UNIVERSIDADE D
COIMBRA

Gustavo Miguel Martins Leite

CREATION OF A REPLICATED EVENT SOURCING APPLICATION

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

September of 2023

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

DEPARTMENT OF INFORMATICS ENGINEERING

Gustavo Miguel Martins Leite

Creation of a Replicated Event Sourcing Application

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

September 2023

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

Gustavo Miguel Martins Leite

Criação de uma Aplicação de Event Sourcing Replicada

Dissertação no âmbito do Mestrado em Engenharia Informática, especialização em Engenharia de Software, orientada pelo Professor Doutor Filipe Araújo e apresentada ao Departamento de Engenharia Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra.

Setembro 2023

Acknowledgements

I would like to thank Professor Filipe Araújo for the guidance and his advice during this dissertation and my family and friends for the support on my time at the University of Coimbra.

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).

This work is funded by the FCT - Foundation for Science and Technology, I.P./MCTES through national funds (PIDDAC), within the scope of CISUC R&D Unit - UIDB/00326/2020 or project code UIDP/00326/2020

Abstract

Traditional data storage typically has the current state of data stored in a SQL or NoSQL database, and when a transaction occurs, that data is updated. The primary objective is to store the current state, and even though it works, it can be limiting if more information than the current values is needed. Information such as what transactions lead to the current values is lost. Even though it is possible to have historical registration and transaction logs in traditional data storage, it is a possibility that it can become a complex task to manage and use.

Event Sourcing is a different way of storing data where instead of storing the current state, it is the sequence of transactions that is stored. The current state is built by replaying the events, and no data is lost, which gives event sourcing robust audit and analytical capabilities.

This dissertation's objective is to tackle the challenge of taking a centralized Event Sourcing application and replicating it to make it more reliable and available across different regions. To achieve this goal, an application with a replicated event store had to be developed and placed on the internet to have the system distributed across multiple areas of the world.

As a result, this work presents the architecture of the created replicated system, along with the used technologies, including SpringBoot, Axon and MongoDB. In addition, this document also presents a performance analysis of the write and read operations speeds of the said application placed on the cloud. These results gathered from the performance testing proved encouraging for event-sourced geo-replicated systems, opening new ways for future applications.

Keywords

State. Event Sourcing

Resumo

O armazenamento tradicional de dados tem tipicamente o estado atual dos dados armazenados numa base de dados SQL ou NoSQL, e quando ocorre uma transação, esses dados são atualizados. O objetivo principal é armazenar o estado atual, e mesmo que funcione bem, pode ser limitativo se for necessária mais informação do que os valores atuais. A informação, tal como as transações que levam aos valores atuais, é perdida. Ainda que seja possível ter registos históricos e registos de transações no armazenamento tradicional de dados, é uma possibilidade de se tornar uma tarefa complexa de gestão e utilização.

Event Sourcing é uma forma diferente de armazenamento de dados onde, em vez de armazenar o estado atual, é a sequência de transações que é armazenada. O estado atual é construído através do processamento dos eventos, e não se perdem dados, o que dá ao Event Sourcing capacidades robustas de auditoria e análise.

O objetivo desta dissertação é enfrentar o desafio de pegar numa aplicação centralizada de Event Sourcing e replicá-la para a tornar mais fiável e disponível em diferentes regiões. Para atingir este objetivo, uma aplicação com armazenamento de eventos replicado teve de ser desenvolvida e colocada na Internet para ter o sistema distribuído por várias áreas do mundo.

Como resultado, este trabalho apresenta a arquitetura do sistema replicado criado em diferentes graus de especificidade, juntamente com tecnologias utilizadas, incluindo SpringBoot, Axon e MongoDB. Além disso, este documento apresenta também uma análise de desempenho das velocidades das operações de escrita e de leitura da referida aplicação colocada na nuvem. Estes resultados recolhidos a partir dos testes de desempenho revelaram-se encorajadores para os sistemas Event Sourcing geo-replicados, abrindo novos caminhos para futuras aplicações.

Palavras-Chave

Estado. Event Sourcing

Contents

Acronyms

ALB Altice Labs.

AWS Amazon Web Services.

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

CQRS Command and Query Responsibility Segregation.

DDD Domain Driven Design.

DEI Department of Informatics Engineering.

ES Event Sourcing.

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

JPA Java Persistence API.

WORM Write-Once-Read-Many.

List of Figures

List of Tables

Chapter 1

Introduction

This document reports the work I did for my Master's Degree Dissertation in Informatics Engineering from the Faculty of Sciences and Technology at the University of Coimbra (FCTUC), under the scope of the POWER project in association with Altice Labs (ALB), in the curricular year of 2022/2023. This project is taking place at the Centre for Informatics and Systems of the University of Coimbra (CISUC) at the Department of Informatics Engineering (DEI) of the University of Coimbra.

1.1 Context, Problem and Motivation

In traditional systems, what is usual to see is data being persisted in SQL tables or NoSQL databases. When a transaction occurs, changes are applied in the data at the databases to update the intended information [34]. Even though these systems do their job, they are limiting if a business needs more than what the current state is because information ends up lost. Industries such as accounting, banking or medicine rely on history to operate. Finance is about keeping track of every related financial transaction, doctors use medical history to make reasoned decisions, and both do not erase what came before, they add new entries. It is possible to create historical registration and transaction logs in traditional systems, but the more complex a system is, the more complex the management and usage of historical data becomes.

In Event Sourcing (ES), data is stored as a sequence of transactions and the current state is constructed from the transaction log by processing all events by order. Every time a new state change occurs, an event with the information regarding that change is created and appended to the event log. Naturally, ES fits the industries referenced before.

But there are more advantages to having data stored as a log than traditional data storage. Due to how ES stores data, no information is lost. Besides knowing the current state, it is also possible to tell how it came about. This aspect gives ES systems solid audit and analytical capabilities that traditional storage cannot offer. Since the state is a construction from the replay of events, another key

advantage is the possibility of rebuilding systems on the fly to any specific date or event.

ES is a relatively recent technological concept which gives this work the purpose of studying it to gain knowledge and insight into it under the scope POWER research project in which the Department of Informatics Engineering (DEI) and ALB are involved.

1.2 Objectives

The main objective of this work is to have an event-sourced application in a replicated environment to increase its performance, availability and reliability, with the purpose of developing it in a way that, to our knowledge, has not been attempted. To accomplish this objective, it is required to:

- Design and develop an application using Event Sourcing
- Configure the replication setup
- Place the entire system on the cloud
- Evaluate the replicated application determine the performance

1.3 Outcome

This work culminated in an event-sourced application seamlessly replicated across continents and with the accumulation of data concerning the established distributed system.

The app is a simplified banking app that can create clients and accounts for them, deposits and withdraws from existing accounts, transfers between accounts through Sagas and allows for balance and account checking.

After placing the system on the cloud, the performance testing showed that having an implementation like this works and adds value to the read speed. Plus, the system benefits from incremented availability capabilities since it has replication at its core.

1.4 Document Structure

This report is divided into the following chapters:

- **Chapter 2:** is dedicated to presenting the studied core concepts related to the project.

- **Chapter 3:** contains and describes the research carried out on potential technologies that could be used.
- **Chapter 4:** contains and describes the research carried out on different Event Sourcing implementations.
- **Chapter 5:** is dedicated to presenting and describing the Architectural Drivers. These are the functional requirements.
- **Chapter 6:** contains a description and representation, with the C4 Mode, of the architecture of the system aimed to be developed.
- **Chapter 7:** contains a description and representation of the setup layouts for the experiments along with the experimental methodology.
- **Chapter 8:** is devoted to present and analyse the data gathered from the experiments described in chapter 7
- **Chapter 9:** is devoted to project schedule and planning. It includes a description of the planned first and second semesters, the executed schedule for the first semester, risk assessment and threshold of success.
- **Chapter 10:** is dedicated to the final analysis of this dissertation, containing the major faced challenges, future work and some final thoughts on the dissertation endeavour

Chapter 2

Background: Concepts

This chapter describes the research on the core concepts related to this work. These concepts are Event Sourcing (ES), Command and Query Responsibility Segregation (CQRS) and Saga, each with its corresponding section. These concepts were chosen because Event Sourcing is the central concept of this dissertation, CQRS, for its connection to ES and Saga. Some important patterns related to data consistency in distributed systems are also explored. In addition to these, a brief section regarding some relevant concepts related to highly available systems is also included.

2.1 Command and Query Responsibility Segregation

Command and Query Responsibility Segregation (CQRS) [43] is a pattern that separates read and update operations from each other.

In traditional architectures, the same database with the same model is used for reading and writing, which can be enough for simple and traffic light applications but for more complex systems, it can become more challenging. In systems with higher complexity, the mismatch between read and write representations of data, data contention, or complex queries required to retrieve information can prove too much to handle on traditional architectures. In CQRS, having separate query and update models helps to mitigate and solve many of these problems.

The CQRS segregation can also be physical by having different data stores for each model, which can even be of different types. A write data store might be a relational database, while the reading one can be a read-only replica of the data store or something different as a graph database.

In separate write and read databases, they must be kept in sync. A solution for this is that whenever there is an update on the write model, in the same single transaction, as the update occurs, create an event that is going to be used by the read model to update its data.

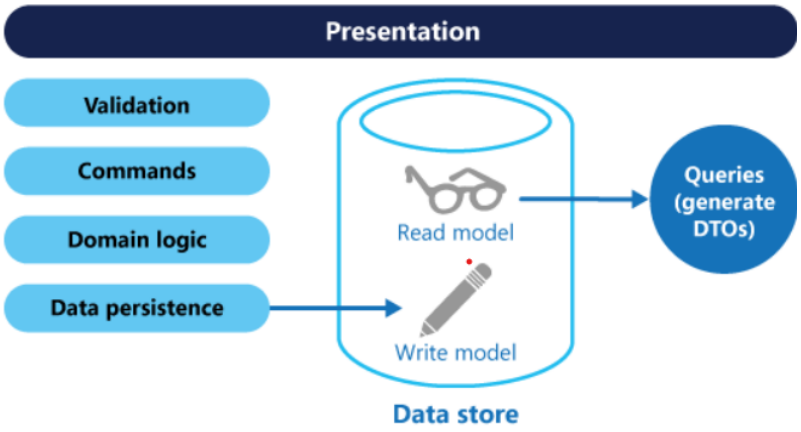


Figure 2.1: CQRS Example 1 [43]

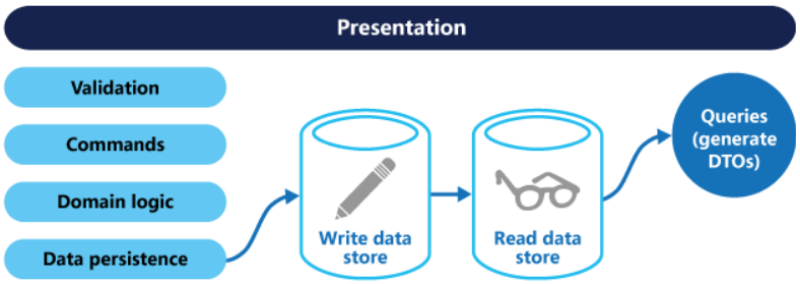


Figure 2.2: CQRS Example 2 - Separated Data Stores [43]

Benefits:

- **Separation of Concerns:** Segregating the read and write sides improves the maintainability and flexibility of a system.
- **Independent Scalability:** CQRS allows independent scaling of the read and write workloads according to different necessities. For instance, multiple read-only replicas can increase query performance since read stores typically encounter a much higher load than write data stores.
- **Flexibility and Optimization:** CQRS by supporting different database types, it is possible to add a read model that suits a new business necessity, therefore increasing performance. For example, a company has a write relational database where they have data about their clients, and these clients have a relation with other people. If a new business decision demands data about these relations between people, CQRS enables the addition of a read model as a graph database that is much faster at obtaining required information than a more complex query. [21].
- **Security:** Easier control over what entities can perform data changes.

Issues:

- **Complexity:** By having more than one model, the application design can become more complex.

- **Messaging:** In CQRS implementations where messaging is required, like when the read and write models are in separated databases, the system must be prepared to handle failures, duplicated messages, and priorities.
- **Eventual Consistency:** in separated databases can become difficult to detect when a user issues a request based on outdated data

When to use:

- Heavy user access to the same data in parallel
- Different performance needs for the write and read models
- Separation of concerns is an objective or necessity for the team and/or project
- Versioning of models
- Integration of other systems, as event sourcing

2.2 Event Sourcing

Event Sourcing (ES) is a way of persisting data where the system only stores facts and all state is "a first-level derivative" of said facts [21].

The idea of ES is to ensure that "every change made to the state of an application is captured in an event object and that these events are themselves stored in the sequence that they were applied for the application and state lifetimes" [30].

Typically, most applications that work with data maintain the current state of data by updating it as users work with it, for example, with CRUD operations (create, read, update, delete).

In ES, instead of just storing the current state on a table, it is stored as a sequence of events having the current state being a summation of that sequence, essentially as an event log. Events are simple objects generated when a system performs an action on data. They are also immutable, stored using append-only operations, and have all the necessary information to implement the intended action. Instead of directly updating a data store, these events are recorded in an event store so they can be processed at the appropriate time [44]. According to Martin Fowler is even correct to say that, in ES, "we are persisting two different things, an application state and an event log" [30].

Figure 2.3, a representation of event sourcing, has on its left the event log, the sequence of events saved in an event store of all departure and arrival events of all ships, and on the right, the “current” state of each chip. The first action that occurred, in this example, was the departure of “King Roy” from San Francisco at a certain point in time, followed by the arrivals of “Prince Trevor” in Los Angeles and “King Roy” in Hong Kong. If there is a need to know the current state of one or all ships, all that has to be done is to go through all the events, from beginning

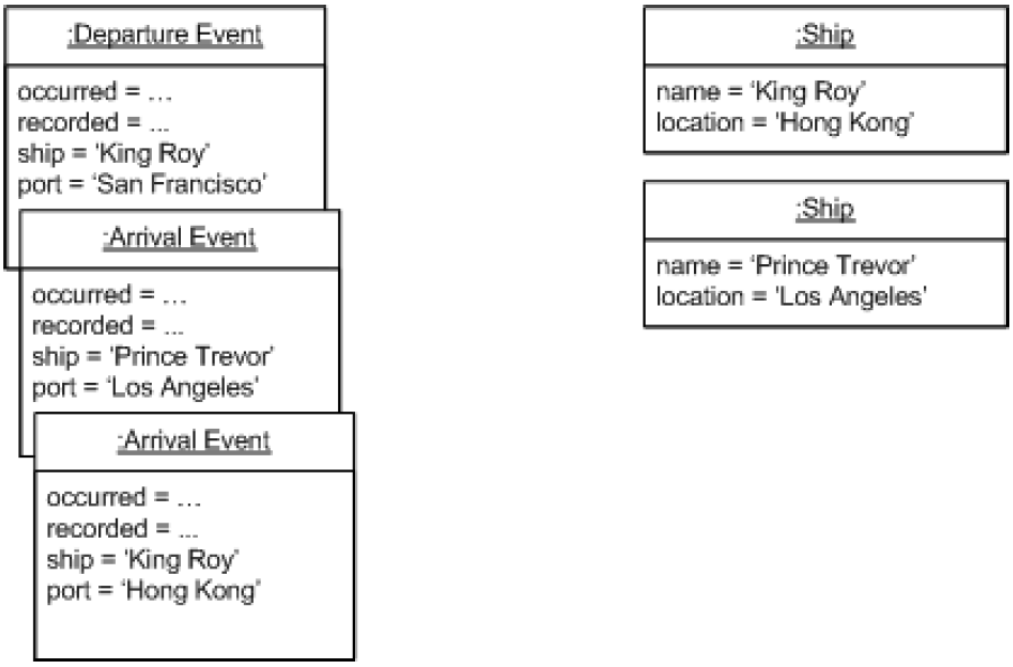


Figure 2.3: ES Example 2 - Ship Management [30]

to end, and derive the current state, resulting in the right side of figure 2.3. In addition to seeing where the ships are, is also possible to see where they have been. [30].

The example of figure 2.3 shows that if ES was not used, only the current state would be saved, and previous ship information would be lost. To prevent data loss, relational databases can have logs, and a system can have tables to store data change history. But saving a history of changes in log entries or a table is still event sourcing, but not implementing it as an event sourced system ends up adding complexity to the system.

ES Capabilities [30]:

- **Complete Rebuild:** Ability to completely discard an application state and, from a clean application, rebuild it by re-running all the events in the event stream.
- **Temporal Queries:** Ability to rebuild an application state to any point in time in the past or event.
- **Event Replay:** In the circumstance of an incorrect event, it is possible to go through the event log in reverse order until the incorrect event and replay the new correct event and later events as well.

Manipulation of Events Prevention

Given an event log, it is relevant to ascertain the past is not tampered with. A way to do this is to store the event log in a Write-Once-Read-Many (WORM) drive[21]. WORM drives are data storage devices that do not allow information to be changed once data is written on them, thus protecting the event log veracity.

Benefits of Event Sourcing: [44]

- **Simplified implementation and management**
  - Does not demand working with complex, hard-to-understand database tables.
  - Helps to prevent conflicts from updated systems. However, attention is necessary to avoid requests that might result in an inconsistent state.
  - It aids in the correction of the random bugs that occur sometimes, but no one knows why. Knowing at what moments those bugs occurred is only needed to rebuild the system at those exact moments and check what happened [20] [44].
  - The event log can be used for application performance analysis.
- **Performance and Scalability** for applications.
- **Flexibility and Extensibility:** caused by the decoupling between operations that trigger the events and the tasks that perform operations in response to said events.
- **Testing new software:** since actions are performed according to the events, new software can be tested in all operations and states that the previous system had gone through [21] [44].
- **Does not lose information:** all actions and changes are recorded
- **Business Analytics:** the event log can be used to detect user behaviour trends or to obtain other useful business information from the beginning to any point in time until the current moment
- **Security:** helps to prevent super-user attacks [21] with WORM drives

Issues of Event Sourcing: [44]

- **Consistency:** a system is only eventual consistent after creating or updating a projection of data after replaying events. Also, there's a delay between the creation, storage and handling of the events, which allows for the arrival of new events that describe further changes.
- **Concurrency when storing events:** multi-threaded applications and multiple instances of applications might store events concurrently. The consistency of events is vital since their wrong order may have consequences on the truthfulness of data. Timestamps and sequential identifiers together help mitigate the problem
- **There are no queries on the event stream:** to extract data from the log, its necessary to go through it and extract the required information
- **Idempotency:** Event consumers must be idempotent and must not reapply the update described in an event if the event has already been handled.

When to use: [44]

- Where business analysis is important.
- When it's necessary to capture the intent, purpose, or reason in data. Each event has a specific event.
- When there's a need to restore a system to previous states.
- When there's a need to keep a history and an audit log.
- Flexibility, extensibility, performance, and scalability are targeted Quality Attributes.

2.2.1 ES with CQRS

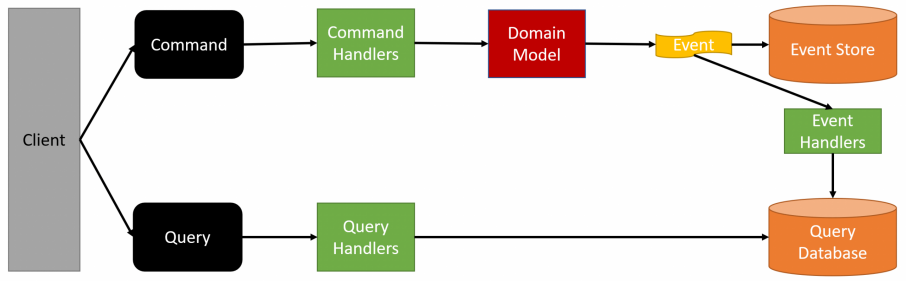


Figure 2.4: CQRS with ES - Representation [19]

CQRS is a pattern frequently used with ES. As previously established, CQRS systems use separate read and write models tailored to their responsibilities and necessities. When both patterns are combined, the default write model is the event store, and the read models are whatever fits best the queries that the business process requires. The read models use the event stream to derive the state with the information they need and can update it at each new event that changes their data [19]. The combination requires that the read models have to be updated according to the new incoming events, which can require processing time and resource usage that adds complexity to the implementation of the system alongside the other CQRS issues previously indicated [43].

2.2.2 Projectons

Projections (also known as View Models or Query Models) [38] are a transformation of the event stream into a meaningful model for the application that can be used on UI needed to be displayed or handled [17]. As it is in event sourcing, it derives the current state from the event stream, which can be done asynchronously. It is also possible to have different projections from the same event stream, as seen in figures 2.5 and 2.6.

Replaying all events in an event stream to get the current state can be very inefficient if the stream has a lot of events. A way to improve performance with

Figure 2.5: Projection example 1 [17]

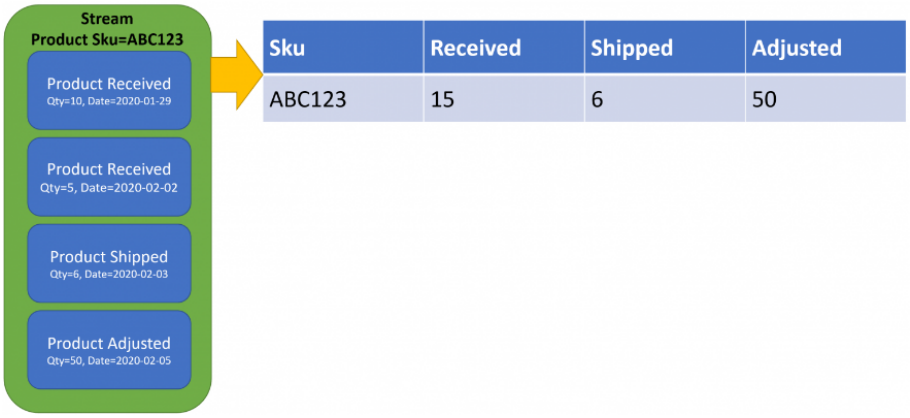


Figure 2.6: Projection example 1 [17]

projections is to have a projection that represents the current state and continue updating it as events occur [17].

One thing to note is that projections are not read models. The relation between them is that a read model is made of projections and that projections are a way of populating a read model with specific parts of the whole model [38].

2.2.3 Optimization

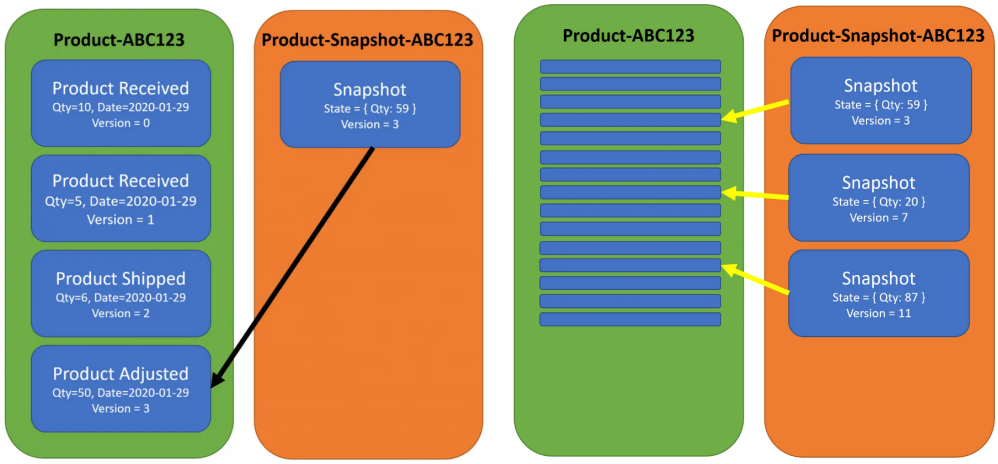
ES can become slow since, conceptually, to derive state, all events in the event stream must be re-runned.

- **Snapshots**

  A snapshot is a cached application state derived from the event log.

  When a stream has a lot of events, it can become inefficient to fetch every event from the event log and replay them to get the current state. Snapshots are a way of solving this problem by acting as a "checkpoint" of the state of an application. A snapshot can be created from the first event in the stream or a previous snapshot up any point in time. Snapshots should also

be saved in a different place [18][21], separate database, in memory or cache [26].



(a) (b)

Figure 2.7: Snapshot examples: (a) Snapshot example 1 [18] ; (b) Snapshot example 2 [18];

So, with snapshots, instead of replaying all events necessary to get the required state, the system uses the closest past snapshot and the following events up to the desired moment. It is also possible to have different types of snapshots pointing back to the same event stream with their distinct perceptions of it.

Snapshots are an optimization technique for the write model. If blended with read models, coupling is introduced and can become hard to untangle [26].

However, snapshots should be avoided [21][26] because they can lead to versioning problems because in doing a snapshot, state is being saved, so the same thing as storing state in SQL or any other structure. State is hard to version, and the need to use snapshots may hint at a model's design flaw [26]. However, sometimes, they may be necessary if the number of events is that big, as a highly active stock exchange.

- **Keep data in memory** [21]

  In keeping some data in memory, the system is decreasing the amount of times it needs to go and load events from the event store.

- **Multiple Event Streams**

  Most event source systems have millions of tiny logs instead of one giant log. It can be considered a document database in which a given document has all its corresponding events [21].

  In having smaller, shorter-lived active streams, downloading events is not a significant overhead since events are concise because they only contain the information needed. In addition to affecting performance, the stream's lifecycle also makes it easier to maintain [26].

The downsides of this approach are that it can be artificial and produce unnecessary overhead. If a stream is broken to reflect each work hour, it may turn out that it won't reflect the actual business flow and too small streams cause more significant overhead, and in tight performance requirements, potential overhead needs to be cut [26].

The history of events allows ES to provide auditability, diagnostics, and an added modelling aspect to consider: the lifecycle over time [26].

Businesses have a workflow, and its pattern is the source for breaking streams into smaller ones. Keywords like "closing the books", "day of work", "end of the day", "summary", or "shift" can be hints for modelling the problem having consideration on business activities. This not only allows to define what streams should exist, define when summary events exist (an event that contains all the needed business summaries), and archive strategy (moving data from the event store to "cold storage" – storage for not active data) [26].

– **Cash Register Example:**

  The following figure 2.8 exemplifies a cash register system that has two streams, one for each shift. As the first shift starts and all events related to that time frame are stored and read if needed. As soon the shift finishes, the system creates summary events (CashPayment and CreditCardPayment) containing all necessary business information about that shift, in the case of a cash register, to verify whether the state in the system is consistent with the actual amount of money in the cash register. As these summaries are created, shift two proceeds as the first. When one of the shifts starts again, if they need any information from the previous correspondent shift, the data comes from the summary events [26].

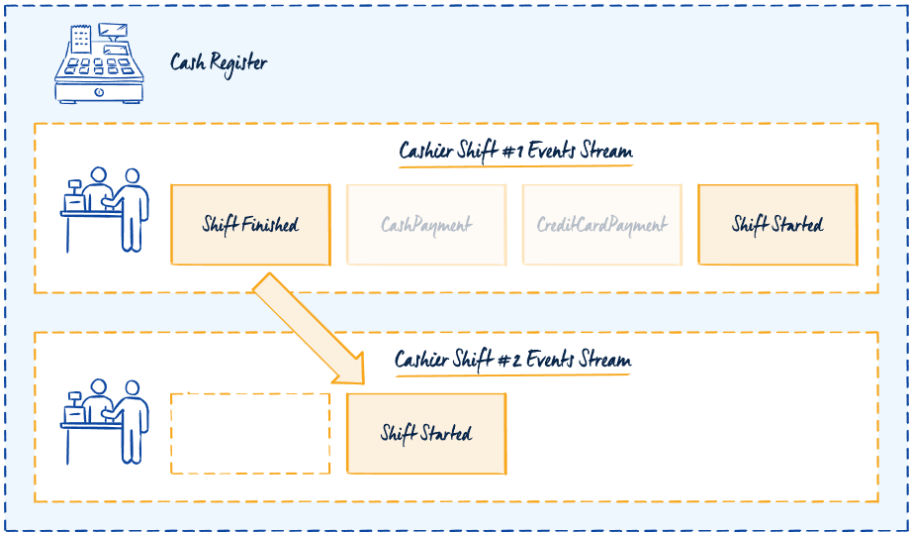


Figure 2.8: Cash Register example [26]

Another good point of discussion in this example is the archive strategy. Since each shift is independent and if any data from the previous

shift is required, it comes through summary events, all events from a previously active shift can be archived in cold storage.

– **Product Management Example:**

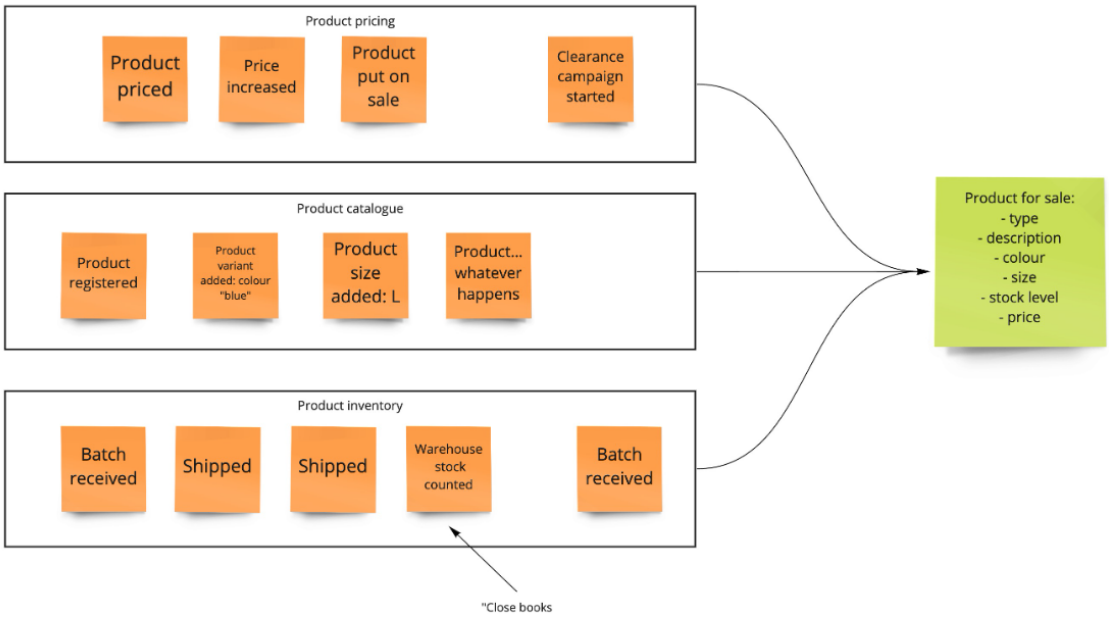


Figure 2.9: Product State example [58]

In the example of figure 2.9 the system instead of having one big stream that has all events related to all the business changes, it has it divided in tree differences ones based on different concerns such as product pricing, characteristics and inventory [58].

2.2.4 Handling of mistakes

Since events are immutable, if a mistake occurs, for instance, A sends to B 100€ instead of 90€, this error has to be corrected, and there are two ways of doing it.

- **Partial Reversal:** B sends back 10€ to A – this situation makes it harder to understand what happened as more individuals are involved in the error situation.

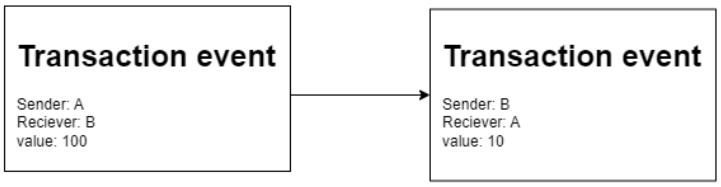


Figure 2.10: Partial Reversal example

- **Full Reversal:** A receives back the 100€, and then sends the correct amount of 90€ to B – Full reversals make it easier to understand what the error was [21].

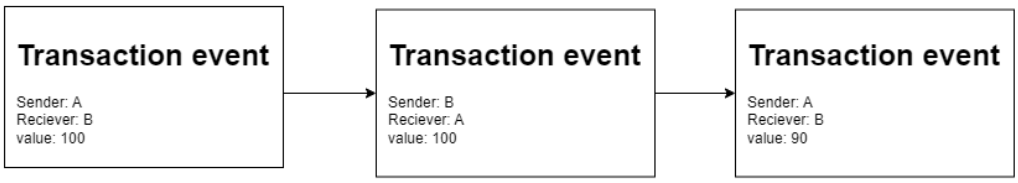


Figure 2.11: Full Reversal example

2.3 Saga

Saga is a pattern that targets the achievement of data consistency across microservices in a distributed system.

Transactions represent operations made by a user that incites a state change. They must comply with the ACID properties (Atomicity, Consistency, Isolation and Durability) so a system's data is consistent with the real-world operations and expected values.

ACID properties consist of four requirements to ensure data consistency before and after transactions [5]. Atomicity requires that all or none of the operations of a transaction must occur. Consistency means that a transaction must bring the data from one valid state to another valid one, in other words, the data has to be consistent before and after the said transaction. Isolation entails that concurrent transactions produce the same result as if executed sequentially. Durability requires that the changes of committed transactions remain even if a system failure occurs [5][45].

In a single service, transactions are easily ACID, but in a multi-service system, a strategy is needed to ensure that distributed transactions comply with the referenced properties.

By establishing the usage of a sequence of local transactions, the saga pattern is one strategy that fixes the distributed transactions issue. Local transactions are units of work performed by a single microservice participating in a saga, where each local transaction updates the database of one microservice and publishes a message or event to trigger the next local transaction in the next microservice in the saga (figure 2.12) [45].

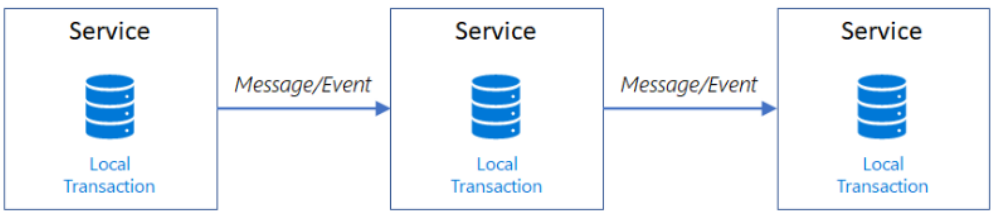


Figure 2.12: Saga example 1 [45]

In the case of a failure, the saga executes compensatory transactions, with the same local transaction–trigger logic, to undo all the previous changes that were made in all previous services in reverse order, as in figure 2.13.

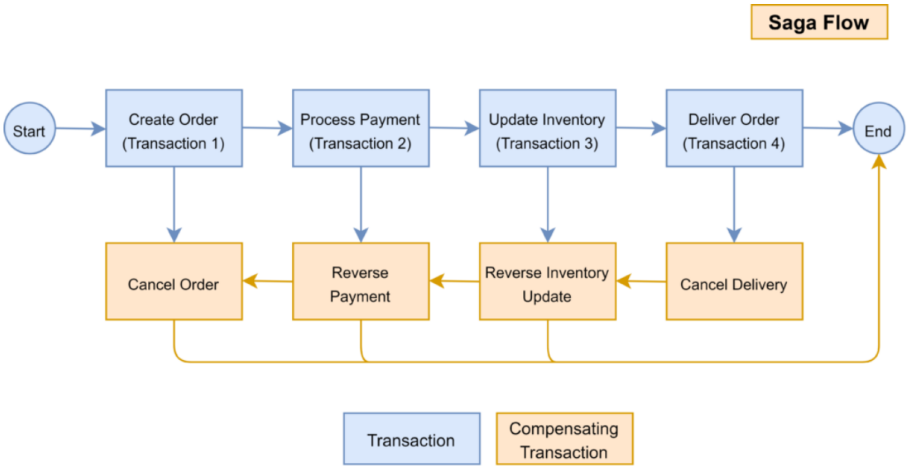


Figure 2.13: Saga example 2 [16]

Implementation

There are different ways of implementing sagas, being the most common are the Choreographed and Orchestrated approaches.

- **Choreographed**

  In this approach, each microservice in the transaction publishes events that trigger local transactions in the following service in the chain without any centralized point of control (figure 2.14). [45]

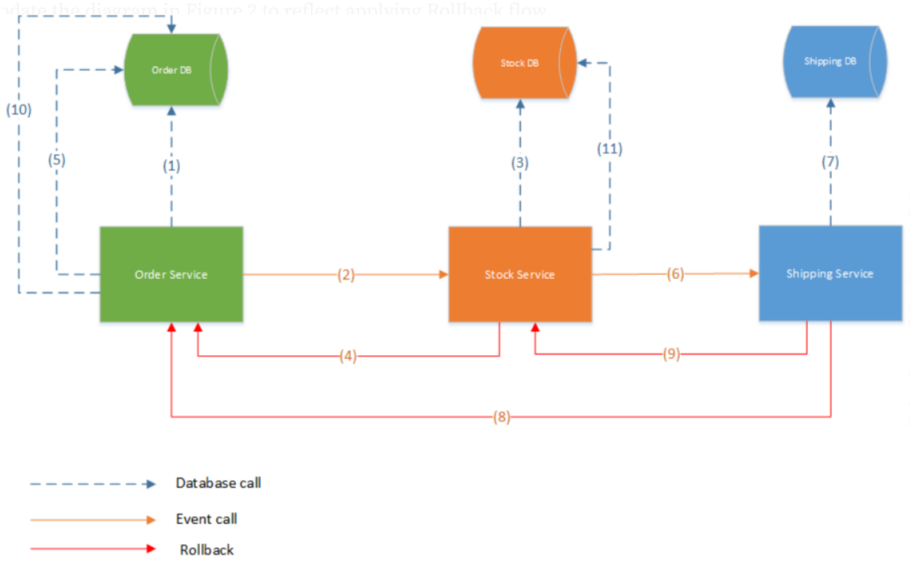


Figure 2.14: Choreographed Saga example [37]

Pros: [45]

- Good for simple workflows that don't have a lot of participating services and don't need to be coordinated.
- Simple and easy to build up [37]
- Additional implementation or maintenance is not required.

- Doesn't have single point of failure

Cons: [45]

- Doesn't have single point of failure
- Workflows can become complex and confusing when adding new steps, as it becomes more difficult to track which services listen to which triggering messages/events.
- Risk of cyclic dependency between services since they have to consume each other's messages.
- Difficult to test because all services must be running to simulate one transaction.

- **Orchestrated**

  The Orchestrated approach has a centralized controller, an orchestrator, that handles and tells the saga participants which local transaction to execute based on events. The orchestrator is also responsible for handling failures with the invocation of the necessary compensatory transactions (figure 2.15) [45].

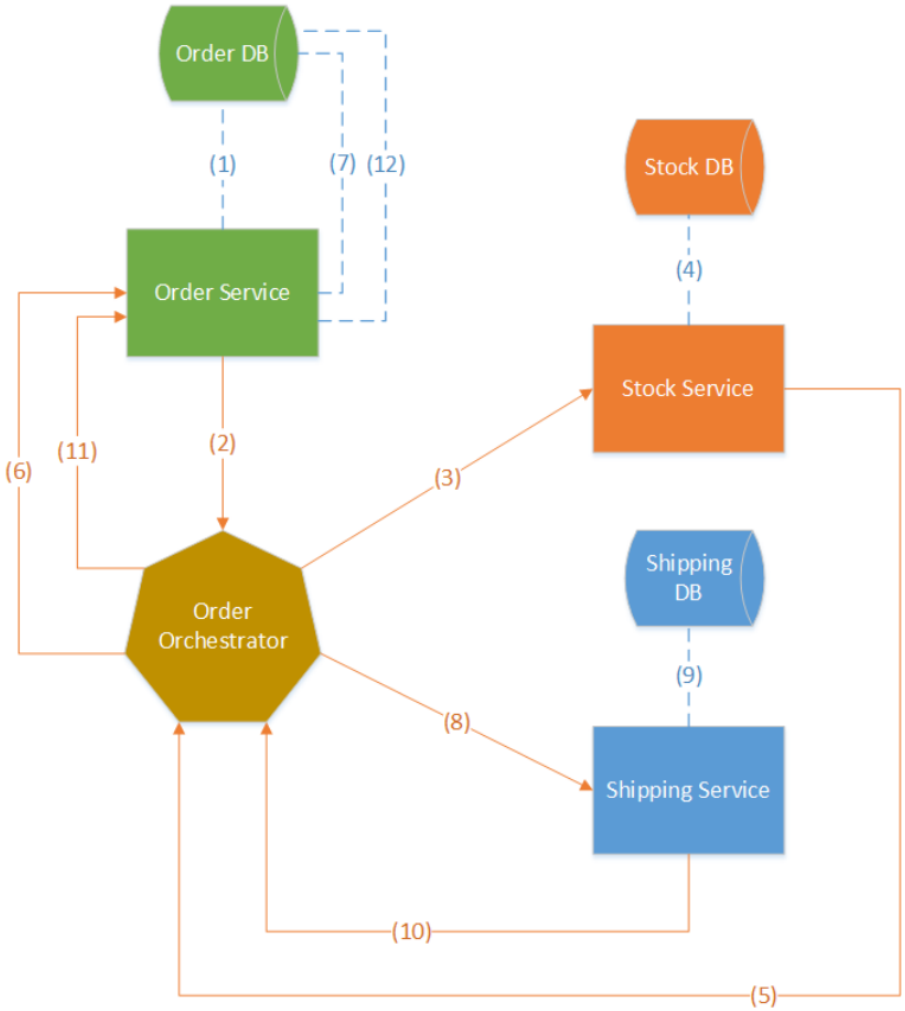


Figure 2.15: Orchestrated Saga example [37]

Pros:[45]

- Good for complex workflows with a lot of services
- Good for the addition of new participants.
- Provides control over the flow of activities.
- Has no cyclical dependencies due to the existence of the orchestrator.
- Clear separation of concerns simplifies business logic.

Cons:[45]

- Coordination logic is required, which adds complexity.
- There's a point of failure, the orchestrator that operates the complete workflow.

2.4 High Available Systems

Availability is a non-functional requirement that a system can have. Typically, a system built with availability in mind is trying to achieve a high percentage of uptime.

A way of achieving availability is to have the system running in different machines at different locations as a distributed system, whether through replication, where each instance of the system has the same responsibilities or by dividing the system's tasks into other distinct instances.

Some concepts of distributed systems useful for this projects are going to be presented.

Consistency

Data consistency regards whether data replicated across different nodes of a distributed system match between replicas.

Eventual Consistency is a consistency model used in distributed systems to achieve availability that eventually guarantees data consistency. The inconsistent window is when there is an update to some data in one node, and the other nodes need to receive that update too [56] [36].

CAP Theorem

CAP stands for three desirable properties in systems with distributed databases: Consistency, Availability and Partition Tolerance. Consistency, in a distributed environment, means all reads after a write operation must return the new value. Availability that every request to a function node of the system must get a response within a time limit, and Partition Tolerance that, between nodes, messages can be delayed or lost [32][53].

The theorem states that, at most, a system with a distributed database can only have two of the three properties. If a system has to be consistent and available,

it can not afford to have partitioning. If a system has to be consistent and have partitioning tolerance, availability suffers because the data has to be consistent before any response. If a system has to focus on availability and support partitioning then data consistency is not assured.

PACELC Theorem

PACELC theorem is an evolution of the CAP theorem. It states that if there is partitioning (P), the system can either prioritize availability (A) or consistency (C), else (E) if the system has replicated data and no partitioning, the system can prioritize latency (L) or consistency (C) [6].

Quorum Replication

In quorum-based replication, a write operation is only considered complete when more than half of the cluster members of the distributed system have confirmed that same operation. For example, a five-node cluster can continue to accept writes as long as at least three are operational [40].

2.5 Final Remarks

This chapter has provided a comprehensive overview of the relevant concepts as Event Sourcing and others related to it. ES is not a trivial pattern, and understanding it and its details is fundamental to not fall under bad practices when implementing it. CQRS is a pattern that connects to event sourcing in an effective manner to optimize queries, and Sagas are a noteworthy pattern for implementing reversals when mistakes or operations encounter errors.

Chapter 3

Background: Technologies

In this chapter, the focus is to describe the research performed on technologies related to the project. The objective is to study and compare the technologies that could be used.

3.1 Axon

AxonIQ[13] has a set of technologies for developing event-driven systems. From their offerings, those that will be presented in more detail are the Axon Framework and AxonServer.

3.1.1 Axon Framework

Axon Framework is an open-source framework from *AxonIQ* tailored to develop applications based on the ES, CQRS and Domain Driven Design (DDD) concepts, including Sagas. The objective of this framework is to allow the developer to focus on the business logic and not on the infrastructure on which the application is running. In other words, the developer only has to worry about what to do with the events and not about how the system handles them. It has extensive documentation [7], a variety of free tutorials and guides [8][9] and is updated regularly.

An axon application is essentially a spring boot application with Axon's functionalities of event management, supports java as a programming language and can also be used with kotlin. To run an axon application is required a connection to an axon server [11][12].

Axon provides the ability to integrate with other technologies such as Kafka and Mongo.

Another thing that makes the use of axon easier, is the axon initializer [AxonIQ Initializr] which essentially generates a base project ready to use with all the dependencies needed already incorporated for the application.

3.1.2 Axon Server

Axon Server is a companion to the Axon Framework necessary to run an axon application. It has been developed in java on top of Spring Boot, is the default event store and is built to offer an easy setup, fast connection, high performance, and low maintenance [11]. It also comes in two versions, the open-sourced Standard Edition and the commercially licenced Enterprise Edition. Cluster and replication capabilities are exclusively in the paid Enterprise Edition.

The axon server aims to [14]:

- reduce configuration management because is purposely integrated with Axon Framework and performs service discovery (auto-detection of devices and services) and message rooting without any configuration.
- be prepared to scale horizontally to have no problems in having 2 or 50 instances.
- offer a real-time graphical overview to show what is happening in the system.
- promote scalable event sourcing by containing a storage engine designed specifically for storing events, offering scalability without much tunning.

3.1.3 Axon Application Example

A simple functional example using the axon framework was developed to understand how it works. This example is an adaptation of one from a live tutorial in the axons YouTube channel [8] and is a simple implementation of an event-sourced application of a bike renting store. The application created had to be capable of creating bikes for rental, renting bikes and receiving the rented bikes.

For the sake of avoiding unforeseen complications and saving time, this example was written in one class. It is relevant to be noted that this is not the correct way to develop an application.

The Axon Reference Guide [7] and the Axon Quick Start Guide [9] were also used to develop the axon example.

Setup:

The Axon initializr [10] was used to create a Maven project with all the necessary dependencies. The chosen were: the predefined Axon Framework and Axon Test, Spring Data JPA, H2 Database, Spring Web and None for the Axon Server.

Application Structure:

The application is divided into three major components: Command Model, Query Model and Commands and Events.

The Commands are the intents of state changes provided by the user. In this example, if the user requires a new bike, this action creates a command that is going to be handled in the Command Model.

The Command Model is responsible for handling its entitled commands and generating the events that represent the commands' state changes and storing them in the event store.

The Query Model is responsible for the projections of the current state.

Commands and Events:

The specified commands for the application were *BikeOrderCommand*, *BikeRentCommand* and *BikeReturnCommand*. *BikeOrderCommad* regards the addition of new bikes to rent out in the system, BikeRentCommand the intent to rent out a bike and *BikeReturnCommand* the return of a rented bike.

Regarding events, there is one event for each command. Events in event sourcing represent state changes, so for each intent, there is a confirmation of the action. The events are *BikeAddedEvent*, which represents a new bike in the system, *BikeRentedEvent*, which means that a bike became rented and *BikeReturndEvent*, which suggests that a bike has been returned.

Each command and event have all the information necessary to perform its intended action. The following figures (3.1a and 3.1b) are an example of what is inside a command and an event.

(a) (b)

Figure 3.1: Command and Event examples: (a) Axon Command Example; (b) Axon Event Example;

In Figures, 3.1a and 3.1b is possible to see @TargetAggregateIdentifier. This annotation is to identify the String *bikeId* variable as the identification of the instance of the exact bicycle, so Axon understands to which bike these commands and events refer.

Command Model (Aggregate):

Axon uses the concept of aggregates for this portion of the application. An aggregate is, according to Martin Fowler, a "cluster of domain objects that can be

treated as a single unit" [31], in other words, is a domain which handles the objects that are part of it. For the case of the Bike Rental Application experiment, the only aggregate is the *BikeAggregate*, identified by the @Aggregate annotation (figure 3.2). In this way, in this system, each bike is going to be an instance of an aggregate.

```
70      @Aggregate
71      public static class BikeAggregate {
            3 usages
72          @AggregateIdentifier
73          private String bikeId;
            1 usage
74          private int bikeName;
            3 usages
75          private String renter;
            4 usages
76          private boolean rented;
77
78          //Base Constructor is required by Axon
79          public BikeAggregate() {}
80
81          @CommandHandler
82          public BikeAggregate(BikeOrderCommand command) {...}
86
87          @CommandHandler
88          public void handle (BikeRentCommand command){...}
96
97          @CommandHandler
98          public void handle (BikeReturnCommand command){...}
103
104         @EventSourcingHandler
105         public void on(BikeAddedEvent event) {...}
112
113         @EventSourcingHandler
114         public void on (BikeRentedEvent event){...}
119
120         @EventSourcingHandler
121         public void on (BikeReturnedEvent event){...}
125     }
```

Figure 3.2: Axon Aggregate Example

The aggregate has four major components: variables, empty constructor, command handlers and event handlers. All of these can be identified in figure 3.2.

The variables in the aggregate are all the ones necessary to store the information of the current state that influences the consistency decisions to see if an action/-command is possible. One of these variables is the String *bikeId* that, through @AggregateIdentifier, is telling Axon that for each aggregate instance, for each bike, that value is its id. Axon knows to which aggregate each command and

event corresponds through the values marked with @AggregateIdentifier (aggregate) and @TargetAggregateIdentifier (events and commands).

The command handlers are responsible for handling commands and are identified as such to axon with @CommandHandler. Command handling is the portion of the code responsible for deciding if the state change of the command should be applied or not.

Figure 3.3 shows the renting command handler in more detail. Its task is to verify if the requested bike is available to be rented by checking the value of the aggregate's rented variable. If the bike is not rented, an event is created. If the bike is already rented, the action is rejected. The apply method in line 91 (figure 3.3) does two things: sends the event to the event store and applies the state changes in the aggregate through the matching event sourcing handler.

```
87      @CommandHandler
88      public void handle (BikeRentCommand command){
89          if (!this.rented){
90              apply(new BikeRentedEvent(bikeId, command.renter));
91          }
92          else{
93              System.out.println("Already Rented");
94          }
95      }
```

Figure 3.3: Axon Command Handler Example

The aggregate event handlers, identified with the @EventSourcingHandler, are responsible for handling the published events by using them to change the aggregate state.

```
113     @EventSourcingHandler
114     public void on (BikeRentedEvent event){
115         this.renter = event.getRenter();
116         this.rented = true;
117     }
```

Figure 3.4: Axon Event Sourcing Handler Example

Upon an applied event, the matching event handler receives the event and changes the aggregate state accordingly (figure 3.4). These event handlers are called when an event is applied and when a system is reconstructed, such as after a system goes down, when it comes up, it uses the events already stored to reconstruct the aggregate's current state.

Query Model:

The query model is the part of the application responsible for responding to queries. In this experiment is a Java Persistence API (JPA) repository that uses the newly published events or the events from the event store to construct the current state for readings. This means that the axon application also uses the con-

cept of CQRS. The database used with JPA is H2 Database, which is an in-memory database, as established in the setup with the Axon Initializr.

The repository consists of several entities, bicycles, in the case of this application. The class in the following figure 3.5, annotated with @Entity, represents the current state of a bike in the read model. The @Id annotation on the bikeId variable, like @AggregateIdentifier (in aggregates) and @TargetAggregateIdentifier (in events and commands) annotations, tells Axon which bike this instance of the class is about.

```
300     @Entity
301     public static class BikeStatus {
            2 usages
302         @Id
303         private String bikeId;
            2 usages
304         private int bikeName;
            4 usages
305         private String renter;
            4 usages
306         private boolean rented;
307
308         public BikeStatus() {}
            1 usage
309         public BikeStatus(String bikeId, int bikeName) {...}
315
316         public String getBikeId() { return bikeId; }
319
320         public String getRenter() { return renter; }
323
324         public boolean getRented() { return rented; }
327
328         public int getBikeName() { return bikeName; }
            1 usage
331         public void markRented(String renter){...}
            1 usage
335         public void markFree(){...}
339     }
```

Figure 3.5: Axon Query Model Entity Example

To update the data of these entities of the query model, they have their event handlers. @EventHandler identify these handlers (figure 3.6), and like the @EventSourcingHandler for the aggregate, upon an applied event, they receive it and employ the state change to the correct instance of the bike.

To return the queried bicycles, it is only necessary to send the desired bikes from the repository (line 293 - figure 3.6), which are returned as JSON.

```
@RestController
public static class BikeRentalStatusUpdater{
    5 usages
    private final BikeStatusRepository repository;

    public BikeRentalStatusUpdater(BikeStatusRepository repository){...}

    @EventHandler
    public void on (BikeAddedEvent event){...}

    @EventHandler
    public void on (BikeRentedEvent event){
        repository.findById(event.getBikeId()).get().markRented(event.getRenter());
    }

    @EventHandler
    public void on (BikeReturnedEvent event){...}

    @GetMapping("/status")
    public List<BikeStatus> statuses(){
        return repository.findAll();
    }
}
```

Figure 3.6: Axon Query Model Entity Example

Application Deployment:

As stated before in the 3.1.2 subsection, an Axon application to work needs an Axon Server. So to run this example was necessary, at first, to run an instance of an Axon Server [15] and then the application.

3.2 EventStoreDB

EventStoreDB is a purposely designed database to store events for event sourcing from *Event Store Limited*. EventStoreDB OSS is the free, open-source version, and EventStoreDB Enterprise is the paid version that comes with the paid support subscription [42].

EventStoreDB supports gRPC [33], TCP and HTTP communication protocols, being gRPC the recommend since it is their primary protocol while TCP is planned be phased out and HTTP is depricated. The available clients to develop software that use Event Store DB vary depending on the protocol to use and their type. There are two types of clients: the recommended official supported clients and the community developed clients. gRPC is the one with more official clients, having .NET, Java, Node.js, Go and Rust as official supported programming languages, and has two community clients, Ruby and Elixir. TCP has three official clients with .NET, JVM client and Haskell and has Node.js, Elixir, Java 8, Go and PGP for the community clients. HTTP only has available community clients with PHP, Python, Ruby and Go [39].

Regarding clustering, Event Store DB is capable of it to achieve high availability.

A node can have three roles in a cluster: Leader, Follower and ReadOnlyReplica. The Leader's role is to ensure that the data is committed and persisted before acknowledging the client, they are elected, and only one can exist at a time in a cluster. The Follower is also role based on an election process, and its purpose is to form a quorum. The ReadOnlyReplica is the role for any node that isn't a cluster member that exists for scaling reads [41].

Regarding replication, it uses a quorum-based replication model (section 2.4).

A replicator tool [57] to help with the replication process is also available that can be used to replicate data between two EventStoreDB clusters, but also for migration to the cloud, event filtering during migrations and event transformation regarding stream name, type, data or schema [57].

3.3 MongoDB

MongoDB is an open-source non-relational (NoSQL) database from *MongoDB Inc*.

In being a NoSQL database, MongoDB allows for a more flexible data structure, which means that data does not need to be stored in previously defined strict tables as in a relational SQL database. In the case of Mongo, data is stored as documents that are saved in BJSON (Binary JSON) [27].

Documents are data structures composed of field-value pairs. The flexibility provided by Mongo, for not being a relational database, allows the fields to be from a variety of data formats as a simple string or more complex as arrays, other documents, or arrays of documents. These documents are stored similarly to tables in relational databases called collections. [46]

MongoDB is capable of clustering and replication. In MongoDB, a group of Mongo instances that maintain the same data set is called a replica set. A node in a replica set can have three types of members: Primary, Secondary and Arbiter. Primary and Secondary types result from replica set elections where the primary replica is elected [47]. The Primaries are responsible for receiving and confirming all write operations in a quorum-like approach, and only one can exist at a time in a replica set. The Secondary replicas are read-only, and their responsibility is to replicate the primary's data to themselves, so the secondary's data are equivalent to the primary's. The Arbiter is a replica which participates in elections but does not hold data [48]. MongoDB's replication facility also provides automatic failover. [46]

MongoDB also uses the method of sharding [49], distribution of data through various machines, which allows horizontal scalability to support the deployment of very large data sets and high quantities of operations. [46]

3.4 Apache Cassandra

Apache Cassandra is an open-source, NoSQL, distributed database from The Apache Software Foundation [29].

Being a NoSQL database, Cassandra allows for a more flexible data structure. Data is stored in Cassandra using flexible tables that allow unstructured data to be stored. This database is also column oriented, which means that data is stored by column and not by row.

As a distributed database Cassandra has clustering and replication capabilities and supports partitioning. A node is a completely capable instance of Cassandra with the responsibility of holding their partitions of data and nodes communicate with each other in a peer-to-peer connection where all nodes have equal responsibility. To achieve fault tolerance and not have a single point of failure, each node not only holds to their main partition but also hold replicated partitions from other nodes [28].

Looking at the CAP, theorem Cassandra by default is an AP (prioritizes Availability and has Partitioning tolerance) but can be configured to be CP (prioritize Consistency with Partitioning tolerance).

As in NETFLIX's implementation of ES [35], Cassandra's capabilities can be taken as an advantage to have events ordered in a custom way that doesn't negatively affect the consistency.

3.5 Apache Kafka

Apache Kafka is an open-source, publish-subscribe, fault-tolerant streaming system [23] [4].

As a publish-subscribe system, Kafka is, at its most basic, a queue where a producer (a generator of messages) publishes a message, and a consumer (a potential receiver) goes to the said queue, to the stream of messages, to read the intended message. If these messages are events and Kafka has big enough queues to have a lot of events available to be consumed, Kafka naturally fits event sourcing as an event store. According to Jedrzej Rybicki [52], one of Kafka's strengths is that the contents of these streams are written to disk for a defined period of time with fault tolerance as an objective through partitioning and replication.

A topic (stream) can have different partitions to handle messages. The problem with this is that, in ES, events need to have an order, and Kafka can only assure the order of a partition, but not cross-partition. This detail is something to be kept in mind if Kafka is utilized [54].

Confluent provides two takes of how to implement Kafka in an event sourcing system. [51]

The first take, seen in figure 3.7, is to use Kafka as an event handler for the appli-

cation state in a CQRS system. Kafka's role is that upon a new event is written in the event store, to make available the latest event to the applications read model so it can update its state.

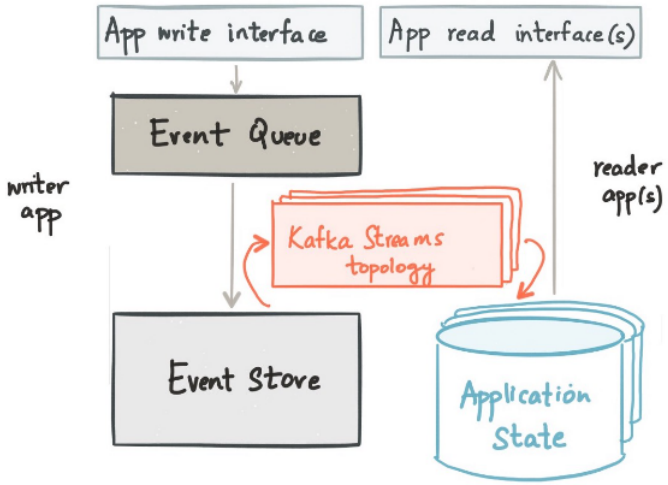


Figure 3.7: Kafka implementation Confluent's take 1 [51]

The second take (figure 3.8) is to, in a CQRS system, use one Kafka topic (stream of messages) as an event store and utilize other streams to store the application state.

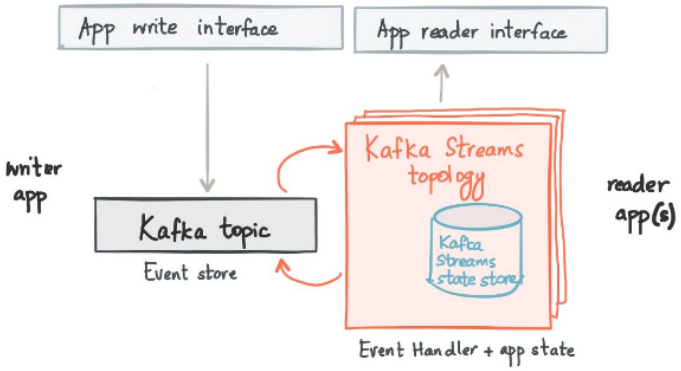


Figure 3.8: Kafka implementation Confluent's take 2 [51]

Confluent presents a real-world implementation of this second take for the New York Times articles consultation. In this example, there's one stream where all article assets are as separate messages, and then these contents are replicated into smaller streams for the different nodes in the system [22][24].

3.6 H2 Database

H2 Database is an open-source Java SQL database that can be embedded in Java applications or used in a client-server mode [1][2].

The most appealing feature is that this database can be run as an in-memory database, which means that data will not be persisted on disk, and every time the application is closed, the data will be lost. This feature allows the H2 database to be a good option for development and testing purposes but not the best option for production.

Within H2s scope, beyond the ability to be an embedded in-memory database or deployed in a server, it also offers transaction support, multi-version concurrency, browser-based Console application, encrypted databases and full-text search.

3.7 Final Remarks

This chapter has provided a comprehensive overview of some useful technologies for developing an application with event sourcing as its core.

The technologies presented here were the ones that appeared most helpful to the topic at hand. Some of them were created with Event Sourcing and event-driven applications as their selling point, as the Axon products and EventStoreDB, and others may not have been created purposely for Event Sourcing but are capable of filling their role if implemented correctly as Kafka, MongoDB, Cassandra and H2.

Decision

The technologies chosen were Axon Framework and Server, MongoDB and H2 Database.

The Axon technologies offer a good jump start to develop an event-sourced application since they already give the tools to develop one. MongoDB is a database that offers replication possibilities, including sharding, plus Axon Framework offers extensions for MongoDB, which allows for a straightforward configuration between the two. Within a CQRS model where Mongo is the writing model, H2 will fill the role of the database responsible for responding to queries, a role that, within the scope of this project, this in-memory fits in.

Previous experience with these technologies also waited in this decision.

Chapter 4

State of the Art

In this chapter, the research performed on real-world implementations of Event Sourcing will be presented. The purpose is to understand ES implementations, understand why this concept was used, and in what way.

4.1 Netflix's Download Feature

Netflix is an entertainment and streaming company that, in 2016, launched its download feature to allow its users to download content for offline viewing [35].

How Netflix's streaming works is that every time a user chooses to see something, the application will get from the server the metadata for the content (URLs and language) and a license (to decrypt the content). As the streaming session goes on, it will generate session events such as start, pause, resume, or keep alive. When the session ends, the license is realised, and a stop session event is published.

The download feature would have to embrace the streaming lifecycle and adapt it to an offline environment, where the content would be downloaded in its entirety. So for the download feature, when the users choose something to download, it downloads the correspondent metadata, license and encrypted content. During the viewing sessions, session events will be created and only uploaded when the user goes online. To control the time a user can have and watch content offline on a subscription-based service, each piece of content will only be able to be watched this way for roughly a year, and the licences expire every month, requiring the user to go online to renew it.

Other business requirements for this offline feature were: limitations on the number of devices with downloads on them, external studios' caps on downloads of their content and limitations on how many times a user can watch the same thing over a year.

This feature revealed itself as heavily reliant on history and not just on the present, so Netflix's team opted to use event sourcing instead of a traditional model.

Event Sourcing Implementation:

The offline system works with two core services: License Service and Download Service. Customer "A" wants to download movie "X", so the License service asks the Download Service if "A" is allowed to do that.

The Download Service, through the Download Repository, this service's event source processor, goes to the event store and replays the events referring to "A" and the movie "X" to get the number of downloads for the last year, verifies if "A" as reached the yearly download cap for that movie and a positive or negative message is sent to License Service.

The Licence Service receives the message and, if it is positive, sends a command to the License Repository (event sourcing processor) to create the "License Created", with the customer and movie ids and expiration date, an event referring to the license creation, and to publish that event to the event store. When the License service receives the confirmation that the "License Created" event has been persisted in the event store, the service asks the DRM for a license and sends it to the user.

Event Store Implementation:

To store the event log, the event store used on this download feature, the chosen database was Cassandra. This NoSQL database can organize information according to different types of columns for data organization and faster reads. In Netflix's case, their database schema has five columns, three of which can be used for search that represent the user and the service, the movie and the time value related to the event that that row represents.

Snapshots:

Snapshots are another component of this event-sourced feature. If a snapshot is triggered, the events are serialized to another table on Cassandra, the "Snapshot Table". Consequently, the next time a query to the event store is done, the snapshot is read first and then the following events from the event log (subsection 2.2.3).

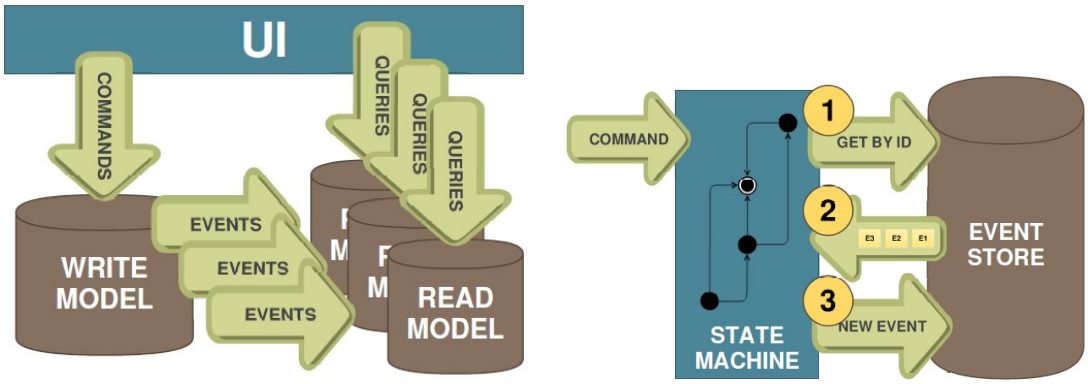
4.2 Scalable Event Source Application

At the AGH University of Science and Technology in Poland, a team set out to study and determine the possibility of implementing a scalable application in a reactive fashion that used the concepts of CQRS and Event Sourcing [25].

To prove the premise, they created a prototype based on a real-world interactive flight scheduling application. The choice was made regarding some of the application's capabilities that made them a good fit for CQRS and ES as maintaining schedules, reporting and data analytics, and all handled within reasonable response times.

Implementation:

The basic idea is to have the system divided into three components due to CQRS (fig 4.1a): the "UI", the "Write Model", and the "Read Model". Because ES is used within this system, the "UI" sends commands and queries, and the "Write Model" (fig 4.1b) handles commands and produces events that are stored in the event store and published to the "Read Models", which are responsible for handling queries.



(a) (b)

Figure 4.1: Basic Idea [25]: (a) CQRS implmentation; (b) Event Sourcing Implementation;

To scale out this "Write Model" (fig 4.2), the team decided to use sharding to partition the commands by the hashing data identifier that that command relates to instead of replicating the command processing units. This decision allows the system to balance the load, straightforwardly add new nodes, and not deal with conflict resolutions because each node handles its data. Another decision made to diminish latency is to have the current state cached, so it is not required to replay events and reconstruct the current state to apply business rules to the commands.

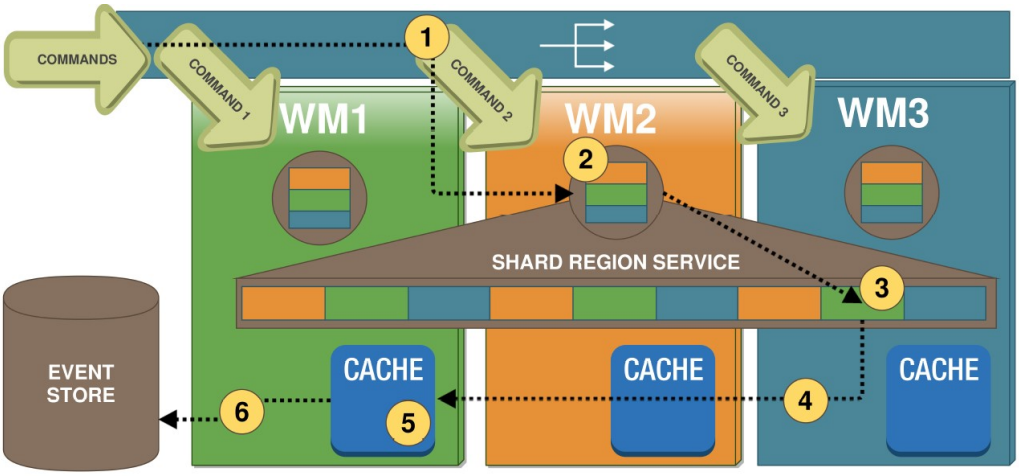


Figure 4.2: Scalable System [25]

The technologies used were Akka, as the base technology, to provide middleware to facilitate communication between entities in the system in a message-passing style, Spray to expose REST endpoints and serialize JSON, Cassandra and Kafka

for the event store and messaging queue and Neo4j (graph database) as an in-memory database as the read model.

Results of the Experiment:

The team were able to create their system, proving that it was possible to create an application with their set of requirements while using CQRS and ES and that it is horizontally scalable. The experiment also showed that scaling the write model resulted in lower response times.

They also concluded that their implementation, with all its upsides, has downsides as eventual consistency and the necessity of being prepared for duplications, losses and retries due to distribution. Another downside found has a form of mitigation as Snapshots that helps the speed of recreating the current state as it becomes slower over time.

4.3 Final Remarks

This chapter has provided an overview of different implementations of Event Sourcing under or linked to real-world problems. Netflix's example showed an implementation that had to fit into their established system and have an offline aspect that they managed to implement through two different services. The team at AGH University of Science proved that ES+CQRS is horizontally scalable with their distributed prototype by replicating their command handling components with additional tactics such as sharding and caching.

Chapter 5

Architectural Drivers

This chapter contains information related to the objectives for the new system to achieve. Here the functional requirements and constraints that impact the architecture are presented. The following architectural drivers were chosen given that the overall objective is to create a highly available system with fast readings while using ES.

The goal is to build a distributed system capable of handling regular query requests, data change requests and have the users' data replicated through other replicas. Some problems that arise come from replication. Each state holder in the system must be able to update its data with events received from other event store replicas. Additionally, when an operation is executed in one replica, the action should not be repeated by other replicas because it must be performed only once.

5.1 Requirements

The requirements presented in this section are those considered the most relevant and are based on the knowledge gained from the axon experiment. Their purpose is to provide a better understanding of what will be implemented and alternative scenarios.

The defined ten requirements for the system are described in tables 5.1 through 5.12.

Challenges risen from Replication, REQ06 (table 5.6) and REQ07 (table 5.7):

Due to new events coming from other event stores, the aggregates, the state holders for business logic, must apply the newly replicated events to update their state. This necessity creates the danger of executing an operation twice if there are exterior consequences. An example is if an action leads to sending an email when the respective event is replicated, the email should not be sent again. Certainly, a challenge that has to be faced.

REQ01 – Respond to state change requests	
Level	Sea
Actor	ES Library
Objective	Having code responsible for handling state-changing requests
Preconditions	· User request for state change
Postconditions	· An event is created if the business logic and current state allow it
Main Scenario	1. State changes reach the system 2. Command Handler uses current state and business logic to check if the operation is possible 3. New event is created
Alternative Scenario	3. The operation is aborted because the state change was refused by the business logic

Table 5.1: REQ01 – Respond to state change requests

REQ02 – Change state with events	
Level	Sea
Actor	ES Library
Objective	Having code capable of applying state changes to aggregates and read models
Preconditions	· A new event is published in the system
Postconditions	· State changes are applied to state holders
Main Scenario	1. New published event is received by the event handler 2. Event handler applies state change to its state holder
Alternative Scenario	1. An error occurs and the event is not received by the event handler

Table 5.2: REQ02 – Change state with events

REQ03 – Store and publish events	
Level	Sea
Actor	ES Library
Objective	Have code capable of storing and publishing events to event handlers
Preconditions	· A newly created event
Postconditions	· Event stored in the event store and sent to respective event handlers
Main Scenario	1. Newly created event is received 2. Event is stored 3. Event is sent to the event handlers
Alternative Scenario	1. Error in storing event a. System logs error b. The system stops to avoid inconsistencies

Table 5.3: REQ03 – Store and publish events

REQ04 – Respond to queries	
Level	Sea
Actor	ES Library
Objective	Having code capable of responding to query requests from the user
Preconditions	· Query request
Postconditions	· Query response
Main Scenario	1. Query request is received 2. Query requested information to the data store 3. Return queried data
Alternative Scenario	2. Requested query returns an error a. Error is logged b. An error message is sent to the user

Table 5.4: REQ04 – Respond to queries

REQ05 – Update Read Data Store through the Event Store	
Level	Sea
Actors	Read Data Store, Event Store
Objective	Read Data Stores to be capable of updating their state in case of not having the latest state changes when a query is issued
Preconditions	· Last processed event is older than the latest event in the event store
Postconditions	· Read Data Store is updated
Main Scenario	1. Query request is received 2. Read data store compares the serial number of the last processed event with the latest event from the event store 3. Read data store request unprocessed events 4. Event store sends requested events 5. Read data store updates its state
Alternative Scenario	3. No update is required

Table 5.5: REQ05 – Update Read Data Store through the Event Store

REQ06 – Update Aggregate through the Event Store	
Level	Sea
Actors	API, Aggregate, Event Store
Objective	Aggregates to be capable of updating their state in case of not having the latest state changes when a state verification is issued
Preconditions	· Last processed event is older than the latest event in the event store
Postconditions	· Read Data Store is updated
Main Scenario	1. State verification is issued 2. The Aggregate compares the serial number of the last processed for that event with the latest event from the event store 3. Aggregate requests unprocessed events 4. Event store sends requested events 5. Event Handler updates the aggregates state
Alternative Scenario	3. No update is required

Table 5.6: REQ06 – Update Aggregate through the Event Store

REQ07 – Replicate Events Between Event Stores	
Level	Sea
Actors	Event Stores
Objective	Event store replicas to be capable of replicating data over each other to update event logs
Preconditions	· Event store replicas · Inconsistency between replicas
Postconditions	· Consistent event stores
Main Scenario	1. Event stores detect inconsistencies 2. Replicas replicate new data among them 3. Replicas become consistent
Alternative Scenario	3. Conflicts are detected a. Compensation actions are created

Table 5.7: REQ07 – Replicate Events Between Event Stores

REQ08 – Idempotency	
Level	Clam
Actors	Event Handlers, Read Data Stores
Objective	Not run the same event more than one time
Preconditions	· Previously processed event is sent to be reprocessed
Postconditions	· Actors do not re-run the already applied event
Main Scenario	1. Already processed event is received by the actors. 2. Actors verify the serial number of the last event and compare it with the one from the newly received event. 3. The new event number is equal to or lower than the number of the last processed event, so the actors reject the event execution.
Alternative Scenario	3. The new event number is higher than the number of the last processed event. a. Actors process the newly received event.

Table 5.8: REQ08 – Idempotency

REQ09 – Support multiple multi-region clusters	
Level	Sea
Actor	System Prototype
Objective	Support different regions with a low-latency service
Preconditions	· Functional individual system with data stores ready for replication
Postconditions	· Distributed system with multiple instances working as one
Main Scenario	1. Two users in different regions use the service. 2. Each uses the cluster corresponding to its region. 3. The changes made by both users can be seen in all clusters
Alternative Scenario	3.a. Connection to service is lost. 3.a.1 User regains control with the message that the transaction was not executed. 3.a.2 Any data changed before the failure is reverted. 3.b. Both users use the same data 3.b.1 Inconsistency is solved by compensation events. 3.b.2 One wait for the other to finish

Table 5.9: REQ09 – Support multiple multi-region clusters

REQ10_v1 – Write Operation	
Level	Fish
Actor	System Prototype
Objective	Write the new event in a primary event store so it can be replicated in other replicas
Preconditions	· A new event is created. · Connection to the Event Store
Postconditions	· The new event is stored in the event store and is replicated
Main Scenario	1. User requests a change of the current state. 2. A new event is created and published. 3. System returns control over to the user. 4. The new event is written in the local event store. 5. Event store replicates the new information
Alternative Scenario	4. Connection to the event store is lost. a. Any data changed before the failure is reverted. 5. The user queries before writing the event. a. User may query not updated information

Table 5.10: REQ10_v1 – Write Operation

REQ10_v2 – Write Operation	
Level	Fish
Actor	System Prototype
Objective	Write the new event in a primary event store so it can be replicated into other replicas
Preconditions	· A new event is created. · Connection to the Event Store
Postconditions	· The new event is stored in the event store and is replicated
Main Scenario	1. User requests a change of the current state. 2. A new event is created and published. 3. The new event is written in the local event store. 4. System returns control over to the user. 5. Event store replicates the new information
Alternative Scenario	3. Connection to the event store is lost. a. User regains control with the message that the transaction was not executed. b. Any data changed before the failure is reverted.

Table 5.11: REQ10_v2 – Write Operation

REQ10_v3 – Write Operation	
Level	Fish
Actor	System Prototype
Objective	Write the new event in a primary event store so it can be replicated in other replicas
Preconditions	· A new event is created. · Connection to the Event Store
Postconditions	· The new event is stored in the event store and is replicated
Main Scenario	1. User requests a change of the current state. 2. A new event is created and published. 3. The new event is written in the local event store. 4. Event store replicates the new information to a quorum. 5. System returns control over to the user.
Alternative Scenario	3. Connection to the event store is lost. a. User regains control with the message that the transaction was not executed. b. Any data changed before the failure is reverted

Table 5.12: REQ10_v3 – Write Operation

Requirement REQ10

The tenth requirement has three versions. These versions represent the conditions on which a write operation is performed concerning the user's interaction.

The objective of the versioning of REQ 10 is to implement it according to the requirement prioritization and go from version 1 to the third during development.

MoSCoW Analysis

Due to the number of requirements, a MoSCoW Analysis was made to prioritize these requirements. The objective is to have a clearer picture of how the development of this project is going to be managed during the dissertation period.

The criteria used were: selecting the requirements associated with what was promised as replicated application as *must-haves*, requirements associated with iterative improvements as *should haves*, and as *could have* the last version consistency implementation on a replicated application.

The analysis was the following:

- **Must Have:** REQ01 (table 5.1), REQ02 (table 5.2), REQ03 (table 5.3), REQ04 (table 5.4), REQ05 (table 5.5), REQ06 (table 5.6), REQ07 (table 5.7), REQ08 (table 5.8), REQ09 (table 5.9), REQ10_v1 (table 5.10)
- **Should Have:** REQ10_v2 (table 5.11)
- **Could Have:** REQ10_v3 (table 5.12)

5.2 Constraints

Constraints are imposed limits that the architecture must abide by where does not exist major flexibility to change them. This section presents the constraints of this project.

5.2.1 Technical Constraints

Technical constraints are the limits imposed on the technologies and concepts used on the project and are the ones with most impact in the architecture.

Identified Technical Constraints:

- Concepts to use:
  - Description: In this project, the concept of ES has to be studied and used
  - Flexibility Points: None
  - Alternatives: None
- High Availability Goal:
  - Description: In this project, to achieve high availability concepts as replication must be implemented
  - Flexibility Points: None
  - Alternatives: None

5.2.2 Business Constraints

Identified Business Constraints:

- Development Time:
  - Description: The development time of this project coincides with the end of the dissertation time frame
  - Flexibility Points: The deadline of the dissertation can be prolonged.
  - Alternatives: None

5.3 Final Remarks

This chapter examined the architectural drivers that are the basis for the design of the system. Through the exploration of requirements and constraints, it was established a clear understanding of the functional and the limiting aspects that

shape the architecture. The functional requirements regard the event sourcing operations and the related steps to ensure a consistent system and the constraints regarding concepts, objectives and time for the dissertation's work.

Chapter 6

Architecture

This chapter's purpose is to describe the proposed system architecture. The first section's goal is to visually describe the architecture, and the last section of this chapter is reserved for the compliance analysis of the architecture with the requirements in Chapter 5.

6.1 C4 Diagrams

To visually represent the system, the C4 model was chosen to represent and visualize the architecture. Here three different diagrams are going to be presented, each one more in-depth and in detail than the last: Context Diagram, Container Diagram and Component Diagram.

6.1.1 Context Diagram

The context diagram is the most abstract created. Its purpose is to show who and what interacts with the system. Figure 6.1 (legend figure 6.2) demonstrates that the only interaction of the system is with a user. The user can perform operations such as queries or data changes. There are no interactions with external systems.

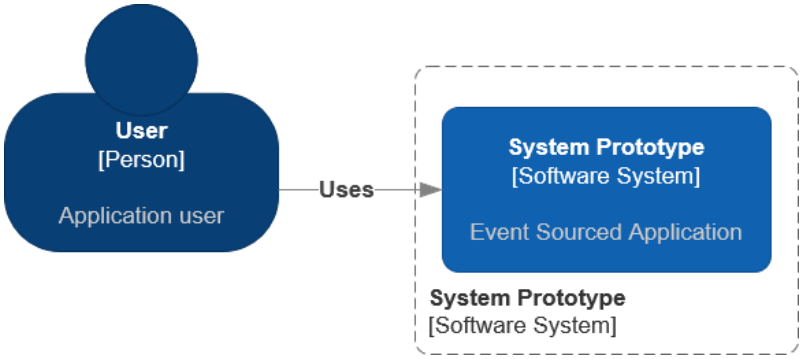


Figure 6.1: System Context

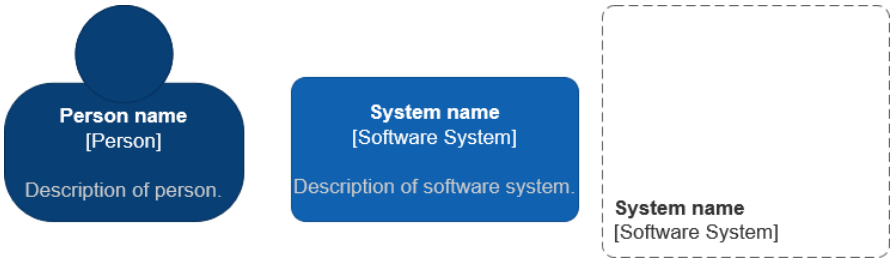


Figure 6.2: Legend for System Context

6.1.2 Container Diagram

The given container diagram is a zoom in the software system. Its purpose is to show what elements constitute the system, how they interact with each other and some choices regarding technologies to be used.

Figure 6.3 (legend figure 6.4) is the proposed container diagram. It demonstrates what constitutes the System Prototype. There are multiple replicas of the system that work as a whole.

The user interacts with the system with API calls to one replica (correspondent to its region). The API uses the Axon Framework to use ES and makes queries to the Read Data Store with JPA. The Event Store is responsible for storing all events that can be used to reconstruct and update the state.

The Axon Framework is responsible for handling the event creation, event-related state changes and store events. The framework is also for constructing and updating the current state of the Read Data Store.

The Axon Server is necessary for the Axon Framework to work and manage all Event Sourcing operations. It is the engine that makes ES and the framework function. The chosen version for the Axon Server is the free Standard Edition that, by it self, does not support replication.

MongoDB is used for the Event Store, and an in-memory H2 is used for the Read Data Store. The MongoDB replication capabilities are responsible for the replication and consistency between event stores. In a replicated environment, there is more than one instance in the MongoDB Replica Set. Because of how MongoDB replication works (section 3.3), the writing of new events is done on the primary Event Store replica and then replicated to the secondary replicas, but the application will read from the closest instance by setting Mongo's Replica Set read preference do that.

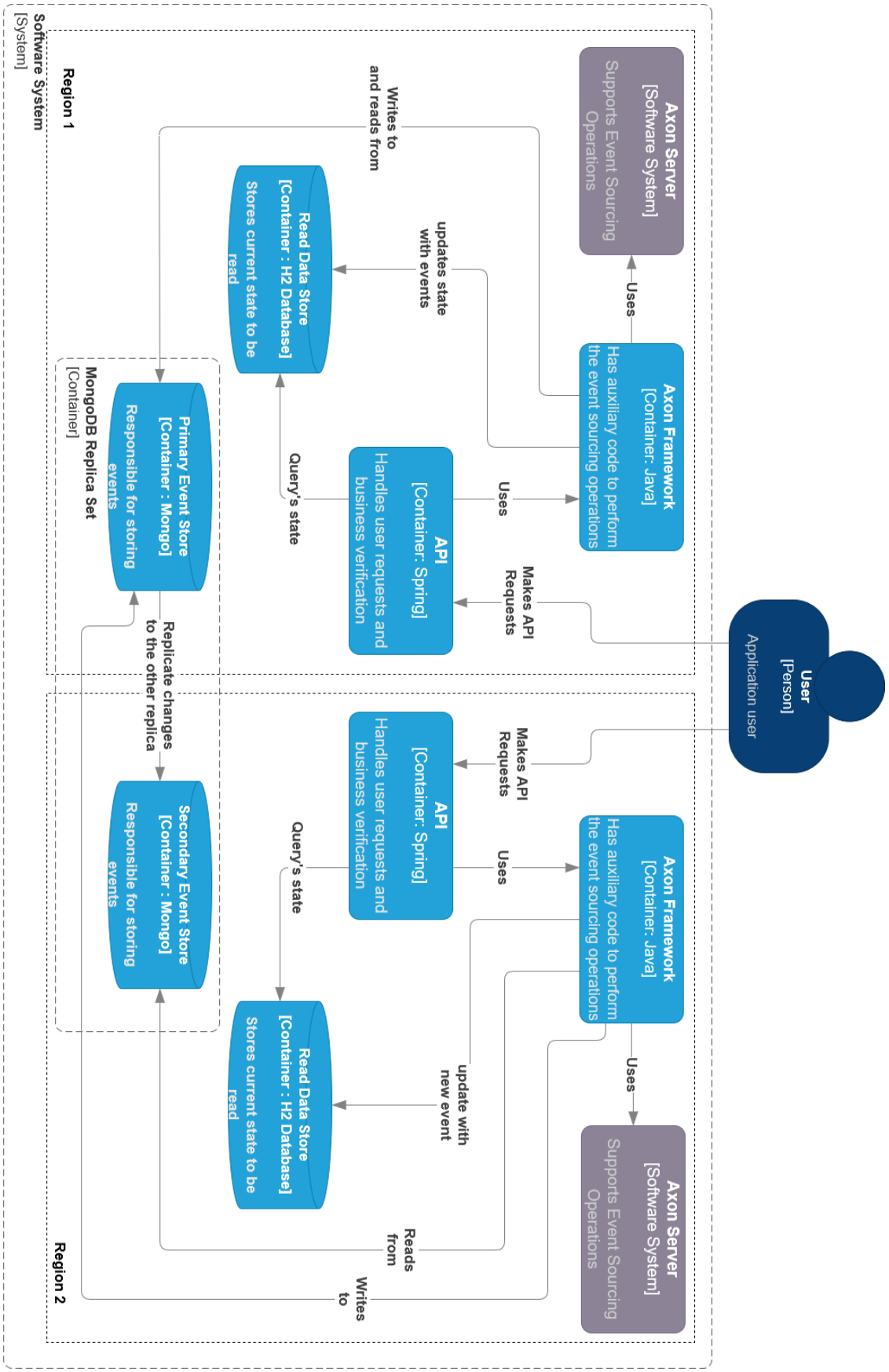


Figure 6.3: Container Diagram

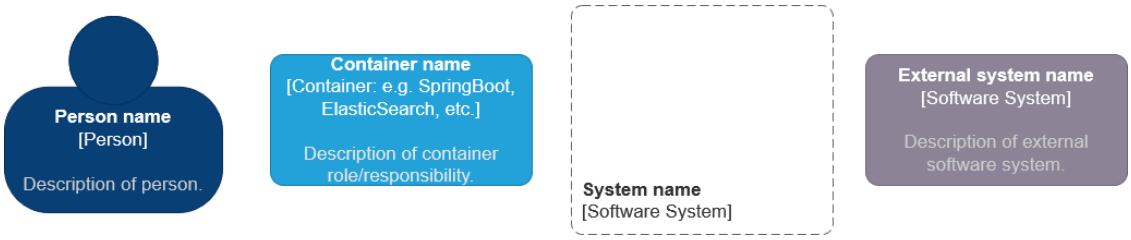


Figure 6.4: Legend for Container Diagram

6.1.3 Component Diagram

The following diagram (figure 6.6) is a zoom of the API and the Axon Framework to see the components involved in the system's event sourcing procedure. It was the experimentation done with Axon and how the framework works that led to the selection of these components and their responsibilities.

The API is constituted by a Rest Controller, the Domain Model and a JPA Repository. The Rest Controller receives the user's requests, the Domain Model contains all business logic and data to perform data changes and the JPA Repository is used to respond to queries.

The Axon Framework contains the Command Handler, Apply component and Event Handlers.

If the Rest Controller receives a command to check the state (for example, checking balance), the Rest Controller uses JPA to make a query to the database.

If the Rest Controller receives a command to change state, it uses the Command Handler. The Command Handler uses an Aggregate to check if the user's intent is possible. If business logic and the current state allows for a new change, a new event is created, and the Apply component is used to publish it. If not, nothing changes.

The apply component is responsible for storing the event store and publishing it to the event handlers. The Aggregate's Event Handler applies the data change from the event to the Domain Model, and the Read Data Store Event Handler applies the data change from the event to the data in the read database.

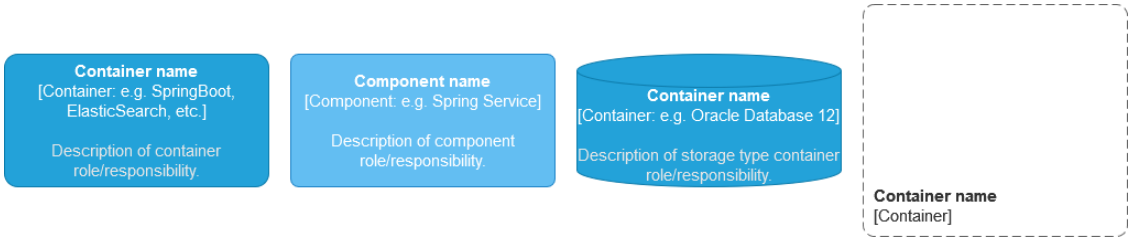


Figure 6.5: Legend for Component Diagram

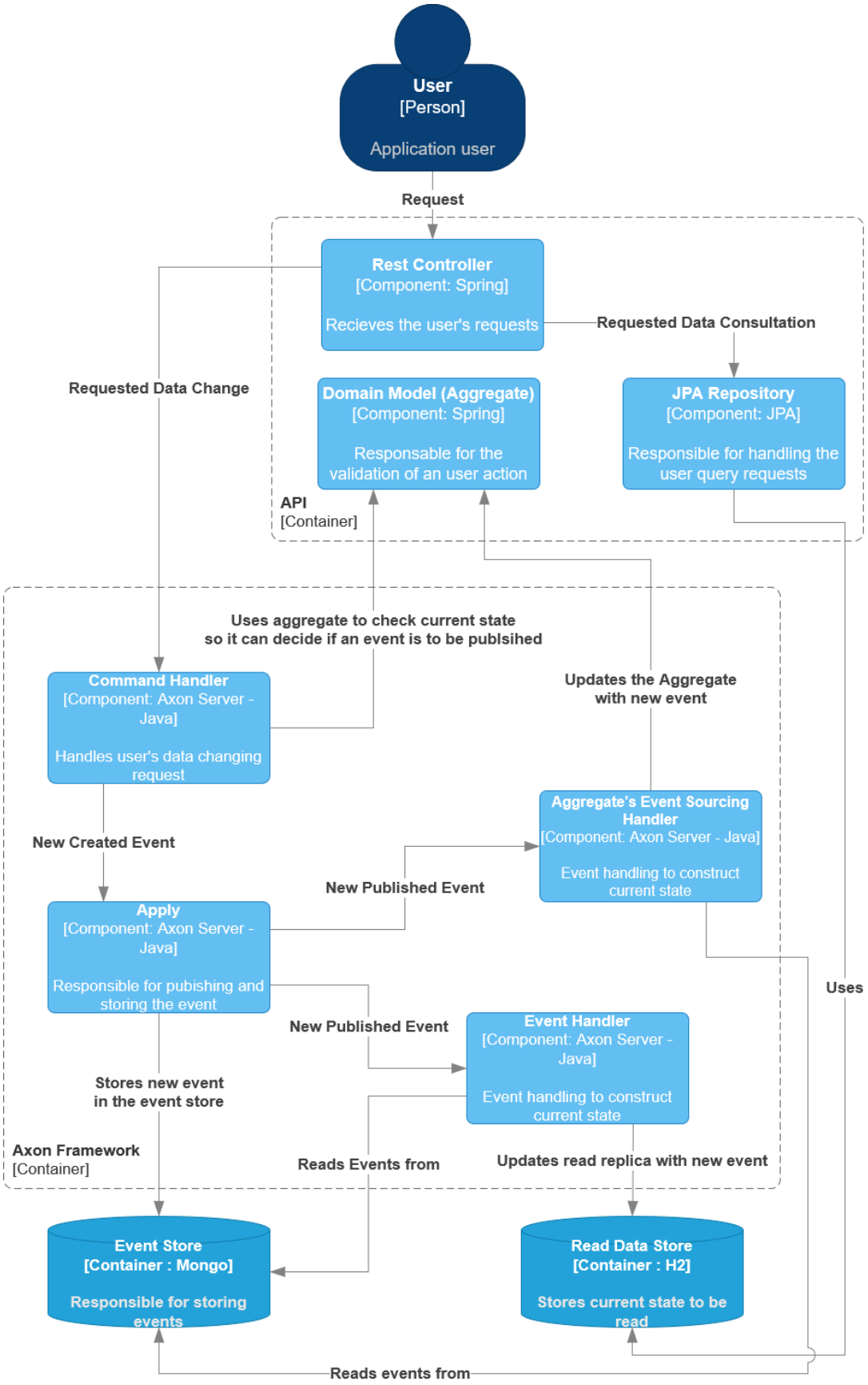


Figure 6.6: Component Diagram

6.2 Analysis

The proposed architecture is analysed in this section to check if it complies with the defined requirements presented in chapter 5.

- **REQ01 – Respond to state change requests:** is accomplished through the components Rest Controller, which receives the users' requests, and Command Handler, which checks the current state and business logic to legitimize the creation of an event.
- **REQ02 – Change state with events:** is accomplished through Event Handlers that apply the changes on state holders (Domain Model (Aggregate) and Read Data Store).
- **REQ03 – Store and publish events:** The Apply component responsibility is to write on the Event Store and publish the events to the Event Handlers.
- **REQ04 – Respond to queries:** is accomplished through the components Rest Controller, which receives the users' requests, and through JPA responds to the query.
- **REQ05 – Update Read Data Store through the Event Store:** Axon is used for this requirement to establish a connection between the Read Data Store and Event Store.
- **REQ06 – Update Aggregate through the Event Store:** is accomplished through Event Handlers requesting and applying the changes of the events from the event store.
- **REQ07 – Replicate Events Between Event Stores:** is achieved through the use of MongoDB replication capabilities.
- **REQ08 – Idempotency:** Responsibility of Event Handlers.
- **REQ09 – Support multiple multi-region clusters:** is achieved by having different instances of the application running with the connected MongoDB event stores.
- **REQ10 – Write Operation:** The apply component is responsible for writing the events in the Event Store, but the moment the user regains control is determined based on the requirement version.

6.3 Final Remarks

This chapter delved into the blueprints of the system with the C4 architecture model. The several C4 diagrams depict the system's key components, interactions, and responsibilities, establishing a robust visualization of the system's structure at different levels of detail. These models guide the development efforts, promoting a structured and coherent implementation that aligns with the vision outlined in this chapter.

Chapter 7

Experimental Setup

This chapter presents and describe the application, the preparation and the different setups used for the experiments, and how they were conducted to assess the performance of the Spring Boot + Axon Application with MongoDB as an event store under various setups and scenarios.

7.1 Application

The system used in this experiment uses the concepts of Event Sourcing, CQRS and Sagas (chapter 2). In this system, the application is a Spring Boot Application that uses Axon Framework, powered by Axon Server Standard Edition, that uses MongoDB as the event store and an in-memory H2 Database as the read model. The application is intended to simulate a banking app, and Sagas are implemented for transactions between accounts. Axon's snapshot mechanism is also set as default.

The following are the versions of the different components used on the system:

- Java version: 11
- Spring Boot version: 2.7.3
- Axon Framework version: 6.3
- Axon Server version: 2023.1.0
- MongoDB version: 6.0.8

7.1.1 Application Description

As a simplified banking system, the application is capable of creating clients and accounts, allows for the operations of deposits and withdraws, permits transfers between accounts and offers balance and client information checking through general queries.

Client Creation

The client creation operation results in the creation of an event that represents the registry of a new client.

Account Creation

The account creation requires a registered client to be created. A command to create an account is processed by the client aggregate (subsection 3.1.3 - Command Model (Aggregate)) and results in the publishing of an event that represents the creation of the account.

Deposit and Withdrawal Operations

With a created account, the operations of deposit and withdrawal are possible. In Axon, every time a command is received, the current state for the command's specific aggregate is reconstructed. For each deposit or withdrawal, the account state is reconstructed from the events referring to that account to get the balance, so the aggregate can use business rules to see if a deposit or withdrawal is possible and publish the event referring to a value addition or removal. The deposit operations have no business verifications, but for the withdrawals, the balance needs to be equal to or higher than the value of the operation.

Transfer Between Accounts

The money transfer between accounts is possible due to the use of Sagas. When a command requiring a transfer between two accounts is processed, a transfer event is published that starts a Saga. This Saga states and manages the sequence of events, alongside compensatory actions in case of failures, that are required to perform a complete money transaction. Through the exchange of commands and events, the Saga communicates with the aggregates of both accounts, first to check the existence of the receiving account, next to withdraw the money from the sending account and then to deposit the transfer value into the receiving account, finishing the Saga. Each step moves to the next after a confirmation event from each account aggregate. If anything fails, the Saga has a deadline that upon a transaction not finishing within a time frame, compensatory events are sent to both account aggregates to null the changes.

Queries

For user reading operations, the application offers two general query options to get the current state related to all the clients and all the accounts registered in the application. This information is in the Read Data Store and is returned as a JSON file.

Conclusion

It is relevant to note that, since it is an event-sourced Axon application, every time a user wants to perform a state change, a command is generated for the application to process, which leads to the current state being reconstructed every time from the events stored in the Event Store.

For the experiments, it was decided to use the deposit action as the write opera-

tion and the accounts information query as the read operation.

7.2 Cloud

To test the system, it was uploaded to Amazon Web Services (AWS) by having each part of it a devoted EC2 machine across three different setups and two different regions (Stockholm-EU and North Virginia-USA). The configurations of these machines were done by hand, without scripts.

The machines chosen were the t3.medium for the Spring Boot App with Axon and H2, and t3.micro for Axon Server and MongoDB, all running Ubuntu 22.04. The t3.medium machines have 2 vCPUs and 4 GiB of memory and the t3.micro machines have 2 vCPUs and 1 GiB of memory.

The load testing tool Locust [3] was used to stress test and get analytics. Each region is an EC2 t3.medium machine running Locust to flood the event-sourced system at every test to record the time needed to handle requests.

Here is the list of EC2 two machines used in each region along with their type and Elastic IP (fixed IP service by AWS):

- EUA N. Virginia:
  - AppSpringBootAxon_USA: - t3.medium - 3.213.107.10
  - AxonServer_USA: - t3.micro - 34.203.142.197
  - LocustTester_USA: - t3.medium - 35.170.157.0
  - Setup2_MongoDB3: - t3.micro - 3.233.184.125
- EU Stockholm:
  - AppSpringBootAxon_EU - t3.medium - 13.50.124.60
  - AxonServer_EU - t3.micro - 13.51.41.205
  - LocustTester_Stockholm - t3.medium - no Elastic IP due to region limit
  - Single_MongoDB - t3.micro - 13.50.136.213
  - Setup2_MongoDB1 - t3.micro - 16.171.188.51
  - Setup2_MongoDB2 - t3.micro - 13.49.255.47

7.2.1 Setup 1

This first setup (fig 7.1) has four EC2 machines: a non-replicated single event store, the Spring Boot + Axon application with in-memory H2, an Axon Server and Locust.

The operating procedure is Locust making requests to the App and having it respond to them by using Axon's technologies for the event sourcing capabilities and writing and reading events from the single instance of Mongo.

The purpose of this setup is to get data related to reads and writes in a non-replicated event store within the same region.

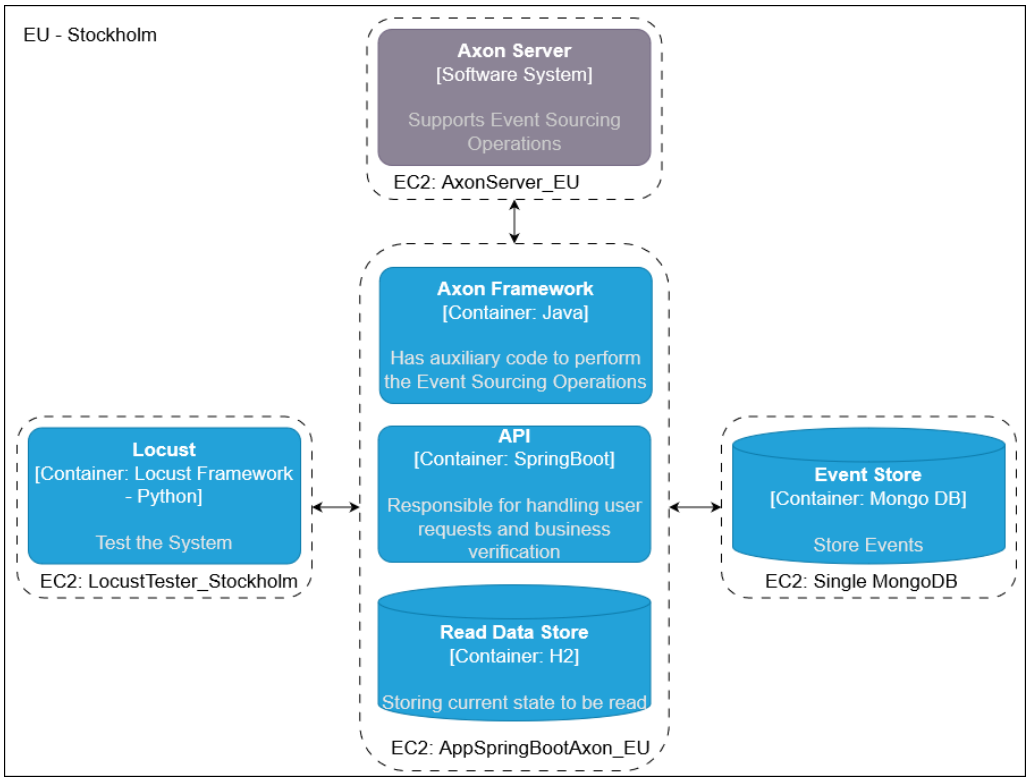


Figure 7.1: Experimental Setup 1

7.2.2 Setup 2

This setup (fig 7.2) has nine EC2 machines across North Virginia and Stockholm. Both regions have Locust, App and Axon Server machines. The event store in this layout is replicated across three MongoDB instances, with two members of the replica set in Stockholm (including the primary) and only one in North Virginia. The Mongo replica set is also configured to have the closest replica to respond to the queries [50], so the App of Stockholm loads events from one of the replicas in Stockholm and the App from North Virginia loads from North Virginia.

The operating procedure is, in each region, Locust making requests to the App and having it respond to them by using Axon's technologies for the event sourcing capabilities. Due to how MongoDB replication works (section 3.3), the writing of new events is performed on the primary replica and then replicated to the secondary replicas, so the Apps from both regions will write on the primary replica in Stockholm.

The purpose of this setup is to get data related to reads and writes in a replicated

event store across different regions by having Locust making requests from both sides.

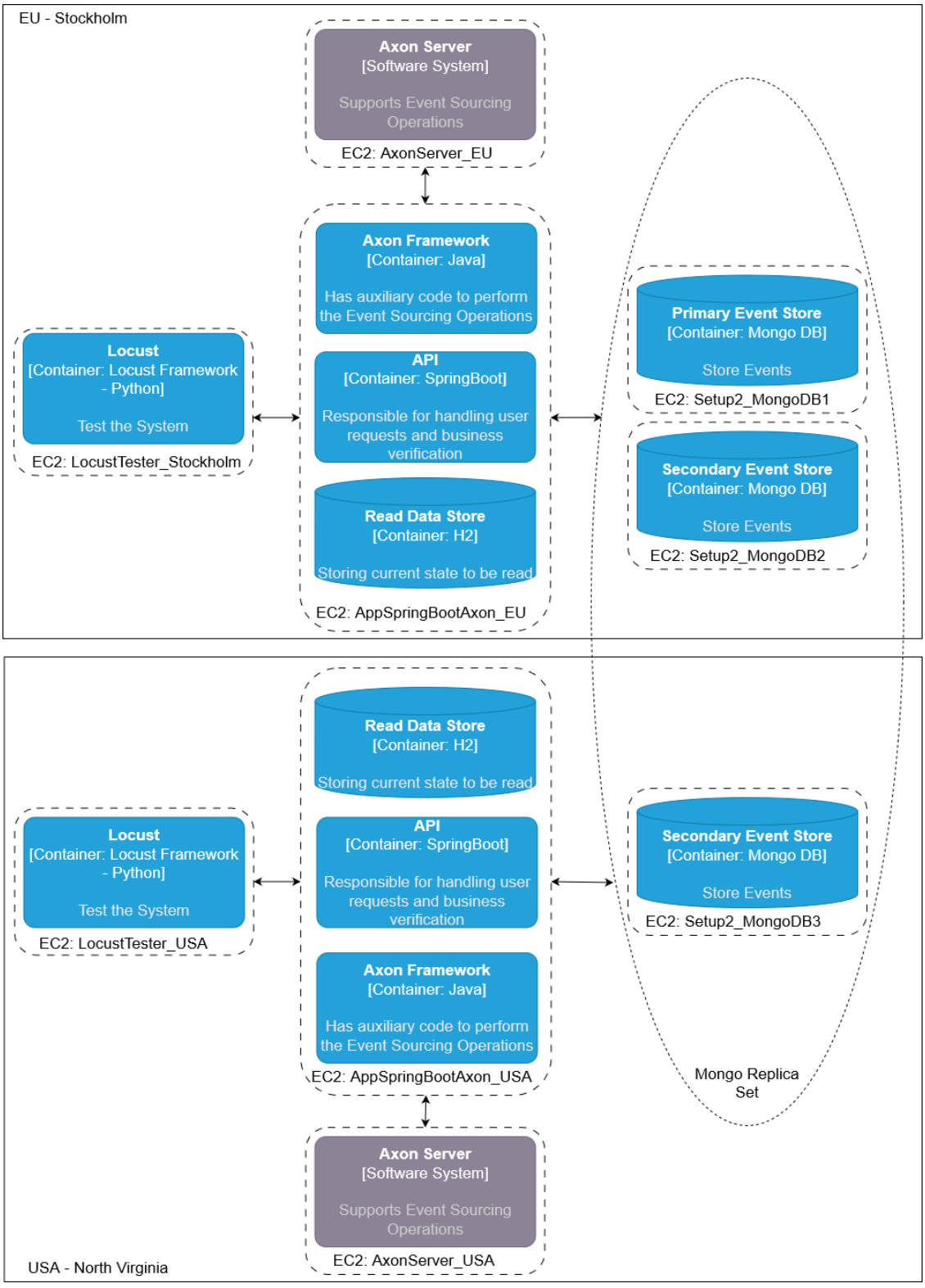


Figure 7.2: Experimental Setup 2

7.2.3 Setup 3

This setup (fig 7.3) has four EC2 machines: the Spring Boot + Axon application with in-memory H2, an Axon Server, Locust and a non-replicated single event store in the other region.

The operating procedure is Locust making requests to the App and having it respond to them by using Axon's technologies for the event sourcing capabilities and writing and reading events from the single instance of Mongo.

The purpose of this setup is to get data related to reads and writes in a non-replicated event store in a different region.

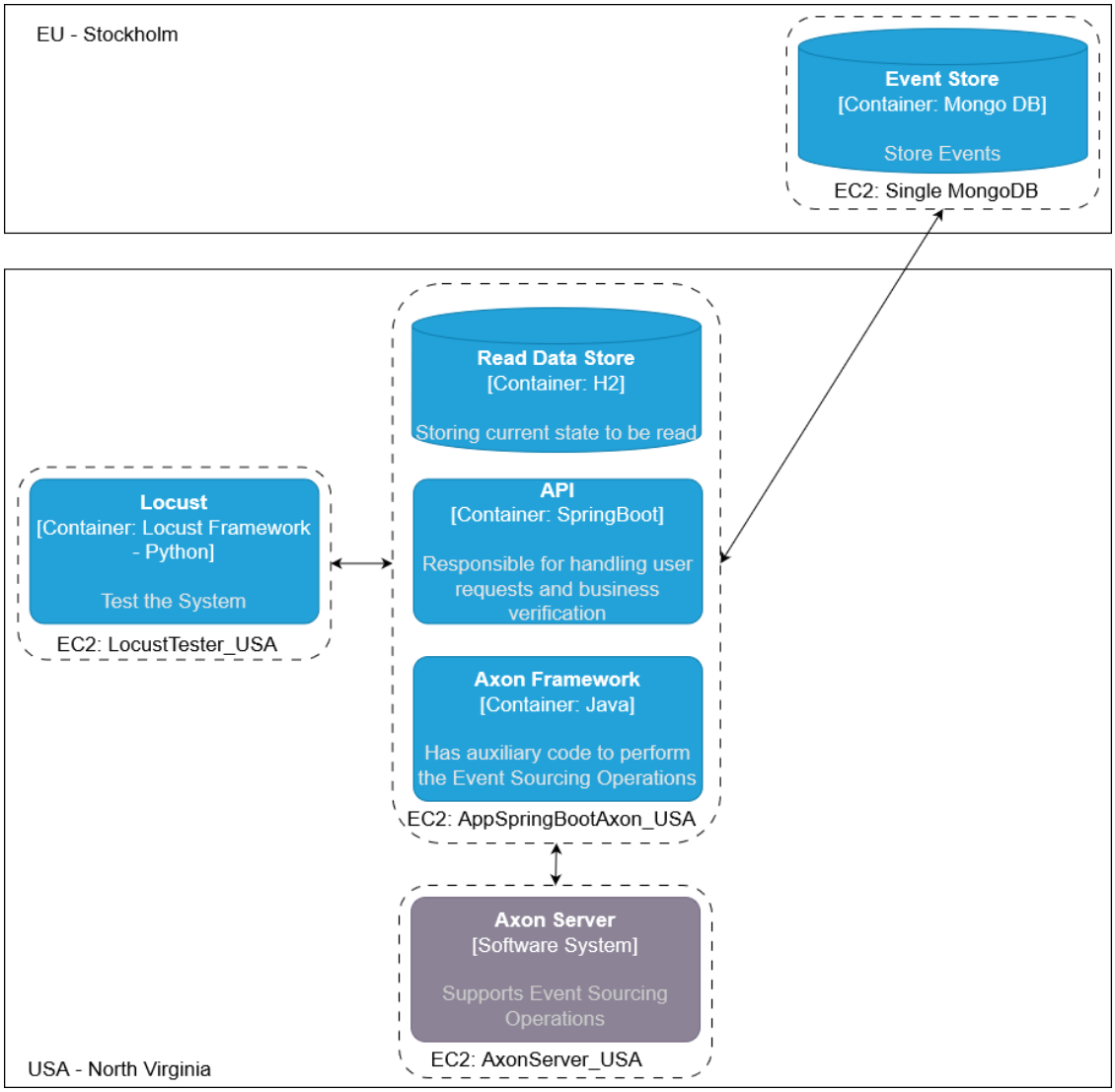


Figure 7.3: Experimental Setup 3

7.3 Experiment Methodology

The experiment process encompassed a systematic approach to evaluate the response times for specific operations and investigate the impact of different configurations on the system's performance. The experiments focused on two key operations: a writing operation (deposit) and a reading operation (read the account details).

The performance evaluations were performed using the Locust load testing framework. Locust allows the creation of virtual users that interact with the application as per specified tasks. Each experiment has two primary operations: deposit (writing) and reading account details (reading). The Python code used in the experiments, in the listing 7.1, exemplifies this. Also, the reading operation cadence was double the writing operation's cadency because it was assumed that there would be more reading operations than writings for this case.

```
import time
from locust import HttpUser, task, between

class QuickstartUser(HttpUser):
    wait_time = between(1, 5)

    @task(2)
    def account_list_request(self):
        self.client.get("/AccountStatus")

    @task(1)
    def deposit_Money(self):
        self.client.post("/deposit",
           json={"accountID":"153ba28f-27a1-4bcb-ba3c-d01358ce
        15b7", "amount": 100})

    def wait(self):
        # Function to add a custom wait/ramp-up time
        import time
        ramp_up_time = 40  # Desired ramp-up time in seconds
        time.sleep(ramp_up_time)

    def on_start(self):
        # This function is called when a user starts running
        # Used to add a ramp-up time before any metrics are
           collected
        self.wait()  # Adjust the wait time here to set
           ramp-up time
```

Listing 7.1: Locust Python Code

7.3.1 Testing Scenarios

Each test consisted of three stages, each utilizing varying numbers of virtual users: 100 users, 500 users, and 1000 users. These steps were executed over a fixed duration of 15 minutes, with 40 seconds of ramp-up time. Also, the database was reset before every test to have the same starting point. The number of users and the duration time were set on the Locust UI before starting the experiment. The ramp-up time is defined on the code in the listing 7.1.

The experiments were categorized into the three setups, each with specific testing scenarios:

Setup 1: EU Reads and Writes

In the first setup (fig 7.1), the performance of the SpringBoot Application with MongoDB is examined under a baseline scenario. Locust was used to measure the response times for the deposit and reading operations while employing users for both (write and read) operations. This approach helped establish a reference point for subsequent analyses.

Setup 3: USA Reads and Writes with EU Database

In this setup (fig 7.3) is similar to the first and involved testing the system's performance under the condition that the application was situated in North Virginia (USA) while the data storage was in Stockholm (EU). The objective was to measure the effects of data storage across geographical regions while comparing it to "Setup 1: EU Reads and Writes".

Setup 2: Variations in EU and USA Operations

The second setup (fig 7.2) encompassed a more diverse range of scenarios. Here, we investigated the impact of geographical distribution on the system's performance. Five distinct combinations were evaluated:

- EU Reads and Writes
- USA with Reads and Writes
- EU and USA with Reads and Writes
- EU with Reads and USA with Writes
- USA with Reads and EU with Writes

These five combinations were tested by running the Locust machines with variations on the Python code in listing 7.1 by running either both read and write operations or only performing "def account_list_request (self)" or "def deposit_Money (self)".

This setup aimed to provide insights into how the location of operations and data storage influenced response times.

7.4 Final Remarks

This chapter outlined the methodology for evaluating the performance of the event-sourced SpringBoot Application with MongoDB as the event store. Through controlled experiments utilizing Locust, it is possible to investigate the response times of deposit and reading operations under various scenarios. The approach encompassed a range of user loads and geographical distributions, enabling a broad analysis of system behaviour.

In total, learning the AWS tools, configuring and arranging all experiments, as well as conducting various tests for all the experiments incurred an expense close to $45 on the AWS platform.

Chapter 8

Results and Discussion

In this chapter, the experiment's outcome will be analysed and discussed. The analysed data is the mean of the response times of the writing and reading operations performed on the different setups presented in the previous chapter. The data is displayed in the form of linear graphics and tables. The linear graphics are colour-coded regarding the region and operation: red for writes and blue for reads from Europe, magenta for writes and cyan for reads from the USA.

8.1 Application Correctness

The experiments proved that the application worked in both the US and Europe and together across the ocean. After the tests, the information queried to the application was the same from both continents, showing that the shared event storage with different query storage in each region was working.

Another noteworthy aspect of Event Sourcing experienced during the experiments was eventual consistency (section 2.2 - Issues of Event Sourcing). This consistency model was spotted when manually querying the application's state from the different regions at the end of an experiment to verify the correctness of the system. After the more demanding tests, the query models need time to catch up and update their state with all the new events. So when several queries were made in both regions immediately after an experiment had finished, the queries showed the read data storage being updated.

8.2 Setup 1: EU (Writes and Reads)

The experiment of this section is about the performance tests on the first setup presented in the previous chapter in subsection 7.2.1. In this experiment, the system is entirely on AWS's Stockholm region and has only one instance of MongoDB as the event store and is meant to be the reference point for the other experiments.

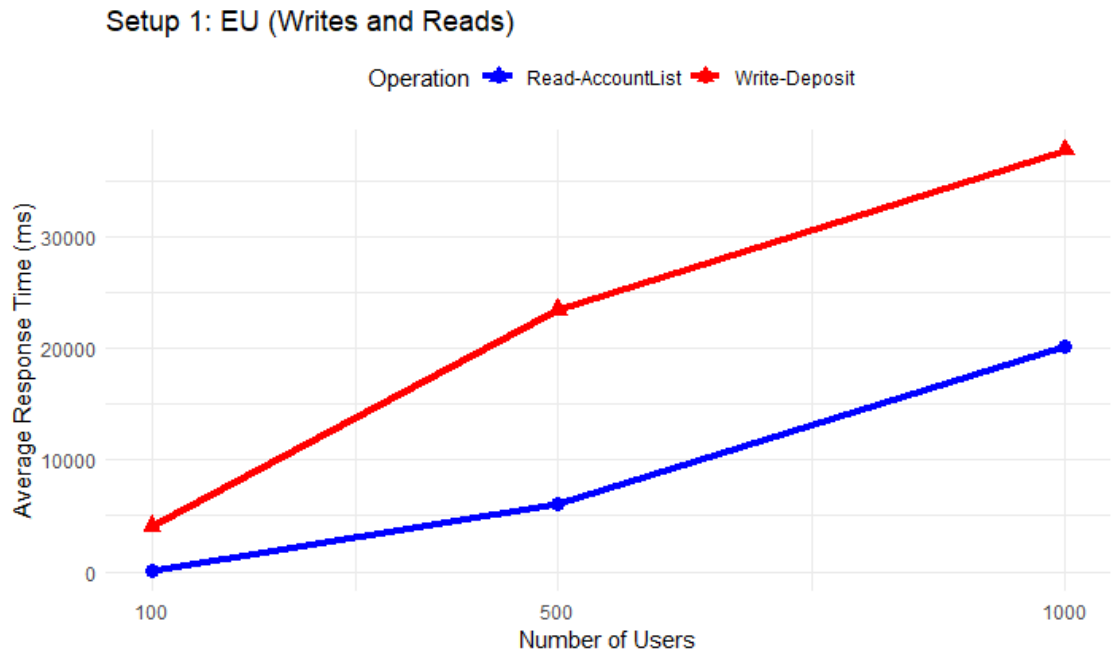


Figure 8.1: Setup 1: EU (Writes and Reads)

Setup 1: EU (Writes and Reads)		
Number_of_users	ReadAccountList (ms)	WriteDeposit (ms)
100	66	4092
500	6030	23453
1000	20127	37687

Table 8.1: Setup 1: EU (Writes and Reads)

The data shown in figure 8.1 and table 8.1 is the average response times for the read and write operations on Setup 1 for 100, 500 and 100 users. This first experiment shows how fast the read operations are in relation to the writing operations, showing the effect of CQRS because, with this pattern, the queried data does not need to be constantly recreated from the event store, unlike the write operation.

An expected point that the data shows is the increment of response times as more users join in, including the reading operations, as the application is overwhelmed with requests.

8.3 Setup 3: USA (Writes and Reads) with EU database

The experiment of this section is about the performance tests on the third setup presented in the previous chapter in subsection 7.2.3. In this experiment, the system has the application on AWS's North Virginia region and has only one instance of MongoDB in Stockholm as the event store.

The data shown in figure 8.2 and table 8.2 shows similar behaviour to "Setup 1" (section 8.2) regarding the reading times being lower the writing ones and the

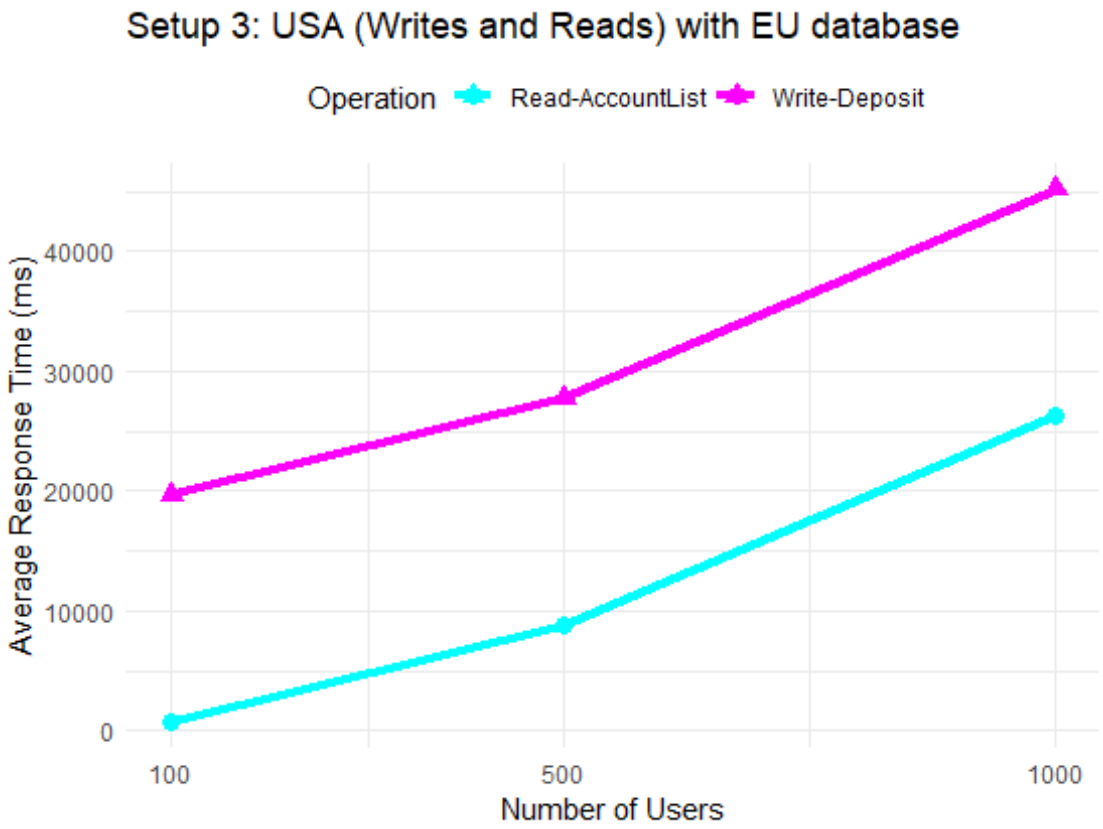


Figure 8.2: Setup 3: USA (Writes and Reads) with EU database

Setup 3: USA (Writes and Reads) with EU database		
Number_of_users	ReadAccountList (ms)	WriteDeposit (ms)
100	724	19650
500	8883	27747
1000	26257	45129

Table 8.2: Setup 3: USA (Writes and Reads) with EU database

increment of these times as more users are added. The difference is that they are slower. Since the event store is across the Atlantic Ocean, it is expected that the write operations will be slower, leading to an overall slower application as more resources from the application's EC2 machine are used between waiting for the event store and handling new requests.

8.4 Setup 2

The experiments of this section are about the performance tests on the second setup presented in the previous chapter in subsection 7.2.2. In these experiments, the system has the application on both AWS's North Virginia and Stockholm regions and a replica set with three instances of MongoDB as the event store, two (including the primary) in Europe and one in the USA. The primary replica is Europe, meaning that writing operations are performed in Stockholm. However,

every event store query is done on the nearest instance, which means that the application in Europe reads from one of the European instances, and the USA's applications upload events from the American replica.

8.4.1 Setup 2: EU (Writes and Reads)

This experiment is similar to the Setup 1's (section 8.2). Both setups have applications in Europe and are tested with writing and reading operations. The difference is that instead of having just one instance of MongoDB, this setup has a replica set with two replicas in Europe and one in the USA.

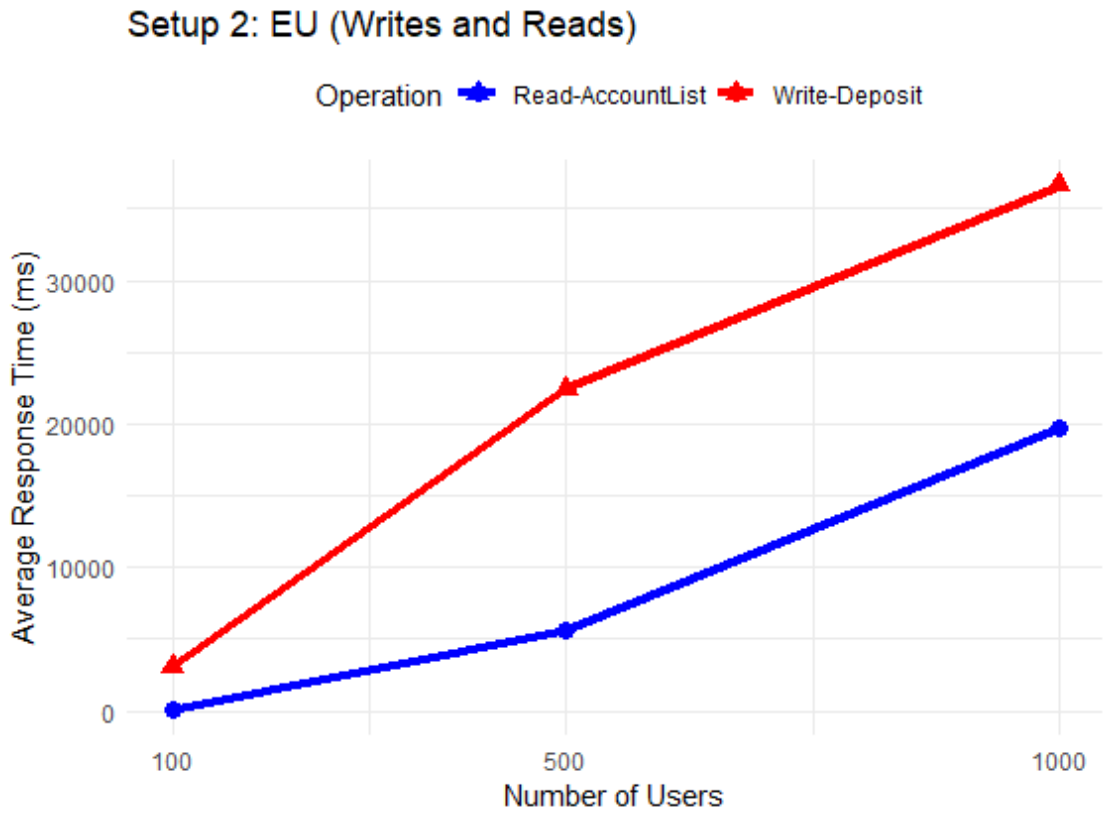


Figure 8.3: Setup 2: EU (Writes and Reads)

Setup 2: EU (Writes and Reads)		
Number_of_users	ReadAccountList (ms)	WriteDeposit (ms)
100	55	3049
500	5568	22439
1000	19710	36604

Table 8.3: Setup 2: EU (Writes and Reads)

The data shown in figure 8.3 and table 8.3 shows identical behaviour to Setup 1 (section 8.2) but slightly faster. When comparing the tables 8.1 and 8.3, "Setup 2" with the replica set beats "Setup 1" every time in both reads and writes with a difference in around 1 000 ms on writing operations. This behaviour may be explained by the application's EC2 machine having to wait less for the replicated event store than a non-replicated one, and since the application spends less time waiting, fewer resources are being consumed at a time for the writing operations, and more are available for handling new requests.

The resource hypothesis is also supported when analysing the ratio of the total number of requests that each experiment processed per second during the testing phase. Through table 8.4 is possible to compare the throughput of requests that the experiments "Setup 1", "Setup 3" (section 8.3), and "Setup 2: EU (Writes and Reads)" had during testing. The "Setup3" experiment, as the slowest experiment of the three, has the lowest ratio of processed requests/s and the "Setup 2: EU (Writes and Reads)", as the fastest, is the one with the highest request/s coefficient of the three.

Total Throughput (total requests per second)			
Number_of_Users	Setup 1	Setup 3	Setup 2: EU (Writes and Reads)
100	21,87	9.37	23.35
500	32.27	26.09	33.55
1000	32.40	26.35	33.03

Table 8.4: Total Throughput Comparison between Setup 1: EU (Writes and Reads), Setup 3: USA (Writes and Reads) and Setup 2: EU (Writes and Reads)

8.4.2 Setup 2: USA (Writes and Reads)

This experiment is similar to the Setup 3's (section 8.3). Both setups have applications in the US and are tested with writing and reading operations. The difference is that instead of having just one instance of MongoDB, this setup has a replica set with two Mongo instances in Europe and one in the USA. As the primary Mongo instance is in Stockholm, all writing operations are done in Europe, but every event reading operation is done in North Virginia.

Setup 2: USA (Writes and Reads)		
Number_of_users	ReadAccountList (ms)	WriteDeposit (ms)
100	729	19690
500	8972	27777
1000	26875	45746

Table 8.5: Setup 2: USA (Writes and Reads)

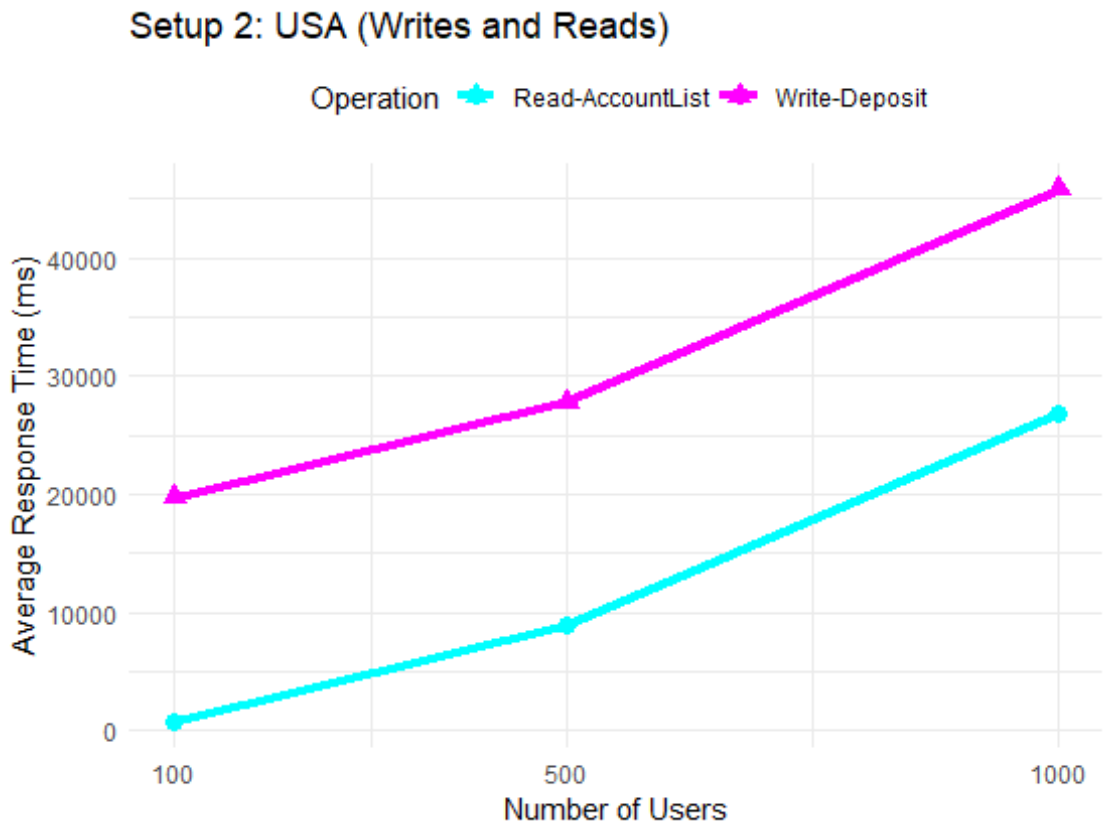


Figure 8.4: Setup 2: USA (Writes and Reads)

The data shown in figure 8.4 and table 8.5 reveals essentially the same behaviour as "Setup 3" (section 8.3). This setup is slower than the previous "Setup 2: EU (Writes and Reads)" experiment (section 8.4.1), its European equivalent, just like Setup 3 is slower than Setup 1 since the writing operation is performed in the primary event store that is situated in Europe.

8.4.3 Setup 2: EU (Writes and Reads) and USA (Writes and Reads)

This experiment puts both regions under reading and writing operations. Both regions have applications tested with writing and reading operations and have the Mongo replica set with two Mongo instances (including the primary) in Europe and one in the USA. Both applications load events from the closest Mongo replica, so the Europe application reads events from one of the European Mongo instances, and the US application reads events from the American replica.

Setup 2: EU (Writes and Reads) and USA (Writes and Reads)				
N.Users	Read_EU (ms)	Write_EU (ms)	Read_USA (ms)	Write_USA (ms)
100	72	4401	3243	19706
500	5959	23598	15516	30889
1000	20601	38294	39463	55101,02

Table 8.6: Setup 2: EU (Writes and Reads) and USA (Writes and Reads)

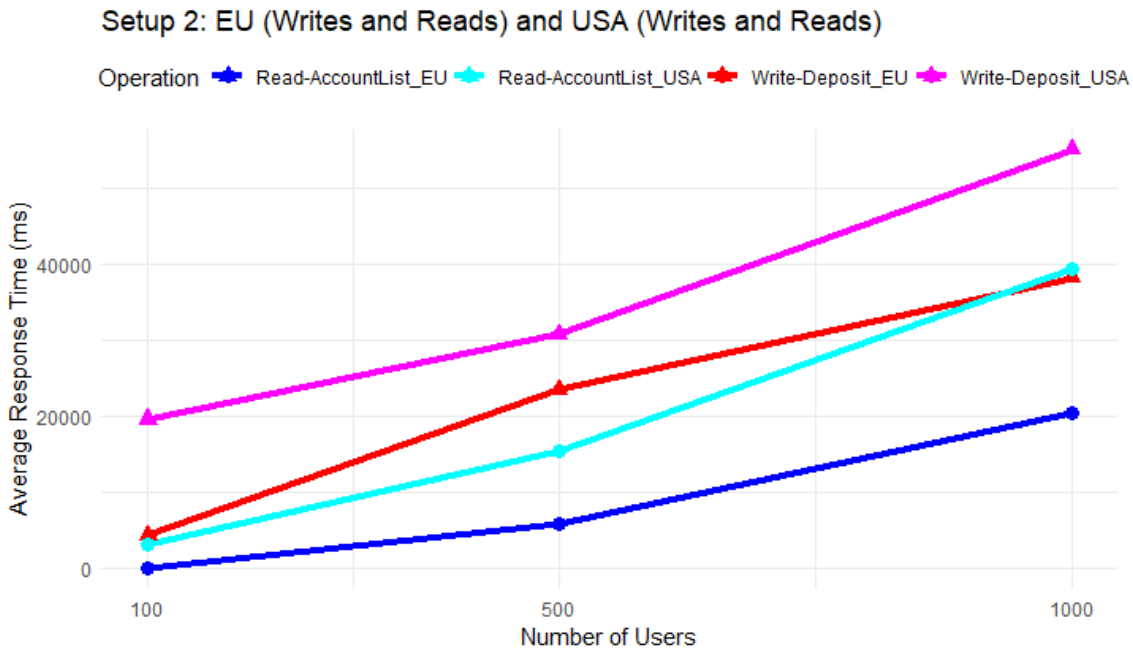


Figure 8.5: Setup 2: EU (Writes and Reads) and USA (Writes and Reads)

Figure 8.5 and table 8.6 show that, in comparison with all previous experiments, this one got the slower results. From Europe's side, although slower than "Setup 1" (section 8.2) and "Setup 2: EU (Writes and Reads)" (section 8.4.1) experiments, the response times are still close, but for the American's side the difference is bigger. The write and read times on the are much slower when comparing with "Setup 3" (section 8.3) or "Setup 2: USA (Writes and Reads)" (section 8.4.2). The differences for 1 000 users in write times can reach 10 000 ms, and in read times can get as far as 13 000 ms.

These results hint that when overwhelming both sides with so many requests, the side that will suffer more is the one without the primary event store. When privileging the writing operations on one side, the other will have to wait more for Mongo replies, consuming more machine resources and making the application overall slower.

8.4.4 Setup 2: EU (Writes) and USA (Reads)

This experiment puts one of the regions doing reading operations and the other writing operations. The European application is tested with writing operations, and the American with reading operations. The experiment is on the second setup, so the system has the Mongo replica set with two Mongo instances (including the primary) in Europe and one in the USA.

Figure 8.6 and table 8.7 show how fast the reading operations are if the application is not being asked to do writing tasks, with all reading operations response-times being under 15 ms.

On the other hand, only flooding the application with writing operations will

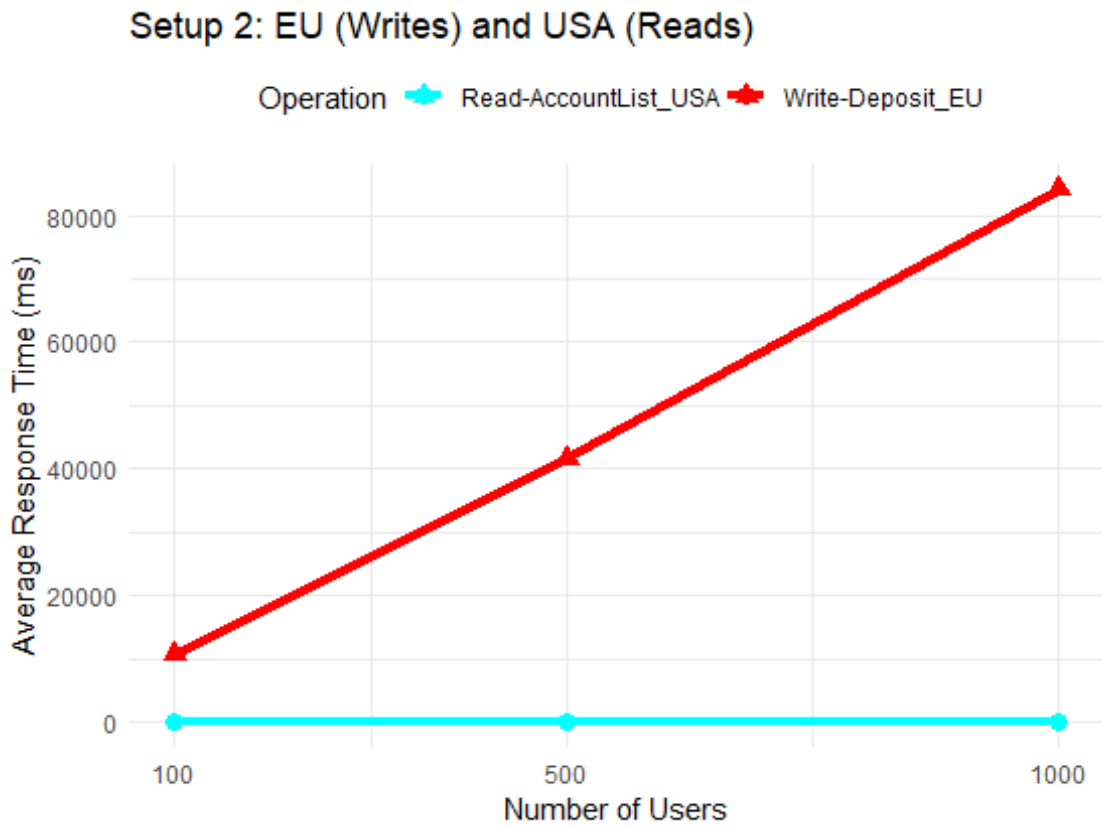


Figure 8.6: Setup 2: EU (Writes) and USA (Reads)

Setup 2: EU (Writes) and USA (Reads)		
Number_of_users	WriteDeposit_EU (ms)	ReadAccountList_USA (ms)
100	10688	4
500	41671	14
1000	84047	14

Table 8.7: Setup 2: EU (Writes) and USA (Reads)

make the application slower. When comparing the write times of this experiment with all other experiments presented before, these are the slowest values for writing operations until now for both regions. This data is another evidence of how heavy the write operations are and how slow the application can become.

8.4.5 Setup 2: EU (Reads) and USA (Writes)

This final experiment puts one of the regions doing reading operations and the other writing operations, like the last presented experiment. This time, the European application is tested with reading operations and the American with writing operations. This experiment is on the second setup, so the system has the Mongo replica set with two Mongo instances (including the primary) in Europe and one in the USA, where the American application writes events on the primary and loads events from the American replica.

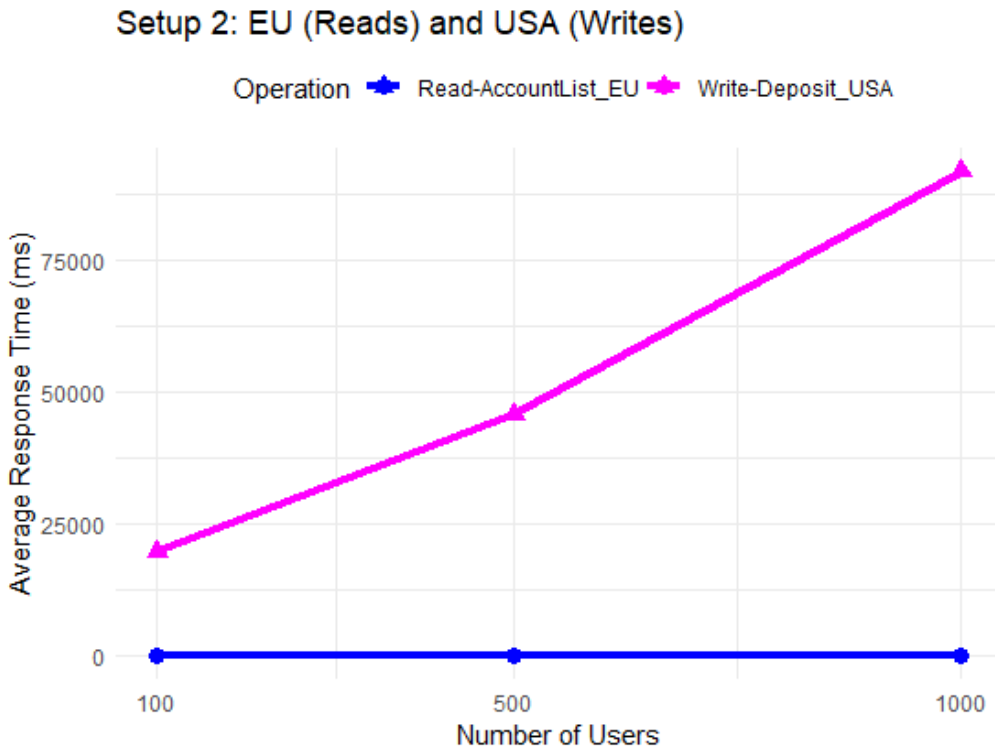


Figure 8.7: Setup 2: EU (Writes) and USA (Reads)

Setup 2: EU (Reads) and USA (Writes)		
Number_of_users	ReadAccountList_EU (ms)	WriteDeposit_USA (ms)
100	3	19819
500	4	45937
1000	7	91911

Table 8.8: Setup 2: EU (Reads) and USA (Writes)

Figure 8.7 and table 8.8 show similar behaviour to the previous "Setup 2: EU (Writes) and USA (Reads)" experiment (section 8.4.4). The readings are again really fast and around the same values as the American readings from the previous experiment, and the writings are slower than all experiments. The largest difference between these two last experiments is that for writing tasks for 100 users, the writing operation is close to all other American writing operations, even though it is still slower.

8.5 CPU Utilization

In an attempt to support the potential correlation between fluctuations in response times and application resource consumption, two new tests were conducted on "Setup 1" (refer to Section 7.2.1). These tests involved the analysis of CPU utilization on the system's EC2 machines. One run encompassed writing and reading operations, while the other focused exclusively on reading operations.

Figure 8.8 has the two linear graphics created through AWS statistical capabilities that show the average CPU utilization from both tests. The left graphic on the figure is the first test with writing and reading operations and the second with just reading operations. The blue line is the applications EC2 machine, the orange is Axon Server's EC2, and the green is MongoDB's.

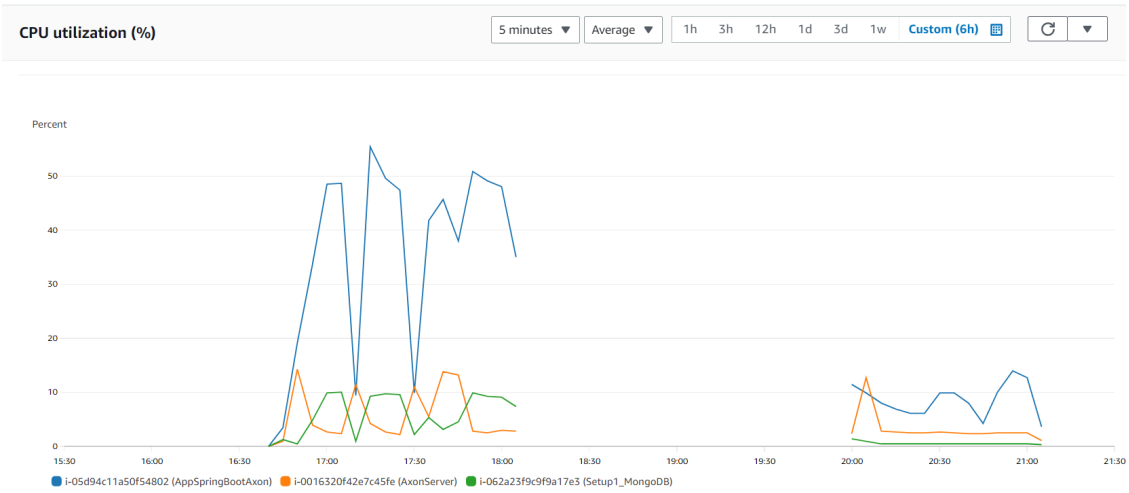


Figure 8.8: Average CPU utilization from "Setup 1" (section 7.2.1) EC2 machines with read and write operations at the left and only read operation on the right

By analysing both graphics in figure 8.8, it shows how demanding writing operations are on the machine, and due to the EC2 machines not being that powerful, how they might influence the overall performance of the application including the reading operations, which should take only a couple of milliseconds and end up taking a very long time.

8.6 Final Remarks

This chapter allowed us to demonstrate that the proposed implementation with replication works. The data presented in this chapter show that having CQRS with the system replicated is more beneficial than just making requests from one continent to the other when comparing just the read times by themselves (section 8.4.4) with the 70 ms trans-Atlantic ping [55]. Also, the data shows that the writing operations from the US are slower because of the trans-Atlantic connection to the primary MongoDB instance of the replica set.

Chapter 9

Planning

This chapter will present and explain the planned schedule for this dissertation, compare it to the executed one and the risks that may affect the project.

This dissertation period is divided into two semesters, the first (part-time) for research and establishing what to develop and the second (full-time) focused on development and implementation. The schedule presentation uses Gantt charts for tasks and their time frame visualization.

9.1 First Semester

The first semester was planned to be part-time, which means that the time was shared with other courses taken at the University.

Planned

The Gantt chart in figure 9.1 represents the planned schedule.

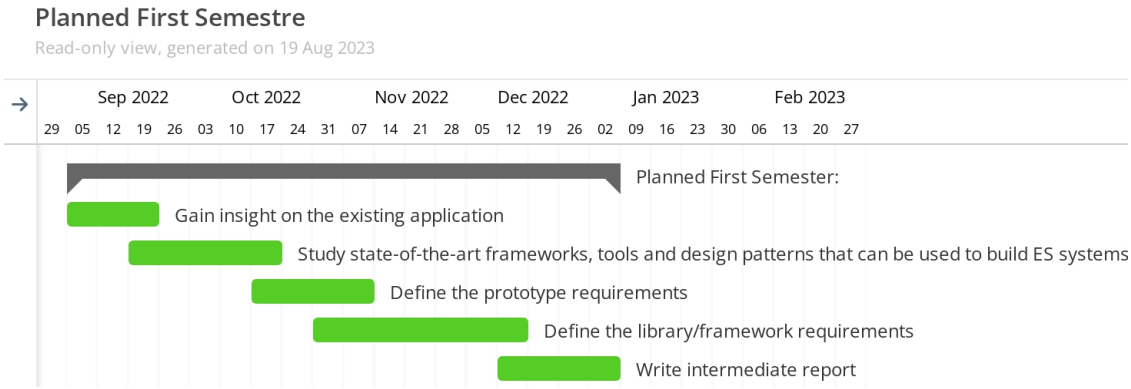


Figure 9.1: Planned First Semester Schedule

Executed

The Gantt chart in figure 9.2 represents the executed schedule.

Figure 9.2: Executed First Semester Schedule

Descriptions of the tasks of the first semester are presented in the following list:

- **Research on Event Sourcing, CQRS and Saga:** Research on the concepts related to Event Sourcing.
- **Experiment With Axon:** To better understand Axon, an Event Sourcing framework, an experimentation scenario with the axon framework was created.
- **Research on dynamic event allocation in logs:** Research performed on the idea of grouping related events in logs as operations take place. This task was carried out because of research made on the Experiment With Axon task and on research made on multiple event logs.
- **Research on real-world Event Sourcing implementations and Event Sourcing in distributed systems:** Research performed on existing implementations of Event Sourcing and this concept in distributed environments. This task was carried out because the research made on the dynamic event allocation in logs proved not to be a good approach for the project.
- **Requirement and architecture definitions:** This task was done with the objective of developing an event-sourced application in a distributed environment.
- **Write intermediate report:** This task had the objective of writing this report as a way to keep the progress of the dissertation's project.

9.2 Second Semester

Planned

The second semester is planned to be a full-time. The Gantt chart in figure 9.3 represents the planned schedule.

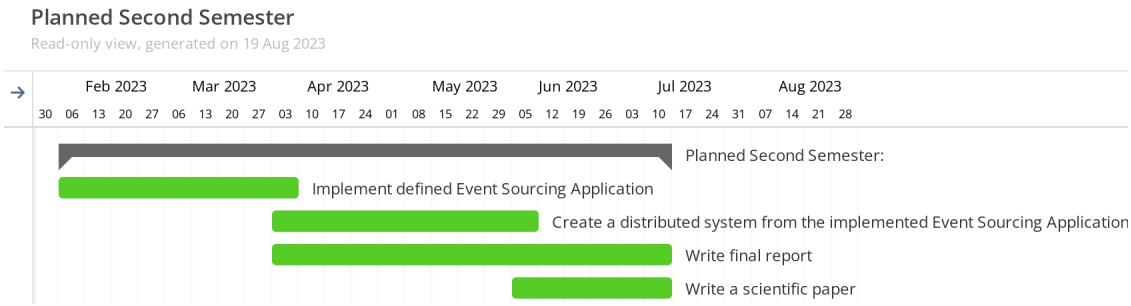


Figure 9.3: Planned Second Semester Schedule

Below is the list presenting the descriptions of the tasks scheduled for the second semester:

- **Implement defined Event Sourcing Application:** Implement the requirements defined as Must Haves in the MoSCoW analysis.
- **Create a distributed system from the implemented Event Sourcing Application:** Implement the requirements defined as Should Haves in the MoSCoW analysis.
- **Write a scientific paper extracting the lessons and knowledge gained from the process:** Write a scientific paper about the topics researched in this work.
- **Write final report:** Write the final curricular report.

Executed

The Gantt chart in figure 9.4 presents the executed schedule for the second semester.

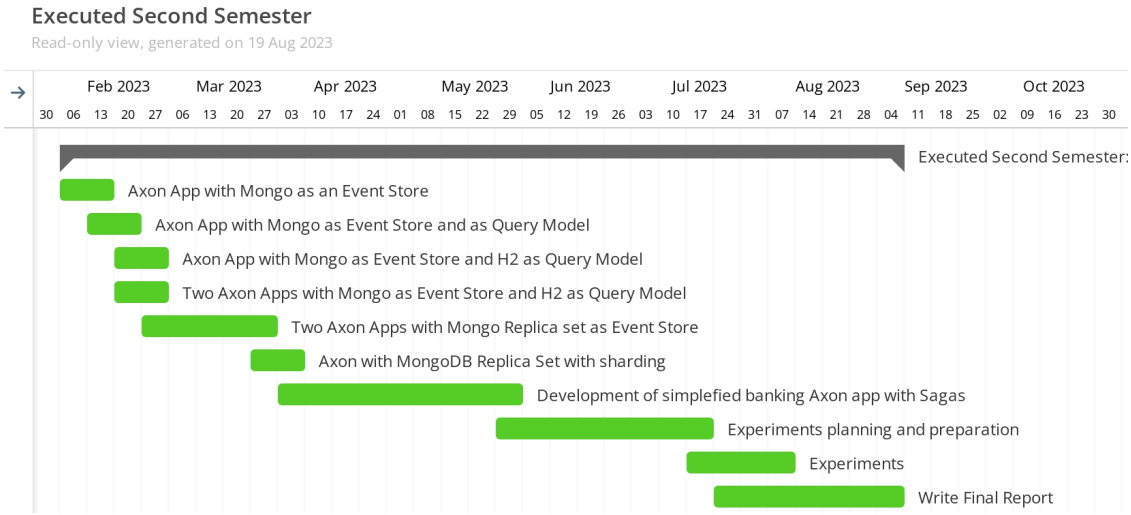


Figure 9.4: Executed Second Semester Schedule

Below is the list presenting and describing the tasks executed during the second semester:

- **Axon App with Mongo as an Event Store:** This task required taking the experiment done with Axon in the first semester and configuring it to use MongoDB as an event store running on docker.
- **Axon App with Mongo as Event Store and as Query Model:** Took the application using MongoDB as an event store and attempted using the same and other Mongo instances for the Read Model. This approach was left in favour of an in-Memory H2 database.
- **Axon App with Mongo as Event Store and H2 as Query Model:** Took the application using MongoDB as an event store and configured H2 as an in-memory Database for the query model.
- **Two Axon Apps with Mongo as Event Store and H2 as Query Model:** Configured two different applications running independently with each other with their own H2 Query Database and Axon Server with a single Event Store running on docker.
- **Two Axon Apps with Mongo Replica set as Event Store:** Took the setup with two different applications running independently with each other with their own H2 Query Database and Axon Server with a Replica Set of three instances of MongoDB for the Event Store running on docker.
- **Axon with MongoDB Replica Set with sharding:** Took the setup with the replica set and attempted to apply sharding on the MongoDB cluster. This approach ended up not being completed due to time constraints.
- **Development of simplified banking Axon app with Sagas:** This task required developing the Apps business logic, events, commands and Sagas.
- **Experiments planning and preparation:** Researched how to upload and test the replicated setup running locally on docker to the internet with Amazon Web Services. This task took defining the setups and layouts to test, learning and choosing the best options of the available AWS services and tools, uploading the setup to the internet, learning and testing the Locust testing framework and preparing the Axon App for testing.
- **Experiments:** Running the different tests on AWS.
- **Write Final Report:** Complete this document with new chapters and information from the realised work of the second semester with improvements to the old chapters.

9.3 Risk Management

In this section, the potential problems and uncertainties that may affect the dissertation's success and a final assessment are going to be presented.

The identified risks are the following:

- **Lack of experience with new tools**

  Description: The master's student has no previous experience with some of the tools he has to work with to develop the proposed system. This inexperience can lead to some delays.

  Mitigation Plan: Search for tutorials, documentation and examples to learn about these technologies and how to work with them.

- **Requirements change**

  Description: As the project progresses, several meetings will occur between the master's student, his adviser and other stakeholders. These may lead to a change of plans and objectives for the project. These changes can lead to delays.

  Mitigation Plan: When a decision may seem uncertain, the master's student should prepare for those changes, such as thinking and researching the alternatives that look more plausible.

- **Not implementing every requirement in the project time-frame**

  Description: The number of requirements and other architectural drivers identified with the purpose of being implemented may be too large for the project's time length.

  Mitigation plan: The master's student can ask to delay the deadline once from July to September.

Final Risk Mitigation Assessment

During the dissertation time frame, the "Lack of experience" risk was faced and, to circumvent the problem, a lot of time was spent learning new tools. Even though the risk of not "Not implementing every requirement in the project time frame" did not come through, the mitigation plan was used to extend the deadline to September to perform experiments and analyse data for this dissertation.

Chapter 10

Conclusion

This chapter has the final analysis of the work done for this dissertation with an identification of the major struggles faced, what can be done and further explored in the future and a final personal reflection on the work performed on this project for the last year.

10.1 Difficulties

This section outlines some of the challenges faced while developing and implementing my dissertation project that posed substantial obstacles that slowed the project's progress.

The major challenges faced during this project were configurations. Axon's technologies, although relatively known, are not popular enough to find adequate problem-solving information beyond the official documentation and forums. This situation led to several struggles and debugging that slowed down the progress of the development.

10.1.1 Version Incompatibilities

The first relevant hurdle involved version incompatibilities between Maven, Java, Axon framework components and MongoDB. The interplay between these technologies, including serializers, plus the not-so-abundant available information, often resulted in compatibility issues, causing delays at the beginning of development in trying to set up MongoDB as the event store.

10.1.2 MongoDB Configuration

Configuring a MongoDB replica set on docker proved to be an intricate endeavour. Overcoming the difficulties of setting up the replica set and ensuring the connection with the Spring Boot Axon application proved time-consuming. The

container set-up and consequent configuration had intricacies regarding their docker-compose file led to a few problems regarding the connectivity between MongoDB instances and the Axon application. Also, the change from a single MongoDB instance to a replica set required a slight change in configuring the connection between the App and the replica set, which also took time to be understood and solved.

Lastly, the shading configuration within the MongoDB replica set proved more demanding than expected. The focus was to have a sharded replica set setup to compare its performance to the non-sharded setup. This effort ended up being sidelined in favour of focusing the work on the eventual experiments.

10.1.3 Saga implementation

Implementing a Saga for the money transfer was filled with setbacks. To properly implement a Saga, it is required to implement the sequence of actions to perform the transfer and compensatory actions in case of an error or failure. To implement the compensatory actions, the Saga needs to know at which point the transfer failed so it knows how and when to act. In the case of the application developed for this dissertation, if the transfer has not finished in a certain amount of time, compensation actions are initiated to add and remove money from where it was withdrawn and deposited. The first setback was that the Saga was not persisting its data, so the deadline manager (the time-trigger) wouldn't activate without any errors or warning messages that could help route the problem. The problem was the serializer. Due to MongoDB being the event store, the application had to use Jackson as the serializer and define that in the app's properties file. But changing the properties file to have other serializers would brake the application. The solution found was to manually configure the XStream serializer just for the Saga in a configuration java file.

After having the Saga persisting data on Mongo and the deadline manager triggering on time, the function wouldn't run, leading to deep debugging sessions of the code to find that problem was the function required to have a specific parameter.

Due to the lack of information about these problems, the saga implementation took a significant amount of time, requiring persistence and time to find the origin of the issues and then solutions to circumvent the problems.

10.2 Future Work

The work that could be done in the future is described in this section. Even though this Master's Dissertation has finished, ES is a concept that can be explored further with new implementations and other technologies.

10.2.1 Performance Analisys of Sagas

During the experimental phase of this work, the performance analysis of money transfer operations that use Sagas was not conducted. In the future, delving into the performance aspects of Sagas can help gain insights into their efficiency and responsiveness under an application like the one developed for this dissertation. The measurements should be time elapsed between the user request for a money transfer and the completion of the transfer itself.

10.2.2 Implementation and testing with a Sharded MongoDB setup

During the course of this study, sharding was not implemented within the MongoDB setup for the application. Sharding presents an interesting avenue for future exploration due to its potential to enhance the scalability and performance of the application. In future research, incorporating sharding into the MongoDB setup and conducting testing would provide valuable insights into how the system performs.

10.2.3 Implementation and exploration with other technologies

The current implementation utilized MongoDB, Axon, and H2 database technologies. Future endeavours could explore the integration of alternative technologies, such as Kafka or Cassandra or any other, to broaden the scope of this study and gain insights into different technological setups. Investigating the implementation and performance of the application with alternative technological contexts could provide valuable comparative insights and potentially uncover superior solutions.

10.3 Final Thoughts

To conclude, I think this dissertation has been completed successfully. The predefined functionalities were implemented, and the system was successfully tested and analysed with different experiments across different layouts and setups. However, more work can be done in a way that iterates on what has been presented here or in a way that explores Event Sourcing differently, as stated in section 10.2.

The time invested in studying and implementing the concepts of ES, CQRS and Sagas on a system allowed me to gain knowledge on these concepts that might prove useful in the future. It also allowed me to gain knowledge on the different technologies and tools that I had not used before, such as Axon or MongoDB, and improve my understanding and expertise on others with some previous experience, such as Docker and AWS.

In conclusion, this dissertation was a valuable experience that allowed me to be a part of a project with multiple institutions and to gain experience working on a

long-year project that demanded me to be critical and capable of problem-solving at an individual level in both theoretical and practical levels.

References

[1] H2 database engine. URL https://www.h2database.com/html/main.html. Available at https://www.h2database.com/html/main.html, Accessed: 2023-07-27.

[2] H2 database - introduction | tutorialspoint. URL https://www.tutorialspoint.com/h2_database/h2_database_introduction.htm. Available at https://www.tutorialspoint.com/h2_database/h2_database_introduction.htm, Accessed: 2023-07-27.

[3] Locust - a modern load testing framework. URL https://locust.io/. Available at https://locust.io/, Accessed: 2023-08-13.

[4] Apache kafka tutorial. URL https://www.tutorialspoint.com/apache_kafka/index.htm. Available at https://www.tutorialspoint.com/apache_kafka/index.htm, Accessed: 2022-12-14.

[5] Acid properties in dbms - geeksforgeeks, 2022. URL https://www.geeksforgeeks.org/acid-properties-in-dbms/. Available at https://www.geeksforgeeks.org/acid-properties-in-dbms/, Accessed: 2022-10-28.

[6] Daniel Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, 45(2):37–42, 2012.

[7] AxonIQ. Introduction - axon reference guide. URL https://docs.axoniq.io/reference-guide/. Available at https://docs.axoniq.io/reference-guide/, Accessed: 2023-01-10.

[8] AxonIQ. *Live Coding - Axon Framework and Pivotal Cloud Foundry* [video], 2017. URL https://youtu.be/lBKZOTe9QM4. Available at https://youtu.be/lBKZOTe9QM4, Accessed: 2023-01-10.

[9] AxonIQ. *Axon Quick Start Guide* [video playlist], 2019. URL https://youtube.com/playlist?list=PL4O1nDpoa5KQkkApGXjKi3rzUW3II5pjm. Available at https://youtube.com/playlist?list=PL4O1nDpoa5KQkkApGXjKi3rzUW3II5pjm, Accessed: 2023-01-10.

[10] AxonIQ. https://start.axoniq.io/, 2021. URL https://start.axoniq.io/. Available at https://start.axoniq.io/, Accessed: 2023-01-10.

[11] AxonIQ. Axoniq products - more than axon framework, 2022. URL https://www.axoniq.io/axoniq-products. Available at https://www.axoniq.io/axoniq-products, Accessed: 2022-12-10.

[12] AxonIQ. Axon framework, 2022. URL https://developer.axoniq.io/axon-framework/overview. Available at https://developer.axoniq.io/axon-framework/overview, Accessed: 2022-12-10.

[13] AxonIQ. Axoniq - event-driven systems with axon framework and server, 2022. URL AxonIQCompany. Available at AxonIQCompany, Accessed: 2023-07-25.

[14] AxonIQ. Axon server, 2022. URL https://developer.axoniq.io/axon-server/overview. Available at https://developer.axoniq.io/axon-server/overview, Accessed: 2022-12-10.

[15] AxonIQ. Download, 2022. URL https://developer.axoniq.io/download. Available at https://developer.axoniq.io/download, Accessed: 2023-01-11.

[16] baeldung. Saga pattern in microservices | baeldung on computer science, 2022. URL https://www.baeldung.com/cs/saga-pattern-microservices. Available at https://www.baeldung.com/cs/saga-pattern-microservices, Accessed: 2022-10-27.

[17] Derek Comartin. Projections in event sourcing: Build any model you want! - codeopinion, 2021. URL https://codeopinion.com/projections-in-event-sourcing-build-any-model-you-want/. Available at https://codeopinion.com/projections-in-event-sourcing-build-any-model-you-want/ Accessed: 2022-10-11.

[18] Derek Comartin. Snapshots in event sourcing for rehydrating aggregates - codeopinion, 2021. URL https://codeopinion.com/snapshots-in-event-sourcing-for-rehydrating-aggregates/. Available at https://codeopinion.com/snapshots-in-event-sourcing-for-rehydrating-aggregates/ Accessed: 2022-10-12.

[19] Derek Comartin. Cqrs and event sourcing code walk-through - codeopinion, 2022. URL https://codeopinion.com/cqrs-event-sourcing-code-walk-through/. Available at https://codeopinion.com/cqrs-event-sourcing-code-walk-through/, Accessed: 2022-10-11.

[20] Ibuildings Dutch PHP Conference. ***keynote: event sourcing - greg young - dpc2016*** [video], 2017. URL https://youtu.be/8JKjvY4etTY. Available at https://youtu.be/8JKjvY4etTY, Accessed: 2022-10-07.

[21] GOTO Conferences. *Event Sourcing • Greg Young • GOTO 2014* [video], 2014. URL https://youtu.be/8JKjvY4etTY. Available at https://youtu.be/8JKjvY4etTY, Accessed: 2022-10-07.

[22] Confluent. *Event Sourcing 101: Incorporating Event Storage into Your System* [video], 2022. URL https://www.youtube.com/watch?v=ds0SgB4jG_s&ab_channel=Confluent. Available at https://www.youtube.com/watch?v=ds0SgB4jG_s&ab_channel=Confluent, Accessed: 2022-12-14.

[23] Inc. Confluent. What is kafka? | confluent, 2022. URL https://www.confluent.io/what-is-apache-kafka/. Available at https://www.confluent.io/what-is-apache-kafka/, Accessed: 2022-12-14.

[24] Inc. Confluent. The source of truth: Why the new york times stores every piece of content ever published in kafka - confluent, 2022. URL https://www.confluent.io/kafka-summit-nyc17/source-truth-new-york-times-stores-every-piece-content-ever-published-kafka/?utm_source=youtube&utm_medium=video&u. Available at https://www.confluent.io/kafka-summit-nyc17/source-truth-new-york-times-stores-every-piece-content-ever-published-kafka/?utm_source=youtube&utm_medium=video&u, Accessed: 2022-12-14.

[25] Andrzej Debski, Bartlomiej Szczepanik, Maciej Malawski, Stefan Spahr, and Dirk Muthig. In search for a scalable & reactive architecture of a cloud application: Cqrs and event sourcing case study. *IEEE Software*, 99, 2017.

[26] Oskar Dudycz. Snapshots in event sourcing, 2021. URL https://www.eventstore.com/blog/snapshots-in-event-sourcing. Available at https://www.eventstore.com/blog/snapshots-in-event-sourcing, Accessed: 2022-10-12.

[27] IBM Cloud Education. What is mongodb? | ibm, 2020. URL https://www.ibm.com/cloud/learn/mongodb. Available at https://www.ibm.com/cloud/learn/mongodb, Accessed: 2022-12-13.

[28] The Apache Software Foundation. Apache cassandra | apache cassandra documentation, 2022. URL https://cassandra.apache.org/_/cassandra-basics.html. Available at https://cassandra.apache.org/_/cassandra-basics.html, Accessed: 2022-12-14.

[29] The Apache Software Foundation. Overview | apache cassandra documentation, 2022. URL https://cassandra.apache.org/doc/latest/cassandra/architecture/overview.html. Available at https://cassandra.apache.org/doc/latest/cassandra/architecture/overview.html, Accessed: 2022-12-14.

[30] Martin Fowler. Event sourcing, 2005. URL https://martinfowler.com/eaaDev/EventSourcing.html. Available at https://martinfowler.com/eaaDev/EventSourcing.html, Accessed: 2022-10-07.

[31] Martin Fowler. Ddd_aggregate, 2013. URL https://martinfowler.com/bliki/DDD_Aggregate.html. Available at https://martinfowler.com/bliki/DDD_Aggregate.html, Accessed: 2023-01-11.

[32] Seth Gilbert and Nancy Lynch. Perspectives on the cap theorem. *Computer*, 45(2):30–36, 2012.

[33] gRPC Authors. grpc, 2022. URL https://grpc.io/. Available at https://grpc.io/, Accessed: 2022-12-12.

[34] Mattias Holmqvist. Event sourcing explained - why you should care - part 1 | blog | serialized, 2018. URL https://serialized.io/blog/event-sourcing-explained-part-1-why-you-should-care. Available at https://serialized.io/blog/event-sourcing-explained-part-1-why-you-should-care, Accessed: 2023-01-13.

[35] InfoQ. *Scaling Event Sourcing for Netflix Downloads* [video], 2017. URL https://youtu.be/rsSld8NycCU. Available at https://youtu.be/rsSld8NycCU, Accessed: 2023-01-12.

[36] Keboola. What is eventual consistency and why should you care about it?, 2021. URL https://www.keboola.com/blog/eventual-consistency. Available at https://www.keboola.com/blog/eventual-consistency, Accessed: 2022-10-20.

[37] Thanh Le. [microservices architecture] what is saga pattern and how important is it? | by thanh le | the startup | medium, 2020. URL https://medium.com/swlh/microservices-architecture-what-is-saga-pattern-and-how-important-is-it-55f56cfe
Available at https://medium.com/swlh/microservices-architecture-what-is-saga-patter
Accessed: 2022-10-27.

[38] Event Store Limited. Beginner's guide to event sourcing | event store, . URL https://www.eventstore.com/event-sourcing. Available at https://www.eventstore.com/event-sourcing, Accessed: 2022-10-27.

[39] Event Store Limited. Introduction | eventstoredb documentation, . URL https://developers.eventstore.com/server/v21.10/#protocols-clients-and-sdks. Available at https://developers.eventstore.com/server/v21.10/#protocols-clients-and-sdks, Accessed: 2022-12-12.

[40] Event Store Limited. Clustering | eventstoredb documentation, . URL https://developers.eventstore.com/server/v21.10/cluster.html#cluster-nodes. Available at https://developers.eventstore.com/server/v21.10/cluster.html#cluster-nodes, Accessed: 2022-12-12.

[41] Event Store Limited. Clustering | eventstoredb documentation, . URL https://developers.eventstore.com/server/v21.10/cluster.html#cluster-node-roles. Available at https://developers.eventstore.com/server/v21.10/cluster.html#cluster-node-roles, Accessed: 2022-12-12.

[42] Event Store Limited. Introduction | eventstoredb documentation, . URL https://developers.eventstore.com/server/v21.10/#getting-started. Available at https://developers.eventstore.com/server/v21.10/#getting-started, Accessed: 2022-12-12.

[43] Microsoft. Cqrs pattern - azure architecture center | microsoft learn, 2022. URL https://learn.microsoft.com/en-us/azure/architecture/

patterns/cqrs. Available at https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs, Accessed: 2022-10-07.

[44] Microsoft. Event sourcing pattern - azure architecture center, 2022. URL https://learn.microsoft.com/en-us/azure/architecture/patterns/event-sourcing. Available at https://learn.microsoft.com/en-us/azure/architecture/patterns/event-sourcing, Accessed: 2022-10-11.

[45] Microsoft. Saga pattern - azure design patterns | microsoft learn, 2022. URL https://learn.microsoft.com/en-us/azure/architecture/reference-architectures/saga/saga. Available at https://learn.microsoft.com/en-us/azure/architecture/reference-architectures/saga/saga, Accessed: 2022-10-26.

[46] Inc. MongoDB. Introduction to mongodb — mongodb manual, 2022. URL https://www.mongodb.com/docs/manual/introduction/. Available at https://www.mongodb.com/docs/manual/introduction/, Accessed: 2022-12-13.

[47] Inc. MongoDB. Replica set elections — mongodb manual, 2022. URL https://www.mongodb.com/docs/manual/core/replica-set-elections/#std-label-replica-set-elections. Available at https://www.mongodb.com/docs/manual/core/replica-set-elections/#std-label-replica-set-elections, Accessed: 2022-12-13.

[48] Inc. MongoDB. Replication — mongodb manual, 2022. URL https://www.mongodb.com/docs/manual/replication/. Available at https://www.mongodb.com/docs/manual/replication/, Accessed: 2022-12-13.

[49] Inc. MongoDB. Sharding — mongodb manual, 2022. URL https://www.mongodb.com/docs/manual/sharding/. Available at https://www.mongodb.com/docs/manual/sharding/, Accessed: 2022-12-13.

[50] Inc. MongoDB. Read preference — mongodb manual, 2023. URL https://www.mongodb.com/docs/manual/core/read-preference/. Available at https://www.mongodb.com/docs/manual/core/read-preference/, Accessed: 2023-08-13.

[51] Neha Narkhede. Event sourcing, cqrs, stream processing and apache kafka: What's the connection? | confluent, 2016. URL https://www.confluent.io/blog/event-sourcing-cqrs-stream-processing-apache-kafka-whats-connection/. Available at https://www.confluent.io/blog/event-sourcing-cqrs-stream-processing-apache-kafka-whats-connection/, Accessed: 2022-12-14.

[52] Jedrzej Rybicki and Juelich Supercomputing Center JSC. Application of event sourcing in research data management. *ALLDATA*, pages 22–26, 2018.

[53] Salomé Simon. Brewer's cap theorem. *CS341 Distributed Information Systems, University of Basel (HS2012)*, 2000.

[54] Anton Stöckl. Event sourcing: Why kafka is not suitable as an event store | itnext, 2018. URL https://itnext.io/event-sourcing-why-kafka-is-not-suitable-as-an-event-store-796e5d9ab63c. Available at https://itnext.io/event-sourcing-why-kafka-is-not-suitable-as-an-event Accessed: 2022-12-14.

[55] Verizon. Monthly ip latency data | verizon enterprise solutions, 2023. URL https://www.verizon.com/business/terms/latency/. Available at https://www.verizon.com/business/terms/latency/, Accessed: 2023-08-24.

[56] Ivy Wigmore. What is eventual consistency? - definition from whatis.com, 2014. URL https://www.techtarget.com/whatis/definition/eventual-consistency. Available at https://www.techtarget.com/whatis/definition/eventual-consistency, Accessed: 2022-10-20.

[57] Alexey Zimarev. Event store replicator, 2021. URL https://www.eventstore.com/blog/event-store-replicator. Available at https://www.eventstore.com/blog/event-store-replicator, Accessed: 2022-12-12.

[58] Alexey Zimarev. Reporting models and event sourcing - alexey's place, 2021. URL https://zimarev.com/blog/event-sourcing/changes-in-event-sourced-systems/. Available at https://zimarev.com/blog/event-sourcing/changes-in-event-sourced-systems/, Accessed: 2022-11-17.