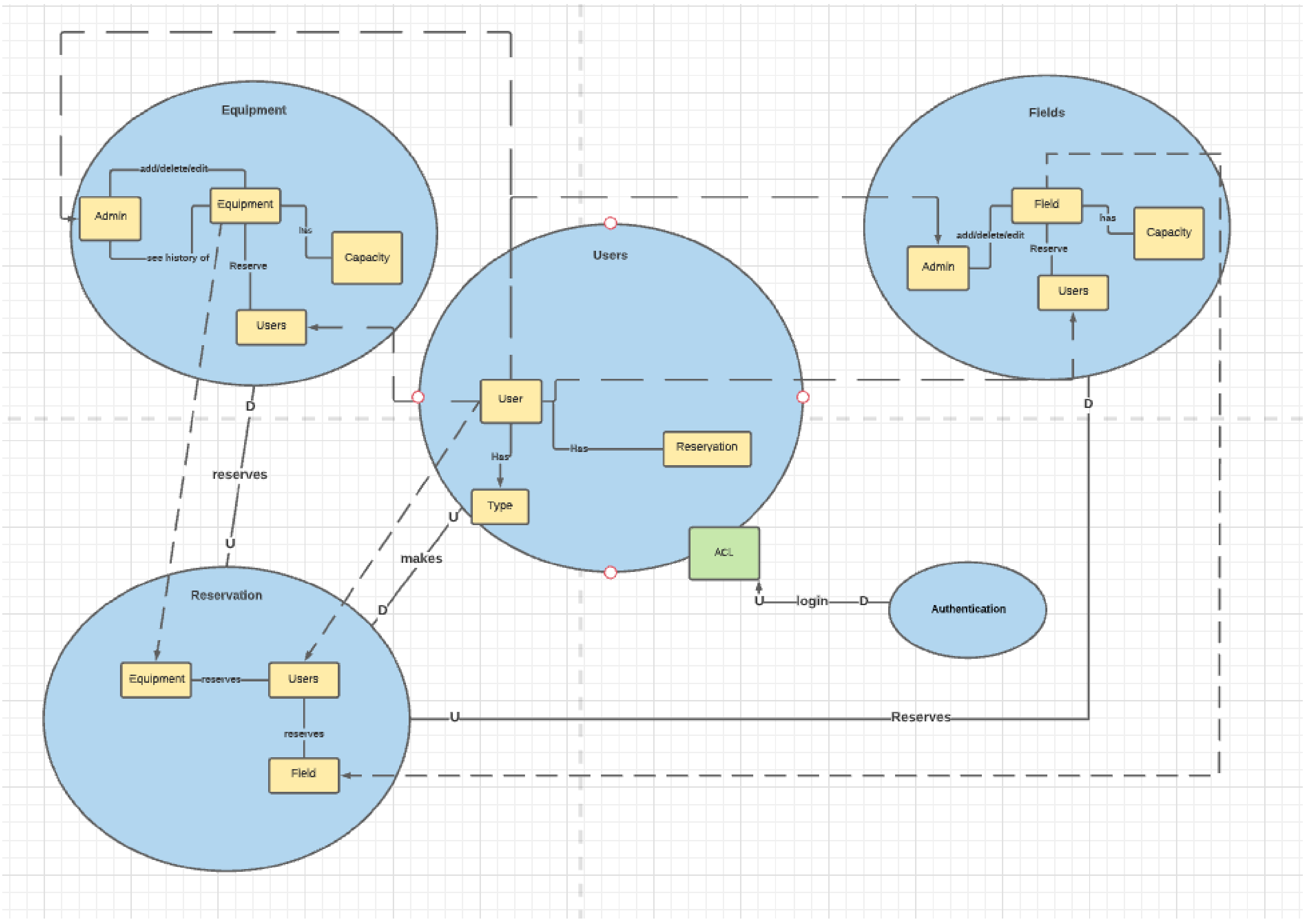
Assignment 1

TASK 1: SOFTWARE ARCHITECTURE

Step 1: Bounded contexts and context maps:
At first, we identified the contexts present in our scenario and the set boundaries between them using Domain-Driven Design (DDD). We recognized the following 5 contexts:

- **User**: The application will be used by users and admins, who are a special type of users. Users may want to reserve fields and participate in lessons(for normal users), or to monitor the activity in order to maintain the integrity of the reservation system and of the used assets (for admins). Users are the main target audience for which the application is made, so User is a core domain.
- **Field**: Users can reserve certain areas in order to practice a large variety of sports. These include outside sports fields (for example for playing football or tennis) or indoor halls (for example for taking swimming lessons or skating). These sport areas can be reserved for lessons (in a certain time slot), groups or individual practice.
- **Equipment**: Users can also reserve equipment that they can use during the reservations. Users may want to reserve fields in order to practice a certain sport, but they may also want to reserve equipment instead of buying their own. Therefore, this is also an important requirement of the application.
- **Reservation**: Users can make reservations for either equipment or fields or lessons. These reservations will contain a timeslot in order for us to maintain an order of the reservations and to avoid collisions of the timeslots. Reservation gives the main action an user can perform when using the application, therefore Reservation is a core domain.
- **Authentication**: The application doesn’t want users using other users' subscriptions. To prevent this, authentication is needed. Authentication will make the application more resistant to malicious users, but without it the application will still work. Authentication is thus a generic domain. There will need to be an Anti Corruption Layer between User and Authentication.

We can now model the main concepts of each bounded context in a context map:



In order to clearly visualize the interactions between bounded contexts, we drew a context map containing the contexts and the relationships between them. Upon closer inspection of the illustration, we specifically noted the bounded contexts that would form an upstream or downstream.

We started from the User core domain and chose an attribute type to specify if the user would be a regular user or an admin. This is important because an admin would have extensive privileges compared to a normal user, such as seeing the history of the reservations or deleting equipment and fields that are out of use. Additionally, we put the Authentication layer in relation with the user because the user is the one that requests their data to be stored safely and therefore is in need of security.

The main action performed by a user is making a reservation, therefore between User and Reservation, the User is the upstream context and Reservation is the downstream context.

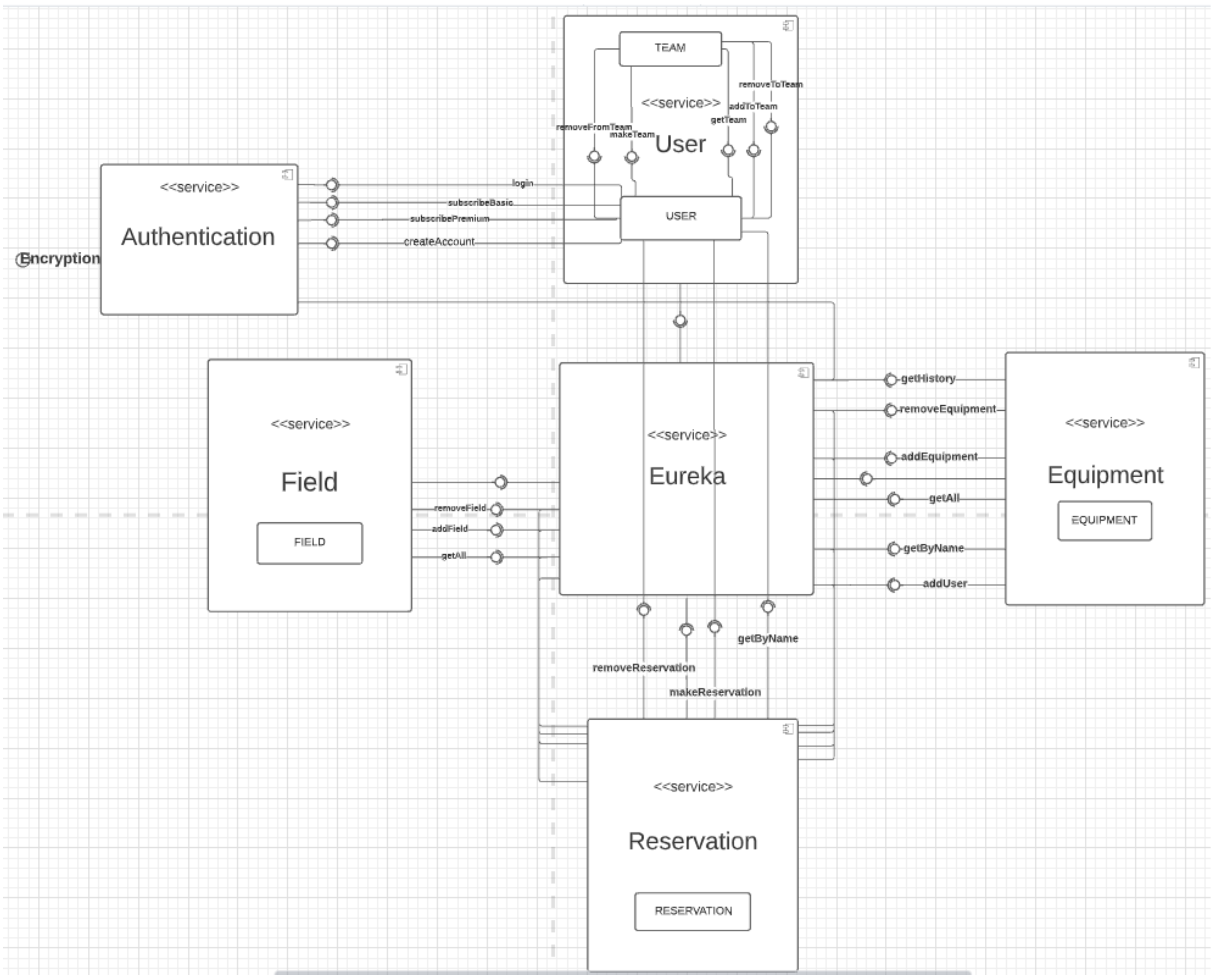
A reservation can be made for either a Field or Equipment, therefore these are the downstream contexts related to reservation. We chose to model Field and Equipment as separate entities because we thought there should be a clear distinction between the two. Even though they have some attributes in common, they are important enough to be considered separately. On top of that, if changes are to be made to just a field or equipment, which is expected to happen quite often (for example fields are out of service or some piece of equipment went missing and it must be removed), we believe it is easier to just model them separately.

Step 2: Component diagram

The component diagram shows the entire microservice architecture of our application. This includes the microservices, the controllers and the methods used by each microservice in order to communicate with the others. After the performed analysis, we ended up with 5 microservices, as described earlier:

- **User**: Manage users (students,teachers,admins) data.
- **Equipment**: Manages information about renting equipment.
- **Field**: Manages information about reserving fields.
- **Reservation**: Manages information about reservations made by users.
- **Authentication**: Security layer, makes sure all the data is stored safely and malicious users cannot make use of the information of others.

We chose to design our microservices this way because we think this design allows for easier implementation, it has a clear structure and it’s easier to extend if new features are going to be added in the future.
Other possible designs that we had in mind when we first started developing the application would be, for example, to make the user and the team as separate microservices, or to include the reservation microservice into one of the other microservices instead of making it a separate instance.
We believe our design is better than these approaches because if new types of users are added, instead of making a whole new microservice for them, we can just add a new type of user to the existing microservice. This extensively improves the extensibility of the application. Also, by making reservation a separate microservice, we avoid increasing the complexity of one microservice too much. This makes our application easier to test and easier to spot future bugs.

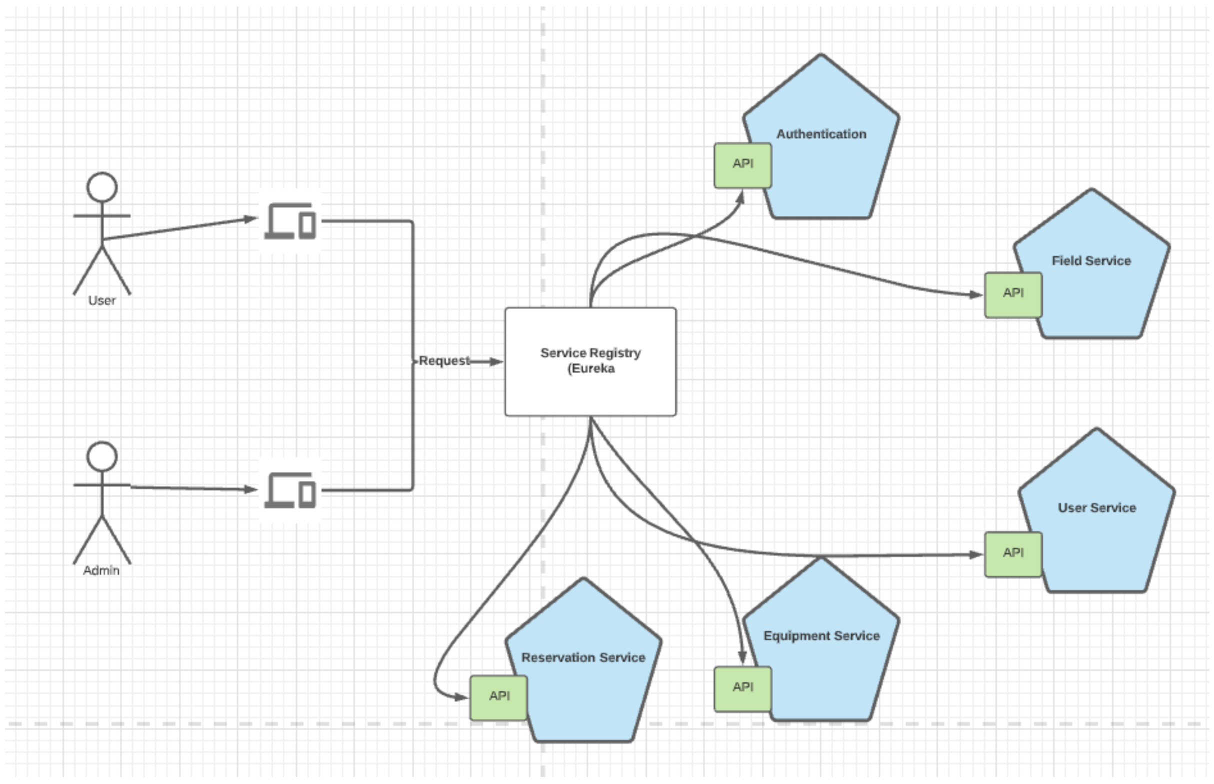


Step 3: Microservice architecture

After considering the bounded contexts and analyzing the context map, we chose which microservices we are going to implement in our application by doing a one-to-one mapping of bounded contexts into the microservices. The backend microservices include the following services:

- **User service**: Manage users (students,teachers,admins) data.
- **Reservation service**: Manages the reservation data for reservations made by the user.
- **Equipment service**: Manages information about renting equipment.
- **Field service**: Manages information about reserving fields.
- **Authentication service**: Manage the authentication.

The following diagram shows how the communication is done between the microservices we chose to implement:

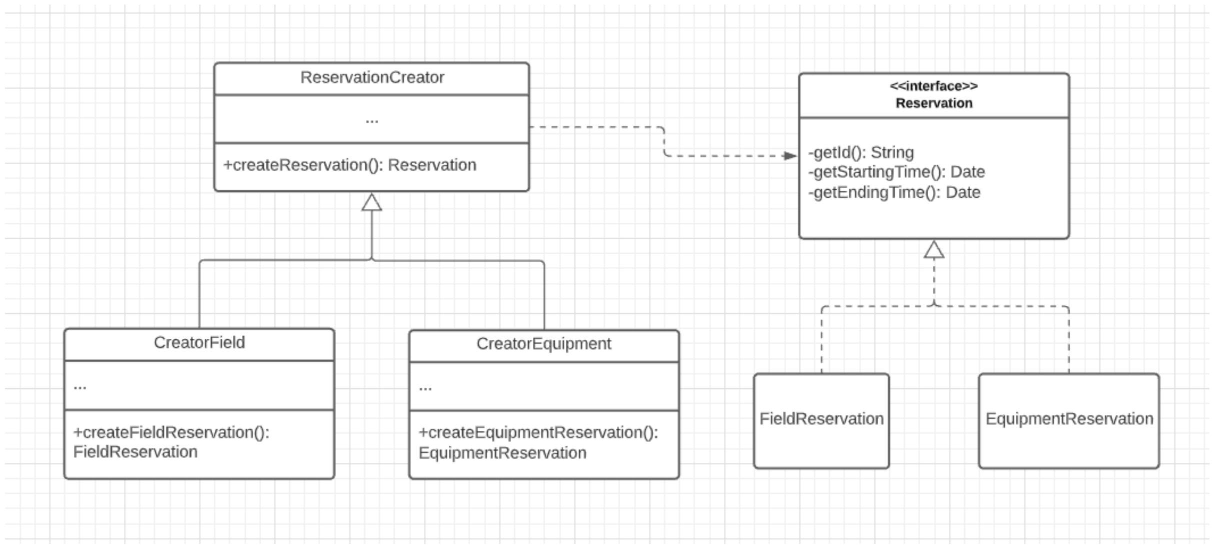


TASK 2: DESIGN PATTERNS

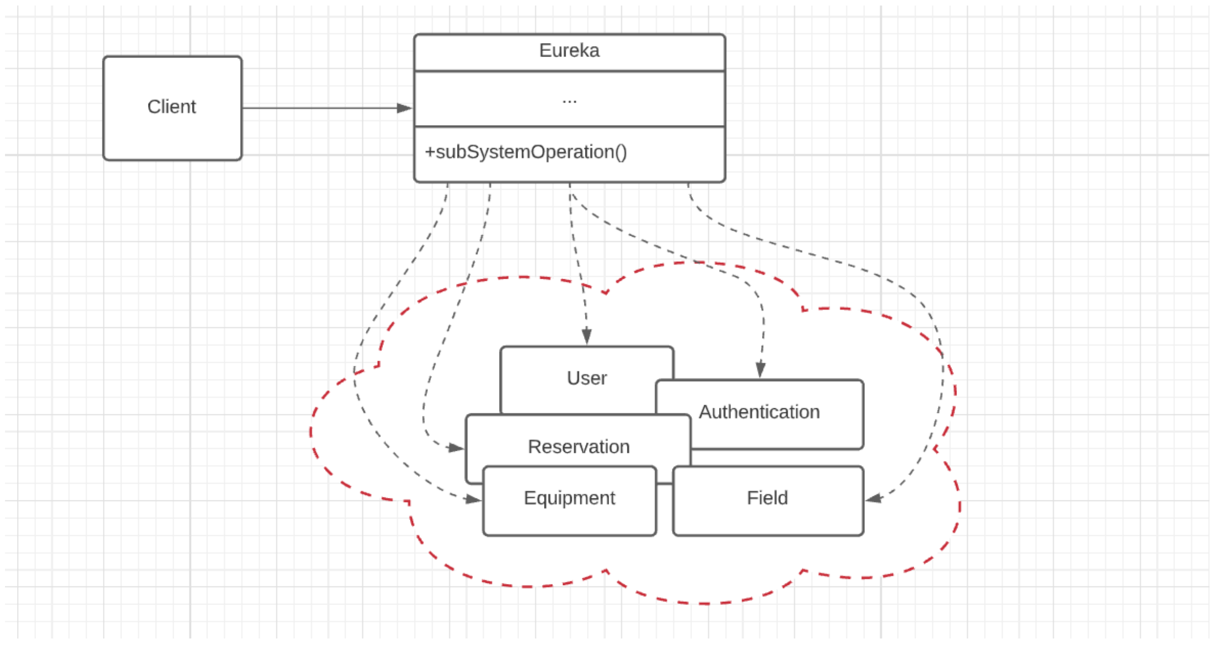
2.1 - 2.2

The first design pattern that we chose to implement is the Factory Design Pattern. This is a creational design pattern that lets you make related objects without specifying their concrete implementation. It provides an interface superclass for creating objects, but allows subclasses to alter the type of objects that will be created. The Factory replaces direct object construction calls with calls to a special factory method defined in the subclasses.
In our case, this design pattern is used to make reservations. A user can make specific types of reservations (for now, a user can reserve equipment or a field), so we made an interface class Reservation and made EquipmentReservation and FieldReservation implement this class.

We decided to implement it this way because since EquipmentReservation and FieldReservation have different attributes(for example isLesson and Paid), the design pattern makes it easier to create new specific objects from the abstract class. Furthermore, this design pattern makes it easier to add more types of reservations in the future. For example, if a restaurant will be added to the system and people need to make a reservation for that as well. This increases the extensibility of our application and makes it easier to spot bugs in the code.



The second design pattern we chose is the Facade Structural Design Pattern. Facade is a design pattern that provides a simplified interface to a complex set of classes and it isolates multiple dependencies within a single facade. A facade provides limited functionality in comparison to working with the subsystem directly. However it provides the client with exactly what it needs through links to the subsystems. The client uses the facade instead of calling the subsystems directly. Subsystems aren't aware of the facade's existence. They operate within the system and work together directly. In our case the subsystems are our various microservices and the facade implemented is the separate Eureka microservice. This is implemented by the @EnableEureka above the other microservices.


Client
Eureka
...
+subSystemOperation()
User
Authentication
Reservation
Equipment
Field