NTNU

Kunnskap for en bedre verden

Bachelor thesis

IE303612 - Bachelor Thesis

Event Sourcing

Candiate numbers:

10014

10058

Totalt antall sider inkludert forsiden: 87

Innlevert Ålesund, 31.05.2018

Obligatorisk egenerklæring/gruppeerklæring

Den enkelte student er selv ansvarlig for å sette seg inn i hva som er lovlige hjelpemidler, retningslinjer for bruk av disse og regler om kildebruk. Erklæringen skal bevisstgjøre studentene på deres ansvar og hvilke konsekvenser fusk kan medføre. Manglende erklæring fritar ikke studentene fra sitt ansvar.

Du/dere fyller ut erklæringen ved å klikke i ruten til høyre for den enkelte del 1-6:		
1.	**Jeg/vi erklærer herved at min/vår besvarelse er mitt/vårt eget arbeid, og at jeg/vi ikke har brukt andre kilder eller har mottatt annen hjelp enn det som er nevnt i besvarelsen.**	☒
2.	**Jeg/vi erklærer videre at denne besvarelsen:** • ikke har vært brukt til annen eksamen ved annen avdeling/universitet/høgskole innenlands eller utenlands. • ikke refererer til andres arbeid uten at det er oppgitt. • ikke refererer til eget tidligere arbeid uten at det er oppgitt. • har alle referansene oppgitt i litteraturlisten. • ikke er en kopi, duplikat eller avskrift av andres arbeid eller besvarelse.	☒
3.	**Jeg/vi er kjent med at brudd på ovennevnte er å betrakte som fusk og kan medføre annullering av eksamen og utestengelse fra universiteter og høgskoler i Norge, jf. Universitets- og høgskoleloven §§4-7.**	☒
4.	**Jeg/vi er kjent med at alle innleverte oppgaver kan bli plagiatkontrollert.**	☒
5.	**Jeg/vi er kjent med at NTNU vil behandle alle saker hvor det foreligger mistanke om fusk.**	☒
6.	**Jeg/vi har satt oss inn i regler og retningslinjer i bruk av kilder og referanser på biblioteket sine nettsider**	☒

Publiseringsavtale

Studiepoeng: 20

Veileder: Hao Wang

Fullmakt til elektronisk publisering av oppgaven

Forfatter(ne) har opphavsrett til oppgaven. Det betyr blant annet enerett til å gjøre verket tilgjengelig for allmennheten (Åndsverkloven §2).
Alle oppgaver som fyller kriteriene vil bli registrert og publisert i Brage med forfatter(ne)s godkjennelse.
Oppgaver som er unntatt offentlighet eller båndlagt vil ikke bli publisert.

Jeg/vi gir herved NTNU i Ålesund en vederlagsfri rett til å gjøre oppgaven tilgjengelig for elektronisk publisering: [✓ ja] nei

Er oppgaven båndlagt (konfidensiell)? ja [✓ nei]
(Båndleggingsavtale må fylles ut)
- Hvis ja:
Kan oppgaven publiseres når båndleggingsperioden er over? ja nei

Er oppgaven unntatt offentlighet? ja [✓ nei]
(inneholder taushetsbelagt informasjon. Jfr. Offl. §13/Fvl. §13)

Dato: 31.05.2018

PREFACE

When we read the problem description for this thesis, we heard of event sourcing for the first time. We saw event sourcing as an interesting concept as an alternative to CRUD, which we had used for all previous programming projects where data storage was involved.

Our goal for the thesis was to find out how you can use event sourcing in a modern system. We wanted to understand what advantages event sourcing has over CRUD. While also looking into what new problems and complexities arise when developing an event sourced system.

We learned a lot while working on this thesis. We ended up with a distributed system, using microservice architecture and event-driven architecture for our solution. These were very interesting concepts that were somewhat new to us. We had heard about them before, but never developed a system using them. We think we ended up with a very interesting solution and many new experiences to take with us for future projects.

We would like to thank DRIW for a very interesting assignment that we have learned a lot from. We also want to thank our supervisor Hao Wang for valuable insight, advice, and feedback we got along the way.

ABSTRACT

Most systems today are typically using CRUD operations to handle creating, reading, updating, and deleting data. The problem with CRUD is that it every time you update or remove data you lose important business information.

This is a problem that can be solved with event sourcing. Event sourcing provides a full audit log of everything that has happened in a system. This provides traceability for the all changes that has happened. It also fits naturally into an event-driven architecture, which helps you keep a system loosely coupled. CQRS is often combined with event sourcing, this gives flexible read models when it comes to structure and scaling.

The thesis is inspired and supported by DRIW who provided the problem description. The problem was to develop an example architecture for an order system based on event sourcing, implementing a prototype of this, and figuring out what benefits event sourcing brings in this kind of system.

The result of our thesis is a prototype system, using event sourcing for data storage. This system handles simple business logic involving orders, picking, and invoicing. The prototype has been developed as a distributed system using microservice and event-driven architecture. Apache Kafka is used for publish/subscribe messaging and to provide a persistent message queue in the prototype implementation. All events saved in the event store is made available on Kafka topics that other application can subscribe to. The events received from Kafka is used to change the state of the receiving system or to update read models. This prototype also includes simulator applications that simulate user interaction with the system.

Event sourcing is a very interesting concept and combining it with CQRS it allows you to disconnect the write model from the read model. This can help avoid the difficulties of mapping normalized database rows to objects, by having read models in the format you require. If you have complex queries that are difficult to perform on your read model you can always create a new read model optimized for this query.

Event sourcing is more complex than the CRUD approach to data storage. Therefore, CRUD is most likely the better option if you don’t require an audit log and are not developing an event-driven system. Event sourcing combined with CQRS gives an eventually consistent read model. This is not strictly bad but is something the developer need to be aware of.

We ended up with a solid prototype of a distributed ordering system by using event sourcing and CQRS along with microservice and event-driven architecture.

The prototype received very positive feedback from DRIW. We found two benefits of event sourcing that interested them. That it provides traceability and it fits naturally into an event-driven architecture.

TERMINOLOGY

Abbreviations

API – Application Programming Interface
CAP – Consistency, Availability, Partitioning tolerance
CLOB – Character Large Object
CQRS – Command, Query, Responsibility, Segregation
CQS – Command, Query, Separation
CRUD – Create, Read, Update, Delete
CSV – Comma-Separated Values
DDD – Domain Driven Design
DTO – Data Transfer Object
FK – Foreign Key
HDD – Hard Disk Drive
HTTP – Hypertext Transfer Protocol
JDBC – Java Database Connectivity
JPA – Java Persistence API
JSON – JavaScript Object Notation
JVM – Java Virtual Machine
LRU – Least Recently Used
MQTT – Message Queuing Telemetry Transport
NoSQL – Non-relational SQL
ODBC – Open Database Connectivity
OOP – Object Oriented Programming
PK – Primary Key
POJO – Plain Old Java Object
REST – Representational state transfer
RFC – Request for comment
RPC – Remote Procedure Calls
SQL – Structured Query Language
SSD – Solid State Drive
UUID – Unique Universal Identifier
VM – Virtual Machine

TABLE OF CONTENT

FIGURE LIST

LISTINGS

1 INTRODUCTION

This bachelor thesis was written at NTNU in Ålesund in collaboration with DRIW. DRIW is a local software development company that develops systems for wholesale distributors. Their products include order system, warehouse management, transport management, etc. They have started development of a new product named TRACE, and in that regard, they are considering if event sourcing can be beneficial in a new order module they are developing.

The goal of the thesis project is to develop an example architecture for an order system, based on event sourcing. The solution should include a prototype implementation of a system based on this architecture, that can handle simple business logic. The business logic should include ordering of products, picking of products for orders and invoicing of orders. The implementation should also include a simulation of user interaction with the system.

After our meeting with DRIW about this assignment we interpreted it as that they also wanted to know about any advantages or disadvantages we could find when applying event sourcing to a system like this.

The thesis is structured in such a way that it will give the reader insight into how event sourcing works in general and an explanation of various concepts and theory we have used or discussed. Followed by a description of the different tools and technologies that you would need to understand in order recreate the solution.

After that, we will look at which approaches we used to build the event sourced system, which results we got from building this system, what challenges we encountered, a retrospective on what we could've done differently, and then we will end the thesis with a conclusion.

We presume that the reader has some basic knowledge of software development and object-oriented programming. The thesis has been written according to those presumptions.

1.1 Limitations

As already mentioned the business logic of the system should be simple. This could result in some unforeseen problems with implementing more complex business logic.

2 THEORY

In this section, we will go through some of the theory that will be necessary to have some knowledge of to reproduce our results.

2.1 Software Architecture

The software architecture of a system tells you something about how a system should be organized and how the overall structure should be designed. It describes a high-level structure of a system and its components. More specifically it shows the relationships between components and how they interact with each other. (1)

Which software architecture to use is decided in the early stages of the development process, this will help the developers get a good overview of the system. Good software architecture will also make it easier to communicate with the stakeholders, it lowers the overall cost, makes the software easier to maintain and debug, and makes your system more scalable.

We will now go through some of the architectures we have used or discussed in the thesis. We are going to look at some of the advantages and disadvantages of each one and see how they compare to each other.

2.1.1 Monolithic architecture

A monolith is an application built as a single unit running as a single executable. (2)
It can be difficult to implement and maintain a monolithic architecture as the code base grows. However, it does have some performance advantages. It can communicate with the different parts of the application with method calls, shared memory or message passing. This gives much lower latency than HTTP and RPC calls.

Some problems of monolithic architecture are: (3)

1. Large monoliths can be difficult to maintain and evolve.
2. Can suffer from dependency hell where adding and updating libraries can cause compilation issues or make the program operate unexpectedly
3. Any change in one module of a monolith requires restarting the whole application. This can result in considerable downtime. This can also result in slow down development and testing.
4. Deployment may be suboptimal due to conflicting requirements of different modules. This can result in a one size fits all configuration which can be expensive for some modules.
5. Monoliths can limit scalability. One way to handle additionally incoming requests is to create a new instance of the monolith to split the load. But traffic may only cause stress to some modules, in which case it is inefficient to create new instances of modules that don’t require it.
6. Monoliths create a technology lock-in. Modules are bound to use the same language and framework

2.1.2 Microservice Architecture

Microservice architecture is somewhat contradictory to the monolith architecture. A monolith keeps all functionality in a single large executable, while microservices split functionality up into many smaller executables. Microservices solve some of the problems mentioned about monolithic architecture, in the previous section: (3)

1. Microservices implement a limited amount of functionality. Making their code base small.
2. Microservices can gradually transition to a new version. The new version could run alongside the old version and let other services can gradually transition to using the new service

3. Changing a module of a microservice architecture does not require a complete restart of the whole system. Only the affected module needs to be restarted. The small services also help speed up restart which causes development and testing to speed up.
4. Microservices naturally work well with containerization. The developers have more freedom when it comes to configuring the environment the service will run in.
5. Scaling a microservice architecture does not require duplicating all modules of the system. Developers can deploy and remove instances of specific services depending on load.
6. The microservice module is only "locked in" by the technology used for communication between them. Otherwise, developers can freely develop each service in different languages and frameworks.

A microservice is a small application that can be run independently, scaled independently and tested independently. It should have a single responsibility, only do one thing, and be easily understood. (4)

The microservices architecture is based on having your application run as small independent services that can communicate with each other. This architecture will make your applications easier to scale and faster to develop. This is an alternative approach to the monolithic architecture which runs all its services in a single container or executable.

Another benefit of this architecture is that it can provide better reliability for your system. Let's say one of your services fail or is temporarily unavailable. The rest of your system should be able to continue functioning, albeit with somewhat reduced functionality until the failing service is available again.

As mentioned microservices can help scale your system by simplifying creation and removal of application instances. It can also help gradually transition to new versions, but this will come at the cost of more complex deployments. Some of this can be alleviated by using continuous tools.

Designing the boundaries of microservices can be difficult, but it is important to get them right to avoid having tightly coupled microservices. If the bounds of the microservices are not designed carefully, you could also end up with a distributed monolith. A distributed monolith gives problems of both microservices and monoliths.

2.1.3 Event-Driven Architecture

When developing microservices and distributed systems you need coordination and communication between the applications and services. This is where event-driven architecture can be useful.

An event-driven architecture is a software architecture where applications publish, detect and respond to events. An event represents a change in the state of the system. The architecture includes producers, processors, responders and communication links. (5)

This approach can be used to produce events for state changes. When the state of something in your system is updated, a new event is created. That event will then be published to a topic which works as a logical separation of messages, subscribers of that topic will be notified of this change. A service may do some work when it receives the notification of a state change. Then it may publish another event to its subscribers that a state has changed. This will cause the system to be driven by events and hence the name event-driven.

Event-driven architecture can be used to solve consistency problems between distributed and microservice applications where using two-phase or distributed transactions is difficult or impossible because of lack of support in databases. (6) By using the event-driven architecture, applications can use events to tell other systems that they need to perform a certain task. If the task fails, they can publish an event to signal the failure so that other listening systems can make changes necessary to recover from this failure.

One of the advantages of using an event-driven architecture is that you have a loosely coupled system. This means that the components and services of your system can have little knowledge of each other. An important benefit you get from loose coupling is the system becomes much more scalable. (7)

The loose coupling of an event-driven system can also be problematic as it becomes difficult to see the flow of the events in the system as it grows. The only way to figure out the flow is to monitor the system. This can make it hard to modify and debug the system. (8)

2.2 Design

In this section, we will first take a look at software development approaches followed by a selection of design patterns we discuss later in the report or have applied in our solution. Software design approaches are important to understand the relationships between modules and what functionality each module has.

2.2.1 Domain-Driven Design

The idea behind domain-driven design (DDD) is to design software applications in such a way that it automates or emulates a real-world process or system. The software development team will work together with a domain expert to come up with a conceptual description of the system using a ubiquitous language (UL). (9)

This conceptual description is a model that the system developers can use to build software, while still being able to collaborate with anyone who is involved in the project by discussing and making complex design decisions.

Among several modeling terms, Eric Evans describes an aggregate as, "…a cluster of associated objects that are treated as a unit for the purpose of data changes…". (9)
An aggregate thus defines the consistency boundaries for groups of related entities. When you request or store aggregates you want to make sure that transaction isn't crossing aggregate boundaries. (10)

2.2.2 Command-query separation

The term CQS was first introduced by Bertrand Meyer. (11) CQS is based on having command and query methods. Only command methods can create side effects like changing the state of objects. Query methods should only return information and they should not create any side effects.

CQS can be implemented in OOP languages by having getters that return values and does not change state. The setters will have a void return type and only change the state.

2.2.3 Command-query responsibility segregation

CQS and CQRS are similar concepts, but CQRS is applied on a higher level. CQS talks about separating command and query methods. While CQRS talks about separating command and query messages and having separate objects to handle these messages. (12) For example, splitting a CustomerService class into a CustomerWriteService and CustomerReadService. Where commands are handled by the CustomerWriteService and queries are handled by the CustomerReadService. Udi Dahan and Greg Young were the first to mention CQRS. (12,13)

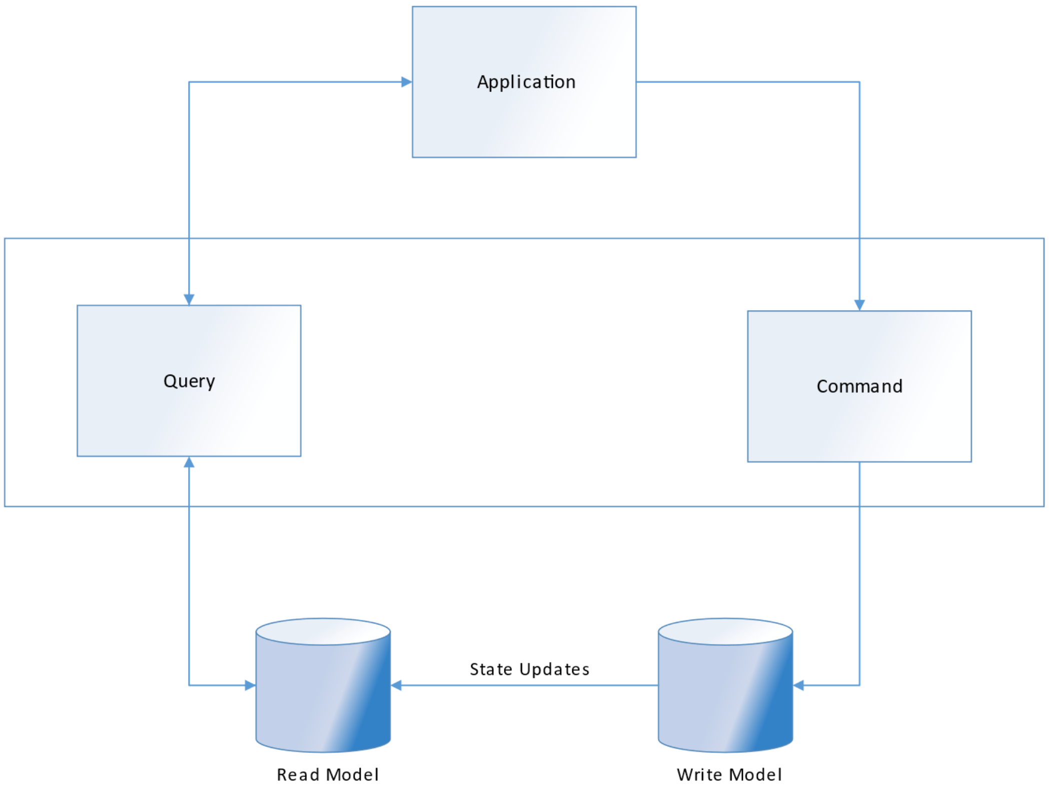


Figure 1: CQRS model.

With CQRS there are separate read and write models. Commands received will be validated against the write data store, if the command is valid it will be applied to the write data store to mutate its state. When the changes have been applied to the write data store the changes will be published as a message or event to the read data stores which can update their state based on the changes to the write data store. See Figure 1 for a visualization.

This separation allows you to build multiple read models which can be optimized for the queries your system requires. These read models can have a different structure from the write model. This allows for more efficient queries and avoids conversions of normalized database rows into objects which can be difficult. You can keep the write model normalized for optimized writes while you have read models that are denormalized to allow for efficient queries. (14) It also allows you to scale the read side of the system independently of the write side. This can be advantageous in a system that has more read operations performed than write operations.

Commands could also be handled asynchronously. If for some reason the write data store is down or have other problems, the commands can be stored in a queue and processed whenever the write data store comes online again. (13) However, this does require some way of notifying users the result of the command when it eventually gets processed. This could be solved with web sockets.

CQRS can add several benefits in some systems, but it doesn’t fit everywhere. In many systems, it works well to have the same data store for write and read operations. In these cases, it can add a lot of complexity if you attempt to apply CQRS to them. (15)

The messaging style between the write and read model makes the read model eventually consistent which is something the developer must consider when developing a system with CQRS.

2.2.4 Design Patterns

Design patterns help solve common problems in software development. Applying design patterns saves time by reusing solutions others have found before you. It can also help create a better understanding among the developers as it creates a common vocabulary for certain classes and objects that help developers recognize the purpose of classes and objects.

2.2.4.1 Visitor

The visitor pattern lets you define a new operation to perform on an object structure without altering the classes of objects it operates on. When dealing with polymorphic types the visitor pattern can be useful if you need to perform operations based on the concrete type of an object. The visitor pattern works well when the type of operation to perform is changed, but it can be difficult to maintain if the class structure changes often. Adding new elements to the class structure requires updating all implementation of the visitor. The visitor can also accumulate state as it visits elements. (16)

2.2.4.2 Façade

The façade pattern is a way of hiding the complexity of multiple subsystems by implementing a façade class that will provide a simple interface that the client can use to access the system. Apart from simplifying and unifying subsystems, the façade pattern also great for avoiding tightly coupled clients and subsystems. (17)

2.2.4.3 Builder

The builder pattern is a way of separating the construction of a complex object from its representation, this will allow the same construction process to create different representations. (16) The benefit of this pattern is that it allows encapsulation of how a complex object is constructed by allowing the objects to be constructed in multiple steps. This also means that you can hide the representation of the product from the client and because the client only sees an abstract interface you can easily swap product implementation. (17) The builder pattern can also involve validation of the parameter value passed to it. To ensure that the required parameters are set and the parameter values set are allowed. (18)

2.2.4.4 Dependency Injection

Dependency injection is a technique where an object declares its dependencies and another object, or a framework will inject the required dependencies when the object is instantiated. This technique enables loose coupling because the object that has its dependencies injected does not need to know about the creation of the objects that are injected.

2.3 Distributed system and processing

In this section, we will describe what a distributed system is and what technologies are being used to handle communication between components on a network.

2.3.1 Distributed Systems

Distributed systems are systems where multiple computers are involved by passing around messages to each other over a network. (19) Some of the benefits you get when developing distributed systems are resource sharing, openness, concurrency, scalability and fault tolerance. These benefits also create some of the main design issues that need to be considered, because these systems are larger and more complex to design. (1)

2.3.2 CAP Theorem

In 2000 Eric Brewer stated that a distributed data store could only provide two of the following 3 guarantees: (20)

- Consistency
  - All reads occurring after an update should see the updated state.
- Availability
  - All requests will receive a response.
- Partition tolerance
  - The system should continue to operate even when messages are delayed or even dropped between the nodes in the data store.

It was later shown to be impossible to provide all three guarantees. (21)

2.3.3 Consistency

One way of dealing with the CAP theorem is to relax the systems consistency guarantees. This involves reducing the consistency guarantees from a strong consistency to a weak consistency.

2.3.3.1 Strong consistency

After an update is performed any read access should return the updated value. (22)

2.3.3.2 Weak consistency

After an update is performed it is not guaranteed that a read access will return the updated value. There is an inconsistency window, which is the period between an update is performed and the moment when any observer is guaranteed to see the updated value. (22)

2.3.3.3 Eventual Consistency

Eventual consistency is a form of weak consistency. It makes the guarantee that if no new updates are performed on the data, it will eventually reach the state of the last update performed. (22) Before the last update is performed the data is said to be stale.

2.3.4 Serialization/Deserialization

When working with a distributed system, you often have to pass objects around to the different applications in the system. This involves serialization and deserialization of the object being sent.

Serialization is the process of converting data structures or object state into a format that can be stored or transmitted. It is also possible to later reconstruct the serialized data in the same or another computer environment, also known as deserialization.
Two common serialization formats are XML and JSON. These formats are language

independent which is very useful because you don't have to depend on a specific programming language or platform.

2.3.4.1 JavaScript Object Notation

JSON stands for JavaScript Object Notation is a lightweight-interchange format that is easy for humans to read and write. It's also easy for machines to parse and generate which makes it great for transmitting and storing data. (23)

A JSON formatted string can be created using two different kinds of structures. The first structure is a collection of name-value pairs that define a JSON object. The second one is an ordered list of values, which is known as a JSON array. So then, a JSON string can be a list of objects in the form of an array or it can be an object that contains a list and so forth.

2.3.5 Unique Universal Identifier

UUID stands for Universal Unique Identifier and can be used for generating a value that is almost certainly going to be unique. It is a 128-bit number that is made up of 32-hexadecimal digits. This makes UUIDs a great alternative to use as an identifier in a database because we know in advance that the chance of two UUIDs being the same is negligible. (24)

In distributed systems, it's difficult to make sure that you'll generate unique numbers when hundreds or thousands of values must be generated every minute or second. However, with UUIDs you can be confident that a generated value is going to be unique, a good example of how this could be useful is that means you don't have to check with e.g. a database if the id already exists. You can simply just set the unique identifier of that object or entity on the client side when the object is created.

2.3.6 Communication

When developing distributed applications, we need to facilitate communication between the applications. In this section, we will mention some common types of communication for these types of systems.

2.3.6.1 Hypertext Transfer Protocol

"HTTP is an application-level protocol with the lightness and speed necessary for distributed, collaborative, hypermedia information systems." (25)

The HTTP protocol is built on top of the TCP protocol for reliable transport. It is based on a client-server architecture. Where the client sends an HTTP request, and the server answers the request with an HTTP response. It is possible to have an application acting as both a client and a server at the same time.

Each request sent by a client defines a method. Methods indicate what action the receiver should perform on a resource identified by the URI included in the request. There are several different methods that can be used, but some of the more common ones are: (25)

- GET
  - Requests a resource identified by a given URI.
- POST
  - Requests that the server to accept the data included in the request.
- PUT
  - Requests that the server accepts the data included in the request if the data already exists with the given URI the data in the request should be considered a modified version that the server is requested to accept.

- DELETE
    - Requests that the server deletes the data identified by the given URI.

A developer does not need to follow these method definitions strictly. But if the system developed should be publicly accessible, it makes it easier to understand for others, if developers use the methods as defined in the HTTP RFC.

Every response message contains a status code, this code is used to indicate what the status of the request is. The status code is defined by a three-digit number and there are 5 different categories for these response codes: (25)

- 1xx: Informational – Not in use
- 2xx: Success, the request was successful
- 3xx: Redirection, Further action must be taken to satisfy the request
- 4xx: Client Error – Request has bad syntax or cannot be performed
- 5xx: Server Error – The server failed to fulfill an apparently valid request

The first digit of the code defines what category the code belongs to. Each status code consists of the status code integer and a reason phrase. Example: 404 – Not Found. This code is a client error code. It is mostly used to tell the client that the resource defined in the request, could not be found. The reason phrases defined in the RFC are only recommendations, it is not required to use them as defined. However, the category definitions of a status code are not optional. (26)

2.3.6.2 Representational State Transfer

Some of the earlier standards for web services were heavyweights. They had much more functionality than many systems required, and there was a lot of overhead involved in creating, processing and interpreting the messages. A more lightweight approach was developed, this was the REST architecture. (1)

The REST stands for Representational State Transfer. (27) The architecture was developed alongside the HTTP protocol. REST uses the HTTP protocol for transferring data and uses the URI, methods and status codes built into HTTP.

A REST web service allows clients to perform stateless operations on the resources it provides access to. When the REST service receives a request, it will look at the URI and method defined in the HTTP request to decide what to with the request. The URI is used to identify what resource the action should be applied to and the method is used to identify what action to take on the given resource.

A benefit of using REST interfaces is reduced coupling between systems. The client systems and the server systems can evolve independently if they stay consistent with the defined interface.

2.3.6.3 Messaging system

While HTTP and REST work well for many use cases, it is not the best solution for communication between microservices and distributed applications. Using synchronous HTTP calls can even be an anti-pattern in some cases. (28) Asynchronous HTTP through polling is possible but can result in many unnecessary HTTP calls being made.

Another solution to this kind of communication is to use a messaging system. This enables asynchronous communication between services. Some messaging systems also provide persistent message queues which can be used to buffer messages for a service if the service goes down. When the service comes back up it will start reading from the log of buffered messages and continue its operation.

2.3.7 Two general problem

The two general problem is a thought experiment named by Jim Gray. (29) It involves coordinating the actions of two generals when they are communicating over an unreliable link. It is similar to the Byzantine generals problem described by Lamport et al. (30)

The problem involves coordinating an attack on a city between two or more generals. They can only communicate with each other by messengers. These messengers have a small change of getting lost after they leave their camp to deliver a message to another camp.

With two generals named X and Y. A messenger leaves X to deliver a message to Y. To be sure that the message reached Y, a messenger will travel back to X to confirm this. If this messenger is lost, X is not sure if Y got the initial message and Y is not sure if X got the confirmation. If the confirmation reaches X, Y is still not sure if X got the confirmation. Therefore, X must send a confirmation back to Y. This will just keep going and both generals will be uncertain if the other general got his message.

> *There is a simple proof that no fixed length protocol exists: Let P be the shortest such protocol. Suppose the last messenger in P gets lost. Then either this messenger is useless or one of the generals doesn't get a needed message. By the minimality of P, the last message is not useless so one of the general doesn't march if the last message is lost. This contradiction proves that no such protocol P exists.*

- Jim Gray, Notes on Data Base Operating Systems p465. (29)

2.4 Storage and caching

Most applications and systems store data. Often by persisting the data in files or databases. Data can be copied to a faster storage medium than the original copy, this is called caching.

Examples of caching can be moving data from an HDD or SSD to memory or storing data retrieved over network communication to an HDD or SSD for faster access next time it is required.

2.4.1 Cache replacement algorithms

In both hardware and software caches are often implemented to increase performance. In hardware, a cache is a fast memory that stores copies of data. Accessing this cache is more efficient than accessing the original data. (31) Caches can also be implemented in software. For example, you can keep data loaded from a file or a database in some data structure in application memory for more efficient access if used repeatedly.

There is usually some limitation to the size of a cache. Therefore, you can end up in situations where you need to evict data from the cache, to make room for new data. To help decide what data to evict we use a cache replacement algorithm.

2.4.1.1 Least Recently Used

The LRU algorithm is a cache replacement algorithm based on evicting the data that has not been used for the longest time. This requires having some timestamp associated with the data. The timestamp is updated for every access to the specific data entry. The timestamp is used when the cache is full to decide what data to evict if a data entry needs to be added. The algorithm finds the oldest entry by timestamp and removes the entry from the cache.

2.4.2 Databases

A database is an organized collection of data. It allows you to structure data using different kinds of database models. Something all the models have in common is that pieces of information are linked together somehow so that the computer quickly and easily can access the data you’re looking for. (32)

2.4.2.1 Relational database

The relational database was invented by Edgar F. Codd at IBM in 1970. The data stored in tables which is made up of columns and rows with data stored inside it. It is then indexed so that it will be easier to find relevant information. Through its lifecycle the data can be read, changed/updated and/or deleted. (33)

Additionally, each row has a unique identifier called a primary key. These unique identifiers can be used to link together rows in different tables and thereby creating a relationship between tables.

Each table has a single primary key which can be used to define the relationship between tables. When a table row is connected to a row in another table, it has a foreign key column that references the primary key of the table in connects to.

In a database table, each column must have a name and a datatype. While the name doesn’t have to be specified in a specific way, there are conveniences for it. The name describes which attribute the column will have and the datatype describes what kind of data is going to be stored in it. The column’s type defines what kind of values it can contain. For instance, if the column is supposed to contain numbers, it will be defined as such and an error will be generated if you try to insert a new record that doesn’t contain exclusively numbers in that specific column.

Below is a list of the datatypes we have used:

- UUID – Universal unique identifier. Formatted as a 32-digit hexadecimal number.
- nvarchar – A Unicode string variable, with a max size between 1 to 4000 characters.
- CLOB – Character large object. Used to store large character data, such as XML or JSON strings.
- timestamp – A unique number that stores exact time and date for when a database entry was created or updated.

A table can have multiple foreign keys, but only one primary key. This can create different kinds of relationships between tables known as "one-to-many", "many-to-one", "one-to-one" or "many-to-many".

An example of a "one-to-many" relationship could be the relationship between a customer table and an order table. A single row in the customer table would be able to link to multiple rows in the order table, but not the other way around. Each order would only have one link back to the customer table, and this is all handled using primary- and foreign keys.

To access, update and manage data in a relational database management system (RDBM) we use Structured Query Language (SQL). It's a simple language that was designed at IBM by Donald D. Chamberlain and Raymond F. Boyce after they had learned about Edgar F. Codd's relational model. (34)

A benefit of the relational database is that it has been around for quite some time now which means that it's easier to understand when sufficient standards exist for the approach. The database language (SQL) used to query data has ISO level standards that specify the grammar and usage, in addition to the language being easy to use and learn.

One of the problems with relational databases is large join queries. As the database grows and more relationships are created, the join queries become difficult to perform efficiently. To make these kinds of queries more efficient a graph database could be used.

2.4.2.2 Graph database

A graph database uses the graph data structures which consists of vertexes/nodes, edges/relationships, and properties. The vertices can be viewed as an entity such as a student, a class or a subject. The edges can be viewed as relationships between the students, classes they're attending or subjects they are studying. Graph databases are thus built from entities and the relationships that exist among them. (35)

In a relational database, you strictly must follow the data structure of your data tables. Some of the benefits of using a graph database are that you are allowed more flexibility towards the data structure, which allows you to easily change the structure without having to make several changes in other nodes. (35) They also provide fast queries based on the relationships between nodes and complex interactions between them.

2.4.2.3 Index

An index is a data structure that enables faster searching in a database at the cost of additional writes and storage space to maintain the index data structure. (36)

2.4.3 Create Read Update Delete

The acronym CRUD is short for created, read, update, and delete. These are the four basic functions of a persistent storage, such as a relational database. These functions or command are used to get data into and out of a database. (37) This model allows you to create data, read data from storage, update the current data with new values or delete existing data. (38)

2.4.4 Event Sourcing

Event Sourcing is an alternative to CRUD. It can be described as a pattern or an architecture which allows you to capture all changes that have been made to a system as a sequence of events. Below is a closer look at what an event is, as described by Betts D et al.: (14)

- Events are something that has happened in the past.
- In event sourcing an event is immutable, it cannot be changed or undone. To nullify an event, you could create a new event that counteracts it.
- An event has a single publisher and can have multiple consumers that may receive the events.
- When using event sourcing, an event usually describes some business intent. The name of the event is typically described in the past tense.

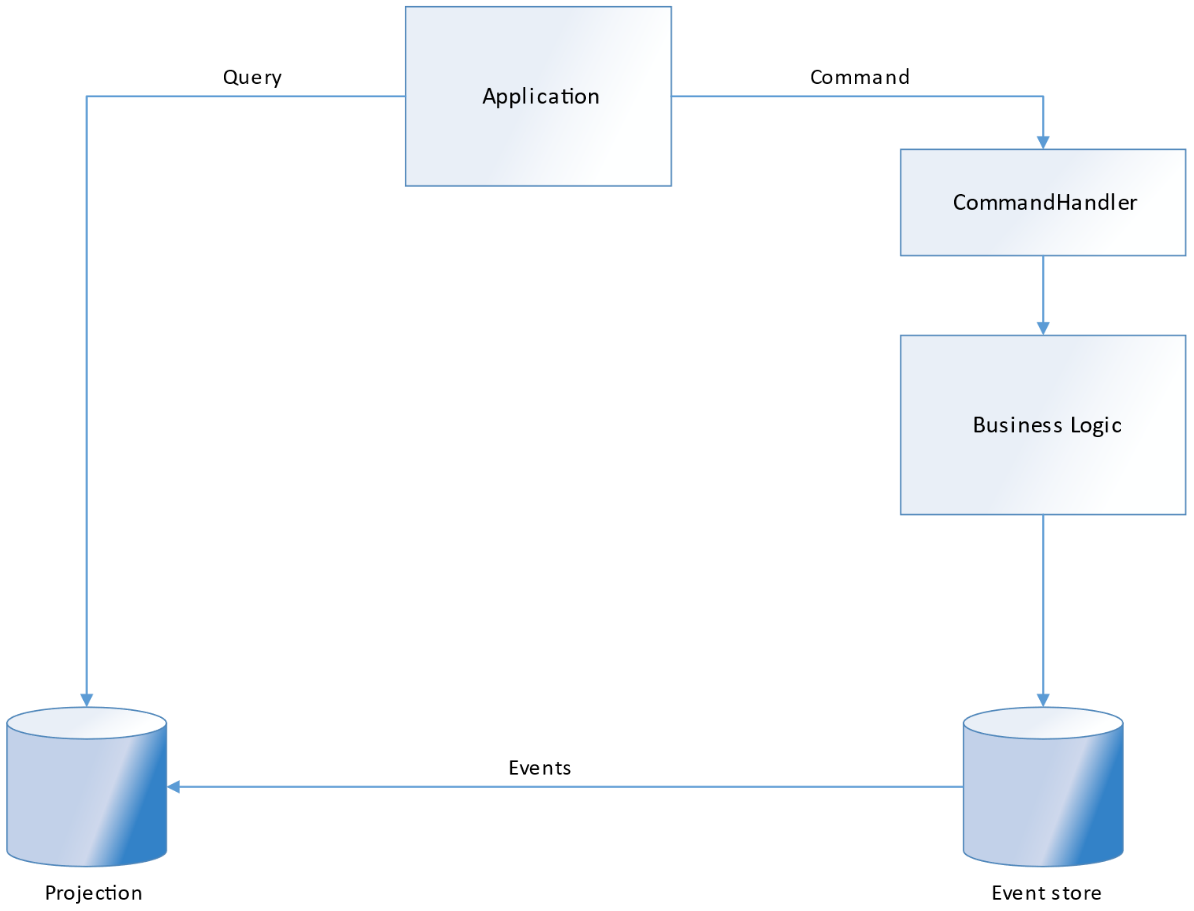


Figure 2: CQRS with event sourcing.

When using CQRS with event sourcing an event is typically a result of a command. The command term in this context comes from the CQRS pattern described in 2.2.3 and is defined as an operation that effects some change to the system.

It is not required to use CQRS with event sourcing, but the two patterns do however accompany each other nicely. In most event sourced systems without CQRS, you will encounter a problem when trying to query for a specific resource, that is because the event store stores the changes made to the system and not the current state. (39) For example, querying for all customers named Bob. To find all customers named Bob you would need to load all customer events. Then rehydrate all customer aggregates, so that you could check if their name is Bob in their current state. With CQRS you can build read models that keep the current state and use these for such queries.

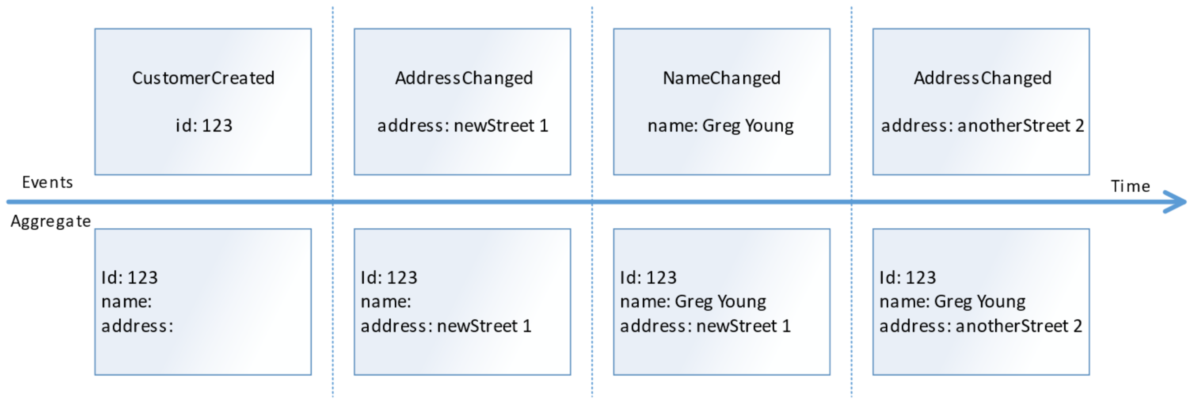


Figure 3: Illustration of how events can be applied to aggregates.

When the event sourcing system receives a command, the command is validated for required values and business requirements. If the command passes validation it will result in one or more events being created. These events will be applied to an aggregate which is defined in the command. Aggregates stem from domain-driven design as described in 2.2.1. In event sourcing, each aggregates state is built up from all the events belonging to it.

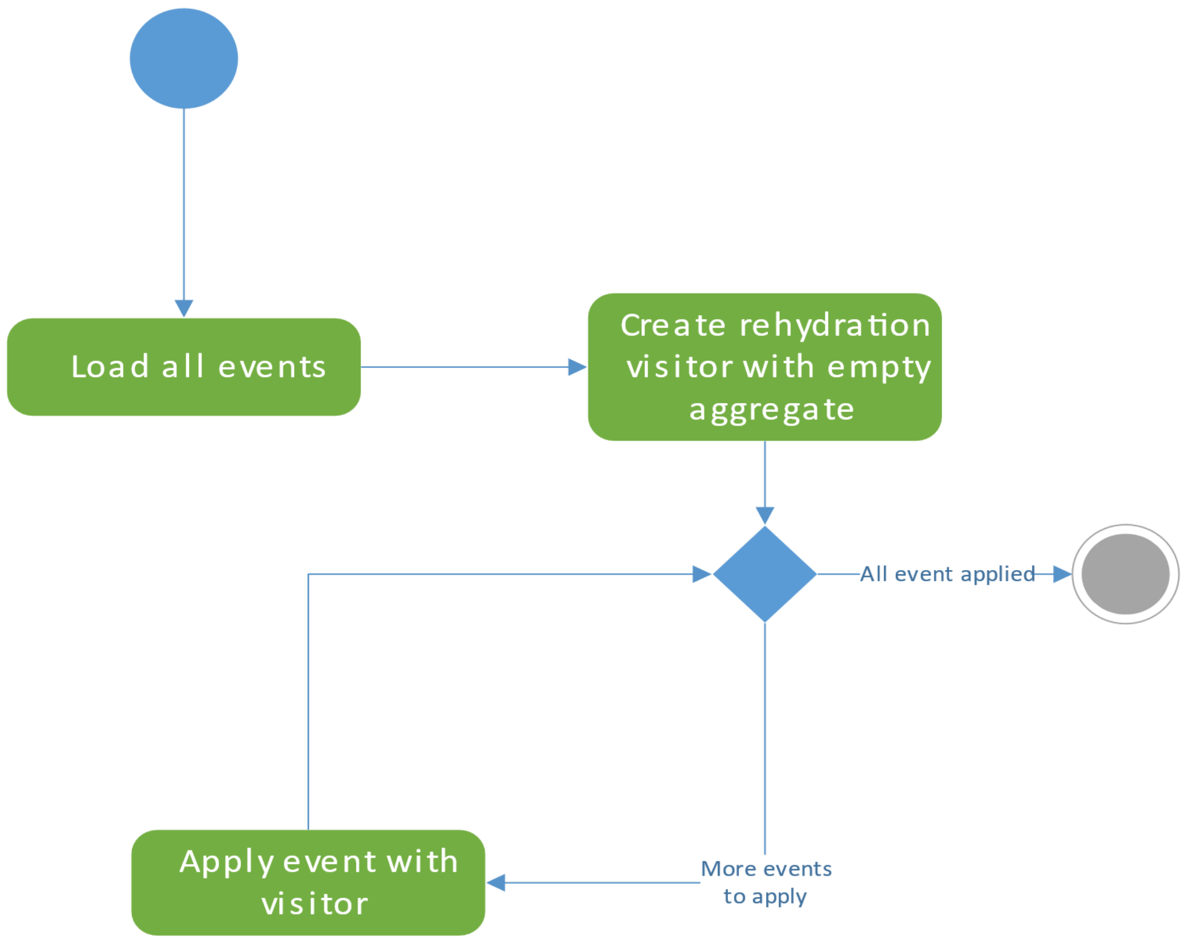


Figure 4: Aggregate rehydration activity diagram.

The events belonging to an aggregate that have been saved to the event store should never be updated or removed. Instead of removing or updating events to correct for errors you should create a new event that counteracts the effects of the problematic event. (40) But there are cases where updating events may be useful, for example when migrating events to a new schema. (41)

When validating a command, it may be necessary to check the current state of an aggregate. For example, you probably don't want an order line to be added to an order that has already been processed and sent to a customer. To check for things like this you need to rehydrate the aggregate. The process of rehydrating the aggregate involves querying the event store for all events belonging to the aggregate sorted from the oldest event to the newest. Then you apply all these events to an empty aggregate. The result is the current state of the aggregate. (14) See Figure 4 for an activity diagram describing the rehydration process.

Depending on how you define your aggregates and how long the system has been running, the rehydration process can become slower over time. Even if the events are small and the operations to apply them are performed quickly, it may start to become a bottleneck in your system if you have thousands of events that must be applied to an aggregate to check the current state of it. To solve this problem, you can implement snapshotting.

A snapshot is the state of an aggregate at a certain point in time, that have been persisted or cached in memory. Using snapshots, you can avoid having to load the full event stream of an aggregate. Instead, you load the events that have occurred after the time the snapshot was made. This way you load the aggregate state from a snapshot and only apply events that occurred after the timestamp or version of the snapshot. (40)

2.4.4.1 Advantages

Event sourcing is useful to have because your applications current state will always be build up from a sequence of past events. That means you will be able to build up your applications state to any point in time, which will allow you to easier test and debug a system because you can always go back and recreate what has already happened.

A feature of event sourcing known as smoke testing can be used to simulate the use of a real system to make sure that it will work properly before it's used in production. If you have an event store with all the changes leading up to the current state, you can rerun those events and if you get the state you were expecting you can be confident that the tested system will work in production as well.

Another major benefit is that you are going to have full traceability of a system which supports business intelligence. You will be able to analyze the data you have and reveal interesting patterns in your business analysis. Furthermore, you'll be able to improve certain aspects of your business intelligence and correct errors that previously weren't visible.

When everything that happens in the system is made up of events, it would also be natural to have the system be driven by these events. That is where the event-driven architecture that is described in chapter 2.1.3 comes in.

2.4.4.2 Disadvantages

We must deal with the eventually consistent nature of the system. If command handling is implemented asynchronously, other applications using the system must be aware of this and should be developed to handle error situations that can occur after they have sent a command to the event sourced system. In an order system, situations can occur where two users place an order for an item that there is only one left off in stock. If the orders are placed around the same time, the first users' command will reach the event store and is accepted. But the eventually consistent read models may not have updated with this information before the second user submit their command. This command will be rejected at the event store and the application will have to handle this. Either by displaying a message in the UI or by sending an email notifying the user that the order could not be placed.

Event sourcing is more complex than CRUD. Most developers have worked with systems using CRUD, while event sourcing may not be as well-known as CRUD.

The event store does not have an explicit schema for events. In an SQL database, there is an explicit schema to follow with defined values and data types. In an event store, all events of an aggregate are stored. These events can differ in what values they hold and what datatypes are used. Therefore, the event stores schema is implicit. It’s up to the event store to be aware of how events have been stored. This makes schema evolution and data conversion more difficult. (42)

2.4.4.3 Schema evolution

There are a couple of techniques that can be used to handle schema evolution for event store, as described by Overeem et al.: (41)

- Multiple versions
    - This technique uses multiple versions of an event. A version number is used to extend the event structure, which can be read by all the event listeners. The event listeners should contain knowledge of different versions of events to support them. The event store will remain intact because older version won't be changed.
- Upcasting
    - This technique uses a component known as an upcaster that will change an event before giving it to the application. This differs from the multiple versions technique because the event listeners are not aware of the different versions of events. The listeners only need to support the latest version because the upcaster changes the event.
- Lazy transformation
    - This technique also uses an upcaster to change the events before they are given to the application. Additionally, the result of the change is stored in the event store. The change is therefore only applied once for every event. On following reads, the change is no longer necessary.
- In place transformation
    - This technique is typically used with NoSQL databases as described by Betts et al. (42). This technique involves reading the data from an event store, transforming it to use the new schema and then write the updated data back to the database.
- Copy and transformation
    - This technique involves copying and changing events before sending them to a new store. This means that while the new event store is created, the previous event store will stay intact.

2.4.4.4 Projections

A projection is used to build the current state of your application from a stream of events. It takes the event stream and projects it to a structural representation. A projection can be projected to any structural representation. For example, a relational database like Postgres or MS SQL, or a graph database such as Neo4j. (43)

2.5 Existing solutions

We didn’t manage to find any existing solutions that specifically involved implementing an order- and warehouse system using event sourcing. However, we did find some examples of applications using event sourcing and CQRS together, or similar concepts in other types of systems.

2.5.1 Relational database transaction log

Relational databases use transaction logs to keep track of changes and to recover from crashes. This is similar to event sourcing in many ways, but events capture intent along with what happened. (14)

2.5.2 Learning analytics

Stein Kjetil Sørhus at NTNU used event sourcing in his master thesis about applying learning analytics to better understand how students handle OOP programming exercises. Event sourcing was used to store events that were collected from student running automated tests while working on programming exercises. The data was then used for analysis. (44)

2.5.3 Analysis of learning analytics system

Andreas Haugen Pedersen at NTNU analyzed the system made by Stein Kjetil Sørhus mentioned in 2.5.2. The analysis was done to check if the learning analytics system followed event sourcing and CQRS principles. (45)

2.5.4 Evaluation of NoSQL databases for event store implementation

Johan Rothsberg at Linköping university explored the possibility of using NoSQL databases in an event store implementation. He compared an existing relational database implementation with an implementation using a Neo4j graph database. He concluded that the existing implementation using a relational database performed better than the NoSQL alternative. (38)

2.5.5 Applying CQRS to increase performance

Rajković et al. show that the CQRS pattern can be applied to improve response time and reduce the amount of data transferred in a medical information system. (46)

2.5.6 Flight Scheduling

Debski et al. developed a prototype flight scheduling system using event sourcing and CQRS. They used Akka for message passing between entities; Scala language for development; Apache Cassandra for the event store implementation and Apache Kafka as a persistent message queue. They also performed some scalability tests on the applications read and write parts. (47)

2.5.7 Akka

Akka is a framework for developing distributed systems using the actor model. It uses an event sourcing model for persisting state of actors in the system. Their implementation also has snapshot capability. (48)

2.5.8 Eventuate

Eventuate is a framework for developing asynchronous microservices. They have two versions, one is Eventuate Tram which uses JDBC or JPA for persistence and the other being Eventuate ES which uses event sourcing. (49)

2.5.9 Event Store

Event Store is an open source event store implementation of an event store. It has a .NET and HTTP API. (50) There is also a JVM API on GitHub that works with Scala and can be used from Java by using tools from Akka. (51)

2.5.10 NEventStore

A library for abstracting different storage implementations when event sourcing is used for data storage. (52)

2.5.11 Blockchain

The concept of blockchain technology is very similar to how the event sourcing pattern works. You have a type of storage or ledger that is made up of a chain of events and you can only append events to the chain. (53)

The difference is that in the blockchain technology, that we know from the cryptocurrency market, is that the ledger that stores all the events is public and that the authenticity of the ledger continuously must be verified by what is called a miner. The miners will take a block of events and try to solve a mathematical problem based on a cryptographic hash algorithm.
This part was previously done partly by individuals with huge computing power, but today large computer centres are beginning to take over this duty. (54)

The way a blockchain works is that each event is run through a cryptographic hash function which produces a unique value that is stored in the succeeding event. This means that the hash value will be changed if any changes are made to that event.
This creates a link between nodes since the hashed value is stored in the event, if you change one event the preceding event will also be changed. So, if you want to change something, you would have to change every single event ever made on that chain. This makes the blockchain very secure. (55)

3 MATERIALS AND METHOD

In this section, we will look at the software development methods, materials, libraries, and frameworks which we have used in our thesis.

3.1 Method

A brief look at the software development methods we have used. Our team only consisted of two people, so it was difficult to strictly follow any of the methods we are familiar with as they are suited better for larger teams.

3.1.1 Project planning

We have used the agile software development management method SCRUM to plan our project. SCRUM is achieved by having a group of three to nine developers develop small, concrete updates to a system in short iterations called sprints. The length of a sprint should be between one to four weeks. (1)

SCRUM defines a product- and a sprint backlog. The product backlog contains a list of issues that need to be completed in the lifetime of the project, while the sprint backlog contains a set of issues that have been taken from the backlog.

The goal of each sprint is to complete as many issues as possible before the sprint is completed. The items have been completed and tested can be put out in production. While the issues that did not get completed can be returned to the backlog and be used in a future sprint.

3.1.2 Literature review

A literature review is a way of showing the reader that you have good control over the subject. That is done by carefully surveying the literature the thesis is based on. You have to be critical of the information that's been gathered and make sure that the sources are trustworthy.

When you have analyzed the gathered material and compared the different points of view, you will gain a deeper understanding of the subject you are working with and that will ultimately contribute to a better result.

3.2 Materials

The tools we have used in our thesis are all software development and managing tools.

3.2.1 Java

Java is an object-oriented programming language. It enables us to write code in a human-readable format which later can be compiled into bytecode that can be executed by the Java Virtual Machine (JVM).

Java can run on many platforms and operating systems. Java will run on any system that has a JVM implementation. These systems can be anything from household appliances to server machines. This can save development time when an application is required to run on different operating systems.

Java is one of the most used languages when it comes to developing distributed web applications and services. For these kinds of applications frameworks like Java EE and Spring are often used. According to tiobe.com Java is currently the most popular programming language. (56)

3.2.1.1 Generics

Generics were introduced in Java Development Kit 5 (JDK) and it was described by Oracle as a way of allowing "…a type or method to operate on objects of various types while providing compile-time type safety.", this is very convenient because it makes it allows you to reuse code. Type variables, classes, interfaces, methods, and constructors can all be declared as a generic.

3.2.1.2 Enum

An enum is a data type used for representing a variable as a fixed set of predefined constants. The natural convention for the naming of an enum type's field is to write them in uppercase letters, this is because they are constants.

```
public enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

Listing 1: Example of an enum.

3.2.2 IntelliJ IDEA

IntelliJ is a software development tool. It is an integrated development environment (IDE) which supports several languages, technologies, and frameworks that can be used in the software development process. (57) It works well when developing multiple application at the same time by using the run dashboard which gives a simple way to start, stop and check the state of multiple applications running at the same time.

3.2.3 Gradle

This is a tool used for automating the building process by declaring the project configuration in a separate file. In the Gradle file, you declare which plugins, libraries or dependencies your application needs to execute. (58)

3.2.4 Git

Git is a version control system that you can use to track the changes that have been made to source code. Git allows you to make changes to a file and easily distribute those changes among the people involved. (59) An example of a version control hosting service would be Bitbucket, which is a web-based solution for development projects that use Git.

3.2.5 Jira

Jira is a project management tool that is used to track issues. With Jira, you can have a list of tasks, issues or bugs that you wish to solve during the lifetime of your project. You can then add a set of these issues to your teams' weekly workload. The people involved will then be able to see what issues have been solved, which are currently in progress and which ones are yet to be solved. (60)

3.2.6 Office 365 OneDrive

A cloud service for students that allows you to store and edit files in the cloud. This tool is provided by NTNU's SharePoint website and it's a good way of collaborating when writing a report or thesis.

3.2.7 VM AutoDeploy

VMDeploy is a service provided by the LAB at NTNU in Ålesund to host virtual machines that will run on the university's server. We used it to develop and test our applications.

3.2.8 Postman

This is an API development environment that you can use for sending HTTP requests to test your application. You can send any type of HTTP request and receive a response in either XML or JSON. (61)

3.2.9 Postgres

This is an open source object-relational database system that uses and extends the SQL language combined with many features that safely store and scale most complicated data workloads. (62)

3.2.10 Kafka

Apache Kafka is a distributed messaging service. Its basic functions are to publishing messages to topics, this is done by a producer. and subscribing to topics to be notified of new messages to the topic. Kafka can be distributed over servers in a cluster and can be configured to replicate data over multiple nodes for fault tolerance. Each partition of the system will have a leader, other nodes called followers will copy the leader's data. If the leader were to fail one of the followers is "promoted" to be the leader. Kafka will attempt to distribute the leader role for different partitions to different servers to spread the load. There are two roles for applications using the Kafka system, producer, and consumer. A producer publishes messages to a given topic and consumers "consume" messages on a given topic. (63)

3.3 *Libraries and frameworks*

A list of the libraries and frameworks we have used.

3.3.1 Spring framework

Spring Framework is an open source Java application framework. It was created in 2003 because of the complexity of the early J2EE applications. It is not a competitor to Java EE, but it integrates some selected specifications from Java EE. (64)

Spring is meant to handle all the "plumbing" involved in creating large enterprise applications and letting the developer focusing on implementing business logic. Creating an application that receives REST requests and reads or writes to a database can be done quickly with some configuration through properties files and annotations in the java code.

Spring has an inversion of control container. The container keeps track of beans, which are objects whose lifecycle are managed by Spring. This container is used to perform dependency injection.

There is a version of Spring called Spring Boot. This version comes with a built-in tomcat server. This simplifies running Spring applications as you don't need to deploy the application to an application server. The application can be built to a jar file that contains all necessary files and dependencies, including the Tomcat application server.

Spring Boot annotation that we have used:

- @ConfigurationProperties – Annotation used for externalized configuration. Used to inject configuration property values into a class or bean.
- @PostConstruct – Annotation used to define a method as initialization method which will be executed after dependency injection is done.
- @Autowired – Annotation for bean injection that tells Spring where an injection needs to happen. It allows you to skip configuration because that is handled by Spring.
- @Component – Annotation used to detect and configure beans. Spring will scan through your project and mark this as a bean.
- @Repository – Annotation used to indicate that a class is a repository. A repository is "a mechanism for encapsulating storage, retrieval, and search behavior which emulates a collection of objects". (9) This annotation also includes the @Component annotation.

- @EnableScheduling – Annotation used for detecting @Scheduled annotations and enabling a background task executor.
- @Scheduled – Annotation used for configuring and scheduling when a method can be executed.
- @EnableAsync – Annotation used for detecting and enabling asynchronous method execution.
- @Async – Annotation used for indicating that a method will run asynchronously on a separate thread.

3.3.2 Spring Data JPA

The Spring Data JPA is a way of accessing data in a relational database. JPA stands for Java Persistence API which is a set of definitions, protocols, and tools that will help you develop your software application. This means that the Spring Data JPA has all the building blocks in place so that a developer quickly and easily can map database tables and entries to java objects.

Instead of writing SQL queries you can use annotations that will tell the JPA framework which class objects will be mapped to tables a relational database and which attributes an entry might have. By using this information, the JPA framework will perform ORM mapping of database rows. Below is a common list of annotations that we have used in our thesis:

- @Entity
  - Tells JPA that this class will be mapped to a table.
- @Id
  - Tells JPA that the following variable is a primary key.
- @GeneratedValue
  - Tells JPA that the following variable will be an auto-generated value. Typically used with the @Id annotation to generate a primary key.
- @ManyToOne
  - Tells JPA that the class will have a many-to-one relationship with the specified table.
- @OneToMany
  - Tells JPA that the class will have a one-to-many relationship with the specified table.

Spring provides the developer with high-level abstractions from the database in form of repositories. These repositories are generic interfaces that the developer can extend. When extending the interface, the developer provides the type of the entity and its identifier through generics. (65)

3.3.3 Spring Framework JDBC

Spring Framework JDBC is a lower abstraction over the database than using a JPA framework. It lets the developer take care of defining connection parameters, specifying SQL statements, declaring parameters and providing parameter values and performing operations on each row of the result set. While Spring still takes care of opening and closing connections, preparing and executing statements, iteration over the result set, transactions, and exceptions. (66)

The Spring JdbcTemplate is used to connect to a database and execute SQL queries. It makes it easier for the developer by handling the creation and release of resources, as well as making it simpler to execute SQL queries or updates, checking for errors and handling exceptions.

3.3.4 Spring Web

This is a Spring module that provides support to help you create web applications. It makes use of different annotation such as @RestController, @RequestMapping, @RequestParam and @PathVariable, which very useful when you are developing a RESTful web service. Below is a short list of most common Spring Web annotations:

- @RestController – Handles HTTP request, this annotation bundles together the @Controller and @ResponseBody annotation.
- @RequestMapping – Tells Spring which type of HTTP request method the specified method should map to. Can also be replaced with @GetMapping, @PostMapping, @PutMappnig or @DeleteMapping.
- @RequestParam – Tells Spring to retrieve the URL query parameter and map it to the method argument.
- @PathVariable – Tells Spring to retrieve the URL path parameter and map it to the method argument.

3.3.5 Spring Integration Kafka

The Spring Integration Kafka is an extension to Spring which provides support for core concepts that will help you develop an event-driven Kafka application. One of the main features is the @KafkaListener annotation, this will mark a method to be the target of a Kafka message listener on the specified topics.

3.3.6 Lombok

Project Lombok focuses on trying to avoid writing repetitive boilerplate code. Getters, setters, constructors, hashCode and equals methods can be generated by Lombok.

Below you can see a short list of some of the most used annotations:

- @NoArgsConstructor – Tells Lombok to create an empty constructor for the class.
- @Getter – Tells Lombok to create getter methods for all the fields of the class.
- @Setter – Tells Lombok to create setter methods for all the fields of the class.
- @Data – This annotation does a couple of things, it tells Lombok to use a bundle of features which include some of the most common annotations. Those are the @Getter, @Setter, @ToString, @EqualsAndHashCode and @RequiredArgsConstructor annotations.

3.3.7 OkHttp

OKHttp is an open source project designed to be an efficient HTTP client. It can be used to send HTTP requests and read their response. It allows all requests to the same host to share a socket, reusing connections and threads will reduce latency and saves memory. (67)

3.3.8 Jackson

Jackson is an API used to process JSON for Java. Its main functionality is to serialize POJO to JSON strings and to deserialize JSON strings into POJO. (68) This functionality is provided by the ObjectMapper class from the Jackson library. The ObjectMapper then takes care of the conversion of the java objects from and to JSON strings. (69)

3.3.9 Liquibase

Liquidbase is a tool used for database version control. Typically, you would choose to use this tool when you expect your database schema to change over time. You will have a change-log file which defines a database schema and the changes made in it over time. You can then append changes to the file and thus have a complete record of all the changes which have been applied to your schema. Another useful feature is that it generalizes some SQL syntax which makes it easier to switch database implementation. (70)

3.3.10 Postgres JDBC driver

The Java Database Connectivity (JDBC) is an API that defines how a client may access a database. (71) The Postgres JDBC driver is then the driver you need to connect your Java application to the Postgres database management system. (72)

3.3.11 H2 Database

H2 is a database management system for relational databases. It can be embedded into Java applications. The database can be configured to run in-memory which means that the data is volatile and therefore it will not be stored on the disk. This is a good feature to have when you are developing and testing your application. Some of the advantages of using the H2 Database include extremely fast queries, it supports SQL and JDBC API, and it can be used with the PostgreSQL ODBC driver. (73)

4 RESULTS

The system has been developed as a distributed system with microservice- and event-driven architecture. Events are published to Kafka and REST requests are used for communication between the applications. For data storage, we use event sourcing. The applications have been split up to separate different functionality into smaller and more manageable applications. We ended up with a total of 14 applications in our system.

4.1 Software architecture

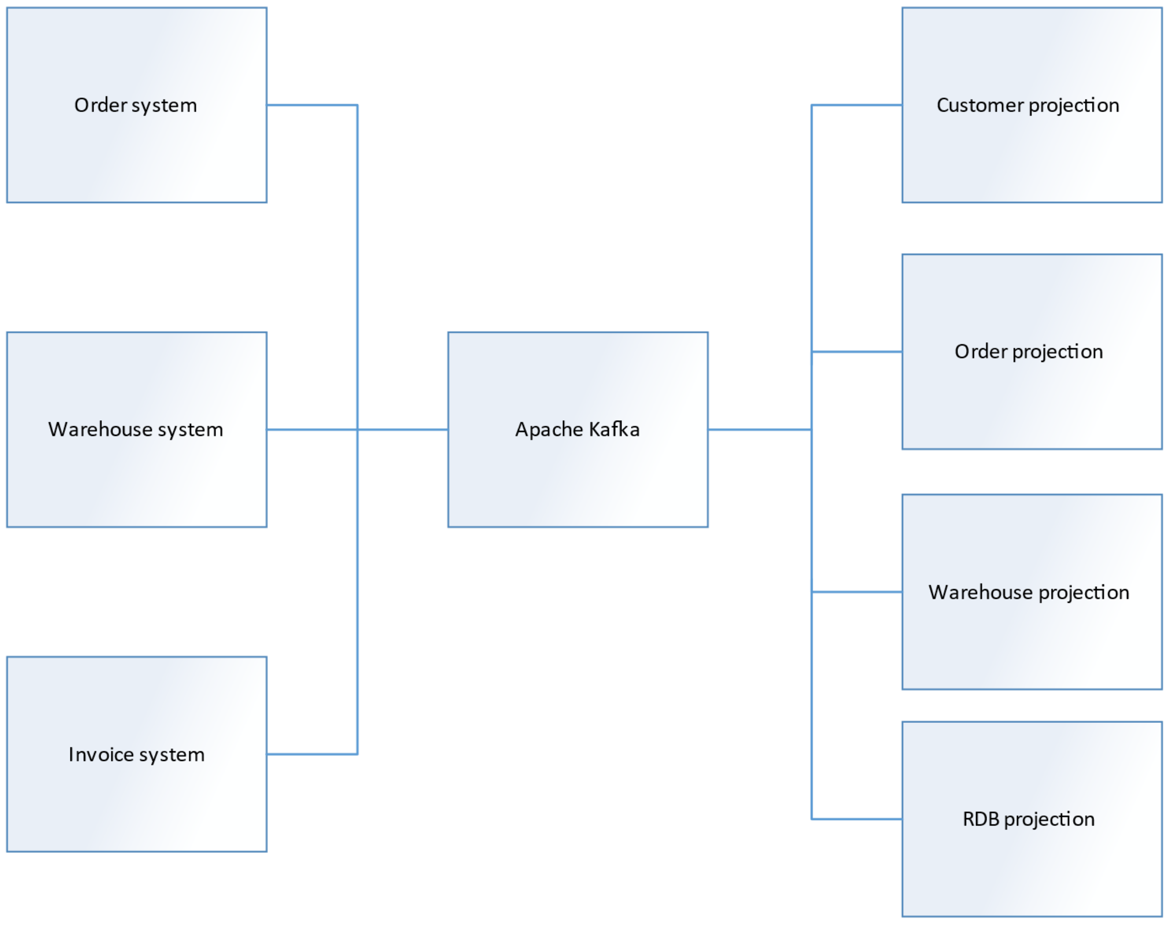


Figure 5: High-level context model of the system without simulator applications.

Figure 5 shows a high-level context model of the system, each node represents a microservice application. Some of the projection applications have been left out to make it easier to visualize.

Following the CQRS pattern, we have separated the write and read models. The components on the left side make up the command side of the system. These applications receive commands through REST requests and validate these commands against the data in their event store. If the command pass validation the changes requested by the command are applied to the event store by producing one or more events. When events are applied to the event store they are also published to their respective Kafka topic.

The query side of the system subscribe to the Kafka topics they are interested in and receive the events published to these topics. These events are used to change the data in the read model according to the events published by the command side. These read models can be queried through REST interfaces.

4.1.1 Microservice architecture

The system has been split into multiple applications. These applications have been separated based on domain and functionality. The different applications can be grouped together in four categories:

- Event store
- Projection
- Service
- Simulator

An event store application receives commands from services and simulators. These commands will be validated against the events in the event store to check if the command can be applied or if it should be rejected. If a command passes validation, one or more events will be generated and saved to the event stores database. The events created represents the changes in state, requested by the command. When the events have been saved to the database they will also be published to the respective Kafka topic. The event store makes up the write model of the CQRS pattern.

Projection applications make up the read models of the CQRS pattern. A projection application is subscribed to one or more Kafka topics based on what type of events the read model is interested in. The projection uses the events it receives from Kafka to build its read model. Information in the read model can then be queried through REST interfaces. The projection application decides by itself how the read model should be structured and how it stores it.

A service uses information it receives by querying projection to apply commands to an event store application.

A simulator application simulates user interaction with the system. The simulators will read from the projections and create commands based on the current state of the system. These commands are then sent to the event store applications by using REST requests.

The service and simulator can look quite similar in the way they work, but the simulator is meant to simulate user input to the system, while a service could perform some scheduled task that produces some commands as input based on the state of the system.

Using a microservice architecture gives us the benefit of smaller more manageable code bases for each application. It can also be beneficial to have the projections as microservices as you could just deploy multiple instances of the projections to scale up the read side of the system.

4.1.2 Event-driven architecture

The event store applications publish events to Kafka when they perform different operations on aggregates. Other applications can listen to events and perform actions based on the events it consumes.

As described in 2.1.3 event-driven architecture gives our system a low coupling between the different event store applications and projections. If one of these applications were to go offline it would have little to no effect on other applications. During this kind of downtime, Kafka will work as a buffer for the events that occur so that when the application comes online again it can start processing any events that have occurred during the downtime.

Kafka becomes a very central part of our architecture, as seen in Figure 5. If Kafka were to fail it would halt all communication between the different parts of the system. But as mentioned in 3.2.10 Kafka is designed to be distributed over multiple nodes for fault tolerance. This means that Kafka is very reliable because of the low probability that all the distributed Kafka nodes will fail simultaneously.

4.2 Aggregates

For our system we ended up with 6 aggregate types that we will go through in this section.

4.2.1 Customer

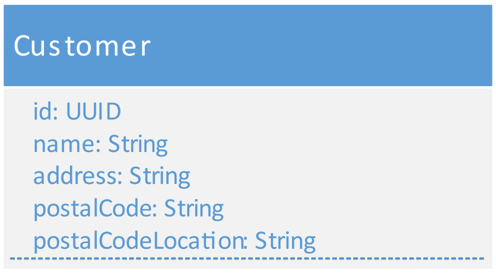


Figure 6: The Customer aggregate class.

The customer aggregate represents a customer in the system and contains:

- Id
- Name
- Address values

4.2.2 Product

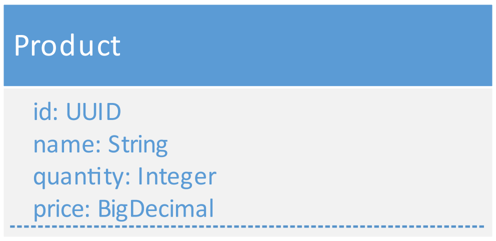


Figure 7: The Product aggregate class.

The product aggregate represents a product that can be ordered through the system. It contains:

- Id
- Name
- Quantity
- Price

4.2.3 Order

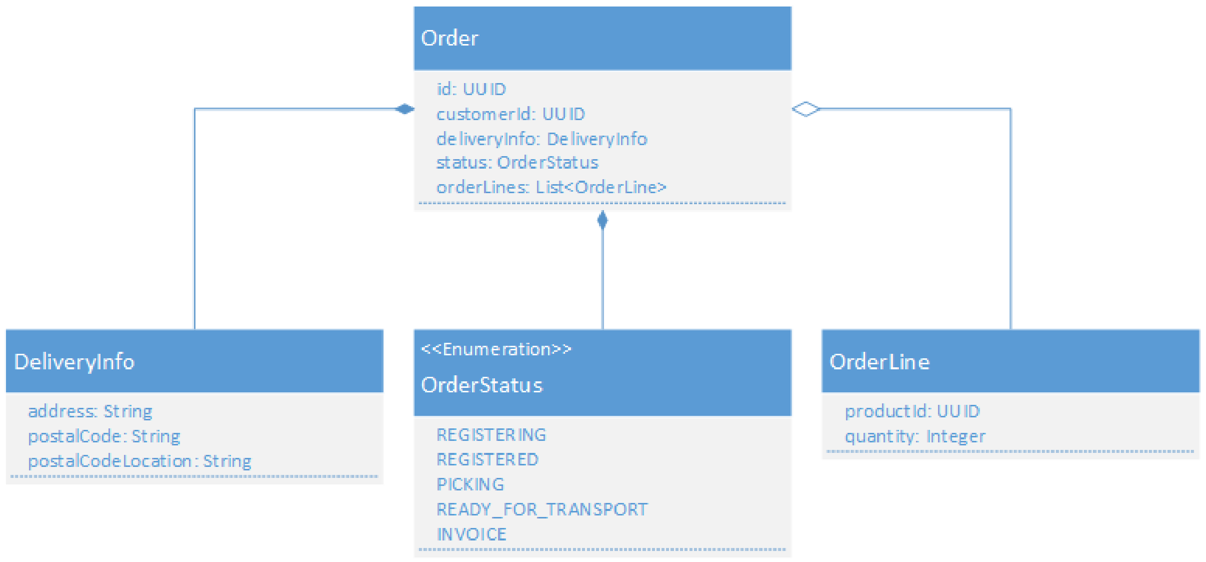


Figure 8: The Order aggregate class diagram.

The order aggregate represents an order placed by in the system. It contains:

- Id
- Customer id
- Address
- Status
- Order lines

The customer id is a UUID value that references the customer the order belongs to. The status value represents the status of the order, these status values are defined in the OrderStatus enum. An enum is used to ensure type safety during compilation.

The list of order lines contains all the order lines of the order. The order line object has a UUID reference to the product ordered and an integer value for the quantity.

4.2.4 Stock Unit

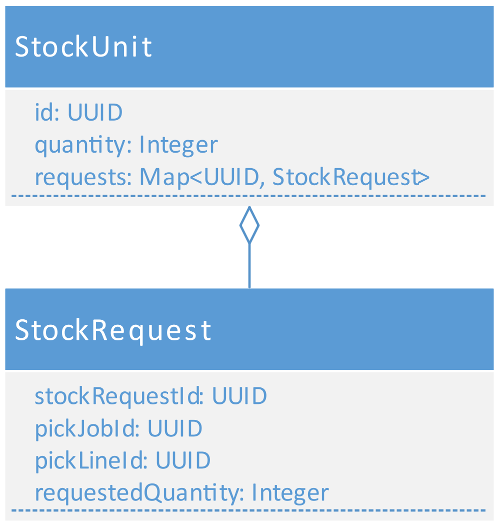


Figure 9: StockUnit aggregate class diagram.

The stock unit represents a product in the warehouse. It contains:

- Id
- Quantity
- Requests

The id UUID value is created to match the id of a product aggregate. The quantity value keeps track of the quantity currently available for picking. The requests map keeps track of requests for picking the stock. This was implemented so we only allow picking to start if the quantity is high enough.

The stock request holds references to the pick job and pick line that caused the request and the quantity needed.

4.2.5 Pick job

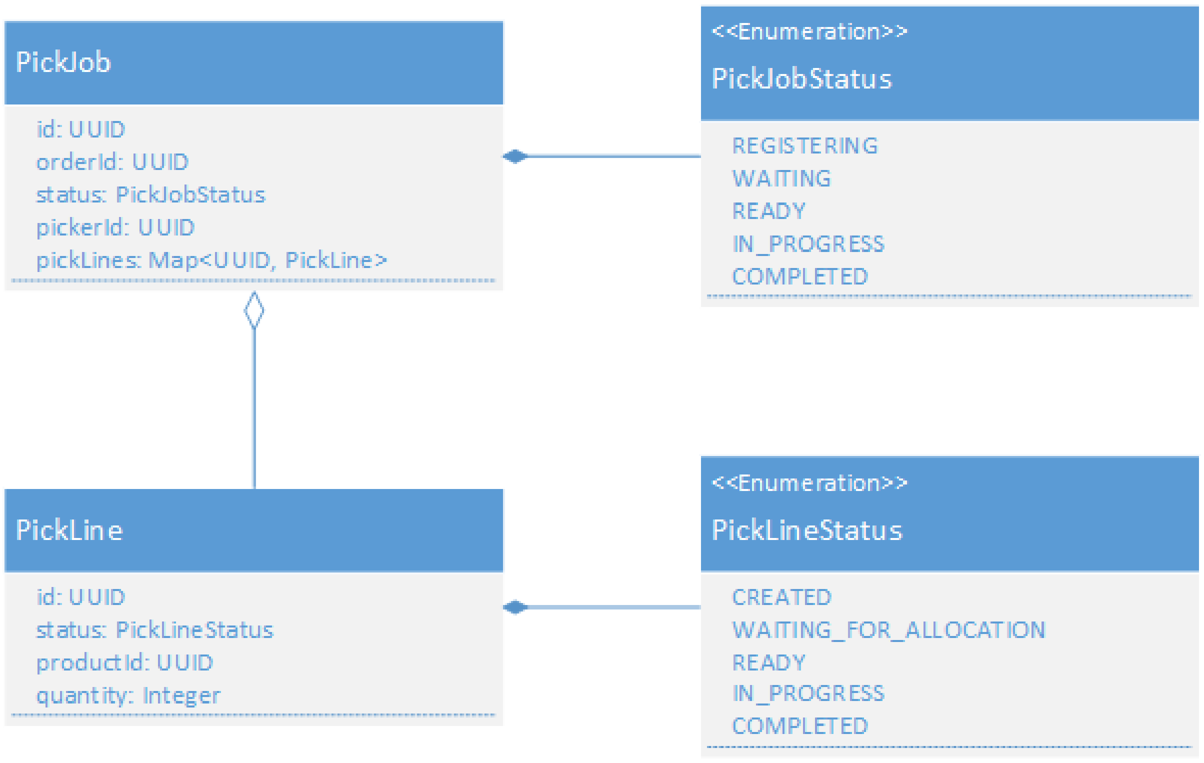


Figure 10: PickJob aggregate class diagram.

The PickJob class represents a pick job in the warehouse. Each pick job references the order aggregate it belongs to. The pick job also references the id of the picker that is working on.

The status value represents the status of the pick job, the values are defined in the PickJobStatus enum to ensure type safety during compilation.

The pickLines map contains all the pick lines that must be completed to complete the pick job. Each pick line corresponds to an order line in the order aggregate that created the pick job.

The PickLine also has an enum value that represents the status of the pick line.

4.2.6 Invoice

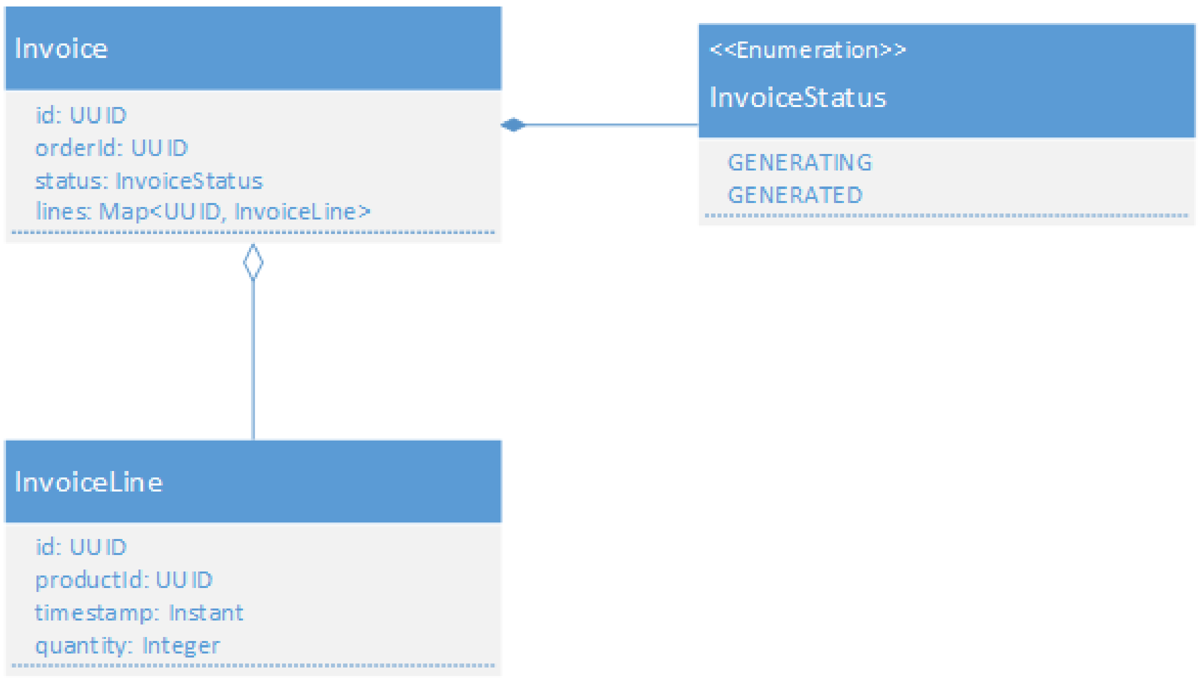


Figure 11: Invoice aggregate class.

The invoice aggregate represents an invoice for an order, the order it belongs to is referenced in the orderId field. The invoice has a status field whose values are defined in an enum to ensure type safety during compilation. The invoice has a set of invoice lines that corresponds to an order line in the order the invoice belongs to.

4.3 Kafka

```
cd ~/development/kafka_2.11-1.0.0/bin

kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor
1 --partitions 1 --topic orders

kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor
1 --partitions 1 --topic customers

kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor
1 --partitions 1 --topic products

kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor
1 --partitions 1 --topic pick-jobs

kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor
1 --partitions 1 --topic stock-units

kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor
1 --partitions 1 --topic invoices

echo "Topics created"
sleep 3s
```

Listing 2: Script used to create the Kafka topics, linux version.

The Kafka topics were set up so that there is one topic for each aggregate type. And all events belonging to an aggregate type are stored in the same topic. This results in each topic having multiple event types. Which we must deal with when deserializing the JSON received from Kafka.

4.4 Events

```
@JsonTypeInfo(use = JsonTypeInfo.Id.CLASS, include = JsonTypeInfo.As.PROPERTY,
property = "@class")
public abstract class Event {

    private UUID id;
    private UUID aggregateId;
    private Instant timestamp;

    /** Constructors etc... **/
```

Listing 3: Abstract Event class.

The event classes all inherit from an abstract event class that defines fields for all the values that are required for all events.

```
@JsonTypeInfo(use = JsonTypeInfo.Id.CLASS, include = JsonTypeInfo.As.PROPERTY,
property = "@class")
public abstract class CustomerEvent extends Event {

    public abstract void accept(CustomerEventVisitor visitor);

    /** Constructors etc... **/
```

Listing 4: Abstract CustomerEvent class with visitor pattern.

Each aggregate type has their own abstract event class that all events for the aggregate inherit from. This abstract class defines an abstract method called visit that enables the visitor pattern.

The visitor pattern is used when we need to deal with the concrete type of an event object, which is one of the ways to use it as described in 2.2.4.1. When we deserialize events for an aggregate from the event store, we get the events as objects of the aggregates abstract event type.

The @JsonTypeInfo annotation is used to handle polymorphic deserialization. The class of the event is included in the JSON object as a property named "@class". This helps Jackson figure out the concrete class of the serialized event when performing deserialization.

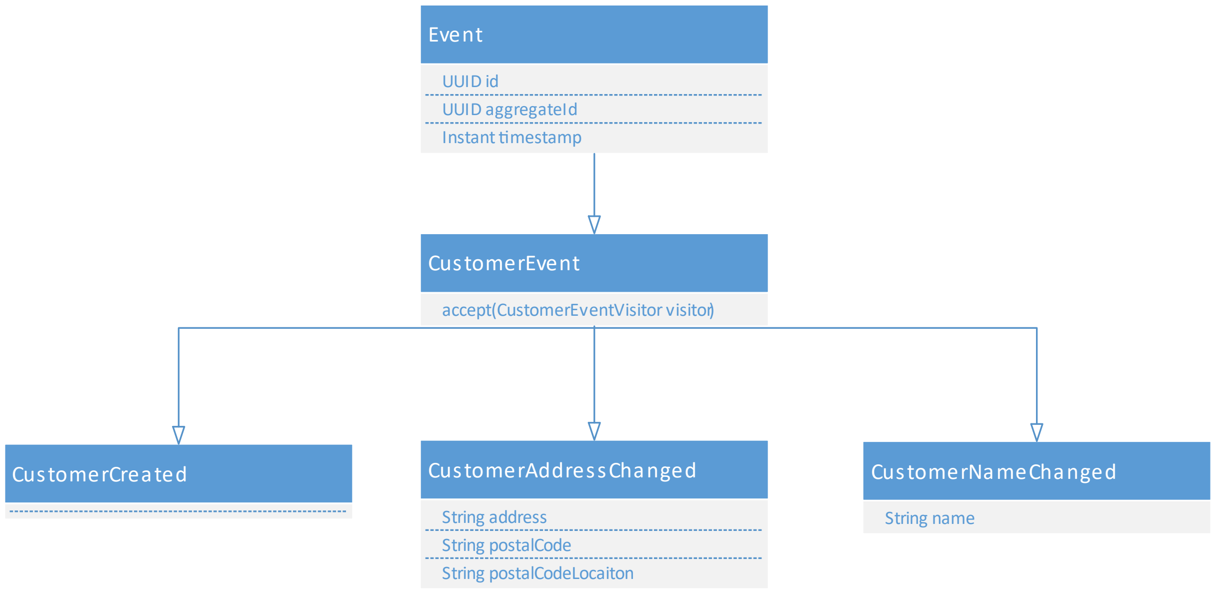


Figure 12: The visitor pattern is used to find the concrete type of an event.

Figure 12 shows an example of event class inheritance for the customer aggregate.

4.5 Library

We created a commons library that contains classes that were reusable in multiple microservice applications. This helped keep different classes and interfaces consistent between the systems. When modifying classes that multiple applications use we don't have to change it in multiple locations.

Example of classes in commons library

- Events
  - Event data structures
  - Event handler interfaces
  - Event visitor interfaces
- Commands
  - Command data structures
  - Command handler interfaces
- Aggregate classes
- Utility
  - TimeUtil
  - JsonParser
  - HttpClient
- API
  - Projection API
  - Command API

4.6 Event store

We created our own event store implementation using a Postgres database.

4.6.1 Structure

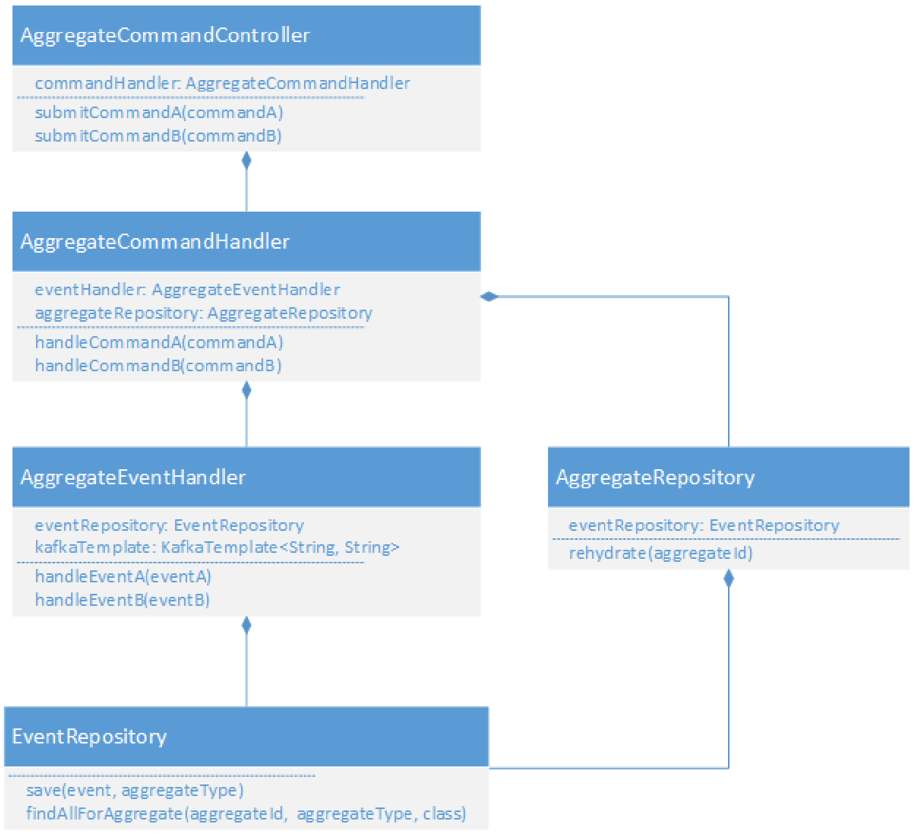

AggregateCommandController
commandHandler: AggregateCommandHandler
submitCommandA(commandA)
submitCommandB(commandB)
AggregateCommandHandler
eventHandler: AggregateEventHandler
aggregateRepository: AggregateRepository
handleCommandA(commandA)
handleCommandB(commandB)
AggregateEventHandler
eventRepository: EventRepository
kafkaTemplate: KafkaTemplate<String, String>
handleEventA(eventA)
handleEventB(eventB)
AggregateRepository
eventRepository: EventRepository
rehydrate(aggregateId)
EventRepository
save(event, aggregateType)
findAllForAggregate(aggregateId, aggregateType, class)

Figure 13: A generic event store class diagram.

See Figure 25 for a sequence diagram describing the process of handling commands.

On the top level, there is a REST interface for receiving commands. Commands are received in an HTTP POST request that contains the command as a JSON object in the body of the request. The REST controller class will take this request and deserialize it to a java object and pass into the command handler.

When the command handler receives a command, it takes the aggregate id included in the command and asks the aggregate repository to rehydrate the aggregate.

See Figure 14 for an activity diagram of the rehydration process. To perform the rehydration the aggregate repository queries the event repository for all events for the given aggregate id, ordered by timestamp in ascending order to get the oldest events first. The event repository performs this query on the underlying PostgreSQL database, and return the resulting events to the aggregate repository. The aggregate repository then creates an instance of a rehydration visitor and passes it an empty aggregate. The rehydration visitor is then used to apply all the events to the aggregate, by changing the state of the aggregate object. The result of this process is an aggregate object with values according to the current state of the aggregate. The command handler then validates the command it received against the aggregates state. To check if it can apply the command without breaking business rules or leaving the system in an inconsistent state. If the command passes validation, events are created to satisfy the changes requested in the command. These events are then passed on to the event handler. The event handler takes the events it receives and saves them to the event store. If successful, the events are also published to the appropriate Kafka topic.

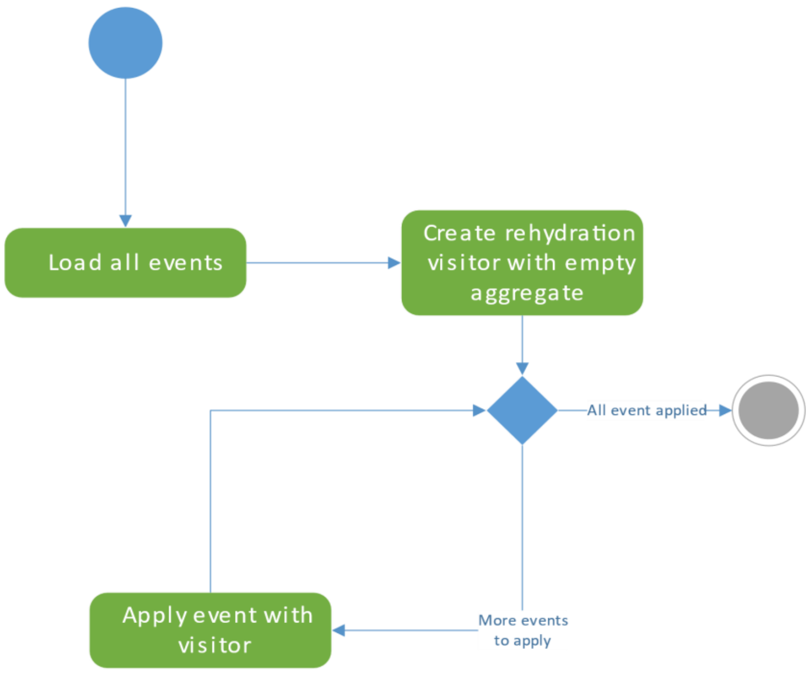
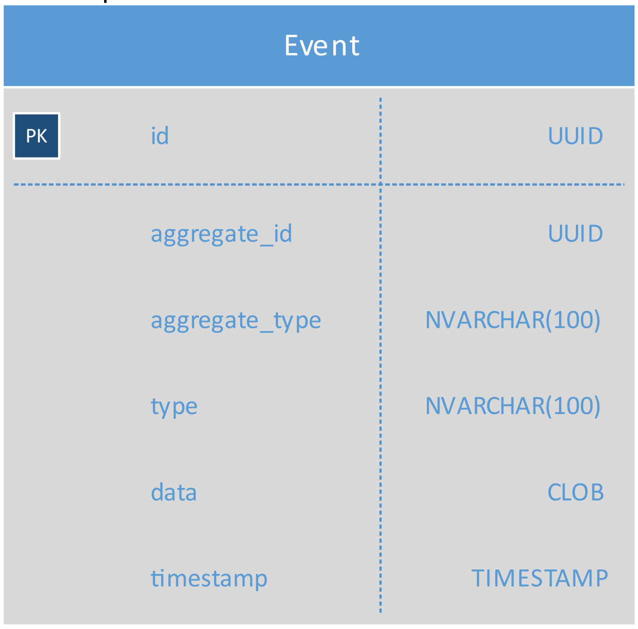


Figure 14: Activity diagram describing the rehydration process.

4.6.2 Database table

See Database configuration for the Liquibase XML configuration used to set up the database with event and snapshot tables and indexes.



Event		
PK	id	UUID
	aggregate_id	UUID
	aggregate_type	NVARCHAR(100)
	type	NVARCHAR(100)
	data	CLOB
	timestamp	TIMESTAMP

Figure 15: The event database entity.

The table used to hold the events has 6 columns:

- id
    - A UUID column that acts as an id for each event. This is the primary key of the table
- aggregate_id

  - A UUID column that indicates what aggregate the event belongs to.
- aggregate_type
  - A nvarchar column that indicate what type of aggregate the event belongs to, Ex: CUSTOMER, ORDER, PRODUCT, etc. The values in this column were string representations of values in that AggregateType java enum.
- data
  - A CLOB column that contains the event serialized into text, our system used JSON as serialization format.
- type
  - A nvarchar column that indicates what type of event is stored in the row. The value for this row was created by using the simple class name from java. Ex: someEvent.getClass().getSimpleName().
- timestamp
  - A timestamp column that indicates when the event was added to the event store, needed to order the event correctly when rehydrating an aggregate.

4.6.3 Querying the event store

There are only two read queries we need to perform towards the event store. There might be other queries that can be done for debugging and testing reasons, but these two are the only ones needed for the system to work:

1. Retrieve all events for a given aggregate id and order them by timestamp value in ascending order. To get the oldest events first.
2. Retrieve all event for a given aggregate id, after a given timestamp, and order them by timestamp in ascending order.

The first query is used to get all events for an aggregate when you need to rehydrate it. The second one is used to rehydrate an aggregate when you have a snapshot version for it. In this case it retrieves all events that occurred after the current state of the snapshot, to only get the events that have not already been applied.

We also added an index on the aggregate id column since this is the column used in the WHERE clause of the queries. This increase search performance as described in 2.4.2.3. This index is created by the create script for the database shown Database configuration.

4.6.3.1 JPA

```
@Query(value = "" +
        "SELECT e " +
        "FROM EventRecord e " +
        "WHERE e.aggregateId = :aggregateId " +
        "AND e.aggregateType = :aggregateType " +
        "ORDER BY timestamp ASC")
List<EventRecord> findAllAggregateEventsWithCustomQuery(
        @Param("aggregateId") UUID aggregateId,
        @Param("aggregateType") String aggregateType);
```

Listing 5: JPQL Query that returns all events with a specified aggregate id.

Only the first query mentioned before has been implemented in JPA. The query to retrieve all events after a timestamp was not implemented in JPA as the only event store to support snapshots had been switched to use JDBC.

```
@Entity
@Table(name = "event")
@NoArgsConstructor
@AllArgsConstructor
@Data
public class EventRecord implements Serializable {

    @Id
    private UUID id;
    @Column(name = "aggregate_id")
    private UUID aggregateId;
    @Column(name = "aggregate_type")
    private String aggregateType;
    private String data;
    private String type;
    private Timestamp timestamp;

}
```

Listing 6: EventRecord entity that maps to the event table.

The events are mapped from their EventRecord JPA entities into the Event objects by parsing the json string stored in the data field.

```
public <T extends Event> List<T> getAllForAggregate(UUID aggregateId,
AggregateType aggregateType, Class<T> clazz) {
    List<EventRecord> eventRecords =
         repository.findAllAggregateEventsWithCustomQuery(
                          aggregateId,
                          aggregateType.toString());
    List<T> events = new ArrayList<>();
    for (EventRecord eventRecord : eventRecords) {
        T event = parser.readJson(eventRecord.getData(), clazz);
        events.add(event);
    }
    return events;
```

Listing 7: Converting EventRecord entities to event.

4.6.3.2 JDBC

We started to implement queries against the event store using Springs JdbcTemplate. This has been implemented as a repository class with generic methods for querying for events. The repository class can be used for retrieving all events of an aggregate or just the events after a given timestamp, to be used when loading a snapshot.

```
public <T extends Event> List<T> findAllForAggregate(UUID aggregateId,
AggregateType aggregateType, Class<T> clazz) {
    final String sql =
            "SELECT data " +
            "FROM event " +
            "WHERE aggregate_id = ? AND aggregate_type = ? " +
            "ORDER BY timestamp ASC";
    final Object[] args = new Object[] {aggregateId, aggregateType.toString()};
    return jdbc.query(sql, args, (rs, rowNum) ->
            jsonParser.readJson(rs.getString("data"), clazz));
}
```

Listing 8: Query for loading all events for a given aggregate.

This query loads all the events for a given aggregate. The query can be used for different classes if they extend the Event base class as defined in the method signature.

```
public <T extends Event> List<T> findAllForAggregateAfter(UUID aggregateId,
AggregateType aggregateType, Class<T> clazz, Timestamp timestamp) {
    final String sql =
            "SELECT data " +
            "FROM event " +
            "WHERE aggregate_id = ? AND aggregate_type = ? AND timestamp > ? " +
            "ORDER BY timestamp ASC";
    final Object[] args = new Object[] {
        aggregateId, aggregateType.toString(), timestamp
        };
    return jdbc.query(sql, args, (rs, rowNum) ->
            jsonParser.readJson(rs.getString("data"), clazz));
}
```

Listing 9: Query for loading all events for a given aggregate, after a given timestamp.

This query can be used to load all event for a given aggregate, but only the ones after a given timestamp. This can be used along with snapshots to avoid loading unnecessary events. As the previous query, this can also be used for all type of classes if they extend the Event base class.

4.6.4 Snapshot

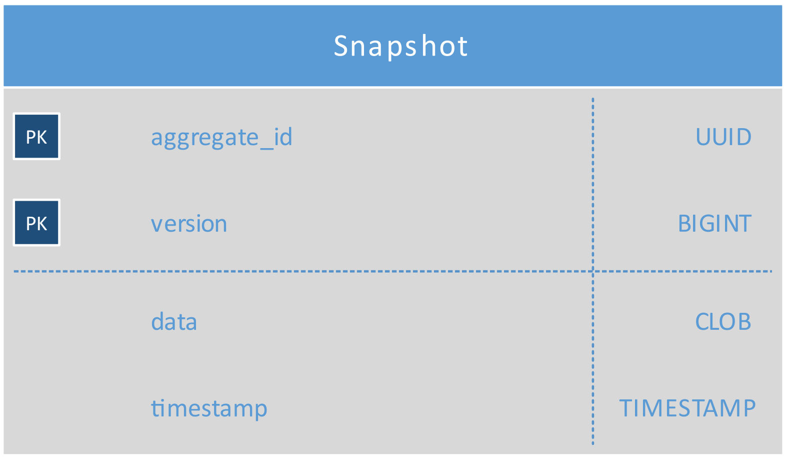


Figure 16: Snapshot database entity.

Snapshots are stored in a database table. Along with the aggregate state in JSON format we include aggregate id, a version number and a timestamp. The aggregate id and version make up a composite primary key for the table, as we do not need multiple snapshots of one aggregate at the same version.

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Snapshot {

    private UUID aggregateId;
    private Integer version;
    private String data;
    private Instant timestamp;

}
```

Listing 10: Class that describes a Snapshot.

Snapshots are performed by taking a rehydrated aggregate and serializing its state to JSON. This JSON string is stored in a Snapshot object that includes a version and timestamp value for the snapshot. The version value indicates how many events were used to create the snapshot. If the snapshot was created after applying 1000 events to an aggregate the version value would be 1000. The timestamp value is the timestamp of the last event that was applied to the snapshot.

The main reason we implemented snapshots was the StockUnit aggregate. The StockUnit aggregate keeps track of the quantity of a product in the warehouse system, this value changes for every order that requests a given stock unit. Therefore, the number of events for a StockUnit aggregate will keep increasing for every order, because of the QuanityChanged events. Because of this, we decided to implement snapshots on the StockUnit aggregate.

Snapshotting of other aggregates may also be necessary if the system had been running and building up events over a longer amount of time.

```
@Async
public CompletableFuture<Snapshot> createSnapshotForAggregate(Snapshot snapshot)
{
    LOG.info(
         "Creating snapshot for aggregate: " +
         snapshot.getAggregateId() +
         " version: " +
         snapshot.getVersion());
    repository.saveSnapshot(snapshot);
    return CompletableFuture.completedFuture(snapshot);
}
```

Listing 11: Method that will save a snapshot.

See Figure 30 for an activity diagram describing the rehydration process including snapshotting. The submitSnapshot methods take the values passed to it and creates an object of the Snapshot class. This is then passed to the createSnapshotForAggregate method in the SnapshotService class that will save the snapshot. @Async is used to tell Spring to run this method asynchronously. This is done to ensure that saving of snapshots does not block the rehydration process, which would slow down the processing of commands.

4.7 Projection

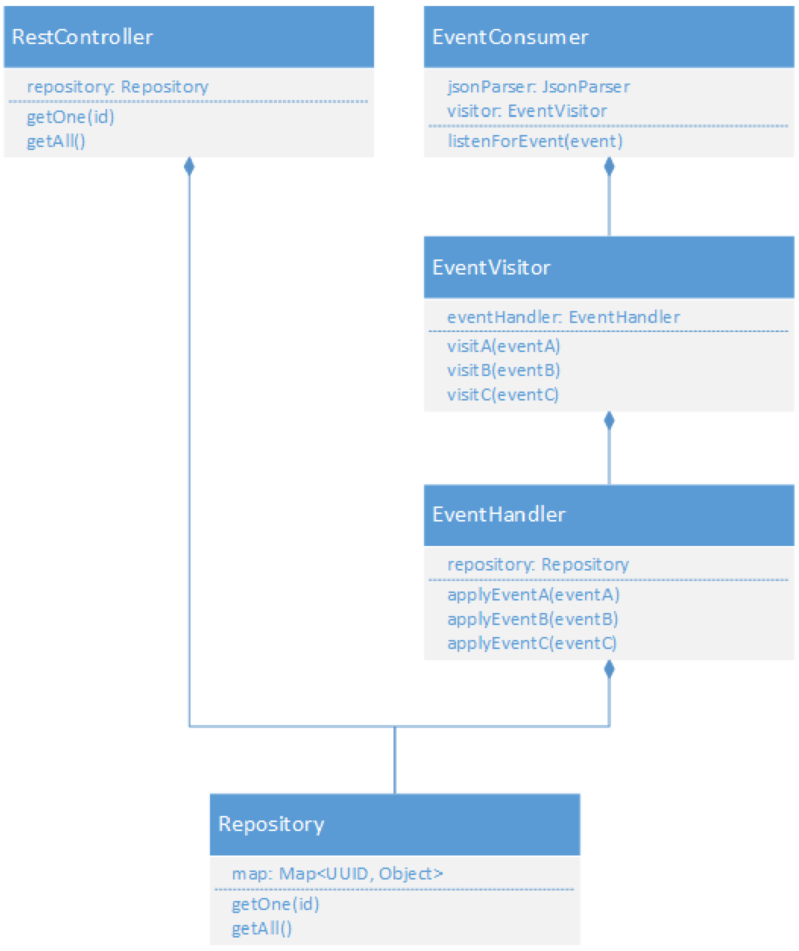


Figure 17: Generic projection application architecture.

See Figure 17 for a class diagram of the general structure of the projection applications. The general architecture for a projection is to have one or more Kafka listeners, depending on what type of events the projection is interested in.

```java
@KafkaListener(topics = "customers")
public void listenForCustomerEvent(ConsumerRecord<String, String>
consumerRecord) {
    CustomerEvent event =
         jsonParser.readJson(consumerRecord.value(), CustomerEvent.class);
    event.accept(visitor);
}
```

Listing 12: A Kafka listener consuming an event belonging to a customer aggregate.

See Figure 26 for a sequence diagram of the general event handling process in the projection applications. When the Kafka listener consumes an event, it receives the event serialized in JSON format. It takes the JSON and parses it to the appropriate event type based on what topic it is consuming from.

After the event is parsed into a java object it is passed on to a visitor that will pass the concrete type of the event to an event handler. The event handler will make the necessary changes to the projections repository based on what type the events is.

4.8 CQRS

We applied the CQRS pattern to our system. This pattern fits into an event sourced system and provides us with some benefits as mentioned in 2.2.3. One of these benefits being that we can separate the write and read models. Combining this with the microservice design of the projections we can easily scale the read models by deploying multiple instances of them. Another benefit is that we can structure the read model however we want and can decide how the read model is stored.

4.8.1 Commands

Command classes are implemented as data structures. They carry data but have no functionality on their own.

```
@Data
public class ChangeCustomerNameCommand {

    private final UUID customerId;
    private final String name;

}
```

Listing 13: Example of a command class.

A command is sent to the event store application as JSON in the body of an HTTP POST request and is received by a command controller.

```
@RequestMapping(method = POST, value = "change-name")
public void submitChangeCustomerNameCommand(
        @RequestBody ChangeCustomerNameCommand command) {
    commandHandler.handleCommand(command);
}
```

Listing 14: A REST controller method that changes the name of a customer.

This controller deserializes the JSON text into a command object and passes it to the command handler that will validate the command. If the command passes validation the appropriate events will be created and submitted to the event store, these events mutate the state according to the command.

Each command has its own URL suffix in the command REST controller. As an example, the customer controller has these URL suffixes:

- /customer/create – To create a new customer.
- /customer/change-address – To change the address of a customer.
- /customer/change-name – To change the name of a customer.

These suffixes follow the same pattern which is taking the name of the command class and writing it in lisp case also known as kebab case.

4.8.2 Queries

Queries are performed against projections using REST calls. Read calls use the GET HTTP method. The general way the projection REST controllers are set up is to have a getAll and getOne query. Some projections have queries to retrieve objects based on specific values, an example of this is to retrieve all orders with a specific status.

```
@RequestMapping(method = RequestMethod.GET)
public List<Order> getAll() {
    return orderRepository.getAll();
}
```

Listing 15: A REST controller method that returns a list of all orders.

To get all objects of a resource the getAll query is performed by making a GET call directly to the resource.

```
@RequestMapping(value = "{id}", method = RequestMethod.GET)
public ResponseEntity getOne(@PathVariable("id") UUID id) {
    Optional<Order> orderOptional = orderRepository.getOne(id);
    if (orderOptional.isPresent()) {
        return ResponseEntity.ok(orderOptional.get());
    } else {
        return ResponseEntity.notFound().build();
    }
}
```

Listing 16: A REST controller method that returns a specific instance of a resource.

To retrieve a single object of a resource the getOne call can be used. This is done by performing a GET call on the resource and including the id of the resource as a path variable. If no object is found for the given id a response with code 404 will be returned. The method that handles this is shown in Listing 16.

```
@RequestMapping(params = "status", method = RequestMethod.GET)
public List<Order> getAllByStatus(@RequestParam("status") OrderStatus status) {
    return orderRepository
            .getAll()
            .stream()
            .filter(order -> order.getStatus().equals(status))
            .collect(Collectors.toList());
}
```

Listing 17: A REST controller method that returns all orders with a specific status.

To retrieve all orders of a specific status, the desired status can be included as a query parameter. When the request is received for orders of a specific status the Java 8 stream API is used to filter the orders.

4.9 Configuration

application.properties file:

```
logging.file=logs/logfile.log

simulator.customerProjectionUrl=http://158.38.101.50:8082/customers
simulator.customerCommandUrl=http://158.38.101.50:8080/customers
simulator.productProjectionUrl=http://158.38.101.50:8084/products
simulator.productCommandUrl=http://158.38.101.50:8080/products
simulator.orderProjectionUrl=http://158.38.101.50:8081/orders
simulator.orderCommandUrl=http://158.38.101.50:8080/orders
```

application-dev.properties:

```
simulator.customerProjectionUrl=http://localhost:8082/customers
simulator.customerCommandUrl=http://localhost:8080/customers
simulator.productProjectionUrl=http://localhost:8084/products
simulator.productCommandUrl=http://localhost:8080/products
simulator.orderProjectionUrl=http://localhost:8081/orders
simulator.orderCommandUrl=http://localhost:8080/orders
```

Listing 18: Example configurations from the order command simulator application.

To configure the system for different environments we used Spring profiles. We created a application.properties and application-dev.properties and configured the dev properties file with properties to use for running the applications locally on our own computer. These properties are mostly IP-addresses for different command and projection APIs and the IP-address of the Kafka host.

This way all API URLs will point to localhost when the application is run in dev profile. This is easy to set up in IntelliJ's run configurations. While running the application without dev profile would point all API URL to the NTNU server that was running the applications. The properties for the different files can be seen in Listing 18.

```
@Configuration
@ConfigurationProperties(prefix = "simulator")
@Getter
@Setter
public class PropertyValues {

    private String customerCommandUrl;
    private String customerProjectionUrl;
    private String productCommandUrl;
    private String productProjectionUrl;
    private String orderCommandUrl;
    private String orderProjectionUrl;

}
```

Listing 19: PropertyValues class used to load custom properties.

To load custom property values that are not configured by Spring we created a class called PropertyValues annotated with the @ConfigurationProperties annotation. In the @ConfigurationProperties we have defined the prefix our custom properties. The properties being loaded can be seen in Listing 18. By using the prefix and the field names in the PropertyValues class, Spring will be able to find and load the values from application.properties. As with other Spring properties, this class would load the values from the application.properties or application-dev.properties depending on what configuration profile the application is running in.

4.10 Simulator

The purpose of the simulator applications is to generate input to the other applications. In the case of event sourcing with CQRS, this involves generating commands to change the state of the system and sending these to the event store. The way we have designed the simulator applications they are generating commands at a set interval and sends them with REST calls to the command controller of the event store it is simulating input for.

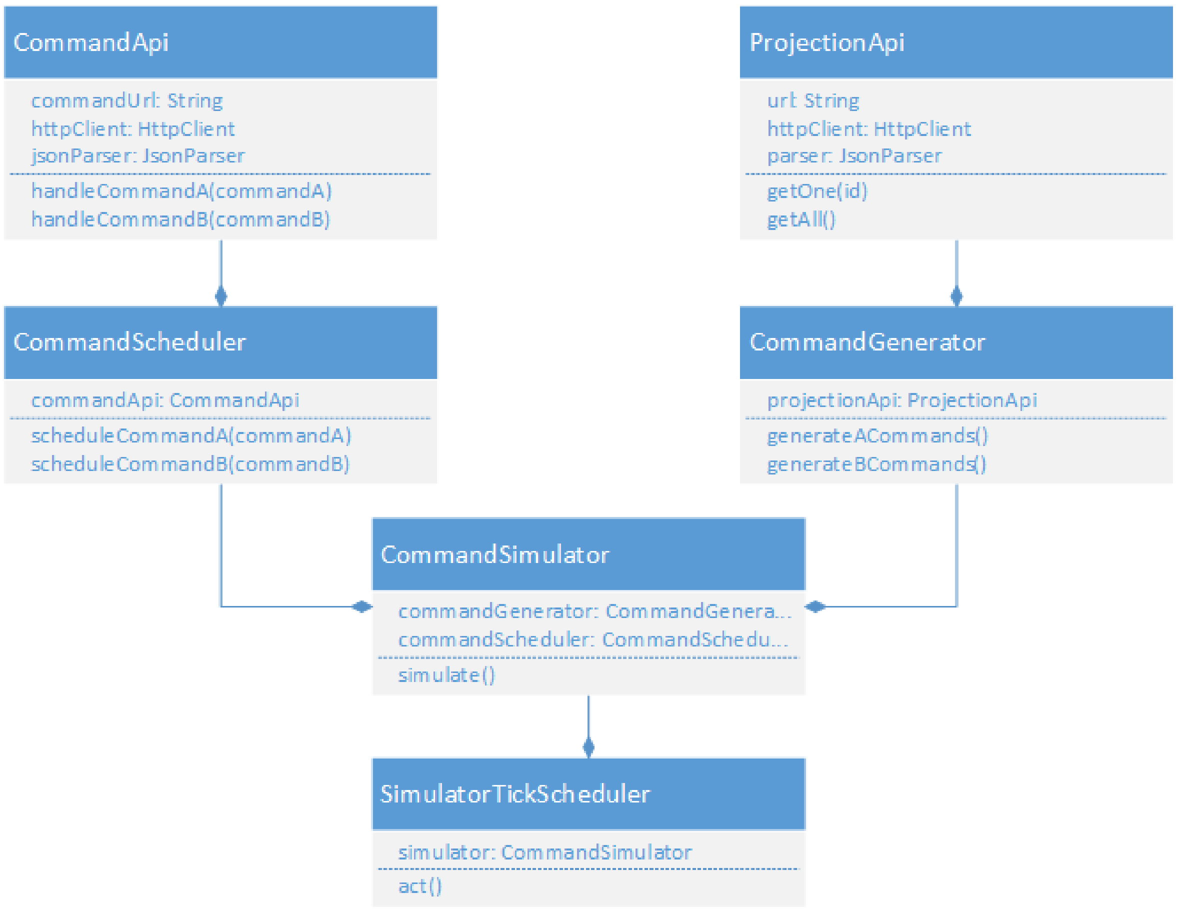


Figure 18: The general structure of the simulator application.

See Figure 27 and Figure 28 in the appendix for sequence diagrams of the command generation process. The SimulatorTickScheduler is the class that triggers the command generation. It will call on the act method in the CommandSimulator at a set interval. This is implemented by using the Spring @Scheduled annotation and defining a value for the scheduling rate, which is the time between each call to the act method.

The CommandSimulator class is the class that ties the application together. When it is called on by the SimulatorTickScheduler it will ask the CommandGenerator class to generate a set of commands that can be submitted. When the CommandGenerator has generated the commands, the CommandSimulator will pass them to the CommandScheduler with a randomly generated timestamp for when the command should be submitted. This is done to spread the commands out over a period instead of sending a lot of commands at once and then nothing until the next simulator tick. The commands will be uniformly distributed over a period from current time to current time plus the scheduling rate. This should result in a steady stream of commands.

The CommandGenerator uses the ProjectionApi to query a projection for the current state of the system. The commands will be generated based on data returned by the projection.

The ProjectionApi is a class from the commons library that acts as an abstraction over the HTTP calls that are being performed.

The CommandScheduler will schedule the commands received from the CommandSimulator by using a ThreadPoolTaskScheduler that allows to schedule Runnable objects to be executed at a specific time. The commands are wrapped by a Task class that will be executed by the thread pool.

The CommandApi is used to submit the commands to a command controller. Like the ProjectionApi class, the CommandApi class acts as an abstraction over the HTTP calls being made.

4.10.1 Random timings of scheduled commands

When the commands are scheduled they are scheduled at a random time between the start of the command generation and the start of the next command generation.

```
currentTime = Instant.now();
endTime = currentTime.plus(Config.simulatorBatchLength, ChronoUnit.MILLIS);
```

Listing 20: How the values used to genereate a random time is generated.

The currentTime and endTime Instant values are fields in the CommandSimulator class. These values are used to generate the random value that is used for scheduling the commands. The simulatorBatchLength value is a constant value defined in the Config class. This value represents the time between each tick of the SimulatorTickSchduler in milliseconds.

```
//In CommandSimulator class
private Instant getRandomInstantInBatch() {
    return TimeUtil.getRandomInstantBetween(currentTime, endTime);
}

//In TimeUtil class
public static Instant getRandomInstantBetween(Instant a, Instant b) {
    long aMillis = a.toEpochMilli();
    long bMillis = b.toEpochMilli();

    return Instant.ofEpochMilli(random.nextLong(aMillis, bMillis));
}
```

Listing 21: Methods used for generating a random time in milliseconds.

The getRandomInstantInBatch method is used in the CommandSimulator class to generate a random Instant value between the current values in the currentTime and endTime field of the class. This is done by using the getRandomInstantBetween method In the TimeUtil class. It is used to get a random instant value uniformly distributed between the given beginning and end Instant values.

4.10.2 Random timings in pick simulation

```
private double getGaussDistributedPickTime() {
    return random.nextGaussian() *
            Config.PICK_TIME_STDEV + Config.PICK_TIME_MEAN;
}
```

Listing 22: Method used to generate a random normally distributed value.

The time it takes for the pick command simulator to complete one pick line is randomly generated using a normal distribution.

The formula shown in Listing 22 is used to generate the random value. It uses javas ThreadLocalRandom class to generate a random value with a standard normal distribution, this value is then scaled by multiplying it with a defined standard deviation and adding a defined mean. This produces values distributed in a normal distribution with the defined values as standard deviation and mean. The mean and standard deviation values are defined in the Config class as static constants (public static final).

4.10.3 Address register

A simple helper class was created to be able to use proper addresses in commands that set addresses in customer and order aggregates.

At startup of the simulator application, the @PostConstruct annotation will be detected by Spring, and the method annotated will be run. This allows the AddressRegister to load the addresses from a CSV file at startup. It will loop through all lines of the file and store the address, postal code and postal code location in a list as an instance of the AddressEntry class.

Later the CommandGenerator will use the AddressRegister when it needs an address for a command. This is done by calling the getRandomAddress() method that returns a random AddressEntry object from the list that was created at startup.

4.11 Event store applications

In this chapter we will describe some of the applications in more details and will describe how the aggregates react to changes in other aggregates.

The general functionality of the event store applications where described in 4.1.1 and they follow the event store structure described in 4.1.2. The event store applications handle one or more aggregates and for each aggregate, it manages the event store repeats the structure, except for the event repository which is generic and can handle multiple event types.

4.11.1 Order system

The order system manages the customer, product and order aggregates. It reacts to events produced by the warehouse and invoice systems.

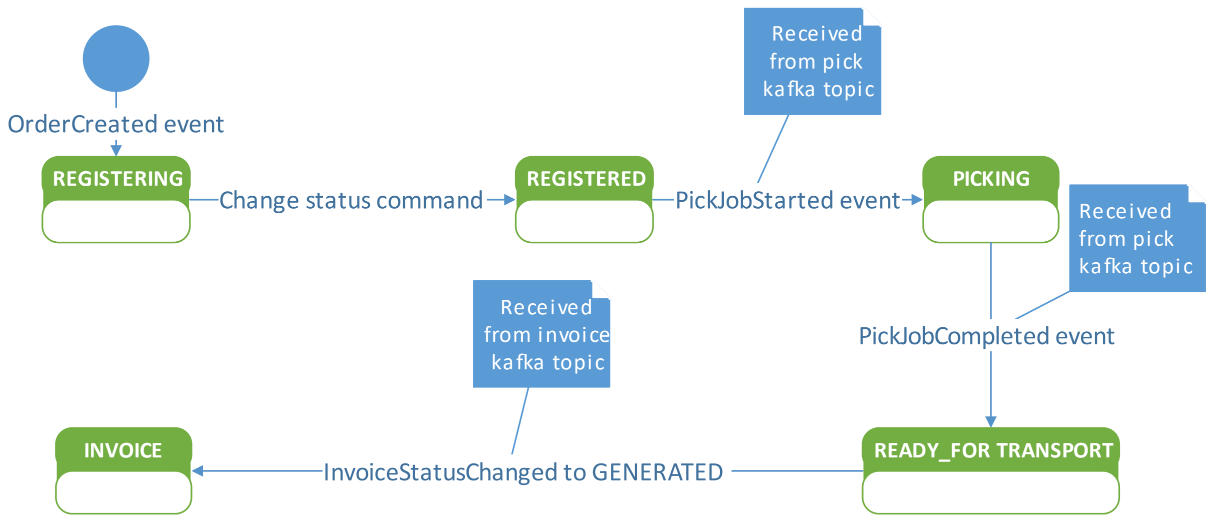


Figure 19: The states of the order aggregate.

Figure 19 shows the different states of the order aggregate and how it is affected by the events produced by other aggregates. The order is created when an OrderCreated event is created. During the REGISTERING state, it is possible to add order lines to the order. The order is moved to the REGISTERED state by a change status command. When it reaches the REGISTERED state, the order will be waiting for the warehouse system to publish a pick job started event. This will cause the order to move into the PICKING state, which signals that picking for the order is in progress.

In the current implementation of the system, the product and customer aggregates will not react to any changes in other aggregates.

4.11.2 Warehouse system

The warehouse system manages the stock unit and pick job aggregates. It reacts to events produced by the order system.

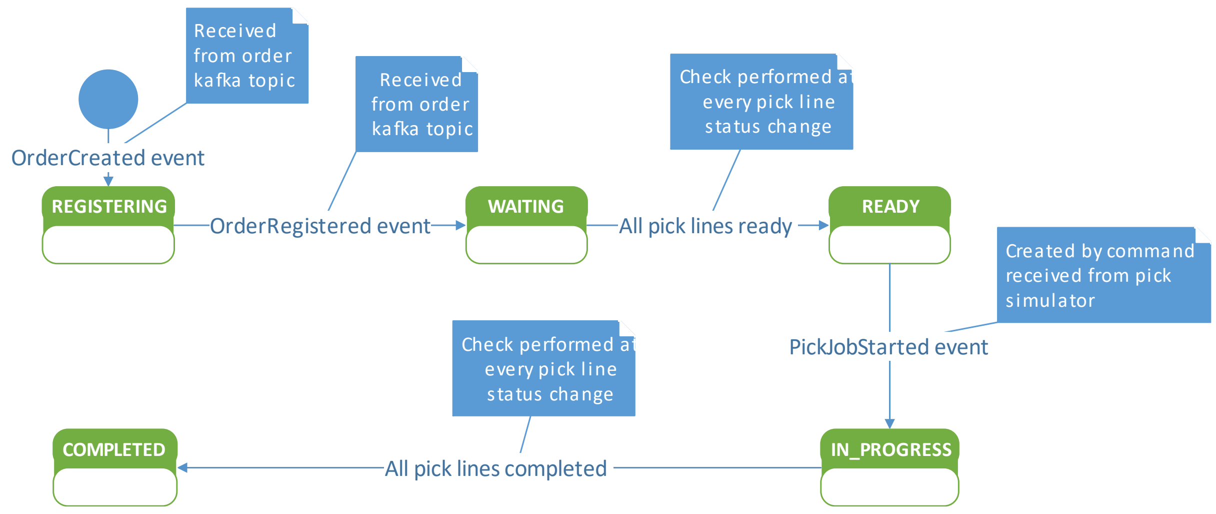

Received from order kafka topic
Received from order kafka topic
Check performed at every pick line status change
OrderCreated event
REGISTERING
OrderRegistered event
WAITING
All pick lines ready
READY
Created by command received from pick simulator
Check performed at every pick line status change
PickJobStarted event
COMPLETED
All pick lines completed
IN_PROGRESS

Figure 20: The states of the pick job aggregate.

As seen in Figure 20 the pick jobs are created when the orders are created. The pick job will get pick lines corresponding to the order lines of the order it belongs to.

When the warehouse system receives an OrderRegistered event from the order system, it will move the pick job into the waiting state. In this state, it will be waiting for all the pick lines to reach the ready state, which indicates that they are ready for picking.

When all pick lines belonging to the pick job reach the ready state the pick job will be moved to the ready state. In the ready state, the pick job is waiting for the pick command simulator to start the pick job.

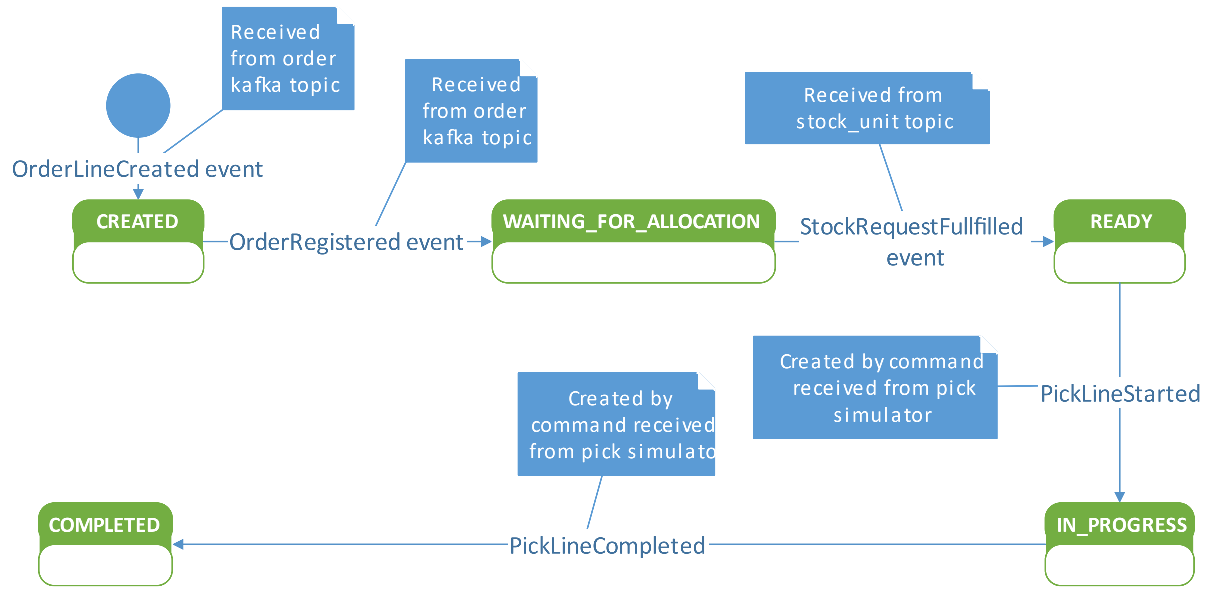

Received from order kafka topic
Received from order kafka topic
Received from stock_unit topic
OrderLineCreated event
CREATED
OrderRegistered event
WAITING_FOR_ALLOCATION
StockRequestFullfilled event
READY
Created by command received from pick simulator
PickLineStarted
Created by command received from pick simulato
COMPLETED
PickLineCompleted
IN_PROGRESS

Figure 21: The states of pick lines in the pick job aggregate.

When the pick job has been started, it will be moved to the in-progress state. When all pick lines have been completed it will be moved to the completed state.

Pick lines are created when order lines as created for the corresponding order as seen in Figure 21. When the order corresponding to the pick job is moved to the registered state the pick lines are moved to the waiting for allocation state.

In this state, the pick lines are waiting for the pick allocations service to allocate the pick lines for picking. When the pick allocation service allocates a line, the lines are moved to the ready state.

In the ready state, the pick line is waiting for the pick command simulator to start picking that line by sending a command that created a pick line started event.

When the pick command simulator starts picking a line it will be moved to the in-progress state.

In the in-progress state, the pick lines are waiting for the pick command simulator to finish the pick line by sending a command that creates a pick line completed event. When this event is created the pick line is moved to the completed state.

4.11.3 Invoice system

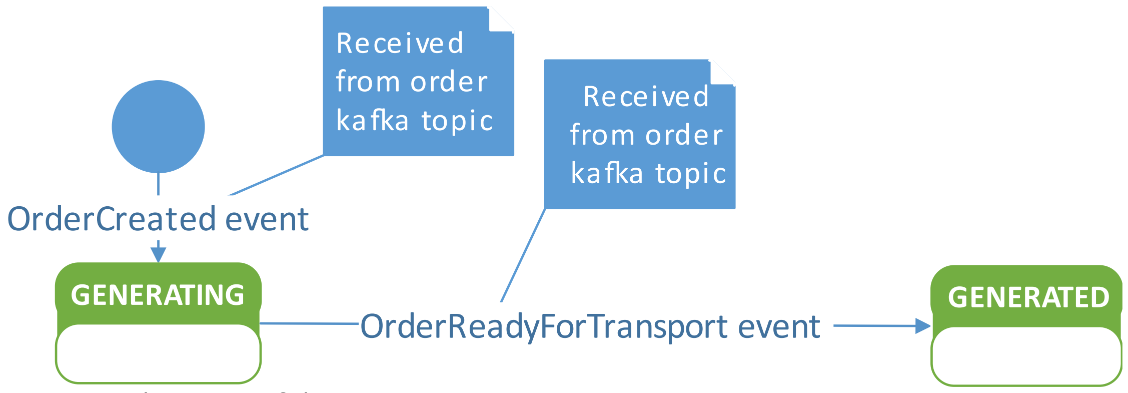

Received from order kafka topic
Received from order kafka topic
OrderCreated event
GENERATING
OrderReadyForTransport event
GENERATED

Figure 22: The states of the invoice aggregate.

The invoice aggregate is created when an order is created. The invoice will receive a new invoice line every time an order line created event is received for the order that corresponds to the invoice. When the invoice system receives an order ready for transport event the invoice will change state to generated.

4.12 Projection applications

We have developed a total of 6 projections, all of these except one was implemented using a hash map to store data. The only projections to not use a hash map was the relational database projection.

4.12.1 Hash map projections

The hash map projections were implemented to be volatile and they are required to build up their state from the Kafka event stream at every startup. This was done by setting a random UUID value as the Kafka group id. The hash map projections are:

- Customer projection
- Product projection
- Order projection
- Invoice projection
- WMS projection

The Warehouse Management System (WMS) projection has read models for stock units and pick jobs.

4.12.2 Relational database projection

The relational database projection was implemented with an H2 in-memory database. This means that it needs to build its state at every startup. But it should be a trivial change to switch it over to a PostgreSQL database for non-volatile storage.

The current implementation of the RDB projection builds up a read model of customers, products and orders which can be queried with SQL queries.

Because the customer, product, and order events are stored in different topics in Kafka the events may be received at different paces, depending on the number of event in each topic. Which could lead to a situation where a product referenced by an order line does not exist yet. When the create order line event is received.

```
private Product getProduct(UUID id) {
    if (productRepository.exists(id)) {
        return productRepository.getOne(id);
    } else {
        Product product = new Product();
        product.setId(id);
        return productRepository.save(product);
    }
}
```

Listing 23: Eager loading of products.

To solve this, we implemented functionality to eagerly create product aggregates as seen in Listing 23. This way the product is created if it does not exist already, it will contain only an id at this point. There will be a period where the product is an empty object with only the id. But eventually when the create product event belonging to it is received from the product Kafka topic it will be updated with the proper values.

4.13 Simulator applications

The general purpose of the simulator applications was described in 4.1.1. And some more general functionality as described in 4.10. We have developed 4 simulators for our system and in this chapter, we will describe more specifically how they generate the commands they use to simulate user interaction.

4.13.1 Customer command simulator

The customer command simulator is responsible for creating and populating the values in customer aggregates.

It runs in batches with a defined length. For each batch a random number will be generated, this number will be used to decide how many new customers to create in the current batch.

The simulator will also query the customer projection for information about the existing customers. Based on this information the simulator will generate commands to set the name and address of customers.

For each customer, without a name, a change name command will be created.

For each customer, without an address, a change address commands will be created. The address will be chosen randomly from the address register.

4.13.2 Product command simulator

The current implementation of the product command simulator is quite simple. It creates a large number of products in a batch, this is only done once for every run of the simulator application.

4.13.3 Order command simulator

The order command simulator simulates orders being created in the system. This is done by querying the product, customer and order projection for information. Based on the information returned by the projection the commands are generated as follows.

Every batch the simulator runs a random number is generated to decide how many new orders to create this batch. Similar to the customer command simulator.

For each order without an address defined a change address command is generated, the address used is picked randomly from the address register.

For each order without a customer, a change customer command is generated. The customer assigned to the order is randomly picked from the customer information returned earlier by the customer projection.

For each order that does not have any order lines associated with it a random number of create order line commands will be created. The quantity for the order line is generated randomly and the product is chosen randomly from the information retrieved from the product projection earlier.

4.13.4 Pick command simulator

The pick command simulator simulates having several pickers working on the pick jobs created in the warehouse system.

The simulator has a defined number of pickers. This number decides how many pick jobs can be worked on at once. At start up the simulator generates a set of random UUIDs that acts as the picker id for the simulated pickers.

The simulator will query the pick job projection for all pick jobs. It will then filter this result to get only the pick jobs in the in-progress state. It will then loop through this list to find out what pickers are currently busy.

When it has found out what pickers are free to take on a pick job. The simulator will filter the list of pick jobs to get all pick jobs in the ready state.

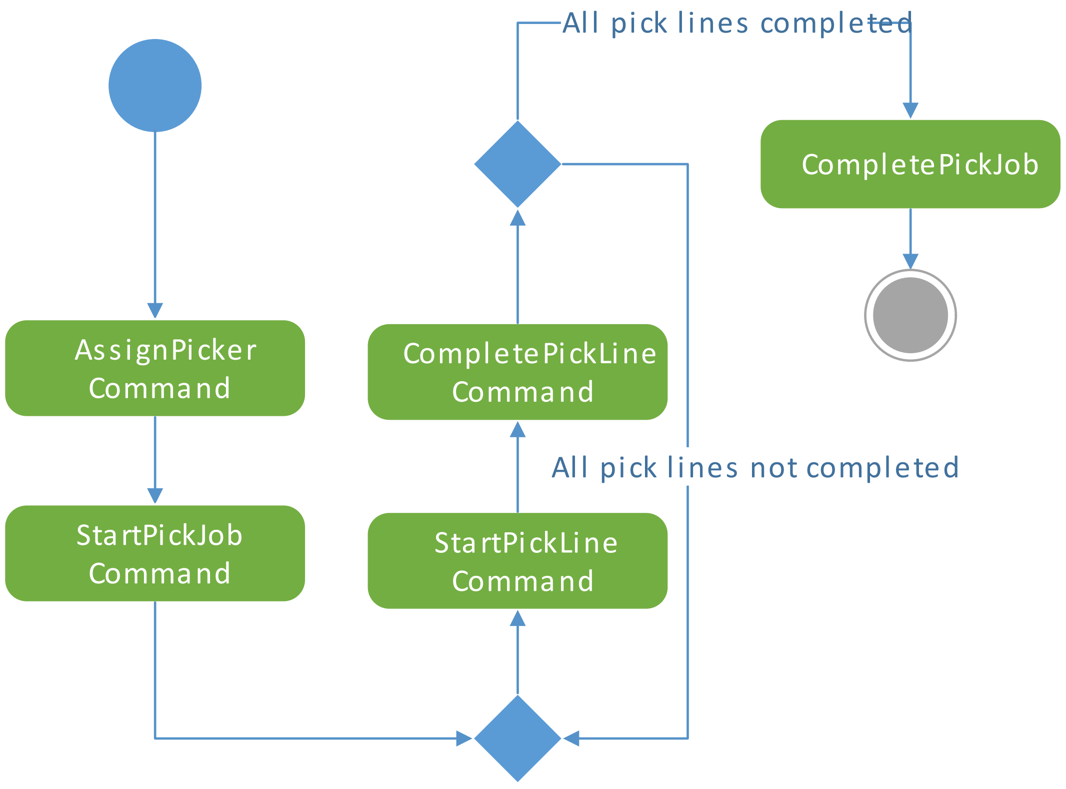


Figure 23: Pick simulator picking activity diagram.

The simulator will then go through the pick jobs in the ready state and generate all commands necessary for the pickers to complete the pick job they are assigned to.

See Figure 23 for an activity diagram of the picking process that the simulator follows. This includes a command to assign the pick job to the picker. Then a command to set the pick job to an in-progress state. After this, the simulator will send out commands to start the pick lines associated with the pick job. Between the command to start a pick line and the command that completes it, there is a normally distributed random delay as described in 4.10.2.

After all the pick lines are completed a complete pick job command will be submitted to set the pick job to the completed state.

4.14 Pick allocation service

Because of the aggregate boundaries and microservice boundaries are stopping us from making changes to the pick job aggregate and the stock unit aggregate we created the pick allocation service. Its intention is to move the pick lines to a ready state if there is a large enough quantity of product to pick. But in the current implementation, the quantity in stock is not properly implemented so the pick allocation service currently just moves pick lines into the ready state.

The stock requests in the stock unit aggregate were meant to provide a consistent way to change the quantity of product in stock and allow picking to start with the approved requests. As seen in Figure 21 the pick lines will react to events generated by approving stock requests. This is done by querying the stock allocation projection to check for all stock requests. It will then go through the stock requests and send commands to the warehouse system to approve the stock requests.

4.15 System diagram

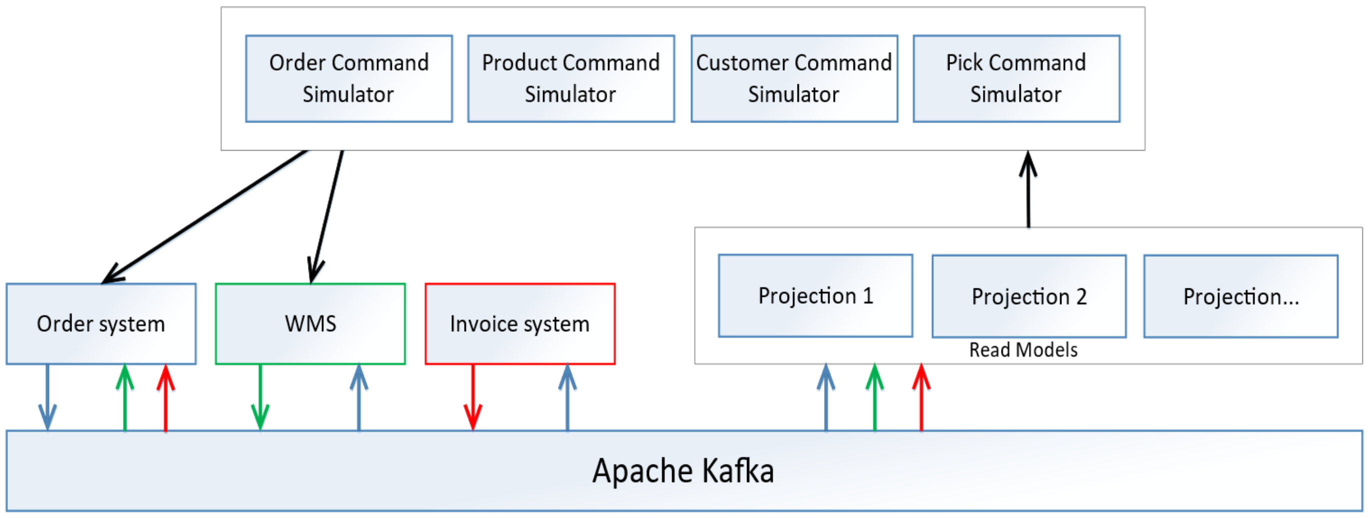


Figure 24: System diagram.

This diagram shows a high-level overview of the system. The simulators query projections for information. Based on this information, commands are generated to change the state of the system. These commands are sent to the event store applications who validate and apply them by creating events. The events are stored in an event store and publishes the events to topics on Apache Kafka. The events are then picked up by other event store applications or the projections.

5 DISCUSSION

In this section we are going to reflect on the quality of the sources we have found, design choices we have made, problems with the current solution that we would like to solve, and what improvements we would have implemented if we had more time.

5.1 Sources

In the early stages of the thesis we did a literature review. The goal of the literature review was to find as many reliable sources as possible to further support our understanding of event sourcing. Among the sources, we found there were books written by professionals that use event sourcing in their own systems, as well as blog posts and YouTube videos made by those who were some of the first to describe the concept.

Along the way, we changed our idea of how we should develop the system several times. We found in our research that big companies like Netflix use event sourcing to solve the problems they had with scalability in some of their services.(74) When we saw how they had implemented event sourcing and the thought process behind getting a good result, that caused us to change our perception of how we should develop our own system. So, we could then take the knowledge we previously had and combine it with new ideas to get the best result possible.

We also used several bachelors- and masters theses, blog-posts from credible sources such as Greg Young and Martin Fowler, and multiple journal articles to gather as much information as possible about the subject. By assimilating this material and comparing it to the limited number of books we could find, we steadily gained a good understanding of how we should implement the solution.

In the field of software engineering, there is a lack of standards to follow in general and especially when you are working with technologies that are relatively new. In our case that meant we had to take this concept of an event sourcing pattern and try to use it in our own version of an ordering system. We think we did a good job of using existing technologies to our advantage and in using the right architecture and design patterns.

5.2 Technological choices

5.2.1 Programming language (Java)

We chose to write the software in the Java programming language. That was because it is the language we have the most experience with, it is being used by DRIW and because of all the existing Java specific libraries. Additionally, it can be used with the Spring framework and Spring Boot which provides several features and modules that make certain things faster and easier.

5.2.2 PostgreSQL

We chose to use the PostgreSQL database because we were already familiar with relational databases and because it has proven to perform better than the graph database Neo4j at reading and writing events. (38)

5.2.3 Kafka

The reason we chose Kafka to handle event messaging was that it seemed like a good fit for our project and because event sourcing is listed as a popular use case for Kafka. (75) We also found a benchmarking of Kafka that showed that it performs very well. (76)

5.3 Project

In our thesis, we tried to follow the agile software development method SCRUM as much as possible throughout the project. The problems we encountered with SCRUM were that we spent a lot of time performing a literature review, while these software development methods promote early delivery and continual improvement to the systems.

So, we ended up with splitting the project in two where the first phase was mostly involved with preparing the thesis by doing the literature review. While the second phase mostly involved developing the software.

5.4 Reflection

5.4.1 Kafka event store

To start out we attempted to implement the event store using Kafka. This worked to some degree, but we were missing some functionality. Since Kafka works as a stream of messages it is difficult to query it for specific events. This is required to be able to validate new commands as you need to be able to retrieve all events for a given aggregate to perform the rehydration of the aggregate. To solve this, we ended up with our simple implementation of an event store using a Postgres database. We still use Kafka as a messaging backbone to transport events produced to other application that are interested.

5.4.2 How small to make a microservice?

If we had fully followed the microservice architecture rule of single responsibility we should have split up some of the applications into multiple applications.

We have one application that acts as an event store for the order, customer and product aggregates. These should have been split into 3 applications. They ended up as one application because it was the first application we developed early in the project and we had not considered using microservices yet. We didn’t prioritize splitting the applications, but it is something we would have done if we had more time.

5.4.3 JPA or JdbcTemplate

We started out using Spring Data JPA repositories for connecting to the Postgres SQL server. This was done because it was quick and simple and allowed us to quickly start developing the system without spending a lot of time on our database communication. We later started switching over to Spring Data JDBC template as we felt using JPA for our simple database communication was unnecessary.

Having JPA perform a mapping from row to object just to have the object discarded immediately seemed unnecessary. With Spring Data JDBC we could rather perform the JSON parsing to respective event object immediately from the result set.

5.4.4 Alternative event implementation (Polymorphism over visitor)

The current implementation of events requires a visitor to be able to deal with the concrete type of events when they are deserialized from the event store or received from a Kafka topic. The approach to this using a visitor makes it somewhat difficult to add new events as you must update the implementation of all visitors of the new events aggregate type.

This could have been done differently by having an apply to aggregate abstract method on the events. That every event override. For a customer event this could look like: applyToAggregate(customerAggregate). This way we would only need to create the new event class and override the abstract method.

This approach would, however, make it difficult to implement the event handling differently in all the different applications as they may want to deal with the event in different ways. The approach we currently use with the visitor pattern allows each application to create their own event handling by creating a new implementation of the visitor.

We think our current implementation works better that the polymorphic way, but there is probably room for improvement.

5.4.5 Evolving data and schema

In the event implementation, we used Jackson to include the full class name in the JSON object, as shown in 4.4. This should, in theory, enable us to use the multiple version techniques described in 2.4.4.3, To evolve the schema. It should be possible to create a new version of the event in a different package and Jackson should be able to tell them apart at deserialization. It does, however, require changes to the event handler and visitor interfaces or to create a new version of each to include the new event version.

So, it should be doable in theory, but it is might be easier to use something like an upcaster as described in 2.4.4.3.

5.4.6 Lots of different opinions about ES and CQRS

One of the problems we had during our study/research of event sourcing was that there is no defined standard for these kinds of systems. The community around event sourcing and CQRS are also much smaller than the CRUD community, so it can be more difficult to find good solutions to problems encountered. There also seem to be many opinions about how to implement these kinds of systems, some can be conflicting and make it difficult to figure out what the correct solution is.

5.4.7 Snapshot interval

We wanted to do some more analysis on what the interval between snapshots should be. We were planning to run some tests to find a good value for this, but we did not have time to do this. Currently, we have just set the interval value to 1000, without having any specific reason to believe this is an optimal value.

5.4.8 Simple simulator implementations

The current implementations of the simulators are quite simple in the way the query the projections and create commands based on this and schedule these commands to run over a period until the next command generation batch. It would be interesting to have some more complex and random behavior, but we found the current solution to work well to the test the system and see the event-driven parts working. It also recovers well from restarting the simulators in the middle of batches.

5.5 Problems

Some problems with the current solution that we would fix if we had more time.

5.5.1 Two generals problem (event store and Kafka)

There is a problem with our implementation of the event handlers that take care of saving new events to the event store and publishing them to Kafka.

If the event is saved to the event store, but the Kafka publish fails it would leave the Kafka messages inconsistent compared to the event store. If we turn the actions around and try to publish to Kafka first and then save to the event store, we end up with the same problem. If the Kafka commit is performed successfully and saving to the event store fails, they will also be inconsistent. Either both actions need to be performed or none of them. This requires something like a two-phase commit.

A solution this could be to change the implementation a bit. We could have the event handler only take care of saving to the event store, this way its either successful or fails without leaving inconsistencies. To take care of publishing to Kafka we could set up a scheduled task that runs with short intervals and publishes any new event in the event store to Kafka. This could be done by including a bit or Boolean value on each event in the event store that tells you if it has been published successfully, or the task could keep track of the timestamp of the last successfully published event and use that to find the new events to publish. This way if the Kafka publishing fails it can just retry it later.

5.5.2 Synchronization and race conditions

There are currently some race conditions in the event store applications that have not been addressed. Currently, the commands are received on the rest controllers which Spring will multithread. This could cause multiple commands for the same aggregate to reach the event store close to each other. Depending on the time it takes to reach the event store they could be processed at the same time. This could cause inconsistencies in the event store if they both pass validation without reading the changes the other command makes.

To solve this, we could lock the event store based on the aggregate id of the commands. This would still allow multiple different aggregates to be updated at the same time, but if multiple commands for one aggregate is received they would have to wait for the command before them to be fully applied.

5.5.3 Get all rest calls

We had some problems with the get all REST requests after the system had been running for a while. They would load so much data that some of the simulator applications did not have enough memory to load and process all of it. This is something we could have solved by including pagination functionality to the REST requests. This is something we did not have enough time to implement but should be implemented if we continued developing the system.

5.5.4 Event duplication

Kafka provides at least once delivery. Therefore, we could end up with duplicate events in the Kafka stream. This should be taken care of to avoid side effects.

It could be solved by implementing some deduplication functionality. This could be done by keeping the ids of all events received events in a set and checking for each event received if it has already been received.

5.5.5 Product and stock unit primary key

The current implementation creates stock unit instances when the warehouse system receives a product created event. Currently, the id of the stock unit is set to the same as the created product. This creates a tight coupling between the two aggregate types that should probably be avoided.

This was implemented this way at first because we handled the event-driven parts of the systems in the event store which can only be searched by primary key. So we made the stock unit and product ids the same to be able to find the stock unit instance when a product event was received. We should have done the event-driven processing on a higher level and used the projections to search for the stock unit, and instead give the stock unit its own unique id. The product id could then be added as another value in the aggregate.

5.6 Improvements

The improvement that could have been made if we had more time. The solution works without these, but some of these could help increase performance or speed up development if they were fixed.

5.6.1 Split up applications

Currently, we are not completely following the microservice principle of single responsibility. Examples of this are the order system event store and the warehouse system event store. These event store applications handle multiple aggregates each. The order system handles order, customer and product. The warehouse system handles pick job and stock unit aggregates. If we were to follow these principles fully these two applications should have been split into 5 applications focusing on one aggregate each.

The reason why it ended up like this is that we started developing the system before we had fully grasped all the theory parts. After we learned about the single responsibility principle we considered splitting the applications but decided against it because of time constraints and because we thought that splitting these small applications further would just create additional overhead in our case.

If we were to continue development of the system we would probably consider this if the applications grew larger or for reasons like scaling, deployment, and technology used for the implementation as mentioned in 2.1.2.

5.6.2 Creating a more generic solution

The event store applications generally follow the same architecture. There may be some more generic implementation for these applications so that it would be easier to add a new event store to the system. We didn't prioritize this for our solution. It would have little to no impact on how the system works but would simplify the process of expanding it. If we were to continue expanding the system, it may have been something worth spending some time on figuring out.

5.6.3 Business logic and validation

Improving business logic was one of the things we wanted to do but couldn't because of the time constraints. Currently, the business logic of the system is quite simple and mostly serves to test out the event sourcing and event-driven nature of the system. We decided that focusing on the architecture and implementation of the system was more important than improving the business logic.

5.6.4 Production configuration profile

Currently, the default Spring configuration profile is the one we use for production settings as shown in 4.9. This was mostly for simplicity when running the applications on the auto deploy virtual machines as we could just run the applications without having to set any profiles. When running the applications in IntelliJ IDEA we set the dev profile to be used in the IDE, but in any proper system, we should not run the applications in a production environment as default. This was just for simplicity during development and we would change this if we ever were to get this up running as a proper system.

5.6.5 In memory aggregate cache

One improvement we could make to increase the performance of aggregate rehydration would be to store rehydrated aggregates in a cache in memory. This way if an aggregate is updated often it would not be necessary to load all events for it and replay them to get the current state of the aggregate for every update. Instead, we could have it cached in memory after the first time it's retrieved.

Depending on the number of aggregates it may be necessary to have some algorithm to decide which aggregates to keep in memory. For this, we may be able to use something like the LRU cache replacement algorithm.

5.6.6 Async commands

In the current implementation of the system, the command REST controller is multithreaded by Spring so multiple commands can be processed at the same time, but each command request is handled synchronously.

To improve the throughput of the command REST controller we could change the command handling to be handled asynchronously.

5.6.7 Ensuring required values have been set

Currently, our implementation has some cases where a no argument constructor is used to create objects, and then setters are used to set all the values. A couple of times during development this resulted in required values not being set. The reason for using the no argument constructors and setters were that it looks cleaner in code that the full constructor.

We think a better solution to this is to use the builder pattern described in 2.2.4.3. This would work similar to setting the values with the setters, but when building the object, we could check if the required values have been set and throw an exception if anything is missing or have illegal values.

5.6.8 Add façade in front of AggregateRepository and EventHandler

As seen in Figure 13 the aggregate command handler is aware of both the event handler used for saving/writing events and the repository used to retrieve/read events.

We could have abstracted away from the fact that there is one class dealing with saving and one class dealing with reading events by using the façade pattern described in 2.2.4.2 and placing a façade in front of these. This would help decouple the classes and give the impression that there is one class dealing with both.

5.6.9 Store events "forever" in Kafka

In the current implementation of the system, the events are saved in the event store and published to Kafka. The events published to Kafka are stored "forever", because the events are only available to other applications through Kafka. This results in two full copies of the events which waste storage space.

From the article about the flight scheduling system mentioned in 2.5.6. We got the idea that we could have the events in the Kafka topics be stored for a limited amount of time. This would be implemented so that when an application first starts up and need to load all events it will first start loading them from the event store. When finished loading them from the event store it would connect to Kafka and start receiving event there. This way

Kafka only need to store the events long enough that it acts as a buffer for the events that occur while the application is loading events from the event store.

Currently, there is no way for other applications to load events directly from the event store. This would have to be implemented for this to work.

5.6.10 Initial data of an aggregate

In our current implementation, some values that would be required in a proper implementation are not set by the create command that creates the aggregate instance.

We should have included all the required values in the create command, but with our current simple business logic, this would reduce the number of events. We decided to keep our current way of handling the create command to give the system more traffic to test the system with. In a more production-ready system, this should have been changed.

5.6.11 Avoid mixing asynchronous and synchronous

Currently, we are mixing synchronous HTTP calls with the asynchronous messaging. The stock allocation service application should probably be changed to use messages for communication instead of HTTP calls.

Mixing these kinds of communication in a microservice system can be an anti-pattern as described in 2.3.6.3.

5.6.12 Persistent projections

In the current implementation of the projections, they are stored in memory, as described in 4.12. This makes them volatile. Because of this, they must be rebuilt at every startup. We have not had any problem with this, but for a system that has been running for a while building the projection from the beginning at every startup would most likely take more time as the event store grows.

To solve this, we should change them to be persisted. This would only require them to catch up with events that have occurred since they went offline, which should reduce downtime.

5.6.13 Smoke testing

Smoke testing as described in 2.4.4.1 could have been implemented to check if the system operates as expected after an update, by running through all events that have occurred and checking if the state is the same as before the update.

6 CONCLUSION

The result of our thesis is a working prototype of an ordering system based on event sourcing and CQRS. This prototype handles simple business logic involving orders, picking, and invoice. It also includes simulator applications that simulate user interaction with the system.

We also found two benefits of event sourcing that DRIW found interesting. It provides full traceability and fits naturally into an event-driven architecture.

We built the system as a distributed system, using microservice and event-driven architecture. The microservices are running on Spring BOOT. The event-driven architecture uses Apache Kafka as a publish/subscribe messaging service. The event store applications publish their events to Kafka topics, which other applications can subscribe to. These applications can use the events received through Kafka to react to state changes in other applications. This could involve changing state in the receiving system or updating a read model.

We implemented read models for each aggregate type in the system. Most of the read models are stored in memory in a hash map data structure, but we did implement one read model using a relational database.

Based on our experiences through this project, we think event sourcing is a very interesting concept that can be very beneficial in some applications. It provides a full audit log of all changes that have happened, this gives full traceability of the system and can be beneficial for applying business intelligence and analysis. It also fits naturally into an event-driven architecture when you make the events in the event store available to other applications. This could be done through publishing to topics in a messaging system, like Apache Kafka which we used. CQRS also gives some benefits through the separation of read and write models. This allows you to build read models that are optimized for reading and gives a lot of freedom to what kind of data store is used for these read models and how you structure them.

Event sourcing doesn’t fit into every system because it adds more complexity than a CRUD approach. CRUD is also something most developers are already familiar with, while event sourcing is not as well known. CRUD is most likely a better option for applications where you are not very interested in an audit log, applying business intelligence, or developing an event-driven system.

We have learned a lot from working on this thesis. We learned about event sourcing which is a very interesting concept that we had not heard of before. We were also able to apply microservice and event-driven architecture which we had heard of before, but never applied in any of the projects we have previously worked on. And in the end, we were able to produce an interesting prototype solution that got positive feedback from DRIW. Below is a statement we got from DRIW in Norwegian.

> *Dette må være en av bacheloroppgavene med best timing. Vi setter nå (i mai 2018) i gang arbeid med den nye ordremodulen i TRACE produktet til Driw. I forbindelse med dette er de vurderingene som er gjort i bacheloroppgaven til nytte for de valgene vi gjør nå fremover. For oss er det to av funnene som Oscar og Robert har gjort som er veldig interessant. Det ene er hvor godt event sourcing passer med en eventbasert arkitektur. Det spiller på lag. Det andre er hvor bra modellen er for sporbarhet. Dvs. koble ikke bare når endringen har skjedd, men i hvilken sammenheng den skjedde. Dette er interessant. Erfaringene som Robert og Oscar har gjort seg i denne oppgaven lever videre i produktdesignet på den nye ordremodulen i Driw.*

- Arne Unneland, DRIW AS.

7 REFERENCES

1. Sommerville I. Software engineering. Tenth edition, global edition. Boston, Mass. Amsterdam Cape Town: Pearson Education Limited; 2016. 810 p. (Always learning).

2. Fowler M. Microservices [Internet]. martinfowler.com. [cited 2018 May 22]. Available from: https://martinfowler.com/articles/microservices.html

3. Dragoni N, Giallorenzo S, Lafuente AL, Mazzara M, Montesi F, Mustafin R, et al. Microservices: yesterday, today, and tomorrow. arXiv:160604036 [cs] [Internet]. 2016 Jun 13 [cited 2018 May 15]; Available from: http://arxiv.org/abs/1606.04036

4. Thönes J. Microservices. IEEE Software. 2015 Jan;32(1):116–116.

5. Liu L, Özsu MT, editors. Encyclopedia of database systems. New York: Springer; 2009. 5 p. (Springer reference).

6. De La Torre C. Challenges and solutions for distributed data management [Internet]. [cited 2018 May 20]. Available from: https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/architect-microservice-container-applications/distributed-data-management

7. Michelson B. Event-Driven Architecture Overview [Internet]. Boston, MA: Patricia Seybold Group; 2006 Feb [cited 2018 May 12]. Report No.: 681. Available from: http://www.customers.com/articles/event-driven-architecture-overview

8. Fowler M. What do you mean by “Event-Driven”? [Internet]. martinfowler.com. [cited 2018 Apr 25]. Available from: https://martinfowler.com/articles/201701-event-driven.html

9. Evans E. Domain-driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional; 2004. 563 p.

10. Fowler M. DDD_Aggregate [Internet]. martinfowler.com. [cited 2018 May 21]. Available from: https://martinfowler.com/bliki/DDD_Aggregate.html

11. Meyer B. Object-oriented software construction. 2nd ed. Upper Saddle River, N.J: Prentice Hall PTR; 1997. 1254 p.

12. Young G. Command Query Separation? | Greg Young [Internet]. [cited 2018 May 20]. Available from: http://codebetter.com/gregyoung/2009/08/13/command-query-separation/

13. Dahan U. Clarified CQRS [Internet]. [cited 2018 May 20]. Available from: http://udidahan.com/2009/12/0/

14. Betts D, Domínguez J, Melnik G, Simonazzi F, Subramanian M. Exploring CQRS and Event Sourcing. 1st ed. Microsoft patterns & practices; 2013. 376 p.

15. Fowler M. CQRS [Internet]. martinfowler.com. [cited 2018 May 20]. Available from: https://martinfowler.com/bliki/CQRS.html

16. Gamma E, editor. Design patterns: elements of reusable object-oriented software. Reading, Mass: Addison-Wesley; 1995. 395 p. (Addison-Wesley professional computing series).

17. Freeman E, Robson E, Sierra K, Bates B, editors. Head First design patterns. Sebastopol, CA: O'Reilly; 2004. 638 p.

18. Bloch J. Effective Java. Third edition. Boston: Addison-Wesley; 2018. 392 p.

19. Coulouris GF, editor. Distributed systems: concepts and design. 5th ed. Boston: Addison-Wesley; 2012. 1047 p.

20. Brewer EA. Towards Robust Distributed Systems (Abstract). In: Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing [Internet]. New York, NY, USA: ACM; 2000 [cited 2018 May 20]. p. 7–. (PODC '00). Available from: http://doi.acm.org/10.1145/343477.343502

21. Gilbert S, Lynch N. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. SIGACT News. 2002 Jun;33(2):51–59.

22. Vogels W. Eventually Consistent. Commun ACM. 2009 Jan;52(1):40–44.

23. JSON [Internet]. [cited 2018 May 1]. Available from: https://www.json.org/

24. Leach P, Mealling M, Salz R. A Universally Unique IDentifier (UUID) URN Namespace. 2005 [cited 2018 May 15]; Available from: https://www.rfc-editor.org/info/rfc4122

25. Berners-Lee T, Fielding R, Frystyk H. Hypertext Transfer Protocol -- HTTP/1.0. 1996 [cited 2018 Mar 12]; Available from: https://www.rfc-editor.org/info/rfc1945

26. Fielding R, Gettys J, Mogul J, Frystyk H, Masinter L, Leach P, et al. Hypertext Transfer Protocol -- HTTP/1.1. 1999 [cited 2018 Mar 12]; Available from: https://www.rfc-editor.org/info/rfc2616

27. Fielding RT. Architectural Styles and the Design of Network-based Software Architectures. University of California, Irvine; 2000.

28. De La Torre C. Communication in a microservice architecture [Internet]. [cited 2018 May 25]. Available from: https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/architect-microservice-container-applications/communication-in-microservice-architecture

29. Gray J. Notes on Data Base Operating Systems. In: Operating Systems, An Advanced Course [Internet]. London, UK, UK: Springer-Verlag; 1978 [cited 2018 May 19]. p. 393–481. Available from: http://dl.acm.org/citation.cfm?id=647433.723863

30. Lamport L, Shostak R, Pease M. The Byzantine Generals Problem. ACM Trans Program Lang Syst. 1982 Jul;4(3):382–401.

31. Silberschatz A, Galvin PB, Gagne G. Operating system concepts. 9. ed., internat. student version. Hoboken, NJ: Wiley; 2014. 829 p.

32. Introduction to Computer Information Systems/Database - Wikibooks, open books for an open world [Internet]. [cited 2018 May 22]. Available from: https://en.wikibooks.org/wiki/Introduction_to_Computer_Information_Systems/Database#Database_Definition_and_Examples

33. Codd EF. A relational model of data for large shared data banks. Communications of the ACM. 1970;11.

34. Chamberlin DD, Boyce RF. SEQUEL: A structured English query language. In ACM Press; 1976 [cited 2018 May 15]. p. 249–64. Available from: http://portal.acm.org/citation.cfm?doid=800296.811515

35. Intro to Graph Databases Series - YouTube - YouTube [Internet]. [cited 2018 May 21]. Available from: https://www.youtube.com/playlist?list=PL9Hl4pk2FsvWM9GWaguRhlCQ-pa-ERd4U

36. Database index. In: Wikipedia [Internet]. 2018 [cited 2018 May 22]. Available from: https://en.wikipedia.org/w/index.php?title=Database_index&oldid=830318096

37. CRUD Database [Internet]. [cited 2018 May 12]. Available from: http://docs.jboss.org/tools/latest/en/seam/html/crud_database_application.html

38. Rothsberg J. Evaluation of using NoSQL databases in an event sourcing system. Linköpings Universitet; 2015.

39. Young G. CQRS and Event Sourcing | Greg Young [Internet]. [cited 2018 May 20]. Available from: http://codebetter.com/gregyoung/2010/02/13/cqrs-and-event-sourcing/

40. GOTO Conferences. GOTO 2014 • Event Sourcing • Greg Young [Internet]. [cited 2018 Apr 25]. Available from: https://www.youtube.com/watch?v=8JKjvY4etTY&t=

41. Overeem M, Spoor M, Jansen S. The dark side of event sourcing: Managing data conversion. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER). 2017. p. 193–204.

42. Scherzinger S, Regensburg O, Klettke M, Storl U. Cleager: Eager Schema Evolution in NoSQL Document Stores. :4.

43. The Good of Event Sourcing: Projections - DZone DevOps [Internet]. dzone.com. [cited 2018 May 22]. Available from: https://dzone.com/articles/the-good-of-event-sourcing-projections

44. Sørhus SK. Applying Learning Analytics in the course TDT4100 at NTNU [Internet]. 2015 [cited 2018 May 22]. Available from: https://brage.bibsys.no/xmlui/handle/11250/2358386

45. Pedersen AH. Bruk av Event Sourcing for logging og visualisering av bruk av nettressurser [Internet]. 2015 [cited 2018 May 22]. Available from: https://brage.bibsys.no/xmlui/handle/11250/2352356

46. Rajković P, Janković D, Milenković A. Using CQRS Pattern for Improving Performances in Medical Information Systems. :6.

47. Debski A, Szczepanik B, Malawski M, Spahr S, Muthig D. A Scalable, Reactive Architecture for Cloud Applications. IEEE Software. 2018 Mar;35(2):62–71.

48. Persistence • Akka Documentation [Internet]. [cited 2018 May 22]. Available from: https://doc.akka.io/docs/akka/2.5.4/scala/persistence.html

49. Eventuate [Internet]. [cited 2018 May 22]. Available from: https://eventuate.io/

50. Documentation | Event Store [Internet]. [cited 2018 May 22]. Available from: https://eventstore.org/docs/

51. EventStore.JVM: Event Store JVM Client [Internet]. Event Store; 2018 [cited 2018 May 22]. Available from: https://github.com/EventStore/EventStore.JVM

52. NEventStore [Internet]. [cited 2018 May 23]. Available from: http://neventstore.org/

53. Blockchain or Event Sourcing | LinkedIn [Internet]. [cited 2018 Apr 25]. Available from: https://www.linkedin.com/pulse/blockchain-event-sourcing-lee-hambley/

54. D'Aliessi M. How Does the Blockchain Work? [Internet]. Michele D'Aliessi. 2016 [cited 2018 May 26]. Available from: https://medium.com/@micheledaliessi/how-does-the-blockchain-work-98c8cd01d2ae

55. What is Blockchain Technology? A Step-by-Step Guide For Beginners [Internet]. Blockgeeks. 2016 [cited 2018 May 26]. Available from: https://blockgeeks.com/guides/what-is-blockchain-technology/

56. TIOBE Index | TIOBE - The Software Quality Company [Internet]. [cited 2018 Mar 13]. Available from: https://www.tiobe.com/tiobe-index/

57. IntelliJ IDEA: The Java IDE for Professional Developers by JetBrains [Internet]. JetBrains. [cited 2018 May 22]. Available from: https://www.jetbrains.com/idea/

58. Gradle Build Tool [Internet]. Gradle. [cited 2018 May 22]. Available from: https://gradle.org/

59. Git - About Version Control [Internet]. [cited 2018 May 22]. Available from: https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control

60. Atlassian. Jira | Issue & Project Tracking Software [Internet]. Atlassian. [cited 2018 May 22]. Available from: https://www.atlassian.com/software/jira

61. Postman [Internet]. Postman. [cited 2018 May 22]. Available from: https://www.getpostman.com/

62. PostgreSQL: About [Internet]. [cited 2018 Apr 27]. Available from: https://www.postgresql.org/about/

63. Apache Kafka [Internet]. Apache Kafka. [cited 2018 Mar 13]. Available from: https://kafka.apache.org/intro

64. Spring Framework Overview [Internet]. [cited 2018 May 22]. Available from: https://docs.spring.io/spring/docs/current/spring-framework-reference/overview.html

65. Spring Data JPA - Reference Documentation [Internet]. [cited 2018 May 22]. Available from: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories.definition

66. Data Access [Internet]. [cited 2018 May 22]. Available from: https://docs.spring.io/spring/docs/current/spring-framework-reference/data-access.html#jdbc

67. OkHttp [Internet]. [cited 2018 May 22]. Available from: http://square.github.io/okhttp/

68. jackson: Main Portal page for the Jackson project [Internet]. FasterXML, LLC; 2018 [cited 2018 May 22]. Available from: https://github.com/FasterXML/jackson

69. baeldung. Intro to the Jackson ObjectMapper [Internet]. Baeldung. 2016 [cited 2018 May 22]. Available from: http://www.baeldung.com/jackson-object-mapper-tutorial

70. Liquibase | Database Refactoring | Databases [Internet]. [cited 2018 May 2]. Available from: https://www.liquibase.org/databases.html

71. Khazanchi A, Kanwar A, Saluja L. JAVA DATABASE CONNECTIVITY (JDBC) - DATA ACCESS TECHNOLOGY [Internet]. International Journal of Engineering and Computer Science; 2013 [cited 2018 May 22]. Available from: https://www.ijecs.in/index.php/ijecs/article/download/2085/1931/

72. PostgreSQL JDBC: Connecting To The PostgreSQL Database [Internet]. [cited 2018 May 22]. Available from: http://www.postgresqltutorial.com/postgresql-jdbc/connecting-to-postgresql-database/

73. H2 Features [Internet]. [cited 2018 May 23]. Available from: http://www.h2database.com/html/features.html

74. Scaling Event Sourcing for Netflix Downloads [Internet]. [cited 2018 Apr 25]. Available from: https://www.infoq.com/presentations/netflix-scale-event-sourcing

75. Apache Kafka [Internet]. Apache Kafka. [cited 2018 Mar 13]. Available from: https://kafka.apache.org/intro

76. Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines) [Internet]. [cited 2018 May 10]. Available from: https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines

APPENDIX

A. Database configuration

The liquibase XML for setting up the event store database. This XML includes the index on aggregate id in the event table and a snapshot table used for storing snapshots.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<databaseChangeLog
        <!--  schema info tags --> >
    <changeSet id="event_table" author="1">
        <createTable tableName="event">
            <column name="id" type="uuid">
                <constraints nullable="false"
                             primaryKey="true"
                             primaryKeyName="event_pk"/>
            </column>
            <column name="aggregate_id" type="uuid">
                <constraints nullable="false"/>
            </column>
            <column name="aggregate_type" type="nvarchar(100)">
                <constraints nullable="false"/>
            </column>
            <column name="data" type="clob">
                <constraints nullable="false"/>
            </column>
            <column name="type" type="nvarchar(100)">
                <constraints nullable="false"/>
            </column>
            <column name="timestamp" type="timestamp">
                <constraints nullable="false"/>
            </column>
        </createTable>
    </changeSet>
    <changeSet id="2" author="1">
        <createIndex tableName="event"
                 indexName="aggregate_id_index"
                 unique="false">
            <column name="aggregate_id" type="uuid"/>
        </createIndex>
    </changeSet>
    <changeSet id="3" author="1">
        <createTable tableName="snapshot">
            <column name="aggregate_id" type="uuid">
                <constraints primaryKey="true"
                             primaryKeyName="snapshot_aggregate_id_pk"/>
            </column>
            <column name="version" type="bigint">
                <constraints primaryKey="true"
                             primaryKeyName="snapshot_version_pk"/>
            </column>
            <column name="data" type="clob">
                <constraints nullable="false"/>
            </column>
            <column name="timestamp" type="timestamp">
                <constraints nullable="false"/>
            </column>
        </createTable>
        <createIndex tableName="snapshot"
                     indexName="snapshot_aggregate_id_index"
                     unique="false">
            <column name="aggregate_id" type="uuid"/>
        </createIndex>
    </changeSet>
</databaseChangeLog
```

B. Diagrams

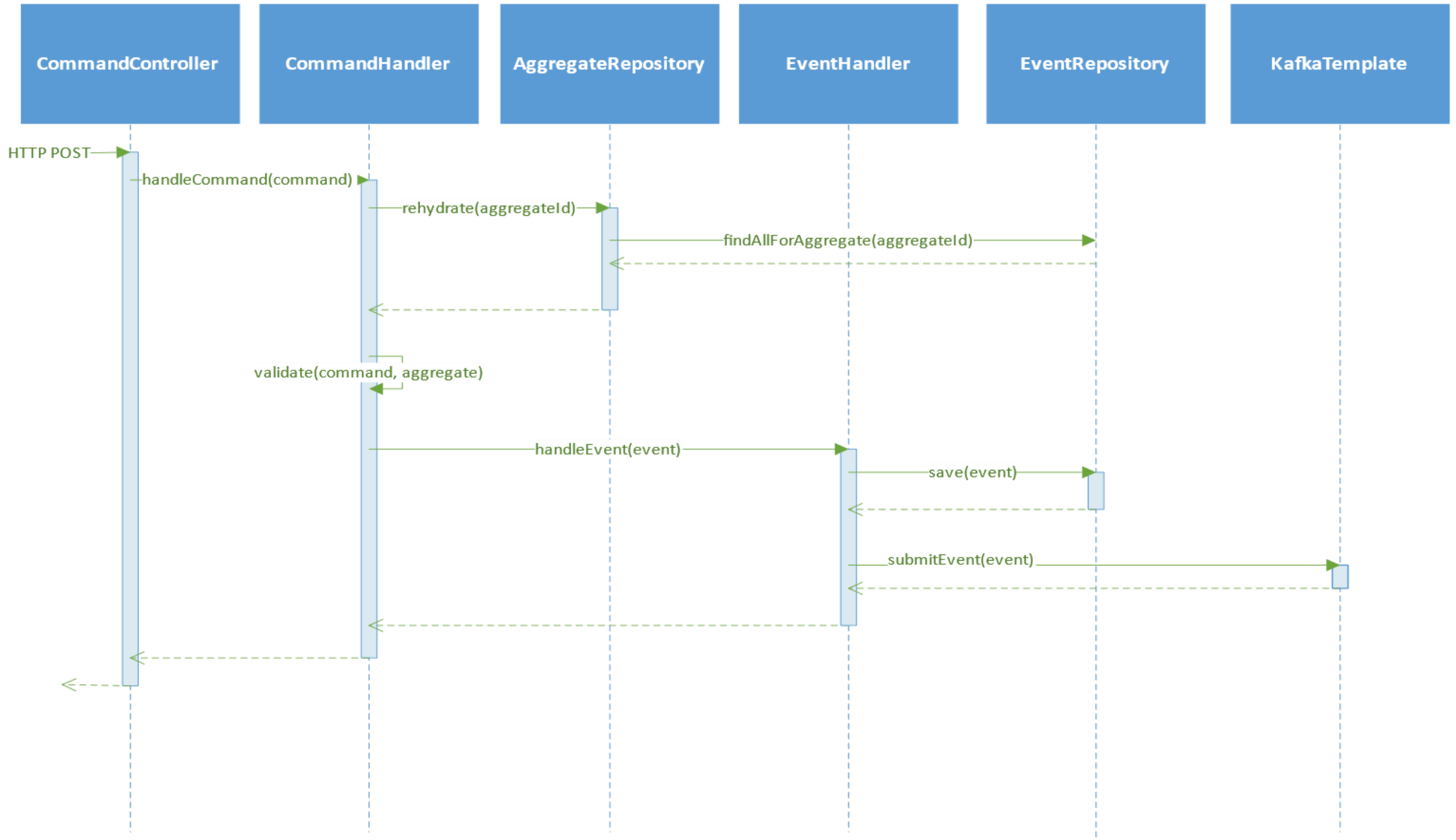


Figure 25: Command handling in event store.

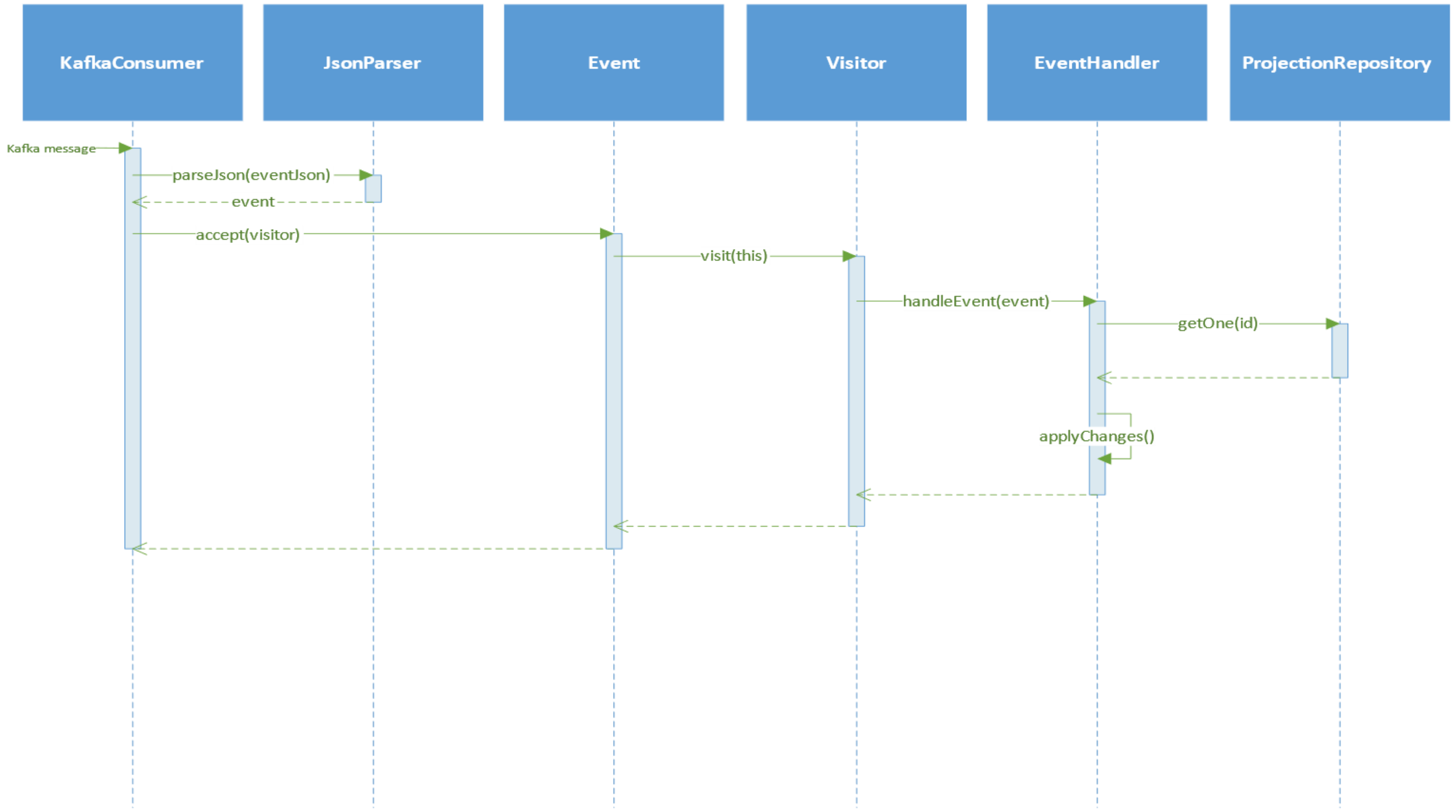


Figure 26: Consuming events in projection.

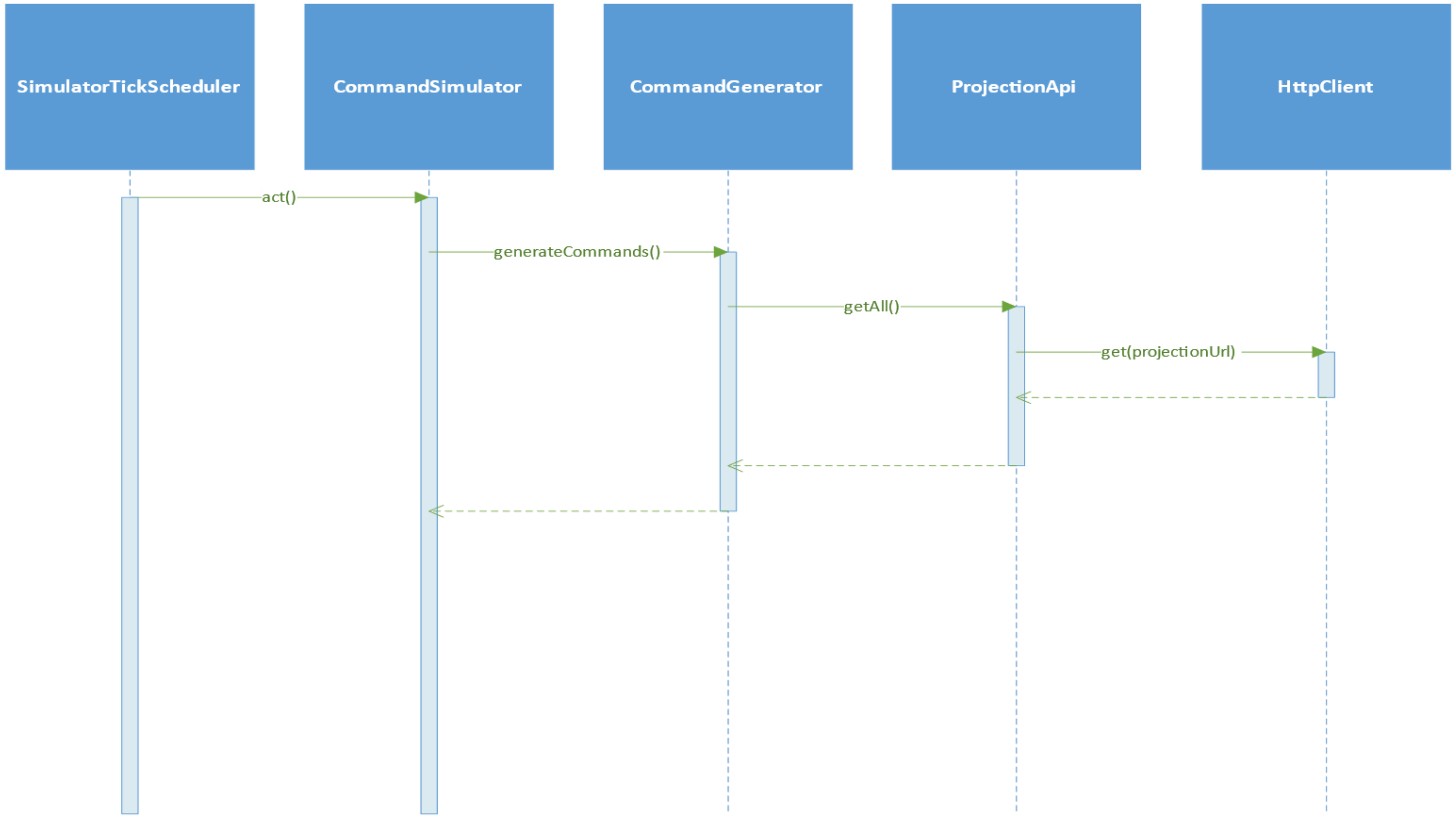


Figure 27: Simulator command generation, continues in Figure 28.

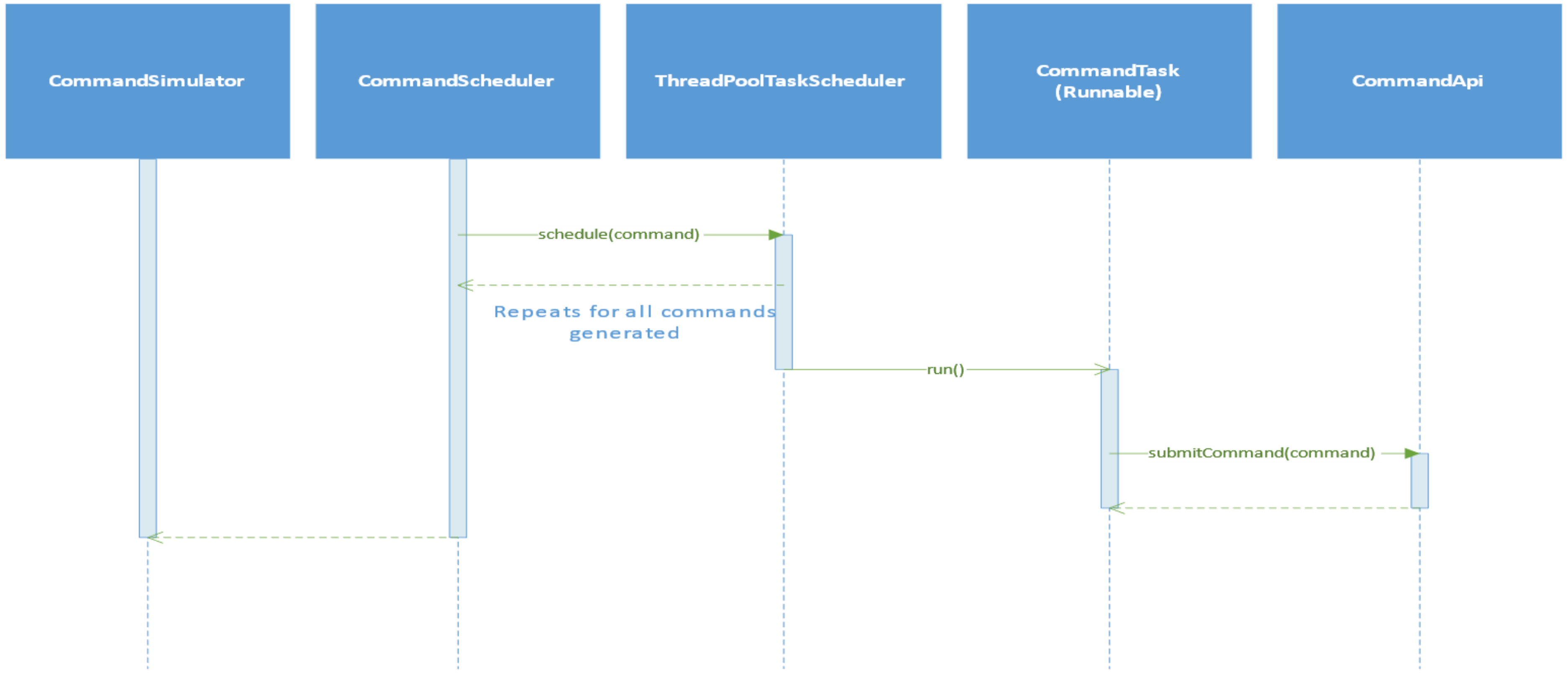


Figure 28: Simulator command scheduling, follows Figure 27.

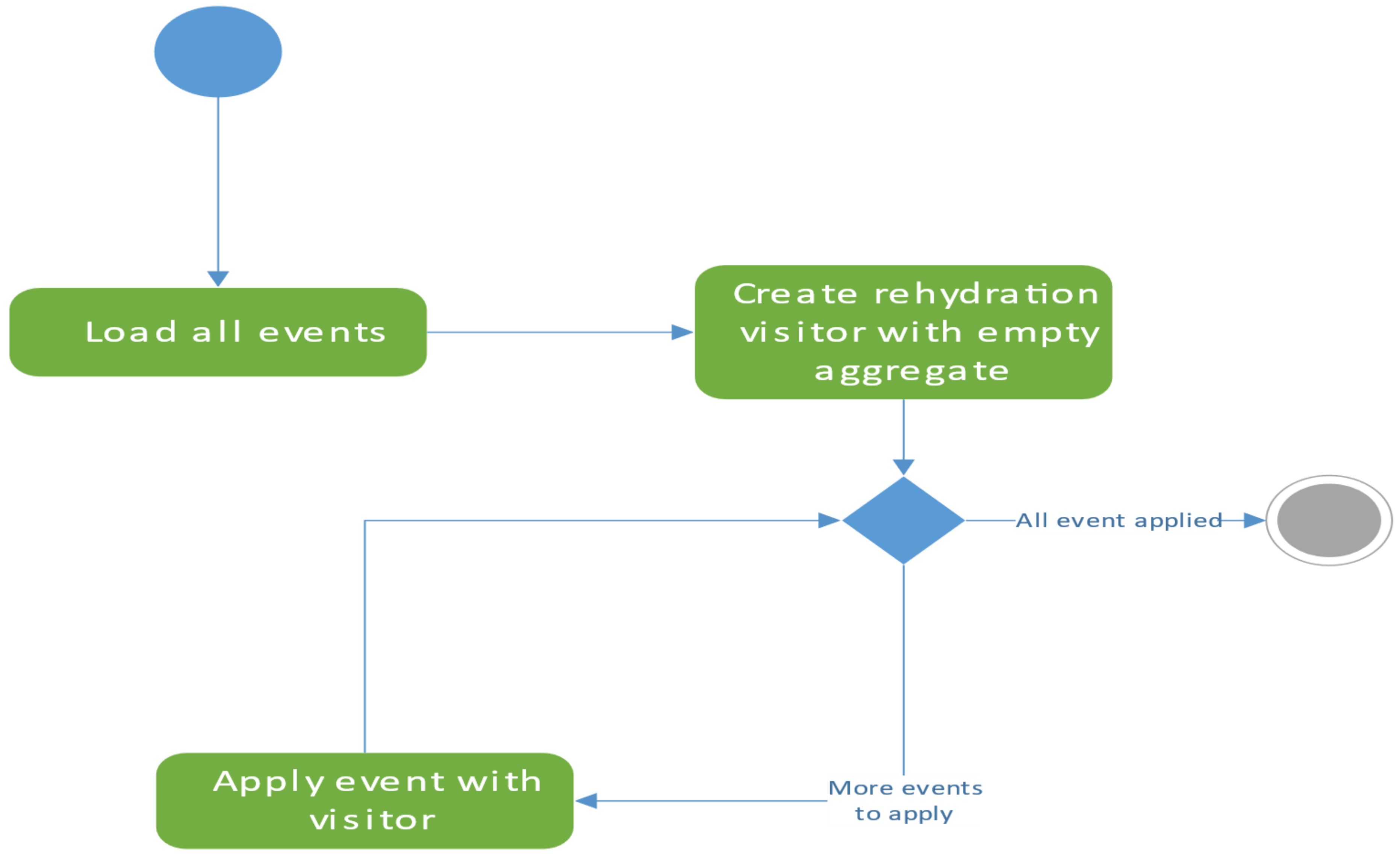


Figure 29: Aggregate rehydration.

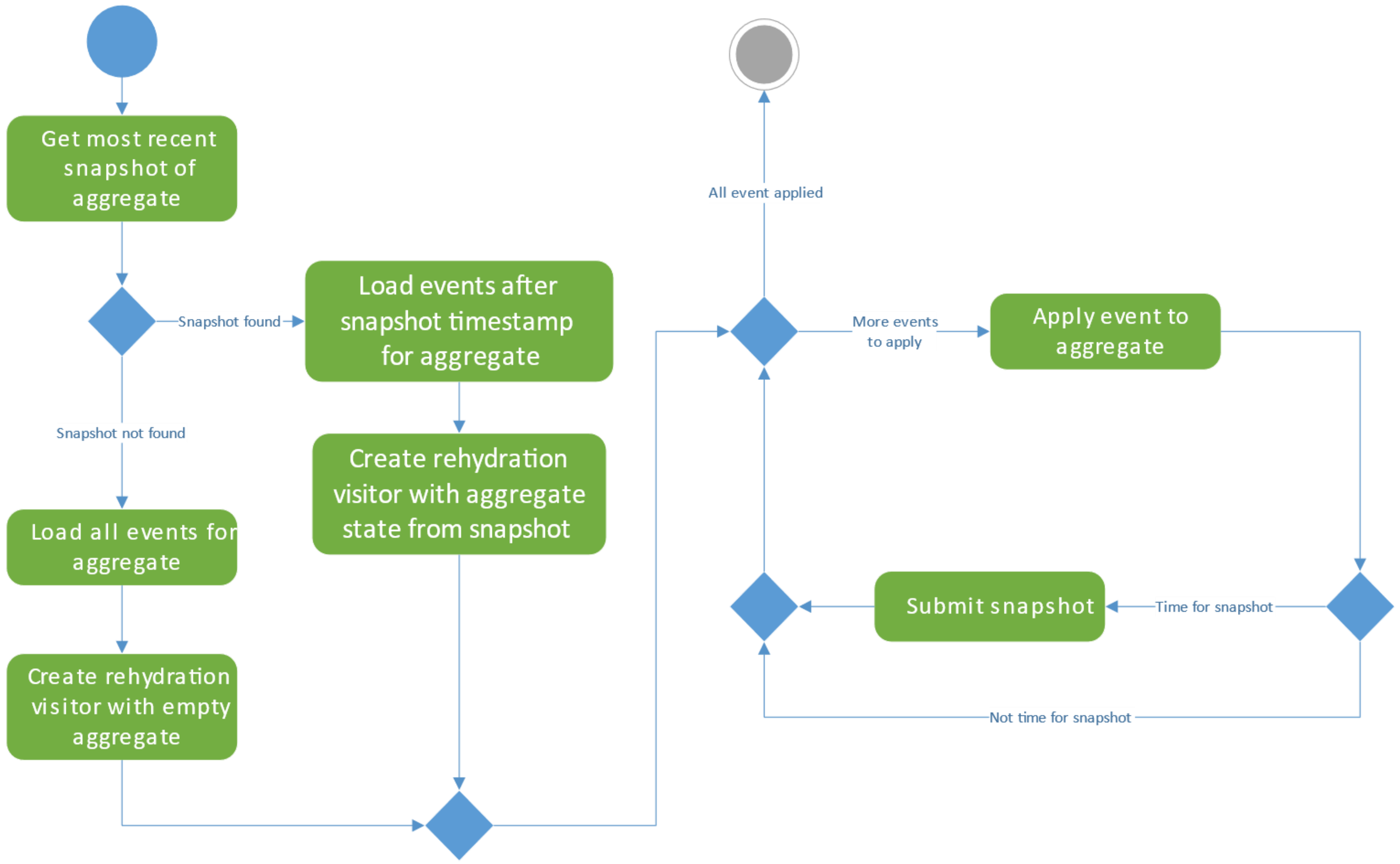

Get most recent snapshot of aggregate
Snapshot found
Load events after snapshot timestamp for aggregate
Snapshot not found
Create rehydration visitor with aggregate state from snapshot
Load all events for aggregate
Create rehydration visitor with empty aggregate
All event applied
More events to apply
Apply event to aggregate
Submit snapshot
Time for snapshot
Not time for snapshot

Figure 30: Aggregate rehydration with snapshotting.