Tri Nguyen

Elementary Event Storage

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Thesis

27 December 2018

Metropolia

Author Title Number of Pages Date	Tri Nguyen Elementary Event Storage 34 pages + 9 appendices 27 December 2018
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineering
Instructors	Janne Salonen, Head of ICT Department

Software is an essential tool to improve business process. In order to improve business process, people need better software. As a consequence, software requirements are now pushed to a higher level. Cloud software architecture has gained the attention of many software developers and architects by its versatility and utility. Event Sourcing is one of the patterns which are currently used by many software vendors.

This thesis introduces Event Sourcing, a new architectural pattern which is widely used in cloud architecture. In addition, CQRS (Command Query Responsibility Segregation) will also be discussed. CQRS is a pattern that separates read and write concerns for an application and together with Event Sourcing it is a good combination for cloud software architecture. Finally, the first implementation of Event Storage to demonstrate Event Sourcing pattern is discussed.

This study is an open source project and will make use of all popular open source technologies. However, the Event Storage project will use NoSQL instead of SQL database to make a proof-of-concept whether NoSQL databases can be a good fit for Event Storage in the long run.

As a result of this study a working implementation covering basic functionalities of an Event Storage library was created. Besides, the author has also provided a base layer of architecture and examples to facilitate implementation for library users. Although CQRS and Event Sourcing are still not a mature architecture, they helped to create more potential for cloud software architecture.

Keyword	Event sourcing, event storage, cqrs

Contents

List of Figures

List of Abbreviations

1 Introduction 1

2 Theoretical Background 2

2.1 Domain Model 2
2.2 Data Mapper 3
2.3 Query Object 4
2.4 Repository 5
2.5 Active Record 5
2.6 Event Sourcing 7
2.6.1 Definition 7
2.6.2 Terminology 8
2.6.3 Event Storage 9
2.6.4 Difference versus Active Record 9
2.6.5 Advantages of Event Sourcing 10
2.6.6 Drawbacks / Limitation 11
2.7 Command Query Responsibility Segregation (CQRS) 11
2.7.1 Definition 11
2.7.2 Write Model 12
2.7.3 Read model 12
2.7.4 Overview 13
2.8 CQRS and Event Sourcing 14

3 Event Storage Project 15

3.1 Motivation 15
3.2 Database structure 16
3.3 Features 17
3.3.1 Creating and appending an event stream 18
3.3.2 Querying for Events 19

3.3.3 Getting entity state from appended events 22

4 Implementation of CQRS + Event Sourcing 25

4.1 Write-side 25
4.1.1 Overview 25
4.1.2 Implementation of simple Command Handler 25
4.2 Read-side 28
4.2.1 Overview 28
4.2.2 Features 29

5 Conclusion 31

5.1 Conclusion of Event Storage project 31
5.2 Conclusion of thesis topic 31

References 33

Appendix 1: Append event to stream implementation 1

Appendix 2: Implementation of getting events from a stream 2

Appendix 3: CurrentBalanceProjection implementation 3

Appendix 4: Examples implementation of building projection 4

Appendix 5: TotalDeposit projection implementation 5

Appendix 6: WriteCommandHandler implementation 6

Appendix 7: BankWriteController implementation 7

Appendix 8: BankReadController 8

Appendix 9: TransactionHistory implementation 9

List of Figures

Figure 1 Domain Model pattern. Taken from [3]

Figure 2 Example of a Data Mapper. Taken from [4]

Figure 3 Query Object pattern. Taken from [5]

Figure 4 Active Record pattern. Taken from [6]

Figure 5 Example of Event Sourcing. Taken from MSDN [7]

Figure 6 An example of event. Taken from [8]

Figure 7 Events that trigger changes. Taken from [8]

Figure 8 An example Event Store with event streams and events. Taken from [10]

Figure 9 A writable interface contains the method which mutates the object

Figure 10 A read-only interface. The property can be read, but cannot be written

Figure 11 InvoiceService implementation with both read and write responsibility

Figure 12 InvoiceService splitted to InvoiceQueryService and InvoiceCommandService

Figure 13 New CQRS Model with Event Storage. Taken from [11]

Figure 14 Flowchart of creating and appending Event Stream. Implementation can be found from Appendix 1

Figure 15 The events persisted in MongoDB

Figure 16 SQL Query for Use Case 1

Figure 17 Events Query for Use Case 1

Figure 18 Example of all AccountDeposited events in 2018

Figure 19 SQL Query for Use Case 2

Figure 20 Event Query for Use Case 2

Figure 21 Example of results for Use Case 2

Figure 22 The process of rebuilding entity's state from events.

Figure 23 AccountTotalDeposits is an aggregation from BankAccountCreated and BankAccountDeposited events

Figure 24 CurrentBalance state is an aggregation of BankAccountCreated, BankAccountDeposited, BankAccountWithdrawed events

Figure 25 Command interface

Figure 26 A command with defined payload

Figure 27 Simple command router implementation

Figure 28 Execution of CreateCommand code block

Figure 29 Method to dispatch the command to handler

Figure 30 An example controller to dispatch command

Figure 31 Event registration for BankAccount

Figure 32 Sample response for bank account information

Figure 33 Event registration for Transaction history. Implementation can be found from Appendix 10

Figure 34 Sample response for bank account transaction

List of Abbreviations

ES	Event Sourcing
CQS	Command Query Segregation
CQRS	Command Query Responsibility Segregation
SQL	Structured Query Language
DDD	Domain Driven Design
DAO	Data Access Object
ORM	Object Relational Mapping
CRUD	Create, Read, Update, Delete
JSON	JavaScript Object Notation

1 Introduction

The requirements of business are continuously growing, and software is now an essential tool to drive and enhance business processes. The non-stop development of hardware (such as processing unit, memory, storage and network) is pushing software performance to a new level. However, software developers never stop to create more approaches to make profit from the improvements of hardware.

Since 2005, Martin Fowler, a well-recognized software developer specialized in object-oriented analysis and design introduced the term “Event Sourcing” as a mechanism to develop scalable software which is highly dependable on extracted events from business process [1]. Additionally, Event Storage was one of the approach to make use of the advanced technologies provided from the hardware paradigm.

Furthermore, since the beginning of software development, data has been a critical source of information. By making use of data, business managers can estimate a retrospective of business process whether later improvements to the process can be suggested or predicting incoming challenges from retrospective.

Event Sourcing was proved as a good pattern in software development where it changed the way software treats data which is more reliable and robust than the legacy approach such as Active Record. With Event Sourcing, combined with DDD (Domain-Driven Design), data is now treated as a first-class citizen, and the way data is processed is more important than ever.

This report reflects the author knowledge and experience gained by working as a Software Developer at Papula-Nevinpat, where the author learned how ES (Event Sourcing) can be good pattern in business-critical systems. Moreover, the author will introduce first implementation of Event Storage which is powered by NodeJS, Typescript and MongoDB.

2 Theoretical Background

All the patterns introduced here will be cited from Patterns of Enterprise Application Architecture, written by Martin Fowler in 2003 [2].

2.1 Domain Model

Domain Model is a pattern which encourages design where any business domain can be represented in terms of behavior and data. This pattern was introduced by Martin Fowler in his book Patterns of Enterprise Application Architecture in 2003. By containing behavior and properties, any entity in the business domain can become independent if there is a change in rules or behavior in other entity.

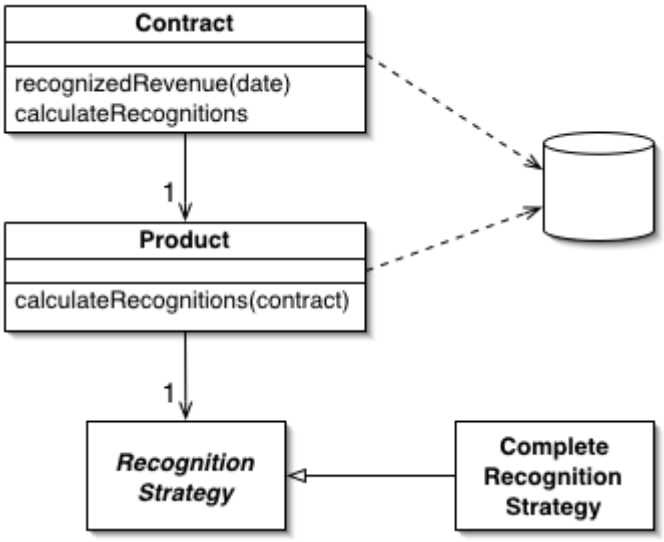


Figure 1. Domain Model pattern. Taken from [3]

The picture above explains how different entities in the system is separated. The Contract object only contains the most relevant methods and properties, same applies to Product. This separation of concerns also helps reducing complexity and coupling between business logic layer and data persistence layer, where there will be everchanging and complicated business rules needed.

While this may be true, the use of this pattern much depends on the complexity of the behavior in the software system. Consequently, applying the knowledge to an application development team will take a significant amount of time and effort.

2.2 Data Mapper

In-memory objects and relational databases contain different mechanism in structuring data. While objects can have inheritance and collections, these terms do not exist in relational databases. To organize the data properties and business logics within a domain object, Data Mapper can be used as a connection between what is represented in the database and what is describe as code, as objects.

Data Mapper is first introduced by Martin Fowler as one of Data Source Architectural Patterns from the book Patterns of Enterprise Application Architecture [2]. Data Mapper acts as a mediator between the in-memory object and relational database. This layer will separate these two components, while being responsible in transferring data between them. With the use of Data Mapper, the object layer will not need to know about the database, the SQL interface and the schema of the database. This will make the Data Mapper layer easier to be tested, and to be replaced or to enable capability to work with multiple databases. See figure 2:

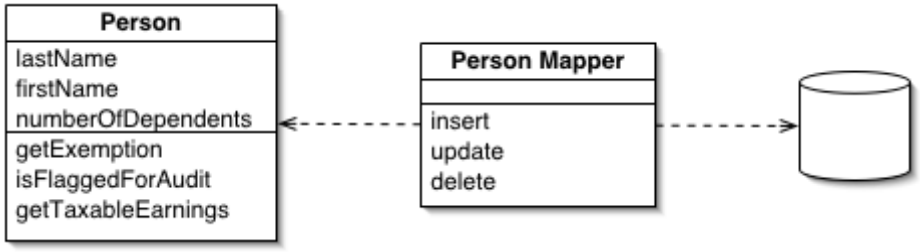


Figure 2. Example of a Data Mapper. Taken from [4]

Many different approaches are introduced for building a Data Mapper

- **Simple:** Mapping object field to an equivalent in a database table on a field-on-field basis

- **Mapping Data to Domain Fields:** Implicit access to the fields in the domain objects, using special OOP techniques (i.e.: reflection.)

- **Metadata-Based Mappings:** determines how field in domain objects will be mapped to database columns using explicit mapper code for each domain object

Data Mapper is considered a versatile pattern, the separation between objects and data source helps each of the components to evolve independently. A developer can work with the domain without the knowledge how it is stored in the database. Along with *Domain Model*, these two patterns enhance the ability to implement more complicated business logics.

However, it is not trivial to implement this pattern. The developer must have a good knowledge in the domain field and how SQL can be built to create this mapper. And if the business logics are simple, using this pattern would be considered as over-complicated. Last but not least, *Active Record* is also proved as a better fit for simpler use cases.

2.3 Query Object

Having SQL queries in line with business logics can cause many different problems. First, it prevents the software to evolve when the database layer needs to be changed, even if the differences between new SQL syntaxes are minor. Secondly, a basic level of database is required so queries can be formed as demonstrated in Figure 3. Query Object solves these problem by creating a structure of object than can transform into a SQL query.

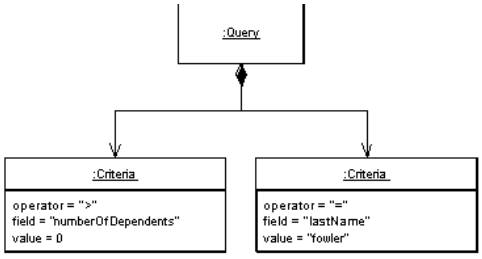


Figure 3. Query Object pattern. Taken from [5]

Query Object brings a lot of benefits, it encapsulates the database schemas, it facilitates when changes from database and database schemas are needed. However, the query object is a good choice when the query is not very complex, and for most use cases, it is only useful when being used along with Domain Model and Data Mapper patterns.

2.4 Repository

As business grows, the number of domain classes is getting higher, the size of a domain classes can become large and querying operations are complex, it is beneficial to create an additional layer which query construction code is concentrated.

Repository is a Data Source Architectural pattern that encapsulates the logic required to access and manipulates data sources. Repository usually treated as an abstraction which works directly with the database where adding, removing, querying operations can be executed by executing actions from database access code.

As a pattern, Repository brings a lot of benefits for enterprise application. First, this pattern operates as an intermediary between the domain model and data mapping layer, helping developer to focus more on data persistence logic by an object-oriented view of the persistence layer. Also, it supports the purpose of separation of concern where clients never need to think in term of SQL, as query can be written in an object-oriented way, readability and clarity in queries code will be dramatically improved.

This pattern is introduced as a follow-up of Query Object pattern, a more concrete example of this pattern will be introduced in Section 4, where the author uses combined knowledge of different architectural patterns in a single project.

2.5 Active Record

Originated from Domain Model, Active Record is another architectural pattern which integrates data access logic in the domain object, as indicated in Figure 4. Every Active Record is:

- Responsible for persisting and loading data to the database.
- Correctly represents the columns represented from the database table
- Contains business logic

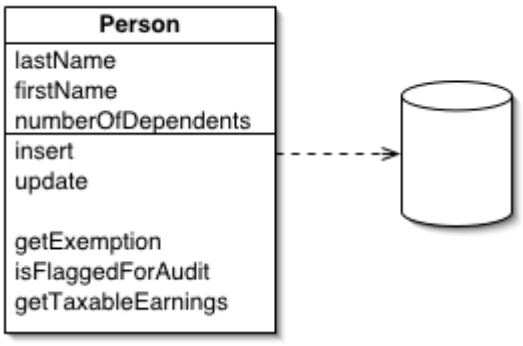


Figure 4. Active Record pattern. Taken from [6]

From the example above, by implementing Active Record pattern, a Person class typically contains:

- Its properties (firstName, lastName, numberOfDependents)
- The methods for data manipulation (insert, update)
- Business logic methods (getExemption, isFlaggedForAudit, getTaxableEarnings)

The pattern provides much convenience for the Person class. Now the Person class not only correctly represented the columns from Person table in database, but also contains domain logics to modify and persist data back to the database, as we simply say CRUD (Create, Read, Update, Delete) operations.

The pattern can be used as a mediator between data persistence layer and business layer, as a more advanced implementation of Domain Model (where Domain Model does not contain data persistence logic). It is a good and easy choice if the firsts implementation of the business entity is simple.

However, while Active Record enhances control over persistence layer, it is now closely coupled to the schema design. As the development project progress, with ever changing and more complex data model (i.e. inheritance and database relationship), it will be challenging to refactor or enhance further, which might lead to a redesign.

2.6 Event Sourcing

All the patterns introduced above were implemented as well-known software, for example Entity Framework from Microsoft and Hibernate from RedHat are very popular ORM software. Likewise, the author would like to introduce another different pattern, where data modelling will be approached differently.

2.6.1 Definition

Event sourcing is a method of storing application's state. As demonstrated by Figure 5, the events occurred in the system are persisted into a persistence layer, called Event Store.

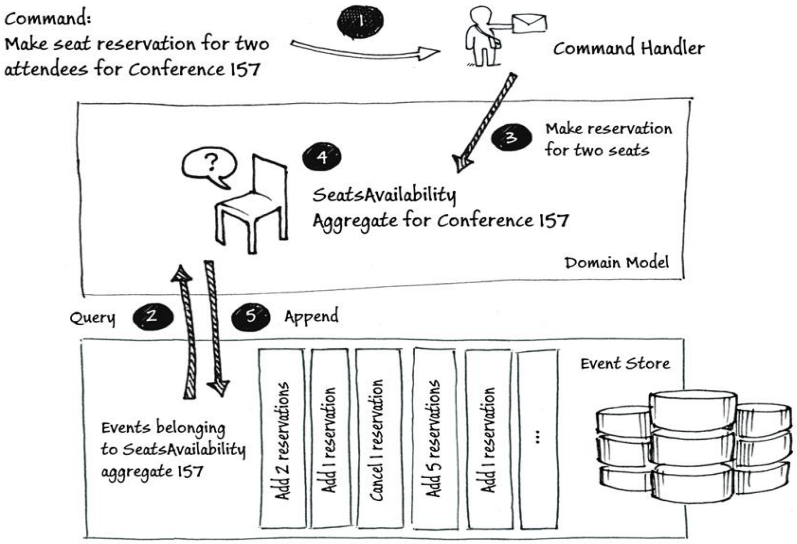


Figure 5: Example of Event Sourcing. *Taken from MSDN [7]*

Accounting can be considered as one of the most relevant areas where event sourcing can be beneficial. In real-life scenario, it is not practical for accountants to record the current balance of a bank account. Preferably, it is compulsory to maintain a list of transactions which occurred since the creation of an account. As a result, the balance of an account will be reflected through the aggregation of transactions.

It is still being in discussion whether Event Sourcing is a good software pattern for enterprise software. In 2005, Martin Fowler introduced Event Sourcing as another approach

for state handling technique using CRUD (Create, Read, Update, Delete) operations, known as Active Record (which was introduced by the same author). [6]

As recently in QConf 2017, a conference for software developer held in New York, Netflix presented their implementation of Event Sourcing in their Video Download feature which resulted a very success integration.

2.6.2 Terminology

An **event** is considered as something that has happened in the past. The figure below is an example of an event, contains the information which will be stored by Event Storage layer.

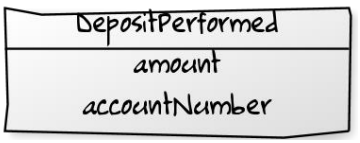


Figure 6: An example of event. Taken from [8]

In an Event Sourced system (a software system that implements an Event Sourcing patterns), an event is treated as the source of truth. As event is a fundamental part of the system, it is crucial for an event to meet those requirement**:**

- An event must be immutable, no manual update/remove operation on an event is allowed, as an event in the past cannot be changed.

- Event data must be sufficient and comprehensible. An event should consist:

    - Time, source of occurrence

    - Version number of the state which the event occurs

The events which have triggered the entity's state to change will be stored in a timely-ordered collection, this collection is termed **Event Stream**, described in Figure 7.

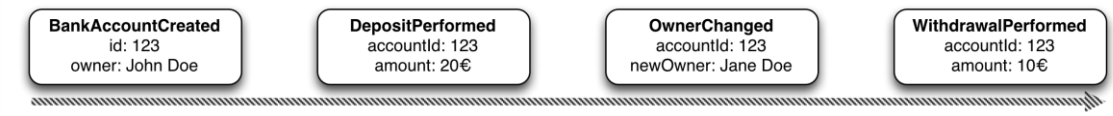


Figure 7: Events that trigger changes. Taken from [8]

2.6.3 Event Storage

Event Storage is the most fundamental part of an Event Sourced system, the figure below explains how an event storage works. An Event Storage contains all event streams, where all the events occurred in the system are persisted.

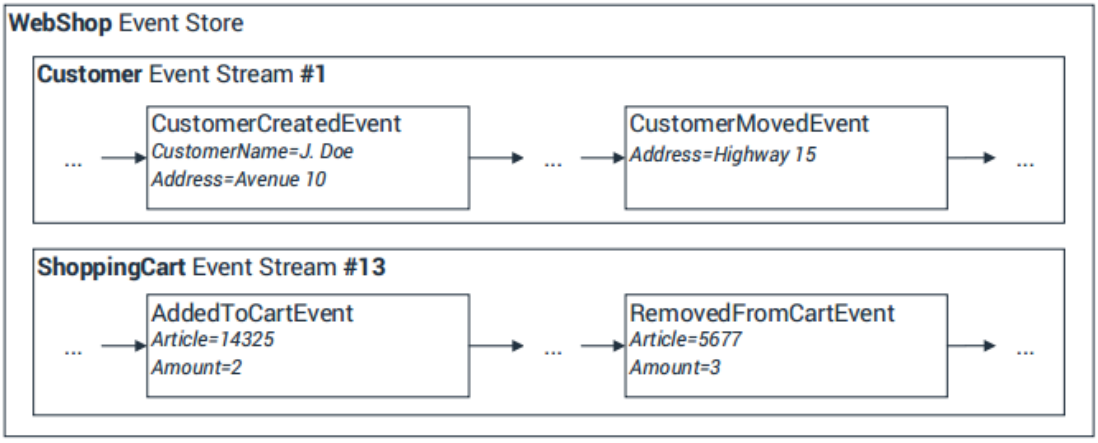


Figure 8: An example Event Store with event streams and events. Taken from [10]

2.6.4 Difference versus Active Record

A difference between Active Record and Event Sourcing is how the handling of a state is executed. While Active Record always stores the current state of the object, event sourcing persists a stream of events which modify the state. The sequence of events which change the entity's state are captured and persisted in the database, provides availability to the history of the entity. [9, 2.]

In addition, Active Record and Event Sourcing are also different in behavior and structure. While Active Record embraces the behavior of CRUD, Event-Sourcing only ap-

pends data (events), no deletion is executed. On the structural side, while Event Sourcing at first only requires two tables (Events, Streams), depends on the number of entities used from the application, Active Record would require one table for each entity, with multiple constraints and reference.

In short, the difference between Active Record and Event Sourcing can be demonstrated using the table below:

Table 1. Main differences between Active Record and Event Sourcing

	Active Record	Event Sourcing
Data Persistence	CRUD	Append-only No deletion is allowed
Structural	One table per entity	Two tables (Events and Streams)

2.6.5 Advantages of Event Sourcing

As cited in On the Potential of Event Sourcing for Retroactive Actor-based programming, since the event log persists every event that have mutate / modify / change the state of the application, it provides the capability to retrospect. Reconstruction of the state of the is always possible to re-applying events from the beginning of the event log to a target point in time. In addition, the authors also approve the practicability of event sourcing to provide state recovery of actors in an Actor-based system [9, 2.]

In fact, Martin Fowler suggested in his blog that Event Replaying is also a benefit that can be gained through an Event Sourcing system. Whenever a past event was incorrect, or an application state was faulty due to incorrect application of an event sequence, events can be replayed, and state can be rebuilt. This technique is favorable for any system that communicates asynchronously. [1]

2.6.6 Drawbacks / Limitation

Event Sourcing brought many advantages to the way software is constructed. Conversely, it is also worth noting that Event Sourced system might as well has its own drawbacks.

From the technical point of view, event sourcing has a very steep learning curve to implement in a real-world application. For any large-scale enterprise application, any attempt to build an event-sourced application by migrating from current running system will also take a significant time.

Similarly, maintaining event-sourced application is still questionable. There has been discussion whether operating an event-sourced application in the long run is a good approach. Because business operations might change over time, the cost of change are not clearly determined.

As an example, Michiel, Marten and Slinger, authors of *The Dark Side of Event Sourcing: Managing Data Conversion* addressed that event sourcing architectural is something relatively new, including the lack of standardized tools and the large amount of data in event storage which will introduce challenges for data conversion in event sourced application. To address the problems, the authors have proposed some techniques and frameworks to tackle the challenges, to improve the process of upgrading event storage while not disrupting the current running system [10, 1.]

Otherwise, Benjamin, Gerhard and Franz refer a more general drawback in operation which is disk space demand for persistence reason will continuously increase. As the systems is running and actors are communicating, new messages and events will be constantly appended by the system [9, 4.]

2.7 Command Query Responsibility Segregation (CQRS)

2.7.1 Definition

CQS is a principle of imperative computer programming which is devised by Bertrand Meyer. The principle states that every method should either be a command that performs an action, or a query that returns the data to the caller.

CQRS is an architectural pattern which applies CQS, first introduced by Greg Young and Udi Dahan where objects in system are separated into two categories:

- **Command**: objects responsible only in mutating data (Write-model)
- **Query**: objects responsible only in returning data (Read-model)

2.7.2 Write Model

Write model shown below is an object contains only methods to modify the state of the object, and its method must not return any value.

```
export interface ICanWriteProperties {
    setAge(age: number): void;
    setName(name: string): void;
}
```

Figure 9. A writable interface contains the method which mutates the object

2.7.3 Read model

Read model is an object contains only methods to returns the state and related data of acquired object, and it must not directly or indirectly mutate the state of the object.

The loaded object from read model can be considered as a Domain Model which contains only properties and without any method to mutate it. See Figure 10:

```
export interface ICanReadProperties {
    Name: string;
    Age: number;
}
```

Figure 10. A read-only interface. The property can be read, but cannot be written

With only Read concerns/responsibility, the read model now is likely to be less coupled from write operations. The read model is now facilitated to be evolved and shaped different from the write side, as long as it can be rebuilt from the persistence layer.

2.7.4 Overview

The terminology is simple, but CQRS provides great opportunity for changes in architectural perspective. For example, the original method where a service for managing invoice is described as below as CRUD operations, as demonstrated in Figure 11 below:

Figure 11. InvoiceService implementation with both read and write responsibility.

By applying CQRS on the InvoiceService, the service is now separated into two services as shown below. There is now a Query side (InvoiceQueryService) and a Command side (InvoiceCommandService). See Figure 12 below:

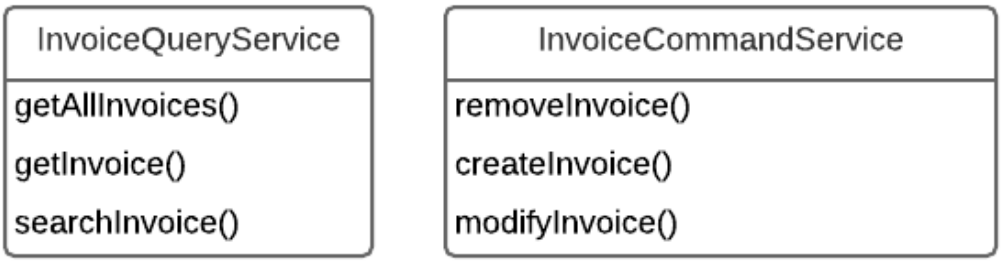


Figure 12. InvoiceService splitted to InvoiceQueryService and InvoiceCommandService

With CQRS applied, there is now a clear separation between Read and Write model. While this enhance manageability, it also adds extensibility for both models, where either Read model can now implement a different data model or Write model can have another approach for data manipulation.

However, while this pattern proves useful for some use case, it is believed to increase accidental complexity if a user interface is proven to be very complex or composed by different data model. Employing CQRS into a project would require consideration and good evaluation whether the business domain is complex or not.

2.8 CQRS and Event Sourcing

Since CQRS and Event Sourcing has been introduced, Greg Young believes that CQRS is a stepping stone for event sourced application. The picture below describes how an application which applies CQRS and Event Sourcing is a viable architecture, where Event Storage is used a write side, and a Data Access Object (DAO) is used as the read side.

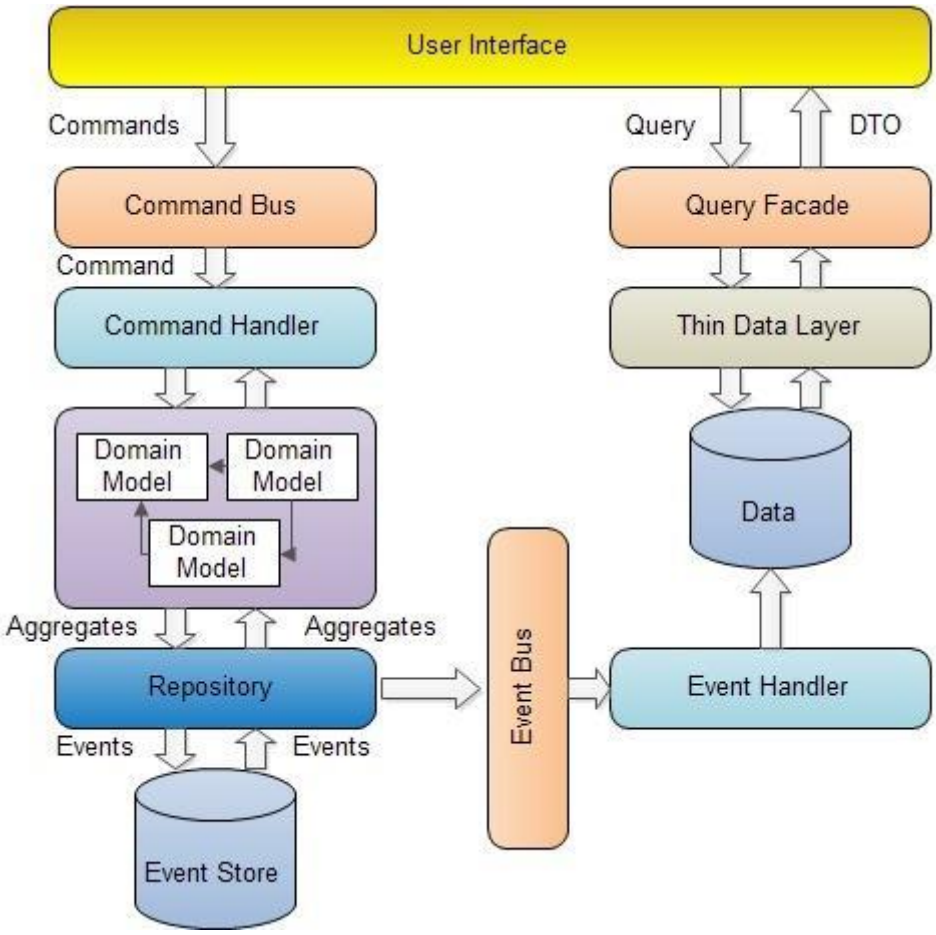


Figure 13. New CQRS Model with Event Storage. Taken from [11]

3 Event Storage Project

3.1 Motivation

The motivation behind this project is to give a brief introduction of implementing some mentioned enterprise application architecture and how Event Storage can be applied for a simple banking application. The interface of Event Storage implementation is much inspired from Jeremy D. Miller's project Marten [12].

For prototyping purpose, only the use of open source software is highly considered.

The implementation will benefit from NodeJS as a runtime environment. Developed and released since 2009, Node.js is an open-source and cross-platform JavaScript runtime that allows JavaScript to be executed in the server-side. Powered by Google Chrome's

V8 Engine, Node.js is now the most widely used runtime environment for both server-side and web development.

The programming language used in this implementation is Typescript. Developed by Microsoft since 1st October 2012**,** Typescript proved its successful by being the main language in Angular, a front-end framework built by Google. Now it is a very viable choice for building simple and large-scale software project.

Lastly, MongoDB will be used as the database to test out the functionality of this implementation. As the goal of this project is to implement a simple Event Storage, MongoDB is considered a good fit because of its gradual learning curve, good performance, and support from the community.

3.2 Database structure

Since for most Event Storage implementation, a SQL database has been used for the implementation. For this project, the author would like to experiment whether a NoSQL implementation is possible, as MongoDB as the choice for database. The structure of the schema should be simple and straightforward, as only one Events collection will be created. See Table 2 below:

Table 2. Collection row definition

Column Name	Data Type	Description
_id	String	Identifier of the event, must be unique
Data	JSON	The data of the event
StreamId	String	The identifier of the event stream
Type	String	The name of event
AppendDate	Date	The date the event is persisted to the database
Version	Number	The version which the event applies to

3.3 Features

To serve the purpose of this report, all business-logic related software flow will be omitted. This report will only cover the basics of Event Storage implementation.

The events that are created from resulted changes will be stored in the database. In order to store them, an implementation to communicate with a database was created, called the Repository. The Repository is responsible for working with every database operation, and its first features are storing events, querying events and rebuilding application state from events.

3.3.1 Creating and appending an event stream

The graph below in Figure 14 represents the software flow of creating or appending an event stream. First, the Repository will determine whether a stream identity for these events exists in the database:

- If not, then all to be appended events will share the same newly created stream identity. This is a process of *creating new Event Stream*.

- If so, then the events will share the existing stream identity and appended to the database. This process is called *appending events to Event Stream*.

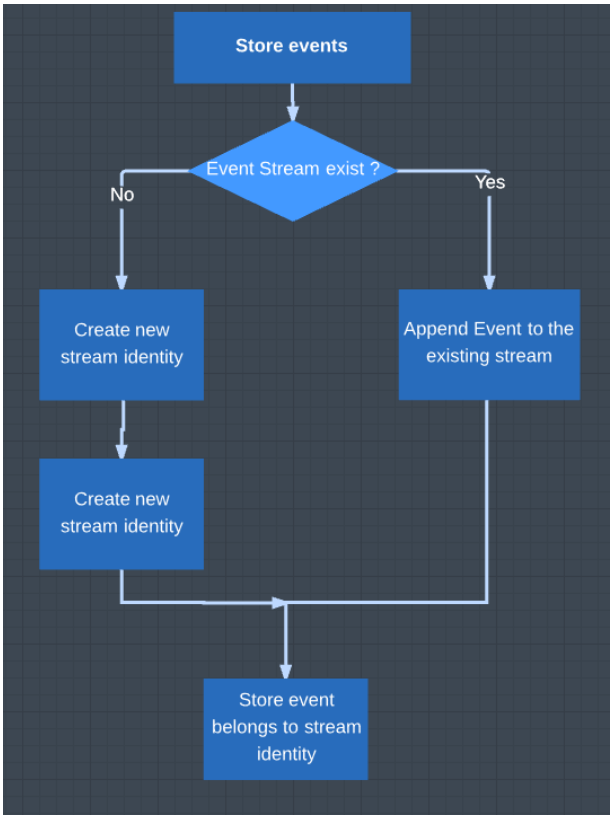


Figure 14: Flowchart of creating and appending Event Stream. Implementation can be found from Appendix 1

The events shown in Figure 15 will now be persisted into the database, in this case MongoDB collection.

_id	Data	AppendDate	StreamId	Type	Version
5bdf0b0e1f30b54540298b36	{ 1 fields }	2018-11-04T15:06:54.754Z	5bdf0b0e1f30b54540298b35	AccountDeposited	1
5bdf0b0e1f30b54540298b37	{ 1 fields }	2018-11-04T15:06:54.757Z	5bdf0b0e1f30b54540298b35	NameSet	2
5bdf0b0e1f30b54540298b38	{ 1 fields }	2018-11-04T15:06:54.757Z	5bdf0b0e1f30b54540298b35	AccountDeposited	3
5bdf0b0e1f30b54540298b39	{ 1 fields }	2018-11-04T15:06:54.757Z	5bdf0b0e1f30b54540298b35	NameSet	4
5bdf0b0e1f30b54540298b3a	{ 1 fields }	2018-11-04T15:06:54.758Z	5bdf0b0e1f30b54540298b35	AccountDeposited	5
5bdf0b0e1f30b54540298b3b	{ 1 fields }	2018-11-04T15:06:54.758Z	5bdf0b0e1f30b54540298b35	NameSet	6
5bdf0b0e1f30b54540298b3c	{ 1 fields }	2018-11-04T15:06:54.758Z	5bdf0b0e1f30b54540298b35	AccountDeposited	7
5bdf0b0e1f30b54540298b3d	{ 1 fields }	2018-11-04T15:06:54.758Z	5bdf0b0e1f30b54540298b35	NameSet	8
5bdf0b0e1f30b54540298b3e	{ 1 fields }	2018-11-04T15:06:54.758Z	5bdf0b0e1f30b54540298b35	AccountDeposited	9
5bdf0b0e1f30b54540298b3f	{ 1 fields }	2018-11-04T15:06:54.758Z	5bdf0b0e1f30b54540298b35	NameSet	10
5bdf0b0e1f30b54540298b40	{ 1 fields }	2018-11-04T15:06:54.758Z	5bdf0b0e1f30b54540298b35	AccountDeposited	11
5bdf0b0e1f30b54540298b41	{ 1 fields }	2018-11-04T15:06:54.758Z	5bdf0b0e1f30b54540298b35	NameSet	12
5bdf0b0e1f30b54540298b42	{ 1 fields }	2018-11-04T15:06:54.759Z	5bdf0b0e1f30b54540298b35	AccountDeposited	13
5bdf0b0e1f30b54540298b43	{ 1 fields }	2018-11-04T15:06:54.759Z	5bdf0b0e1f30b54540298b35	NameSet	14
5bdf0b0e1f30b54540298b44	{ 1 fields }	2018-11-04T15:06:54.759Z	5bdf0b0e1f30b54540298b35	AccountDeposited	15
5bdf0b0e1f30b54540298b45	{ 1 fields }	2018-11-04T15:06:54.759Z	5bdf0b0e1f30b54540298b35	NameSet	16

Figure 15. The events persisted in MongoDB

3.3.2 Querying for Events

Because the events are appended, the Repository can be used to query them. However, there are different techniques in querying events, depends on the business use cases, which makes Event Storage a reliable solution for critical business processes.

To serve the purpose of comparison, we will simulate the Active Record implementation, where a relational database has been used to store banking data. As three tables (Accounts, Deposits, Withdraws) will be used for this system.

Use case 1: Query the number of bank account deposits happened in 2018 (Querying specific event)

The original approach would be query for all row from bank deposits table where timestamp is in 2018. This can be simplified with a simple SQL query, the author has omitted SQL JOIN statements for simplicity in Figure 16 below:

```
1   SELECT * FROM Bank.Deposits
2   WHERE timestamp ⩾ '2018-01-01' AND timestamp < '2019-01-01'
```

Figure 16. SQL Query for Use Case 1

Indeed, this use case can be dealt with in a different approach with Event Storage, where we would query for AccountDeposited events instead. To simplify from using MongoDB query language, the author would translate it into SQL query as below.

```
1   SELECT * FROM Bank.Events
2   WHERE type='AccountDeposited'
3   AND AppendDate ⩾ '2018-01-01' AND AppendDate < '2019-01-01'
```

Figure 17. Events Query for Use Case 1

The result of the query below would be a list of AccountDeposited events.

5bdf0b0e1f30b54540298b46 { 1 fields }	2018-11-04T15:06:54.759Z	5bdf0b0e1f30b54540298b35	AccountDeposited	17
5bdf0b0e1f30b54540298b48 { 1 fields }	2018-11-04T15:06:54.759Z	5bdf0b0e1f30b54540298b35	AccountDeposited	19
5bdf424c063d00371062f05f { 1 fields }	2018-11-04T19:02:36.726Z	5bdf424c063d00371062f05e	AccountDeposited	1
5bdf424c063d00371062f061 { 1 fields }	2018-11-04T19:02:36.730Z	5bdf424c063d00371062f05e	AccountDeposited	3
5bdf424c063d00371062f063 { 1 fields }	2018-11-04T19:02:36.730Z	5bdf424c063d00371062f05e	AccountDeposited	5
5bdf424c063d00371062f065 { 1 fields }	2018-11-04T19:02:36.731Z	5bdf424c063d00371062f05e	AccountDeposited	7

Figure 18. Example of all AccountDeposited events in 2018

Use case 2: Query bank account withdraws and deposits from user A in 2018

For reporting purpose, the most straightforward approach would be query for everything from all tables, with JOINs statements and having user identifier as a criterion. See Figure 19:

```
SELECT * FROM Bank.Account a
INNER JOIN Bank.Deposits d
    ON a.AccountID = d.AccountID
INNER JOIN Bank.Withdraw w
    ON a.AccountID = w.AccountID
WHERE a.AccountID = 'A'
```

Figure 19. SQL Query for Use Case 2

By using Event Storage, only events are considered. In this case, we assume that the stream of events are the bank account history, where StreamID is the identifier of the user, as indicated below:

```
SELECT * FROM Bank.Events
WHERE Type='AccountDeposited' OR Type='AccountWithdrawed'
AND StreamID='A'
```

Figure 20. Event Query for Use Case 2

The results of Use Case 2 should be a list of events of AccountDeposited and Withdrawed where all of these events belong to the stream which has the same identifier as the user identifier. See Figure 21:

5bdf0b0e1f30b54540298b38 {} { 1 fields }	2018-11-04T15:06:54.757Z	5bdf0b0e1f30b54540298b35	AccountDeposited	3
5bdf0b0e1f30b54540298b3a {} { 1 fields }	2018-11-04T15:06:54.758Z	5bdf0b0e1f30b54540298b35	AccountDeposited	5
5bdf0b0e1f30b54540298b3c {} { 1 fields }	2018-11-04T15:06:54.758Z	5bdf0b0e1f30b54540298b35	AccountWithdrawed	7
5bdf0b0e1f30b54540298b3e {} { 1 fields }	2018-11-04T15:06:54.758Z	5bdf0b0e1f30b54540298b35	AccountDeposited	9
5bdf0b0e1f30b54540298b40 {} { 1 fields }	2018-11-04T15:06:54.758Z	5bdf0b0e1f30b54540298b35	AccountDeposited	11
5bdf0b0e1f30b54540298b42 {} { 1 fields }	2018-11-04T15:06:54.759Z	5bdf0b0e1f30b54540298b35	AccountWithdrawed	13
5bdf0b0e1f30b54540298b44 {} { 1 fields }	2018-11-04T15:06:54.759Z	5bdf0b0e1f30b54540298b35	AccountWithdrawed	15
5bdf0b0e1f30b54540298b46 {} { 1 fields }	2018-11-04T15:06:54.759Z	5bdf0b0e1f30b54540298b35	AccountDeposited	17

Figure 21. Example of results for Use Case 2

3.3.3 Getting entity state from appended events

The Repository can query for all appended events that belong to the same stream identity and rebuilt the current state of the entity. The figure below describes the process how the Bank Account object is re-constructed using events.

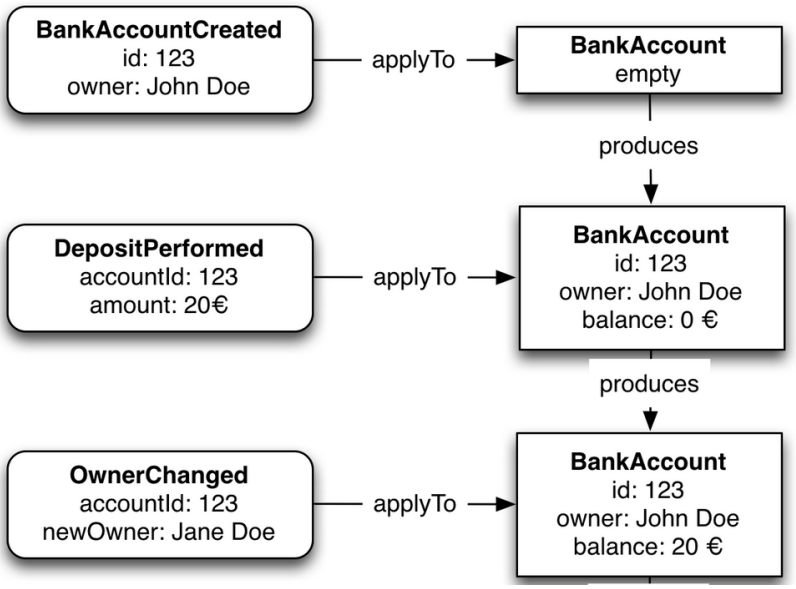


Figure 22: The process of rebuilding entity's state from events.

However, depending on different use cases, where only some specific properties of an entity are considered, a projection can be created, as it only cares/be built from some specific events, which can be understood as an aggregation of specific events
For instance, a simple BankAccount entity that can append and apply events from table below.

Table 3. Bank account events

Event name	**Occurrence**
BankAccountCreated	A bank account is created
BankAccountDeposited	A deposit issued
BankAccountWithdrawed	A withdraw issued
BankAccountSuspended	Account is suspended

For example, when an application needs to rebuild a state which represents the total of deposits from a bank account, only events related to deposits are required to rebuild the state entirely, this will be illustrated in Use Case 1.

Use Case 1: Get the total of deposits from account

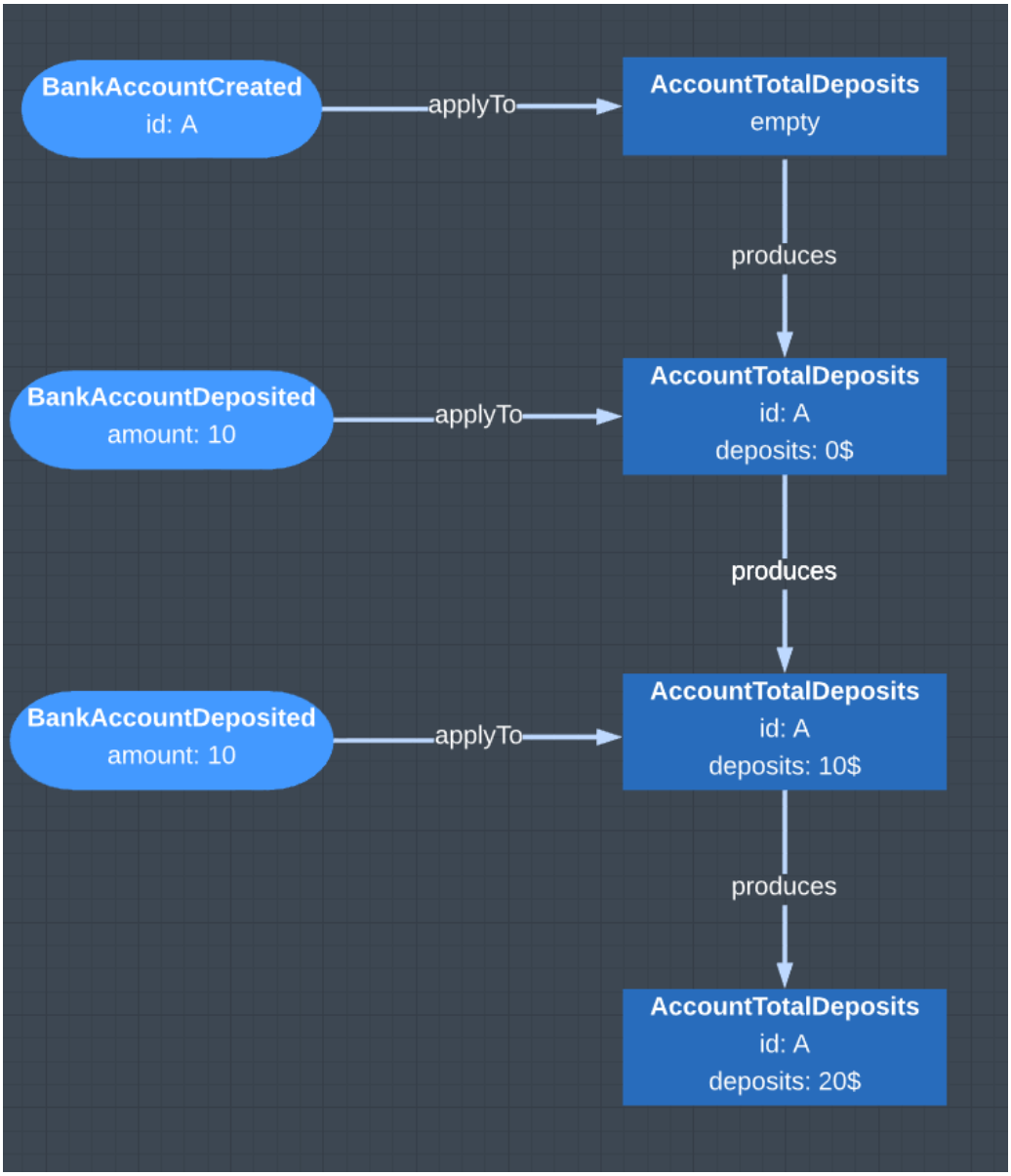


Figure 23. AccountTotalDeposits is an aggregation from BankAccountCreated and BankAccountDeposited events

The implementation for TotalDeposits projection can be found at Appendix 5

In another use case where the application needs to fetch the current balance of a bank account, all events which affect account balance must be taken into account for state rebuilding.

Use Case 2: Get the current balance of account

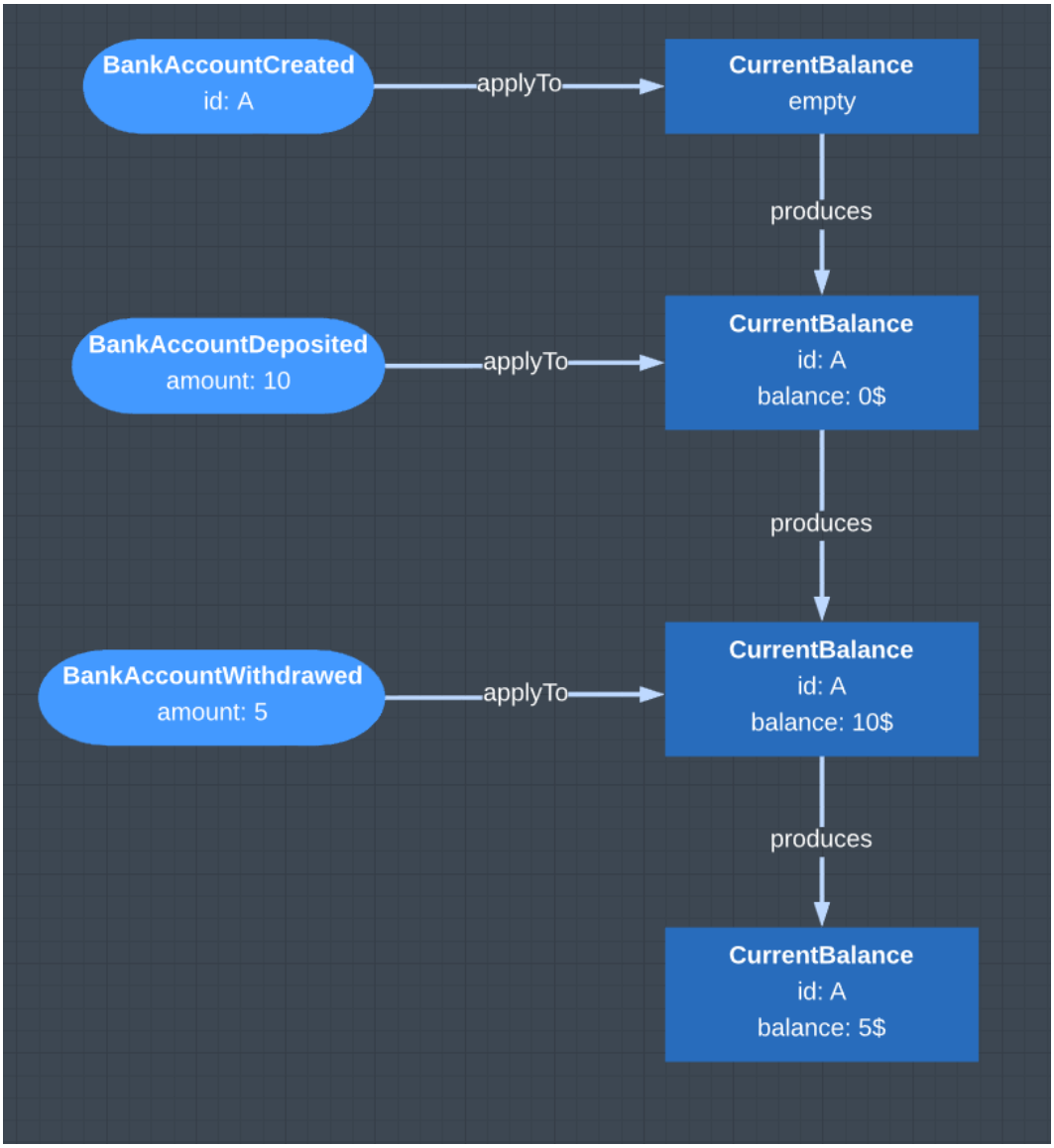


Figure 24. CurrentBalance state is an aggregation of BankAccountCreated, BankAccountDeposited, BankAccountWithdrawed events

The implementation for CurrentBalance projection can be found at Appendix 3

4 Implementation of CQRS + Event Sourcing

This projection will be a combination of Event Storage features project from Section 3, along with minimal implementation of CQRS pattern for a simple banking web service.

4.1 Write-side

4.1.1 Overview

The write side will be implemented as a set of Node.js Express controllers with POST methods. Each controller will be considered as a command to be dispatched to the Command Handler, as illustrated in Table 4. For simplicity, we will not follow any strict guidelines from REST API design.

Table 4. Write-side features

Feature	**Controller**	**Command**
Create a bank account	POST /account	CreateCommand
Deposit to an account	POST /account/deposit	DepositCommand
Transfer from one account to another	POST /account/transfer	TransferCommand

4.1.2 Implementation of simple Command Handler

a) Defining Commands

Firstly, a command interface should be created, which should contain a payload, which stores the required information to perform the operation. A generic interface is demonstrated below, where T is the class definition of the payload. See Figure 25 below:

```
export interface ICommand<T> {
    payload: T;
}
```

Figure 25. Command interface

Secondly, a payload type is defined, and a command with corresponding payload can be created. See Figure 26.

```
export class CreatePayload {
    constructor(public Name: string, public Amount: number) {

    }
}

export class CreateCommand implements ICommand<CreatePayload> {
    payload: CreatePayload;

    // We create command payload from constructor
    constructor(name: string, amount: number) {
        this.payload = new CreatePayload(name, amount);
    }
}
```

Figure 26. A command with defined payload

b) Creating Handlers

As commands are implemented, there should be a handler to execute the command and each command should lead to different behavior of the system with different behavior. For this reason, a router should be an essential component of a command handler. To illustrate further, the author will implement a simple command router, as demonstrated with the figure below.

```
readonly commandRouter = {
    'CreateCommand': this.createBankAccount.bind(this),
    'DepositCommand': this.depositBankAccount.bind(this),
    'TransferCommand': this.transferBankAccount.bind(this),
}
```

Figure 27. Simple command router implementation

The figure above shows a simple implementation. Specifically, the core idea is still a key-value object, as correspondent command name should lead to correct action dispatched.

Therefore, the execution part of the command can now be implemented, as shown in Figure 28:

```
private createBankAccount(createCommand: CreateCommand): void {
    this.bankWriteSvc.createBankAccount(
        createCommand.payload.Name,
        createCommand.payload.Amount
    );
}
```

Figure 28. Execution of CreateCommand code block

Finally, a method to dispatch the command to command handler, demonstrated in Figure 29:

```
public dispatchCommand(command: any): void {
    try {
        const handler = this.commandRouter[command.constructor.name];

        if (handler) {
            handler(command);
        } else {
            throw new Error(`No handler found for this command ${command.constructor.name}`);
        }

    } catch (e) {
        console.error(`Error occured for handler ${command.constructor.name}`, e);
    }
}
```

Figure 29. Method to dispatch the command to handler

The complete implementation of the write-side handlers can be found at Appendix 5

c) Dispatching command from HTTP controller

Dispatching commands are straightforward using HTTP controllers. In the sample below, we make use of payload type definition to be the request body. Due to simplicity, validation logic will be omitted from the code. See Figure 30:

```
@Post('/account')
create(@Body() createPayload: CreatePayload): Promise<any> {
    const command = new CreateCommand(createPayload.Name, createPayload.Amount);
    this.handler.dispatchCommand(command);

    return null;
}
```

Figure 30. An example controller to dispatch command

The implementation for the controller can be found in Appendix 6

4.2 Read-side

4.2.1 Overview

The read-side of the project will be implemented as a set of Express controllers using GET methods. This set of controllers will be the query side, where every read operation is executed. The query side will not include command handler implementation but injecting directly event storage instance to the controller with dependency injection.

The features mentioned below in Table 5 are implemented as a practice of feature from Event Storage project (Section 3.3.3.). The read-side HTTP controller implementation can be found from Appendix 8.

Table 5. Read-side features

Feature	**Controller**	**Parameters**
Get bank account basic information	GET /account/:id	Id: Id of the account

Get bank account transaction history	GET /account/:id/transaction	

4.2.2 Features

As the features are already implemented while Event Storage project was introduced, this section will only demonstrate what response can a web service can provide by getting a specific entity states, building projection operation can be found at Appendix 4.

a) Get bank account basic information

Bank account represents the state of a basic bank account, which accepts all the events appended to the event stream (i.e.: account is created, deposits, withdraws, account name is changed). See Figure 31 below:

```
public WireUpEvents(): void {
    this.RegisterEvent(AccountCreated, this.accountCreated);
    this.RegisterEvent(AccountDeposited, this.accountDeposited);
    this.RegisterEvent(NameSet, this.accountHolderSet);
    this.RegisterEvent(AccountWithdrawed, this.accountWithdrawed);
}
```

Figure 31. Event registration for BankAccount

The picture below demonstrates a response given from server side for bank account information from BankAccount projection. See Figure 32:

```
{
    "Name": "Jason Do",
    "Balance": 2000
}
```

Figure 32. Sample response for bank account information

b) Get bank account transactions

Transaction history object illustrated in Figure 33, represents a state which only accepts events that changed the account balance (i.e.: account created with starting balance, deposits from account, withdraws from account)

```
// I only cares about AccountCreated, AccountDeposited, AccountWithdrawed
private WireUpEvents(): void {
    this.RegisterEvent(AccountCreated, this.accountCreated);
    this.RegisterEvent(AccountDeposited, this.accountDeposited);
    this.RegisterEvent(AccountWithdrawed, this.accountWithdrawed);
}
```

Figure 33. Event registration for Transaction history. Implementation can be found from Appendix 10

Figure 34 below shows a sample response from the server


```
{
    "History": [
        {
            "Type": 2,
            "Amount": 1000,
            "Description": "First transaction"
        },
        {...},
        {...},
        {...},
        {...},
        {...},
        {
            "Type": 1,
            "Amount": -2000,
            "Description": "Transfer from 2000 euros 5bf094b1a28752505844a65f to 5bf094aca28752505844a65d"
        },
        {
```


Figure 34. Sample response for bank account transaction

5 Conclusion

5.1 Conclusion of Event Storage project

All the basic features introduced in this report have been implemented successfully and used for proof-of-concept and new feature testing. By implementing a very simple Event Storage project, building an Event Sourced application with Node.js are now facilitate, as there was no good Event Storage implementation for Node.js provided by the open-source community so far.

Typescript has been a very helpful programming language to use. The structure of the project is very clean, implementing more features is achievable. It is also beneficial for new contributors to familiarize themselves into the project.

At the beginning of this project, there was skepticism towards the selection of MongoDB as a storage layer for this project since MongoDB could be a not suitable fit because of its characteristic of being a non-relational database. But the more feature implemented, the more good results the author could achieve from usage of MongoDB.

Furthermore, there are some possible enhancements that can be done. The first step is to ensure the condition of being transactional of the project, one approach is to utilize MongoDB 2 Phases Commits (2PC) feature, another approach is to have all events to be appended in the same commit.

On the other hand, due to the constraint of time and scope of the project, persisting projection while appending events is not implemented. This feature would drastically help with querying projections. Currently, the projection can be built imperatively by fetching and applying the events, similar to example in Section 3.3.3.

5.2 Conclusion of thesis topic

Event Sourcing is a new pattern that can change the way people build software. By perceiving the system as a stream of events, the possibility to track and rebuild state of the

application is always available. Combining with CQRS pattern to split read-side and write-side, the whole software infrastructure can become more robust and flexible.

As a side note, Event Sourcing also has its own drawbacks. As the business requirements changed, events will need to be changed as well. Alongside with the changes of events, some events might become obsolete and new events must be introduced. Eventually, there will be many events in the system and migration is required. At the time being, there is no pre-built solution for it, as the development have to come up with different strategy and build their own framework to proceed. Moreover, because of the prematurity of Event Sourcing, there is a lack of material and good guidance for development teams who would like to tackle their problems with this pattern.

In conclusion, besides from the drawback, it is notable that Event Sourced application can be a great step further for Cloud infrastructure, where microservices and Big Data are playing important roles in digitalization processes of enterprises.

References

1 Martin Fowler's blog - Event Sourcing [online].

URL: https://martinfowler.com/eaaDev/EventSourcing.html
Accessed: 26.10.2018

2 Fowler, Martin 2003, Patterns of Enterprise Application Architecture, Addison Wesley.

3 Domain Model

URL: https://martinfowler.com/eaaCatalog/domainModel.html
Accessed: 12.11.2018

4 Data Mapper

URL: https://martinfowler.com/eaaCatalog/dataMapper.html
Accessed: 28.10.2018

5 Query Object

URL: https://martinfowler.com/eaaCatalog/queryObject.html
Accessed: 28.10.2018

6 Active Record

URL: https://martinfowler.com/eaaCatalog/activeRecord.html
Accessed: 28.10.2018

7 Microsoft Developer Network (MSDN) – Introducing Event Sourcing

URL: https://msdn.microsoft.com/en-us/library/jj591559.aspx
Accessed: 28.10.2018

8 Event Sourcing in Practice presentation

URL: https://ookami86.github.io/event-sourcing-in-practice
Accessed: 19.11.2018

9 Erb B, Habiger G, Hauck FJ. On the Potential of Event Sourcing for Retroactive Actor-based Programming. ACM. 2016:1-5asdfasdf

10 Overeem M, Spoor M, Jansen S. The dark side of Event Sourcing: Managing Data Conversion. Software Analysis, Evolution and Reengineering (SANER), IEEE. February 2017:193-204

11 Introduction to CQRS

URL: https://www.codeproject.com/Articles/555855/Introduction-to-CQRS

Accessed: 28.10.2018

12 JasperFX Marten Project

URL: http://jasperfx.github.io/marten/

Accessed: 28.10.2018

13 Designing the infrastructure layer

URL: https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design

Accessed: 28.10.2018

14 Typescript

URL: https://en.wikipedia.org/wiki/TypeScript

Accessed: 20.10.2018

15 Node.js

URL: https://en.wikipedia.org/wiki/Node.js

Accessed: 17.10.2018

16 MongoDB

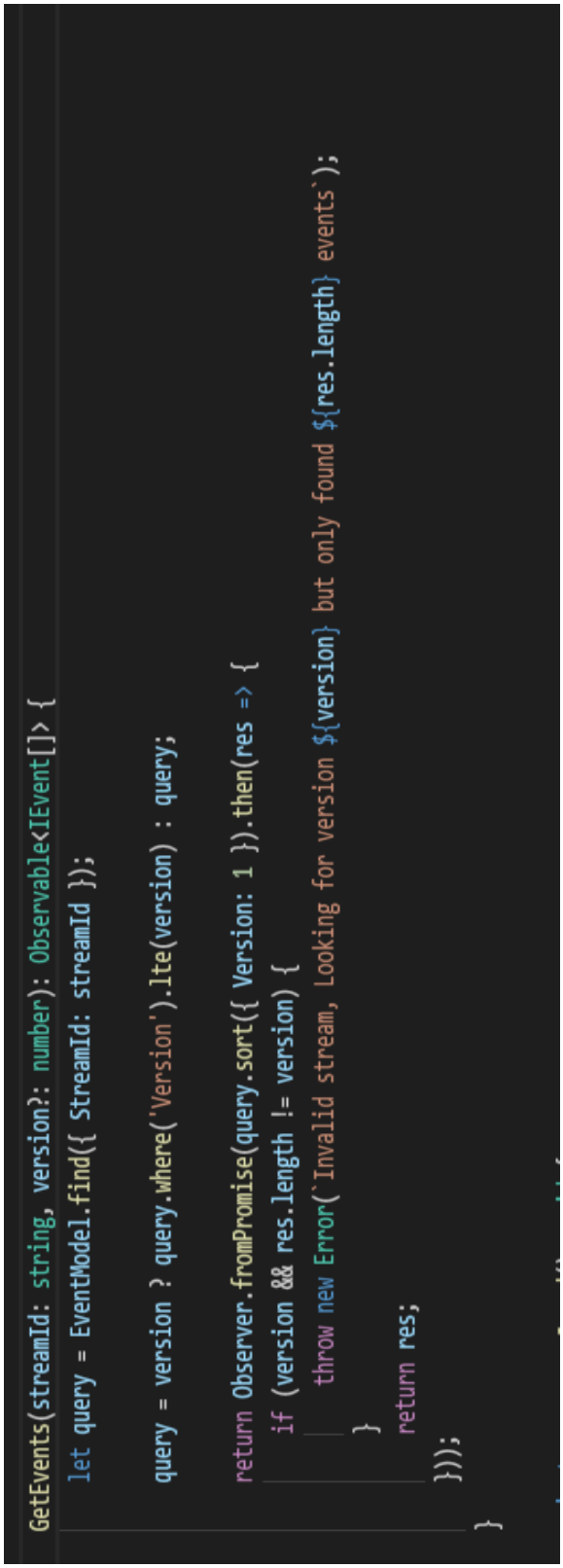
URL: https://en.wikipedia.org/wiki/MongoDB

Accessed: 16.10.2018

Appendix 1: Append event to stream implementation

```
AppendStream(streamId: string, events: any[], newStream: boolean = false, version?: number): void {
    try {

        const actionId = Math.random();
        const streamState$ = newStream ? Observable.of(new StreamStateBase()) : this.GetStreamState(streamId).map((state: any) => {
            const innerExpectedVersion: number = version || (state.CurrentVersion + events.length);
            const outerExpectedVersion: number = state.CurrentVersion + events.length;

            console.log('outerExpected', state, state.CurrentVersion, events.length);

            if (innerExpectedVersion != outerExpectedVersion) {
                throw new Error('Version mismatch !');
            }
            return state;
        });

        streamState$.subscribe((res: StreamStateBase) => {
            console.log('map streamstate', res);
            this.persistEvents(this.fromEvents(events, streamId, actionId, res.CurrentVersion));
        }, err => {
            throw new Error(err);
        })
    } catch (err) {
        throw new Error(`Error while appending stream ${streamId} ${err}`);
    }
}
```

Appendix 2: Implementation of getting events from a stream



```
GetEvents(streamId: string, version?: number): Observable<IEvent[]> {
    let query = EventModel.find({ StreamId: streamId });

    query = version ? query.where('Version').lte(version) : query;

    return Observer.fromPromise(query.sort({ Version: 1 }).then(res => {
        if (version && res.length != version) {
            throw new Error(`Invalid stream, Looking for version ${version} but only found ${res.length} events`);
        }
        return res;
    }));
}
```

Appendix 3: CurrentBalanceProjection implementation

```
import { ProjectionBase } from '../../../../src/infrastructure/interface/stream-state-base';
import { AccountCreated, AccountDeposited, AccountWithdrawed } from '../bank-account-events';

export class CurrentBalance extends ProjectionBase {

    private balance: number = 0;
    public get Balance(): number {
        return this.balance;
    }

    constructor(streamId: any) {
        super(streamId);
        this.WireUpEvents();
    }

    // I only cares about AccountCreated, AccountDeposited, AccountWithdrawed
    public WireUpEvents(): void {
        this.RegisterEvent(AccountCreated, this.accountCreated);
        this.RegisterEvent(AccountDeposited, this.accountDeposited);
        this.RegisterEvent(AccountWithdrawed, this.accountWithdrawed);
    }

    // Appliers
    public accountCreated = (ev: AccountCreated): void => {
        this.balance = ev.startBalance;
    }

    public accountDeposited = (ev: AccountDeposited): void => {
        this.balance = this.balance + ev.Value;
    }

    public accountWithdrawed = (ev: AccountWithdrawed): void => {
        this.balance = this.balance - ev.Value;
    }
    // End of appliers
}
```

Appendix 4: Examples implementation of building projection

```
async getBankTransactionProjection(userId: string): Promise<TransactionHistory> {
    const events = await this.eventStore.getEvents(userId).toPromise();

    const transactionProjection = new TransactionHistory(userId);
    events.forEach((ev: any) => transactionProjection.ApplyEvent(ev.Data, ev.Type));

    return transactionProjection;
}

async getBankTotalDepositProjection(userId: string): Promise<AccountTotalDeposit> {
    const events = await this.eventStore.getEvents(userId).toPromise();

    const depositProjection = new AccountTotalDeposit(userId);
    events.forEach(ev => {
        depositProjection.ApplyEvent(ev)
    });

    return depositProjection;
}

async getBankCurrentBalanceProjection(userId: string): Promise<CurrentBalance> {
    const events = await this.eventStore.getEvents(userId).toPromise();

    const currentBalance = new CurrentBalance(userId);
    events.forEach((ev: any) => {
        if (ev) {
            currentBalance.ApplyEvent(ev.Data, ev.Type)
        };
    });
    return Promise.resolve(currentBalance);
}
```

Appendix 5: TotalDeposit projection implementation

```
import { ProjectionBase } from '../../../../src/infrastructure/interface/stream-state-base';
import { AccountCreated, AccountDeposited } from '../bank-account-events';

export class AccountTotalDeposit extends ProjectionBase {
    private totalDeposit: number = 0;

    constructor(streamId: any) {
        super(streamId);
        this.WireUpEvents();
    }

    // I only cares about AccountCreated and AccountDeposited
    public WireUpEvents(): void {
        this.RegisterEvent(AccountCreated, this.accountCreated);
        this.RegisterEvent(AccountDeposited, this.accountDeposited);
    }

    public get TotalDeposit(): number {
        return this.totalDeposit;
    }

    // Appliers
    public accountCreated = (ev: AccountCreated): void => {
        this.totalDeposit = ev.startBalance;
    }

    public accountDeposited = (ev: AccountDeposited): void => {
        this.totalDeposit = this.totalDeposit + ev.Value;
    }
    // End of appliers
}
```

Appendix 6: WriteCommandHandler implementation

```
import { IWriteBankAccount } from './../services/write.interface';
import { BankAccountRepository } from './../services/bank-repository';
import { CreateCommand, DepositCommand, TransferCommand } from './interface/command.interface';
import { Service } from 'typedi';

@Service()
export class WriteCommandHandler {

    // Make sure we don't use any read method here
    bankWriteSvc: IWriteBankAccount;

    readonly commandRouter = {
        'CreateCommand': this.createBankAccount.bind(this),
        'DepositCommand': this.depositBankAccount.bind(this),
        'TransferCommand': this.transferBankAccount.bind(this),
    }

    // Dependency Injection is used
    constructor(public bankRepo: BankAccountRepository) {
        this.bankWriteSvc = bankRepo;
    }

    private createBankAccount(createCommand: CreateCommand): void {
        this.bankWriteSvc.createBankAccount(
            createCommand.payload.Name,
            createCommand.payload.Amount
        );
    }

    private depositBankAccount(depositCommand: DepositCommand): void {
        const { Account, Amount } = depositCommand.payload;
        if (Amount < 0) { throw new Error('Deposit amount must be greater than 0'); }
        this.bankWriteSvc.depositAccount(Account, Amount);
    }

    private transferBankAccount(transferCommand: TransferCommand): void {
        const { From, To, Amount } = transferCommand.payload;
        if (Amount < 0) { throw new Error('Withdraw amount must be greater than 0'); }

        this.bankWriteSvc.transferMoney(From, To, Amount);
    }

    public dispatchCommand(command: any): void {
        try {
            const handler = this.commandRouter[command.constructor.name];

            if (handler) {
                handler(command);
            } else {
                throw new Error(`No handler found for this command ${command.constructor.name}`);
            }

        } catch (e) {
            console.error(`Error occured for handler ${command.constructor.name}`, e);
        }
    }
}
```

Appendix 7: BankWriteController implementation

```
import { JsonController, Post, Param, Body } from 'routing-controllers';
import { Service } from 'typedi';
import { WriteCommandHandler } from '../../bus/command-handler';
import {
    CreateCommand, CreatePayload,
    DepositCommand, DepositPayload, TransferPayload, TransferCommand
} from '../../bus/interface/command.interface';

@Service()
@JsonController()
export class BankWriteController {
    constructor(private handler: WriteCommandHandler) {

    }

    @Post('/account')
    create(@Body() createPayload: CreatePayload): Promise<any> {
        const command = new CreateCommand(createPayload.Name, createPayload.Amount);
        this.handler.dispatchCommand(command);

        return null;
    }

    @Post('/account/deposit/:id')
    deposit(@Param('id') id: string, @Body() depositPayload: DepositPayload): Promise<any> {
        const command = new DepositCommand(id, depositPayload.Amount);
        this.handler.dispatchCommand(command);

        return null;
    }

    @Post('/transfer')
    transfer(@Body() transferPayload: TransferPayload): Promise<any> {
        const command = new TransferCommand(transferPayload.From, transferPayload.To,
            transferPayload.Amount);
        this.handler.dispatchCommand(command);

        return null;
    }
}
```

Appendix 8: BankReadController

```typescript
import { Service } from 'typedi';
import { JsonController, Get, Param, Res, ParamMetadata } from 'routing-controllers';
import { IReadBankAccount } from './../services';
import { BankAccountRepository } from './../services/bank-repository';
import { CurrentBalanceResponse } from './../model/response/current-balance.response';
import { TransactionHistoryResponse } from './../model/projection/transaction-history';

@Service()
@JsonController()
export class BankReadController {
    bankReader: IReadBankAccount;

    constructor(bankRdr: BankAccountRepository) {
        this.bankReader = bankRdr;
    }

    @Get('/account/:id')
    async get(@Param('id') accountId: string): Promise<any> {
        const res = await this.bankReader.getBankCurrentBalanceProjection(accountId);

        return CurrentBalanceResponse.FromCurrentBalance(res);
    }

    @Get('/account/:id/transaction')
    async getTransaction(@Param('id') id: string): Promise<any> {
        const res = await this.bankReader.getBankTransactionProjection(id);

        return TransactionHistoryResponse.FromProjection(res);
    }

}

```

Appendix 9: TransactionHistory implementation

```
export class TransactionHistory extends ProjectionBase implements IAmTransactionHistory {
    readonly History: ITransactionInfo[] = [];
    Name: string;

    constructor(streamId?: any) {
        super(streamId);
        this.WireUpEvents();
    }

    // I only cares about AccountCreated, AccountDeposited, AccountWithdrawed
    private WireUpEvents(): void {
        this.RegisterEvent(AccountCreated, this.accountCreated);
        this.RegisterEvent(AccountDeposited, this.accountDeposited);
        this.RegisterEvent(AccountWithdrawed, this.accountWithdrawed);
    }

    // Appliers
    private accountCreated = (ev: AccountCreated): void => {
        this.Name = ev.name;
        this.addToHistory({
            Type: TransactionType.Deposit, Amount: ev.startBalance,
            Description: 'First transaction'
        });
    }

    private accountDeposited = (ev: AccountDeposited): void => {
        const value = ev.Reason ? ev.Reason.indexOf('Transfer') > -1 ? -ev.Value : ev.Value : ev.Value;
        this.addToHistory({ Type: TransactionType.Deposit, Amount: value, Description: ev.Reason || '' });
    }

    private accountWithdrawed = (ev: AccountWithdrawed): void => {
        const value = ev.Reason ? ev.Reason.indexOf('Transfer') > -1 ? -ev.Value : ev.Value : ev.Value;
        this.addToHistory({ Type: TransactionType.Withdraw, Amount: value, Description: ev.Reason || '' })
    }
    // End of appliers

    private addToHistory(transaction: ITransactionInfo): void {
        this.History.push(transaction);
    }
}
```