final report on empirical research

by Anupam Siwakoti

Submission date: 04-Jun-2022 01:59PM (UTC+0545)
Submission ID: 1846228931
File name: Research_Report_Anupam_Siwakoti.docx (7.95M)
Word count: 13534
Character count: 74490

LEEDS BECKETT UNIVERSITY

The British College
KATHMANDU

Coursework Submission Coversheet
(individual coursework only)

The British College

Faculty of Arts, Environment and Technology

LBU Student Id: C7227267

FOR CHECKING BY THE STUDENT:

Please ensure all information is complete and correct and attach this form securely to the front of your work before posting it in a coursework collection box.

Award name: BSc (Hons) Computing

Module code: 14806

Module name: Production Project

Module run:

Coursework title: **Software Development with Domain Driven Design and related Architecture**

Due Date: JUNE 5th, 2022

Module leader: (In LBU):

Module Supervisor: (In TBC): Resham Bahadur Pun

TURNITIN Checked: YES NO *(please circle)*

Submission date & time: Date: JUNE 5th, 2022 Time: 1 pm

Total Word Count: 13795 **Total Number of Pages (including this front sheet): 107**

In submitting this form with your assignment, you make the following declaration:
I declare, that the coursework submitted is my own work and has not (either in whole or part) been submitted towards the award of any other qualification either at LBU or elsewhere. I have fully attributed/referenced all sources of information used during the completion of my assignment, and I am aware that failure to do so constitutes an assessment offence.

Signed: Date: 2022/06/04

Teacher's Signature: _____ **Date:** _____

I. Abstract

Intent of this research is to have practical experience on concepts of Domain Driven Design and its combination with different other architectures. In programming when we are capable of introductory concepts such as class, objects etc., then there is a need to use this concept with real world problems to have a computing solution. This report understands the importance of this factor and provide a guide map for a fresh-out graduate. Also, we look on how project is setup for microservices, how we apply paradigms of DDD and architecture with frameworks etc., this report encapsulates all the important aspects that a programmer needs for being ready for the job market. As this report includes combined information from the developers over the period, their experience on DDD is captured in this report as I put my hand on their work and their ideology for this research. Combination and differentiation of different architectures when combined with DDD is clearly spotted when we go deeper in the research.

II. Declaration

DECLARATION BY THE CANDIDATE

I the undersigned solemnly declare that the project research report on **Software Development with DDD and related architectures** is based on my own work carried out during the course of our study under the supervision of **Resham Bdr Pun.**

I assert the statements made and conclusions drawn are an outcome of my research work. I further certify that

I. The work contained in the report is original and has been done by me under the general supervision of my supervisor.

II. The work has not been submitted to any other Institution for any other degree/diploma/certificate in this university or any other University of India or abroad.

III. We have followed the guidelines provided by the university in writing the report.

IV. Whenever we have used materials (data, theoretical analysis, and text) from other sources, we have given due credit to them in the text of the report and giving their details in the references.

Anupam Siwakoti
C7227267

III. Table of Contents

IV. List of Figures and Tables

Figures

Tables

V. List of Abbreviations

We have used some of words which are interchangeable with each other, also we have used some shortcuts:

Bounded Context -> context

ES -> Event Sourcing

Domain -> Business Domain

CQRS -> Command Query Responsibility Segregation

Hexagonal Architecture -> Onion (barely used)

DDD -> Domain Driven Design

BBoM -> Big Ball of Mud

Domain Events - > Events

Micro-services -> Services

Persistence layer -> data layer

Also, if you want to reference the research with prototype code, we can find it at, https://github.com/Anupam1223/DDD-prototype.git.

1. Introduction

1.1 - Problem and Motivation

Every software is developed for having computing solution for an ongoing business domain. Business domain may range from astronomical research to general retailing vendor. As a software engineer, we face different scenarios every time when a business owner makes a call as every business has its own problem to be solved. So, there isn't any step wise rule for developing a software, furthermore, to succeed the software must pass through two parameters, time, and memory. Hence, Software Development itself has become very advanced and complex in terms of technology and resources with the advancement of cloud computing and vast amount of data being circulated every second. For developing a quality software, one must also have adequate knowledge on working with external services. External services may include message brokers, background tasks etc., all these information might overwhelm an inexperience developer. Success of any software also relies on its architecture i.e., whether it can be modified to adjust new features that the time brings and does it reflect the core business problem. A clear map on how we will dissect each of the business entity to make them work efficiently keeping in mind that it needs to adjust the changes that the time will bring is a must. Following software architectures that has been proposed and applied by many developers over the time will help us achieve all the factors mentioned above.

1.2 - Research Questions

RQ1: *How is real-time working business incorporated to software using DDD paradigms?* We answer this question by learning about DDD and what experienced engineers have to say about the practices of DDD. We also review software's which are designed using DDD principles.

RQ2: *How is the domain model fused with layered architecture as a monolith, hexagonal architecture as a microservice, and with CQRS and Event Sourcing?* Again, we will learn the practices of layered architecture and apply that with DDD model. We will learn to design microservices using required framework, then we will learn about hexagonal architecture and later fuse the domain model with these two. Detail analysis of CQRS and event sourcing is done and later applied with the DDD model.

RQ3: *What obstacles, benefits and outcomes does these architectures provides in comparison with each other?* When we finally implement all these architectures, we will have detailed discussions about how we overcame the obstacles and what may be the benefits of these architectures.

1.3 - Expectation from the research

This research paper focuses on a widely implemented and celebrated architecture, i.e., domain driven design which was first proposed by Eric Evans in his book '*Domain-Driven Design: Tackling Complexity in the Heart of Software'*. We can use the principles of DDD with various other software patterns, among them Layered architecture, Onion architecture (clean architecture or hexagonal architecture), CQRS and event sourcing will be discussed and demonstrated in this research. Well, all these architectures and patterns were proposed, validated and is being in use for many years, so this research paper will not briefly discuss on what these architectures are, rather we will combine them with each other, i.e., a prototype will be developed for each combination and later be compared with one another. This will provide base for any inexperience developer to have practical introduction to different software architectures, their combination, and their differences.

1.4 - Research Process

Plenty of books and articles are written on the principles of Domain Driven Design, lots of software's have been designed using DDD. These are the main area for reviewing idea for the research. There is no data collection involved as this is quantitative based research. We will have a prototype design which we will follow along with the research defining each paradigm, it starts with defining DDD paradigms to fusing the paradigms with stated architectures.

1.5 - Prototype Domain: Homrent

Throughout the paper we will follow a common business domain to demonstrate principles of DDD, and its combination with layered architecture, hexagonal architecture and CQRS and Event sourcing. Homrent is a flat renting company that rents flat to the tenant. It has an owner that has its profile with details of housing, whenever a house is booked it should be removed from available houses, management of the booked house

must be done. When we research about DDD, we will continue to apply those DDD principle with Homrent business domain.

2. Background Work and Literature Review

Cosmic Python is hosted at https://github.com/cosmicpython/book, author of this book Harry Percival and Bob Gregory has discussed all kind of possible patterns that are used with DDD principles and hexagonal architecture. When I first read the book not even a single concept made it to me because it's very complex for a fresh out graduate like me to understand practices that has been over two decades. It talks about repository pattern, unit of work pattern, dependency inversion, Aggregates etc. Without experience it is as close to impossible to grasp this knowledge. I read blogs related to CQRS and event sourcing, onion architecture etc. and found out that even though they have been described properly with words but lack in providing practical example that can be easily executed and tested. Saying that, cosmic python does have a prototype to follow, and it is open sourced, have tested modules and follow TDD (test driven development).

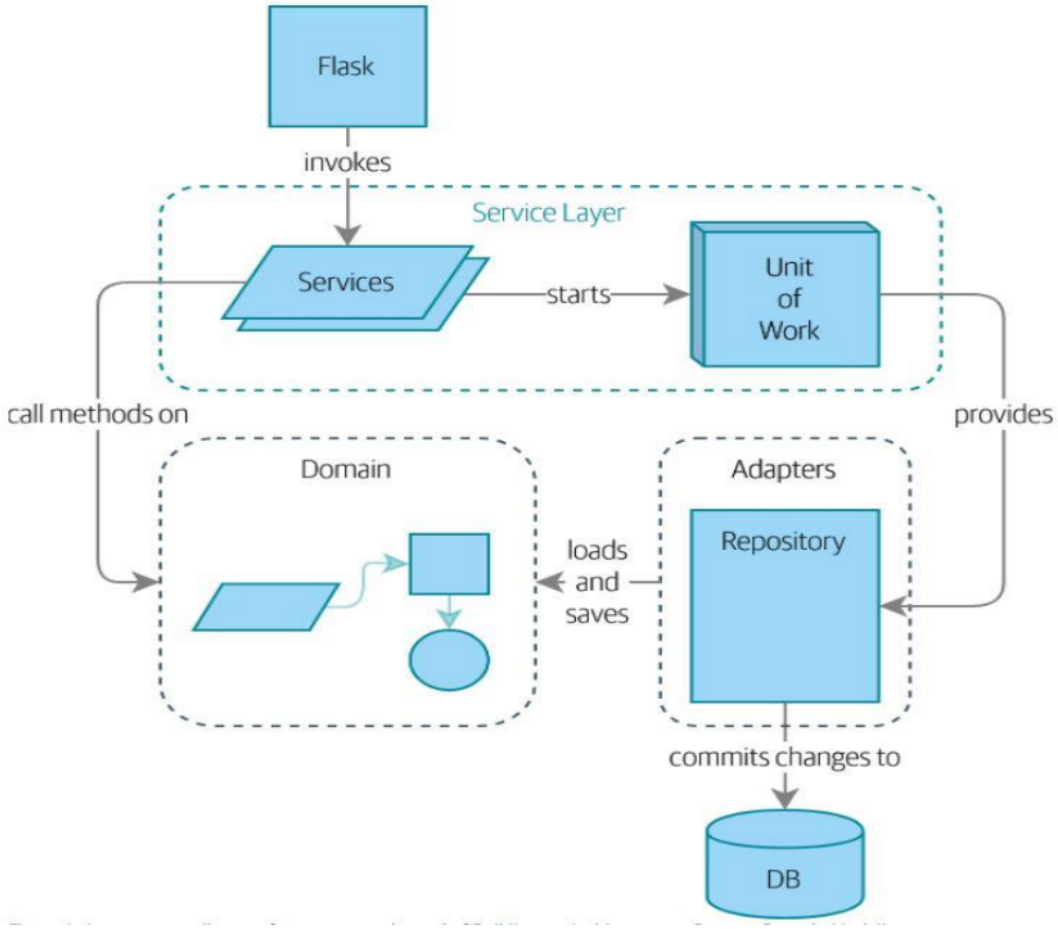


Fig1.1: pattern used in cosmic python

This book revolved around the figure given above, we can see many of individual patterns are used together with DDD.

One of the blogs was, In Search for a Scalable & Reactive Architecture of a Cloud Application: CQRS and Event Sourcing Case Study (Debski, Szczepanik, Malawski from AGH University of Science and Technology, Poland and Spahr, Muthig from Lufthansa Systems, Germany). This paper focuses on the effect of CQRS and its applicability using akka tools for developing a prototype, interactive flight scheduling application from Lufthansa Systems (LSY). Throughout the report the main agenda is to focus on its

reactive manifesto, the need for building reactive and responsive system. It concludes with the note that CQRS + ES is the architectures that brings resiliency and flexibility. It also includes a concept that help in scalable performance when executing commands in write model i.e., sharding, method for load balancing of command execution.

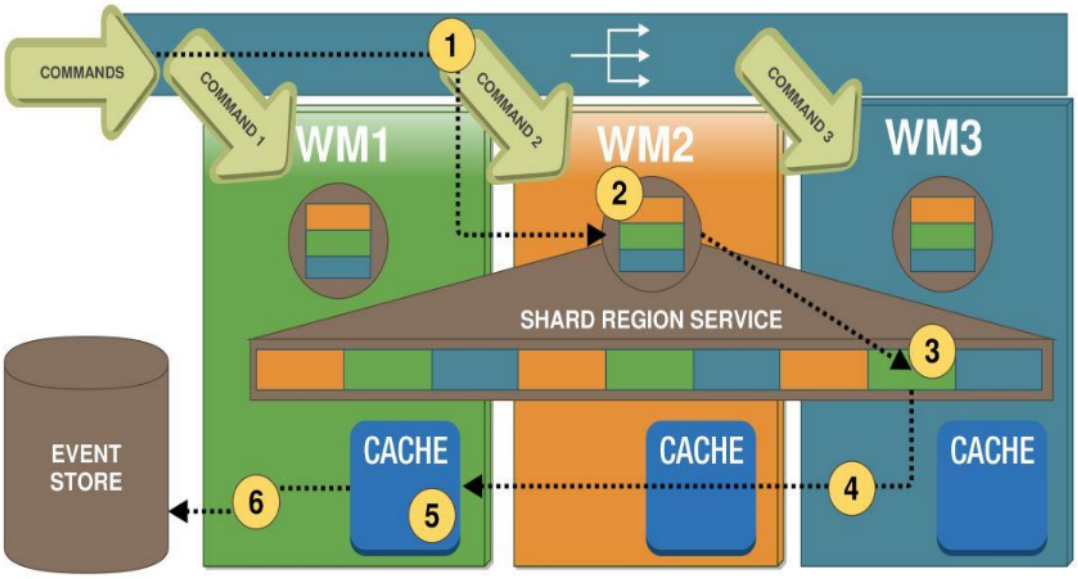


Fig1.2: sharding concept discussed in the report discussed above

Patterns, Principles, and practices of Domain-Driven Design (Millet with Tune, 2015) also have stated multiple principles on how we can implement DDD with different architectures such as event sourcing, CQRS. This book gives the detailed description on these patterns but lacks in giving simpler practical explanations.

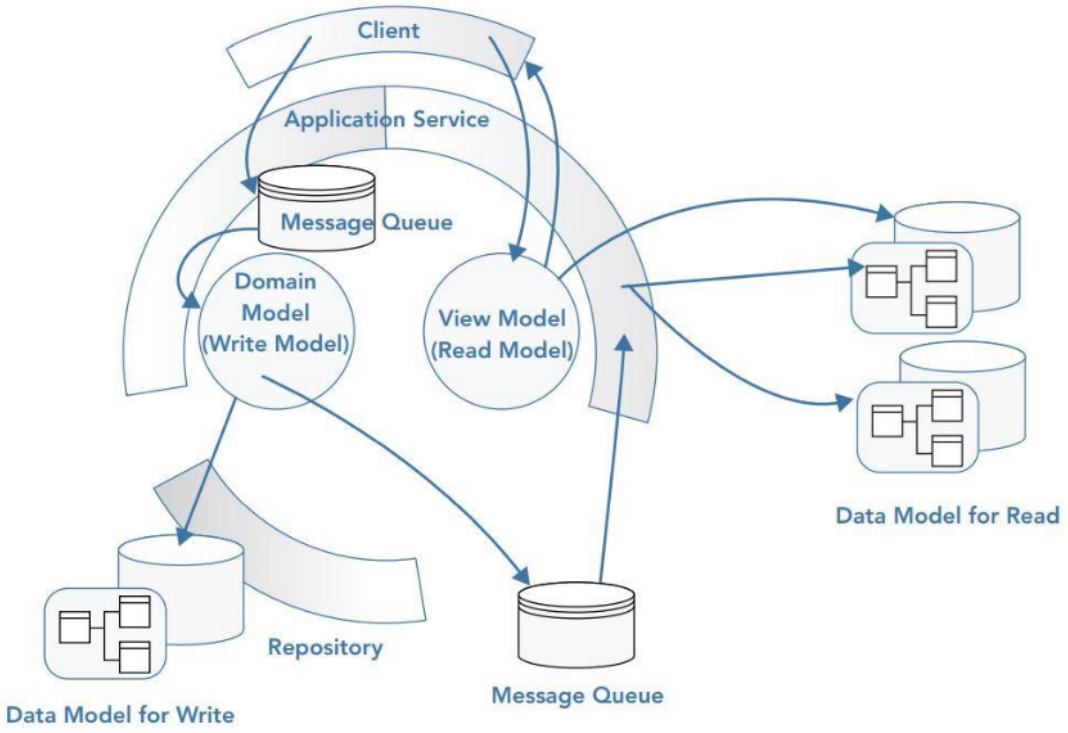


Fig1.3: Concept about implementing CQRS talked in the book above

Learning Domain-Driven Design, Aligning Software Architecture, and Business Strategy (Khononov, 2022), Again this book also has some great explanation on understanding the principles of DDD and how can we apply those principles with our problem, but it doesn't state things that a fresh out graduate need. A flow of explaining details on how things are working is missing. Every basic concept such as Transactional record or active record has been discussed properly and a single pattern i.e., hexagonal pattern Is used throughout for the explanation, but it lacks the comparison between architecture, a simple pattern is not enough to understand the exact structures behind building a software. This book provides theoretical based concepts on each detailed paradigm of DDD, such as in part I chapter 4 it talks about integrating bounded context and its methods (customer-supply, cooperation), talks about how these integration method helps in achieving context map for physical design but lacks the practical explanation of such concepts i.e., no code is provided as a reference.

Architecture followed in voice communication application, Krispcall, As I was one of the engineers that was working for developing backend of this application, it gained me enrich experience on how big enterprises application are made. This application is big inspiration behind this research, link for the website is https://krispcall.com. Back-end portion of Krispcall strictly follows the paradigms of Domain Driven Design. It has very advanced architecture, follows messaging protocol, python as programming language, GraphQL, Rest APIs for building API's, Docker for wrapping up the application. Patterns such as Adaptors, factory, unit of work, repository are used but we are interested in the domain driven paradigms. Every bounded context has a single domain model, i.e., organization bounded context has its own domain, and every service, commands, events work around this model. Now since our research is not about the communication protocols, I am referencing this application because it contains some of the architectures that we are working upon. I gathered information from this application regarding the integration of endpoint, how to build a domain model with abstraction over aggregate.

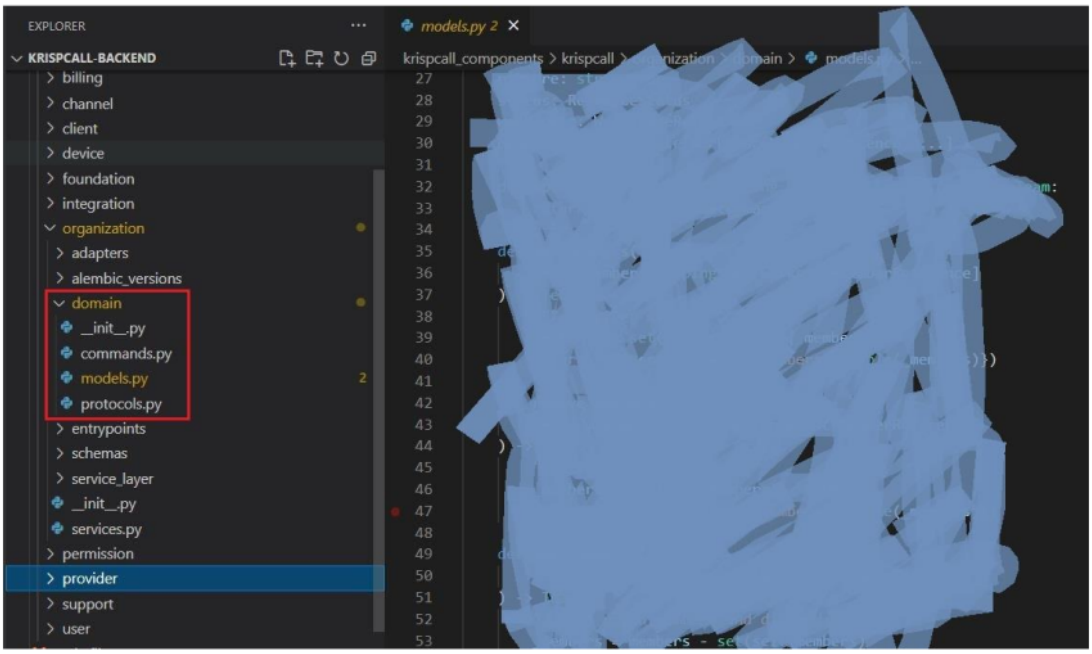


*Fig*1.4*: piece of code from krispcall*

2.1 - Review of technologies

2.1.1 - Spring Framework

Spring framework is one of the leading java tool kits and popular among java enterprise applications. It covers almost every aspect required for developing an enterprise application. Spring framework compromises of technologies, such as Aspect-Oriented Programming (AOP), Dependency injection (DI), cloud dependency, in-memory database, plain old java object (POJO) which is very efficient for rapid developed of java projects. Also managing dependency and further extension of external technology is way to simpler. Spring framework itself is huge and provides a portfolio of projects,

- Core infrastructure projects, consist of libraries that support foundation for building spring-based application
- Cloud-native projects, for developing microservices that has cloud-native capabilities
- Data management project, contains libraries with interfaces that robust the work related to data persistence

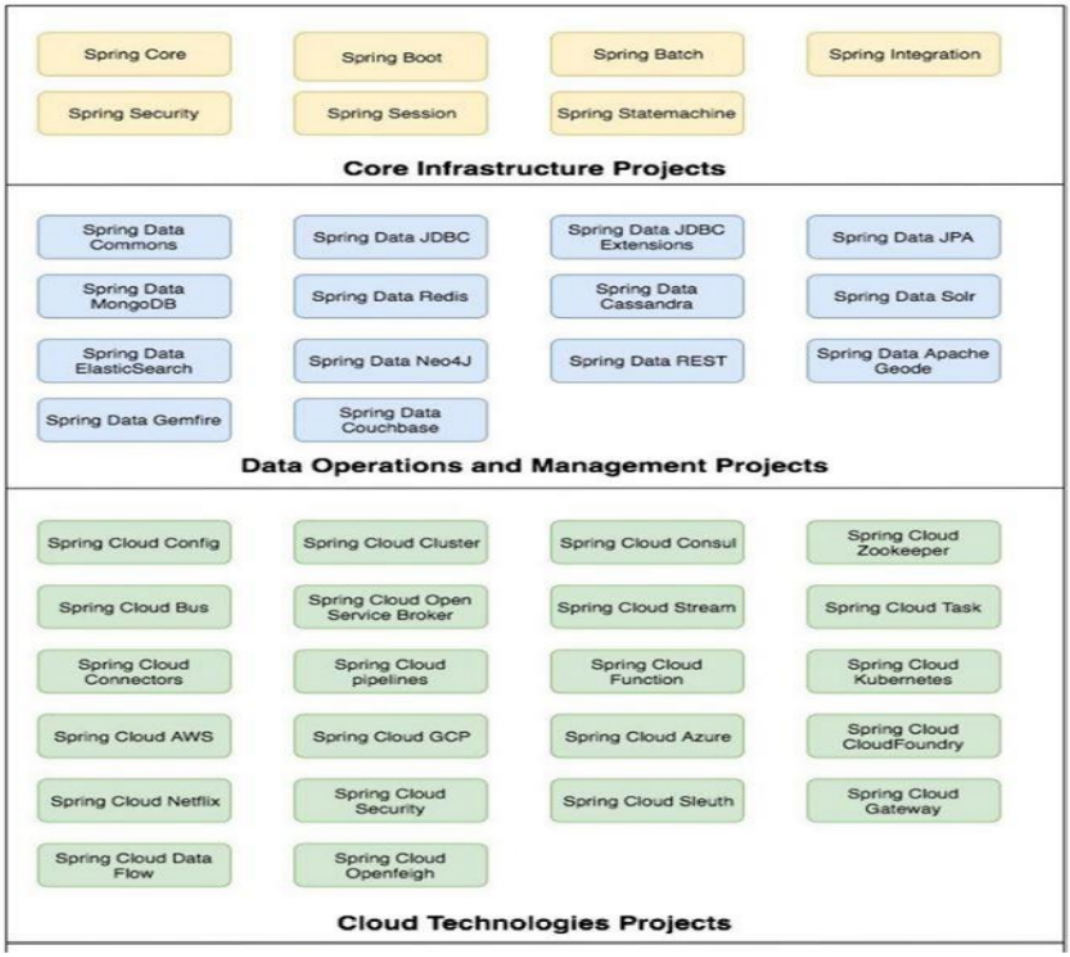


Fig2.1: portfolios included in spring framework

We are just going to use subset of libraries, spring boot, spring data JPA, spring cloud stream etc.

Spring Boot is the foundational base for any spring based microservices application, it helps in developing Rest Interface, Data, and messaging functionalities by implementing abstraction due to which developers don't need to worry about underlying information for achieving rest interface. For initial establishment of the application, spring boot provides starter kit which help in exposing API's, configuring messaging brokers and connect with underlying database, some of the starter kits used with the prototype are, spring-boot-starter-web, spring-boot-starter-data-jpa, and spring-cloud-starter-stream-rabbit.

2.1.2 - Axon Framework

It is an extension over spring framework, the primes reason behind developing it is to support developer in developing DDD with CQRS + ES patterns. In this library it contains the description for basic blocks of DDD such as Aggregates, Command gateways, Event Handlers, Command Handlers, Sagas etc. Later in the research when we are in the phase of implementing CQRS we will discuss about it. The strongest point of this framework is that it provides annotation support such as @Aggregate for defining aggregate, this simple word and the framework is ready to initialize the lifecycle of an aggregate. Axon framework provides an appropriate UI where we can see the running of application, events, and commands.

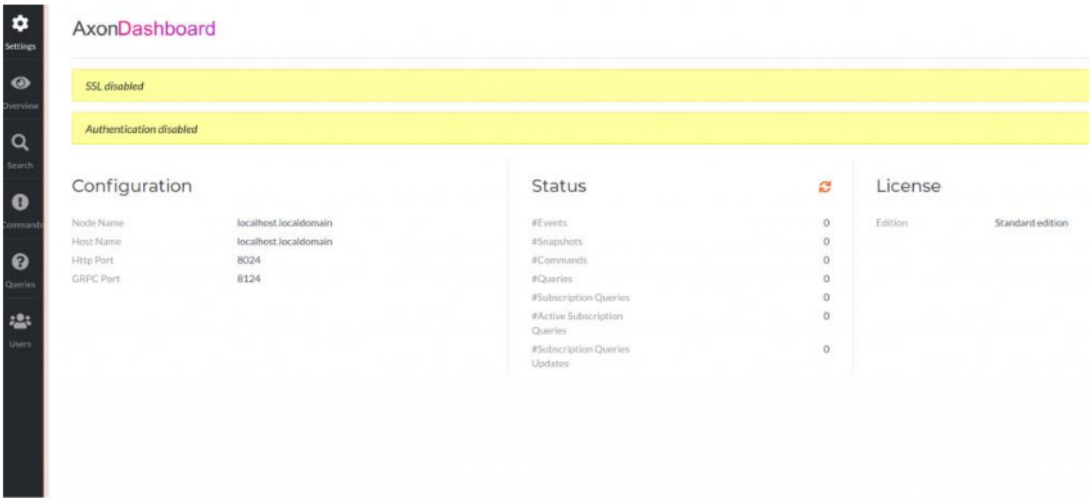


Fig2.2: Dashboard of axon framework

2.1.3 - Postman

The most useful tool for checking the endpoints as we don't have UI for our API. Any form of information such as raw text, JSON objects, xml file can be sent as a request body. All the advanced framework such as FastAPI, Django, Flask, Spring can use this for checking the API's. It also has the power of checking the authorization and authentication of the endpoint by decoding the request. It has the capability of listening almost all type of HTTP requests (PUT, POST, PATCH, GET). Even form data can be sent using this technology.

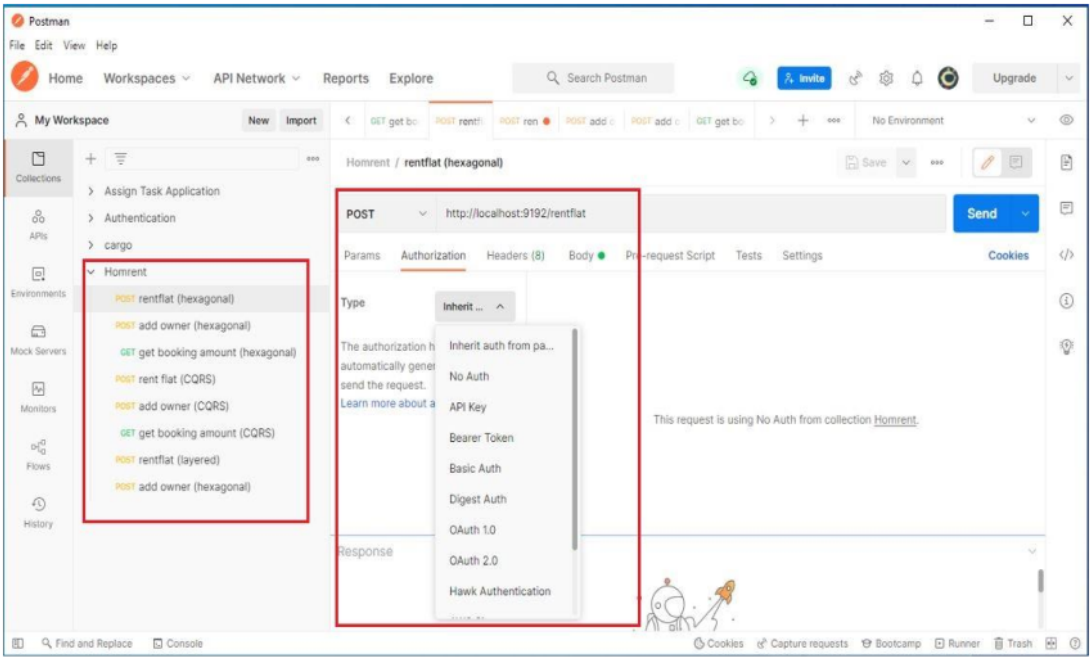


Fig2.3: postman showcasing endpoints

We can see all the functionality that we can use with postman while testing endpoints, there are params, authorization, headers, body (where we send our JSON payload).

2.1.4 - RabbitMQ

As stated earlier, it is a messaging-queuing technology that follows Advanced Message Queuing Protocol (AMQP). Let's see how we configured it with our demo project, Homrent. First, add RabbitMQ with relevant version in pom as a dependency. It is used as a bridge between two microservices for message communications.

In-order to use RabbitMQ first you must download it, secondly start the service.

For starting service,

- Open RabbitMQ terminal from start after installing it
- Use **rabbitmq-plugins enable rabbitmq_management** command to enable it

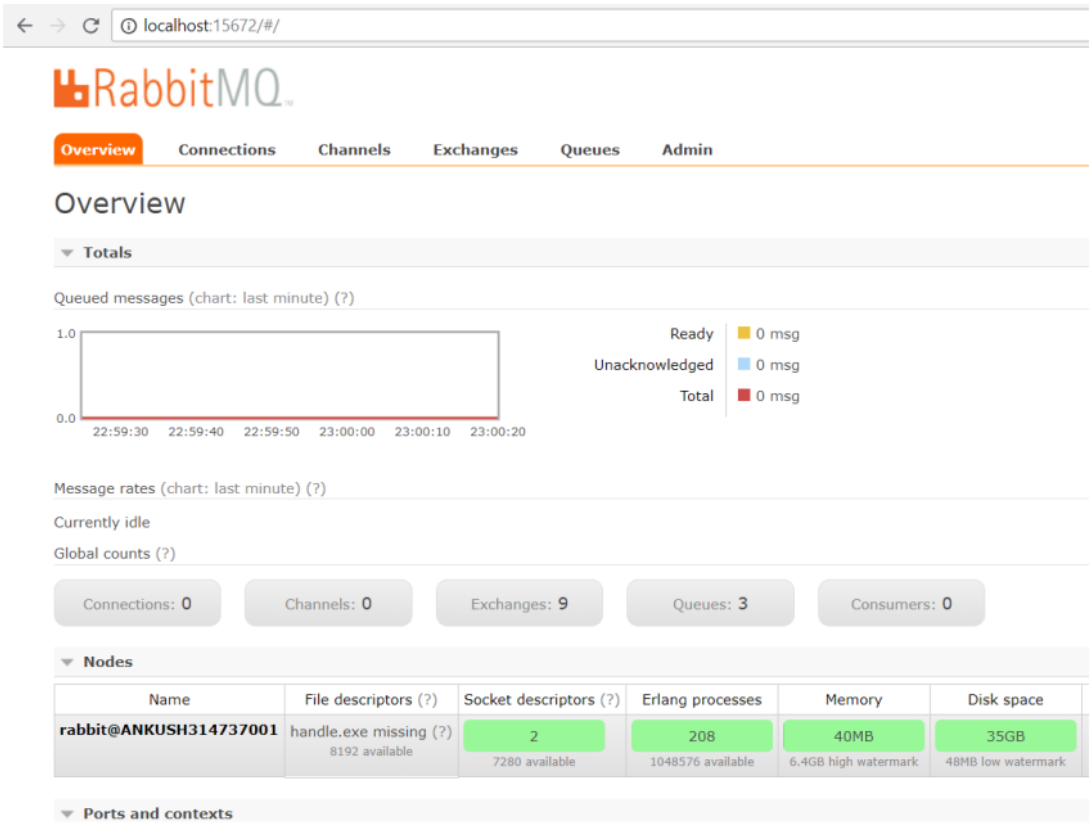


Fig2.4: GUI of RabbitMQ

We can see there are connections, channels, exchanges, queues, and admin in the menu, connections say the connection with the application, exchanges describe the attachment between two applications. Queues are the actual queue, there may be named queue of commands, events etc.

3. Methods

3.1 – Introduction

In this section we will discuss about how the research proceeded from searching the materials required to setting up the environment. This research being **quantitative** we needed previously written books, blogs, articles, research, and recorded interviews, talks etc. Figuring out the correct framework and language was quite challenging but through several YouTube videos I figured that java language has perfect pool of framework that is specially developed for implementing principles of Domain Driven Design. Postman was recommended by a blog which contained FastAPI content in it.

3.2 – Research Setting

For the coding environment I have used VS code IDE, one of the most powerful IDE that has extension for almost all programming language and their frameworks. It is such powerful that we can even run WSL2 (Linux environment) in it, even extension for including docker files and wrapping up with Kubernetes is very easy, implementation and logging of background tasks is way easier. I searched up for general Java package,

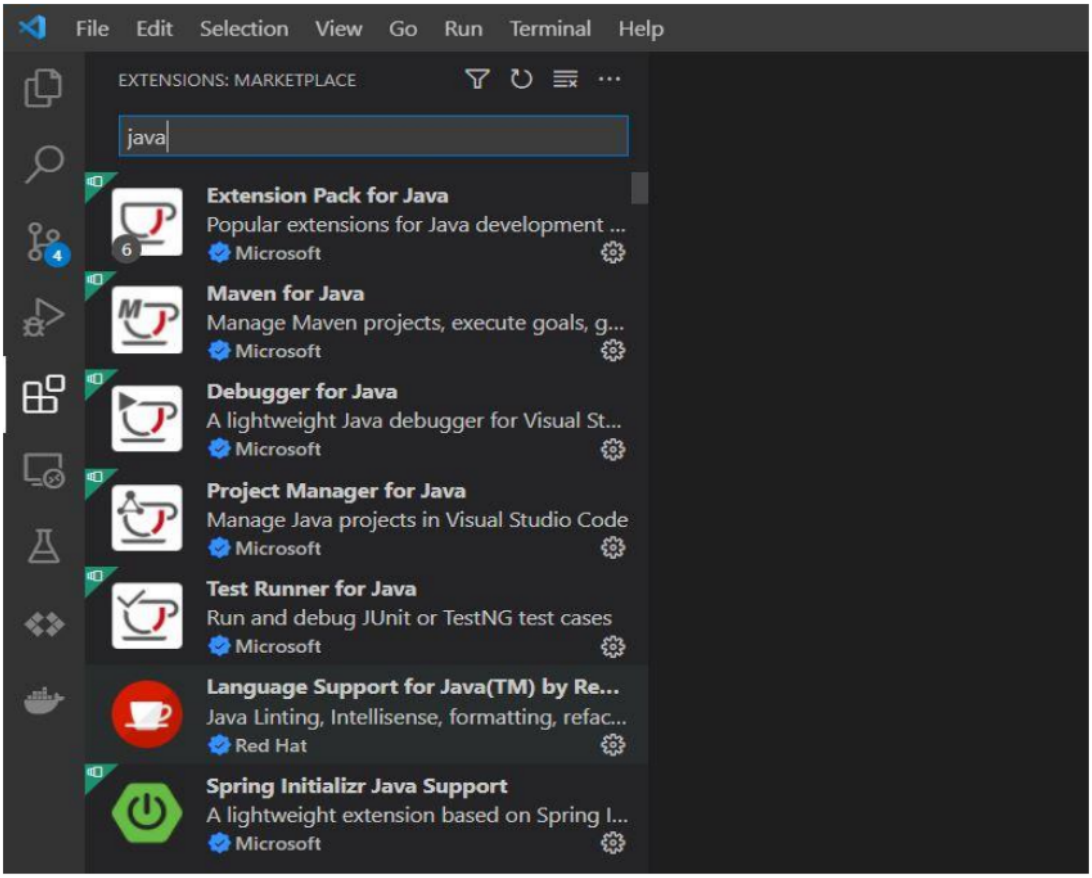


Fig3.1: Extensions of java installed for supporting the research

In the figure, there is an extension pack which includes general purpose libraries required for general purposes, but that's not enough for our project so we can see spring java support which is what required for us. For embedding axon framework, we just need to add dependency in pom.xml file of spring project.

Now that we have environment setup, we also need to have some idea on initializing the spring project. I went to official documentation of spring framework, and it suggested a link where we can add dependencies on our demand and it's very easy.

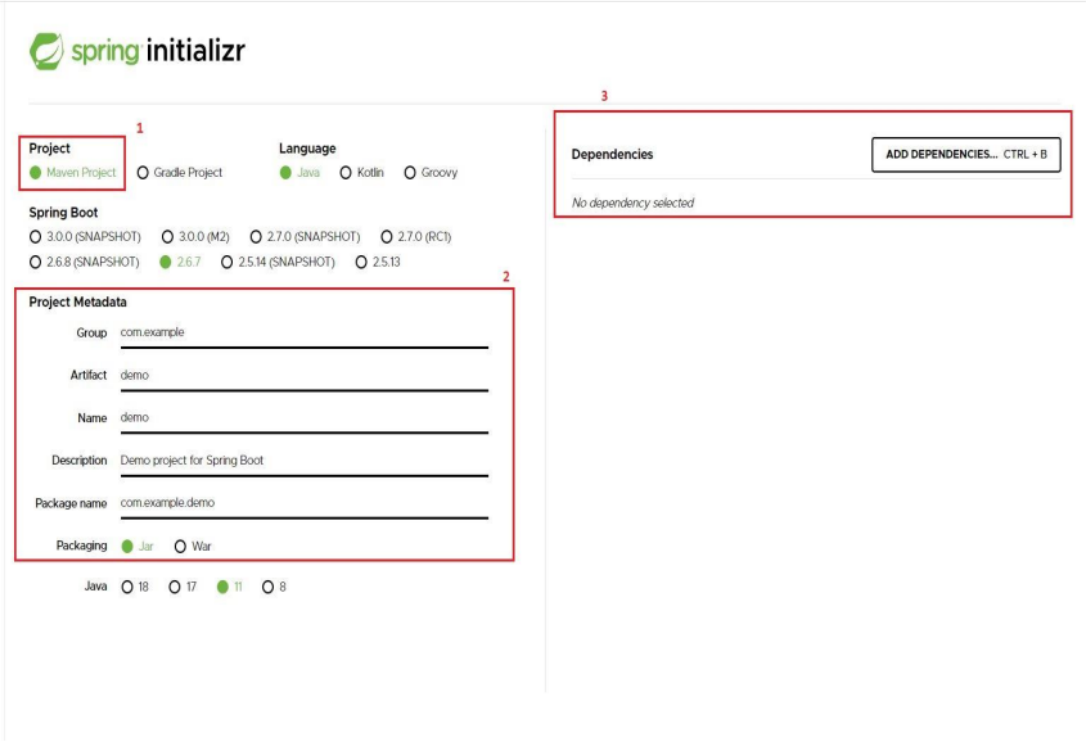


Fig3.2: initializing spring project

First, we choose the type of project management tool, two choices maven and Gradle project. I have used maven project because it follows POM (Project Object Model). Second, we setup what we want our project name to be, in microservices we will have different group for each bounded context. Third, we now select the dependencies we need, **Spring Data JPA** is one of the examples.

Now we have structure for developing our project, for testing I setup local environment. I will explain it in steps

- Go to resources and create a file, application.yml

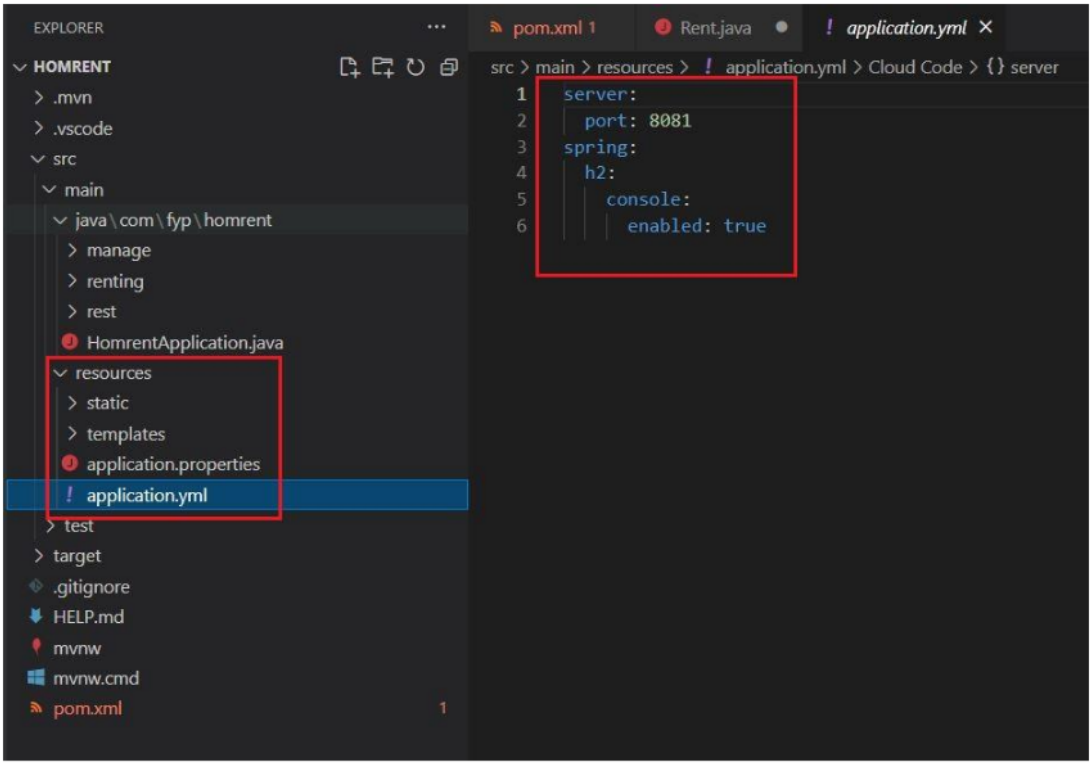


Fig3.3: setting local environent for prototype

- Go to your main application file, HomrentApplication.java is the main file for this project and run the project, for running you can just click play button which is at the top right corner of vs code IDE

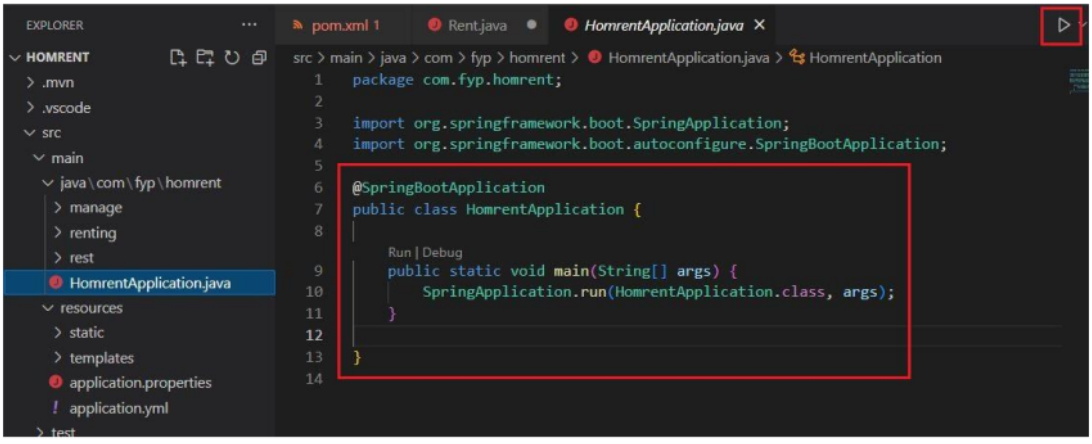


Fig3.4: Running Spring project

If terminal suggest to visit localhost:8081 then the project is sucessfully installed.

3.3 – Research Process

Now that we have set the framework its ready to gradually apply the findings from our research. As we have discussed that we are applying DDD principles with layered, hexagonal and CQRS with event sourcing.

When developing the prototype, we follow the principles by learning them from various sources (already discussed). Once all the paradigms of domain driven design are figured out then we gradually apply them with these three architectures. In section 4 we cover this.

3.4 – Limitations

Our prototype has a very simple problem domain because it's just a walk through for this research. Because of the time limitations there are not much functionality. It's both good and bad as the one who is just hooked with the concepts of patterns it would be helpful for them. This prototype demo is not hosted over internet, we need to setup local environments for it to run but it's quite easy if our framework setup is accurate.

RabbitMQ is considered as traditional tool for messaging queues, Kafka is quite popular but since integration of RabbitMQ with spring framework is quite popular, so I used it, working of RabbitMQ is not that bad. Generally, when Implementing event stores, developers use external services but since we have very minimal implementation, we use inbuilt event stores provided by axon framework.

4. Prototype development

4.1 - Domain Driven Design

"Domain-Driven Design (DDD) is a process that aligns your code with the reality of your problem domain" (millet and tune, 2015). DDD not only acknowledges the need of pattern, framework, methodologies, principles but also emphasizes on communication between domain expert and developers working together to build software that correctly reflects the business. With implementing DDD constraints we can develop software that can describe the business itself, conveys the exact business problem and adapt easily with the future enhancement. Basically, DDD describes different approaches, tools and techniques which dissects business complexity and still manages to focus on the core business idea.

To feel the positive changes brought by DDD we need to understand the hurdles behind maintaining and developing a software. There is a term BBoM, described by Brian Foote and Joseph Yoder in the paper "Big Ball of Mud," which suggests that when a software works fine but can't answer how its working then we will end up developing a software where it will be difficult to read and understand the codebase. This often happens when business logic complexities are mixed with technical complexities. Technical complexities may include database connection, API integration, Broker integration and so on.

To prevent ourselves form ending up with BBoM we must have clear distinction between business model and external technical complexities. DDD is thus divided into two parts, strategical design, and tactical design. Strategical design includes analysis of business domain, dissecting it into multiple subdomains and mapping these different subdomains to provide intended business solution. Let's look at each of the DDD constraints that falls under strategic design.

4.2 - Implementing DDD constraints to our prototype

When we talk about any software architecture then it obviously has its principles on aligning different aspect of problem, how will it be modeled and so on, let us shortly review them and start implementing it with our reference business i.e., Homrent

4.2.1 - Ubiquitous language

This is the starting phase of implementing DDD, where software engineers start to ask questions about the business to the business expert. The product (software) should exactly express the business idea which sometimes isn't documented or codified but resides inside the mind of business expert. The only way to achieve this is by having enough conversation with the business expert. When having conversation, we start to hear some words and phrase which help in defining the exact intent of business, these words or phrases are highlighted by the consent of both expert and developer and later used while writing the code. By doing so gap between the non-technical expert and developer is reduced. Ubiquitous language is ever evolving which is much clearly expressed when we start to dissect bounded context. Let's take the example form our reference domain Homrent: conversation between business expert and developer

Business expert: "when flat is rented, it should be out of the available list"

Developer: "oh! So, it raises **flat Rented** event"

Hence the phrase **flat rented** is now will be used by both (business expert and developer), to understand the cause and effect of renting the flat.

Later when we learn about bounded context, we will be more familiar with how ubiquitous language plays an important role in designing software that speaks for itself.

4.2.2 - Business Domain

Solving any kind of problem needs thorough understanding of the problem. As an engineer we need to correctly dissect each part of the business domain: its terminology, the exact reason behind developing the software. The services that the business promise to provide its customer. To address this, DDD can be divided into two parts, strategic and tactical design. Strategic Design includes deep understanding of the business, to which we call tunneling the brain of the business expert. In this stage, developer will have multiple meetings with domain expert to decompose larger business domain to smaller subdomains.

Subdomains

Generally, a subdomain has its own individuality of providing solution. Such dissecting of the problem will saturate core business domain from the generic and supporting one. Now, what's core and how does it differ from supporting and generic subdomain? Let's visualize this with a table

- **Core subdomain**: The main motive behind building a software, or the problems that the software will solve to generate revenue. For example, google is famous for its search algorithms due to which it stands out from its competitors, that's why its core subdomain of google. As our prototype domain, Homrent is a minimal implementation, our core domain is to rent the flat.
- **Generic subdomain**: Subdomains which are a bit tough to work through and needs qualified human resource but doesn't provide any competitive edge is termed as generic subdomain. They are widely accepted and used by all the companies, for example there is a concept called web-socket, which is used by every companies now a days for implementing background tasking. This kind of extensions falls under generic subdomain. In our prototype, we have implemented RabbitMQ as an extension while taking our project from layered architectures to microservices.
- **Supporting subdomain**: out of three subdomains these are the simpler one, as name suggests it supports the other subdomains to work well. We need services such as crud operations, which is very basic and is very important for any software to be success.

Now that we learned about all three types of subdomains, we need to understand about the bounded context.

4.2.3 - Bounded Context

Programming module under which related subdomains (sometimes single) are included to solve a certain part of the business. Bounded context helps to segregate ubiquitous language in a context. For example, in our prototype we have a term called owner, let's see this as a conversation

Business expert: Well, current **owner** can leave before the lease time, but he will not have refund.

Developer: by current owner you mean tenant? oh! actual **owner** is the one who owns.

Here we have two owners, when we design bounded context, both will have their own context i.e., current owner as tenant in renting bounded context and actual owner in manage bounded context. Modules shouldn't be bigger with that said it shouldn't be too small either rather it should be useful.

Our core sub-domain was to rent a flat, hence we will have renting as a different bounded context. We have another subdomain on which we will work, i.e., manage hence it's another bounded context. We can have a single bounded context for both which we will use in a monolith structure, but for microservices we will use them as a separate bounded context which will be deployed in the cloud and communicate through message brokers. Figure 1.1 explains how we design actual problem(sub-domains) to bounded context (solution space).

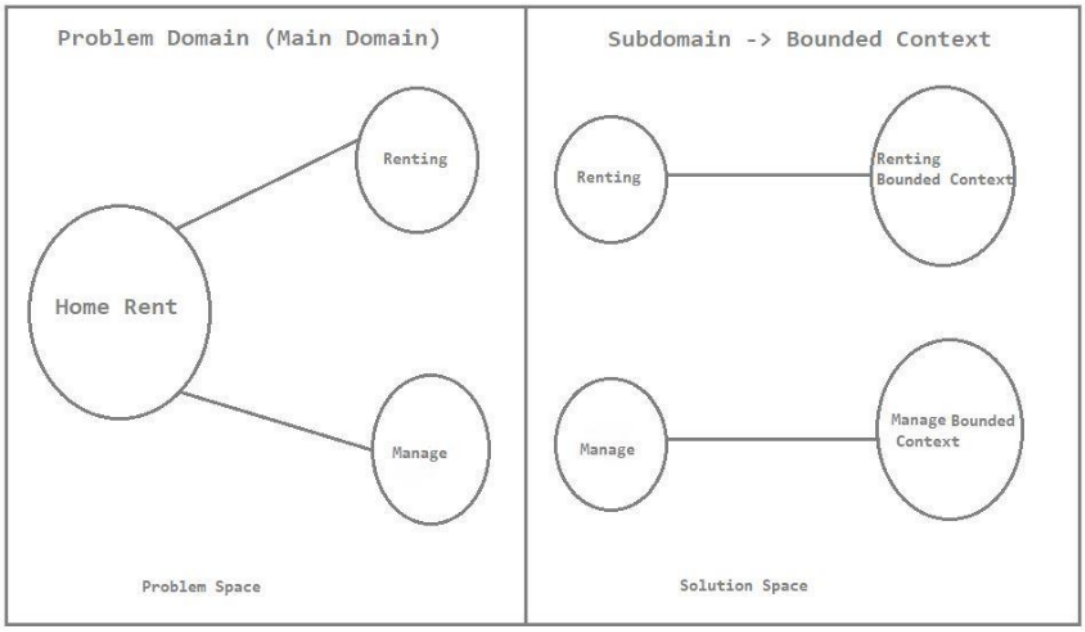


Fig 4.1: Homrent subdomains solutioned as separate bounded context

Let's revise, from beginning we've been discussing about domains, renting the home is the main purpose of our system, we figured out renting and managing owner of the house

were two branched problems. In problem space we figured that out now in solution space we designed physical map or context on how our modules will be built.

As discussed earlier, first we deploy as a monolith architecture where each bounded context is deployed as a single web archive but for other two (CQRS and hexagonal) we will deploy each bounded context as a different unit. Each bounded context is considered as a separate entity, which is developed, maintained, and deployed as a separate unit. Such several bounded contexts need to communicate with each other to solve domain problem. DDD talks about various communication techniques between bounded contexts. It needs thorough meetings with developers who are in control of these bounded context to stick in a common ground on which technique to be applied, Abundant understanding is necessary because both sides are involved.

Integrating bounded contexts

As discussed earlier, there is a need of integrating bounded contexts for efficient communication and smooth working of the application.

Cooperation

Simplest out of them, where just good communication between both side is enough.

Partnership

It is the simplest form of cooperation where the communication is in ad hoc manner i.e., with every change in one of the contexts the other team is notified, and both adapt to this change on common ground. Frequent communication, discussion is a must, no one team is in upper hand when deciding the language. In our prototype, the team working on Rent will communicate with other team i.e., manage every time there is a change in the API.

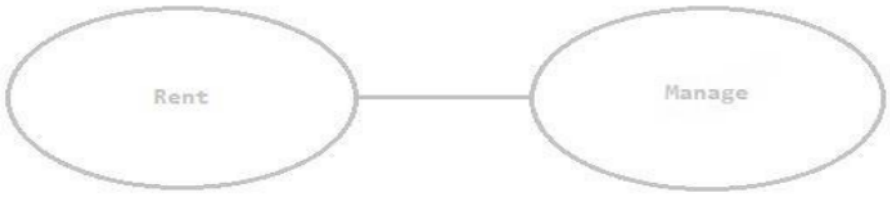


Fig4.2: partnership between rent and manage bounded context

Integration issues, deployment issues or may be communication issues are solved with frequent meetings. In our prototype, whenever any change is made in rent that may affect the owner bounded context, a meeting is held to negotiate in a common term.

Shared Kernel

Even though we separate context based on the use case of domain as bounded context, there is chance where some of the content may be similar in multiple bounded contexts. In such case, we place the common content as shared domain. Here, we should take care of the consistency of the shared model in all the bounded context. In our prototype, we have used shared domain between Rent and Manage. Let's see an example, we will discuss about all of this gradually.

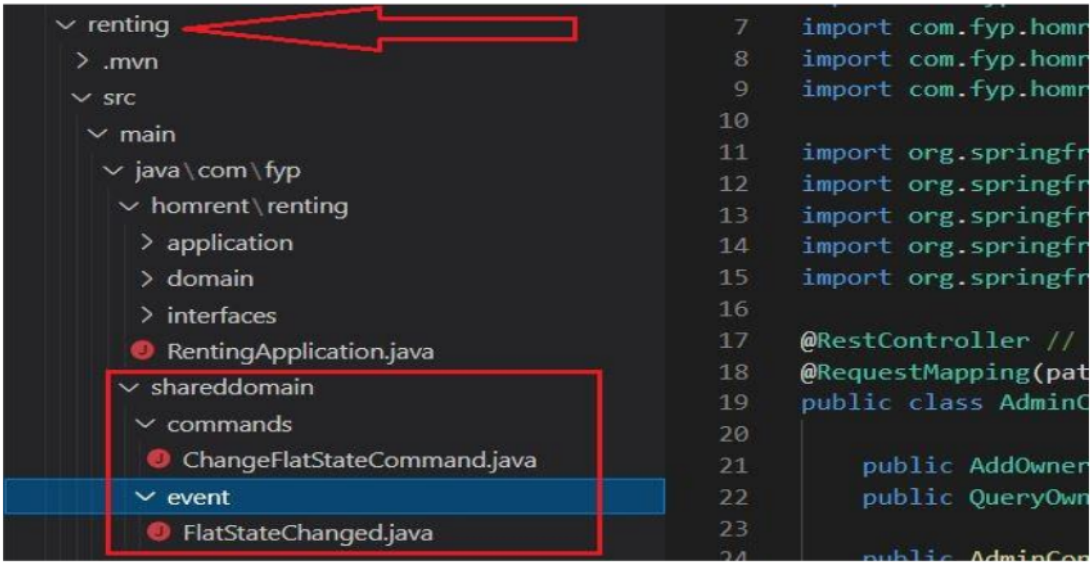


Fig4.3: example of shared domain module in renting bounded context

Don't stress about the content given above, we are yet to discover all those, just focus on the model shared domain. Commands and event included here is present in Manage bounded context, both teams came in a common term and created a module which is frequently used by both.

4.2.4 - The Domain Model

Domain model, as name suggests we model the problem as it is discussed in the earlier phase i.e., strategic design. Traditional CRUD operation is not the answer, we deal with

complex business rules, state transitions and invariants: rule that should not be crossed. Domain model should be modeled such that it can manage the business complexity ensuring clarity of what the modules has to offer. We follow object-oriented paradigms to define each of our domain models. In short, the domain model is the actual implementation of business logic inside a specific bounded context.

In our prototype, we have two bounded contexts, renting and manage. Each will have a model which includes the actual business logic. To achieve that we use the concepts of domain model, let's look at them individually.

Aggregate

In any regular CRUD operation, we generally only consider the consistency of a particular class, for example in a CRUD operation related to an organization, employee class maintains consistency of its own. But in any huge business with complex functionality, it doesn't work that way, lots of classes should maintain consistence with each other. **Aggregate acts as an encapsulation around value objects and entities enforcing transactional consistency between them**. In case of our prototype: Homrent, in renting bounded context, we need to maintain state consistency with booking state, about tenants. Similarly, in manage bounded context, owner needs to have consistency with the flat him/her offers.

Aggregate is considered as central object that helps in maintaining consistency, everything in a domain model starts and ends with it. Say, if a person wants to rent a house, it starts with rent aggregate where current state of the house is present. Doing so we get the booking state, state of the tenant, rented info etc.

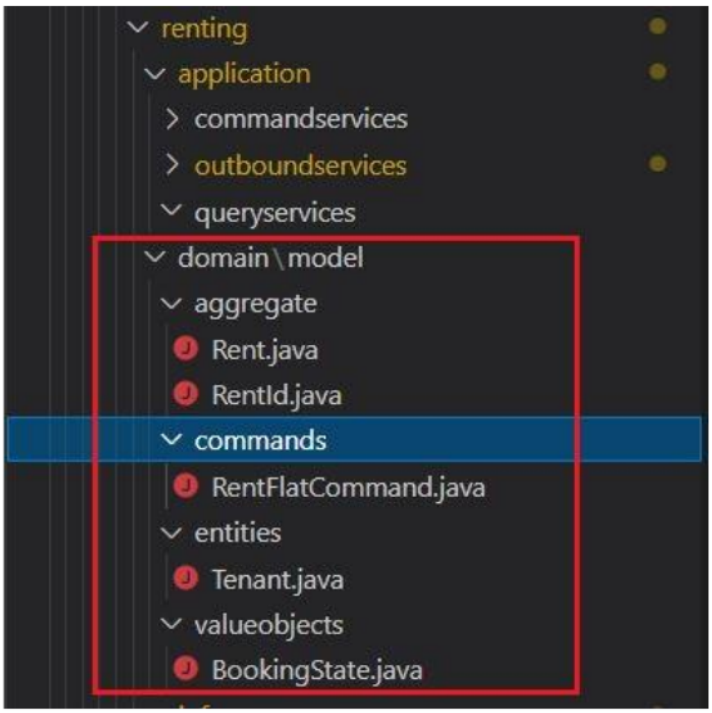
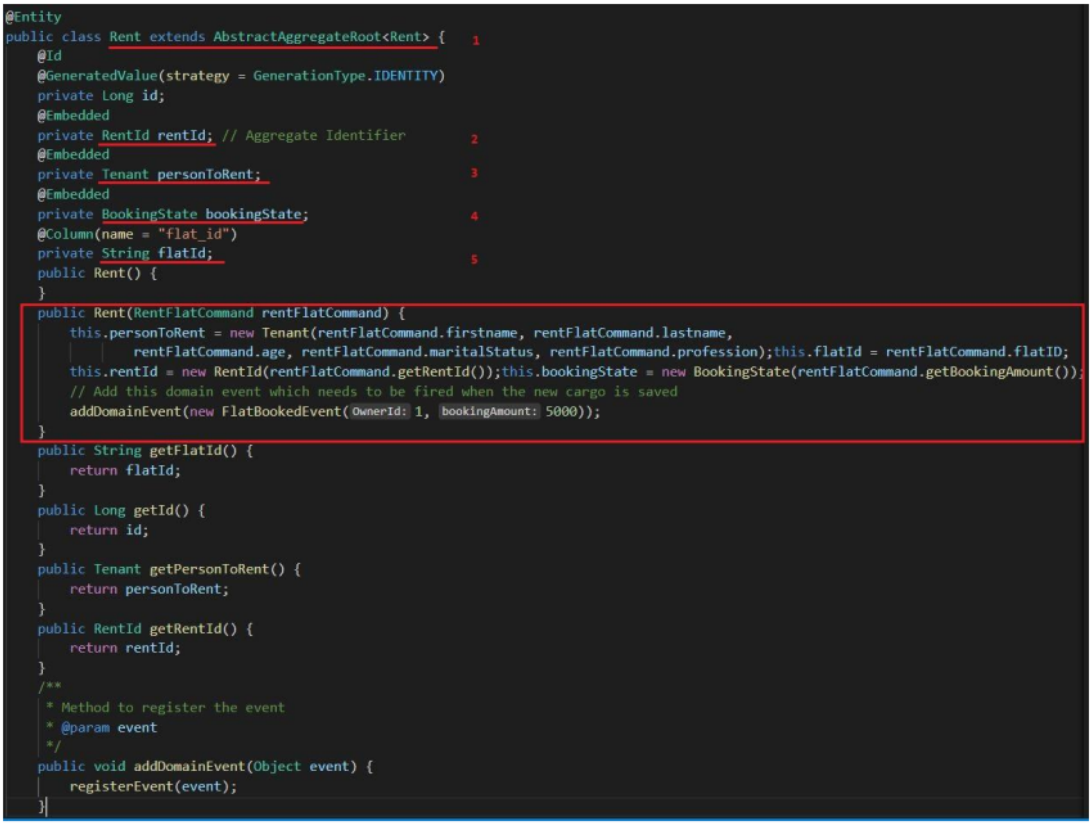


Fig4.4: Implemented domain model

In the figure above, we can see Rent and RentId inside aggregate module. **RentId is aggregate root**, used while referencing other aggregates. Say for example, we need to communicate between renting and manage bounded context, aggregate is what represents a bounded context, so manage bounded contexts will use RentId for communication. By doing so we will just keep the reference of aggregate rather than instantiating object of one aggregate in another.



```java
@Entity
public class Rent extends AbstractAggregateRoot<Rent> {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Embedded
    private RentId rentId; // Aggregate Identifier
    @Embedded
    private Tenant personToRent;
    @Embedded
    private BookingState bookingState;
    @Column(name = "flat_id")
    private String flatId;
    public Rent() {
    }
    public Rent(RentFlatCommand rentFlatCommand) {
        this.personToRent = new Tenant(rentFlatCommand.firstname, rentFlatCommand.lastname,
                rentFlatCommand.age, rentFlatCommand.maritalStatus, rentFlatCommand.profession);this.flatId = rentFlatCommand.flatID;
        this.rentId = new RentId(rentFlatCommand.getRentId());this.bookingState = new BookingState(rentFlatCommand.getBookingAmount());
        // Add this domain event which needs to be fired when the new cargo is saved
        addDomainEvent(new FlatBookedEvent( OwnerId: 1, bookingAmount: 5000));
    }
    public String getFlatId() {
        return flatId;
    }
    public Long getId() {
        return id;
    }
    public Tenant getPersonToRent() {
        return personToRent;
    }
    public RentId getRentId() {
        return rentId;
    }
    /**
     * Method to register the event
     * @param event
     */
    public void addDomainEvent(Object event) {
        registerEvent(event);
    }
```

Fig4.5: Rent aggregate

Above screenshot is taken from my prototype, we are not going to see the technical details i.e., framework related annotation right now, we will talk about them later when we develop individual monolith, microservices. Now we look at how an aggregate is written. Rent aggregate encapsulates Tenant and BookingState as the business rules inside this bounded context ask the consistency of this objects. Whenever, a flat need to be rented, a rentFlat command is fired by the user, program flows to the rent aggregate and the state of Tenant class and BookingState is maintained in an atomic unit.

For an aggregate to be aggregate it must satisfy some rules,

Enforce Consistency

State of aggregate can be mutated; this makes the data vulnerable to have state inconsistency. Aggregate bears the responsibility of validating the business invariants

before bringing any change. For modifying the state of aggregate specific methods are written, also called as commands. Before any change, aggregate plays with these commands, verification, validation is done. Commands can be played in two ways: one by implementing a separate method in aggregate object. Another way is to include a parameterized constructor that encapsulates the operation related to command.

As we can see rentFlatCommand is parameterized for Rent object. Inside which object of Tenant, RentId, BookingState is instantiated along with other static variables. If this command is validated successfully, then all these variables and objects are persisted together, if not its halted. By this way, aggregate enforce consistency. Here Tenant is entity and Booking State is value objects, we will learn about them individually don't worry about it now.

Hierarchy of entities and value objects

As we know, entity cannot be used as an independent pattern in DDD, it falls under aggregate. In many scenarios, multiple objects should share transactional boundaries whether it's between entities or between entities and value objects; say when state of one object is dependent on the business rule of another object. The answer to this is changes to multiple objects must be done in one atomic transaction.

Referencing other aggregates

Since value objects and entities are all under aggregate, there is a chance of aggregate growing much bigger which may arise problem in scalability and performance. Only the business object that are bounded by invariants and need consistency between them is covered inside an aggregate. Information that can eventual consistency i.e., consistency with time can reside outside the boundary of the aggregate. In summary, aggregates should be as small as possible including objects that work together to solve any business rule.

Entity

As we know DDD is not a pre-defined step of pattern which can be directly applied, it's about the business problem. To classify entity from the set of business objects we must have to have a proper communication with the business expert, or we can easily misplace entity with value objects. Generally, entities are objects that has identity of its own and

are unique. Now with that said let's compare this with our example, we have Tenant as an entity, it's because a person who rents a flat is unique. It has its own uniqueness and identity. A person can change its name, but does that take away the uniqueness of the person? No. Entity has a lifecycle, 'Tenant is looking for a house', 'Tenant booked a house', 'Tenant left the house', so basically to have a lifecycle it must have an identity. The requirement for being an entity is that it must be unique for each instance of entity. Let's look at a table and see the example

Table 1: illustrating about entities

ID	FIRSTNAME	LASTNAME
1	Rakesh	Altaf
2	Muhammad	Sheikh
3	Muhammad	Sheikh

The value of the entity remains immutable comparing that to value objects. Here, we have two Muhamad sheikh, but that doesn't make change because they both have their individual identity. There, value remains unchanged throughout except for some conditions. Value objects define the properties of entities, that's the major difference between entities and value objects. Entities are fundamentally about identity, looks for answers about who rather than what.

Now that we gathered simple information on entities let's see how we have implemented in our prototype.

```java
@Embeddable
public class Flat {
    @Column(name = "location")
    public String location;
    @Column(name = "booking_amount")
    public int bookingAmount;
    @Column(name = "monthly_rent")
    public int monthlyRent;
    @Column(name = "type")
    public String type;
    public Flat() {}
    public Flat(String location, int bookingAmount, int monthlyRent, String type) {
        this.location = location;
        this.bookingAmount = bookingAmount;
        this.monthlyRent = monthlyRent;
        this.type = type;
    }
    public int getBookingAmount() {
        return bookingAmount;
    }
    public String getLocation() {
        return location;
    }
    public Integer getMonthlyRent() {
        return monthlyRent;
    }
    public String getType() {
        return type;
    }
    public void setBookingAmount(int bookingAmount) {
        this.bookingAmount = bookingAmount;
    }
    public void setLocation(String location) {
        this.location = location;
    }
    public void setMonthlyRent(Integer monthlyRent) {
        this.monthlyRent = monthlyRent;
    }
    public void setType(String type) {
        this.type = type;
    }
```

Fig4.6: implementation of entity in our prototype

Flat entity encapsulates information related to flat, so first why did we consider flat as an entity? A flat has its own unique identity. At most, the owner of a house may change and that happens quite fewer. Similarly, its immutable, the address, location monthlyRent and bookingAmount changes on longer period. Flat is encapsulated under owner aggregate inside manage bounded context.

Although entities have identity of their own, they cannot stand alone. It starts with the instantiation of the aggregate and destroys with the aggregate. It's considered as the secondary identifiers after the aggregates.

Value Objects

Unlike entities, value objects are mutable and doesn't have any uniqueness in them. As name suggests, they are objects carrying values. If we take the example from our

Note

There are three types of equality when it comes to comparing object:

1) **Reference equality:** When two reference variable points to same object.
2) **Identifier equality:** When two object have same identifiers, like same primary key in tables
3) **Structural equality:** Two objects are equal when all their attribute matches.

prototype, we have BookingState as a value object. Let's consider there is home and its currently not rented, whenever it's booked the state change to booked. So here when again the tenant leaves the house, the state changes to not rented. There is no identity of its own, it's just an object that carries value. That's why its inside entity describing the entity. Let there be an object of bookingState b1 and its value be rented, similarly let there be another object b2 and its value be rented as well, comparing b1 and b2 should result in same. Because value objects follow **reference equality and structural equality** whereas entity objects follow, **reference equality and identifier equality**.

Thumb rule while applying DDD ideology, using value objects to describe properties of other objects. Let's see example from our prototype.

```java
@Embeddable
public class BookingState {
    @Column(name = "amountpaid")
    public Float amountPaid;
    // @Column(name = "isTotalAmountPaid")
    // public boolean isComplete = false;

    public BookingState() {

    }

    public BookingState(Float amountPaid) {

        this.amountPaid = amountPaid;
        // this.isComplete = isComplete;
    }

    public Float getAmountPaid() {
        return this.amountPaid;
    }

    public void setAmountPaid(Float amountPaid) {
        this.amountPaid = amountPaid;
    }

    // public boolean getIsComplete() {
    // return this.isComplete;
    // }

    // public void setIsComplete(boolean isComplete) {
    // this.isComplete = isComplete;
    // }
}
```

Fig4.7: value objects from our example

In conclusion, value object is an object that define itself with the value it carries at the time.

Domain Commands

When we discussed about aggregate, we encountered a term as command, command are basically the only instructions that are conveyed from UI to the endpoint. They are the **do-this-instruction** to the aggregate and reason why aggregate changes its state. In our prototype, we must rent the house so, while conversing with the domain expert, both developer and the expert met in common ground of calling this command as **RentFlatCommand.** This command contains the information that the user must sent it from the UI to make state change. Generally, data transfer as a JSON object, but while communicating between endpoints framework does this on it own. Events are the result of command, but saying that what I mean is, whenever a command is executed because

of it an event is generated. Basically, commands and events are the only pattern that changes the state of aggregate.

```
public class RentFlatCommand {

    public String rentId;
    public String firstname;
    public String lastname;
    public int age;
    public boolean maritalStatus;
    public String profession;
    public String flatID;
    public Float bookingAmount;

    public RentFlatCommand() {
    }

    public RentFlatCommand(String firstname, String lastname, int age,
            boolean maritalStatus, String profession, String flatId, Float bookingAmount) {
        this.firstname = firstname;
        this.lastname = lastname;
        this.age = age;
        this.maritalStatus = maritalStatus;
        this.profession = profession;
        this.flatID = flatId;
        this.bookingAmount = bookingAmount;
    }
```

Fig4.8: command that changes the state of Rent aggregate

Domain Events

As we learned that commands, events are generally the after effect of command. Domain event was not included in the original book written by Eric Evan “Domain Driven Design: Tracking complexity in the heart of software”. It was more of a discovered thing later when people started to apply the principles of DDD.

Basically, Domain Events are the message of what has recently happened for example, hotel booked, booking completed etc. Since it describes what has just happened it should be named as a past tense. The goal is to describe an important task that has completed and carry important information regarding the completion of the task. Say for example, if there are other criteria after filing the form for flying abroad, we will name the event as FormFilled and this event will carry the information that might be helpful for further process.

As everything starts and ends with an aggregate, events are published by an aggregate, other interested aggregate or module can subscribe to it. Its like a notification that carries information. In our prototype, we have an event FlatBookedEvent, whenever a flat is booked, rent aggregate raises this event. It carries the information about the owner and booking amount. The reason behind carrying this information is because, cross validation is done by another bounded context to see if whole amount is paid before releasing the key to the tenant.

```
public class FlatBookedEvent {

    private int OwnerId;
    private int bookingAmount;

    FlatBookedEvent() {
    }

    public FlatBookedEvent(int OwnerId, int bookingAmount) {
        this.OwnerId = OwnerId;
        this.bookingAmount = bookingAmount;
    }

    public void setOwnerId(int OwnerId) {
        this.OwnerId = OwnerId;
    }

    public int getOwnerId() {
        return OwnerId;
    }

    public void setBookingAmount(int bookingAmount) {
        this.bookingAmount = bookingAmount;
    }

    public int getBookingAmount() {
        return bookingAmount;
    }
}
```

Fig4.9: event after renting the flat

Sagas

Bounded contexts are built individual, have their own context and work but when it comes to working of software, all these corelated bounded context must work together. In another term, flushing out business process inside a business domain is called saga. Saga's spams around multiple bounded contexts, probably the only domain model that has this ability. Aggregate is the participant of the saga; they react to multiple events and orchestrate the business process. In simple language, any work has a flow, for example when we make tea it has steps, similarly in programming everything happens one after another. Sagas encapsulates this steps that spans around multiple bounded contexts.

Sagas listens to the event published by any of the context and fires commands to the subsequent components. The only responsibility it carries is that, if any step inside of saga is terminated the whole process should be re-evaluated to the step that it was in before the start of the saga.

We generally have two kinds of saga; one is orchestration, and the other is choreography. In orchestration saga, a central component is what takes care of the whole process i.e., publishing of events and invocation of command.

Difference in orchestration saga and choreography saga

In orchestration saga, the responsibility of handling a entire process that includes multiple publication of events and invocation of commands is given to a single component. But in choreography saga, all events and commands are handled by their own handlers, there is no component that handles this entire process.

In our prototype we have used choreography saga. But I have a dummy code that I wrote for showcasing orchestrating the RentFlatCommand.

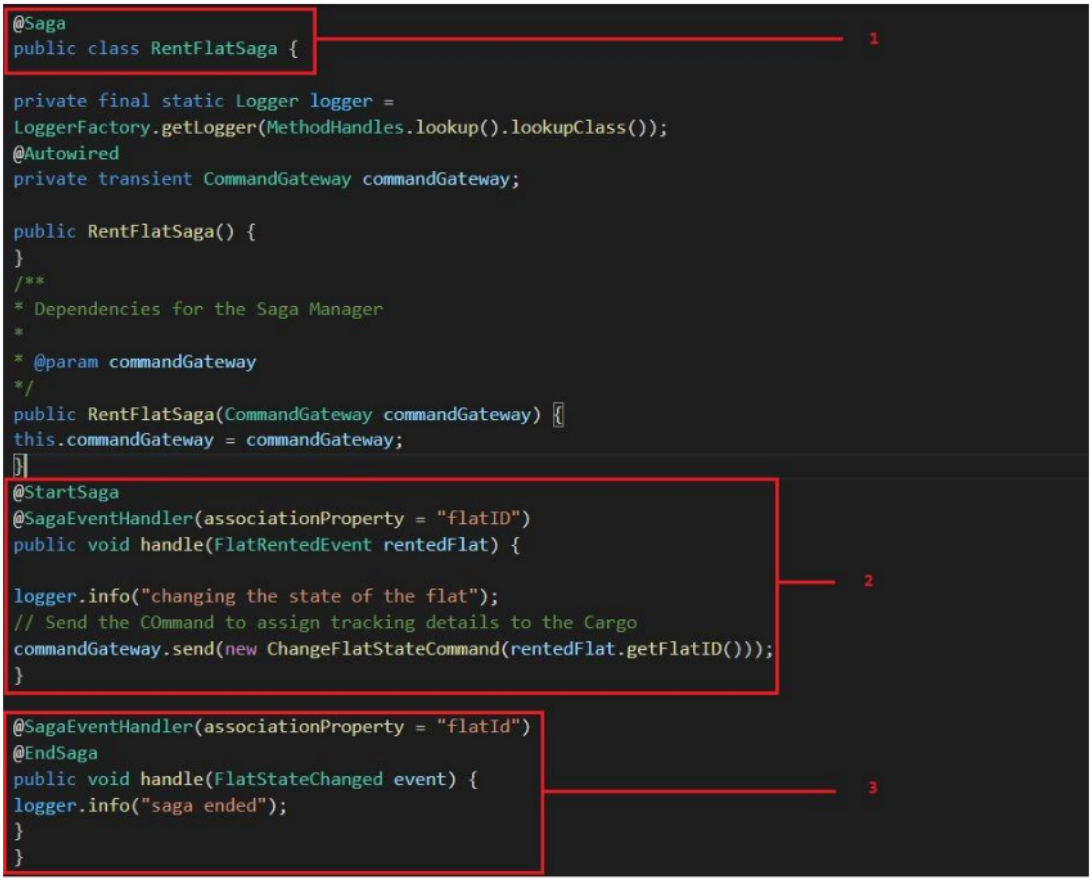


Fig5: Orchestration saga for rent flat command

In above figure, there are steps given in terms which the saga flows. In step 1 we specify to the framework (Axon framework) that this class is saga. Now there are gateways (like as a queue), in above figure we can see CommandGateway which are provided by axon framework to make it easier for handling commands. In step 2 we start the saga using (@StartSaga). Now here is how these flows, first the user requests the RentFlatCommand this is straight sent to aggregate, now aggregate verify validate the business invariants and make changes to its state if everything aligns. After that event is raise related to the command just executed i.e., FlatRentedEvent is raised. When this event is raised, now here comes the saga, we have given an annotation in step 2 i.e., @SagaEventHandler this make sure that after the FlatRentedEvent is raised the program

should flow from here. Now that we reached the place where the event is subscribed, another invocation of command is done i.e., ChangeFlatStateCommand.

In step 3, there is another @SagaEventHandler and it's associated with FlatStateChanged event. When this event is raised it is subscribed by this procedure in highlighted in step 3. Now this also marks the end of the saga as it has another annotation as @EndSaga. So, this is how a single component RentFlatSaga handled all the subscription to events and invocation of commands.

Well that marks the end for the brief introduction of the principles of Domain Driven Design. We used a prototype along i.e., Homrent to describe these principles. In the next chapter we will look after various software architectures on which we will use these principles to see the differences.

4.3 - Domain Driven Design's Principles with layered Architecture (Monolith)

Layered or N-layered architecture is considered as traditional software architecture as its barely used in designing today's software. As its name suggests, software includes different layer, layer on top of the other is in control of flow. Every architecture is built while keeping in mind about SoC: Separation of Concern, so as Layered architecture.

4.3.1 - Layered Architecture

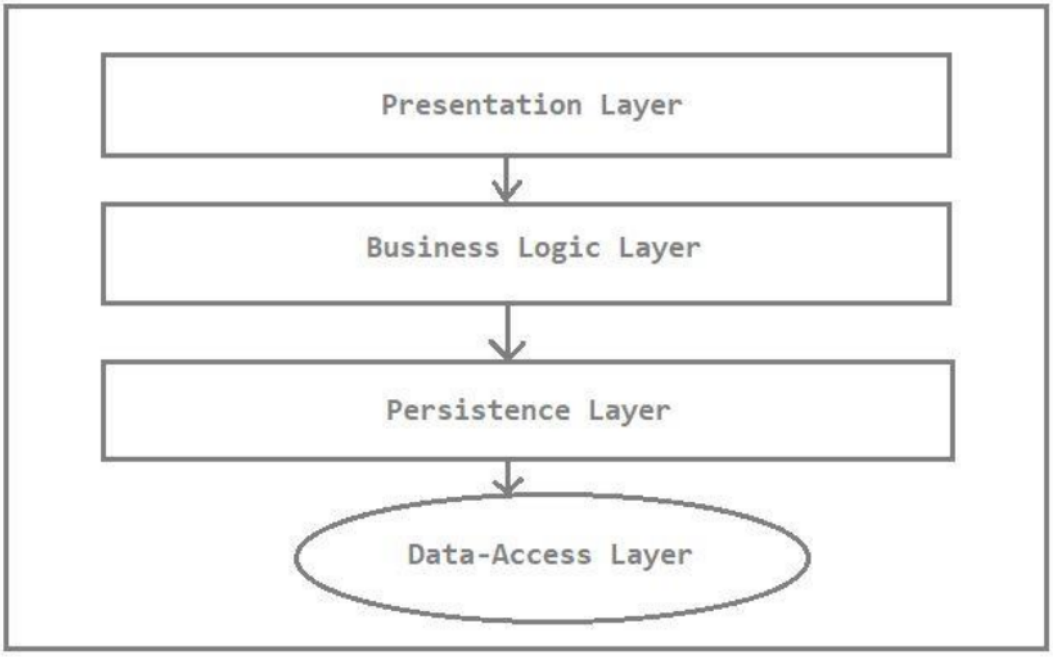


Fig5.1: Layered Architecture

Presentation layer consists of UI, front end part that is visible for the user. In our prototype we don't have any presentation layer. The only way user interact with the software is via form or clickable buttons. When user submit any form or buttons, the information is sent to the business logic layer. Now, its not that simple in real world we do have API integration, Frameworks, message brokers etc., but the general idea is that data is sent from front end to logic layer where it's processed. Persistence layer associates the service

layer with data layer. It contains codes that help in storing and retrieving information from the database.

4.3.2 - Monolith Architecture

Monolith architecture stands among the foundation for developing enterprise application. With the advent of microservices we barely see the use of monolith architecture. Monolith architecture focused on

- Enforcing Transactional Consistency
- Easier maintainability
- Centralized data management
- Compact relationship among modules

With the point stated above we can distill out that it has its own pros and cons, one of the main reasons behind it being obsolete is that it doesn't adapt to the concept of cloud computing. Every bounded context is packed under same WAR file and deployed together. Although it reduces the need for external dependencies such as RabbitMq, Kafka etc., and is quite simple to bring in use but it doesn't meet the demand for software built today.

Let's see how we have implemented monolith architecture in our prototype

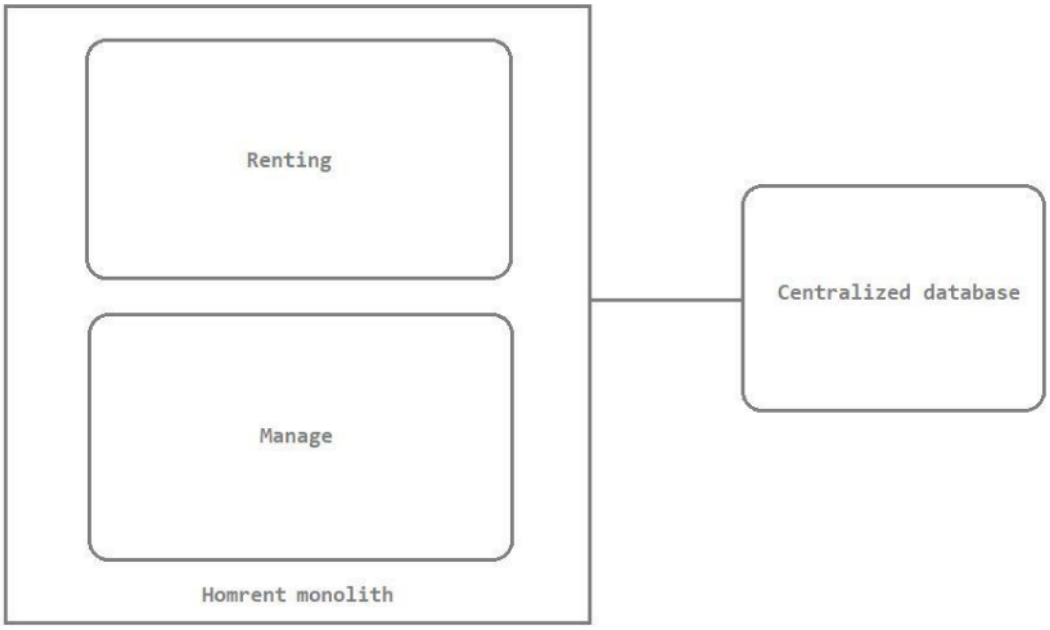


Fig5.2: monolith structure for prototype Homrent

We have developed a prototype that follows DDD principles with layered and monolith architecture.

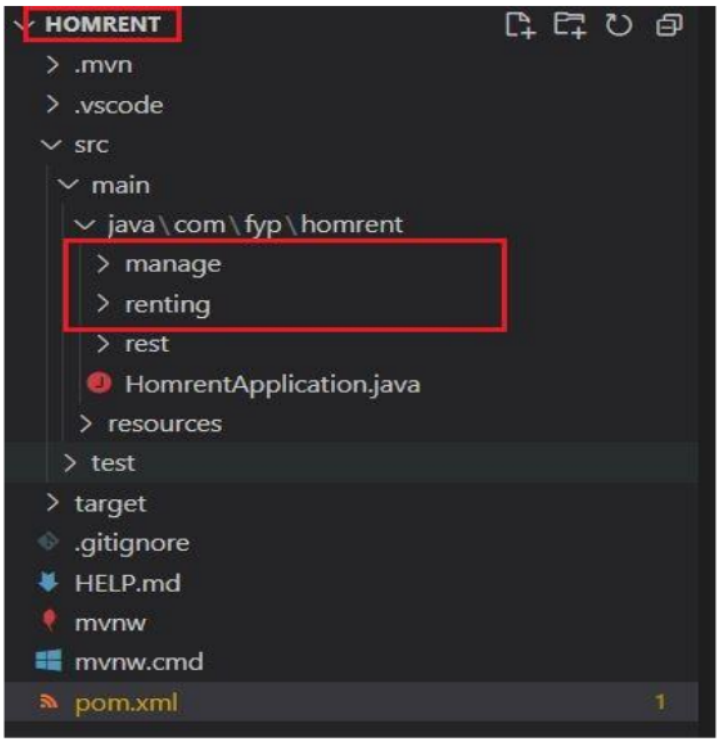


Fig5.3: monolith structure for Homrent

In the figure above we can see that, manage and renting both bounded contexts are under same Homrent WAR file, and later will be deployed together.

As we know we don't have presentation layer so we will use postman: software that help us in testing the endpoints that we will generate while developing the API.

We are using Spring boot framework, quite popular now a days for developing both monolith and microservices java-based projects. Its component will be discussed along with the ss that we will provide gradually with the report.

The domain model that we discussed earlier is same for all this demo projects, simpler changes will be made according to the architecture that we may use but the general framework will be the same. Inside renting bounded context we have following structure.

Fig5.4: renting bounded context in layered architecture

The data module consists of data-access layer, the model contains modules related to data persistence whereas services is the business logic layer. There is also another module named rest, this module listens and response to the incoming requests.

```
package com.fyp.homrent.renting.data;

import com.fyp.homrent.renting.model.aggregate.Rent;
import org.springframework.data.jpa.repository.JpaRepository;

public interface RentRepository extends JpaRepository<Rent, Long> {

    Rent findByRentId(String RentId);
}
```

Fig5.5: data layer of layered architecture

In the figure above, we see RentRepository is extending JpaRespository which is provided by spring framework, what it does is it provide us the flexibility of just defining method which is automatically configured by the JpaRepository. **findByRentId(String RentId)** is just declared later everything is done by spring framework.

Model consists of business logic layer also in other terms it's the domain model

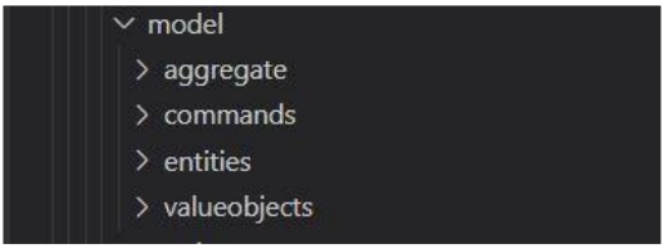


Fig 5.6: domain model of layered architecture

Inside aggregate we have the rent aggregate,

```java
@Entity
public class Rent extends AbstractAggregateRoot<Rent> {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Embedded
    private RentId rentId; // Aggregate Identifier
    @Embedded
    private Tenant personToRent;
    @Column(name = "flat_id")
    private String flatId;
    public Rent() {
    }
    public Rent(RentFlatCommand rentFlatCommand) {
        this.personToRent = new Tenant(rentFlatCommand.firstname, rentFlatCommand.lastname,
        rentFlatCommand.age, rentFlatCommand.maritalStatus, rentFlatCommand.profession); this.flatId = rentFlatCommand.flatId;
        this.rentId = new RentId(rentFlatCommand.getRentId());
    }
    public String getFlatId() {
        return flatId;
    }
    public Long getId() {
        return id;
    }
    public Tenant getPersonToRent() {
        return personToRent;
    }
    public RentId getRentId() {
        return rentId;
    }

    /**
     * Method to register the event
     *
     * @param event
     */
    public void addDomainEvent(Object event) {
        registerEvent(event);
    }
}
```

Fig5.7: aggregate in layered aggregate

As discussed earlier, the domain model is almost same for every architectural type shown in this report. We have discussed about value objects, entities, commands, and events we can find it here https://github.com/Anupam1223/DDD-prototype.git. In this above figure, there is an annotation @Entity this is provided by framework which help the spring framework to recognize that it's the aggregate as we don't have any annotation for aggregate. When we defined JpaRepository for Rent aggregate, now whenever we have a database connection, be it postgresql, mysql or oracle we will have Rent as a Table, i.e., @Entity and spring framework will do their magic. For this prototype we have used H2 database, which is an in-memory database provided by spring framework itself.

To embed any database system with spring framework you must set configuration. Inside resources folder there is a file application.properties, inside of which you should give configuration for H2 database.

Also, you should mention the version of H2 database in pom.xml file, which is like dependency management of spring framework.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.6.3</version>
        <relativePath/>
        <!-- lookup parent from repository -->
    </parent>
    <groupId>com.fyp</groupId>
    <artifactId>homrent</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>homrent</name>
    <description>Demo project for Spring Boot</description>
    <properties>
        <java.version>11</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
```

Fig5.8: dependency management of H2 database in pom.xml file

Along with H2 database, we also can see multiple other dependencies. We can handpick these dependencies in this site, https://start.spring.io/.

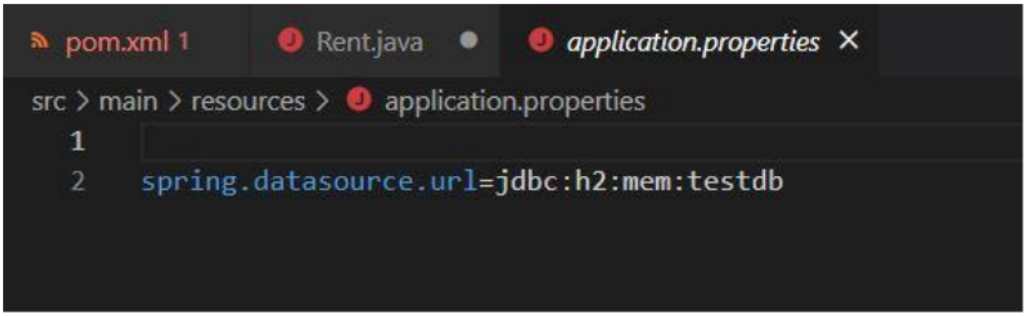


Fig: 5.9: H2 database configuration in application.properties

Along with business layer and data persistence layer we also need some internal and external services. Business domain cannot work itself; it needs to be used by internal services. Similarly, having a working bounded context isn't enough it must have the ability to listen and response to request, we have separate module for this named as rest. Let's first have a look at internal services.

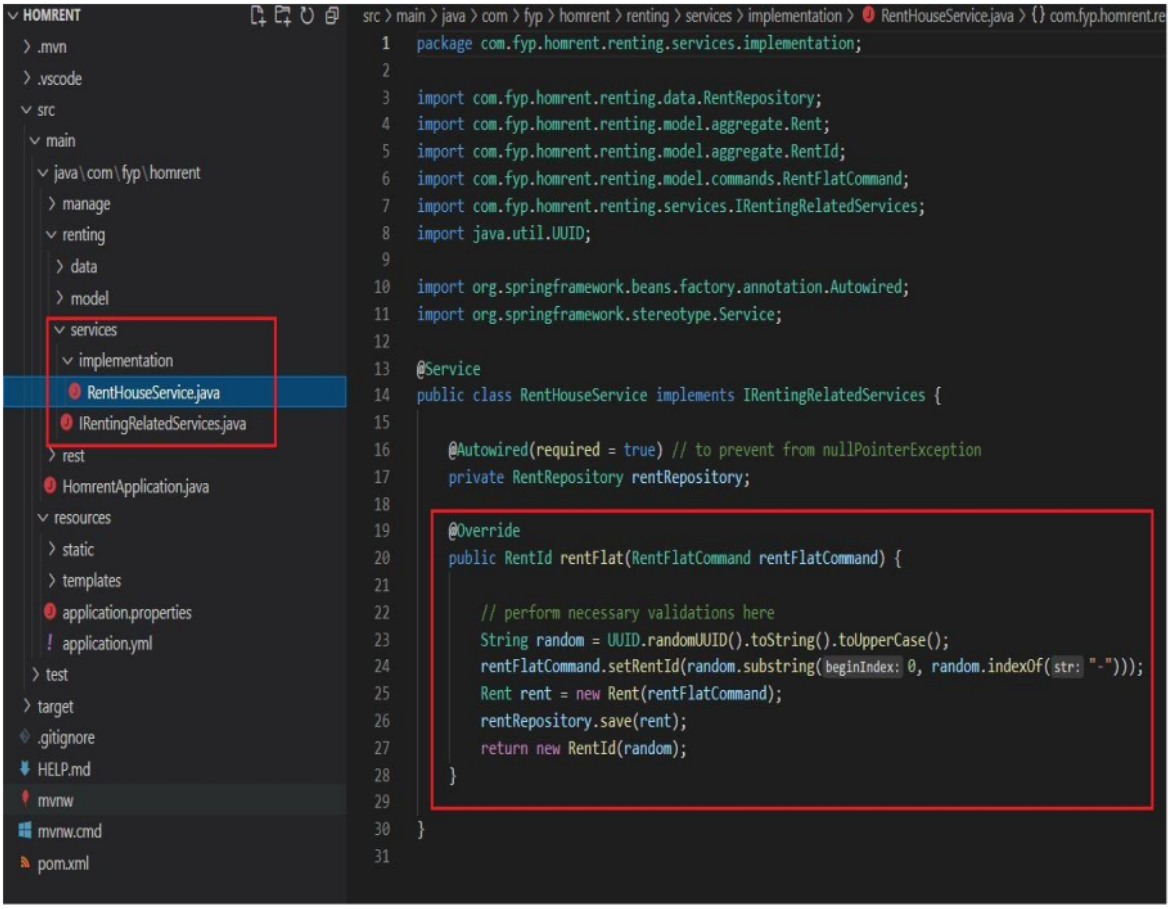


Fig6: service that help in renting flat

This service is invoked whenever an external service gets request on renting the flat. Let's see how its invoked by external service.

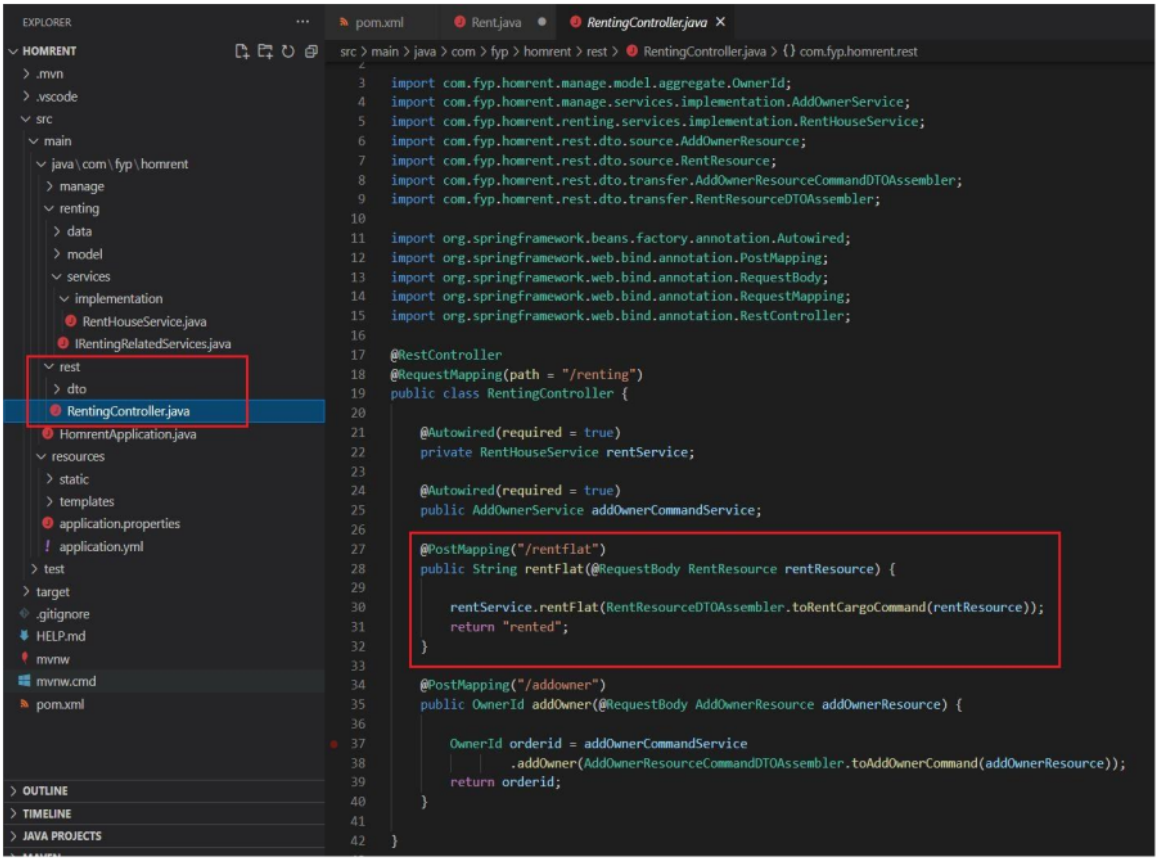


Fig6.1: rest module for integrating APIs

We have an endpoint, /rentflat which listens to the request of renting a flat. As we can see from this endpoint our rentService is calling its method rentFlat. We can also see dto folder just below rest, this folder contains modules that helps in convert object of rentResource to rentFlatCommand so that it can be understood by the services. Inside of rentFlat we are sending an argument RentResourceDTOAssembler.toRentCargoCommand(rentResource), it is to convert resource to command.

4.4 - Domain Driven Design's Principles with Hexagonal Architecture (Microservices)

Also called as port and adaptors, hexagonal architecture looks like a hexagonal when we draw its specifications.

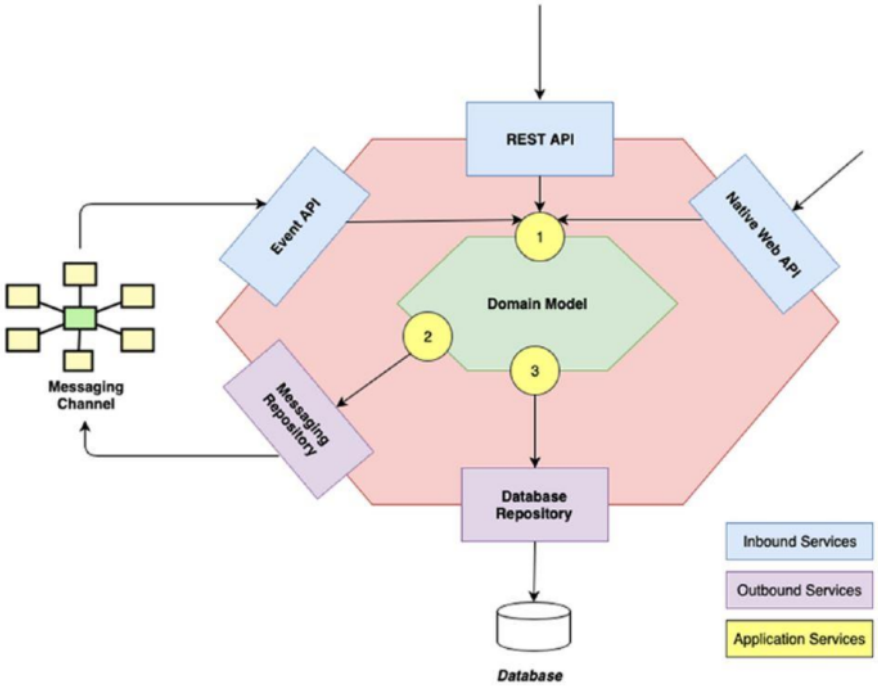


Fig6.2: hexagonal architecture

From the figure above we can assume why the architecture is named as hexagonal. Every side of the architecture are the gateway for flowing of information. Information either flow in or out from the application, so considering that there are three types of services: inbound, outbound, and application services.

4.4.1 - Inbound Service

In any application there must be an interface that act as an entrance for the request sent to the application. As we know hexagonal architecture follows the concept of ports and adaptors, while talking about inbound services lets also talk about inbound ports and inbound adaptors.

- An inbound port facilitates an interface through which inbound adaptors can send information to the actual business logic layer. In the above figure, number 1 points the inbound adaptor, they are implemented as application services.
- An inbound adaptor in technical word is a class that incorporates external messages, basically a listener to the request made by any of the client. some of the examples are, REST API, EVENT LISTENERS.

Let's understand this by an example If a login request is sent to the application, then username and password is decoded in inbound adaptors and this information are sent to the application services (ports) which are ready to adapt this information. Hence that's why it's called ports and adaptors.

4.4.2 - Outbound Service

When request from the inbound services is processed by the domain model, there must be another interface that facilitates the response for the request. There are also outbound ports and adaptors.

- An outbound port provides an interface that helps in either publishing the events from the domain model or persists the change. In the figure we can see number 2 and 3, these are outbound ports. They are also implemented as application services.
- An outbound adaptor receives the instructions from the outbound adaptors and does accordingly, for example if a message needs to be published in the queue, then the outbound port calls messaging repository and if wants to persist then calls database repository.

As we saw earlier, JpaRespository provided by spring framework implemented in data layer, is an example of outbound adaptor and rest module where we discussed about DTO objects, that module is inbound adaptor in our case. Let's look at these one by one,

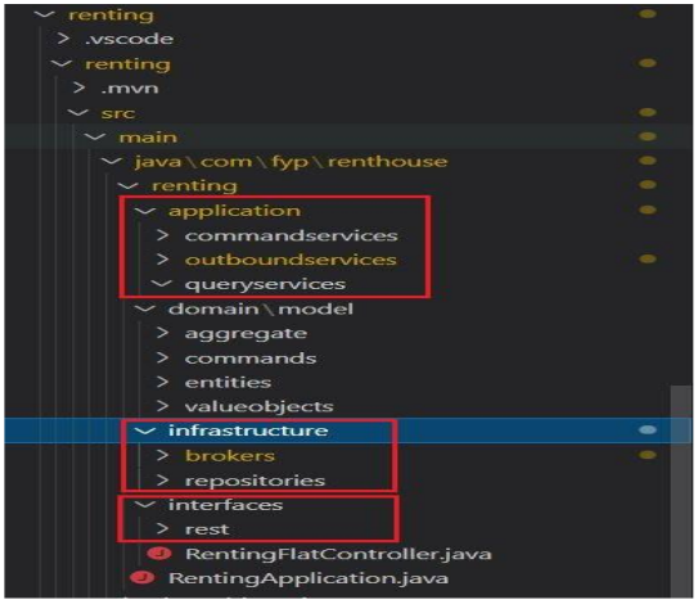


Fig6.3: implementing hexagonal architecture

As highlighted in the figure above, application module contains command services, outbound services, and query services. Command services and query services are inbound ports that gets information from inbound adaptors.

We can also see inside infrastructure module, there are other two modules' brokers and repositories

Fig6.4: implementation of inbound and outbound adaptors

Inside of brokers module there is another module, i.e., rabbitmq, inside of this module we have RentEventSource.java as an interface which is an adaptor that adapts to the information regarding of publishing the message to the message broker. Similarly, there is another module repositories which contains java file that defines the persistence interface.

Both modules, brokers and repositories generate response out of the application. We can also see rest module just below it; this module is inbound adaptor which gets the initial request from the client. We've discussed about this module while in layered architecture.

Till now we saw, the frame of hexagonal architecture used in our project but now let's have a deeper dive and see the code that we use to implement for achieving this in real life.

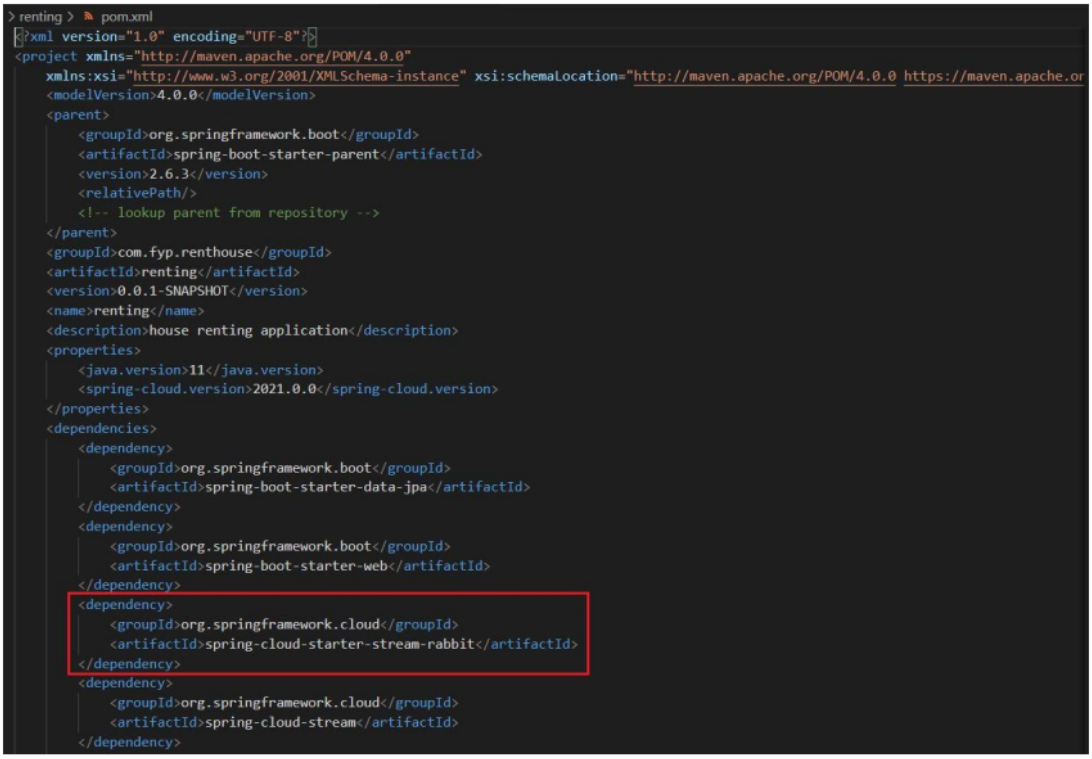
4.4.3 - Microservices

when defining domain driven design, we discussed about bounded context. In layered architecture we defined a single WAR file for all these bounded contexts, it is quite hard to achieve cloud participants also there is only one database for all the bounded context, but in microservices we will have individual WAR file, individual database system (any kind), induvial broker and supports cloud computing. Each of these bounded contexts has separate API and are decentralized which means they are developed, tested, and deployed individually. Now the question arises, well how do all these services/bounded contexts communicate with each other. For that we use third party dependency, such as message brokers, these brokers are the medium through which events raised by any of the contexts is listened by another context.

We have three different WAR file for hexagonal architecture, **rent**, **manage**, and **owner.** We have implemented RabbitMQ as a message-queuing software.

4.4.4 - Configuring RabbitMQ with our microservices

Let's include the axon framework in our pom.xml file



```
> renting > pom.xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.or
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.6.3</version>
        <relativePath/>
        <!-- lookup parent from repository -->
    </parent>
    <groupId>com.fyp.renthouse</groupId>
    <artifactId>renting</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>renting</name>
    <description>house renting application</description>
    <properties>
        <java.version>11</java.version>
        <spring-cloud.version>2021.0.0</spring-cloud.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-stream</artifactId>
        </dependency>
```

Fig6.5: adding dependency for RabbitMQ in pom.xml

Also we need to add configuration in application.properties file

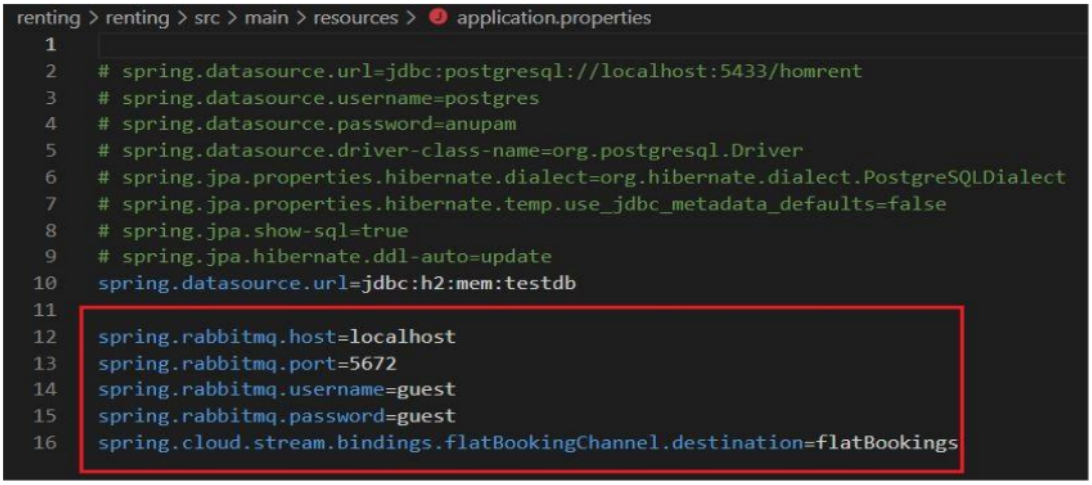


Fig6.6: configuration of RabbitMQ in application.properties

After that we define a channel for flowing message,

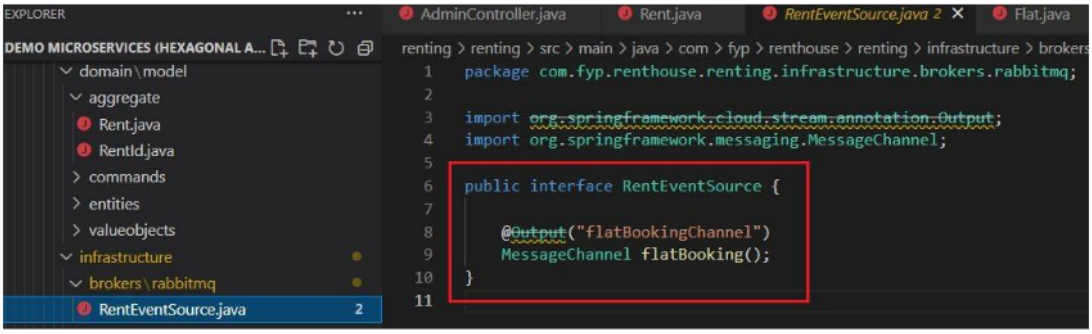


Fig6.7: defining message channel in RabbitMQ

In figure 3.6, we can see at line no 16, we have configured the flatBookingChannel which destination flatBooking that means flatBookings listens to the messages in flatBookingsChannel.

But how is message sent to this channel? Answer is simple, remember we have learned about outbound adaptors?

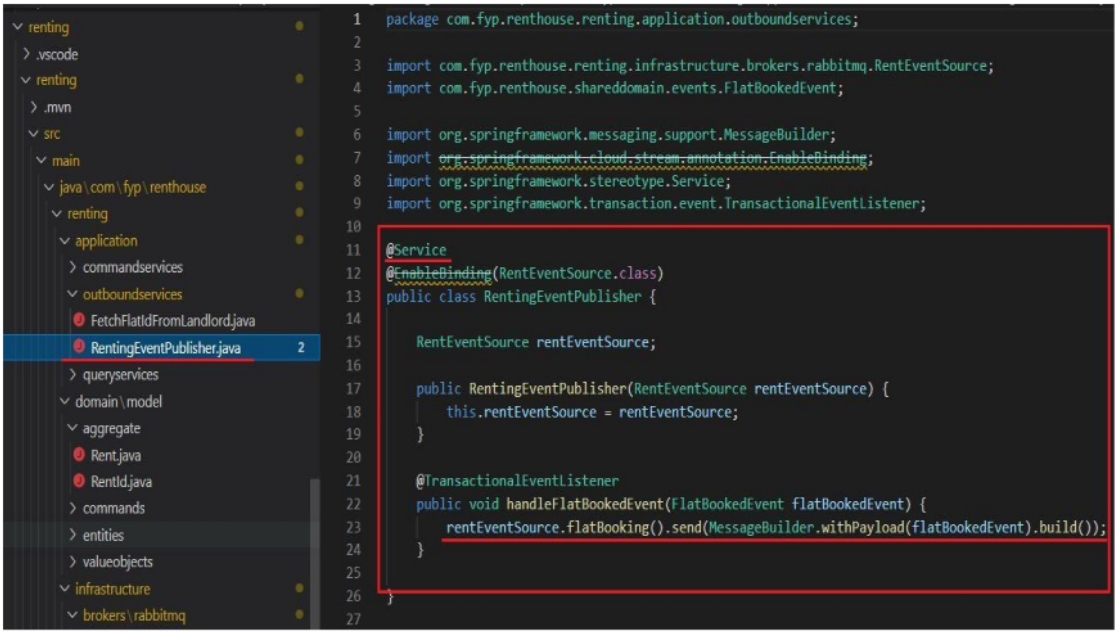


Fig6.8: publishing events in flat booking channel

@Service annotation provided by spring boot framework label it as a service. @TransacionEventListener annotation listens if FlatBookedEvent is raised. Line no 23 shows that rentEventSource builds the payload required by RabbitMQ along with the event to the corresponding message channel.

Another question, well rent bounded context sent the event in the channel but how is it listened by another context?

Let's see how will manage bounded context listen to the message in the flatBookingChannel.

manage > manage > src > main > resources > application.properties

```
1
2   spring.datasource.url=jdbc:h2:mem:testdb
3
4   spring.rabbitmq.host=localhost
5   spring.rabbitmq.port=5672
6   spring.rabbitmq.username=guest
7   spring.rabbitmq.password=guest
8   spring.cloud.stream.bindings.input.destination=flatBookings
9   spring.cloud.stream.bindings.input.group=flatBookingsQueue
```

Fig6.9: RabbitMQ configuration in manage to listen flat booking queue

This is listened by stream listener defined by @StreamListener annotation.

```
// @RestController
@Service
@EnableBinding(Sink.class)
public class ManageFlatController {

    @Autowired(required = true)
    private ManageRepository manageRepository;

    @Autowired(required = true)
    FetchOwnerInformation fetchBalance;

    public ManageFlatController(FetchOwnerInformation fetchBalance) {
        this.fetchBalance = fetchBalance;
    }

    @StreamListener(target = Sink.INPUT)
    public void receiveEvent(FlatBookedEvent flatBookedEvent) {

        JustTest test = new JustTest(flatBookedEvent);
        System.out.print("Id " + flatBookedEvent.getOwnerId() + " amount -> " + test.getBookingAmount());
        int bookingAmount = fetchBalance.fetchBookingAmount(flatid: 1);

        if (flatBookedEvent.getBookingAmount() < bookingAmount) {
            test.setRemaining_amount(bookingAmount - flatBookedEvent.getBookingAmount());
        } else {
            test.setRemaining_amount(remaining_amount: 0);
        }
        manageRepository.save(test);
    }
}
```

Fig7: listening the event present in the flat booking channel

ManageFlatController listens to the event FlatBookedEvent, in figure 3.9 we have specified input as flatBookings also specified the queue name flatBookingsQueue. whenever RabbitMQ receives event published from the rent bounded context, manage bounded context get the notification and with correct utilization of annotation provided by spring framework subscribe to the event. Publishing of event from one end which could be later subscribed by another end is termed as pub-sub.

4.5 - Domain Driven Design's Principles with Hexagonal Architecture Along with CQRS and Event Sourcing (Microservices)

This is our final implementation of DDD with software architectures. CQRS and Event sourcing being new and advance topic we need extra extension with spring framework to develop our prototype. Axon framework is the perfect fit for us, it's light weight, easy extension and have ample lib-code to help us support our motive. Implementing CQRS with event souring is quite challenging as incorporates idea which is completely different from what we are used to. Every business logic is based on events.

4.5.1 - Event Sourcing

Until now we store a current state of an aggregate. Let's understand this through an example, there is a lifecycle of a flat. A flat is uploaded by an owner, it's out there for rent, when a person is interested its booked and finally with clear financial settlement its rented. I will draw a table for every stage,

Table 1.2: persistence of aggregate for traditional approach

Stage 1

FlatId	Owner	Type	state	Rent fee
1	Aster	2BHK	open	10000

Stage 2

FlatId	Owner	Type	state	Rent fee
1	Aster	2BHK	booked	10000

Stage 3

FlatId	Owner	Type	state	Rent fee
1	Aster	2BHK	rented	10000

Finally, in our database we will only have the information of that of stage 3. This was how we based our development of software's. Event sourcing is based on completely different

ideology, we base our persistence on time and events. For now, forget about the traditional approach, think about storing every change that the aggregate goes are stored in a row. For example,

Table 1.3: persistence of aggregate in case of event sourcing

S N	Owner	Type	Flat Id	state	Rent fee	Modified on	Event type
1	Aster	2BHK	1001	**open**	10000	10/03/2000	Flat added
2	Aster	2BHK	1001	**booked**	10000	10/10/2000	Flat booked
3	Aster	2BHK	1001	booked	8000	10/20/2000	Rent Amount Changed

The table above tells the story about the flat that is out there for rent. Having this example and visualizing is quite important because it's not quite easy when we align this with domain driven design. Now if we want to infuse DDD with event sourcing, events drive the lifecycle of an aggregate. Every event of an aggregate is stored with a time factor.

With traditional approach traversing through the history was impossible, what we have now is what we get as evidence. Say you booked the flat but somehow it wasn't recorded, there would be quite of headache for the business as the data doesn't speak for itself, quite work around with bank details is needed. But suppose if you have event source data store, then you can traverse and evaluate events before and after.

4.5.2 - Event Store

Events are stored in event store; they are the database for events but also are message broker because they publish events to its subscribers. Each event after being stored are published. Event store is the backbone for the application that follows event driven architecture. CQRS (Command Query Responsibility Segregation) and event sourcing complement each other very well so using both together will contribute to developing

advance software's. While designing prototype Homrent we have implemented an inbuilt event store provided by axon framework and implemented CQRS with event sourcing.

4.5.3 - Command Query Responsibility Segregation (CQRS)

As name, we segregate the responsibility of command and query. By that what we mean is there are two general operation we perform when we use event driven architecture and that is storing the change in aggregate which is done by command, and another is querying the information regarding the aggregate. A figure is designed to explain how we can integrate this concept with our prototype.

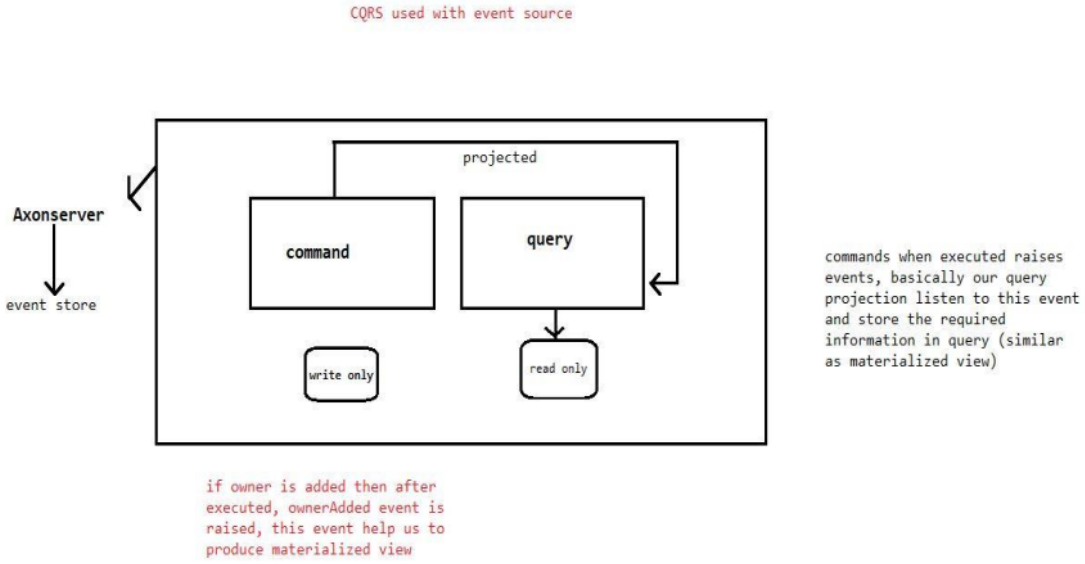


Fig7.1: explanation of CQRS with Event Sourcing

First, there are two separate modules for both command and query as you can see in the figure above. Along with separate modules they have separate flow, persistence mechanism, separate logical process etc., Under command module we see write only database, a command generally is what make changes to an aggregate so it can use database system that is better at storing the data. Similarly, query side is generally for showing information out about the aggregate. It may use database technology that priories analytical components.

Now let's generalize the figure and see what's going on, first we see that the whole application using axon framework (extension added in spring framework). Also, it specifies that we use event store internally provided by axon framework to make things easier. Whenever a command say rent flat command is requested by the user then, command module calls the aggregate, aggregate then reviews all its previous stages if already have instantiated before, else creates a new instance of the aggregate. We have command associated with aggregate whereas queries are associated with projections.

Now whenever command is executed an event is raised which is saved in event store, event store publishes the event and projections subscribe to those events so that it can be stored in the database of query side. It's not that easy to grasp the idea, right? Well let me simplify it,

Command side and query side being different module, they have different database. Whenever command is executed only the database of command side is relevant with the stage of aggregate, but query side will not have any idea about it. We need a technique by which we can maintain relevancy between command side and query side. Hence, event store does that for us as we learned above that not only, they store the events in command side but also acts as a message broker between command and query side. We have subscriber on query side that listen to these messages published by event store. We now project only those data that we want our client to see. Not every information needs to be shown so query as name suggests it process queries that are asked multiple times and are projected using projection module. Now that we learned about it theoretically let's see how we have implemented all of these in our prototype.

Let's start with adding axon framework as the dependency in pom.xml file,

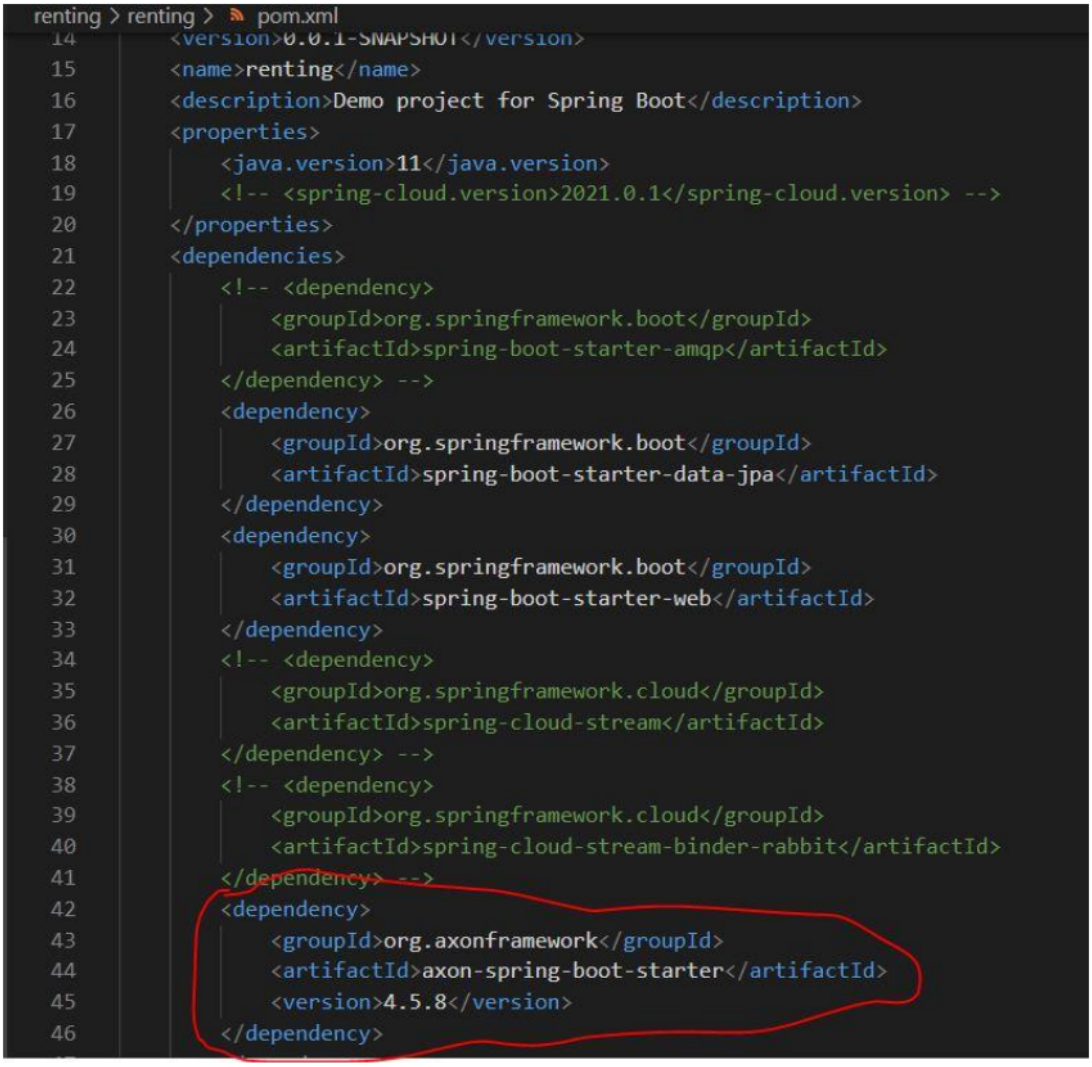


Fig7.2: Configuring axon framework in pom.xml

When shifting from traditional approach to CQRS and event sourcing, we need to bring changes in the domain model along with service models (inbound, outbound and application). Axon framework makes the work easier for us. We will look at the flow how does a request from client is processed by our application.

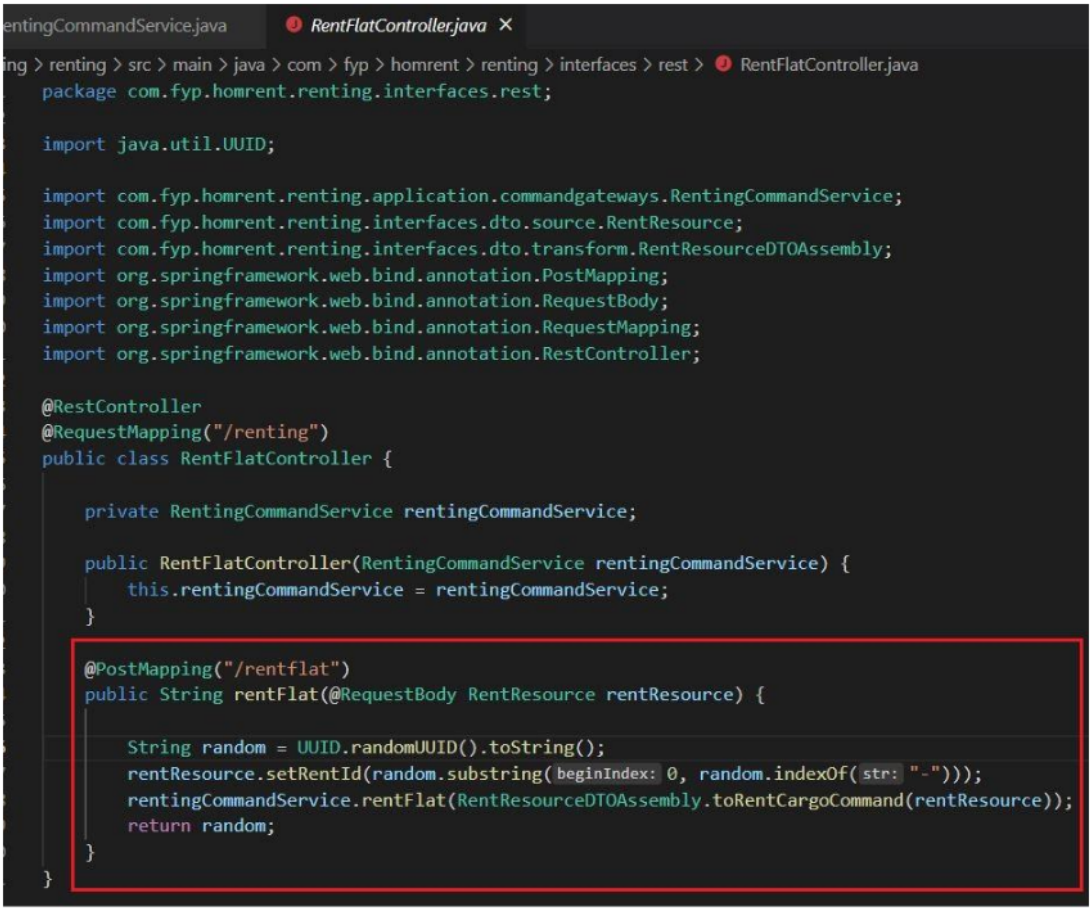


Fig7.3: Endpoint for accepting the incoming request

@RestController conveys that this is where request is made and @RequestMapping gives name for the endpoint, it read as **localhost/renting**. To specifically rent flat there is another path **localhost/renting/rentflat**, i.e., @PostMapping annotation accepts the request regarding the request that wants to make change.

Inside of rentFlat method we are initializing RentingCommandService, rentFlat method of RentingCommandService is called and the DTO objects are used to convert request to command. Program flows to the method rentFlat which is inside of RentingCommandService.

RentingCommandService.java × RentFlatController.java

renting > renting > src > main > java > com > fyp > homrent > renting > application > commandgat

```java
package com.fyp.homrent.renting.application.commandgateways;

import com.fyp.homrent.renting.domain.commands.RentFlatCommand;
import org.axonframework.commandhandling.gateway.CommandGateway;
import org.springframework.stereotype.Service;

@Service
public class RentingCommandService {

    private final CommandGateway commandGateway;

    public RentingCommandService(CommandGateway commandGateway) {
        this.commandGateway = commandGateway;
    }

    /**
     * Book a Cargo
     *
     * @param rentingCargoCommand
     */
    public void rentFlat(RentFlatCommand rentFlatCommand) {
        commandGateway.send(rentFlatCommand);
    }
}
```

Fig7.4: Screen Shot of RentingCommandService

We can see inside of rentFlat method RentFlatCommad is sent as parameter this is sent to command gateway by CommandGateway.send() method. This method is provided by axon framework. Command gateway then search for the aggregate because we know commands are associated with aggregate.

renting > renting > src > main > java > com > fyp > homrent > renting > domain > aggregate > Rent.java > {} com.fyp.homrent.renting.domain.aggreg

```java
import com.fyp.homrent.renting.application.outbounservices.acl.ChangeFlatState;
import com.fyp.homrent.renting.domain.commands.RentFlatCommand;
import com.fyp.homrent.renting.domain.entities.Tenant;
import com.fyp.homrent.renting.domain.events.FlatRentedEvent;

import org.axonframework.commandhandling.CommandHandler;
import org.axonframework.eventsourcing.EventSourcingHandler;
import org.axonframework.modelling.command.AggregateIdentifier;
import org.axonframework.modelling.command.AggregateLifecycle;
import org.axonframework.spring.stereotype.Aggregate;

import lombok.NoArgsConstructor;

@Aggregate
@NoArgsConstructor
public class Rent {
    @AggregateIdentifier
    private String rentId;
    public Tenant tenant;
    public String flatID;
    public RentingStatus rentingStatus;

    @CommandHandler
    public Rent(RentFlatCommand rentFlatCommand) {
        FlatRentedEvent flatRentedEvent = new FlatRentedEvent(rentFlatCommand.getRentId(),
                rentFlatCommand.getFirstname(), rentFlatCommand.getLastname(),
                rentFlatCommand.getAge(), rentFlatCommand.isMaritalStatus(), rentFlatCommand.getProfession(),
                rentFlatCommand.getFlatID(), RentingStatus.RENTED);
        ChangeFlatState.changeFlatStatus(rentFlatCommand.getFlatID());
        AggregateLifecycle.apply(flatRentedEvent);
    }

    @EventSourcingHandler
    public void on(FlatRentedEvent event) {
        this.rentId = event.getRentId();
        new Tenant(event.getFirstname(), event.getLastname(), event.getAge(), event.isMaritalStatus(),
                event.getProfession());
        this.flatID = event.getFlatID();
        this.rentingStatus = event.getRentingStatus();
    }
}
```

Fig7.5: aggregate for CQRS and event sourcing

@Aggregate helps command gateway to find out the aggregate. After that @CommandHandler asks the request because it contains the business logic that needs to be performed. Command Handler initialize the FlatRentedEvent and create its object. Because the task was to rent the flat. From here the lifecycle of an aggregate starts. AggregateLifecycle which is provided by axon framework starts the life cycle and first searches for @EventSourcingHandler because making to change to query side it must

need to store this change. Also, we see @AggregateIdentifier in line number 19, this identifies the aggregate and reviews all the stages of aggregate until now and makes ready with the latest stage of the aggregate.

After @EventSourcingHandler the lifecycle of aggregate includes of handling the event. One thing to be noticed is that axon framework stores data in event store whenever a command is executed. When data is stored in event store then it needs to be subscribed by the query projection.

```
18  @Entity
19  @Table(name = "status_projection")
20  @AllArgsConstructor
21  @NoArgsConstructor
22  @Data
23  public class FlatStatusProjection {
24
25      @Id
26      @GeneratedValue(strategy = GenerationType.IDENTITY)
27      private Long flatStatusId;
28      @Column
29      private String flatId;
30      @Column
31      private String username;
32      @Column
33      @Enumerated(EnumType.STRING)
34      private RentingStatus status;
35
36      public FlatStatusProjection(String flatId, String username, RentingStatus status) {
37          this.flatId = flatId;
38          this.username = username;
39          this.status = status;
40      }
41  }
```

Fig7.6: data that will be projected

The attributes shown above are the data that is going to be projected. This projection is done by event handlers. Before that we must also give a service that supports projection to project the data.

```java
package com.fyp.homrent.renting.application.querygateways;

import javax.persistence.EntityManager;
import com.fyp.homrent.renting.domain.projections.FlatStatusProjection;
import org.springframework.stereotype.Service;

@Service
public class RentingProjectionService {

    private EntityManager entityManager;

    public RentingProjectionService(EntityManager entityManager) {
        this.entityManager = entityManager;
    }
    /**
     * Stores the status of flat
     *
     * @param flatStatus
     */
    public void storeRenting(FlatStatusProjection flatStatus) {
        entityManager.persist(flatStatus);
    }

}
```

Fig7.7: projection service

This service help in persisting data. Okay now let's see the event handler that projects the data

RentingCommandService.java | FlatRentedEventHandler.java

renting > renting > src > main > java > com > fyp > homrent > renting > interfaces > eventhandler > FlatRentedEventHandler.java > {} com.fyp.homrent.re

```java
package com.fyp.homrent.renting.interfaces.eventhandler;

import java.lang.invoke.MethodHandles;

import com.fyp.homrent.renting.application.querygateways.RentingProjectionService;
import com.fyp.homrent.renting.domain.events.FlatRentedEvent;
import com.fyp.homrent.renting.domain.projections.FlatStatusProjection;

import org.axonframework.eventhandling.EventHandler;
import org.springframework.stereotype.Service;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

@Service
public class FlatRentedEventHandler {

    private RentingProjectionService rentingProjectionService; // Dependencies
    private final static Logger logger = LoggerFactory.getLogger(MethodHandles.lookup().lookupClass());

    public FlatRentedEventHandler(RentingProjectionService rentingProjectionService) {
        this.rentingProjectionService = rentingProjectionService;
    }

    @EventHandler
    public void on(FlatRentedEvent flatRentedEvent) {

        logger.info("flatId {}", flatRentedEvent.getFlatID());

        FlatStatusProjection flatStatus = new FlatStatusProjection(flatRentedEvent.getFlatID(),
                flatRentedEvent.getFirstname(), flatRentedEvent.getRentingStatus());
        rentingProjectionService.storeRenting(flatStatus);
    }
}

```

Fig7.8: event handlers in CQRS and event sourcing

The lifecycle of aggregate finds this event handler by @EventHandler annotation. FlatStatusProjection object is created and the service that we just discussed above will persist the data that this projection carries.

We concluded the implementation of implementing CQRS and event souring with DDD using axon framework, you can check full code at https://github.com/Anupam1223/DDD-prototype.git.

5. Product Testing

5.1 – Testing of Layered Architecture

5.1.1 – Local environment setup

We run only on service i.e., Homrent because it's monolith

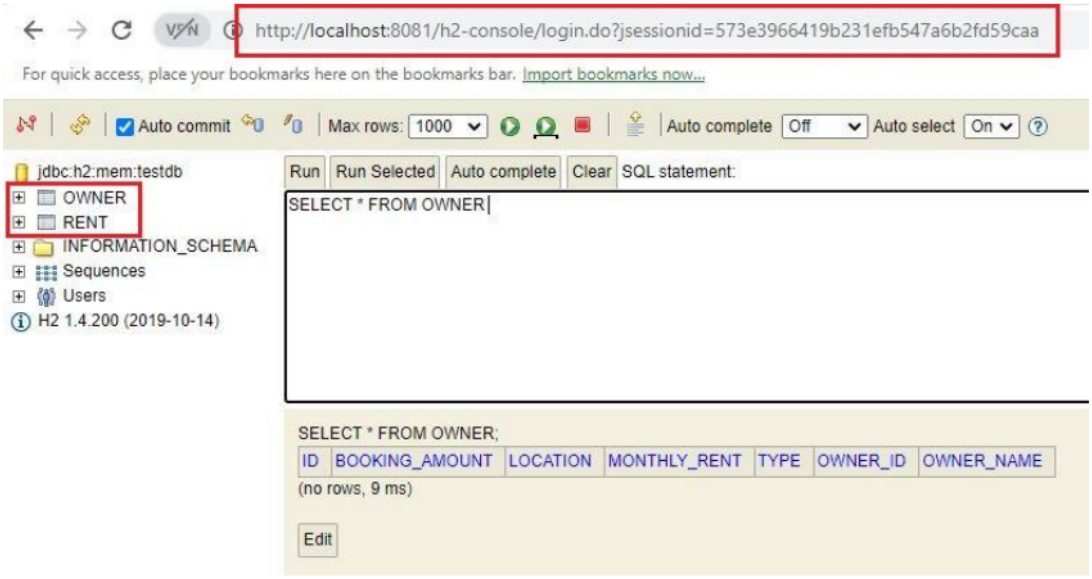


Fig8: local server for Homrent set at 8081

5.1.2 - Requesting our endpoint with JSON payload

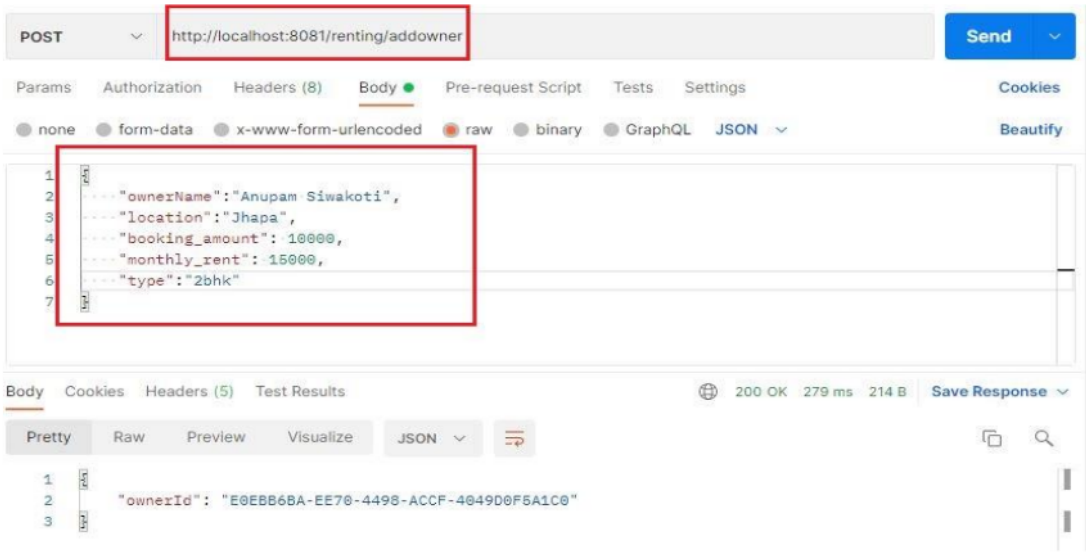


Fig8.1: requesting to add owner on port 8081

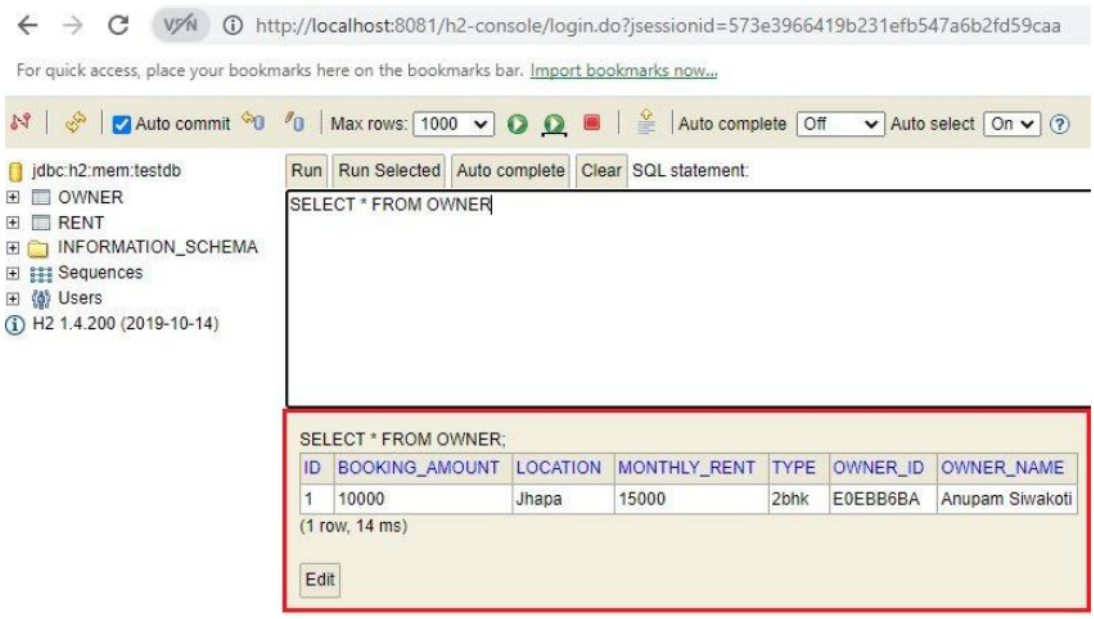


Fig8.2: owner added with above request

Now that we added owner and flat, lets rent this flat

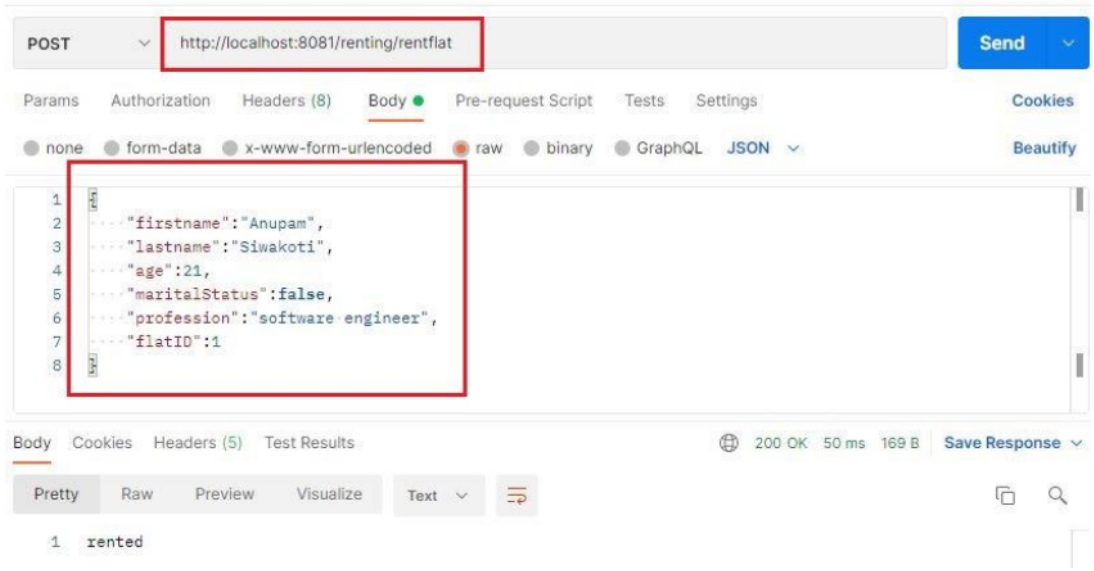


Fig8.3: sending flat renting request on port 8081

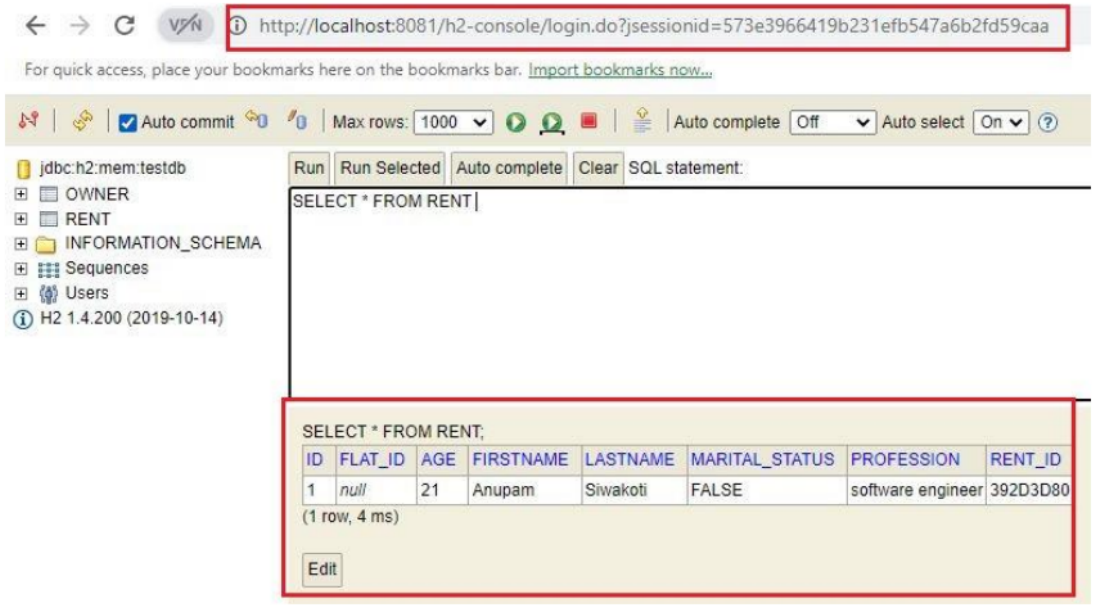


Fig8.4: flat rented with above request

5.2 – Testing of Hexagonal Architecture

5.2.1 - Local environment setup

We run three services, rent, manage, and owner

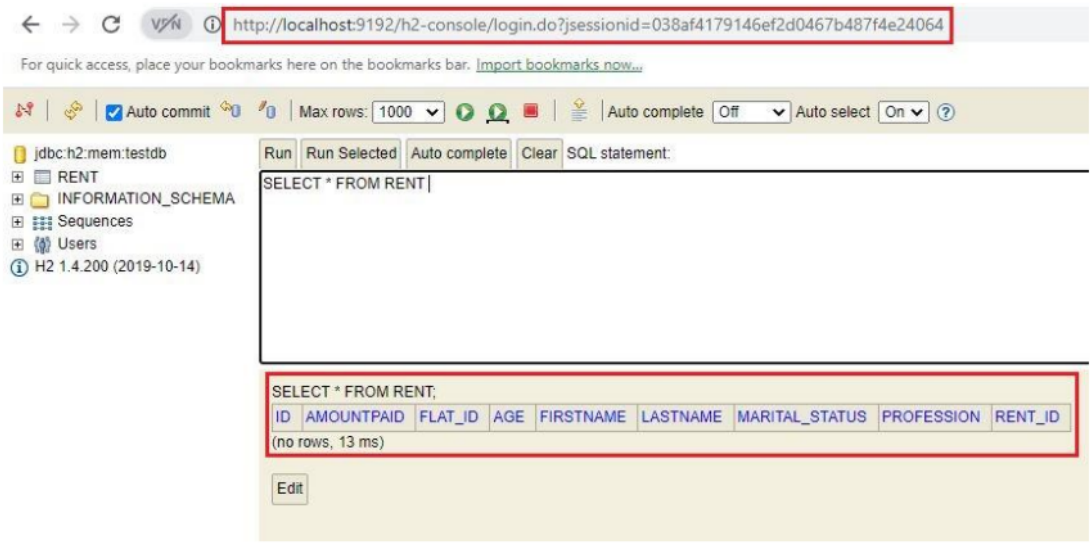


Fig8.5: running rent service in local host

The server for rent bounded context (service) is allocated the port 9192, above figure illustrate that H2 console (in-memory database) for rent is running.

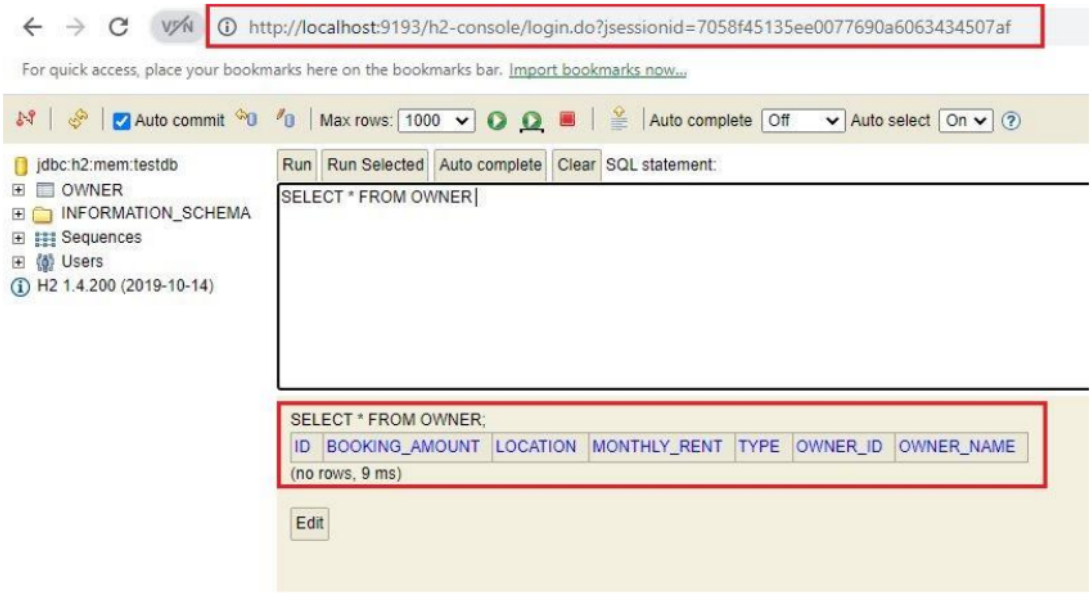


Fig8.6: running owner service in local host

Owner bounded context (service) is running on 9193.

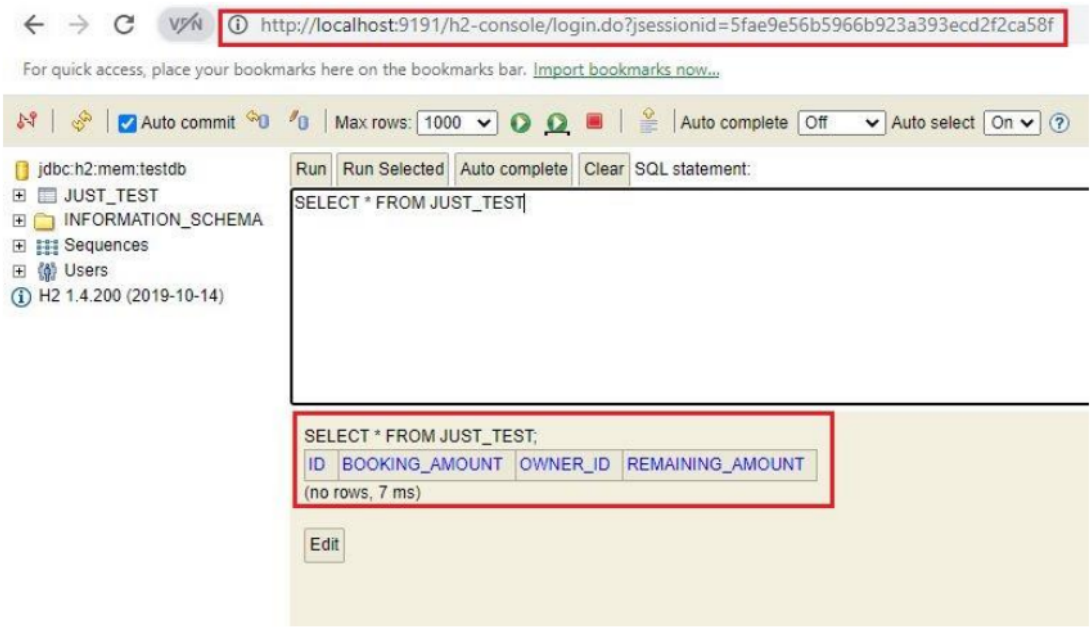


Fig8.7: running manage service in local host

5.2.2 - Requesting our endpoint with JSON payload

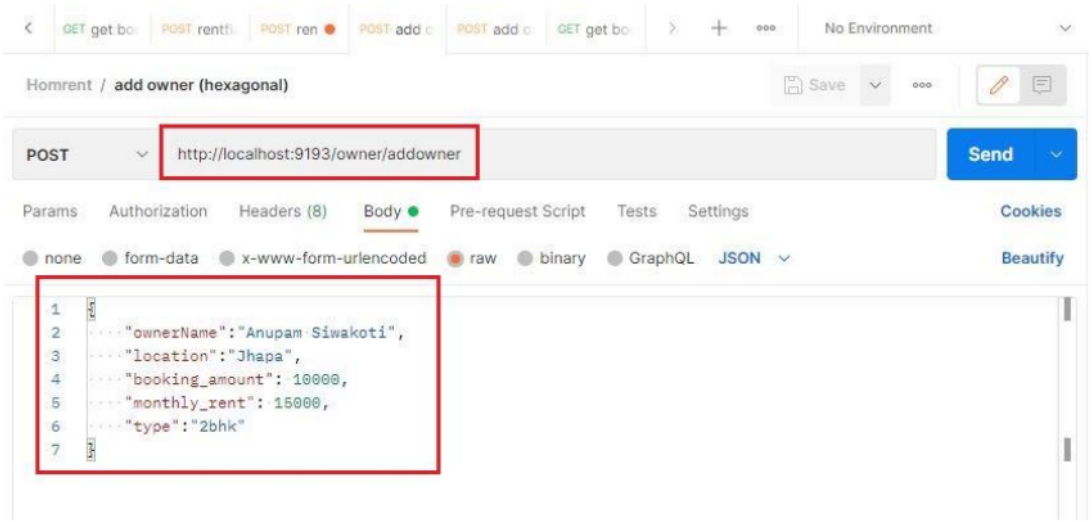


Fig8.8: requesting to add owner on port 9193

Figure above is sending request to addowner with payload

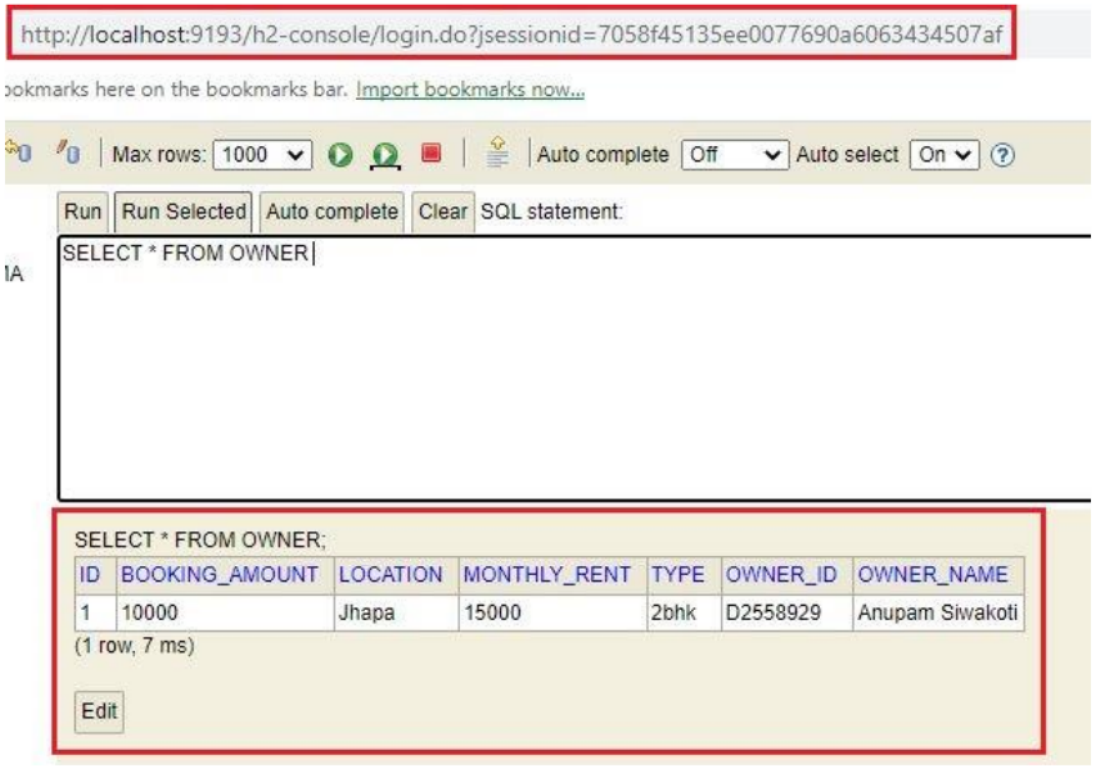


Fig8.9: owner added with above request

Now that the owner is added let's rent this flat.

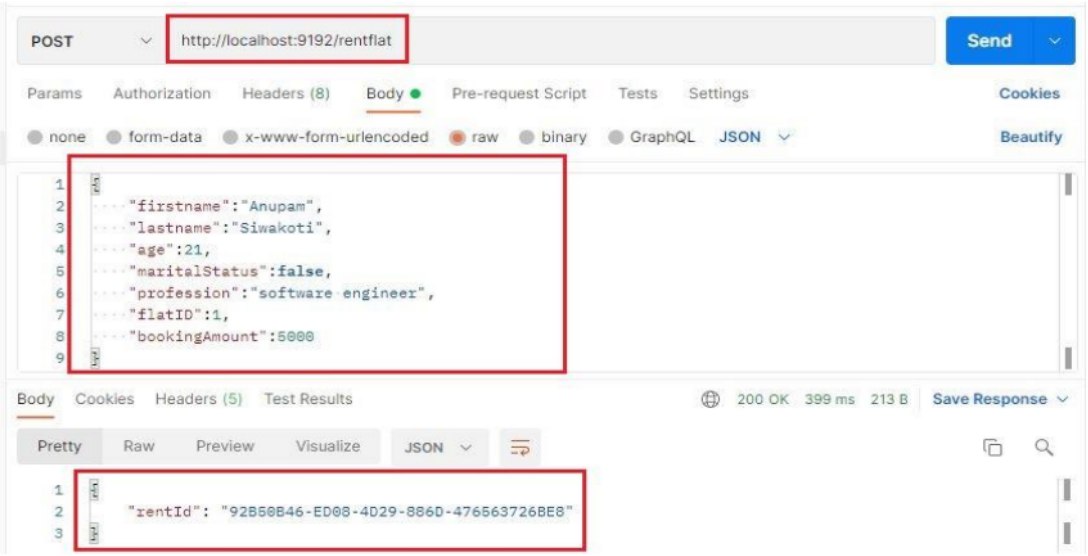


Fig9: requesting to rent flat on port 9192

Figure above sends the renting request, booking amount of the flat is 1000, but we have 5000 in the payload this needs to be handled by manage bounded context. First the request is sent to rent bounded context and flatRented event is raised, which is sent to RabbitMQ message queue. Manage bounded context subscribe to this event and automatically stores that remaining amount as 5000.

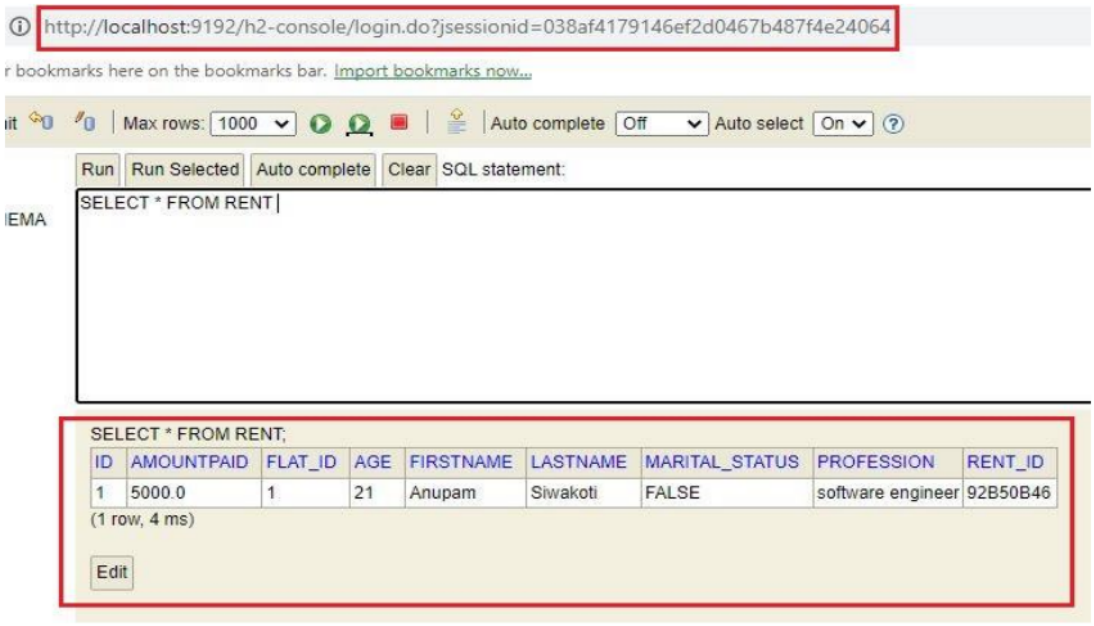


Fig9.1: flat rented

Fig9.2: remaining amount is recorded

5.3 – Testing of CQRS with Event Sourcing

5.3.1 - Local environment setup

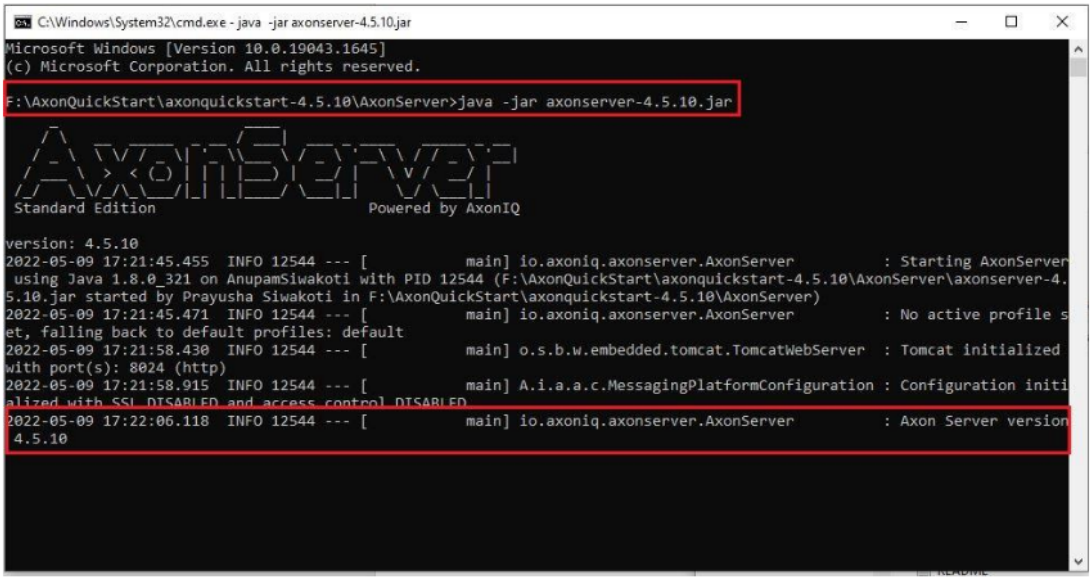


Fig9.3: running axon server

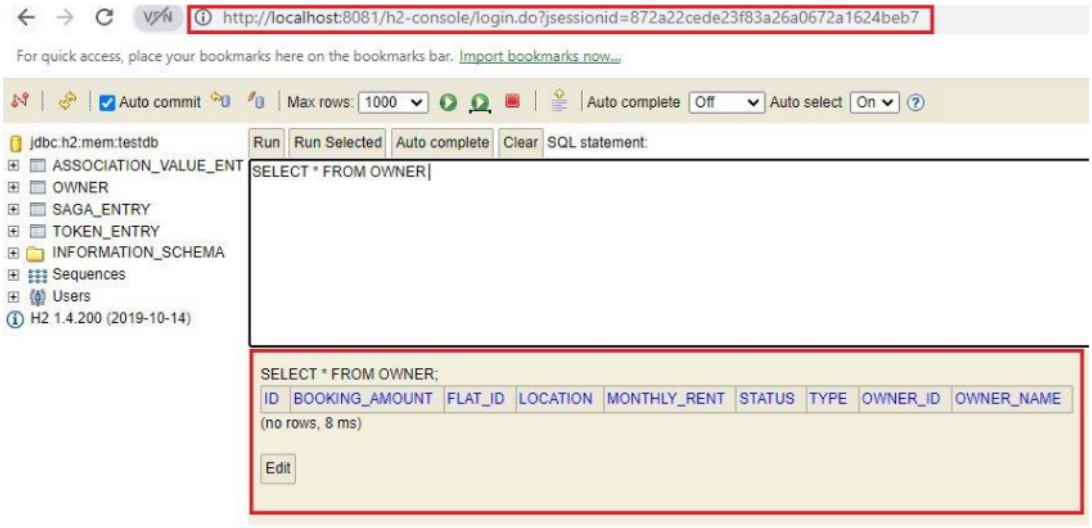


Fig9.4: running owner service on port 8081 in local host

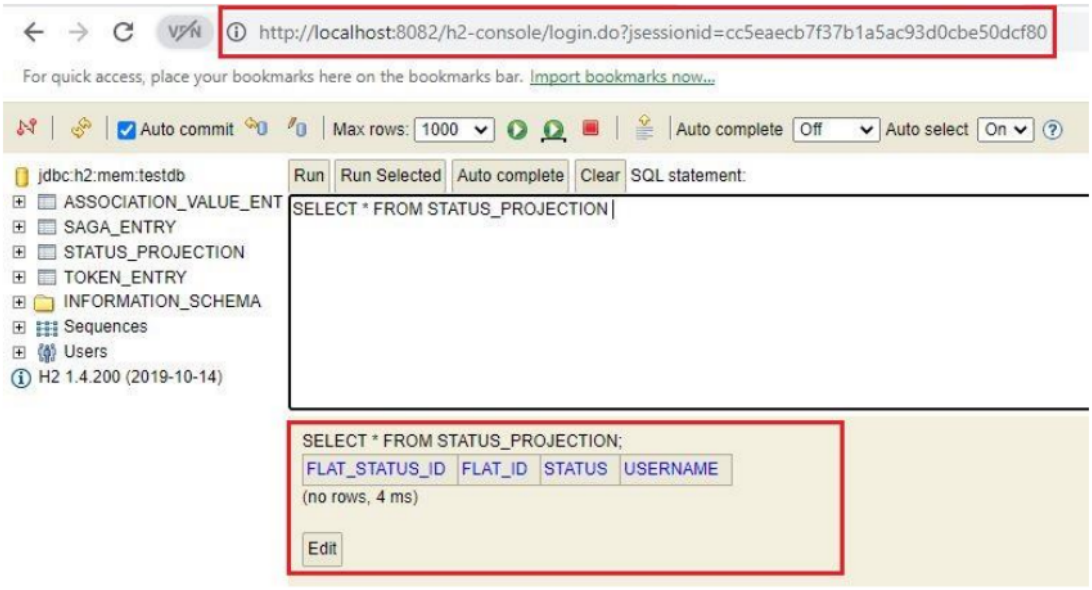


Fig9.5: running manage service on port 8082 in local host

5.3.2 - Requesting our endpoint with JSON payload

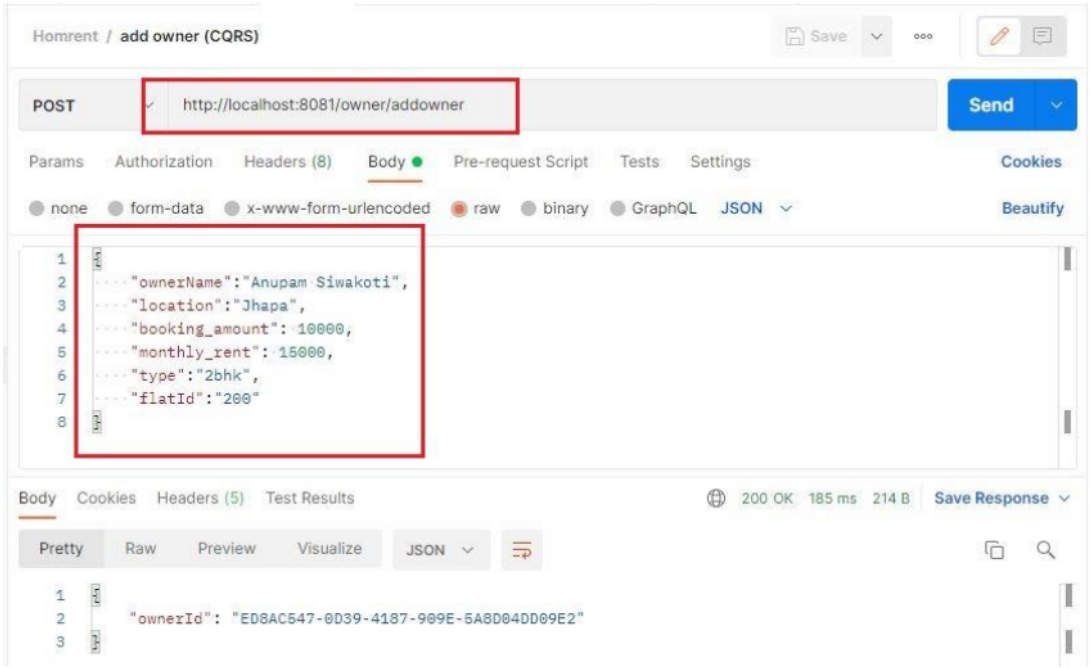


Fig9.6: adding owner requesting from postman

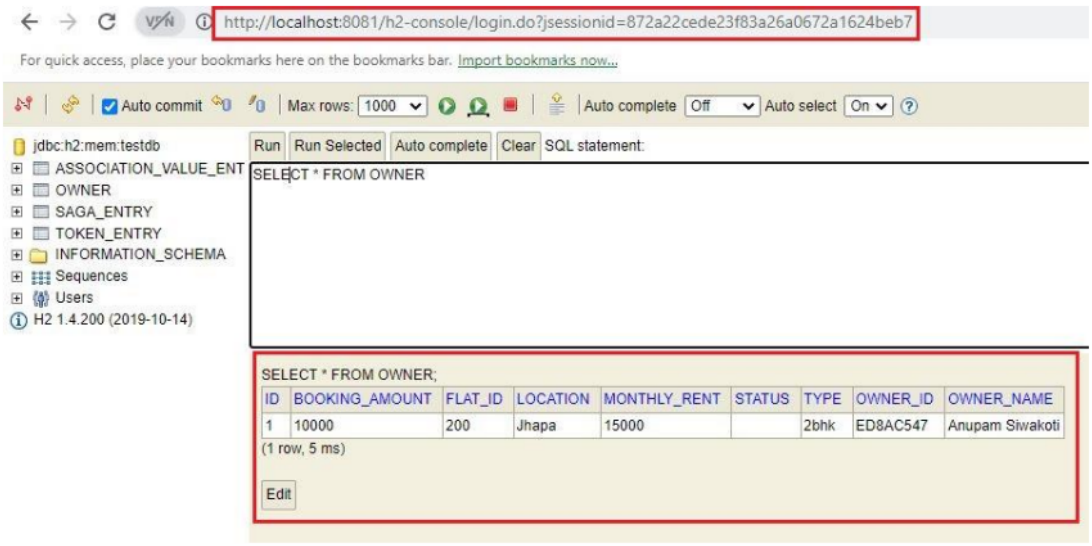


Fig9.7: owner added with request from above picture

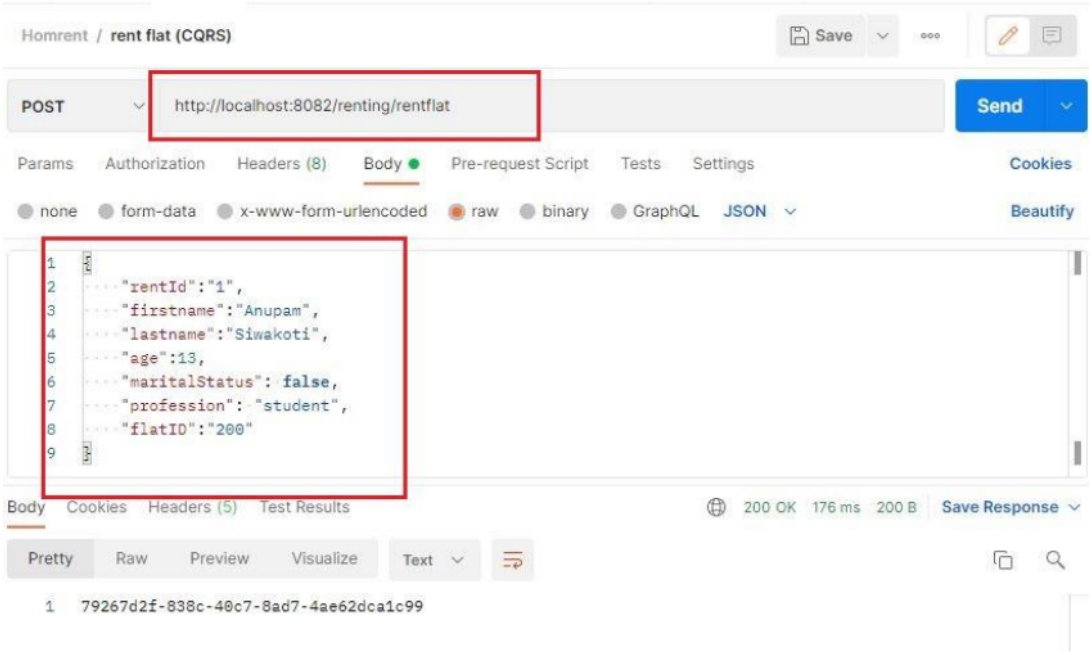


Fig9.8: requesting to rent flat

Flat is rented in event store by axon server, and later its projected into this table

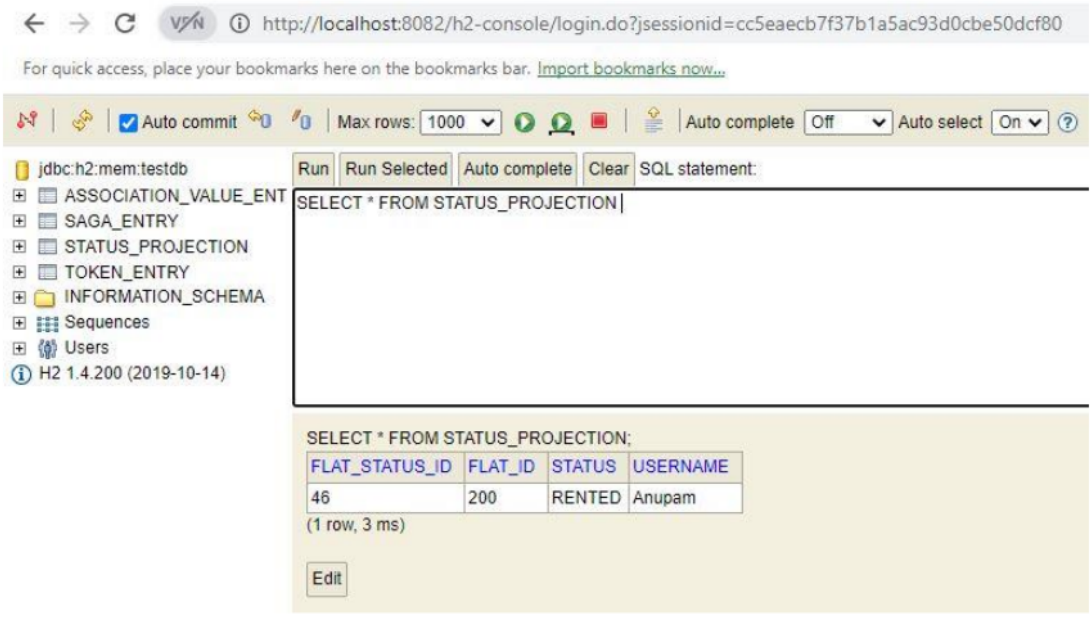


Fig9.9: flat is projected after stored from event store

6. Discussion

We will discuss on these combinations based on Time, Memory, effort, Intelligence required, affective, efficiency. Our prototype is very small, but I got the taste of which architecture is capable of what. So, the prototype may not give huge differences in execution time, memory factor but surely has a framework like the big enterprise application that follows these architectures.

Time

Layered architecture is very simple as compared to hexagonal and CQRS, so time that we need to indulge this architecture is less. But we are implementing the principles of DDD so, most of time is spent in understanding the business logic, converting the logic to logical maps, and converting those logical maps into physical modules. Resources to implement layered arch is not much, single database, simpler API, no message brokers, on top of that its monolith so no cloud communication. But hexagonal architecture and CQRS need extra attention when finally, logic map of the business is ready for its physical form. Again, compared to hexagonal, CQRS and event sourcing needs more time because it completely opposes the traditional way of software programming. Also segregating command side and query side, using projection logic, operating event store, maintaining internal message bus, command bus and the list goes on. There was no execution time difference between layered and hexagonal architecture but in CQRS + ES we can feel the lack of execution time as first data is stored in event store and later projected.

Memory

For this factor, hexagonal architecture wins the race because we have built it as microservices, so load is balanced between all these services which are hosted in the cloud. Maintaining consistency of data between these services may be challenging but memory wise its simpler. CQRS architecture needs to bear pressure in terms of memory because we are indulging it with event sourcing. Stage of every instance of aggregate is preserved, also separate database for command and query, this might need more powerful system and memory.

Affective

CQRS pattern is more affective for businesses today, as analytics is taking over. Segregation gives freedom for people associated with query side, also hitting same database again and again for changing and retrieving may shut the system down for huge enterprises. Hexagonal architecture is equally effective when used as microservices and properly complements domain driven design. The inbound and outbound services of hexagonal properly fits with the command and event system of DDD.

Intelligence required

CQRS and event sourcing is much more advanced, concept it carries along with the external technologies that we may require to achieve this is quite advanced as well. To operate this, we need people having experience on system development for quite long. Hexagonal architecture when developed as microservices also need engineers with ample knowledge on web development, because handling and maintaining relation with services maintained in cloud is quite hard. Also, implementing DDD itself is very challenging.

While discussing about these architectures lets quickly have the pros and cons of event sourcing so that we can conclude on which architecture fits what kind of software's.

- Searching is way more dynamic; we can search with respect to time, status, event etc.,
- History of the flat is well preserved, helps in case of some business error
- Analytics department is overpowered as diverse topic can be analyzed upon.
- Events are source of truth for the business, completely shadows the traditional way of persisting aggregate

But has its own price, some of its downsides are

- Trained expertise with profound knowledge on event driven architecture is required, as it requires new pattern of thought process.
- Every state of an aggregate is preserved which requires more data storage

- Each aggregate must be reviewed with all its stages before aligning the changes, which bears more time and memory
- Not suited for simpler applications only weighs in better when applied for big enterprises applications.

Web based application which are inclined more toward searching and analytics are must to use event driven architecture, but if your system is more store based for example, application that record data for census, event driven architecture is not that helpful.

7. Project Management

Proper management is done for structuring this research, I have included some of the project management tools that I used.

7.1 - Final status project task list

RESEARCH TASK LIST

TASK	STATUS	PRIORITY	DUE DATE	ASSIGNEE	NOTES
Initial Project Plan					
Initial Project Topic	completed	High	1/22/2022	Resham Bdr Pun	
Aims and Objectives	completed	High	1/22/2022	Resham Bdr Pun	
Initial Project management	completed	Medium	1/22/2022	Resham Bdr Pun	
Research Prototype					
Prototype concept	completed	High	6/1/2022	Resham Bdr Pun	*searching concept for prototype*
collection of resources	completed	High	6/1/2022	Resham Bdr Pun	*collection of software's and articles*
Prototype Implementation					
DDD model implementation	completed	High	6/1/2022	Resham Bdr Pun	
DDD model with layered architecture	completed	High	6/1/2022	Resham Bdr Pun	
DDD model with Hexagonal architecture	completed	High	6/1/2022	Resham Bdr Pun	
DDD model with CQRS + ES	completed	High	6/1/2022	Resham Bdr Pun	
Research Report					
literature review and technology review	completed	High	6/5/2022	Resham Bdr Pun	
methodology	completed	Medium	6/5/2022	Resham Bdr Pun	
prototype development	completed	High	6/5/2022	Resham Bdr Pun	
prototype testing	completed	High	6/5/2022	Resham Bdr Pun	
Discussion	completed	High	6/5/2022	Resham Bdr Pun	
Project Management and Evaluation	completed	Medium	6/5/2022	Resham Bdr Pun	
Summary	completed	Medium	6/5/2022	Resham Bdr Pun	
References	completed	Medium	6/5/2022	Resham Bdr Pun	

Fig10: task list with its status on completion of research

7.2 – Final Gantt Chart

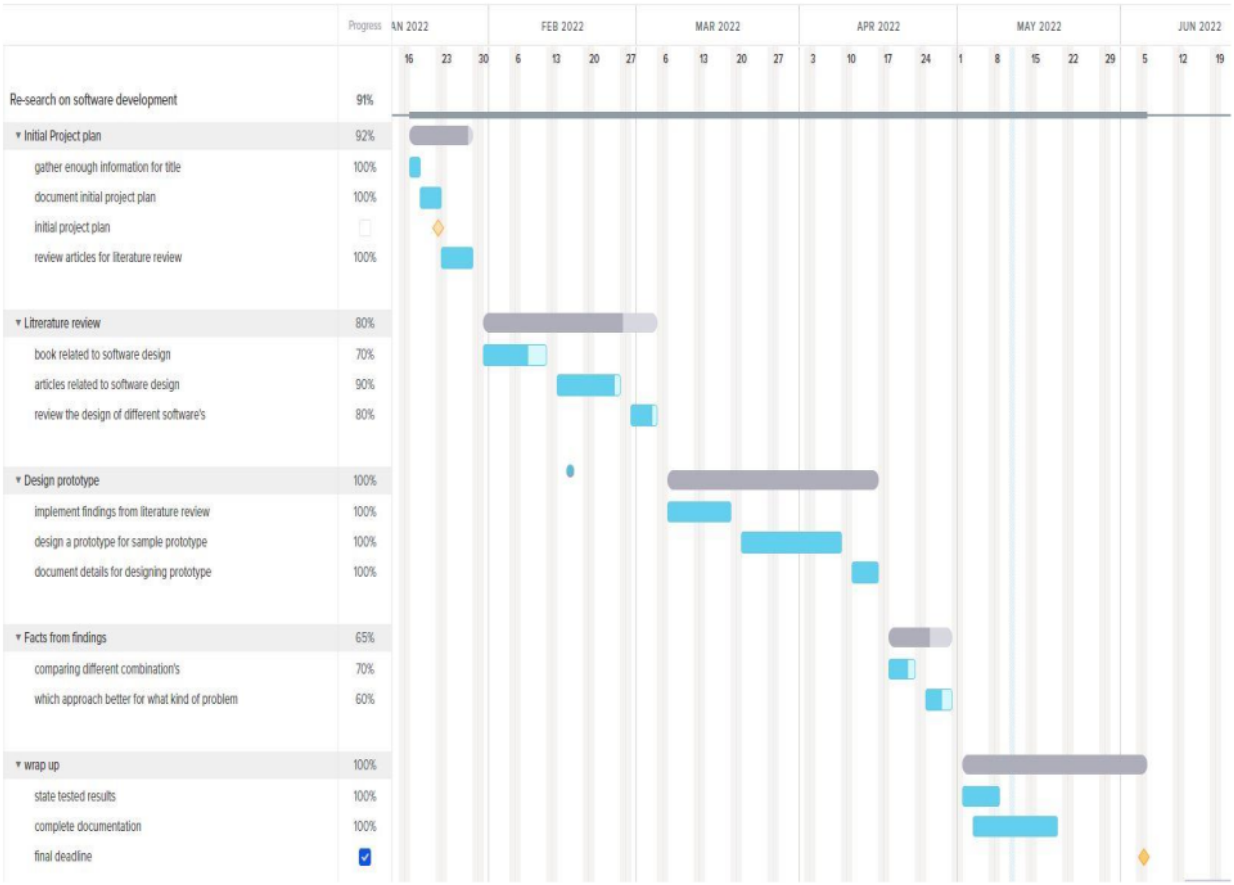


Fig10.1: final Gantt chart after completion of research

7.3 – Work Breakdown Structure (WBS)

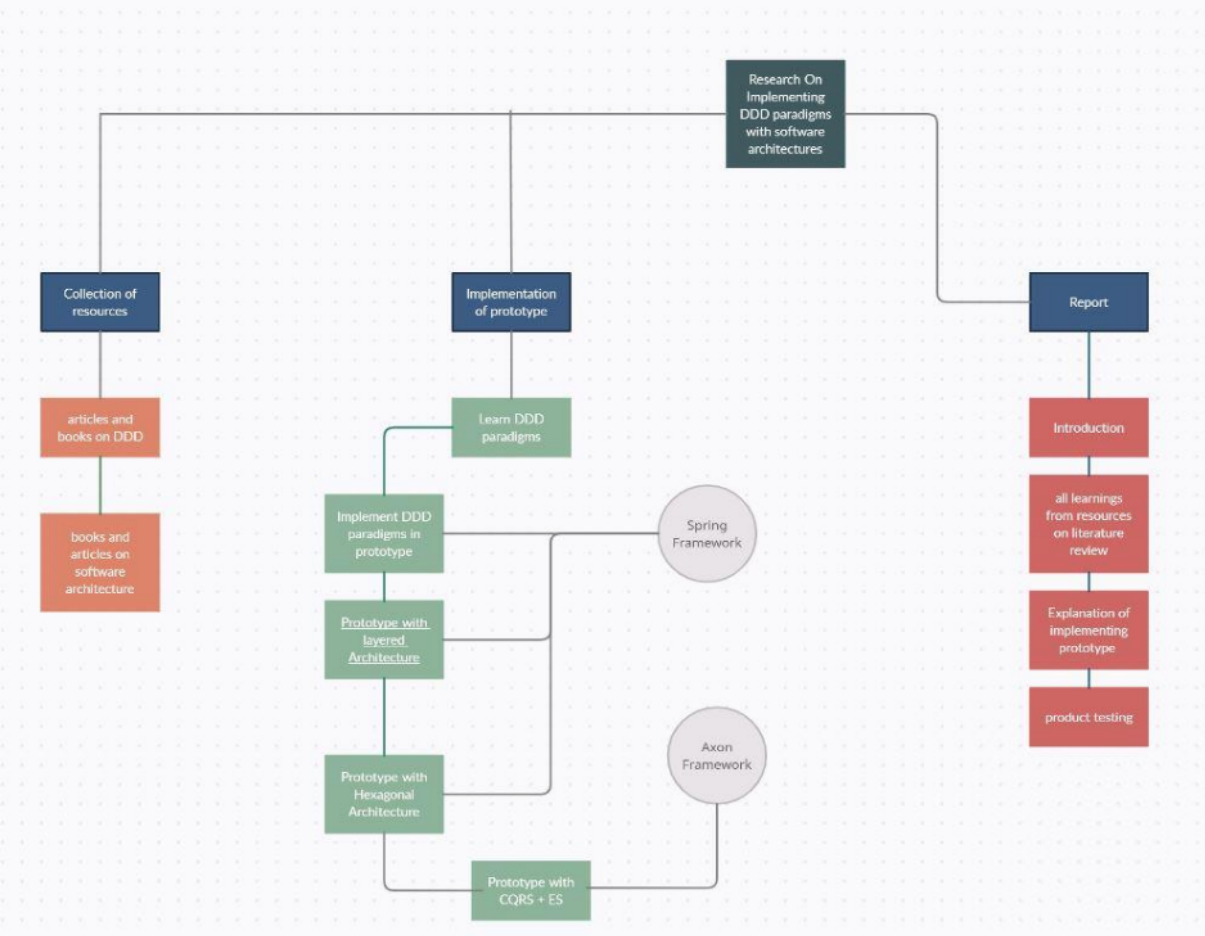


Fig10.2: WBS for our research

8. Summary

The journey from collecting the resources to end of this report took quite a while, it wasn't an easy journey but a fruitful one. We read their book, blogs articles and even Interviews. After that when got a little idea on what DDD may be about we started to look for an idea for prototype that can be followed along with this research. Later I found that not many of this kind of research is performed, also we face language barrier when conducting research in software topics. Examples in internet may be developed in completely different programming languages and it comes with its own difficulties. It took a little time to figure out matching language and framework. Factors such as building project played an important role as we were also implementing microservices, from one of the articles I figured out that maven project under spring framework is very convenient for beginners.

Finalizing the prototype was challenging because we need a problem domain that can address each aspect of what this research demands. We then learn about aggregates (encapsulation over entity and value object) that maintains transactional consistency of a problem domain, entities (objects which have identity of their own and its meaning is defined by the value objects) and value objects which doesn't have identity of their own, their value is their own identity. Also, when talking about aggregate and DDD we must include Events and Commands. Even though events were not captured in the older versions of DDD books, later it was defined and covered as a separate chapter. Events are the cause of commands. Commands changes the status of the aggregate every time it performed successfully. With that knowledge, again it was quite hard to visualize aggregate, entities, commands etc., from our prototype. Further implementation of strategic design (bounded context, context map) helped in understanding the concept of how we separate are of concern.

Once the strategic design of prototype is ready for implementation, now it was important for us to understand the implementation hurdles and benefits of monoliths and microservices. Layered architecture where lower component is completely dependent upon the layer just above it. This was later addressed by hexagonal architecture where we follow dependency inversion in persistence mechanism. Implementing microservices with hexagonal architecture was one of the hardest procedures in this research.

Separating services and maintaining communication between such services in local environment was required. After several google and article searches, I found out about the messaging queue RabbitMQ which is easily adaptable with spring framework. Running each service on different port required configuration in application.yml file. Another challenging task was to implement CQRS + ES with axon framework as it requires complete modification in though process. Segregation of command and query within a service (messaging queue within the service), applying ES with every state preserved where there Creating and updating involved. But with annotation provided in axon framework and its documentation we were able to complete this research and provide some useful outcomes about the combination of these architectures with Domain Driven Design.

9. References

Books

- Avram, A. and Marinescu, F., 2006. Domain-Driven Design Quickly. 2nd ed. [Erscheinungsort nicht ermittelbar]: C4Media.
- Khononov, V., 2021. Learning domain-driven design. 2nd ed. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media.
- Millett, S. and Tune, N., 2015. Patterns, principles, and practice of domain-driven design. 1st ed. Indianapolis, IN: John Wiley & Sons.
- Nair, V., n.d. Practical domain-driven design in enterprise Java. Mountain View, CA, USA: Apress.
- Vernon, V., 2013. Implementing domain-driven design. Upper Saddle River, NJ: Addison-Wesley.

Articles

- Al-Debagy, O. and Martinek, P., 2018. A Comparative Review of Microservices and Monolithic Architectures. 2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI).
- Arthur, J. and Azadegan, S., n.d. Spring Framework for Rapid Open Source J2EE Web Application Development: A Case Study. *Sixth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing and First ACIS International Workshop on Self-Assembling Wireless Networks (SNPD/SAWN'05)*.
- Chang, K. and Lee, C., 2021. Mapping Nearest Neighbor Compliant Quantum Circuits onto a 2-D Hexagonal Architecture. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp.1-1.
- Debski, A., Szczepanik, B., Malawski, M., Spahr, S. and Muthig, D., 2018. A Scalable, Reactive Architecture for Cloud Applications. IEEE Software, 35(2), pp.62-71.
- Nunes, D. and Schwabe, D., 2006. Rapid prototyping of web applications combining domain specific languages and model driven design. Proceedings of the 6th international conference on Web engineering - ICWE '06,
- Uludağ, Ö., Hauder, M., Kleehaus, M., Schimpfle, C. and Matthes, F., 2018. Supporting Large-Scale Agile Development with Domain-Driven Design. Lecture Notes in Business Information Processing, pp.232-247.
- Zhong, Y., Li, W. and Wang, J., 2019. Using Event Sourcing and CQRS to Build a High-Performance Point Trading System. Proceedings of the 2019 5th International Conference on E-Business and Applications - ICEBA 2019,

Online resources

- Blog.nebrass.fr. 2022. *Playing with CQRS and Event Sourcing in Spring Boot and Axon*. [online] Available at: https://blog.nebrass.fr/playing-with-cqrs-and-event-sourcing-in-spring-boot-and-axon/ [Accessed 14 March 2022].
- Calus, B., 2022. *Event Sourcing and CQRS with Axon!* [online] Medium. Available at: https://medium.com/reaktika/event-sourcing-and-cqrs-with-axon-16984f682190 [Accessed 12 March 2022].
- Ch, K. and rakant (2020). *CQRS and Event Sourcing in Java | Baeldung*. [online] www.baeldung.com. Available at: https://www.baeldung.com/cqrs-event-sourcing-java [Accessed 26 March 2022].
- Culttt. (2014). *Strategies for Integrating Bounded Contexts*. [online] Available at: https://www.culttt.com/2014/11/26/strategies-integrating-bounded-contexts [Accessed 1 Feb 2022].

- eventuate.io. (n.d.). *Why Event Sourcing?* [online] Available at: https://eventuate.io/whyeventsourcing.html [Accessed 22 March 2022].
- khalilstemmler.com. (n.d.). *Decoupling Logic with Domain Events [Guide] - Domain-Driven Design w/ TypeScript | Khalil Stemmler*. [online] Available at: https://khalilstemmler.com/articles/typescript-domain-driven-design/chain-business-logic-domain-events/ [Accessed 1 Feb 2022].
- Mitra, S. (2018). *Layered Architecture Up and Running just in 5 minutes:: Spring Boot Part 1 - Java Code Geeks - 2022*. [online] Java Code Geeks. Available at: https://www.javacodegeeks.com/2018/10/layered-architecture-running-just-5-minutes-spring-boot-part-1.html [Accessed 6 Feb 2022].
- Ozkaya, M., 2022. *Layered (N-Layer) Architecture*. [online] Medium. Available at: https://medium.com/design-microservices-architecture-with-patterns/layered-n-layer-architecture-e15ffdb7fa42 [Accessed 3 Feb 2022].
- rreselma (n.d.). *An illustrated guide to CQRS data patterns*. [online] Enable Architect. Available at: https://www.redhat.com/architect/illustrated-cqrs [Accessed 16 March 2022].
- Singh, D., 2022. *Spring Cloud Stream with RabbitMQ: Message-Driven Microservices*. [online] Stack Abuse. Available at: https://stackabuse.com/spring-cloud-stream-with-rabbitmq-message-driven-microservices/ [Accessed 18 Feb 2022].
- Technology, V.C. and (n.d.). *Articles Tutorials | AspNet Boilerplate*. [online] ASP.NET Boilerplate. Available at: https://aspnetboilerplate.com/Pages/Documents/EventBus-Domain-Events [Accessed 20 March 2022].

List of Appendices

Meeting record

School of Computing, Creative Technologies, and Engineering 2021/22 **Level 6 Production Project**	
MEETING RECORD SHEET:	**Meeting Number: 1**
Student: Anupam Siwakoti	**Student I.D.: C7227267**
Date of Meeting: Feb 25, 2022	**Supervisor: Resham Bdr Pun**

Actions agreed at previous meeting (completed or comment):

1	Completion of initial project plan	☐
2		☐
3		☐
4		☐
5		☐
6		☐

Comments of student (if any):
This was the first meeting with our supervisor, I discussed with him about the topic, Software Development with DDD and related architectures. He agreed upon this title and the doorway for my further research.

***ABOVE here** – student to complete before Meeting with supervisor. **BELOW here** – complete at the Meeting.*

Next meeting (date/time)**: March 2nd, 2022**

Agreed Actions to complete before next meeting:

1	Start to read books, blogs, and articles on DDD
2	Start writing the literature review
3	Think about the prototype
4	
5	
6	

Comments of supervisor (if any):
I accepted the topic of the student, now he will start his research on the suggested topic.

School of Computing, Creative Technologies, and Engineering 2021/22 **Level 6 Production Project**	
MEETING RECORD SHEET:	**Meeting Number: 2**
Student: Anupam Siwakoti	**Student I.D.: C7227267**
Date of Meeting: March 2nd, 2022	**Supervisor: Resham Bdr Pun**

Actions agreed at previous meeting (completed or comment):

1	Started with the research (reading books etc.,)	☐
2	Came up with the prototype idea (Homrent)	☐
3	Started writing literature review	☐
4	Local setup for prototype	☐
5		☐
6		☐

Comments of student (if any):

This was the second meeting with our supervisor, I discussed with him about my prototype idea, A home renting software using all the architecture which I am going to discuss.

***ABOVE here** – student to complete before Meeting with supervisor. **BELOW here** – complete at the Meeting.*

Next meeting (date/time)**: March 5th, 2022**

Agreed Actions to complete before next meeting:

1	Start the development of layered architecture
2	Continuing writing literature review
3	
4	
5	
6	

Comments of supervisor (if any):

I accepted the prototype which he is continuing throughout the report. His literature review reflects the knowledge that he gained from what he has learned.

School of Computing, Creative Technologies, and Engineering 2021/22

Level 6 Production Project

MEETING RECORD SHEET:	**Meeting Number: 3**
Student: Anupam Siwakoti	**Student I.D.: C7227267**
Date of Meeting: March 5th, 2022	**Supervisor: Resham Bdr Pun**

Actions agreed at previous meeting (completed or comment):

1	Using spring framework for layered architecture	☐
2	DDD concepts are implemented (which will be same for all architectures)	☐
3	Continuing with writing the literature review	☐
4		☐
5		☐
6		☐

Comments of student (if any):

This was the third meeting with our supervisor, he saw the progress.

ABOVE here – student to complete before Meeting with supervisor. BELOW here – complete at the Meeting.

Next meeting (date/time)**: March 12th, 2022**

Agreed Actions to complete before next meeting:

1	Local server setup, database management (in memory)
2	Write down all the step-up procedure for the project setup
3	
4	
5	
6	

Comments of supervisor (if any):

He is continuing with implementing prototype along with his learnings.

School of Computing, Creative Technologies, and Engineering 2021/22 **Level 6 Production Project**		
MEETING RECORD SHEET:		**Meeting Number: 4**
Student: Anupam Siwakoti	**Student I.D.: C7227267**	
Date of Meeting: March 12th, 2022	**Supervisor: Resham Bdr Pun**	
Actions agreed at previous meeting (completed or comment):		
1	**Completed the implementation of layered architecture with DDD**	☐
2	**Started to learn about hexagonal architecture**	☐
3	**Reviewing architecture of different software's**	☐
4		☐
5		☐
6		☐

Comments of student (if any):
This was the fourth meeting with our supervisor, he reviewed the implementation of the layered architecture. He was positive on the progress

ABOVE here – student to complete before Meeting with supervisor. BELOW here – complete at the Meeting.

Next meeting (date/time)**: March 21st, 2022**

Agreed Actions to complete before next meeting:	
1	Learn on about hexagonal architecture
2	Write down all the step-up procedure for the project setup
3	Learn about the message broker for pub-sub
4	
5	
6	

Comments of supervisor (if any):
No comments, I saw his layered architecture, he is on progress.

School of Computing, Creative Technologies, and Engineering 2021/22

Level 6 Production Project

MEETING RECORD SHEET:	**Meeting Number: 5**
Student: Anupam Siwakoti	**Student I.D.: C7227267**
Date of Meeting: March 21st, 2022	**Supervisor: Resham Bdr Pun**

Actions agreed at previous meeting (completed or comment):

1	Learned about hexagonal architecture	☐
2	Started implementing hexagonal architecture with spring	☐
3		☐
4		☐
5		☐
6		☐

Comments of student (if any):

This was the fifth meeting with our supervisor, he said to continue it.

***ABOVE here** – student to complete before Meeting with supervisor. **BELOW here** – complete at the Meeting.*

Next meeting (date/time)**: March 27th, 2022**

Agreed Actions to complete before next meeting:

1	Complete implementation of hexagonal architecture
2	If possible, start implementation of CQRS and Event sourcing
3	
4	
5	
6	

Comments of supervisor (if any):

Continuous progress on his prototype

School of Computing, Creative Technologies, and Engineering 2021/22

Level 6 Production Project

MEETING RECORD SHEET:	**Meeting Number: 6**
Student: Anupam Siwakoti	**Student I.D.: C7227267**
Date of Meeting: March 27th, 2022	**Supervisor: Resham Bdr Pun**

Actions agreed at previous meeting (completed or comment):

1	Completed hexagonal architecture implementation	☐
2	Started implementing CQRS and Event sourcing	☐
3		☐
4		☐
5		☐
6		☐

Comments of student (if any):

This was the sixth meeting with our supervisor, he said to continue it.

ABOVE here – student to complete before Meeting with supervisor. BELOW here – complete at the Meeting.

Next meeting (date/time)**: April 3rd, 2022**

Agreed Actions to complete before next meeting:

1	Complete implementation of CQRS and event sourcing
2	Start writing research report, this should include the findings
3	
4	
5	
6	

Comments of supervisor (if any):

Continuous progress on his prototype

School of Computing, Creative Technologies, and Engineering 2021/22

Level 6 Production Project

MEETING RECORD SHEET:	**Meeting Number: 7**
Student: Anupam Siwakoti	**Student I.D.: C7227267**
Date of Meeting: April 3rd, 2022	**Supervisor: Resham Bdr Pun**

Actions agreed at previous meeting (completed or comment):

1	Completed CQRS and Event sourcing implementation	☐
2		☐
3		☐
4		☐
5		☐
6		☐

Comments of student (if any):

This was the seventh meeting with our supervisor, he said to continue it.

ABOVE here – student to complete before Meeting with supervisor. BELOW here – complete at the Meeting.

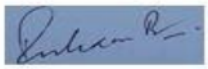
Next meeting (date/time)**: April 10th, 2022**

Agreed Actions to complete before next meeting:

1	Complete fifty percent implementation of report
2	
3	
4	
5	
6	

Comments of supervisor (if any):

Continuous progress on report writing



School of Computing, Creative Technologies, and Engineering 2021/22

Level 6 Production Project

MEETING RECORD SHEET:	**Meeting Number: 8**
Student: Anupam Siwakoti	**Student I.D.: C7227267**
Date of Meeting: May 6th, 2022	**Supervisor: Resham Bdr Pun**

Actions agreed at previous meeting (completed or comment):

1	Completed above 50% report	☐
2		☐
3		☐
4		☐
5		☐
6		☐

Comments of student (if any):

This was the eight meeting with our supervisor, he said to continue it.

***ABOVE here** – student to complete before Meeting with supervisor. **BELOW here** – complete at the Meeting.*

Next meeting (date/time)**: May 9th, 2022**

Agreed Actions to complete before next meeting:

1	On next meeting complete the report and submit a draft for review
2	
3	
4	
5	
6	

Comments of supervisor (if any):

Continuous progress on report writing

final report on empirical research

ORIGINALITY REPORT

5%	4%	1%	4%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Submitted to The British College Student Paper	1%
2	Submitted to The Robert Gordon University Student Paper	1%
3	Submitted to Brunel University Student Paper	1%
4	Vijay Nair. "Practical Domain-Driven Design in Enterprise Java", Springer Science and Business Media LLC, 2019 Publication	<1%
5	Andrzej Debski, Bartlomiej Szczepanik, Maciej Malawski, Stefan Spahr, Dirk Muthig. "In Search for a Scalable & Reactive Architecture of a Cloud Application: CQRS and Event Sourcing Case Study", IEEE Software, 2017 Publication	<1%
6	Submitted to Swinburne University of Technology Student Paper	<1%
7	herbertograca.com Internet Source	

		<1 %
8	github.com Internet Source	<1 %
9	www.studymode.com Internet Source	<1 %
10	Submitted to Gayatri Vidya Parishad College of Engineering (Autonomous) Student Paper	<1 %
11	archive.org Internet Source	<1 %
12	landongn.com Internet Source	<1 %
13	d.lib.msu.edu Internet Source	<1 %
14	www.simplilearn.com Internet Source	<1 %
15	codurance.com Internet Source	<1 %
16	www.javacodegeeks.com Internet Source	<1 %
17	courses.iicm.tugraz.at Internet Source	<1 %

Exclude quotes	On	Exclude matches	Off
Exclude bibliography	On		