



**Politecnico
di Torino**

Politecnico di Torino

Computer Engineering

A.y. 2023/2024

Graduation session December 2024

Development of a Back Office For Transaction Management in the Petroleum Sector

Supervisor:

Luigi De Russis

Christian Pio Petrucci

Candidate:

Giuseppe Maggio

Table of Contents

List of Figures	v
1 Introduction	1
1.1 Context	1
1.2 Goal	2
1.3 Thesis structure	3
2 Project analysis	4
2.1 Back Office	4
2.1.1 What is a back office	4
2.1.2 Importance of the Back Office	4
2.1.3 Example of Back Office	5
2.2 Thesis Domain	6
2.2.1 Overview of the System Structure	6
2.2.2 Categorization of Services and Transaction Analysis	7
2.3 Technical Analysis of the Current System	8
2.3.1 Overview of the Current System	8
2.3.2 Back End Analysis	8
2.3.3 Front End Analysis	10
2.3.4 Related Problems	10
2.4 Company Requirements	10
2.4.1 Functional Requirements	10
2.4.2 Non Functional Requirements	12
2.5 Architectural Patterns	13
2.5.1 Monolithic Architecture without IAM	13
2.5.2 Microservices Architecture with IAM	15
3 System Architecture	18
3.1 Architectural choice	18
3.2 Backend Analysis	19
3.2.1 Spring Boot	19

3.2.2	Spring Data JPA, Oracle e MongoDB	20
3.2.3	System Integration and Apache Camel	22
3.2.4	IAM: Keycloak	26
3.2.5	Spring Security	29
3.2.6	Grafana, Loki, Prometheus	35
3.3	FrontEnd Analysis	36
3.3.1	Angular	36
3.3.2	Angular Material	38
3.4	User Interface Design: Patterns, Heuristic Evaluation, and Proto- typing with Figma	39
3.4.1	Design Patterns	39
3.4.2	Heuristic Evaluation	39
4	System Implementation	42
4.1	Introduction	42
4.2	Project Setup	42
4.2.1	Maven and Dependencies	43
4.2.2	Docker and Services	44
4.2.3	BackEnd structure	44
4.2.4	FrontEnd structure	45
4.3	Security Services Implementation	46
4.3.1	KeyCloak Setup	46
4.4	Spring Security SetUp	49
4.4.1	Angular Security Implementation	53
4.4.2	Spring Cloud Gateway	55
4.5	Sales Points Implementation	56
4.5.1	Retrieving List Implementation	56
4.5.2	Retrieving Details Implementation	58
4.5.3	Updating Implementation	59
4.5.4	FrontEnd visualization	59
4.6	Terminal Implementation	60
4.6.1	Retrieve List Implementation	61
4.6.2	Adding Implementation	61
4.6.3	FrontEnd visualization	62
4.7	Transaction Implementation	63
4.7.1	Retrieving List Implementation	64
4.7.2	Transaction's Detail	64
4.7.3	FrontEnd Visualization	65
4.8	Batch Implementation	67
4.8.1	Back End - Sales Point Creation	68
4.8.2	Back End - Daily Transaction	72

4.8.3	Retrieving List Implementation	72
4.8.4	Retrieve CSV file	73
4.8.5	FrontEnd Visualization	73
4.9	Analysis implementation	74
4.9.1	Kafka Connect	74
4.9.2	Back End Implementation	76
4.9.3	Front End Visualization	77
5	Results and Comparison	79
5.1	Technology improvements	79
5.1.1	Java 21 and Tomcat	79
5.1.2	Angular vs AngularJs	80
5.1.3	IAM	81
5.2	Performance improvements	81
6	Conclusion	84
	Bibliography	86

List of Figures

2.1	Enterprise using ERP	5
2.2	Enterprise using CRM	6
2.3	Monolithic Pattern Diagram	14
2.4	Microservice Pattern Diagram	16
3.1	BackOffice Architecure	19
3.2	SpringData JPA structure	21
3.3	System Integration	23
3.4	from: Distributed Computing in Java 9, by Raja Malleswara Rao Pattamsetti, Packt Publishing, 2017, ISBN: 9781787126992	24
3.5	Apache Camel Structure	25
3.6	OAuth 2.0 authorization code flow	28
3.7	Servlet Filter	30
3.8	Authorization Flow	31
3.9	OIDC Code Flow	32
3.10	Here, the Web Browser has established an authenticated session with the Trusted Site. Trusted Action should only be performed when the Web Browser makes the request over the authenticated session. [7]	34
3.11	A valid request. The Web Browser attempts to perform a Trusted Action. The Trusted Site confirms that the Web Browser is authen- ticated and allows the action to be performed [7]	34
3.12	A CSRF attack occurs when a malicious site tricks the browser into sending a request to a trusted site. The trusted site perceives the request as valid and authenticated, since it originates from the user's web browser, and proceeds to execute the requested action. CSRF attacks are feasible because websites authenticate the browser, not the user.[7]	35
3.13	Google Trends: Javascript vs Typescript	37
3.14	Learning Curve: Familiar UI vs Unfamiliar UI	41

4.1	Setup of the modules	43
4.2	Realm main page	47
4.3	Realm's urls	47
4.4	Realm's roles	48
4.5	Realm's users	48
4.6	Example of assigned role to an user	48
4.7	Sales Points Page	59
4.8	Sales Point's detail	60
4.9	Terminals Page	62
4.10	Add terminal Page	63
4.11	Transaction's List	65
4.12	66
4.13	Transaction's details	67
4.14	Batch's List	73
4.15	Upload Batch	74
4.16	First Dashboard	77
4.17	Second Dashboard	78
5.1	Performance Comparison	82

Chapter 1

Introduction

1.1 Context

The services offered by a company are often the primary focus for us as end users. Departments such as marketing, sales, and customer service make up what is known as the front office. However, beneath this visible layer lies an essential engine that drives the company's efficiency: the back office.

In this crucial area, **Pay Reply** stands out, working closely with large companies to design, develop, and manage the complete lifecycle of diverse software solutions for both internal operations and external users. Pay Reply collaborates with major players in the petroleum sector, supporting the oil and fuel industry, as well as in the finance sector, working with banks and companies that provide digital payment services. Through its work, Pay Reply contributes to both the visible and invisible aspects of a company's success, ensuring that the front office and back office operate in harmony.

The development of an efficient and secure back office plays a crucial role in a company's overall performance. A well-designed back office streamlines internal processes, reduces operational bottlenecks, and supports smoother, more reliable workflows across departments. There are various types of back-office systems, each tailored to specific functions—such as finance, human resources, inventory management, and compliance—yet all share a common goal: to simplify and optimize the management of critical business areas. The focus of this work is a back office system designed to monitor petroleum-related transactions. From a high-level perspective, the service offered includes the installation of sales points when needed and when they are not already present. The sales points are equipped with payment terminals that enable customers to make electronic payments for various petroleum products, such as refueling with diesel, thereby generating a transaction record. For company owners, this model provides the flexibility to allow

customers to purchase products and make payments either through self-service or with the assistance of a gas station attendant, offering a cost-effective and efficient approach to managing operations. However, it also demands a more structured method to monitor and control these sales points effectively. This is where the back office system comes into play, providing a comprehensive overview of sales points, terminals, and transaction activities. The system empowers operators to detect and respond swiftly to anomalies, ensuring continuous and reliable service.

1.2 Goal

Driven by the need to track and analyze company transactions, the idea of creating a back-office system to offer as a service to businesses emerged over fifteen years ago. Since then, the technologies used have not been updated, and some, like AngularJS, have reached end-of-life, no longer receiving updates or security patches. This has led to significant security vulnerabilities, outdated code that fails to adhere to best practices, resulting in slower development of new customer-requested features, bug fixes and challenges in integrating new external services.

This context is the foundation of my thesis work, which focuses on completely rebuilding the core structure of the back-office service, both on the Back End and Front End. By leveraging modern technologies that enable faster development, adhering to the state of the art, and enhancing system security, the new solution addresses these shortcomings. At the end of the project, a final comparison with the current system will demonstrate the improvements achieved. We can divide the work I carried out into three stages:

1. **Study of system requirements and architecture's design**

The purpose of this phase is to analyze the functional and non-functional requirements of the back office system. This includes identifying which technologies are mandated for its development and where there is flexibility to explore different implementation options. The final objective is to establish the chosen system architecture.

2. **Implementation of the selected System Architecture** In this phase, we proceed with the full implementation of the system by constructing the entire connected infrastructure and subsequently developing the various features that the system is designed to provide. Each component is carefully integrated to ensure seamless operation and alignment with the specified requirements.

3. **Results and Comparison** In this final phase, we analyze the strengths of the new implementation, focusing on the impact of the modern technologies employed. Where possible, performance comparisons are made with the

existing software, allowing us to identify the improvements achieved with the new system, and potential future enhancements.

1.3 Thesis structure

The thesis is organized into several chapters that go on to analyze and explore the various stages of development:

Chapter 2. Project analysis: In this chapter, I provide a detailed exploration of what a back office is and its relevance to businesses today. Next, I examine and analyze the functional and non-functional requirements for software development. Based on this analysis, two distinct architectural design choices are presented, highlighting their advantages and disadvantages. These options are positioned at opposite ends of the spectrum of possible architectural choices.

Chapter 3. System Architecture: Having analyzed the pros and cons of the two architectural choices, this chapter shows the choice that was ultimately implemented, which is a combination of both options. The technologies used are numerous, and in this section, I describe them while explaining the reasons behind these choices, emphasizing how an understanding of design and the technologies utilized are closely interconnected. Furthermore, I analyze the user interface, discussing the design patterns employed, the results of heuristic evaluation, and the software prototypes developed during the design phase.

Chapter 4. System Implementation: I explore how the software implementation was carried out, beginning with the overall structure of the project. I illustrate the user interaction flows, starting with authentication and authorization, progressing through the various endpoints, and culminating in the achievement of the final goal.

Chapter 5. Result and Comparison: After examining the entire flow, from the importance of the back office to the actual implementation of our solution, this chapter contains the obtained results. I provide a comparison of several metrics between my solution and the currently used software, highlighting estimates that demonstrate improvements in performance, cost, and development.

Chapter 6. Conclusion: The sixth and final chapter examines the results achieved throughout the thesis and outlines the potential future developments of the work.

Chapter 2

Project analysis

2.1 Back Office

2.1.1 What is a back office

What is a Back Office and why it is so important for companies? The Back Office it is the part of a company responsible for providing all business functions related to its operations. Compared to the Front Office, the Back Office it is invisible to clients and provide support for the operations in the Front Office or help to perform company's business operations. These tasks are often administrative or operational and are crucial for an organization's day-to-day functioning Usually the departments that are part of the back office are:

- Human resources
- Operations
- IT
- Accounting
- Compliance

2.1.2 Importance of the Back Office

Although the support of the back office it is not visible to customers, it plays an important role in their experience. It enable the front office to deliver an high-quality service taking care that all the company's operation run smoothly. His work has always been a fundamental part of business, and the evolution of technology plays a crucial role. Speeding up and automatize the works of the back office means:

- Increase the efficiency of the company reducing human errors and automation of repetitive tasks
- Decrease costs and in this way optimize the resource allocation
- Increase the scalability of the company
- To improve the support in compliance with regulations and industry standards

2.1.3 Example of Back Office

Clear examples of Back Office software are the CRM (Customer Relationship Management) and the ERP (Enterprise Resource Planning). According to a study conducted in 2023 by eurostat, 43% of enterprise in Europe have an ERP, and 25,8% use a CRM.

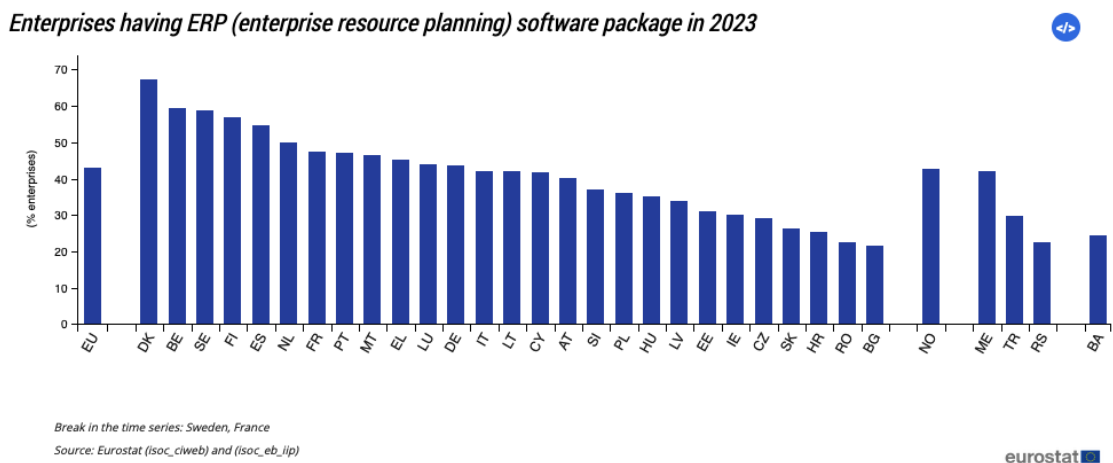


Figure 2.1: Enterprise using ERP

ERP applications binds together different parts of an organizations, allow them to communicate easily, reach all the information needed almost in real-time and create an integrated system. In this way we have a single source that coordinates all the different company's process despite their different nature



Figure 2.2: Enterprise using CRM

A CRM software is a system for managing all of company's interactions with current and potential customers. Every companies that interacts with customers have benefits having a CRM system, tacking track of many different sources and connection teams together making everyone's job easier

2.2 Thesis Domain

2.2.1 Overview of the System Structure

This Back Office system is designed specifically for companies in the petroleum industry that need to track and manage transactions at their retail points. To understand how this system works, it is essential to examine the entire structure. In this context, we define the **Sales Point** as any physical location where services are provided, and where payment transactions occur. These points can range from rest areas, gas stations, to electric vehicle charging stations.

At each of these points of sale, there are multiple **Terminals**. A terminal is the device responsible for facilitating the actual payment transaction. Terminals may include traditional POS systems, which are point-of-sale devices used for card payments, or more advanced kiosks that integrate POS systems but also offer

additional features and services. These advanced terminals may provide services like product purchases, ticketing, or digital top-ups, thus broadening the scope of customer interaction.

Once a terminal is used to initiate a transaction, it is processed and evaluated by a third-party service responsible for managing the payment process and interfacing with banking systems. It is important to note that the core function of the Back Office does not involve handling payments or their validation. Instead, its focus is on collecting, displaying, and analyzing transaction data.

2.2.2 Categorization of Services and Transaction Analysis

In industries like petroleum, where competition is constantly growing and customer demands are evolving, the role of services offered is becoming increasingly crucial. This is where the categorization of transactions into three main types becomes important: **FUEL**, **VAS (Value-Added Services)**, and **BANKING**.

- The **FUEL** category refers to the core service provided by the business, which is the fueling of vehicles with products such as gasoline, diesel, and their alternative substitutes. While this is the primary revenue-generating service, businesses must also expand their offerings to stay competitive.
- The **VAS (Value-Added Services)** category includes additional services that go beyond fuel sales. This can include the sale of third-party products, such as snacks, beverages, or car accessories, as well as loyalty programs that provide discounts through cards or points. These services help build customer loyalty and enhance the customer experience, turning a simple fuel stop into a multifaceted service offering.
- The **BANKING** category is also essential and reflects the growing trend of enabling customers to perform financial transactions, such as paying for utility bills (e.g., electricity, gas), making bank transfers, or purchasing insurance. These services not only provide added value but also encourage repeat visits, as they make the point of sale a convenient, all-encompassing service station.

Through the Back Office system, companies can perform in-depth analyses of their transaction data, identify trends in customer demand, evaluate the effectiveness of marketing campaigns, and refine their strategies for service optimization. Moreover, being able to track transactions across multiple categories allows companies to detect errors or discrepancies in their retail operations that would otherwise be difficult to spot.

2.3 Technical Analysis of the Current System

2.3.1 Overview of the Current System

The current Back Office system operates with a monolithic architecture. The Back End is built using Spring, a framework for enterprise-level Java applications that simplifies the development process. The Front End is developed using AngularJS, an older JavaScript framework. For data storage and management, the system relies on an Oracle Database, with database interactions handled via MyBatis, a persistence framework that simplifies mapping between Java objects and SQL statements. While functional, the system faces significant challenges due to outdated technologies and architectural choices. Let us delve into the specifics of each component and the associated issues.

2.3.2 Back End Analysis

The Back End is developed in Java SE 8, a version released in 2014. Although Java 8 introduced key features such as lambdas and streams, it lacks the modern capabilities offered by newer versions like Java 17 (LTS) or Java 23, which is the current version at the time of this analysis. This older version presents several challenges. It does not include the modular system introduced in updated version, which simplifies dependency management. Additionally, it does not benefit from the performance optimizations and resource efficiency improvements found in recent releases. Most critically, Java 8 has reached its End of Life (EOL) and no longer receives security updates or patches, exposing the system to potential vulnerabilities and exploitation risks.

The system runs on *Tomcat 8.0.53*, a servlet container released in 2018. Like Java 8, Tomcat 8 has reached its EOL, and the recommended version is now Tomcat 10.x or higher, which offers better performance, modern features, and ongoing security updates. The combination of outdated Java and Tomcat creates significant vulnerabilities, compatibility issues, and limits integration with modern services or WAR applications built with current technologies.

Challenges in System Integration

As with any software system, there is a continuous maintenance phase, during which businesses request new features or integration with external services. However, the use of outdated technologies creates a vicious cycle:

1. New WAR files or services may require modern Java and Tomcat versions.
2. The current system cannot support these versions due to compatibility issues.

3. This necessitates costly and time-consuming workarounds, increasing the overall cost of maintaining the system.

MyBatis

The system uses **MyBatis** for database interaction, a persistence framework that bridges Java objects and SQL statements by allowing developers to define custom queries. This is achieved through XML configuration files where the queries are defined.

Maintaining MyBatis can become increasingly challenging due to its reliance on manually crafted mappings and SQL, which can lead to repetitive code and increased complexity. Unlike modern ORM frameworks such as Hibernate, the one used in my thesis's work, MyBatis does not offer a full abstraction layer for database interactions, which can hinder development speed and increase maintenance overhead.

Authentication and Authorization

The system's authentication and authorization module was developed internally and is managed locally. While this approach offers full control over the implementation, it introduces significant challenges due to the need for continuous updates to address vulnerabilities and comply with new standards. Without constant monitoring and updates, locally developed systems are vulnerable to attacks. Further this renders very difficult the possibility to extend the Back Office with other services allowing a single point of authentication, and enabling the Single Sign-On (SSO).

Batch and Crontab

Many system operations, such as adding new sales points or performing specific analyses required by a particular company, are executed by **batch** jobs that run at scheduled times. The scheduling tool used to control the execution of each batch is **Crontab**, which allows defining a specific schedule for running scripts or commands.

Some of the batch jobs in use perform predefined queries at regular intervals, such as gathering data or generating reports. Others require the processing of CSV input files, with the generation of output CSV files. One significant issue in the current system is the need to have a specific name for each input CSV file, with no notification provided until the batch job is executed, which is managed by cron. This requirement to follow strict naming conventions makes operations that should be simple and repetitive very difficult. Additionally, the lack of a notification system complicates error management since the user is not alerted

until the batch is completed, increasing the likelihood of incorrect operations and potentially compromising the system's reliability.

2.3.3 Front End Analysis

The Front End is built using **AngularJS**, a framework released in 2010 by Google and based on JavaScript. It has been completely surpassed by its next-generation counterpart, Angular, and Google ended its support at the end of 2021. As a result, no security patches will be provided, leaving the application vulnerable to threats like Cross-Site Scripting (XSS). The transition to Angular, which includes native features such as TypeScript support and a modular architecture, has significantly reduced the AngularJS community, making it increasingly difficult to update libraries and resolve issues.

These limitations make it increasingly challenging to maintain and scale the Front End. Organizations that continue to rely on AngularJS face higher costs and risks, especially in industries with strict security and regulatory requirements.

2.3.4 Related Problems

The combination of an outdated Back End, reliance on deprecated technologies like AngularJS, and a custom authentication system has rendered the Back Office system inefficient, insecure, and challenging to maintain. The inability to integrate modern features, coupled with performance limitations, hinders the system's scalability and its ability to meet the evolving demands of the petroleum industry. Transitioning to updated technologies and architectures is essential to ensure the system's longevity, security, and efficiency.

2.4 Company Requirements

In this section I am going to analyze the Functional and Non Functional Requirements of the Back Office system, as requested by the company. The system was designed to meet the business needs of the organization. The Functional Requirements provide detailed information on the required operations and the features implemented to support them. On the other hand, Non Functional Requirements include aspects such as the system's performance, security and scalability ensuring that not only the Back Office meets the operations required but it is capable of evolving on business needs and be secure.

2.4.1 Functional Requirements

- **Access and Security:**

The system must require user to authenticate through a login page with email and password. Every user will have a role that discriminates the different operations that is allowed to perform. I was free to decide between a local login, with a dedicated page, database and a session or go through an IAM solution.

- **Operational Section Management:**

The system is divided in different sections, accessible through a menu and support different operations like search, filter, insert and detail.

- **Sales Point:**

In this section the user is able to search all the sales point on the system, with a related possibility of filtering between different fields and dates. Data will be shown in a table, using a pagination system and having the possibility to sort in the different fields. Selecting a specific sale point the user is able to see the detail and update it if has the authority. Further we will be able to download a CSV file with the entire collection of sales point.

- **Terminal:**

In this section the user is able to search all the terminals on the system. Every terminal is related to a specific Sales Point and the user must have the possibility of filtering through the relevant information. There is the possibility of adding a new terminal and associate it with a specific sales point. By default the terminal is '*Inactive*', when a third part company confirm the installation, it will be changed in '*Active*'

- **Transaction:**

In this section the user is able to search all the transactions on the system. Every transaction is related to a specific sale point and terminal and we are able for filtering for the different fields of the transaction. Data are shown in a table, using a pagination system and having the possibility to sort in the different fields. Selecting a specific transaction the user is able to see the detail. Further the user must have the possibility to see dashboards related to the execution of the transactions

- **Batch:**

In this section the user is able to search a list of batch execution. There is the possibility to filtering the research. Moreover the user must be able to add a new schedule in the system uploading a CSV file and to download for each schedule the result of the batch in a CSV file. It is required the possibility to add a Batch, programmed with **Crontab** that allow the user to add new sales point and one to retrieve all the transactions at the end of the day.

Not all operations will be allowed for end users, but only for the employees of the agency owning the Back Office. Specifically, users will not be able to create new terminals, new batches, or new user accounts. These tasks are delegated to the company upon specific client request.

2.4.2 Non Functional Requirements

- **Technological Architecture:**

- **REST APIs:**

- Every backend operation must be reach through a set of endpoints , using HTTP methods (GET,POST, PUT, DELETE) following all the best practice developing the REST APIs.

- **FrontEnd Technology:**

- For the development of the FrontEnd part is required to use the Angular framework, with the LTS version.

- **BackEnd Technology:**

- For the development of the BackEnd part is require to us Spring Boot, exploiting its potential to manage microservice.

- **Database:**

- The system requires the use of Oracle database for the data storage. While Oracle is mandatory, the specific design of the tables are left to the discretion of the programmer, allowing flexibility in defining the schema according to the application's needs and requirements. Further is give the possibility to use a second database different from Oracle.

- **Security:**

- Every user must be authenticated for have access in the system, the implementation of this is left to the programmer. There is flexibility in choosing between and Identity and Access Management (IAM) solution or design a custom authentication system. If case of the implementation of the second choice care must be taken in encrypt Passwords and save the hash into the database, handling credentials and roles.

- **Scalability:**

- The real system is expected to handle a large amounts of data, the system must be able to scale and do not reduce the performance as the amount of data increase.

2.5 Architectural Patterns

The architecture of the software define the structure and the organization of the system, showing the different components and how they interact. Choosing the right architecture can impact drastically the performance of the system, the maintainability and the scalability. There are many different possible architectural patterns, such as monolithic, microservice and layered structure. Each of them has its strengths and weaknesses. For my work I identified two possible different architectural choice, with different pro and cons. Many different possibilities could be found, but I decided to show and study the most different choices among them, thus going deeper into those which are their most intrinsic differences. In this analysis, I will evaluate the presence or absence of an IAM (Identity and Access Management) system for managing users and their permissions.

2.5.1 Monolithic Architecture without IAM

The first architecture that I identified is a Monolithic pattern without the presence of an IAM service. In this particular choice, the entire application is built as a single unit. All the components are contained and deployed as one entity. The various modules will communicate with each other through simple call function, without the need of network. The filesystem of the machine server will be use for store the batch file the user may provide, and a backup will be store even in the database. In this case the authentication and authorization processes would need to be handled from the application itself.

- **Pros**

- **Simplicity of development**

The monolithic architecture is faster and easier to develop, having all the component in one big system make simpler to manage the dependencies and the communication

- **Simplicity of deployment**

We have only one deployment unit to deploy, without dependencies.

- **Low cost at beginning**

As a result of having to deploy a single unit, infrastructure costs will also be lower

- **Easier Testing**

Testing results be more straightforward because all component are located in on environment. This allows for end-to-end testing to verify that the application behaves as expected.

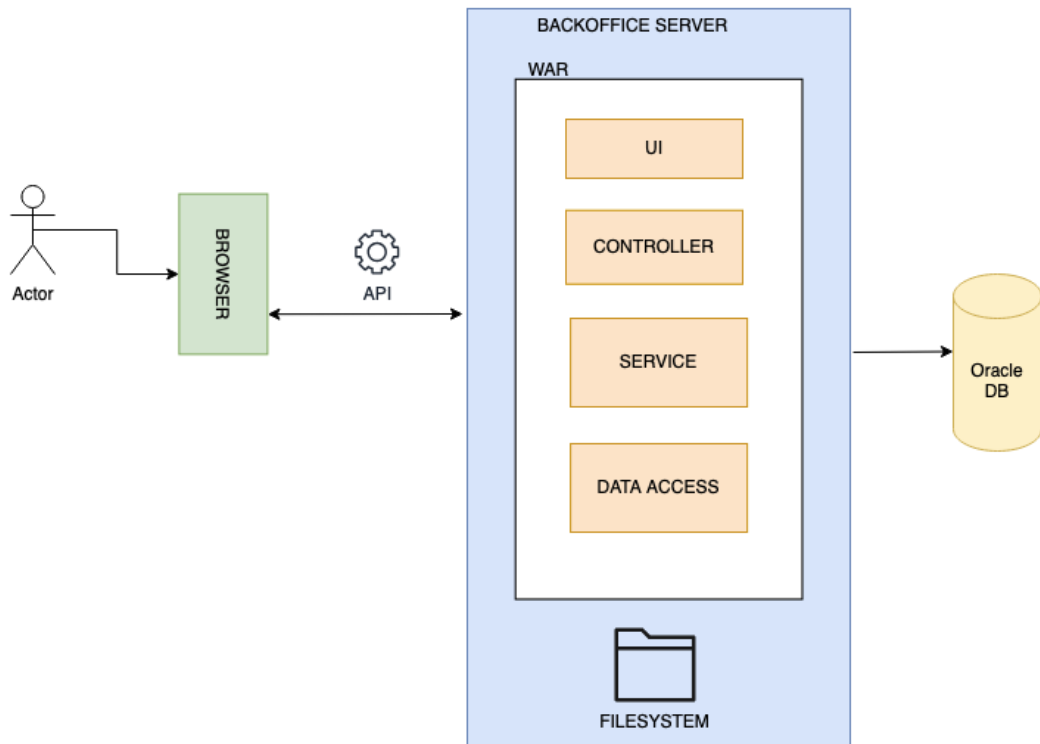


Figure 2.3: Monolithic Pattern Diagram

- **Cons**

- **Performance issues**

There is a single Database and this is a bottleneck for our system, it is possible to improve the queries but there is a limit of optimization

- **Less flexibility**

In this type of pattern we are tight to the technologies we used, changing it means rebuilt all the entire application. The risk is that the application at some point is big enough to make impossible the possibility of a migration.

- **High code coupling**

With the increase of the system we risk to end up with a *spaghetti code*, making it harder to understand and decreasing in this way the time for release a new feature.

– **Slow deployment**

If in one hand the deploy is easy to do having everything in one piece, on the other hand even a small change requires a redeployment of the whole application.

– **Security**

The complexity related to the registration and authentication of a user, together with all the implied processes (domain federation, password recovery, multi-factor authentication, user banning, password strength validation, password expiration, ...) and the corresponding hardening necessary to prevent known attacks determine a strong push toward the adoption of well-know and field-proof solutions

2.5.2 Microservices Architecture with IAM

A second possibilities is given by the use of a Microservices pattern with the presence of a IAM service. In this choice I split the entire BackOffice in three main microservice: **BatchService**, **SalesManagementService** and **AnalyticsService**. The *BatchService* will include all the logic related of the schedule, performing the CRUD operations for the batch. In the *SalesManagementService* we found the core of our system, all the logic about Sales Point and Transaction resides there. In the end we have the *AnalyticsService*, that is responsible of taking, analyze and send all the statistics that the company needs. For this service I decide to propose a schema-less database like MongoDB, having less restriction about the structure of the database allows this service of growing up, including more different statistics over time without effecting the performance. There will be a direct communication between the *SalesManagementService Database* and the *Analytic Database*, provided by a message broker. In this way every update is made in the first one will be reflected in the analysis without interrupt the main flow operation, increasing the performance. As before the **BatchService** will use both the filesystem and the database of the machine for save the batch file. The authentication and the authorization is handle by an IAM service.

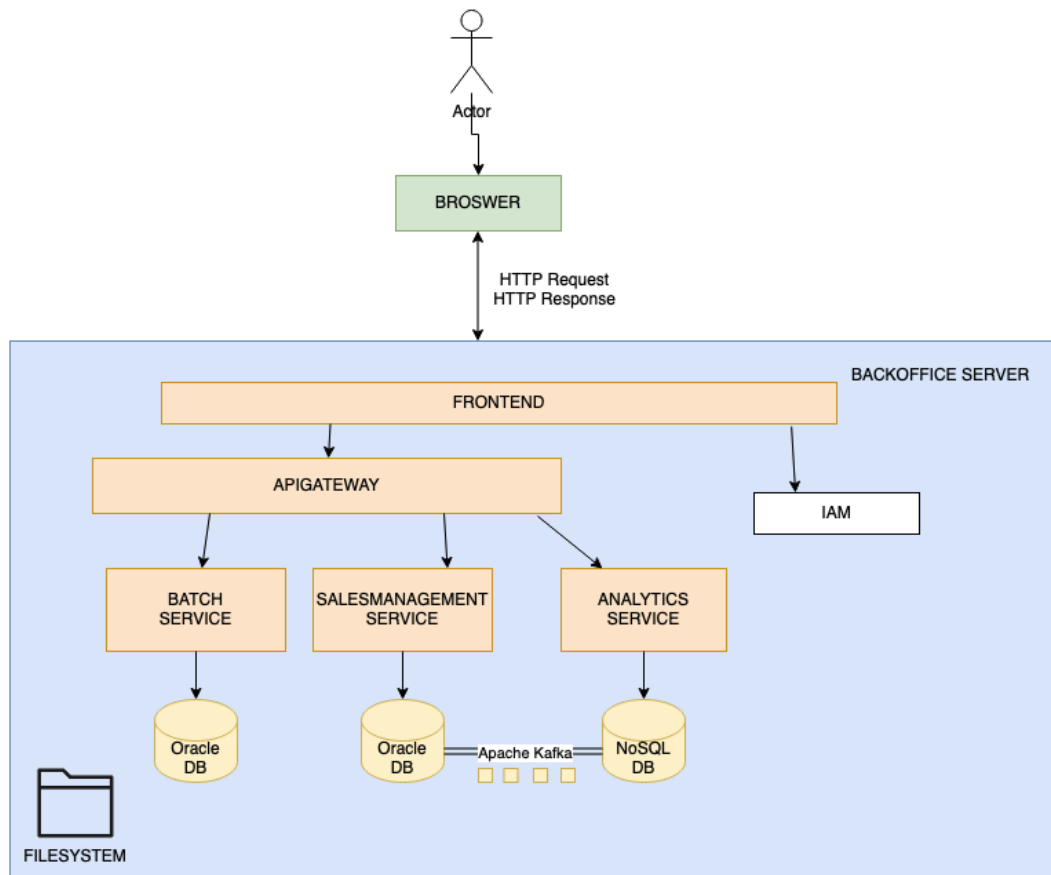


Figure 2.4: Microservice Pattern Diagram

- **Pros**

- **Scalability**

- Splitting the structure of the system in microservices allow them to scale independently, based on demands.

- **Decoupling of services**

- From the moment the every service works on a specific part of the system, this allow the developers to update the system faster and in a clean way. In a competitive world, Time is a key factor.

- **Improve flexibility**

- Each microservices can be developed with different technology, frameworks and databases. Changing technology or update it in a new version does

not block the entire system because affect only a piece of it

– **Fault isolation**

If a microservices fails, the overall system will continue to operate, and make the research from the problem faster and easy to reach.

– **Security with IAM**

The IAM service manages the authentication and authorization of the users, avoiding gaps and problems that might occur in a personalized implementation. The overall security will improve, with a better handling of tokens and technology use. Further modern IAM use a personalized login page, making the development faster for this point of view.

• **Cons**

– **Increased Complexity**

In this patter the complexity increase due to the need to manage different and independent services. Communication between services become more challenging compared to the monolithic option.

– **Increased latency**

Due to the network communication between different services, there is a potential latency issues. This may affect the performance.

– **Consistency**

Each microservices it is connected with a specific database and store own data, can be challenging ensure consistency among them. The usage of possible distributed transaction will increase the complexity of the sistem.

– **IAM Implementation**

IAM improves the security but its implementation may add complexity to the system

Chapter 3

System Architecture

3.1 Architectural choice

I have decided to adopt a hybrid approach, combining elements from both of the previous patterns to strike a balance between pros and cons. In more detail, I have chosen a *microservice architecture* with the presence of an *Identity and Access Management (IAM)* system. The key modification is removal of the *BatchService*, which is considerably smaller in scope compared to the core business functionality, and integrated into *SalesManagementService* rather than exist as a own service. In this architecture we can find two main services:

- **SalesManagementService**, responsible of all the main functions of the BackOffice, related to Sales Point, Transactions and Batch
- **AnalyticsService**, responsible of queries MongoDB and manipulate the data for extract statistics

The reason that lead me to remove the *BatchService* as a single service was due to its size and simplicity, a good tradeoff for this BackOffice is between performance and complexity. In this way we avoid a possible performance bottleneck arise from the communication between the services. On the other side I have opted to keep the microservice architecture, leaving the flexibility to extend the system or to change technology if needed in a more easy way compared to a monolithic option. Further analytics operation can be decrease the performs of the system, it may need to read entire tables and making complex operation, having it in a separate service with a separate database improve the performance of the Back Office, without interrupting the core functionalities. The presence of IAM, compared to a personalized solution, increase the security of the system.

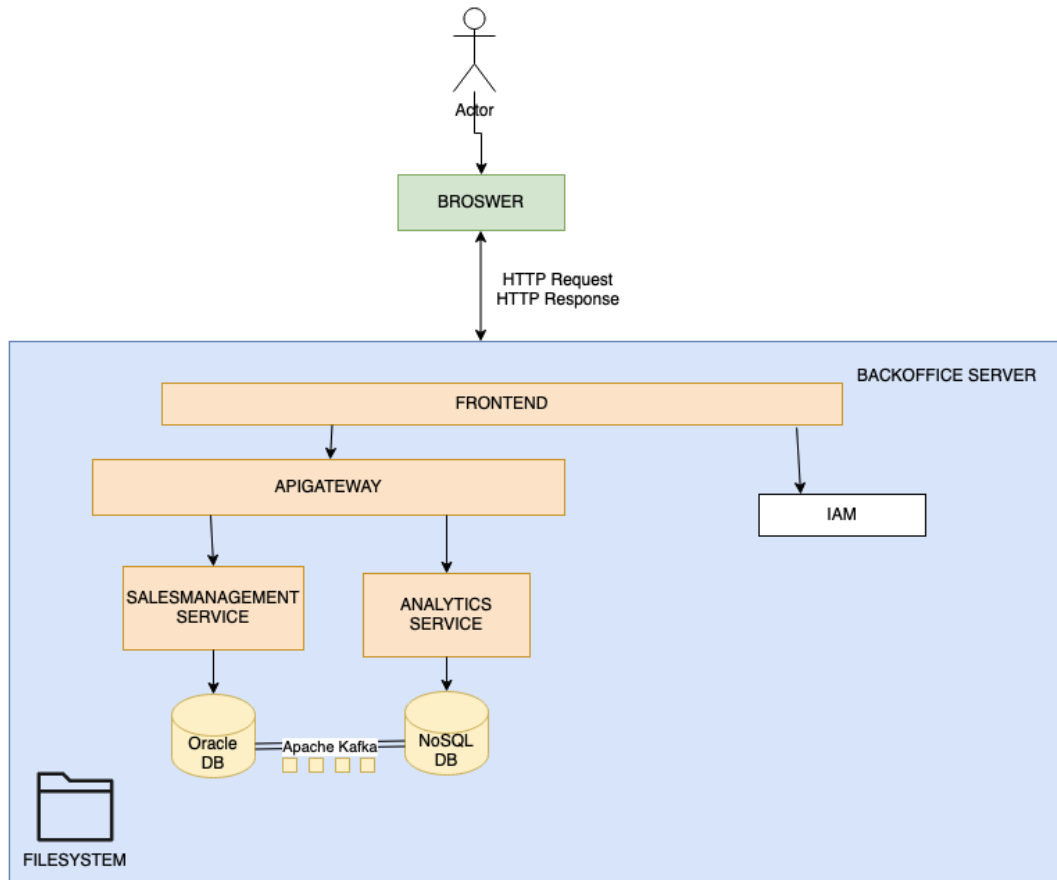


Figure 3.1: BackOffice Architecture

3.2 BackEnd Analysis

The BackEnd of our software contains many different technologies and frameworks inside, trying to take advantage of each of the.

3.2.1 Spring Boot

Spring Boot is a framework for developing web applications and it builds on top of Spring Framework. Maintaining all the core concepts of a Spring Application, **Inversion of Control**, **Dependency Injection** and **Aspect Oriented Programming**, Spring Boot aims to make the development easier, enabling to focus more on writing the business logic rather than the configuration of our system.

Convention Over Configuration is the main principle of Spring Boot, the framework will provides defaults options for many settings and configurations:

- **Auto-Configuration**

Spring Boot auto configure automatically our applications based on the dependency we add. *springboot-starter-web* is the main starter for our web application, allowing spring boot to setup our application providing an MVC architecture and have a template for our RESTful APIs and REST endpoints

- **Embedded Server** Our application is bundled with an **embedded Tomcat** server. In this way our application can run from the beginning without needing an external web server. If needed it is possible to change it with our external web server or to replace Tomcat with Jetty or Undertow.

- **Property Files** External configuration for our application is made possible by changing the **application.properties** or **.yml**,

3.2.2 Spring Data, Oracle e MongoDB

The Data Access Layer of our application is manage by Spring Data, a Spring Framework that makes easy to use data access technologies, relational and no-relational database as our case. In particular I used **Spring Data JPA**(Jakarta Persistence API) for accessing to Oracle database and **Spring Data MongoDB** for accessing Mongo Database. This allowed me to evaluate and compare various database access techniques and assess them against the system currently in use.

Spring Data JPA

Allows the developer to interact with the relational database through Java objects, making the implementation easier and more readable. This thanks to an **ORM (Object-to-Relational-Mapping)** software layer that automatically converts tables rows into objects and vice-versa, keeping the two representations in sync. In the standard configuration the ORM implementations is provided by **Hibernate**. The main concepts of Spring Boot JPA are:

- With the **@Entity** annotation, we declare that the class represents a table in a relational database. This allows us to define relationships between tables, manage IDs, enforce constraints on columns, and apply many other configurations, all through the use of annotations.
- **Auditing** is supported by JPA, that insert extra properties and annotations on entities like the creation date of the record or the last modified date of it using the **@CreatedDate** Annotation and the **@LastModifiedDate**

- The **Entity transaction**, a database transaction that can be either committed or rolled back according to the application state, creating a safe code where to operate
- The **Entity Manager**, an object that encapsulates the connection to the database, and offer the methods for queries it through the **Query** object, that encapsulate a custom query in *JPQL*, a custom query language that JPA provide to perform queries with object-oriented concepts
- **Repositories**, classes that extends interfaces already built in in Spring Boot JPA as *SimpleJpaRepository* and increase the level of abstraction over database access, reducing even more the boilerplate. It offers the possibility to automatically generate and implementing at runtime simple CRUD queries just analyzing the function name, and to implement even *Pagination and Sorting*. There is the possibility to customize our queries with the **@Query** Annotation, that use *JPQL*, and to perform our methods inside a Transaction with the **@Transactional** Annotation

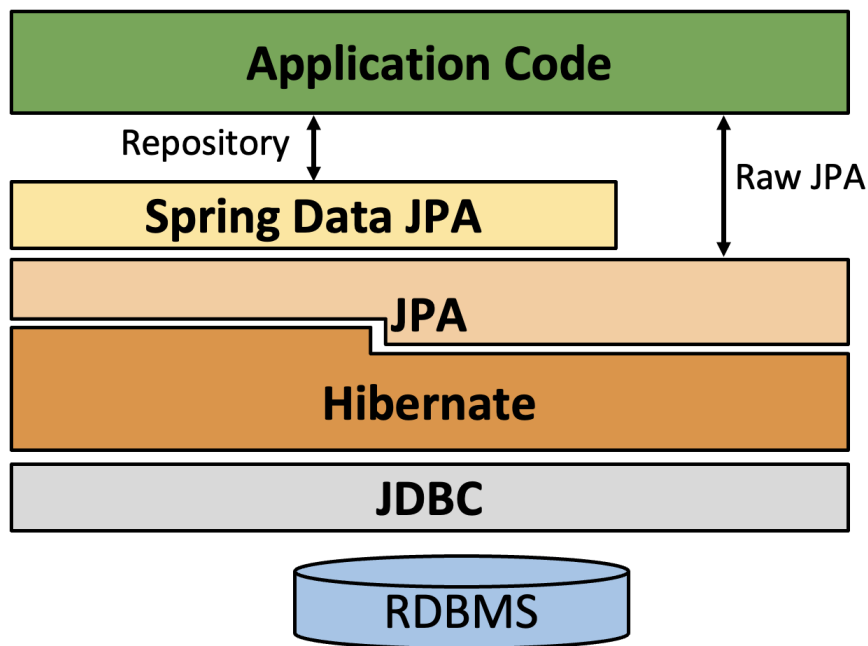


Figure 3.2: SpringData JPA structure

Spring Data MongoDB

Spring Data MongoDB allow us to get access to a mongo database in a convenient and easy way. The major abstractions are provided by the **MongoTemplate** helper class. It provides a rich set of features for interaction with the database, offering a set of built in operation for manage the CRUD operations, query the MongoDB documents and a way for mapping the MongoDB documents into our domain object

- **Documents**, as for Spring Data JPA works with Entity for manage the database's tables, Spring Data MongoDB use the **@Documents** Annotations for mapping a class in a mongo db document. With these classes we have the possibility to create relationships between documents, normalized and denormalized and create indexes
- **Mongo Repositories**, is the counterpart os JpaRepository. Extending the *MongoRepository* Spring provide us a set of ready-to-use methods for CRUD operations, the possibilities to define custom queries and to implements already the sorting and the pagination. In addition of the **@Query** Annotations that as before offer at the developer the possibility of customize the query, Spring Data MongoDB provide us support for perform **Aggregations**, bringing the documents into a pipeline that perform a set of operations and transformation for getting the result we desire.

3.2.3 System Integration and Apache Camel

System integration means connecting different systems or tools into a single, extensive system that functions as one. When it comes to software, system integration is often defined as the process of connecting disparate IT systems, services, and/or software to make them all operate together smoothly.

The main goal of following this approach is to allow different tools to be used for specific purposes, as most tools are designed to serve only a limited range of needs. Previously, the only solution for achieve the same functionality was to building large, monolithic applications to cover all features. This approach is complex, with an increase of the costs and often poorly suited to the individual functions that each tool is intended to provide. [1]

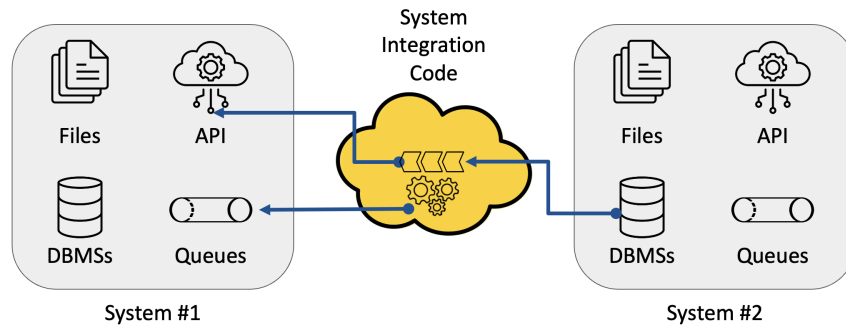


Figure 3.3: System Integration

System Integration can take place in different levels:

- **Data level:**

We are in this level when we are facing a copy of data between two different databases. We want to transfer continuously the new data that arrive, guaranteeing that any change that happens in one DBMS is reflected into the other DBMS

- **Application logic level:**

At this level two applications communicate invoking business logic or service of the other, often using APIs that are exposed.

- **Presentation level:**

The user interface receives information from various integrated systems and presents it in a unified, coherent manner to the end-user. This level ensures that data from different sources is displayed consistently, providing a seamless user experience.

It is clear that integrating different systems requires effort and costs.

Enterprise Integration Pattern (EIPs) is a standardized solution for common integration problems, providing different methodology, design and communication flows for integrating systems, simplifying integration challenges.

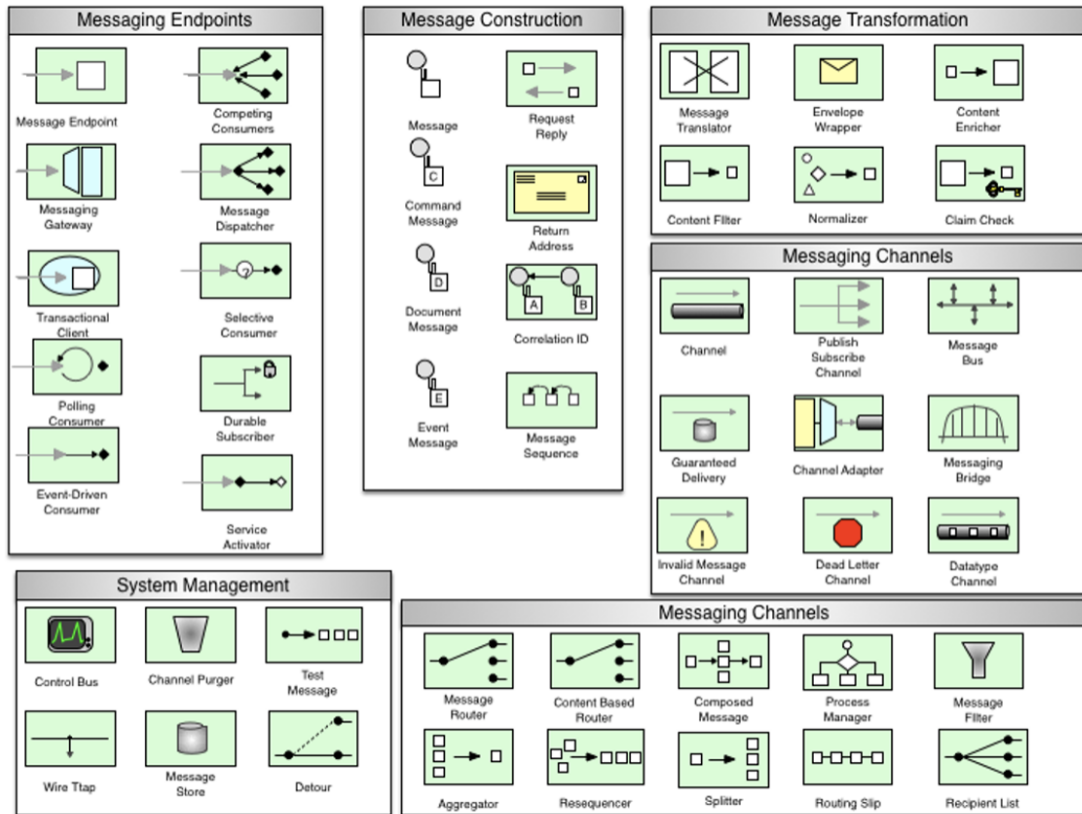


Figure 3.4: from: Distributed Computing in Java 9, by Raja Malleswara Rao Pattamsetti, Packt Publishing, 2017, ISBN: 9781787126992

How integrate the EIPs in our application? Spring offers for developers two main approaches:

Spring Integration

The first approach is the use of *Spring Integration*, which gives us the ability to make different systems interact by exchanging light messages and also to be able to integrate our system with other external ones through declarative adapters. These adapters provide a high-level abstraction over Spring’s capabilities for remote communication, messaging, and scheduling. The main objective of Spring Integration is to enable a straightforward approach to building enterprise integration solutions, while preserving a separation of concerns that supports maintainable and testable code. [2]

Apache Camel

Apache Camel is an open source integration framework that simplifies integration tasks implementing standard EIPs. It provides an object-base implementation of most EIPs and offers a collection of components for interaction with different APIs and protocols.

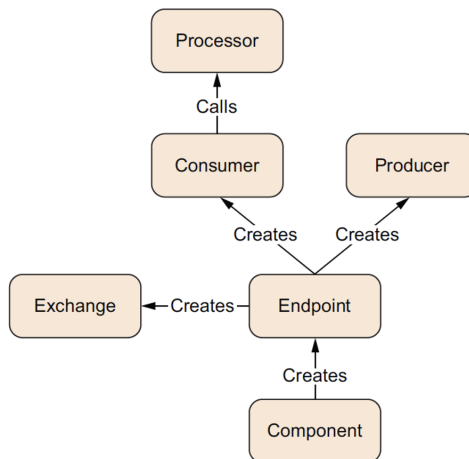


Figure 3.5: Apache Camel Structure

Camel architecture consist of:

- **Components:** In Apache Camel, components are the essential building blocks for connecting to external systems and services. They serve as bridges that enable Camel to interact with various endpoints, such as databases, queues, APIs, and files.
- **Endpoints:** Objects created by components that represent externally connectable systems that act as sources or destinations of messages. We identify:
 - **Producer:** The entity that is capable of sending messages to an endpoint
 - **Consumer:** The entity that consumes messages from an endpoint
- **Routes:** Routes in Camel are central to the processing logic. A route consists of a flow that begins at an endpoint, passes through a series of processors and converters, and concludes at another endpoint. It is possible to chain routes by designating another route as the final endpoint of a preceding one. Additionally, routes can incorporate various features such as Enterprise Integration Patterns (EIPs), as well as support for asynchronous and parallel

processing. A route represents the sequential movement of a message from an input queue, through various operations—such as filters and routers—until it reaches its destination queue (if applicable). The simplest definition of a route is a series of connected processors. Each route in Camel is assigned a unique identifier, which is used for logging, debugging, monitoring, and controlling the execution of routes. Furthermore, routes have a single input source for messages, effectively linking them to an input endpoint. [3]

Usage in our system

The BackOffice I developed makes use of our server's FileSystem for the entire Batch management. Specifically, when a user from the appropriate screen sends a .csv file to the server to run a particular batch, it is parsed, converted to json and saved in the file system thanks to apache camel. From there the system will forward it to a specific directory waiting for the time of its execution.

Each batch has a scheduling that is done through **Crontab**, a tool that allows particular actions to be executed periodically. Specifically, scripts are to be generated that will be executed periodically or REST Endpoints that are called when the timeout expires. Managing it in this way is complex, but with SpringBoot we have two ways to manage it much more quickly and efficiently: The **@Scheduled** Annotation or with **Apache Camel**. The Cron Component of apache camel has made the configuration of it much more immediate, avoiding the use of special scripts.

3.2.4 IAM: Keycloak

As anticipated at the beginning of this chapter, the Back Office will manage the identity of the users with an Identity and Access Management (IAM) service, but what exactly is an IAM, what are its responsibilities and how it works?

In the early days of computing, IAM was a relatively simple concept. Most computers were mainframe systems used in academic and government settings, and user access was typically controlled by basic username and password systems.

By the 1970s, the rise of multi-user systems necessitated stronger access control and authentication procedures. In the late 1960s, researchers at the Massachusetts Institute of Technology (MIT) developed one of the earliest identity and access management (IAM) systems, known as the Compatible Time-Sharing System (CTSS). One of the significant challenges for IAM at that time was the lack of consistent authentication and access control systems. This led to the creation of protocols such as Kerberos in the 1980s at MIT, which aimed to provide a uniform approach to authentication and access control across multiple systems over insecure channels, like digital networks. As technology progressed, IAM

systems became increasingly sophisticated, evolving from simple username and password authentication to more advanced mechanisms for authentication and access control.[4]

IAMs follow two main approach, **OAuth 2.0** and **OIDC 1.0**

OAuth 2.0

OAuth introduces an authorization layer that separates the role of the client from that of the resource owner. In OAuth, the client requests access to resources managed by the resource owner and hosted on the resource server, receiving its own set of credentials distinct from those of the resource owner.

Rather than using the resource owner’s credentials to access protected resources, the client obtains an access token—a string that specifies a particular scope, duration, and other access attributes. Access tokens are provided to third-party clients by an authorization server with the resource owner’s consent. The client then uses this access token to request access to the protected resources hosted by the resource server.

OAuth defines four roles:

- **Resource Owner:** An entity that has the authority to grant access to a protected resource.
- **Resource Server:** The server that hosts protected resources. It receives and responds to requests for these resources using access tokens.
- **Client:** An application that requests access to protected resources on behalf of the resource owner and with their authorization. The term “client” does not imply any specific implementation characteristics, such as whether the application runs on a server, desktop, or other devices.
- **Authorization Server:** The server responsible for issuing access tokens to the client after successfully authenticating the resource owner and receiving authorization.

[5]

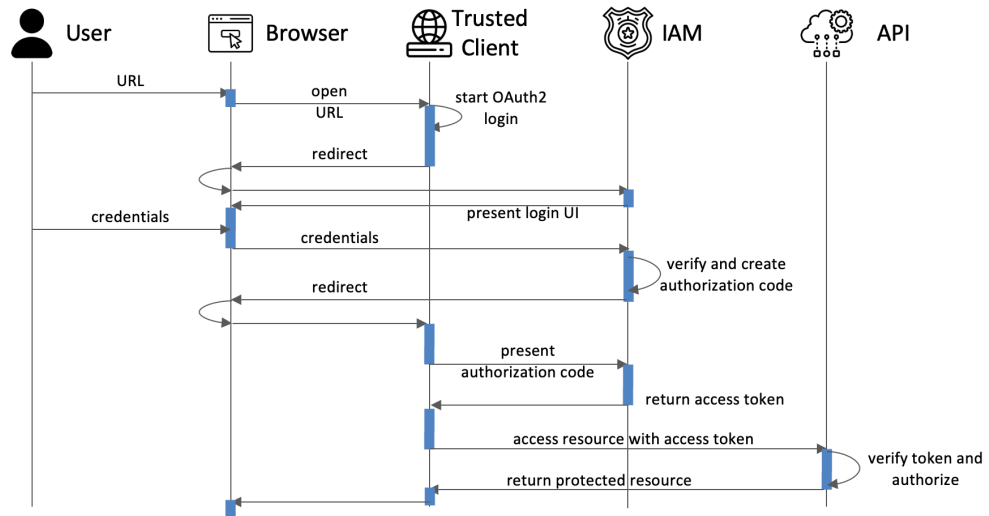


Figure 3.6: OAuth 2.0 authorization code flow

In order to act as client, an application must be registered with the IAM that will then authenticate the application via a credential pair: **clientID** and **clientSecret**

1. The user try to get a resource without providing any Authorization Token
2. The client starts the login flow redirecting the browser to the UI login page that is handle by the IAM
3. The user put his own credentials and the IAM proceeds to verifying them.
4. If the credentials are correct, user will be redirected to the client with an authorization code
5. Client contacts the IAM for get the access token, bringing in the request the authorization code it received
6. After the client receive the access token, it can proceed with the resource request
7. The resource server receive the request with the access token, verify it and send back the resource

OIDC 1.0

OpenID Connect 1.0. is an authentication layer that is built on top of the OAuth 2.0. While OAuth 2.0 is primarily focused on authorization, allowing applications to access resources on behalf of a user, OIDC extends this protocol to include

authentication This means OIDC provides a way to verify the user's identity in addition to managing permissions to access resources In addition to the Access Token, client receive the **ID Token**, a *JSON Web Token (JWT)* that contains information about the authenticated user as other data relevant to the process itself

KeyCloak

The IAM that I used during the development is **KeyCloak**, an open-source software that provide:

- **Single Sign-On and Single Sign-Out:** Users can access multiple applications with a single set of credentials.
- **OpenID Connect Support**
- **User Management:** An Admin Console allows centralized management of users, roles, role mappings, clients, and configurations.
- **Login Flows:** Optional features include user self-registration, password recovery, email verification, and password update requirements.
- **Session Management:** Both administrators and users can view and manage active user sessions.
- **Token Mappers:** Customize tokens and statements by mapping user attributes, roles, and other values as needed.
- **Service Provider Interfaces (SPI):** A range of SPIs are available for customizing server functionality, including authentication flows, user federation providers, protocol mappers, and more.

[6]

In keycloak page the Administrator is able to create new roles, assign them to users and create groups and assign users and roles to them as well. The administrator has a total and transparent control of what is happening and have the possibilities to control and manage all the users without that this may possible block the entire system.

3.2.5 Spring Security

We have seen how through *keycloak* we manage authentication and user's roles, but how do we integrate the flow of OIDC into our software and implement security controls? The de-facto standard for applications developed with Spring Boot is

the **Spring Security** framework, which implements security in a declarative way. This allows the system to be maintainable, to minimize code duplication and to separate the concerns about the security. How Spring Security do this? It use a set of **Servlet Filters** that take an incoming HTTP Request, and throw filters grant proper access only to those who are entailed to do so.

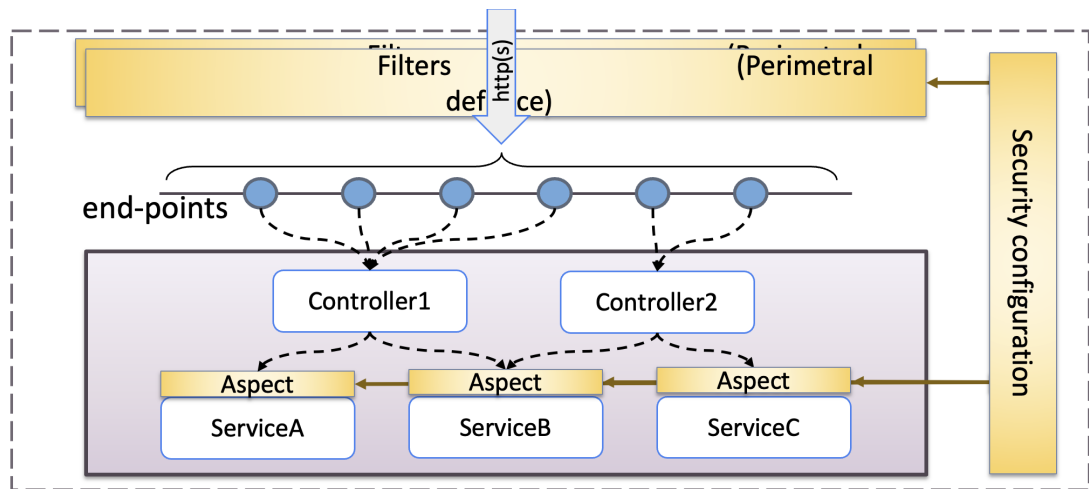


Figure 3.7: Servlet Filter

Spring Security use filter chains that intercepts incoming requests before they reach our application's controllers. This filter chain is a sequence of filters that each performs a specific function, such as: **Authentication**, **Authorization** and **Cross-Site Request Forgery (CSRF) protection**.

As our authentication is managed by an IAM, how Spring Security create this connection?

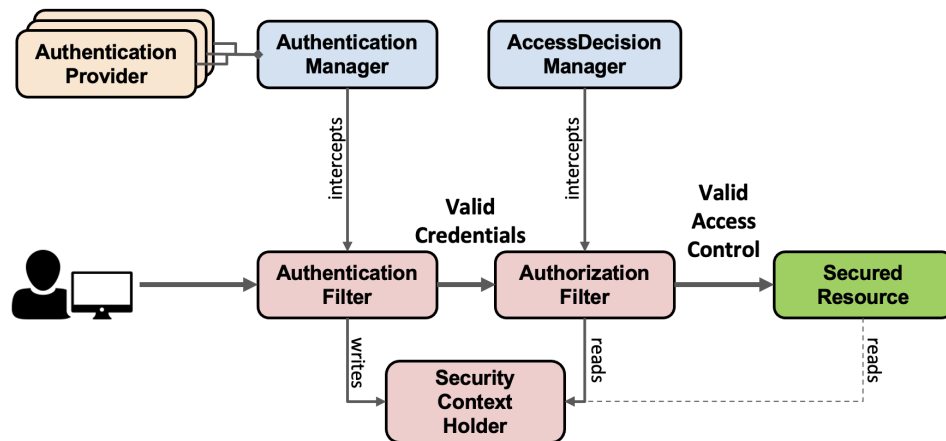


Figure 3.8: Authorization Flow

1. The **Authentication Filter** intercepts the incoming request and checks if the user is authenticated
2. Supposed the user it is not authenticated, the Authentication Filter will contact the **Authentication Manager** that collaboration with the **Authentication Provider** will validate the credentials.

Spring Security offer us different types of Providers for handle the authentication, and it is here that the connection with our IAM begin, using the **OidcAuthorizationCodeAuthenticationProvider** that is responsible for validating an authorization code and exchange it with Access and ID tokens

3. After that the user was authenticated, the **Security Context Holder** will stores the authentication details, ensuring that the user does not need to authenticate again for each request in the same session.
4. The **Authorization Filter** will determines if the user has the permission to access to the request source. **AccessDecision Manager** checks the roles or the granted authorities
5. Now if the filtering processes give a positive result, the requested service can be provided

Particular attention has to be taken with the OIDC authorization code flow. Our software has to be split in two main layers:

- **OAuth 2 Client** It is responsible for being exposed to the internet, receiving incoming HTTP requests, interacting with the IAM (Identity and Access

Management) system, and managing the entire JWT lifecycle. Additionally, it exposes a set of endpoints to handle tasks such as token issuance and validation

- **OAuth 2 Resource Server** It is responsible for validating the JWT (JSON Web Token) that the *OAuth 2 Client* automatically attaches to the request header. This token is forwarded by the client after the authentication process is completed, ensuring that the request is securely authorized before proceeding to access the protected resources.

The Logout process

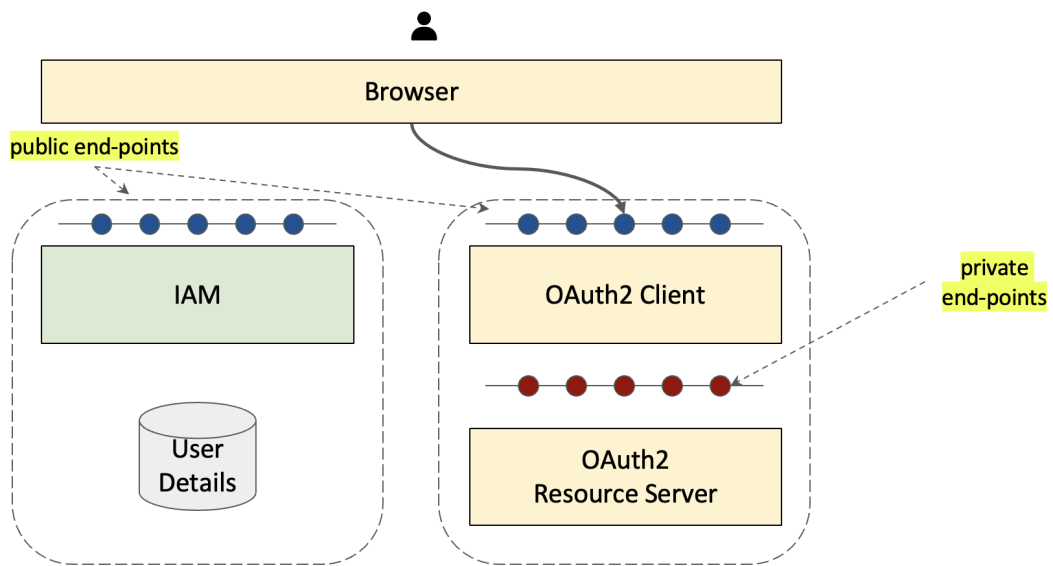


Figure 3.9: OIDC Code Flow

As we can see in this picture, this is the high-level architecture of software that uses OIDC. When a user is authenticated, a cookie containing the JWT will be sent with each request. In a custom authorization policy, it would be sufficient to invalidate the session related to the specific JWT in order to log the user out.

However, in this particular flow, the logout process must be handled differently. The IAM creates its own session with the browser when the login process begins. As a result, if the user makes a new request after invalidating the session only in the OAuth2 Client, they will be redirected to the IAM login page. The IAM, upon verifying that its own session with the browser is still valid, will issue a new authorization code, thereby bypassing the logout action performed at the application level.

To achieve an effective logout, our software needs to trigger the RP-initiated logout. When a logout action is requested, the application must also end the user session inside the IAM, ensuring that the user is properly redirected to the IAM login page, and the session is fully terminated.

SPA extentions

A single-page application (SPA) is a type of web application that loads a single HTML page and dynamically updates its content as the user interacts with it, without requiring a full page reload. While this architecture provides a smooth user experience, it introduces certain challenges when integrating with an external identity provider during the authorization process. Specifically, the usual page redirection mechanism does not work seamlessly with the SPA model, as the redirection will not automatically cause the browser to load the login page from the IAM RL.

To properly handle authentication and authorization in a SPA, we must design both the SPA and the OIDC client to manage this flow efficiently. This involves coordinating the login redirection and token management between the client and the SPA.

- The OIDC client must provide appropriate endpoints to the SPA. One of these endpoints should allow the SPA to determine whether the user's identity has already been established. Another endpoint should provide the necessary redirect URL that the SPA will use during the login process. This ensures that the SPA can properly initiate the login flow when required.
- The SPA must be designed to check the user's identity upon initialization. If the user is not authenticated, the SPA must unload itself and trigger a redirection to the login page provided by the IAM. This involves temporarily handing over control to the IAM's login system. Once the user completes the authentication process successfully, the SPA should be reloaded in the browser, at which point it can retrieve the authentication tokens and restore the user's session seamlessly.
- Additionally, the SPA should be able to handle scenarios where the user's session expires or when explicit logout actions are triggered. In such cases, the SPA should again use the provided endpoints to redirect the user back to the IAM for re-authentication or session termination, ensuring that security is maintained throughout the user's interactions with the application.

CSRF

Cross-Site Request Forgery¹ (CSRF) attacks occur when a malicious web site causes a user's web browser to perform an unwanted action on a trusted site. These attacks have been called the "sleeping giant" of web-based vulnerabilities, because many sites on the Internet fail to protect against them and because they have been largely ignored by the web development and security communities. CSRF attacks often exploit the authentication mechanisms of targeted sites. The root of the problem is that Web authentication normally assures a site that a request came from a certain user's browser; but it does not ensure that the user actually requested or authorized the request. [7]

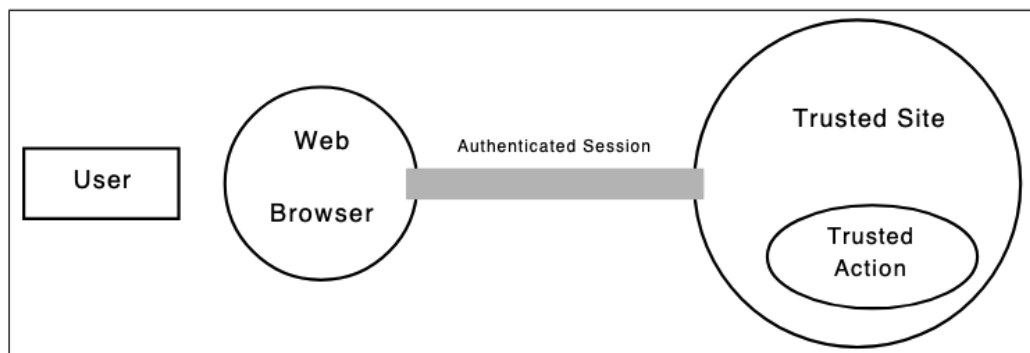


Figure 3.10: Here, the Web Browser has established an authenticated session with the Trusted Site. Trusted Action should only be performed when the Web Browser makes the request over the authenticated session. [7]

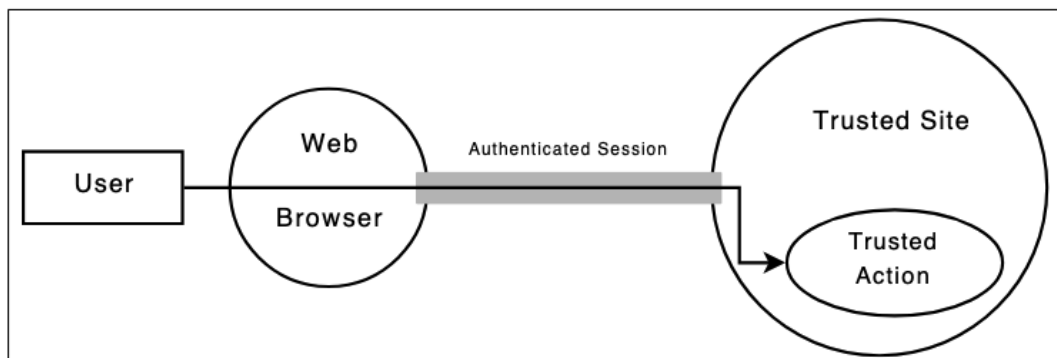


Figure 3.11: A valid request. The Web Browser attempts to perform a Trusted Action. The Trusted Site confirms that the Web Browser is authenticated and allows the action to be performed [7]

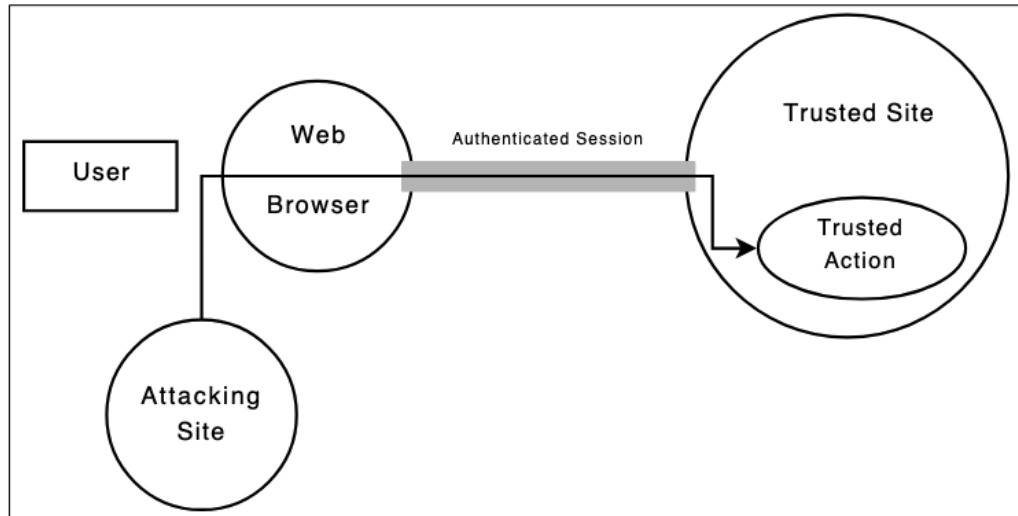


Figure 3.12: A CSRF attack occurs when a malicious site tricks the browser into sending a request to a trusted site. The trusted site perceives the request as valid and authenticated, since it originates from the user’s web browser, and proceeds to execute the requested action. CSRF attacks are feasible because websites authenticate the browser, not the user.[7]

We can prevent this type of attack by including an additional parameter in all non-GET requests. This parameter is generated by the server when the session is established and stored in a non-HTTP-only cookie. Whenever our SPA application performs a request, it will include this cookie in the request headers to prove that the request is originating from the application itself and not from a malicious site. This ensures that the server can verify the authenticity of the request and mitigate the risk of CSRF attacks.

3.2.6 Grafana, Loki, Prometheus

Monitoring the performance of our software is crucial for maintaining high levels of **Quality of Service (QoS)**. In a microservice architecture, where multiple services communicate with one another across different environments, the complexity of monitoring can increase significantly. Each service may have its own dependencies and performance metrics, making it difficult to get a clear view of the overall health of the system. However, there are robust tools available that can help streamline the monitoring process and ensure that potential issues are detected early before they impact the user experience. Some of the most widely-used tools for this purpose are **Prometheus**, **Loki**, and **Grafana**.

Prometheus

Prometheus is an open-source tool that collect in real-time metrics from our services, tracking performance and errors in a microservices environment. Prometheus works by analyzing metrics from endpoints that expose them, typically in a RESTful format, and then storing them in a time-series database. Its query language, **PromQL**, allows us to perform complex calculations on your collected metrics and set up custom alerts based on thresholds. Prometheus provides deep visibility into individual services and the interactions between them, enabling quick identification of performance bottlenecks or system failures.

Loki

Loki is a log aggregation system that works seamlessly with Prometheus but focuses on managing logs rather than metrics. Loki organizes logs based on labels such as service name , which aligns with the way Prometheus handles time-series metrics. By using Loki, we can efficiently gather, search, and analyze logs generated by our microservices. This becomes particularly valuable when diagnosing issues that may not be evident from metrics alone

Grafana

Grafana is an open-source platform for monitoring and observability that integrates with both Prometheus and Loki. It allows us to create visually rich, customizable dashboards that present metrics and logs in a user-friendly format. Grafana supports a wide variety of data sources, and its real-time visualizations make it easier to understand performance trends and spot anomalies.

3.3 FrontEnd Analysis

As required, the framework used to develop the front-end was **Angular**. Angular is an open-source front-end framework created and mantained by Google, which is nothing more than version 2 of AngularJs but because of their major differences they are referred to as two separate frameworks. Let us now analyze the structure of Angular, its features and its advantages and disadvantages

3.3.1 Angular

Typescript

Angular is built on Typescript, a superset of Javascript that expands its functionality.



Figure 3.13: Google Trends: Javascript vs Typescript

Although JavaScript remains more popular, TypeScript supports modern object-oriented programming (OOP), making the code cleaner and easier to manage. Angular is suitable for building both small and large applications, and TypeScript's IDE support, improved code maintainability, and easier error detection make it a far superior programming language for Angular compared to JavaScript.

Component-Based Architecture

Angular, like other frameworks as React, is built around a component-based architecture that means that each UI element is a reusable component, that encapsulate the HTML, CSS and the logic part. It makes the code more modular and easier to scale.

Two-Way Data Binding

Two-way data binding allows you to bind a value to an element and simultaneously enable that element to propagate changes back to the underlying data. This is one of the main differences between Angular and React, as Angular provides the ability to easily move data from a child component to a parent component through this binding mechanism, while React requires a more manual approach for doing the same

Directives

In Angular, directives are one of the core building blocks used to extend the functionality of HTML by attaching custom behavior to DOM elements. They allow us to create reusable code. There are three types of directives in Angular:

- **Component Directives** Components in Angular are a special type of directive that comes with an associated HTML template and styles. Unlike other directives, components define the view (UI) and behavior (logic) of a specific

portion of the application. Each component in Angular controls a part of the screen, known as the view, and includes its own selector, template, and data-binding logic.

- **Structural Directives** Structural directives are used to manipulate the DOM by adding or removing elements. These directives change the structure of the view.
- **Attribute Directives** Attribute directives are used to modify the behavior or appearance of existing elements in the DOM, without changing the structure. They work by changing the attributes of the host element.

Dependency Injection DI

Dependency Injection (DI) is a design pattern used to create and supply specific components of an application to other components that depend on them. In Angular, these dependencies are often services, though they can also be values like strings or functions. During the application's bootstrap process, an injector is automatically created to instantiate dependencies as required, using a configured provider for each service or value. [8]

Command Line Interface (CLI)

The Angular CLI (Command Line Interface) is a tool provided by Angular that allows us to create, build, test, and maintain Angular applications more efficiently by providing a set of commands for generating code, managing dependencies, and automating repetitive tasks.

3.3.2 Angular Material

Angular Material is a UI component library for Angular that follows the **Google's Material Design** guidelines. This library offers a set of reusable, well-tested, and accessible UI components designed to ensure a consistent and intuitive user experience. Angular Material components are built to integrate seamlessly with Angular functionalities, such as forms, and to support a wide range of use cases, making it easier to develop modern and responsive user interfaces.

Benefits of Angular Material

- **Consistent Design:** By adhering to Material Design guidelines, Angular Material ensures that the look and feel of components are consistent, thereby enhancing usability and familiarity for users.

- **Reusable Components:** The library includes components such as buttons, cards, dialogs, menus, and navigation bars that can be utilized across different parts of the application, reducing development time and improving code maintainability.
- **Integration with Angular:** Angular Material fully leverages Angular's features, such as data binding and directives, allowing us to build interactive applications without manually managing the DOM.
- **Form Support:** Input and form components are designed to work seamlessly with Angular's reactive forms, facilitating validation and state management of forms.

3.4 User Interface Design: Patterns, Heuristic Evaluation, and Prototyping with Figma

In the **Human-Computer Interaction** (HCI) it is very import to design intuitive and user-friendly interfaces for give the best user experiences possible. In this section I will explore three interconnected elements of interface design: **design patterns**, **heuristic evaluation**, and **High-level prototyping**.

3.4.1 Design Patterns

Design Patterns are proven and reusable solution that solve recurring problems that user can face during the interaction with the system, suggesting specific solution for specific problem. Even if every user interface is uniquer with its set of goals it is very important to follow specific patterns and do not force the users to learn new conventions. We can consider Design patterns as templates that provide guidance on how to structure the interface, ensuring consistency and predictability across different parts of the system.

In our Back Office, an example of a design pattern is the way users navigate between the different sections. The **Accordion Menu** is a perfect example of this—it's a vertical menu with different sections that can expand to reveal subsections.

3.4.2 Heuristic Evaluation

Evaluation test the usability, functionality and the acceptability of an interactive system. In more details:

- With **usability** we want to know how well the user can use the system's functionality

- **Functionality** want to measure how the system's functionality accord the user's requirements and enable them to perform their tasks.
- Testing **acceptability** means evaluating the enjoyment and emotional response to a system by the user

The different phases for performing an Heuristic Evaluation consist on

1. Define a set of tasks, that the evaluators will analyze going through the design or on a prototype
2. At each step, check the design according to each of the heuristics
3. Take notes about the comments of the evaluators
4. Each evaluator should provide a list of usability problems and which heuristic has been violated, and why. Taking care to specify where this possible problem it was found

Understand how to use a new software has a **learning curve** that can be more or less steep. A learning curve serves as a graphical illustration of how production efficiency evolves over time. The foundational idea behind this concept is that, typically, there is an initial phase during which the costs or investments involved exceed the benefits or returns generated. However, once this learning curve is navigated and overcome, the expectation is that the returns will begin to surpass the initial investments. This phenomenon can be attributed to the accumulation of experience and knowledge, which allows for improved processes, reduced errors, and increased productivity. Over time, as individuals or organizations become more familiar with the tasks at hand, they tend to streamline their efforts, leading to more efficient operations and ultimately higher profitability. Thus, the learning curve not only highlights the relationship between time and productivity but also emphasizes the importance of experience in driving long-term success.

The development of a new Back Office would imply a certain slope in this curve, but if instead we take into account the fact that there is already a Back Office that has been used by the company for many years and the end-users are clear about the paths they need to take to complete their tasks this would raise the ratio of Time to Learn. Users who are familiar with older software versions often experience resistance when adapting to new systems. They may rely heavily on existing habits, which can make the learning curve steeper for them than for new users who come without preset expectations or workflows. This effect, called "unlearning," requires targeted training that addresses the specific pain points of experienced users to help them replace old habits with new ones. [9]

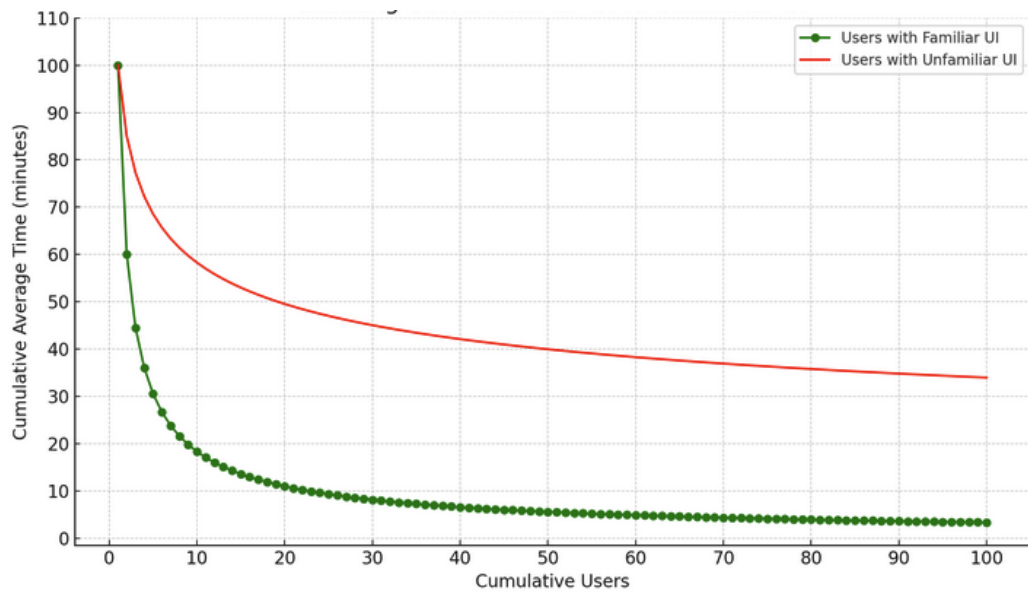


Figure 3.14: Learning Curve: Familiar UI vs Unfamiliar UI

I decided then to take the role of the evaluator and perform the heuristic evaluation of the software they currently have. From the results I got I went to create what is a UI that complies as much as possible with the evaluation heuristics and modifying the parts that currently do not so as to improve the user experience, with a design that is more modern but keeps the same overall structure so as to cut down the learning curve, lower the training time and increase productivity .

The heuristics I analyzed during the software analysis were the **10 Nielsen's Usability Heuristics**:

1. **Visibility of system status**
2. **Match between system and the real world**
3. **User control and freedom**
4. **Consistency and standards**
5. **Error prevention**
6. **Recognition rather than recall**
7. **Flexibility and efficiency of use**
8. **Aesthetic and minimalist design**
9. **Help users recognize, diagnose, and recover from errors**
10. **Help and documentation**

Chapter 4

System Implementation

4.1 Introduction

In this chapter, we will take a closer look at the work done, focusing on the different parts of the implementation. We'll examine how the design choices were translated into concrete code and configurations, ensuring that each feature aligns with the initial project requirements. This analysis will explore not only the surface-level functionalities but also the underlying mechanisms that support each request made to the system.

One key aspect we'll investigate is how each component works together to validate incoming requests. For each operation, the system must verify that all parameters are correct, relevant, and within defined constraints. We'll delve into the logic implemented to handle these checks, highlighting how different types of data are processed, validated, and either accepted or flagged for errors based on preset conditions.

4.2 Project Setup

The first step addressed once we defined the design and schema of our project was to create the entire software setup, linking the various Microservices together and building a clear and robust infrastructure.

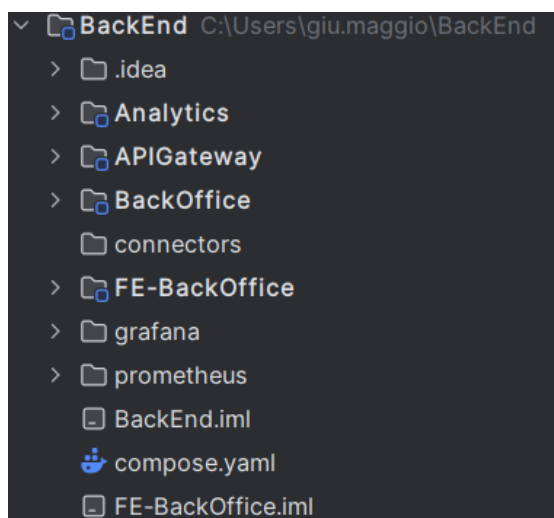


Figure 4.1: Setup of the modules

Without going into detail about each folder at the moment, the general structure consists of a package within which we find three Springboot project modules: the **Apigateway**, **Analytics**, and **BackOffice**. And a module containing the FrontEnd Angular project, **Fe-BackOffice**.

4.2.1 Maven and Dependencies

All three Springboot microservices use the **Maven** tool, which simplifies the process of building, managing and deploying our software by:

- **Managing project's dependencies**, going to include external libraries in our project, downloading them and appropriately configuring them. All the dependencies that our project will need will be shown in a single file, pom.xml (Project Object Model)
- **Convention-over-configuration** approach, allowing developers to not define every configuration of the project since it will be handled by the tool. It in fact when the project is created, builds a **default structure** making it easier to maintain and giving the developers only the task of going to work with the files placed accordingly with the it
- **LifeCycle Management**, Maven has a defined build lifecycle that includes different phases. *Compilation*, *testing*, *installation* and *deploying* are some of these, allowing to automate common tasks and accelerating the workflow.

4.2.2 Docker and Services

The entire project run within a Docker container managed by **Docker Compose**. This set up allows the orchestration of all the different services essential for the application's operation in a cohesive and interconnected environment where each service performs a distinct role.

I used the **Docker Compose Support** for Spring Boot that enables a seamless integration between a Spring Boot application and the services we defined in a *docker-compose.yml* file. When we run our application, it automatically detect the services we defined and automatically start them without any additional setup, making it easier to manage dependencies within a containerized environment.

The services that I included in the *docker-compose.yml* file are:

- **keycloak service**: is one of the different methods we have for run the Keycloak IAM services
- **mongodb service**: execute an instance of MongoDB inside a container
- **kafka service**: allows kafka, a message broker, to run inside a Docker container. This add in our application to exchange and process flows of messages and connect the different components we have.
- **kafka-ui service**: it is an UI interface for Kafka where we can find the *topics*, *producers*, *consumers* and *messages* in our Kafka broker.
- **kafka-connect service**: it connects different data resources, in our case the OracleDB and MongoDB, making them able of exchanging information through *connectors*
- **prometheus service**: makes possible to quickly deploy a Prometheus instance to collect metrics from our containers
- **loki service**: run a Loki instance for collection the logs of our services
- **grafana service**: set up a monitoring and visualization environment, providing a detailed overview of system health, performance, and other operational parameters

I will provide a more detailed overview of each service throughout this chapter.

4.2.3 BackEnd structure

The architecture of each microservice we will explore follows a service layer pattern. The main classes that make up the services can be categorized into the following components:

- **DTOs (Data Transfer Objects):** DTOs are objects responsible for transferring the necessary data and information across services and sending responses through the network in API endpoints. They are designed to encapsulate the data that needs to be transferred between layers or across different microservices.
- **Controller:** Controllers are classes responsible for receiving and handling HTTP requests. They serve as the entry points for each microservice, routing the incoming requests to the appropriate service layer for the desired response. Controllers often contain basic logic for managing the flow of data but delegate complex business logic to the service layer.
- **Service and ServiceImplementation:** Service classes encapsulate the business logic of the application. They manage the core functionality of the service, interacting with repositories and entities to execute the required actions. The service layer is where the main business rules and workflows are processed.
- **Entity:** Entities represent the data model of the application, mapping the database tables into Java objects. These objects validate the columns and relationships defined in the database schema and ensure that the data is structured correctly in the application.
- **Repository:** Repositories are classes responsible for data persistence and retrieval. They interact with the database to perform CRUD (Create, Read, Update, Delete) operations on entities. By using JPA (Java Persistence API), repositories provide an abstraction layer that improves flexibility and performance when interacting with the database.

4.2.4 FrontEnd structure

Apart from considering Angular entry points like *app.component.ts*, the FrontEnd has been structured to ensure the code is readable and easy to manage. The main sections are as follows:

- **Component:** This folder is dedicated to the UI components, with each specific page or feature having its own subfolder. Each subfolder contains the associated HTML, TypeScript, and CSS files, along with the logic responsible for the page's behavior and interactions. These components are reusable and are designed to encapsulate the UI elements specific to different areas of the application.
- **Enviroments:** This folder stores TypeScript files that define the application's environment-specific configurations.

- **Interfaces:** The interfaces folder contains various TypeScript interfaces that define the shape of the data used throughout the application. The interfaces are typically used for objects exchanged between the FrontEnd and the Backend and are connected with the DTOs structure.

- **Service:** This folder contains services that handle the communication between the FrontEnd and the Backend, centralizing the logic for making HTTP requests and business logic. Isolating the HTTP logic in services, the application becomes more maintainable, as components only deal with the UI and delegate data management to the services.

4.3 Security Services Implementation

4.3.1 KeyCloak Setup

Let us go on to analyze the implementation of the chosen IAM, **keycloak**. The first concept and set up that needs to be done is the **realm**. The realm is the primary unit of keycloak used to manage and isolate users and their roles by configuring them all in one instance. Although not initially defined in the functional requirements, seeing the existing software in operation I noticed that there were multiple logically separate groups of users and this difference between them was handled entirely by code. The keycloak realms instead will allow the possibility of separating these groups entirely, thus defining the roles for each of them separately. This creates an isolation that is guarantees security,enhances code readability and simplify the implementation and management. After logging in with the admin credentials, defined in the docker-compose.yaml at the keycloak service definition, a realm can be created simply by entering the name. The main screen we are presented with is as follows

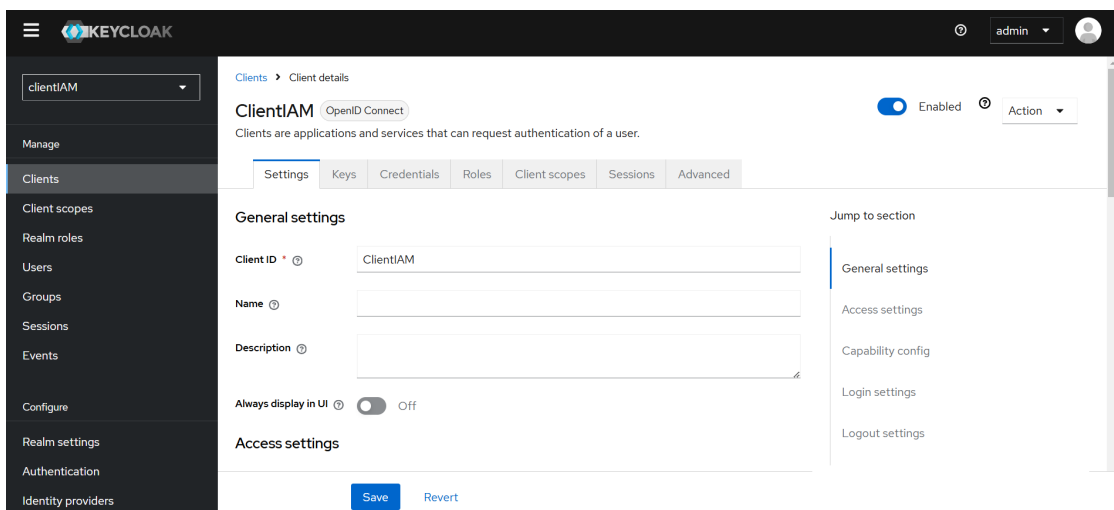


Figure 4.2: Realm main page

The created realm is **ClientIAM**. Before I started with the definition of users and roles, I had to define in the realm the urls to refer to:

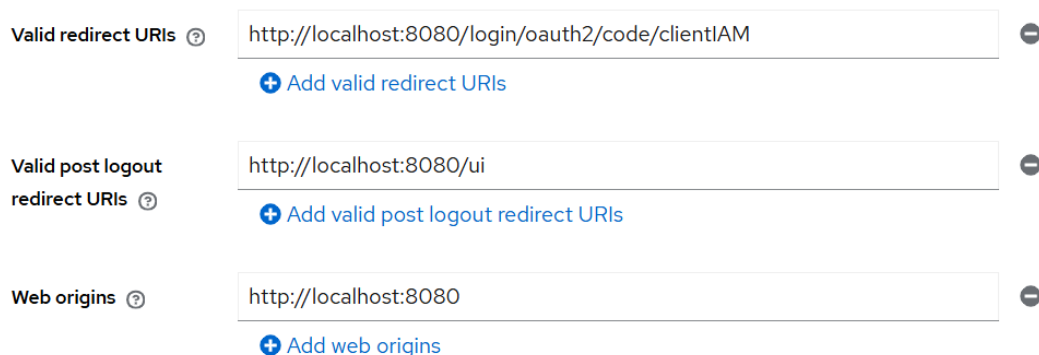


Figure 4.3: Realm's urls

respectively I defined:

1. The url to redirect to after logging in correctly.
2. The url to redirect to after logging out correctly.
3. The CORS orgin

To show the potential and simplicity of how you can manage users and their roles I created three different roles: **Admin**, **simpleConsultant** and **ManagerLevel1** and three different users to assign them to respectively

System Implementation

The screenshot shows the 'ClientIAM' management interface. At the top, it indicates 'OpenID Connect' is 'Enabled' with a toggle switch and an 'Action' dropdown. Below this, a navigation bar includes tabs for 'Settings', 'Keys', 'Credentials', 'Roles' (which is selected), 'Client scopes', 'Sessions', and 'Advanced'. A search bar for roles is present, along with 'Create role' and 'Refresh' buttons. The main area displays a table of roles:

Role name	Composite	Description
Admin	False	-
managerLevel1	False	-
simpleConsultant	False	-

At the bottom right, there is a pagination control showing '1-3' and navigation arrows.

Figure 4.4: Realm's roles

The screenshot shows the 'Users' management interface. It states 'Users are the users in the current realm.' and includes a 'Learn more' link. A 'User list' tab is active. The interface features a search bar, 'Add user', 'Delete user', and 'Refresh' buttons. The user list table is as follows:

Username	Email	Last name	First name
consultant1	consultant1@gmail.com	-	consultant1
giuseppe	giu.maggio@reply.it	Maggio	Giuseppe
manager1	manager1@gmail.com	-	manager1

Navigation controls at the bottom right show '1-3' and arrows.

Figure 4.5: Realm's users

The screenshot shows the 'Role mapping' interface for the user 'consultant1'. It indicates the user is 'Enabled' and has an 'Action' dropdown. The 'Role mapping' tab is selected, with other tabs for 'Details', 'Credentials', 'Groups', 'Consents', 'Identity provider links', and 'Sessions'. The interface includes a search bar, a checked 'Hide inherited roles' checkbox, and 'Assign role', 'Unassign', and 'Refresh' buttons. The role mapping table is:

Name	Inherited	Description
default-roles-clientiam	False	\${role_default-roles}
ClientIAM simpleConsultant	False	-

Navigation controls at the bottom right show '1-2' and arrows.

Figure 4.6: Example of assigned role to an user

Very important was for the company that users can have multiple roles at the

same time and as we can see it is possible, at the moment the consultant user has two roles, the one created by us and the one that is assigned by default by KeyCloak when creating a user.

4.4 Spring Security SetUp

After creating the realm with the users associated with their roles, we need to connect this service with the **ApiGateway** and the various microservices. The **SpringSecurity** tool makes this process much easier and faster. The first step taken was to identify our ApiGateway as an oauth2 client provider. It is in fact responsible for communicating with KeyCloak, receiving tokens after logging in, and going to verify request cookies by validating them. If each step of this process is successful then the request will be routed into the required microservice which will take the role of oauth2 resource server. By defining the parameters in *application.yml*, the file responsible for handling the configuration of the Spring project, I connected the gateway service with the keycloak service and with the realm created.

```

1 spring
2   security:
3     oauth2:
4       client:
5         provider:
6           keycloak:
7             issuer-uri: http://localhost:9090/realms/clientIAM
8       registration:
9         kcClient:
10          provider: keycloak
11          client-id: ClientIAM
12          client-secret: *****
13          scope:
14            - openid
15            - roles
16            - offline_access
17          authorization-grant-type: authorization_code
18          redirect-uri: http://localhost:8080/login/oauth2/
           code/clientIAM

```

Next, I defined the configuration classes (annotated with **@Configuration** and **@EnableMethodSecurity**) for the purpose of defining logout behaviors and defining Spring Security's SecurityFilterChain.

```

1 @Configuration
2 @EnableMethodSecurity(prePostEnabled = true, securedEnabled = true
  )

```



```

3 public class SecurityConfig{
4     @Bean
5     public SecurityFilterChain securityFilterChain(HttpSecurity
6         httpSecurity) throws Exception {
7         httpSecurity
8             .authorizeHttpRequests(authorize -> authorize
9                 .requestMatchers("/", "/login", "/logout",
10                    "/me", "/ui/**").permitAll()
11                    .anyRequest().permitAll()
12                )
13                .oauth2Login(Customizer.withDefaults())
14                .logout(logout -> logout.logoutSuccessHandler(
15                    logoutSuccessHandler()))
16                .csrf(csrf -> csrf
17                    .csrfTokenRepository(
18                        CookieCsrfTokenRepository.withHttpOnlyFalse())
19                    .csrfTokenRequestHandler(new
20                        SpaCsrfTokenRequestMatcher())
21                )
22                .addFilterAfter(new CsrfCookieFilter(),
23                    BasicAuthenticationFilter.class);
24         return httpSecurity.build();
25     }
26 }

```

As we can see in lines seven to ten, our filter chain allows access without authorization verification for a series of URLs. Specifically, we permit access without a token during login (“/login”) and logout (“/logout”), for any URL related to a Front End page (“/ui/**”, “/”), as specific resource requests will then be restricted. Additionally, there’s another route (“/me”) located in the *APIGateway* microservice, that is essential to maintain the best practices of the OIDC flow. This endpoint is called by the FrontEnd to handle and verify authentication and permissions, enabling access to specific sections as required.

```

1 @GetMapping("/me")
2     public Map<String, Object> me (
3         @CookieValue(name="XSRF-TOKEN", required=false)
4         Optional<String> xsrf
5     ){
6         Authentication authentication = SecurityContextHolder.
7             getContext().getAuthentication();
8         OidcUser principal = authentication != null &&
9             authentication.getPrincipal() instanceof OidcUser ? (OidcUser)
10             authentication.getPrincipal() : null;
11         String name = principal != null ? principal.getName() : ""
12         ;
13         Map<String, Object> me = new HashMap<>();

```

```
10     me.put("loginUrl", "/oauth2/authorization/kcClient");
11     me.put("logoutUrl", "/logout");
12     me.put("xsrftoken", xsrf);
13     me.put("principal", principal);
14     return me;
15 }
```

The **principal** is the Spring Security object that encapsulates all user information, which is extracted from the JWT during authentication. The `/me` endpoint, particularly as shown from lines 10 to 15, returns all necessary information for the FrontEnd. Specifically:

- **loginUrl**: The URL to which we must redirect, exiting the Single Page Application to perform login through the Keycloak UI.
- **logoutUrl**: The URL to initiate logout, which will log the user out not only from the BackOffice software but also from the IAM. The logout process, including endpoint creation, is fully managed by Spring Security.
- **xsrftoken**: A token used to protect the software from Cross-Site Request Forgery (CSRF) attacks.
- **principal**: An object containing all user information that Keycloak forwards, such as email, first and last name, and role, for example.

```
1 private OidcClientInitiatedLogoutSuccessHandler
   logoutSuccessHandler(){
2     OidcClientInitiatedLogoutSuccessHandler successHandler =
   new OidcClientInitiatedLogoutSuccessHandler(crr);
3     successHandler.setPostLogoutRedirectUri("http://localhost
   :8080/ui");
4     return successHandler;
5 }
```

The following function manages the logout process. Specifically, it sets the redirect URI to navigate to upon successful logout. Now, let's analyze how Spring Security is configured across the various microservices and how the filter chain operates at that level to check and manage the **JWT (JSON Web Token)**.

```
1 spring.security.oauth2.resourceserver.jwt.issuer-uri=http://
   localhost:9090/realms/clientIAM
```

In the application properties, I defined the URI of the authorization server that issues the JWTs. Spring Security will use this URL to retrieve the authorization server's metadata and will be able to validate the token's issuer and its signature.

```
1 @Configuration
2 @EnableMethodSecurity(prePostEnabled = true, securedEnabled = true
3 )
4 public class SecurityConfig {
5     @Bean
6     protected SessionAuthenticationStrategy
7     sessionAuthenticationStrategy() {
8         return new RegisterSessionAuthenticationStrategy(new
9         SessionRegistryImpl());
10    }
11    @Bean
12    public SecurityFilterChain filterChain(HttpSecurity
13    httpSecurity) throws Exception {
14        httpSecurity
15            .authorizeHttpRequests(authorize -> {
16                authorize.anyRequest().authenticated();
17            })
18            .oauth2ResourceServer(oauth ->
19                oauth.jwt( jwt->
20                    jwtAuthenticationConverter(
21                        jwtAuthenticationConverterForKeycloak())
22                )
23            .sessionManagement(sessionManagement ->
24                sessionManagement.sessionCreationPolicy(
25                    SessionCreationPolicy.STATELESS)
26            )
27            .csrf(csrf -> csrf.disable())
28            .cors(cors -> cors.disable());
29        return httpSecurity.build();
30    }
31    @Bean
32    public JwtAuthenticationConverter
33    jwtAuthenticationConverterForKeycloak() {
34        Converter<Jwt, Collection<GrantedAuthority>>
35        jwtGrantedAuthoritiesConverter = jwt -> {
36            Map<String, Object> resourceAccess = jwt.getClaim("
37            resource_access");
38            Object client = resourceAccess.get("ClientIAM");
39        }
40    }
41 }
```

```

36     LinkedTreeMap<String, List<String>> clientRoleMap = (
LinkedTreeMap<String, List<String>>) client;
37     List<String> clientRoles = new ArrayList<>(
clientRoleMap.get("roles"));
38
39     return clientRoles.stream()
40         .map(role -> new SimpleGrantedAuthority("ROLE_"
" + role))
41         .collect(Collectors.toList());
42     };
43
44     JwtAuthenticationConverter jwtAuthenticationConverter =
new JwtAuthenticationConverter();
45     jwtAuthenticationConverter
setJwtGrantedAuthoritiesConverter(
jwtGrantedAuthoritiesConverter);
46
47     return jwtAuthenticationConverter;
48 }
49 }

```

In this configuration class, which is common across all microservices, we can see how the JWTs are handled. Specifically, the function `jwt.jwtAuthenticationConverter(jwtAuthenticationConverterForKeycloak())` in line 19 communicates with the Keycloak provider to validate the JWT. If the conversion is successful, a **STATELESS** session is set, which is standard for RESTful APIs, ensuring that each request is processed independently without maintaining any user state. For the same reason, CORS and CSRF protection are disabled.

In line 30, we define the function `jwtAuthenticationConverterForKeycloak()` and its logic. Specifically, I have designed the function to access the Keycloak realm I created, ClientIAM, and retrieve the roles of the users, modifying the structure to make them compatible with Spring Security's role-based annotations, such as **@PreAuthorize**.

4.4.1 Angular Security Implementation

Security in the FrontEnd is managed using **Auth Guards**, a technique that allows us to protect different routes with great flexibility and ease, enabling us to handle various scenarios effectively. Auth Guards can be divided into different types; in our case, we use the most common one, **canActivate**, which determines access to each route where it is defined, following the logic we set up in `auth.guard.ts` **auth.guards.ts**:

```

1 export const canActivateGuard: CanActivateFn = (

```

```

2   route : ActivatedRouteSnapshot,
3   state: RouterStateSnapshot) => {
4     if( inject(AuthService).isLoggedInCheck()){
5       return true
6     }
7     else {
8       inject(Router).navigate(['/login'])
9       return false;
10    }
11  }
12 ];

```

The actual authentication check is delegated to **auth.service.ts**, which is injected into **auth.guard.ts**. This setup allows the guard to return true if the authentication is successful or to redirect to the login page if the check fails. **auth.service.ts**:

```

1  @Injectable({
2    providedIn: 'root'
3  })
4  export class AuthService {
5    private http = inject(HttpClient);
6    public me: MeInterface | null = null;
7
8    constructor(private cookieService: CookieService, private route:
9      Router) {
10     this.getMe().subscribe((value) => this.me = value)
11   };
12
13   getMe():Observable<MeInterface | null> {
14     let url = 'http://localhost:8080/me';
15
16     return this.http.get<MeInterface | null>(url).pipe(
17       tap((value) => {
18         this.me = value;
19         if(this.me?.principal === null){
20           this.route.navigate(["login"])
21         }
22       }),
23       catchError(this.handleError)
24     );
25
26   getXSRFToken() {
27     return this.cookieService.get('XSRF-TOKEN');
28   }
29   private handleError(error: HttpResponse): Observable<never>
30   {
31     ...
32   }

```

```
31 |
32 |     isLoggedInCheck(): boolean{
33 |         return this.me?.principal !== null
34 |     }
35 |
36 |     handleLogin(): any {
37 |         window.location.href = this.me!.loginUrl;
38 |     }
39 |
40 | }
```

In the `auth.service.ts`, each function is responsible for handling access and authorization. Specifically, a call is made to the `http://localhost:8080/me` endpoint, which, as discussed, retrieves user information that is then stored in a dedicated interface, *MeInterface*. Based on the presence or absence of the principal within the returned object, we can determine whether the user is correctly logged in.

Additional functions offered by this service include retrieving the `XSRFToken` (via `getXSRFToken()`), which is necessary for making non-GET requests, and the `handleLogin()` function, which enforces redirection to the IAM login page during the login process.

4.4.2 Spring Cloud Gateway

Having covered the main aspects of the security of our system, we shall dive into the principal concept of the **ApiGateway** service: **Routing**. To implement this feature, I utilized the **Spring Cloud Gateway**, which is a powerful and flexible API gateway built on top of the Spring Framework.

```
1 | spring:
2 |   application:
3 |     name: ClientIAM
4 |   cloud:
5 |     gateway:
6 |       mvc:
7 |         http-client:
8 |           type: autodetect
9 |         routes:
10 |          - id: BO-Server
11 |            uri: http://localhost:8081
12 |            predicates:
13 |              - Path=\backOffice\**
14 |            filters:
15 |              - StripPrefix=1
16 |              - TokenRelay
17 |          - id: Analytics-Server
```

```

18     uri: http://localhost:8082
19     predicates:
20       - Path=\analytics\**
21     filters:
22       - StripPrefix=1
23       - TokenRelay
24   - id: ui
25     uri: http://localhost:4200
26     predicates:
27       - Path=\ui\**
28   - id: home
29     uri: http://localhost:8080
30     predicates:
31       - Path=/
32     filters:
33       - RedirectTo=301, http://localhost:8080/ui

```

As shown above, the implementation is simple and straightforward. In the code shown, I have defined the paths for the various microservices: *BackOffice*, *Analytics*, *FrontEnd (UI)*, and the *ApiGateway* itself. Each URL will have a specific predicate that identifies where the request should be forwarded. Additionally, through the filter property, we remove this identifier to make the URL compatible with what the microservice expects.

Furthermore, it can be seen that Spring Cloud Gateway integrates seamlessly with Spring Security. By adding the **TokenRelay** filter, we enable the system to forward the access token along with the request, ensuring secure and authenticated communication between the services.

4.5 Sales Points Implementation

In this section, we will examine in detail the Sales Point component, specifically focusing on how they are retrieved using pagination, ordering and optional filtering, as well as how the details of a Sales Point can be accessed and modified. Adding new Sales Points is not automatic but must occur through a Batch process, which we will explore later in the corresponding section.

4.5.1 Retrieving List Implementation

```

1 @GetMapping("/salesPoints")
2   @ResponseStatus(HttpStatus.OK)
3   public Page<SalePointDTO> getSalesPoints
4   (

```

```

5     @RequestParam(required = false) Integer nPage, Integer maxItem
6     ,
7     @RequestParam(required = false)
8     String name, String siteId, String country,
9     String region, String provincia, String city, String street,
10    String sortBy, String order
11  ){
12    Map <String, String> filters = new HashMap<>();
13    Map <String, String> sorting = new HashMap<>();
14
15    if(name != null) {
16      filters.put("name", name);
17    }
18    ...
19    return t_siteService.getSalesPoints(nPage!=null ? nPage : 0,
20    maxItem!=null ? maxItem : 20, filters, sorting);
  }

```

The endpoint is a GET request to the URL */salesPoints*, where various parameters for filtering, sorting, and paging can be added optionally. The request, along with any received parameters, is then forwarded to the core service implementation for processing.

```

1  @Override
2  public Page<SalePointDTO> getSalesPoints(
3      int nPage,
4      int maxItem,
5      Map<String, String> filters,
6      Map<String, String> sorting) {
7
8      Sort sort = Sort.unsorted();
9      if(!sorting.isEmpty()) {
10         if(sorting.get("order")!= null && sorting.get("order").
11         equals("1")) {
12             sort = sort.by(sorting.get("sortBy")).descending();
13         }
14         else {
15             sort = sort.by(sorting.get("sortBy")).ascending();
16         }
17     }
18
19     Pageable page = PageRequest.of(nPage, maxItem, sort.isEmpty()?
20     sort.unsorted() : sort);
21     Page<T_SITE> sitePoints = t_siteRepository.findAllWithFilters(
22         filters.containsKey("name") ? filters.get("name") : null,
23         ...
24         ...

```



```

23     ,
24     page);
25     List<SalePointDTO> listSalePointResponseDTO = sitePoints.
getContent().stream()
26         .map(sitePoint -> sitePoint.toDTO())
27         .collect(Collectors.toList());
28     return new PageImpl<>(listSalePointResponseDTO, page,
sitePoints.getTotalElements());
29
30 }

```

The DTO exchanged is wrapped within a *Java Page object*, which automatically provides valuable pagination information, such as the total item count, current page, and total number of pages. After configuring the various filters, the repository connected to the **SalesPoint** entity is called. This repository class manages database interactions, executing queries and handling data retrieval or modifications as required.

```

1  @Query("SELECT t FROM T_SITE t WHERE "
2      + "(:name IS NULL OR t.NAME LIKE %:name%) AND "
3      + "(:siteId IS NULL OR t.SITEID = :siteId) AND "
4      ...
5      public Page<T_SITE> findAllWithFilters(
6          @Param("name") String name,
7          @Param("siteId") String siteId,
8          ...
9          Pageable page
10         );
11 }

```

The **@Query annotation** allows for executing a custom query instead of relying solely on the default queries provided by *JpaRepository*.

4.5.2 Retrieving Details Implementation

The endpoint for retrieving the details of a specific sales point, in line with what has been discussed earlier, is `/salesPoints/saleId`. This endpoint allows you to retrieve detailed information for a specific sales point identified by its unique `saleId`, passed as a path variable. In this case, the full potential of JPA is utilized, as calling the function with the predefined format, *findById(saleId)*, allows JPA to automatically derive the query and communicate with the database accordingly.

4.5.3 Updating Implementation

Using the PUT method and calling `/salesPoints/saleId`, the user can modify a specific sales point. In this case, we can observe the simplicity of how this request is managed and secured using the `@PreAuthorize('hasRole('Admin'))` annotation, which ensures that the request is rejected if the user making the request does not have the specified role.

To modify or insert a new record in the database using JPA, the procedure used involves creating an object of the required entity, in this case, **T SITE**, with the updated data provided by the user and passed through the DTO in the service. The `saveAndFlush` function of the repository allows us to quickly save or update the object in the database. It searches for the corresponding record in the table based on the ID provided, determining whether the operation is an insertion or an update.

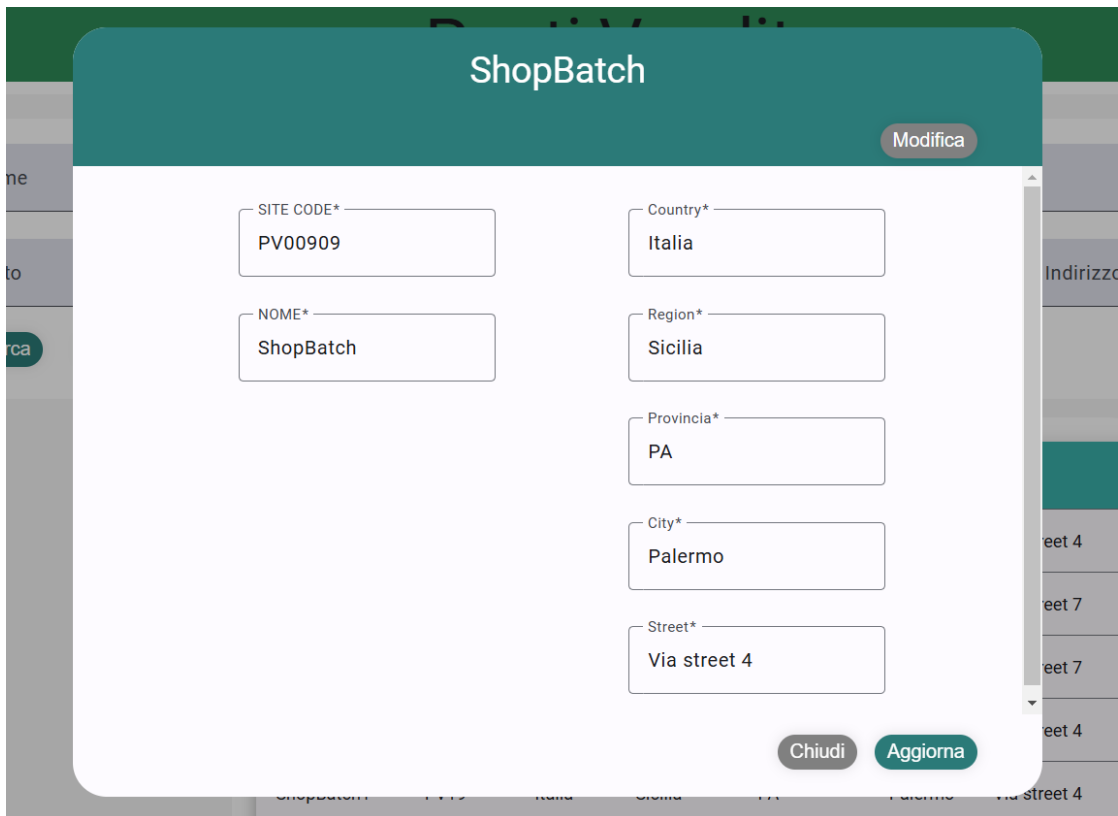
4.5.4 FrontEnd visualization

Let's now take a look at the UI and user experience.

Nome	Site ID	Stato	Regione	Provincia	Città	Via
ShopBatch	PV00909	Italia	Sicilia	PA	Palermo	Via street 4
ShopBatch2	PV022	Italia	Piemonte	TO	Torino	Via street 7
ShopBatch3	PV0292	Italia	Piemonte	TO	Torino	Via street 7
ShopBatch1	PV01	Italia	Sicilia	PA	Palermo	Via street 4
ShopBatch1	PV19	Italia	Sicilia	PA	Palermo	Via street 4

Figure 4.7: Sales Points Page

This is the main page for sales points, consisting of a filtering section at the top, a table on the lower left displaying the sales points, and on the right, a section for analytics based on the overall sales point data, not limited to the displayed subset, as we are working with a paginated table. The user can select or remove filters and perform searches as needed. In a familiar manner, clicking on a column header sorts the data by that field in ascending or descending order.



The image shows a modal window titled "ShopBatch" with a "Modifica" button in the top right corner. The form contains several input fields with the following values:

Field	Value
SITE CODE*	PV00909
Country*	Italia
NOME*	ShopBatch
Region*	Sicilia
Provincia*	PA
City*	Palermo
Street*	Via street 4

At the bottom right of the modal, there are two buttons: "Chiudi" and "Aggiorna".

Figure 4.8: Sales Point's detail

When the user clicks on a specific row, a modal opens to display the details of that sales point. If the user has the necessary permissions, they can click the "Modifica" button, as shown in the image, to update the Sales Point information.

4.6 Terminal Implementation

Let us now examine the terminal section. On this page, users can view the terminals associated with their sales points, along with their status and related information. Users can apply filters to precisely display the data they need, such as viewing all inactive terminals or those belonging to a specific sales point. Additionally, it is possible to create a new terminal by associating it with a sales point. As per the specifications, any newly created terminal will be set to inactive by default.

4.6.1 Retrieve List Implementation

The endpoint for retrieving all terminals is a GET request to the URL `/terminal`. Optional parameters are expected to manage filtering, pagination, and sorting. For pagination, if no specific parameters are provided, default values are applied. The request is then forwarded to the *TerminalService* for processing. The service mirrors the one provided for retrieving sales points, with differences in the filtering criteria and the use of a distinct Entity and repository. The two entities, T SITE and T TERMINAL, are connected through a **@OneToMany** relationship, as each sales point can have multiple terminals associated with it.

Unlike JDBC, JPA enables **lazy loading** of related table data. This means that while the retrieval of sales point data linked to terminals is prepared, it is not executed unless explicitly requested. This approach improves performance by reducing unnecessary data fetching.

4.6.2 Adding Implementation

Calling the same URL, `/terminal`, but using the POST method, allows for the addition of a new terminal to the system.

```

1      @Override
2      public TerminalDTO addTerminal(AddTerminalDTO terminal) {
3          Optional<T_SITE> site = siteRepository.findBySITEID(
terminal.siteCode);
4          if(site.isEmpty()){
5              throw new SiteIdNotFoundException(terminal.getSiteCode
());
6          }
7          T_TERMINAL terminalEntity = new T_TERMINAL();
8          terminalEntity.setModalita(terminal.modalita);
9          terminalEntity.setStato("INATTIVO");
10         terminalEntity.setModello(terminal.modello);
11         terminalEntity.setTERMINALID(terminal.terminalId);
12         terminalEntity.setSITECODE(site.get());
13         terminalEntity.setNumeroCassa(terminal.numeroCassa);
14
15         T_TERMINAL res = terminalRepository.saveAndFlush(
terminalEntity);
16         return res.toDto();
17     }

```

During the process of adding a new terminal, the system first performs a lookup in the sales points database to verify that the sales point to which the terminal is being assigned exists and is valid. If the sales point is not found or is deemed

invalid, a custom exception is thrown, and an appropriate error message is displayed on the front end.

In this scenario, the operation was not implemented using JDBC since it is a lightweight and quick operation, especially when compared to more data-intensive tasks such as retrieving thousands of transaction records.

4.6.3 FrontEnd visualization

Let's now take a look at the UI and user experience.

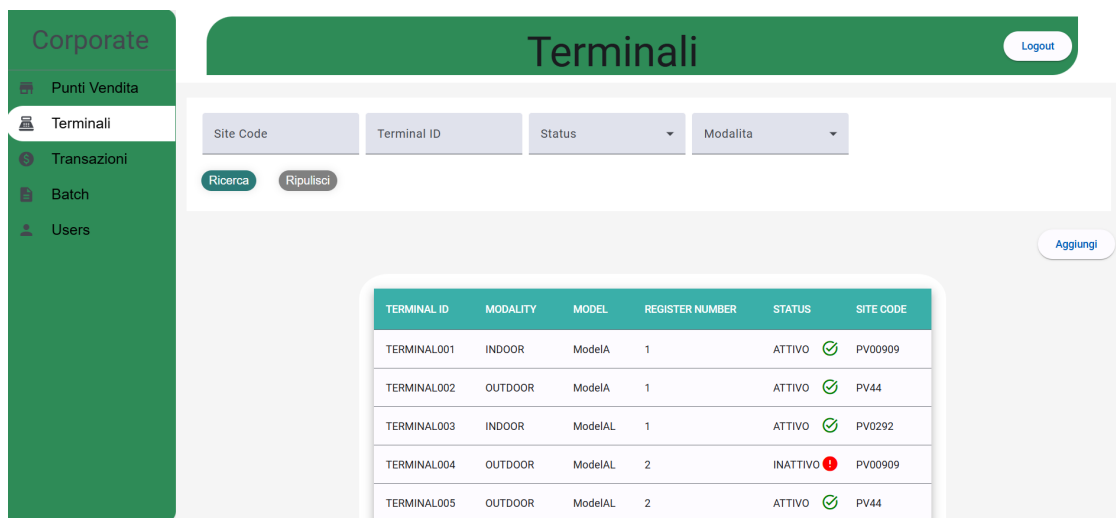


Figure 4.9: Terminals Page

Clicking on the corresponding item in the side menu navigates to the terminals page, which follows the same structure as all other list pages. A filtering section at the top allows users to refine the displayed data. Notably, the filters for sales point and status are particularly useful, enabling users to quickly view inactive terminals and identify their locations.

Terminal ID	Modalita'	Modello	Quantita'	Status
TERMINAL004	OUTDOOR	ModelAL	2	INATTIVO
TERMINAL005	OUTDOOR	ModelAL	2	ATTIVO

Figure 4.10: Add terminal Page

The "Aggiungi" button, located at the top right, opens a modal containing a form for adding a new terminal. All fields in the form are mandatory. Notably, the form does not include a field for the terminal's status. This omission is intentional because when a request to add a new terminal is made, it is created in an *Inactive* state by default. The status will only be updated to *Active* once the installation is confirmed by the technical team.

4.7 Transaction Implementation

Let's now examine the transactions section. In this area, users can perform searches with optional filters, pagination, and sorting. Additionally, users have the ability to view detailed information about each transaction. They can also access transaction-related analytics, providing insights into various operational metrics. However, the analytics feature will be explored in detail later when discussing the microservice dedicated to this functionality. For security reasons, modifying or

adding a transaction directly from the UI is not allowed. However, an endpoint is available in the BackOffice, accessible only to users with an Admin role. This endpoint enables controlled modifications or corrections by executing specific HTTP requests, ensuring that sensitive operations are properly regulated while allowing for necessary adjustments when required.

4.7.1 Retrieving List Implementation

The operation, as expected, is a GET request to the `/transaction` URL, which includes optional filtering and pagination parameters. This follows the same structure as the Sales Points, maintaining consistency across the entire project. This uniform approach simplifies the codebase and enhances user experience, as similar components follow a consistent design and behavior throughout the application.

As with the previous case, the Service layer prepares the objects needed for filtering and pagination. It then retrieves the list of transactions by calling a custom query within the repository. Each transaction includes fields like “type” and “outcome”, which have fixed, predefined values. To reinforce security, both fields are defined using specific Enums. This ensures that only valid, expected values can be used, preventing unauthorized or inconsistent entries and filtering attempts on invalid fields.

4.7.2 Transaction’s Detail

In accordance with RESTful principles, by contacting the URL `/transaction/transactionId` with a GET method, we can retrieve specific information about each transaction. For obvious reasons, this detailed information is not displayed in the main table when the transactions are initially retrieved. This endpoint allows for a more focused view of individual transaction data, providing users with deeper insights into a specific transaction’s details.










4.7.3 FrontEnd Visualization

TRANSACTION ID	SITE CODE	CATEGORY	TYPE	OUTCOME	DATA TRANSAZIONE	TERMINAL ID
1	PV00909	CHIUSURA	FUEL	OK	2024-11-12 - 21:03:23	TERMINAL001
2	PV0292	CHIUSURA	FUEL	ERROR	2024-11-12 - 21:05:57	TERMINAL003

Figure 4.11: Transaction's List

The layout remains similar to the previous design for a consistent user experience. At the top of the page, there is a filter section that allows users to personalize their view of the data according to specific parameters. Below this, there is a section featuring three tab bars, each representing one of the three possible transaction types: FUEL, VAS, and BANKING. These tabs enable users to quickly switch between and view the respective transaction types, providing an efficient way to navigate through different categories of transactions.

System Implementation

FUEL		VAS			BANCARIO		
TRANSACTION ID	SITE CODE	CATEGORY	TYPE	OUTCOME	DATA TRANSAZIONE	TERMINAL ID	
1	PV00909	CHIUSURA	FUEL	OK 	2024-11-12 - 21:03:23	TERMINAL001	
2	PV0292	CHIUSURA	FUEL	ERROR 	2024-11-12 - 21:05:57	TERMINAL003	
3	PV44	ACQUISTO	FUEL	ERROR 	2024-11-12 - 21:05:57	TERMINAL002	
4	PV44	STORNO	FUEL	OK 	2024-11-12 - 21:05:57	TERMINAL005	
5	PV0292	STORNO	FUEL	OK 	2024-11-12 - 21:05:57	TERMINAL006	
6	PV44	ACQUISTO	FUEL	OK 	2024-11-12 - 21:05:57	TERMINAL002	
7	PV44	PREAUTORIZZAZIONE	FUEL	OK 	2024-11-12 - 21:05:57	TERMINAL002	
8	PV00909	NOTIFICA	FUEL	OK 	2024-11-12 - 21:05:57	TERMINAL004	
9	PV00909	STORNO	FUEL	ERROR 	2024-11-12 - 21:08:09	TERMINAL001	

Items per page: 1 - 9 of 9 |< < > >|

Figure 4.12

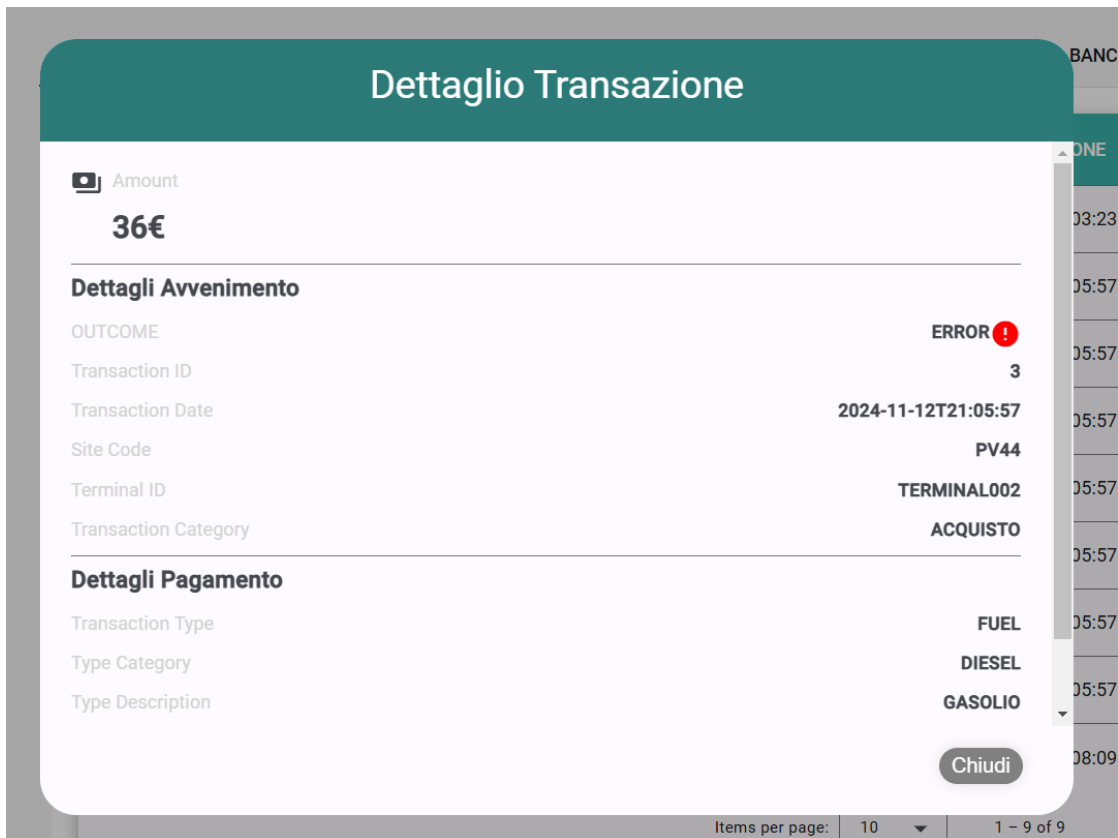


Figure 4.13: Transaction's details

Clicking on a row in the table opens a detailed view where all transaction information that cannot be displayed in the main table is shown. This detailed view includes the banking data related to the transaction, the transaction result, the purchased product, and the quantity involved. This feature provides users with a comprehensive understanding of each transaction without cluttering the primary table.

4.8 Batch Implementation

The batch section allows users to view all executed batches, download their results as CSV files, and create new batches. To address the issue present in the current Back Office, I added a **Type** section during batch creation. This addition enables users to specify the functionality being requested without relying on a specific naming convention for the CSV input file.

Two batches have been developed within this foundational Back Office service structure: the creation of new Sales Points and the generation of a daily transaction

report.

- Sales Point Creation Batch: This batch is initiated by the user uploading an input file. It processes the data and executes the necessary operations to create new sales points based on the provided information.
- Daily Transaction Report Batch: This batch runs automatically at a specific time each day, triggered by a Crontab. It generates a CSV file containing all transactions that occurred during the day, providing a comprehensive summary of daily activity.

We will now analyze the Back End implementation of these two batch types.

4.8.1 Back End - Sales Point Creation

```

1      @PostMapping("/batch")
2      public BatchDTO createBatch(
3          @RequestParam("file") MultipartFile file,
4          @RequestParam("type") BatchType type
5          ) throws IOException {
6
7          BatchDTO batch = batchService.createBatch(file,type,
8          OutComeBatch.TO_DO,0);
9          try {
10             String jsonResponse = producerTemplate.
11             requestBodyAndHeader
12                 ("direct:processFile",
13                 new CamelCreateRequestBody(type,file.
14                 getInputStream(),batch.creationDate.format(DateTimeFormatter.
15                 BASIC_ISO_DATE)+"T"+batch.creationDate.format(DateTimeFormatter
16                 .ISO_LOCAL_TIME).replaceAll(":", "").replaceAll("s+", "")),
17                 "CamelFileName",
18                 file.getOriginalFilename().substring
19                 (0, file.getOriginalFilename().lastIndexOf(".")),
20                 String.class);
21
22             } catch (Exception e) {
23                 throw new InternalError("File non creato nel folder ma
24                 solo nel DB: " + file.getOriginalFilename() );
25             }
26             return batch;
27         }

```

The batch creation process involves two levels: the File System and the Database. This dual-layer approach ensures improved reliability and traceability. Storing

the batch in both the database and the file system allows for better management of metadata and status tracking (via the database), while also providing efficient access and processing of large files (via the file system). Relying solely on one of these methods could result in limitations, such as reduced scalability or difficulties in managing batch statuses and historical records.

The creation is initiated by sending a POST request to the endpoint `/batch`. Upon receiving a Multipart File as input, the file is stored in the database with a status of *TO DO*. After the database entry is successfully created, the batch name is updated to include the current date. This step helps avoid naming conflicts during file storage in the file system. Once the name is modified, the *processFile* route of Apache Camel is invoked to proceed with batch processing.

```

1     from("direct:processFile")
2         .log("Received file: ${header.CamelFileName}")
3         .process(exchange -> {
4             CamelCreateRequestBody requestBody = exchange.getIn().
getBody(CamelCreateRequestBody.class);
5             exchange.setProperty("type", requestBody.getType());
6             exchange.setProperty("creationDate", requestBody.
getCreationDate());
7                 exchange.getIn().setBody(requestBody.getFile()
);
8         })
9         .log("${exchange.getIn().getHeaders('creationDate')}")
10        .log("Converted CSV to JSON: ${body}")
11        .to("file:BackOffice/BATCH/ToDo?fileName=${header.
CamelFileName}-${exchange.getProperty('creationDate')}-${
exchange.getProperty('type')}.csv");

```

The first Apache Camel route takes the input data: the “type”, “creationDate”, and the CSV file—and saves it in the file system under a dedicated folder named “ToDo”. This folder serves as the repository for all batch files awaiting processing. The file name is manipulated to include metadata about the type of operation to be executed. This approach ensures that, even within the same “ToDo” folder, all future batch files can be clearly identified and distinguished based on their operation type.

Another route is configured to listen to the “ToDo” folder. It is triggered automatically whenever a new file is added, ensuring seamless processing of the newly submitted batches.

```

1     from("file:BackOffice/BATCH/ToDo?recursive=true&delay=10000&move
=.done")
2         .process(exchange -> {
3             String fileName = exchange.getIn().getHeader("
CamelFileName" String.class);

```

```

4         ...
5         ...
6         // Save the original file content and headers
7         String fileContent = exchange.getIn().getBody(String.
class);
8         exchange.setProperty("originalFileContent", fileContent)
;
9         exchange.setProperty("originalHeaders", exchange.getIn()
.getHeaders());
10        });
11        .log("New file detected: ${header.CamelFileName} in folder
${header.CamelFileParent} of type ${header.type} in date ${
header.creationDate}")
12        .choice()
13        .when(header("type").isEqualTo("CREATE_SITE"))
14        .doTry()
15        .unmarshal(csv)
16        .process(exchange -> {
17            List<List<String>> jsonArray = exchange.getIn().
getBody(List.class);
18            List<AddSalePointRequestDTO> transformedList = new
ArrayList<>();
19            for (List<String> entry : jsonArray) {
20                AddSalePointRequestDTO transformedEntry = new
AddSalePointRequestDTO(
21                    entry.get(0),
22                    entry.get(1),
23                    ...
24                );
25                transformedList.add(transformedEntry);
26            }
27            exchange.getMessage().setBody(transformedList);
28        })
29        .marshal().json(JsonLibrary.Jackson)
30        .bean(batchServiceImpl, "addSalePoint(${exchange.getMessage().
getBody()}, ${header.creationDate})")
31        .doCatch(IOException.class, BatchNotFound.class)
32        .log("Error processing file: ${header.CamelFileName}
")
33        .end()
34        .choice()
35        .when(simple("${body.duplicatesList.size()} == 0"))
36        .log("File Batch ${header.CamelFileName} è stato
eseguito correttamente ")
37        .process(exchange -> {
38            String originalFileContent = exchange.getProperty(
"originalFileContent", String.class);
39            Map<String, Object> originalHeaders = exchange.
getProperty("originalHeaders", Map.class);

```

```

40         exchange.getIn().setBody(originalFileContent);
41         exchange.getIn().setHeaders(originalHeaders);
42     })
43     .to("file:BackOffice/BATCH/Finished?fileName=${
header.CamelFileName}")
44     .otherwise()
45     .log("Il File Batch ${header.CamelFileName} contiene
dei SiteId duplicati: ${body.duplicatesList}")
46     .log("Nel File Batch ${header.CamelFileName} i
seguenti Site sono stati inseriti correttamente ${body.
uniqueList}")
47     .process(exchange -> {
48         String originalFileContent = exchange.
getProperty("originalFileContent", String.class);
49         Map<String, Object> originalHeaders = exchange.
getProperty("originalHeaders", Map.class);
50         exchange.getIn().setBody(originalFileContent);
51         exchange.getIn().setHeaders(originalHeaders);
52     })
53     .to("file:BackOffice/BATCH/Error?fileName=${header.
CamelFileName}")
54     .endChoice()
55     .otherwise()
56     .end();

```

This route is more complex but serves as a "gateway" for all future batches. When a new file is placed in the *ToDo* folder, it is picked up and processed. The file name is parsed to extract the operation type. Based on this type, a switch determines which route or function to execute for the batch.

For the creation of sales points, the input CSV file is parsed, and a specific DTO is generated for each record. A dedicated service for creating new sales points is invoked directly by the route.

In the current BackOffice system, if any issues are found in the data within the CSV file, the entire batch fails. To improve this process, I modified the behavior to make the service more detailed and user-friendly. The updated implementation ensures that the batch is completed even when errors are present, and it highlights the specific rows in the input file that caused issues. This approach introduces a distinction between **ERROR**, **PARTIAL ERROR**, and **COMPLETE**, making the service more efficient and enhancing the overall user experience.

Once the service completes its execution, Apache Camel captures the result and moves the output file to one of the corresponding folders: "ERROR", "PARTIAL ERROR", or "FINISHED", providing clear feedback about the batch outcome.

4.8.2 Back End - Daily Transaction

The creation of a CSV file containing the day's latest transactions is managed by a route connected to a Crontab, which triggers the route every day at midnight (00:00). The implementation using Apache Camel offers simplicity in both management and modification, making it a robust and flexible solution for scheduling and handling such periodic tasks.

```

1      from("cron:tab?schedule=0+0+*****?")
2          .bean(batchServiceImpl, "getLastDayTransactions")
3          .process(exchange -> {
4              LocalDate day = LocalDate.now().minusDays(1);
5              exchange.setProperty("date", day.toString());
6          })
7          .marshal().csv()
8          .to("file:BackOffice/BATCH/Finished?fileName=
TransactionsDay-${exchange.getProperty('date')}.csv")
9          .process(exchange -> {
10             InputStreamCache inputStreamCache = exchange.getMessage
11             ().getBody(InputStreamCache.class);
12             byte[] content = (inputStreamCache.readAllBytes());
13             MultipartFile file = new CustomMultipartFile(content, "
TransactionsDay-"+ exchange.getProperty("date") + ".csv");
14             exchange.getIn().setBody(file);
15         })
16     .bean(batchServiceImpl, "createBatch(${exchange.getMessage().
getBody()}, 'TRANSACTIONS_DAYBEFORE', 'COMPLETE', 1)");

```

When the cron job is triggered, it executes the batch process, which calls a specific service to fetch the latest transactions from the database and generates a CSV file from the retrieved data. Apache Camel simplifies this process by enabling the execution of a function and the transformation of its output into a custom CSV file with minimal code. This eliminates the need to write complex, dedicated functions, significantly reducing development effort while maintaining flexibility and scalability.

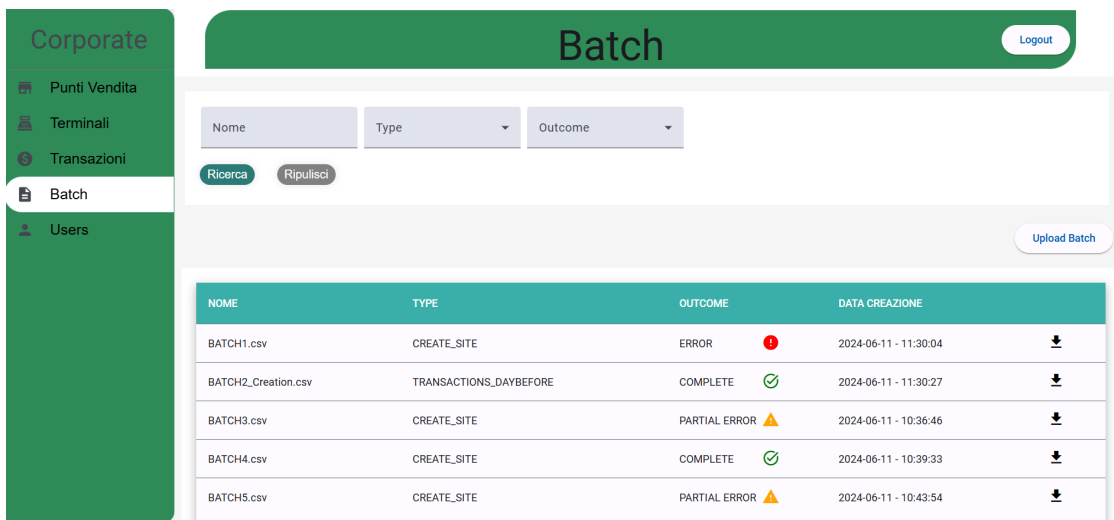
4.8.3 Retrieving List Implementation

Performing a GET request to the endpoint `/batch` invokes a service structured similarly to previous retrieval services. This service allows the user to obtain a filtered and sorted list of batches, ensuring consistency in functionality and ease of use across the system's retrieval operations.

4.8.4 Retrieve CSV file

The download of the CSV file containing the output of a batch is executed by making a GET request to `/batch/batchId`. This endpoint retrieves the file, provided it exists, in *BLOB* (Binary Large Object) format, which is stored in the database.

4.8.5 FrontEnd Visualization



NOME	TYPE	OUTCOME	DATA CREAZIONE	
BATCH1.csv	CREATE_SITE	ERROR	2024-06-11 - 11:30:04	↓
BATCH2_Creation.csv	TRANSACTIONS_DAYBEFORE	COMPLETE	2024-06-11 - 11:30:27	↓
BATCH3.csv	CREATE_SITE	PARTIAL ERROR	2024-06-11 - 10:36:46	↓
BATCH4.csv	CREATE_SITE	COMPLETE	2024-06-11 - 10:39:33	↓
BATCH5.csv	CREATE_SITE	PARTIAL ERROR	2024-06-11 - 10:43:54	↓

Figure 4.14: Batch's List

In this section, the user can visually explore the entire list of batches, with a clear graphical representation of each batch's outcome. The interface includes icons that immediately highlight the result of each batch, making it easier for the user to quickly identify whether the batch was successful, partially successful, or encountered errors. On the right side of each batch entry, there is a download icon that allows the user to easily download the corresponding CSV file, providing quick access to the batch's output. This layout enhances the user experience by combining a visually intuitive design with practical functionality, ensuring users can efficiently monitor and retrieve batch results

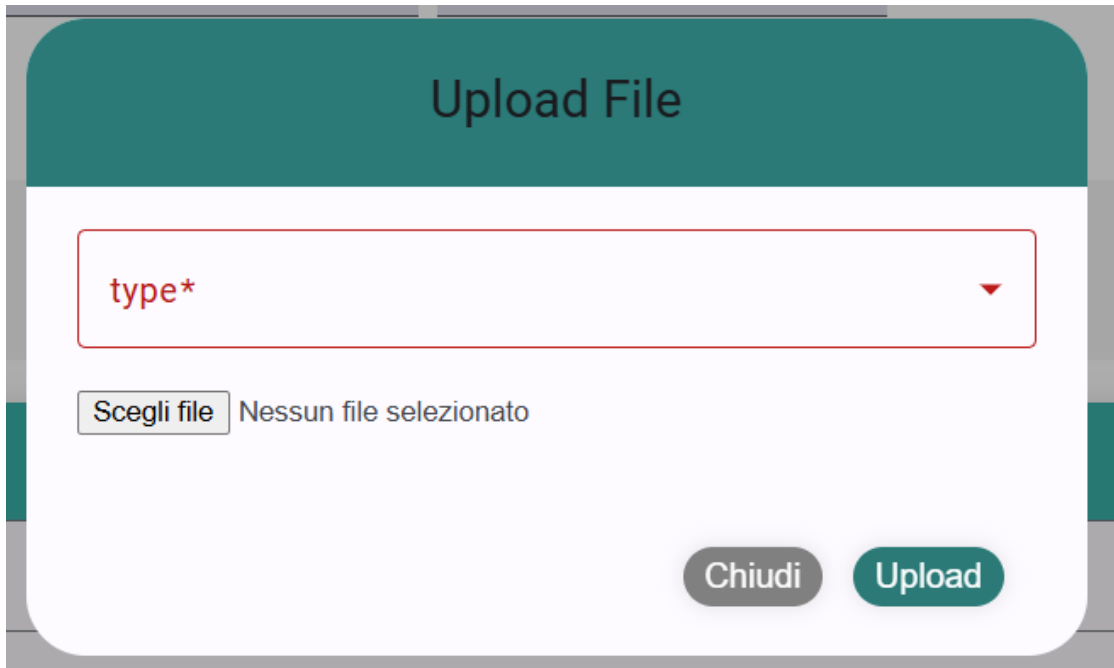


Figure 4.15: Upload Batch

By clicking the "Upload Batch" button located at the top of the page, a modal window is opened, allowing the user to select the type of operation that the batch should perform. This step ensures that the batch is configured correctly according to its specific function. Additionally, the modal provides an interface for the user to upload the file by accessing their local file system. This user-friendly feature simplifies the process of batch file submission, making it easier for users to perform batch operations without navigating through complex steps, and streamlines the overall batch management workflow.

4.9 Analysis implementation

The Analysis microservice is responsible for processing transaction data to generate comprehensive statistics. Its development involved two key components: integrating the Oracle and MongoDB databases via Kafka Connect and implementing both the back-end and front-end functionalities.

4.9.1 Kafka Connect

The creation of Kafka Connect connectors is achieved via a POST request to the endpoint `/connectors` on the port where the Kafka Connect Docker service is

running. Two connectors were implemented:

1. The first connector writes new transaction data to a message in a Kafka topic.
2. The second connector collects these messages from the topic and inserts the data into MongoDB.

This setup creates a reliable data pipeline between the transactional database and the analysis system.

```
1   {
2   "name": "oracle-connector",
3   "config": {
4     "connector.class": "io.debezium.connector.oracle.
OracleConnector",
5     "tasks.max": "1",
6     "database.hostname": "oracle-db",
7     "database.port": "1521",
8     "database.user": "****",
9     "database.password": "*****",
10    "database.dbname": "*****",
11    "database.pdb.name": "*****",
12    "database.server.name": "oracle",
13    "database.history.kafka.bootstrap.servers": "kafka:29092",
14    "database.history.kafka.topic": "schema-changes.oracle",
15    "table.include.list": "public.T_TRANSACTION",
16    "plugin.name": "oracle",
17    "snapshot.mode": "always",
18    "key.converter": "org.apache.kafka.connect.storage.
StringConverter",
19    "value.converter": "org.apache.kafka.connect.json.
JsonConverter",
20    "key.converter.schemas.enable": "false",
21    "value.converter.schemas.enable": "false",
22    "topic.prefix": "kafka_transactions_"
23  }
24 }
```

Every time a CRUD operation is executed to the T_TRANSACTION, this connector captures the event and publishes it to the kafkatransactions topic. This ensures that any updates, inserts, or deletes made to the T_TRANSACTION table in the Oracle database are seamlessly streamed to Kafka in near real-time.

```
1   {
2   "name": "mongo_transactions_sink",
3   "config": {
4     "connector.class": "com.mongodb.kafka.connect.
MongoSinkConnector",
```

```
5     "tasks.max": "1",
6     "topics": "kafka_transactions",
7     "connection.uri": "mongodb://web2:password@mongo:27017",
8     "database": "analyticsDb",
9     "collection": "transactions",
10    "key.converter": "org.apache.kafka.connect.storage.
StringConverter",
11    "value.converter": "org.apache.kafka.connect.json.
JsonConverter",
12    "value.converter.schemas.enable": false,
13    "key.converter.schemas.enable": false,
14    "transforms": "ExtractTransaction,FilterFields,RenameFieldId",
15    "transforms.ExtractTransaction.type": "org.apache.kafka.
connect.transforms.ExtractField$Value",
16    "transforms.ExtractTransaction.field": "after",
17    "transforms.FilterFields.type": "org.apache.kafka.connect.
transforms.ReplaceField$Value",
18    "transforms.FilterFields.include": "id,outcome,site_code,
date_transaction",
19    "transforms.RenameFieldId.type": "org.apache.kafka.connect.
transforms.ReplaceField$Value",
20    "transforms.RenameFieldId.renames": "id:_id"
21 }
22 }
```

The following connector listens to the topic and, by performing a data filtering operation, inserts or updates transaction data in the document in MongoDB. The delete operation is not considered, as transactions will not be deleted from the database.

4.9.2 Back End Implementation

The statistics I developed include a count of failed transactions, identified by an OUTCOME value of "KO", and the one that ended well, "OK", for each of the three categories, as well as an analysis of failed transactions ("KO") by sales point. All statistics are calculated based on transactions from the past week. This approach ensures that the representation is not distorted by the inclusion of historical data, providing a clearer and more accurate analysis. Both statistics are collected through a single GET endpoint, `/transactions/fails`. Without the constraint of retrieving only the most recent transactions, there would have been no need to develop a custom query, as Spring Data provides automatic query derivation based on documented conventions.

To implement this functionality, two queries were created to query the MongoDB database and retrieve the "KO" and "OK" transactions from the past week. The

service layer handles filtering by category and sales points, returning the processed result to the Front End.

4.9.3 Front End Visualization

Clicking the button in the top-right corner of the transactions menu page allows users to switch to the statistics view.

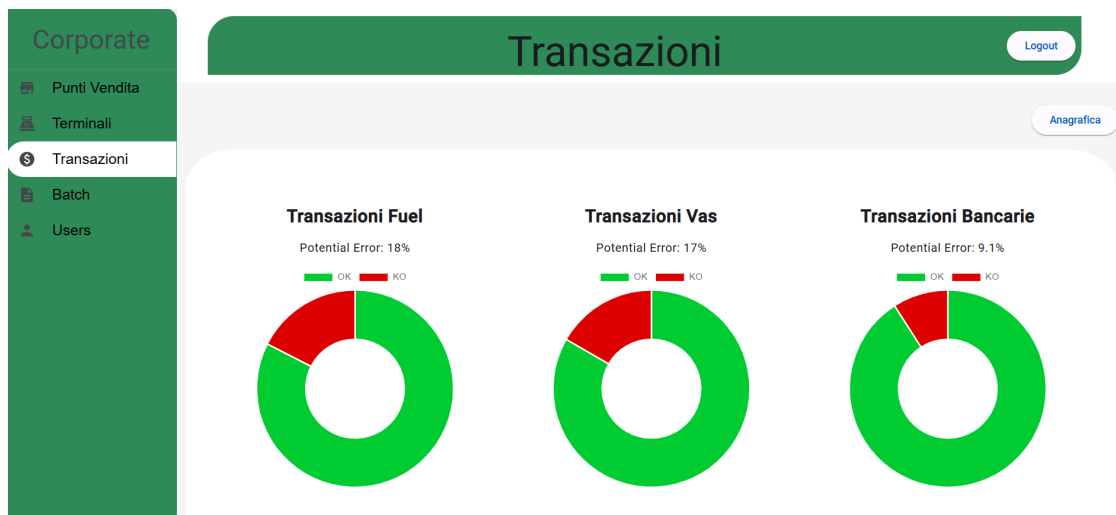


Figure 4.16: First Dashboard

The first dashboard section displays donut-pie charts that present the performance of transactions across the FUEL, VAS, and BANKING categories. These charts calculate the error rate percentage for each category, making it immediately accessible and clear for the user to identify potential issues at a glance. This intuitive visualization enhances the user's ability to monitor and respond to transaction performance effectively.

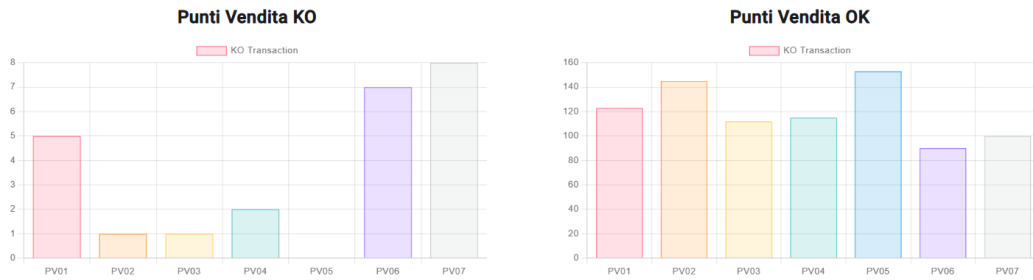


Figure 4.17: Second Dashboard

The second dashboard features two bar charts that provide a comprehensive view of transaction performance across sales points. The first chart illustrates the number of failed transactions (KO) with the x-axis representing the sales points and the y-axis showing the corresponding KO counts. The second chart focuses on successful transactions (OK), using the same axes for consistency. Together, these visualizations enable users to analyze the performance of each sales point, quickly identifying locations with the most significant issues or those excelling in transaction success rates. By offering this dual perspective, the dashboard facilitates both targeted problem-solving for underperforming locations and recognition of high-performing ones, supporting overall performance optimization.

Chapter 5

Results and Comparison

After showcasing the final steps in the implementation of the back office and observing how it gradually took shape, this chapter will focus on evaluating the results achieved. It will include a comparison with the software currently in use and an analysis of the improvements introduced.

5.1 Technology improvements

All the technologies employed in the current back office have undergone significant upgrades, leading to substantial improvements in performance, security, and scalability.

5.1.1 Java 21 and Tomcat

The transition from Java 8 to Java 21 brings substantial improvements across multiple dimensions:

- **Performance Enhancements:** One of the standout enhancements is the considerable reduction in application startup time, achieved through the improved Garbage Collector and the enhanced Java Virtual Machine (JVM). Additionally, features like Class Data Sharing (CDS) optimize the reuse of class metadata across processes, enabling faster initialization. This is particularly crucial in a development environment where frequent server restarts are necessary for testing updates.

A measured improvement of **76.10%** in startup time was recorded. The previous BackOffice system required an average of 2 minutes and 57 seconds to start, while the upgraded system now starts in just 37 seconds. Furthermore, the shift to a microservices architecture amplifies these benefits, as changes

often require restarting only a single microservice rather than the entire system. This minimizes downtime and accelerates the testing and deployment process.

- **Security Updates:** Although Java 8 continues to receive security patches, Java 21 introduces the latest advancements and new features that go far beyond mere updates. These improvements make applications more secure and robust by incorporating modern enhancements like better garbage collection, advanced JVM capabilities, and enhanced APIs. This upgrade ensures that the application is future-proof, ready for the eventual end of life (EOF) of Java 8, when it will no longer receive security patches.
- **New Features and Enhancements:** Java 21 has simplified the development of specific features, thanks to the introduction of new functionalities like enhanced modularity.

Moreover, upgrading to Java 21 enables the use of the latest version of Tomcat, which brings further improvements in performance, security, and functionality. This update also aligns with the ongoing evolution of Java standards, making it easier to integrate with modern libraries and frameworks.

5.1.2 Angular vs AngularJs

The transition from AngularJS to Angular 18 has brought significant enhancements that have greatly enhanced both performance and developer experience. One of the most notable changes is the complete overhaul of the framework's architecture. AngularJS relied heavily on two-way data binding, which could cause performance bottlenecks. Angular 18, however, has adopted a more efficient unidirectional data flow, coupled with reactive programming patterns, which not only improves the performance but also makes the code easier to maintain.

Another significant improvement is in the area of development speed and build times. Angular 18 supports a more modern and efficient build process using the Angular CLI, significantly reducing development time. The toolchain has become much faster and more customizable, with features like incremental compilation and faster hot module reloading, which were absent in AngularJS. This makes iterative development much more efficient, as changes are reflected almost instantly without needing to rebuild the entire project from scratch. Without AngularCLI, every change in the FrontEnd requires a manual build of the entire project, which involves managing modules and dependencies manually. The process of visualizing the implemented changes can take anywhere between 40 and 90 seconds, notably impacting development efficiency. The new version of Angular significantly reduces build times to under 3 seconds, representing a **96.67%** improvement in speed. Additionally, AngularJS has reached its end of life (EOF), leaving open CVEs

(Common Vulnerabilities and Exposures) with high-risk scores that will no longer receive security patches from Google. This makes applications built on AngularJS increasingly vulnerable to potential attacks. In industries like the oil sector, where security breaches can have catastrophic consequences, this poses a significant risk. Vulnerabilities in web applications can lead to unauthorized access to critical data, disruption of operations, and even physical damages in sensitive systems. As such, upgrading to a more secure, modern framework like Angular 18 not only improves performance but also greatly mitigates these security risks.

5.1.3 IAM

The introduction of an Identity and Access Management (IAM) system has greatly improved the security of the application by shifting the responsibility for authorization away from developers and delegating it to the IAM system. This approach ensures that the application remains up-to-date with the latest security improvements without placing an additional burden on the development of the Back Office. The codebase has been significantly simplified and reduced, allowing for faster development cycles. Furthermore, the creation of multiple realms and user groups, provided by Keycloak, enables a quick and seamless way to offer Back Office services to different companies. This is achieved without making direct changes to the code; instead, administrators can manage access and roles entirely through the IAM's user interface. This flexibility streamlines the process of scaling and offering customized access controls while enhancing overall security.

5.2 Performance improvements

In the table below, the performance measurements for various operations between the two Back Offices are displayed. These measurements were obtained after ensuring the databases were standardized with an equal amount of data.

Operation	Old Back Office	New Back Office
GET (No Filter) Sales Points	0.697s	0.246s
GET (No Filter) Terminals	1.020s	0.422s
GET (No Filter) Transactions	2.123s	0.989s
GET (No Filter) Statistics	1.401s	0.879s
GET (No Filter) Batch	2.276s	1.023s

Table 5.1: GET Requests without Filtering

Operation	Old Back Office	New Back Office
GET (With Filter) Sales Points	3.190s	0.489s
GET (Details) Sales Points	2.951s	0.382s
GET (With Filter) Terminals	1.553s	0.679s
GET (With Filter) Transactions	2.677s	1.243s
GET (Details) Transactions	2.129s	1.011s

Table 5.2: GET Requests with Filtering

Operation	Old Back Office	New Back Office
PUT Sales Points	2.29s	1.176s
POST New Batch	1.983s	0.899s

Table 5.3: Other Operations (Insert, Update)

The calculations result in an average improvement of:

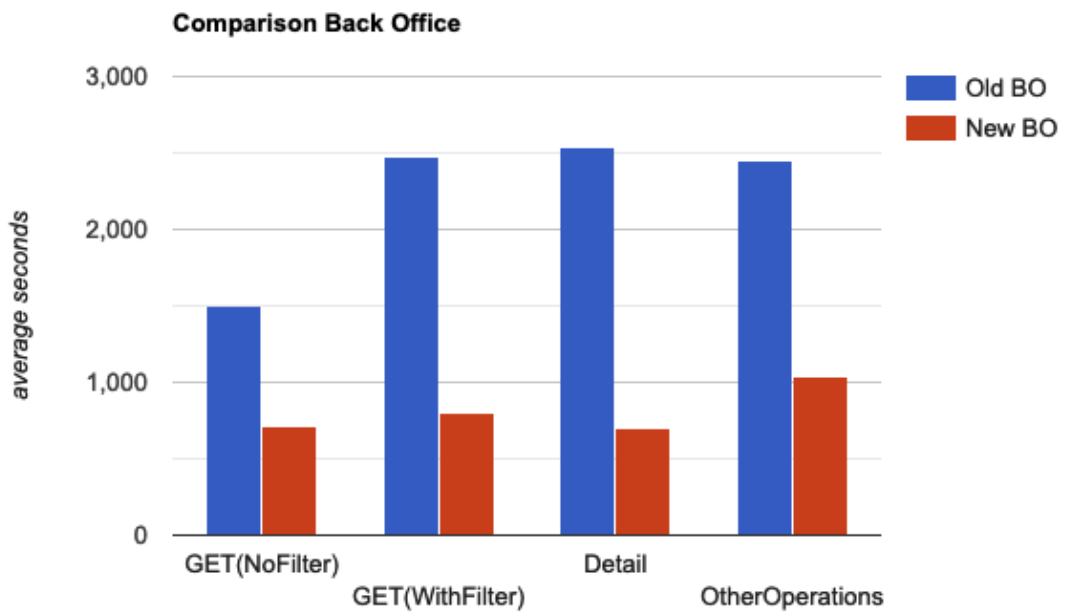


Figure 5.1: Performance Comparison

- GET Requests without Filtering: 53.81% improvement
- GET Requests with Filtering: 78.82% improvement
- Detail operations: 73.66% improvement

- Other Operations (Insert, Update): 79.9% improvement

The performance improvement was significant, thanks to better query management, an optimized code structure, and the adoption of modern technologies, leading to an average improvement of 71.54%.

Improvements have also been made to the user experience when using the BackOffice. The introduction of batch type definitions has eliminated the need to assign specific names to each file, a requirement that often led to errors only discovered during batch execution.

The batch process for creating sales points has been enhanced to continue execution even when errors are present in the file. This allows correctly formatted sales points to be processed while providing detailed error reporting for problematic entries.

Additionally, the transactions section now enables seamless and more intuitive navigation between different categories. This contrasts with the previous BackOffice, which required navigating through separate menu sections for each category.

Chapter 6

Conclusion

As we reach the conclusion of this thesis, the crucial role of business support software, such as Back Office systems, becomes even more apparent. In industries like oil and gas, companies face the constant challenge of expanding across territories, making a centralized Back Office system indispensable for monitoring and managing activities at various sales points and terminals. The ability to consolidate these operations into a single, streamlined platform ensures that companies can maintain oversight and control over their distributed assets.

Given the critical importance of Back Office systems in ensuring smooth operations and safeguarding sensitive data, it is imperative that these systems evolve with technological advancements. The rapidly changing landscape of technology necessitates that Back Office systems stay updated with the latest software, tools, and security protocols. With the constant threat of cyberattacks, especially in sectors like oil and gas, where sensitive data is crucial, ensuring robust security standards is essential.

Failure to update these technologies not only leads to a degradation in system performance but also poses significant risks in terms of compatibility with other systems. As demonstrated, performance improvements of over 70% have been achieved through the upgrade, highlighting the direct impact of technological advancements on operational efficiency. Moreover, an outdated system can hinder the ability to collaborate seamlessly with third-party software, complicating integrations and making it difficult to adapt to new business requirements.

The work done in this project has resulted in the design and development of the new Back Office, incorporating improved functionalities, security standards, and a more efficient architecture. The update not only addresses performance concerns but also enhances the overall user experience. With the core features in place, the next phase of development can focus on adding additional functionalities and testing the system thoroughly. The ultimate goal is to replace the legacy Back Office with the new, upgraded version, offering a more streamlined, secure, and

scalable solution. This migration will not only make the system more efficient but also provide a foundation for faster, more effective development of future features, ensuring the continued growth and success of the company in a highly competitive and evolving market.

Bibliography

- [1] Petteri Raatikainen. «What are system integrations? Methods, challenges and best practices». In: *ONEIO* (2024). URL: <https://www.oneio.cloud/blog/what-are-system-integrations#what-is-system-integration> (cit. on p. 22).
- [2] Spring Team. *Spring Framework Documentation*. 2024. URL: <https://spring.io/projects/spring-integration> (cit. on p. 24).
- [3] Lakshmeesh achar. «A Basic Guide to Apache Camel Architecture». In: *Medium* (2024). URL: <https://medium.com/@lakshmeeshachar/a-basic-guide-to-apache-camel-architecture-dccaee8690ee> (cit. on p. 26).
- [4] Iwan Buchler. «The History of Identity and Access Management». In: *LinkedIn* (2023). URL: <https://www.linkedin.com/pulse/history-identity-access-management-iwan-buchler/> (cit. on p. 27).
- [5] *The OAuth 2.0 Authorization Framework*. Standards Track. Internet Engineering Task Force (IETF), 2012. URL: <https://datatracker.ietf.org/doc/html/rfc6749#page-4> (cit. on p. 27).
- [6] KeyCloak Team. *Keycloak features and concepts*. 2024. URL: https://www.keycloak.org/docs/latest/server_admin/ (cit. on p. 29).
- [7] William Zeller and Edward W. Felten. «Cross-Site Request Forgeries: Exploitation and Prevention». In: *Princeton University* (2008). URL: <https://people.eecs.berkeley.edu/~daw/teaching/cs261-f11/reading/csrf.pdf> (cit. on pp. 34, 35).
- [8] Angular Team. *Angular Documentation*. 2024. URL: <https://angular.dev/guide/di> (cit. on p. 38).
- [9] Rina Rai Amita Jain. «Maximize User Adoption of New Software in 3 Steps». In: *Capterra* (2020). URL: <https://www.capterra.com/resources/maximize-user-adoption/> (cit. on p. 40).