UNIVERSITY OF ZAGREB

**FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING**

Dalijo Vorkapić

**APPLICATION AND IMPLEMENTATION OF MICROSERVICE ARCHITECTURE**

SEMINAR PAPER

Zagreb, 2023.

**Table of Contents**

[1. Introduction 1](#_Toc136534687)

[2. Implementation 4](#_Toc136534688)

[2.1 Microservices workflow 7](#_Toc136534689)

[2.2 Microservices discovery 12](#_Toc136534690)

[2.3 API Gateway 14](#_Toc136534691)

[2.4 Distributed tracing 15](#_Toc136534692)

[2.5 Code 17](#_Toc136534693)

[2.6 Containerization 19](#_Toc136534694)

[3. Results 23](#_Toc136534695)

[4. Conclusion 28](#_Toc136534696)

# Introduction

Microservices architecture is a type of software architecture where software system is composed of smaller, independent services that communicate and exchange data with each other. Each microservice has one and only responsibility, tied to a specific domain or a business function. This allows for greater flexibility, modularity, agility, resilience, and maintainability of the overall system. By breaking down a monolithic application into smaller services, microservices enable organizations to deliver new features faster, respond more quickly to changing business requirements, and reduce the risk of downtime and failure. Additionally, they promote a culture of continuous improvement and innovation, as individual teams can develop, test, and deploy services independently of others. An example of microservices architecture blueprint can be seen in *Figure 1.1*.

A diagram of a computer

Description automatically generated with low confidence

Figure 1.1: Microservices Architecture

Microservices architecture is mostly compared to a more traditional approach, monolithic architecture. In it, the entire system is developed as a single, self-contained unit, as seen in *Figure 1.2*. The logic is tightly coupled, making it difficult to make changes to individual components without affecting the rest of the system. This type of architecture is often used for smaller systems.

A screen shot of a computer

Description automatically generated with low confidence

Figure 1.2: Monolithic Architecture

Through the next sections, this paper will go through the implementation idea and process of microservices architecture, by giving insight to various technologies used, communication, discovery, containerization, distribution and many more aspects of the microservices implementation.

# Implementation

Since microservices architecture is widely spread today, there are many great application examples of implementation of microservices architecture. Each implementation might be slightly different, but they all share the same core principles of microservices, like modularity, independence, flexibility, communication, etc.

In order to show the most important principles, technologies, and approaches of microservices architecture, this paper will show the implementation of a **Weather Forecast Application.** Technologies used to develop this application are shown in *Figure 2.1*.

A screenshot of a phone

Description automatically generated with low confidence

Figure 2.1: Technologies used

The purpose of this application is not a complete functional use, but rather to show the key principles of microservices, communication, microservices discovery, containerization, as well as technologies used to enable such features, and much more.

The Weather Forecast Application can interact with two outside entities, the user, and the weather stations. Based on the provided location, the user can retrieve and view the forecast data for specified location. But, how to even have any weather information to use for a forecast. For that, the weather station entity comes into play. Its function is to gather weather measurements based on many different parameters, and to provide the weather measurements to the Weather Forecast Application. The application then stores the measurements and processes and analyzes them in order to make predictions for future weather conditions. Such black box schema of this system, meaning it only shows outside entities, inputs, and outputs, while it hides the inside functionalities and logic of the application, can be seen in Figure *2.2*. This black box schema can also be viewed as a base schema of a monolithic application, and in the next step, we will dive into the monolith structure, and break it into smaller pieces, the microservices.

A diagram of a weather forecast application

Description automatically generated

Figure 2.2: Black box schema of application

After defining outside entities, types of input/output communication and the application block, it is the time to dive one level deeper, into the Weather Forecast Application block, and see what is inside. On this level of hierarchy, the application is divided into microservices, as can be seen in the Figure *2.3*.

A diagram of a weather forecast application

Description automatically generated

Figure 2.3: Microservices architecture of application

## Microservices workflow

Regarding the microservices, firstly let’s take a look at the **Weather Collection Service**. As the name suggests, its duty is to collect weather measurements and its parameters. It receives the measurements from the weather stations, an outside entity that communicates with the whole weather forecast application. Since the main goal of a microservice is a principle of one responsibility, with the responsibility to collect weather measurements and communicate with the weather stations, the weather collection service achieves such principle.

As seen on *Figure 2.3,* the weather collection service communicates with other service, the **Data Processing Service**. The context of that communication is next: upon receiving a new measurement from the weather station, the weather collection service automatically passes the measurement to the data processing service. Next, the data processing service stores the measurements, and based on currently received measurement and other measurements stored in its database, the data processing service processes and analyses the group of measurements based on the location, and makes predictions for future weather condition occurrences and parameters, like temperature, humidity, wind, air quality, etc. Afterwards, it stores the predictions and parameters in its own database.

This way of communication is **called asynchronous communication**, where the data processing service is subscribed to the event of weather collection service, and when the weather collection service triggers the event with the new weather measurement, the data processing service receives the measurement through the event subscription. With this way, a non-blocking communication is made, meaning that the data processing service can do its own logic without waiting for a response from the weather collection service. This communication is achieved with the use of **Apache Kafka**, an event streaming platform. In its terminology, the weather processing service is producer, meaning it produces an event that is sent to the data processing service, which is a consumer of the event. As seen on *Code 2.1*, the weather collection service sends the weather measurement event through the kafkaTemplate, on topic of measurement, to which the data processing service is subscribed.

@Service  
@RequiredArgsConstructor  
@Slf4j  
public class WeatherMeasurementService {  
  
  
 private final KafkaTemplate<String,WeatherMeasurementEvent> kafkaTemplate;  
  
 public String addWeatherMeasurement(WeatherMeasurementRequest weatherMeasurementRequest) {

...  
 kafkaTemplate.send("measurementTopic",weatherMeasurementEvent);  
 return "New Weather Measurement added!";  
}

Code 2.1: Kafka send event

In the data processing service, there is a service function designated to be subscribed to the kafka topic event, and the function will execute once the event occurs. Such function is annotated with the @KafkaListener(topic), as seen in *Code 2.2*.

@Service  
@RequiredArgsConstructor  
@Slf4j  
public class DataProcessingService {  
  
 private final DataProcessingRepository dataProcessingRepository;  
 private final WeatherMeasurementRepository weatherMeasurementRepository;  
  
 @KafkaListener(topics = "measurementTopic")  
 public String getWeatherMeasurement(WeatherMeasurementEvent weatherMeasurementEvent) {

...

}

Code 2.2: Kafka event subscription

Both producer and consumer properties are defined on this microservices, providing definition for a topic of communication and event serialization parameters. Such properties can be seen in *Code 2.3*, showing dana processing service consumer attributes.

*# Kafka Properties*spring.kafka.bootstrap-servers=localhost:9092  
spring.kafka.template.default-topic=measurementTopic  
spring.kafka.consumer.group-id= measurementId  
spring.kafka.consumer.key-deserializer=org.apache.kafka.common.serialization.StringDeserializer  
spring.kafka.consumer.value-deserializer=org.springframework.kafka.support.serializer.JsonDeserializer  
spring.kafka.consumer.properties.spring.json.type.mapping=event:com.fer.sem2.dataprocessingservice.event.WeatherMeasurementEvent

Code 2.3: Kafka consumer properties

Referring to the *Figure 2.3,* the next element in communication chain is **Forecast Service**. The service is triggered upon the user’s (outside entity) request for the weather forecasts, based on the location it provided in the call. After the request, the forecast service needs the weather predictions and its parameters to make the forecast data view for the user. To achieve it, it calls up the data processing service. But this communication differs from the first one between weather measurements and data processing services. Now, the **synchronous communication** is needed, because the forecast service needs the data immediately, so it can show it to the user that requested it. To achieve the synchronous communication, the application uses **WebClient** paired with the **Resilience4J** library. The WebClient enables synchronous communication between separated services while the Resilience4J ensures fault tolerant communication and stability patterns. Firstly, when the get method for forecast is triggered, Resilience4J comes in the play and provides several fault tolerant functionalities to the function. It adds the time limiter as the time to wait for a response from another service, number of retry calls and a fallback function in case of an error. This parameters are defined in the properties file of the host, in this case, the forecast service, and can be seen in *Code 2.4* The defined parameters are called as annotations above the function that use it, in this case the get forecast function shown in *Code 2.5*. The next step is actual callup to the other microservice, where WebClient builds a uri to access the data processing service, and through the uri it calls its defined function getProcessedWeather to fetch the predictions. The WebClient build is seen in *Code 2.6.*

*#Resilience4j Properties*resilience4j.circuitbreaker.instances.forecast.registerHealthIndicator=true  
resilience4j.circuitbreaker.instances.forecast.event-consumer-buffer-size=10  
resilience4j.circuitbreaker.instances.forecast.slidingWindowType=COUNT\_BASED  
resilience4j.circuitbreaker.instances.forecast.slidingWindowSize=5  
resilience4j.circuitbreaker.instances.forecast.failureRateThreshold=50  
resilience4j.circuitbreaker.instances.forecast.waitDurationInOpenState=5s  
resilience4j.circuitbreaker.instances.forecast.permittedNumberOfCallsInHalfOpenState=3  
resilience4j.circuitbreaker.instances.forecast.automaticTransitionFromOpenToHalfOpenEnabled=true  
  
*#Resilience4j Timeout Properties*resilience4j.timelimiter.instances.forecast.timeout-duration=3s  
  
*#Resilience4j Retry Properties*resilience4j.retry.instances.forecast.max-attempts=3  
resilience4j.retry.instances.forecast.wait-duration=5s

Code 2.4: Resilience4J definition

@PostMapping  
@CircuitBreaker(name="forecast", fallbackMethod = "fallbackMethod")  
@TimeLimiter(name = "forecast")  
@Retry(name = "forecast")  
public CompletableFuture<Forecast> getForecast(@RequestBody ForecastRequest forecastRequest) {  
 return CompletableFuture.*supplyAsync*(() -> forecastService.getForecast(forecastRequest));  
}

Code 2.5: Resilience4J parameters

public Forecast getForecast(ForecastRequest forecastRequest) {  
  
 *log*.info("Fetching Processed Weather for Forecast, for location: {}",forecastRequest.getLocationName());  
 *log*.info("Contacting Data Processing Service...");  
  
 ProcessedWeatherResponse[] processedWeatherResponses = webClientBuilder.build()  
 .get().uri("http://data-processing-service/api/data-processing",  
 uriBuilder -> uriBuilder  
 .queryParam("locationName", forecastRequest.getLocationName())  
 .build())  
 .retrieve()  
 .bodyToMono(ProcessedWeatherResponse[].class)  
 .block();

...

}

Code 2.6: WebClient communication

Another communication the forecast service needs to achieve, is the communication with the **Notification Service**. The idea is next, when the forecast service receives the weather predictions from the data processing service, the forecast service goes through and processes the data received. If it finds an alarming weather prediction, like a weather prediction for storms, blizzards, droughts, etc., the forecast service notifies the notification service to send alerts to all the users. In terms of that communication, the notification service is subscribed to the alerting weather condition event of the forecast service, which then asynchronously triggers the event when such alarming prediction is found. To achieve that, the application once again uses Apache Kafka.

In order for user to make the request for forecasts, the user needs to be provided with a list of locations. Based on offered locations, the user can choose his location and send the location through the forecast request. The **location service** is in charge for providing the location list to the user as well as providing the option to add the new location to the location database.

## Microservices discovery

After the microservices workflows and communications have been defined in the previous section, the next question that occurs is, when microservice wants to communicate with another microservice, how do they find each other. The answer is **discovery server**. Its duty is to enable service discovery in a distributed environment, by maintain a registry of all registered services. When microservices start up, they register themselves with the discovery server. By doing so, they announce their presence and provide essential metadata as their network location (address and port), service endpoints, and health status, like up, down, etc. When microservice is looking to interact with other service, it can query the discovery server to obtain information about available instances of that service. For the purpose of a discovery server, **Netflix Eureka** server is used in this implementation. It operates on a client-server model, where microservices act as clients and the Eureka server acts as the central registry.

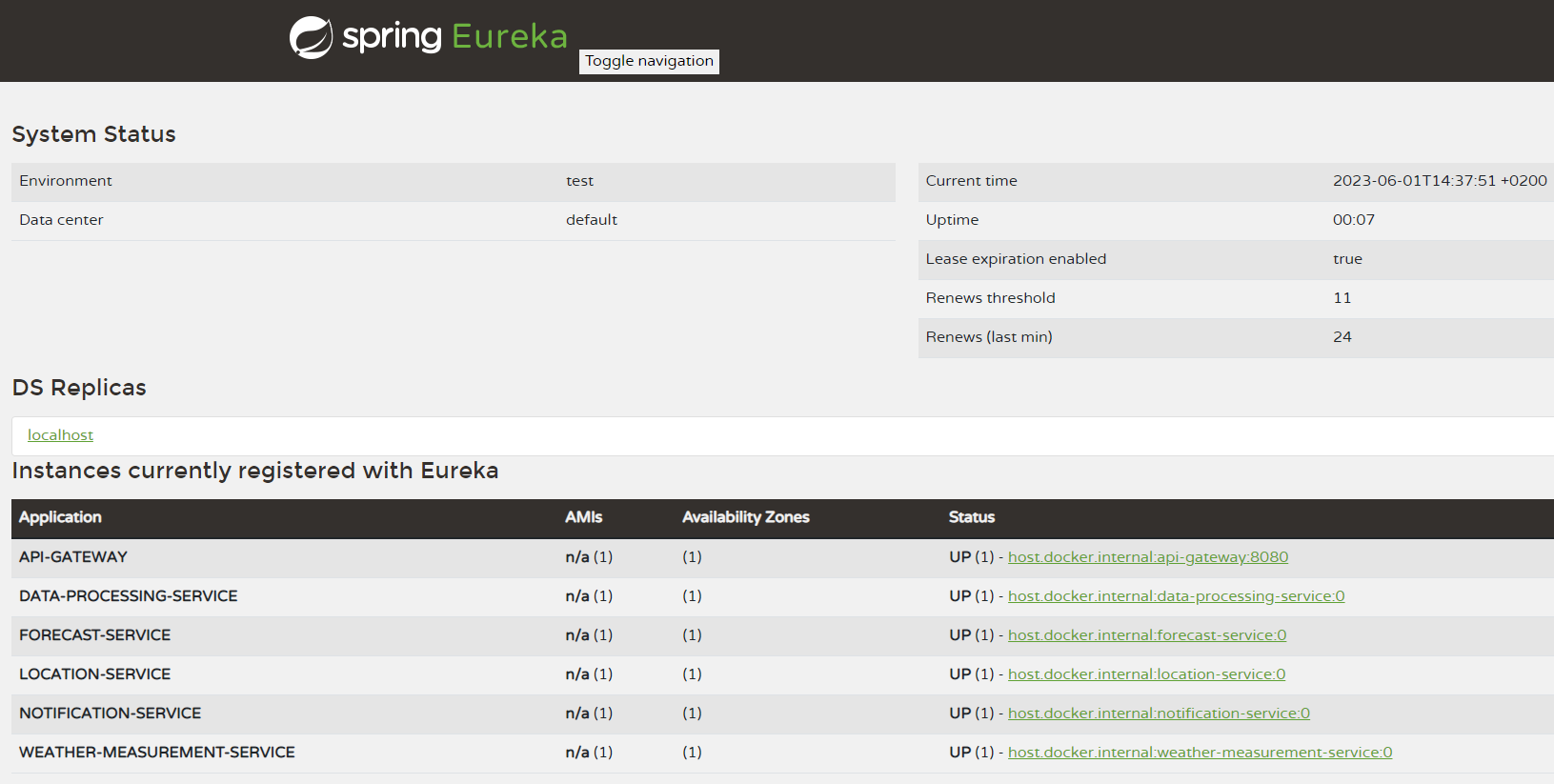


Figure 2.4: Eureka interface

As seen on *Figure 2.4*, Eureka contains all the microservices as the registered instances. It checks their health status, availability, addresses and ports. Eureka server initializes with the annotation @EnableEurekaServer, in the main application, as seen in *Code 2.7.* Properties of Eureka server are visible in Code 2.8*.*

@SpringBootApplication  
@EnableEurekaServer  
public class DiscoveryServerApplication {  
 public static void main(String[] args) {  
 SpringApplication.*run*(DiscoveryServerApplication.class, args);  
 }  
}

Code 2.7: Eureka server initialization

eureka.instance.prefer-ip-address=true  
eureka.client.register-with-eureka=false  
eureka.client.fetch-registry=false  
eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka/  
server.port=8761

Code 2.8: Eureka properties

After the server is initialized, each microservices registers to the discovery server with the annotation @EnableEurekaClient, with syntax same as in the *Code 2.7.*

## API Gateway

By looking at *Figure 2.3*, the last piece of puzzle that is left to explain is the **API Gateway**. It is visible that all the requests from the outside entities go firstly to the API Gateway, and then they are distributed to the needed service. Since this implementation contain several microservices and a discovery server, each one of them has its unique address and port. With that there are a lot of entry points that need to be tracked and linked to each request from the outside. As a fix for this problem, API Gateway acts as a single-entry point for all the request, providing a centralized and unified interface to interact with multiple services. It acts as an intermediary between clients and the underlying microservices. To achieve that, this implementation uses **Spring Cloud Gateway.** It provides a way to route, filter, and control traffic to and from services. It also provides load balancing, meaning it can distribute incoming requests across multiples instances of a same service. Spring Cloud Gateway integrates with Netflix Eureka discovery server. In API Gateway properties lies the setup for all the routes inside the application, and some of them are shown in *Code 2.9.*

logging.level.root=INFO  
logging.level.org.springframework.cloud.gateway.route.RouteDefinitionLocator = INFO  
logging.level.org.springframework.cloud.gateway = TRACE  
  
*# Weather Measurement Service route*spring.cloud.gateway.routes[0].id=weather-measurement-service  
spring.cloud.gateway.routes[0].uri=lb://weather-measurement-service  
spring.cloud.gateway.routes[0].predicates[0]=Path=/api/weather-measurement

…..

*# Forecast Service route*spring.cloud.gateway.routes[3].id=forecast-service  
spring.cloud.gateway.routes[3].uri=lb://forecast-service  
spring.cloud.gateway.routes[3].predicates[0]=Path=/api/forecast  
  
  
*# Discovery server route*spring.cloud.gateway.routes[4].id=discovery-server  
spring.cloud.gateway.routes[4].uri=http://localhost:8761  
spring.cloud.gateway.routes[4].predicates[0]=Path=/eureka/web  
spring.cloud.gateway.routes[4].filters[0]=SetPath=/

Code 2.9: API Gateway routes setup

## Distributed tracing

Distributed tracing is a technique used to monitor and analyze the flow of requests as they travers through a distributed system composed of multiple interconnected components, such as microservices. **Zipkin** distributed tracing system is used for this implementation. A unique identifier called trace ID is assigned to each incoming request at its entry point. This identifier is then propagated through subsequent service-to-service calls, allowing the request to be traced across the entire system. A prompt that lists all recent requests through the microservices is seen in *Figure 2.5Figure 2.5*, where spans present number of hops between services. In *Figure 2.6*, a request distribution and time through multiple services is visible.

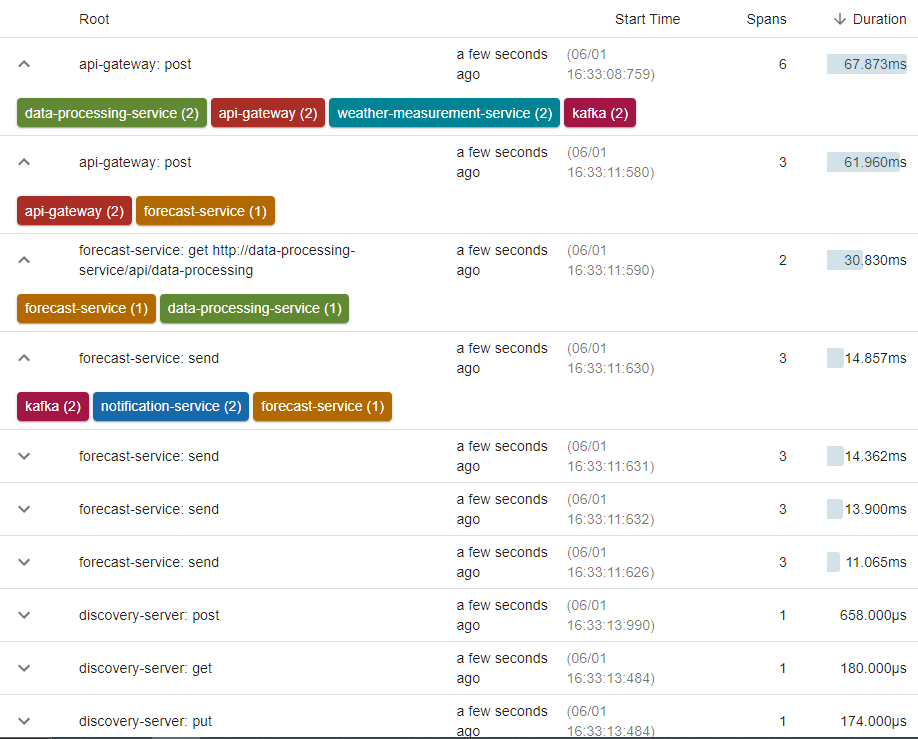


Figure 2.5: Requests tracing

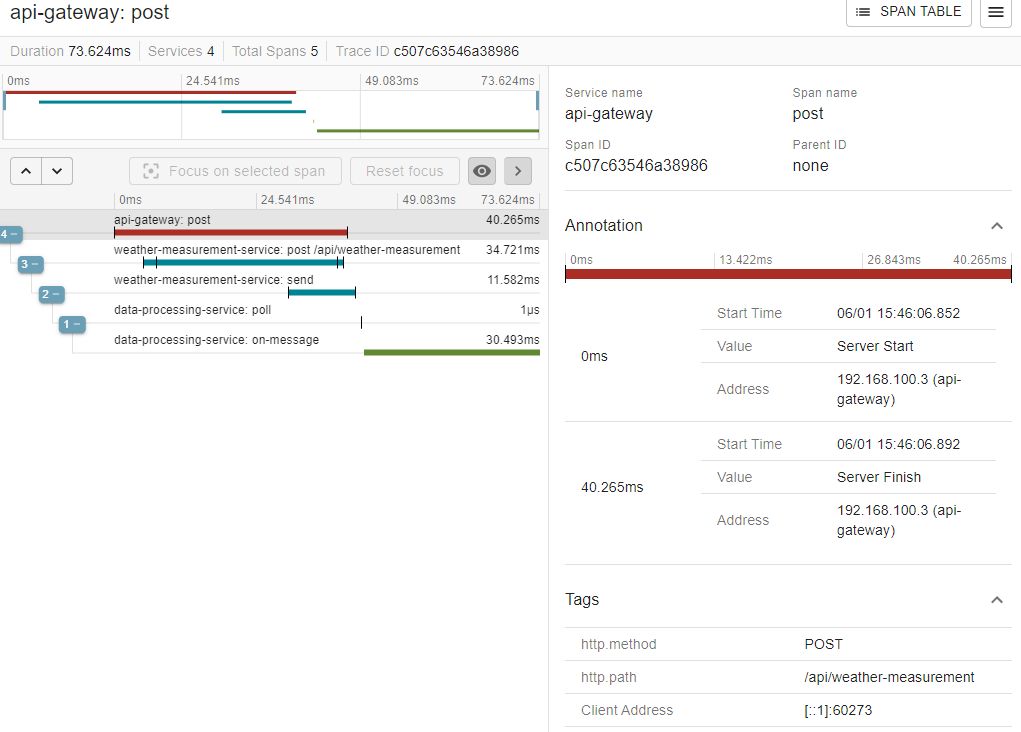


Figure 2.6: Request distribution

## Code

The whole code of this implementation is written in **Java** 17 programming language, in combination with **Java Spring Boot** framework. For build automation and dependency managements in this Java project, **Apache Maven** tool is used. In Java, **Lombok** library is used to reduce boilerplate code, and **SLF4J** library as a logging interface for the application.

Regarding the databases, in this implementation there two databases. A **MySQL** relational database is used as a shared database between weather collection service and data processing service, and **Mongo** noSQL database for storing locations and their attributes.

Regarding a MySQL database, *Figure 2.7* shows two entities with corresponding attributes, the data processing and weather measurement entities since the database is shared between those services. For location service, an entity for Mongo is created, and its attributes are shown in *Figure 2.8*

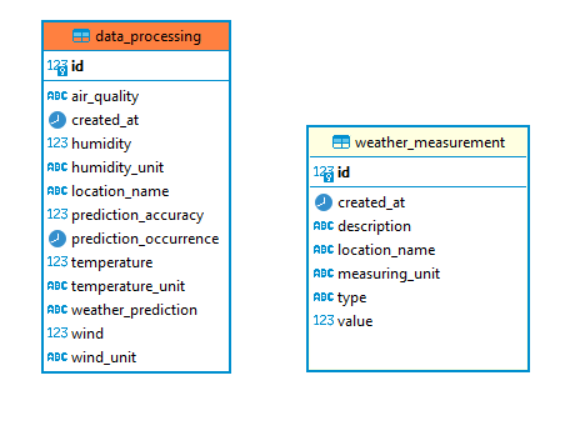


Figure 2.7: MySQL entities



Figure 2.8: Mongo entity

In terms of code structure, microservices, discovery server and API Gateway are organized into separate modules, that all belong to the main weather forecast application module, as seen in Figure *2.9*. This ensures that each module can run separated from the rest.

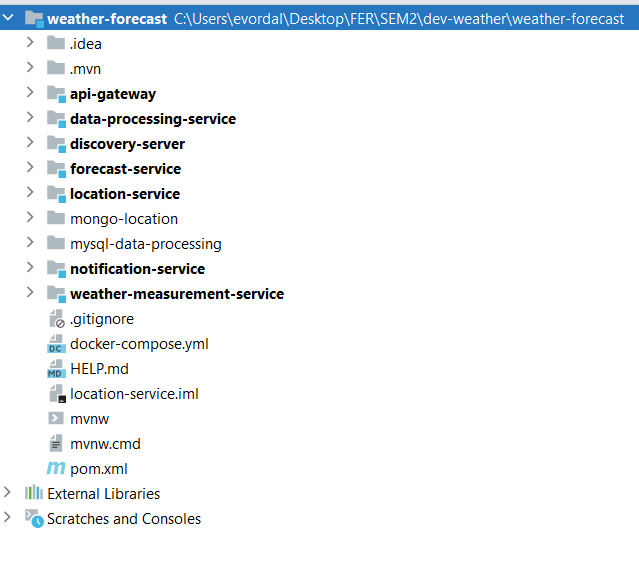


Figure 2.9: Modules structure

## Containerization

As mentioned in the previous section, each microservice, alongside API Gateway Discovery Server is ran as a separated application module. Additionally, the implementation contains of additional separated components, like the Zipkin server, Kafka broker and the databases servers for data processing and locations. That’s why containerization plays a major role in this implementation context, providing packaging of services and their dependencies into a self-contained unit. By that it ensures portability, isolation between services, ensuring that each service runs independently, scalability, etc. For this implementation containerization, **Docker** and **Docker Compose** are used. Docker allows to build, package, and run applications in isolated containers. Docker Compose defines and manages those containers. It provides coordination and orchestration between multiple Docker containers to ensure they work together and communicate seamlessly. A look inside of docker compose file, and some of the application containers setup can be seen in *Code 2.10.*

To create Docker containers for the application modules, firstly the images of each module need to be created. It is a executable software package that contains everything from the module itself, and it serves as a template for creating Docker containers, which are instances of the image that can be run on a Docker engine. In order to build all this images, a Java-based tool, **Jib**, is used to simplify the process of building container images. It also integrates seamlessly with Maven build tool.

Regarding this project, Docker images are shown in*Figure 2.10: Docker imagesFigure 2.10*, and Docker containers, that are built based on the images and the docker compose file, are shown in *Figure 2.11*. It is visible that in the end there are 12 separated containers, each running in its own independent environment. Not only services are contained, as it can be seen, the implementation has Docker containers for databases, Kafka broker, zipkin server, etc. Through the docker compose file it is defined which container depends on other containers, ports and environment variables for communication and connection between each other, order of running, and much more.

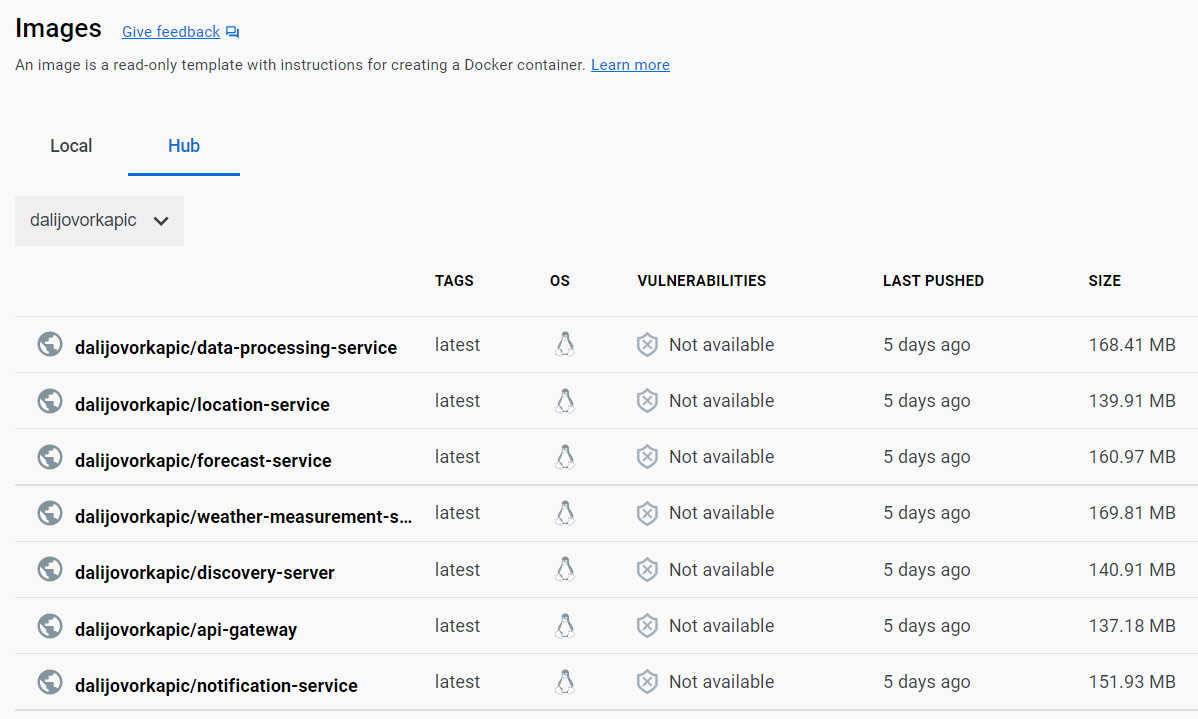


Figure 2.10: Docker images

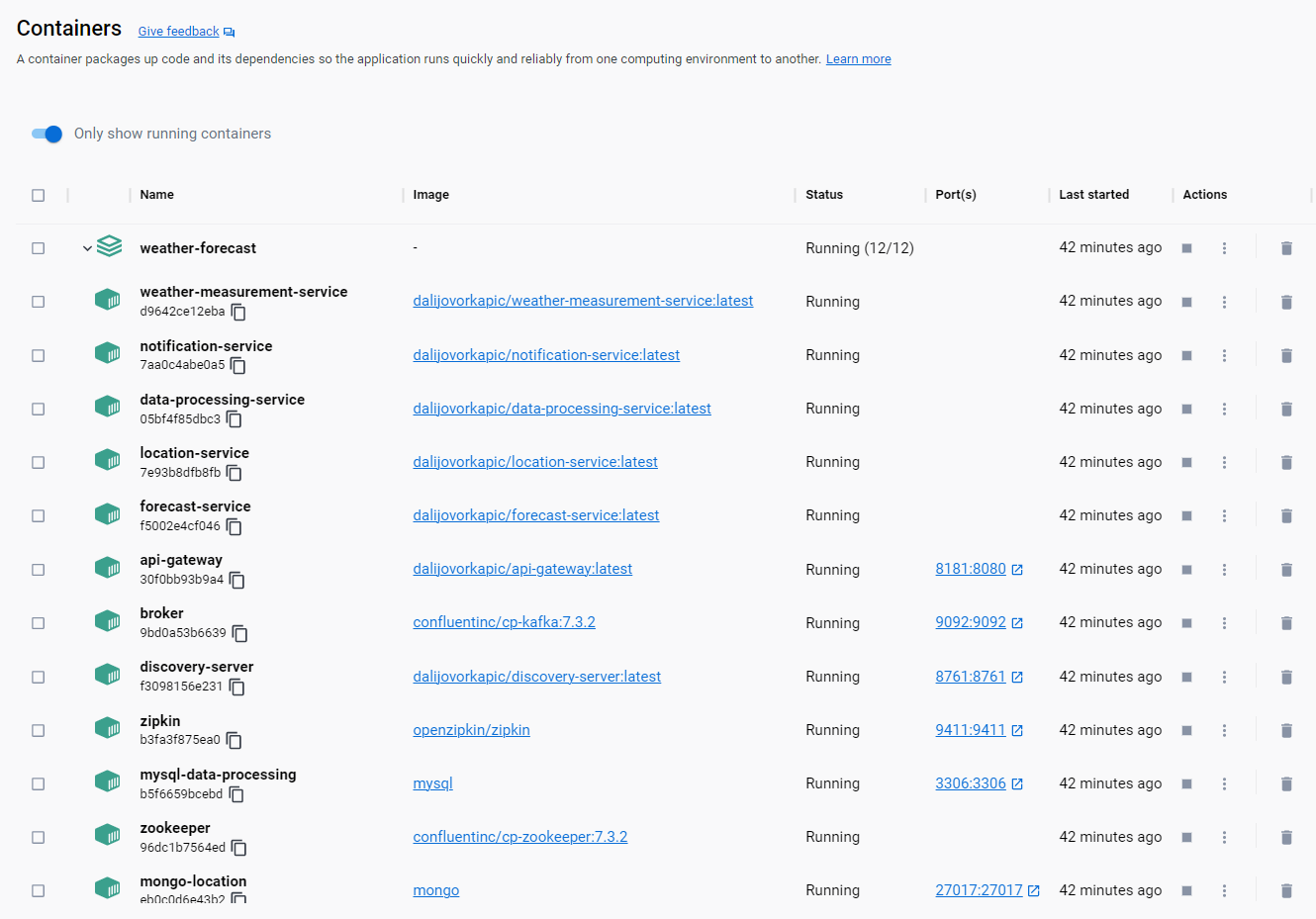


Figure 2.11: Docker containers

---  
version: '3'  
services:  
  
 *## MySQL Docker Compose Config* mysql-data-processing:  
 container\_name: mysql-data-processing  
 image: mysql  
 restart: unless-stopped  
 environment:  
 MYSQL\_USER: root  
 MYSQL\_PASSWORD: mysql  
 MYSQL\_DATABASE: data-processing-service  
 volumes:  
 - ./mysql-data-processing:/var/lib/mysql  
 ports:  
 - "3306:3306"  
  
  
 *## Eureka Server Docker Compose Config* discovery-server:  
 image: dalijovorkapic/discovery-server:latest  
 container\_name: discovery-server  
 ports:  
 - "8761:8761"  
 environment:  
 - SPRING\_PROFILES\_ACTIVE=docker  
 depends\_on:  
 - zipkin  
  
 *## API Gateway Docker Compose Config* api-gateway:  
 image: dalijovorkapic/api-gateway:latest  
 container\_name: api-gateway  
 ports:  
 - "8181:8080"  
 expose:  
 - "8181"  
 environment:  
 - SPRING\_PROFILES\_ACTIVE=docker  
 - LOGGING\_LEVEL\_ORG\_SPRINGFRAMEWORK\_SECURITY= TRACE  
 depends\_on:  
 - zipkin  
 - discovery-server  
  
 *## Data Processing Service Docker Compose Config* data-processing-service:  
 container\_name: data-processing-service  
 image: dalijovorkapic/data-processing-service:latest  
 environment:  
 - SPRING\_PROFILES\_ACTIVE=docker  
 - SPRING\_DATASOURCE\_URL=jdbc:mysql://mysql-data-processing:3306/data-processing-service  
 depends\_on:  
 - mysql-data-processing  
 - discovery-server  
 - zipkin  
 - api-gateway  
 - broker  
  
 *## Forecast Service Docker Compose Config* forecast-service:  
 container\_name: forecast-service  
 image: dalijovorkapic/forecast-service:latest  
 environment:  
 - SPRING\_PROFILES\_ACTIVE=docker  
 depends\_on:  
 - discovery-server  
 - zipkin  
 - api-gateway  
 - broker  
  
 *## Location Service Docker Compose Config* location-service:  
 container\_name: location-service  
 image: dalijovorkapic/location-service:latest  
 environment:  
 - SPRING\_PROFILES\_ACTIVE=docker  
 depends\_on:  
 - mongo-location  
 - discovery-server  
 - zipkin  
 - api-gateway

.. . . .

Code 2.10: Docker compose file

# Results

This section will show the whole workflow of the implementation, inputs, and output results.

Firstly, the user is prompted with location choices, by making a get request to the api gateway. The api gateway redirects the call to the location service that fetches the locations from its mongo database and returns the result to the user, as seen in *Figure 3.1.*

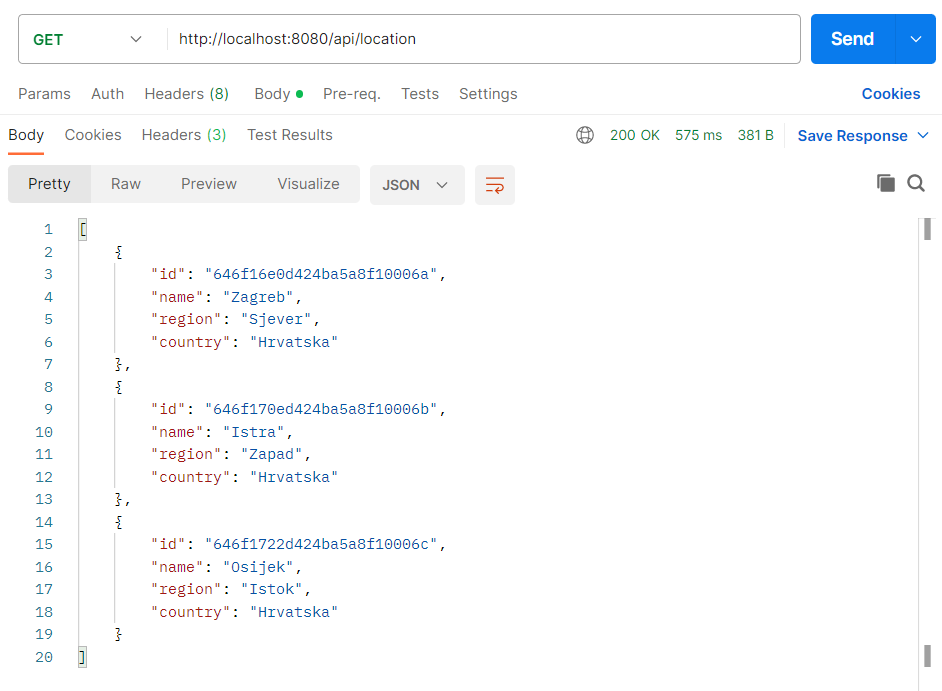


Figure 3.1: Location fetch

After the location fetch, the user choses a location and sends a request for the forecast, by calling the api gateway with a specified forecast route. Api gateway then processes the route and redirects the request to the forecast service. Next, the forecast service, as mentioned before, synchronously contacts the data processing service and fetches weather predictions as seen in *Figure 3.2*, which it prepares for the view and returns it back to the user as a result, seen in *Figure 3.4Figure 3.4*. If during the process of data, the forecast service spots an alarming weather condition, it send alert to the notification service. That communication can be seen in *Figure 3.2* and *Figure 3.3*.

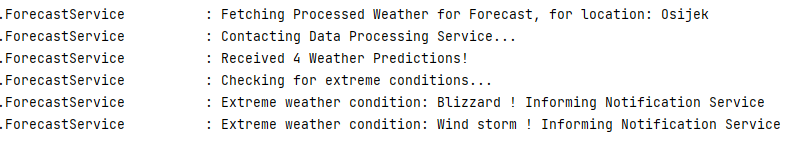


Figure 3.2: Processed weather exchange

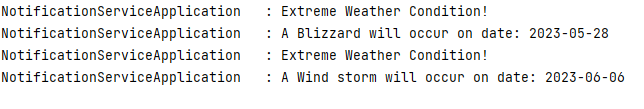


Figure 3.3: Notification alert

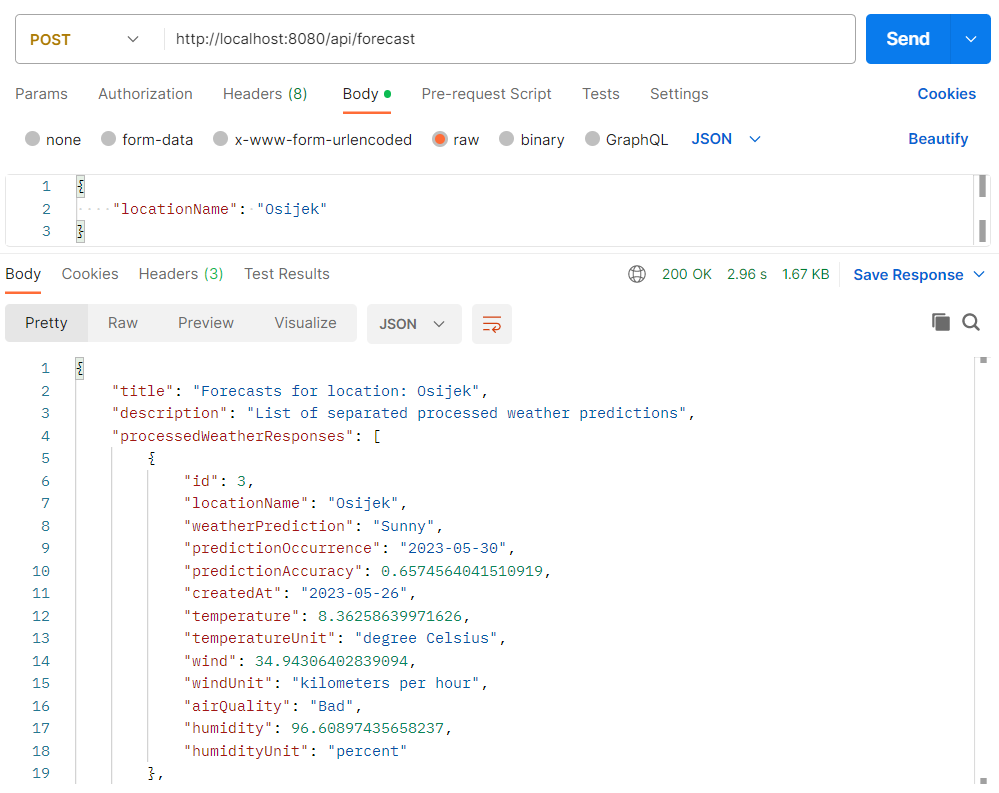


Figure 3.4: Forecasts fetch

On the other side of the system, when a weather station creates a new weather measurement, it sends it in form of a post request to the api gateway, which routes the request to the weather collection service. The example of weather measurement creation can be seen in *Figure 3.5*. As soon as the measurement is made, the weather collection service calls the data processing service and passes the measurement, which then the data processing service analyses, as seen in prompt in *Figure 3.6*.

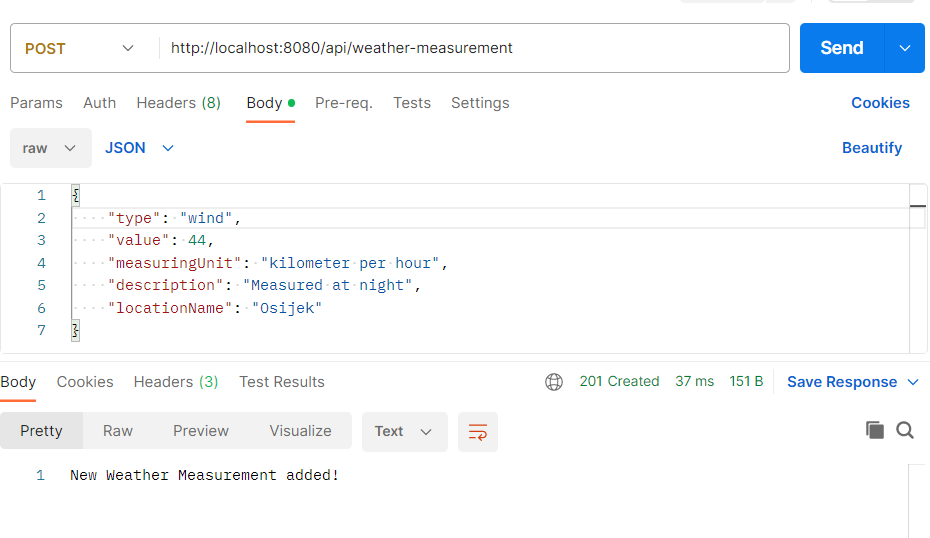


Figure 3.5: Weather measurement

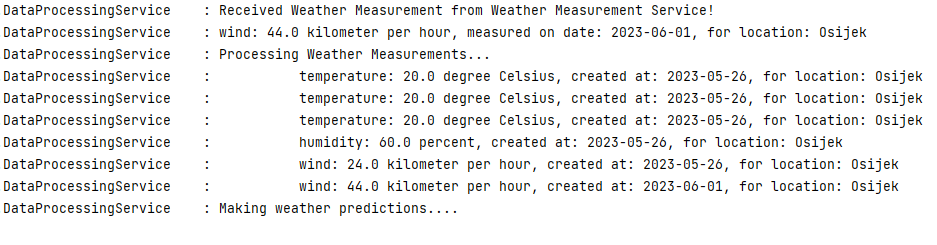


Figure 3.6: Measurement processing

# Conclusion

In conclusion, the implementation of microservices offers numerous benefits and advantages for bigger systems and applications that are seeking for a scalable, resilient, and flexible architecture. The ability for each service to be distributed and independent makes them well-suited for cloud-native and distributed systems. Microservices promote technology diversity, by opening the possibility of separate service development in different environments and programming languages, and by many technologies used for inner system communication, discovery, and routing between individual services.

However, it’s important to note that microservices implementation can also introduce additional complexity and challenges, like service coordination, complex inner system communication, data consistency, etc.

My opinion about microservices, is that the microservices architecture is a fulfilling architecture of software development, by offering various software development principles, blueprints and technologies that can be very useful in everyday engineering life of a software engineers. But, as everything in life, microservices also need to be taken with a grain of salt and need to be thoroughly planned and assessed as to whether they are needed considering the size and complexity of the required system.