ASSIGNMENT 1

Group 31b

Alexandra-Ioana Neagu, Bozhidar Andonov, Ferhan Yildiz, Jannick Weitzel, Luuk van de Laar, Tudor-George Popica

TASK 1. SOFTWARE ARCHITECTURE

1. Domain-Driven Design and Bounded Contexts

For the scope of the current software (a specialized reservation system for sports centers) we have split the design process into multiple steps:

1. Identify the **entities** that would play a role in implementing the system.
2. Think about the way that these entities **interact** with each other.
3. Group them into bounded contexts, which would later be mapped to **microservices**.

Through analyzing the description provided for the project we identified the following entities: users, teams, reservations, sports halls, sports fields, lessons, equipment, security, and authentication.

We have found commonalities in some of these entities:

- For both sports halls, sports fields, lessons, and equipment, we need to store a unique code (UUID) to be able to identify each object stored in a database, as well as a name and the total amount of units available in the sports center. Thus, we decided that one bounded context should be **Bookable**, which encompasses all the things that can be booked in the sports center.
- A different bounded context is **Reservations**, which encapsulates the biggest part of the functionality in the software, namely the actions that can be taken on objects by the users or admins (booking, adding/removing objects in the database, retrieving objects, etc.).
- Both individual users and teams share similarities, such as name, email, password, and the unique code by which they are identified. Therefore, we decided to settle for the bounded context **Users**, which store both these entities.
- A computationally expensive and dense part of the software consists in authorizing access to users, and so we decided that **Authentication** is a fourth bounded context, encompassing login/register functionalities for users. Although this is directly linked to the users, it is a separate bounded context since it deals with a specific service on which the user is dependent, but the two can be split, to allow for scalability and atomicity.

The microservices we settled on are *Users* (corresponding to the Users bounded context), *Reservations* (corresponding to the Bookable and Reservations bounded contexts), and *Authentication* (corresponding to the Authentication bounded context). Why we decided on such a split is discussed later.

2. Microservices

Each microservice represents a part of the software that can be run independently from the others, which enables greater scalability of the application. Hence, each microservice needs to have its own internal structure, independent of other services. We decided that a layered architecture would work best: it is well-supported by Spring and is something we are all familiar with.

We were heavily inspired by the Model-View-Controller (MVC) architecture and therefore have identified the following layers to be used in each microservice:

- **Repositories**, which make the physical connection with the databases.
- **Services**, which contain the business logic of each microservice and are the middle layer between *Repositories* and *APIs*.
- **APIs**, which are used to communicate with the outer world (other microservices and users in our scenario).

We also decided on an architecture in which each microservice has its own separate database (using **PostgresSQL 13**), with its own specific tables, corresponding to the entities each microservice is responsible for. Following is a general overview of how each microservice is structured and what they are responsible for.

2.1 The *Authentication* microservice

This microservice falls under the category of **generic domains**, which are not core domains, but the core depends on them.

The main role of the authentication microservice is to know which users are currently (or will in the future be) interacting with the application and providing access only to those that already have an account. Therefore, before they are able to form teams and actually book equipment/sports facilities/lessons users should **register** with this microservice by providing an email and a password. If the provided email does not exist, then the user is created successfully and they are able to use the API of the other services as intended. However, they need to **log in** every time their session window expires. This operation generates a JSON Web Token (*JWT*, for short) that can be used to identify the user later on. The process is as follows: Whenever the users try to make a request to the other microservices, their JWT is forwarded to the Authentication microservice to check whether it is still valid. If it is, then the microservice gives a green light to the requesting microservice so that it can proceed with handling the request. If no JWT is present, then the user is prompted to log in or register with the authentication microservice. This way, the application becomes more secure.

The application also has **one admin user**, identified by a unique token, that is able to do **CRUD operations** on the different bookables, as we want to be able to add, remove and modify those.

2.2 The *Users* microservice

This microservice falls under the category of **core domains**, which is the main focus of the system. It handles almost all information regarding users, including their subscription status, usernames, as well as the teams that they have formed.

We have settled for **two entities** for the implementation of this service:

- **User**, which stores a single user, who is identified by a UUID, a name, an email, and a password. It also has a boolean variable indicating the type of subscription the user has (basic/premium), with premium allowing up to 3 reservations daily, and a List of reservations corresponding to each user which is checked every time they try to make a booking so that the constraint on bookings per day for a subscription is followed.
- **Team**, which stores multiple users who have been merged in a team. It has a unique team UUID, a name, and members (which is stored as a list of users, internally). Single users can therefore be represented as teams of size 1. Whenever a reservation is made,

only the team passes along and *not* a single user as this unifies all reservation cases and provides an efficient solution for one of the central issues in the design of this microservice, with a minimal amount of code duplication.

In terms of **functionality**, we have implemented *APIs* connected to *Services* for both users and teams. Thus, we can add and remove them from the database.

While both the Authentication microservice and the Users service have Users in common, they manage different aspects of said user. This service, in particular, is responsible for the **application-specific details of the user**, while the authentication service is simply responsible for user credentials and security.

Since both the Users and the Authentication microservice have an entry for the same user, which is split into two, they have to synchronize their databases. That is why we introduce a UUID in both microservices. This ID is only initialized when the user is created (by **registering**) and it is never changed afterward. This makes sure that there is not much need for synchronization after the initialization of the user, and also allows for better scalability. For instance, if we did not introduce this ID and we let users be identified by their email, in the case that users were to change their email (which is pretty common in a large-scale application) that would have to be reflected in both the Authentication and the Users database, which would introduce network overhead.

One could ask why we did not just merge the two microservices into one when they share information in common. The answer for that is, again, **scalability**. Many users could be trying to authenticate at the same time, while others may be trying to retrieve their teams, for instance. Instead of having a single service be the bottleneck, we split the functionality between these two services so these operations can be parallelized.

2.3 The *Reservations* microservice

This is also a microservice that falls under the category of **core domains**. It is responsible for all the functionality related to reservations and keeps track of everything that can be booked — Lessons, Sports Halls/Fields, and Equipment.

For this service, we have decided on **five entities**:

- **Reservation**, which corresponds to a reservation made by a team (we are representing single users as teams with one element). It links a Bookable with a Team and also has a start time of the reservation, as well as a unique id (UUID, again). To get the Team of the booking, the service should communicate with the Users microservice.
- **Bookable**, which encompasses all the different things that a user can make a reservation for: Halls/Fields/Equipment/Lessons. It has a name, description, maximum capacity/amount, and a unique identifier (UUID). All the other entities discussed below inherit from this class.
- **Sports Facility**, which can be either a *hall* or a *field*. These can be booked after 16:00.
- **Lesson**, which has a sports facility related to it, since lessons take place in one of the halls in the sports center. These can be booked after 16:00.
- **Equipment**, which has a sport associated with it, which is one of the requirements for making the application. These can be booked after 16:00.

In terms of **functionality**, we have implemented *APIs* connected to *Services* for each individual ‘bookable’. Thus, we can add, retrieve, delete, and book all of the bookable objects listed above.

We have also made sure that we can retrieve all the available objects for a certain time frame. The APIs also let admin users do **CRUD** operations on the bookables.

Naturally, for the service to be functional it should also keep track of these reservations. How does it do that? Well, it links a Bookable from the reservations database to a team from the Users database. Therefore, whenever information for the team is needed, a request to the Users microservice is sent from the Reservations database with the specific *teamUUID*. An important observation is that reservations are made for teams only.

One might ask why we did not make it for users as that seems more intuitive — a user has a reservation after all. Plus, the team does not have a limit to how many bookings they can make but users do. The reason behind this decision lies in the fact that if we decided to link Users to Bookables instead of Teams, then the information stored in the Reservations would grow and could contain more duplicates. It also results in a more complex API that can take both a single user and a team as arguments for booking, which introduces more complications when handling requests. Instead, we let the Users microservice handle all of that complexity (by adding a list of *reservationUUID*s for each user) and we only focus on what we are responsible for — reservations.

One could also ponder why we did not split this service into two, as it takes care of not 1, but 2 bounded contexts that have been identified earlier — reservations and **bookables**. That means it is also responsible for all of the things that can be reserved. While another service can indeed be created out of that, we decided that our solution is more **scalable**. Making a request to book or retrieve a reservation is often accompanied by information about the subject that is being/has been reserved. If we were to split this microservice into two, then the reservations microservice would need to communicate a lot to the other service. This introduces a lot of overhead when handling requests and is not desirable when wanting to achieve scalability.

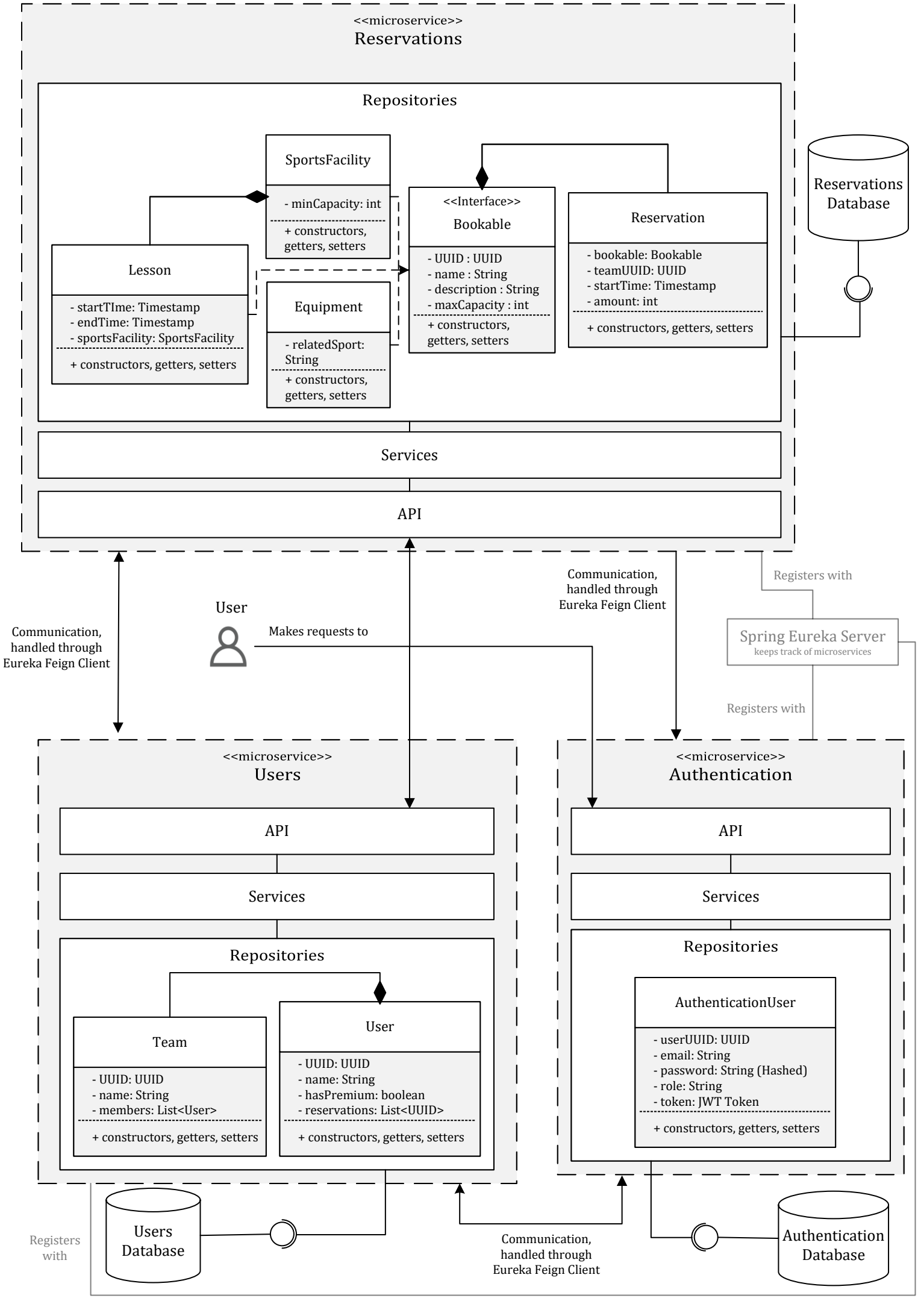
3. Interservice communication

Services are more independent than in a standard application but that comes at a cost of them having to ask for external information, as not all the microservices have the full picture. That is why communication between services is a vital topic for this application. The usual approach would be to send requests straight from one microservice to the other by hardcoding the names and ports of the other services. That, however, becomes cumbersome and infeasible when multiple instances of the same microservice are up and running.

That is why we opted for using **Spring Eureka**. We have devised a discovery service that makes use of **Spring Eureka Server**, which keeps a registry of which services are available. Each individual microservice is then registered as a **Spring Eureka Client** in that register. Each such client also has a list of **Feign Clients** — other microservices that this service depends upon. These Feign Clients are represented as interfaces with API endpoints, whose signatures match those of the actual client. Whenever a microservice requests something from a Feign Client, Spring automatically forwards it to the correct physical microservice. This makes communication simpler, more effective, and increases scalability. This communication is initiated inside the **Service** layer of the requesting microservice and arrives at the **API** layer of the requested service.

4. Architecture diagram

Below is a diagram, visualising what has already been said in this document. Its purpose is a visual aid in understanding what has been mentioned.


<<microservice>>
Reservations
Repositories
SportsFacility
- minCapacity: int
+ constructors, getters, setters
<<Interface>>
Bookable
- UUID : UUID
- name : String
- description : String
- maxCapacity : int
+ constructors, getters, setters
Reservation
- bookable: Bookable
- teamUUID: UUID
- startTime: Timestamp
- amount: int
+ constructors, getters, setters
Lesson
- startTIme: Timestamp
- endTime: Timestamp
- sportsFacility: SportsFacility
+ constructors, getters, setters
Equipment
- relatedSport: String
+ constructors, getters, setters
Reservations Database
Services
API
Communication, handled through Eureka Feign Client
Registers with
User
Makes requests to
Spring Eureka Server
keeps track of microservices
Registers with
Communication, handled through Eureka Feign Client
<<microservice>>
Users
API
Services
Repositories
Team
- UUID: UUID
- name: String
- members: List<User>
+ constructors, getters, setters
User
- UUID: UUID
- name: String
- hasPremium: boolean
- reservations: List<UUID>
+ constructors, getters, setters
<<microservice>>
Authentication
API
Services
Repositories
AuthenticationUser
- userUUID: UUID
- email: String
- password: String (Hashed)
- role: String
- token: JWT Token
+ constructors, getters, setters
Registers with
Users Database
Communication, handled through Eureka Feign Client
Authentication Database

TASK 2. DESIGN PATTERNS

We have settled on implementing the **Builder** pattern in the Users microservice and the **Chain of Responsibility** pattern in the Reservations microservice.

1. The Builder Design Pattern

We have decided to implement the Builder Pattern. With code maintainability and scalability in mind this seemed like a beneficial design pattern to implement, given that the Builder Pattern allows us to separate and encapsulate the construction of our object from the representation and the business logic. We have implemented this pattern for our User entity class and our Team entity class, with the thought in mind that in the future these classes could quickly become a lot more complex as our system would require more fields and logic from these classes. For example, our system might require the User class to keep track of an extra Birthday field and an isBirthday method, for say personal birthday discounts. If we keep adding fields and logic like this to our class it would quickly become unmaintainable.

To counter this we have implemented the Builder Pattern by creating another Builder class inside of our classes. For the User class and Team class these are called UserBuilder and TeamBuilder respectively. To explain it further we will take the UserBuilder as an example. The UserBuilder class has the same fields as the User class does. It has its own constructor, which doesn't take any parameters and just initializes a UserBuilder object with a new UUID and its hasSubscription field set to false (by default, upon creation, Users start out with a basic subscription). Our UserBuilder also has a method for every field, which initializes the relevant field using the given input and returns that UserBuilder object, which allows us to chain together initialization methods. After we've initialized every field we give our UserBuilder as an input for the constructor of the User class itself, using the UserBuilder's build method. Before returning the created User object, the build method internally calls a private method, validateUserObject, which validates if the object doesn't break any assumptions of the system. The User class creates the actual User object, using the initialized fields of the UserBuilder object. Here's an example of what constructing a User object using our UserBuilder class would look like:

```
new User.UserBuilder().name("John").hasPremium(true).build();
```

This creates a User with a random UUID, the name John, a premium subscription, and an empty list of Reservations, that is checked behind the scenes for integral validity. As you can see, by using the UserBuilder we reduce the parameters to our constructor and we provide them in more easily interpretable chained method calls. Now we also don't need to pass "null" for optional parameters while constructing our User. Constructing the object using the builder also helps us with testability, by making object set up more intuitive and maintainable.

Even though the Builder Pattern makes the classes simpler to understand and more readable, a disadvantage is that it does increase our code base of the entities by quite a lot. Although this is the case, we still think that the advantages it introduces more than make up for the extra lines of code, so ultimately we didn't think this disadvantage was detrimental enough to defer us from implementing this design pattern.

The implementation of the Builder Pattern can be found in the User and Team entity classes, which can be found by following the path sem-repo-31b/users/src/main/java/users/entities. On the next page you can see the class diagrams for the TeamBuilder and the UserBuilder.

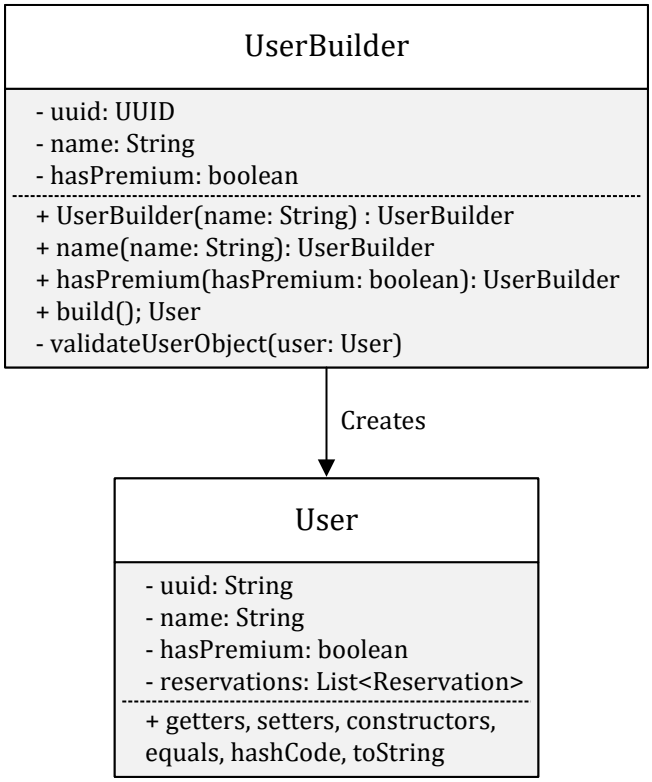


Figure 1: The UserBuilder

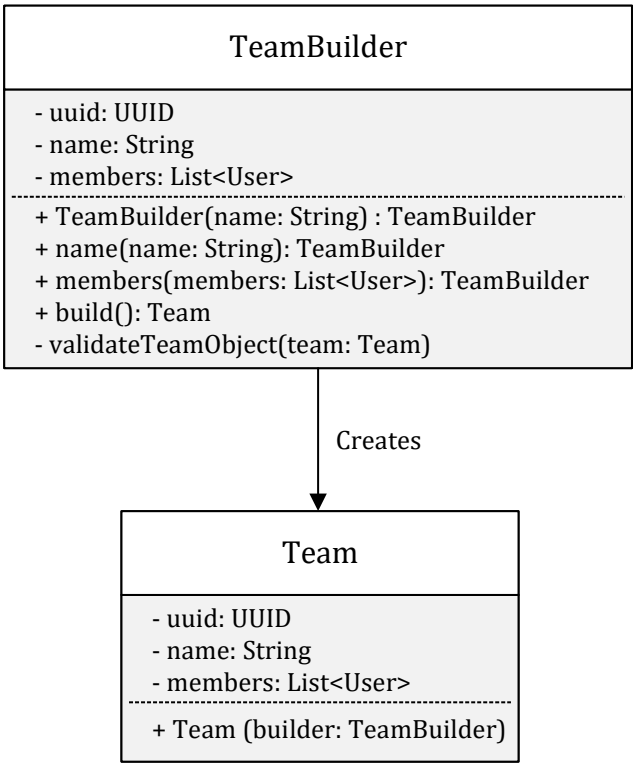


Figure 2: The TeamBuilder

2. The Chain of Responsibility Design Pattern

Apart from the builder design pattern, we have also decided to implement the **Chain of Responsibility** pattern. We also settled on the reservations microservice being the one to host this pattern. More specifically, we have made the choice to decouple the validation of the parameters when making a booking for equipment and the actual creation of the reservation. The usage of this pattern also means that we can easily use parts of the chains or even the whole chain in all other classes and methods in the reservation package that need them as there is a lot of shared logic between them. This would greatly reduce code duplication and make it more maintainable.

In order to implement this pattern, we have made a validator interface which makes sure every validator has a `handle` and a `setNext` method. The handle method is implemented by each concrete implementation of the validator interface and is responsible for validating the input that is given to it. If it is valid, then we proceed to the next validator and if not, an `InvalidParameterException` with an appropriate message for each validator is thrown. As we will see later, we have a few different implementations of those handle methods. The setNext method, on the other hand, takes another Validator as a parameter and sets it as the next Validator to execute. Before we get to concrete implementations of the validators, however, we should also mention that there is also the Base Validator abstract class which adds a `Validator next` field and has a concrete implementation of the setNext method. It also adds a checkNext method, which checks whether there is another validator in the chain - if not, then the chain has finished and we can go back to handling the request. Otherwise, it calls the handle method of the next validator.

Now, concrete implementations of the BaseValidator class need to implement their own handle method. We have devised **4** different validators:

- The **EquipmentUuidValidator** checks whether the UUID that is provided already exists in the Equipment. If it does, then the validator retrieves that piece of equipment and saves it as a result. Otherwise it throws an exception that the UUID is invalid.
- The **StringToTimestampValidator** takes the time when the equipment has to be used as a String and converts it to a Timestamp. It does this by trying to fit the String to a date, if this fails this validator will throw an InvalidParameterException with the message “Timestamp cannot be parsed”. If it doesn’t fail, it sets the result of the validator to the parsed timestamp.
- The **EquipmentTimestampValidator** checks if a Timestamp makes sense when booking for equipment. There are two questions that need to be asked here: Is the timestamp in the future and is it between 16:00 and 23:00 as that is the only time when equipment can be booked. If the answer to any of those questions is no, then we throw an InvalidParameterException with appropriate messages. Otherwise, we continue with the next request.
- The **EquipmentAmountIsAvailableValidator** checks whether the amount of equipment that is requested for booking is available during the provided time period.

All of these can now form an arbitrary chain together. We have made use of these chains in the `bookEquipment` and `getAllAvailableEquipment` methods in the `EquipmentService` class, located in the reservations microservice.

Here is an example of a chain being initialised:

```
Validator stringToTimestampValidator = new StringToTimestampValidator();
Validator equipmentTimestampValidator = new EquipmentTimestampValidator();
stringToTimestampValidator.setNext(equipmentTimestampValidator);
stringToTimestampValidator.handle(...);
```

This creates a simple Chain of Responsibility of length two. with the stringToTimestampValidator being first and the equipmentTimestampValidator second\ in the chain. Instead of having all the logic here, in the method, we let the chain handle it.

The only thing we have to be careful of when constructing this chain is that every link has to be correctly added. If you make a mistake here, you could end up trying to make a reservation with invalid parameters which would then break the program. Despite this we have still opted to use it as we thought the benefits far outweigh the drawbacks.

The implementation of the Chain of Responsibility Pattern can be found in the validators package in the reservation microservice. You can find this with the following path sem-repo-31b/reservations/src/main/java/reservations/validators. On the next page you can see the class diagram for the Chain of Responsibility.

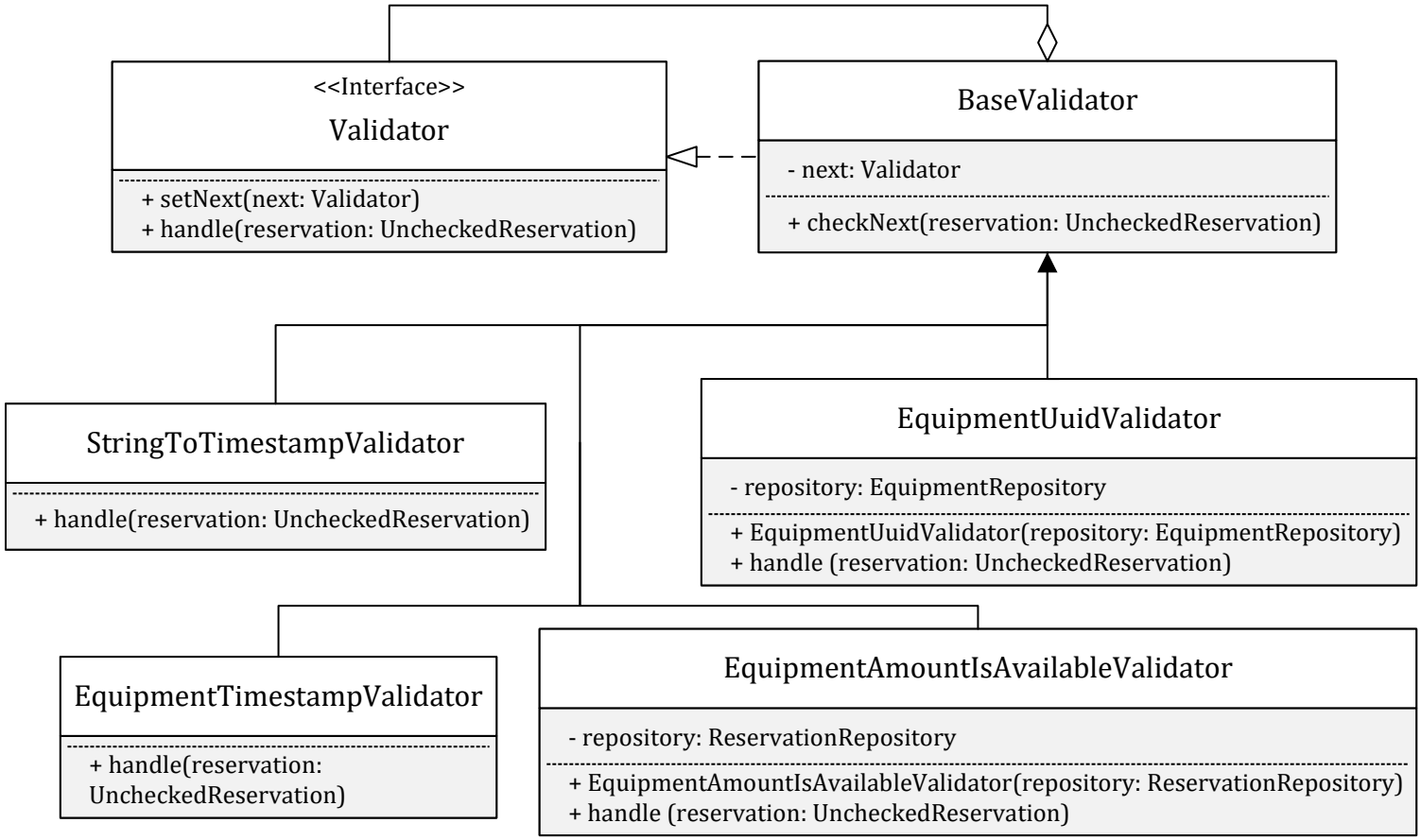


Figure 1: Class hierarchy

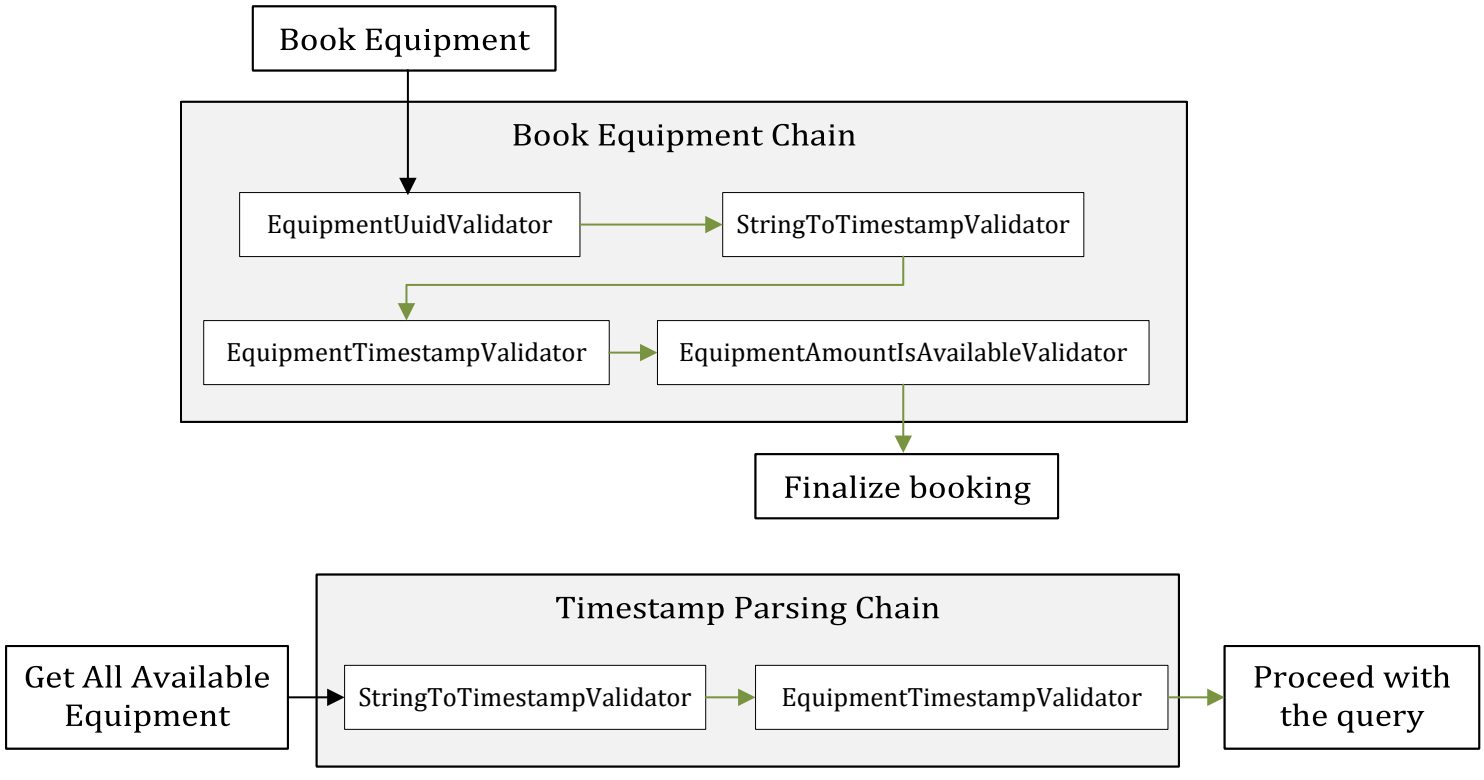


Figure 2: Chains, that have been constructed for two of the methods in EquipmentService

If one of the validators in the chains is not successful, an InvalidParameterExcception with the appropriate message will be thrown by that validator.