ADD Iteration 1

Step 1: Considered Inputs

Descriptions

User Stories

Name	Description
US01 - Register User	As a User I want to register myself in the application so that I use it

Quality Attributes

Name	Description
QA1 - Maintainability	The prototype should be implemented in a way so that future problems/bugs found during the development, deployment and possibly support can be resolved quickly without major costs.
QA2 - Complexity	The complexity of prototype shouldn't surpass certain limits due to the simplicity of the context.
QA3 - Modifiability	Future updates of the prototype should be simple and low cost. For example, the prototype must be implemented in a way that it will not fail if, for some reason, the ports of the services are changed.
QA4 - Scalability	The prototype must support scalability by thinking in future features that can be implemented. For example, the addition of a new microservice should be as simple as smooth as possible.

Concerns

Name	Description
CON1 – Prototype accessible from the web	The developed prototype must be accessible using a web browser.

Name	Description
CON2 – Developer staff and deadline	The team must develop the prototype in six weeks, being the delivery deadline on the 22nd of December of 2022. </br>The team was constituted by four members.
CON3 – Authentication and Authorization	The prototype must contain Authentication, so that only registered user can access it, and Authorization, so that certain features can be limited to certain users.
CON4 – Multiple ports	With multiple microservices, using different ports, there can be confusion in the frontend (Single Page Application) on which port to call.

Architectural Constraints

Name	Description
CRN1 - Microservices	The monolithic developed in the first part of the project must be divided into microservices.
CRN2 – Data Migration	The data in the monolithic, developed in the first part of the project, must be migrated to their respective new database (in each microservice).
CRN3 - Deployment	The deployment of the prototype should be, ideally, based on containers.

Technology Constraints

Name	Description
TC1 – The use of two or more programming languages	The prototype must be developed using two, or more, programming languages.

Name	Description
TC2 – Open-source tools only	Only open-source tools are allowed in the development of the prototype.
TC3 - GraphQL	The team must analyse GraphQL and its suitability to the project. However, the tool might, or not, be used in the final version of the prototype.
TC4 - Open API (Swagger)	The Open API (Swagger) must be used for the documentation of the API.
TC5 - ContextMapper	ContextMapper must be used for the creation of a context map.

Priorities

User Story

Name	Importance	Difficulty
US01 - Register User	High	High

Quality Attributes

Name	Importance	Difficulty
QA1 – Maintainability	High	Medium
QA2 – Complexity	Medium	Low
QA3 – Modifiability	High	Medium
QA4 - Scalability	Low	Medium

Concerns

Name	Importance	Difficulty
CON1 – Prototype accessible from the web	High	Medium

Name	Importance	Difficulty
CON2 – Developer staff and deadline	High	High
CON3 – Authentication and Authorization	High	Medium
CON4 – Multiple ports	Medium	Medium

Architectural Constraints

Name	Importance	Difficulty
CRN1 – Microservices	High	High
CRN2 – Data Migration	High	High
CRN3 – Deployment	High	High

Technology Constraints

Name	Importance	Difficulty
TC1 – The use of two or more programming languages	High	Medium
TC2 – Open-source tools only	High	Low
TC3 - GraphQL	Medium	High
TC4 - Open API (Swagger)	High	Medium
TC5 - ContextMapper	Medium	Medium

Step 2: Establish iteration goal and select inputs to be considered in the iteration

The iteration goal is to decompose the monolithic into microservices while addressing the following main drivers:

```
* User Stories: US1
* Quality Attributes: QA1
* Concerns: CON3
```

```
* Architecture Constraints: CRN1, CRN2, CRN3
* Technology Constraints: TC1, TC3, TC4
```

Step 3: Choose one or more elements of the system to decompose.

The elements to refine are the previously mentioned drivers.

The monolithic will be decomposed into microservices.

Step 4: Choose design concepts that satisfy the selected drivers

Design Decisions and Location	Rationale and Assumptions
Update the existing Domain Driven Model for the application	The diagram describing the relationships between the different aggregate roots, entities and value objects, created in the first part of the project, must be updated due to the addition of new features.
Create a Context Map for the application	A context map specifying all the contexts of the application and their relationships must be created.
Create a Deployment Diagram for the application	There must be a diagram describing the different components and their interactions.
Create Sequence Diagrams for the application	Diagrams to demonstrate the flux of information related to the primary features.

Design Decisions and Location	Rationale and Assumptions
Create a Components Diagrams for the application	There must be a diagram describing the different components and their interactions.

Step 5: Instantiate architectural elements, allocate responsibilities, and define interfaces

MicroService Migration

The monolithic was decomposed into microservices by root entities defined in the **Domain Driven Model**. This can also be seen as business capabilities.

Microservice Name	Business Capability	Responsibility
User Auth	User management	Management of users, authentication and authorization.
Sandwich	Sandwich Management	Management of sandwichs.
Order	Order Management	Management of orders made by costumers.
Shop	Shop Management	Management of shops.
Promotions	Promotions Management	Management of local and global promotions of sandwichs.

In the first part of the project, it was decided to merge the manager and costumer aggregate in the microservice User Auth to address **CON3**.

Patterns

Pattern	Description
Strangler Fig Apllication	This pattern aims to incrementally re-write small parts of the codebase until we have strangled all our old codebase and we can be totally removed it. Using this pattern rollbacks are easier, reduces the risks when the codebase is updated. https://www.freecodecamp.org/news/what-is-the-strangler-pattern-in-software-development/
API Composition	It is a run-time composition that loads data in-memory through an API Composer Service built on the top of two or more services. This pattern should be used whenever is possible, but because of its in-memory load nature must be used carefully and only when the data to load is relatively small. https://www.linkedin.com/pulse/api-composition-pattern-microservices-arpit-bhayani/

Microservice Comunication

Step 6: Sketch views and record design decisions

Domain Driven Design

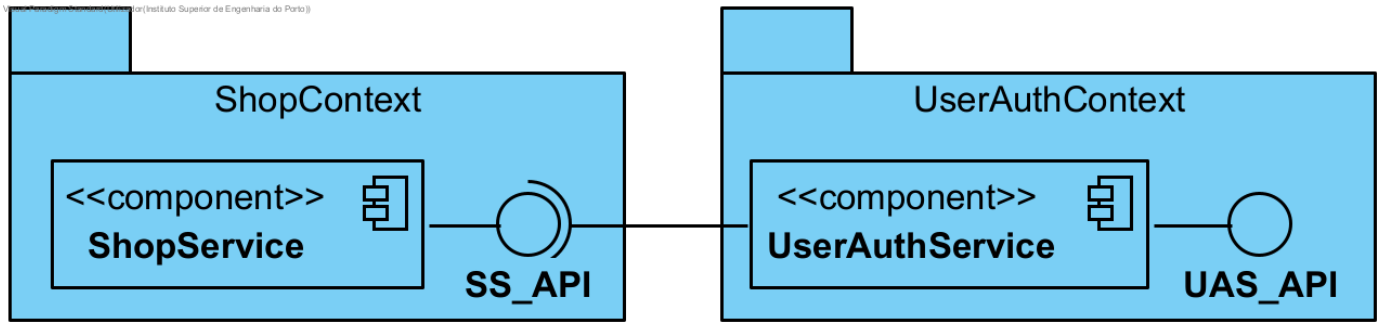
The Domain Driven Design model remains almost the same. The updates were:

User Aggregate

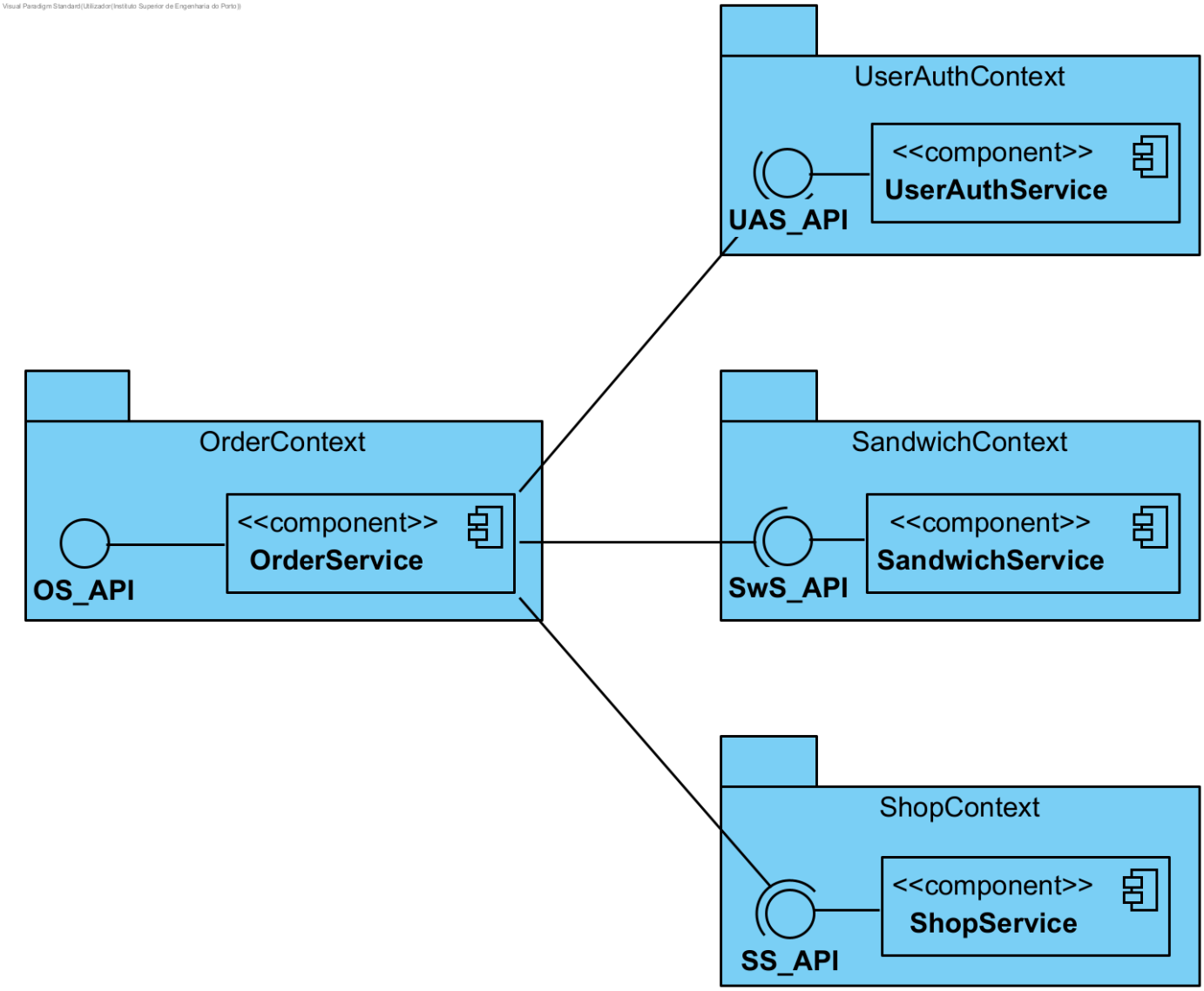
The Manager Aggregate and Costumer Aggregate in order to address the **CON3**. The User class has the attributes for the authentication and authorization and is the superclass of Manager and Costumer. Manager and Costumer classes remains with their unique attributes from the last iteration.

Component Diagrams

In order to increase the visibility of the component diagrams, the group decided to create one diagram for each context with the connections/binding made to its API.



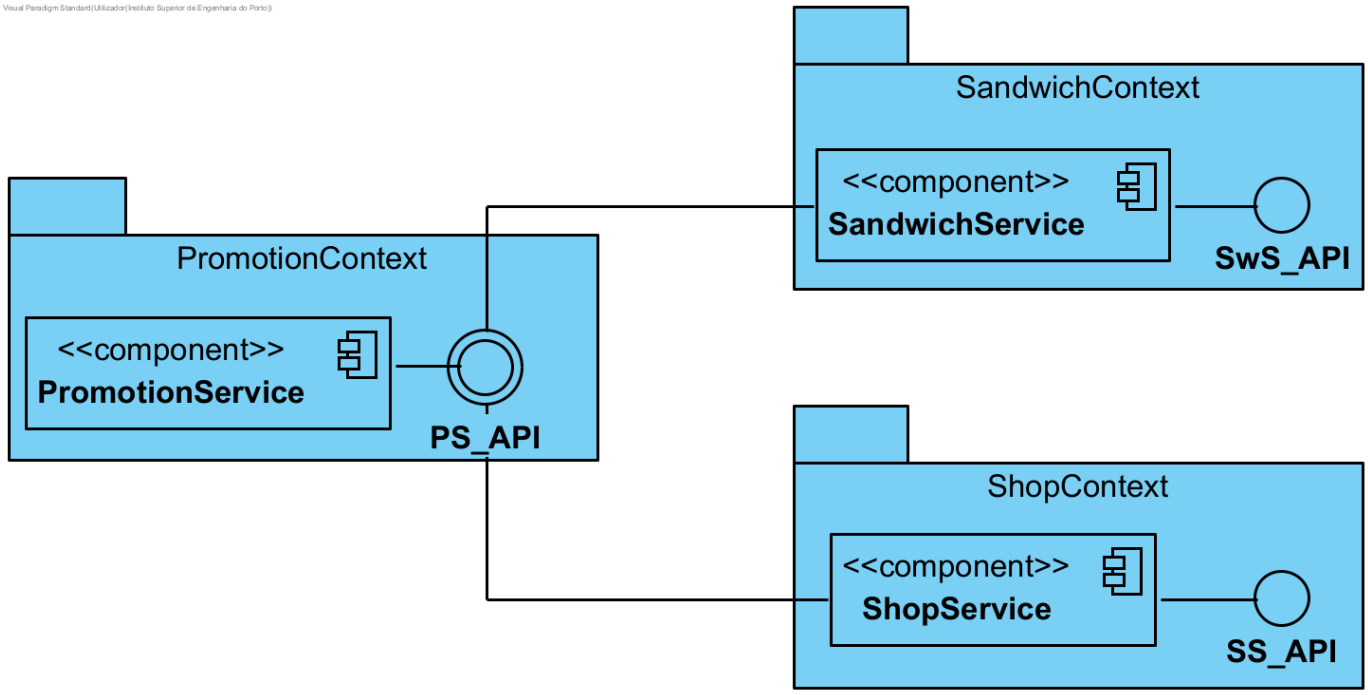
UserAuthService uses the Shop API to verify if the shop associated to the manager exists.



OrderService uses the Shop API to verify if the shop associated to the order exists.

OrderService uses the UserAuth API to verify if the costumer associated to the order exists.

OrderService uses the Sandwich API to retrieve all the sandwiches within an order.



SandwichService uses the Promotion API to retrieve the promotions of a sandwich.

ShopService uses the Promotion API to retrieve the promotions of a shop.

Context Map

Domain Driven Design

The context map describes the different microservices that were created for the prototype, having the monolithic built in the first part in mind. The team achieved, after many meetings, this context map that explains and demonstrates how the contexts communicate with each other. The definition of some properties used in the context map are explained below.

- OHS (Open Host Service): This protocol defines and gives access to different contexts, which means that it can communicate with other bounded contexts through domain events that use different bounded context.
- Upstream and Downstream: Upstream term define the context that transmit information/data to other contexts and can influence the downstream contexts. Upstream contexts usually have impact in different context because of the fact that they have data for the other contexts to work properly.

Step 7: Analyse current design, and review iteration goal + achievement of design purpose

Iteration 1		
Not Addressed	Partially Addressed	Completely Addressed
CRN3	QA4	QA1
TC1	CON3	QA3
-	CON4	TC2
-	CRN1	TC4
-	CRN2	TC5
-	TC3	-

Second Iteration

Step 1: Considered Inputs

This iteration there weren't inputs updates.

Step 2: Establish iteration goal

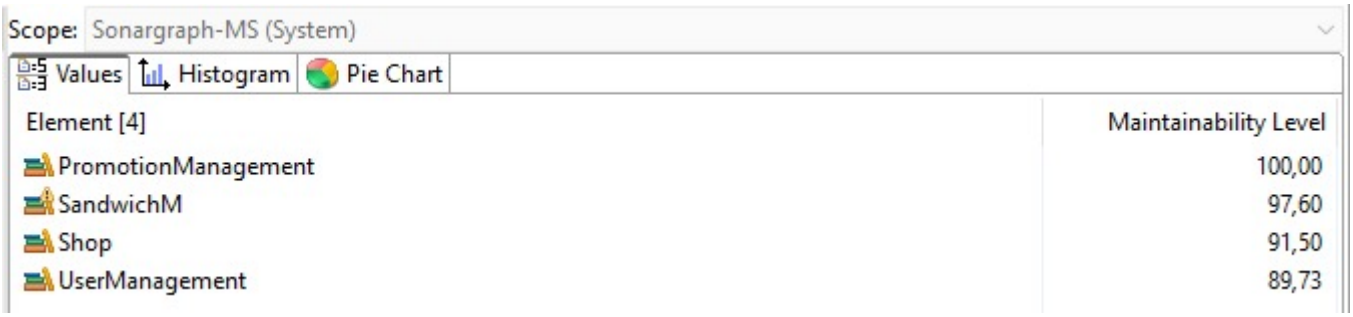
The iteration goal is to re-evaluate the microservices communication and create a data migration process. The drivers related to this are the following:

CRN1 – Microservices;

CRN2 – Data Migration;

CRN3 – Deployment.

Acording QA1 is expected to have a high percentage of maintenability. Using the sonorgraph this percentage for the project and for each microservice was measured respectively:



Step 3: Choose what to refine

The elements to refine is the communication between microservices, that at the moment are made by HTTP requests and must be by GraphQL requests.

Step 4: Choose design concepts that satisfy the selected drivers

Design Decisions and Location	Rationale and Assumptions
Create a Microservices Communication Diagram	The diagram describing the communication between the different microservices.

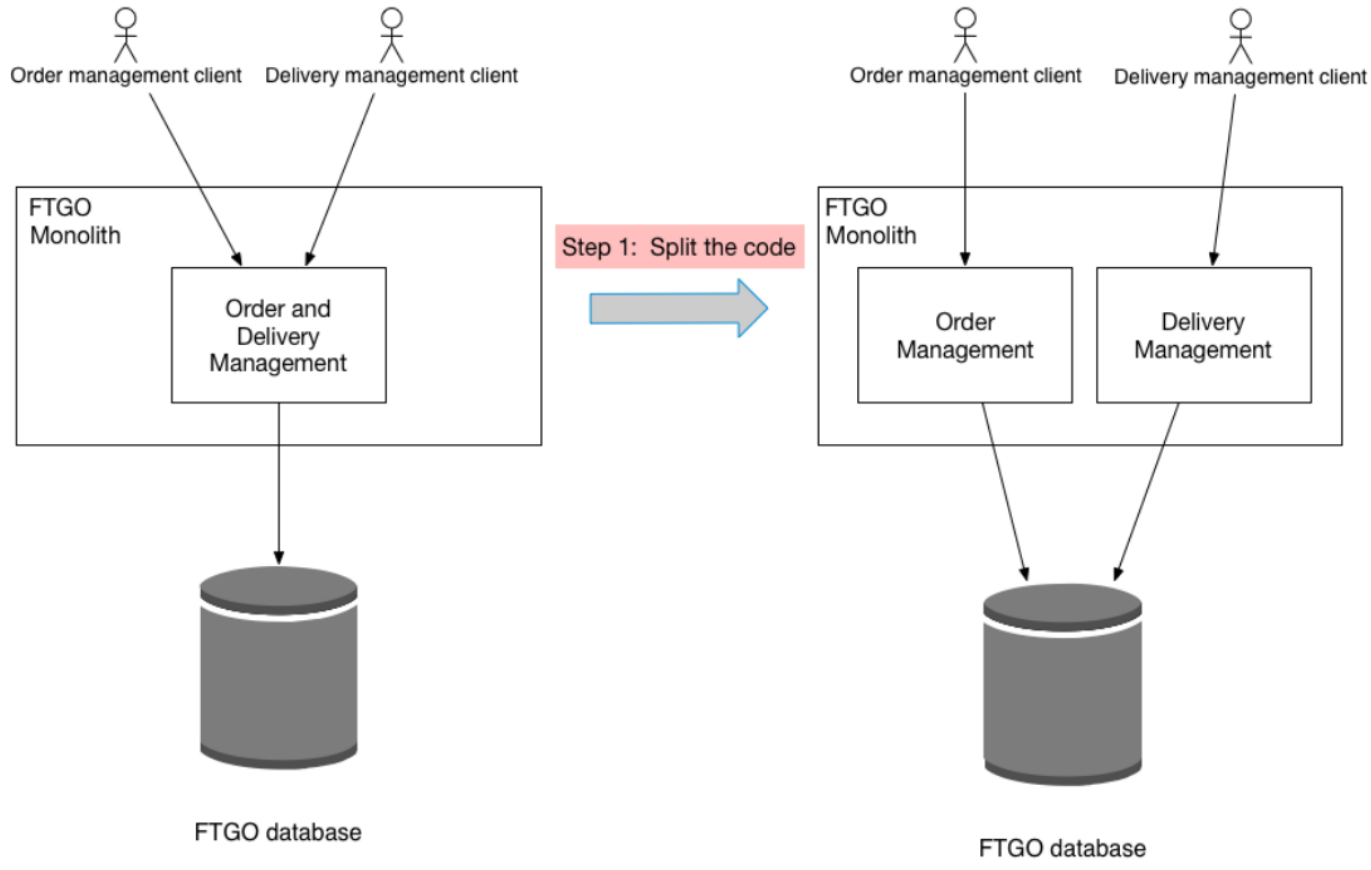
Design Decisions and Location	Rationale and Assumptions
Create a Components Diagrams for the data migration	There must be a diagram describing the different components, involved in the data migration and their interactions.
Create a Deployment Diagram	There must be a diagram describing the deployment of the prototype.

Step 5: Instantiate architectural elements, allocate responsibilities, and define interfaces

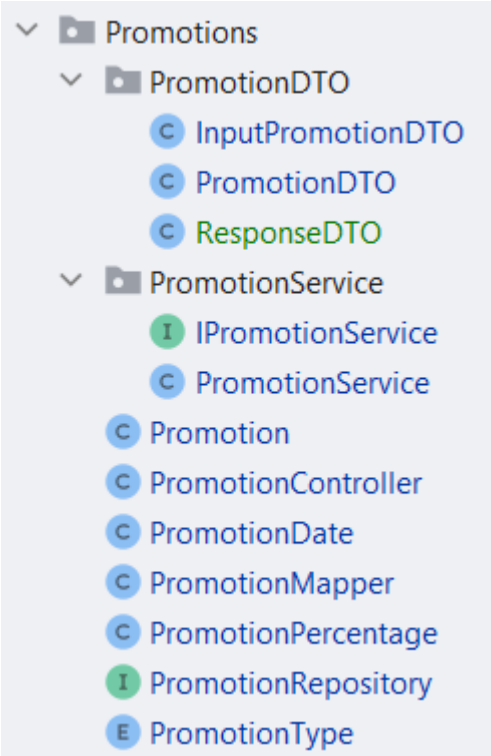
Data Migration

The team members never migrated a monolithic project to a microservices project so firstly we decided follow step by step the Strangler Fig Apllication pattern.

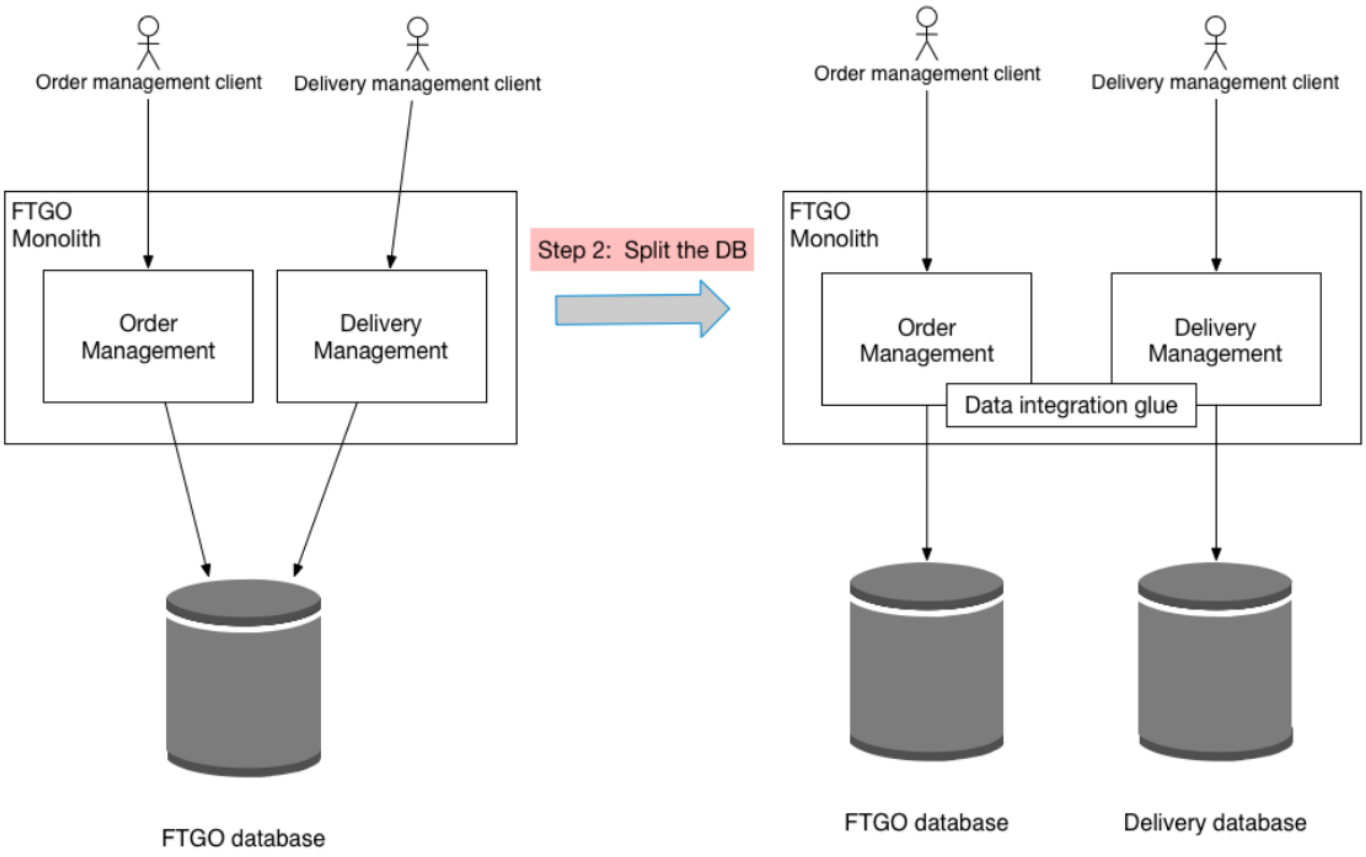
Step 1



The first step consists in separate the codebase in different aggregates. This step was already done for the project first iteration.



Step 2



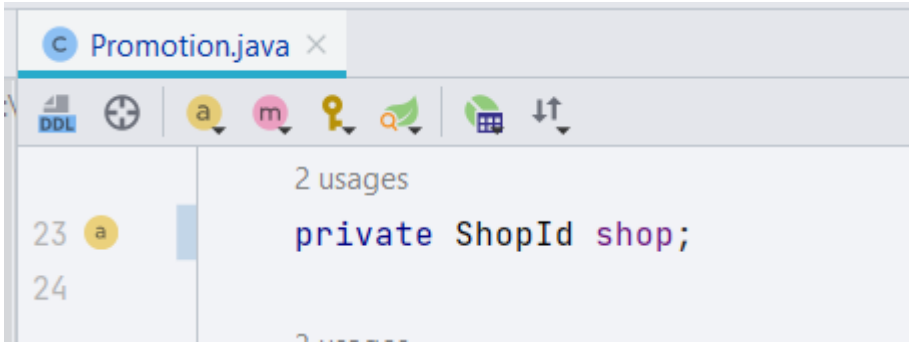
Secondly is needed to create a new database for the aggregate that will be converted to a microservice. Besides that is necessary to have communication beetween

aggregates by the data integration glue (foreign keys beetween aggregates database tables).

The database creation:

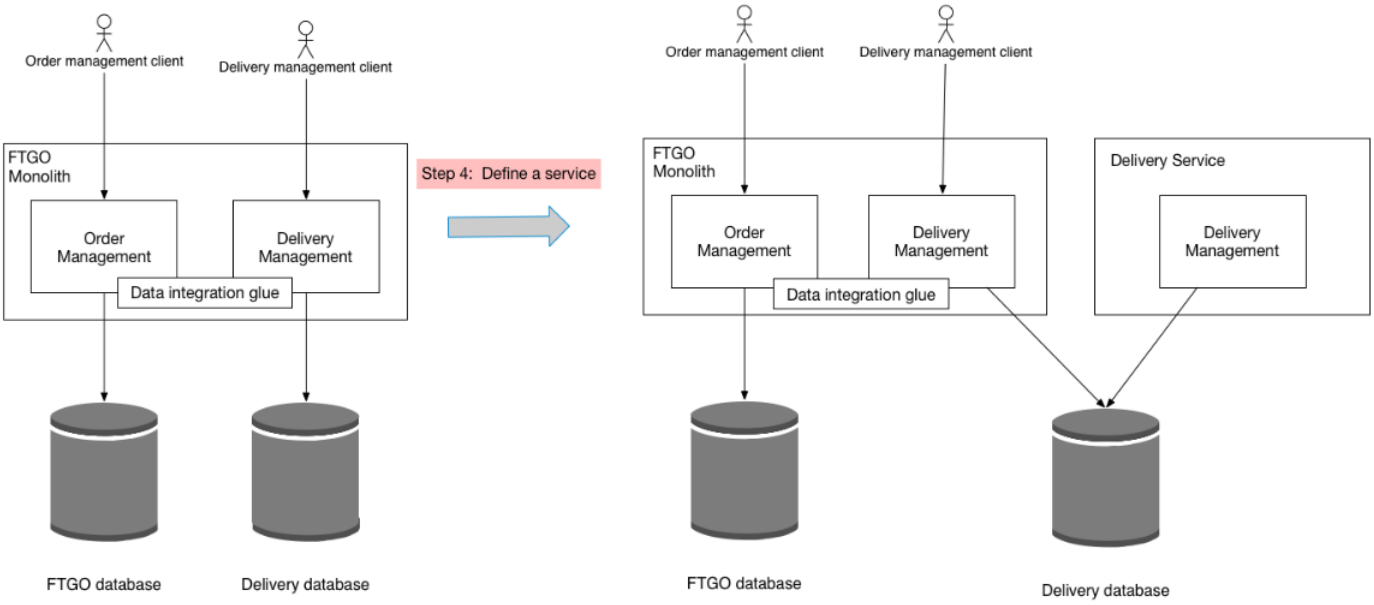
```
mysql> create database promotiondb;
Query OK, 1 row affected (0.01 sec)

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| arqsoftdb          |
| information_schema |
| mysql              |
| performance_schema |
| promotiondb        |
| sys                |
+--------------------+
6 rows in set (0.00 sec)
```

Data Integration Glue:

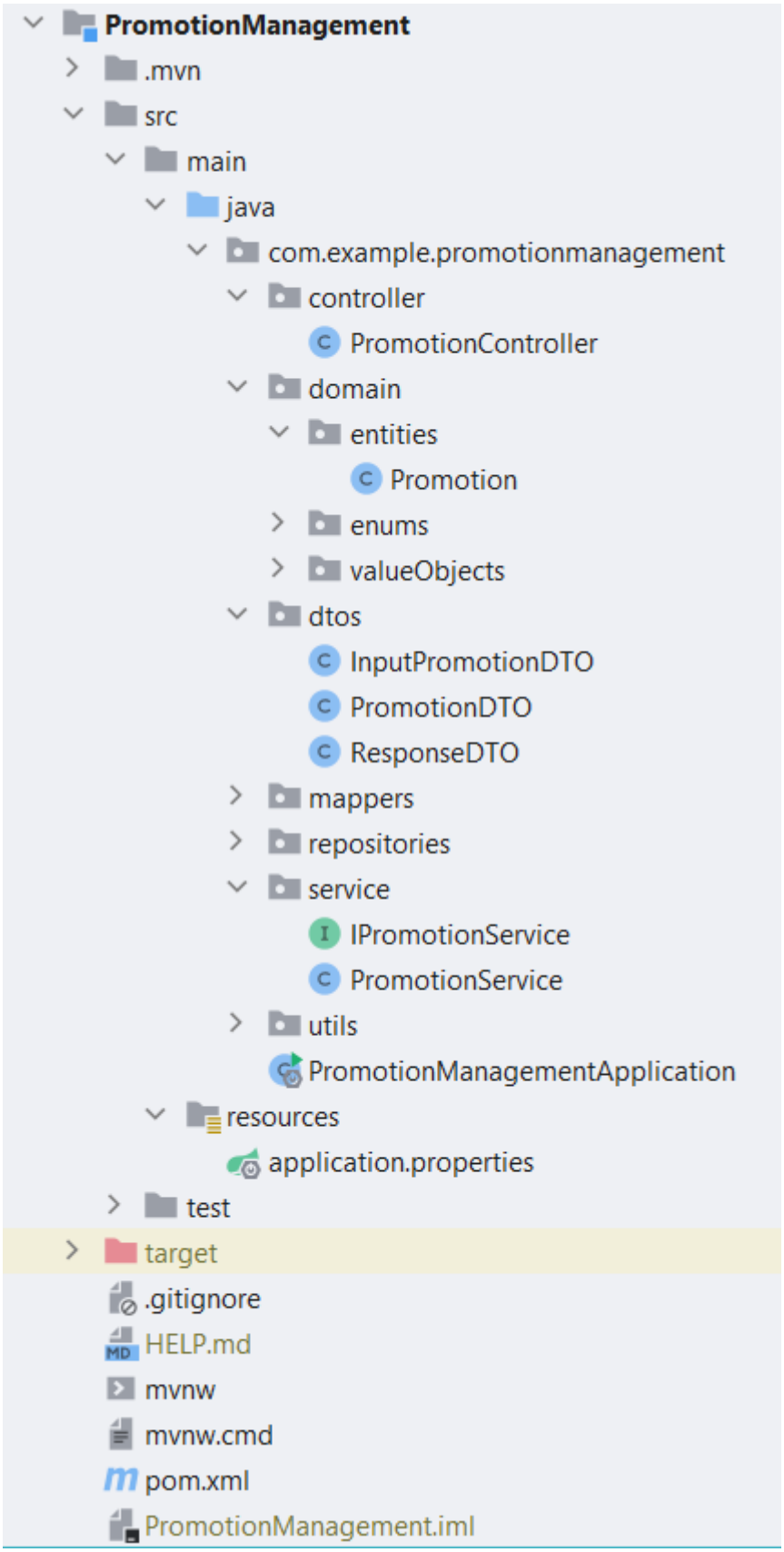


Now is not used the object shop but the shop Id.

Step 3

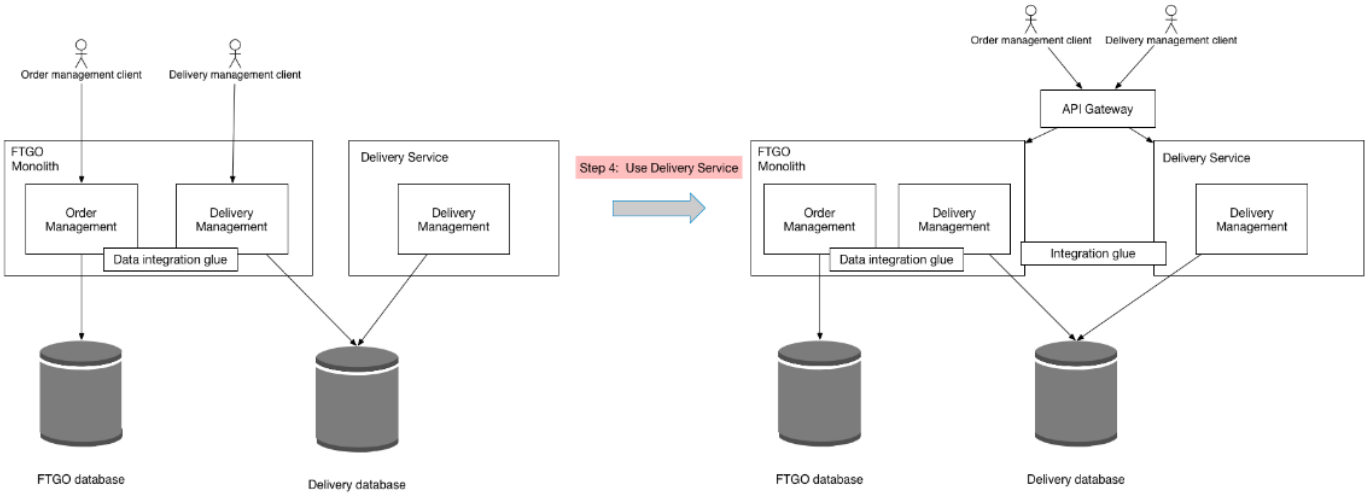


The next step is to create a new project that is going to be the new microservice, deploy it and connect it to the database created in the step before.




```
spring.datasource.url=jdbc:mysql://localhost:3306/promotiondb
spring.datasource.username=admin
spring.datasource.password=admin2223
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.show_sql=true
springdoc.api-docs.path=/api-docs
```

Step 4



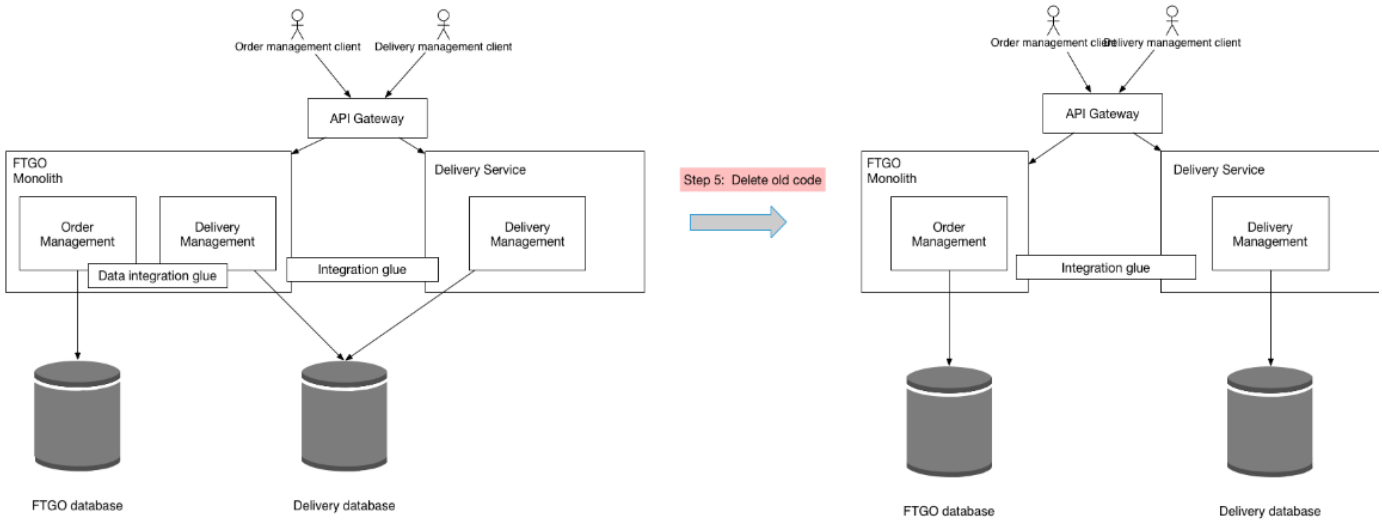
In this step the client does not communicate anymore directly with the monolith management solution but by an API Gateway. Besides that, the codebase is gradually being copied and tested from the monolith solution to the new microservice.

```
type Mutation{
    addPromotion(inputPromotionDTO : InputPromotionDTO!) : PromotionDTO
    deletePromotionById(promotionId: Int!) : Int
    editPromotion(promotionId: Int!, inputPromotionDTO : InputPromotionDTO!): Int
}

type Query{
    listPromotions: [PromotionDTO]
    getPromotionById(promotionId: Int!) : PromotionDTO
}

type PromotionDTO {
    promotionId: ID
    type: String
    sandwichId: ID
    shopId: ID
    percentage:Float
    startDate:String
    endDate:String
}

input InputPromotionDTO {
    sandwichId: ID
    type: String
    shopId: ID
    percentage:Float
    startDate:String
    endDate:String
}
```

```
mutation{
addPromotion(
    inputPromotionDTO:
  {sandwichId:1
  type:"Global"
  shopId:5
  percentage:0.3
  startDate:"2022-09-02"
  endDate:"2022-09-03"}){
    promotionId
    type
    sandwichId
    shopId
    percentage
    startDate
    endDate
  }
}
```

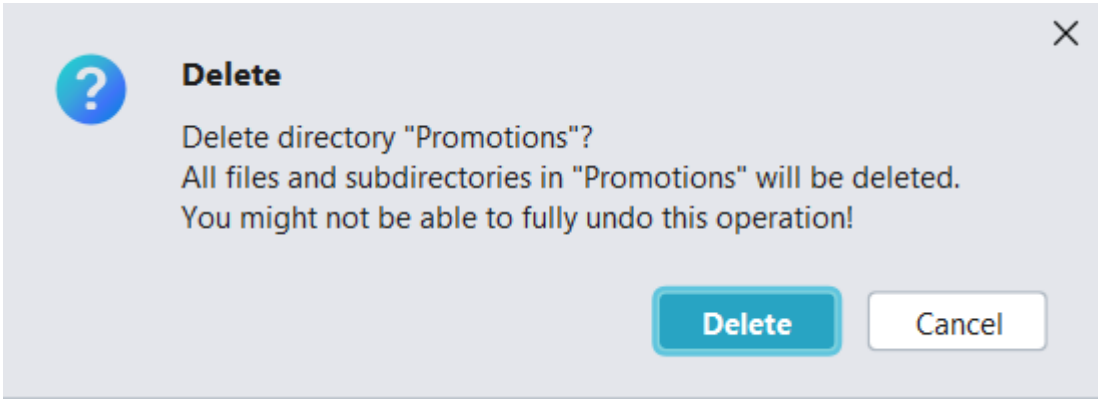
```
query{
  getPromotionById(promotionId:4){
    promotionId
    type
    sandwichId
    shopId
    percentage
    startDate
    endDate
  }
}
```

Step 5



The final step is delete the aggregate codebase that was copied to the new microservice.

This is a valid way of data migration and also go to the encounter the selected drivers of scalability, maintainability and modifiability.



GraphQL

GraphQL is a query language that allows the user to request only the information that he needs. The APIs are organized in terms of types and fields, not endpoints so, we can access the full capabilities of our data from a single endpoint. With GraphQL, the addition of new fields and types to an API dont affect the already existing queries, helping in the modifiability of the application.

The monolith application was built in a way that all request answer with a generic answer (ResultDTO, composed by a generic object_Result, responsible for retrieving the data or an error message, and a statusCode that allows the requester to know the code of the answer). Due to that generic implementation, the team was not able to implement the framework. One way to resolve this problem is to use the *JSON.stringify()* but using this method there isn't any purpose to use GraphQL. So,

instead to return ResultDTO, will be returned the DTO itself. For that reason, any exception raised in the process of the query or mutation, will result in a GraphQL Error since he cannot return the expected DTO.

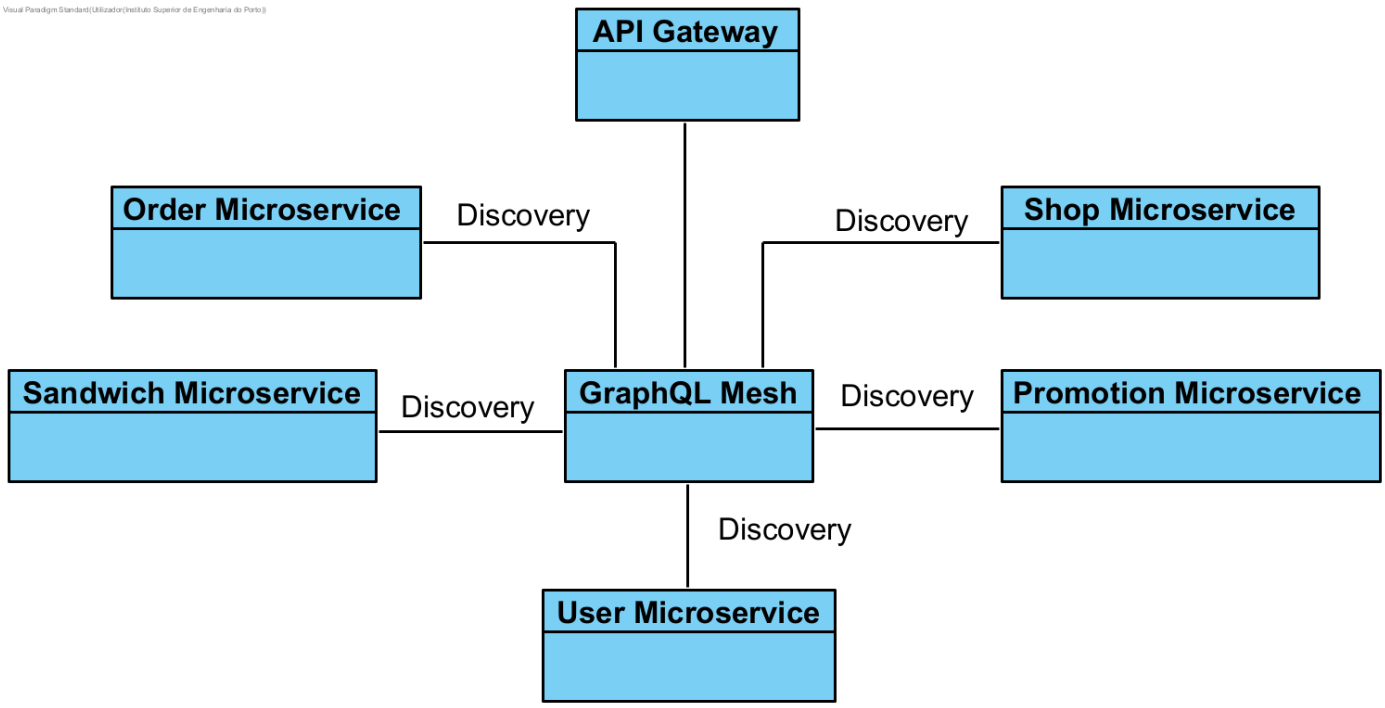
Nevertheless, the team was able to understand the advantages of a framework like GraphQL. The customization in every request is in fact an asset that will increase the performance of most applications, especially if the applications have microservices communicating with each others.

Deployment

Docker will be used to create containers for the different components and microservices of the prototype.

Step 6: Sketch views and record design decisions

Service Discovery Diagram

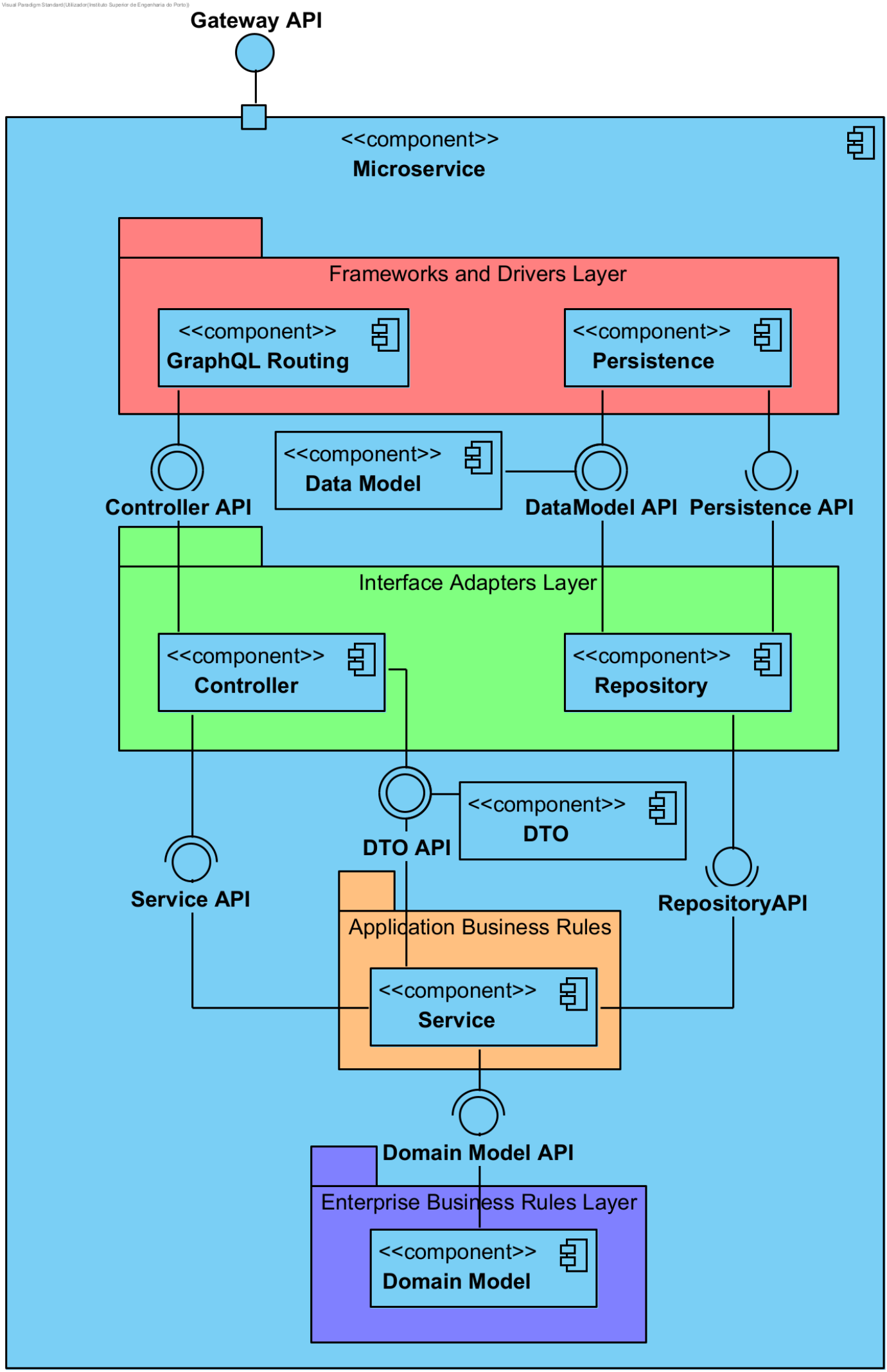


GraphQL Mesh represents a central server that will register all the microservices, within the network. The addresses are composed by the host and port.

The other microservices comunicate by the GraphQL Mesh in the central server, giving its information to be used.

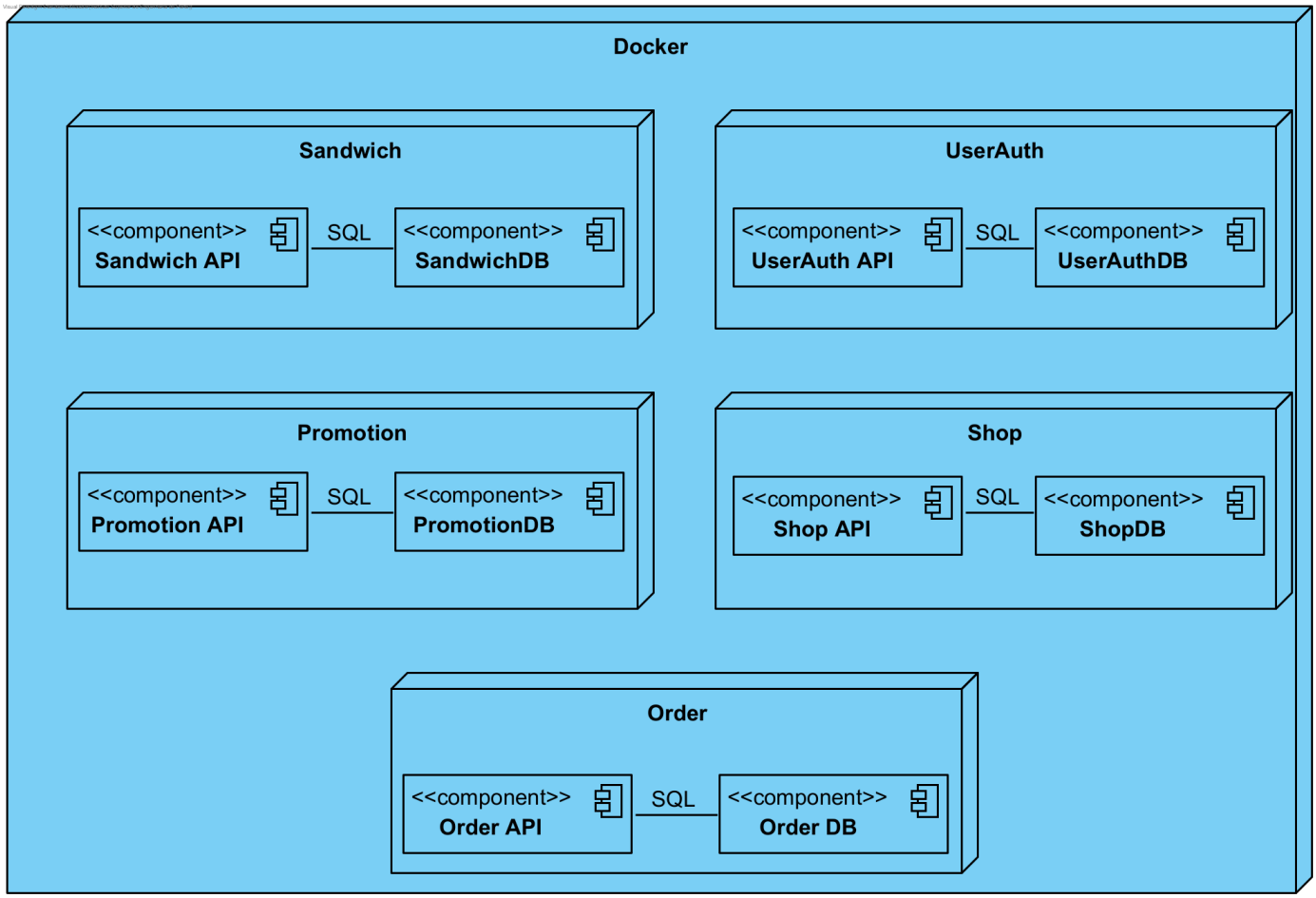
All the microservices registered in the central server are available for everyone to use. To use them it's not necessary to know the address, just the name.

Component Diagram

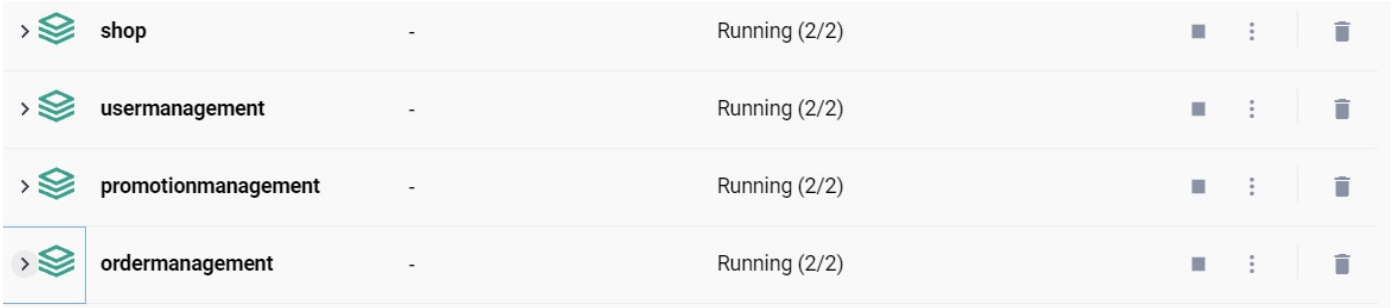


In the monolitic application the onion architeture was used and the team decided to implement each microservice with the same architeture because it's the one the team is used to work. The main difference are in the routing, now is done by GraphQL instead of HTTP and there is an API Gateway.

Deployment Diagram



Docker is the tool that supports the deployment of the system. There is a big container which will have smaller one with each microservice and respetective database.



Step 7: Analyse current design, and review iteration goal + achievement of

design purpose

Kanban Board

Iteration 2		
Not Addressed	Partially Addressed	Completely Addressed
-	QA4	QA1
-	CON1	QA3
-	-	TC1
-	-	TC2
-	-	TC3
-	-	TC4
-	-	TC5
-	-	CRN1
-	-	CRN2
-	-	CRN3
-	-	CON2
-	-	CON3
-	-	CON4

Road Map

Person	Work	Completed
Luís	Promotion Migration	Completely Addressed
Luís	GraphQL Implementation	Completely Addressed
Luís	ADD - 1st Iteration	Completely Addressed
Luís	ADD - 2nd Iteration	Completely Addressed
Luís	ATAM	Completely Addressed

Person	Work	Completed
Daniel	Sandwich Migration	Completely Addressed
Daniel	GraphQL Implementation	Completely Addressed
Daniel	ADD - 1st Iteration	Completely Addressed
Daniel	Gateway Implementation	Partially Addressed
Rui	Shop Migration	Completely Addressed
Rui	GraphQL Implementation	Completely Addressed
Rui	ADD - 1st Iteration	Completely Addressed
Rui	Contract-based testing	Partially Addressed
Daniela	US01 - Register User	Completely Addressed
Daniela	User Migration	Completely Addressed
Daniela	Order Migration	Completely Addressed
Daniela	GraphQL Implementation	Completely Addressed
Daniela	Deployment	Partially Addressed

Architecture Trade-off Analysis Method (Lightweight ATAM)

Risks

One of the risks of the previous part remains, currently there is no data backup, which means that, if for some reason, there is a data corruption or a database delition by accident, it won't be possible to be restore the data to its original format. However, this its not a big risk due to the size of the project. GraphQL Denial of Service also represents a risk to the project. One way to solve part of this problem is to implement the GraphQL with a version higher than 7.0.0, so the team used 16.6.0 version. After five tries the application is blocked for five seconds due to the rate limit plugin. The number of tries and the seconds are configurable.

```
"@graphql-mesh/plugin-rate-limit": "0.2.4",
```

```
plugins:
  - rateLimit:
      config:      You, 7 minutes ago • Uncommitted c
        # Add as many rules as you want
        - type: Query
          field: foo
          max: 5 # requests limit for a time period
          ttl: 5000 # time period
          identifier: '{context.userId}'
```

Non-Risks

The use of Spring framework, Java, ASP.NET and MySQL as they are tested and stable tools. The services discovery tool used, GraphQL Mesh, can also be seen as a

non-risks due to its usability, testability and support. Due to the use of value objects and domain primitives the validation of entity properties is validated upon the instantiation of the object (Fast Fail). All exceptions raised within the system are caught and processed. A secure authentication method is used to prevent the access to functionalities restricted to a specific user type.

Sensitivity point

One possible sensitive point can be the user authentication server, that when down, will prevent the use of the prototype due to security purposes.

Trade-off

The API Gateway can be seen as a trade-off since it adds more complexity to the prototype. The GraphQL Playground interface must communicate with the gateway instead of communicating directly with the target microservices, which will reduce performance. The team thinks this decision is the most correct since it keeps the prototype more organized and maintainable. Another trade-off is the use of service discovery (GraphQL Mesh) which, besides requiring more time for implementation, it also adds more complexity. Although, it increases the security, by hiding the microservices addresses, it also increases the scalability of the prototype. The data migration is also a topic that has to be discussed under this subject. The inexperience in data migration added more complexity and workload to the team, but, after a study the team came across with a solution that easily increases the modifiability, scalability and maintainability of the project.

Risk Matrix

Risk Criteria	Scalability	Availability	Performance	Security	Data Integrity	Total Risk
US01						
QA1	2	7	3	8	8	28
QA2	2	7	4	8	8	29

Risk Criteria	Scalability	Availability	Performance	Security	Data Integrity	Total Risk
QA3	2	6	4	9	8	29
QA4	2	6	4	9	8	28

Low : [0-3] Medium : [4-6] High : [7-9]