

Repository for Project

ARQSOF 2022/2023

Roadmap

The software being developed is a brownfield system, so it is proposed the following roadmap for each ADD iteration

Iteration 1

- Goal: Refine software architecture
- Description: In this iteration it is intended that the design of the previously developed software architecture is refined to include new architectural drivers
- Design Concepts:
 - Refine the Domain Model
 - Decompose the monolithic GFAB into a microservice-based application

Iteration 2

- Goal: Support quality attribute scenarios and concerns
- Description: In this iteration it is intended that the design of the quality attribute scenarios and concerns is settled
- Design Concepts:
 - Analyze, and choose Tactics
 - Relate Architectural and Deployment Patterns with choosen Tactics
 - Relate Externally Developed Components with choosen Tactics

Note

- Despite having this organization some drivers might be related to each other and can possible be addressed in various iterations.

Architectural Drivers

Design Purpose

A well detailed plan for the migration of the monolith legacy system to a new micro-services model is to be designed, implemented, tested and deployed in order to promote the minimum amount of damage to the current system deployed

Quality Attributes

ID	Scenario	Quality Attribute	Importance	Technical risk
QA-1	The use of GraphQL is mandatory.	Architecture Performance	H	H
QA-2	The prototype to be developed must be accessible using a web browser or a simple application that transfers data to and from a server that supports some protocols allowing the use of API, such as Curl and Postman.	Accessability	H	M
QA-3	A distributed Microservices architecture is the target.	Architecture Performance	H	M
QA-4	The OpenAPI (Swagger) specification must be used for API documentation..	Learnability	L	L
QA-5	Only open-source tools and technologies are allowed, except some mentioned.	Security Design	H	L
QA-6	Maintainability is a characteristic that the team has to carefully consider during the architecture design. Appropriate metrics should be used.	Code Coverage	M	L
QA-7	Testing on various quality dimensions is mandatory for the prototype, including the business rules captured in domain layers and what can ensure the correct functioning of the application and its API. More than the number of tests, it is important to explain and document what the tests allow to verify.	Code Coverage	H	M

Primary Functionality

ID	Use Case	Description	Priority	Difficulty
UC1	Create Sandwiches	The sandwich is labeled with designation, price and a description with it's language	High	Medium
UC2	List Sandwiches	View a characterization regarding the sandwich's designation, description and price	Medium	Low
UC3	Register Shop	The shop is labeled with designation, an address, a manager and opening hours	High	High
UC4	List Shop	View a characterization regarding the shop's designation, address and manager's name	Medium	Low
UC5	Register Order	The order is labeled with specific day and shop, the items and their quantities	High	High

ID	Use Case	Description	Priority	Difficulty
UC6	List Order	View a characterization regarding the order's number, promotion, items, and price	Low	Low
UC7	Register Customer	The register is labeled with name, a tax identification number, an address, an email and authentication data	High	Low
UC8	List Customer	View a characterization regarding the customer's username, email, tax identification number, and address	Low	Low
UC9	Create promotion	The promotion is labeled with type, the percentage and time effect	Medium	Medium
UC10	List promotion	View a characterization regarding the promotion's type, percentage and time of effect	Low	Low
UC11	Login	The user logs in to the prototype	High	Low
UC12	Define N minimum/maximum of deliveries	The shop can define the N minimum and maximum of deliveries for each period of N minutes	Medium	Medium
UC13	List deliveries	The manager can list the deliveries for certain period of time	Medium	Medium
UC14	Register deliveries	The manager registers a delivery	Medium	High

Architectural Concerns

Architectural concerns encompass additional aspects that need to be considered as part of architectural design but that are not expressed as traditional requirements. They can be general, specific, internal or issues.

General Concerns

These are "broad" issues that one deals with in creating the architecture, such as establishing an overall system structure.

Specific concerns

These are more detailed system-internal issues such as exception management, dependency management, configuration, logging, authentication, authorization, caching, and so forth that are common across large numbers of applications.

Internal Concerns

These requirements are usually not specified explicitly in traditional requirement documents, as customers usually seldom express them. Internal requirements may address aspects that facilitate development, deployment, operation, or maintenance of the system.

Issues

These result from analysis activities, such as a design review , so they may not be present initially. For instance, an architectural evaluation may uncover a risk that requires some changes to be performed in the current design.

Our concerns:

ID	Concern
CRN-1	Accessible from web browser, Postman or Curl.
CRN-2	Establishing an overall initial system structure.
CRN-3	Leverage the team's knowledge about Java technologies, including Spring, GraphQL, ASP.NET and also MongoDB knowledge.
CRN-4	Allocate work to members of the team.

Constraints

ID	Constraint
CON-1	Explore the microservice architecture.
CON-2	The prototype presentation must include the build and the execution at the command line.
CON-3	Local method calls need to be replaced by synchronous remote calls, or better options.
CON-4	Service discovery is desirable.
CON-5	Direct dependencies in the database are to be eliminated depending on the adopted data management strategies.
CON-6	This prototype must be developed in the near six weeks.
CON-7	Integrate metrics (Sonargraph-Explorer).
CON-8	An on-prem solution is desirable with deployment based on containers

Technologies Explorations

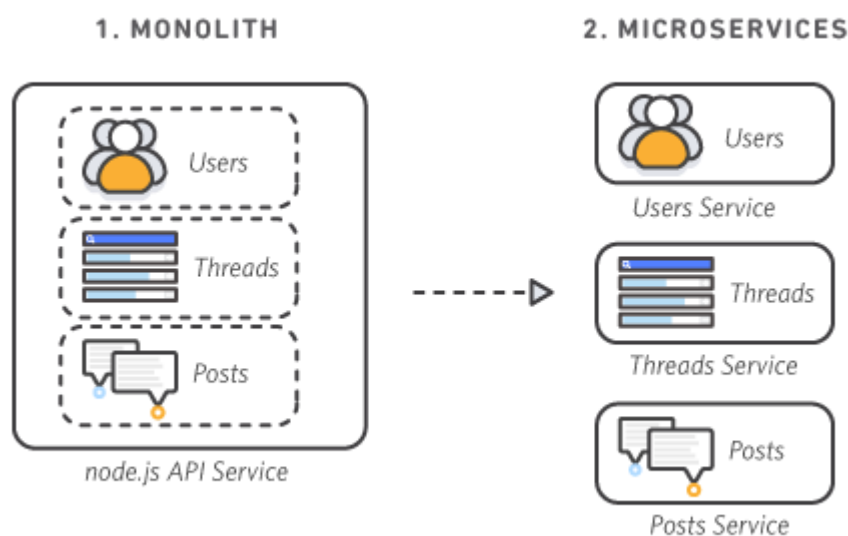
Microservices

Microservices are an architectural and organizational approach to software development where software is composed of small independent services that communicate over well-defined APIs. These services are owned by small, self-contained teams. Microservices architectures make applications easier to scale and faster to develop, enabling innovation and accelerating time-to-market for new features.

Why use microservices over monolithic architecture

With monolithic architectures, all processes are tightly coupled and run as a single service. This means that if one process of the application experiences a spike in demand, the entire architecture must be scaled. Adding or improving a monolithic application's features becomes more complex as the code base grows. This complexity limits experimentation and makes it difficult to implement new ideas. Monolithic architectures add risk for application availability because many dependent and tightly coupled processes increase the impact of a single process failure.

With a microservices architecture, an application is built as independent components that run each application process as a service. These services communicate via a well-defined interface using lightweight APIs. Services are built for business capabilities and each service performs a single function. Because they are independently run, each service can be updated, deployed, and scaled to meet demand for specific functions of an application.



Breaking a monolithic application into microservices

Benefits of Microservices

- Agility

Microservices foster an organization of small and independent teams. Teams act within a small and well understood context, and are empowered to work more independently and more quickly.

- Flexible Scaling

Microservices allow each service to be independently scaled to meet demand for the application feature it supports. This enables teams to right-size infrastructure needs, accurately measure the cost of a feature, and maintain availability if a service experiences a spike in demand.

- Easy Deployment

Microservices enable continuous integration and continuous delivery, making it easy to try out new ideas and to roll back if something doesn't work.

- Technological Freedom

Microservices architectures don't follow a "one size fits all" approach. Teams have the freedom to choose the best tool to solve their specific problems.

- Reusable Code

Dividing software into small, well-defined modules enables teams to use functions for multiple purposes. A service written for a certain function can be used as a building block for another feature. Because of this developers can create new functionalities without writing the code from scratch.

- Resilience

Service independence increases an application's resistance to failure. In a monolithic architecture, if a single component fails, it can cause the entire application to fail. With microservices, applications handle total service failure by degrading functionality and not crashing the entire application.

GraphQL

What is GraphQL

GraphQL is a query language for APIs and a runtime for fulfilling those queries with your existing data. GraphQL provides a complete and understandable description of the data in your API, gives clients the power to ask for exactly what they need and nothing more, makes it easier to evolve APIs over time, and enables powerful developer tools.

Why use GraphQL over Rest API

- GraphQL is faster

GraphQL is way faster than other communication APIs because it facilitates you to cut down your request query by choosing only the specific fields you want to query.

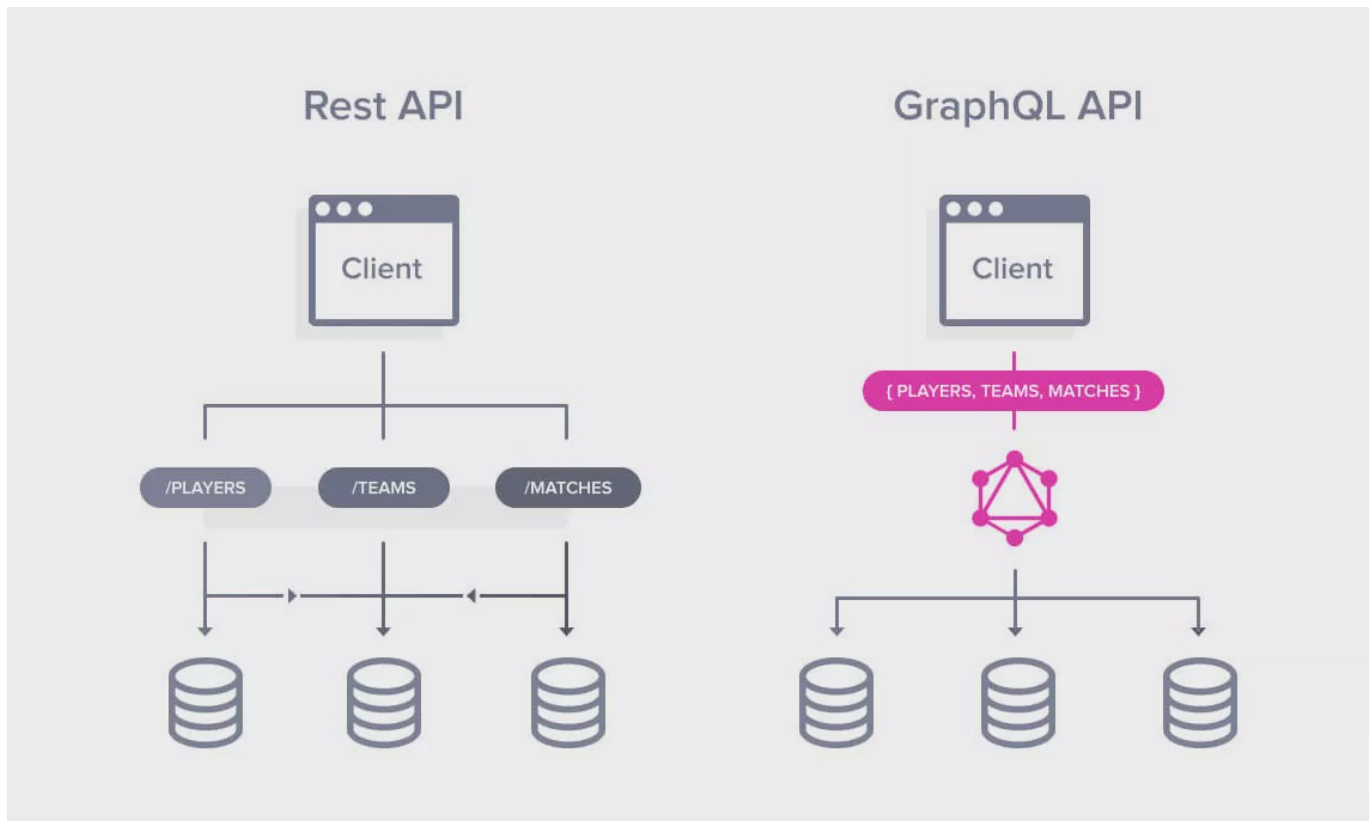
- Best for complex systems and microservices

We can integrate multiple systems behind GraphQL's API. It unifies them and hides their complexity. The GraphQL server is also used to fetch data from the existing systems and package it up in the GraphQL response format. When we have to migrate from a monolithic backend application to a microservice architecture, the GraphQL API can help us to handle

communication between multiple microservices by merging them into one GraphQL schema (gateway).

- No over-fetching and under-fetching problems

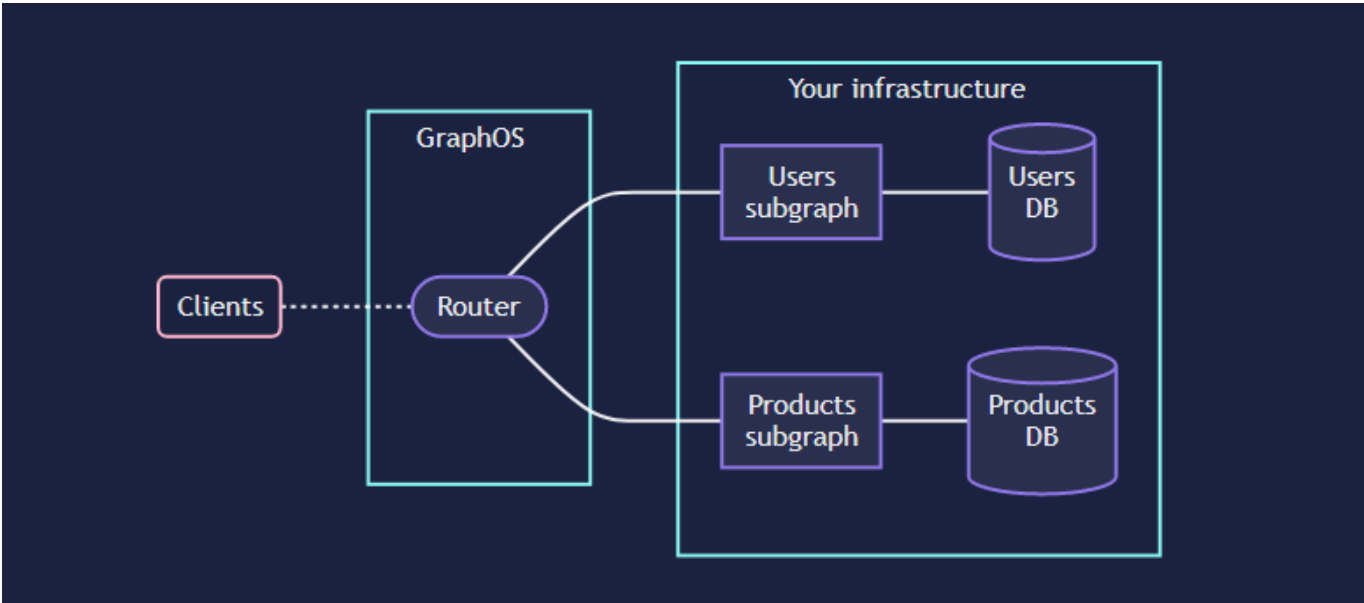
The main advantage of GraphQL over REST is that REST responses contain too much data or sometimes not enough data, which creates the need for another request. GraphQL solves this problem by fetching only the exact and specific data in a single request.



Apollo

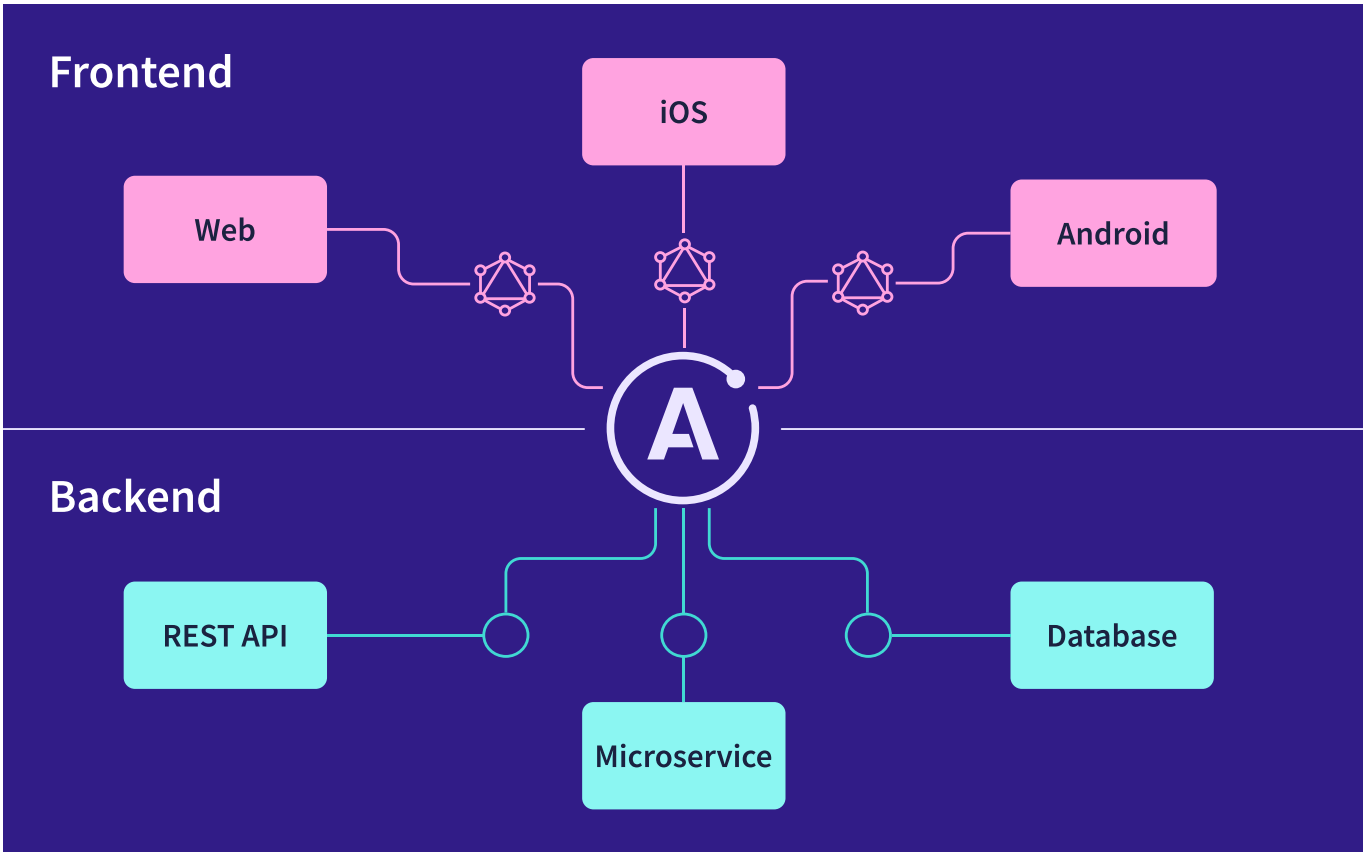
What is Apollo

Apollo is a platform for building a supergraph, a unified network of all your data, services, and capabilities that connects to your application clients (such as web and native apps). At the heart of the supergraph it uses GraphQL.



Apollo server

Apollo Server is an open-source, spec-compliant GraphQL server that's compatible with any GraphQL client. It's the best way to build a production-ready, self-documenting GraphQL API that can use data from any source.



Iteration 1

Step 1

- Goal: Review Inputs

- Possible Questions:

Question	Answer
Inputs available and correct?	As far as the feedback of the stakeholder, the defined architectural drivers are correct
All drivers available?	As far as what was retained from the domain problem and the stakeholder feedback, all drivers have been defined
Is it clearly established what is the purpose for the design activities?	Yes, the purpose of this iteration is to structure the software architecture in a coarse-view
Have primary functionality and quality attribute scenarios been prioritized (ideally by the most important project stakeholders)?	Yes, but in this iteration we will only structure our system.
Are initial architectural concerns defined?	Yes

Step 2

- Goal: Establish iteration goal by selecting drivers

Kanban Board

Not Addressed	Partially Addressed	Addressed
QA-1		
QA-3		
QA-5		
CRN-2		
CRN-3		
CRN-4		
CON-1		
CON-2		
CON-3		
CON-4		
CON-5		

Step 3

- Goal: Choose elements of the system to refine

Since we are working on a brownfield system, the main focus of this iteration is to support the new functionalities of the pre-existing GFAB component and split it into various microservices.

Step 4

- Goal: Choose one or more design concepts that satisfy the selected drivers

Given the iteration goal selected drivers in Step 2, it is necessary to define which design concepts will be taken in account to realize the elements to refine selected in Step 3. The design concepts proposed are the following:

Design Decisions and Location	Rationale
Decompose the existing monolithic application by using business capability pattern	It is necessary to distinguish the GFAB business processes, and create a service to support each of its main functionalities
Decompose the existing monolithic application by using Sub-Domain pattern (Bounded Contexts)	For this approach we must split the GFAB domain into multiple smaller domains, each corresponding to a different business area.
Decompose the existing monolithic application with a supporting tool	For this approach we supply an external tool with the GFAB main concerns as input. The tool then outputs a possible decomposition into various services.
Apply microservices pattern to the output of monolith decomposition	It is obligatory to migrate the existing solution to microservices, so microservices pattern should be applied
Database Per Service	Keep each microservice's persistence data private to the service and accessible only via its API. The service's database is not accessible directly by other services.
API Composition	The application performs the join rather than the database. For example, a service (or the API gateway) could retrieve a user and their orders by first retrieving the user from the user service and then querying the order service to return the user most recent orders.
Aim to build a gateway apollo server API	It is necessary to create an apollo server that will bring together all of our microservices
Use relational database.	A relational database is a database based on the relational model of data.
Alternative	Rationale
Command Query Responsibility Segregation (CQRS)	By following this design pattern, we can separate data-update versus data-querying capabilities into separate models. Maintaining one or more materialized views that contain data from multiple services. The views are kept by services that subscribe to events that each services publishes when it updates its data.

Alternative	Rationale
Shared Database	In this solution services share a common database; a service publishes its data, and other services can consume it when required. The Shared Database option could be viable only if the integration complexity or related challenges of Database per Service-based services become too difficult to handle; also, operating a single Shared Database is simpler.
Database View	With a view, a service can be presented with a schema that is a limited projection from an underlying schema- we limit the data that is visible to the service. It gives us control over what is shared, and what is hidden.
Change Data Capture	With change data capture, rather than trying to intercept and act on calls made into the monolith, we react to changes made in a datastore. For change data capture to work, the underlying capture system has to be coupled to the monolith's datastore.
Spring Gateway, Apollo Gateway...	Since we use GraphQL in all our communications we felt that apollo server would fit great as our gateway and one element of our Team already worked with it and can pass knowledge.
Keep mongoDB(non relational database)	We chose to use non relational database this time around because the domain objects are already fixed and will not change and to our docker environments setup it was easier to do with SQL relational databases.

Step 5

- Goal: Instantiate architectural elements, allocate responsibilities and define interfaces

To satisfy the structure of the chosen design concepts, the following elements are proposed to be created:

Design Decisions and Location	Rationale
Elaborate refined domain model	In order to comprehend the new domain concepts of the iteration, it is necessary to establish a refined domain model with these concepts
Map use cases by actors	To establish actors and their responsibilities it is necessary to map the use cases being addressed by their actors.
Map use cases to domain objects	Domain objects of use cases help in identifying the dependencies existent for each use case
Map system elements to logical components	As the monolith solution is being decomposed in new components it is necessary to map these as logical components in order to understand which interfaces are being produced and consumed by the components

Design Decisions and Location	Rationale
Map system elements to physical components	As the monolith solution is being decomposed in new components it is necessary to map these as physical components in order to understand which communication protocols are being used in component communication
Structure packages of new components	By doing so, developers will have a better insight of the architecture and responsibilities of each component

Step 6

- Goal: Sketch views and record design decisions

Business Capabilities Decomposition:

Considering each Use Case a business function it's possible to describe business capabilities. Those are organized in the following table according the supplied information:

Use Cases	Business Capability
UC1- Create Sandwiches / UC2 -List Sandwiches	Sandwich
UC7-Register Customer / UC8 - List Customer / UC11-Login	User
UC3 - Register Shop / UC4 - List Shop / UC12-Define N minimum/maximum of deliveries	Shop
UC5-Register Order / UC6- List Order	Order
UC13-List deliveries / UC14 - Register Delivery	Delivery
UC9-Create promotion / UC10-List promotion	Promotion

Justifications:

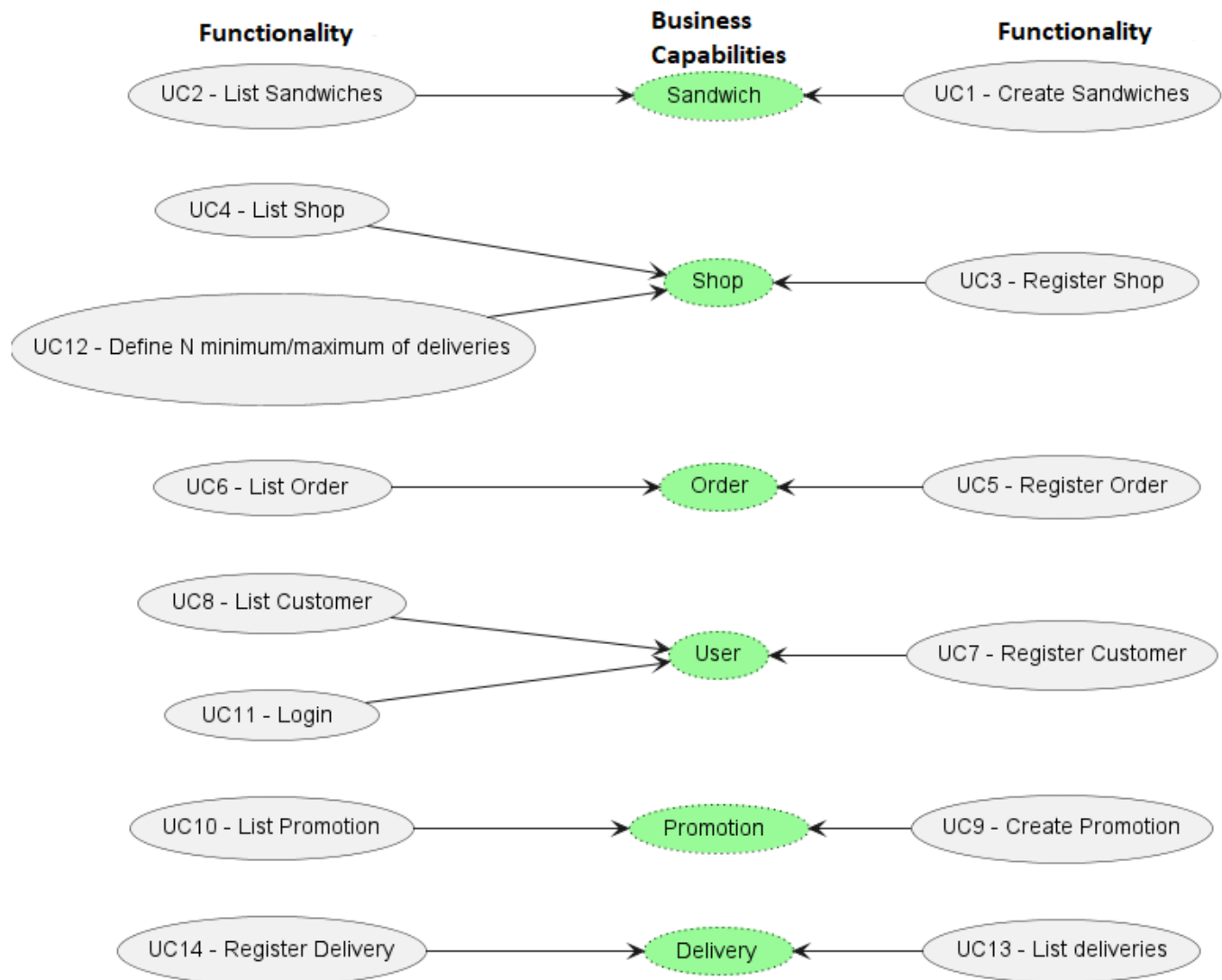
It's possible to decompose in at least 6 services.

Some use cases need the interaction of more than one service thought. For example if we want to register an Order(UC-5) we will need to access Sandwiches or Shops that exist on our system.

It was thought a sixth service, with the addition of the Delivery. Previously our domain had 5 aggregates but with more requirements from our client we feel the need to create a Delivery Aggregate that will going to be represented as a new microservice and Business Capability.

Other Use Cases are only contained in one service only and comply with Single Responsibility and Common Closure Principle (CCP).

Business Capabilities Decomposition Diagram



Sub-Domain Decomposition:

Taking into account the monolith's domain, we were able to split it into 6 separate sub-domains, namely: **Sandwich, Promotion, Order, Shop, User** and **Delivery**.

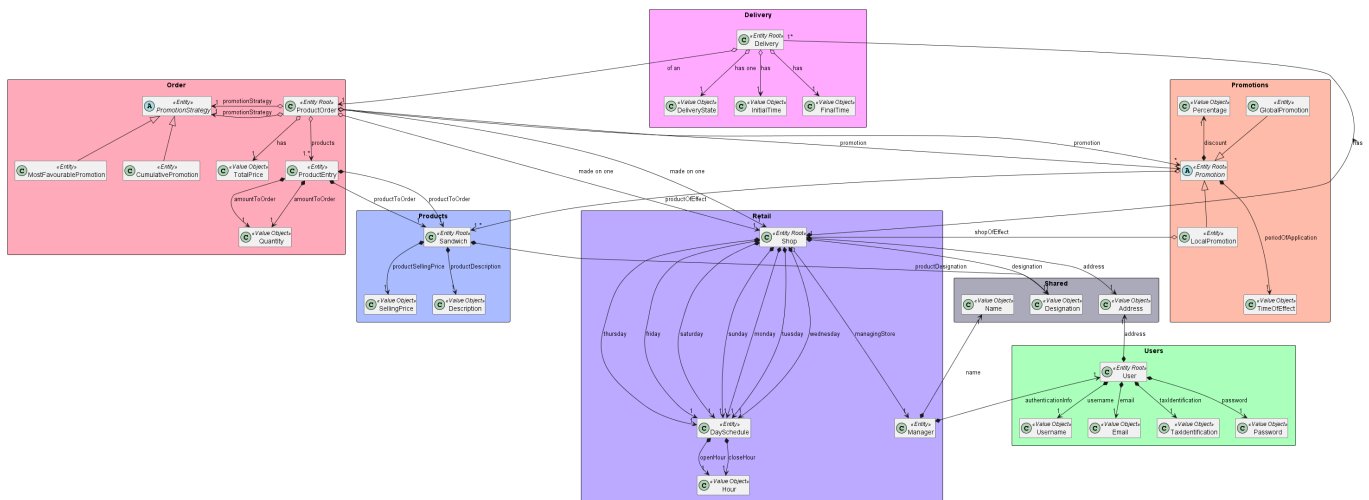
Most of the bounded contexts will have an Upstream/Downstream relationship since the downstream end of the relationship depends on data or behavior of the upstream end. Meaning that the upstream end will influence the downstream context.

The **Order** bounded context will be responsible for creating the various orders regarding user needs. For this feature, we need to know the sandwich/es that the customer wants to buy the promotions selected and the Shop where he want to by it.

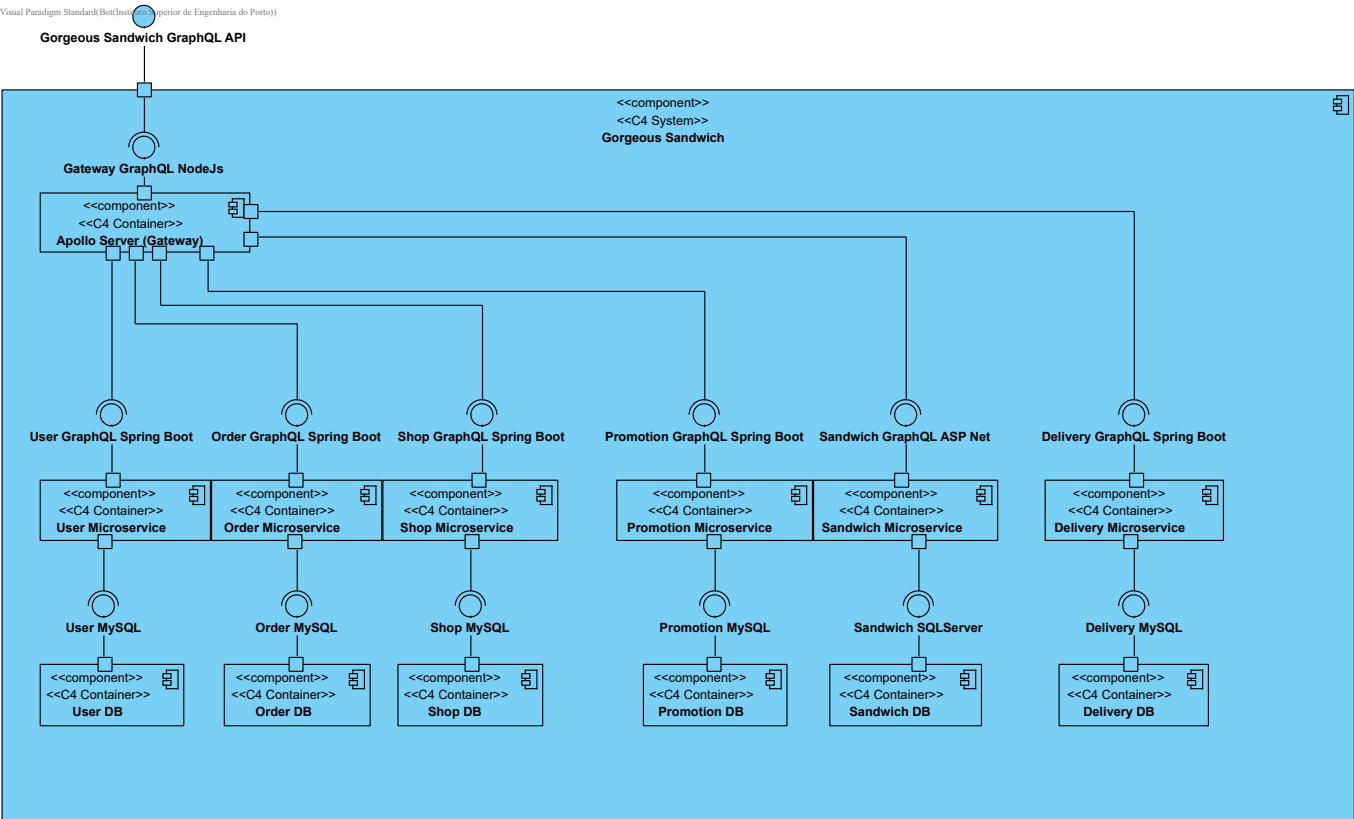
The **Promotions** bounded context will be responsible for making the order price cheaper for the User. For this feature, we need to know the sandwich/es where the promotion applies and the stores where the promotion is available.

The **Shop** bounded context will be the structure responsible for all the action. For this feature, we need to know the User that it's its Manager and the Deliveries it has to do.

Context Mapper Diagram

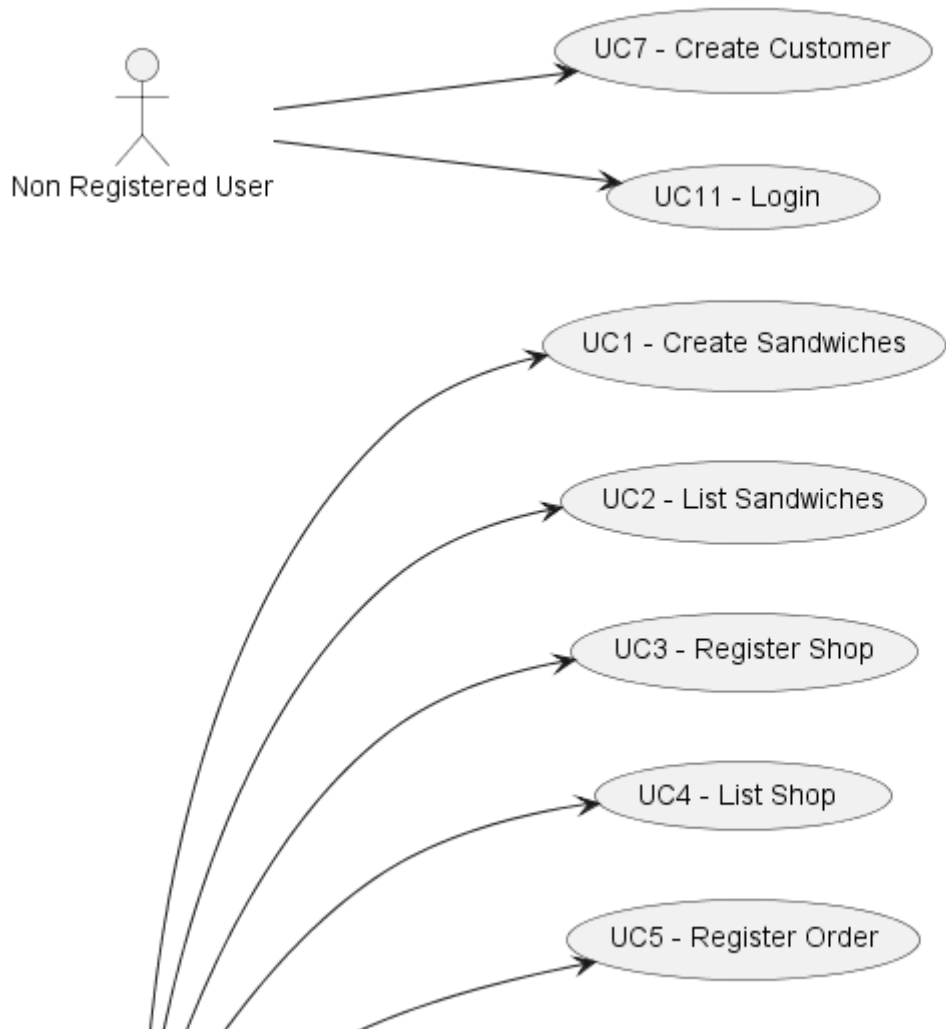


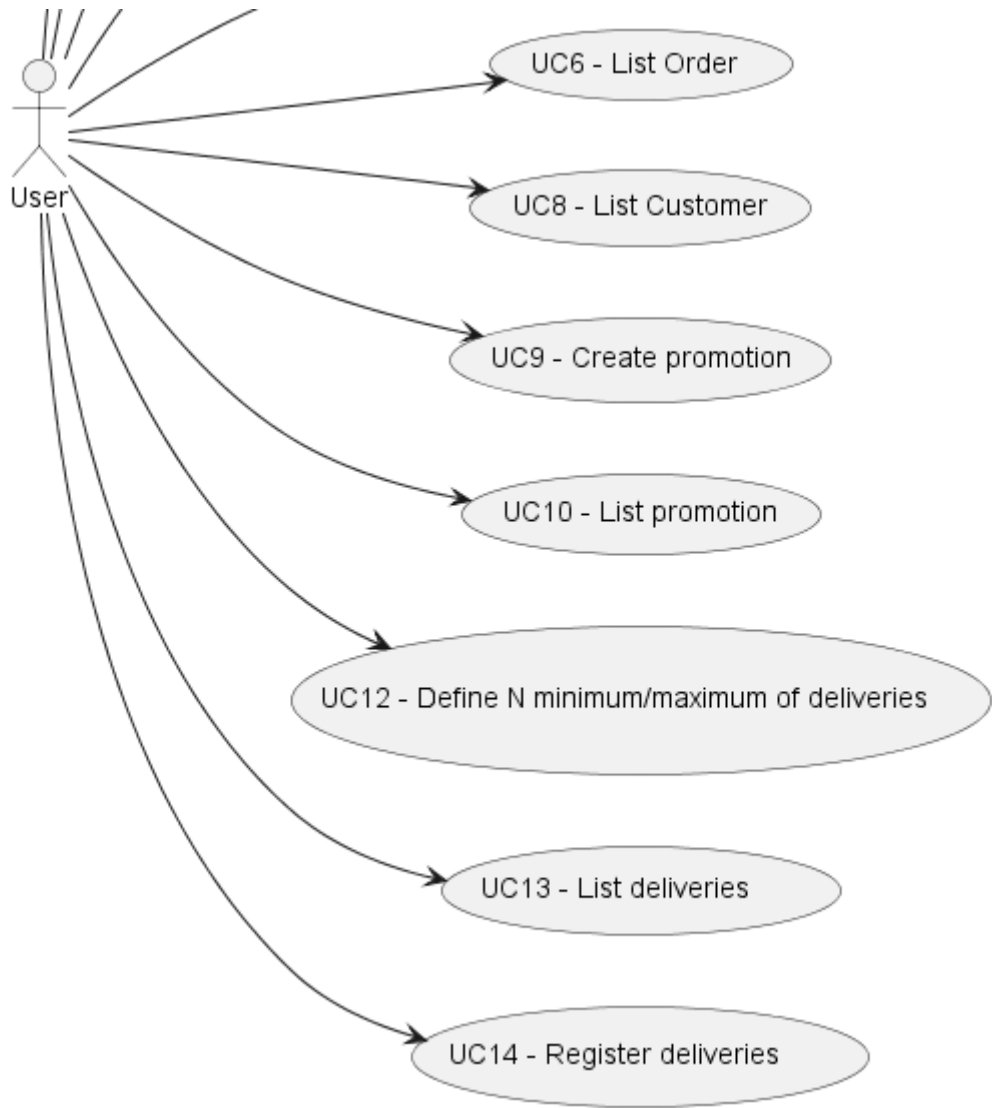
Component Diagram



Allocation View :

Use Case Diagram





Step 7

- Goal: Perform analysis of current design and review iteration goal and achievements of design purposes

As the first step for this iteration, the domain model was refined in order to meet the stakeholder's new expectations. Afterwards, the GFAB monolith was decomposed by following the Bounded Context (Sub-Domain) approach, which allowed the team to obtain a better understanding of the GFAB's main business capabilities, and each of the responsibilities of the newly created services. The existing REST API will be fractured in microservices next, using GraphQL this time around.

The following table represents the update of the kanban board after the iteration:

Not Addressed	Partially Addressed	Addressed
	QA-1	
		QA-3
		QA-5
		CRN-2
		CRN-3

Not Addressed	Partially Addressed	Addressed
		CRN-4
		CON-1
		CON-2
		CON-3
		CON-4
		CON-5

Iteration 2

Step 1

- Goal: Review Inputs
- Possible Questions:

Question	Answer
Inputs available and correct?	As far as the feedback of the stakeholder, the defined architectural drivers are correct
All drivers available?	As far as what was retained from the domain problem and the stakeholder feedback, all drivers have been defined
Is it clearly established what is the purpose for the design activities?	Yes, it is to attack primary functionalities using DDD pattern.
Have primary functionality and quality attribute scenarios been prioritized (ideally by the most important project stakeholders)?	Yes, the primary functionalities with most importance initially are the register and login of an user.
Are initial architectural concerns defined?	Yes

Step 2

- Goal: Establish iteration goal by selecting drivers

Kanban Board

Not Addressed	Partially Addressed	Addressed
	QA-1	
QA-2		
QA-4		

Not Addressed	Partially Addressed	Addressed
QA-6		
QA-7		
UC1		
UC2		
UC3		
UC4		
UC5		
UC6		
UC7		
UC8		
UC9		
UC10		
UC11		
UC12		
UC13		
UC14		
CRN-1		
CON-6		
CON-7		
CON-8		

Step 3

- Goal: Choose elements of the system to refine

The goal of this iteration is to apply the design defined on iteration 1, finally doing the microservices and gateway API. We will also change our REST API structure and use GraphQL so the UC's made in GFAB will be refactored. In order to realize this design it is necessary to refine the following elements:

- GFAB (Gorgeous Food Application Back-end), which is the monolithic backend of GFA, breaking it down into microservices. In the end, this component will no longer exist.
- UserAPI, which is the microservice responsible for the User.
- ShopAPI, which is the microservice responsible for the Shop.
- PromotionAPI, which is the microservice responsible for the Promotion.
- OrderAPI, which is the microservice responsible for the Order.
- DeliveryAPI, which is the microservice responsible for the Delivery.

- SandwichAPI, which is the microservice responsible for the Sandwich.
- ApolloGatewayAPI, which is the component that will connect all the microservices as our gateway.

Step 4

- Goal: Choose one or more design concepts that satisfy the selected drivers

Given the iteration goal selected drivers in Step 2, it is necessary to define which design concepts will be taken in account to realize the elements to refine selected in Step 3. The design concepts proposed are the following:

Design Decisions and Location	Rationale
Use 3 tier deployment pattern to physically architecture GFA	The 3 tier deployment pattern fits the needs of the requirements of GFA, in which there is a tier for our ApolloGatewayAPI, a tier for the respective microservice and a tier for the database
Use .NET for the SandwichAPI microservice	The stakeholders suggested that we should use more than one programming language for different microservices. For this reason we chose the Sandwich entity for this since it doesn't depend on any other service.
Deploy all our components and their databases into Docker	Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. In this case we will deploy all our components into docker.
Use Sonargraph Explorer.	The core idea is to have a better understanding of our application using a powerful static analysis tool with a focus on metrics and dependency visualization.
Use Voyager, Sandbox, Banana Cake Pop and GraphQL to have instructions clear to others.	Those tools allow you to describe the structure of your APIs so that machines can read them. It makes that information available for the client using it and easier to understand.
Unit testing and integration testing using Postman.	The more coverage an application has the more reliable it is.
Alternative	Rationale
Node.js for building SandwichAPI	The element of our time who got this microservice has a better understanding of C#.
Deploy the application on Azure, AWS or ISEP servers	Since we are using microservices, deploying them all into those external resources would cost money.
Use Swagger	Swagger is only available if we used REST but all our components use GraphQL

Alternative	Rationale
Sonarqube	The process of integrating Sonarqube to the application is more complex than Sonargraph Explorer. Sonargraph Explorer was also suggested by the stakeholders.

Step 5

- Goal: Instantiate architectural elements, allocate responsibilities and define interfaces

To satisfy the structure of the chosen design concepts, the following elements are proposed to be created:

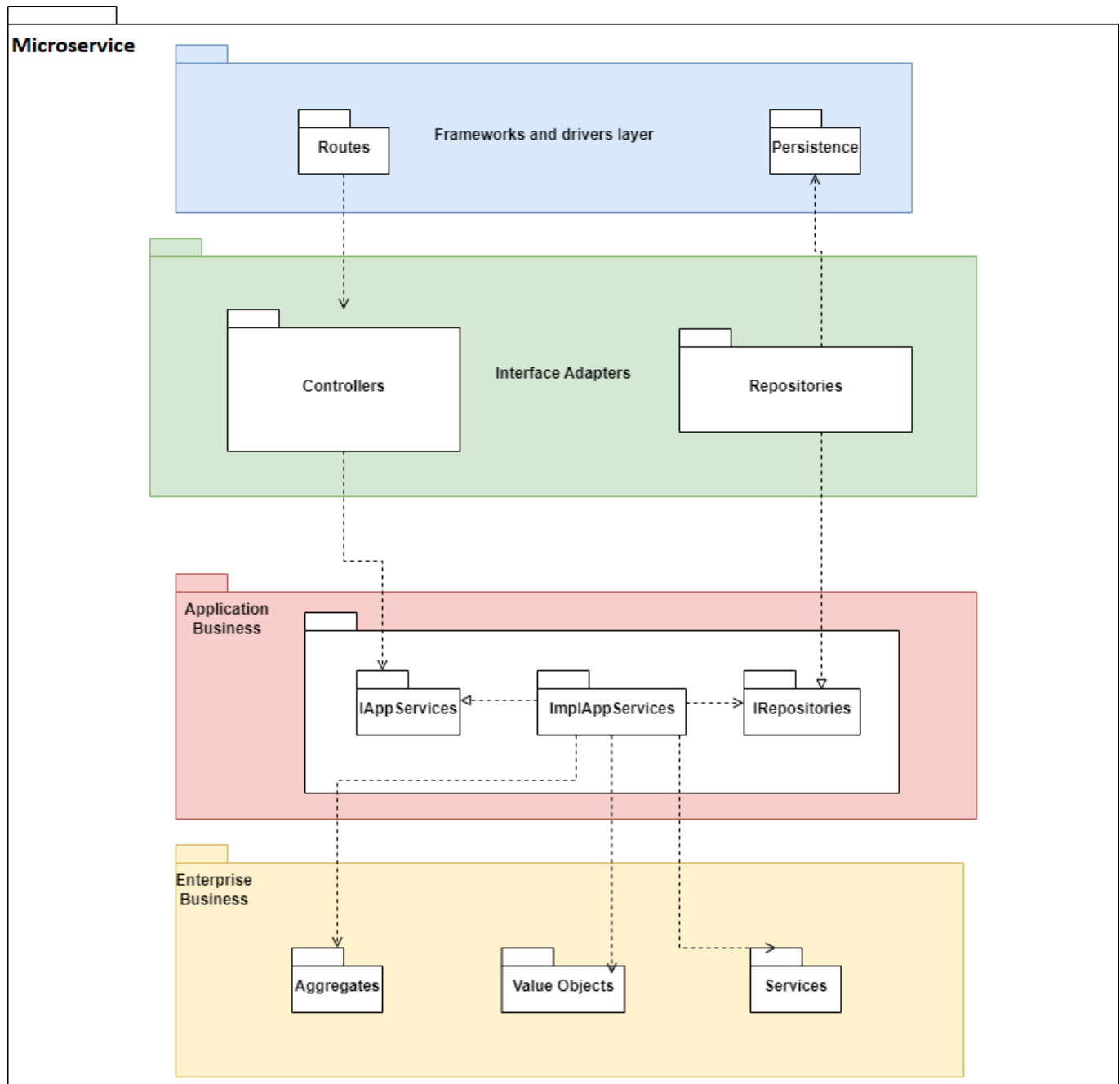
Design Decisions and Location	Rationale
Decompose GFAB in microservices	Our REST API application needs to be fractured into different components.
Change all the microservices into using GraphQL	It was requested from the stakeholders GraphQL usage.
Elaborate ApolloGatewayAPI	To explicit the specification of the produced functionalities so consumers can understand how requests and responses are performed and structured and bind all microservices together .
Deploy all the components into Docker.	Process of deploying our application and its documentation.

Step 6

- Goal: Sketch views and record design decisions

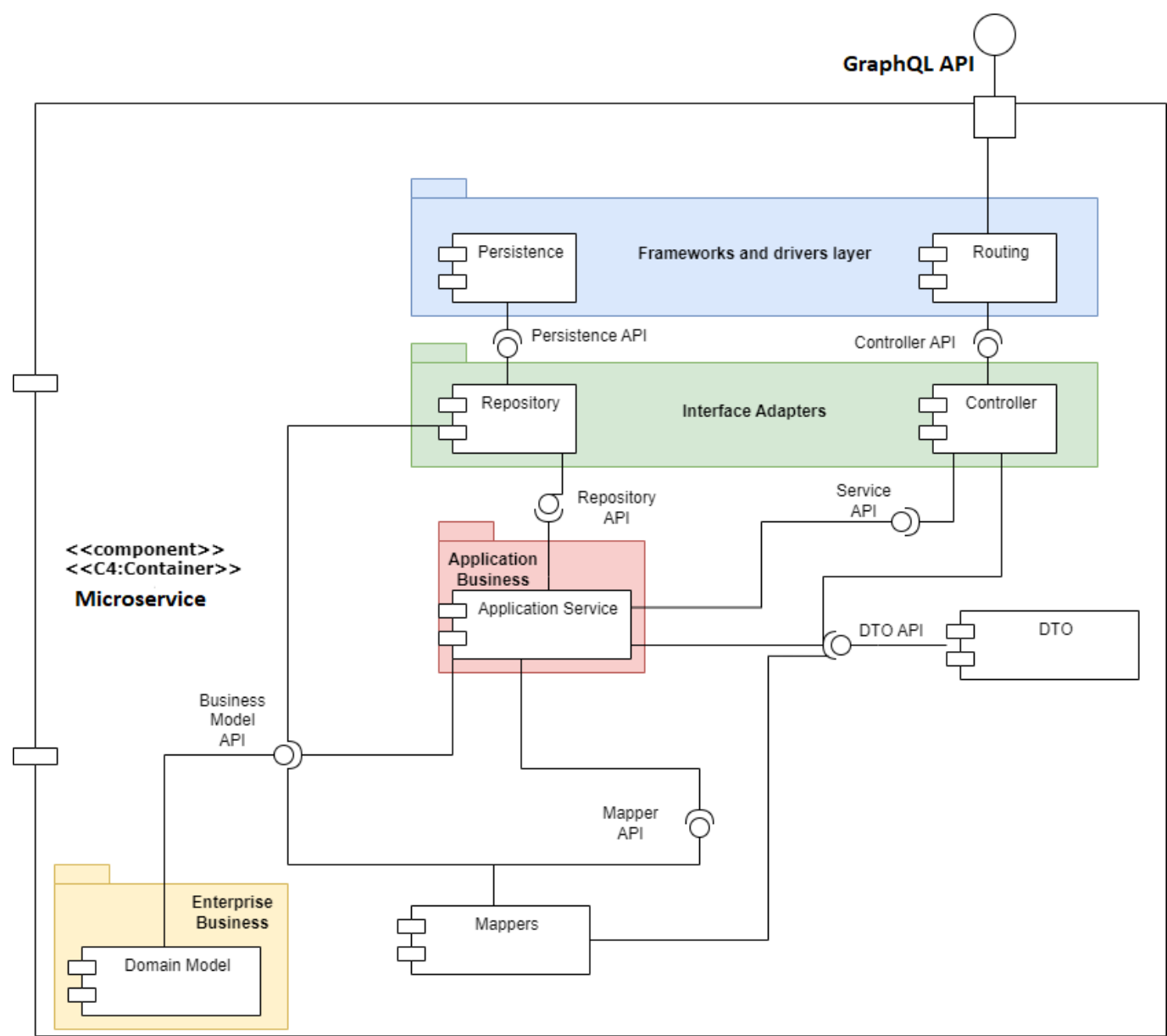
Module View :

Packages Diagram of Microservices

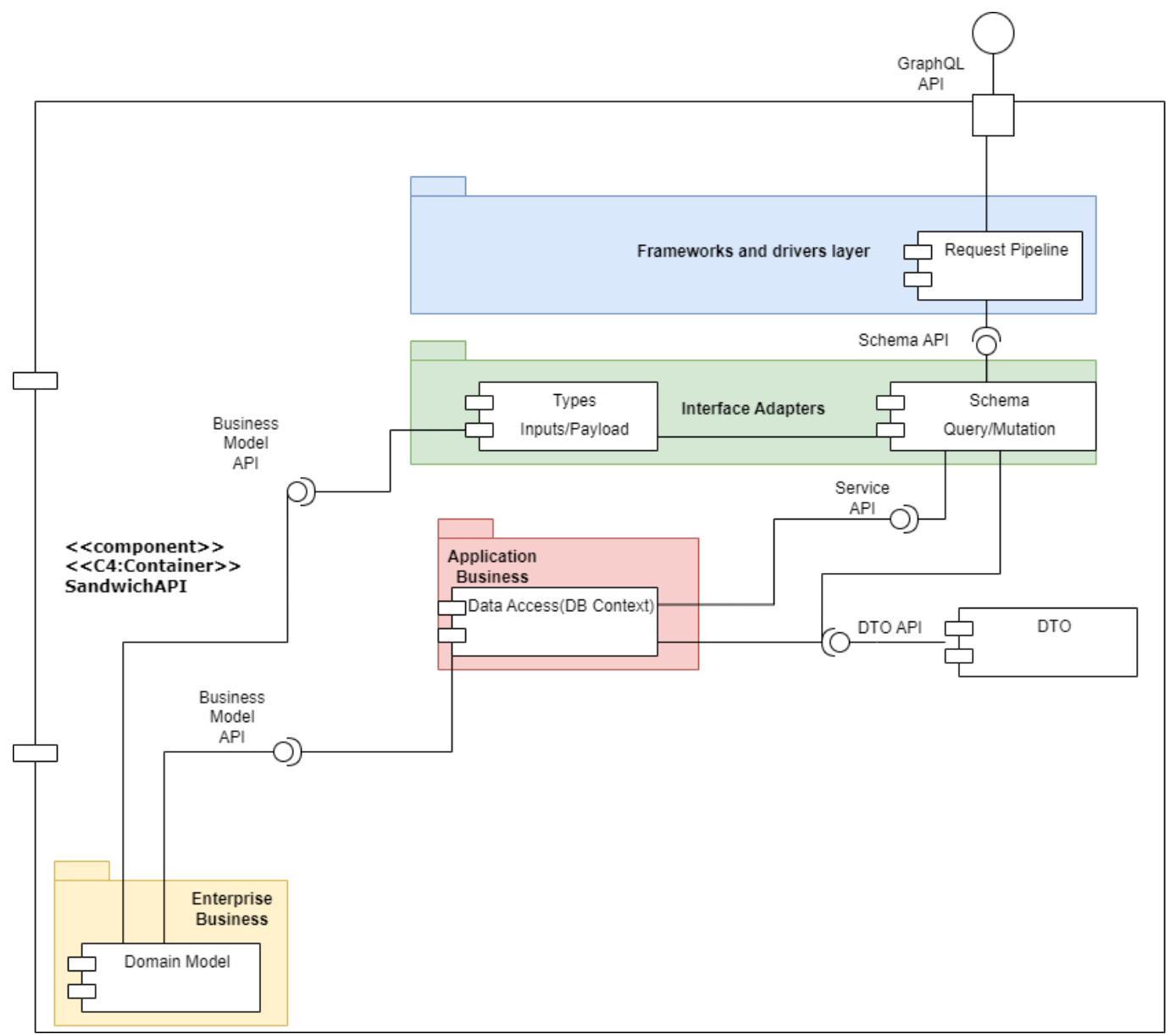


Component & Connector View :

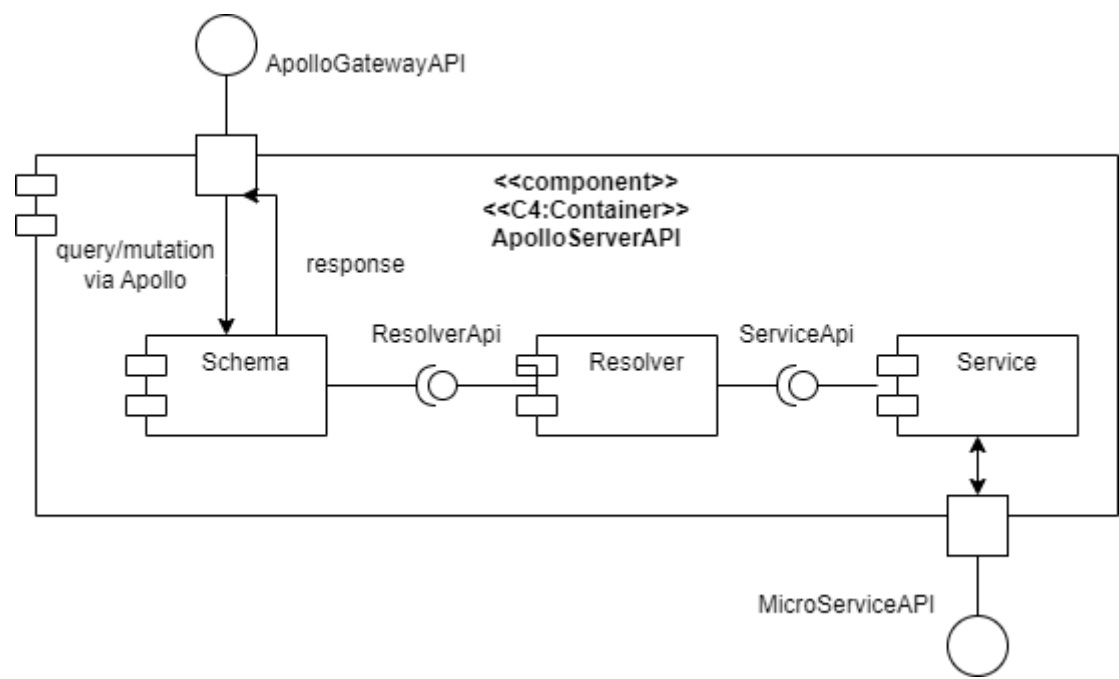
Components Diagram Fine-Grain View of all microservices except Sandwich



Components Diagram Fine-Grain View of Sandwich microservice .NET

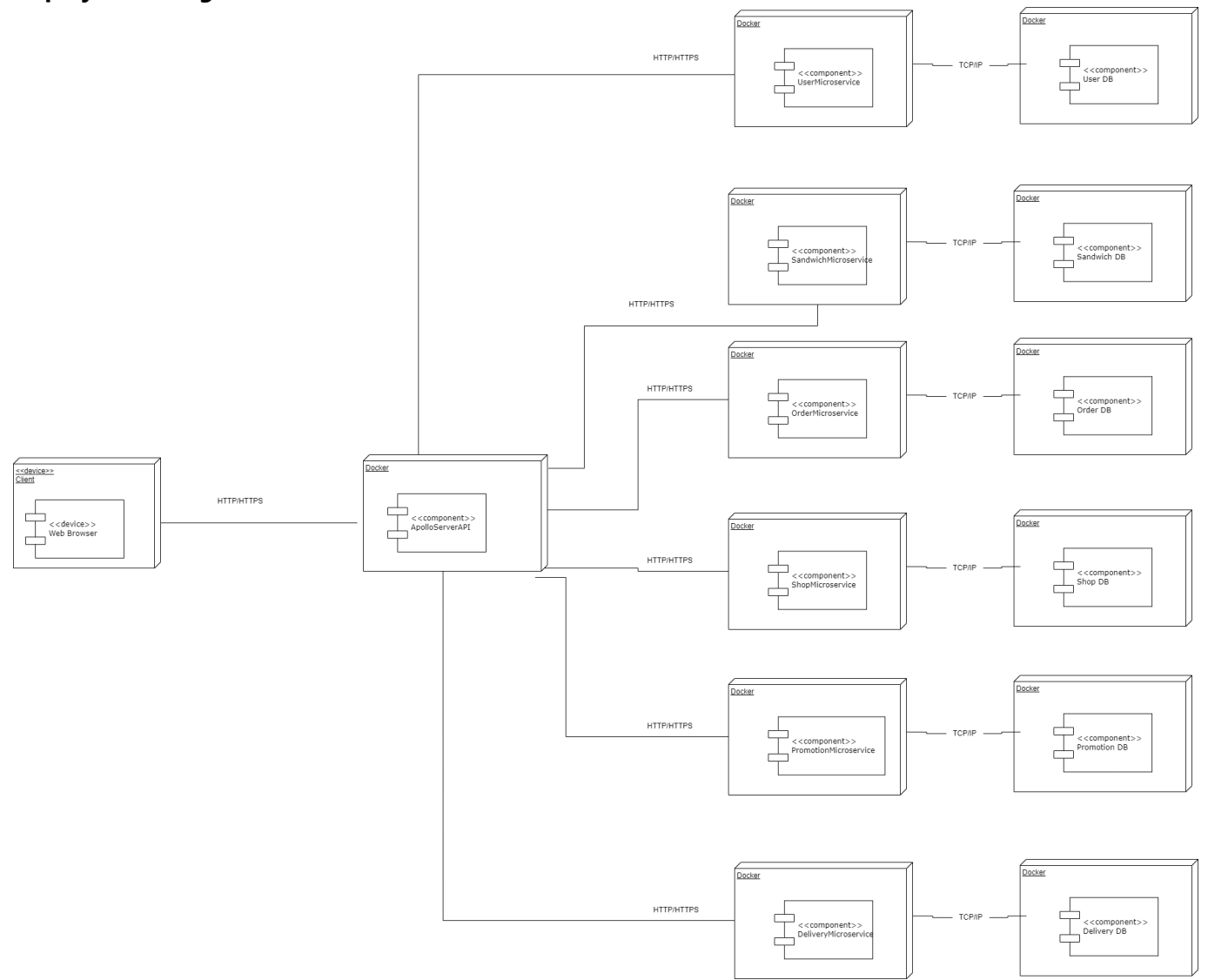


Components Diagram Fine-Grain View of ApolloServerAPI gateway



Allocation View :

Deployment Diagram



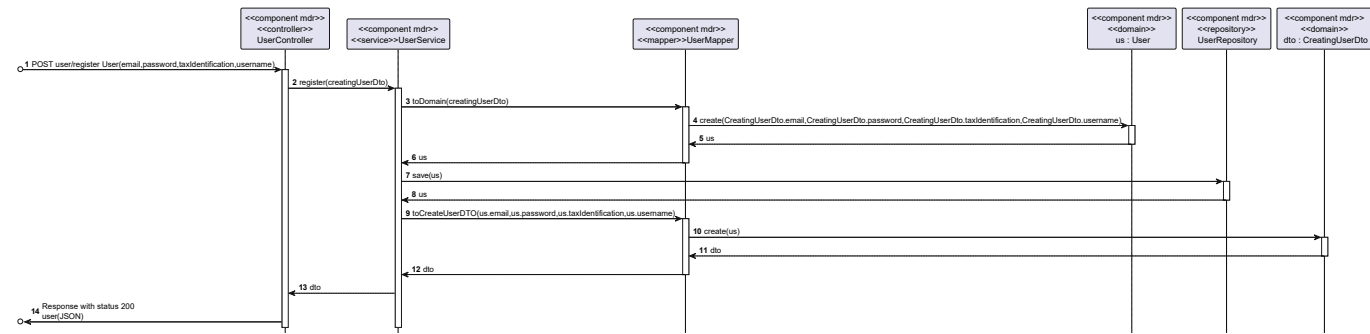
- Responsability Table for Defined Elements

Element	Responsibility
GFU (Gorgeous Food User)	Service responsible for providing the functionalities related to User aggregate root
GFS (Gorgeous Food Sandwich)	Service responsible for providing the functionalities related to Sandwiche aggregate root
GFO (Gorgeous Food Order)	Service responsible for providing the functionalities related to Order aggregate root

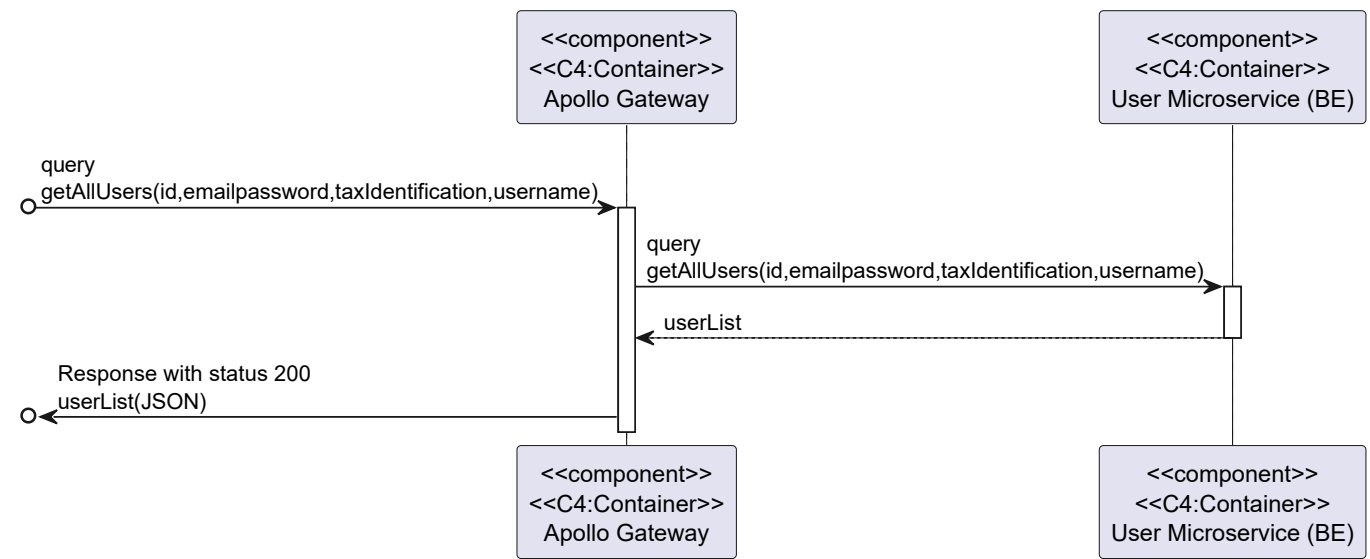
Element	Responsibility
GFP (Gorgeous Food Promotion)	Service responsible for providing the functionalities related to Promotion aggregate root
GFSH (Gorgeous Food Shop)	Service responsible for providing the functionalities related to Shop aggregate root
GFD (Gorgeous Food Delivery)	Service responsible for providing the functionalities related to Delivery aggregate root
GFG (Gorgeous Food Gateway)	Service responsible for binding all the microservices above, our gateway.
GFUD (Gorgeous Food User Database)	Database responsible for storing the information managed by the GFU Service
GFSD (Gorgeous Food Sandwich Database)	Database responsible for storing the information managed by the GFS Service
GFOD (Gorgeous Food Order Database)	Database responsible for storing the information managed by the GFO Service
GFPD (Gorgeous Food Promotion Database)	Database responsible for storing the information managed by the GFP Service
GFSHD (Gorgeous Food Shop Database)	Database responsible for storing the information managed by the GFSH Service

Element	Responsibility
GFDD (Gorgeous Food Delivery Database)	Database responsible for storing the information managed by the GFD Service
API Gateway	This layer enables communication with remote services. It is responsible for managing requests and responses to and from objects of the domain
Enterprise Business	Also known as Entity Layer , represents the business rule.
Application Business	Also known as Use Case Layer represent the rules related to the automatization of our system.
Interface Adapters	Our business should deal only with the most convenient data format for it, and so should our external agents, as DBs or UIs. But, this format usually is different. For this reason, the interface adapter layer is responsible for converting the data.
Framework and Drivers Layers)	In truth, we don't code here. That is because this layer represents the lowest level of connection to external agents. For example, the driver to connect to the database or the web framework. In this case, we're going to use spring-boot as the web and dependency injection framework.
Sandwich (Model)	Produces models and functionalities related to Sandwich aggregate root
Shop (Model)	Produces models and functionalities related to Shop aggregate root
Order (Model)	Produces models and functionalities related to Order aggregate root
User (Model)	Produces models and functionalities related to Users aggregate root
Promotion (Model)	Produces models and functionalities related to Promotion aggregate root
Delivery (Model)	Produces models and functionalities related to Delivery aggregate root

Generic CRUD behaviour within the UserAPI component that demonstrates Onion Architecture



Generic CRUD behaviour within our app that demonstrates the communication between gateway and microservices



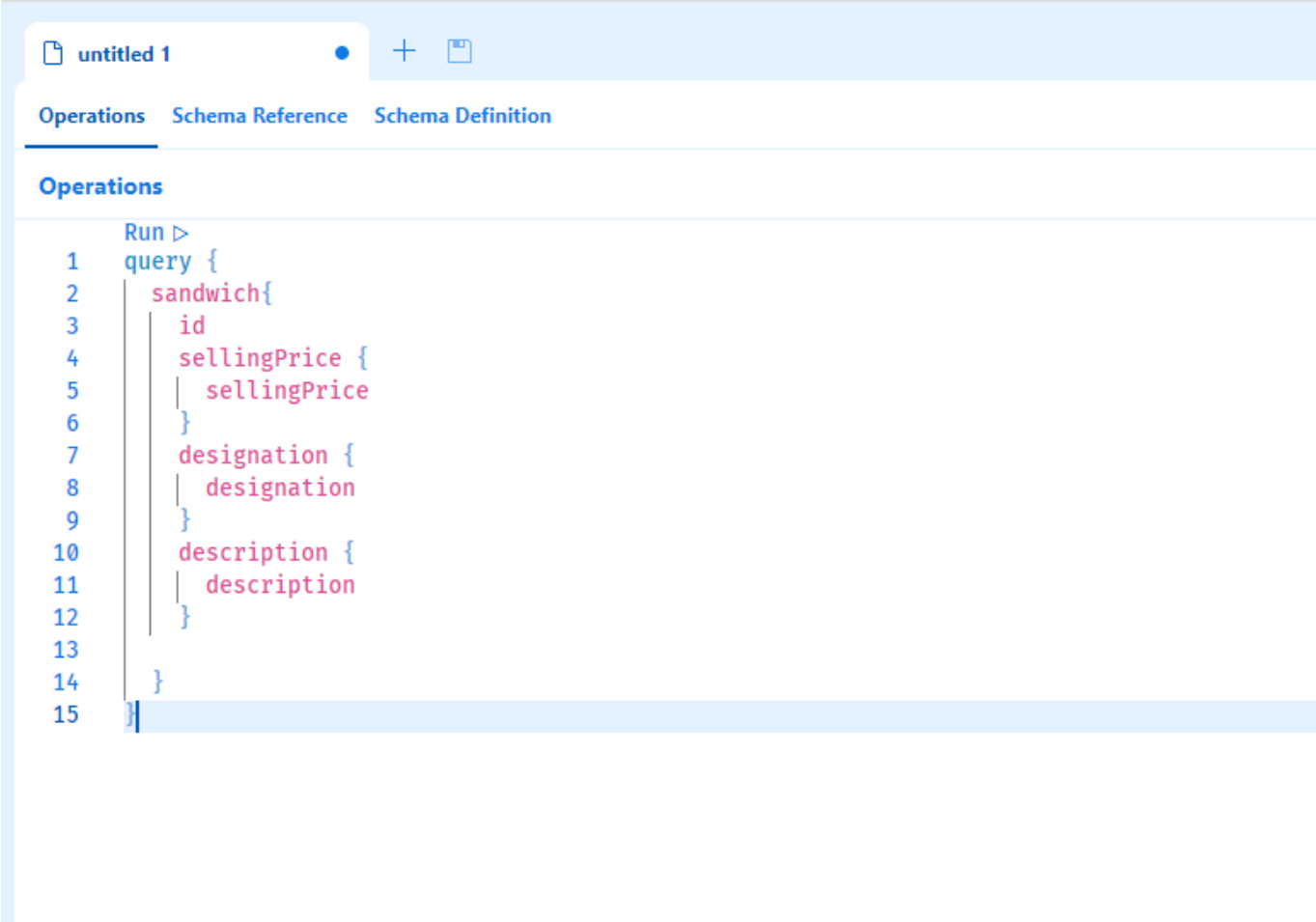
Simplify API

Banana Cake Pop

Banana Cake Pop is a GraphQL tool that makes it easy and enjoyable to test your GraphQL server implementations. It works well with Hot Chocolate and any other GraphQL server. It lets us see our query, mutation and objects.

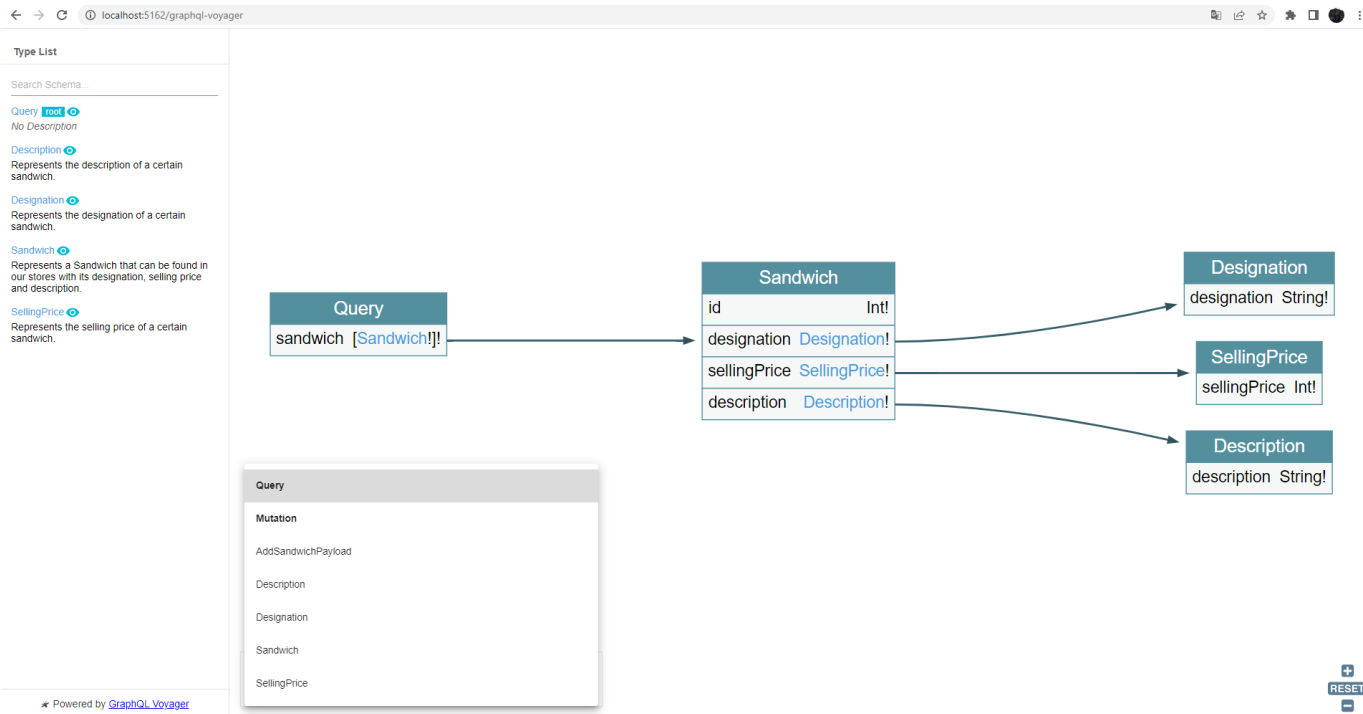


And we can also do requests



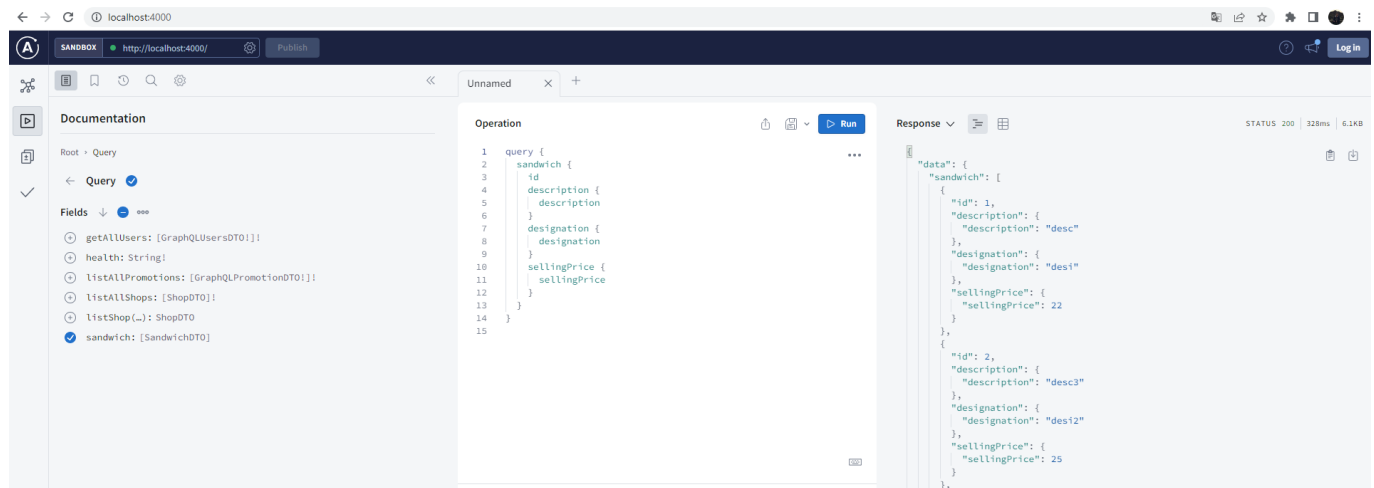
Voyager

Voyager is a GraphQL tool that gives us the general idea of our domain, shows as its objects, queries, mutations or subscriptions. When we have a large domain on our application it gives us a lot of help.



Sandbox

Sandbox is pretty similar to Banana Cake Pop and it's embedded in our gateway.



Tests

Unit tests to all the new components were added(SandwichAPI and DeliveryAPI) and Integration tests were created. Our unit tests are made to verify that the our Domain Objects will be created without errors,they verify Business Rules. It doesn't make sense to create a sandwich with a number below 0 for example.

Our integration tests with the use of Postman let us know if the communication between components it's working and if the results given to us are correct.


Unit Tests Sandwich

```
Starting test execution, please wait...
A total of 1 test files matched the specified pattern.

Passed! - Failed:    0, Passed:   11, Skipped:    0, Total:   11, Duration: 23 ms - SandwichGQL.dll (net6.0)
```

Integration Tests Sandwich(requests to the gateway)

ARQSOF-SANDWICH - Run results

 Run on Today, 20:42:04 · [View all runs](#)

Source	Environment	Iterations	Duration	All tests	Avg. Resp. Time
Runner	none	1	789ms	3	20 ms

All Tests

Passed (3)

Failed (0)

Skipped (0)

Iteration 1

POST

Create Sandwich

http://localhost:4000/

/ Create Sandwich / Create Sandwich

Pass

Status code is 200

POST

Get All Sandwiches

http://localhost:4000/

/ GetAllSandwiches / Get All Sandwiches

Pass

Status code is 200

Pass

There are sandwiches

Step 7

- Goal: Perform analysis of current design and review iteration goal and achievements of design purposes

In this iteration we successfully transformed our monolithic GFAB into a microservice-based application with a gateway using GraphQL, test it and deploy it to Docker.

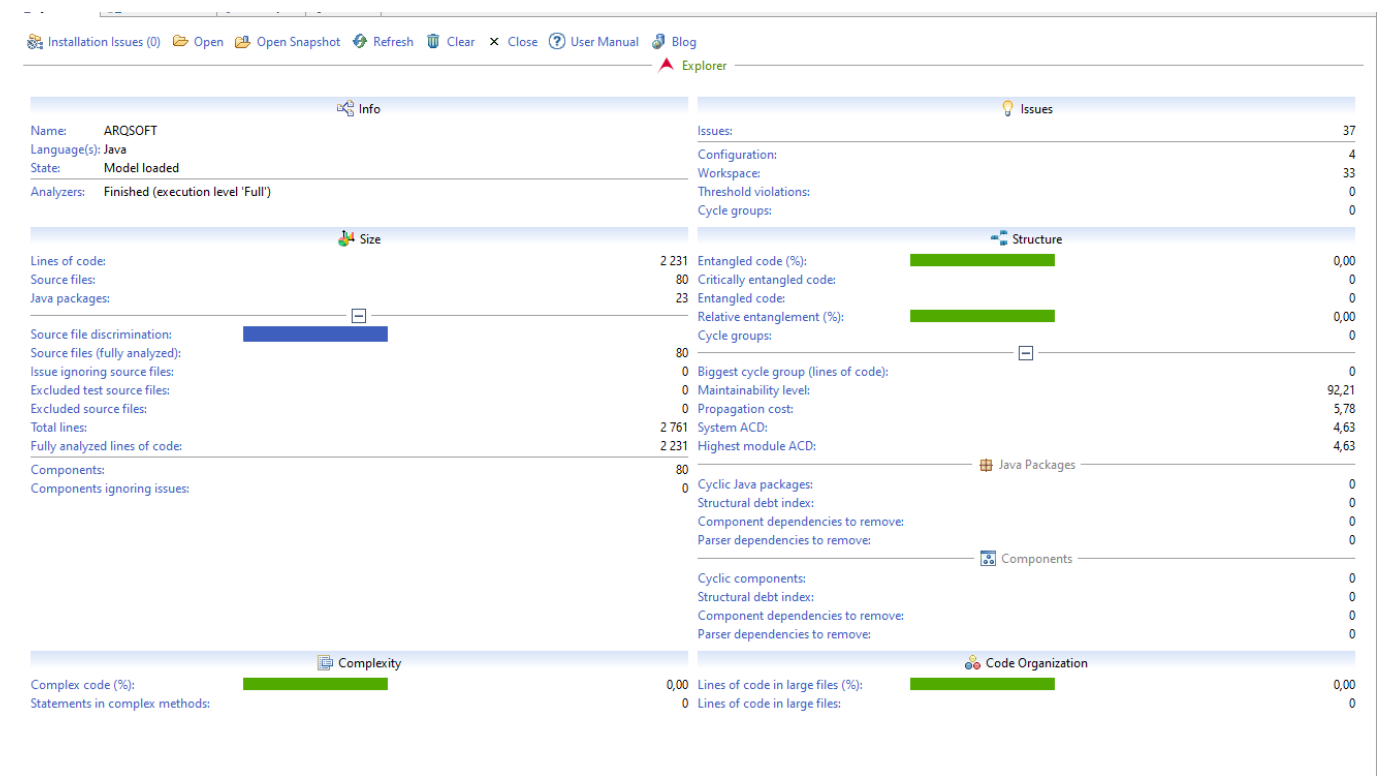
The following table represents the update of the kanban board after the iteration:

Not Addressed	Partially Addressed	Addressed
		QA-1
		QA-2
		QA-4(other API's for GraphQL)
		QA-6
		QA-7
		UC1
		UC2
		UC3
		UC4
		UC5
		UC6

Not Addressed	Partially Addressed	Addressed
		UC7
		UC8
		UC9
		UC10
		UC11
UC12		
UC13		
UC14		
		CRN-1
		CON-6
		CON-7
		CON-8

Metrics and Sonagraph Explorer

Overview



Maintainability

The maintainability of this project is **92.21%**, which is almost perfect. Such metrics, when compared with the ones from the last projects, demonstrate why the microservices architecture allows for a better maintainability compared with a monolith strategy.

Others

There are no cyclic groups in the project, as well as no structural debt.

Metric	Value
Components	80
Total lines	2761
Lines of code	2231
Entagled code	0%
System ACD	4,63
Propagation cost	5,78
Number of types	84
Average complexity	1,51
NCCD	0,85

Report HTML

[Report HTML](#)

Deployment

Docker

The deployment methodology used for deployment was Docker, taking advantage of the docker compose strategy.

Image Build

To run the docker compose is necessary to build the image of each microservice and the gateway

```
docker build ./shop -t gorgeous-sandwich.shop:0.0.1-SNAPSHOT &
docker build ./promotion -t gorgeous-sandwich.promotion:0.0.1-SNAPSHOT &
docker build ./user -t gorgeous-sandwich.user:0.0.1-SNAPSHOT &
docker build ./SandwichGQL -t gorgeous-sandwich.sandwich:0.0.1-SNAPSHOT &
docker build ./order -t gorgeous-sandwich.order:0.0.1-SNAPSHOT &
docker build ./apollo-server -t gorgeous-sandwich.gateway:0.0.1-SNAPSHOT &
```


The name chosen for the tags was "gorgeous-sandwich." plus the name of the microservice. The objective of said name convention is to visually associate the microservices as a "family" of services.

Compose File

```
version: "2"
services:
  shopmicroservice:
    container_name: shop_ms
    image: gorgeous-sandwich.shop:0.0.1-SNAPSHOT
    restart: on-failure
    ports:
      - "5000:8080"
    depends_on:
      - shopdb
    environment:
      - spring.datasource.url=jdbc:mysql://shopdb:3306/shops?
        autoreconnect=true&allowPublicKeyRetrieval=true&useSSL=false
      - spring.datasource.username=admin
      - spring.datasource.password=@Dmin123
      -
    spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
      - logging.level.org.springframework.web=INFO

  shopdb:
    image: mysql
    container_name: shopdb
    environment:
      - MYSQL_USER=admin
      - MYSQL_ALLOW_EMPTY_PASSWORD=true
      - MYSQL_PASSWORD=@Dmin123
      - MYSQL_DATABASE=shops
    command: mysqld --lower_case_table_names=1 --skip-ssl --
      character_set_server=utf8
    restart: on-failure
    ports:
      - "5001:3360"

  promotionmicroservice:
    container_name: promotion_ms
    image: gorgeous-sandwich.promotion:0.0.1-SNAPSHOT
    depends_on:
      - promotiondb
    environment:
      - SPRING.DATASOURCE.URL=jdbc:mysql://promotiondb:3306/promotions?
        autoreconnect=true&allowPublicKeyRetrieval=true&useSSL=false
      - SPRING.DATASOURCE.USERNAME=admin
      - SPRING.DATASOURCE.PASSWORD=@Dmin123
      -
    SPRING.JPA.PROPERTIES.HIBERNATE.DIALECT=org.hibernate.dialect.MySQL8Dialect
```

```

    ports:
      - "5003:8080"
    restart: on-failure

promotiondb:
  image: mysql
  container_name: promotiondb
  environment:
    - MYSQL_USER=admin
    - MYSQL_ALLOW_EMPTY_PASSWORD=true
    - MYSQL_PASSWORD=@Dmin123
    - MYSQL_DATABASE=promotions
  command: mysql -l --lower_case_table_names=1 --skip-ssl --
character_set_server=utf8
  restart: on-failure
  ports:
    - "5004:3360"

usermicroservice:
  container_name: user_ms
  image: gorgeous-sandwich.user:0.0.1-SNAPSHOT
  environment:
    - SPRING.DATASOURCE.URL=jdbc:mysql://userdb:3306/users?
autoreconnect=true&allowPublicKeyRetrieval=true&useSSL=false
    - SPRING.DATASOURCE.USERNAME=admin
    - SPRING.DATASOURCE.PASSWORD=@Dmin123
    -
  SPRING.JPA.PROPERTIES.HIBERNATE.DIALECT=org.hibernate.dialect.MySQL8Dialect
  ports:
    - "5005:3360"
  depends_on:
    - userdb
  restart: on-failure
userdb:
  image: mysql
  container_name: userdb
  environment:
    - MYSQL_USER=admin
    - MYSQL_ALLOW_EMPTY_PASSWORD=true
    - MYSQL_PASSWORD=@Dmin123
    - MYSQL_DATABASE=users
  command: mysql -l --lower_case_table_names=1 --skip-ssl --
character_set_server=utf8
  restart: on-failure
  ports:
    - "5006:3360"

sandwichmicroservice:
  image: gorgeous-sandwich.sandwich:0.0.1-SNAPSHOT
  container_name: sandwich_ms
  depends_on:
    - sandwichdb
  ports:
    - "5007:80"

```

```
sandwichdb:
  image: "mcr.microsoft.com/mssql/server:2022-latest"
  container_name: sandwichdb
  environment:
    ACCEPT_EULA: "Y"
    SA_PASSWORD: "pa55w0rd!"
    MSSQL_PID: "Express"
  ports:
    - "5008:1433"

ordermicroservice:
  image: gorgeous-sandwich.order:0.0.1-SNAPSHOT
  container_name: order_ms
  depends_on:
    - orderdb
  ports:
    - "5008:8080"
  restart: on-failure
  environment:
    - SPRING.DATASOURCE.URL=jdbc:mysql://orderdb:3306/orders?
autoreconnect=true&allowPublicKeyRetrieval=true&useSSL=false
    - SPRING.DATASOURCE.USERNAME=admin
    - SPRING.DATASOURCE.PASSWORD=@Dmin123
    -
  SPRING.JPA.PROPERTIES.HIBERNATE.DIALECT=org.hibernate.dialect.MySQL8Dialect

orderdb:
  container_name: orderdb
  image: mysql
  command: mysqld --lower_case_table_names=1 --skip-ssl --
character_set_server=utf8
  restart: on-failure
  ports:
    - "5009:3360"
  environment:
    - MYSQL_USER=admin
    - MYSQL_ALLOW_EMPTY_PASSWORD=true
    - MYSQL_PASSWORD=@Dmin123
    - MYSQL_DATABASE=orders

gateway:
  image: gorgeous-sandwich.gateway:0.0.1-SNAPSHOT
  container_name: gateway
  depends_on:
    - sandwichmicroservice
    - promotionmicroservice
    - shopmicroservice
    - usermicroservice
  ports:
    - "5010:4000"
```

The strategy used in this file is to make use of the default network that is created when a docker compose file is created and group containers into a network. Furthermore, each microservice depends on its assigned database, which is always built first due to such dependency. The gateway is the last service to be built due to its dependencies on all the microservices.

To run this docker compose file the following command must be executed:

```
docker compose -f Dockercompose.yml run gateway
```

Architecture Tradeoff Analysis Method

Design

- The User of our application at the moment don't accept other types, which limits the experience of our application and makes significant changes to our application and domain model inevitable and also lowers our modifiability.(Risk)
- Our team has given priority to performance and features, leaving security to the side. At the moment we keep encrypted user passwords in our databases, offering some protection but that's all we have for now.(Risk)
- Wanting to implement GraphQL in all our communications, due to its difficulty, specially on the communication between microservices and gateways took us a lot of time and we couldn't finish the Delivery microservice.

Microservices

- Creation of unit tests only for the entity layer affecting our security and reliability.(Risk)
- Lack of CORS implementation that affects our security.(Risk)

Apollo Server Gateway

- We couldn't divide our server per modules, despite the code being well documented, all the code is in the same class(index.ts).(No risk)