1 2 9 0

UNIVERSIDADE D
COIMBRA

João Miguel Rainho Mendes

Implementation of an Event Sourcing Application

Curricular internship – Internship Report

Dissertation in the context of the Master in Informatics Engineering, specialization in Software Engineering, advised by Prof. Filipe Araújo, PhD student Jaime Correia and PhD student André Bento, and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

July 2022

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE Đ
COIMBRA

DEPARTMENT OF INFORMATICS ENGINEERING

João Miguel Rainho Mendes

Implementation of an Event Sourcing Application

Curricular internship - Internship Report

Dissertation in the context of the Master in Informatics Engineering, specialization in Software Engineering, advised by Prof. Filipe Araújo, PhD student Jaime Correia and PhD student André Bento, and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

July 2022

Acknowledgements

I would like to thank Professor Filipe Araújo, for the support, advice, and encouragement during this internship. Likewise, I would also like to thank PhD students Jaime Correia and André Bento for the advice, support, and feedback given throughout this internship. I also would like to thank all the individuals associated with Power Project for their availability and support. Finally, I want to thank my friends and family for the support given over these past months.

This work is funded by the project POWER (grant number POCI-01-0247-FEDER-070365), co-financed by the European Regional Development Fund (FEDER), through Portugal 2020 (PT2020), and by the Competitiveness and Internationalization Operational Programme (COMPETE 2020).

Abstract

With the advancements in technologies the complexity and size of systems have been increasing. Consequently, there is a need for bigger systems (for example banks) to keep a track of the history of requests. This happens because an administrator cannot easily check the requests processed by the system in real time. One way to create the history of requests is by using Event Sourcing. Using this method each request is saved in the log as an atomic event that can also be used for various operations (for example event reprocessment or an auditing). But since this method is fairly new there is a lack of expertise.

This internship proposes the implementation of Event Sourcing. To do this a collaboration between the Department of Informatics Engineering (DEI) and Altice Labs was formed. The objective of this internship is to implement Event Sourcing in a system prototype given by Altice Labs, while simultaneously creating a library of Event Sourcing to help future implementations. To implement this concept research was made on the state of the art, requirements were defined, and an architecture of the system was made. Finally, the implementation of Event Sourcing was performed along with the testing of the system.

With the conclusion of this internship, Event Sourcing was implemented in the system. With this a high level of understanding of Event Sourcing was obtained. Furthermore, various scripts to create snapshots of events were created. Finally, a library was conceived. This library will allow an easier implementation of Event Sourcing in the future as it has the bases for some of the more complex operations found in Event Sourcing.

Keywords

Complexity. Event Sourcing. Altice Labs.

Resumo

Com o avanço da tecnologia, a complexidade e tamanho dos sistemas tem vindo a aumentar. Consequentemente, existe uma necessidade pela parte de maiores sistemas (por exemplo bancos) de manter um histórico de pedidos. Isto acontece pois um administrador não consegue facilmente ver os pedidos processados pelo sistema em tempo real. Uma maneira de criar o histórico de pedidos é usando Event Sourcing. Ao usar este método cade pedido é guardado numa base de dados como um evento atómico que pode também ser usado para várias operações (por exemplo o reprocessamento de eventos ou fazer uma auditoria). No entanto como este método é bastante novo, existe uma falta de experiência.

Este estágio propõe a implementação do Event Sourcing. Para fazê-lo, iniciou-se uma colaboração entre o Departamento de Engenharia Informática e a Altice Labs. O objetivo deste estágio curricular é implementar o Event Sourcing num protótipo de sistema dado pela Altice Labs, e simultaneamente criar uma libraria de Event Sourcing para ajudar implementações futuras. Para implementar este conceito foi feita uma pesquisa sobre o estado da arte, foram definidos requisitos, e foi feita uma arquitetura do sistema. Finalmente, a implementação do Event Sourcing foi realizada juntamente com o teste do sistema.

Com a conclusão deste estágio, Event Sourcing foi implementado no sistema. Através disto um grande nível de conhecimento sobre Event Sourcing foi adquirido. Além disso, várias scripts para criar snapshots de eventos foram criadas. Finalmente, uma libraria foi concebida. Esta libraria vai facilitar a implementação de Event Sourcing no future devido a ter as bases para as operações mais complexas de Event Sourcing.

Palavras-Chave

Complexidade. Event Sourcing. Altice Labs.

Contents

Acronyms

ALB Altice Labs.

BSON Binary JSON.

CISUC Centre for Informatics and Systems of the University of Coimbra.

CQL Cassandra Query Language.

CQRS Command Query Responsibility Segregation.

DEI Department of Informatics Engineering.

ES Event Sourcing.

FCTUC Faculty of Sciences and Technology at the University of Coimbra.

GUI Graphical User Interface.

HTTP Hypertext Transfer Protocol.

IDE Integrated Development Environment.

JSON JavaScript Object Notation.

RDBMS relational database management system.

TCP Transmission Control Protocol.

UC University of Coimbra.

VCS Version Control System.

List of Figures

List of Tables

Chapter 1

Introduction

This report represents the project associated with a Master's Degree in Software Engineering from the Faculty of Sciences and Technology at the University of Coimbra (FCTUC), in the curricular year of 2021/2022. This project is taking place in the Centre for Informatics and Systems of the University of Coimbra (CISUC) at the Department of Informatics Engineering (DEI) of the University of Coimbra. This project is also associated with Altice Labs (ALB) in the context of the Power Project and was done with their cooperation.

1.1 Context, Problem, and Motivation

In recent times, the software systems are getting ever more complex [28][27], and because of this companies have to waste an increasing amount of money just to manage the performance of their system [29][30]. Furthermore, this applies to all kinds of systems, from small apps to multilayered systems. More precisely, the ALB system has some clear flaws that could be fixed to upgrade it. Namely, it is impossible to see what happened in the system in a clear and simple way (similar to doing an auditing), and the possibility to check information regarding the user or the requests processed at a given date. A good way to apply the points stated above is by using a methodology named Event Sourcing (ES).

While using ES, each action is saved as an event that is automatically saved in a database as a part of a sequence of events. For instance, we can imagine a bicycle selling system. In this system when a bicycle is sold the only thing that is done is update the current number of bicycles. But with ES an event is also saved, this event can contain different information about the operation. For instance, details of the buyer, the date on which the bicycle was bought, the id of the bicycle, and, how much time the system took to process the whole operation. By doing this, the developers can see how their systems reacts to tasks in real-time and the flow of events that made the system get to its current state. Additionally, since each event is a single operation, it is possible to accurately check the state of the system and its users at a given time, when a system error occurred, and, if necessary, make a compensation. But this compensation in itself is a complex operation, as will be further explored later in the report. There can be various ways of compensating an action, and there does not exist a method that is clearly superior to every other.

Nevertheless, the concept of ES is fairly new in the tech industry. Consequently, there does not exist a great understanding and expertise on it. In order to gain expertise, the Department of Informatics Engineering(DEI) will be involved with ALB in the Project Power. The latter will be in charge of producing a working prototype for a Charging System (a system that handles the increase and decrease of cellphone credit) in which the intern will implement ES.

Regarding the prototype mentioned above, it is responsible for the management of various transactions. More precisely, debit, credit, and reservation operations. When a request arrives on the system the information will go through various microservices where it will be analysed and transformed for different purposes. In this path, two different microservices possess databases that are responsible for keeping the user's data which will be accessed and altered if need be. Furthermore, it will be in those microservices that ES will be implemented.

1.2 Objectives

The main objective of this internship is to upgrade the charging system for the Power Project. For this upgrade, the concepts of ES will be utilized. That is to say, that while the system is working normally (resolving credit, debit, and reservations requests) there are operations that must be done, namely:

- When a request arrives at the system an atomic event corresponding to the request needs to be formed and saved in a database one hundred percent of the time.
- In the case of an error/rollback in the transactions, the system must be capable of creating a rollback/compensation event or deleting the original event one hundred percent of the time.
- In order to reload the system in a fast manner, periodic system snapshots must be created. This needs to always happen when the trigger decided by the user activates.

A secondary objective is the creation of an ES library, as there is not much information on it and to help future implementations. To achieve this the objectives stated above will be implemented in the most modular way possible.

Because the work will be done in a prototype created by ALB and only modified by the intern, it is necessary to reach solutions that will please both parties while not compromising the performance of the system. In the case of ALB they expect an implementation of ES that works for their current system. Hence, it is expected by them that all of the main functionalities of ES (described in section 3.1.2) are working correctly. Regarding the objective of DEI, as mentioned before, it is expected that by the end of the internship a library of ES is created. In addition to this, it is required for the intern to apply changes or add new functionalities if they are required by the ALB team. As such, there may exist some restrictions in the implementation or there might exist the need to redo certain parts of the project.

All this being said, the intern will be responsible for searching for various ways of applying the concepts, technologies that can be used, defining requirements, and implementing the solution. At the end of the internship, it is expected that the prototype will be running with no issues and a library of ES is created, tested, and can be used without issues.

1.3 Document Structure

This report is divided into the following chapters:

- **Chapter 2** is dedicated to the planning of the project schedule. It contains a description of the planning done for the first and second semester, moreover, some diagrams are also provided to help further the understanding of the planning. In addition to this, a part of Chapter 2 is focused on risk management and the threshold of success.
- **Chapter 3** is focused on the research done by the intern. It contains research done on various topics of importance to this internship, additionally, there is also a comparison between various technologies that could be used and a final decision on which technology is going to be used.
- **Chapter 4** is dedicated to the Architectural Drivers. It provides a description of the current system, the functional requirements, and their restrictions and quality attributes.
- **Chapter 5** contains the architecture of the system, presented in the form of a C4 Model.
- **Chapter 6** describes the tools and processes utilized in the system's development, as well as a detailed description of the system's modules.
- **Chapter 7** explains the testing performed on the system as well as an analysis of the findings.
- **Chapter 8** consists of an analysis of the internship, more concretely, what were the plans versus what was done, work that can be done in the future, difficulties in this internship, and finally, some thoughts on this internship.

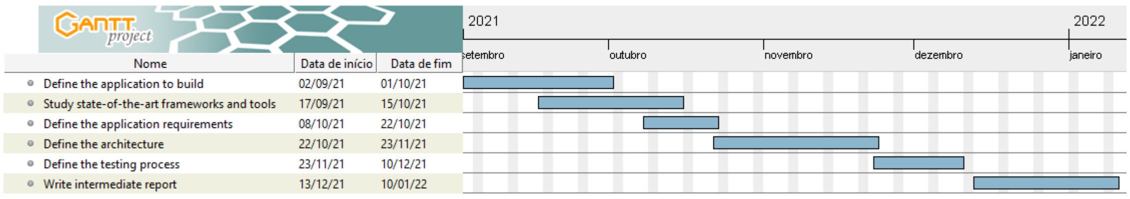
Chapter 2

Planning

In this chapter, the internship schedule and the risks that may occur while doing the internship will be explained. The internship is divided into two semesters, which have completely different purposes. Both chapters will have a Gantt chart to help the visualization of the time and tasks, this chart was done using Gantt Project [32].

2.1 First Semester

The supervisors performed the planning for the first semester. Firstly, the application was defined and took one month. Secondly, the research was done on the state of the art and the technologies that would be used, and this occupied roughly one month. In the next two weeks, the requirements were defined. After this, the system architecture was defined in one month. Lastly, the testing process was defined in two weeks. The remaining time was used to write the intermediate report. This can be observed in Figure 2.1, the Gantt chart.



Nome	Data de início	Data de fim
Define the application to build	02/09/21	01/10/21
Study state-of-the-art frameworks and tools	17/09/21	15/10/21
Define the application requirements	08/10/21	22/10/21
Define the architecture	22/10/21	23/11/21
Define the testing process	23/11/21	10/12/21
Write intermediate report	13/12/21	10/01/22

Figure 2.1: Schedule of the first Semester

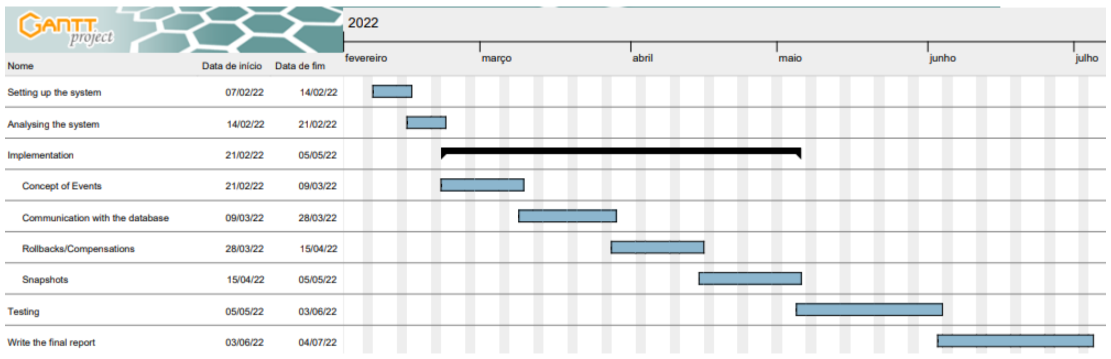
To decide when the meetings should occur, we used an Agile [2] methodology, more precisely SCRUM [3]. In particular, every week a meeting would be done with the intern and with the three supervisors. In every meeting, it would be discussed what was done in the last week, any doubts the intern might have, and what was supposed to be done next week. Additionally to this meeting, another weekly meeting would be done with the ALB team. In these meetings, the details of the project would be discussed. For example, what concepts were supposed to be used, or the requirements of the project.

As time progressed, some changes to the requirements of the project were done. This affected the planning by having the necessity to revisit the requirements and change them. Nevertheless, there were no major adjustments to the planning proposal.

2.2 Second Semester

Afterwards the intermediate presentation, the plan was to setup and do an analysis of the system, this would take two weeks. After this initial step, the implementation of the solution would start. This was planned to occupy three months. Succeeding the end of the implementation, testing was done. It was planned to take one month. Finally, after both the implementation and testing were finished, the intern would dedicate the rest of the time, roughly one month, to the elaboration of the final report.

To help visualize what was said above, a Gantt chart is shown in Figure 2.2.



Nome	Data de início	Data de fim
Setting up the system	07/02/22	14/02/22
Analysing the system	14/02/22	21/02/22
Implementation	21/02/22	05/05/22
Concept of Events	21/02/22	09/03/22
Communication with the database	09/03/22	28/03/22
Rollbacks/Compensations	28/03/22	15/04/22
Snapshots	15/04/22	05/05/22
Testing	05/05/22	03/06/22
Write the final report	03/06/22	04/07/22

Figure 2.2: Schedule of the second Semester

In the second semester, as well as in the first, weekly meetings were done. As said in the section above, this follows SCRUM [3]-like rules. More precisely, there were weekly meetings with the supervisors and with the ALB team. The purpose of these meetings was to discuss what was done in the week before, see if it was done correctly and what changes needed to be done, if any, and plan for the next week.

Finally, like with every project, there are hazards that could prevent the plan from being carried out. To counteract this, a risk analysis was carried out, which is detailed in the next section.

2.3 Risk Assessment

This section details some of the dangers that could jeopardize the curricular internship's success.

Some potential risks that were found in the first semester:

- **Collaboration with external groups may cause developmental delays.** This project is involved with an external entity, namely, ALB. Since the intern will apply the Event Sourcing (ES) concepts in a prototype provided by ALB, any delays with the prototype will delay the implementation. A possible mitigation plan is for the intern to create a smaller system while the prototype is not delivered, in which the intern might be able to test possible implementations.
- **Delays in access to infrastructures.** Due to the necessity of hardware to allocate this service, it is expected that these will be provided by the Department of Informatics Engineering (DEI). The implementation will be delayed if the system is not set up properly, correctly, and on time. Free tier infrastructures could be employed as a mitigating strategy.
- **Changes in requirements.** As there are going to be a great number of meetings with the stakeholders to discuss the project's state it is entirely possible that there may be changes. These changes will, naturally, create a delay in the life-cycle of the project. A feasible answer is to switch from the existing implementation method to a truly Agile [2] one, like SCRUM [3]. Following the SCRUM [3] method rules, the intern would develop and present prototypes with more frequency in an attempt to get more feedback.

2.4 Threshold of Success

This section details all the conditions that must be met in the internship for it to be considered a success.

The following goals must be completed:

- The time period allotted for system development cannot exceed the time allotted for the curricular internship.
- All of the system features described in this document must be implemented, tested, and ready for deployment.
- The created system must meet the quality criteria defined in this document.

Chapter 3

State of the art

In this chapter, the research performed on the technologies and concepts related to the project will be described. It starts with section 3.1 which introduces the basic concept of CQRS, followed by ES and Saga's. These concepts were chosen because, in the case of CQRS, it has a strong connection to ES, this will be explained in the next section. In the case of Saga's, besides being a way to coordinate microservices, they are already implemented in the current version of the system. After these concepts are explained, a study and comparison of the technologies employed will be presented in section 3.2.

3.1 Concepts

In this section, there are descriptions of concepts that will be used in the curricular internship. More precisely, section 3.1.1 will be CQRS, section 3.1.2 will explore ES. Finally, in section 3.1.3 the topic of Saga's will be briefly introduced.

3.1.1 Command Query Responsibility Segregation

Command Query Responsibility Segregation (CQRS) is a simple pattern. Typically, we use the same database with the same data model for both consulting (reading) and updating (writing), and although this works well for traditional software architectures in more complex ones this might prove challenging.

CQRS tries to cater to more complex architectures where having the same data model brings more complexity than having a different one. In other words, in CQRS we have two different models, one for writing data and one for reading data. In these systems, there are two types of actions, commands that update data and queries that read data. These actions are asynchronous, they must be based on tasks instead of data and the query must never change the database. Regarding the data models, by having two different models it is possible to cater them for specific purposes.

Concerning the number and size of the database, they can be catered for their needs. In terms of numbers, more reading databases can exist if there is an advantage, for example, if the database needs to be close to the system applications. In some extreme cases, the two databases are even physically separated to boost system performance. The same can be said about the size, it is very common for the reading database to be bigger than the writing one as reading operations occur on a much larger scale than updating operations.

In order to have synchronicity between the reading and writing databases, the simplest solution is to create an event that updates the reading model whenever there is a writing operation.

As expected, having this approach which differs from the standard CRUD [4] approach brings both benefits and drawbacks that will be enumerated in the next section.

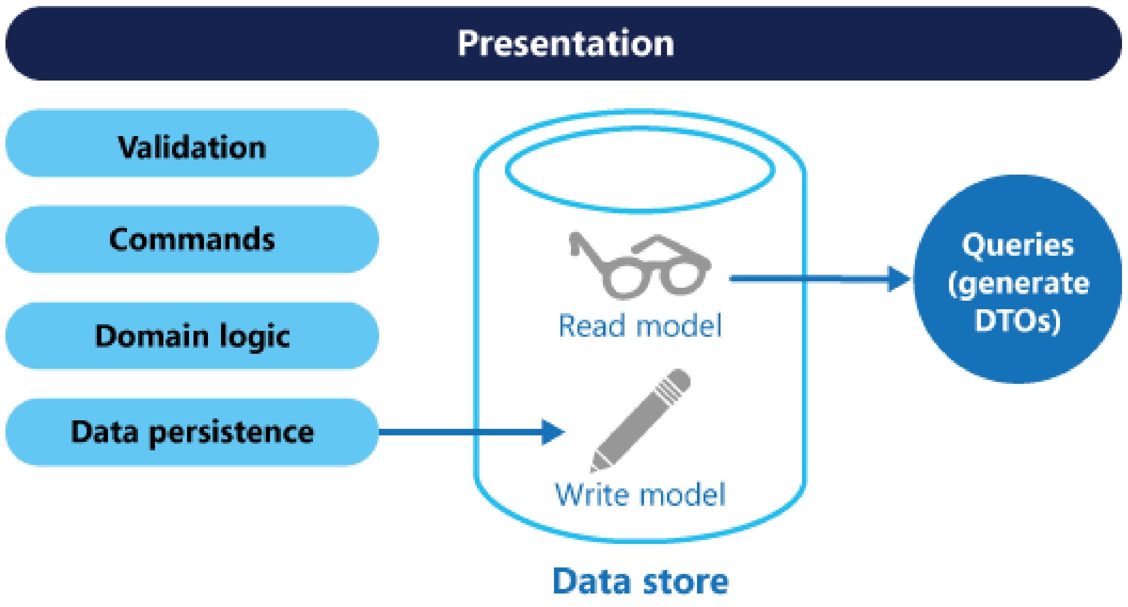


Figure 3.1: CQRS example [21]

Advantages

- **Scalability and Optimization:** Since the reading and writing actions are independent, it is possible to personalize the size of the database and its model for different needs.
- **Security:** It is easier to guarantee that the writing database is only being altered by certain entities.
- **Simpler Consultation:** We can save time by retaining the material in a reading model rather than a model for both reading and writing.

Disadvantages

- **Complexity:** Due to having two models instead of one the system may become more complex.
- **Messaging:** Although CQRS does not usually need a messaging service, if there is a need for one additional attention is required for duplicated messages or even error messages.
- **Consistency:** As a result of having a single model for reading the information it may become obsolete without a constant updating method.

When to Use

To conclude, there are situations where the use of CQRS is more appropriate. These are when the **System has a lot of user traffic**, when the **Number of information writing and reading differs greatly**, and in **Big Systems where there are multiple developers teams**.

3.1.2 Event Sourcing

Event Sourcing (ES) is an emerging way to persist data and process operations that has been gaining more and more popularity in recent times. In some systems, when there is a change of state, the system will process the requests and make necessary changes to the databases if needed. But with ES there are more steps to this process. When the system performs an action, an event is created. An event is an atomic, immutable, simple object that represents a single action in the system and has the necessary data to perform the action if there is a need to. Also, it will not influence the database keeping the system information. These events are kept and will be processed only at the right time. Additionally, they are specifically made for their corresponding system and only have meaning for specialists in that domain. Some event examples can be seen in Figure 3.2 .

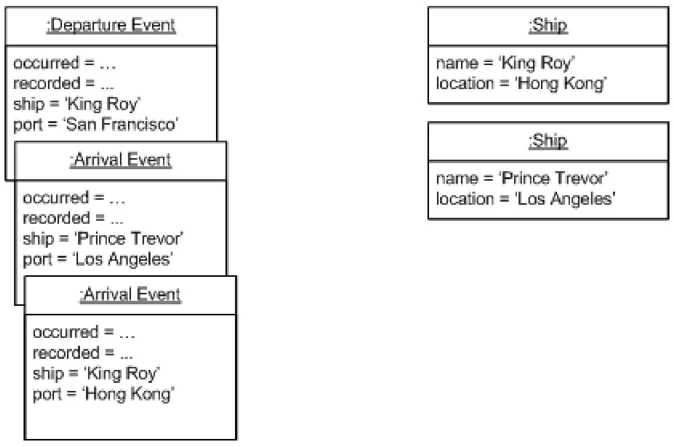


Figure 3.2: Event example [13]

Due to the nature of these events, they are not kept in the same database as other information. More concretely, in Event Source systems there are normally databases specialized in safeguarding the events created by the system and databases specialized in safeguarding data in a more standard way (CQRS, Section 3.1.1).

In ES systems (Figure 3.3), we can also find an Event Handler. This Event Handler has the logic behind the system handling, that is to say, that all of the operations needed for the events are there. Some of the operations are the creation of events, the connection to the database, and, fetching and processing events. These handlers can be of two types, transaction scrips [14] and domain models [15]. One way to go is using a transaction script, but this is not always the more appropriate option as transaction scripts are the go-to for simple applications, while domain models are catered for systems that are more complex.

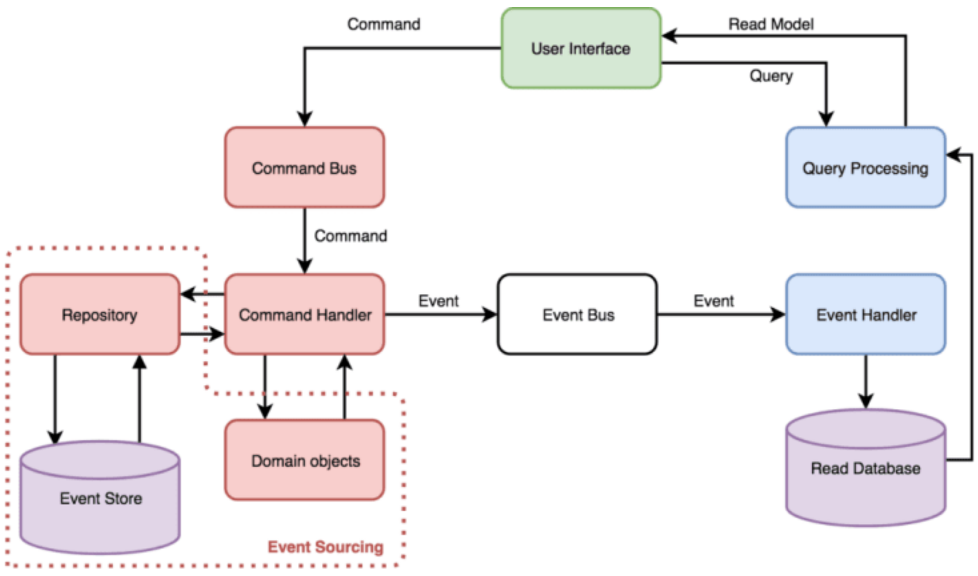


Figure 3.3: Event Sourcing system example [16]

With all this explained, there are some operations that can be done when the events are properly saved on the database.

Operations

All of the operations characteristics of ES revolve around the saved events in the log with the three main ones being [13]:

- **Complete Rebuild:** By reading all of the history of the events and performing the corresponding actions we can get to the current state of the application on an empty shell.
- **Temporal Query:** Similarly to the Complete Rebuild, it involves rerunning the events in the database, the main difference is that in the Temporal Query

the rerun is only until a certain timestamp or event. Sometimes it gets more complex depending on multiple time-lines, similarly to branches.

- **Event Replay:** If a past event was incorrect and there exists a need to rectify it we can go through the log in the reverse order, find the incorrect event, change it and run the new events and all that came after. Despite the fact that this is the standard method, it has drawbacks that will be discussed later. In the same way that we can rectify the events, it is also possible to change their order.

Advantages

With the knowledge of what is ES and the main Operations we can deduce a list of advantages:

- **Easy Implementation and Management:** Since the events are relatively simple and contain all the information needed to perform the action there is no need for them to actively change the database until the performed action is supposed to happen.
- **Analysis of the System:** With the way the events are saved, in particular by respecting the temporal order, and since they are of a simple nature, we can pinpoint certain things. For example, the behavior of the system, bugs/errors that may happen, and user tendencies.
- **Performance and Scalability:** Due to the nature of events they are immutable and need a simple operation of addition (no update or delete) to be saved. This permits that while the system creates and adds events to the database in the background, the process that originated them continues.
- **Easy Understanding:** Comparatively to a database, reading an event log is much easier due to the low complexity of it.

Disadvantages

Although there are advantages with ES there are also disadvantages that come with it:

- **Difficulty in updating the format:** Since the event information is permanent, we cannot update the events. Consequently, if there is any need to change the system information we need to add a compensation event to every event in the log. Or in a worse case, we might need to change the format of every event.
- **Difficulty with threads:** Temporal Consistency in the log is everything. Knowing this, the use of threads brings an issue. They may try to access a resource at the same time, which will bring inconsistency to the events.

- **No standard approach with databases:** Currently, there are no standard ways of reading events. This may cause the developer to expend more time than if a standard approach to saving information was used.
- **Event Flux:** Depending on the number of events in the log, replaying through all of them can take a long time. While doing snapshots of the current state of the system may help to attenuate this. It is expected that every operation that requires seeing past events to take some time.

When to Use

To conclude, as a result of all the advantages and disadvantages of ES there are several scenarios that are more appropriate for their use. The three identified scenarios [13] are **When it is necessary to create an audit log**, **When there is a need to restore the system to a previous state**, and **When there is a need to do compensatory actions**.

Relation with CQRS

CQRS is commonly used with ES due to their natures. More precisely, when using these two techniques together the event log serves as the default writing model in CQRS. Consequently, when the reading model needs to be updated, we can run the event log to get a representation of the current state. Additionally, due to the nature of events as described in section 3.1.2, the scalability and performance will be further maximized.

3.1.3 Saga

Using an architecture with microservices is becoming the norm for large-scale systems, however, there is a problem with how to handle transactions that span multiple services in a consistent way.

A transaction represents a single operation performed by a user. It contains the change of state of an entity and all the necessary information for that change of state to happen (for example, the time of day, location, etc...). They follow the ACID [22] properties. These are atomicity, consistency, isolation, and durability. While they have these characteristics this alone does not bring data consistency. It is also necessary to manage the transactions.

The Saga pattern fixes that issue. Using a sequence of local transactions (an atomic action performed by a single user of a Saga), each local transaction updates the information in the database of a single service and serves as the trigger for the next local transaction to happen. That is to say, a transaction in the microservice A will trigger the next transaction in the microservice B, and so on until the last location transaction is done, as can be seen in Figure 3.4. However, if a single local transaction fails all the previous actions are undone and a compensatory action will occur.

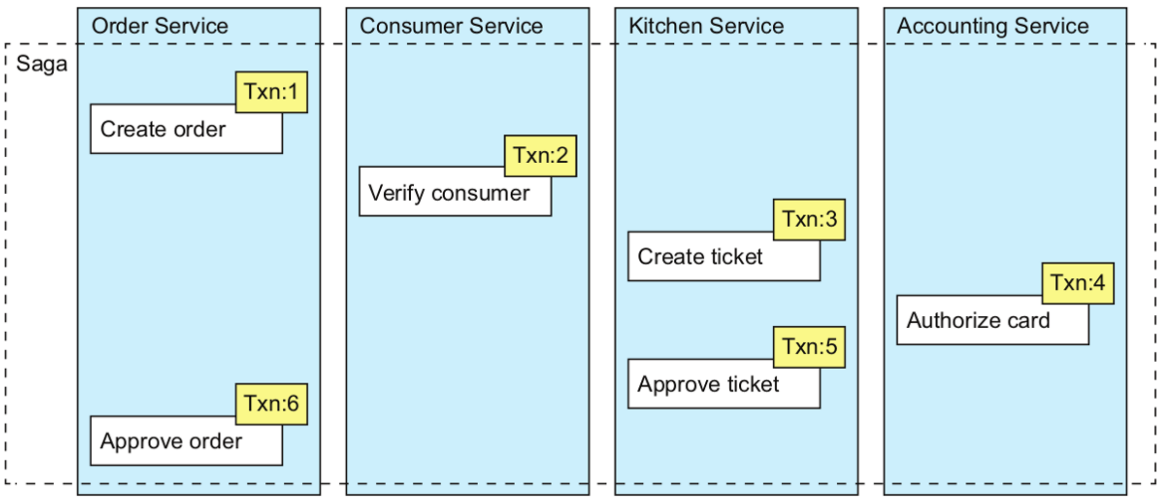


Figure 3.4: Saga example

Local transactions are divided into three types:

- **Pivot:** The first transaction, if it is applied the Saga must run until the end.
- **Retryable:** The transaction following the Pivot and that always succeeds.
- **Compensatory:** Reversible transactions that have the opposite effect of a past transaction.

There are two types of Sagas that are more commonly used, Choreographed, and Orchestrated. They differ in the approach to the transaction coordination, and both have some flaws and benefits, Table 3.1 and Table 3.2, which will be explained in the next subsection.

Orchestrated

In orchestrated Saga's there exists a centralized object (orchestrator) that tells all the users what transactions to execute based on certain events. For example if the microservice A behaves in a specific manner the orchestrator will order the other microservices to behave in a specific way. Additionally, the orchestrator is also responsible for compensatory transactions.

Orchestration pros and cons	
Pros	Cons
Useful when control over the transactions is needed	Reliance on an orchestrator
Good for complex workflows	Makes the workflow more complex
Not reliant on a single point	
Does not have a reliance on other users (no cyclic dependencies)	

Table 3.1: Pros and Cons of *Orchestration*

Choreographed

In choreographed Saga's all the user's transactions trigger the next transaction in other services, more precisely, a point of control does not exist.

Choreography pros and cons	
Pros	Cons
Does not require additional implementation	Difficult testing
Not reliant on a single point	Reliance on other users (cyclic dependency)
Good for simple workflows	Complex workflow due to dependencies

Table 3.2: Pros and Cons of *Choreography*

3.2 Technologies

In this section, research will be presented, more precisely, a brief introduction to the advantages of frameworks, if they are needed for the project, and also a study of databases that might be used, including a brief description of each one. After this what technology will be used in the project will be decided.

3.2.1 Support Tools

Due to ES being a relatively new approach to handle data there are not a great number of support tools available to use. Nevertheless, there exist some strong options that will be presented shortly.

Event Store

Event Store is a database designed exclusively to keep events. Developed by Event Store Ltd, it is available in an opensource version and a paid one.

Event Store supports three connection protocols, gRPC [23], Transmission Control Protocol (TCP), and Hypertext Transfer Protocol (HTTP), this latter one being the less used of the three due to performance issues, and thus will not be explored in great detail. The clients supported by this database vary depending on the type of protocol used. .NET, Java, Node.js, Go, and Rust are officially supported when using gRPC, furthermore, there are community made clients for Ruby and Elixir. When using a TCP connection, the supported clients are .NET, JVM client, Haskell, furthermore there exist community made clients for Node.js, Elixir, Java 8, Go, and PHP.

As was already said, with Event Store we can store events in a database. These events are then saved in streams that can be accessed and modified with costume commands such as *$all* (returns all the events for a respective stream) and *$maxcount* (specifies the number of seconds an event can live for). Furthermore, support for technical issues is also available, although it is necessary to pay, and ranges from an answer in the *Next business day response* to *2 or 8 hours response*.

Axon Framework

Axon Framework is an opensource framework used to implement ES and CQRS. This tool supports the Java programming language and enables the connection to their server *Axon Server*. Using the framework the developer will be able to focus on the high-level functionalities as the base work is already done.

What differentiates Axon from other tools available on the market is the explicit and extensive documentation [24], recurrent updates, and while normal ES permits the user to check how the system is behaving through time. Axon also has a Dispatch Interceptor and Handle Interceptor that permits the system to intercept a message before it is dispatched and while it is being handled. On the other side, to use this framework the Axon Server is also needed. Furthermore, while they have the basic concept of an event, the concept of a compensation event is not implemented.

No support tools

Although there are numerous advantages to the utilization of support tools they are not a guarantee of success.

Greg Young [9], an expert on CQRS and ES dislikes its use when coupled with the topics. In two of his talks [10] [11], he justifies this by saying that they bring unnecessary dependencies and complexity to the systems, or in his words, they bring "magic". This "magic" is the capability of the support tool to do certain tasks, and in a failure situation, the developer might not understand how the task is made, making it much harder to fix. In addition, the system is already reliant on the support tool so its removal can be time consuming.

Decision

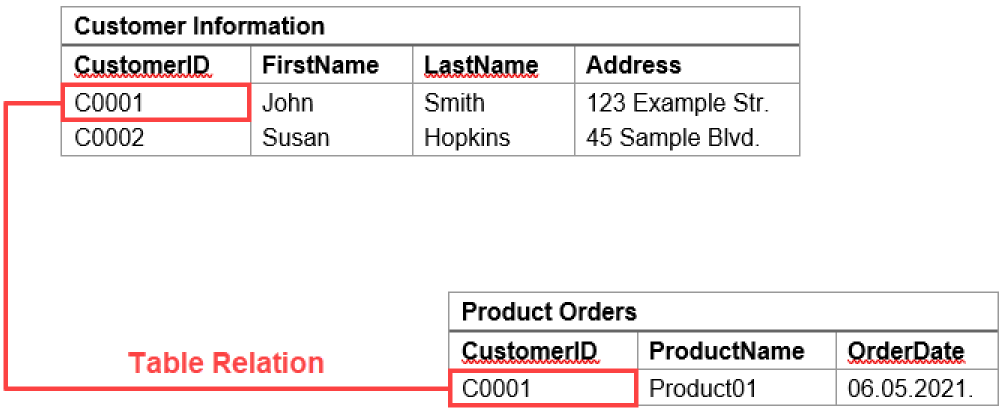
While a support tool may be beneficial, the cons are bigger than the pros. If one were to be used the dependencies and complexity of the system would go up unnecessarily. As such, an ES support tool will not be used.

3.2.2 Databases

MySQL

MySQL is an opensource SQL relational database management system (RDBMS) [8] developed in 1995 and later being acquired by *Oracle* in 2010.

Being an RDBMS MySQL saves the information in different tables, let's take the example of a customer and its products orders. Obviously, they have different data and are at different tables, but to connect them, a relation is built. That is to say, the respective primary key of the customer (normally a unique number given to him) is also present as a foreign key (Figure 3.5) in the product order table. This gives MySQL a high degree of flexibility and maintenance since data updates need to affect only one table and are easily controlled by developers.



Customer Information

CustomerID	FirstName	LastName	Address
C0001	John	Smith	123 Example Str.
C0002	Susan	Hopkins	45 Sample Blvd.

Product Orders

CustomerID	ProductName	OrderDate
C0001	Product01	06.05.2021

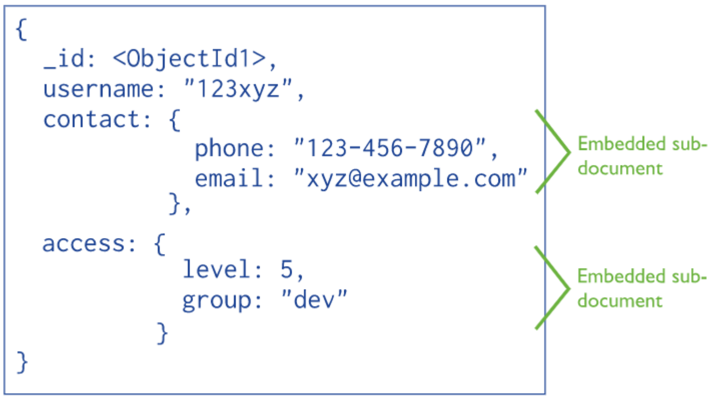
Figure 3.5: Foreign key example [20]

When MySQL was implemented it was not designed with the concept of a data center in mind, instead, it is a single-noded system that must rely on sharding. However, these solutions are manual and add to the complexity, lowering the performance and having a negative impact on the overall scalability of the system.

MongoDB

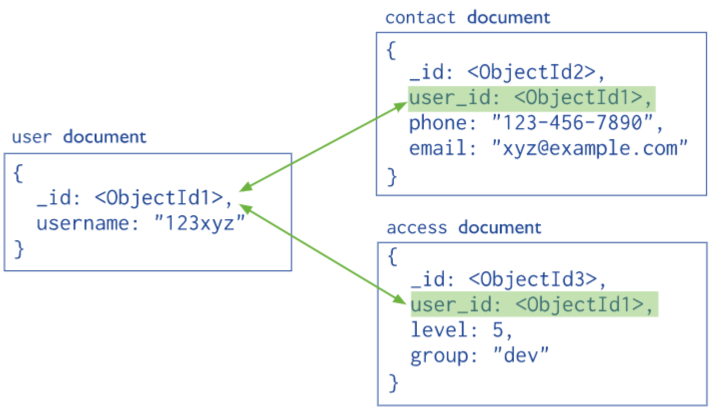
MongoDB is an opensource NoSQL database developed by MongoDB Inc.. It supports various data formats, this happens because in MongoDB each record is saved in Binary JSON (BSON) [5] that later can be retrieved in a JavaScript Object Notation (JSON) [6] format. Enabling the flexibility of data formats and are capable of having arrays, documents, and arrays of documents saved.

In MongoDB there are two types of structures [19], *Embedded Data* in which a type of data can be stored inside a different data type (Figure 3.6), and, *References* in which there is a reference data table with the relation between two different data sets (Figure 3.7).



```
{
   _id: <ObjectId1>,
   username: "123xyz",
   contact: {
               phone: "123-456-7890",
               email: "xyz@example.com"
            },
   access: {
              level: 5,
              group: "dev"
           }
}
```

Figure 3.6: Embedded data in MongoDB [19]



user document

```
{
   _id: <ObjectId1>,
   username: "123xyz"
}
```

contact document

```
{
   _id: <ObjectId2>,
   user_id: <ObjectId1>,
   phone: "123-456-7890",
   email: "xyz@example.com"
}
```

access document

```
{
   _id: <ObjectId3>,
   user_id: <ObjectId1>,
   level: 5,
   group: "dev"
}
```

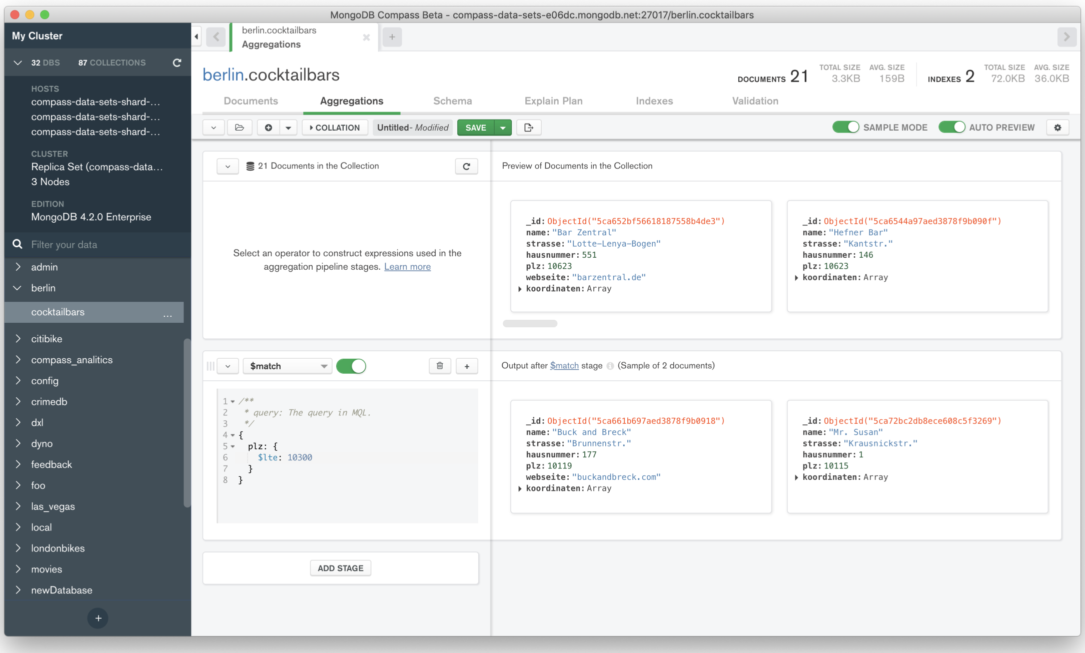
Figure 3.7: Reference table in MongoDB [19]

They use the concept of sharding [7]. Which is to say that the data is distributed in various machines to speed up the deployment of very large information sets. Thus increasing scalability and having a safety net in case of error.

Regarding data availability, there exists one master node with some slave nodes, in the eventuality that the master node goes down one of the slave's nodes will take its place, however, this can take a while.

For their user usage, the most common utilizations come through the command line, or through the MongoDB Graphical User Interface (GUI), *Compass* (Figure 3.8), in which the user can visualize the schema, create queries, and validate BSON Schemas.

However, there exists some negatives, the size file is limited to 16 Mb which can be a hindrance in higher density data. In addition, with the wrong indexing, the speed will go down drastically, and managing the indexes can be time consuming. Finally, since there are no join functions that exist in some SQL databases the users have to do it manually.



```
/**
 * query: The query in MQL.
 */
{
  plz: {
    $lte: 10300
  }
}
```

```
_id: ObjectId("5ca652bf56618187558b4de3")
name: "Bar Zentral"
strasse: "Lotte-Lenya-Bogen"
hausnummer: 551
plz: 10623
webseite: "barzentral.de"
koordinaten: Array
```

```
_id: ObjectId("5ca6544a97aed3878f9b090f")
name: "Hefner Bar"
strasse: "Kantstr."
hausnummer: 146
plz: 10623
koordinaten: Array
```

```
_id: ObjectId("5ca661b697aed3878f9b0918")
name: "Buck and Breck"
strasse: "Brunnenstr."
hausnummer: 177
plz: 10119
webseite: "buckandbreck.com"
koordinaten: Array
```

```
_id: ObjectId("5ca72bc2db8ece608c5f3269")
name: "Mr. Susan"
strasse: "Krausnickstr."
hausnummer: 1
plz: 10115
koordinaten: Array
```

Figure 3.8: MongoDB Compass [36]

Cassandra

Cassandra is an opensource NoSQL database released in 2008 and developed by *Apache Software Foundation*.

It is similar to MongoDB in the way that there are no foreign keys and instead a reference data table is used.

Regarding the data format, the developers made a new query language called Cassandra Query Language (CQL), it is similar to SQL, making the queries used also similar, reducing the learning curve.

In terms of data accessibility, Cassandra has various master nodes inside of a cluster, due to this, when a master node goes down one is always available to take its place and by doing this there exists practically no downtime.

Decision

Due to its scalability, the type of database that will be used is MongoDB. Additionally, since the data is saved in BSON files, there is a high degree of flexibility that comes with it.

3.2.3 Final Remarks

In conclusion, although using a *Support Tool* made for ES could be helpful in reducing the workload for the developer since some features would be implemented automatically, the negatives would have more impact, namely, the scalability and dependencies of the system would be increased. Knowing this the choice made is to use a normal database to save the events created.

Regarding the choice of databases, the chosen one is MongoDB. This database will permit the system to be scalable and to have a high degree of flexibility in the creation of events.

Chapter 4

Architectural Drivers

This chapter contains information regarding the current state of the system and its Architectural Drivers, that is to say, the requirements that have an impact on the architecture, such as the functional requirements, the technical and business restrictions, and the quality attributes.

4.1 Current System

As refereed before, in this project the intern will implement ES in an already built system made by Altice Labs (ALB), as such this section will provide a description of the system. This system is composed of ten microservices that will be described in a latter part of this thesis.

4.1.1 Overall Architecture

This system's purpose is charging. More precisely, it allows the connected users to request debit (removing money from their account) or credit (adding money to the account) operations and even creating/making reservations (reversing money to be used in a future operation).

There are four main event types that will be used in this internship:

- **Create**: This operation creates a new account in the system.
- **Delete**: This operation deletes an account.
- **Topup**: This operation adds balance to an account.
- **Voice**: This operation removes balance from an account (this balance cannot go below 0).

As it is possible to see in Figure 4.1 the system is composed of ten distinct microservices all with distinct functionalities that produce different types of information. Of the ten that can be observed, there are two stateful [31] services, these are the *Record Management Service* and the *Balance Management Service*. These two are connected to a MongoDB database, as it will be necessary to fetch information for the system use.

Regarding the communication between microservices, it is mostly done using a Kafka stream and a consumer producer model (that is to say, some microservices produce information for the others to consume)[12]. The communication not done using a Kafka stream is done using an HTTP communication protocol. The only communication done using an HTTP communication protocol is between the *Eligibility Service* and the *Rule Execution Service*.

For the choice of programming language and database, JAVA and MongoDB were chosen with the main objective of scalability.

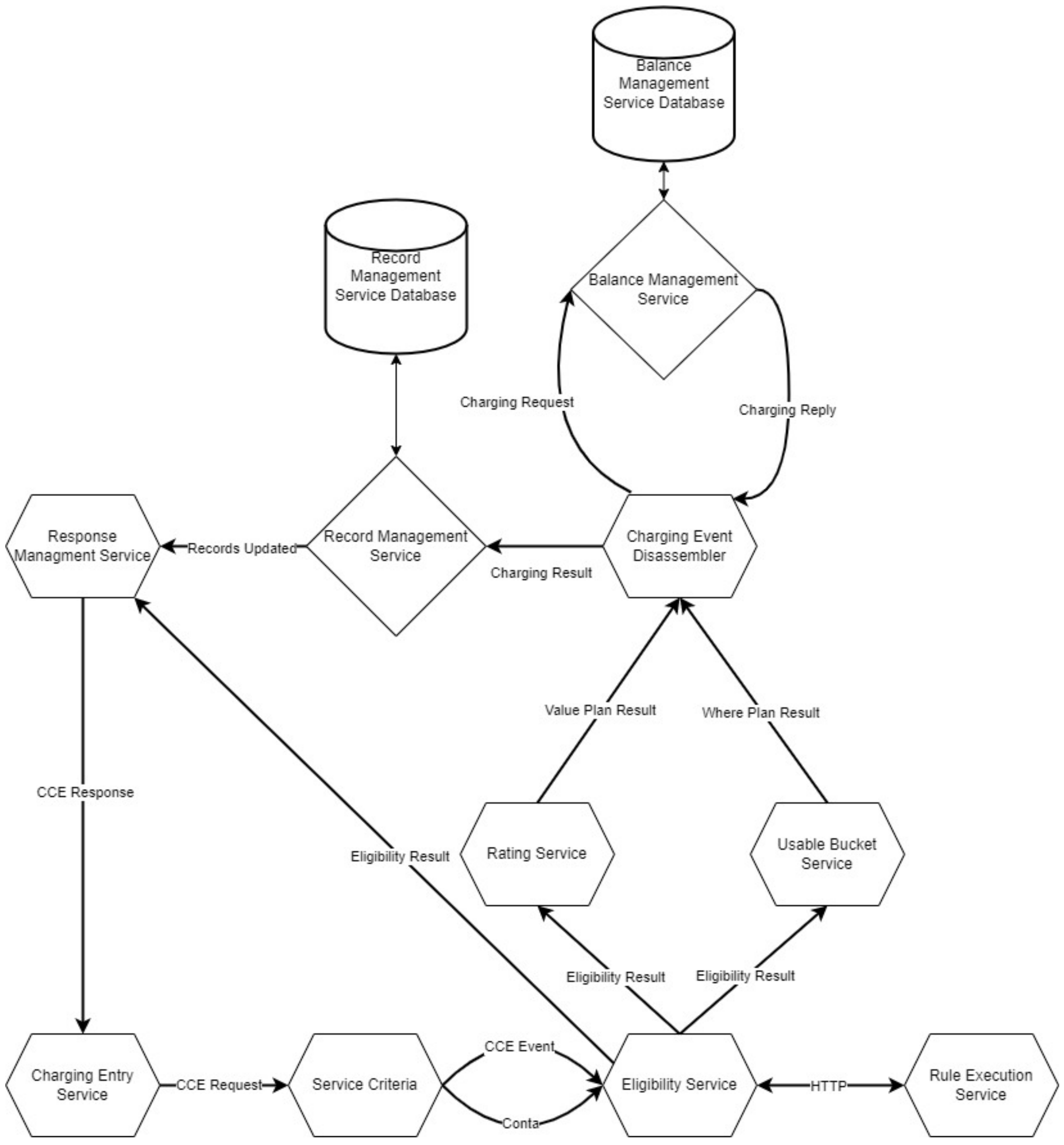


Figure 4.1: Diagram of the System

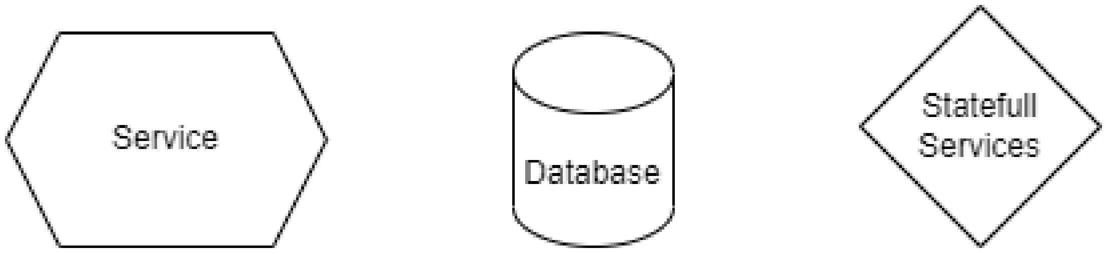


Figure 4.2: Legend for the System Diagram

4.1.2 Microservices functionalities

In this section, the functionalities of each microservice will be explained (Table A.3, Table A.4), and each data type created by the microservices will be presented with a brief description (Table A.1, Table A.2), and finally a description of the databases used (Table A.5).

4.2 Requirements

In this section, we can find the requirements needed for this project. They serve as a way to better understand what will be implemented in the system, the conditions necessary and, alternative scenarios. In this case, all the requirements are related to ES.

REQ01 - Creating an Event	
Level	Sea
Actor	Microservice
Objective	While the microservice performs the needed actions, events are created and stored in the database
Preconditions	• Connection to the database
Postconditions	• The created event is stored in the database
Main scenario	1. An action arrives at the actor 2. While the action is being performed an event is created 3. The created event is stored in the database after the action is processed
Alternative scenario	3a. There is no access to the database 3a.1. The system will notify the programmer and will try to establish a connection

Table 4.1: REQ01 - Creating an Event

REQ02_01 - Creating a Compensation Event	
Level	Sea
Actor	Microservice
Objective	The microservice creates the compensatory event and stores it in the database
Preconditions	• Connection to the database • The microservice needs to rectify a previous action
Postconditions	• The created events are stored in the database
Main scenario	1. An action arrives at the actor 2. While the action is being performed an event is created 3. The event is kept in cache 4. The process of the action ends 5. A compensation event is created through the event kept in cache 6. The original event and compensation are stored in the database
Alternative scenario	6a. There is no access to the database 6a.1. The system will notify the programmer and will try to establish a connection

Table 4.2: REQ02_01 - Creating a Compensation Event

REQ02_02 - Deleting an event	
Level	Sea
Actor	Microservice
Objective	The microservice deletes the event
Preconditions	• Connection to the database • The microservice rectifies a previous action
Postconditions	• The event is deleted
Main scenario	1. An action arrives at the actor 2. While the action is being performed an event is created 3. The created event is stored in the database 4. After the action is nullified, the event is deleted
Alternative scenario	3a. There is no access to the database 3a.1. The system will notify the programmer and will try to establish a connection 4a. There is no access to the database 4a.1. The system will notify the programmer and will try to establish a connection

Table 4.3: REQ02_02 - Deleting an event

REQ03 - Automatically creating a snapshot	
Level	Sea
Actor	Microservice
Objective	The microservice creates a snapshot of the event database
Preconditions	• Connection to the database
Postconditions	• The created snapshot is stored
Main scenario	1. The system checks if the conditions for creating a snapshot are met (for example, the time of day) 2. The system checks all the accounts ids 3. For each id a folder is created 4. For each id a snapshot is created 5. The snapshot is saved in the respective folder 6. The event database is cleared
Alternative scenario	1a. The condition is not valid 1a.1 The actor keeps processing actions 4a. The snapshot is not created 4a.1. The system will input the result on a log

Table 4.4: REQ03 - Automatically creating a snapshot

REQ04 - Changing the conditions for the snapshot	
Level	Cloud
Actor	Programmer
Objective	The condition for the snapshot condition must be changed and applied
Preconditions	• There exists a condition
Postconditions	• The condition is changed
Main scenario	1. The actor changes the condition 2. The change is applied
Alternative scenario	1a. The condition is not valid 1a.1. The system will display an error message notifying the actor

Table 4.5: REQ04 - Changing the conditions for the snapshot

REQ05_01 - Reprocessing events by id	
Level	Cloud
Actor	Administrator
Objective	The events are reprocessed
Preconditions	• A snapshot of the microservice event database exists • There exists an event log • There exists a "blank" microservice • There exists a connection to the database
Postconditions	• All the events are reprocessed
Main scenario	1. The actor chooses the id of the account that will be recuperated 2. The system will access the snapshot with the correct id 3. The events of the snapshots are restored updating the database 4. The actor chooses to reprocess events in the microservice
Alternative scenario	2a. A snapshot matching the correct id does not exist 2a.1. The system will display an error message notifying the actor 4a. An error occurs while processing the events 4a.1. The system will notify the programmer and the microservice stops processing the events

Table 4.6: REQ05_01 - Reprocessing events by id

REQ05_02 - Reprocessing events by date	
Level	Cloud
Actor	Administrator
Objective	The events are reprocessed
Preconditions	• A snapshot of the microservice event database exists • There exists an event log • There exists a "blank" microservice • There exists a connection to the database
Postconditions	• All the events are reprocessed
Main scenario	1. The actor chooses the date 2. The system will access the snapshot matching that date 3. The account information present in the snapshot will be restored by updating the database 4. All the snapshots regarding that account that were made after the chosen date will be accessed 5. The events of the snapshots are restored updating the database 6. The actor chooses to reprocess events in the microservice
Alternative scenario	2a. A snapshot matching the correct date does not exist 2a.1. The system will display an error message notifying the actor 4a. A snapshot matching the correct id does not exist 4a.1. The system will display an error message notifying the actor 6a. An error occurs while processing the events 6a.1. The system will notify the programmer and the microservice stops processing the events

Table 4.7: REQ05_02 - Reprocessing events by date

In these Requirements, we have two tasks that have two ways of being done, namely REQ02 and REQ05. In the case of REQ02, this happened due to the meetings with the ALB team that occurred in the first semester. In the case of REQ05, this was done to have a more complete system.

4.2.1 Requirement02

In those meetings, one of the topics discussed was how to tackle the compensatory events that were needed for this system. We reached two main approaches, REQ02_01 which involves creating a compensatory event that is the opposite of the event that needs compensation, and REQ02_02, which is deleting the event that is rollbacked. Both approaches have consequences that influence the system.

In the case of REQ02_01, since there does not exist a need to backtrack the event log the microservice is always available to process new events, reducing the risk of a bottleneck and having no need to dedicate time to reprocessing events. On the other hand, since we had a new event to the event log its size will grow which can be a downside, as a bigger event log impacts negatively the creation of snapshots (REQ04) and the creation of a copy of the system (REQ05_01 and REQ05_02) due to the need to process more events.

Regarding REQ02_02, we also have no need to reprocess the events. Furthermore, since we delete an event the event log size will not be affected. On the other hand, since the event was deleted, it is not possible for the developers to see what happened in the system.

One thing that must be pointed out is the impact of the event log on REQ03. Depending on its size it can influence it positively or negatively. This happens because a greater number of events will change the time needed to process them all.

In conclusion and due to all of these nuances, both of the methods will be applied and compared in section 6.2.2. The superior method will be applied in the final version of the system.

4.2.2 Requirement05

For REQ05, two ways were created to give the administrator more options. While doing the reprocessing by id is the standard and easier manner, the administrator might want to see the changes in the system as a whole. On the other hand, reprocessing by date it is a more lengthy method, and, it gives another viable and useful function to the administrator.

Contrary to REQ02, both options will be present in the final version of the system. This happens because one is not clearly better than the other, since they serve different purposes.

4.3 Restrictions

Restrictions are rules that the architecture must abide by, in other words, there does not exist flexibility regarding these rules (in some cases there exists a very low degree of flexibility). Pre-existing projects, internship decisions, and, deadlines are examples.

4.3.1 Technical Restrictions

Technical Restrictions are those that have more influence on the Architecture. Some examples of these, are programming languages, protocols, and technologies.

The Technical Restrictions identified for this project are:

- Certain concepts must be used
  - Description: In this project, ES, CQRS, and, Saga's must be used.
  - Flexibility Points: None.
  - Alternatives: None.

4.3.2 Business Restrictions

Business Restrictions are those that, while not having a direct influence on the Architecture can still have an impact on it.

The Business Restrictions identified for this project are:

- Development Time
  - Definition: The development time for this project must not exceed the duration of the internship.
  - Flexibility Points: The internship time can be extended.
  - Alternatives: None.

4.4 Quality Attributes

The non-functional needs of the system, often known as quality attributes, will be discussed in this section. Each quality attribute will have a brief description and their impact on the system will be categorized (H for High, M for Medium, and, L for low).

For the architecture of the system, the following quality qualities are the most important:

- **Scalability (H):** If the system is under an above average workload, if more resources are added to it, it is expected (under normal circumstances) an increase in its performance.

  - How to evaluate: When the system is complete, his performance will be evaluated in different working environments.

- **Portability (H):** If there is a need to create a new system (for example in REQ05), the programmer in charge of the creation must be able to do it without changing the source code.

  - How to evaluate: A test shall be made in which a copy of the system is created, and this copy will have to work identically to the original.

- **Performance (M):** Under normal circumstances, the database must be able to process at least one insert or one read per second.

  - How to evaluate: The capability of the database will be tested in different working environments.

Chapter 5

Architecture

This section describes the system architecture. To achieve the documentation, the C4 model was used and resulted in several images. Furthermore, the last section of this chapter is reserved for an analysis of the architecture. This is done to see if it complies with the requirements specified in Chapter 4.

5.1 C4 Diagram

5.1.1 System Context Diagram

In this section, we can find the highest level diagram created. This diagram exemplifies how an average user of the system interacts with it. As we can see in Figure 5.1, the only interaction is between the user and the system. The user can make various operations, more precisely, debit, credit, and, reservations. Besides this, there are no external systems that interact with the prototype. This is demonstrated in Figure 5.1 and with the associated legend, Figure 5.2.

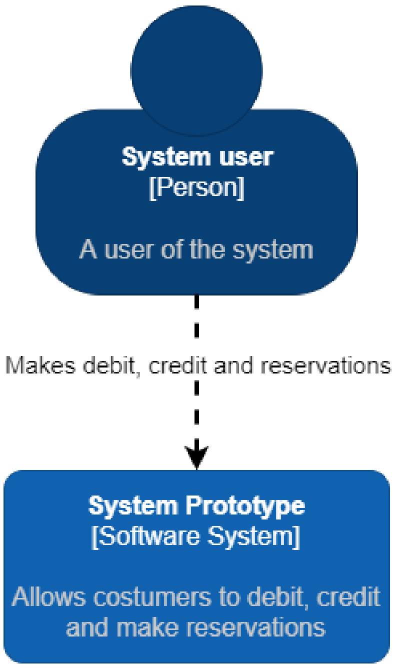

System user
[Person]
A user of the system
Makes debit, credit and reservations
System Prototype
[Software System]
Allows costumers to debit, credit and make reservations

Figure 5.1: System Context

Person name
[Person]
Description of person.
System name
[Software System]
Description of software system.

Figure 5.2: Legend for the System Context

5.1.2 Container Diagram

The next diagram is more detailed than the previous one. In this diagram, it is possible to see the various containers in a high level way. It also shows the choices regarding technologies and how the containers communicate. This is demonstrated in Figure 5.3 and with the associated legend, Figure 5.4.

As can be seen in Figure 5.3 there are three major choices for the color: blue, grey, and, red. They, respectively, symbolize the services that will not need change, the services that are going to change, and, the new infrastructures that will be added. Almost all of the communications between services are done using Kafka Streams (section 4.1.1). Regarding the flow within the system, it can be seen in Figure 5.3. The user requests will be processed by the various microservices. The most important ones are the *Balance Management Service*, and *Record Management Service* as these are the ones where ES will be applied.

In the *Balance Management Service* the requests are processed. To do this it will read and update the data present in the *Balance Management Service database (Reading Model)*. Additionally, some changes to this service will be made that will enable it to create Events and store them in the *Balance Management Service database (Writing Model)* for later use. After all of this, it will send a reply to the *Charging Event Disassembler*.

Regarding the *Record Management Service*, it is responsible for updating the values that represent the information about the customers. Like what happened in the above mentioned service, events are going to be created and because of this it will be necessary to introduce a new database for this service, in which all the events are going to be stored, this being the *Record Management Service database (Writing Model)*.

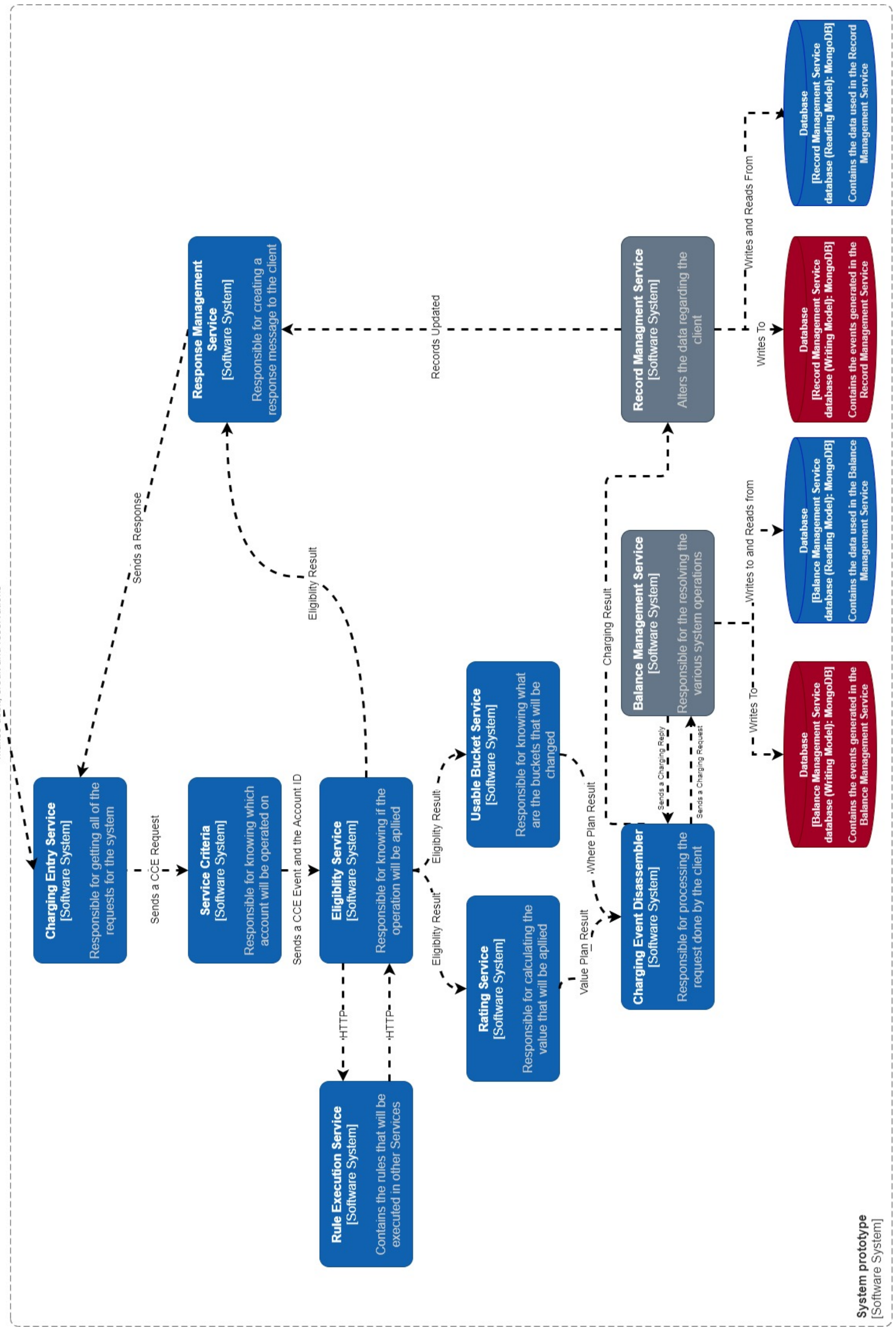

System user
[Person]
A user of the system
Makes debit, credit and reservations
Charging Entry Service
[Software System]
Responsible for getting all of the requests for the system
Sends a Response
Response Management Service
[Software System]
Responsible for creating a response message to the client
Sends a CCE Request
Service Criteria
[Software System]
Responsible for knowing which account will be operated on
Sends a CCE Event and the Account ID
Eligibility Service
[Software System]
Responsible for knowing if the operation will be apllied
Eligibility Result
HTTP
HTTP
Rule Execution Service
[Software System]
Contains the rules that will be executed in other Services
Eligiblity Result
Eligiblity Result
Usable Bucket Service
[Software System]
Responsible for knowing what are the buckets that will be changed
Rating Service
[Software System]
Responsible for calculating the value that will be apllied
Where Plan Result
Value Plan Result
Charging Event Disassembler
[Software System]
Responsible for processing the request done by the client
Charging Result
Sends a Charging Reply
Sends a Charging Request
Balance Management Service
[Software System]
Responsible for the resolving the various system operations
Records Updated
Record Managment Service
[Software System]
Alters the data regarding the client
Writes To
Writes and Reads From
Writes To
Writes to and Reads from
Database
[Record Management Service database (Reading Model): MongoDB]
Contains the data used in the Record Management Service
Database
[Record Management Service database (Writing Model): MongoDB]
Contains the events generated in the Record Management Service
Database
[Balance Management Service database (Reading Model): MongoDB]
Contains the data used in the Balance Management Service
Database
[Balance Management Service database (Writing Model): MongoDB]
Contains the events generated in the Balance Management Service
System prototype
[Software System]

Figure 5.3: Container Diagram

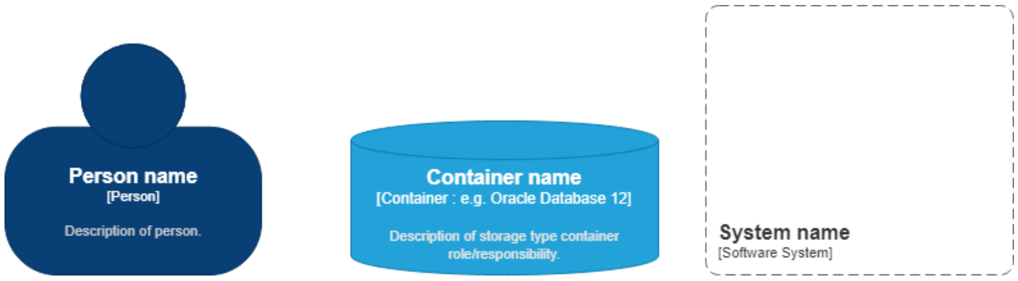

Person name
[Person]
Description of person.
Container name
[Container : e.g. Oracle Database 12]
Description of storage type container role/responsibility.
System name
[Software System]

Figure 5.4: Container Diagram Legend

5.2 Analysis

In this section, the architecture will be analysed to see if it complies with the Architectural Drivers in section 4.

- **Requirements** (section 4.2)
  - **REQ01**, **REQ02_01**, **REQ02_02** are all available in the *Balance Management Service* and *Record Management Service*. The events created are going to be kept in their respective databases, *Balance Management Service database (Writing Model)* and *Record Management Service database (Writing Model)*.
  - **REQ03**, **REQ04**, **REQ05_01**, and **REQ05_02** will be present in the machine hosting the system.
- **Restriction** (section 4.3)
  - In this project, the concepts of Event Sourcing, CQRS, and Saga's will be utilized.
- **Quality Attributes** (section 4.4)
  - **Scalability (H):** The chosen technology, *MongoDB*, has proven that it is scalable [18].
  - **Portability (H):** This is guaranteed by **REQ03** and **REQ05**, which respectively produce a snapshot and inserts it into a new environment.
  - **Performance (M):** The chosen technology, *MongoDB*, has proven that it is scalable [17].

Chapter 6

Implementation

This chapter describes the implementation of Event Sourcing (ES) on the system. At the start, the tools and practices used in the development will be explained. After that, there will be a brief description of how the system was deployed. Finally, the implementation done in the second semester will be explained. This is going to be subdivided into different sections.

6.1 Approach

This section is dedicated to the tools and procedures used during the internship to construct the system. It explains why a certain Integrated Development Environment (IDE) was chosen for the system implementation, how the Version Control System (VCS) was designed, how the data was analysed, and how the system was deployed.

6.1.1 Integrated Development Environment

An IDE is a piece of software that allows you to edit source code and may also include tools to assist you in the development of applications, such as a compiler or a debugger. Throughout this whole internship, the default IDE used was IntelliJ [33], by JetBrains. While this IDE can be more resource demanding for a machine, it has features that make this disadvantage worth it, namely a code completion system and a debugger. Furthermore, it is possible to integrate version control, and a database system while having a great number of plugins that the user can add. It is important to highlight that some of these functions are only available in the premium version. This was the version used in this internship, as JetBrains gives a special offer to students and teachers [34], allowing them to use their IDE for free.

6.1.2 Version Control System

To store and manage the source code for this project, a VCS was used. In a VCS, whenever someone is changed a commit is created. This allows the tracking of the project history. Furthermore, if needed the project state can be reverted to a previous commit or even restored. The VCS used in this project was GitHub [35], as in the current times it is the standard. This is due to GitHub having a good performance and extensive documentation. Regarding the repository, the intern created a personal repository dedicated entirely to this project.

6.1.3 Data analyses

To accurately and reliably analyze the results, MongoDB Compass [36] was used throughout the second semester. Although with IntelliJ [33] it is possible to integrate MongoDB this was not used as MongoDB Compass offers more features. As explained in section 3.2.2, with Compass we can visualize what happens in the database in real time, query the information, and check for outliers.

6.1.4 Deployment

This system is deployed on a single machine from the University of Coimbra (UC). In this section, the current deployment will be explained.

UC provided one machine to use in this internship. This machine has 4 CPU cores, 8GB of RAM, and 100 GB of storage. Currently, this machine contains all the scripts created throughout this internship, and can host all of the microservices of the system described in section 5. Despite this, for implementation purposes, some of them are not run on the machine.

Regarding the IP address, the current one is *10.17.0.159*. To access this machine the ssh protocol [37] was used. For this, a private and public key were created by the intern and copied to the machine. On top of that, OpenVPN was also used to access DEI internal network [38]. Furthermore, some microservices were deployed on the intern personal machine for implementation purposes. Since these microservices needed to communicate with the microservices inside the machine, some ports were opened to outside access, namely, port 22 (SSH access), port 27017 (MongoDB), and, port 9092 (Kafka).

To deploy this system, Docker [39] and Docker compose were used. With this method the various microservices were containerized and can be, built, deployed, ran, updated, and stooped in a fast and easy manner. Concerning Docker compose, this technology uses a YAML [40] file, that enables the specification of various factors in the deployment and configuration of the microservices (environment variables, open ports, etc...).

6.2 Development

In this section, the development in the second semester will be explained. The topics explored will be events and their compensation, how the snapshots are being created, and finally, how the reprocessing of events happens.

6.2.1 Event

The most important concept for this internship. As ALB already had created an event class to use in this system there was no need for the intern to create one. This event is then added to the database when the microservice finishes the processing. It is not added when the event arrives because there is always a possibility that the event is rollbacked (a different method was created for this), or it is invalid (the event would have to be removed).

Regarding the data contained in the event, it has:

- **_id**: an event token identifier.
- **consumedTimestamp**: an integer that represents the time when the request was consumed by the microservice.
- **createdTimestamp**: an integer that represents the time when the request was created.
- **entryTimestamp**: an integer that represents the time when the user made an operation to the system.
- **eventCreatedTimestamp**: an integer that represents the time when the event was created.
- **eventId**: a string identifier created by the microservice.
- **eventNumber**: an integer that represents the number of events processed by the microservice.
- **optimistic**: a boolean that defines if the event is always processed (cannot be rollbacked or is invalid).
- **origin**: a string identifier that represents the origin of the request.
- **payload**: This is an embedded data that contains:
  - **_t**: a string identifier that represents the model of the request.
  - **accountId**: a string that represents the id of the account that will have its balance changed.
  - **bucketId**: a string that represents which account bucket suffered an alteration.

  - **operation**: a string that represents the type of the event.
  - **reservation**: a boolean, this boolean is true if the operation decrements balance from the account, and is false if it does not.
  - **value**: an integer, this represents the value added/removed from the account.

- **readTimestamp**: an integer that represents when was the last time the event was read by the system.
- **requestId**: a string that represents the request id.
- **retryNumber**: an integer that represents the number of times that the system tried to process the event.
- **sagaId**: a string that represents the saga of which this event is a part.
- **sagaTimeout**: an integer that represents the time needed for the saga to timeout.
- **topicPartition**: a string that represents the topic of this event.

6.2.2 Compensation

In this section, the two ways of compensation are going to be explained, and in the end, there will be a brief conclusion.

Deleting the Event

To apply this method it was only needed to delete the event generated by the microservice when the processing ended. While this process is indeed faster and simpler it raises a problem. This being, while analyzing of the system it is as if the event never entered the system. This can change the analysis to something that is not true as it permanently deletes some requests made by users.

Compensation Event

This method is harder to implement. As always, we only add the compensation event when the processing for the original one has ended. In cases where the system processes a great number of events without interruptions, they would be kept in a list and added after they all were processed.

Regarding the content of the compensation event, it consists of the inverse that happens when processing a normal event. While we can change the type of event (for example, a adding balance event will create a remove balance event) this is not optimal as in some cases there can be events that do not have an inverse. What was done was changing the quantity added in events that change the balance. For instance, an event that added 5€ would be changed to adding -5€. But this does not work for events that create or delete accounts. To invert the events it was needed to change the type. For instance, a create account event would be changed to a remove account event. All this being said, the method to invert events depends heavily on the system.

Comparing the two approaches

As said before, these two methods affect the event log in a different manner. But knowing that just deleting the original event impacts negatively the analysis of the system (an ES major advantage), it was decided to keep the Compensation Event method on the system.

6.2.3 Creation of snapshots

Since the creation of snapshots is a periodic procedure, a tool called Cron [41] was used. With Cron we can schedule different operations to be performed periodically. In this case, a script that created the snapshots is running at midnight of each day. This script takes all the events in the database and organizes them in folders with a matching id and day. For example, all the events regarding the account with the *id 0* that happened on *10-06-2022* will be kept in the folder named *id0-10-06-2022*. Furthermore, the details of the account will also be saved for the reprocessing of events. After this process, the events are then deleted from the database to prevent event duplication in the snapshots and to control the size of the database.

It was done this way to better organize the events as with various accounts it can become hard to track events. Additionally, by keeping the date we can also retrieve events by date. In order to keep the events consistent, while the script is creating the snapshots and the folder the system cannot process events. Instead, they will be kept on cache and processed after all the snapshots are created. Furthermore, a log of this process is also created so the user can see if any problems occurred. Finally, to change the time at which the snapshots are created, it is only necessary to change the Cron configuration file.

6.2.4 Reprocessing Events

The reprocessing of events is a two-stage operation. To accomplish this a script was created that takes the events stored in the snapshots and copies them to the database. This script needs two arguments, an integer that says if the events are to be restored by id or date, and the id or date depending on the integer chosen. Since the script behaviour changes depending on the first argument, it works in two different ways:

- **By id**: The user inputs an id, for example the *id 0*, and all the events in folders that correspond to the account with *id 0* are copied to the database. If no folder corresponds to that id an error message will show up.

- **By date:** The user inputs a date, and if there is a folder with that date, all the events in that folder and folders with date past that will be copied to the database. Furthermore, since the date might not be the account creation date the account details on that date will also be copied. For example, the user inputs the date *10-06-2022*, all accounts from *10-06-2022* will be restored and the events from *10-06-2022* onward will be copied. If there is no folder corresponding to the date an error message will show up.

After this first part was done and the events are in the database the microservice will process them. This can be done when starting the microservice and inputting a number that corresponds to the reprocessing action. When the reprocessing is over the microservice will behave as normal.

6.2.5 Creation of the Event Sourcing Library

Overview

As stated before, a library of ES was also created using Maven. This library's focus is to tackle the most difficult aspects of ES not present in other libraries, namely the event reprocessing and the compensation actions. Although it only tackles these aspects it is possible to include more functions (for example an event creation if needed or the access to the database).

To implement it various interfaces[44] were created. With these interfaces we can call the methods with empty bodies. These methods are going to be used with the main class of the library. With the interfaces created the next thing needed were the handlers. In the handlers we can find the initializations of methods that the main class needs. This detail is important, it is not the methods contained in the interface, instead the methods belong to the main class.

Regarding the main class in itself (named processor), it contains an Instance so we can call and use it in other projects, it also contains the interfaces created. Lastly, we can also find the functions that are going to be used by the users.

The functions present in this library work through callbacks. For example, the *processRollbackEvent* that is used for the compensation events first checks through the *sagaId* of the event if there is no event present with that respective Id. If no event is found, it adds the original event and then uses a function named *converEventToRollbackEvent* to convert the event and then add it. This last function must be in the client system and can be changed to suit its needs. Both the aspect of event reprocessing and the compensation actions work this way.

Finally, the installation and usage of this library. It can be installed by copying and pasting it in the *Maven* directory of the machine or IntelliJ can be used as it allows the installation of Maven projects directly. Regarding the usage, the user needs only to initialize the processor and then use the functions contained by it (for example processor.function).

Class Description

In this section we can find a description of all the classes from the library, including their functions, arguments used for said functions and purpose.

ReprocessConverter - Interface		
Function	Arguments	Purpose
processEvent	• Event event • int choice	Enables the use of the function

Table 6.1: ReprocessConverter functions and purposes

ReprocessHandler - Class		
Function	Arguments	Purpose
register	• ReprocessConverter reprocessConverter	Sets the interface ReprocessConverter so it can be later used using a function from the Processor

Table 6.2: ReprocessHandler functions and purposes

RollbackConverter - Interface		
Function	Arguments	Purpose
convertEventToRollbackEvent	• Event event	Enables the use of the function

Table 6.3: RollbackConverter functions and purposes

RollbackHandler - Class		
Function	Arguments	Purpose
register	• RollbackConverter rollbackConverter	Sets the interface Rollback-Converter so it can be later used using a function from the Processor

Table 6.4: RollbackHandler functions and purposes

Processor - Class		
Function	Arguments	Purpose
getInstance	• None	Sets the instance of the class so it can be used
setRollbackConverter	• Rollbackconverter rollbackConverter	Sets the RollbackConverter so it can be used
setReprocessConverter	• Reprocessconverter reprocessConverter	Sets the ReprocessCon-verter so it can be used
processRollbackEvent	• MongoDatasource eventUC • Event event	Adds the original event and the compensation event to the database
reprocessEvent	• None	Reprocess all the events contained by the database

Table 6.5: Processor functions and purposes

Chapter 7

Testing

Tests were done on the system to ensure compliance with the architectural drivers specified in Chapter 4. Unit tests are described in section 7.1. Following that, section 7.2 has the compliance of the system to the quality attributes from section 4.4. Finally, the last section is dedicated to scalability tests.

7.1 Unit tests

As the development of the different modules proceeded, unit tests were created. All of the tests were done by following a structure that could change depending on the request. Requests were created that will be processed by the system. After this, the results will be analysed.

7.1.1 Test structure

The tests followed this structure:

- **Setup of the information and resources** (e.g., before we add an amount of money to the account balance, we need to create that account).
- **Choose the parameters for the request** (e.g., choose what kind of request the system would process, the account id, etc...).
- **Perform the request**.
- **Check if the account information is correct** (e.g., check if the right account was created/deleted, and check the balance).
- **Check if the events corresponded to the action performed**.

7.1.2 Strategy

Every resource was examined using this method as a general rule:

- **Test the processing of events:** Test whether the user can send requests to the system and they are processed in the right way.
- **Test the compensation events:** Test whether the user can send requests and after the rollback the account didn't suffer any change. Additionally, it is expected that a compensation event remains in the database.
- **Test the snapshot creation:** Test whether the system created a snapshot and respective folders as intended.
- **Test event reprocessing:** After deleting the user account, reprocess the events. In the end, the user account needs to match the state before the deletion.

7.1.3 Test Suite

The complete test suit is composed of 35 tests. These tests can be done individually or as a group. The full scope of tests can be seen in table 7.1 and table 7.2.

Test description and results	
Test	Result
valid_processing_1type_1event	Passed
valid_processing_1type_multiple	Passed
valid_processing_multiple_multiple	Passed
invalid_processing_1type_1event	Passed
invalid_processing_1type_multiple	Passed
invalid_processing_multiple_1event	Passed
invalid_processing_multiple_multiple	Passed
valid_rollback_1type_1event	Passed
valid_rollback_1type_multiple	Passed
valid_rollback_multiple_1event	Passed
valid_rollback_multiple_multiple	Passed
invalid_rollback_1type_1event	Passed
invalid_rollback_1type_multiple	Passed
invalid_rollback_1type_1event	Passed
invalid_rollback_multiple_multiple	Passed
valid_rprocessID_1account_1event	Passed
valid_rprocessID_1account_multiple	Passed
valid_rprocessID_multiple_multiple	Passed
invalid_rprocessID_1account_1event	Passed
invalid_rprocessID_1account_multiple	Passed
invalid_rprocessID_multiple_multiple	Passed

Table 7.1: Test and their results

Test description and results-continuation	
Test	Result
valid_rprocessD_1account_1event	Passed
valid_rprocessD_1account_multiple	Passed
valid_rprocessD_multiple_multiple	Passed
invalid_rprocessD_1account_1event	Passed
invalid_rprocessD_1account_multiple	Passed
invalid_rprocessD_multiple_multiple	Passed
valid_snapshot_1type_1account	Passed
valid_snapshot_multiple_1account	Passed
valid_snapshot_1type_multiple	Passed
valid_snapshot_multiple_multiple	Passed
invalid_snapshot_1type_1account	Passed
invalid_snapshot_multiple_1account	Passed
invalid_snapshot_1type_multiple	Passed
invalid_snapshot_multiple_multiple	Passed

Table 7.2: Test and their results-continuation

7.1.4 Result

Currently, all the unit tests pass, indicating that no unexpected behavior has been discovered. This does not imply that the code is bug-free. It just implies that the present tests are unable to find any bugs.

In this current state, these tests cover all of the functional requirements (REQ01 to REQ05). Additionally, they also cover the quality attribute of portability as it guarantees that the event reprocessing is working as intended.

7.2 Quality attribute verification

This section examines how well the system adheres to the quality attributes from section 4.4.

- **Scalability**: The system does not fully comply with this quality attribute and should be upgraded in the future. Tests were performed on the system modules most prone to suffer from scalability problems. These tests can be found in the next section, each one including a description, the results, and a discussion.

- **Portability**: To comply with this quality attribute two things were done. First Docker was used and this allows the microservices to deploy in different machines with minimal configuration and in a fast manner. Secondly, REQ_05 permits the user to transfer all the data from the system users to a copy of the system through events.
- **Performance**: The system complies with this attribute using MongoDB. This database guarantees a fast read and write of documents. Additionally, *mongostat* [42] was used and this showed the compliance of the system. Furthermore, in the next section there will be information regarding some of the scripts created.

7.3 Scalability Tests

In this section, the scalability tests results will be shown. These tests were performed on the system parts most prone to be affected by large workloads, namely the normal processing of events, the event rollback, and the script to create the snapshots. Furthermore, in the case of the event rollback tests were done for the two methods (deletion of the event and the compensation event). Each test is described in the subsections below, including the tools used, the many scenarios, the findings, and the conclusions.

7.3.1 Tools used

No additional tools were needed to perform the tests, this happened for two reasons. Firstly, since the system provided a counter for the average time spent on a request no additional tool was used. Secondly, the system permits the easy simulation of various payloads since it permits the specification of various metrics. Namely, the number of requests, type, the time between the requests, warm up requests (requests that would happen before the real payload), etc... . Furthermore, for script testing purposes it is only needed to use the *time* command when using the script to see the completion time (for example, time script.sh). Finally, each and every one of these tests were performed while the system was working in the DEI host machine.

7.3.2 Event processing

The core of this internship, this module is prone to scalability issues as whenever a user makes a request to the system an event is created and processed. Consequently, various tests were done that barrage the system with more time demanding requests.

To maintain consistency, all the requests made for the system were ones that would increase the balance and decrease the balance with zero time between requests. This was made because the accounts that would create and delete accounts sometimes created accounts with the same id and this would change the results. Furthermore, each workload was performed ten times to perform the average preventing outliers.

Four scenarios were created to test the processing of events:

- **Scenario 1**: 10 requests
- **Scenario 2**: 20 requests
- **Scenario 3**: 40 requests
- **Scenario 4**: 80 requests

Results

These are the results of the scenarios mentioned above.

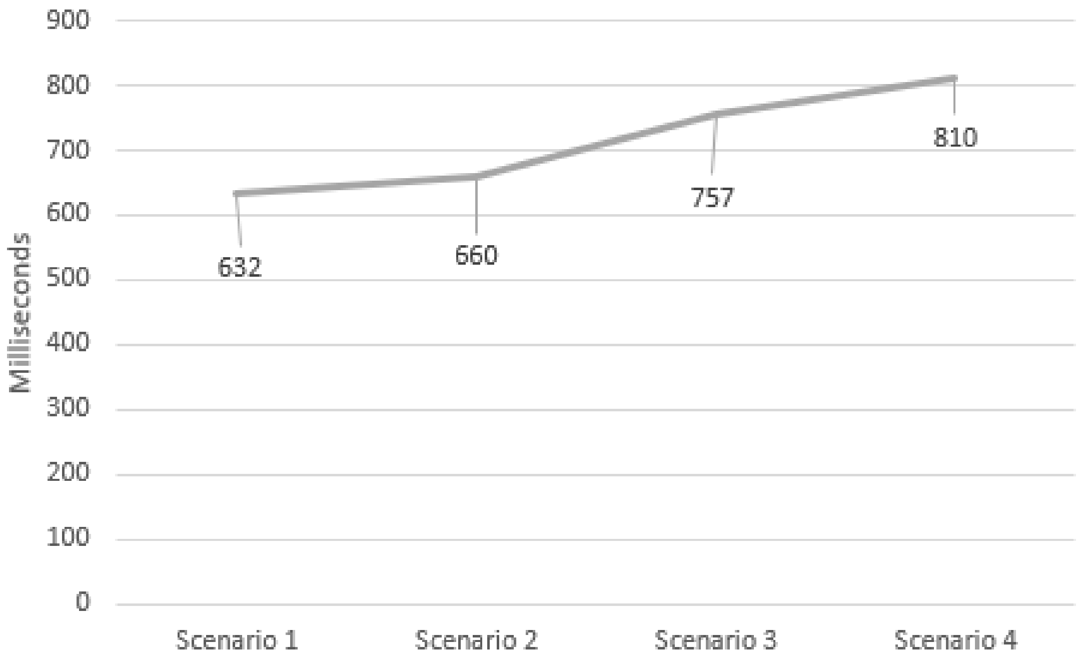


Figure 7.1: Average time spent to process requests in each scenario

Scenarios and results			
Scenario	Median	Max	Min
Scenario 1	638ms	659ms	579ms
Scenario 2	654ms	689ms	647ms
Scenario 3	752ms	791ms	733ms
Scenario 4	817ms	865ms	743ms

Table 7.3: Response time table for Event processing

Discussion

By analysing the data, it can be observed that the average response time is under one second up to eighty concurrent requests. This can be considered a low response time (above 1 second is considered a medium to high response time). Considering that this system is a prototype, this type of behaviour is acceptable as it shows that it can handle a large number of concurrent requests if no rollbacks are involved.

7.3.3 Event Rollback

The Event Rollback is also one of the most important aspect of this internship. Likewise the normal processing of events, the system can be bombarded with various requests that would result in a rollback. Furthermore, as happened in the previous section various scenarios were created to test this system. These scenarios follow the same rules, only requests that change the accounts balance, and with no time between requests. Additionally, since two methods (compensation event and event deletion) to rollback an event were introduced, both of them were tested.

The scenarios are:

- **Scenario 1**: 10 requests
- **Scenario 2**: 20 requests
- **Scenario 3**: 40 requests
- **Scenario 4**: 80 requests

Results for the event deletion

The following figure and tables show how much time on average was spent on the processing of requests with event deletion.

Scenarios and results			
Scenario	Median	Max	Min
Scenario 1	10327ms	10681ms	10155ms
Scenario 2	13804ms	14608ms	12678ms
Scenario 3	14582ms	15208ms	14158ms
Scenario 4	16378ms	16933ms	16291ms

Table 7.4: Response time table for Event Rollback with deletion

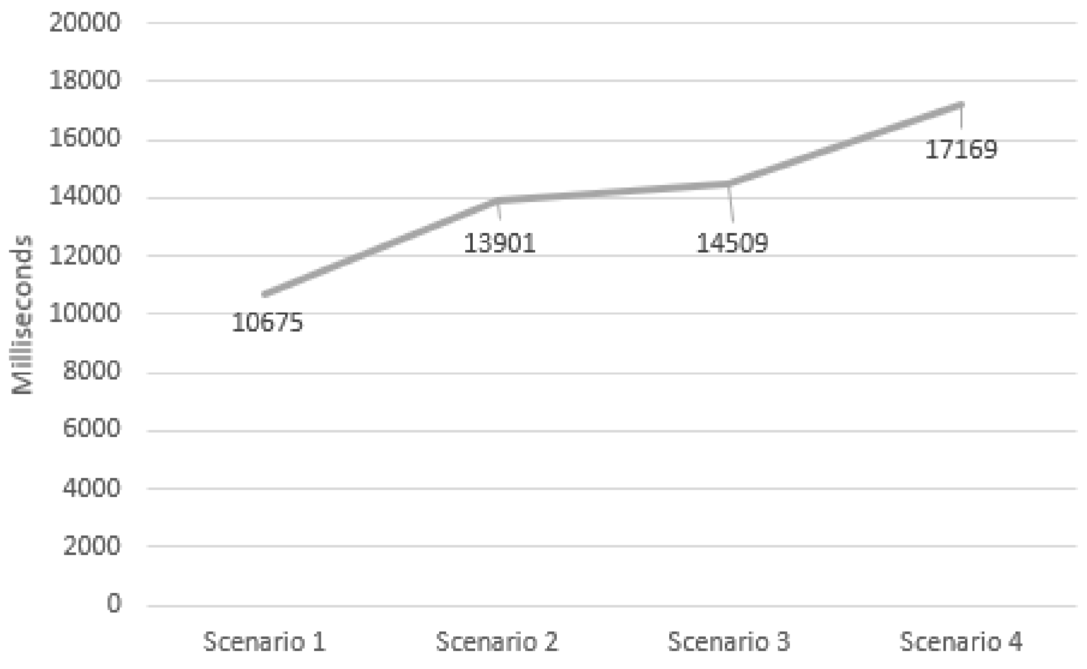


Figure 7.2: Average time spent in each scenario with event deletion

Result for the creation of a compensation event

The graphs and tables below illustrate how much time was spent on average processing requests with compensation events creation.

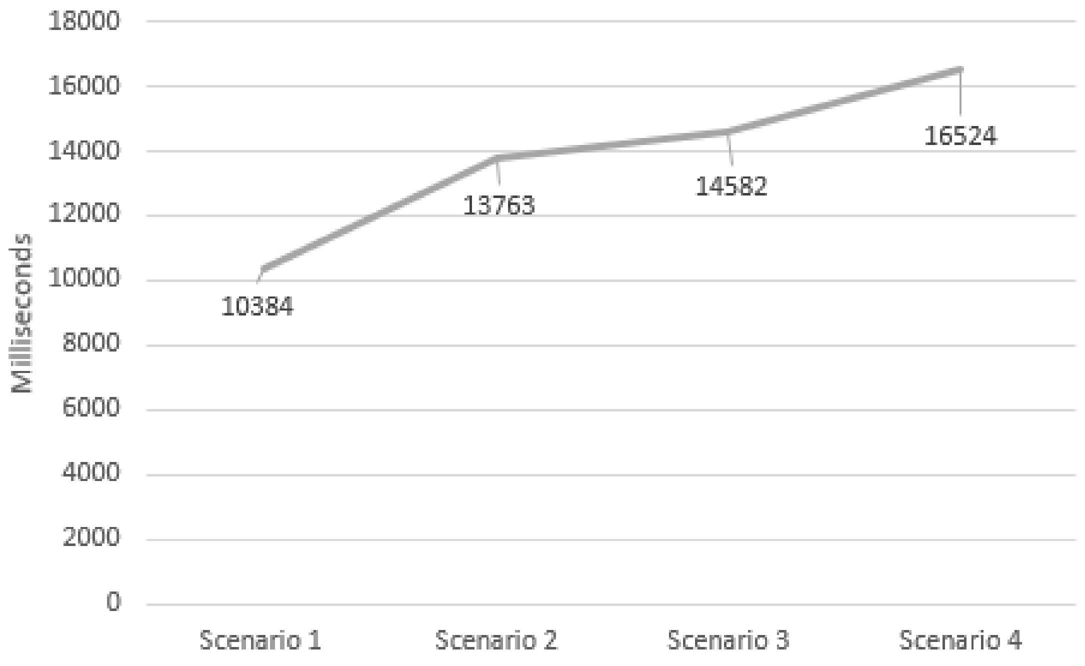


Figure 7.3: Average time spent in each scenario with compensation events creation

Scenarios and results			
Scenario	Median	Max	Min
Scenario 1	10312ms	11564ms	9851ms
Scenario 2	14018ms	15058ms	12503ms
Scenario 3	14524ms	15261ms	13813ms
Scenario 4	17333ms	18077ms	16164ms

Table 7.5: Response time table for Event Rollback with a Compensation Event

Discussion

Contrary to the expectations, the difference between the two methods is not great. The method that was supposed to take a greater time had smaller ones. This is most likely due to the randomness of the tests. Doing more tests would result in more accurate data and the difference in values would likely change. Summarizing, even though this part of the system could be optimized the time spent on the requests in each scenario is acceptable.

7.3.4 Snapshot creation

The final part of the system tested is the snapshot creation script. Since it trims the event log and is an essential part to rebuild the system it was tested. It needs to be able to handle a large number of accounts. The scenarios created will take into account the number of accounts while having a fixed number of events.

The scenarios are:

- **Scenario 1**: 5 accounts with 10 events each
- **Scenario 2**: 15 accounts with 10 events each
- **Scenario 3**: 40 accounts with 10 events each

Result for snapshot creation

The graphs and tables below illustrate how much time was spent on average creating snapshots requests.

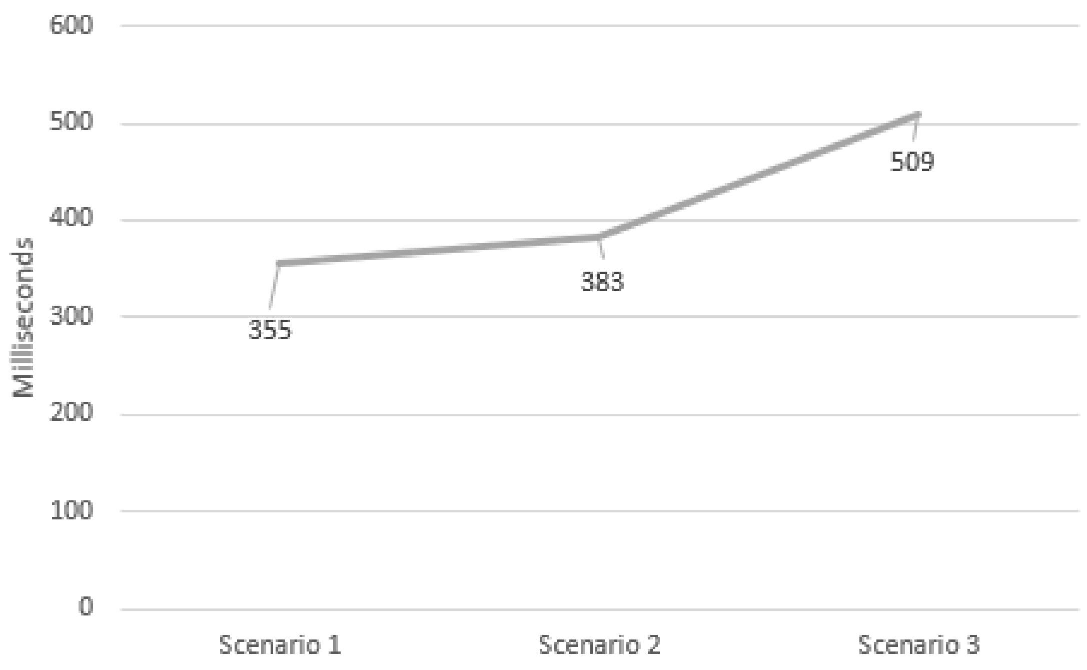


Figure 7.4: Average time for snapshot creation in each scenario

Scenarios and results			
Scenario	Median	Max	Min
Scenario 1	351ms	368ms	349ms
Scenario 2	383ms	419ms	368ms
Scenario 3	518ms	534ms	469ms

Table 7.6: Response time table for Snapshot creation

Results

As expected, the time spent by the script increased as the number of accounts increased. Knowing that, this system should host a great number of users this behaviour is acceptable. In conclusion, if some optimizations were done the system would be left in a better state. Nonetheless, because the results are satisfactory, it is a low-priority activity.

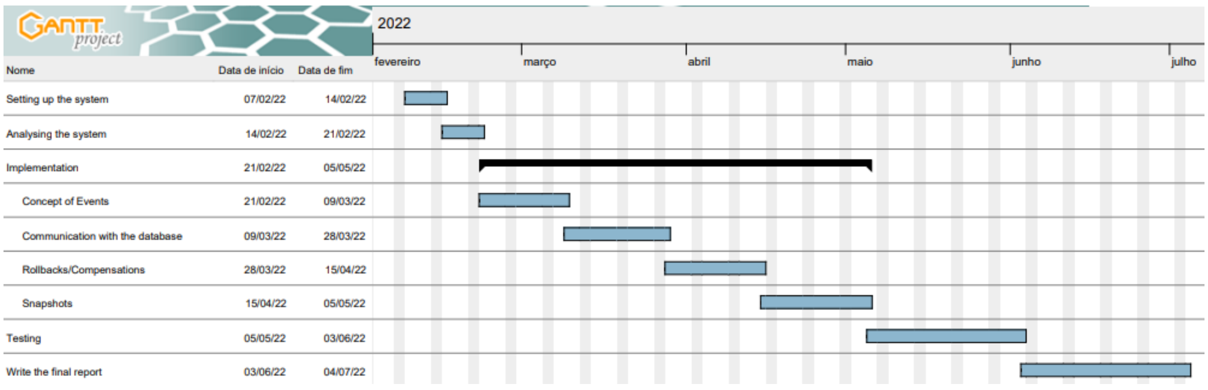
Chapter 8

Conclusion

In this section, we can find an analysis of this internship, more concretely, how it diverged from the original plan, what was developed, what could be done better, and what can be developed in the future. Some personal thoughts on this internship will also be present.

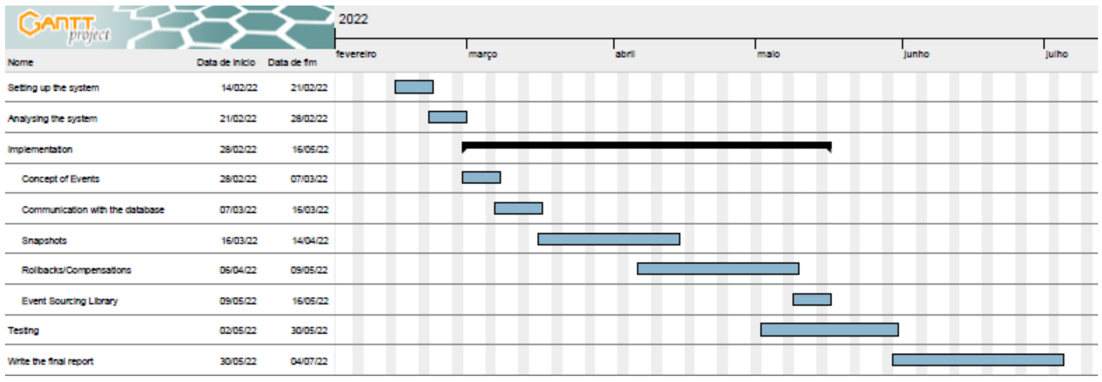
8.1 Planned vs Real schedule

In contrast to the original timetable shown in Figure 8.1, the planned schedule diverged from the actual schedule, as seen in Figure 8.2.



Nome	Data de início	Data de fim
Setting up the system	07/02/22	14/02/22
Analysing the system	14/02/22	21/02/22
Implementation	21/02/22	05/05/22
Concept of Events	21/02/22	09/03/22
Communication with the database	09/03/22	28/03/22
Rollbacks/Compensations	28/03/22	15/04/22
Snapshots	15/04/22	05/05/22
Testing	05/05/22	03/06/22
Write the final report	03/06/22	04/07/22

Figure 8.1: Original Schedule of the second Semester



Nome	Data de início	Data de fim
Setting up the system	14/02/22	21/02/22
Analysing the system	21/02/22	28/02/22
Implementation	28/02/22	16/05/22
Concept of Events	28/02/22	07/03/22
Communication with the database	07/03/22	16/03/22
Snapshots	16/03/22	14/04/22
Rollbacks/Compensations	06/04/22	09/05/22
Event Sourcing Library	09/05/22	16/05/22
Testing	02/05/22	30/05/22
Write the final report	30/05/22	04/07/22

Figure 8.2: Real Schedule of the second Semester

The schedule for the second semester generally followed the same order as the planned one. Firstly, regarding the system setup a slight delay occurred because ALB did not have the system prepared and some problems occurred with the infrastructures provided by DEI.

Secondly, the creation of events and connection to the database was shorter than expected. This happened because the system already had an event class built in and the connection to the database was easier than expected.

Thirdly, it is possible to see that the snapshots and the compensation events took more time than expected. The first timeline was overly optimistic in terms of the amount of time allotted for development as the system complexity and the inexperience of the intern had a negative impact. When creating a new timetable in the future, the intern should consider technologies that are new or about which he or she has little expertise. If this is the case, he should devote extra time to the anticipated issues.

Fourthly, the creation of the ES library was not planned at the beginning of this internship. Consequently, it was added to the schedule. It was made simultaneously with a part of the testing because this permitted the intern to focus on the library while the system was performing the tests.

Finally, some of the testing and the writing of the report were made simultaneously since some tests were made automatically and only the results needed to be written.

8.2 Difficulties

This section outlines some of the challenges that slowed the project's growth and could have been avoided if the intern had more expertise.

8.2.1 Setting up the system

Since some microservices had to be hosted on the intern machine they had to communicate with the microservices hosted in the DEI machine. To achieve this some ports needed to be opened to outside access (section 6.1.4). This was made through trial and error as the intern had no previous experience doing this. More concretely, after adding rules to the firewall that allowed outside access she was turned on. Furthermore, some changes were made to the configuration files of both MongoDB and Kafka.

8.2.2 Snapshots

Another part of this internship that took more time than expected were the snapshots. The original intention was to use Rsnapshot [43] to create periodic snapshots. Unfortunately, this technology has a disadvantage, it would create a snapshot of the entire database. In other words, it would not separate the events. Consequently, a new method using a script was created that would attend to the needs of this internship. Furthermore, separating the events by date was also an idea that came later in the lifecycle of the snapshot's implementation. As such, the script used had to be altered and retested. However, it was necessary time to make sure that this module would be more complete.

8.2.3 Rollbacks/Compensations

Definitely the part where the intern encountered the most difficulties. These difficulties originated due to the complexity of the system. Since various threads were used by the system it became difficult to guarantee that the events would only be added at the end of the process. Consequently, various versions of this process were created in the process to reach the final version. This will be less of a problem in future implementations of ES due to the library created in this internship.

8.3 Future Work

In this section, the work that might be done in the future will be described. Even though this internship is finished, the concept of ES is interesting and new experiences could be made with the system.

8.3.1 Documentation

Through the development of this system, no documentation was created. This choice was made to speed up the development, since the intern would not have to be concerned with documentation. It was acceptable because the intern was the only person in direct contact with the system and gained knowledge of the system as the internship progressed. However, if another individual decides to work with the system proper documentation is a must, as it lessens the learning curve.

8.3.2 Implementing the Axon Framework

A decision made during this internship was not to use any external tools to help the implementation of ES (section 3.2.2). Implementing ES and comparing it with the solution reached by the intern would be an interesting experiment. With this, we could compare the solution created in this internship with one already established in the market. As such, using the Axon Framework would be the logical choice as it is the most complete framework for ES.

8.3.3 Upgrading the library

As stated in section 6.2.5 an ES library was created. This library does not cover all of the aspects of ES, only the most troublesome ones. Knowing this some future work could focus on improving the library. For example, the transfer of the snapshots creation procedure from a script in the host machine to the library, or even the event replay functions. With this, a condensation of all the work done in this internship could be found in the library.

8.4 Final Thoughts

To conclude, I think this internship was done successfully. The functionalities were implemented and the system was left in a usable state. However, work can still be done in this system as stated in section 8.3.

In technical terms, implementing the concepts of ES and CQRS on a system with various microservices allowed me to gain knowledge on these concepts that might prove useful in the future. Furthermore, since these concepts were applied in this specific system it allowed me to understand some of the problems that I might encounter in the future. Moreover, it also helped to deepen my knowledge of databases, some of them I already had some kind of knowledge due to classes, but some of them were completely new to me and might be useful in the future. In addition to all this and although I knew what Docker was, I had no previous experience with it and this internship helped me gain basic knowledge of it

Professionally, it was a new experience but valuable experience to work on a project with multiple institutions and see that it requires a large amount of work to coordinate the efforts. Furthermore, in the first semester it was also necessary to work through the internet. This proved to be interesting, as it made me reevaluate my work ethics. Nevertheless, I know that this work experience will be valuable going forward.

Bibliography

[1] Two-phase commit protocol
https://www.geeksforgeeks.org/two-phase-commit-protocol-distributed-transaction-management/.
Consulted at 14/11/2021

[2] Agile
https://www.cprime.com/resources/what-is-agile-what-is-scrum/.
Consulted at 15/11/2021

[3] Scrum
https://www.scrum.org/resources/what-is-scrum.
Consulted at 15/11/2021

[4] CRUD
https://www.codecademy.com/articles/what-is-crud.
Consulted at 21/11/2021

[5] BSON
https://www.mongodb.com/json-and-bson.
Consulted at 27/11/2021

[6] JSON
https://www.json.org/json-en.html.
Consulted at 27/11/2021

[7] Sharding
https://docs.mongodb.com/manual/sharding/
Consulted at 28/11/2021

[8] RDBMS
https://searchdatamanagement.techtarget.com/definition/RDBMS-relational-database-management-system
Consulted at 28/11/2021

[9] Greg Young
https://2017.dddeurope.com/speakers/greg-young/
Consulted at 02/12/2021

[10] Greg Young - A Decade of DDD, CQRS, Event Sourcing
https://youtu.be/LDW0QWie21s
Consulted at 02/12/2021

[11] Greg Young - 8 lines of code
https://www.infoq.com/presentations/8-lines-code-refactoring/
Consulted at 02/12/2021

[12] Kafka Stream Introduction
https://kafka.apache.org/intro
Consulted at 05/12/2021

[13] Martin Fowler Event Sourcing
https://martinfowler.com/eaaDev/EventSourcing.html
Consulted at 10/12/2021

[14] Transaction Script
https://martinfowler.com/eaaCatalog/transactionScript.html
Consulted at 10/12/2021

[15] Domain Models
https://martinfowler.com/eaaCatalog/domainModel.html
Consulted at 11/12/2021

[16] Example of an Event Sourcing System
https://dev.to/wsantosdev/event-sourcing-parte-4-domain-events-2j7f
Consulted at 11/12/2021

[17] MongoDB Performance
https://www.mongodb.com/blog/post/high-performance-benchmarking-mongodb-and-nosql-systems
Consulted at 11/12/2021

[18] MongoDB Scalability
https://www.mongodb.com/basics/scaling
Consulted at 11/12/2021

[19] MongoDB data Modeling
https://docs.mongodb.com/manual/core/data-modeling-introduction/
Consulted at 11/12/2021

[20] Relation Database Example
https://phoenixnap.com/kb/what-is-a-relational-database
Consulted at 11/12/2021

[21] CQRS
https://docs.microsoft.com/pt-pt/azure/architecture/patterns/cqrs
Consulted at 11/12/2021

[22] ACID
https://www.ibm.com/docs/en/cics-ts/5.4?topic=processing-acid-properties-transactions
Consulted at 11/12/2021

[23] gRPC
https://grpc.io/
Consulted at 12/12/2021

[24] Axon documentation
https://docs.axoniq.io/reference-guide/axon-framework/introduction
Consulted at 15/12/2021

[25] Choreographed and Orchestrated Saga
https://microservices.io/patterns/data/saga.html
Consulted at 20/12/2021

[26] C4 Model
https://c4model.com/
Consulted at 08/01/2022

[27] Complexity of IT systems will be our undoing
https://www.networkworld.com/article/2193597/complexity-of-it-systems-will-be-our-undoing.html
Consulted at 14/01/2022

[28] Software Complexity Is Killing Us
https://www.simplethread.com/software-complexity-killing-us/
Consulted at 14/01/2022

[29] IT departments spend millions tackling performance issues in complex IT
https://www.computerweekly.com/news/252470861/IT-departments-spend-millions-tackling-performance-issues-in-complex-IT
Consulted at 14/01/2022

[30] 76 per cent of CIOs say IT complexity makes it impossible to manage performance
https://www.retaildive.com/news/76-of-cios-say-it-complexity-makes-it-impossible-to-manage-performance/516065/
Consulted at 14/01/2022

[31] Stateful
https://www.merriam-webster.com/dictionary/id
Consulted at 15/01/2022

[32] GanttProject
https://www.ganttproject.biz/.
Consulted at 03/06/2022

[33] IntelliJ
https://www.jetbrains.com/idea/
Consulted at 06/06/2022

[34] IntelliJ special offers
https://www.jetbrains.com/idea/buy/discounts?billing=yearly
Consulted at 06/06/2022

[35] GitHub
https://github.com/
Consulted at 07/06/2022

[36] MongoDB Compass
https://www.mongodb.com/products/compass
Consulted at 07/06/2022

[37] SSH
https://www.ssh.com/academy/ssh
Consulted at 07/06/2022

[38] OpenVPN DEI configuration instructions
https://helpdesk.dei.uc.pt/configuration-instructions/vpn-access/
Consulted at 07/06/2022

[39] Docker
https://www.docker.com/
Consulted at 08/06/2022

[40] YAML
https://yaml.org/
Consulted at 08/06/2022

[41] Cron
https://linux.die.net/man/5/crontab
Consulted at 10/06/2022

[42] Mongostat
https://www.mongodb.com/docs/database-tools/mongostat/
Consulted at 13/06/2022

[43] Rsnapshot
https://rsnapshot.org/
Consulted at 19/06/2022

[44] Java Interface
https://docs.oracle.com/javase/tutorial/java/concepts/interface.html
Consulted at 23/06/2022

Appendices

Appendix A

Overall Architecture

A.1 Data created by the system and their description

Data created by the system and their description (First Part)		
Data	Description	Contains
Records Updated	Details regarding counters that were updated and with which values did the update occur	• List of records for that account with the value
Eligibility Result	The result of checking if the request can be processed	• true/false
CCE Event	Request for credit, debit, or creating/removing of a reservation, this data will be changed as the systems processes the request.	• accountID • value • operation • isReservation
Account ID	The id of the account that will be changed	• account id

Table A.1: Data types and their descriptions (First Part)

Data created by the system and their description (Second Part)		
Data	Description	Contains
CCE Request	Request for credit, debit or creating/removing of a reservation in the base format, this data will be changed as the systems processes the request.	• accountID • value • operation • isReservation
Charging Request	Data created for each account that will have the balance changed.	• accountID • bucketID • value • operation • isReservation
Charging Reply	Answer to the above mentioned request.	• success
Charging Result	Details about the accounts and quantities that were credited or debited	• accountID • bucketID • value • operation • isReservation
Value Plan Result	Value that will be added or taken from the balance of the account	• value
Where Plan Result	The list of buckets that will be changed	• List of buckets

Table A.2: Data types and their descriptions (Second Part)

A.2 Microservices and their functionalities

Microservices and their functionalities (First Part)	
Charging Entry Service	• Receives all of the requests to the service • Publishes a CCE Request
Service Criteria	• Calculates which account will be affected by the operations • Sends a CCE Event and the Account ID
Eligibility Service	• Executes the rules on the Rule Execution Service and checks if the request can be done • Sends the appropriate response (true or false)
Rule Execution Service	• Executes the rules needed for the system operations
Rating Service	• Calculates the change in balance for the account • After this operation, it publishes the value
Usable Bucket Service	• Checks what are the buckets of the account that will be changed • After this operation it publishes the list of buckets

Table A.3: Microservices and their functionalities (First Part)

Microservices and their functionalities (Second Part)	
Microservice	Functionality
Charging Event Disassembler	• For each account balance, it publishes the Charging Request • Receives the result of the operation in the Charging Reply • After all the operations are complete it publishes a Charging Result
Balance Management	• Receives requests for credit, debit, and creation/removal of reservations • Executes the operation • Published the result of said operations in a Charging Reply
Record Management	• Will change the counter associated with the account in question depending on the system rules that can be found in another microservice • Updates the counters and publishes the results in a Records Updated
Response Management	• Depending on the eligibility of the account it will give a suitable answer to the user

Table A.4: Microservices and their functionalities (Second Part)

A.3 Databases and their data

Databases and their information	
Database	Contains
Record Management Service Database	A structure with: • **_id**: the object id of the account created by mongo • **_t**: a string identifier that represents the model of the account • **accountId**: the account id • **id**: the account id • **records**: the two last requests made from the account • **timestamp**: the date in which the last request was made
Balance Management Service Database	A structure with: • **_id**: the object id of the account created by mongo • **_t**: a string identifier that represents the model of the account • **balance**: the balance that the account has • **buckets**: list of numbers that when added will equal the balance • **id**: the account id

Table A.5: Databases and their data