Group-Project-8-OP29-SEM57-Assignment-1

Timen Zandbergen
Merdan Durmuş
Luca Becheanu
Can Parlar
Alexandru Bobe
Matthijs de Goede

Task 1: Software Architecture (30 pts)

The coronavirus pandemic has been challenging for us all, including educational institutions, on which it has a major impact. How can such institutions give each student a maximized and equal amount of physical participation opportunities, according to the guidelines set by the government? To answer this question, our group was given the assignment to implement a corona-proof room scheduling program, using the REST API and the microservice architecture. Starting our development process, we decided on our problem requirements and tried to divide the system into microservices based on the main components of our solution.

We used domain-driven-design techniques to design our system. By the nature of the microservice architecture, we wanted each service to be able to work in pure isolation. On the contrary, different services should also be able to work together in harmony, communicating with each other and the client. We wanted to have approximately equal-sized services. For some components of the system, it was obvious that they required a separate microservice.

An example being the ‘Identity’ service. As one of the main requirements was to have validation functionality so that only validated users could perform API requests, which has nothing to do with the actual functionality of the system, we decided to create a separate microservice, that functions as a “gatekeeper” and can be used by all microservices for authentication and authorizations of users.

The ‘Identity’ microservice has access to a database that contains the netIDs and the passwords of all the users. Before being able to use the application, any client has to authenticate, by providing their netID and their password. If the combination is found in the database, the ‘Identity’ microservice gives the client a limited-time session token that they have to use for every further request and is validated using public-key cryptography.

Secondly, authorization is coupled with authentication. Every user has a role: teacher or student. And each role has a different set of permissions and restrictions. The server uses authorization to verify that the client that made a specific request has the permission to access the resource. Again, the role of a user is embedded in the token. Therefore, before giving a response, the server has to check the token.

We tried to balance the resources used with the level of protection needed. In order to be sure that the algorithms used for authentication, authorisation and encryption are reliable we used the Spring Security framework and JWT Web Token.

Now let's look at the key functionality of the system. We knew that in order to schedule lectures, teachers should be able to create courses and assign students to it. To manage all course-related information we decided to create a ‘Course-Management’ service. While brainstorming about this division we also thought that we should have a service to manage lectures. However, we later on decided that the high coupling between lectures and courses would cause inefficiency if we had them in two different services.

The ‘Course-Management’ microservice allows teachers to create new courses and store them, these courses are also linked to a list of student NetIDs (given by the teacher). These students are enrolled in this course and will be given an opportunity to participate in lectures physically and online. The service also lets teachers create lectures to be scheduled for this course. The lectures have a fixed duration and date and are stored in the ‘Course-Management’ database to later be scheduled for that given day. Each lecture can be cancelled and any course can be completely deleted. Overall, this microservice serves as the first part of our system, storing the needs of the teacher, to later be processed by the ‘Calendar’ service.

Then, we realised that we had to manage the available rooms on campus, and we devoted one microservice for that.

The ‘Rooms’ microservice’s sole purpose is to manage the database that stores the pure information of rooms on campus. It has basic Create/Read/Update/Delete functions and lets a teacher create or delete a certain room. The capacities of the rooms are set on their base values pre-corona. The service only communicates directly with a validated client or the ‘Restrictions’ service to deliver the necessary information.

After that, the most important requirement of our system was that it should be adaptable and flexible with regards to changes to the pre-set rules. Our system should rely on dynamic conditions. We created the ‘Restrictions’ service to store and keep track of every rule and condition used in any part of our system. We thought about combining the ‘Rooms’ and ‘Restrictions’ services a lot while trying to complete our architecture, however, we thought that flexibility should priorities and all changed values should be able to be accessed from anywhere in our system. Thus, we decided to keep the ‘Restrictions’ service separate.

The ‘Restrictions’ microservice is what makes our system adaptable to changes in conditions. It keeps every rule set by the government including capacity limiting, the mandatory time gap between lectures, start and end time of the university. All these values are stored in a separate database, where it can be updated by teachers if needed. The system uses updated values every time a new request is sent. The ‘Restrictions’ microservice fetches the room information from the ‘Rooms’ microservice and calculates the adapted capacity. It also communicates with the ‘Calendar’ service to provide it with the required information.

The ‘Calendar’ microservice is where the magic happens. It forms the heart of the systems architecture and ties the functionality of the system together. It is responsible for assigning a timeslot, on-campus participants and room to requested lectures for a certain course, duration and date in such a way that each student is able to come to campus once, taking all the regulations imposed by the government into account. Next to that, it’s also responsible for the retrieval of personal schedules and attendance management. The latter makes it possible for students to indicate that they refrain from attending the lecture in person so that the number of on-campus attendants of a lecture can be retrieved and reviewed by university staff.

In our initial design, we planned to implement two separate microservices for scheduling lectures and retrieving personal schedules for both students and teachers. However, in such a scenario, the scheduling service would be an in-between service that would solely process data coming from the ‘Course-Management’ and ‘Restrictions’ services and output results to a schedule-retrieval service, without having any persistent storage on its own. Moreover, both services would deal with the same entities, meaning that they would operate in the exact same domain. By merging both services into one, we reduced the network overhead in our system as scheduled lectures can now directly be stored on and retrieved from the ‘Calendar’ microservice without having to send them somewhere else.

The merged service heavily relies on the ‘Course-Management’ service as the latter ought to provide the scheduler with (details about) the lectures to be scheduled, their associated courses, teachers and participating students. It also relies on the ‘Rooms’ and ‘Restrictions’ services for details about the rooms that can be used for scheduling, their corrected capacities, the daily time interval in which lectures can be scheduled, and the time gaps that should be introduced in between any lectures.

To retrieve all required information we used the API endpoints of these services, together with the synchronous HTTP protocol. We created dedicated communication classes on the caller side of the communication in a way that is inspired by the ports and adapters architectural pattern. Adding these extra classes made it possible to make dummy implementations for the interaction with other microservices when they weren’t in place yet, and boosted the testability as they could be mocked altogether.

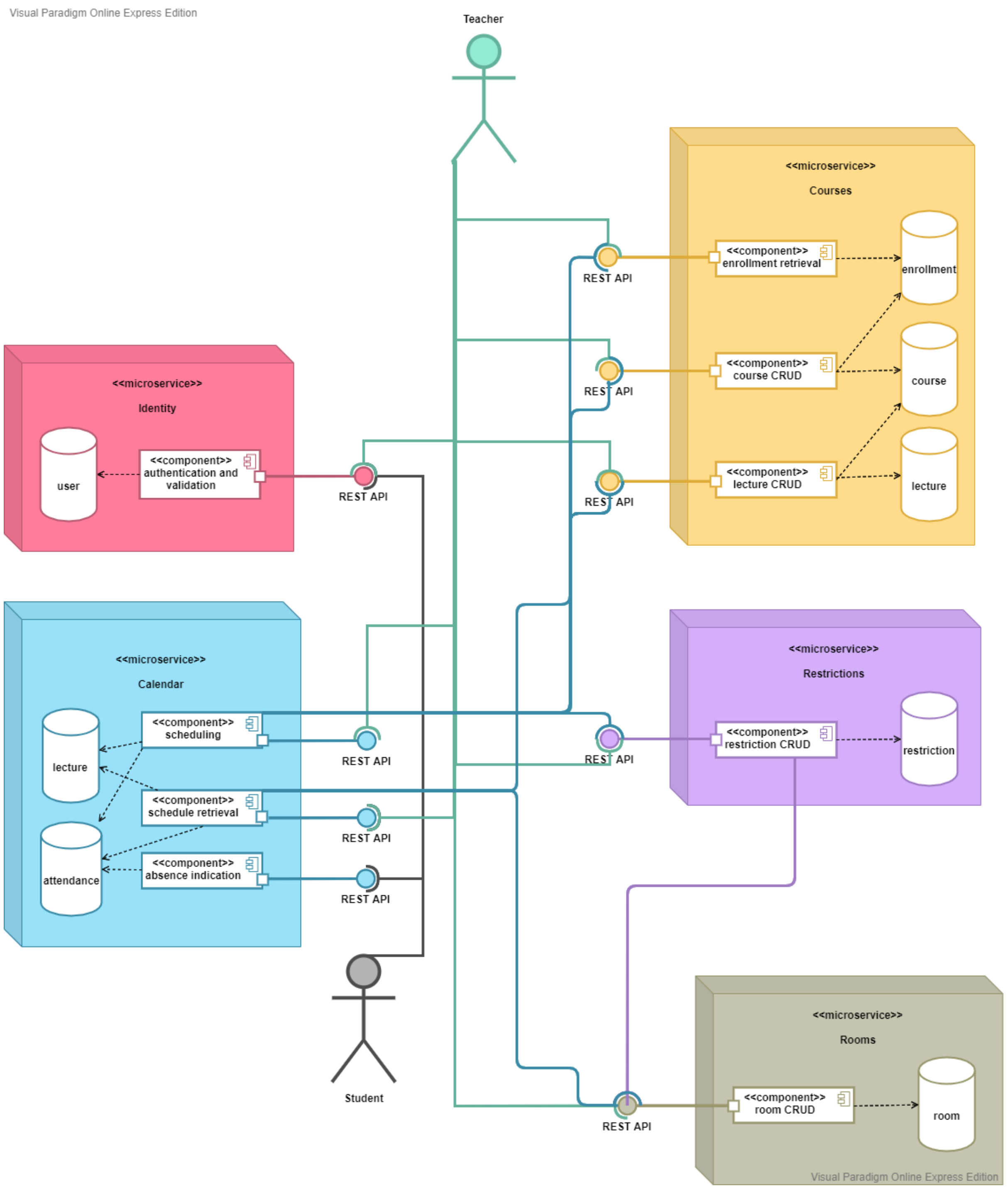
Provided with all the required information, the scheduler schedules all the lectures that had to be scheduled according to the ‘Course-Management’ service and save them into a dedicated ‘Lecture’ database. For each associated course, the participants receive entries in the ‘Attendance’ database indicating whether they are or aren’t selected to attend the lecture on campus. Whenever a student or teacher wants to see his schedule, he or she can make a request to the ‘Calendar’ service, which queries those databases and produces an overview of all upcoming lectures by day or course. And with that, the corona-proof room scheduling is a fact.

To provide you with a global overview of the system, we created a UML diagram showcasing the different microservices and the API interfaces between the different microservices and the client, for which we distinguish ‘Teachers’ from ‘Students’, in Figure 1. Modelling all 37 different API endpoints in our system would make the overview messy.

That's why the endpoints have now been clustered in components that are named after their functionality.

In conclusion, we succeeded in implementing a corona-proof room scheduling service. The resulting system uses the microservice architecture. Although designing this architecture took a lot of time at the beginning of the project, it paid off as we ran into fewer bugs during the implementation phase and it facilitated the distribution of the workload. After discussing some options, we came to the conclusion that a division into five microservices was the way to go: ‘Identity’ service, ‘Course-Management’ service, ‘Rooms’ service, ‘Restrictions’ service and the ‘Calendar’ service. All of them have a fair share in the overall system, but two of them are remarkably important: the ‘Identity’ service, the “gatekeeper” of the system, and the ‘Calendar’ service, the “heart” of the application. We hope that our program will facilitate the teachers’ work and make students’ life a bit easier, during these strange times.

Figure 1 - UML Diagram of the Microservices and Their Interactions



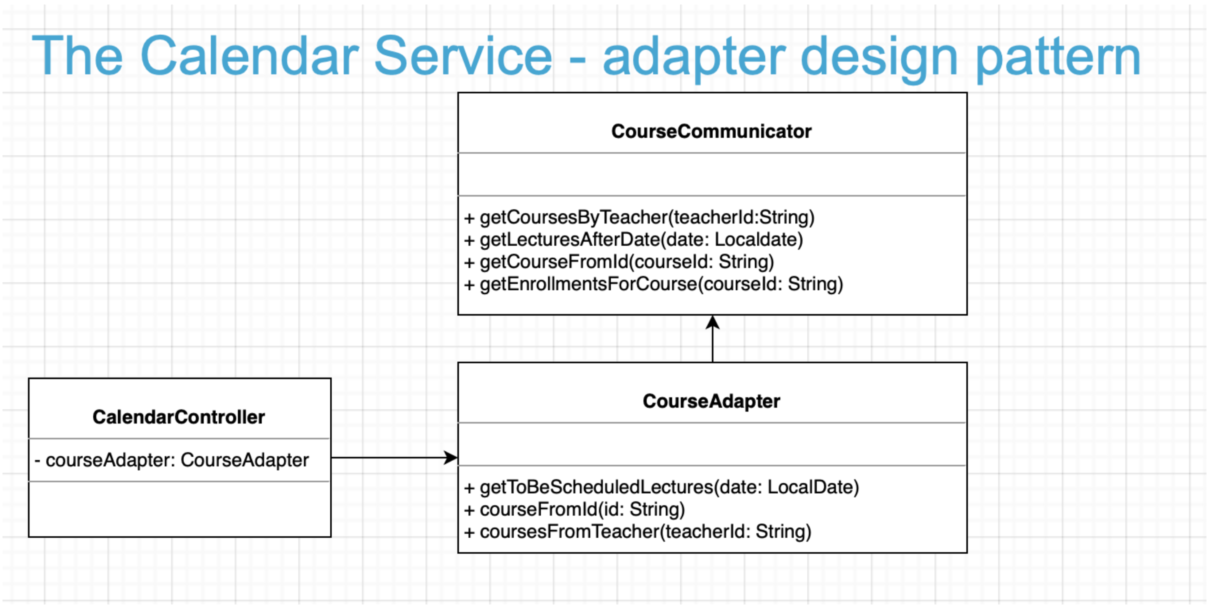
Task 2: Design Patterns (30 pts)

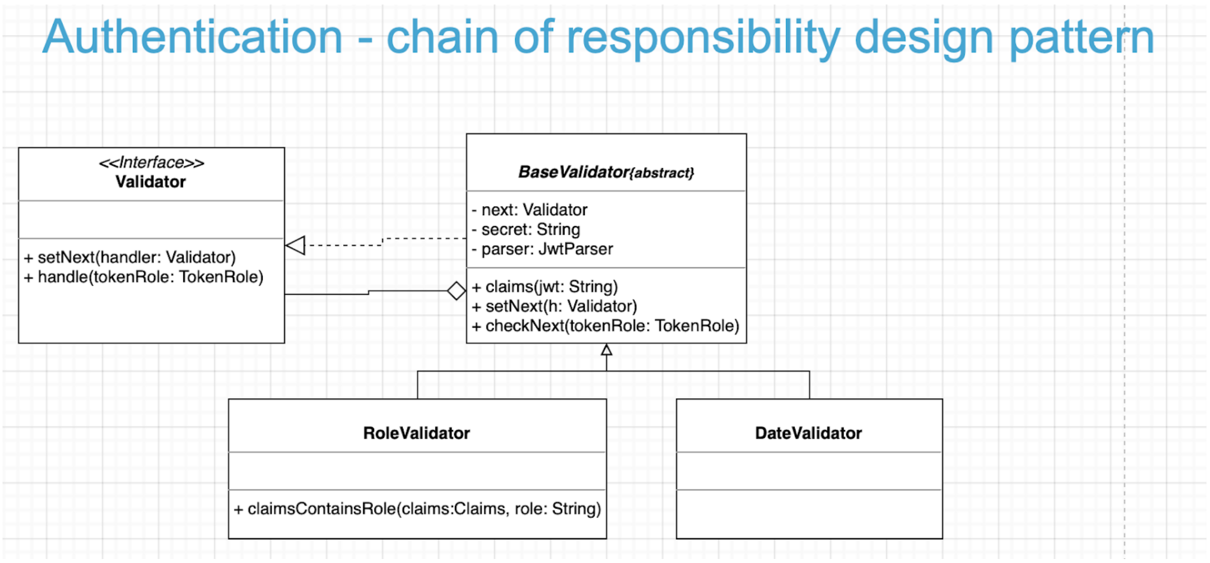
2.1. Write a natural language description of why and how the pattern is implemented in your system (6 pts per each design pattern).

- Within the calendar service, we make use of an adapter, which wraps the communicator class and provides useful methods to retrieve the lectures to be scheduled and information about their courses. We use this pattern because communication happens using so-called 'bare' classes which include less information than the calendar service needs for scheduling. In the communication folder, there's a CourseAdapter which wraps a CourseCommunicator. The CourseCommunicator handles all of the actual communication with the Course service, including mapping the returned values onto their respective classes. While the CourseAdapter only takes the returned inputs and converts them into the desired classes used by the LectureScheduler. This means that if the communication strategy changes, for example, due to switching to asynchronous communication or requiring HTTPS for transfers, still only the CourseCommunicator class needs to be changed.

- The chain of responsibility design pattern has been implemented in the identity service under the folder identity.design.pattern where the handlers are named Validators. We decided to implement this design pattern taking into account the possibility of extending the validation process of the JWT token, should the need arise in the future, for easier modification i.e. adding more objects to the chain. The pattern contains an interface where the shared API for all Validators is located, an abstract class with code that all handlers have in common and two concrete handlers, the former checking if the token has expired and the latter containing actual logic to verify if the requested role is correct and whether it can be found in the token. The chain creation can be found in the validate method of UserController under the folder 'identity.controller', at line 50. There, the validation chain will be called twice, once to verify if the role is that of a teacher and once more for the student. In case the token is invalid or the role incorrect, an exception will be thrown to the user stating they do not have access to the method for which they were making the request.

2.2. Make a class diagram of how the pattern is structured in your code (3 pts per each design pattern).


The Calendar Service - adapter design pattern
CourseCommunicator
+ getCoursesByTeacher(teacherId:String)
+ getLecturesAfterDate(date: Localdate)
+ getCourseFromId(courseId: String)
+ getEnrollmentsForCourse(courseId: String)
CalendarController
- courseAdapter: CourseAdapter
CourseAdapter
+ getToBeScheduledLectures(date: LocalDate)
+ courseFromId(id: String)
+ coursesFromTeacher(teacherId: String)


Authentication - chain of responsibility design pattern
<<Interface>>
Validator
+ setNext(handler: Validator)
+ handle(tokenRole: TokenRole)
BaseValidator{abstract}
- next: Validator
- secret: String
- parser: JwtParser
+ claims(jwt: String)
+ setNext(h: Validator)
+ checkNext(tokenRole: TokenRole)
RoleValidator
+ claimsContainsRole(claims:Claims, role: String)
DateValidator

2.3. Implement the design pattern (6 pts per each design pattern).
The design patterns discussed in question 2.1 are already present in the system.
The adapter (hexagon) design pattern can be found in the scheduling (calendar) service, and the second design pattern is present in the identity service.