ASSIGNMENT 1

Task 1: Software architecture
Task 2: Design patterns

Software Engineering Methods 2020

TEAM MEMBERS:
MICHEL BULTEN;NIKOLAOS EFTHYMIOU;LUCA HAGEMANS;ALEXANDRU LUNGU;ALEX MAAT;EE XUAN TAN

1 Software architecture

The implemented application copes with the problem of assigning rooms and students to lectures under the limitations posed by the Covid-19 normative restrictions. The overall software architecture is based on a Microservices approach with a set of loosely coupled services. As the relevant UML component diagram illustrates (see Appendix, image 1), five microservices have been designed that can be implemented in different servers with the option of using distinct databases respectively:

- *Authorization:* Responsible for user authentication and authorization and for providing the related authentication / authorization tokens to all other services by handling respective requests[1].
- *Courses:* Responsible for storing and managing courses and lectures offered by TU-Delft. Handles requests from the Student service for enrolling students to lectures and sends requests to the Room Scheduler for scheduling each lecture to specific room and time slot(s).
- *Room Scheduler:* Responsible for storing and managing time slots per lecture room as well as scheduling lectures (assign lectures to rooms) based on specific criteria: duration, avoiding year of study overlap, number of enrolled students, size of the room by considering the Covid-19 restrictions. Handles the related requests from the Course and Student services and sends requests to Room service to retrieve room data such as room size.
- *Rooms:* Responsible for storing and managing lecture rooms. Receives requests from the Room Scheduler service to provide data about rooms.
- *Students*: Responsible for storing and managing student enrolments to courses and to assign students to lectures (with a specific room and time slot). Sends requests to the Courses service to retrieve course data and demands for lectures by teachers.

Based on the Hexagonal Architecture, each microservice internally has its own internal business logic and by means of *inbound/outbound adapters* and *ports* handles/sends requests from/to other services or TU Delft subsystems, thus invoking respective logic. The **adapters** are mainly REST APIs using HTTP for *synchronous communication* among microservices and *MySQL adapters* for repository persistence services. Respectively, **ports** function as *service interfaces* with proper (mainly public) methods or as *repository interfaces* to access distinct MySQL database servers for each microservice.

The selected microservices architecture is characterized by quality attributes like *modularity*, *scalability* and easy *integration* with other TU Delft systems. In order to overcome microservices architecture shortcomings, through limiting the number of microservices, the selection was based on a **functional decomposition approach** and specifically on the *"domain-driven design",* according to these domains: *academic structure* (courses and lectures), *student enrolment and lecture attending, room and student assignment mechanisms, and security*. To this aim, the "*single responsibility"* and "*common closure"* principles have been applied through specific steps: i) functional requirements register, ii) describing system operations and external requests, iii) microservices identification and iv) APIs and collaboration definition.

1.1 Authentication Microservice

The main responsibility of the Authentication service is to handle the authentication part between our microservices. It stores the users and a roles table in the database so that other services can request which role a user has, it also makes it possible to add new roles whenever desired, so that it is extendable.

[1] Alternatively, it could be part of the TU-Delft authentication system or an interconnected service that sends requests to that central system to retrieve user's data and authentication token and then to authorize them for accessing the microservices.

The general flow consists of sending a request to the login endpoint with a username and password. The service will then validate the credentials within the database. If a user exists with the username and the password is correct, it generates a token and returns it. The token expires in 10 days, which was an arbitrary choice which we had not decided upon. In hindsight, we would have changed it to 3 hours instead, as this is used in other services as well (eg. Brightspace). There are some points that we would have liked to have implemented if we had more time like using SSL for the login and doing hashing on a potential client. However, due to the lack of time we did not do this.

We have a couple of endpoints for deriving the role a user is: */isAdmin*, */isTeacher*, */isStudent*. These are only accessible if the user has the right role. We decided upon this because this seemed to be the easiest way to do it given the short time we had. If we later want to scale it with more roles, another approach would be better. Furthermore, we have a couple of endpoints in regard to getting a user, updating an user, and getting a list of all users, which might come in handy for the administrator. There also is an endpoint that for example is used by the students service " */getUsername*" which uses the token to derive the username and is then used to get the student on the student service.

1.2 Room Microservice

The main responsibility of the room service is to maintain and provide information about all the rooms known to the system. It has endpoints that can be accessed by admins that can create, edit, and delete rooms. It also has standard *GET* endpoints for getting a list of all the rooms as well as getting a specific room by its id.

The other remaining endpoints on this service return rooms based on criteria that are required by the Room Scheduler Service. The first of these endpoints is *GET getRoomsWithCapacityAtLeast/{numberOfStudents}/{minPer}/{maxPer}*. This endpoint returns a list of all room ids that have the restricted capacity of *numberOfStudents*. *minPer* and *maxPer* are percentages that are used to calculate the restricted capacity of the room. If the room in question is considered to be a small room (number of seats is smaller than 200) it allows *minPer* * *originalCapacity* of the room, otherwise it allows *maxPer* * *originalCapacity* of the room.

The remaining endpoint is used to explicitly get the restricted capacity of a room and it uses the same *minPer* and *maxPer* percentages. This data is needed for the student service to know how many students can be scheduled for this on-campus lecture.

The idea behind having a separate Room service over having the Room and Room Scheduler services combined into one, was that managing both the storage and CRUD operations of rooms as well as all the endpoints for making the schedules for these rooms, would be too much for a single microservice to manage. We considered room scheduling and room managing as two different concerns which is why they ended up being two services.

1.3 Room Scheduler Microservice

The Room Scheduler Service's focus lies in scheduling lectures to rooms. To achieve this the Room Scheduler keeps track of rules for scheduling. These rules include:

- The buffer time (how much time it takes to clean a room after a lecture)
- The duration of a timeslot
- The duration of a break (the time between two timeslots)
- The number of slots per day
- The slot number of the lunch slot
- Restricted capacity percentage of small rooms (rooms that have a normal capacity of < 200)

- Restricted capacity percentage of large rooms (room that have a normal capacity of at least 200)

The reasoning behind this is that these rules should be easily changed by the system owner, for example when the regulations for Corona get stricter, the corona percentages of the rooms can easily be adjusted. When the service assigns lectures to rooms, it uses these rules to find suitable rooms.

When it comes to scheduling, the service keeps track of a list of room slots. These room slots store whether a room is occupied or not during a certain slot. Scheduling happens on a per-lecture basis. For each lecture that needs to be scheduled it gets the date that that lecture needs to be scheduled and how many timeslots the lecture takes up. The system then finds consecutive room slots that meet these criteria: all the slots of this lecture take place in the same room, the room has enough capacity for the students in the lecture but not too much as that would waste space, and it filters out room slots that have lecture for the same year (so that lectures from the same year do not overlap).

The reasoning behind using roomslots like this and doing a per-lecture scheduling is so that it makes it easier to find room slots that meet the criteria, as this would come down to a database query which is more performant than filtering programmatically.

1.4 Student Microservice

The student service is responsible for storing students, courses they are enrolled for, and what rooms they are assigned to. The main reason why we did this was because we wanted students to only interact with one microservice.

To go into depth why we decided the following endpoints for students:

- */setPreferences* which is a *POST* request including a boolean to set their preference. We wanted this so that we can give priority to students that are willing to come to campus.
- */allMyLectures* which is a *POST* request using the token to derive the student. The reason we wanted this endpoint is so that students can retrieve a list with all their upcoming lectures in ascending order.
- */allMyCourses* which is a *POST* request which works similarly to above. We wanted such an endpoint so a student can get a list with all his enrolled lectures.

For administrators we have the following endpoints:

- */initializeStudents* which is a *POST* request including a list of all the students, that way we can store the students in the database. This endpoint is important so that administrators can easily add students to the database without directly interacting with the database.
- */getStudents* which is a *GET* request so that we can retrieve all the students currently stored into the database. We thought it would be helpful for an administrator to be able to check all the students in the database, to ensure no mistakes are made.
- */assignStudentsUntil* which is a *POST* request. A very important one for assigning students to lectures on campus. It does this with a date so it can schedule the student up until that date. It makes sure that we initialize a *courseList* including all the courses with their lectures. We then call a function to only provide us with the lectures up to the given date in the post request. We send a request to the database, to give us a list with students that prefer to go to campus sorted by the last date they visited campus. This way we can evenly divide the student. In short, this endpoint is used so that an administrator can assign all the students to lecture rooms until a certain date.

1.5 Courses Microservice

The courses service focuses on providing the teachers an interface so that they can create courses and lectures. This is its own microservice, because our intention later on was to provide more CRUD operations for both courses and lectures, and to provide a way for a teacher to see their upcoming lectures, but unfortunately due to time constraints this has not been implemented. Which means that currently, the sole responsibility of this microservice is to store courses and lectures.

Endpoints used by teachers:

- */createCourse* which is a *POST* request that should include a course so that the service can store the provided course in the database.
- */createLecture/{courseId}* which is a *POST* mapping in which we give the course ID in the URL, the lecture in the body so that a teacher can create a lecture which is linked to the right *courseId*

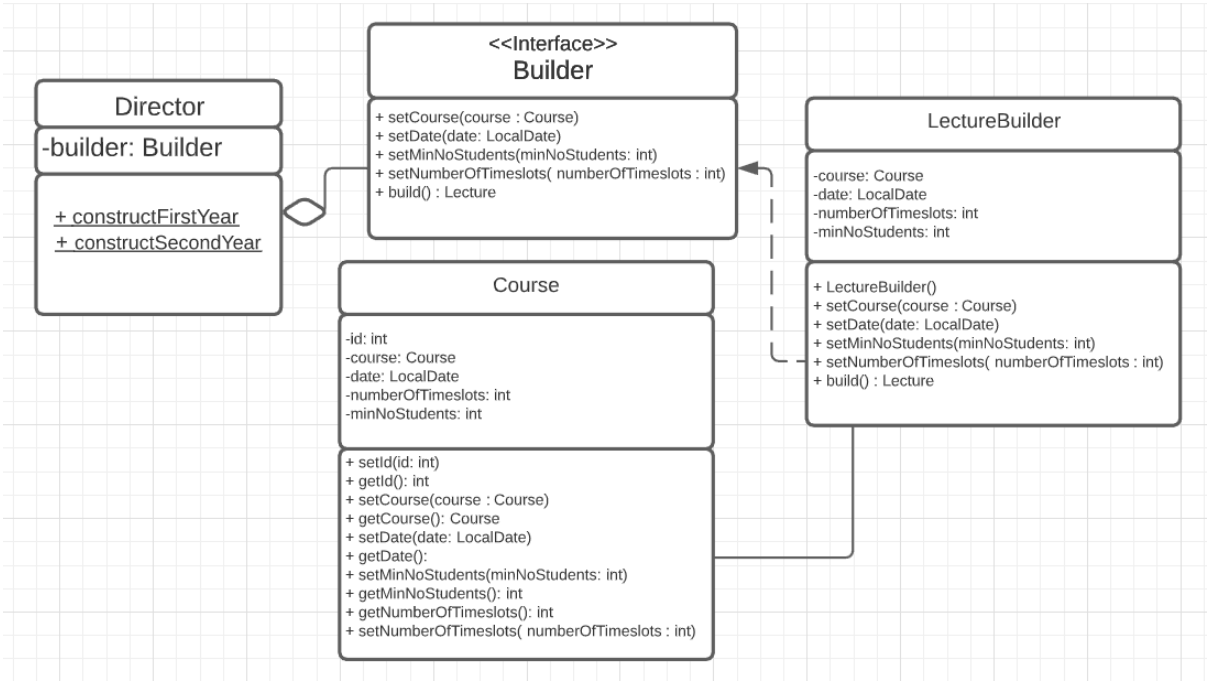
Endpoints used by Administrators or other services:

- */scheduleLecturesUntil* which is a *POST* request that should schedule all the lectures in the courses database until the date given. It does this by sending a get request to the room schedule service.
- */scheduleLecture/{lectureId}* which is a *GET* request that should schedule the lecture provided in the URL. It also does this by sending a request to the room schedule service with the lecture.
- */scheduleForTwoWeeks* which is a *GET* request that should schedule all the lectures in the courses database for the coming two weeks. We decided on creating this endpoint as we thought scheduling for the coming two weeks would be the most frequently done. But this can also just manually be done by the first endpoint.
- */getAllCourses* which is a *GET* request which the student service for example uses to get a list of all the courses including their lectures. It helps the student service with checking if a student is enrolled in a certain course and to get the course details.
- */getAllLectures* which is a *GET* request which the student service also uses to assign the students to the correct lecture rooms and to give students the ability to request their personal schedule. But other services or an administrator can also use this to retrieve a list of all lectures.

2 Design patterns

2.1 The Builder Pattern

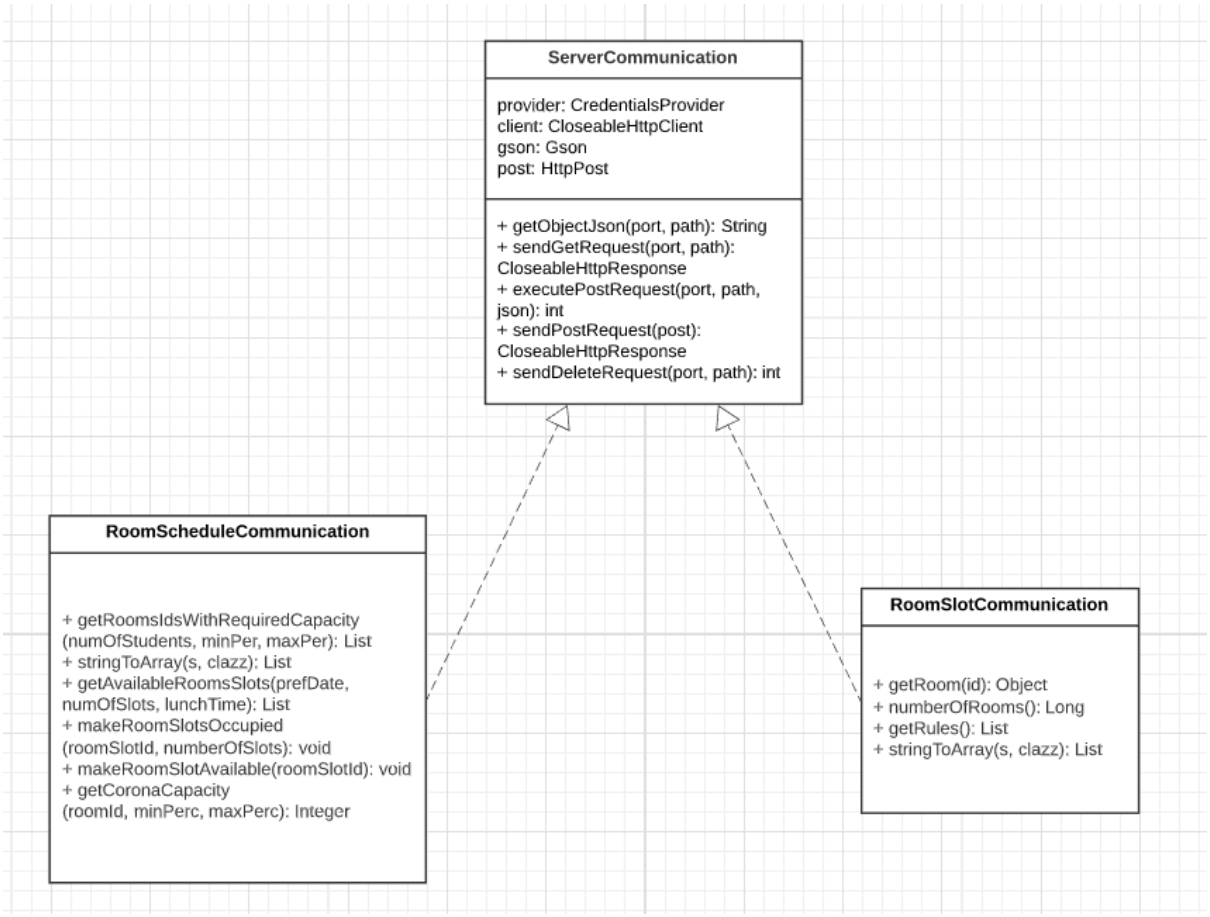
We have implemented the design pattern *Builder* in one of our microservices: The Courses microservice. In this microservice we have two entities, one being *lectures* and the other being *courses*. A lecture has 4 attributes that distinct different lectures from each other. One of the attributes is *minNoStudents* which refers to the minimum number of students this lecture will have. One of the requirements is that first-year lectures have more students than second year lectures. Thus, we have implemented the builder pattern by making a *director class* that creates two different types of lectures: *firstYearLectures* and *secondYearLectures*. These are different by the minimum number of students. So, when one of the two methods is called, a builder interface gives the list of attributes that a lecture must have: course, date, number of time slots and minimum number of students. Then in the *lectureBuilder* class these get created into an object, where we have setters for each of the attributes. An overview for this can also be seen in the UML diagram below.



A builder pattern is typically used to construct objects that are more complex, to make the creation of these objects simpler. In this particular use-case, using a builder pattern may not seem useful yet, however, if we look into the future, where we might want to add extra functionality to the lecture object, it makes more sense. For example, when wanting to schedule special lectures such as tutorial classes. Additionally, the lecture object could include certain information such as the number of Teaching Assistants necessary for that lecture or the lecture is for a group assignment. The builder pattern will then help reduce the complexity when trying to schedule lectures.

2.2 The Facade Pattern

We implemented the facade design pattern in our system in the communication classes. These are the classes that facilitate communication between microservices. Take as an example the class *ServerCommunication* from the room-scheduler service: it is a facade because it makes the communication between microservices easier for other classes such as *RoomScheduleCommunication* and *RoomSlotCommunication*. Furthermore, in *RoomScheduleController* we call the methods from the *RoomScheduleCommunication* class. Because *RoomScheduleCommunication* simplifies communication with other microservices greatly, this class acts as a facade for communication between microservices in *RoomScheduleController*.


ServerCommunication
provider: CredentialsProvider
client: CloseableHttpClient
gson: Gson
post: HttpPost
+ getObjectJson(port, path): String
+ sendGetRequest(port, path): CloseableHttpResponse
+ executePostRequest(port, path, json): int
+ sendPostRequest(post): CloseableHttpResponse
+ sendDeleteRequest(port, path): int
RoomScheduleCommunication
+ getRoomsIdsWithRequiredCapacity (numOfStudents, minPer, maxPer): List
+ stringToArray(s, clazz): List
+ getAvailableRoomsSlots(prefDate, numOfSlots, lunchTime): List
+ makeRoomSlotsOccupied (roomSlotId, numberOfSlots): void
+ makeRoomSlotAvailable(roomSlotId): void
+ getCoronaCapacity (roomId, minPerc, maxPerc): Integer
RoomSlotCommunication
+ getRoom(id): Object
+ numberOfRooms(): Long
+ getRules(): List
+ stringToArray(s, clazz): List

3 Appendix

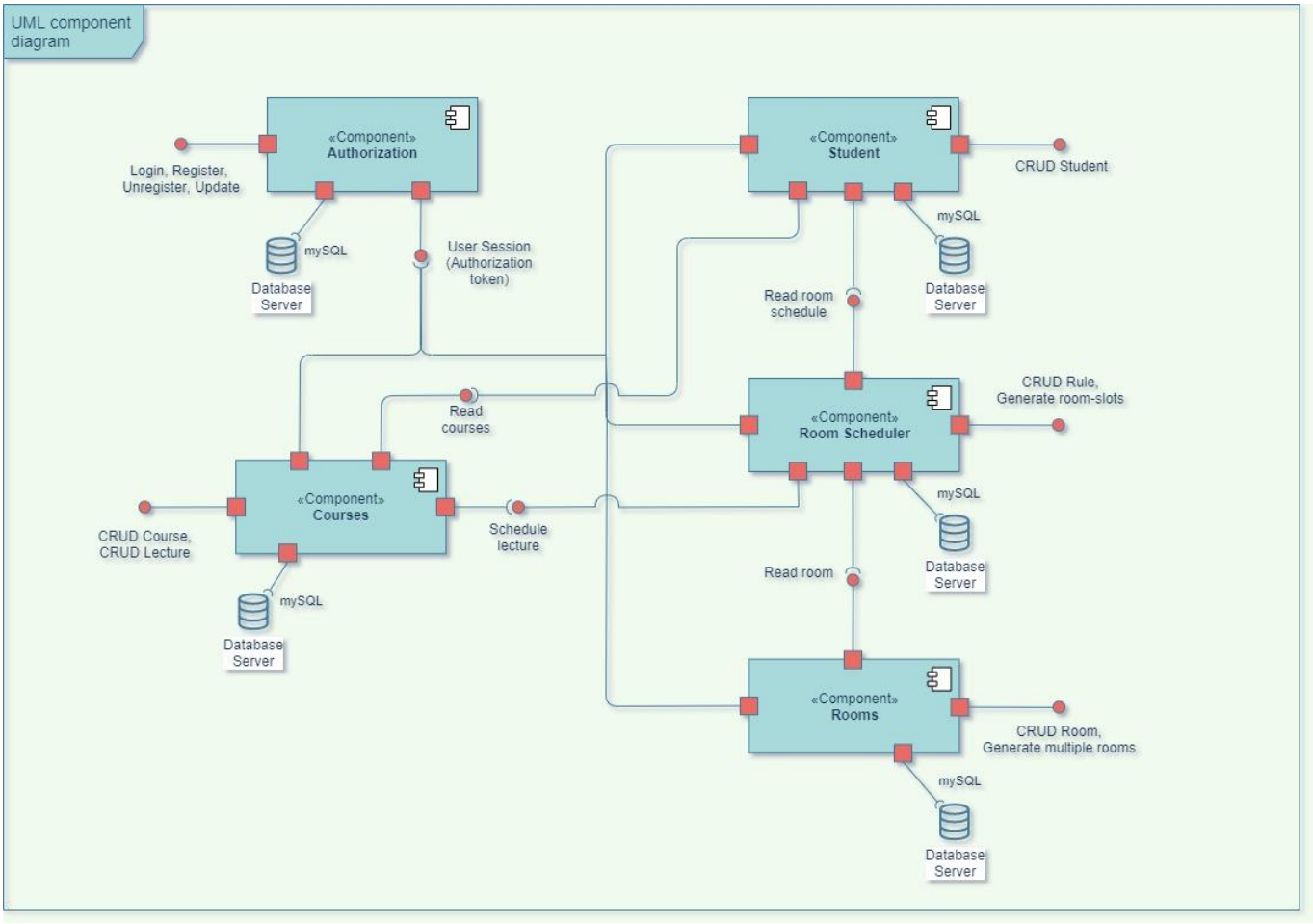


Image 1: UML diagram of the microservices architecture.