Assignment 1

OP3-SEM6

Eva Noritsyna, Carlotta Lichtenauer, Annabel Simons,
Mathanrajan Sundarrajan, Yuanze Xiong and Matteo Fregonara

January 8, 2021

1 Task 1: Architecture of our project

The architecture of our project utilizes the microservices architectural pattern. (See Figure 1). We first developed the microservices as three separate standalone services, which are all connected to individual databases, and then integrated them with RESTful communication. This means that one microservice can call another microservice to get more data that it needs to complete a request. For that, we have used WebClient instead of RestTemplate for its non-blocking features. The way we divided the functionalities between the services depended greatly on minimizing data duplication in their respective databases.

The entry point to the system goes through the authentication service which, after the user has successfully logged in, generates a unique, temporary JSON Web token. The authentication with the token assignation is done through Spring security. This token gets passed on to all other services and has to get validated by communicating back with authentication service anytime a user makes a call to a component, for example when enrolling into a course. The way the JWT is validated when called from the course or room microservice is through an internal filter. With every HTTP request coming into either of the microservices, the request has to first go through the filter which internally makes an API call to the authentication microservice's JWT validation interface. Additionally, the authentication service manages and store information about users.

The course service consists of two main components, the "Enrollment", and the actual course management component. Enrolling in a course is a separate component since it implements its own set of closely related functionalities. In its logic, it makes an individual call to the provided interface of the authentication microservice. The course management component mainly focuses on the other functionalities within courses. It is greatly dependent on the room microservice and therefore requires a data access interface from it. The course service provides an interface to the room microservice since that gets updated whenever courses change their data related to the room microservice. The room microservice mainly exists to provide data to the authentication and course microservice. All data requests in all three microservices are handled through API calls which are closely connected to the individual databases.

2 Task 1: Responsibility and role in the overall infrastructure

The authentication microservice has three main roles; it provides authentication, stores user information, and validates access to other microservices. That means that the entry point to the course management system is through the authentication microservice. After successful authentication, the service provides the user with a JWT, generated by the SpringSecurity infrastructure. The token includes the "userID" as well as the role of the logged-in person, which helps when the service does role-based authorization. Besides, the service provides one interface which allows other microservices to request a role-check for a specific user, which is needed when a teacher is trying to add another teacher to a course, to check that the person they are trying to add is indeed a teacher. Similarly, it works when a teacher enrolls a student into a course to make sure it is a student.

The course microservice has three main roles; providing a course overview, course management, and enrollment. The overview responsibility communicates solely with the course's persistence service and the authentication microservice to validate the user's JWT and only allow access to certain methods based on the role. Similarly, the enrollment component also requires a role check interface from the authentication service, however it uses the explicitly provided one as this time the role that is being checked is of a different "netID". The course management component also communicates with the room microservice, to get data about room availabilities, and to update the availability of the chosen room.

The room microservice provides information about room availability to the course microservice and can only be directly interacted with by a teacher, to receive an overview of all available rooms. This means that it also has the communication with the authentication microservice's JWT validation interface which goes through with the role-based authorization.

3 Task 1: Motivations for the architecture and each individual micro-service

When deciding how to split our application into different microservices, we started by identifying the main entities our system must include: room, course, user, and enrollment. Based on this list, we proceeded to partition the application, choosing to group the course and enrollment entities under a single microservice. This was due to courses and enrollments being closely linked together. Furthermore, we deemed that having separate services for both courses and enrollments was redundant and required excess communication between the two services.

The authentication microservice, as mentioned above, also takes care of functionalities relating to users. Similar to how we combined the functionalities relating to both courses and enrollments under a single service, we decided that users and authentication are closely related and therefore should be grouped.

Finally, we decided to have a room microservice to extract the functionalities relating to the rooms from the rest of the application, thus improving

modularisation and making the codebase easier to understand.

In general, we chose to group several correlated entities and functionalities under a single microservice, such as in the case of the course and user services, to avoid an excessive partition of the application when it was not deemed necessary. This provides several advantages, such as the aforementioned reduced need for excess communication between services, resulting in simpler implementations, less data duplication, and a reduced risk of facing communication issues between microservices.

4 Task 2: Design pattern choices

4.1 Strategy Pattern

We choose to implement the Strategy Pattern in our system. (See Figures 2 and 3). Looking at our requirement list we noticed that we had multiple requirements related to giving an overview in the course microservice, as well as in the room microservice.

In the course microservice, we have to give a general overview of all courses there are for teachers. Teachers have to have an overview in which courses they are involved and students have to have an overview in which courses they are enrolled. We as well implemented a sorted course overview, which sorts the courses descending on their maximum amount of students.

For the room microservice, the teacher has to have an overview of all rooms there are, as well as all non-booked rooms. Furthermore, a teacher can get the room overview related to a specific course. Lastly, the teacher can retrieve a specific room.

For both microservices, we created a "strategy" folder with a Strategy interface in it. As well as, the Concrete Strategies classes which implement the actual behavior of the strategy interface. For the course microservice, we decided to divide these classes into "GetAll", "GetAllSorted", "GetInvolvedInTeacher", and "GetInvolvedInStudent", since as described above these are the four main different behaviors. In the room microservice we divided the Concrete Strategy classes into "GetAllRooms", "GetEmptyRooms", "GetCourseRooms", and "GetRoom".

When the overview logic is needed the individual controller calls the corresponding Context class. This context class stores a reference to the Concrete Strategies classes. By storing this reference the context class puts all the work in the strategy instead of executing it on its own.

Beforehand, we had all the logic in the corresponding microservice controller. Since we now created a different folder with all of the overview logic in it our code is more structured and easier to read.

4.2 Chain of Responsibility Pattern

For the other design pattern, we chose to implement the Chain of Responsibility Pattern. (See Figures 4 and 5). While looking at our implementation of "createCourse" logic, we found that there was a chain of checks handled by if statements related to different objects. Meaning that something had to be true to be able to get to the next check. This logic can be converted into a chain of handlers that individually process a request, and can be reused for different requests. It is upon every individual handler to decide if they approve the request and pass it along to the next handler, with it not being clear beforehand which object will handle it. The chain of responsibility pattern makes the if-logic very organized and clear.

When creating a new course, there are multiple conditions you need to check. For instance, you need to check whether or not the course code is null. If it is, then you have to throw an exception to inform the user/client that they are trying to create an invalid course. Some of the other things you need to check are whether the room is available for the course or if this course already exists. For the checks related to enrollment logic, some of the checks are handled by logic related to courses and some are handled by checks related to authentication service so these are the ones present in the chain of responsibility for enrollment logic.

To implement the Chain of Responsibility Pattern, we first had to create an Interface with two functions. "setNext" which sets which validator should be called after the current validator, and "handle" which handles what the validator does. We create an abstract method that implements the "checkNext" and "setNext" methods, which are common to each of the concrete validators. The concrete classes (validators in their namings) extend the abstract base validator and implement the "handle" method which performs various types of checks.

To keep the code organized, we have stored all the classes and interfaces related to this design pattern inside its package called "validator". As we have now implemented the Chain of Responsibility Pattern, the code which dealt with the checks is much more short and clean now.

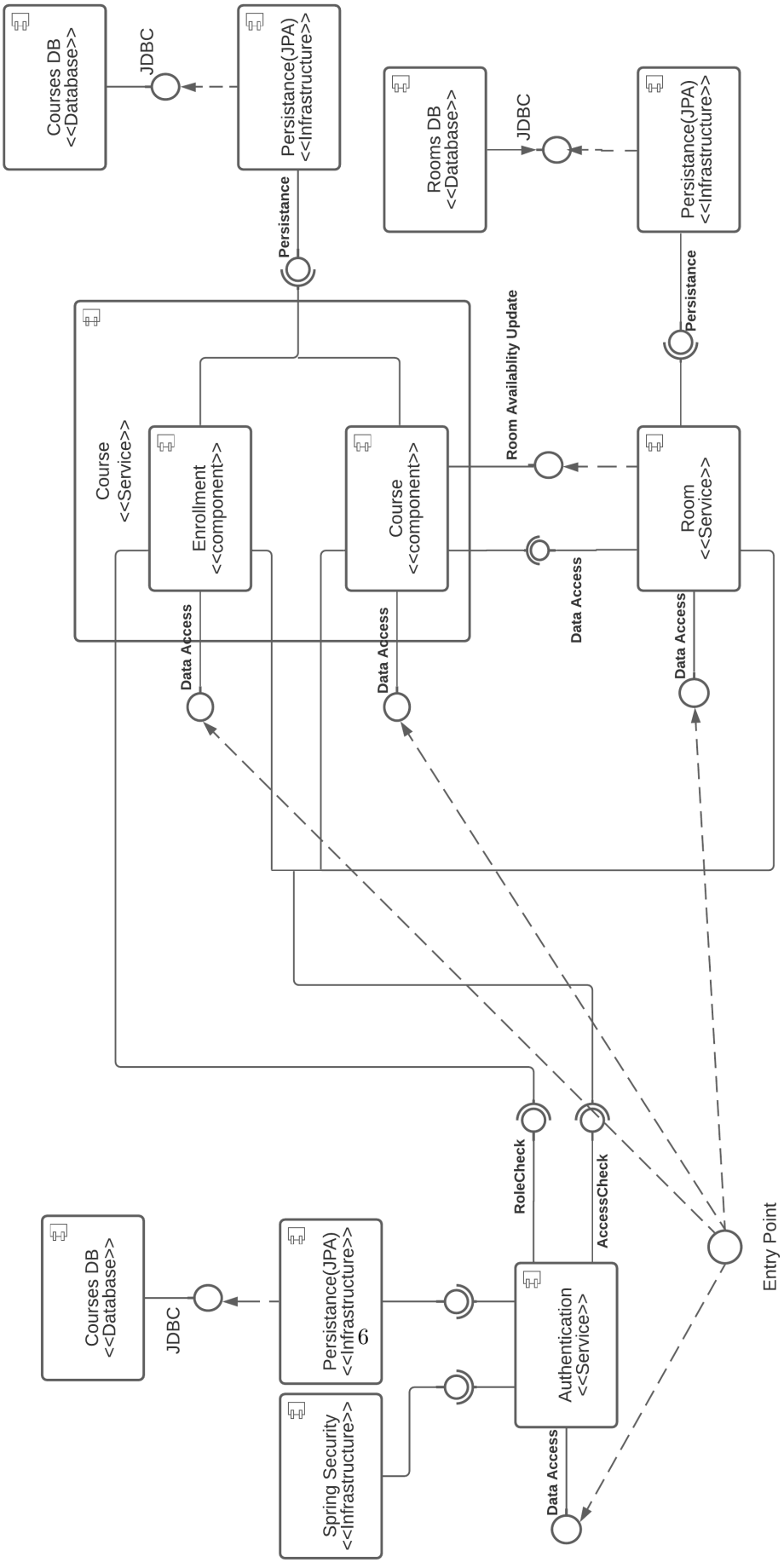

Figure 1: Component Diagram for the microservice architecture.

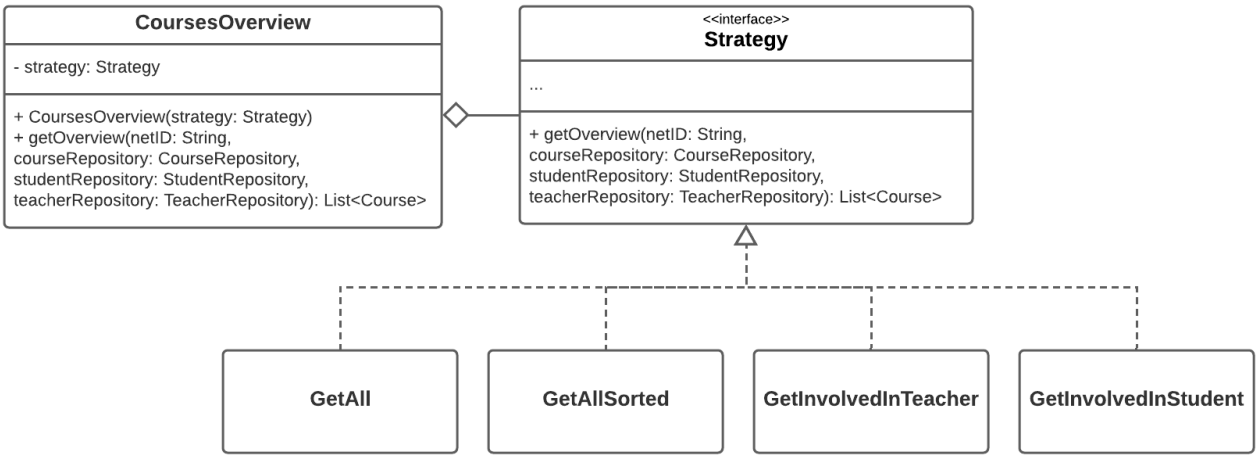

Figure 2: Class Diagram for the course microservice's Strategy Pattern.

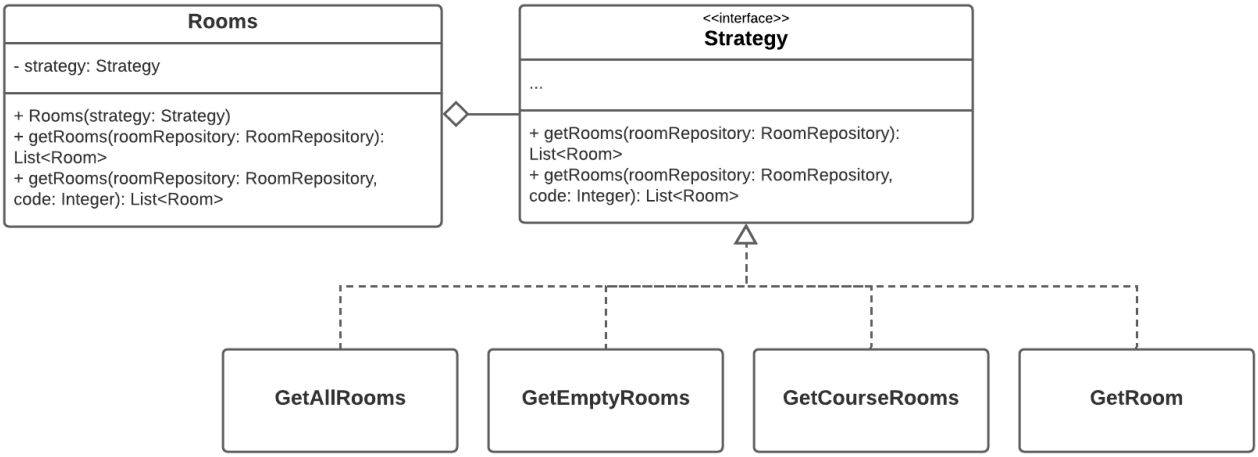

Figure 3: Class Diagram for the room microservice's Strategy Pattern.

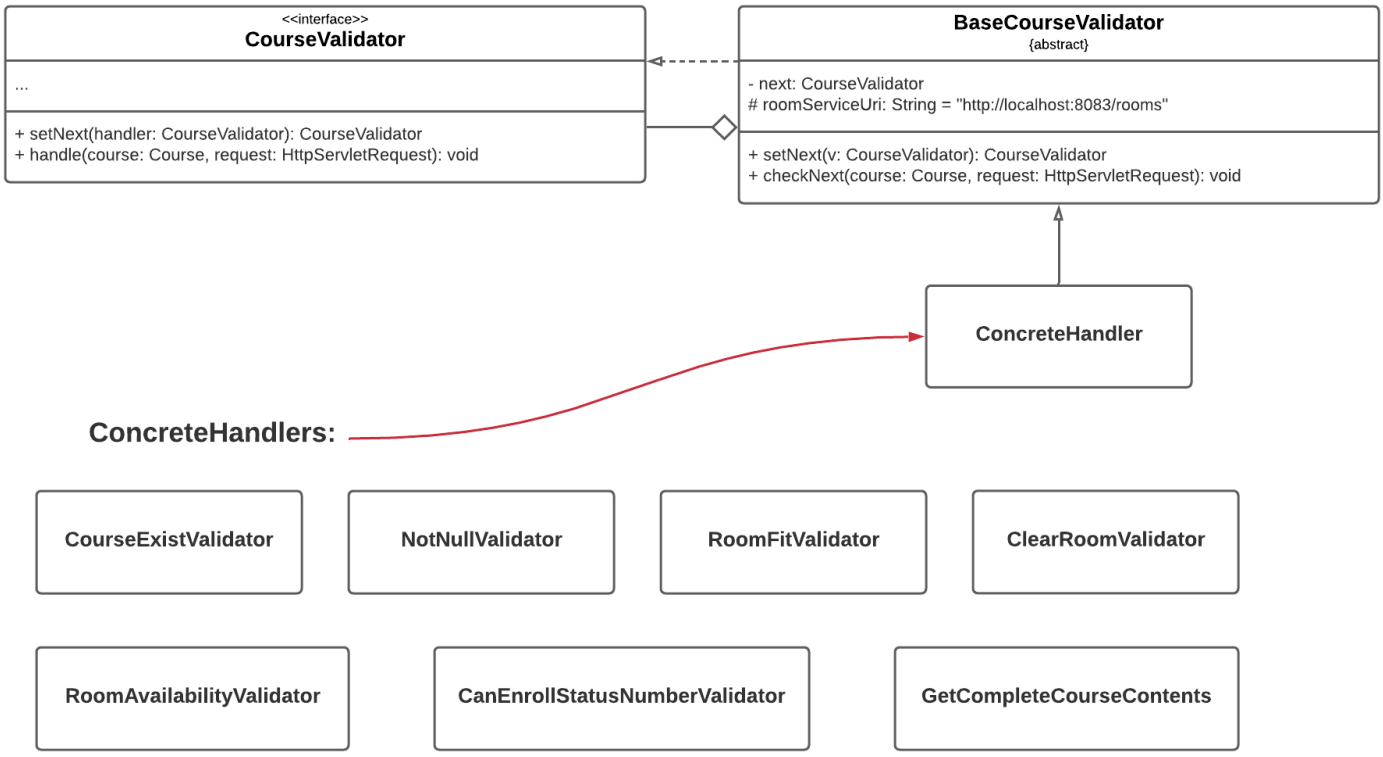

Figure 4: Class Diagram for the course microservice's Chain of Responsibility Pattern.

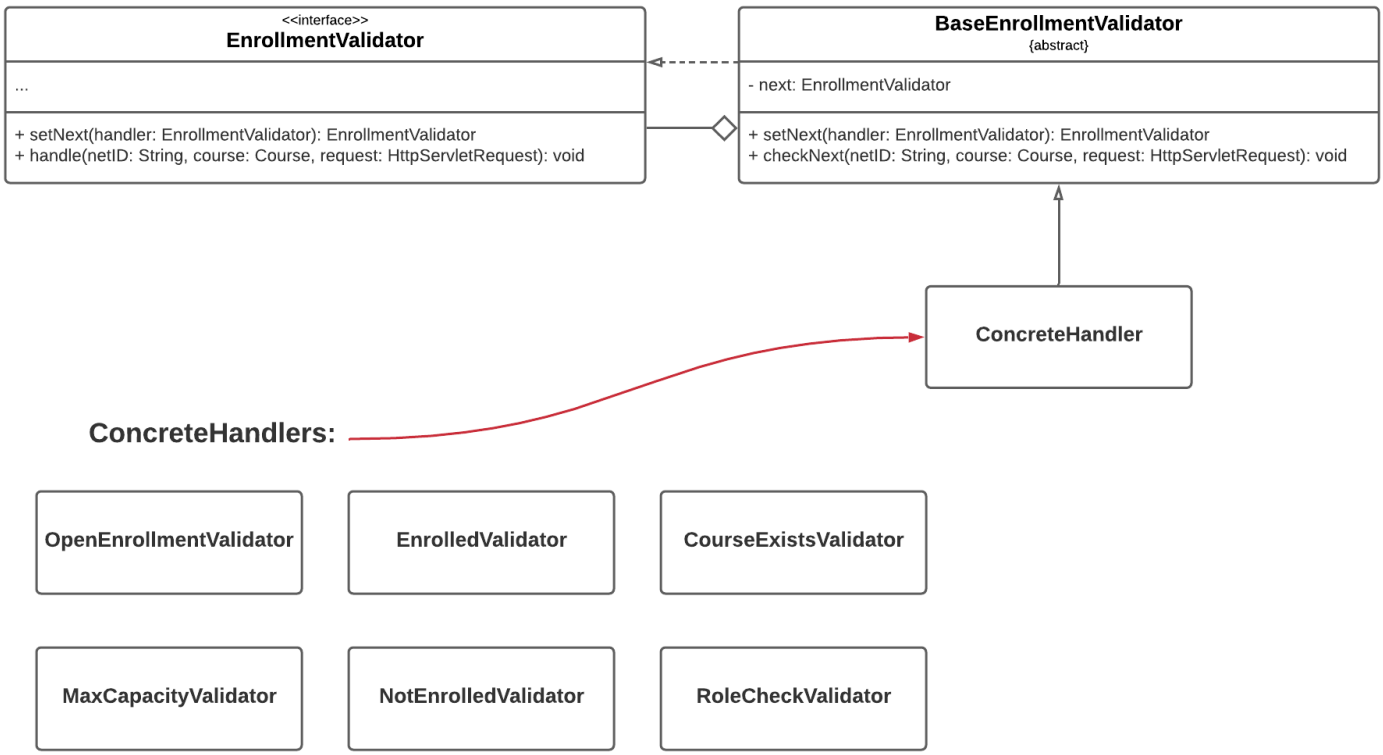

Figure 5: Class Diagram for the course microservice's Chain of Responsibility Pattern used for enrolling.