Software Architecture and Platforms
-
5th Assignment

Simone Ceredi

03/11/2023

Contents

Chapter 1

Analysis

The fifth assignment for the Software Architecture and Platforms course is the re implementation of the E-Scooters case study using a microservices-based approach.

1.1 Requirements

The E-Scooters case study is a management system for E-Scooter rides. The entities involved are Users, E-Scooters and Rides. The objective of the assignment is to implement the management system using a microservices-based architecture and fulfil the following requirements:

- Adopt at least two different technologies;
- Apply any microservices design patterns considered useful for the case study;
- Describe and document properly the project.

1.2 Analysis and domain modelling

The management system allows the interaction between Users, E-Scooters and Rides. Users and E-Scooters can only be created, while rides are composed of a User who starts it by taking an E-Scooter for a ride, then ends it by leaving the E-Scooter. A very simple schema of the entities involved is shown in Figure 1.1.
A user has only a name and a surname, it is identified by a alphanumeric identifier. An E-Scooter only has the identifier. The ride is a bit more complex as other than an identifier it also has to keep track of the user who

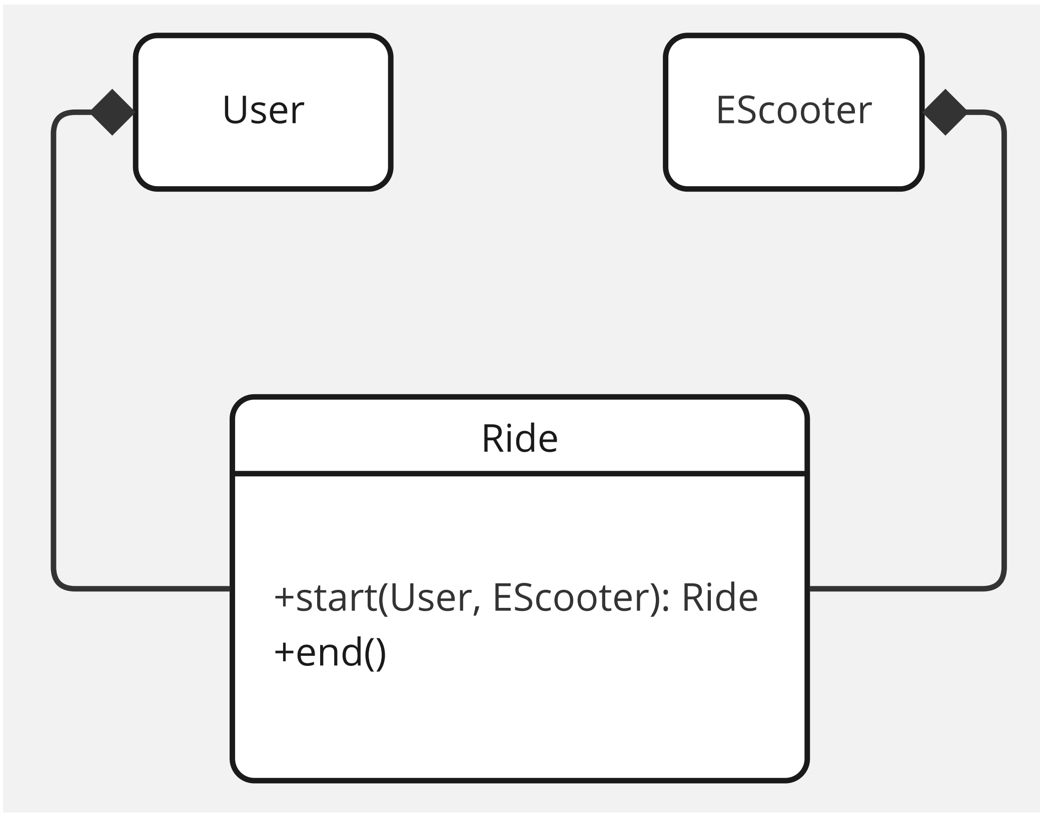

Figure 1.1: Class diagram of the involved entities.

started it and which E-Scooter is used, what day and time the ride started and, if it's finished, it also has to store the end time.
Anyone who uses the E-Scooters management system must be able to create a new user, create a new e-scooter, start and end a ride and also monitor in real time how many rides are currently ongoing.

Chapter 2

Design

This chapter focuses on the design structure of the project from an architectural standpoint and how the various services connect and communicate with each other.

2.1 Architectural design

The chosen architecture is composed of five different microservices, as shown in Figure 2.1. A client of the management system will connect to the dashboard using a browser and every request made will pass through the API gateway service, which will redirect the request to the correct service. All the requests are RESTful. The three main components of the infrastructure are the Users, Rides and E-Scooters services, each one has a dedicated server and a private database. Each is implemented using the hexagonal architecture, so each external facing component can be easily swapped for a different one.
One focal point in the design was the network structure implementation. By using the API gateway pattern, it was possible to avoid direct communication between microservices. This allowed for a very segregated network structure, as shown in Figure 2.2. While the arrows in the image may suggest that they are showing exposed ports, in reality these are the various networks that are created, a task largely simplified by utilizing Docker.

Positive elements

Utilizing this approach allows for a great amount of segregation between services, ensuring that each one can only be accessed by passing through the API Gateway. While not implemented in this simple case study, the

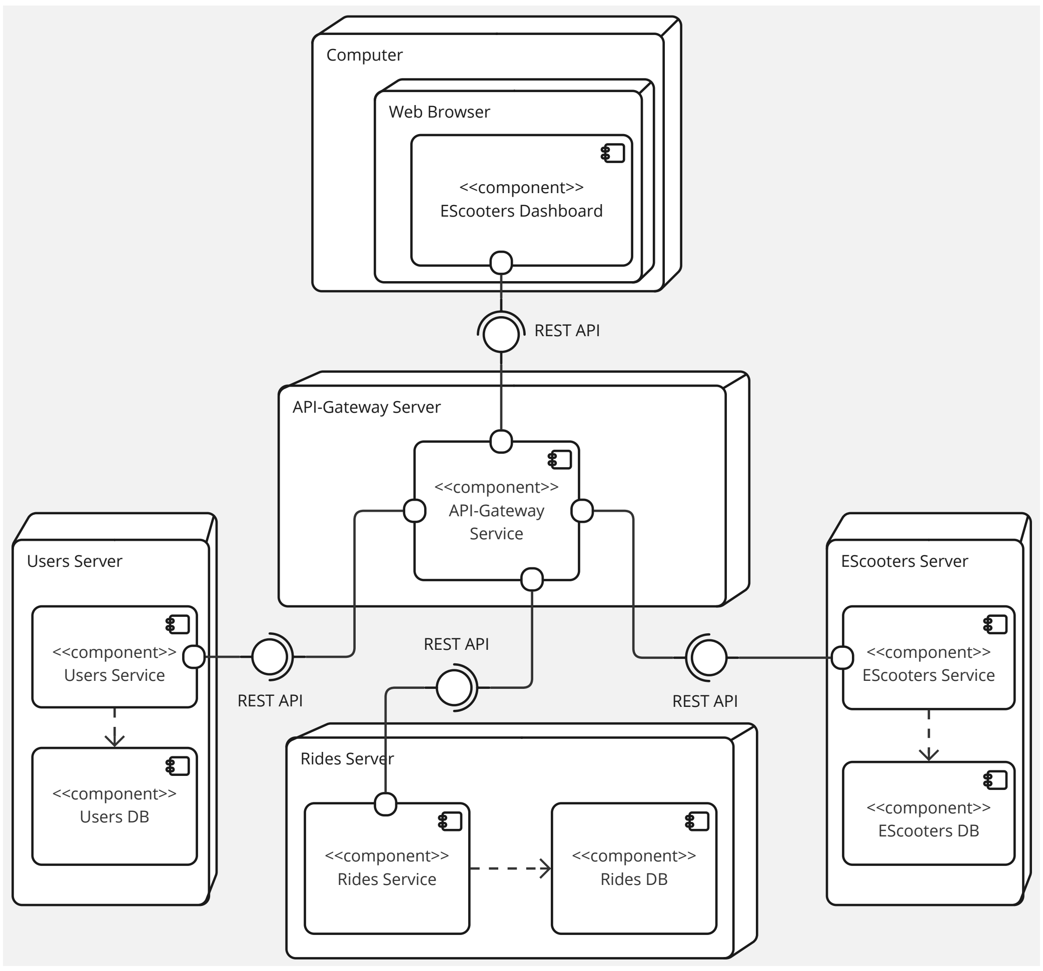

Figure 2.1: Component Diagram showing the involved microservices

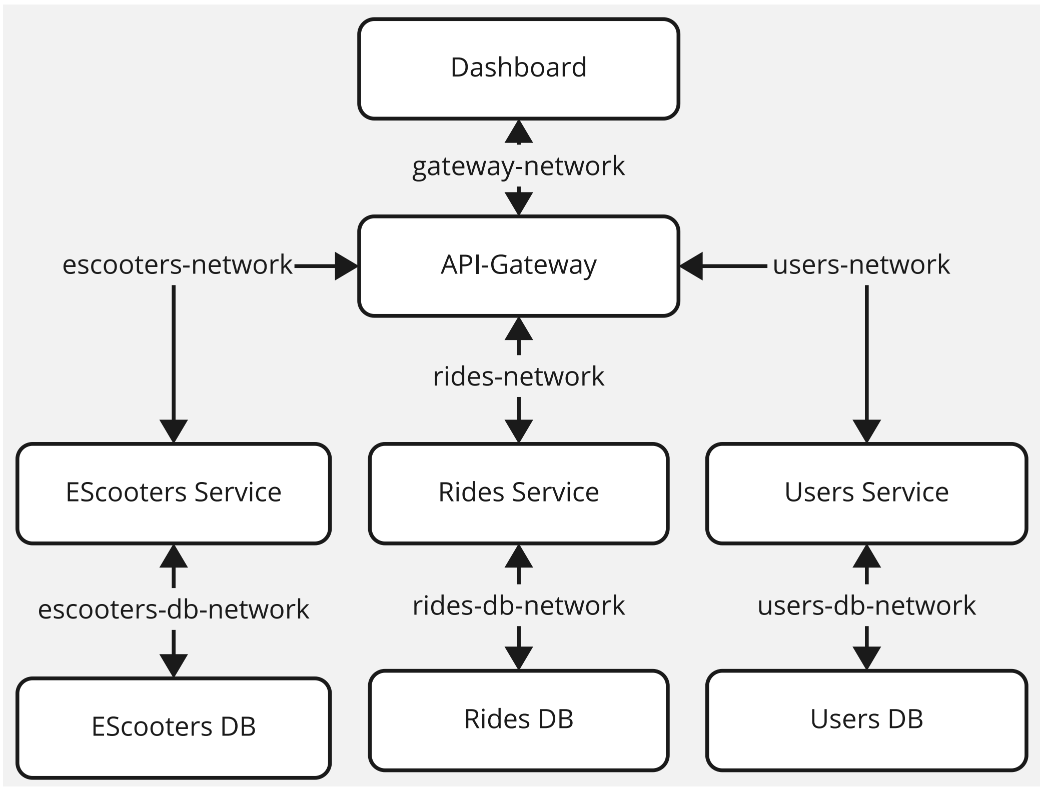

Figure 2.2: A simplified diagram of the network in which the system gets deployed

adoption of the API gateway pattern makes it possible to avoid overloading the various services, and also to scale horizontally when necessary.

Negative elements

While centralizing the network in the API gateway can be beneficial for the control of the state of the entire network, there are also some issues with the approach, as it creates a single point of failure, meaning that without that particular service active, instead of a defined portion of the application not working, the entire application will be unresponsive. Single points of failure are one of the many reasons to switch from a monolithic implementation to a microservices one; thankfully many modern technologies allow the API gateway benefits to outweigh the downsides, as they make it very easy to spin up the service after it goes down, or to maintain two different instances running at the same time, so that in case of failure of one, the other can take its place without any downtime.

2.2 API contracts

All of the API contracts are defined using the Open-API standard, they adhere to RESTful API specification up to level 2, so HATEOAS is not implemented. The use of a standardized method to describe the API allows each microservice to know what to expect when interacting with the others and also allows for easier integration testing, as the correct implementation is always the one described by the Open-API definition.

2.3 Interaction schema

The interactions that a user of the management system can have with the services always follow a procedure similar to the one described in fig. 2.3. The interactions of the user in the management system are modelled using RESTful API calls.

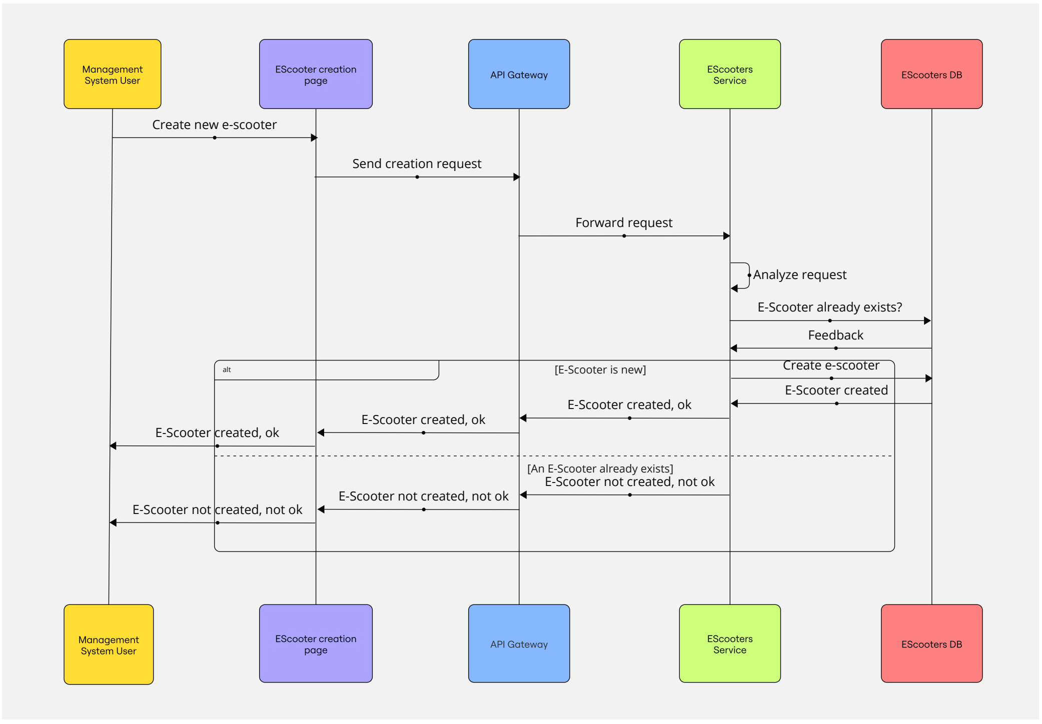

Figure 2.3: A sequence diagram of the interactions between services that occur registering a new E-Scooter

Chapter 3

Detailed Design and Implementation

Each service is designed using the hexagonal architecture, the only ones that are implemented differently are the Dashboard and the API Gateway.

3.1 Dashboard

The dashboard is the simplest service as it only has to be a web server serving some HTML pages, so its design is straightforward. The implementation of this microservice has been done in Go using the Echo framework[1] which allows for a very simple creation of web servers.

3.2 API Gateway

The API Gateway has to redirect every request to the correct microservice, since creating an extensible API gateway from scratch can be tough, the decision was made to utilize the Spring Cloud Gateway[2]. By using this library the creation of the various routes was very simple and only required a YAML configuration file.

[1]`https://echo.labstack.com/`

[2]`https://spring.io/projects/spring-cloud-gateway#overview`

3.3 Users Service

The Users Service has been implemented in Kotlin. It uses Vert.x[3] for the web service creation and communicates with both the API gateway and a Mongo database that stores all of the required user data. The implementation follows hexagonal architecture principles and is fairly simple. To test the service without requiring the database to be running, mock[4] tests are used to evaluate the correctness of the implementation.

3.4 E-Scooters Service

The E-Scooters Service has been implemented in Go. As with the Users Service, its architecture follows the design of the hexagonal architecture. It uses only two libraries: one to connect to the respective Mongo database and another to load some secrets from environment variables. The web server used to communicate with the API Gateway is implemented using only the Go standard library. Since Go version 1.22, the http package[5] is as powerful as any framework and very easy to use.

3.5 Rides Service

The Rides Service has also been implemented in Kotlin, following the hexagonal architecture. It is more complicated than the other two as it also has to manage a Web-Socket connection. The implementation resembles that of the Users Service, with the main difference being that this service also has to communicate with the other two when creating a ride, requiring a more complicated test suite. Vert.x is also used to manage the Web-Socket connection that shows how many rides are currently ongoing.

[3]`https://vertx.io/docs/vertx-core/kotlin/`

[4]`https://mockk.io/`

[5]`https://pkg.go.dev/net/http`

Chapter 4

Conclusions

The orchestration of various microservices is not a trivial problem. Without proper testing, it can be difficult to develop such applications. The implementation of microservices patterns, such as the API Gateway, can simplify the development process.

Even with a simple application like this one, it is easy to see the reasons behind the creation of a microservices-oriented architecture. While it has its drawbacks in complexity, it more than makes up for them with the ability to maintain a more stable service, and, if an error occurs, it enables most of the application to remain active. Developing the system alone, I wasn't able to fully appreciate the advantages that such an architecture provides, but the ability to have separate groups working on separate tasks in separate microservices surely has a lot of benefits in the development process. Tools such as Swagger[1] and integration tests allow for seamless multitasking in different teams, as sticking with the described interface will avoid most errors that can occur during microservices communication.

In conclusion, even if using this architecture makes it tougher to implement and orchestrate the project compared to a monolithic implementation, using a microservices architecture has many benefits that outweigh the downsides.

[1]`https://swagger.io/specification/`