Tunisian republic
Ministry of Higher Education and Scientific Research
University of Tunis El Manar
National Engineering School of Tunis
Information Technology and Communication Department

End Of The Second Year Project

Design and development of a ride-sharing platform in a microservice architecture (passenger-side)

Developed by

ROUATBI Abdelhamid **KHOMSI Asma**

Class : 2nd Year Computer science

Supervised By

Mr. HADDAD Mohammed Ramzi

Academic year : 2022/2023

Acknowledgement

First and foremost, we would like to dedicate this section to thank the following individuals and organizations for their precious contribution to the realization of this project.

Thus, our deepest thanks go to Mr. HADDAD Mohammed Ramzi, our project supervisor, whose countless advices and careful support were a most valuable asset throughout the whole project.

Our thanks also goes to all our professors in ENIT, for all their efforts for the sake of our training and education.

Finally, we would like to thank anyone else who contributed to this project in ways big or small. Your support and feedback were greatly appreciated.

Abstract

This is the report of the second end-of-year project of Software Engineering in the National School of Engineering of Tunis. It includes the design and implementation of the passenger-side of a ride-sharing platform in a microservices architecture.

Keywords: Microservices, Event Storming, Domain-Driven Design, Event-Driven Architecture, Onion Architecture.

Contents

List of Figures

List of Acronyms

DDD Domain-Driven Design

TNC Transportation Network Company

AWS Amazon's Web Services

ELB Elastic load balancer

PHP Hypertext Preprocessor

API Application Programming Interface

EDA Event-Driven Architecture

IoT Internet of Things

DAO Data Access Object

DAL Data Access Layer

VO Value Object

JSON JavaScript Object Notation

BSON Binary Javascript Object Notation

IDE Integrated Development Environment

IAM Identity and Access Management

UI User Interface

General introduction

The concept of a microservices architecture emerged in the early 2010s and has gained a significant popularity thanks to the rise of cloud computing and containerization technologies, and a widespread adoption due to their ability to support continuous delivery and deployment, simplify testing and debugging and improve overall system resilience and fault tolerance. Many large organizations, such as Netflix, Amazon and eBay, have adopted microservices architectures to support their complex, distributed applications. In addition, Domain-Driven Design is a software design approach that emphasizes the understanding of the business domain.

In this context, our project is the design and development of a ride-sharing platform that adopts a microservice architecture and a Domain-Driven Design approach, which allows riders and passengers to look for and arrange trips.
The first chapter consists of a case study of similar previous works and the specifications.
The second chapter introduces the theoretic concepts that were implemented in the project.
The third chapter shows the procedure of the system's design.
In the last chapter, we talk about the technology stack and present a brief demonstration of the functionalities of the platform.

Chapter 1

Specifications and related platforms

Introduction

The goal of this chapter is to study and analyse already existing applications that relate to the field of ridesharing, followed by the specifications.

1.1 Lyft

Lyft is a transportation network company (TNC) that operates a ride-sharing platform, connecting passengers who need a ride with drivers who own a car and are willing to provide a ride. The Lyft platform allows passengers to request rides through a mobile app, and drivers can accept ride requests and earn money by providing transportation services.
It was founded in 2012 and is headquartered in San Francisco, California. It operates in the United States and Canada, providing services in over 600 cities across both countries. In addition to ride-sharing services, Lyft has expanded into other transportation services, such as bike-sharing and scooter-sharing, and is also working on developing autonomous vehicles.[6]

1.1.1 Monolithic Beginnings

When Lyft was first released in 2012, it was built using a basic monolithic architecture. Clients would interact through the internet with Amazon's Web Services Elastic Load Balancer (AWS ELB), which would in turn send traffic to a PHP/Apache monolith that interacts with a MongoDB database.[5]

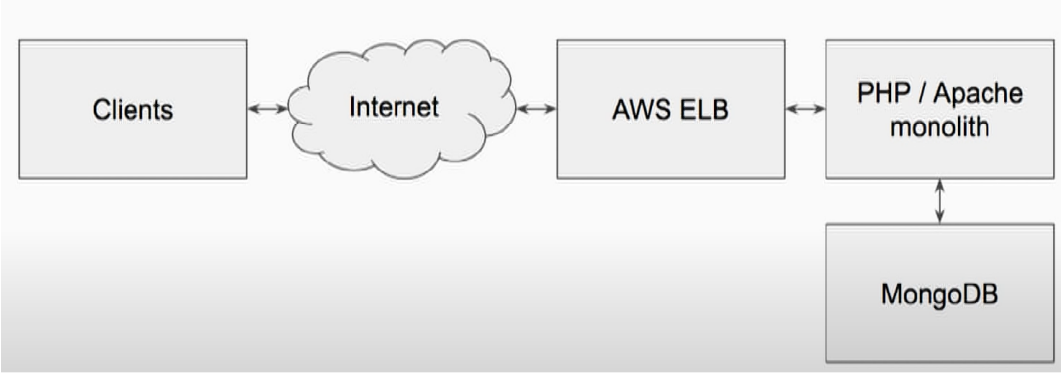

Figure 1.1: Lyft's monolithic architecture.[5]

1.1.2 The need for a new architecture

Comes 2014, Lyft was successful and rising in both popularity and complexity.
As a result, many issues started arising. One such issue was that of the load balancer: PHP and Apache implement a process per connection model. This means it can only handle a limited number of connections at the same time.
However, the load balancer doesn't take this into account, and as a result creates too many pre-connections for the backend to handle when traffic is high. Lyft had to ask AWS to turn off some features such as pre-connections, which hindered performance.
Lyft would eventually resolve to switch its monolithic architecture to a service-oriented architecture.

1.1.3 Service-Oriented Architecture

By 2015, Lyft had partially switched to a service-oriented architecture where the monolith now makes service calls to new python back end services, a DynamoDB database was added and an internal ELB now handles traffic between the different services.
From a deployment standpoint, it was more complicated than ever: Lyft had to use a lot of tooling, such as HA Proxy to complement the serving architecture of PHP not allowing to make concurrent connections and as well as introducing queing scripts for NSQ, a distributed messaging platform that provides a queuing system for building distributed systems.
Another major problem that emerged was that of observability. Lyft had no means of statistics logging or traceability, which made debugging a very tedious task for developers.

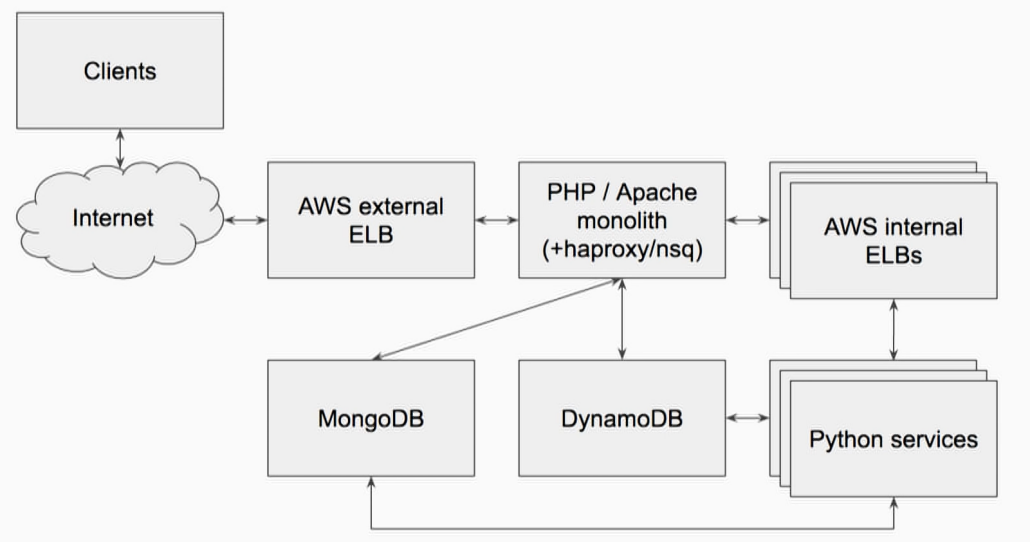

Figure 1.2: Lyft's service-oriented architecture.[5]

From a developer's standpoint, the growing complexity of the system was translated to a diverse use of programming languages, libraries, frameworks and protocols. This made it impossible for developers to properly understand what is available, what is happening and what the best practices are.

In September 2016, Lyft announced what would become the answer to all its problems: Envoy Proxy, a high-performance, open-source service proxy designed for cloud-native applications. It provides a flexible and scalable framework for managing service-to-service communication, including load balancing, health checking, service discovery, rate limiting, circuit breaking, and distributed tracing. Overall, Envoy is a powerful and flexible service proxy that provides a

wide range of features for managing service-to-service communication in modern, cloud-native architectures. Today, Lyft has totally revamped its architecture to a microservices architecture.

1.2 Uber

Uber is a technology company that operates a ride-hailing platform connecting riders with drivers through a smartphone app. The company was founded in 2009 in San Francisco, California, and has since expanded to operate in over 700 cities worldwide. Through the Uber app, riders can request a ride from a nearby driver, track the driver's location, and pay for the ride automatically through the app.
Uber has also expanded into other areas such as food delivery (Uber Eats), electric bikes and scooters (Uber Jump), and freight transportation (Uber Freight).

1.2.1 From monolith to microservices

Prior to the transition to a microservices architecture, Uber primarily had two monolithic services. In 2012, Uber's growth started having its impacts on society, as it expanded to new cities and gained traction among consumers. Its engineering department grew from tens to hundreds of engineers with multiple teams. As a result, operational issues were becoming a daily hassle.[2]

- Availability Risks: A single regression within the monolithic architecture implied the whole system would be brought down.
- Risky and expensive deployments: Since monolithic architecture must deploy or release all of its code at once, deployments became painful and time consuming to perform with the frequent need for rollbacks.
- Poor separation of concerns: With a huge code base, it was difficult to set boundaries between logic and components.

All these issues combined made autonomous and independent work between the teams nearly impossible, which is very inefficient. This made the switch to a microservices architecture the optimal move for Uber, which it started working on in 2014. As a result, Uber's systems became more flexible, which allowed teams to be autonomous and thus accelerated the development of new features.

1.2.2 Limits of Uber's microservices architecture

The transition to microservices allowed Uber to continue its exponential growth. In 2015, it had expanded in over 300 cities worldwide with a particularly strong presence in North America, Europe, and Asia. The company continued to aggressively pursue international expansion in the years that followed, entering new markets and launching new services like UberEATS, which allowed it to expand beyond ride-hailing and into the broader logistics and delivery space.[2]
Along with this success of course came more complex systems. As the company grew even larger, from hundreds to thousands of engineers, new sets of issues started taking place with greatly increased system complexity. Understanding dependencies and communications between services becomes challenging as calls can go many layers deep, causing a latency spike to produce a cascade of issues, and without proper tools, visibility becomes unclear, making debugging tougher than it should be. Additionally, engineers often have to work across multiple services owned by other teams. This requires extensive collaborative work in meetings,

designs and code reviews. What was once a clear separation of services is now clouded as teams code within each other's services. As a result, migrations became once again painful, and the development process slowed down.

1.2.3 The Domain-Oriented approach

By that time, the concept of domain drive design was slowly gaining traction and recognition in the software development community. Moreover, many software development teams had already started adopting DDD as a way to manage complexity and improve alignment with business requirements. Several case studies and success stories of organizations adopting DDD were already available. Thus, Uber opted for the domain-driven solution, which is very effective to handle very complex and large systems. Instead of treating microservices as single, independent entities, they are now grouped around collections of related microservices called Domains. Each domain is designed to belong to a certain layer, which is a collection of domains that dictates what dependencies its microservices are allowed to take on. Each domain is provided a single point of entry into the collection called Gateway. Finally, each domain should not implement logic that belongs to other domains. Since teams often need to include logic in another team's domain anyway, an extension architecture is provided to support well defined extension points within the domain. With this, Uber was able to transform its microservices architecture from a massively complex mess to a more understandable, clear, structured and flexible architecture.[2]

1.3 Specifications

Event Storming is a technique used in software design to quickly identify and explore complex business processes, systems, and models. It is basically a communicative brainstorming method in which knowledge and understanding of a specific, delimited field of knowledge (a domain of expertise) is jointly developed and visualised in a workshop. It's important for an event storming workshop to have the right people present. This includes people who know the questions to ask (typically developers) and those who know the answers (domain experts, product owners). The process involves creating a visual representation of the system on a large wall or whiteboard, using colored stickerse to represent different types of events and actions.

- People who know the questions to ask: Developers, Testers
- People who know the answers: Domain Experts, Product Owners, Clients
- A good moderator and facilitator: Scrum Master, Agile Coach

1.3.1 Terminology

Before going further, it is important to establish the terminology used in an event storming. For this, participants need to understand the meanings of each sticker color.

- Domain events are colored in orange. They are events that occur in the business process. Written in past tense.
- Users and actors are colored in pale yellow. They represent a person who executes a command through a view.
- Business processes are colored in purple. They represent business rules and logic.

- Commands are colored in a sky blue. They represent commands executed by a user through a view on an aggregate that result in the creation of a domain event.
- Aggregates are colored in yellow. They represent a cluster of domain objects that can be treated as a single unit.
- External systems are colored in pink. They represent third-party service providers such as a payment gateway or shipping company.
- Views and read models are colored in green. They represent a view that users interact with to carry out a task in the system.
- Question marks and risks are colored in red. They represent unclear topics or questions that arise during the session.

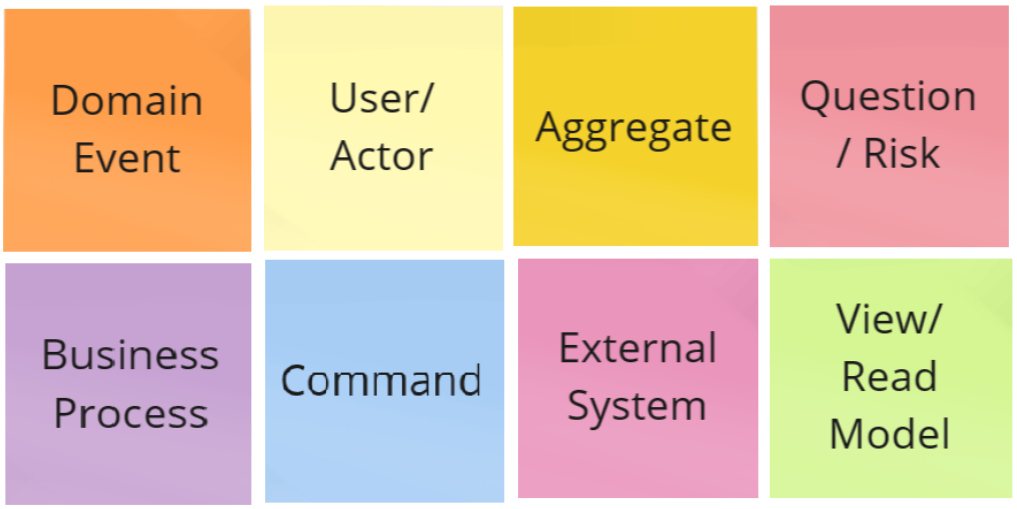

Figure 1.3: Event Storming Legend

1.3.2 Why event storming ?

When compared to other requirements gathering and design methods, such as use cases, it is clear why an event storming is usually the best fit for a domain-driven approach.

- Business domain focus: Event Storming is particularly well-suited for exploring and modeling complex business domains. It allows teams to gain a deep understanding of the business processes and workflows that the software will support, which can help to avoid misunderstandings and missed requirements.
- Rapid ideation: Event Storming is a highly collaborative and interactive process that allows stakeholders and team members to quickly generate ideas and identify potential issues or opportunities. It encourages a creative and open-minded approach to problem-solving.
- Cross-functional involvement: Event Storming encourages participation from stakeholders and team members with a variety of backgrounds and perspectives. This can help to ensure that all relevant viewpoints are considered and that the resulting software is well-designed and user-friendly.

- Visualization: Event Storming produces visual artifacts, such as sticky notes on a wall or a diagram on a whiteboard, that can help stakeholders and team members to better understand and communicate complex ideas. These visual artifacts can also serve as a reference throughout the development process, helping to ensure that everyone is aligned on the project goals and requirements.
- Agile-friendly: Event Storming is well-suited for use in agile development environments, where rapid iteration and continuous feedback are key. It can help teams to quickly identify and address issues or opportunities, and to adapt their plans as needed.

1.3.3 Spicifications identification

The first step of an event storming session is the collection of domain events. Each participant uses only orange Post-Its in the first round. Each orange Post-it stands for a professional event. A professional event is a technical relevant fact that happened in the course of business. The verb on the Post-it must therefore be in the past. The first round is a pure brain storming process about the existing domain events. Events are sorted in the chronological order in which they occur.

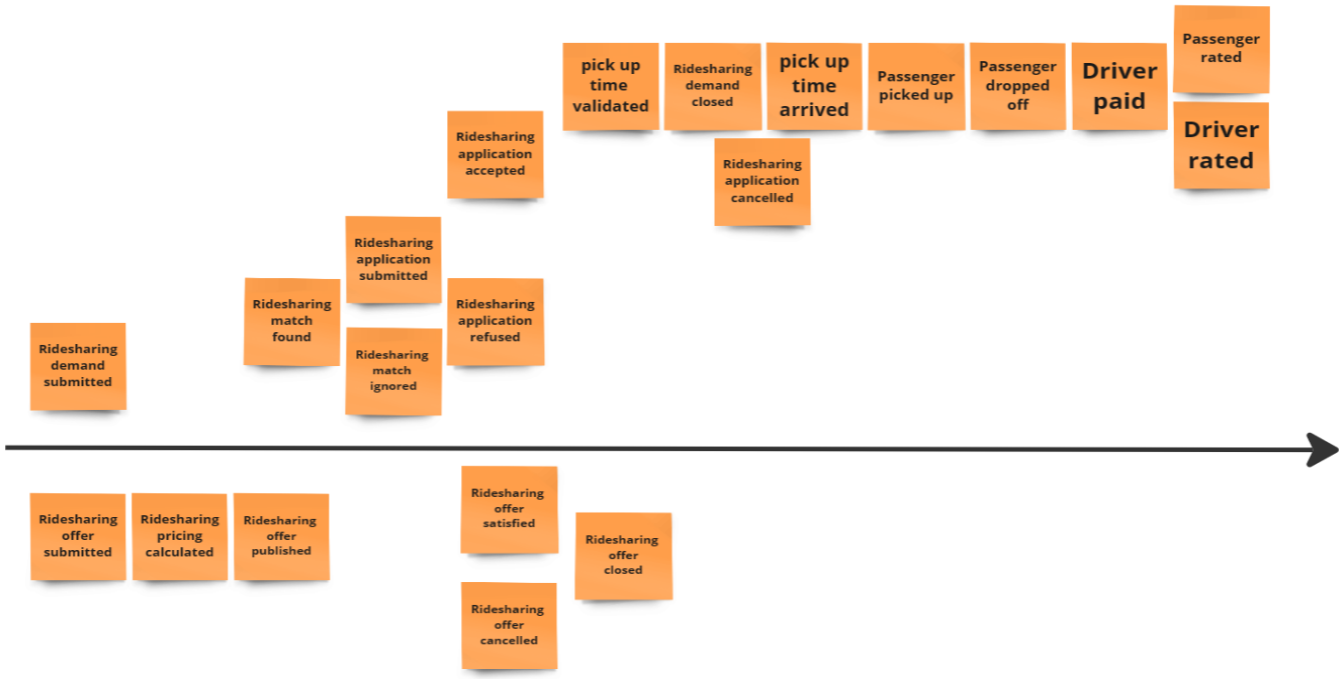

Figure 1.4: Domain Events Diagram

Before proceeding, it is important to check the syntactical correctness of each word as to eliminate risks of misunderstandings later. Next, domain causes must be identified, whether it is user actions, external systems or other domain events.

Figure 1.5: Event Storming Specifications

With this we have now identified the different actors, and the actions they may take.

Conclusion

We have now seen the root cause that leads most companies to throw monolithic architectures for the sake of the more structured and decomposed microservices architecture, along with some of the drawbacks of the latter and a brief introduction to Domain-Driven Design, which will be further discussed in the next chapter. We have also provided the specifications of our project.

Chapter 2

Microservices Design Literature

Introduction

In this chapter, we provide details about the architectural approach of our system and its design. We will talk about microservices, event-driven architectures, and Domain-Driven Design.

2.1 Microservices

We have briefly talked about microservices in the previous chapter, but what are they exactly? Microservices is a software development approach where a large application is broken down into smaller, independent, and modular services that can communicate with each other through APIs. Each service focuses on a specific business capability and can be developed, deployed, and scaled independently. This approach enables agility, flexibility, and scalability in software development.

2.1.1 When should we use a microservices architecture ?

A microservices architecture is most suitable for large and complex systems that demand:

- Scalability: When a high traffic is expected or the need for scaling certain services independently based on demand is anticipated.
- Agility: When development must be optimized and time efficient, and feature releases must not be delayed in any way.
- Flexibility: When a system is complex enough to require different technologies for different services, and teams need to be able to work independently on specific services.
- Resilience: When the system is required to be resilient, which means it tolerates service failures without impacting the entire system.
- Modularity: When a system needs to be easily maintainable and to allow for easier upgrades and changes.

2.1.2 Cons and drawbacks

Although a microservices architecture provides many benefits, it also comes with drawbacks.

- Increased complexity: A microservices architecture can be more complex than a monolithic architecture, especially when it comes to managing service dependencies and interactions. This complexity can make it more challenging to design, develop, test, and deploy services.
- Communication overhead: In a microservices architecture, services communicate with each other via APIs, which can create an overhead in managing API versions, ensuring compatibility between services, and managing the network.
- Operational overhead: Managing a microservices architecture can require additional operational overhead, such as monitoring, logging, and troubleshooting.
- Testing complexity: Testing individual services and their interactions can be more complex and time-consuming in a microservices architecture, and may require specialized tools and expertise.
- Deployment complexity: Deploying and scaling individual services independently can be more challenging than deploying a monolithic application, and requires a sophisticated deployment process.
- Distributed system issues: A microservices architecture introduces the challenges of managing a distributed system, such as ensuring consistency, managing failures, and handling network latency.
- Security concerns: A microservices architecture can introduce additional security concerns, such as securing the API gateways, managing service authentication and authorization, and managing data privacy and protection across services.

To sum up, a microservices architecture requires careful planning, design, and management to overcome its challenges and achieve its benefits. In the scope of this project, we have chosen the Domain-Driven Design approach with an event-driven architecture as a solution.

2.2 Event-Driven Architecture

An event-driven architecture (EDA) is a software architecture pattern where the flow of the application is determined by events or messages that are exchanged between components rather than a linear sequence of steps. In this architecture, components of the system communicate with each other by emitting and receiving events, which trigger the execution of certain actions or processes. These events can be internal to the system, such as a change in the state of an object, or external, such as a user input or a message from another system. EDA is commonly used in distributed systems, where components are distributed across multiple machines and need to communicate asynchronously, making it very efficient to use with a microservices architecture since it also allows for flexibility and scalability, as new components can be added or removed from the system without affecting the overall architecture. EDA can also help in achieving loose coupling between components, as each component only needs to know how to handle the events it receives, rather than having direct knowledge of the internal workings of other components.

2.2.1 Event-Driven Microservices

Event-driven microservices is an architectural pattern that combines the benefits of event-driven architecture and microservices. In this pattern, each microservice is designed to perform

a specific business function or task and communicates with other microservices by sending and receiving events. When an event occurs in one microservice, such as a change in the state of an object, it emits an event that other microservices can consume and act upon. This allows for a highly decoupled and scalable architecture, where each microservice can be developed and deployed independently without affecting the overall system. Event-driven microservices are well-suited for handling large-scale distributed systems, where events can be generated from a variety of sources, including user interactions, external services, and IoT devices. By using event-driven microservices, developers can create highly responsive, fault-tolerant systems that can easily adapt to changing business requirements.

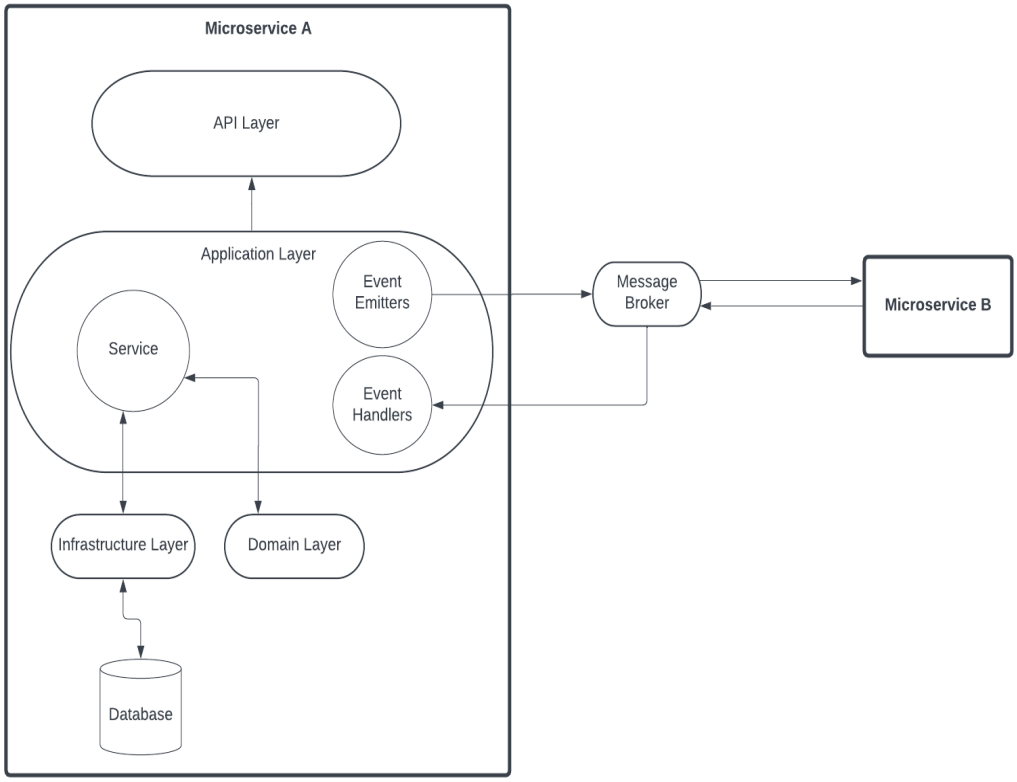

Figure 2.1: Example of an event-driven microservice

One of the key benefits of event-driven microservices is that they enable asynchronous communication between microservices, which can lead to improved performance and reduced latency. This is especially important in systems that need to handle a large volume of events or transactions.

2.2.2 Messaging

Messaging is the means by which different components interact with each other. There are many different ways of implementing it and the choice depends on the specific requirements and use case of the application or system. Nevertheless, all messaging models fall in one of two categories: Synchronous or Asynchronous.

2.2.2.1 Synchronous Messaging

Synchronous messaging is also known as blocking communication. As its name suggests, it is a type of messaging where a sender blocks and waits for a response from the receiver before

continuing. In synchronous messaging, the sender and receiver are actively communicating with each other and both are involved in the message exchange. It is often used in systems that require immediate responses or strict consistency guarantees, such as real-time communication systems or online payment systems. However, it can also lead to reduced scalability and increased latency, as each request needs to be processed before the next request can be handled.

2.2.2.2 Asynchronous Messaging

Asynchronous messaging is a type of communication in software engineering where messages are sent and received between different components or services of a software system without waiting for an immediate response. This means that the sender does not have to wait for the recipient to process the message before continuing with other tasks. Asynchronous messaging is often used in distributed systems, where different components or services may be running on different machines or in different environments. It can help to improve the scalability, reliability, and performance of a system by allowing components to work independently and asynchronously. It can also enable the system to handle high volumes of requests and provide fault tolerance by allowing components to handle messages even if other components are down or unavailable.

2.2.2.3 Pub/Sub Messaging

Pub/sub messaging, meaning publish/subscribe, is an asynchronous messaging pattern in which a publisher sends messages to a message broker without knowing who the subscribers are, and the broker is responsible for delivering those messages to all subscribed consumers. It is a scalable and flexible messaging pattern that allows multiple subscribers to receive the same message, and it can also provide fault tolerance by allowing consumers to receive messages even if other consumers are down or unavailable.

2.2.2.4 Other Messaging Models

There are numerous other messaging models such as point-to-point messaging where messages are sent from a sender to one specific receiver. This is often used in message queues and similar systems. Isochronous messaging is when messages are sent and received at regular and predictable intervals, with a fixed time delay between the sending of each message. This is often used in real-time systems where a constant and predictable rate of data transfer is critical. In multicas messaging, a single message is sent to multiple recipients simultaneously. This is often used in streaming applications and similar systems.

2.3 Domain-Driven Design

Domain-driven design is an approach to software development that emphasizes the importance of understanding the domain of the problem being solved, and modeling that domain in software.[4]

2.3.1 Origins

In the 1990s, Eric Evans, software developer and consultant was working on a large software project for a financial services company. He noticed that the software design was becoming increasingly complex, and that it was difficult to maintain and extend the system. He realized

that the problem was not just a technical one, but that it was related to how the development team understood the business domain of the problem being solved. To address this problem, Evans began to develop a new approach to software development that focused on understanding the domain and modeling it in software. He called this approach "Domain-Driven Design" and began to write about it in articles and presentations. In 2003, Evans published his book "Domain-Driven Design: Tackling Complexity in the Heart of Software", which became a seminal work in the field of software development. The book introduced a set of concepts and techniques for modeling the domain of a problem in software, such as bounded contexts, ubiquitous language, aggregates, entities, value objects, and domain events. Since then, Domain-Driven Design has become widely adopted in the software industry, especially in the development of complex business applications. Domain-Driven Design has influenced the development of other software development practices and methodologies, such as microservices and event-driven architectures.[1]

2.3.2 Motivations

The motivations of Domain-Driven Design are to provide a way of developing software that is centered around the problem domain, rather than being focused solely on technical implementation details. Some of the key motivations of Domain-Driven Design include:

- Improving communication between developers and domain experts: Domain-Driven Design emphasizes the importance of using a shared language, called the "ubiquitous language", to describe the domain concepts and processes. By creating a common language, developers can better understand the domain experts' requirements and build more accurate and useful software.

- Enabling flexible and scalable software development: Domain-Driven Design encourages breaking down a large, complex domain into smaller, more manageable subdomains, called "bounded contexts". Each bounded context can be developed and maintained independently, allowing for more flexible and scalable software development.

- Encouraging a focus on the core domain: Domain-Driven Design encourages developers to focus on the parts of the system that provide the most business value, and to model those parts of the domain in software with precision and clarity. By focusing on the core domain, developers can build more useful software and avoid getting bogged down in peripheral concerns.

- Encouraging a data-driven approach: Domain-Driven Design encourages developers to use the domain data model as the foundation for the software design. By modeling the domain data, developers can build software that closely matches the real-world problem domain, leading to better software quality and more satisfied users.

2.3.3 Key Concepts

In order to achieve its goals, Domain-Driven Design relies on several key concepts that help developers create software systems that are well-suited to the needs of the domain.[8]

- Domain: A domain is a specific area of expertise that the software aims to model or simulate, and understanding the domain is crucial for the creation of software that is consistent with the business at hand. This understanding guides developers to better design and implementation of the system.

- Ubiquitous Language: The ubiquitous language is the shared language used by all stakeholders involved in a project, from domain experts to software developers. It needs to be accurate in reflecting the concepts within the domain and understood by all the parties involved. The goal of using a ubiquitous language is to create a shared understanding of the domain and ensure that everyone involved in the project is speaking the same language, which prevents misunderstandings, and to make sure that good communication happens.
- Bounded Context: A bounded context is a boundary within which a certain shared language and model apply. Different parts of a software system may have different meanings for the same concepts based on the specific context in which they are used. Thus, bounded contexts also provide a certain logic that allows developers to break down a system into smaller, more manageable parts that can be developed independtly.
- Value Objects: A value object is an object that has no identity of its own and is defined solely by its attributes or properties. It represents a set of characteristics. Value objects can be easily copied or shared between entities without affecting the consistency or integrity of the system.
- Entities: An entity represents a concept from the domain that has a clear boundary and meaning in its context. It is used to model and capture the business rules and the logic that applies. Unlike value objects, entities have an identity of their own.
- Aggregates: An aggregate is a cluster of domain objects that can be treated as a single unit. It's composed of one or more entities and value objects that are related to each other and have a common boundary. By encapsulating these objects within an aggregate, developers can more easily reason about the behavior and state of the objects, and can make changes to the system with greater confidence and safety. Aggregates also ensure transactional consistency by enforcing a set of rules that ensure that changes to the objects within the aggregate are made in a consistent and transactional manner.
- Aggregate Root: When changes are made to an aggregate, all of the objects within the aggregate are treated as a single unit, and any changes to the objects must be made in a way that maintains the integrity of the aggregate as a whole. To ensure this, only the aggregate can access its domain objects. Interaction with the outside can only happen through its entry point, which is the aggregate root. .

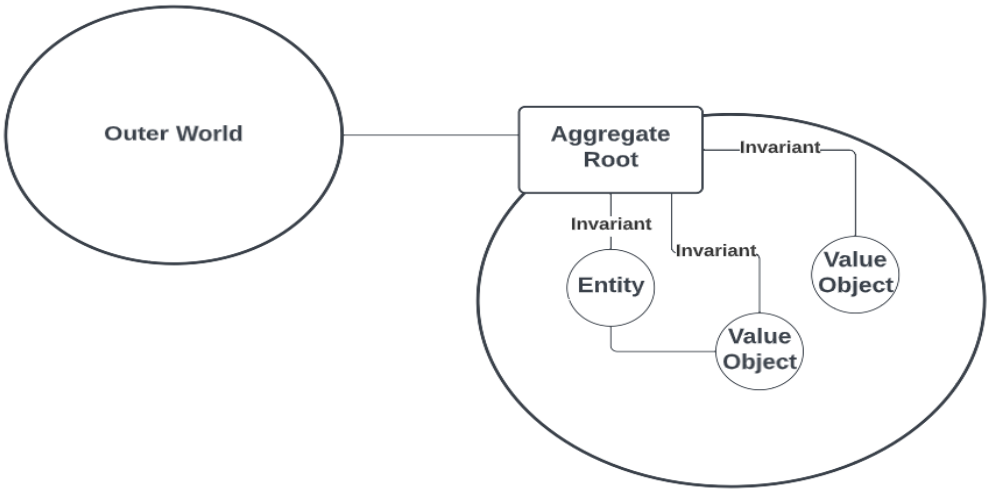

Figure 2.2: Strcture of an aggregate

As shown in 2.2, the outer world may only reference the aggregate root and not the aggregate as a whole. Meanwhile, objects inside the aggregate may reference each other, or even other aggregate roots. Invariants are rules that need to be satisfied at all times, they are an integral part of the consistency guarantees.[8]

- Services: A service is an object that represents cohesive and stateless operations or transformations that are not a natural part of any entity or value object within the domain. It might be used to coordinate the behavior of multiple entities or value objects, or to perform a complex computation or transformation that involves multiple domain concepts.

- Repositories: Repositories provide an abstraction layer to all data persistence mechanisms. It's a pattern that is used to decouple the domain model from the specific implementation details, allowing the model to be developed and tested independently of the data storage technology.[9]

- Domain Events: A domain event represents a fact or an occurrence that has taken place within the domain. It is a message that carries information about what has happened, but does not prescribe any particular action or behavior. Other parts of the system can listen for these events and react accordingly.

- Context Mapping: Context mapping is a technique that aims to manage the complexity that arises when different bounded contexts within a system interact with each other. This process defines clearly the interactions and dependencies between the bounded contexts, as well as the interfaces and communication channels that will be used to connect them. The goal is to create a shared understanding of the architecture and dependencies, and to ensure that the different bounded contexts can work collectively in a coherent and consistent way. This helps to manage the complexity of the system and avoid misunderstandings or conflicts between the different teams.

2.4 Domain Modeling

When it comes to domain modeling, there is no one way to do it. However, some are more efficient than others in the right circumstances.

2.4.1 Anemic Domain Modeling

Implementing a rich domain model is an integral part of Domain-Driven Design. It involves building blocks like Entities, Value Objects, and Aggregates, which is the opposite of the anemic domain models. In anemic domain models, entities are represented by classes that include only data and connections to other entities. Business logic is absent in these classes and is usually placed in managers, services, utilities, helpers, etc.[3] The term "anemic domain model" was coined by Martin Fowler, who argued that this approach violates the principles of object-oriented programming and can lead to a number of problems, including:

- Violation of encapsulation: By exposing internal data structures through accessors, an anemic domain model can expose implementation details and violate the principle of encapsulation.

- Poor performance: By relying on external services or controllers to perform most of the processing, an anemic domain model can lead to poor performance and increased latency.

- Difficulty in maintaining code: Because much of the application's logic is spread across external services and controllers, an anemic domain model can be difficult to maintain and debug.
- Lack of flexibility: An anemic domain model can be inflexible and difficult to adapt to changing requirements, because most of the processing is performed outside of the domain model itself.

In the context of large and complex applications, neglecting to invest in proper domain modeling and development can result in an architecture that has a bulky Service Layer and a weak domain model. As a consequence, the classes that serve as interfaces to the application's functionality can become bloated with business logic over time, while the domain objects themselves are reduced to simple data structures with basic getter and setter methods. The lack of well-defined responsibility boundaries can cause domain-specific logic and responsibilities to become intertwined with other components, leading to a loss of clarity and making the system harder to maintain over time.
The figure 2.3 illustrates a typical 3-layer application with an anemic domain model.

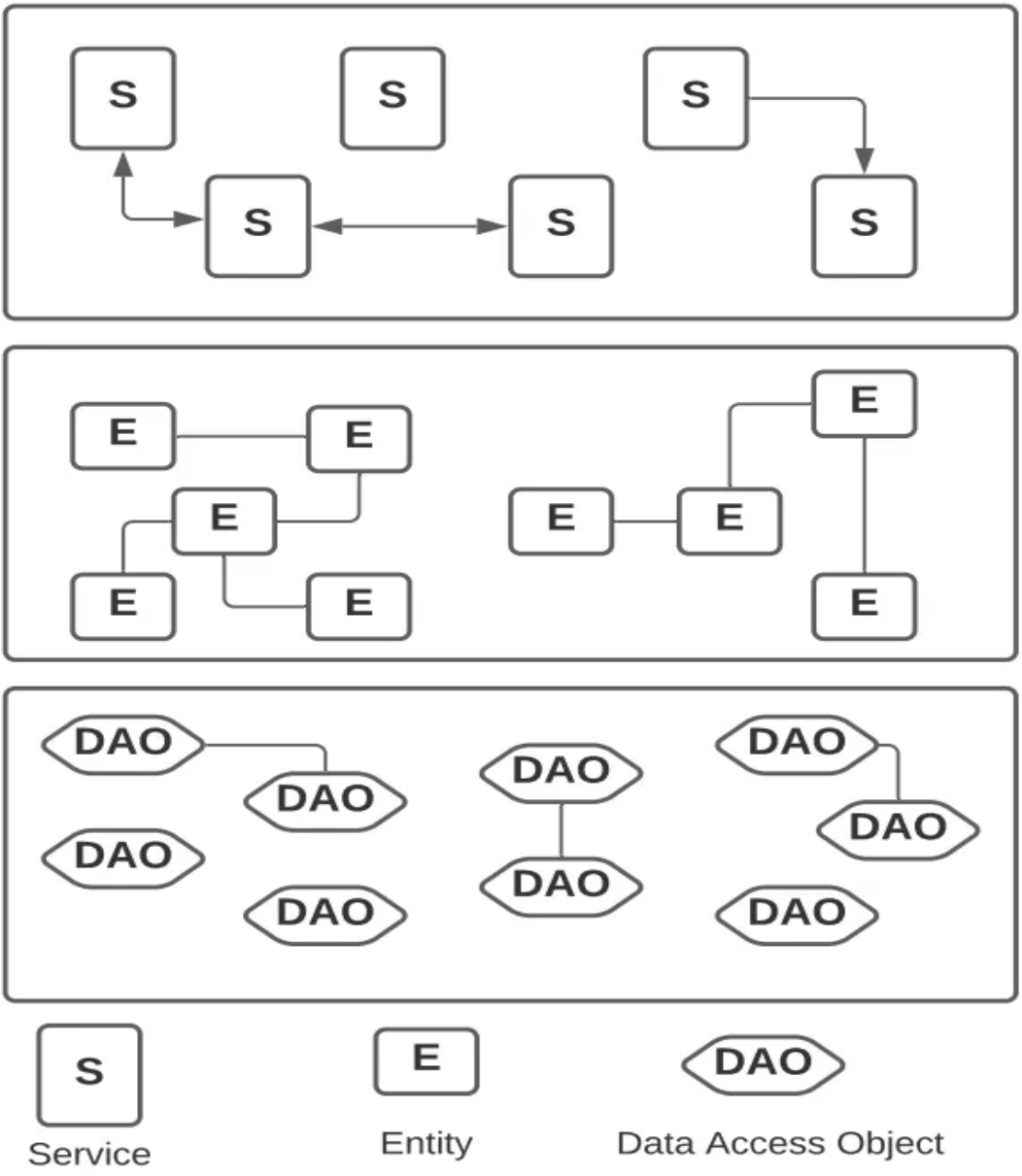

Figure 2.3: Anemic Domain Model[3]

The Data Access Layer (DAL) is comprised of a collection of classes that are responsible for handling the reading and writing of entities. Typically, there is a one-to-one correspondence between the number of Data Access Objects (DAOs) and entities, since each entity represents a distinct business scenario. It's important to note that DAOs don't contain any business logic, but rather serve as utilities for retrieving and storing entities. Within the Service Layer, various modules and classes are utilized to implement specific business logic using an appropriate DAO. The common operations performed by a service typically involve loading an entity through a DAO, modifying its state, and persisting it. According to Martin Fowler, this architectural pattern is known as Transaction Scripts. As the functionality becomes more intricate, the number of operations required between loading and persisting entities grows. Since entities and DAOs are typically unmodified unless there's a need to store new fields, services frequently rely on other services, resulting in a "fat" Service Layer.

2.4.2 Domain-Driven Model

In contrast to the Anemic Domain Model, Domain-Driven Design follows an architecture where the domain model is rich with behavior and encapsulated data, and the services are thin and focused on orchestrating domain objects to perform business operations.

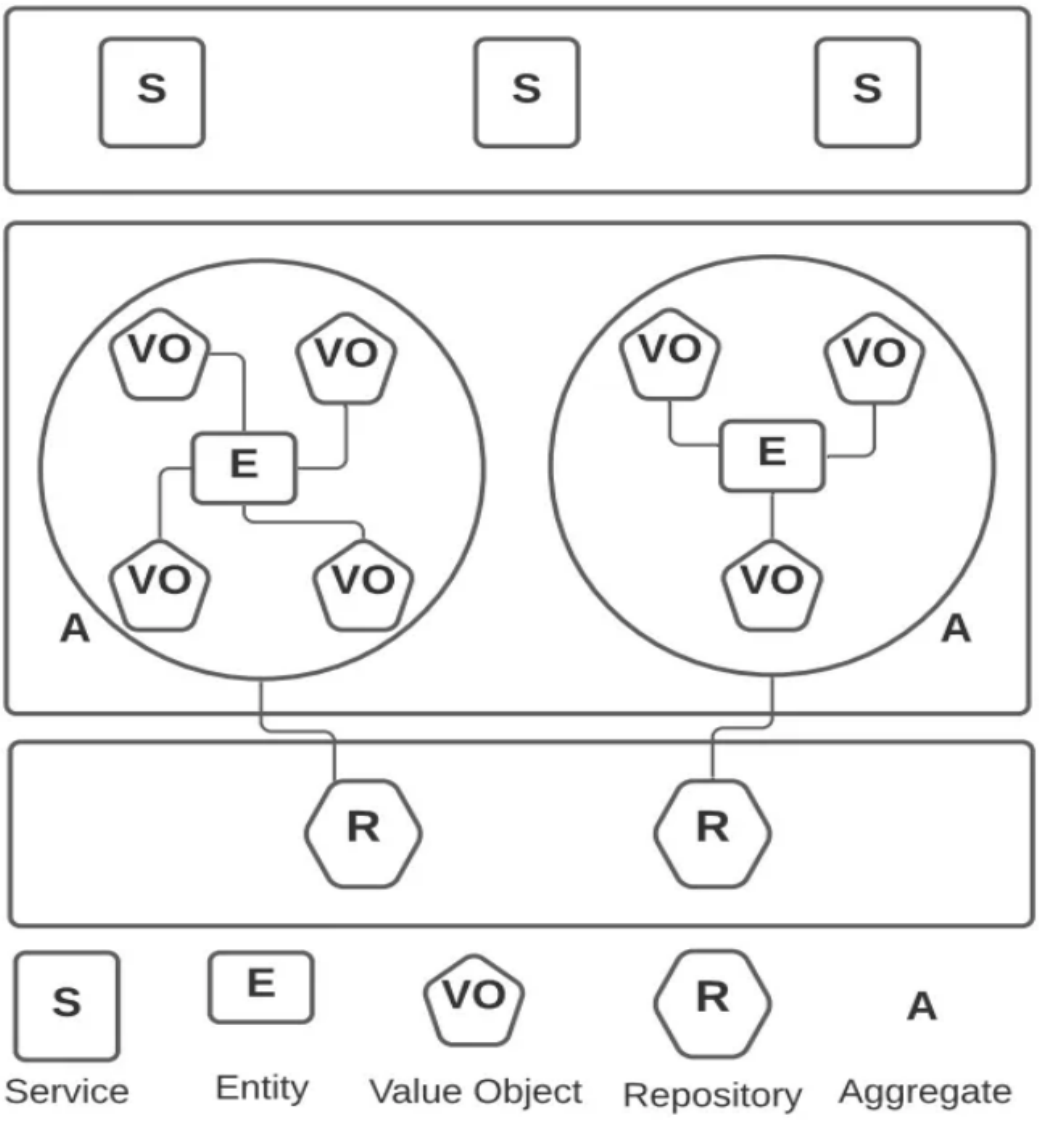

Figure 2.4: Domain Driven Model[3]

One of the key differences between the Anemic Domain Model and Domain-Driven Design is the size and purpose of the Service Layer. In an application that employs Domain-Driven Design, the Service Layer is much thinner because most of the business logic is included in the Domain Layer. Each service is focused on a specific area of the domain and only includes operations related to that area. This approach allows for better encapsulation of domain behavior and a more cohesive model. When comparing the Anemic Domain Model and Domain-Driven Design, we can observe that the Domain Layer in the latter is thicker and includes more objects such as value objects, entities, and aggregates. Additionally, the last layer is thinner than that in the Anemic Domain Model. Each aggregate requires only one repository, resulting in fewer repositories compared to the number of Data Access Objects required in the Anemic Domain Model for loading and persisting entities. This demonstrates how Domain-Driven Design promotes better encapsulation and cohesion of the domain model.

2.5 Onion Architecture

Traditional architectures often suffer from tightly coupled components and a lack of separation of concerns, which can lead to issues with testability, maintainability, and dependability. To address these challenges and provide a better solution, Jeffrey Palermo introduced the Onion Architecture. This approach overcomes the limitations of traditional architectures and offers a way to solve common problems. In Onion Architecture, layers communicate with each other through the use of interfaces, promoting loose coupling between components. This makes the architecture more flexible and easier to modify or replace individual components.

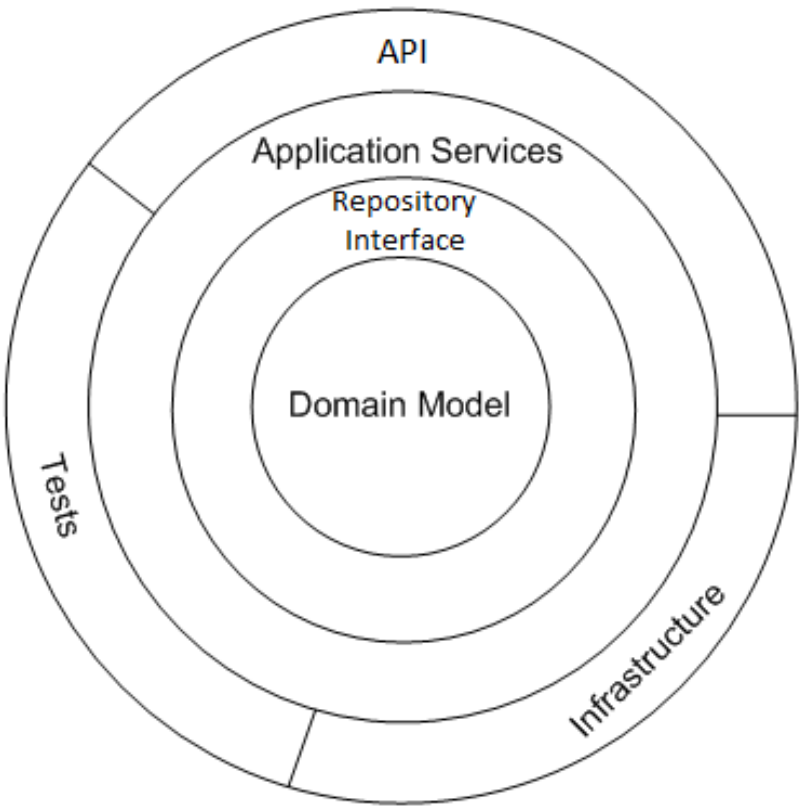

Figure 2.5: Onion Architecture[7]

The Onion Architecture consists of several layers that interface with each other. Internal layers have no dependencies with external layers, with the core representing the domain at the center. Since Domain-Driven Design insists on the focus on the domain, this makes The Onion Architecture a very convenient and powerful model to follow. Thus, each microservice in the ridesharing platform will be broken down into:

- Domain Layer: The central component of the Onion Architecture is the domain layer, which consists of the business and behavior objects: Value Objects, Entities, Aggregates and Events. The primary objective is to gather all domain objects in this core layer, which comprises all application domain objects. In addition to domain objects, domain interfaces may also exist, which don't posess any dependencies. Domain objects are kept flat and lean, without any complex code or dependencies.

- Repository Layer: The purpose of this layer is to establish a separation between the application's domain entities and its business logic. It typically incorporates interfaces that enable object saving and retrieval, often in conjunction with a database. The data access pattern, which is a more decoupled approach to data access, is a key feature of this layer. In addition, a generic repository is created, which includes queries to retrieve data from the source, map that data to a business entity, and store changes made to the business entity back to the data source.

- Application Layer: Contains application services that encapsulate complex logic, coordinate actions between different parts of the application, and orchestrate the flow of data between layers. In an event-driven architecture, event handlers also belong to this layer.

- API Layer: This layer is responsible for managing and exposing the APIs of the back-end systems in a standardized and secure way. It can handle requests from different types of clients and translate them into the appropriate format for the back-end systems to consume.

- Tests: This layer includes all of the code and resources related to testing the software application. The Onion architecture allows the testing of the Domain Model and Application Services in isolation from all the other layers.

- Infrastructure: The Infrastructure layer is responsible for providing a stable and reliable foundation for the application to run on, such as databases and containers.

Conclusion

With that being said, we can now clearly answer the question of why chose this architecture. We talked about microservices, event-driven architecture, domain-driven design and onion architecture. All of these concepts share the same basic goal: Simplicity, Modularity and Scalability.
In the next chapter, we move on to a more concrete conception and design of our ridesharing platform.

Chapter 3

Design of the proposed solution

Introduction

This chapter goes into the details of the design of our ridesharing platform, and the steps that guided us to the blueprint of the project.

3.1 Microservices Decomposition and context mapping

Identifying microservices and how they communicate in a complex system is a crucial step to its success.

3.1.1 Microservices Decomposition

The result of the event storming from figure 1.5 is what will allow us to identify the different microservices that make up our application. By grouping similar business processes and their boundaries, we have now identified the different bounded contexts.

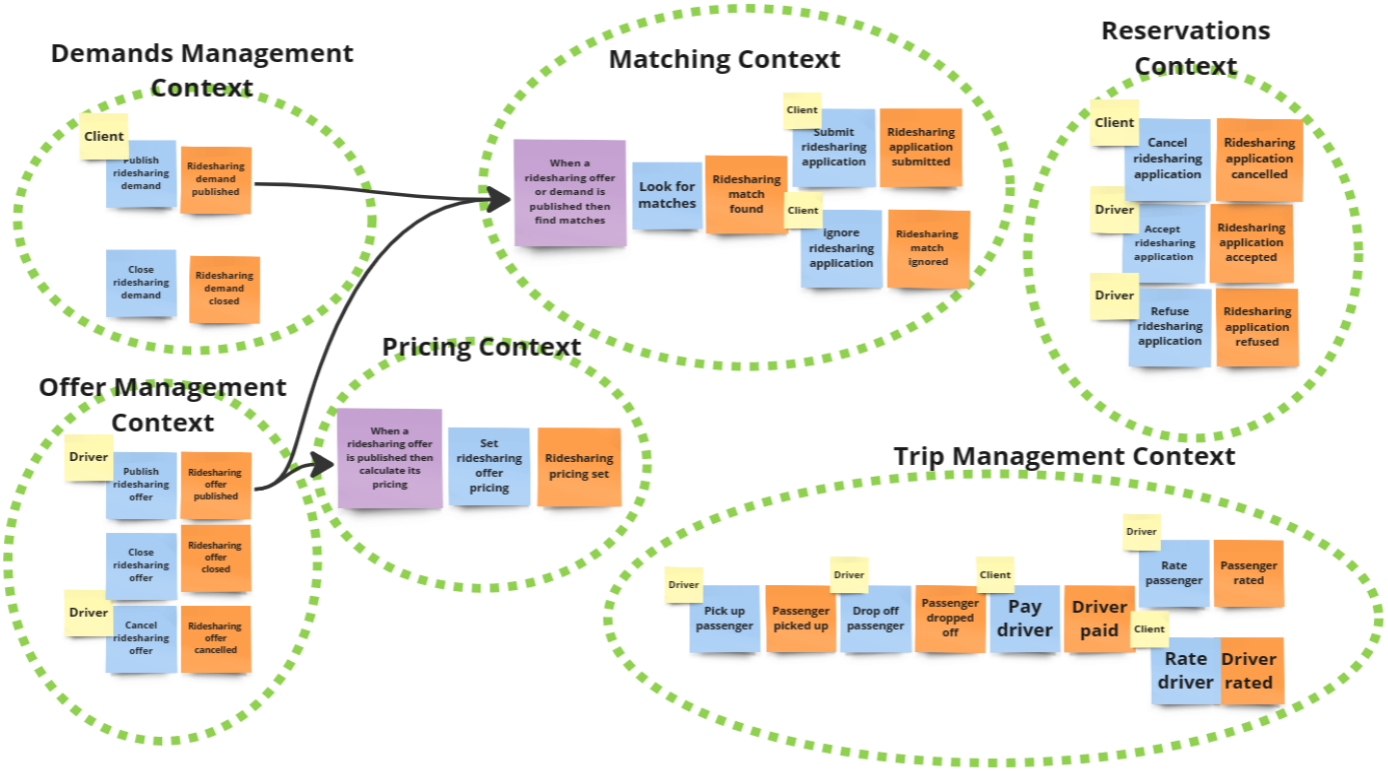

Figure 3.1: Event Storming Bounded Contexts

The final step is to identify the aggregates.

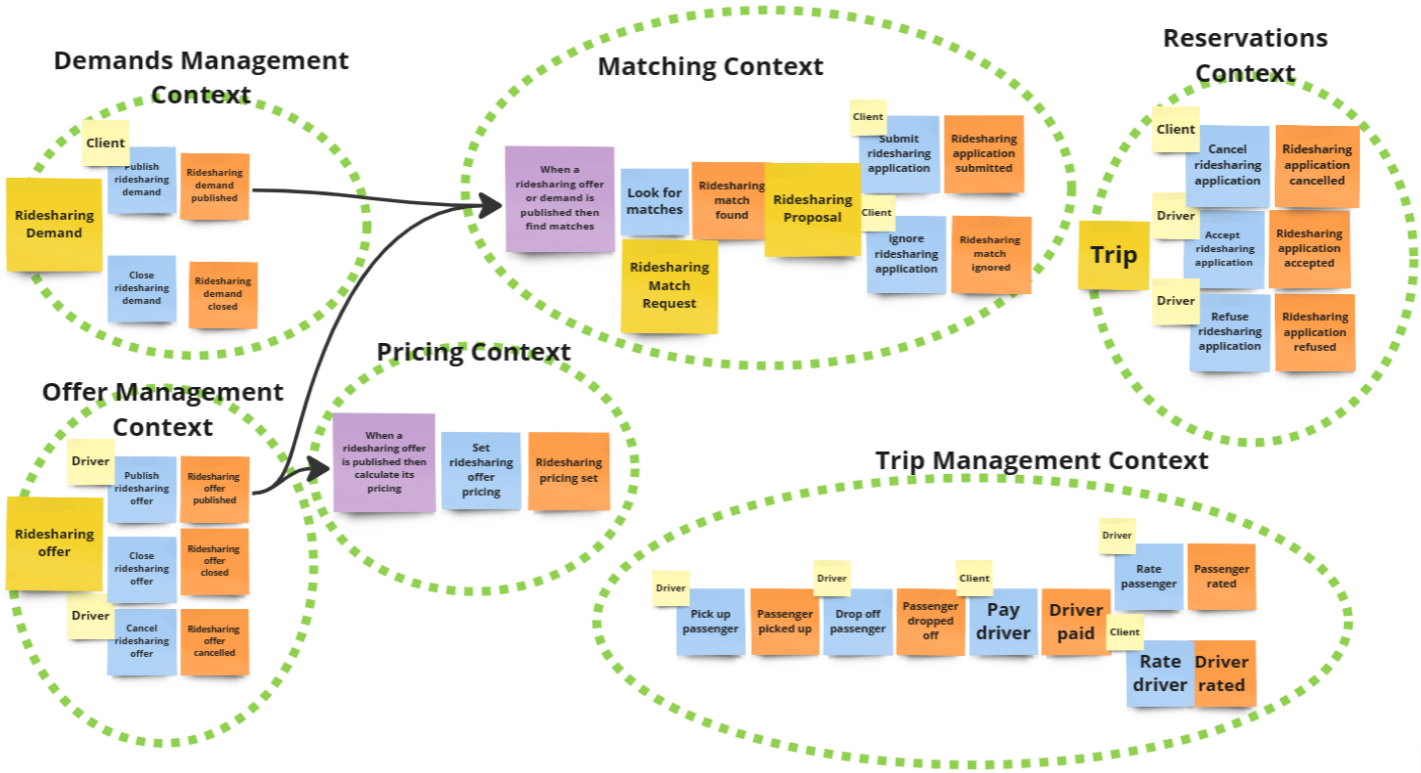

Figure 3.2: Event Storming complete diagram

3.1.2 Context Mapping

Models within distinct bounded contexts have the ability to be developed and executed separately. However, it is important to note that bounded contexts are not entirely self-reliant. In order to accomplish the overarching objectives of the system, the components must interact and coordinate with one another. Similarly, while bounded contexts may evolve independently in their implementation, they must ultimately be integrated with one another. Consequently, touchpoints between bounded contexts will always exist and these points of connection are referred to as contracts. Since each contract affects more than one party, they need to be defined and coordinated. That is the role of the context mapping. In this regard, we have used the following models:

- Shared Kernel: The shared kernel is a component or module that is shared between two or more bounded contexts. It provides a standardized way for these different parts of the system to communicate and collaborate while maintaining their independence.
- Downstream Context : A downstream context refers to a context that depends on another context to fulfill its own responsibilities. It is a context that receives information or services from an upstream context and uses them to provide value to its own stakeholders.
- Upstream Context: The upstream context is responsible for producing the data or services that the downstream context needs to do its work.
- Anticorruption Layer: An anticorruption layer is a design pattern used to protect a bounded context from a legacy or external system with a different design, data model or language. It acts as a translation layer, converting data or requests from the legacy or external system into a form that the bounded context can understand, and vice versa.
- Published Language: In this model, the supplier's public interface is not intended to conform to its ubiquitous language. Instead, it is intended to expose a protocol convenient for the consumers, expressed in an integration-oriented language. Hence, the public protocol is called the "published language."

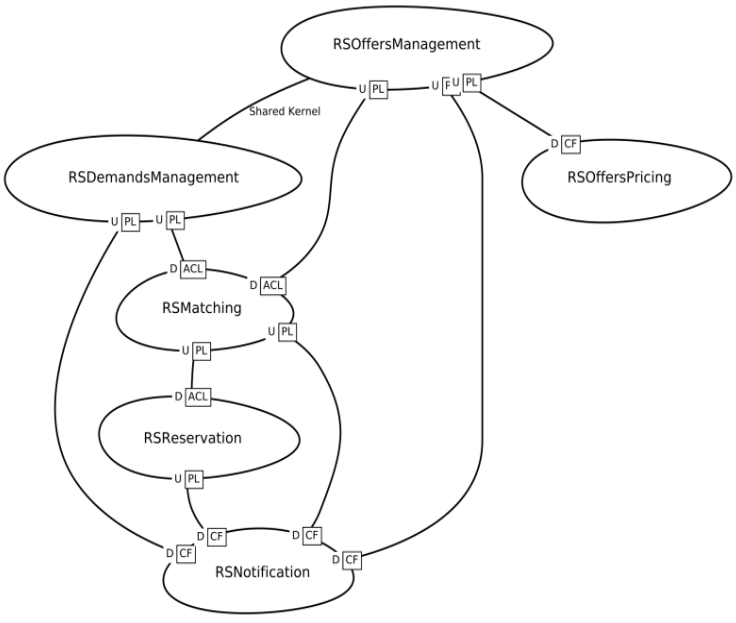

Figure 3.3: Context Mapping

3.2 General Conception

Now that the design and architectural patterns are clearly established, we may proceed to a general conception of our different bounded contexts, a blueprint that every microservice should stick to.

3.2.1 Generic Classes Diagram

First, the DDD package will provide the necessary interfaces to implement the Domain-Driven Design logic. The rs package, which stands for ridesharing will represent our Onion Architecture. Therefore, it will contain a domain package, which contains the core business rules, the Infrastructure package, which contains repository implementations, the application package, which contains the services classes, and the API package, which contains classes that implement resources for API calls.

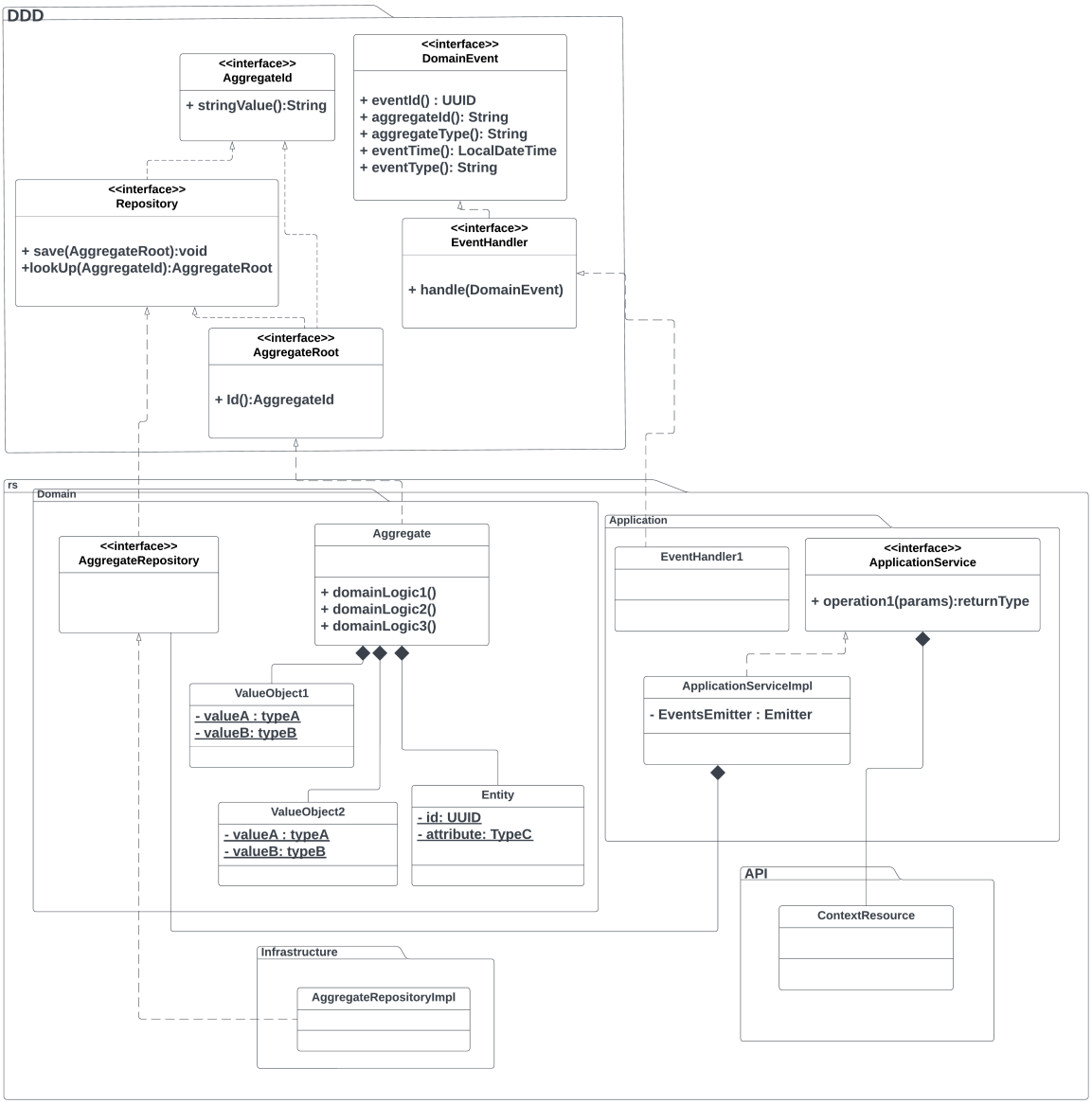

Figure 3.4: Generic Classes Diagram

3.2.2 Generic Sequence Diagram

The sequence diagram demonstrates the typical sequence of interactions between different layers of the architecture.

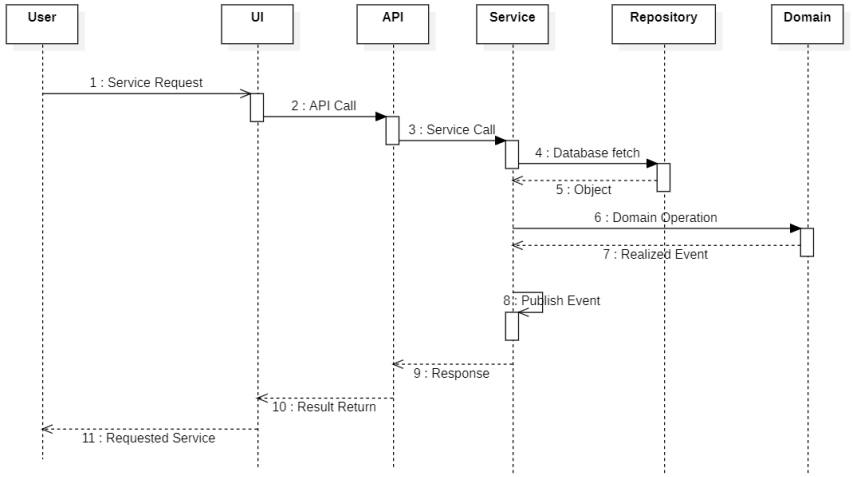

Figure 3.5: Generic Sequence Diagram

3.3 Specific Conception

With everything we've seen so far, we may now proceed to a more in-depth conception of the microservices in our platform. In the scope of this project, we will only be interested in microservices involving the rider's experience, that is the demands management microservice, the matching microservice and the notifications microservice.

3.3.1 Demands Management Microservice

The Demands Management Microservices handles all operations that involve the Ridesharing Demands, from creating to publishing, cancelling, closing, and editing.

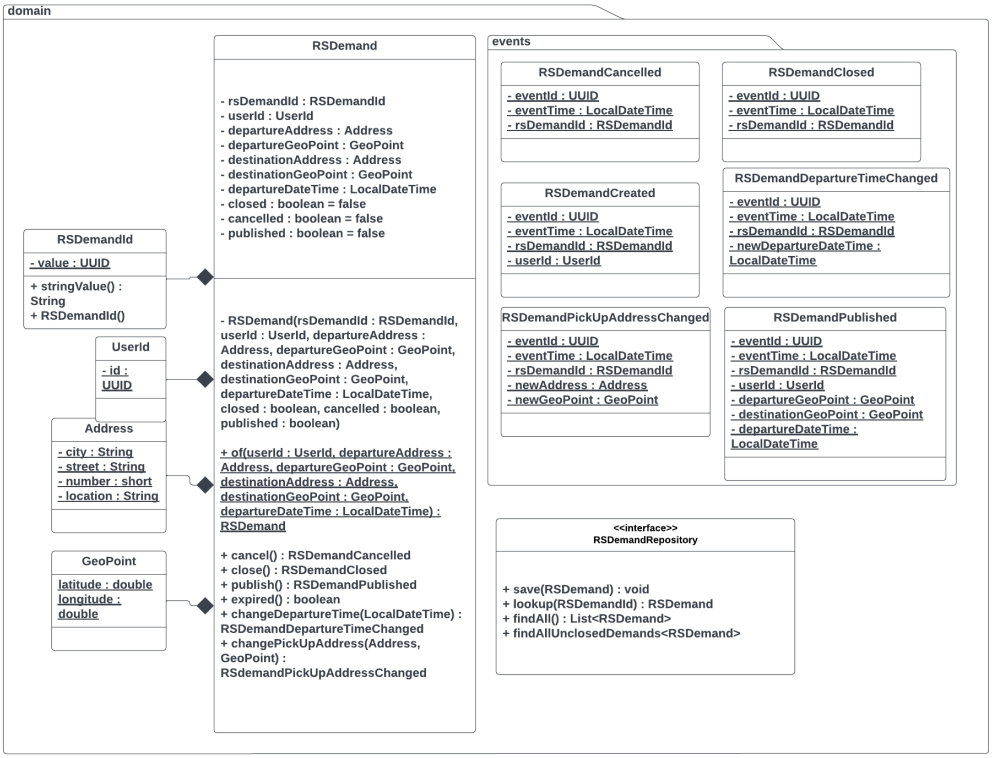

Figure 3.6: Demands Management Microservice Domain Classes Diagram

As show in figure 3.6, the domain layer of the Demands Management Microservice contains the RSDemand aggregate along with the value objects and entities it is composed of. The repository interface is also part of the domain, and the domain events this bounded context produces.

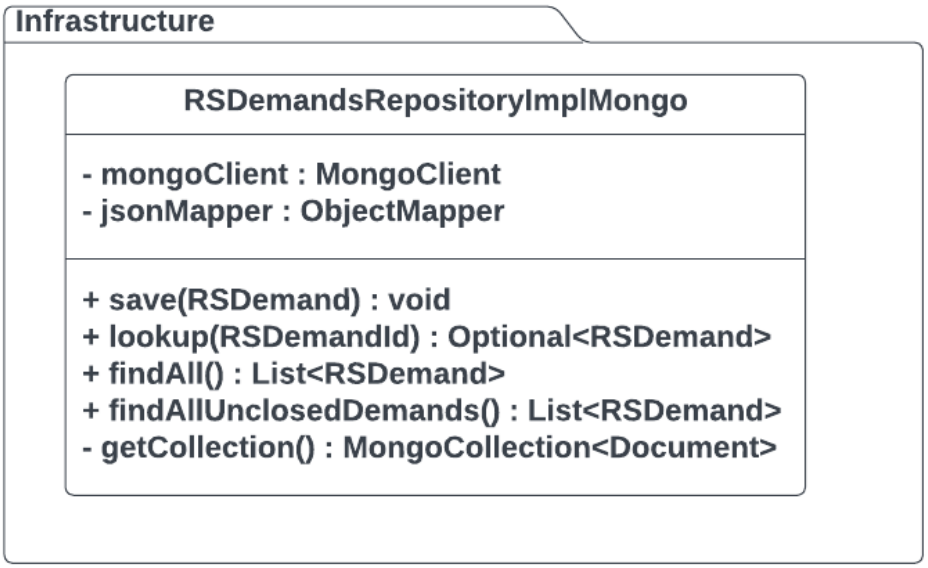

Figure 3.7: Demands Management Microservice Infrastructure Classes Diagram

The figure 3.7 represents the classes that implement the connectivity with the database. In this case, it is simply one class that implements the domain's repository interface. The mon-

goClient attribute allows connectivity with MongoDB and the jsonMapper allows stringifying objects in JSON format.

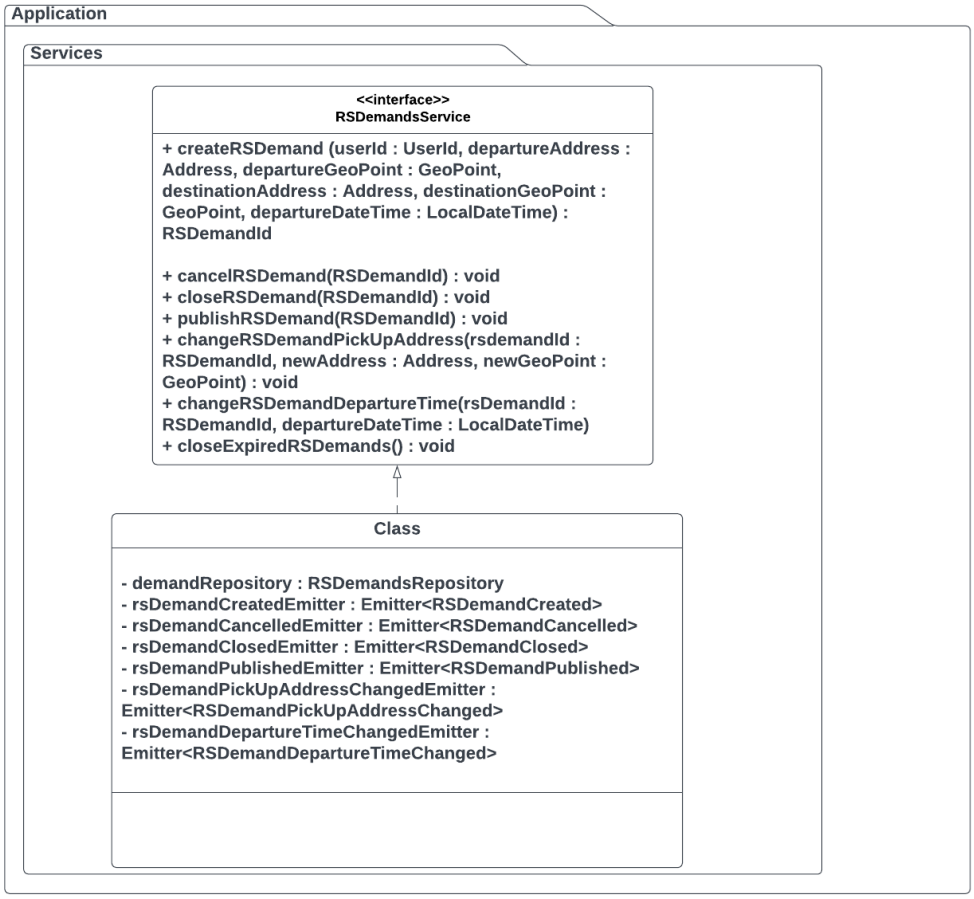

Figure 3.8: Demands Management Microservice Application Classes Diagram

The figure 3.11 represents the application classes. In the demands management microservice, it's only the service interface with its implementation, which uses a repository.

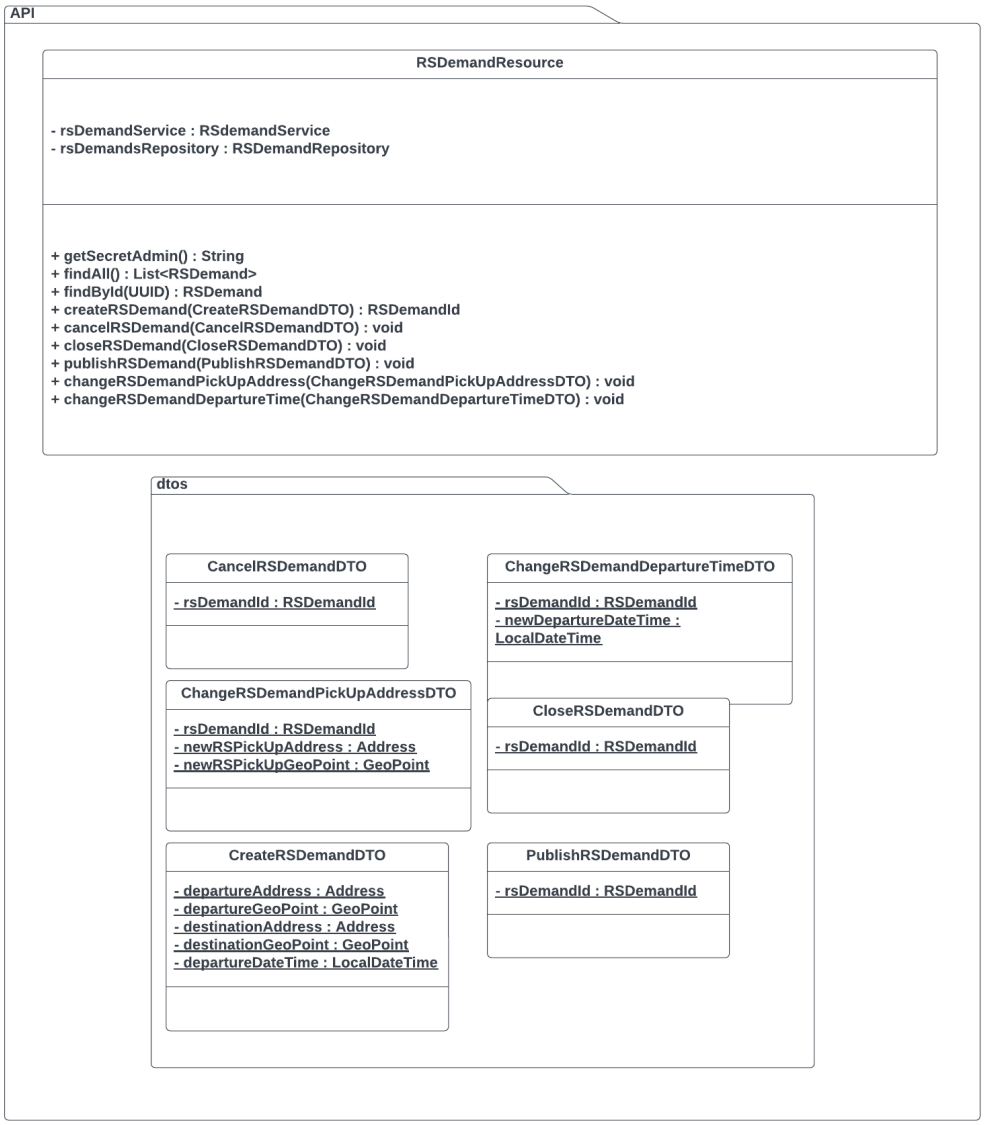

Figure 3.9: Demands Management Microservice API Classes Diagram

The figure 4.2 contains the classes that implement the API of the microservice. It allows eventual user interfaces to interact with the back-end.

3.3.2 Matching Microservice

The matching microservices is responsible for finding compatible offers and demands according to departures and destinations. It is the most complex context of the application.

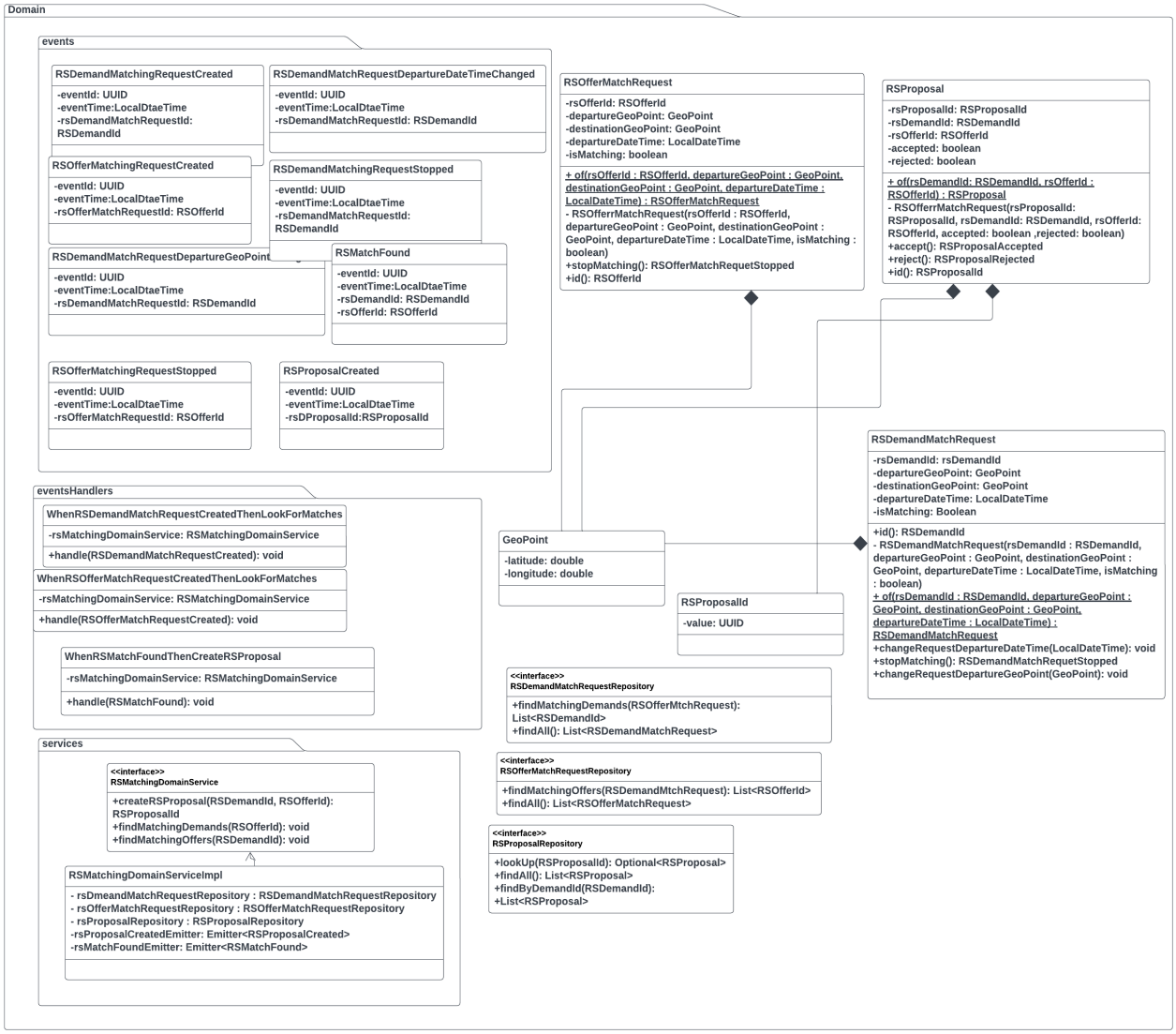

Figure 3.10: Matching microservice classes diagram

Figure 3.10 represents the domain layer of the matching context. Similar to the demands management context, it has a domain events package. It also has event handlers and a domain service that is internal to the context. There are three aggregates: RSDemandMatchRequest, RSOfferMatchRequest and RSProposal.

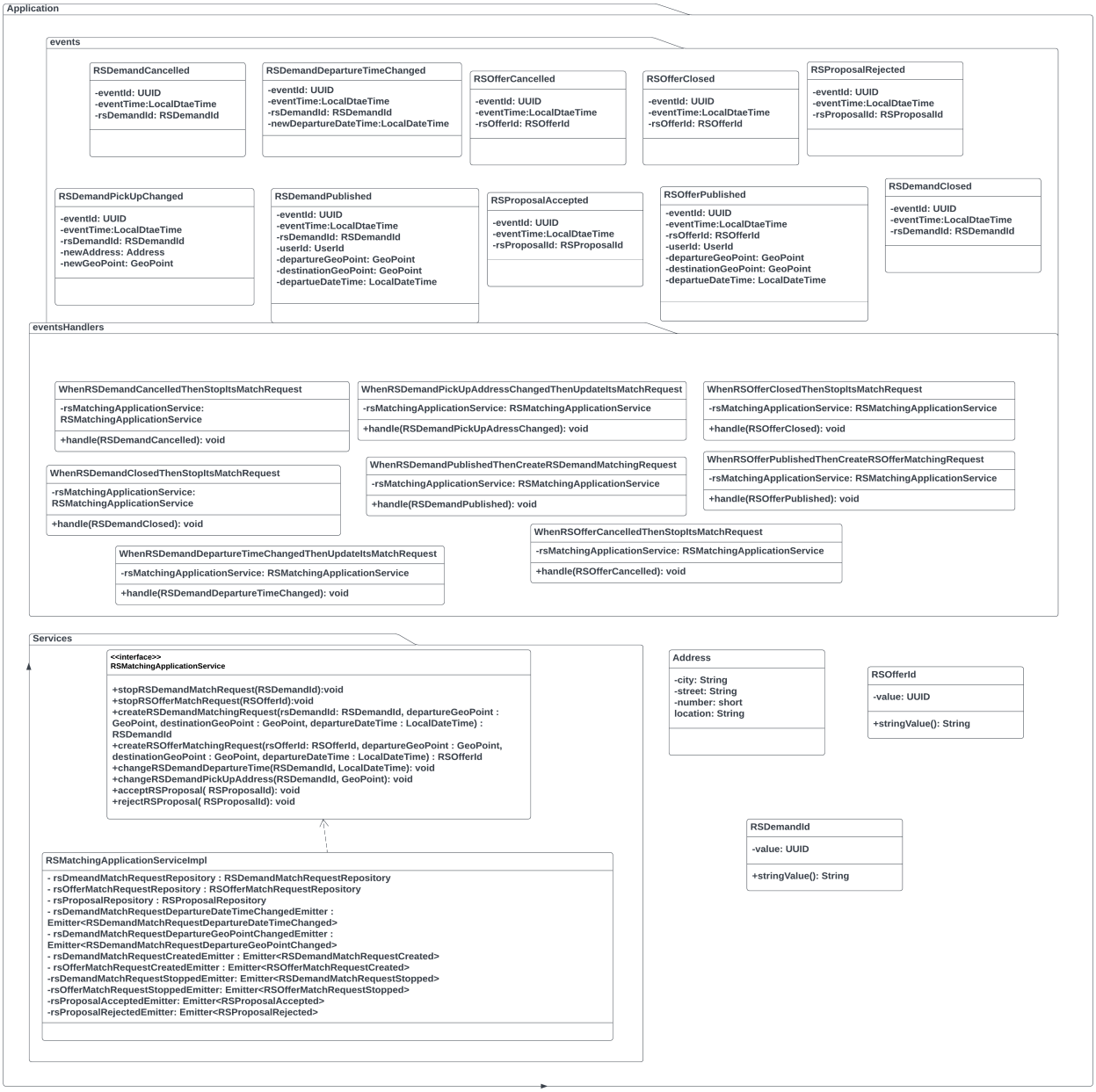

Figure 3.11: Matching microservice application classes diagram

The figure 3.11 represensts the applicaiton layer. In addition to the services, it is here we find value objects and entities that come from other contexts according to the context mapping rules. The application event handlers are responsible for reacting to external triggers.

Figure 3.12: Matching microservice infrastructure Classes Diagram

The infrastructure package contains three repositories for each aggregate that are responsible for the persistence of their corresponding aggregate in a MongoDB database.

Figure 3.13: Matching microservice API classes diagram

Finally, the API package contains the resource classes. One for each aggregate.

Conclusion

In this chapter, we have successfully realized the design and conception of the ridesharing platform. We can now talk about the concrete implementation of all the previous concepts.

Chapter 4

Implementation and validation

Introduction

In this chapter we will talk about the technologies, libraries and frameworks used, and present a brief look on the functionalities of the application.

4.1 Technology Stack

In our project, we have combined different technologies.

4.1.1 Development Environment: Intellij IDEA

IntelliJ IDEA is an integrated development environment (IDE) developed by JetBrains for Java and other programming languages such as Kotlin, Groovy, Scala, and more. It is a popular IDE among Java developers due to its powerful features such as code completion, refactoring, debugging, and version control integration. IntelliJ IDEA is available in two editions, the Community Edition which is free and open-source, and the Ultimate Edition which is a commercial product with additional features such as support for more programming languages, frameworks, and tools.

4.1.2 Frameworks

Quarkus is the main framework we have opted for in this project."Supersonic Subatomic Java", it is a Kubernetes-native Java framework designed for creating lightweight, scalable, and fast microservices and serverless applications. It allows developers to build cloud-native applications using familiar Java technologies, including Hibernate, RESTEasy, Eclipse Vert.x, and Apache Kafka, while optimizing them for containerized environments. Quarkus offers features such as fast startup time, low memory footprint, high-density deployments, and native compilation, which can help to reduce infrastructure costs and improve application performance. Additionally, Quarkus provides a comprehensive extension system that enables developers to customize and extend the framework with additional functionality and integrations.

4.1.3 Libraries

Many libraries were used in this project. Lombok (short for "laconic project") is a Java library that helps to reduce boilerplate code and improve code readability by automatically

generating common code constructs, such as getters, setters, constructors, and more. In addition to lombok are the extension libraries to the quarkus framework that allow it to use other services such as kafka.

4.1.4 Platforms

For the implementation of the messaging process, Apache Kafka was used. It is a publish-subscribe messaging system that is highly scalable and fault-tolerant. We also used Keycloak, it is an open-source identity and access management (IAM) platform that provides authentication, authorization, and security capabilities.

4.1.5 Testing

JUnit is a popular open-source testing framework for Java programming language. It is used to write and run repeatable automated tests, called unit tests, that verify the behavior of individual units of code. JUnit provides a set of annotations, assertions, and other features that make it easy to write and manage tests, and it can be integrated with build tools like Ant, Maven, and Gradle. The framework helps developers catch bugs and errors early in the development process, which can reduce the overall cost and time required to deliver high-quality software.

4.1.6 Infrastructure

The infrastructure of our application consists of Docker and mongoDB. Docker is a platform that allows developers to easily create, deploy, and run applications in containers. Containers are lightweight, portable, and self-contained environments that provide a consistent runtime environment for applications, regardless of the underlying infrastructure. Docker makes it easier to build, package, and distribute applications, as well as to scale and manage them, by abstracting away the underlying infrastructure and providing a simple, standardized interface for working with containers. It is widely used in DevOps and cloud computing to streamline application development and deployment processes. MongoDB is a popular, open-source NoSQL document-oriented database that stores data in a flexible JSON-like format called BSON (Binary JSON). It was designed to provide high scalability, high performance, and high availability, making it a popular choice for modern, data-driven applications that need to store large amounts of unstructured or semi-structured data. MongoDB also provides powerful querying and indexing capabilities, as well as support for horizontal scaling across multiple nodes in a cluster.

4.2 Demonstrations

In order to demonstrate our work, we'll be using Swagger UI, a web-based interface that allowss us to quickly and easily test different API endpoints, explore available methods and parameters, and view response data in real-time.

4.2.1 Administration with Keycloak

Keycloak provides an easy-to-use platform for securing applications and services with strong authentication and authorization capabilities.

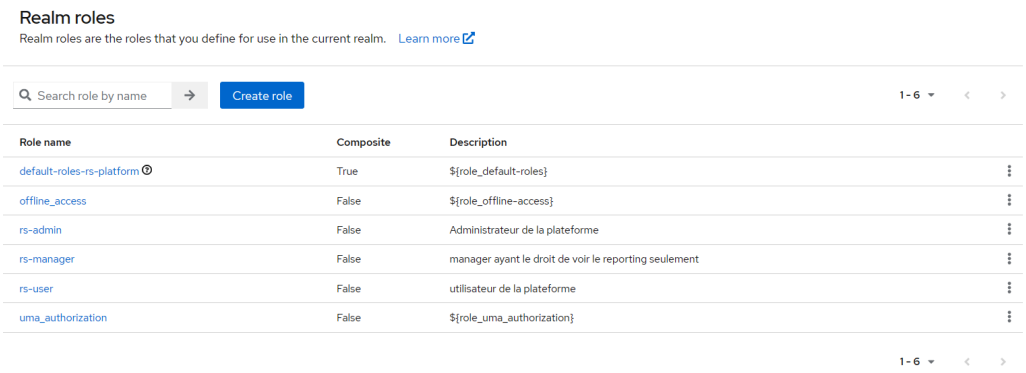

Figure 4.1: Assignable roles

The different roles represent different authorizations.

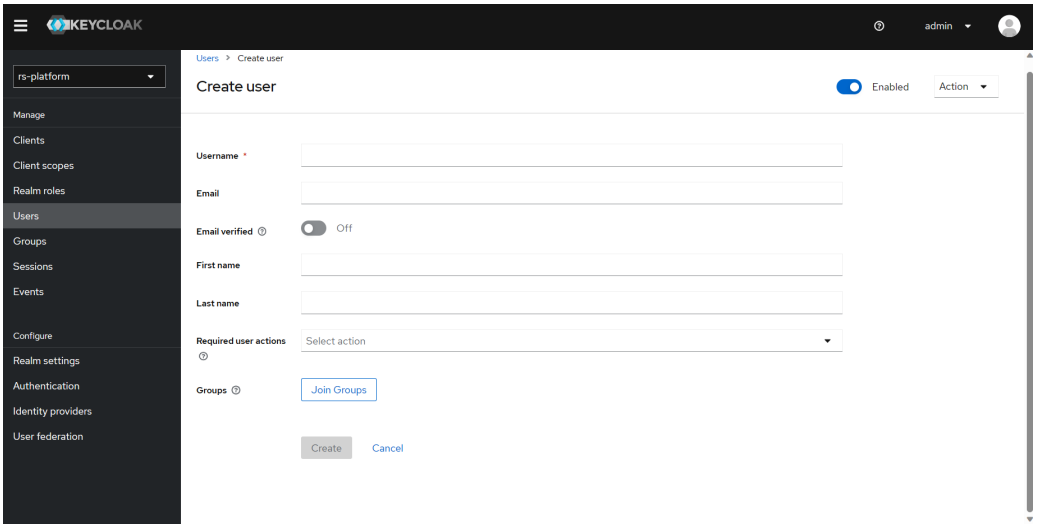

Figure 4.2: Adding user

When a user is created, they are assigned a default role. The admin can change a user's role.

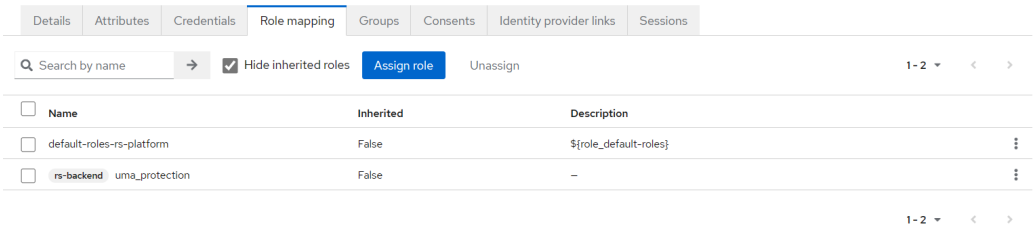

Figure 4.3: Changing a user's role

As shown in figure 4.4, this user has two roles. We can customize by adding or removing roles. When all is done, users should be able to log to their account.

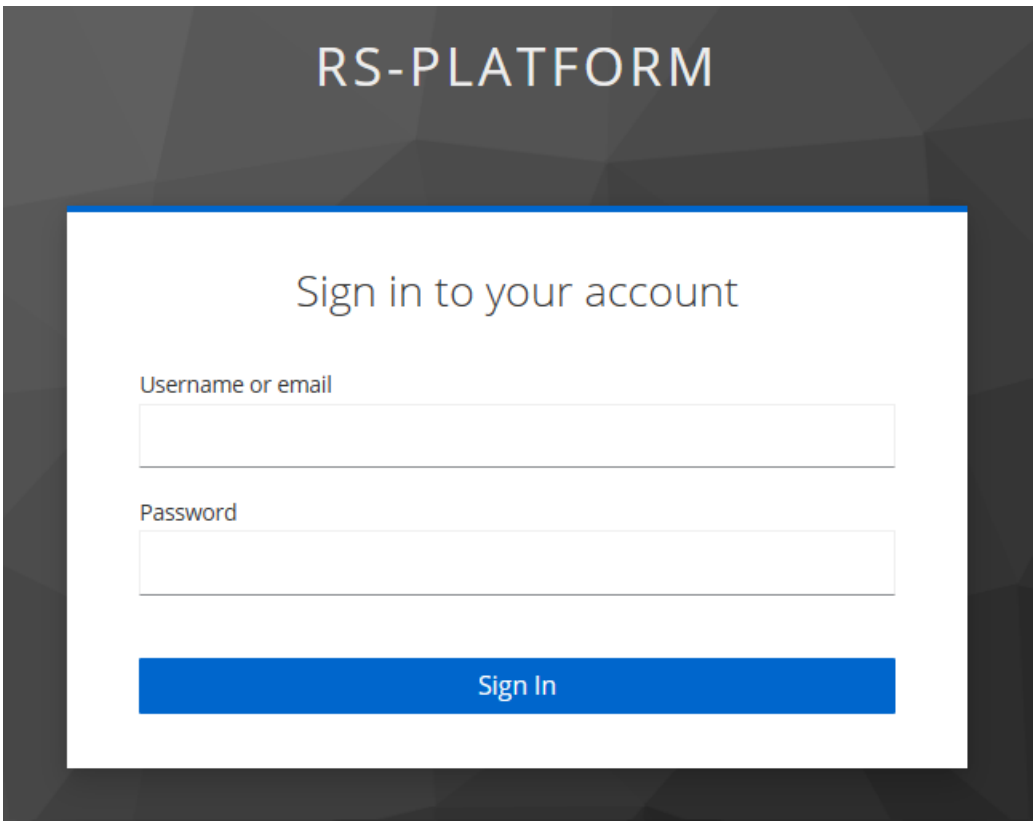

Figure 4.4: Login page

4.2.2 Demands management microservice

The demands management microservice is responsible for the creation of the ridesharing demands, and their maintenance throughout their lifecycle.

Figure 4.5: Demands management services

The figure 4.7 shows the services exposed by the API layer of the microservice, which matches the methods of the resource class (figure 4.2). The figures below are the final result.

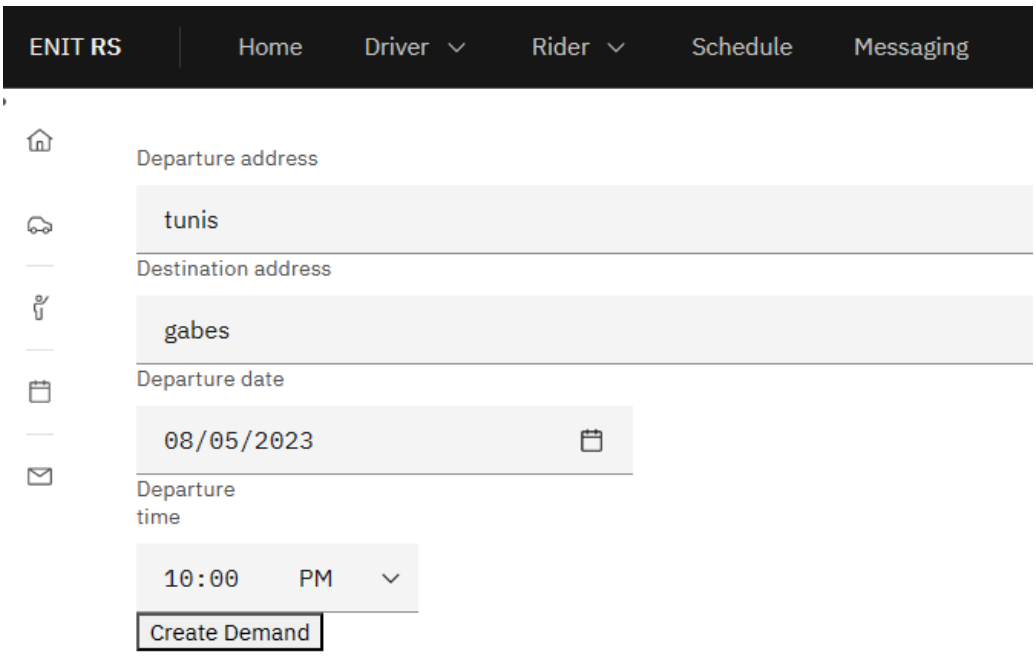

Figure 4.6: Ridesharing demand creation

Once the demand created, the user can now publish it.

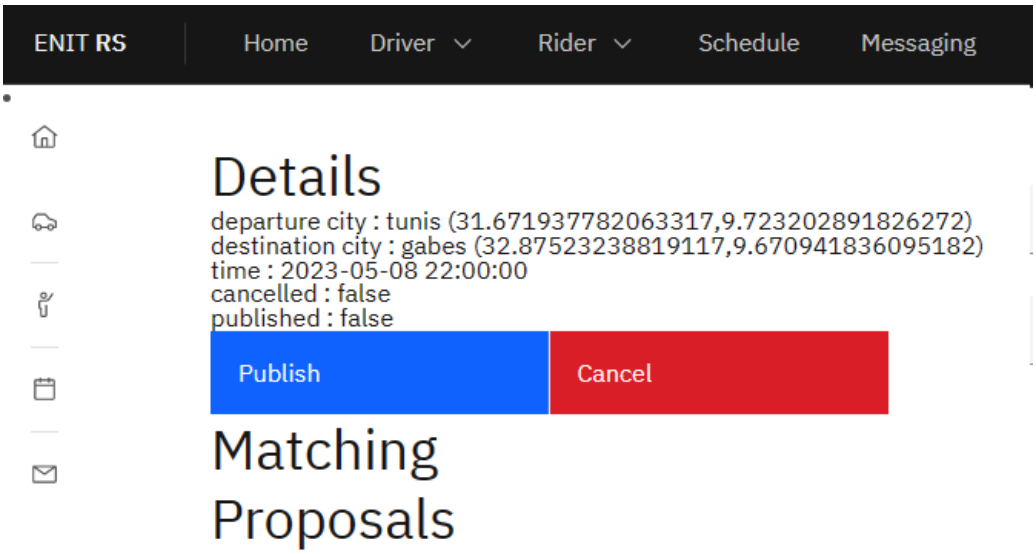

Figure 4.7: Publishing ridesharing demand

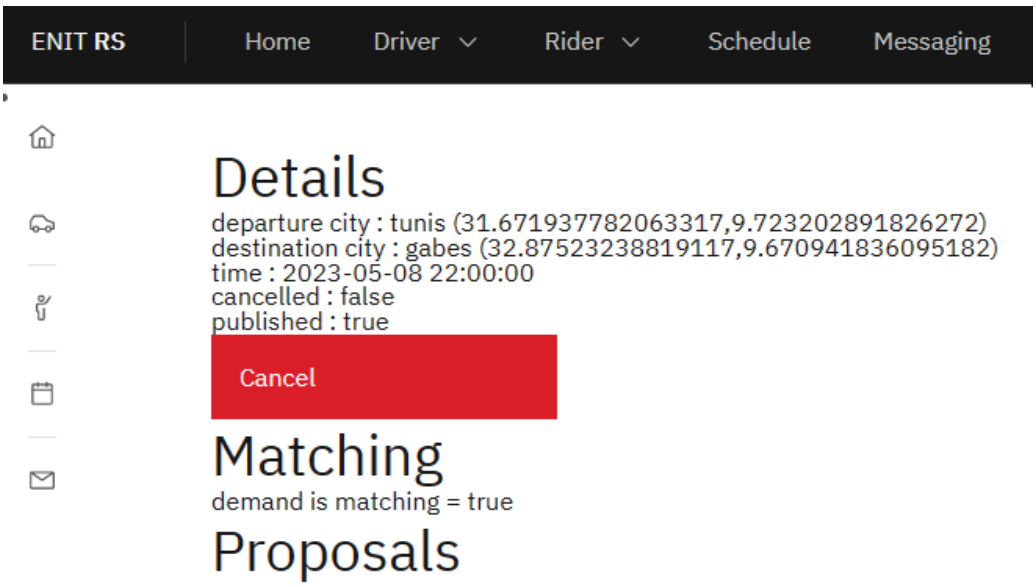

Figure 4.8: Published ridesharing demand

As shown in figure 4.8, the demand is now looking for matches.

4.2.3 Matching microservice

The matching microservice is responsible for creating ridesharing proposals for the riders according to their needs. When a corresponding offer is created, a proposal will be created.

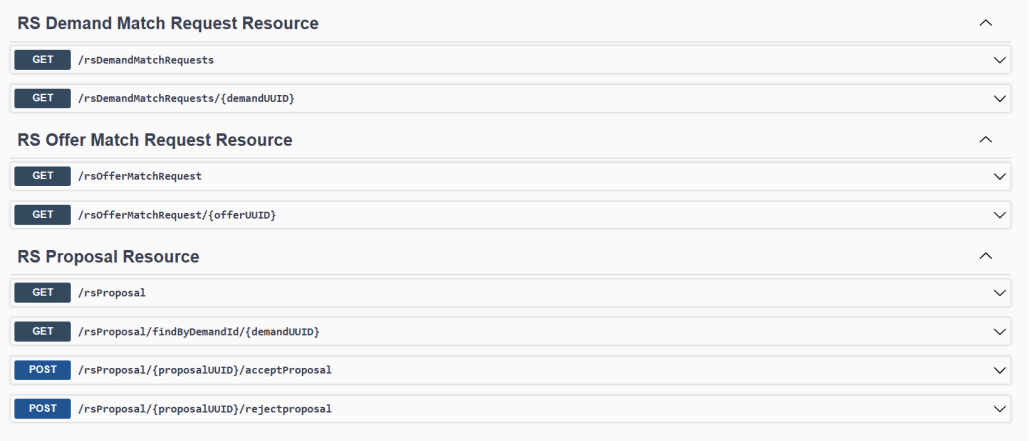

Figure 4.9: Matching services

The figure 4.9 shows the services of the matching microservice. Here too it matches its respective resouce classes as shown in 3.13.

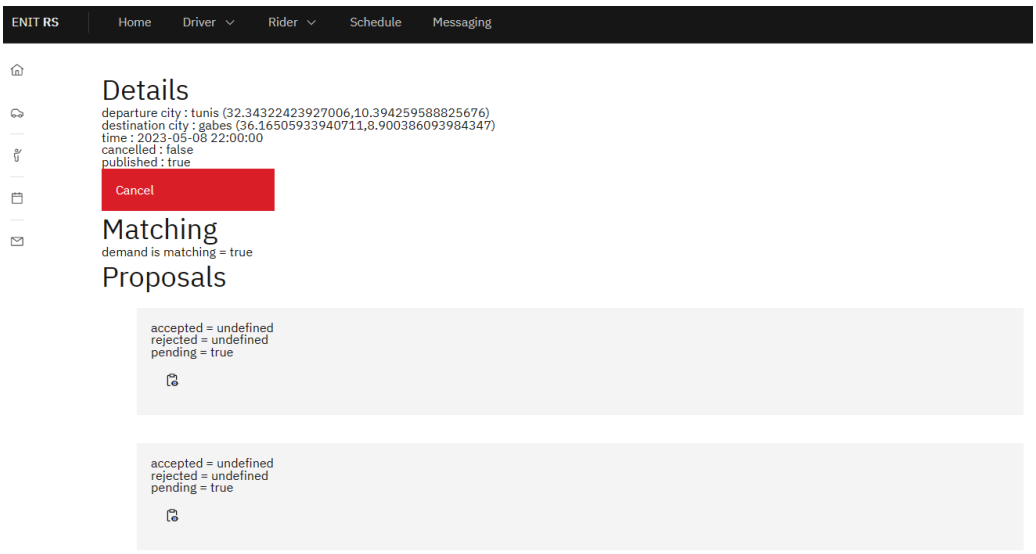

Figure 4.10: Proposals created

As shown in figure 4.10, two proposals were created for the demand.

Conclusion

In this chapter, we talked about the technologies used to implement our work, and have shown the final result of the project.

General Conclusion

The current report illustrates the work carried out during the second-year of the Software Engineering curriculum's end-of-year project at the National School of Engineers in Tunis. The objective was to design and implement a ridesharing platform based on a microservices architecture. We started with a case study of similar works and the specifications. Then, we described the design as well as our proposed solution, and finally ended with the implementation of the platform.

We had initially hoped to complete the front part of our platform, as well as other microservices such as user management, but were unable due to lack of time. This can be further pursued as personal work.

Nevertheless, the project was a success in the sense that we have learned a lot in terms of technical skills as well as teamwork and collaboration. We have gained a basic understanding of the microservices architecture, as well as domain-driven design. We have learned important practices of Software Engineering such as Modularity, Abstraction and testing.

Bibliography

[1] Eric Evans. *domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2003.

[2] Adam Gluck. Introducing domain-oriented microservice architecture. <https://www.uber.com/en-IT/blog/microservice-architecture/?fbclid=IwAR1hpeHQYgfo0JDhrAWvqXKSXDaigZ8aDyMc-HZtpEWZRoeOBCc6gq0OaUo>, 2020.

[3] Mahad Khan. Domain-driven design: Effective domain modeling and its perks. <https://medium.com/ssense-tech/domain-driven-design-effective-domain-modeling-and-its-perks-e4e3e3e0d5ee>, 2021.

[4] Vladik Khononov. What is domain-driven design. <https://www.oreilly.com/library/view/what-is-domain-driven/9781492057802/ch04.html?fbclid=IwAR12Gv0ORxJm32-cHarbXmqO75l3pHFHFMSrnmsBzPtCU4s0jyhxE7Ie-kk>, 2023.

[5] Matt Klein. Lyft's envoy: From monolith to service mesh. <https://www.microservices.com/talks/lyfts-envoy-monolith-service-mesh-matt-klein/?fbclid=IwAR2hzl2NVXTsu8CX7e7wX8vGhvl6bWP4AUQ2rvGq349ajuRhca0nj7WMCjE>, 2017.

[6] Lyft. Inc. 2022 form 10-k annual report. u.s. securities and exchange commission. <https://investor.lyft.com/financials-and-reports/sec-filings/default.aspx>, 2023.

[7] Tapas Pal. Understanding onion architecture. <https://www.codeguru.com/csharp/understanding-onion-architecture/?fbclid=IwAR3-TwCINFcMxVo_qzpjgUq_Z6qsgGlzE_O4X9m0tdrO60jd4ytBQ7cSkFI>, 2018.

[8] Alex soyes. Ddd : Domain-driven design. <https://alexsoyes.com/ddd-domain-driven-design/?fbclid=IwAR0DMBtlmWHFH0IOC3QehOReOEHaz6pOwBCgcwTVlkfpoNJ8xc~:text=Le>, 2022.

[9] Mike Wojtyna. Persisting ddd aggregates. <https://www.baeldung.com/spring-persisting-ddd-aggregates?fbclid=IwAR2P8S6HMAVmMak3X30u9bXOLczxdCVf2khtY4qETnaTNgSYyigQNUlV9Gc>, 2018.