RESTaurant Reservation

Leonardo Emili, Alessio Luciani, Emanuele Mercanti, Andrea Trianni

May 29, 2021

Contents

1 Problem introduction

While there are lots of services to order food deliveries, there is seemingly little choice in the market when it comes to restaurant booking services. The most widespread option to book a restaurant is to use the restaurant's website. This makes it hard to integrate the different "processes" of going to the restaurant, namely searching for where to go, reading the place information (e.g. the menù) and finally booking a table. As things are now one would use Google / Tripadvisor for the search part and restaurants own websites/phone numbers for the booking part. Getting information about the menu is seldom a feature offered by the aforementioned service providers and an actual pain-point for users. Our web app conveys all the different steps into a single, easy-to-use platform where users can search, read the menù and book a restaurant. Additionally, restaurant owners are able to create their own business profile and manage them: updating the menù when needed and accepting the reservations made by users. This last point is particularly important, since restaurants may still receive bookings from different means (phone calls, customers unexpectedly showing up, etc.)

2 Solution design

We started with a system design step. Regarding this, we realized sequence diagrams that describe the sequences of interaction between the simple frontend and the microservice-based backend. These also take into account the communications that happen among different microservices that are involved in the same use case. One example is the restaurant reservation use case that passes through an authentication microservice and another microservice that handles the restaurants' data. Here we describe the microservices that we designed and attach the corresponding sequence diagrams showing the interactions among them.

2.1 Microservices

2.1.1 User auth

The user auth microservice exposes an interface for authentication needs of the user. These are mainly related to login and registration via the frontend web app and identity validation via the backend for operations that need authentication. The main idea is to have an API that responds to requests, by operating on a NoSQL database. The database contains information about the users: Name, Surname, Email address, Password, and Session Tokens. The user registers into the system and a new entry in the database is created. When a user logs in, a new session is created by generating a unique token. This token is stored in the database and sent to the frontend to be cached. The system is purely RESTful and thus stateless. Every request is authenticated via the auth token.
The available operations are:

- Login (Figure 1): The client provides email and password (or token) and receives a session token if the user is registered.
- Registration (Figure 2): The client provides name, surname, email and password, and receives a session token after being registered in the database. The operation fails if a user with the same email already exists.
- Logout: The current login session is canceled.
- Token validation: The calling microservice (the one that needs a request to be authenticated) provides the email and session token of the user that initiated the operation and receives a validation or an error, depending on the validity of the token. This operation is shown in Figure 5.

2.1.2 Restaurant auth

The restaurant auth microservice exposes an interface for authentication needs of the restaurant. It is very similar to the one of the user, but it is separated to better divide contextes. In fact, the scalability needs of the two authentication services are quite different. After the release of such a system, the increase in

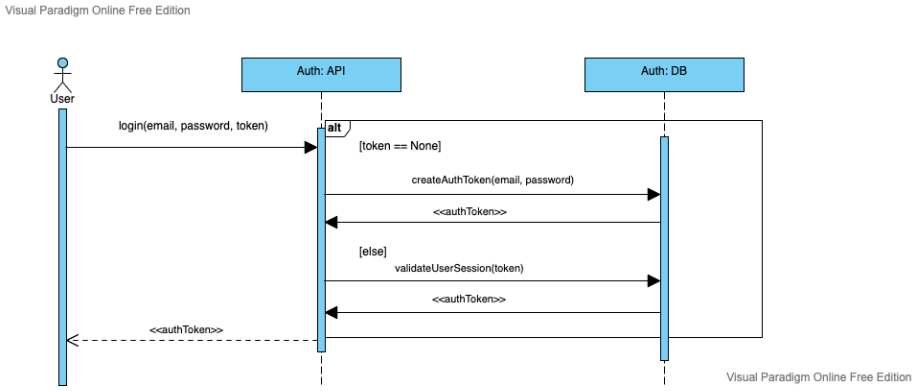

Figure 1: User login sequence diagram.

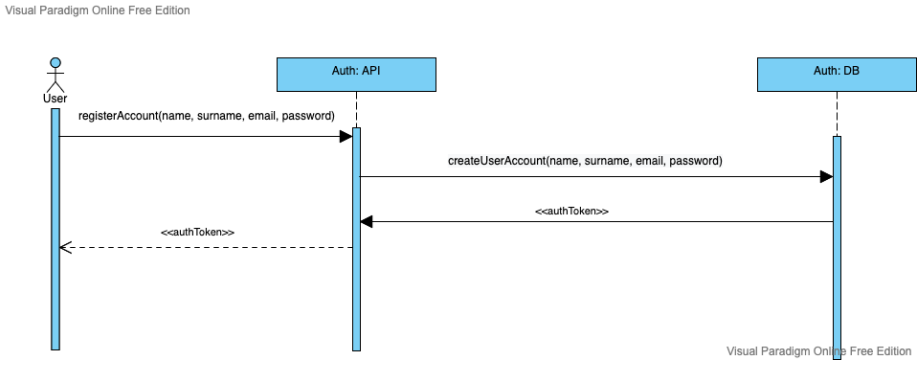

Figure 2: User registration sequence diagram.

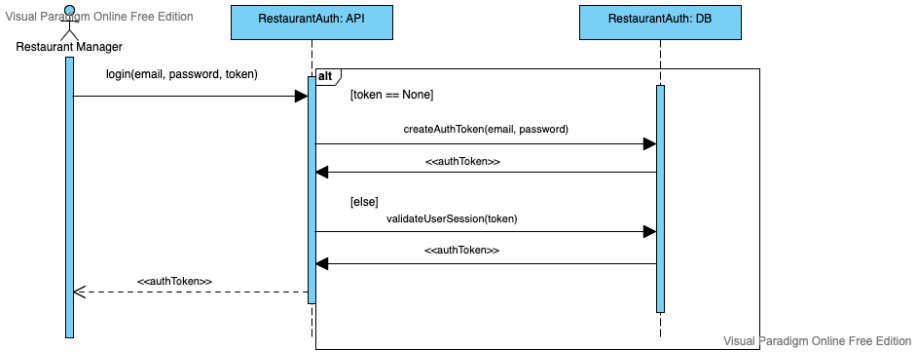

Figure 3: Restaurant login sequence diagram.

the number of users is expected to be much more significant than the one of the number of registered restaurants. This division guarantees individual scalability. Also in this case there is an API that responds to requests, by operating on a NoSQL database. The database contains authentication information about the restaurants: Name, Email address, Password, and Session Tokens. The tokens work the same way they do in the user auth module.
The available operations are:

- Login (Figure 3): The client provides email and password (or token) and receives a session token if the user is registered.
- Logout: The current login session is canceled.
- Token validation: The calling microservice (the one that needs a request to be authenticated) provides the email and session token of the restaurant that initiated the operation and receives a validation or an error, depending on the validity of the token. This operation is shown in Figure 6.

2.1.3 Restaurant data

2.1.4 Booking

Reservation (Figure 5)

2.2 Scalability

By structuring the system into many microservices, there is the possibility to scale the single microservices indipendently. This is useful since the microservices may receive different loads of incoming traffic and this way there is more control over which component should be scaled.

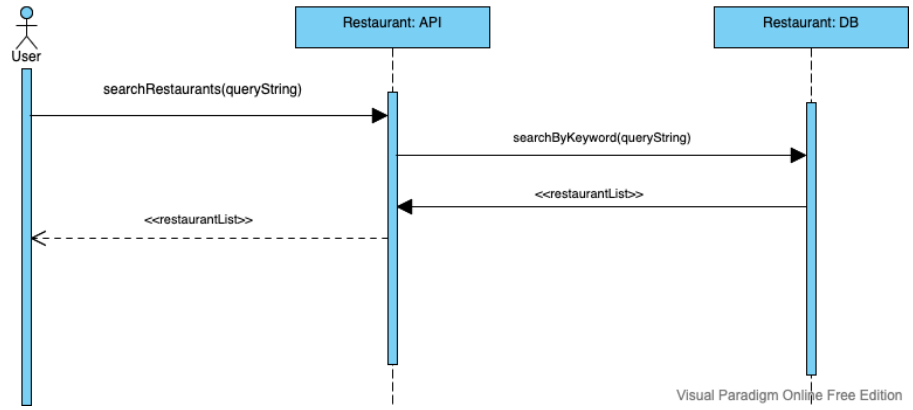

Figure 4: Restaurant search sequence diagram.

Figure 5: Reservation sequence diagram.

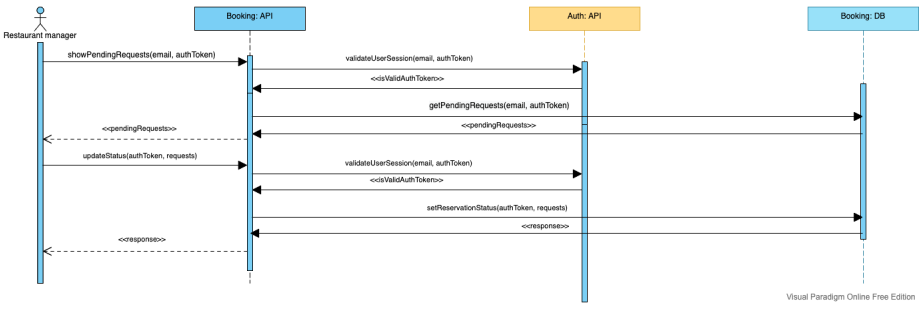

Figure 6: Booking management sequence diagram.

3 Solution implementation

Here we discuss the implementation of our solution. As already anticipated, the core of the system is a microservice architecture that communicates through the REST API. Since every microservice is independent of each other, the technologies that are used internally in the various modules do not have to be the same ones. So, apart from keeping a standardized REST interface among the components, we could develop the microservices with different programming languages and frameworks. Here are described the specific implementations of the single services.

3.1 Microservices

3.2 User auth

This module was built using Typescript that runs on a Node.js environment after being compiled into Javascript. The corresponding database was built using MongoDB that provides a NoSQL structure. The module uses the Express.js library to expose its REST API and the Mongoose library to communicate with the database. The POST http method was used to perform operations that edit entries in the database, such as during the registration. The GET http method, instead, was used for operations that do not modify content in the database, but only read information, such as the token validation.

Listing 1: User auth exposed API

```
POST register
JSON body data
    name: string
    surname: string
    email: string
    password: string
JSON response
    token: string OR error: string


POST login
JSON body data
    email: string
    password: string OR token: string
JSON response
    token: string OR error: string

POST logout
JSON body data
    email: string
```

```
    token: nullable string
JSON response
    token: string OR error: string


GET validate
JSON body data
    email: string
    token: string
JSON response
    token: string OR error: string
```

3.3 Restaurant auth

This module was implemented very similarly to the user auth one.

Listing 2: Restaurant auth exposed API

```
POST register
JSON body data
    name: string
    email: string
    password: string
JSON response
    token: string OR error: string

POST login
JSON body data
    email: string
    password: string OR token: string
JSON response
    token: string OR error: string


POST logout
JSON body data
    email: string
    token: nullable string
JSON response
    token: string OR error: string


GET validate
JSON body data
    email: string
    token: string
```

```
JSON response
    token: string OR error: string
```

3.4 Restaurant data

3.5 Booking

3.6 Web application

For the purpose of the project, we also implemented a web application that serves as a showcase to test the functionalities of our system. The backbone of our website is composed of elements of HTML and CSS, tied up using *V*ue.js to add the responsive component. To enforce types, we opted to use TypeScript in strict mode instead of the plain Javascript. We decided to use *A*JAX to send requests from the client to the servers. The interface is quite simple and implements the most important use cases that may be useful in the context of restaurant reservations, namely, we implemented the following use cases: user authentication, restaurant owner authentication, restaurant search using keywords, reserve a table, and handle an incoming user booking for a table. For a full description of how these functionalities are implemented please refer to their documentation in the microservices section. In particular, all requests that need to be authenticated are provided with an authentication token relative to the user session. Once the user is authenticated by our system, we store the authentication token that is provided by the user auth API in the local storage object. In this way, we are able to maintain the user session open even if the browser was closed. In such cases, we perform a technique known as silent sign-in using the above-cited token-based technique, enabling the user to proceed with their reservation transparently. For security reasons, we do not store user passwords on the client-side, neither encrypted ones, and the random generation of authentication tokens is performed on the server-side. Moreover, the website allows the non-authenticated user to view all available restaurants and see the details of a particular one, exception made for the booking request that can only be performed by a user that is currently logged into the system. A logged user can also see the list of their reservations, that have to be accepted by a registered restaurant owner. On the other side, we allow restaurateurs to see the collection of pending reservations that need to be accepted. Once the status of the reservation is changed on the restaurant side, the change is reflected in the user dashboard as well. It's worth noting that even though in this context the web application is only used to test the real system functionalities (implemented in the back-end), future work could be in the direction of improving the user experience on the website.

4 Solution deployment

Description of the deployment of the solution

Description of docker compose...

5 Test design

Test/validation design (i.e. description of how you will test/validate your solution and why you decided to do that)

6 Experimental results

Experimental results (i.e. the description of the results obtained with your test/validation phase))