Assignment 1

Software architecture

We are developing a system where users can enroll for rowing activities. We found five bounded contexts and each of them will be mapped to one microservice.

Users

The user class will have the following attributes: UserID, Positions, Certificate, Gender, Organization, Level.

A user should have a **UserID** for identification. A **Positions** attribute is needed in order to express what roles he is able to fill (cox, coach, port side rower, starboard side rower, sculling rower). The **Certificate** attribute contains the most powerful certificate that the user possesses (in case the user can fill the cox position). The **Gender** and **Organization** attributes must be filled only if the user desires to participate any competitions that require this, while the **Level** attribute is requested only for certain competitions.

Activities

The activity class will have the following attributes: ActivityID, Date, Time, BoatType, AvailablePositions, Owner. The activity class is an abstract class extended by the subclasses Training and Competition. Competition has Organization, Gender and Competitive as attributes.

An activity should have an **ActivityID** for identification. The **Date** and **Time** attribute are used for storing when the activity takes place. The **BoatType** determines which type of boat is used for the activity, this can be a C4, 4+ or 8+ in our system and is needed to determine which users can take the role of cox in the activity. In the **AvailablePositions** attribute we store the number of needed rowers for each position. The **Owner** attribute is the user that created the activity.

The subclass **Training** does not have any attributes, it is the simplest type of activity without extra rules. The subclass **Competition** has the attributes **Organization** and **Gender** because all members of a competition team need to be of the same gender and organization. The **Competitive** attribute is used to state whether the competition is open to all rowers or to competitive rowers exclusively.

Matchers

The matcher class will have the following attributes: Activity, User.

We need a matcher to link enrolled users to each Activity and store that information in the database. A matcher should have an **Activity** and the corresponding pending **User** to process the matching for each user.

Notifications

The notification class will have the following attributes: ActivityID, UserID, Message.

We need to notify the users if they were accepted for the activity. A Notification should have an **ActivityID** and a **UserID** to send a **Message** to the user who is accepted for the activity.

Authentications

The authentication microservice is accomplished with the help of **Spring Security**.

Dataflow

- Authentication

First, the user needs to register to the app with the help of the Authentication microservice. After they register, the Authentication microservice will give access to the user if they introduce the correct login details.

- Creation of activity

Users can create an activity by sending it to the Activities microservice (**createActivity()**). They need to provide the values for the attributes *Date, Time, BoatType* and *AvailablePositions* and specify whether it is a training or a competition. In the case of a competition, they also need to provide the *Gender, Organization* and whether the rowers must be *Competitive*. The user will become the activity owner and their *UserID* will be linked to the activity. They can edit the available positions by calling the Activities microservice again (**updateActivity()**) or they can cancel the entire activity (**cancelActivity()**).

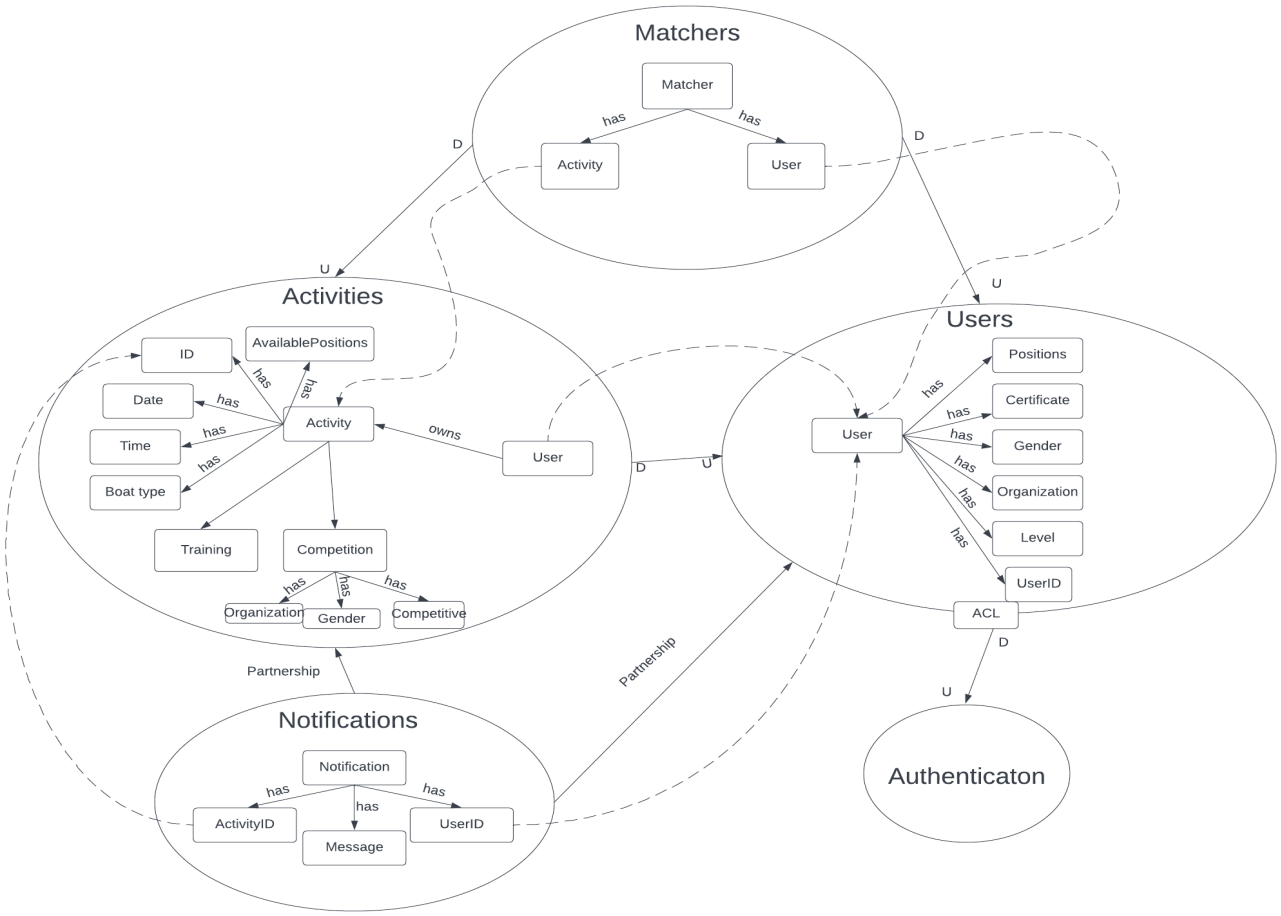
- Matching

When the user decides to enroll in activities, they fill up their available timeslots section and send a request to the Matcher (**requestMatch()**). The Matcher requests the available activities from the Activities microservice (**getActivities()**). After it receives them, the Matcher filters the one that are available for the current user. Afterwards, the Matcher sends the filtered activities to the user (**sendMatchingActivities()**). The user selects one of them and sends it back to the Matcher (**sendSelectedActivity()**). The Matcher stores the User and the selected activity in its database, as a pending match.

- Selection

Matcher sends the request of pending users to the owner of an activity (**sendUserRequests()**). The owner(user) selects users from the requests and sends them to the matchers (**sendAcceptedUsers()**). Then matcher should update the information of the participants (only accepted users) in the activity (**updateParticipants()**). Updated activity should send a request to the notification service to notify the accepted users (**notifyParticipant()**). Lastly, notification service sends a message to the accepted user to notify the selection (**acceptUser()**).

Context Map


Matchers
Matcher
has
has
Activity
User
D
D
U
U
Activities
ID
AvailablePositions
has
has
Date
has
Activity
owns
User
Time
has
has
Boat type
Training
Competition
has
has
has
Organization
Gender
Competitive
D
U
Users
User
has
Positions
has
Certificate
has
Gender
has
Organization
has
Level
has
UserID
ACL
D
U
Partnership
Partnership
Notifications
Notification
has
has
has
ActivityID
UserID
Message
Authenticaton

UML Component diagram