E.T.S. de Ingeniería Industrial,
Informática y de Telecomunicación

Microservices and Observability: Implementing a product recommendation system.

Grado en Ingeniería Informática

Trabajo Fin de Grado

Jaime Higueras Medina

José Enrique Armendáriz Iñigo

Pamplona

upna
Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

Abstract

This project is motivated by the real experience of working with a large microservice system and living the transition from having a poor observability system to having one of the best new observability systems.

We will first cover the microservice architecture explaining the advantages and drawbacks of its architecture. Then the observability of microservices will be studied mentioning its advantages and the utility of having a good observability monitoring system to have a better understanding of the microservice architecture and have a better troubleshooting of it.

Lastly, we will implement a system recommendation of product with the purpose of creating a system based on the microservice architecture and having one of the newest and most efficient observability system you can find to this day.

Keywords

Microservice architecture, Observability systems and Recommendation systems.

Index

Figures Index

Codes Index

Introduction

In the world of software development, microservices have become a popular approach, offering multitude of advantages to deploy, develop and maintain large applications. In the Figure 1, we can see the unstoppable trend in the early years.

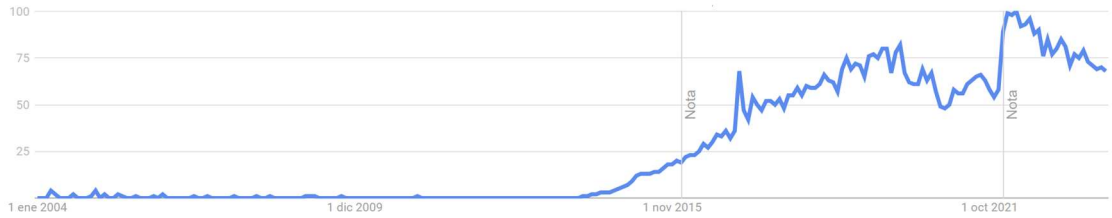

Figure 1: Google trends for the keyword microservices [1].

Microservices architecture [2], [3] is a distributed design approach that allows for the development of complex applications as a suite of small, independent services.

This architecture enhances scalability of the application by allowing independent scaling of microservices, leading to more efficient resources use. It also offers fault isolation, preventing that a single service failure dose not affecting the entire application.

The microservice architecture allows us to use multiples programming languages and data storage technologies, as microservices are loosely coupled and can be tailored to their specific needs.

It aligns with DevOps practices, especially with the continuous integration and continuous delivery. This helps with having lower-risk deployments and speeding up the feedback cycle, allowing continuous application improvement.

While microservices offer numerous benefits, they also come with their own set of challenges.

They can introduce increased complexity in system operations and deployment. Each microservice requires its own runtime environment and data storage, which can lead to higher resource consumption.

The distributed nature of microservices also means that developers must address issues such as network latency and load balancing. Additionally, the need for complex orchestration frameworks that can add to the operational overhead.

There is also the risk of failure cascades, where a problem in one service can affect others. Lastly, coordinating many independent services can be challenging, potentially leading to organizational complexity.

There are many problems when we face microservice systems; however, the main problem is understanding the overall behaviour of the entire system, especially when, with the pass of time the system, becomes gradually more complex.

This is where observability comes in handy. Observability [4] is about more than just tracking and monitoring, it is about getting a full picture of how a system works, performs, and stays reliable.

This project is all about exploring how microservices and observability work together. We want to understand how observability practices can make it easier to develop and maintain applications based on microservices.

Objectives

Throughout this project, we will break down the basics of microservices, looking at why they are used and the problems they can bring. We will also look closely at observability, figuring out how it helps developers and operators understand what is happening in a system.

Once we have the overall understanding of microservice architecture and the role of observability, we will implement a product recommendation system that will have a microservice architecture and all the necessary tools for having a good observability of the overall system.

Outline

Chapter 1: Introduction

A brief microservice architecture introduction mentioning the importance of the observability in its architecture. Definition of the project's objectives.

Chapter 2: Microservice architecture

This chapter starts with a brief introduction to the monolith architecture, the predecessor of microservice architecture, which we cover broadly in this chapter. It highlights microservices' advantages and mentions the drawbacks of its architecture. Moreover, the chapter explains the two main ways that microservices can communicate with themselves and how they expose their services through an API interface. In the last part of the chapter, we cover two advanced patterns in the microservice architecture.

Chapter 3: Observability

With a good observability system, we can minimize many of the problems that microservices carry. This chapter delves into the observability domain, explaining the three observability pillars: metrics, logs and traces, in order to meet a good monitory system.

Chapter 4: Recommendation system

An introduction to recommendation systems, focusing on a content-based approach using a graph database. This chapter explains the recommendation system that is implemented in the final microservice application.

Chapter 5: Practical implementation

The details of the implementation of the product recommendation system in a microservice architecture. It defines the functionalities, API contracts, data models, message queues and the frontend application.

Chapter 6: Observability system

The details of the implementation of the observability system using the innovative Grafana stack (Prometheus, Loki, Tempo, Grafana).

Chapter 7: Testing and validation

This chapter presents test cases for testing and validating the recommendation system application. The goal of this chapter is, on the one hand, to assess the most important functionality of the application and the observability system. On the other hand, showcasing the main functionality of the recommendation system application.

Chapter 2: Microservice Architecture

You should be thinking of migrating to a microservice architecture to achieve something that you cannot currently achieve with your existing system architecture.

Sam Newman – Author of "Building Microservices"

In recent years, the shift from traditional monolithic architecture to microservice architecture has become one of the main strategies for companies to thrive in today's volatile, uncertain, complex and ambitious world [2]. In this chapter we will discuss what a monolithic architecture is, the definition of microservice architecture and its advantages and disadvantages, how microservices can communicate among themselves and provide the API definition. Moreover, we abord two complex microservices pattern at the end of the chapter.

2.1 Monolithic architecture

The monolithic architecture consists of contain the entirely application built as a single component. This approach is not per se bad, in fact, it is suitable for most real applications, which consist of smaller project, where the scalability issues are not expected and focused on rapid development and simplicity.

The main benefits of monolithic architecture include:

- **Rapid development process**: Initially, monolithic architecture is faster to develop due to its simplicity and the absence of distributed complexity.
- **Lack of external dependencies**: Monolithic architecture lacks external dependencies, making debugging processes and testing easier, due to all the code being in one place.
- **Easy deployment**: The deployment is simplified as there is only one deployment each time a feature is released, or a bug is fixed.

However, as the application grows and the business logic becomes more complex, monolithic architecture can slow down the development process, increase code coupling and tie the application with the technologies used in the monolith.

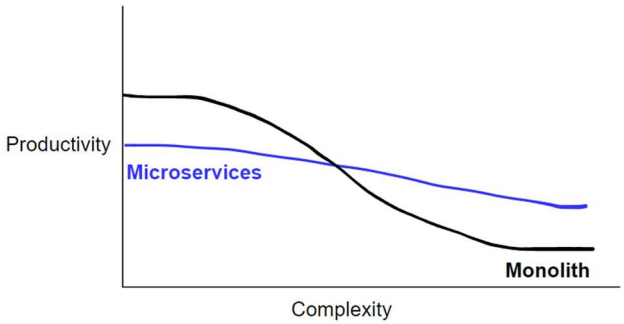

Figure 2: Productivity comparation between monolithic and microservice architectures [5].

To solve these problems, in 2009, Netflix was the first big company that migrated from a monolith architecture to a microservice architecture. This shift aimed to overcome the limitations of monolithic architecture and embrace the advantages offered by microservice architecture [6].

2.2 Definition of microservice architecture

Microservice architecture is a modern software design approach that structures an application as a collection of small, independent, and loosely coupled services. Each microservice is focused on a specific business and can be developed, deployed, and scaled independently.

The microservice architecture treats each *service* as the unit of modularity. Each service has its own responsibility. The services can only be used through its API, which is a difficult constraint to violate. For example, direct access to the microservice's database is not allowed. Instead, we must interact with the database via an API exposed by the service. This constraint preserves the modularity of the application over time [7].

The microservice architecture has two main characteristics:

Modularity

Microservices are based on the principles of doing one thing well. This approach is based on the principle of Single Responsibility by Robert C. Martin, which states:

"*Gather together those things that change for the same reason and separate those things that change for different reasons*" [8].

To create a robust microservices architecture, it is crucial to understand the distinct boundaries of the business. Every identified boundary becomes a well-defined microservice. Each microservice is responsible for implementing the services within a particular scope [3].

For instance, if we have an online shopping service, we will have various functionalities, such as order management, inventory tracking, user authentication and so on. Each of these represents distinct business domain. In the microservices approach, each of these functionalities becomes an individual microservice.

Independence

Each microservice operates independently from others, which means that the services can change and update without the need to impact other microservices or consumers [3][9]. To enforce this separation, all communication between microservices is via network.

Moreover, our microservices must expose their services through a technology-agnostic interface. With the objective of ensuring that the consumers of the microservices are not affected by changes or updates in the microservices. The common way to expose the services of any microservices is through an Application Programming Interface (API) that we will cover later.

2.3 Microservices advantages

By meeting the two main microservice characteristics modularity and independence. The microservices architecture gives us the following advantages.

Technology heterogeneity

With architectures that are composed of multiple and independent services. We can use different technologies and programming languages for different services. This allows us to use the most suitable technology for each job [3].

Microservices not only grant us the ability to stick to familiar technologies well known within the company, but also open the doors to exploring new languages and technologies. This advantage allows us to implement almost all microservices with technologies that are well known while simultaneously introducing a subset of microservices with the latest technologies. This dynamic approach keeps the innovation in the company and it adapts to future advances in the technologies.

Isolated complexity

Microservices allow us to isolate the complexity of the system by breaking down the system into numerous independent services, each service dedicated to a specific business. In this way, the development teams can focus on the individual services without being overwhelmed by the intricacies of the entire application, commonly in large monolith systems [3].

In the end, each microservice becomes an independent unit with its data, logic and dependencies. With this isolation we will be able to differentiate which microservices will have more business complexity. It allows us to dedicate more resources to microservices with more complexity and less resources to microservices that we know are simpler.

Moreover, isolated complexity enhances the system's resilience. If an issue shows up within one microservice, it can be fixed individually without compromising the entire application.

Scaling

The concept of scaling refers to the ability to expand or contract individual services based on demand. Microservices grant the flexibility to scale specific components of the system independently.

In the past, with the monolith architecture, if one small part of the system is performing poorly, the monolith forces us to scale the entire system as one big unit. This problem increases if our system is in an on-demand platform like AWS.

Besides microservices adapting so well to on-demand approach, if we have the problem of a traffic spike in our system, we can scale only the microservices that are engaged at that moment, keeping the rest of the system in less powerful hardware.

Easy deployment

To deploy one tiny code is not the same as deploying large million-line code. With monolith system if we discover a bug to fix or must add a new feature, the architecture forces us to deploy the entire system, which could be a high-risk deployment.

With microservices, we would only have to deploy the single service that has the bug or needs the new functionality without affecting the entire system [3].

Team organization

One of the major problems in large monolithic applications is to handling large teams. It is well known that small teams work better and are more productive than large teams.

Microservice architecture allows us to decentralize and create autonomous teams. Each team becomes responsible for specific microservices. These autonomous teams own the entire lifecycle of their assigned services [3].

Moreover, teams in a microservices environment often become specialized in a particular business domain or technical stack. This specialization leads to higher quality code and more effective problem solving.

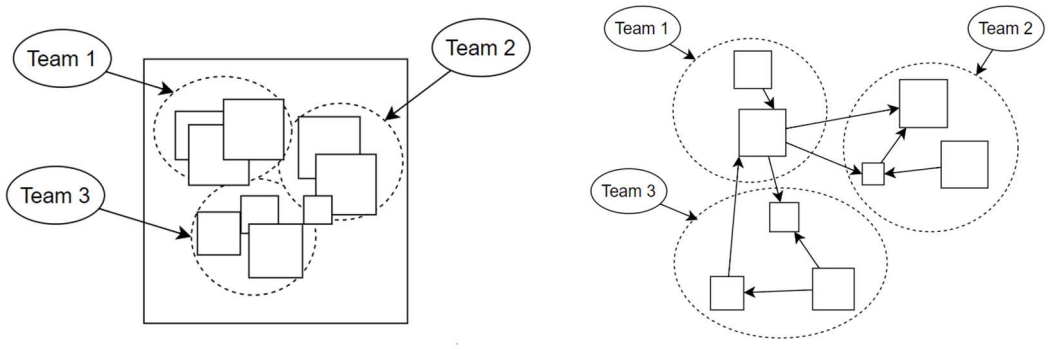

Figure 3: Large monolithic application versus Large microservice application [9].

Although this architecture tends to specialize teams, it does not mean that there is no internal collaboration. Development teams can collaborate with operations, testing and other relevant tasks to ensure a holistic approach of the entire software development lifecycle.

2.4 Microservice disadvantage

So far, we have seen the numerous advantages of the microservices architecture over its monolithic counterpart. But this architecture is not the golden hammer. It also presents several challenges that we have to bear in mind before making the decision about shifting our monolithic application or starting a new application with this architecture.

Complex service integration

Now, with microservices, we must think about how microservices communicate over a network. Which introduces complexities in term of service communication. And in systems with many services, it can be a challenging task.

Moreover, due to all the communication between microservices are through the network, it can introduce latency which impacts the application's performance [10]. In applications where it is critical to have a good response time, this problem forces us to add caching and other mechanisms that add even more complexity.

Data management

Instead of having only one database, each microservice typically manages its own database [3], leading to challenges in maintaining data consistency and integrity across services.

Other critical problems related to data are transactions. In traditional monolithic applications we can ensure the transaction mechanism with the unit of work pattern, which consist of if any operation in a transaction fails, the entire transaction is rollback [11].

But in a microservice architecture we will struggle to reach a proper transaction. The problem is that if we don't have a central database, we will not have a unit of work. To overcome this problem, we might require complex patterns such as the Sage pattern to ensure the consistency of the data between services [12].

Increased operational overhead

Microservices require a robust infrastructure to manage service, scaling, and monitoring. And this infrastructure becomes more complex with the number of microservices. This

problem requires more specialised teams only to handle this kind of infrastructure and it increases the cost of infrastructure maintenance.

2.5 Microservice integration

Until now, we have seen the difference between monolithic and microservice architecture. Then we saw the two main characteristics that any architecture has to meet to be a microservice architecture. And we also saw the advantages and disadvantages related to the microservices.

Now, in this section, we want to focus on how microservices can be coordinated and how microservices expose their own services and resources.

Orchestration vs choreography

In microservices, coordinating the interaction among services can be approached in two different ways.

Orchestration

With orchestration we rely on a central service that, like an orchestra director, guides and drives the job and the work among services. This approach provides centralised control of the flow of our application.

For example, in the Figure 4, the Customer service manages all the process. It calls the Loyalty points service to add some point to the new customer. After that, it calls the Collective service to add the new customer to a collective of our business. And in the end, it calls the Email service to send the corresponding email.

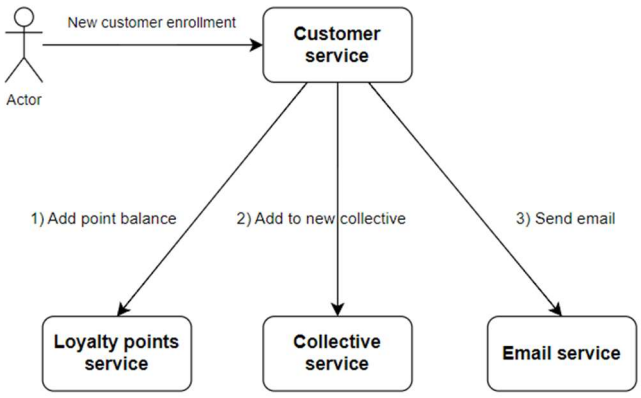

Figure 4: Example of orchestration microservice architecture [3].

The main advantage of this approach is that it is straightforward. We could get the diagram and model it directly into a code. And if the call among services is synchronous, which happens in almost all the cases in this approach, we could know the status of each stage.

One problem is that the orchestrator can become too much of a central governing authority. It means that this kind of service becomes a single point of failure.

And other problems are that the orchestrator becomes extremely tight and brittle to change, which increases the cost and takes us away from independent, and loosely coupled services that we are following with microservice architecture.

We also have to be aware of the availability in a distributed system. The availability of a system operation is the product of the availability of the services that are invoked by the operation [7]. Following the example of Figure 4, if we suppose that every service that calls the customer service has an availability of 99%, it means that the order service will have an availability of $99\%^3 = 97\%$ which is substantially less. If we want high availability we must minimise the number of synchronous communications.

Choreography

Choreography relies on each service knowing its role and responsibilities within in a system. With this approach there is no central point of control, instead, each service works independently, it just subscribes to events and reacts when they occur.

Now, following the same system example, the Customer service publishes an event to an Event broker (message queue) and the other services Loyalty service, Collective service and Email service subscribe to these events.

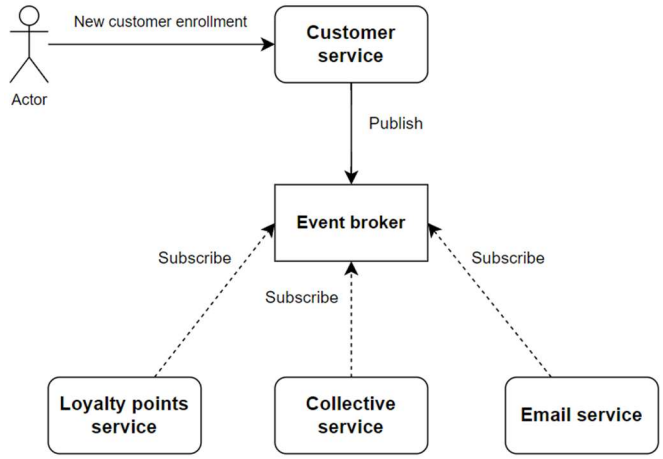

Figure 5: Example of a choreography microservice system [3].

The main advantage of choreography is that it is more prone to loose coupling among services, which enhances system resilience and scalability. Moreover, this kind of system tends to be more flexible and amenable to the change.

An additional benefit of this approach is its remarkable resilience. The message broker acts as an intermediary that temporarily stores messages until they are consumed. This

feature ensures that if any service becomes unavailable, the message will be safely held until the service is restored and ready to process it.

However, the lack of central control makes the system harder to understand, this means that additional effort is necessary. For example, building a monitoring system to track what is happening in our system also requires careful design to ensure that events are handled in a reliable way.

Both orchestration and choreography have their place in microservice architecture. In the real world, many systems use a combination of both, some with some parts with an orchestration approach and others with a choreographed approach to leverage the benefits of both [13].

2.6 API Concept

Once we have seen the two main strategies of internal microservices coordination, now we will delve into how microservices expose their services through a well-defined API.

The acronymous API means Application Programming Interface which is a set of rules and tools that enable the communication among software applications. It exposes a software interface which is considered contract, which describes the services that users are allowed to use and the way that they can communicate with [14].

The main purpose is to abstract the implementation and only expose the actions and the objects that the developers need. Calls to the API are commonly called requests or endpoints.

There are four different API types:

- **SOAP API (Simple Object Access Protocol)**: It consists of communication between client and server through XML messages.
- **RPC API (Remote Procedure Call)**: It allows calls to remote functions on external servers as if they were local software.
- **WebSocket API**: It allows to stablish two-way communication between the client and the server.
- **REST API (Representational State Transfer)**: It is an interface that uses two systems to communicate through the internet.

Although several types of API exist, the fact is that nowadays almost all APIs are REST API [15].

A REST API which acronym means Representational State Transfer, it is an architecture for designing networked applications. REST is based on a series of principles. The key principles are:

Resources

Resources are the key entities or objects that an API will manage. Each resource must be uniquely identified by a URI which differentiates different types of resource [16]. For example:

```
https://example-tfg.com/api/v1/books
https://example-tfg.com/api/v1/users
```

Code 1: Example of http resource requests.

The Code 1 provides a representation of the different types of resource (books and users) that the server has. It is important to bear in mind that all resources should be grouped by names and not by verbs. For instance, we cannot define a resource like *getAllBooks*.

```
Good ->  https://example-tfg.com/api/v1/books
Bad  ->  https://example-tfg.com/api/v1/getAllBooks
```

Code 2: Example of one well defined resource and one bad defined resource.

HTTP methods

The HTTP methods are the verbs used to perform operations on resources. There are many methods nowadays [17], but the most important are:

- **GET**: Retrieve a resource or a collection of resources.
- **POST**: Create a new resource.
- **PUT**: Update an existing resource.
- **PATCH**: Only update a partial existing resource.

When we perform any request to the API it is mandatory to specify the HTTP method. For example, if we want to retrieve all the books saved on the server: example-tfg.com through its API we need to indicate the method GET to get the books resources.

HTTP status codes

When the server responds, it always comes with the HTTP response status code [18]. This code is grouped in five classes. The most used codes are 200 for ok, 201 created, 404 not found and 500 internal server error. The status code groups are:

- **Information**: 100 - 199
- **Success**: 200 - 299
- **Redirection messages**: 300 - 399
- **Wrong on the request side**: 400 - 499
- **Wrong on the server side**: 500 – 599

Following the previous example, we perform the GET operation for the books resource, it could return if all goes well, the response body will contain all books stored and 200 status code. If there is no bookstore in the server, it will return a 404 “resources not found”.

API-first approach

Although we have defined well the API interface, it will not be useful if it does not meet the client’s requirements.

For example, it is typical that the backend side and the frontend side are different teams. It is possible that both sides could have completed the development and the application did not work. This may be due to the fact that the API interface was poorly defined and the two applications could not communicate.

For this reason, as much as it is possible we have to use an API-first approach to define the interface [19]. It consists of iterating the API definition before starting the development process.

First, we write the interface definition with an interface definition language. Nowadays the standard is Swagger [20]. Second, we review the interface definition with the client developers. And after the iteration process of the API definition, we can now implement the services.

In this way, we increase the success of meeting the requirements of the clients.

2.7 Command Query Responsibility Segregation (CQRS)

Once we have seen how microservices expose their resource through exposing an API, we now explore an advanced architectural pattern name Command Query Responsibility Segregation or CQRS. This pattern will be applied practically in Chapter 5, as a backend implementation of the recommendation system. CQRS pattern allows us to have a document database in the product side and a graph database optimized for recommending product.

When we are modelling a microservice architecture, we have to take into account that the use of CQRS pattern may not be beneficial for our system. While CQRS can add significant advantages, it also introduces risks and complexity, even for experienced teams [21].

The three most important scenarios for using the CQRS pattern are [7]:

- **Data aggregation**: When we need to aggregate data across multiple services which result in expensive and inefficient in-memory joins.
- **Database capabilities**: When the microservice that stores data has a database that does not efficiently support the required query.
- **Separation of concerns**: The need to separate concerns when the microservice that has the data is not the service that should implement the query operation.

The CQRS pattern consists of the separation of concerns. This pattern splits the persistent data model and the modules (microservices) that are used into two distinct parts:

- **Command side**: Which performs the CUD operations (Create, Update and Delete). It is the responsible for modifying the data. This side publishes an event through an event buffer (queue) when its database changes.
- **Query side:** Only implements the *Read* operation and subscribes to the event buffer where the command side is publishing events. This ensures that the query side keeps its data synchronized with the command side.

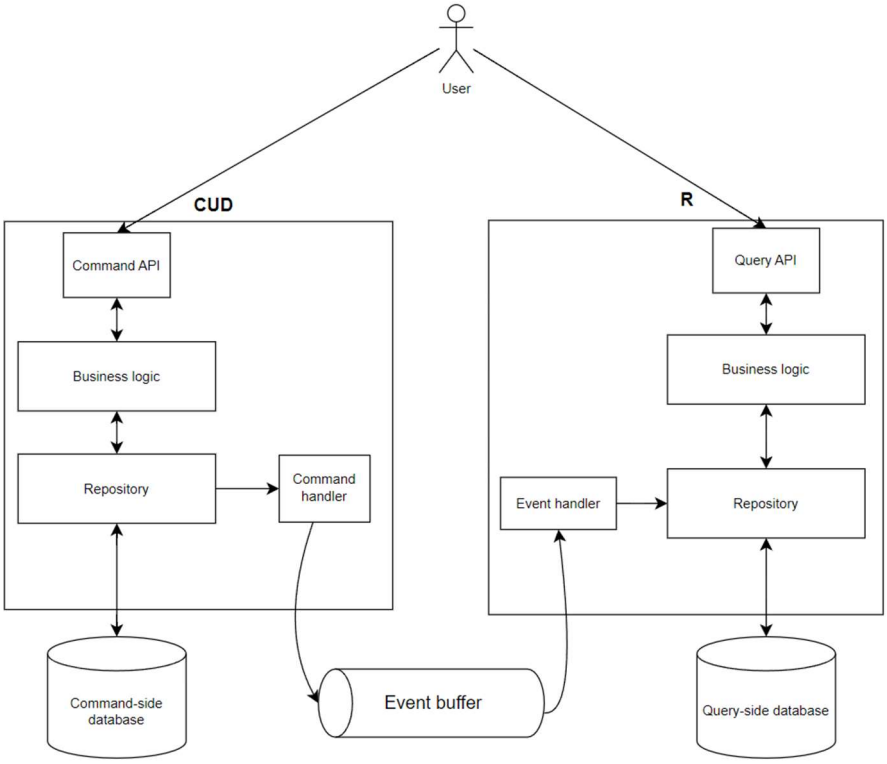

Figure 6: CQRS pattern.

The benefits of CQRS are [7], [22]:

- **Optimize performance**: The *Read* side can be scaled independently and has an optimized schema or database for queries.
- **Improve security**: CQRS ensures to ensure strict access for *Write* operations while allowing broader access for readers.
- **Separation of concerns**: Splits the *Write* side and the *Read* side improves maintainability and flexibility.
- **Simple queries**: Query-side has optimized views which avoid complex joins and queries.

But these benefits do not come free, the drawbacks which present of CQRS pattern are:

- **Eventual consistency** : The problem with having multiple databases is that the read data and the write data may not be synchronized in a short period of time [23].
- **Complexity**: Although the concept of CQRS is straightforward, it leads to a more complex architecture structure.

2.8 Saga pattern

ACID which acronym means Atomicity, Consistency, Isolation and Durability are the 4 key properties that define a transaction. A transaction is a sequence of database operations that satisfies the ACID properties.

Implementing a transaction in a monolith architecture is an easy task. In a microservice architecture we can also perform transactions inside a single service. However, the challenge is implementing transactions in operations that involve multiple microservices [7].

In distributed systems the traditional approach is to use two-phase commit (2PC) [24]. This protocol is designed to ensure that each node needs to know if other nodes successfully stored the data or if they fail [25].

This protocol consists of two phases:

- **Prepare**: The transaction coordinator sends a *prepare* message to all participants in the transaction. Each participant votes 'yes' if the transaction can proceed or 'no' if it cannot.
- **Commit/Rollback**: If all the participants vote 'yes' the coordinator notifies all participants to commit, otherwise, the coordinator will notify them to rollback.

The main problem with distributed transactions is that some modern technologies such as NoSQL database or moder message broker don't support them explicitly. Another problem is that distributed transactions are based on a synchronous communication which we saw that reduces the availability of the services [7].

On the other hand, the Saga pattern as an alternative approach in microservice architecture has gained popularity. Saga is a sequence of local transactions where each transaction updates data within a single service.

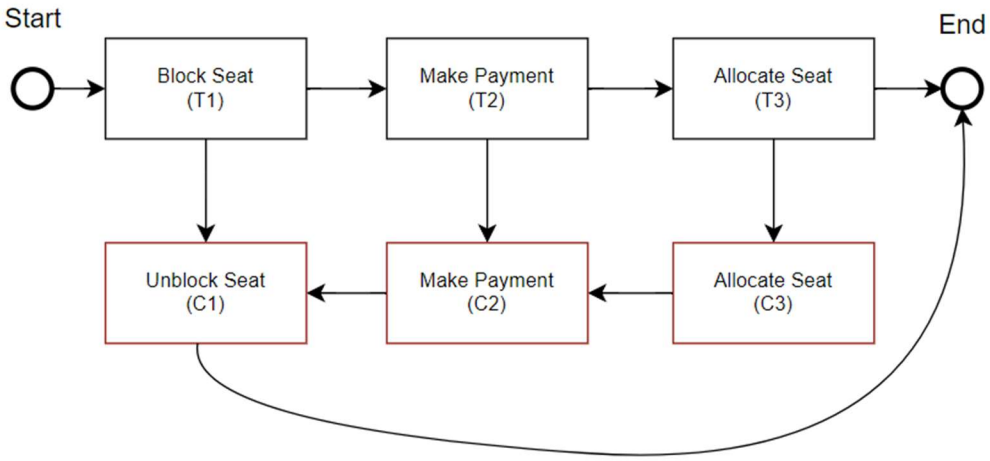

Figure 7: SAGA pattern example [26].

Saga is divided into two parts:

- **Local transaction**: Each service performs its transaction. The completion of a local transaction triggers the next transaction.
- **Compensation**: If a service fails the saga's coordination mechanism must execute the compensating transactions.

There are two types of Saga pattern mechanisms:

- **Orchestrations-base**: This approach uses a central coordinator to manage the sequence of transactions. The orchestrator tells each participant service what transaction to execute and in what order.
- **Choreography-base**: In this approach there is no central coordinator. Each participant service performs its transaction and then publishes an event that triggers the next step in the process. There is no central coordinator, instead, each service subscribes to each other's events and responds accordingly.

In general, the orchestration-based approach is easier to understand for the developers. Additionally, the choreography-based approach has an important benefit, it ensures that all the steps of the saga are executed thanks to the asynchronous communication, even if one of the participants is temporarily unavailable [7].

Chapter 3: Observability

Every application has an inherent amount of complexity. The only question is: who will have to deal with it; the user, the application developer, or the platform developer?

Larry Tesler – Pioneer of Human-Computer interaction

Observability is a concept that has become a hot topic with the disruption of the microservice architecture. It refers to the ability to gain insight into the internal workings of a system by collecting, analysing, and understanding relevant data. Observability provides a means to illuminate the otherwise opaque nature of complex distributed systems, allowing developers and operators to effectively monitor, troubleshoot, and optimize the performance of their applications.

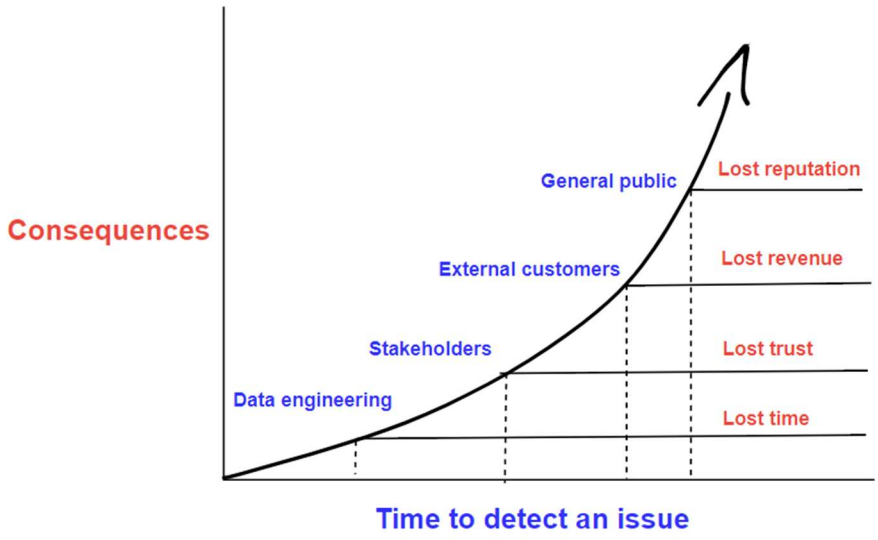

Figure 8: Consequences of the time to detect an issue [27].

With traditional monitoring tools for production software, we can see performance issues or perceive that a problem is occurring. However these tools do not allow us to dive into the root of problems. For example, it does not work for us to have a graph that shows 90% of our incoming request are fast and cannot dive or investigate for those that are not performance like our expectations or even more do not end with a success end.

With observability tools we are looking for understanding and explaining any state of our system that can get into no matter how novel or bizarre are, without shipping new code or looking inside the code system [4].

The main observability goals:

- The primary goal is to infer the internal state of the distributed system from observable outputs, in other words, to understand the inner workings of our applications.
- Observability aims to identify issues before they escalate into user problems, with early detection of issues such as errors, bugs or bottlenecks.
- To understand the inner working and system state solely by observing and interpreting with external tools.
- To reduce the required knowledge in advance of the system when debugging an issue.
- We shift the best debugger from the person who has been there longest on the team to whoever was the most curious and most persistent with the observability tools.

Observability is a measure of how well we can understand and explain any state our system can get into, but this feature is not binary, according to the tools that we use, and the way that we implement this, means we can achieve more or less grade of observability. Moreover, observability comes with its own drawbacks:

- Managing data volume generated by the system is a difficult challenge. This requires efficient data storage, indexing, querying mechanisms and so on.
- Correlating data across services to get a unified overview of the system is complex and required specialized team.
- Observability could hinder the performance of the system adding an overhead utilization of the resources.

3.1 Cardinality

The mathematical definition of cardinality is the measurement of the number of elements in the set [28]. In terms of the databases, cardinality refers to the uniqueness of data values contained in a set. Low cardinality means that there are more duplicated elements in a particular column. High cardinality means that the column contains more unique values[29].

In the context of observability high cardinality information is almost always the most useful. For instance, request IDs and user IDs are high cardinality data. On the other hand, HTTP status code (200, 404, 500) are a low cardinality data.

3.2 Dimensionality

Dimensionality in the context of observability refers to the number of attributes that are used to describe any piece of data. Each dimension provides additional context and granularity of the data.

For instance, in this JSON we can see a six-dimensional data for a particular request:

```
{
  "method": "GET",
  "url": "/api/products/123",
  "response_status": 200,
  "response_time_ms": 150,
  "user_id": "user789",
  "client_ip": "192.168.1.25",
  "server_id": "server42",
  "timestamp": "2023-12-23T12:34:56Z"
}
```

Code 3: Example of six-dimension data.

3.3 The three pillars of Observability

Before diving into the pillars of observability, it is crucial to define what an event is in this context.

An event is a record of everything that occurred in our system while one request hits our services. This could be any action or set of actions triggered by the request. The event captures data such as request ID, user ID, timestamps, and other relevant data during the lifetime of the request [4]. The event has a structured nature that makes them easy to parsable and analysable.

Metrics

Metrics are numerical representations of data measured over an interval of time. They supply information related to the system, like memory usage, CPU load, response times, requests per second, etc [4].

They are useful for real-time monitoring of our system and help in understand the system's health and help us make more informed decisions about scaling and system improvements.

For instance, if we see that almost all weekends the CPU usage level of a particular microservice regularly approaches the critical range of 90% - 100%. We can decide to auto scale this microservice with more instance on weekends to manage this elevated level of load.

Logging

Logs are documents generated by the systems that record all events and changes in the systems. They are very useful for tracking errors and understanding the sequence of events leading to an issue. Moreover, logs can serve for audit purposes and depending on the industry it could be mandatory to save them [4].

Log files are large blocks of unstructured text, designed to be readable by humans but difficult for machines to process. Traditional logs are unstructured, often generate the details of one event in multiple lines of text, like:

```
2023-12-23 09:15:03 Server started successfully.
2023-12-23 09:17:21 User 'admin' logged in from IP address 192.168.1.15.
2023-12-23 09:20:45 Database connection established.
2023-12-23 09:22:30 WARNING: Memory usage exceeded 80% of capacity.
2023-12-23 09:23:10 User 'john_doe' attempted to access restricted resource.
2023-12-23 09:24:05 ERROR: Unable to send email notification to user
'jane_smith'.
2023-12-23 09:25:00 Scheduled backup process initiated.
```

Code 4: Example of unstructured logs application.

The unstructured logs pose a challenge for automated processing and analysis, for this reason the best practices in recent years have recommended creating structured logs data designed for machine readability like:

```
timestamp=2023-12-23 09:15:03 event="Server started" status=success
timestamp=2023-12-2309:17:21 event="User login" user=admin
ip_address=192.168.1.15 status=success
timestamp=2023-12-23T09:22:30     event="System warning" message="Memory usage
exceeded 80%"
timestamp=2023-12-23T09:24:05     event="Email notification error"
user=jane_smith error="Unable to send email"
timestamp=2023-12-23T09:30:00    event="Backup process" status=failure
error="Network timeout"
timestamp=2023-12-23T09:40:00    event="System health check" status=completed
```

Code 5: Example of structured logs application.

But unstructured data is not the only challenge, the volume of data in the production environment is high, especially in large distributed systems, for this reason is not easy or cheap to store and manage logs.

Log aggregation pattern

In traditional applications we configure the place where logs will be stored. But in microservices architecture with modern deployment technologies such as containers and

serverless, this traditional approach does not scale well. In distributed systems, we must log to *stdout* and the deployment infrastructure will handle these logs.

The common approach is to use the log aggregation pattern. It consists of all the microservices sending to a central log aggregator all the logs that they generate. The central log aggregator stores the logs and provides a single point to access logs, allowing querying, searching and visualization [30].

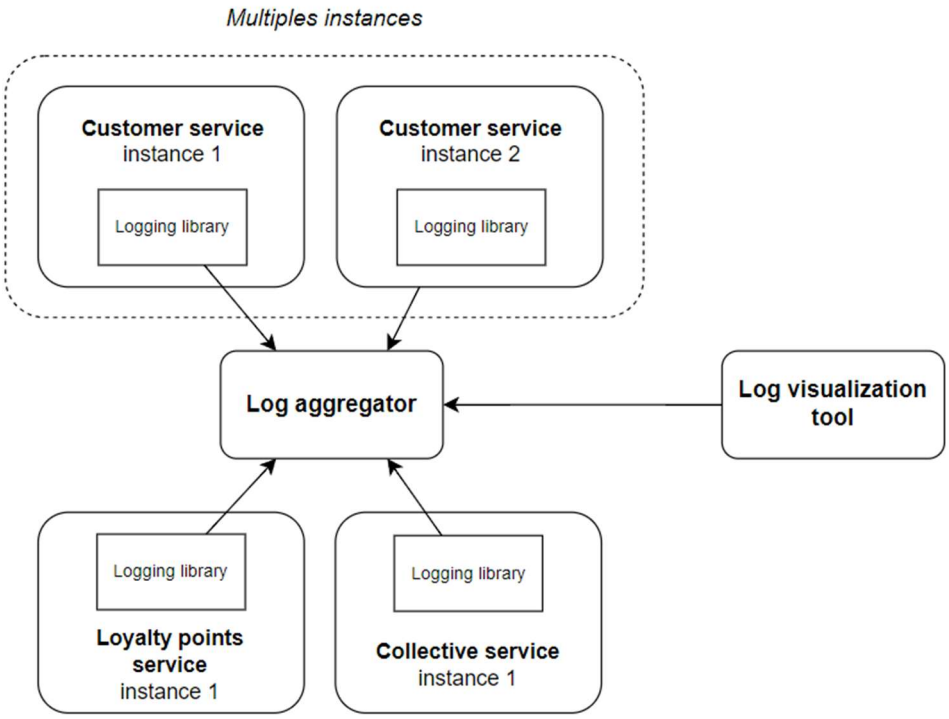

Figure 9: Log aggregation pattern [7].

One of the popular open-source logging infrastructures is the ELK stack [31], but in this project, we used an open-source Loki Grafana solution [32]. One of the differences is that Loki uses less memory space to store the logs than ELK stack. This is only due to the fact that Loki only indexes the metadata of the log rather than the full text of the log. Loki is suitable for systems that do not have much store capability.

Tracing

Tracing in the context of observability is the process of tracking the progression of a single request. In a microservices architecture, where a single request may traverse processes, machines and network boundaries, tracing becomes essential to understand the journey and interactions of these requests. In other words, a trace is simply a series of interconnected events.

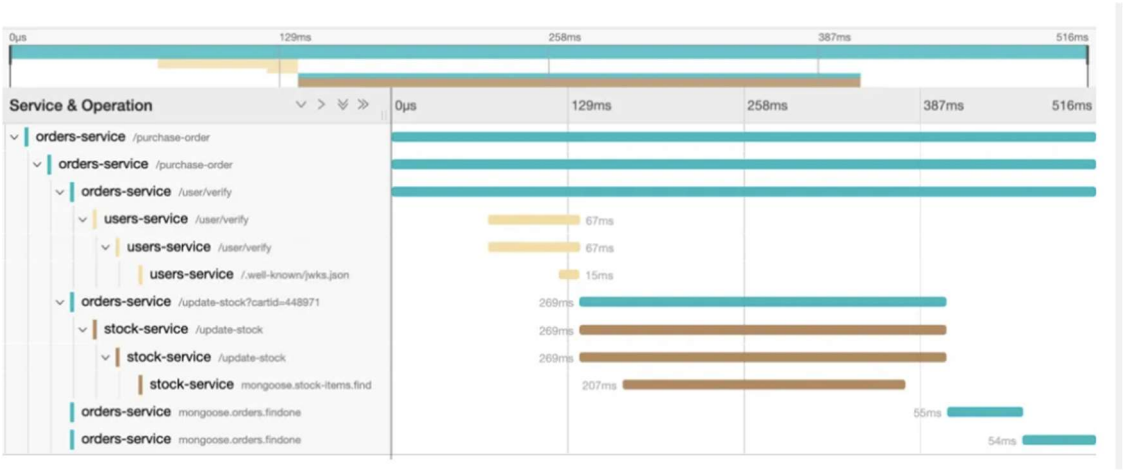

Figure 10: Example of waterfall visualization [33].

In this waterfall-style visualization of a trace, we can see how the initial value is affected by a series of intermediate values, leading to a final value. In this way, traces help us to understand the system dependencies in a clear view.

The tracing implementation is compounded by 4 mandatory keys [4]:

- **Trace ID**: is a unique identifier value that we can map to a particular request. It is used to group all span IDs, allowing the entire request path to be viewed as one unit.
- **Span ID**: is the ID that identifies a particular unit of work. Each service or process the request passes through will have its own span ID.
- **Timestamp**: records the start time of each span ID.
- **Duration**: provides the duration of each span ID.

Besides these keys, a good tracing system uses tracing propagation to ensure the continuity of tracking the request across different systems. Each system or service the request interacts with adds its own span ID maintaining the integrity of the trace.

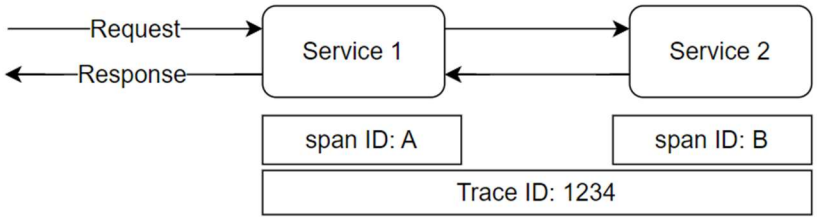

Figure 11: Example of traceID and spanID [34].

For example, in Figure 11, we can see that the request is identified with the trace ID: 1234 and the Service 1 has its own span ID A and the Service 2 has span ID B. This allows tracking the request's path and understanding the interactions between services.

Nowadays, multiple standardization exist to achieve tracing propagation like W3C or B3. These standards define a unified approach to managing the trace across microservices. They add specific tracing information to the HTTP request headers. These standards enable tools to properly track events in a distributed system [35].

3.4 Combining Tracing, Metrics and Logs

So far, we have seen the three pillars separately. Each of them has its advantage: metrics perform well for performance and health monitoring, logs for detailed event information and tracing for revealing the path and lifecycle of requests across services.

In the real world, it is rare to find teams that have all the pillars in they own observability tools, indeed, the most common situation is to find that the teams only gather logs. But with the disruption of microservices architecture and the fact that systems are increasingly more complex, enhancing the observability of these systems is almost mandatory.

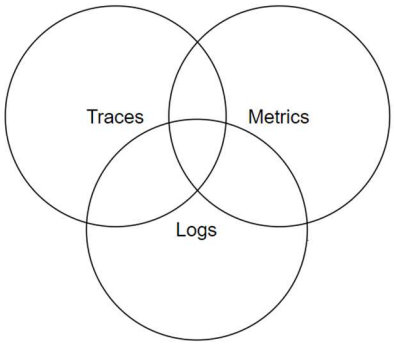

Figure 12: Tracing, metrics, and logs all together [36].

When we combine tracing, metrics, and logs, we create a holistic observability strategy. While each pillar offers value on its own, their integration provides a deeper and more complete understanding of the systems. This integrative approach enables teams to not only detect and react to issues but also proactively optimize systems and enhance user experience.

3.5 Health check pattern

To end this chapter, we will cover the health check pattern, which is an essential part of maintaining the stability and reliability of a distributed system. This pattern is designed to automatically report the status of an individual component within a system.

To understand the utility of this pattern, let's imagine an instance of a microservice that takes 20 seconds to start. It will be a problem if the load balancer routes HTTP requests to this instance. Or we could imagine that an instance of a microservice runs out of its

database, but the instance continues to be operative. It will not be capable to access the database.

The health check pattern consist of exposing a service that responds the status of the microservice. For example, with the Spring Boot Actuator library [37], we can know the status of the microservice calling to the *GET actuator/health*.

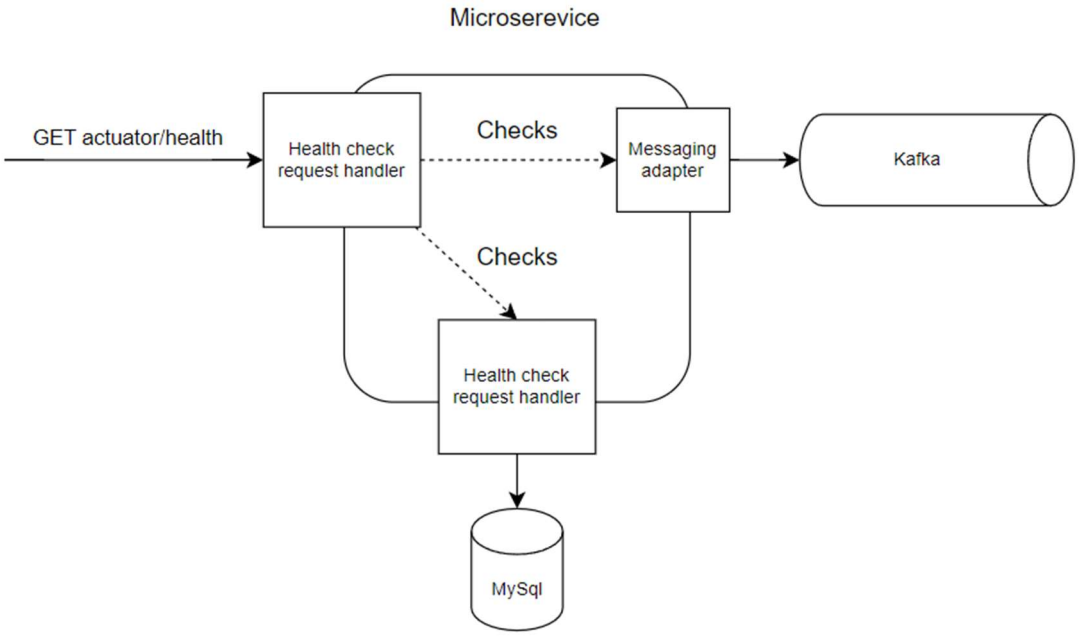

Figure 13: Health check Spring Boot [7].

In Figure 13, we can see that the health pattern implemented checks the status of the dependencies of the microservice. In this case checking the ability to interact with a MySQL database and a Kafka message broker.

Chapter 4: Recommendation System

If people don't really have a choice, how is it really a recommendation?

Michael Schrage – Author of "Recommendation Engines"

In today's digital landscape, it is well known that providing too many options to users might demotivate and make them less satisfied with their decisions.

In the highlighted study "When Choice is Demotivating: Can One Desire Too Much of a Good Thing?" [38], they made an experiment that offers the customers two tasting booths, one with a limited 6 flavours of jams and the second one with an extensive selection of 24 selection of different flavours jam.

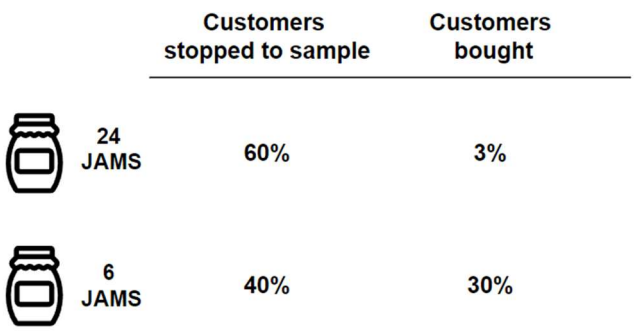

Figure 14: Jam experiment on demotivation of choice.

The result was that customers were more attracted to the tasting booth with 24 jam options but were more likely to purchase jam when only 6 options were available. This discovery manifested:

The pain of choosing is greater than the benefit of making the choice.

In this context, recommendation systems come to solve these problems, guiding and influencing user's choices and enhancing their experience. In fact, nowadays, recommendation systems have emerged as essential tools to help users navigate through the vast array of choices and make informed decisions.

These recommendation systems exist in almost all domains, from e-commerce platforms suggesting products to healthcare with personalized medicine and treatments

recommendations based on a patient's genetic profile, medical history, and preferences. Understanding the mechanisms behind these systems is crucial, as they significantly influence consumer behaviour [39].

In this chapter, we will see in the first section a brief introduction to the recommendation systems and its main categories. Then, in the second section we will discuss our own system recommendation.

4.1 Categories of recommendation systems

Recommendation systems are mainly classified into three different categories [40].

Collaborative filtering

Collaborative filtering-based systems make recommendations based on gathering and analysing data on the users' behaviour. Predictions are made about what the user will like based on their similarity to other users.

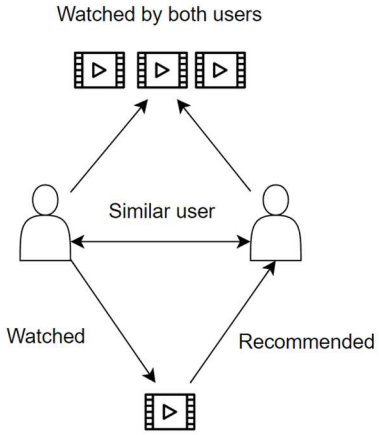

Figure 15: Collaborative filtering approach [41].

For example, we have two users: Alice and Bob. Both have watched and liked the movies "Inception", "The Matrix" and "Interstellar". The system identifies them as having similar tastes based on this overlapping history of watched movies. Now if we suppose that there is a movie called "Dune" watched by Alice. This movie will be recommended to Bob if he has not seen it yet.

One of the benefits of collaborative filtering is that the recommendation system can recommend items without understanding the objects themselves. Another benefit is that it can provide recommendations that users might not discover on their own.

On the other hand, there are two main problems with this approach. One is the cold start problem. New users or items with little or no interaction history pose a challenge, making recommendations difficult initially. The other problem is related to the sparsity of the

data. It occurs when most of the users or items matrix is empty or unknown. This reduces the accuracy of recommendations.

Content-based

Content-based recommendation systems are based on the content or description of the product, item or key term searched by the user. The central assumption of content-based filtering is that if we like a particular item, we will also like a similar item.

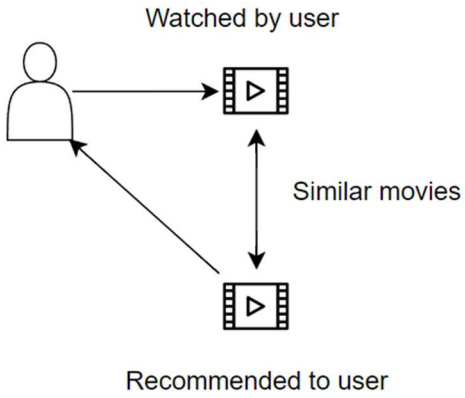

Figure 16: Content-based filtering approach [41]..

For example, if we consider a user named Bob, who has watched the movie "The Hunger Games", with characteristics like science fiction and strong female leads. The system will search for movies with similar characteristics and could recommend Bob to watch the movie "Gravity", which has similar characteristics such as science fiction and strong female leads.

The main benefit is that this approach avoids the cold start problem for items, as recommendations are based on item characteristics. Another benefit offered by this system is the ability to recommend new and unpopular items, that otherwise, could be ignored.

However, finding the right characteristics of the items is hard. Moreover, this system can lead to a filter bubble, where the system continuously recommends similar items, reducing the diversity of the recommendations.

Hybrid-based

It is well known that a hybrid-based system combines both collaborative and content-based methods leverage the strengths and mitigate the weaknesses of both approaches. In fact, the Netflix Prize competition for the best recommendation system revealed that hybrid recommenders, which combine content-based and collaborative filtering systems, produced recommendations superior to either approach alone [39] [42].

This hybrid-based system can be achieved in various ways, such as by making predictions using both methods and combining them, adding content-based capabilities to a collaborative approach, or vice versa.

4.2 Focus on content-based approach

In this project, we will focus on the content-based approach. The decision is driven by the resilience to the common cold start problem issue. This content-based recommendation system will be allocated in the microservice named recommendation-microservice and all the implementations will be explained in the Chapter 5.

Our proposed recommendation system relies on the strengths of the graph database [43]. Which with one query to the graph database, we can achieve a robust and scalable content-based recommendation system.

Core components of a graph database

The Graph database concept is based on a graph structure, representing interested objects and relationships. A labelled property graph is made up of nodes, relationships, properties, and labels [44].

- Nodes contain properties, They are equivalent to documents that store properties in the form of arbitrary key-value pairs. In a relational database, it could be the records of the entities.
- Nodes can be tagged with one or more labels. These labels represent the role that the nodes represent in the database.
- Relationships connect nodes. A relationship always has a direction, a single name, and a start node and an end node.
- Relationships like nodes can also have properties.

Graph databases maintain relationships among data without join operations. In contrast to relational databases, where join-intensive query performance deteriorates as the dataset gets bigger. Graph database performance tends to remain constant while the dataset grows.

Our model graph database

Our model graph database will be compound by:

- Nodes labelled "Product Node". These nodes represent individual products in the graph database. These nodes will contain properties like "productId" and "productName", along with other attributes specific to the product.
- In the model, various types of relationships will be defined among the product nodes. These relationships could be "Category", "Material", "Colour", and so on. These relationships are crucial for the recommendation logic, as they allow the system to understand and quantify the connections between different products based on shared characteristics.

- Relationship weight, all the relationships include a "weight" property, which quantifies the relevance of that relationship. This is a critical aspect of the recommendation system. It allows us to prioritize certain relationships over others.

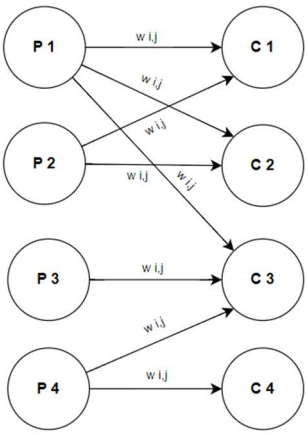

Figure 17: Model graph example.

In Figure 17, we have an example of our graph data model, where products are labelled with (P) and the characteristics with (C). The relationship between nodes are labelled with weights w(i , j) which represent the relevance of the relationship between products and characteristics.

Query to get similar product

Once we have correctly modelled the graph, our algorithm for obtaining similar products to a specific product is straightforward. It consists of counting the shared relationships between products that at least one shared characteristic, bearing in mind the weight of each relationship. And it returns the similar products in descending order.

This algorithm is achieved using the following cypher query.

```
MATCH (p:Product {productId: $productId})-[r]->(char)<-[r2]-(similar:Product)
WHERE  p <> similar

RETURN  similar.productId               as productId,
        similar.productName             as productName,
        SUM(r.weight + r2.weight) / 2    as similarityScore

ORDER BY similarityScore DESC
```

Code 6: Cypher query to fetch similar products by productId.

The left part of the MATCH statement `(p:Product {productId: $productId})` matching Products node with a specific "productId" that is provided as a parameter to the query.

The right part of the MATCH statement `(similar: Product)` takes all the Product nodes in the graph that have any relationship with the left node.

the WHERE statement `p <> similar` ensures that the product `p` is not the same as the 'similar' product found by the match.

The RETURN part specifies what to return as the result:

- `similar.productId as productId`: The ID of the similar product.
- `similar.productName as productName`: The name of the similar product.
- `SUM(r.weight + r2.weight) / 2 as similarityScore:` It calculates the similarity score by summing the weights on the relationship 'r' and 'r2' and then dividing by 2. This average is used to assess the similarity between the products based on the shared characteristics.

Chapter 5: Practical Implementation of an e-commerce

In this chapter, we delve into the implementation of an e-commerce platform with a content-based product recommendation system based on microservice architecture. The objective is to put into practice all the concepts seen so far. Starting with the microservice architecture, continuing with the content-based recommendation system and ending with the observability strategies which we are going to discuss in detail in Chapter 6.

System overview

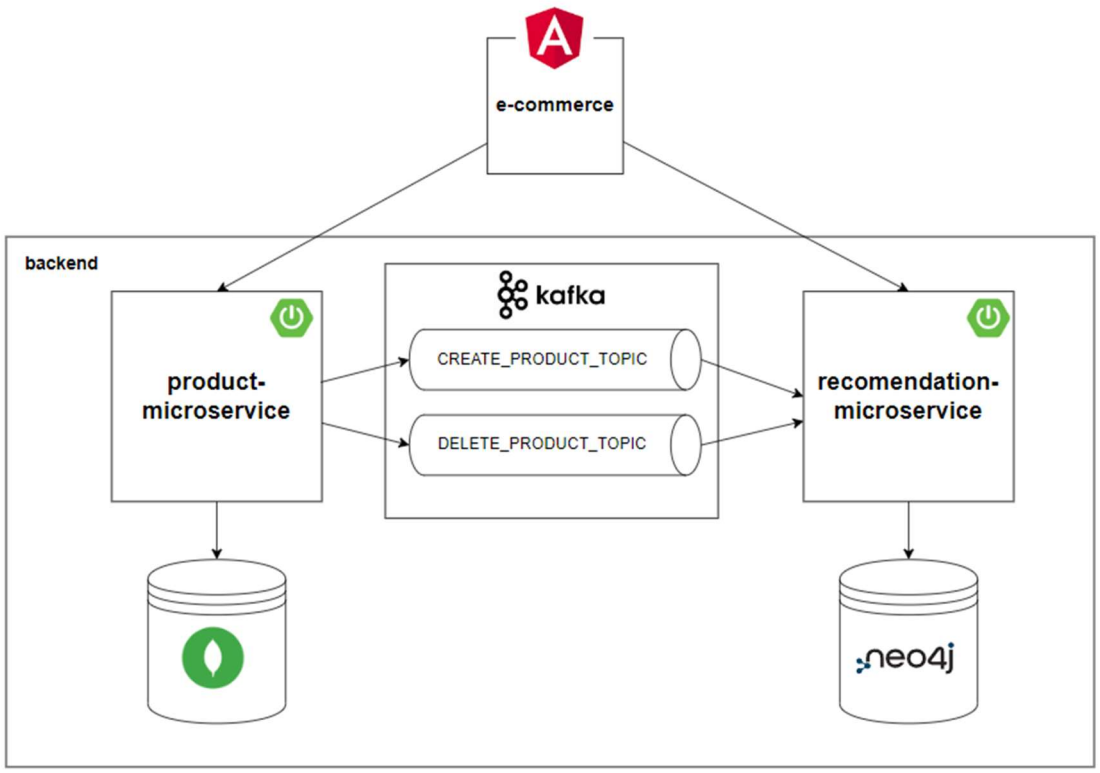

Figure 18: Overview of the system architecture.

The backend architecture consists of two microservices specifically designed to fulfil distinct roles within the system.

- **Product microservice:** Responsible for managing all the business logic around the products. It manages the product catalog, handling all CRUD operations and serving product data to other parts of the e-commerce system.
- **Recommendation microservice:** Responsible for calculating similarity scores among products and recommending products related to a specific product.

Apart from the microservices architecture, the system has two important components.

- **Frontend application:** Used to expose to users a list of the products and the details of each product along with the corresponding related products based on the recommended products provided by the recommendation microservice.
- **Queue message broker:** Provides an asynchronous communication between microservices and allows the use of a choreography approach to communicate the microservices.

5.1 Technologies justification

The following technologies were used to create the entire system:

Java

Java is one of the most popular programming languages for building microservices[45]. It is a robust, object-oriented programming language that is widely used in enterprise applications. The main advantage of this language is that Java is cross-platform. This feature makes Java an ideal choice for environments where applications need to be portable across different computing platforms.

Spring framework

The Spring framework is one of the most popular frameworks to use with Java. It simplifies the development process of Java based applications. Spring boot embedded by default a tomcat server, which is handy and useful in a microservice ecosystem.

In this project we used Spring boot, which is an extension of the Spring framework that eliminates the boilerplate configuration required for setting up the application.

The functionalities that are extensively used are:

Stereotype annotations

Spring provides stereotype annotations: *@Component*, *@Controller*, *@Service*, *@Repository*. These annotations are used to define the layers of the application automatically detected and register *beans* by Spring container. The particularity of these

annotations is that *@Component* is the main stereotype annotation, and the rest of the stereotype annotations are derived from *@Component*.

Inversion of Control (IoC) and Dependency injection

The Spring framework has a powerful Inversion of Control (IoC)container [46], which manages Java objects from instantiation to destruction. This feature allows the use of dependency injection in Java classes. With dependency injection, we reduce the coupling of the code and it helps to have more testable and maintainable code.

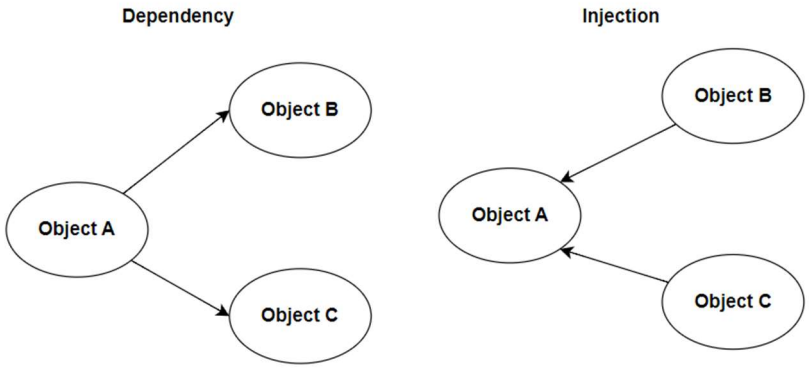

Figure 19: Dependency approach and Injection approach.

On the left side of Figure 19, it shows the typical scenario where object A is responsible for creating and managing object B and object C. The right side depicts dependency injection, here object A is not responsible for creating and managing object B and object B. Instead, these dependencies are injected into object A by the IoC container.

Aspect oriented programming (AOP)

Aspect Oriented Programming (AOP) [47], complements object-oriented programming (OOP) by allowing modularization of concerns such as transaction management. In this project we used AOP to intercept functions in the code and log their names and arguments. In this way, we achieve more reliable logs because it automatically records every service invocation [7].

MongoDB

MongoDB [48] is a document-oriented NoSQL databased based on a JSON schema. Data is stored in documents grouped in collections. Each document has a flexible schema, which means that each document inside in a collection can have different fields and structures. Moreover, the document maps the object in the application code directly, which makes it easy to work with.

As we know, relationships in SQL databases are made by joins, in NoSQL databases, we can add relationships by embedding documents inside other documents. Another approach is to reference the ID of one document to the related document. This approach simulates the traditional foreign key in SQL databases.

Neo4j

Neo4j [49] is a highly performing graph database that excels in managing data in the form of a graph. It is suitable for databases where relationships between data are a key feature, such as social networks, recommendation engines, fraud detection systems and so on.

One experiment to showcase the strength of Neo4j is the social network "friend-of-friend" experiment [4]. This experiment involves a simulated social network of 1,000,000 people, each with approximately 50 friends. The goal is to evaluate and compare the performance of Neo4j against a traditional relational database when querying multi-level relationships.

Depth	RDBMS execution time(s)	Neo4j execution time(s)	Records returned
2	0.016	0.01	~2500
3	30.267	0.168	~110,000
4	1543.505	1.359	~600,000
5	Unfinished	2.132	~800,000

Figure 20: Experiment result RDBMS vs Neo4j [4].

Figure 20 shows the experiment result, that in the search of "friends of friend" (depth 2) both databases perform well. From the search of "friends of friends of friend" (depth 3) and so on, Neo4j performs extremely well.

Kafka

Apache Kafka [50] is a message broker that is used in a broad variety of fields. It has become a key component in microservices architecture due to its ability to facilitate reliable, scalable and efficient communication between decoupled microservices.

The key concepts of Kafka are:

- **Event**: In Kafka, an event (message) is a record of data that is sent from a producer to a Kafka topic. The events in Kafka are immutable and have a default delete policy of one week.

- **Topic**: A topic is where the events are published. Topics in Kafka are multi-subscriber, and they can be consumed by multiple consumers.
- **Partition**: A partition is a division of a topic. This allows topics to be consumed in parallel. Each partition is an ordered and immutable sequence of events.
- **Consumer group**: A consumer group is one or more consumers that consume a topic. The consumer group divides the topic partitions among themselves, so each partition is consumed by only one consumer from the group.
- **Producer**: A producer is any application or service that publishes events to Kafka topics. The producer determines which partition is used in the topic.

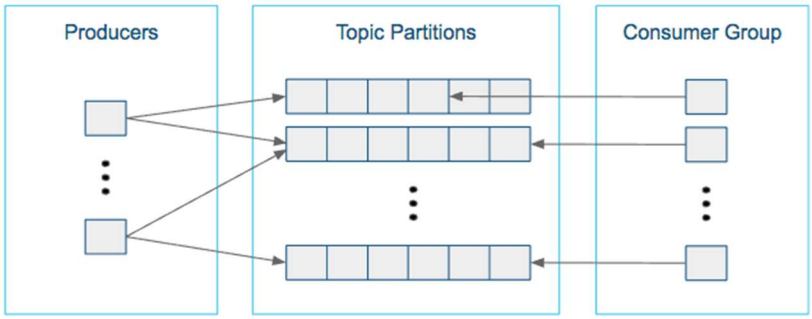

Figure 21: : Kafka architecture [51].

Angular

Angular [52], [53] is an open-source framework for creating Single Page web Application (SPA). It is built on TypeScript and runs on Node.js. This framework is widely used in large enterprise applications that require a sophisticated web application integrated with complex backend systems. Angular is commonly seen with the stack of Java + Spring Boot.

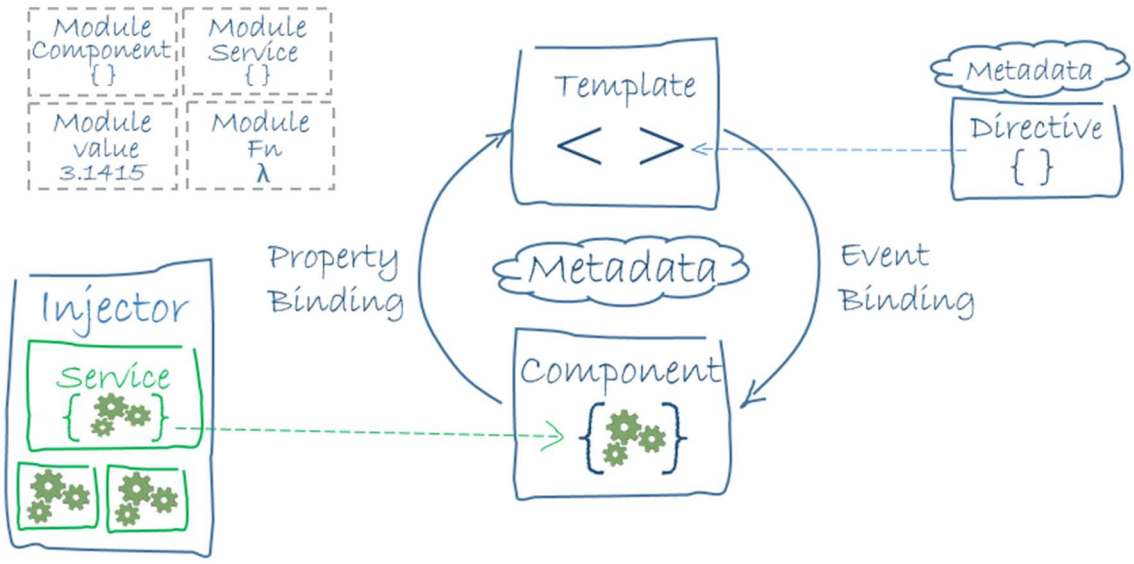

Figure 22: Angular architecture [53].

The principal concepts of Angular are:

- **Components**: The basic building blocks of Angular applications. An Agular application is a tree of Angular components. The component controls the template which is the view for the user.
- **Templates**: Define views in Angular applications. The template is a chunk of HTML that contains specific Angular elements and attributes.
- **Data biding**: It is the mechanism that coordinates the template part and the component part. Angular supports two-way data binding.
- **Services and Dependency injection**: For data that do not depend on a specific component and need to be shared across multiple components, we build a service class. And with Agular's dependency injection, we inject the service into multiple components.

Docker

Docker [54] is a platform that makes it easier to create, deploy and run applications using containers. With containers, we package the application with all the parts that it needs, such as libraries and dependencies. This ensures that the application will run on any other Linux machine. Docker effectively virtualizes the host operating system and isolates application's dependencies from other containers running on the same machine [55].

The core concepts of Docker are:

- **Images**: The Docker image is a template. Docker can build images automatically by reading the instructions from a Dockerfile.
- **Dockerfile**: A Dokerfile is a text document that contains all the commands needed to create an image. Using *docker build,* Docker automatically creates the image.
- **Containers**: Docker containers wrap a piece of software in a complete filesystem that contains everything needed to run the software. This guarantees that the software will always run the same, regardless of its environment.

In a microservice architecture, Docker is particularly useful because it encapsulates a microservice in a container. This isolation ensures that each microservice can have its dependencies managed independently. This makes it easier to scale, update and maintain individual microservices of the system.

5.2 Microservices descriptions

Product Microservice

The product microservice is the core component that manages product data. It offers Create, Read, Update, and Delete (CRUD) operations. This microservice uses a NoSQL database MongoDB, ensuring high performance and scalability for product data.

Functionalities

- CRUD operations, manages the creation, reading, updating and deletion of product records within the database.
- Provides the corresponding endpoints to perform the CRUD operations.
- When a new product is created, it sends the product to a queue to communicate to other services that a new product has been added.
- When a product is deleted, it sends the ID of the product to a queue to communicate to other services that the product has been deleted.

Api contract product microservice

GET – Get all the products

This service is responsible of returning all the available products stored in the MongoDB database. The collection to retrieve these products is named *product*.

- Request URL

```
http://<server>:8080/products
```

- Response body (JSON)

List of the next table:

Param Name	Example	Data type	Required
id	65f0855584e9be567011327c	String	Yes
name	Ocean Breeze T-Shirt	String	Yes
category	T-Shirt	String	Yes
material	cotton	String	Yes
Color	Blue	String	Yes
seasonality	Summer	String	Yes
occasion	Casula	String	Yes
gender	Women	String	Yes
priceRange	Mid-Range	String	Yes
price	30	Integer	Yes
fit	Regular	String	Yes
pattern	Fine lines	String	Yes
sustainability	Eco-Friendly	String	Yes
tags	["eco-friendly", "summer", "casual", "breathable cotton"]	List<Sting>	No

-HTTP status code response

HTTP status code	Reason
200	Response ok
204	No content
500	Internal error

GET - Get product by ID

This service is responsible for retrieving a specific product from the collection of MongoDB named *product*. The param used is the ID of the product.

- Request URL

```
http://<server>:8080/products/{ID}
```

- Response body (JSON)

Param Name	Example	Data type	Required
id	65f0855584e9be567011327c	String	Yes
name	Ocean Breeze T-Shirt	String	Yes
category	T-Shirt	String	Yes
material	cotton	String	Yes
Color	Blue	String	Yes
seasonality	Summer	String	Yes
occasion	Casula	String	Yes
gender	Women	String	Yes
priceRange	Mid-Range	String	Yes
price	30	Integer	Yes
fit	Regular	String	Yes
pattern	Fine lines	String	Yes
sustainability	Eco-Friendly	String	Yes
tags	["eco-friendly", "summer", "casual", "breathable cotton"]	List<Sting>	No

-HTTP status code response

HTTP status code	Reason
200	Response ok
404	Not found
500	Internal error

- Exceptional response

In case that no products exist with the given ID, the response of the service will be null with HTTP status 404 not found.

POST – Create a new product

This service is in charge of creating a product object and saving it in the MongoDB collection named *product*. After saving the object in the database, the service will publish the new product object in a queue named: CREATED_PRODUCT. The topic is provided by the Kafka broker.

The service will validate the incoming data, if any fields in the request body do not match with the corresponding definition of the filed, the service will throw an exception.

- Request URL

```
http://<server>:8080/products
```

- Request param (JSON string)

Param Name	Example	Data type	Required
name	Ocean Breeze T-Shirt	String	Yes
category	T-Shirt	String	Yes
material	cotton	String	Yes
Color	Blue	String	Yes
seasonality	Summer	String	Yes
occasion	Casula	String	Yes
gender	Women	String	Yes
priceRange	Mid-Range	String	Yes
price	30	Integer	Yes
fit	Regular	String	Yes
pattern	Fine lines	String	Yes
sustainability	Eco-Friendly	String	Yes
tags	["eco-friendly", "summer", "casual", "breathable cotton"]	List<Sting>	No

- Request body (JSON)

Param Name	Example	Data type	Required
image	Image.jpg	Multipat/form-data	Yes

- Response body (JSON)

Param Name	Example	Data type	Required
id	65f0855584e9be567011327c	String	Yes
name	Ocean Breeze T-Shirt	String	Yes
category	T-Shirt	String	Yes
material	cotton	String	Yes
Color	Blue	String	Yes
seasonality	Summer	String	Yes
occasion	Casula	String	Yes
gender	Women	String	Yes
priceRange	Mid-Range	String	Yes
price	30	Integer	Yes
fit	Regular	String	Yes
pattern	Fine lines	String	Yes
sustainability	Eco-Friendly	String	Yes
tags	["eco-friendly", "summer", "casual", "breathable cotton"]	List<Sting>	No

-HTTP status code response

HTTP status code	Reason
201	Created
500	Internal error

DELETE - Delete product by ID

This service is in charge of deleting an existing product in the MongoDB collection named *product*. If the product is correctly deleted, the service will publish the productId in the queue named: DELETED_PRODUCT. The topic is provided by the Kafka broker.

- Request URL

```
http://<server>:8080/products/{ID}
```

- Response

No response will be provided, only an HTTP status code.

-HTTP status code response

HTTP status code	Reason
200	Response ok
404	Not found
500	Internal error

- Exceptional response

In case that no products exist with the given ID, the response of the service will be null with HTTP status 404 not found.

Data model

The data is stored in a document-based NoSQL database, in this case MongoDB. The data model is in the form of documents within a collection. There is only one collection for this product-microservice, named *product*. To prevent data inconsistency, it is important to point out that the database is only available for the product-microservice and it is forbidden to share the connection with other microservices.

Bear in mind that NoSQL databases lack constrains and restrictions. Each document in the 'product' collection represents a product with the following schema:

- **_id**: Default unique identifier of the product.
- **category**: String (Upper case) - Category of the product.
- **color**: String (Upper case) - Color of the product.
- **fit**: String (Upper case) - Fit of the product.
- **gender**: String (Upper case) - Target gender for the product.
- **material**: String (Upper case) - The material from which the product is made.
- **name**: String (Upper case) - The name of the product.
- **occasion**: String (Upper case) - The suitable occasion for the product.
- **pattern**: String (Upper case) - The pattern on the product if any.
- **price**: Integer - The selling price of the product. (Numeric values are not case-sensitive)
- **priceRange**: String (Upper case) - Representation of the price range.

- **seasonality**: String (Upper case) - The season of the product.
- **sustainability**: String (Upper case) - Indicates the sustainable attributes.
- **tags**: Array of Strings (Upper case) - Additional keywords or tags associated with the product.

Example of the product document:

```
{
  "_id": " 65f0855584e9be567011327c",
  "category": "T-SHIRT",
  "color": "BLUE",
  "fit": "REGULAR",
  "gender": "WOMEN",
  "material": "COTTON",
  "name": "OCEAN BREEZE T-SHIRT",
  "occasion": "CASUAL",
  "pattern": "FINE LINES",
  "price": 30,
  "priceRange": "MID-RANGE",
  "seasonality": "SUMMER",
  "sustainability": "ECO-FRIENDLY",
  "tags": ["ECO-FRIENDLY", "SUMMER", "CASUAL", "BREATHABLE
COTTON"]
}
```

Code 7: example of product document.

Index consideration: MongoDB automatically generates an index on the _id field, but in case that there is any service that needs to search for any field of the product collection, the MongoDB administrator must add a new index to improve the performance of the queries.

Sequence diagrams

Recommendation Microservice

The recommendation microservice is the implementation of a content-based product recommendation system explained in Chapter 4. The system will recommend products based on its shared characteristics.

Functionalities

- The microservice listens to a Kafka topic in order to receive new products and perform a data transformation from document data in NoSQL to the corresponding graph database.

- The microservice listens to a Kafka topic in order to receive products that have been deleted in order to delete the product in the graph database.
- The microservice exposes a service to fetch all the products in the graph database
- The microservice exposes a service to get a similar product by a *productId*.

Recommendation microservice consumer contract

Create product

The service the product creates is the responsible one for integrating new products into the content-based recommendation system. This process consumes messages from a Kafka topic named: CREATED_PRODUCT. When a message arrives, the service transforms the data into a graph database.

- Kafka message consumption

The microservice subscribes to the CREATED_PRODUCT Kafka topic. The messages are consumed by a Kafka consumer group named: recommendation-group.

Upon receiving a message, the service deserializes the JSON formatted message into a KafkaMessage. The JSON message follows the next schema:

Param Name	Example	Data type	Required
id	65f0855584e9be567011327c	String	Yes
name	OCEAN BREEZE T-SHIRT	String	Yes
category	T-SHIRT	String	Yes
material	COTTON	String	Yes
Color	BLUE	String	Yes
seasonality	SUMMER	String	Yes
occasion	CASUAL	String	Yes
gender	WOMEN	String	Yes
priceRange	MID-RANGE	String	Yes
fit	REGULAR	String	Yes
pattern	FINE LINES	String	Yes
sustainability	ECO-FRIENDLY	String	Yes

The ID which identifies the product must be the same in MongoDB and Neo4j. It allows us to identify the product in both databases.

- Data transformation

The next step of the service is to transform the *KafkaMessage* into the product representation in the graph database.

1. The service creates the product node with the ID and the name of the *KafkaMessage*.

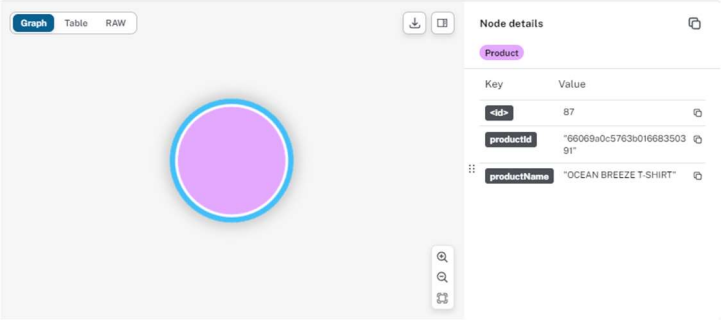

Figure 23: Example of product node with productID and productName.

2. The service creates all the characteristic nodes that are: [Category, Material, Color, Seasonality, Occasion, Gender, PriceRange, Fit, Pattern, Sustainability,]

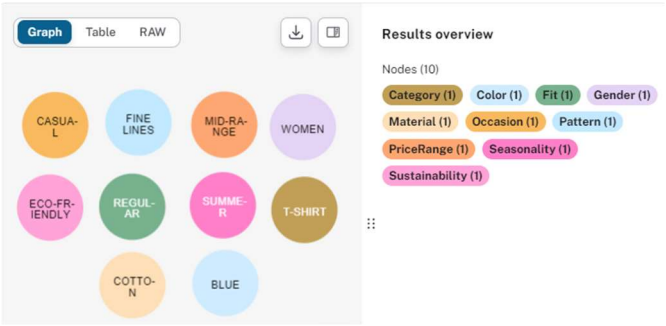

Figure 24: Example of characteristic nodes.

If any characteristic node is already created, the node will not create another one. An example of a Cypher query to create a characteristic node is:

```
MERGE (:Category {name: $categoryName})
```

Code 8: Example of Cypher query to create the characteristic node Category.

3. After all the characteristic nodes are created, the last step is to create the relationships from the product to each characteristic node. Each characteristic node also has a weight property that is determined by the recommendation teams Annex 7: Characteristics weight. An example of a Cypher query to create a relationship between a product and a characteristic node is:

```
MATCH (p:Product {productId: $productId}),
      (c:Category {name: $categoryName})
MERGE (p)-[:BELONGS_TO {weight: $weight}]->(c)
```

Code 9: Example of Cypher query to create a relationship between product and Category node.

The result of the product represented in the graph database is:

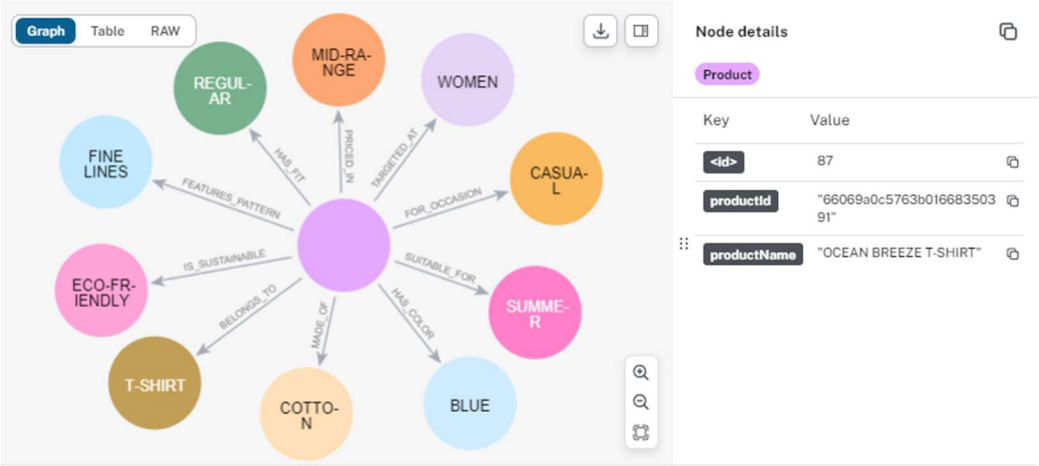

Figure 25: Result of a new product in Neo4j graph database.

Delete product

The deleted product service is the responsible for deleting the products in the content-based recommendation system. This process consumes message form a Kafka topic named: DELETED_PRODUCT. When a message arrives, the service will delete the product form the graph database.

- Kafka message consumption

The microservice subscribes to the DELETED_PRODUCT Kafka topic. The messages are consumed by a Kafka consumer group named: recommendation-group.

Upon receiving a message, the service deserializes a string-formatted message. This string corresponds to the *productId* that must be deleted in the graph database.

- Delete product in the database

The Cypher query that the service will perform is:

```
MATCH (p:Product {productId: $productId}) DETACH DELETE p
```

Code 10: Cypher query to delete a product node.

This Cypher query will delete the product node, the relationships that the product node has and any characteristic node that does not have other relationships with other nodes.

Api contract recommendation microservice

GET - Get all products

This service is responsible of retrieving all the products form the Neo4j graph database. The instance of the database is named: Instance01.

- Request URL

```
http://<server>:8081/products
```

- Response body (JSON)

List of:

Param Name	Example	Data type	Required
productName	OCEAN BREEZE T-SHIRT	String	Yes
productId	6606b40f5763b01668350393	String	Yes

-HTTP status code response

HTTP status code	Reason
200	Response ok
204	Not content
500	Internal error

- Exceptional response

In case no product exists, the service will be return null with http status 204 not content.

GET - Get similar products by ID

This service is in charge of calculating the similarity of all the products in the Neo4j graph database compared to the product received in the request. The service will return an ordered list of all products with its similarity regarding the product that the service received in the request. The order in the list is descended from most to least similar.

The Cypher query performed by the service is already described in the section: Query to get similar product.

- Request URL

```
http://<server>:8081/recommendation/{productId}
```

- Response body (JSON)

List of:

Param Name	Example	Data type	Required
productName	OCEAN BREEZE T-SHIRT	String	Yes
productId	6606b40f5763b01668350393	String	Yes
similarityScore	30	Integer	Yes

-HTTP status code response

HTTP status code	Reason
200	Response ok

204	Not content
500	Internal error

- Exceptional response

In case that the incoming product does not exist the service will return an empty list with HTTP status code 200.

Data model

The data model for the recommendation system is designed to efficiently represent products and their characteristics within the Neo4j graph database. The primary entity is the product node, which is connected through relationships to its characteristic nodes.

Nodes:

- **Product Node**: It represents an individual product. It has the properties *productId* and *productName*.

- **Characteristic Nodes**: These nodes represent the characteristics of any product. The types of characteristics nodes are:

    - Category: Represents the category of the product.
    - Material: Represents the material of the product.
    - Color: Represents the color of the product.
    - Seasonality: Represents the season for when the product is suitable.
    - Occasion: Represents the occasion for which the product is suitable.
    - Gender: Represents the target gender of the product.
    - PriceRange: Represents the price range of the product.
    - Fit: Represents the fit of the product.
    - Pattern: Represents the pattern of the product.
    - Sustainability: Represents the sustainable attributes.

- **Relationships**: These connects the product node to characteristic nodes and are labelled with weight. This weight is used to calculate the strength of the relationship between the product and the characteristic. The types of relationships are:

    - BELONGS_TO: Links a product node to a Category node.
    - MADE_OF: Links a product node to a Material node.
    - HAS_COLOR: Links a product node to a Color node.
    - SUITABLE_FOR: Links a product node to a Seasonality node.
    - FOR_OCCASION: Links a product node to an Occasion node.
    - TARGETED_AT: Links a product node to a Gender node.
    - PRICED_IN: Links a product node to a PriceRange node.
    - HAS_FIT: Links a product node to a Fit node.
    - FEATURES_PATTERN: Links a product node to a Pattern node.
    - IS_SUSTAINABLE: Links a product node to a Sustainability node.

Index consideration: It is not necessary to create an index for the graph database.

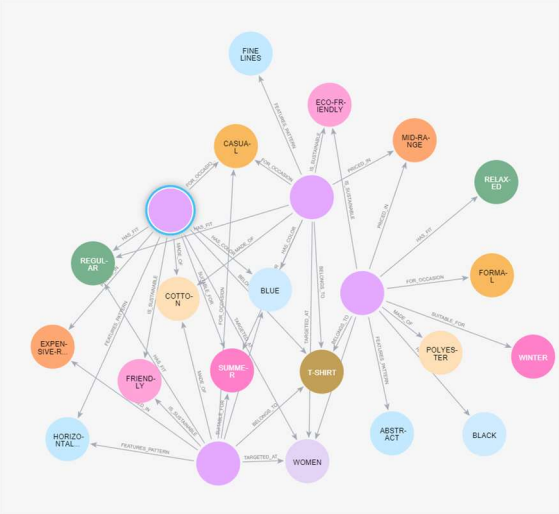

Figure 26: Example of the result of the Neo4j graph database.

5.3 Queues documentation

There are two queues in the system provided by the Kafka broker. The queues provide asynchronous communication between microservices. With this approach we decouple the product-microservice and the recommendation-microservice.

CREATED_PRODUCT

This queue is designed to handle new product creation events. When the product-microservice creates a new product, it publishes a message to this topic. The details of the topic are:

- **Consumer**: The consumer group that are allowed to consume the topic is named: recommendation-microservice.
- **Message key**: In all scenarios, the message key will be null. If we face problems with the order of the message, we can use this feature.
- **Message value**: The message value.
- **Partition strategy**: The topic only has one partition. If we face scalability issues it allows to increment the number of partitions.
- **Retention policy**: The expiration time of the message is one week.

DELETED_PRODUCT

This queue is designed to delete products. When the product-microservice deleted a product, it publishes a message to this topic. The details of the topics are:

- **Consumer**: The consumer group that are allowed to consume the topic are named: recommendation-microservice.
- **Message key**: In all scenarios, the message key will be null. If we face problems with the order of the message, we can use this feature.

- **Message value**: the value is a String with the *productId* to delete in the graph database.
- **Partition strategy**: The topic only has one partition. If we face scalability issues it allows to increment the number of partitions.
- **Retention policy**: The expiration time of the message is one week.

5.4 Frontend application.

The frontend application consists of a Single Page Application (SPA) built with the framework Angular. This application uses the services provided by the product-microservice and the recommendation-microservice. The application consists of two mains views:

Product list view

This is the principal view of the application. The functionality is to show the user all the products available. The page displays a grid of product cards showing the products' title, a brief description of the products, the image of the products and the products' tags.

- Path

```
http://<server>:4200
```

- Interactions

View Details: If the user clicks on a product card the page sends the user to a product detail view for that product card.

Before the view is rendered, the view fetches a list of products from the backend with the API endpoint: */products*. This service is provided by the product-microservice.

Product detail view

When the user clicks on any product card in the main page or in the recommendation section, the user will be redirected to this view. The view consists of two sections.

- Detail section: Provides all the details and characteristics of the product
- Recommendation section: Provides a grid of related product cards, indicating how closely they match the characteristics of the viewed product.

- Path

```
http://<server>:4200/product-detail/:productId
```

- Interactions

View Details: If the user clicks on a related product card, the page sends the user to a product detail page for that product card.

Before the view is rendered, the view fetches the detailed information of the specific product from the backend with the API endpoint: *GET /products/{ID}*. This service is provided by the product-microservice.

The view also fetches the related or recommended product based on the viewed product. It is from the backend with the API endpoint: */recommendation/{productId}*. This service is provided by the recommendation-microservice.

Delete product: If the user clicks on delete product button, the view calls to the endpoint: *DELETE products/{ID}*. This service is provided by the product-microservice. After that, the user will be redirected to the principal view of the application.

Add product view

When the user clicks on the add button in the product list view, the user will be redirected to the add product view. The view displays a form with an input for each characteristic of that has the product.

- Path

```
http://<server>:4200/add-product
```

- Interactions

If the user fills all the inputs of the from, the Add Product button will be shown to the user to perform the creation of the product. When the button is pressed, the user will be redirected to the product list view. Moreover, the user will be informed through a pop-up that the product was created successfully.

Chapter 6: Observability system

In this chapter we will see the implementation of the observability system that allows us to observe the recommendation system we already built. The intentionality of this observability system is to apply the theory of the observability discussed in Chapter 3 and use one of the modern observability stacks that has been gaining a lot of traction in recent years.

System overview

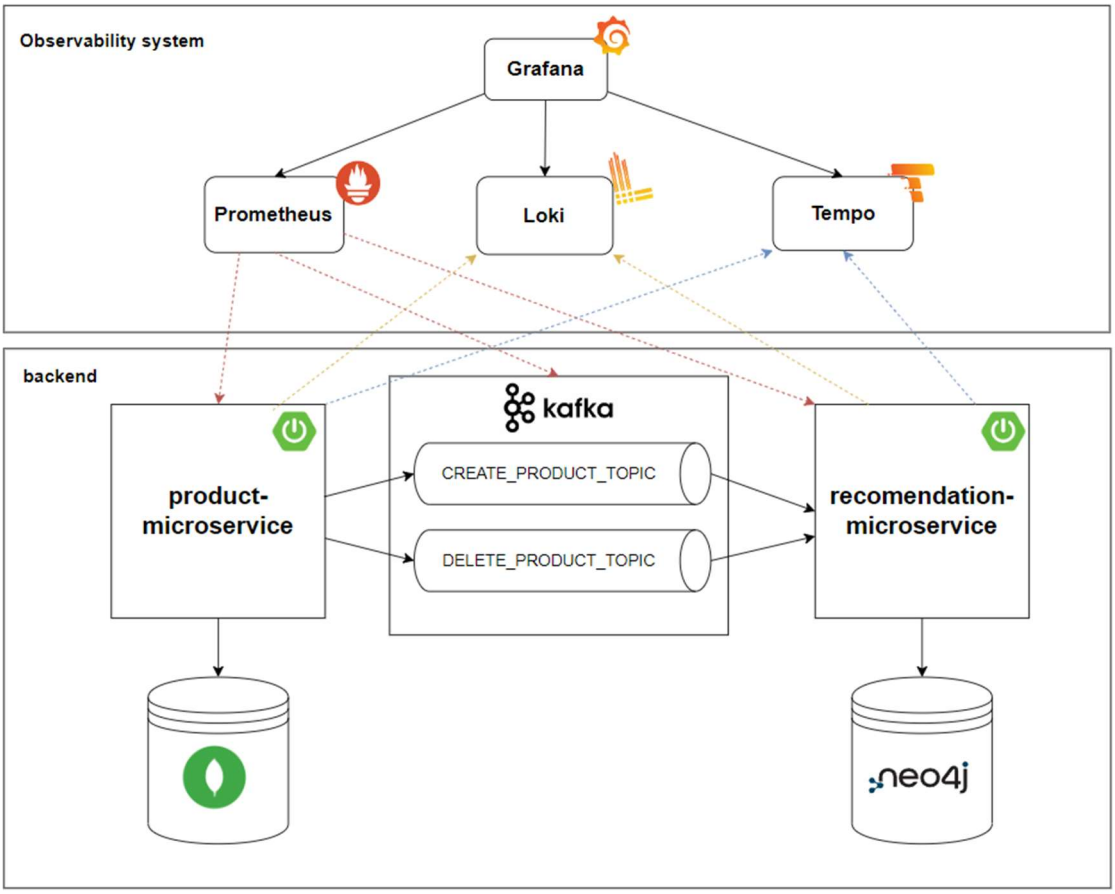

Figure 27: Diagram of the observability system and the backend.

The observability system allows us, at any interval of time, to understand the inner work of the recommendation system. Metrics such as hardware availability or hardware utilization, which are the responsibility of systems operators, are not useful to us. We want to focus on observability insights that help developers manage, understand, and troubleshoot the microservice architecture.

The observability system gathers the next observability data:

- **Metrics:** The system gathers all the metrics related to the microservices. Moreover, the system gathers all the metrics related to the microservices' dependencies. These dependencies are the databases and the message broker.
- **Logs:** All the logs generated by the microservice are gathered by the system.
- **Traces:** All the traces generated by the events that hit the recommendation system are gathered by the system.
- **Health check:** All the microservices expose a service that returns the health of the microservice itself.

Moreover, the system displays all this information in one single tool. This minimizes the cognitive load on the system administrator and reduces the complexity of managing multiple observability solutions.

6.1 Technologies justification

The following open-source technologies were used to create the entire observability system:

- **Prometheus**: A popular metric collection tool [56]. It excels in reliability and the ability to handle hight-cardinality data. Moreover, Prometheus has a huge community and is well aligned with the microservice approach.
- **Loki**: An aggregation system which allows us to collect logs in an easy and cost-efficient way [32].
- **Tempo**: A system that ensures high scalability for traces with minimal operational effort [57].
- **Grafana**: It offers an easy integration and compatibility with Prometheus, Loki, and Tempo. It is ideal for a cohesive observability experience [58].

6.2 Metrics

The system collects all the metrics with Prometheus. It scrapes all the metric data through one specific endpoint at a 2s interval. Each microservice exposes all the metric data related to the microservice and its dependencies.

The following endpoints allow Prometheus to scrape the metrics data are:

- Product microservice metric endpoint:

```
http://<server>:8080/actuator/prometheus
```

- Recommendation microservice metric endpoint:

```
http://<server>:8081/actuator/prometheus
```

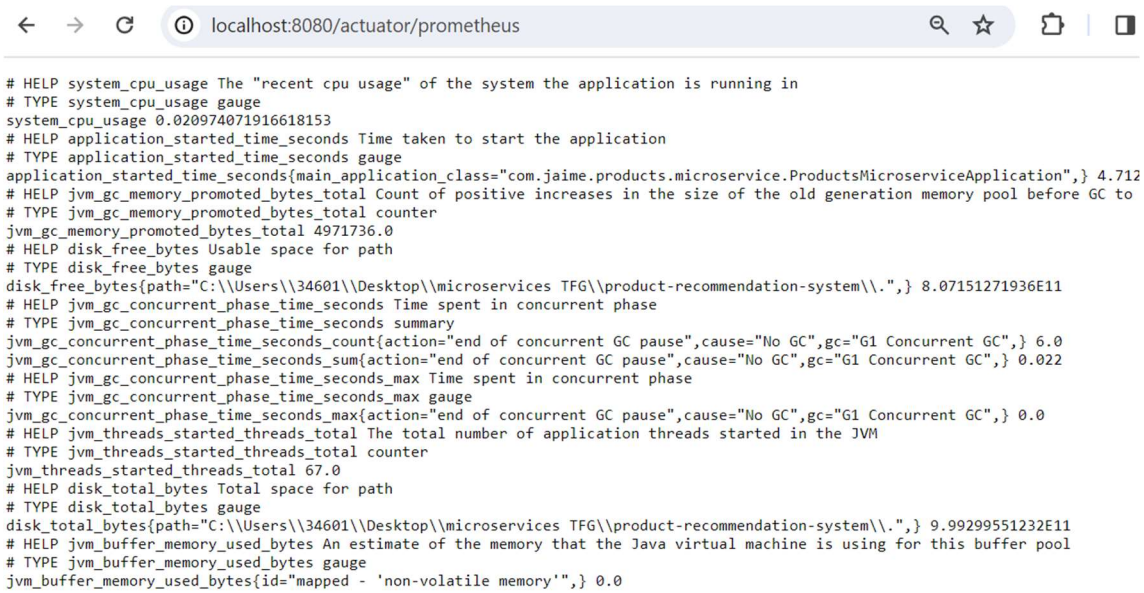

Figure 28: Example of data expose to Prometheus.

In Figure 28, we see part of the metrics exposed by the product-microservice. For example, the data shows metrics such as the current CPU usage by the microservice or the time that the application took to be ready.

The metrics that are used to monitor the system are divided into three categories:

HTTP metrics

For both microservices, the HTTP metrics are:

- **Total request**: This counts the total number of HTTP requests that our microservice has processed.
- **Total request count by service (end-point):** It shows the distribution of requests among the services that expose the microservice.
- **Request per second:** This metric provides the rate between requests / second. This metric allows us to understand the performance under load.
- **Request average duration:** This shows the average duration of the requests in a specific interval of time.
- **HTTP request max duration (second)**: This metric shows the longest request times. This metric allows us to see the worst request time during a specific interval of time.
- **Total 2xx requests**: This metric shows the successful response of the microservice. This metric aggregates all the 2xx response, such as 200, 201 and so on.
- **Total 4xx request**: This metric shows the error responses of the microservice. This metric aggregates all the 4xx response, such as 400, 401, and so on.

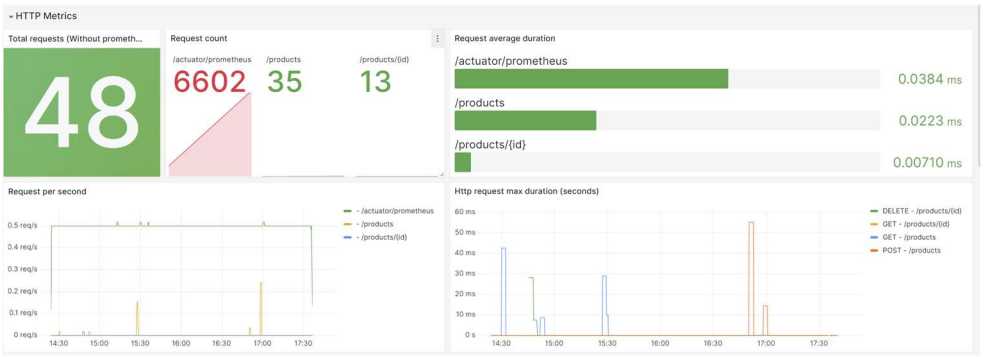

Figure 29: HTTP metrics about recommendation system Grafana.

Spring data

For both microservices, the metric related to the performance of the database is:

- **Max. duration of repository request:** This metric shows the maximum response time for each operation to the database.

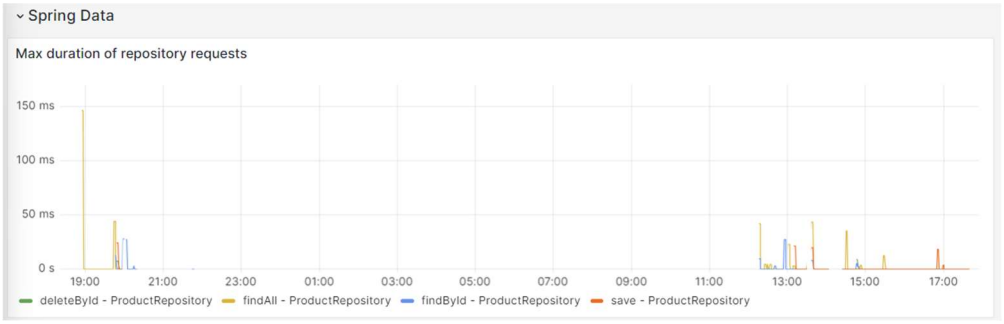

Figure 30: MongoDB database max respond time duration metric of product-microservice.

Figure 31: Neo4j database max respond time duration metric of recommendation-microservice.

Kafka data

For both microservices, the metric used to measure the messages on the topics is:

- **The total number of records sent for a topic (per minute)**: This metric represents the total number of records sent to a topic within one-minute intervals. It allows us to measure the volume of data handled by the system.

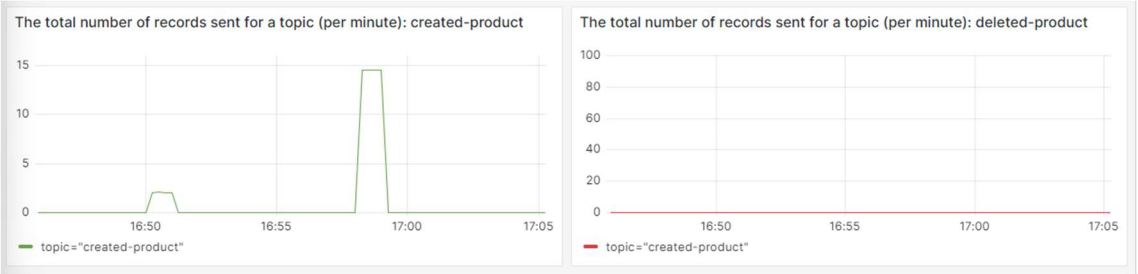

Figure 32: Kafka total number of message/minutes on Kafka topics.

For the recommendation-microservice, there is one more metric which is:

- **Max. Kafka listener**: This metric represents the maximum time taken by the Kafka listeners for processing a message.

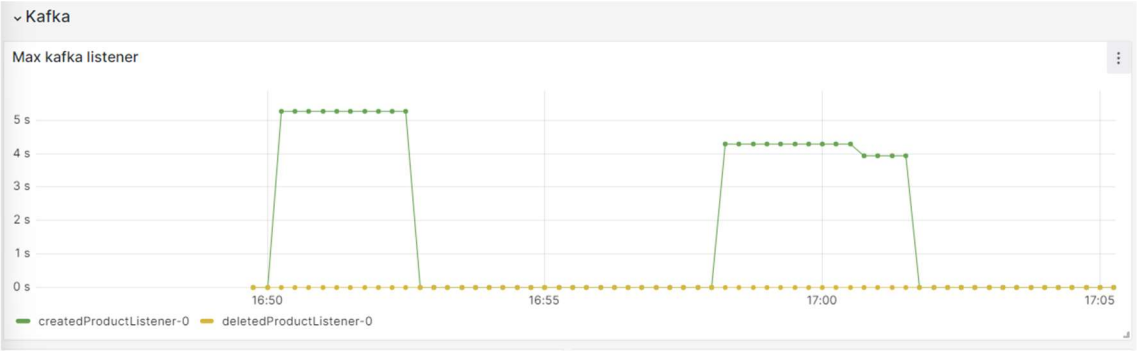

Figure 33: Total max of time taken by the system to process a message.

6.3 Logs

The microservices pull all the generated logs to a log aggregation system, in this case Loki. It does not index the content of the logs, only the metadata of the logs. This makes our system cost-effective due to storage logs are the most expensive part of any observability system.

The event log structure for the overall system has the following fields:

- **Timestamp**: The moment that the log was created.
- **Log level**: The level of the log, which can be:

    - INFO: General information about the application.
    - DEBUG: Detail diagnostic information.
    - WARNING: Indicates a potential issue.
    - ERROR: Indicates an error in the application.
- **ProcesID**: The identifier of the process that creates the log.
- **Thread**: The thread involved in the operation.
- **Source**: The name of the microservice that created the log.
- **TraceId**: Identifier of the global event.
- **SpanId**: Identifier for a single operation.
- **Message**: the content of the log.

An example of an event in the product-microservice is when a client requests to delete an existing product.

```
> 2024-04-01T14:45:05.766+02:00  INFO 17564 --- [http-nio-8080-exec-6] [produc
  t-microservice,660aac517ae4ef87c17b10f5f7feb598,c17b10f5f7feb598]c.j.p.micro
  service.aop.LoggingAspect     : product-microservice: ProductController.dele
  teProductById -> DataFunction(className=ProductController, functionName=dele
  teProductById, args={Arg_1=6606c2d85763b01668350397})
> 2024-04-01T14:45:05.767+02:00  INFO 17564 --- [http-nio-8080-exec-6] [produc
  t-microservice,660aac517ae4ef87c17b10f5f7feb598,c17b10f5f7feb598]c.j.p.micro
  service.aop.LoggingAspect     : product-microservice: ProductServiceImpl.del
  eteProductById -> DataFunction(className=ProductServiceImpl, functionName=de
  leteProductById, args={Arg_1=6606c2d85763b01668350397})
> 2024-04-01T14:45:05.770+02:00 DEBUG 17564 --- [http-nio-8080-exec-6] [produc
  t-microservice,660aac517ae4ef87c17b10f5f7feb598,c17b10f5f7feb598]o.s.data.mo
  ngodb.core.MongoTemplate      : findOne using query: { "id" : "6606c2d85763b
  01668350397"} fields: Document{{}} for class: class com.jaime.products.micro
  service.dao.model.Product in collection: product
> 2024-04-01T14:45:05.778+02:00 DEBUG 17564 --- [http-nio-8080-exec-6] [produc
  t-microservice,660aac517ae4ef87c17b10f5f7feb598,c17b10f5f7feb598]o.s.data.mo
  ngodb.core.MongoTemplate      : Remove using query: { "_id" : { "$oid" : "66
  06c2d85763b01668350397"}} in collection: product.
> 2024-04-01T14:45:05.783+02:00  INFO 17564 --- [http-nio-8080-exec-6] [produc
  t-microservice,660aac517ae4ef87c17b10f5f7feb598,c17b10f5f7feb598]c.j.p.m.ser
  vice.impl.ProductServiceImpl  : ProductService: Product with id: 6606c2d8576
  3b01668350397 deleted
> 2024-04-01T14:45:05.783+02:00  INFO 17564 --- [http-nio-8080-exec-6] [produc
  t-microservice,660aac517ae4ef87c17b10f5f7feb598,c17b10f5f7feb598]c.j.p.micro
  service.aop.LoggingAspect     : product-microservice: KafkaProducer.sendToDe
  leteProduct -> DataFunction(className=KafkaProducer, functionName=sendToDele
  teProduct, args={Arg_1=6606c2d85763b01668350397})
```

Figure 34: Example of an event to delete a product.

6.4 Tracing

The tracing part of the implemented system is divided into 4 main parts:

- **Instrumentation**: Each microservice is instrumented to generate trace data. In this case, we use the Spring Cloud Sleuth library [59] to add to all the events that happen in the microservices the following pattern in the logs:

  ```
  [${spring.application.name:},%X{traceId:},%X{spanId:-}]
  ```

  This pattern allows us to correlate logs with the events.

- **Propagation**: With Spring Cloud Sleuth, trace context propagation is handled automatically. This is handy when microservices communicate among themselves.
- **Storage:** All the microservices push the trace data to tempo which is responsible for storing the traces.
- **Analysis and visualization**: The analysis of the traces can be done with Loki in Grafana searching for the *traceIds* and *spanIds* and looking for the logs. Or with the well-integrated Tempo in Grafana.

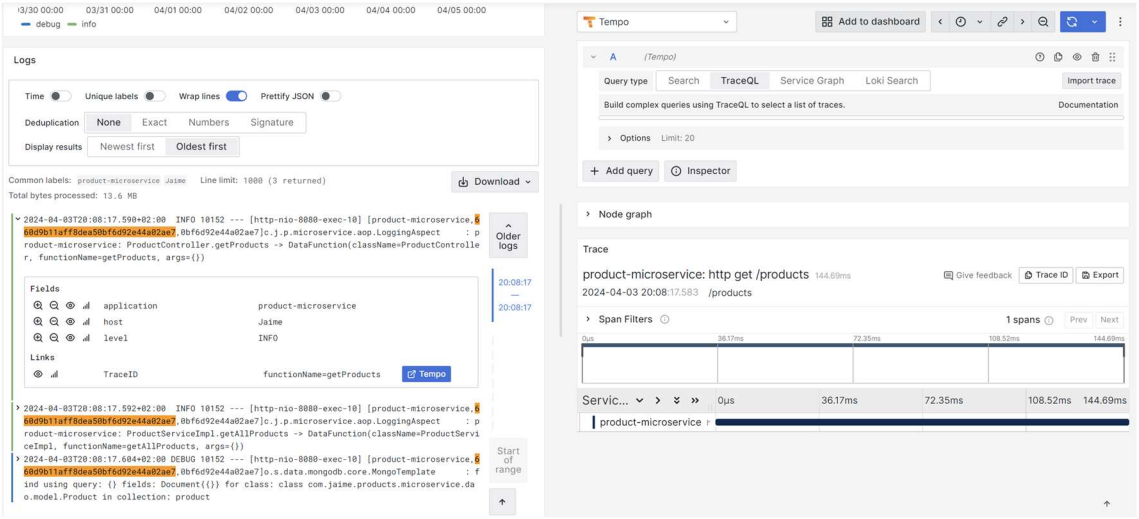

Figure 35: Example of trace Loki in the left and tempo in the right with the same trace.

6.5 Health check

As discussed in Chapter 3, it is important to know the availability and the health of the microservices and their dependencies. We implement this monitoring using the Spring boot Actuator library [37].

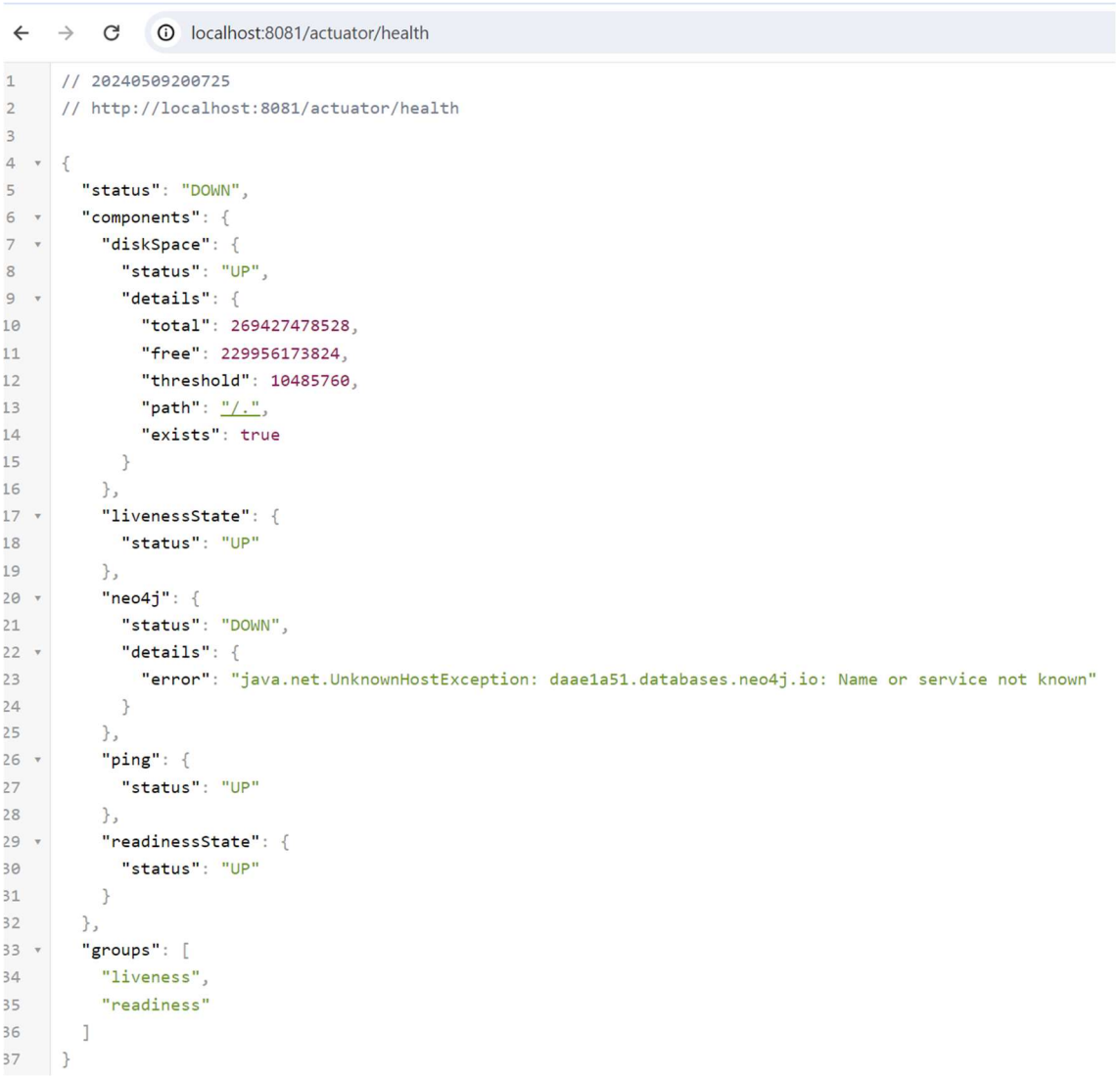

Figure 36: Health check recommendation-microservice.

In the Figure 36, we can see a response from the health check endpoint of the product-microservice. The response indicates that while the microservice itself is operational (ping: "UP") and ready to handle traffic (readinessState: "UP"), the overall status of the microservice is "DOWN". This status is due to the Neo4j database dependency being down.

Chapter 7: Testing and validating the recommendation system

To validate and demonstrate the main services of the recommendation and observability system, we are going to do a battery of test cases. These test cases will help in assessing the systems and show how it works.

7.1 Test case 1: New product creation

Objective

The objective is to verify that a new product is successfully created in the product-microservice and recommendation-microservice. We have to perform the creation of the product in the *add-view* of the Angular application.

Precondition

No specific precondition is required for this test case.

Result

First we populated the fields with the product characteristics that we want to add. After that we press the *Add product* button.

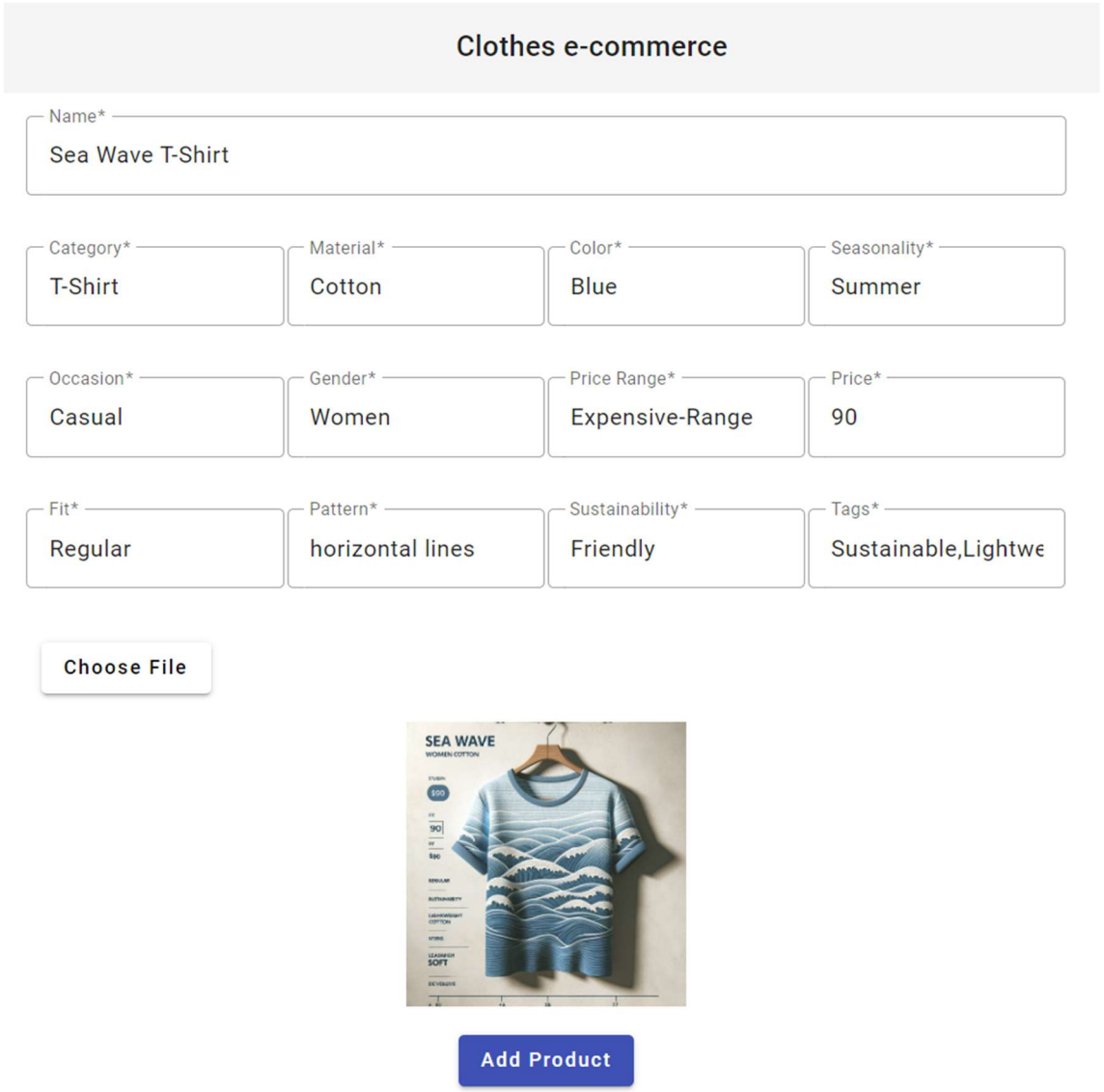

Figure 37: add-view Angular.

The product-microservice processes the request and creates the new product.

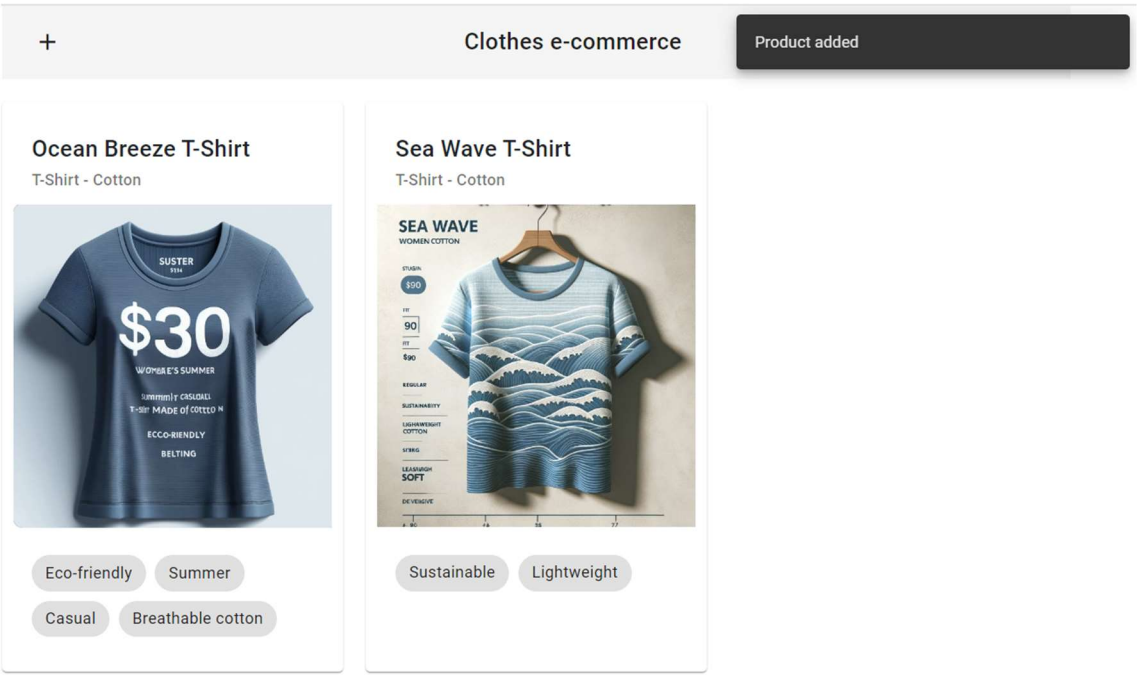

Figure 38: Product list view, Angular app.

In Figure 38, we see that the product is listed, moreover we can see the pop-up that confirms the creation of the product.

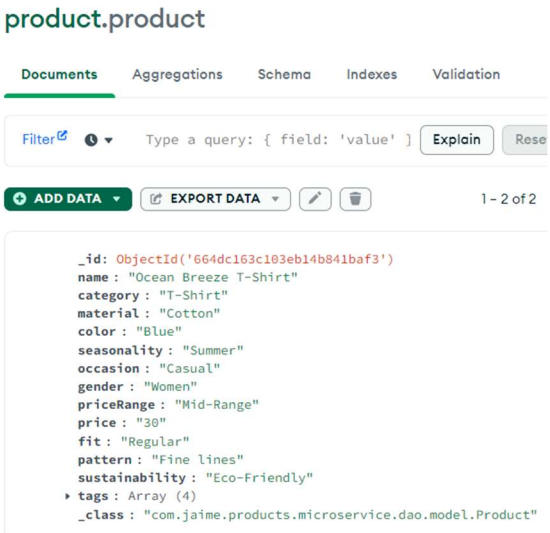

Figure 39: Product collection MongoDB.

if we look at the MongoDB we verify that the product was successfully saved.

After the product-microservice saved the product, it publishes a message with the new product to the Kafka topic CREATED_PRODUCT. This is the part of the asynchronous communication between product-microservice and recommendation-microservice.

Figure 40: Kafka metrics recommendation-microservice Grafana.

We can confirm that if we see the Kafka metrics in Figure 40. These metrics show that one message arrived to the CREATED_TOPIC and it took the recommendation-microservice 4.07 seconds to process the message.

If we look at the logs from the overall system and search by the identifier of the product, we can confirm the path that has taken the request. Starting from product-microservice and ending in the recommendation-microservice.

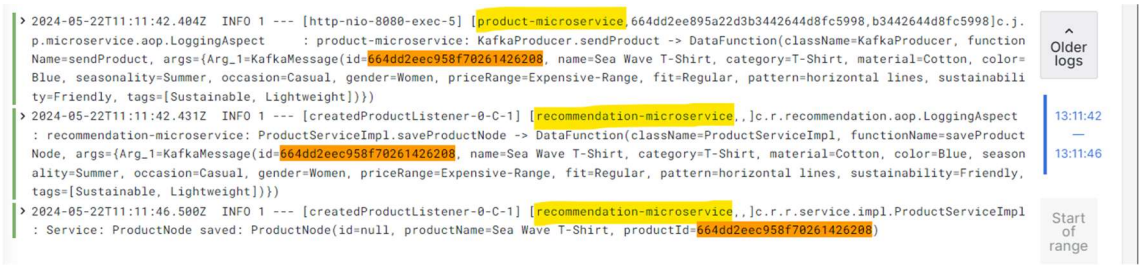

Figure 41: Loki search by productID.

The final verification confirms that the recommendation-microservice has successfully processed the Kafka message and created in Neo4j database all the nodes and the relationships which represent the new product.

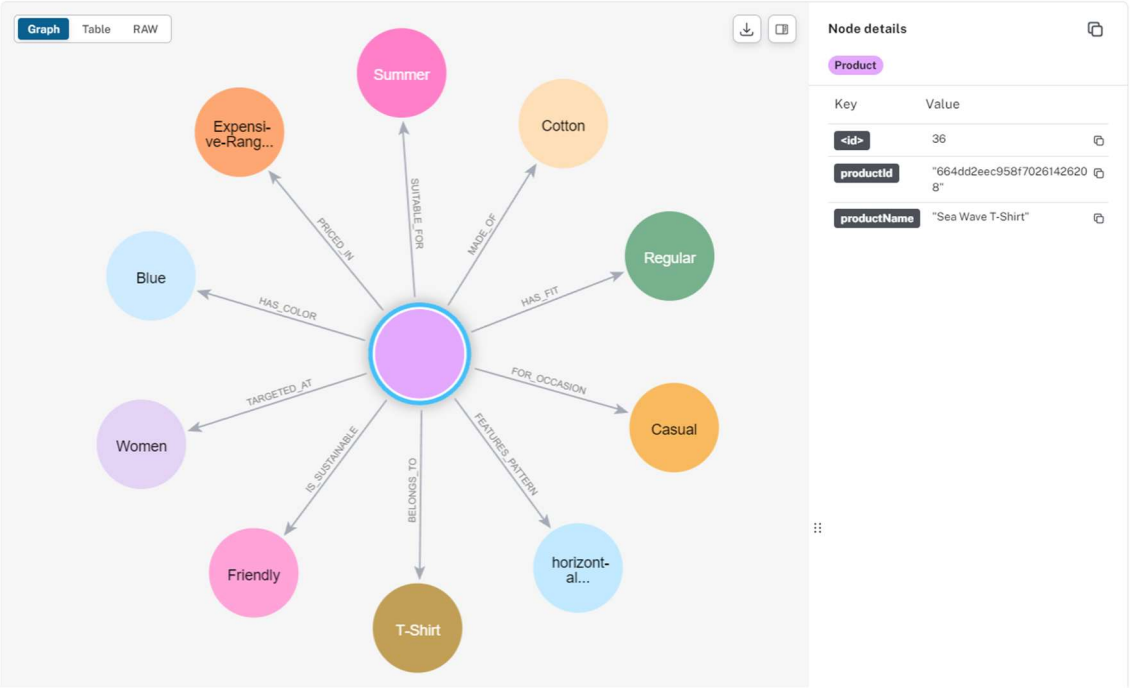

Figure 42: The new product created in Neo4j database.

7.2 Test case 2: List product in the frontend application

Objective

The objective is to verify in the frontend application that there is a visual interface that fetches all the products created and shows the products' title, a brief description of the products, the image of the products and the product's tags.

Precondition

The system has already been created 6 clothes products, 3 t-shirts and 3 jeans.

Result

If we visit the URL: *http://localhost:4200/*, we can see an Angular application that shows a list of clothes products and shows a description of each product.

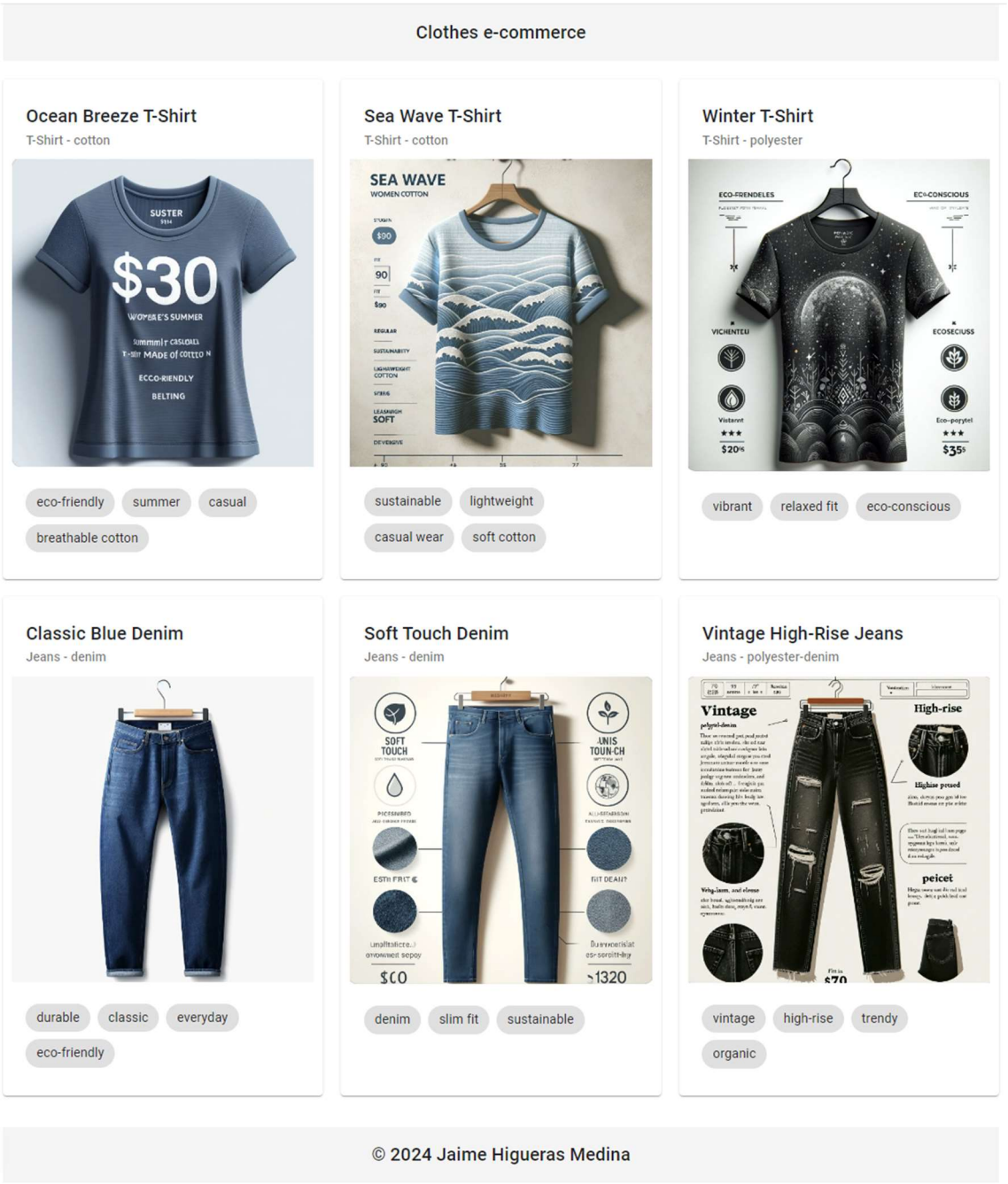

Figure 43: Product list in Angular application.

Internally The Angular application made a GET request to the product-microservice in the endpoint */products*. To fetch a list of all the products that already exist in the system.

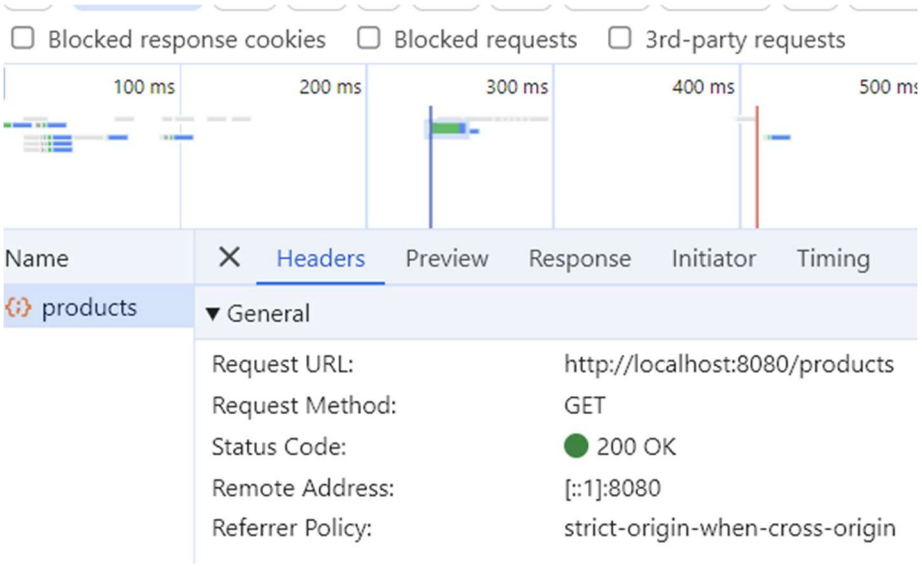

Figure 44: HTTP request to /products.

If we loo at the metrics the GET request to the end-point /products of the product-microservice. The metrics indicate that the request took 12.4 ms to respond and the response was successful.

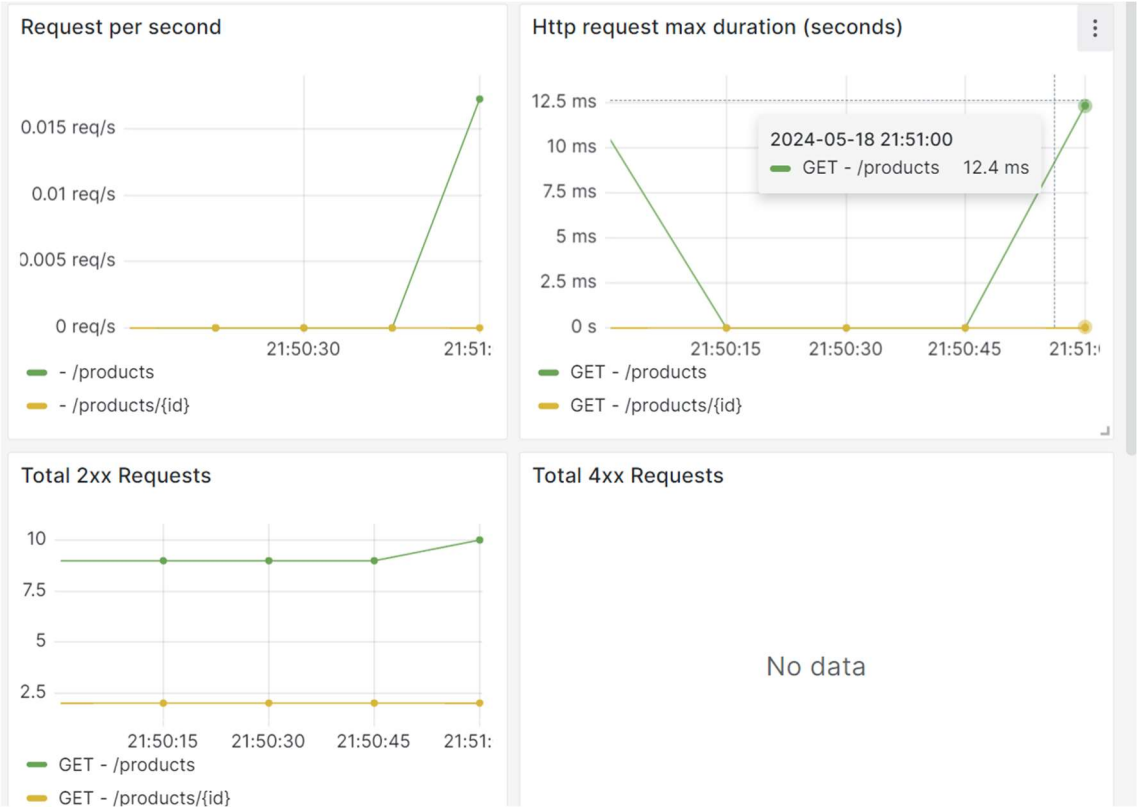

Figure 45: HTTP metrics form the product-microservice.

7.3 Test case 3: Showing the product detail and a recommendation list of similar products in the frontend application

Objective

The objective is to verify in the frontend application that there is a visual interface that shows all the details of one specific product. Moreover, below the product information there is a product list with similar products.

Precondition

The system has already been created 6 clothes products, 3 t-shirts and 3 jeans.

Result

If we visit the URL: *http://localhost:4200/* and click on a specific product the Angular application takes us to *http://localhost:4200/product-detail/productID*. In this view we can see the details of the product and its image.

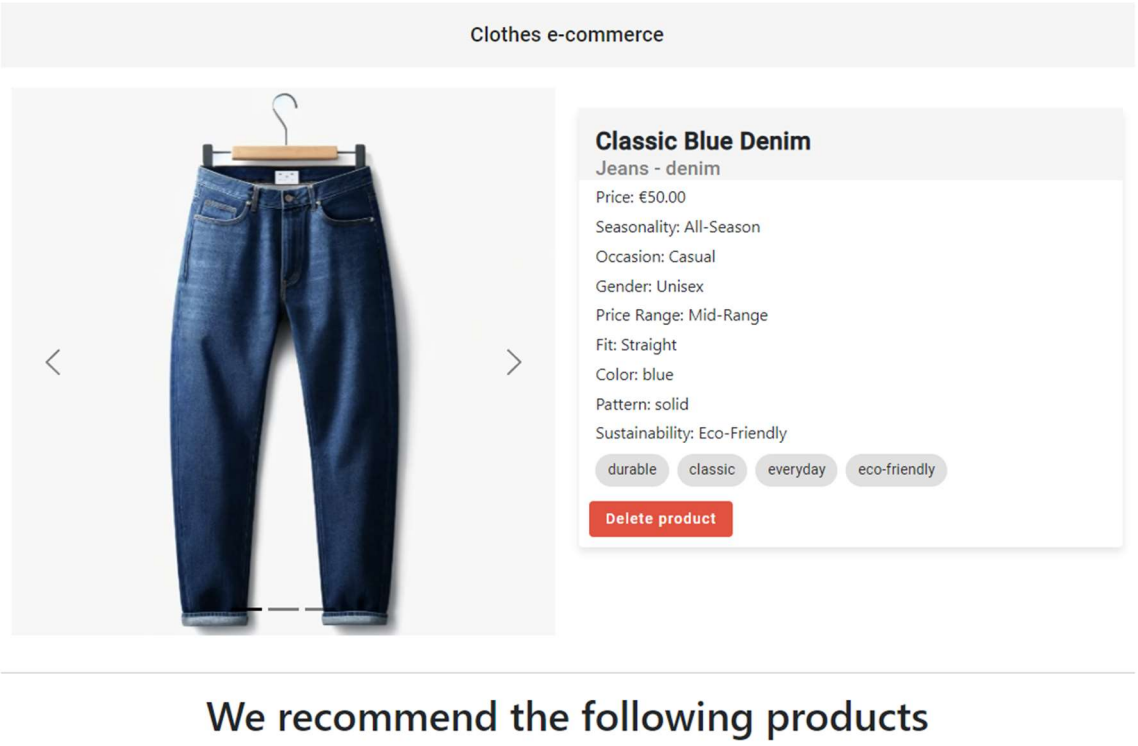

Figure 46: Product detail view Angular application.

If we scroll down in the view we can see the recommended products ordered by the similarity with the product in the view.

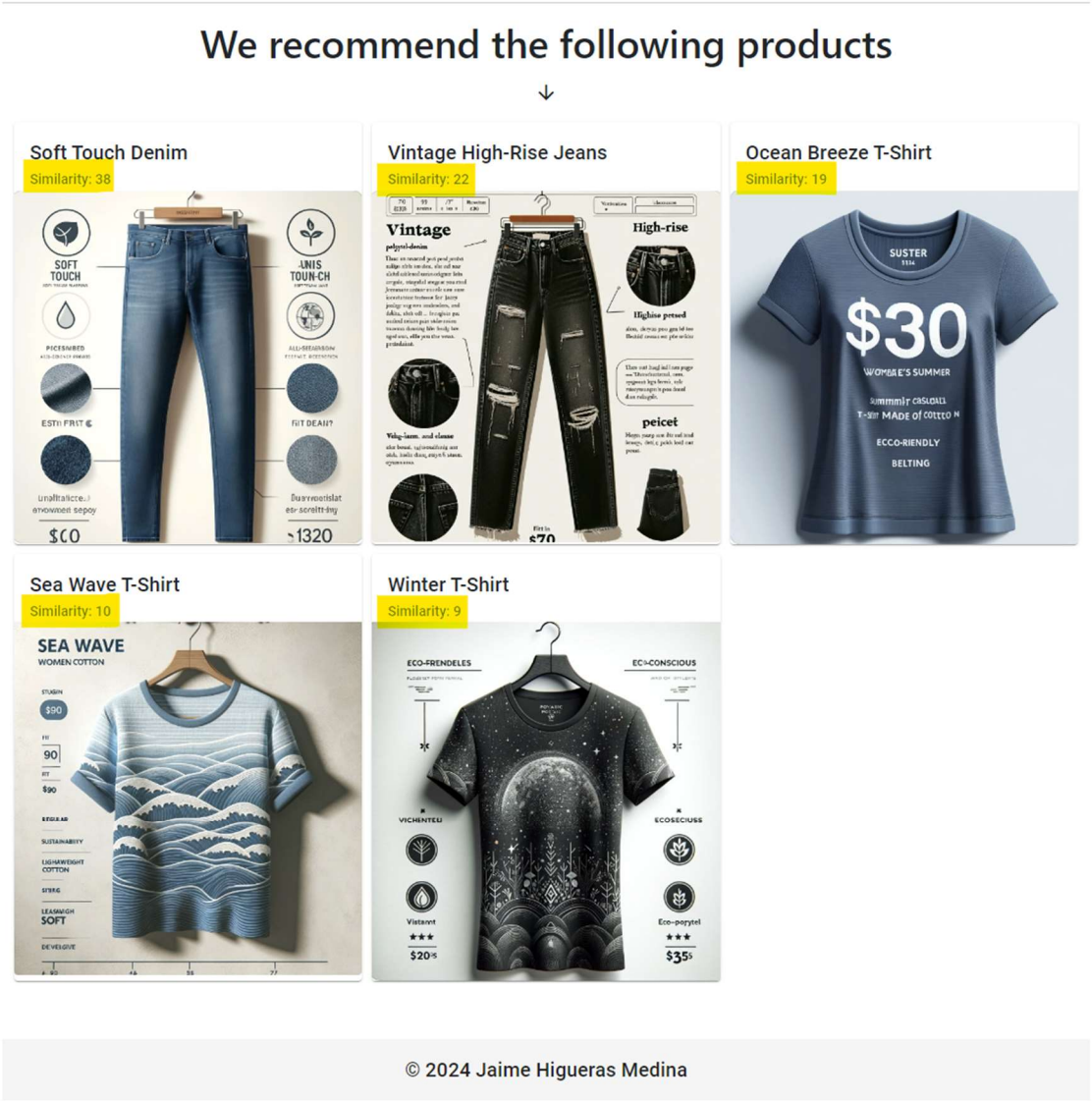

Figure 47: Recommendation product list in product detail view Angular application.

Internally the Angular application made two requestd. One to the product-microservice in the end-point /products/productId to fetch all the information about the product.

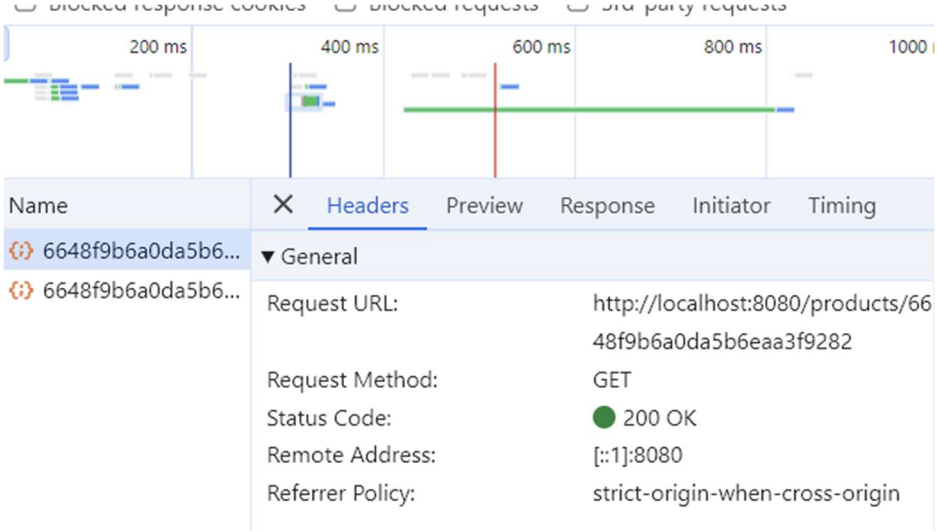

Figure 48: Request to product-microservice.

The second one was to the recommendation-microservice to the end-point recommendation/productId to fetch a list of recommended products based on the productId and ordered by similarity.

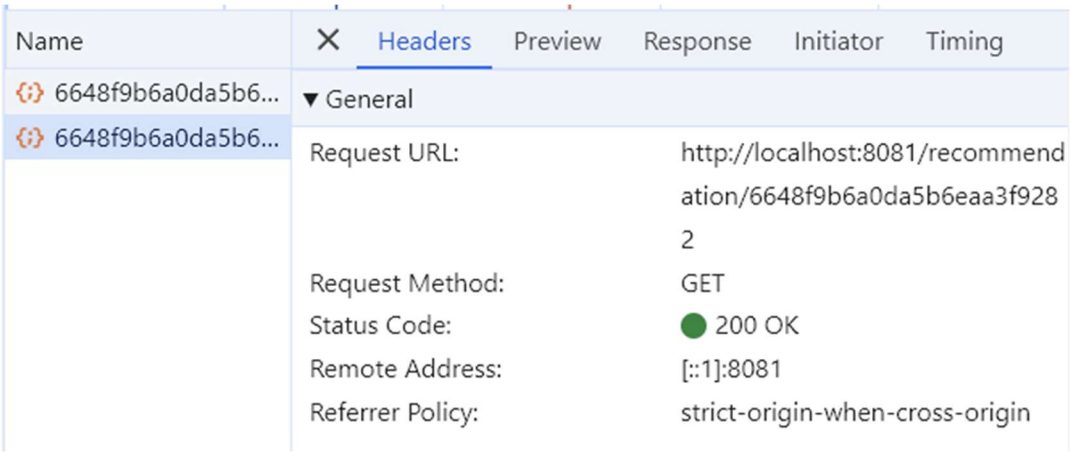

Figure 49: Request to recommendation-microservice

If we look at the metrics of recommendation-microservice in which the maximum duration of the request response was 449 ms.

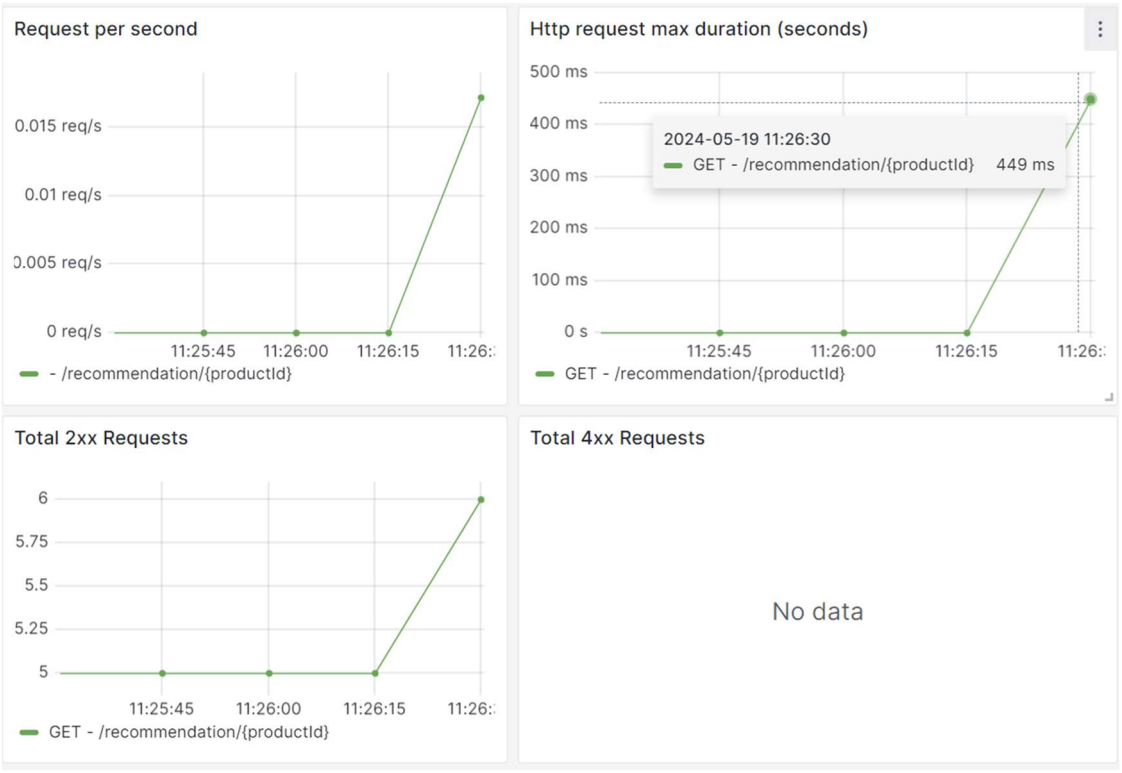

Figure 50: HTTP and Neo4j database recommendation-microservice metrics.

Internally, the request triggered two Cypher queries: one to check the existence of the product in the Neo4j database and the second to fetch the recommended products.

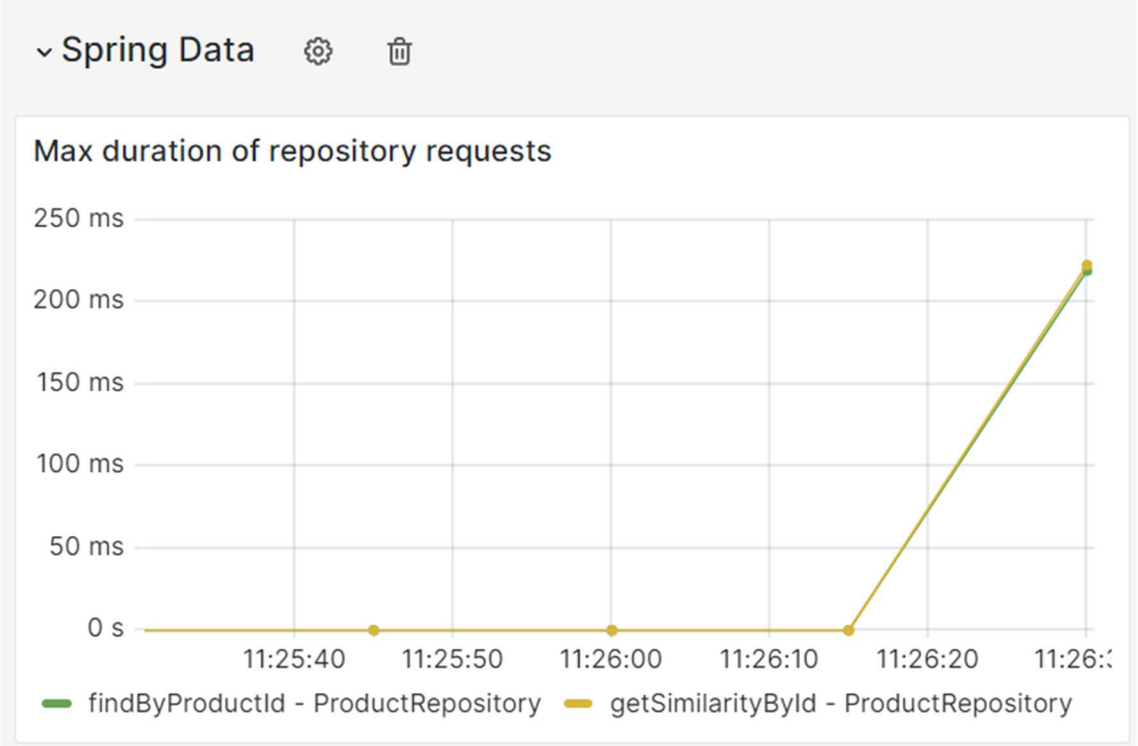

Figure 51: recommendation-microservices queries

7.4 Test case 4: Delete a product from the detail view section

Objective

The objective of this test case is to verify that a product can be successfully deleted from the product detail view in the frontend application. This test ensures that the deletion request is correctly processed by the backend.

Precondition

The system has at least 1 product created.

Result

If we click the delete product button, a warning windows is triggered. We click in the delete button again to confirm the operation.

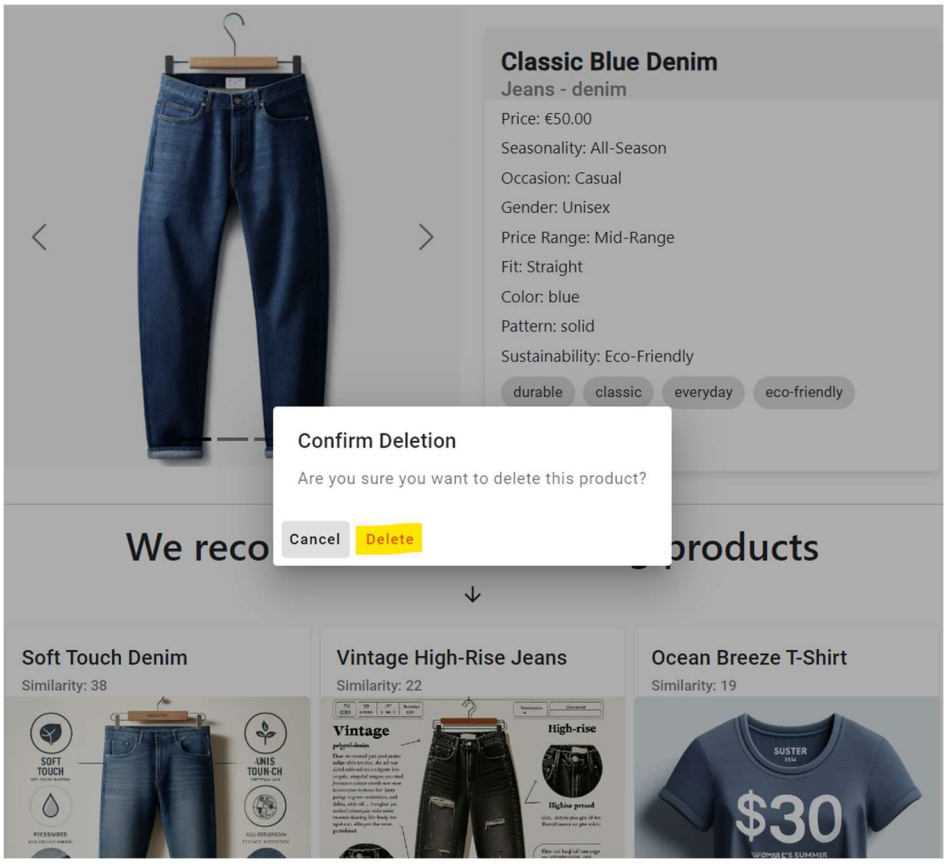

Figure 52: Confirm deletion warning

If we look at the metrics, we can check that the delete product endpoint was triggered in the product-microservice.

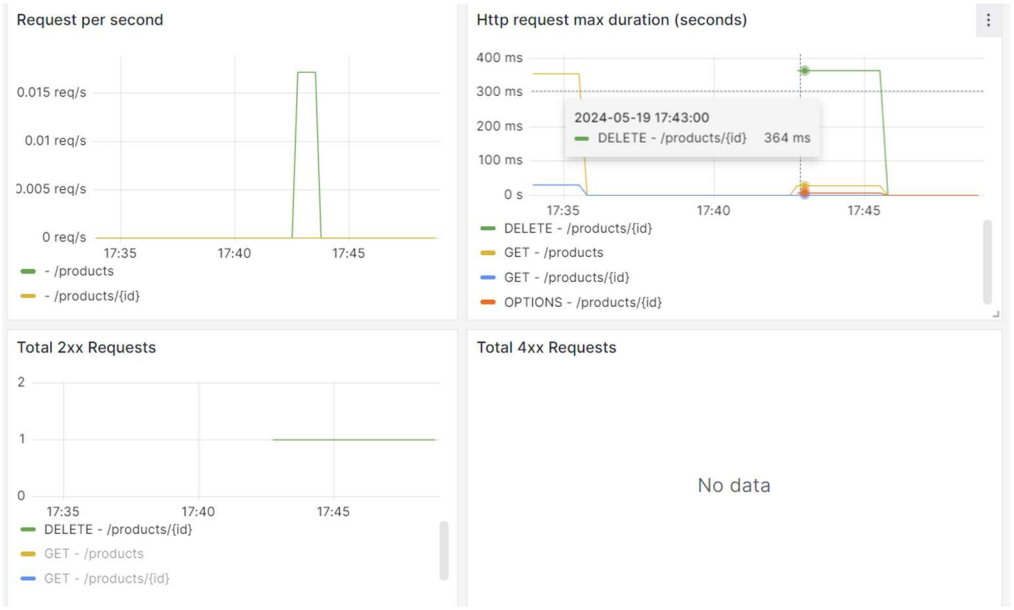

Figure 53: Delete end-point, product-microservice.

The service triggered two queries to the MongoDB database: one to check the existence of the product and the second for delete the product in the database.

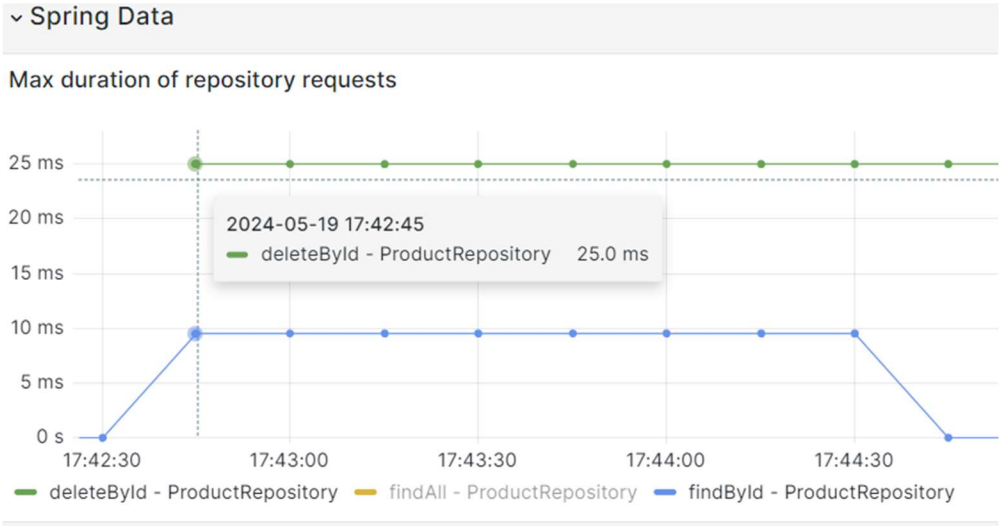

Figure 54: MongoDB queries, product-microservice

If we look at the recommendation-microservice metrics, we can check that the product was successfully eliminated in this microservice too.

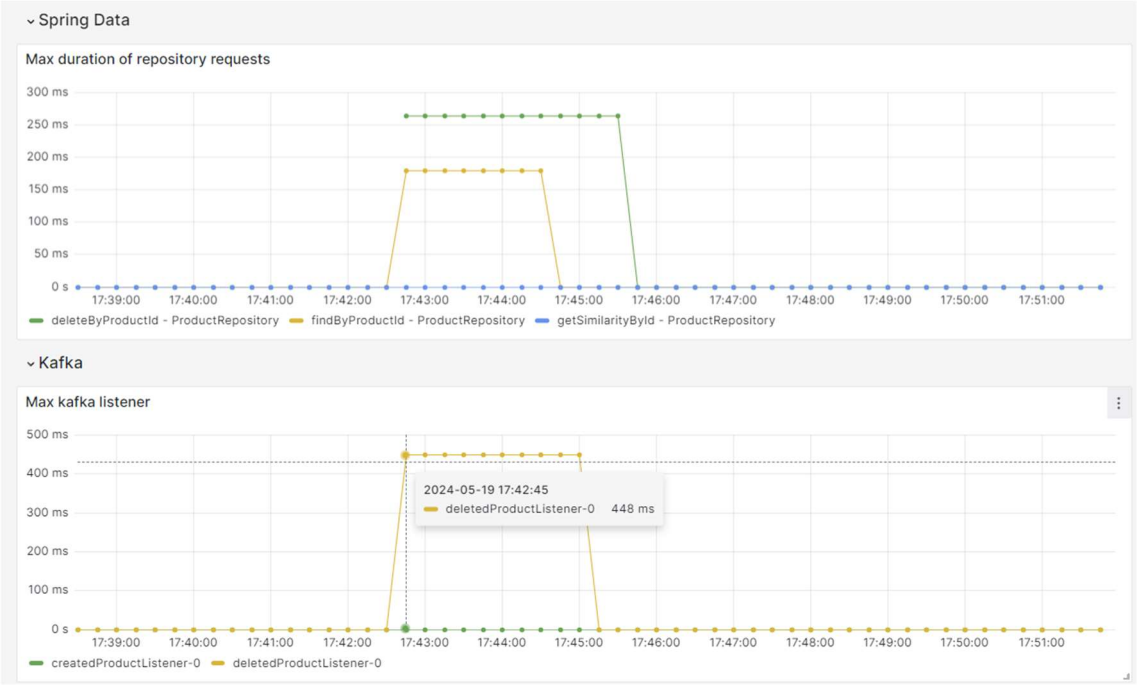

Figure 55: recommendation-microservice delete product

If we filter in the logs panel provided by Loki, we can see all the logs corresponding to the delete process.

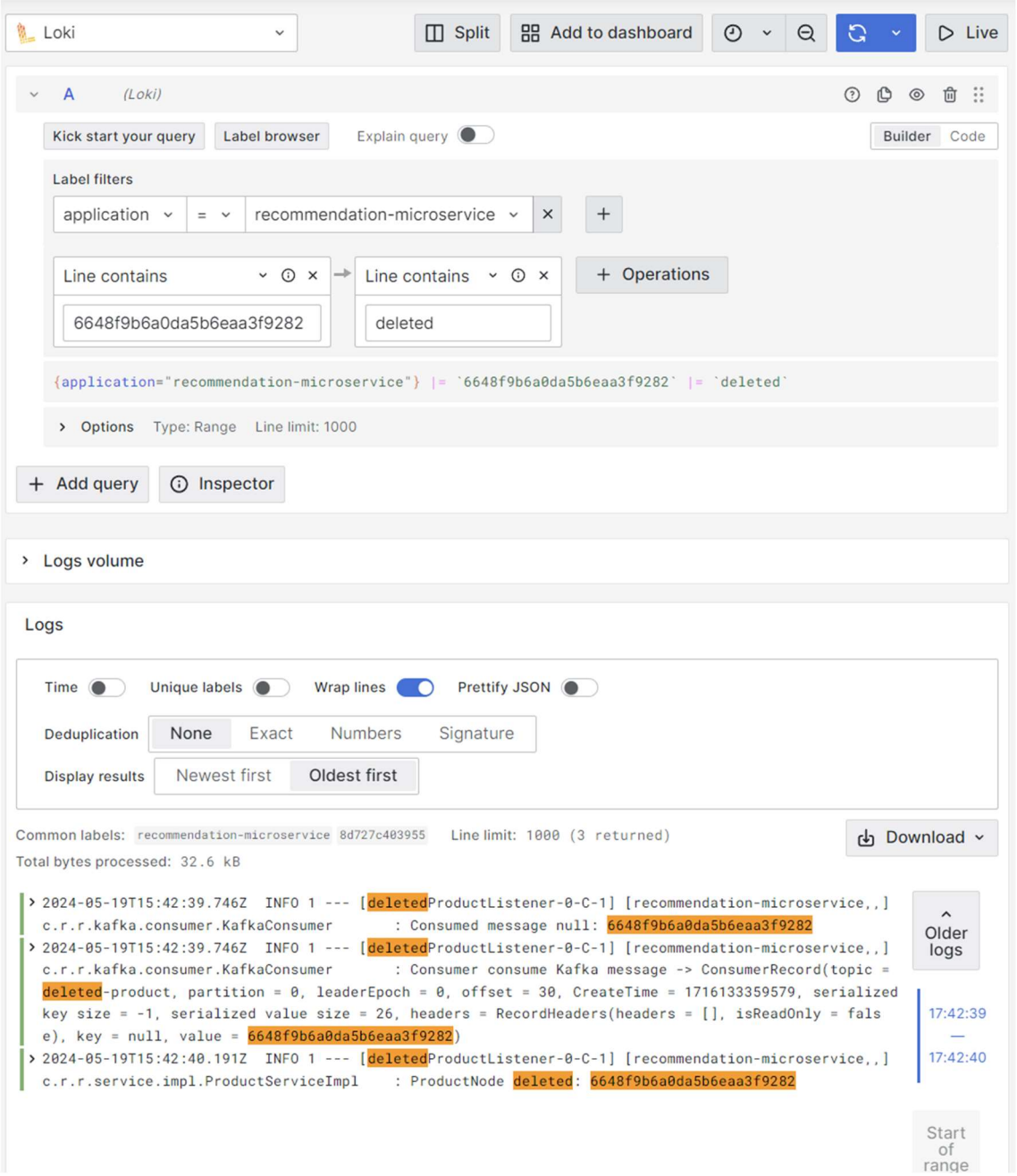

Figure 56: Logs filtered by productId and deleted.

Conclusions and future work

Conclusion

In this project we saw that microservice architecture is suitable for large systems and is becoming the main architecture nowadays in the business world. This architecture has achieved it, thanks to the advantages of productivity and acceleration of the development process that it offers. We also saw that, unfortunately, the microservice architecture also comes with its own drawbacks.

To reduce some of the problems that come with microservices, we have dived into the observability world in the context of distributed systems. With a good observability system, we can enhance the microservice architecture: Thanks to the observability system which provides deep insights into the system's behaviour, allowing us to efficiently monitor and troubleshoot it.

In the third theorical part, we have seen a brief introduction of the main recommendation system categories and explained our proposed recommendation system that fits in the content-based category. This recommendation system consists on using the power of a graph database to make a simple but robust recommendation system based on shared characteristics of the products.

Once we completed the theorical part, we implemented a clothes e-commerce application with a simple user interface and built it in a microservice architecture with its own observability system. With this recommendation system implementation, we put into practise all the theoretical concepts and implemented the content-based recommendation systems that we had proposed.

Future work

Due to having covered many different topics in this project, we know that we have left many things uncovered that are important to address in the future.

In the microservice side, the principal topics we want to cover are:

- **Security**: For demonstration purposes, in this project we did not cover the security topic, but in the real-world application it is something we must implement. Moreover, in microservice architecture every microservice must implement its own security layer.

- **Incorporation of orchestration tool**: Our application is hosted in a containerized approach. But to be able to scale in a horizontal way so the microservices can manage all of them. We have to use orchestration tools such as Kubernetes or Eureka Cloud. These tools offer a robust service discovery feature that dynamically manages the stateless nature of microservices.
- **Service Mesh**: Now, service mesh is an emerging tool to enhance communication between microservices. Service Mesh implements out of the box, all the services like circuit breakers, traffic routing, load balancing, secure communication, and so on, which allows to improve communication between microservices through the network.
- **Testing**: Testing is always important in software development, but especially in microservices architecture, it is critical to have reliable tests. We would like to cover testing topic broadly including unit testing, integration testing, contract testing and end-to-end testing.

In future versions of the observability system, we would like to cover:

- **Metrics**: We would like to create more sophisticated metrics that provide deeper insights. Additionally, we plan to increase the cardinality of the metrics already built, allowing more granular analysis and better understanding of the system.
- **Storage problem**: We also would like to address the main problem that all the observability systems have, which is the storage problem. The observability systems require a substantial amount of storage capacity to function effectively. It will be nice to study all the possibilities and provide a solution for this problem.
- **Exception tracking pattern**: Normally the services rarely throw an exception, it would be nice to use the exception tracking pattern [7]. This pattern consists of the microservices reporting the exceptions to a centralized exception tracking service. This pattern improves the response time to errors in production environments and helps troubleshooting.

As far as the recommendation part goes, the future work is straightforward. Adding a new user collaborative-filtering microservice and aggregating the results of the already existing content-based microservice with the new one. In this way, we can reach a more accurate and personalized recommendation.

References

[1] "Google Trends," Google Trends. https://trends.google.com/trends/explore?date=all&q=microservices&hl=es

[2] "Microservice Architecture pattern," *microservices.io*. https://microservices.io/patterns/microservices.html

[3] S. Newman, *Building microservices*. "O'Reilly Media, Inc.," 2021.

[4] C. Majors, L. Fong-Jones, and G. Miranda, *Observability Engineering*. "O'Reilly Media, Inc.," 2022.

[5] K. Yusov, "Monolith vs. Microservices: Why Companies Switch Back and Forth," *Jelvix*, Sep. 10, 2020. https://jelvix.com/blog/monolith-vs-microservices-architecture

[6] GfG, "The story of Netflix and microservices," *GeeksforGeeks*, May 17, 2020. https://www.geeksforgeeks.org/the-story-of-netflix-and-microservices/

[7] C. Richardson, Microservices patterns: With examples in Java. Simon and Schuster, 2018.

[8] "Clean Coder Blog," May 08, 2014. https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html

[9] L. Atchison, *Architecting for scale: How to Maintain High Availability and Manage Risk in the Cloud*. O'Reilly Media, 2020.

[10] A. Kainz, "Microservices vs. Monoliths: An Operational Comparison," *The New Stack*, May 22, 2023. [Online]. Available: https://thenewstack.io/microservices/microservices-vs-monoliths-an-operational-comparison/

[11]K. Xiang, "Patterns for distributed transactions within a microservices architecture | Red Hat Developer," *Red Hat Developer*, Aug. 14, 2023. https://developers.redhat.com/blog/2018/10/01/patterns-for-distributed-transactions-within-a-microservices-architecture#what_is_the_problem_

[12] "Pattern: saga," *microservices.io*. https://microservices.io/patterns/data/saga.html

[13] P. Vergadia, "Service orchestration on Google Cloud," *Google Cloud Blog*, Oct. 05, 2021. https://cloud.google.com/blog/topics/developers-practitioners/service-orchestration-google-cloud

[14] Wikipedia contributors, "API," *Wikipedia*, Dec. 10, 2023. https://en.wikipedia.org/wiki/API

[15] C. Voskoglou, "APIs have taken over software development | Nordic APIs |," *Nordic APIs*, Oct. 20, 2020. https://nordicapis.com/apis-have-taken-over-software-development/

[16] ByteByteGo, "What is REST API? Examples and how to use it: Crash Course System Design #3," *YouTube*. Aug. 24, 2022. [Online]. Available: https://www.youtube.com/watch?v=-mN3VyJuCjM

[17] "Hypertext Transfer Protocol (HTTP) Method Registry." https://www.iana.org/assignments/http-methods/http-methods.xhtml

[18] "HTTP Response Status Codes - HTTP | MDN," *MDN Web Docs*, Nov. 03, 2023. https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

[19] "What is API-first? The API-first Approach Explained | Postman," *Postman API Platform*. https://www.postman.com/api-first/#:~:text=API%2Dfirst%2C%20also%20called%20the,of%20treating%20them%20as%20afterthoughts

[20] "API documentation & design tools for teams | Swagger." https://swagger.io/

[21] M. Fowler, "bliki: CQRS," *martinfowler.com*. https://martinfowler.com/bliki/CQRS.html

[22] RobBagby, "CQRS pattern - Azure Architecture Center," *Microsoft Learn*. https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs

[23] W. Vogels, "Eventually consistent - revisited," *All Things Distributed*, Dec. 23, 2008. https://www.allthingsdistributed.com/2008/12/eventually_consistent.html

[24] Wikipedia contributors, "Two-phase commit protocol," *Wikipedia*, Feb. 06, 2024. https://en.wikipedia.org/wiki/Two-phase_commit_protocol

[25] U. Joshi, "Two-Phase commit," *martinfowler.com*. https://martinfowler.com/articles/patterns-of-distributed-systems/two-phase-commit.html

[26] "IBM Developer." https://developer.ibm.com/articles/use-saga-to-solve-distributed-transaction-management-problems-in-a-microservices-architecture/

[27] B. Moses, "What is data observability? 5 key pillars to know in 2024," *Monte Carlo Data*, Apr. 09, 2024. https://www.montecarlodata.com/blog-what-is-data-observability/

[28] Wikipedia contributors, "Cardinality," *Wikipedia*, Apr. 09, 2024. https://en.wikipedia.org/wiki/Cardinality

[29] Wikipedia contributors, "Cardinality (SQL statements)," *Wikipedia*, Nov. 14, 2021. https://en.wikipedia.org/wiki/Cardinality_(SQL_statements)

[30] J. C. Geeks, "A comprehensive guide to the log aggregation pattern," *Java Code Geeks*, Aug. 09, 2023. https://www.javacodegeeks.com/2023/08/a-comprehensive-guide-to-the-log-aggregation-pattern.html#:~:text=Log%20Aggregation%20is%20a%20pattern,management%2C%20analysis%2C%20and%20monitoring

[31] "Log Monitoring with Elastic Observability," *Elastic*. https://www.elastic.co/observability/log-monitoring/

[32] "Grafana Loki OSS | Log aggregation system," *Grafana Labs*. https://grafana.com/oss/loki/#:~:text=Loki%20is%20a%20log%20aggregation,our%20fully%20composable%20observability%20stack

[33] T. Aspecto, "Jaeger Tracing: The Ultimate Guide | Aspecto | JaegerTracing," *Medium*, Mar. 07, 2022. [Online]. Available: https://medium.com/jaegertracing/jaeger-tracing-a-friendly-guide-for-beginners-7b53a4a568ca

[34] "Architecture," *Jaeger: Open Source, Distributed Tracing Platform*. https://www.jaegertracing.io/docs/1.30/architecture/#ingester

[35] "Tracing :: Spring boot." https://docs.spring.io/spring-boot/3.3/reference/actuator/tracing.html

[36] A. S, "Logging, Traces, and Metrics: What's the difference?," *Atatus Blog - for DevOps Engineers, Web App Developers and Server Admins.*, Feb. 13, 2023. https://www.atatus.com/blog/logging-traces-metrics-observability

[37] A. W. S. Nicoll, "Spring Boot Actuator Web API Documentation." https://docs.spring.io/spring-boot/docs/current/actuator-api/htmlsingle/

[38] S. S. Iyengar and M. R. Lepper, "When choice is demotivating: Can one desire too much of a good thing?," *Journal of Personality and Social Psychology*, vol. 79, no. 6, pp. 995–1006, Dec. 2000, doi: 10.1037/0022-3514.79.6.995.

[39] M. Schrage, "Recommendation engines," in *The MIT Press eBooks*, 2020. doi: 10.7551/mitpress/12766.001.0001.

[40] H. S. Mohana and M. Suriakala, "A Study on Ontology Based Collaborative Filtering Recommendation Algorithms in E-Commerce Applications," *IOSR Journal of Computer Engineering*, vol. 19, no. 04, pp. 14–19, Jun. 2017, doi: 10.9790/0661-1904011419.

[41] Maruti Techlabs, "Types of Recommendation Systems & Their Use Cases - Maruti Techlabs - Medium," *Medium*, Jan. 05, 2022. [Online]. Available: https://marutitech.medium.com/what-are-the-types-of-recommendation-systems-3487cbafa7c9

[42] eBay Tech Berlin, "47th #ebaytechtalk: Deep Learning for Recommender Systems," *YouTube*. Apr. 12, 2018. [Online]. Available: https://www.youtube.com/watch?v=qQKMDJI_miU

[43] B. M. Sasaki, "Game Discovery: A recommendation algorithm for video games [Community Post]," *Graph Database & Analytics*, May 19, 2022. https://neo4j.com/blog/video-game-discovery-recommendation-algorithm/

[44] I. Robinson, J. Webber, and E. Eifrem, *Graph databases*. "O'Reilly Media, Inc.," 2013.

[45] Statista, "Primary programming languages among microservices developers worldwide 2022," *Statista*, Nov. 15, 2023. https://www.statista.com/statistics/1273806/microservice-developers-programming-language/

[46] "1. Introduction to Spring Framework." https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/overview.html#background-ioc

[47] "Aspect Oriented Programming with Spring :: Spring Framework." https://docs.spring.io/spring-framework/reference/core/aop.html

[48] MongoDB, "MongoDB: the Developer Data platform," *MongoDB*. https://www.mongodb.com/

[49] Neo4j, "NEO4J Graph Database & Analytics | Graph Database Management System," *Graph Database & Analytics*, Mar. 18, 2024. https://neo4j.com/

[50] "Apache Kafka," *Apache Kafka*. https://kafka.apache.org/intro

[51] "Choosing the number of partitions for a topic." https://docs.cloudera.com/cdp-private-cloud-base/7.1.9/kafka-performance-tuning/topics/kafka-tune-sizing-partition-number.html

[52] "Angular." https://angular.io/

[53] Wikipedia contributors, "Angular (web framework)," *Wikipedia*, Apr. 30, 2024. https://en.wikipedia.org/wiki/Angular_(web_framework)

[54] "'Docker overview,'" *Docker Documentation*, Apr. 17, 2024. https://docs.docker.com/get-started/overview/

[55] J. Haley, "Demystifying containers, Docker, and Kubernetes," *Microsoft Open Source Blog*, Jul. 15, 2019. https://cloudblogs.microsoft.com/opensource/2019/07/15/how-to-get-started-containers-docker-kubernetes/

[56] Prometheus, "Prometheus - Monitoring system & time series database." https://prometheus.io/

[57] "Grafana Tempo OSS | Distributed tracing backend," *Grafana Labs*. https://grafana.com/oss/tempo/

[58] Grafana Labs, "Grafana: The open observability platform | Grafana Labs," *Grafana Labs*. https://grafana.com/

[59] "Spring Cloud sleuth," *Spring Cloud Sleuth*. https://spring.io/projects/spring-cloud-sleuth

Annexes

Annex 1: Docker Compose setup for observability and microservices

To manage and run the backend side, including the observability infrastructure and the microservice architecture, we use Docker Compose and with one single command, all the necessary containers are spun up simultaneously, ensuring that each component is correctly configured and interlinked. This setup creates a total of 10 containers. 6 dedicated to the observability system and 4 to the microservice architecture.

To start the system, run in the terminal where docker-compose.yml is, the next command:

docker-compose up.

```
version: "1"
services:
  prometheus:
    image: prom/prometheus:v2.46.0
    command:
      - --enable-feature=exemplar-storage
      - --config.file=/etc/prometheus/prometheus.yml
    volumes:
      - ./docker/prometheus/prometheus.yml:/etc/prometheus/prometheus.yml:ro
    ports:
      - "9090:9090"
  grafana:
    image: grafana/grafana:10.1.0
    volumes:
      - ./docker/grafana:/etc/grafana/provisioning/datasources:ro
    environment:
      - GF_AUTH_ANONYMOUS_ENABLED=true
      - GF_AUTH_ANONYMOUS_ORG_ROLE=Admin
      - GF_AUTH_DISABLE_LOGIN_FORM=true
      - GF_USERS_DEFAULT_THEME=light
    ports:
      - "3000:3000"
  loki:
    image: grafana/loki:2.8.2
    ports:
      - "3100:3100"
    command: -config.file=/etc/loki/local-config.yaml
  tempo:
    image: grafana/tempo:2.2.2
    command: [ "-config.file=/etc/tempo.yaml" ]
```

```
    volumes:
      - ./docker/tempo/tempo.yml:/etc/tempo.yaml:ro
      - ./docker/tempo/tempo-data:/tmp/tempo
    ports:
      - "3110:3100"  # Tempo
      - "9411:9411" # zipkin
  zookeeper:
    image: confluentinc/cp-zookeeper:latest
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000
    ports:
      - 22181:2181
  kafka-ui:
    image: provectuslabs/kafka-ui:latest
    depends_on:
      - kafka
    ports:
      - 8090:8080
    environment:
      KAFKA_CLUSTERS_0_NAME: local
      KAFKA_CLUSTERS_0_BOOTSTRAPSERVERS: kafka:9092
      KAFKA_CLUSTERS_0_ZOOKEEPER: zookeeper:2181
  mongo:
    image: mongo
    container_name: mongodb
    environment:
      MONGO_INITDB_ROOT_USERNAME: rootuser
      MONGO_INITDB_ROOT_PASSWORD: rootpass
    ports:
      - "27017:27017"
  kafka:
    image: confluentinc/cp-kafka:latest
    depends_on:
      - zookeeper
    ports:
      - 29092:29092
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS:
PLAINTEXT://kafka:9092,PLAINTEXT_HOST://host.docker.internal:29092
      KAFKA_CFG_ADVERTISED_LISTENERS: PLAINTEXT://host.docker.internal:9092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
    healthcheck:
      test: [ "CMD", "curl", "-f", "http://zookeeper:2181" ]
      interval: 10s
      timeout: 5s
      retries: 5
  product-microservice:
    image: product-microservice:latest
    ports:
        - "8080:8080"
    volumes:
      - product-images:/app/uploads

  recommendation-microservice:
    image: recommendation-microservice:latest
    ports:
      - "8081:8081"
    volumes:
      - recommendation-images:/app/uploads

  angular-app:
    image: clothes-e-commerce:latest
    ports:
      - "4200:4200"
volumes:
```

```
product-images:
recommendation-images:
```

Code 11 docker-compose.yml.

Annex 2: Neo4j Database

For recommendation-microservice we use Neo4j AuraDB, which is a fully managed Neo4j cloud service to host our graph database.

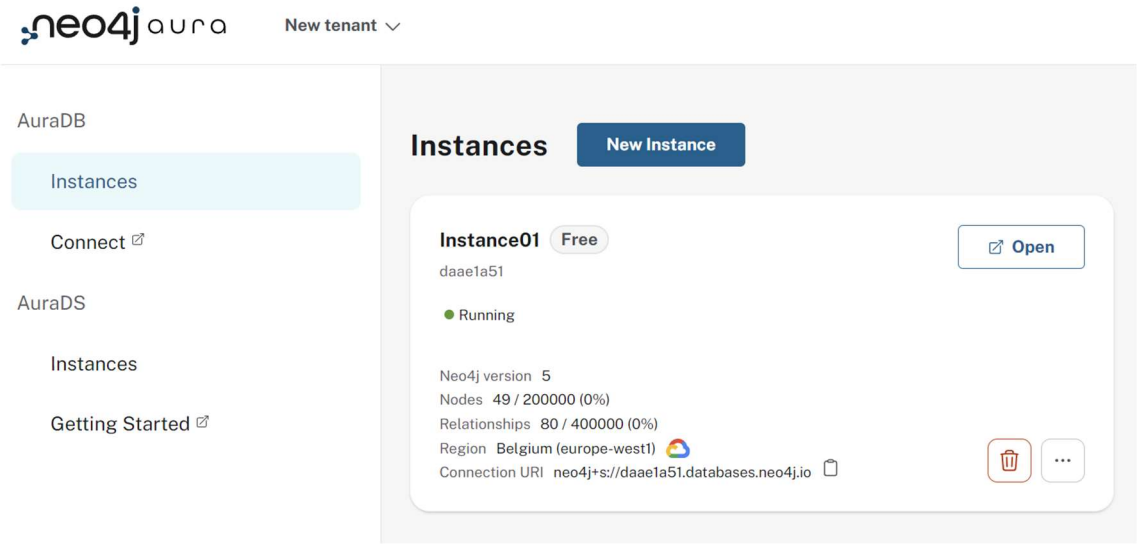

Figure 57: Console Neo4j AuraDB.

Annex 3: Docker fields microservices

The two microservices, product-microservice and recommendation-microservice, are deployed as Docker containers. For creating the Docker images, we used the following Dockerfiles to create the Docker images.

To create the product-microservice image run the next command:

docker build -t product-microservice .

```
# Stage 1: Build the application with Maven
FROM maven:3.8.4-eclipse-temurin-17 AS build
COPY src /home/app/src
COPY pom.xml /home/app
RUN mvn -f /home/app/pom.xml clean package

# Stage 2: Create the final Docker image
FROM eclipse-temurin:17-jdk-focal
EXPOSE 8080

# Copy the JAR file from the build stage to the final image
COPY --from=build /home/app/target/products-microservice-0.0.1-SNAPSHOT.jar /products-
microservice-0.0.1-SNAPSHOT.jar
```

```
ENTRYPOINT ["java", "-jar", "/products-microservice-0.0.1-SNAPSHOT.jar"]
```

Code 12 Dockerfile product-microservice.

To create the recommendation-microservice image run the next command:

docker build -t recommendation-microservice .

```
# Stage 1: Build the application with Maven
FROM maven:3.8.4-eclipse-temurin-17 AS build
COPY src /home/app/src
COPY pom.xml /home/app
RUN mvn -f /home/app/pom.xml clean package

# Stage 2: Create the final Docker image
FROM eclipse-temurin:17-jdk-focal
EXPOSE 8081

# Copy the JAR file from the build stage to the final image
COPY --from=build /home/app/target/recommendation-microservice-0.0.1-SNAPSHOT.jar
/recommendation-microservice-0.0.1-SNAPSHOT.jar

ENTRYPOINT ["java", "-jar", "/recommendation-microservice-0.0.1-SNAPSHOT.jar"]
```

Code 13 Dockerfile recommendation-microservice.

Annex 4: Docker fields Angular application

The Angular application (clothes e-commerce) is also deployed as a Docker container.

To create the image run the next command:

docker build -t clothes-e-commerce .

```
FROM node:20.13.1
WORKDIR /app
ENV NG_CLI_ANALYTICS="false"
RUN npm install -g @angular/cli@17.3.5
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 4200
CMD ["ng", "serve", "--host", "0.0.0.0"]
```

Code 14 Dockerfile Angular application

Annex 5: Configuration files observability system

The configuration files for Prometheus and Tempo are in the docker folder, see the Figure 58. When docker-compose.yml is up, it takes the prometheus.yml and tempo.yml files.

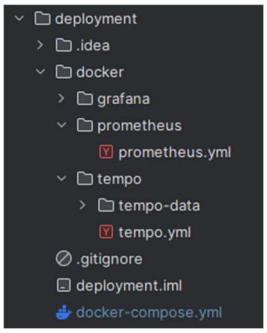

Figure 58: Deployment folder structure.

The Prometheus configuration defines the scrap interval and the targets from which Prometheus collects metrics.

```
global:
  scrape_interval: 2s
  evaluation_interval: 2s

scrape_configs:
  - job_name: 'product-microservice'
    metrics_path: '/actuator/prometheus'
    static_configs:
      - targets: ['host.docker.internal:8080']
  - job_name: 'recommendation-microservice'
    metrics_path: '/actuator/prometheus'
    static_configs:
      - targets: ['host.docker.internal:8081']
```

Code 15 prometheus.yml.

The Tempo configuration defined the port number on which Tempo listens for incoming HTTP requests. It also sets the configuration traces form Zipkin instrumentation. And in the last part it defined the place where the trace data is stored.

```
server:
  http_listen_port: 3200

distributor:
  receivers:
    zipkin:

storage:
  trace:
    backend: local
    local:
      path: /tmp/tempo/blocks
```

Code 16 tempo.yml.

For Loki configuration we had to add to both microservices the logback-spring.xml file. In this file we configured the logging framework, the application name from Spring Boot properties and the Loki logback appender.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <include resource="org/springframework/boot/logging/logback/base.xml"/>
    <springProperty scope="context" name="appName" source="spring.application.name"/>
    <appender name="LOKI" class="com.github.loki4j.logback.Loki4jAppender">
        <http>
            <url>http://host.docker.internal:3100/loki/api/v1/push</url>
        </http>
        <format>
            <label>
                <pattern>application=${appName},host=${HOSTNAME},level=%level</pattern>
            </label>
            <message>
                <pattern>${FILE_LOG_PATTERN}</pattern>
            </message>
            <sortByTime>true</sortByTime>
        </format>
    </appender>
    <root level="INFO">
        <appender-ref ref="LOKI"/>
    </root>
</configuration>
```

Code 17 Loki XML configuration.

Annex 6: Grafana dashboard PromQL Queries

To create the Grafana dashboard we used Prometheus Query Language (PromQL).

HTTP metrics panel

```
# Total request
sum(http_server_requests_seconds_count{job="$job",uri!="/actuator/prometheus"})

# Total request count by service (end-point)
sum(http_server_requests_seconds_count{job="$job",uri!="/actuator/prometheus"})

# Request per second
sum by(uri) (http_server_requests_seconds_sum{job="$job", uri!="/swagger-ui*/**",
uri!="/swagger-ui*/*swagger-initializer.js", uri!="/v3/api-docs", uri!="/v3/api-
docs/swagger-config", uri!="/**"}) / sum by(uri)
(http_server_requests_seconds_count{job="$job"})

# Request average duration
sum by(uri) (rate(http_server_requests_seconds_count{job="$job", uri!="/swagger-ui*/**",
uri!="/swagger-ui*/*swagger-initializer.js", uri!="/v3/api-docs", uri!="/v3/api-
docs/swagger-config", uri!="/**", uri!="/actuator/prometheus"}[1m]))

# HTTP request max duration (second)
http_server_requests_seconds_max{job="$job", uri!="/swagger-ui*/**", uri!="/swagger-
ui*/*swagger-initializer.js", uri!="/v3/api-docs", uri!="/v3/api-docs/swagger-config",
uri!="/**", uri!="/actuator/prometheus"}

# Total 2xx requests
http_server_requests_seconds_count{job="$job", status=~"2.*",
uri!="/actuator/prometheus", uri!="/swagger-ui*/**", uri!="/swagger-ui*/*swagger-
initializer.js", uri!="/v3/api-docs", uri!="/v3/api-docs/swagger-config", uri!="/**"}

# Total 4xx request
```

```
http_server_requests_seconds_count{job="$job", status=~"4.*",
uri!="/actuator/prometheus", uri!="/swagger-ui*/**", uri!="/swagger-ui*/*swagger-
initializer.js", uri!="/v3/api-docs", uri!="/v3/api-docs/swagger-config", uri!="/**"}
```

Code 18 PromQL queries for HTTP metrics panel.

Spring data

```
# Max. duration of repository request
spring_data_repository_invocations_seconds_max{job="$job"}
```

Code 19 PromQL for Spring data panel.

Kafka data

```
# The total number of records sent for a topic (per minute)
sum(increase(kafka_producer_topic_record_send_total{topic="created-product"}[1m]))

# Max Kafa listener

spring_kafka_listener_seconds_max{job="$job",
name!="org.springframework.kafka.KafkaListenerEndpointContainer#0-0",
name!="org.springframework.kafka.KafkaListenerEndpointContainer#1-0"}
```

Code 20 PromQL for Kafka metrics panel.

Annex 7: Characteristics weight

The *CharacteristicsWeight* is an enum class which is placed in the recommendation-microservice. These weights are used in the recommendation algorithm to calculate the similarity score between products.

```
@Getter
@AllArgsConstructor
public enum CharacteristicsWeight {
    CATEGORY(10),
    MATERIAL(9),
    SIZE_RANGE(8),
    PRICE_RANGE(7),
    COLOR(6),
    SEASONALITY(5),
    OCCASION(4),
    GENDER(4),
    BRAND(3),
    FIT(3),
    PATTERN(2),
    TRENDINESS(2),
    SUSTAINABILITY(2);

    private final int weight;

}
```

Code 21 Characteristic weight enum.

Annex 8: Microservice folder structure

The folder structure and the Java classes used to develop the microservices are:

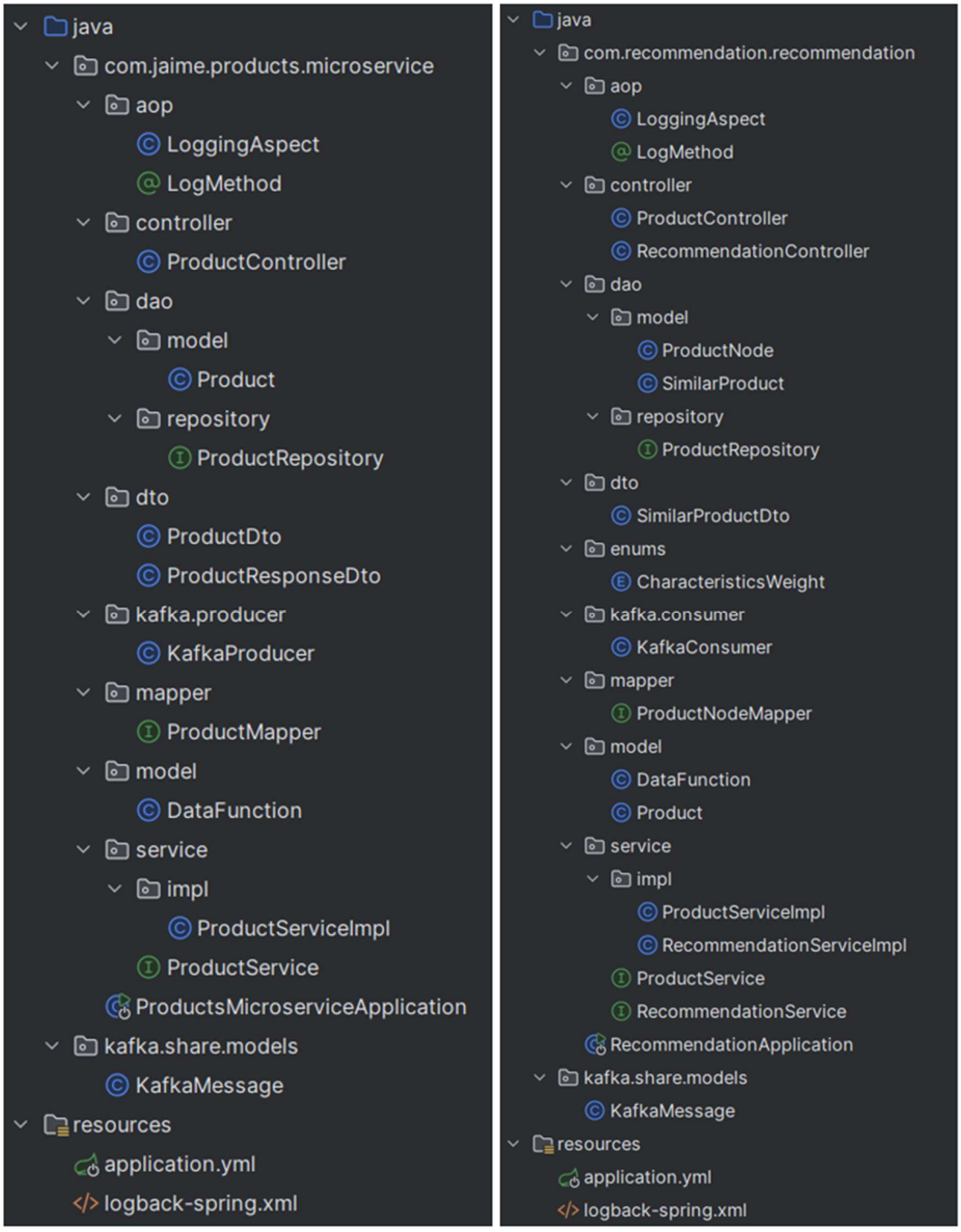

Figure 59: Microservice folder structure.

Annex 9: Angular folder structure

The folder structure and the TypeScript classes used to develop the Angular app are:

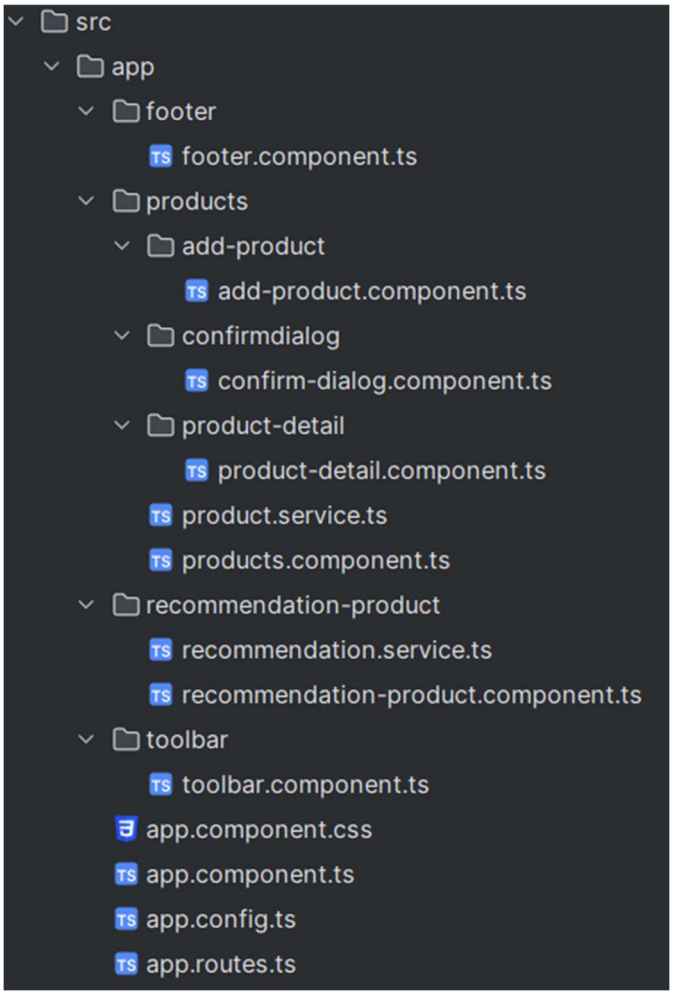

Figure 60: Angular app folder structure