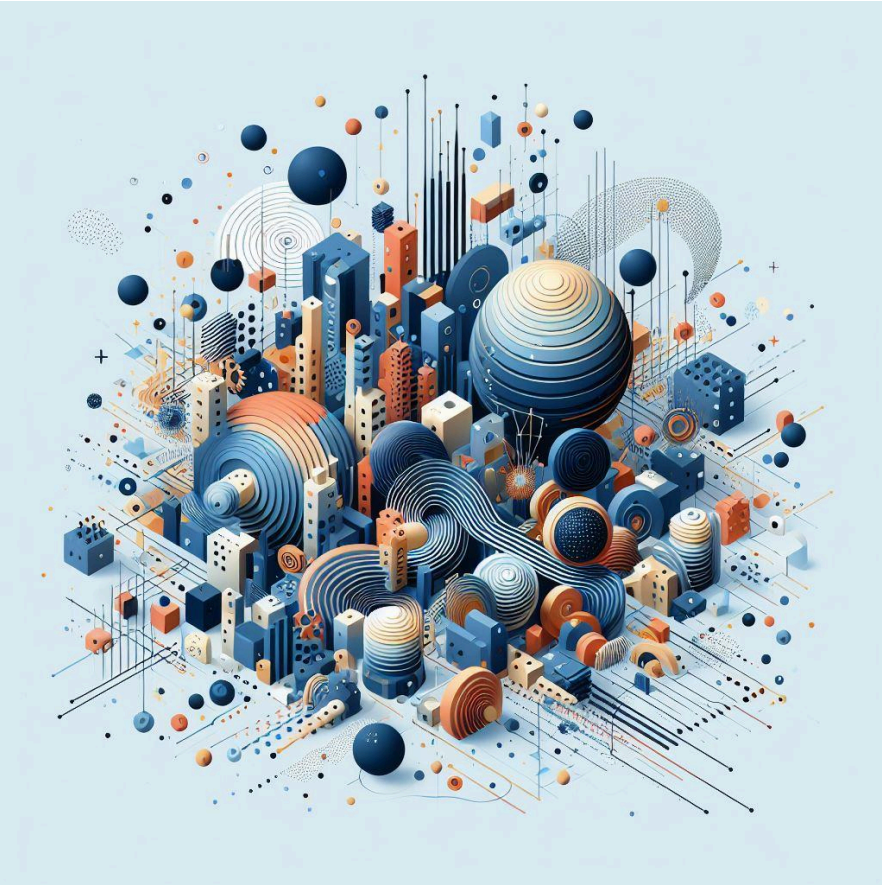
Relazione Assignment 5



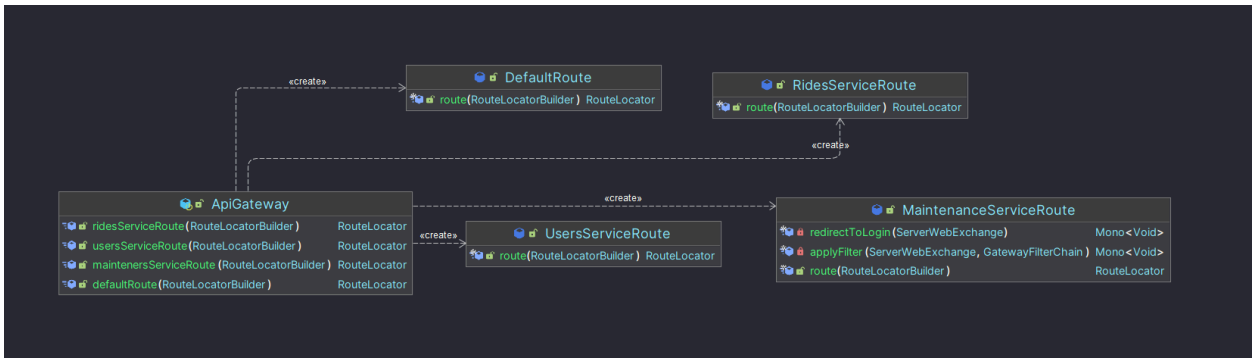
Structure of the project

To implement a microservices architecture based on the Escooters case study, I decided to implement four different services:

- The API Gateway service, hosted on the port 8080;
- The Rides service, hosted on the port 8081;
- The Maintenance service, hosted on the port 8082;
- The Users service, hosted on the port 8888;

These services are each one in a different module of the Assignment 5 project folder. Each one of them is properly accessible from a specific port. All the requests that a service makes are forwarded to ***localhost:8080*** (the URL of the API Gateway). The API Gateway then proceeds to redirect these requests based on the structure of the URL. For example, a request to the URL "***localhost:8080/maintenance/create_escooter***" of the API Gateway will redirect the request to the Maintenance service on the URL "***localhost:8082/create_escooter***". I'm now going to properly describe how each of these is structured and implemented.

API Gateway



The API Gateway in this project is structured using Spring Boot and Kotlin. It uses the resilience4j library for circuit breaking to prevent failures in one service from cascading to other services. The gateway routes incoming requests to the appropriate microservices based on the path of the request.

Here's a brief overview of the structure:

1. ApiGateway.kt: This is the main entry point of the application. It defines the Spring Boot application and the main function to run the application. It also defines beans for each route in the application.

2. CircuitBreakerConfiguration.kt: This file contains the configuration for the circuit breaker. The circuit breaker is designed to prevent failures in one service from cascading to other services.

3. Route classes (DefaultRoute.kt, MaintenanceServiceRoute.kt, RidesServiceRoute.kt, UsersServiceRoute.kt): Each of these classes defines a route in the API Gateway. They each contain a route function that takes a RouteLocatorBuilder and returns a RouteLocator. The route function defines the path for the route and any filters that should be applied to requests on that route.

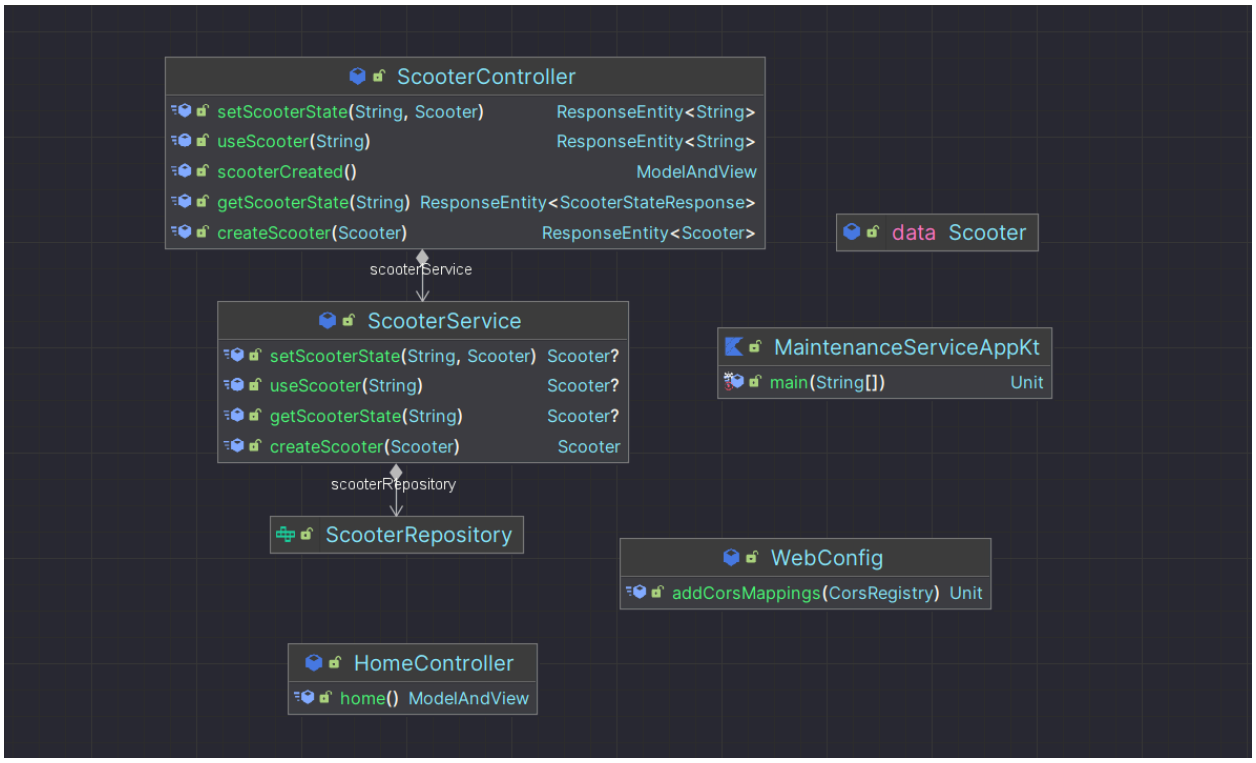
The routing in the API Gateway is managed by the Spring Cloud Gateway. Each route class defines a RouteLocator bean that is used by the gateway to route incoming requests. The RouteLocator is created using a RouteLocatorBuilder and defines the path for the route and any filters that should be applied to requests on that route.

The filters are used to modify the request or response in some way. For example, the MaintenanceServiceRoute class defines a filter that checks if the request is a PUT request and logs the request. It also checks if a certain cookie is set for non-PUT requests and redirects to a login page if it is not.

The uri method is used to specify the destination of the route. This is typically the address of a microservice that will handle the request.

In case of any failure in the microservice, the circuit breaker comes into play. If the failure rate goes beyond a certain threshold, the circuit breaker opens and starts failing fast for a certain duration. After that, it allows a few requests to pass to check if the downstream service is healthy again. If it is, the circuit breaker closes, otherwise it opens again. This prevents failures in one service from cascading to other services.

Maintenance Service



The Maintenance Service is a Spring Boot application written in Kotlin. It's structured in a typical Spring Boot way, with separate packages for controllers, models, repositories, and configuration.

1. MaintenanceServiceApp.kt: This is the entry point of the application. It contains the main function which starts the Spring Boot application.

2. controllers package: This package contains the HomeController and ScooterController classes. HomeController handles the root ("/") GET request and returns the "maintenance" view. ScooterController handles various requests related to scooters by referring to the ScooterService, such as creating a scooter, getting all scooters, getting a scooter's state, setting a scooter's state, and using a scooter.
3. models package: This package contains the Scooter data class, which represents a scooter with fields for id, name, location, and state.

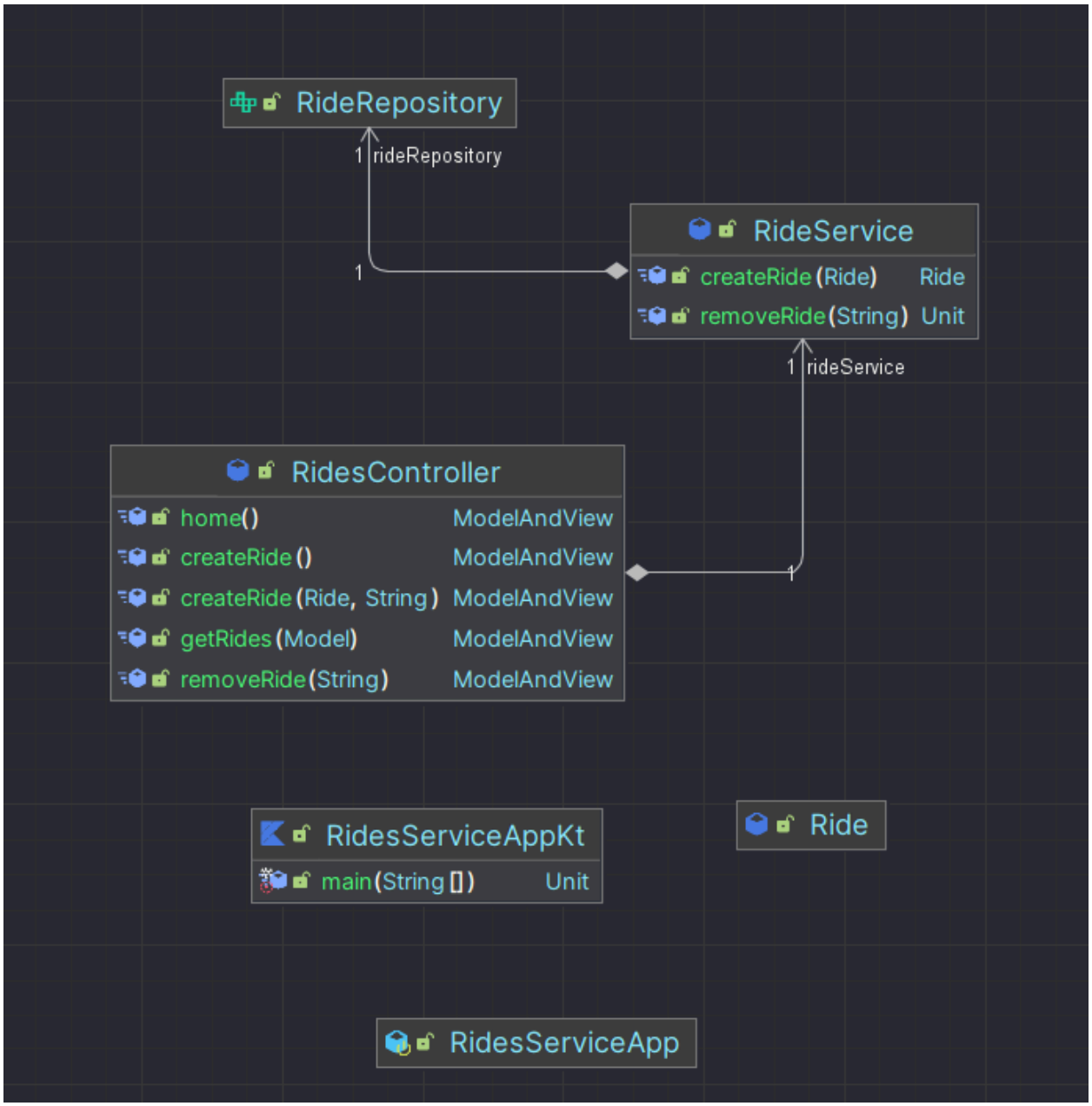
4. services package: This package contains ScooterService.The ScooterService is a service class in Kotlin, part of the Spring Boot framework, that encapsulates the business logic related to the Scooter entity.

5. repositories package: This package contains the ScooterRepository interface, which extends MongoRepository. This provides methods to perform CRUD operations on the "scooters" collection in MongoDB.

6. config package: This package contains the WebConfig class, which is a configuration class that allows CORS from all origins, headers, and methods.

7. resources/templates directory: This directory contains HTML templates for various views, such as "maintenance", "escooter_created", and "escooter_error".

Rides Service



The RidesServiceApp is a Spring Boot application that manages rides for e-scooters. It consists of several components that work together to provide the functionality of the application. Here's a brief description of each component and how they interact:

1. RidesServiceApp.kt: This is the entry point of the application. It contains the main function which starts the Spring Boot application. The @SpringBootApplication annotation indicates that it's a Spring

Boot application, and the @EnableWebMvc annotation enables support for the MVC pattern.

2. Ride.kt: This is the model class that represents a ride. It's annotated with @Document, indicating that it's a MongoDB document. The Ride class has several properties including id, startLocation, endLocation, startTime, and endTime.

3. RideRepository.kt: This is an interface that extends MongoRepository, providing CRUD operations for Ride objects. It's used to interact with the MongoDB database.

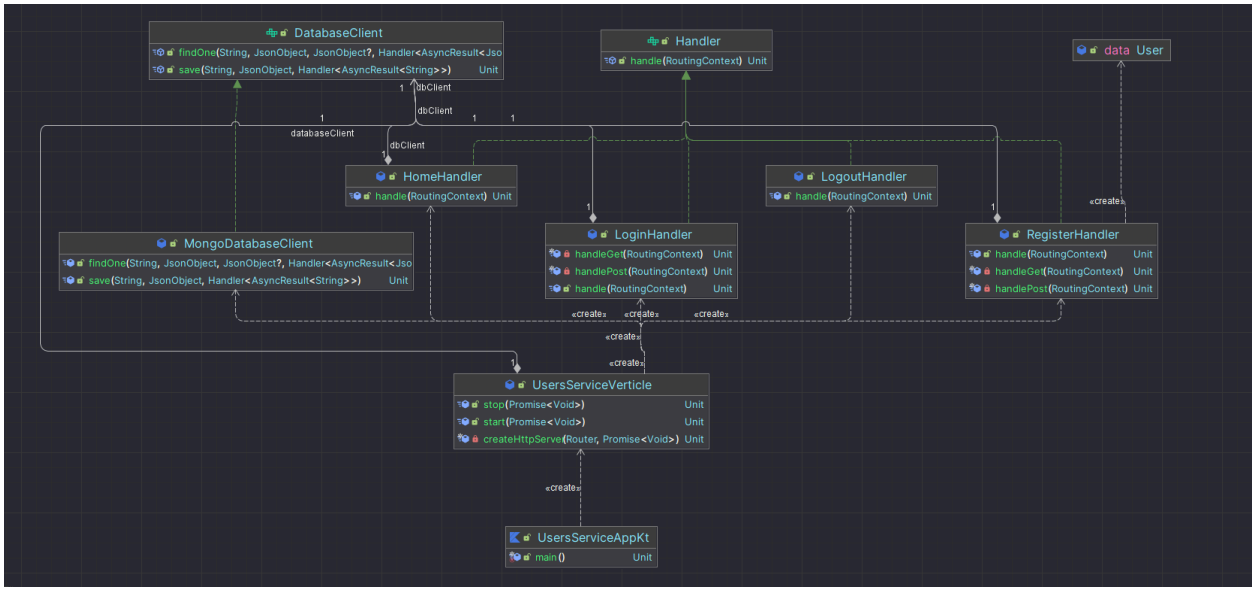
4. RideService.kt: This is a service class that contains business logic related to rides. It uses RideRepository to interact with the database. The RideService class provides methods for creating a ride, getting all rides, and removing a ride.

5. RidesController.kt: This is a controller class that handles HTTP requests. It uses RideService to perform operations related to rides. The RidesController class provides endpoints for creating a ride, getting all rides, and removing a ride.

The components work together as follows:

- When a HTTP request is made to the application, it's handled by the RidesController.
- The RidesController uses the RideService to perform operations related to rides.
- The RideService uses the RideRepository to interact with the MongoDB database.
- The Ride model is used to represent a ride in the application and in the database.

Users Service



The UsersServiceApp is a Vert.x application that provides user-related functionalities such as registration, login, and logout. It consists of several components that work together to provide these functionalities. Here's a brief description of each component and how they interact:

1. UsersServiceApp.kt: This is the entry point of the application. It creates an instance of Vert.x and deploys the UsersServiceVerticle.

2. UsersServiceVerticle.kt: This is a Verticle, a basic component of Vert.x, that sets up the HTTP server and routes for the application. It initializes the MongoDatabaseClient and sets up the routes for the application using a Router instance. Each route is associated with a specific handler that processes the HTTP requests.

3. MongoDatabaseClient.kt and DatabaseClient.kt: These classes provide an interface and implementation for interacting with the MongoDB database. They provide methods for finding and saving documents in the database.

4. Handlers: These are classes that handle specific HTTP requests. They include HomeHandler, LoginHandler, LogoutHandler, and RegisterHandler. Each handler processes a specific type of request,

such as GET or POST, and performs the necessary actions, such as retrieving data from the database or updating the HTTP response.

5. User.kt: This is a data class that represents a user in the system. It includes properties for the user's name, email, password, and whether they are a maintainer.

6. HTML and JavaScript files: These are the frontend components of the application. They include home.html, login.html, registration.html, and login.js. These files provide the user interface for the application and handle user interactions.

The flow of the application is as follows:

- When a user sends a request to the application, the Router in UsersServiceVerticle directs the request to the appropriate handler based on the request's URL and method.
- The handler processes the request. This may involve reading data from the request, interacting with the database using MongoDatabaseClient, and updating the HTTP response.
- The handler sends the response back to the user. This may include HTML content from the frontend files, or data retrieved from the database.

Microservices Pattern

API Gateway

I implemented the API Gateway pattern in my project using the Spring Cloud Gateway library for a minimal configuration of the logic needed for the implementation of a basic API Gateway. The implementation of this pattern was (as previously described) made using some different Route classes that managed requests based on the structure of the URL.

Circuit Breaker

I've used the Circuit Breaker pattern to handle potential failures in my system. This pattern is used to detect failures and encapsulate the logic of handling them, allowing a system to continue operating in the face of failure.

```
Java
 @PostMapping("/create_escooter/")
  fun createScooter(@RequestBody scooter: Scooter): ModelAndView {
    return try {
      circuitBreaker.executeSupplier {
        if (scooter.name == null || scooter.location == null) {
          ModelAndView("escooter_error")
        } else {
          scooterService.createScooter(scooter)
          ModelAndView("escooter_created")
        }
      }
    } catch (e: Exception) {
      ModelAndView("error_page")
    }
  }
```

In the context of my ScooterController class, the Circuit Breaker pattern is used to wrap the calls to the ScooterService methods. This is done using the executeSupplier method of the CircuitBreaker instance.

Here's a brief explanation of how it works in this example:

1. When a request comes in, it's handled by one of the controller methods. Each of these methods uses the executeSupplier method of the CircuitBreaker instance to wrap the call to the corresponding ScooterService method.

2. If the call to the ScooterService method is successful, the result is returned and the circuit remains closed.

3. If the call to the ScooterService method throws an exception, the circuit breaker records the failure.

4. If the number of recorded failures reaches a certain threshold within a certain time window (these parameters are typically configurable), the circuit breaker trips and opens the circuit.

5. Once the circuit is open, further calls to the ScooterService method are not made; instead, the circuit breaker can return a preconfigured fallback response, or throw an exception.

6. After a certain amount of time (the reset timeout), the circuit breaker allows a limited number of test requests to pass through. If those requests succeed, the circuit breaker closes the circuit and resumes normal operation. If those requests fail, the open state is maintained for another reset timeout period.

This pattern helps to prevent an application from trying to perform an operation that's likely to fail, allowing it to continue to handle incoming requests. It also provides a way to fail gracefully when a method call fails, instead of crashing the application.

Here's a breakdown of the configuration:
- failureRateThreshold(50.0f): This sets the failure rate threshold to 50%. If the failure rate of the calls is equal or greater than this threshold, the circuit breaker will trip and open the circuit.

- waitDurationInOpenState(Duration.ofMillis(1000)): This sets the wait duration in the open state to 1000 milliseconds (1 second). This is the time that the Circuit Breaker should wait before it decides whether to allow the calls to go through (half-open state) or not.

- slidingWindowSize(2): This sets the sliding window size to 2. The sliding window is used to record the outcome of the calls. In this case, the last 2 calls are recorded.

- minimumNumberOfCalls(2): This sets the minimum number of calls to 2. The Circuit Breaker will not trip open unless it records at least this many calls.