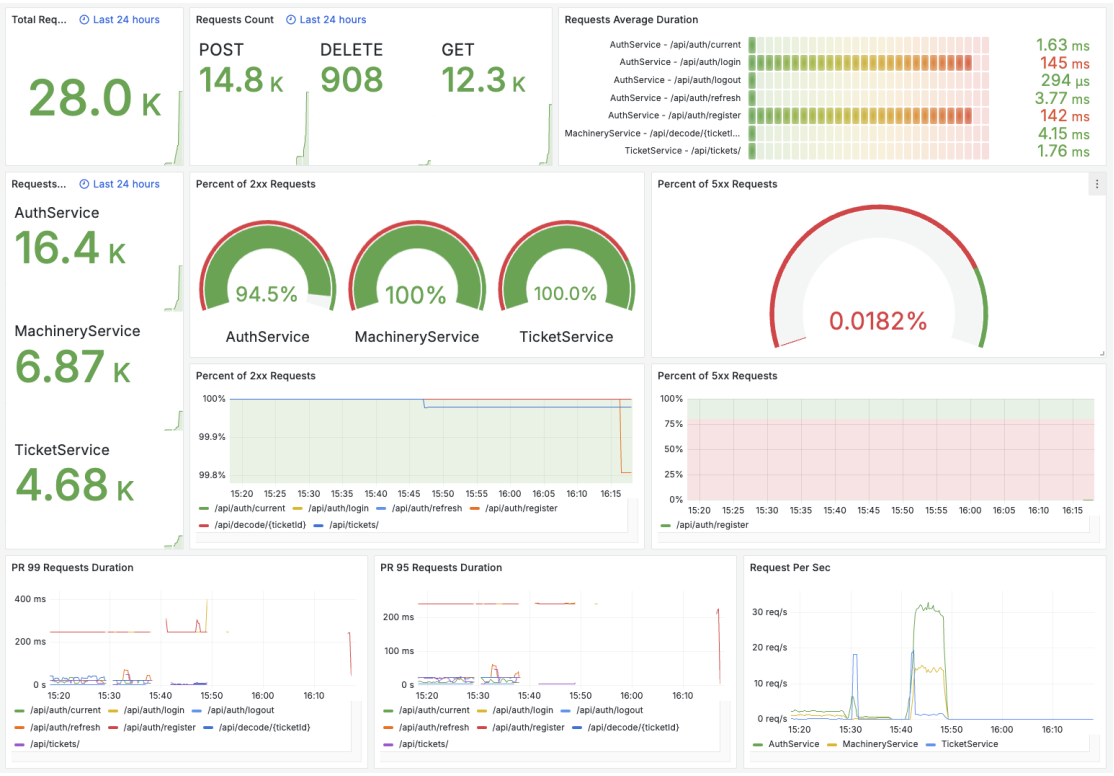

Figure 6.2. Grafana Dashboard used for monitoring in this project

Chapter 7

Application Deployment

After several months during which the application was under continuous changing, the system was finally ready for deployment. Deploying an application means making it available to end-users. As stated earlier, this project focuses on the backend system, which supports a frontend layer that has been thoroughly analyzed in another thesis. While both the backend and frontend components are deployed, this chapter will primarily focus on the deployment of the backend microservices system.

Deploying an application based on microservices is a complex task that requires not only knowledge of best practices but also hands-on experience in configuring the deployment environment. This process includes managing the deployment of microservices along with the infrastructure for data storage, logging, and system monitoring. Ideally, the services should remain available at all times and should be scalable to handle increased loads.

Each microservice is containerized in a Docker image, allowing the creation of multiple containers from the same service image. These images are stored in the Azure Container Registry, with tags used to identify different image versions. Storing images in a container registry simplifies the deployment process, as the services do not need to be rebuilt, but they can simply be instantiated as containers from the pre-built images. Once all the microservices are containerized, the next steps involve setting up the deployment environment and running the complete system.

7.1 Kubernetes Setup

When deploying an application, there are two primary options: using a managed cloud service (e.g., AWS, Azure) or hosting the application on-premises using the company's own servers. The first option is faster and more convenient but can be costly. The second option is more complex to manage, but it can reduce long-term costs if the system's workload is manageable and the application requires minimal maintenance after deployment. Since the AROL R&D team had access to machines capable of hosting the services, we decided to take the opportunity to deploy the system on-premises.

To orchestrate and manage the deployment, we used **Kubernetes**. The core concept of Kubernetes deployment is the **cluster**, that is a group of **nodes**, working machines in which the containerized services run. A cluster also contains a **control plane**, a set of components that make global decisions about the cluster[52]. The control plane is a crucial part of the Kubernetes architecture and it should be always available and reachable. For this reason, it is usually replicated on multiple machines to provide fault-tolerance and availability. In a worker node, multiple **pods** can be created. A pod represents a set of running containers in the cluster. There is an agent, **kubelet**, that runs inside every node and that ensures that containers are running in a pod and are healthy. Figure 7.1 represents the architecture described above.

In order to run the microservices on multiple machines, the first step was to install Kubernetes on the chosen machines, that would act as worker nodes in the cluster. Each machine was set up with the required resources to handle the container workloads. The cluster was created using **k3s**, a lightweight version of Kubernetes, which is enough for our purposes.

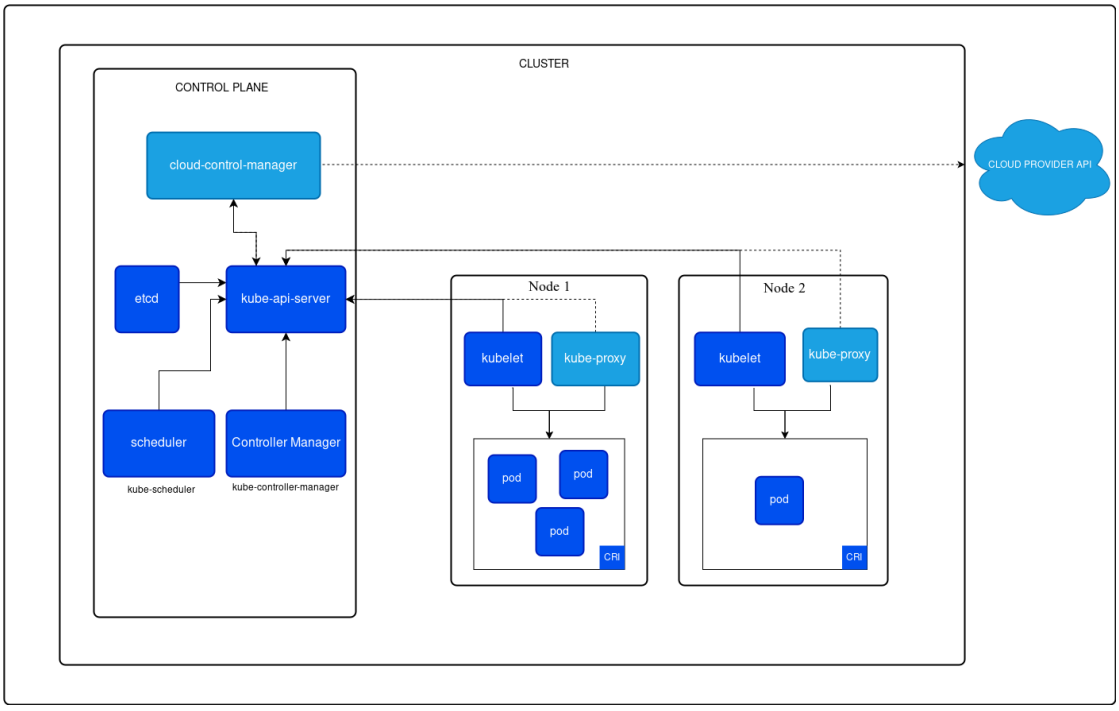

Figure 7.1. Kubernetes Cluster Architecture Diagram[52]

7.1.1 Azure Container Registry Integration

The microservices were containerized using Docker and the images were stored in the Azure Container Registry (ACR). The registry was not accessible without appropriate permissions. In the CI/CD Bitbucket pipeline, an account with "push" permissions was used to store the built images in the repository. In the deployment environment, an account with "pull" permissions was used to retrieve and download the images from the registry.

Kubernetes allows to configure the so called **image pull secrets**, that are secret configurations to pull images with authentication. Before running the containers, it is necessary to authenticate with Azure and sign in the ACR using credentials with pull permissions. Then a **kubectl** command allows the creation of the corresponding image pull secret with an access token obtained in the login step. This secret is used to pull the image before the container creation.

7.1.2 Defining Resources with Manifests

Kubernetes operates with a "desired state" methodology, that means that the user specifies the desired state of the cluster and Kubernetes tries to reach and maintain that state. The configuration is specified using YAML files called **manifests**. Usually each manifest file describe a single resource that we want to define in the cluster.

There are many types of resources, but for the microservices the used ones are the following:

- **Deployments**: Each microservice was defined with a Deployment object in Kubernetes. A Deployment manages a set of Pods to run an application workload, usually one that doesn't maintain state[53]. In the corresponding manifest the desired number of replicas of the microservice can be set.
- **Services**: Each microservice was exposed within the cluster and to the outside world using Service objects. All microservices were exposed for internal communication with **ClusterIP** services. The API Gateway service instead was exposed as a NodePort service, to be reachable from the external world. In this way, external communication is possible only passing through the gateway.
- **ConfigMaps and Secrets**: Microservices require some configuration settings, such as database connection strings or environment variables. These variables were stored using ConfigMaps for non-sensitive information and Secrets for sensitive data. Kubernetes automatically injects these configurations into the containers, ensuring that the services can be reconfigured without rebuilding the images.

7.1.3 Resource scaling

An important aspect of the deployment phase is scaling the service when the incoming requests are overloading the system. Kubernetes provides a way to manage scalability automatically for each Deployment resource. First of all, in the manifest we can specify the minimum number of replicas that we want to have up and running at all times. For example, in our case using a minimum of two replicas for each microservice and for the API gateway guarantees a certain level of availability, because we are reducing the down-time of each microservice if a fatal error occurs. Also, requests can be better distributed by the load balancer, leading to a better resource usage for each Pod.

The interesting parts happens when the minimum number of replicas is not enough to handle all the incoming traffic. In this case, it is necessary to increase that number, but doing it manually is inconvenient and also, once the new minimum number is set, it is necessary to manually intervene again to set it back to the previous value. For this reason, having a way to automatically scale horizontally would be a game changer. Kubernetes provides an **Horizontal Pod Autoscaler (HPA)** that is very useful for this task. As specified in the official documentation, the HPA includes a controller that "periodically adjusts the desired scale of its target (for example, a Deployment) to match observed metrics such as average CPU utilization, average memory utilization, or any other custom metric you specify."[54] In this way, when the metrics are below certain thresholds the number of Pods reflects the minimum number set in the manifests, but when the resource utilization increases, the HPA intervenes and creates new Pods as required to lower down the usage levels below the thresholds again. When the resources consumption goes down and becomes manageable again, the HPA shuts down the extra Pods that were added.

This mechanism not only automates the horizontal scaling of the microservices, but also allows to consume just the required resources at every moment. This means that it is not necessary to always allocate a huge amount of resources that won't be used, just in prediction that the load will increase, but with the HPA the resources will be allocated dynamically. Having a dynamic allocation of resources is crucial to pay only what is used and also allows to have more resource to run parallel tasks in the machines, thus not having to add more machines to perform various activities.

Chapter 8

Performance

When building and deploying an application, it's important to have an idea of its resource usage over time. In the context of microservices, monitoring the resources can help system operators to know when it's time to scale a service or set a usage threshold after which the service is scaled up automatically. An excessive stress on the system can lead to bad performances, because requests require more time to be processed and this leads to slower responses and to accumulate requests in the queue.

In order to collect useful data for usage metrics, there are two main cases that should be considered: a standard traffic scenario and a high traffic scenario. The first one simulates the system behavior most of the time, because usually the system won't be too stressed. The second scenario is the most challenging one, because it reproduces a situation of overload in the system. It is important that the system can respond quickly to a rapid increase in the number of requests, before it is too late.

AROL has more than 20.000 capping machines installed all over the world and hundreds of customers and operators scattered in different geographical areas. Customers can access the customer portal to scan a machine or a part inside it and examine the corresponding information. AROL operators might want to scan a QR code to access information needed for some internal operation, but also they can generate a new QR code associated to an existing product.

Provided this context, the most frequent operations on the system are:

- Check if user is authenticated (used by the frontend SPA very frequently)
- Refresh login session
- Login
- Scan QR code to retrieve product information

Other operations that may be performed with less frequency are:

- Generate a new QR code
- Logout
- Register new user
- Admin management operations on machines, users or tickets

Admin management operations are so rare in the daily context that they can be excluded from the traffic analysis, also because there is generally no more than a couple of system administrators that can access the system with that level of permissions. Thus, there is a very low probability of multiple and frequent parallel requests from them.

In order to simulate real situations, both with low traffic and high traffic, we wrote a script to perform HTTP requests to the system. The requests are the following:

1. GET /api/auth/current - Get current authenticated user to check session validity
2. POST /api/auth/refresh - Refresh current session
3. GET /api/decode/ticketId - Retrieve info from QR scan
4. POST /api/login - Perform login
5. POST /api/tickets - Generate new ticket for creating a QR code
6. DELETE /api/logout - Sign out
7. POST /api/auth/register

The script associates a probability value to each request, based on their probable frequency. The first three requests are the most probable, the login request a bit less because usually the refresh request is enough to prolong an existing session. The last three instead have the lower probability associated.

The script randomly chooses a request based on the associated weights and makes the corresponding call, tracking the response time. Then it waits for a random time before making the next request. The number of parallel requests and the waiting time between them determines the load on the system.

8.1 Low Traffic Scenario

The low traffic situation has been recreated by adding a high random waiting time between sequential HTTP requests. The waiting time range was set to not exceed 2 seconds. This produced a very light load situation, that the microservices handled very well. As shown in the Figure 8.1, all microservices stayed below 4% of CPU usage and between 5% and 10% of memory. Given that the most frequent request are performed to the IAM service, that's also the microservice that handles most request per second, but still

a manageable rate, being lower than 1 req/s on average. The slowest calls to respond are the user registration and login, that achieved an average response time of 175.5 ms and 180.5 ms respectively. Both operations internally hash the password sent by the user and then write on the database or generate authorization tokens. Cryptographic operations like password hashing with salt are particularly resource and time demanding and that's why the IAM service takes more time and CPU for registration and login. This can also explain why the IAM uses more CPU then the other microservices on average. We can also notice that the three most frequent requests (i.e., current, scan and refresh) all respond in less then 30 ms on average.

Request	**Total requests**	**Avg. res. time (ms)**	**Min. res. time (ms)**	**Max. res. time (ms)**
current	76	14.3	4	53
scan	78	30.0	7	87
refresh	75	22.3	8	69
login	32	180.5	158	193
logout	19	7.1	3	10
register	18	175.5	144	214
generate	10	27.7	10	62

Table 8.1. Low traffic: total requests and response times for each tested request

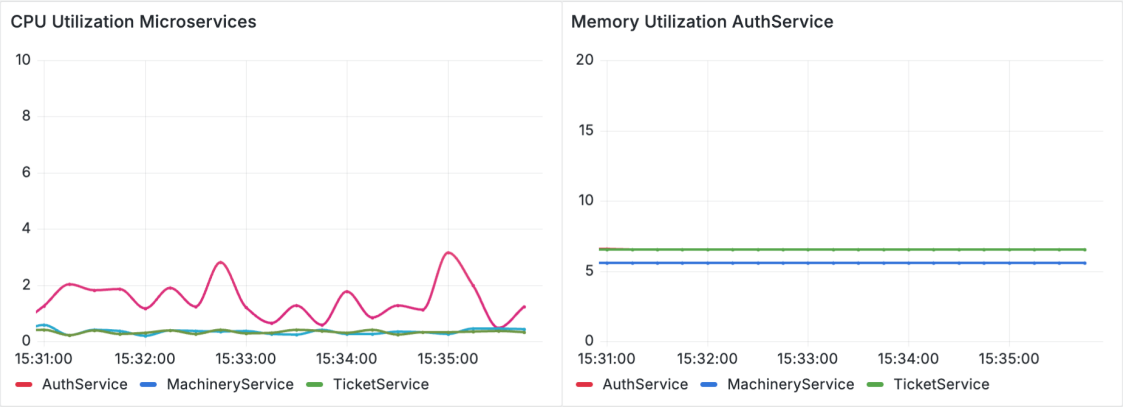

Figure 8.1. Resource consumption of microservices in low traffic situation

8.2 Medium Traffic Scenario

The medium load scenario is very similar to the previous one, but with less waiting time between requests (500 ms maximum in this case), which now are more frequent. We can see

from Table 8.2 that there are now more requests on each microservice, in particular on the IAM service, for the same reasons mentioned before. It's easy to spot an increased CPU utilization in the IAM service (see Figure 8.2, that settles between 2% and 8% on average, but not a significant memory usage which stays between 5% and 10%, probably because the performed operations don't use much memory. About the other two microservices, there seems to be little difference in resources consumption, which means that they can handle higher traffic just fine.

Request	Total requests	Avg. res. time (ms)	Min. res. time (ms)	Max. res. time (ms)
current	282	10.4	3	40
scan	293	18.8	5	176
refresh	284	15.9	5	62
login	98	175.1	136	271
logout	63	6.1	2	16
register	47	178.0	145	226
generate	34	16.8	5	30

Table 8.2. Medium traffic: total requests and response times for each tested request

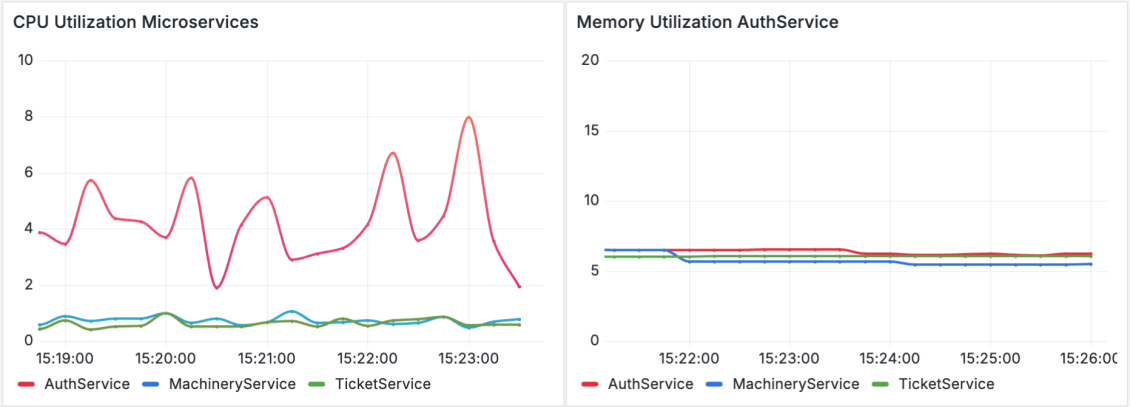

Figure 8.2. Resource consumption of microservices in medium traffic situation

8.3 High Traffic Scenario

Finally, this test targets the high traffic scenario, when many frequent incoming requests must be handled by the microservices, without slowing down excessively or crashing and become unavailable. To simulate this scenario, the waiting time between requests has been removed, in order to make sequential requests as fast as possible. This time the

probability of making a decode request (i.e., scanning a QR code, here called "scan") is higher, because it is probable that this scenario occurs when many QR scans are performed by many customers in a short span of time. As shown by the chart in Figure 8.3, also in this case memory remained stable between 5% and 10%. In Table 8.3 we can see a significant growth in the number of incoming requests, which is now more than 10 times the quantity of the previous scenario. Again, the majority of them are routed to the IAM microservice. While it is clear that both the machinery and the ticket services can easily handle this traffic, using less than 5% of CPU, this is not true for the IAM. In fact, the IAM significantly increased the CPU usage, reaching almost 40%, which is 4 times more than the medium traffic scenario and almost 10 times higher than the consumption of the other microservices. Even if this seems bad, the CPU percentage increased by 4 times in spite of a number of requests per second that was 10 times higher than before. It is improbable that this scenario will ever take place, so it can be considered the worst case scenario, at least until real performance test are done with actual customers.

Request	**Total requests**	**Avg. res. time (ms)**	**Min. res. time (ms)**	**Max. res. time (ms)**
current	3135	1.7	1	22
scan	3934	3.5	2	82
refresh	3196	3.1	2	47
login	1323	136.7	130	325
logout	63	1.1	0	15
register	407	134.9	130	207
generate	369	2.8	2	13

Table 8.3. High traffic: total requests and response times (in milliseconds) for each tested request

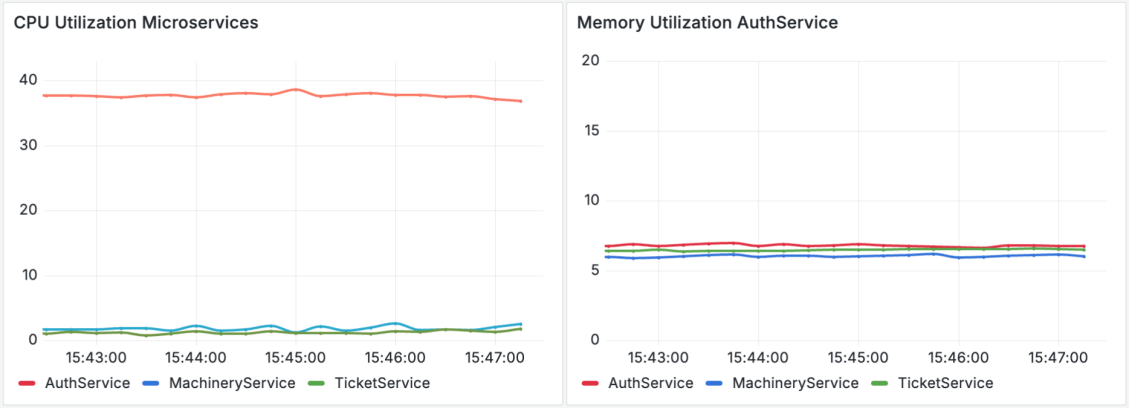

Figure 8.3. Resource consumption of microservices in high traffic situation

8.4 Results

Those tests were useful to have an idea about how the system behaves in normal and critical conditions, both from an external standpoint, which is represented by the response times of the incoming requests, and an internal standpoint, represented by the resource consumption. The analysis of the results leads to some considerations. First of all, the IAM is definitely the microservice that handles more requests. Even if the tests are scripted and therefore not entirely realistic, it seems true that this service is called very frequently to check the user identity or to manage an authenticated session. It uses a lot of CPU to perform important cryptographic operations and therefore the CPU consumption drastically increases when many registration or login calls are handled concurrently. This behavior suggests that this microservice should be constantly monitored to scale it horizontally when the CPU usage reaches a certain threshold. Ideally this threshold can be set at 60% or more, given that at 40% the service was operating correctly without any type of errors or delays. In any case, it is a good idea to always have two instances of the microservice up and running, with a load balancer that evenly distributes the requests among them, in order to lower the load on the service. The same operation can be done for all the other microservices and for the API gateway in particular, not just because of the traffic overloading, but also for resilience. In fact, having two instances means that if one goes down, all request can be temporarily redirected to the other one with minimal downtime.

Finally, comparing the average response times over the three scenarios for each request (see Table 8.4 and Figure 8.4), we can see that in the high traffic scenario, the microservices tend to respond faster. This might seem a strange behavior but it is not so uncommon. Multiple factors can contribute to the increased speed; for example, when more resources are requested, more CPU cores and threads are used and multiple operations can be parallelized, which may lead to quicker response times. Another reason, due to the nature of the simulation script, might be the re-utilization of already opened connections, that leads to cut the three way handshake phase, thus saving some time. Even if the scripted simulation is not completely realistic and the results can be different in a real production scenario, it was useful to analyze the obtained results to better understand some details on the system behavior, which would have been difficult to understand without stressing the application.

Request	Avg. in low traffic (ms)	Avg. in medium traffic (ms)	Avg. in high traffic (ms)
current	14.3	10.4	1.7
scan	30.0	18.8	3.5
refresh	22.3	15.9	3.1
login	180.5	175.1	136.7
logout	7.1	6.1	1.1
register	175.5	178.0	134.9
generate	27.7	16.8	2.8

Table 8.4. Comparison of the average response times in the three scenarios per request

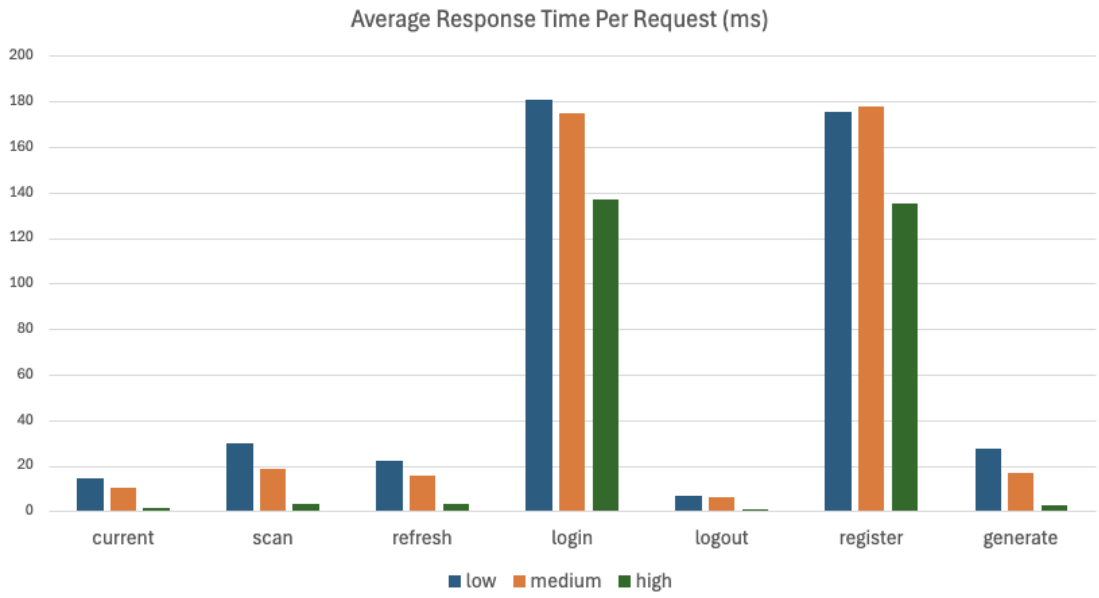

Figure 8.4. Average response time per request

Chapter 9

Conclusions and future work

This thesis has covered the entire lifecycle of the system's development, from requirements collection to design, implementation, and deployment. Additionally, performance tests were conducted to evaluate the robustness and responsiveness of the system. Based on the work done until now, several conclusions can be drawn about the system and its suitability for the company's needs.

One of the key objectives of this project was to build a system capable of supporting AROL's interactions with their customers. The distributed and modular architecture chosen for the application meets this need effectively. With AROL's customers located across different areas of the world, it was crucial to design a system that could help users to interact with the application without introducing excessive delays in response times. A user-friendly and responsive system is important for both external customers and internal AROL operators to ensure that interactions with AROL's products are seamless and efficient. A slow system would not only discourage customers from using the platform but also introduce delays in AROL's internal processes. The architecture's flexibility plays a critical role in the success of this project, because it allows for scaling and optimized performance.

The simulations have demonstrated that the system performs well, even under more demanding scenarios. Most operations have low response times, ensuring that users experience minimal delays during interactions. Furthermore, the microservices architecture ensures that each service consumes minimal CPU and memory resources relative to the incoming traffic load. Only the Identity and Access Management (IAM) service might require additional scaling as computationally heavy operations, such as logins and registrations, increase. However, the architecture's modular nature makes it easy to scale individual services as needed, providing flexibility and availability, thanks to the capability of having multiple parallel copies of the same microservice that handle the requests.

The system developed thus far meets the primary goals outlined during the requirements' collection phase. The application prototype is functional and delivers the necessary features to enhance both internal processes and customer communication. The current

state of the application is a significant step forward, but further steps are required to transition the system into full production. Specifically, the prototype is not yet connected to AROL's real data sources and it's not yet distributed across the world. However, thanks to the modular microservices architecture, integrating these real-world data sources is trivial.

To move closer to a production-ready system, there are some important tasks missing. A possible next step could be to make the application available to a controlled group of internal AROL operators and external customers. This would allow the collection of real usage metrics and feedback from end users, which can identify potential issues, bottlenecks, and areas for improvement. While the initial performance tests provided useful insights, they were based on simulated scenarios. Real-world data instead would provide a more accurate representation of system load, user behavior, and overall performance in a production environment.

In conclusion, this thesis has taken significant steps towards the development of a system that meets AROL's goals of improving internal processes and enhancing customer engagement. The architecture and design choices made in the project allowed to build a solid base for future work. While there is still progress to be made before the system is ready for production, the work done thus far has moved the project significantly closer to that objective. With more development and real-world testing, the application can potentially become a key tool for AROL's competitiveness in the global market.

Bibliography

[1] *AROL Closure Systems - About Us*. URL: https://www.arol.com/arol-canelli/. (accessed: 10/08/2024).

[2] *AROL Customer Care*. URL: https://www.arol.com/customer-care-for-capping-machines/. (accessed: 10/08/2024).

[3] *AROL Group*. URL: https://www.arol.com/arol-group-canelli/. (accessed: 10/08/2024).

[4] URL: https://www.arol.com/beverage-capping-machines/capping-machines-for-pre-threaded-flat-plastic-caps. (accessed: 10/08/2024).

[5] Wikipedia. *Value chain*. URL: https://en.wikipedia.org/wiki/Value_chain/. (accessed: 28/09/2024).

[6] Elizabeth Green. *The Rise of QR Codes: Transforming Customer Engagement in a Digital-First World*. URL: https://www.starleaf.com/blog/the-rise-of-qr-codes-transforming-customer-engagement-in-a-digital-first-world/. (accessed: 08/10/2024).

[7] *What Is Use Case Diagram?* URL: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-use-case-diagram/. (accessed: 28/09/2024).

[8] Chrissy Kidd. *Distributed Systems Explained*. URL: https://www.splunk.com/en_us/blog/learn/distributed-systems.html/. (accessed: 20/08/2024).

[9] Matt Tanner. *What Is a Monolithic Application? Everything You Need to Know*. URL: https://vfunction.com/blog/what-is-monolithic-application/. (accessed: 10/10/2024).

[10] Mike Loukides and Steve Swoyer. *Microservices Adoption in 2020*. URL: https://www.oreilly.com/radar/microservices-adoption-in-2020/. (accessed: 20/08/2024).

[11] *Microservice Architecture pattern*. URL: https://microservices.io/patterns/microservices.html/. (accessed: 10/08/2024).

[12] Mehmet Ozkaya. *Software Architecture Design Principles: SoC, SOLID*. URL: https://medium.com/design-microservices-architecture-with-patterns/software-architecture-design-principles-soc-solid-98611997c30e/. (accessed: 10/10/2024).

[13] Martin Fowler. *Bounded Context*. URL: https://martinfowler.com/bliki/BoundedContext.html. (accessed: 10/08/2024).

[14] Martin Fowler. *Domain Driven Design*. URL: https://martinfowler.com/bliki/DomainDrivenDesign.html. (accessed: 10/08/2024).

[15] David Mosyan. *CI/CD for Microservices*. URL: https://medium.com/@dmosyan/ci-cd-for-microservices-1b5582f3e1fd/. (accessed: 10/10/2024).

[16] *DevOps in the Era of Microservices: Challenges and Solutions*. URL: https://medium.com/@Wicultylearningsolution/devops-in-the-era-of-microservices-challenges-and-solutions-87a5cfe6f384/. (accessed: 10/10/2024).

[17] Microsoft. *Introduction to .NET*. URL: https://learn.microsoft.com/en-us/dotnet/core/introduction. (accessed: 11/08/2024).

[18] *Most popular technologies survey*. URL: https://survey.stackoverflow.co/2024/technology#most-popular-technologies/. (accessed: 10/10/2024).

[19] Microsoft. *What is the .NET SDK?* URL: https://learn.microsoft.com/en-us/dotnet/core/sdk. (accessed: 11/08/2024).

[20] *C# docs*. URL: https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/. (accessed: 10/10/2024).

[21] Microsoft. *Overview of ASP.NET Core*. URL: https://learn.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-8.0. (accessed: 11/08/2024).

[22] *What is Containerization?* URL: https://aws.amazon.com/what-is/containerization/. (accessed: 10/08/2024).

[23] Docker. *What is a Container?* URL: https://www.docker.com/resources/what-container/. (accessed: 12/10/2024).

[24] Docker. *Docker Engine overview*. URL: https://docs.docker.com/engine/. (accessed: 12/08/2024).

[25] *Docker Compose overview*. URL: https://docs.docker.com/compose/. (accessed: 20/08/2024).

[26] Docker. *Docker overview*. URL: https://docs.docker.com/get-started/docker-overview/. (accessed: 20/08/2024).

[27] *PostgreSQL*. URL: https://www.postgresql.org/. (accessed: 20/08/2024).

[28] *Asynchronous message-based communication*. URL: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/asynchronous-message-based-communication/. (accessed: 20/08/2024).

[29] *RabbitMQ*. URL: https://www.rabbitmq.com/. (accessed: 20/08/2024).

[30] *Azure Container Registry*. URL: https://azure.microsoft.com/en-us/products/container-registry/. (accessed: 03/09/2024).

[31] *Kubernetes*. URL: https://kubernetes.io/. (accessed: 16/10/2024).

[32] *What is version control*. URL: https://www.atlassian.com/git/tutorials/what-is-version-control/. (accessed: 10/10/2024).

[33] *Bitbucket*. URL: https://bitbucket.org/product/. (accessed: 10/10/2024).

[34] *Jira*. URL: https://www.atlassian.com/it/software/jira/. (accessed: 10/10/2024).

[35] Tom Collings. *Controller-Service-Repository*. URL: https://tom-collings.medium.com/controller-service-repository-16e29a4684e5. (accessed: 25/08/2024).

[36] *Understanding Dependency Injection: A Powerful Design Pattern for Flexible and Testable Code*. URL: https://medium.com/@sardar.khan299/understanding-dependency-injection-a-powerful-design-pattern-for-flexible-and-testable-code-5e1161dd37dd. (accessed: 16/10/2024).

[37] *Dependency injection in ASP.NET Core*. URL: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-8.0. (accessed: 16/10/2024).

[38] *Access Tokens*. URL: https://auth0.com/docs/secure/tokens/access-tokens. (accessed: 16/10/2024).

[39] *Refresh Tokens*. URL: https://auth0.com/docs/secure/tokens/refresh-tokens. (accessed: 16/10/2024).

[40] *JWT debugger*. URL: https://jwt.io. (accessed: 10/08/2024).

[41] *Introduction to JSON Web Tokens*. URL: https://jwt.io/introduction. (accessed: 10/08/2024).

[42] *Refresh Token Rotation*. URL: https://auth0.com/docs/secure/tokens/refresh-tokens/refresh-token-rotation. (accessed: 16/10/2024).

[43] *NGINX*. URL: https://nginx.org/en/. (accessed: 16/10/2024).

[44] *NGINX Configuration File's Structure*. URL: https://nginx.org/en/docs/beginners_guide.html#conf_structure. (accessed: 16/10/2024).

[45] *Docker Networking Overview*. URL: https://docs.docker.com/engine/network/. (accessed: 16/10/2024).

[46] *Kubernetes DNS for Services and Pods*. URL: https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/. (accessed: 16/10/2024).

[47] Joey Bartolomeo. *The benefits of observability*. URL: https://grafana.com/blog/2020/03/03/the-benefits-of-observability/. (accessed: 16/10/2024).

[48] *Serilog*. URL: https://serilog.net/. (accessed: 16/10/2024).

[49] *Grafana Loki*. URL: https://grafana.com/docs/loki/latest/. (accessed: 16/10/2024).

[50] *Prometheus*. URL: https://prometheus.io/docs/introduction/overview/. (accessed: 16/10/2024).

[51] *Grafana*. URL: https://grafana.com/grafana/. (accessed: 16/10/2024).

[52] Kubernetes. *Cluster Architecture*. URL: https://kubernetes.io/docs/concepts/architecture/. (accessed: 28/09/2024).

[53] Kubernetes. *Deployments*. URL: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/. (accessed: 28/09/2024).

[54] Kubernetes. *Horizontal Pod Autoscaling*. URL: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. (accessed: 10/10/2024).