# Assignment 1 – Group 29A

"Rowing" is a platform that allows people to join rowing events such as trainings and competitions for their team. This program will help organizations fill up open positions with members that have signed up on the platform without having to ask around for available people.

## *FUNCTIONAL REQUIREMENTS*

After being presented with the scenario, the team has made a priority list for each of the requirements using the Moscow analysis method in order to make sure everything that the client has requested will be contained in the final version as true to the nature as it can be.

### 1. MUST HAVES:

- Users must be able to create a unique account on the platform (unique ID and password)
- Users must be able to log in using the created credentials
- Users must be able to fill out a profile (name, gender, schedule, positions and certificates)
- Users must be able to add events to the platform (trainings and competitions)
- Users must be able to see a list of all compatible events they can join
- Users must be able to join one event and queue for an answer
- Event creators must be able to accept/reject enqueued people
- Users must be kicked out of a queue that would collide with other events that they were accepted into

### 2. SHOULD HAVES:

- Event creators should be able to edit the list of people that are needed
- Users should be matched to events that start later than 30 minutes for trainings and later than a day for competitions from the enqueuing time
- The user should be logged out after a certain amount of time

### 3. COULD HAVES:

- An event creator could be able to edit the time of a training
- Users could be able to get notifications about event outcomes

4. **WON'T HAVES:**
   - The platform won't have a GUI
   - The platform won't have a certification validation/verification

## *NON-FUNCTIONAL REQUIREMENTS*

The platform must run on Windows 7 or higher and MacOS 10.8 or higher. It should be implemented in Java 11 using SpringBoot, Gradle and Spring Security. The implementation must be tested enough to have a code coverage of at least 50%.

The system should be able to handle a lot of data and be scalable (allow additions of certificates and boat types).

The passwords should be safely stored without clearly exposing them inside the database. The platform should limit the impact users can have on the system (eg: prevent SQL injections).

The platform won't have an encryption system for password storage (eg. One time pad, Caesar etc.).

## *ARCHITECTURE*

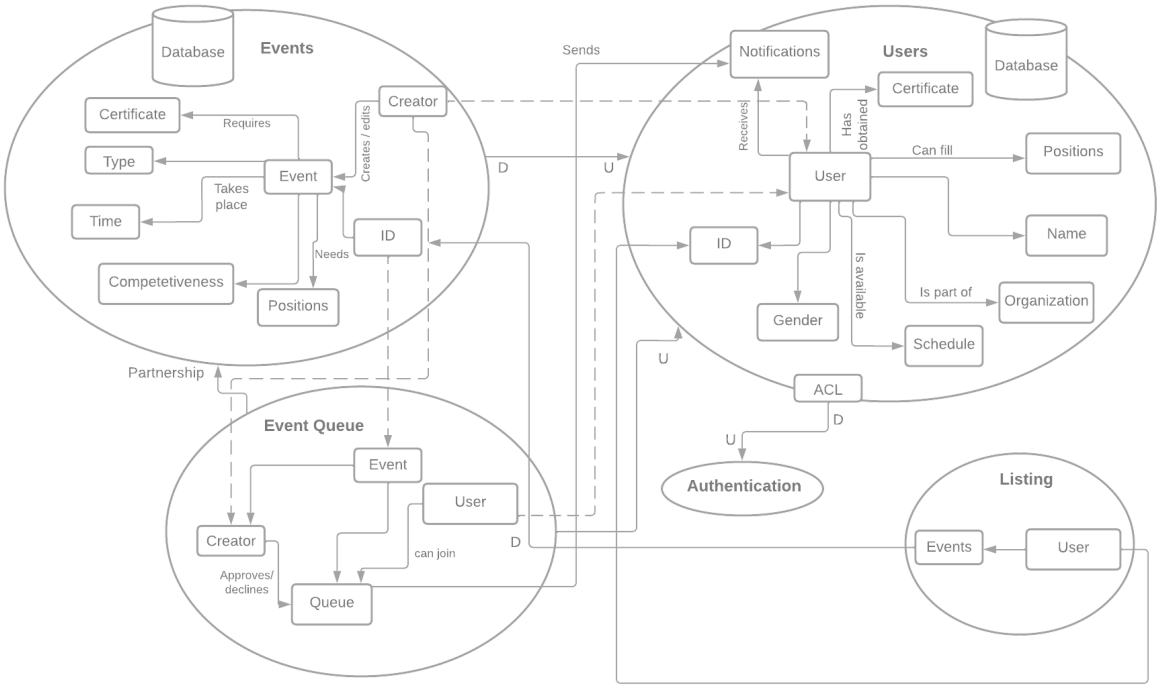
In order to realise the implementation, we have broken down the idea into smaller tasks. The main components of the program will be the **authentication system**, the **user** and the **events** (competitions and trainings). These will make the bounded contexts of the architecture. Another important part will be the position that one user can take when joining an activity, which will not be stored separately inside a database to provide a dropdown option to choose but be written manually by the user when filling the profile.

The authentication system will be a generic domain, as it will be needed to have a functional scheduling platform, but still needed so that we can have multiple people joining at once the platform. Each unique id will link a person to their account and information that will be needed to be matched to an event. This will make sure that users can't access and edit information published by others.

The user is one of the core domains of the structure as all actions will only be triggered on demand. Without having users, added events would be pointless to the platform. After passing the simplistic security measures, people can use "Rowing". They must first add information about themselves before requesting to be matched to any event. Users can interact with the list of events and will receive notifications based on their applications.

The event is the last core domain, as they will make the connection between the users and the real world. Events will be split into 2 categories: trainings and competitions. Users

will have an overview on matching events and will be able to see requirements to join them and the starting time, but not who has already joined the event. Events are allowed to be edited only by the creator. Due to the scalability of bigger events, competition schedules cannot be changed. Each event will be catered towards a certain group of rowers (competitive or amateur).




In order to maintain the performance of the platform in case of maintenance issues, we have decided to split the **events microservice** into 2: one that will only be used by the creators mainly (to create an event or edit information about if) & one that performs the process of enqueuing (allowing a user to join and be accepted/rejected).

After reading the client's requests, we have identified that the user and the events are the most important and most prone to change out of all the objects, which lead to the creation of databases for these entities.

**Users** will have a name, just for formalities, and an unique ID used for identification. In order to assign them to the correct type of competition, we are also storing the gender and the certificates that one has. The organization is something that will not have much impact, but was still requested. The schedule will contain all the available time periods for every day of the week, from Monday until Sunday. In order to avoid having to fill up a timetable each week, these will be recurrent, the user being able to change the schedule if something occurs in the meantime. Due to the fact that one person can be skilled in more than one rowing position, we will have a list that will reflect the users abilities in order to facilitate an easier process of finding an open spot.

**Events** are strongly correlated to users. These will have a creator who must accept or decline applications to join them, but can also edit various data about them, like time or the people the coordinator is looking for. Events can also be cancelled, which must also be done by the creator. In order to help filter easily, we will have 3 important facts about an event: the necessary certificate for rowers, the type of event (competition or training) and the level of experience. Each event will have a list of positions that are being wanted. If the list is full, people can no longer join this event due to the fact that a creator cannot kick already accepted members. There will also be a queue for every event that will contain every person that wants to take part in that activity.

In the eventuality that somebody requests to join multiple events that can take part at the same time, after receiving the first accept, they will be kicked out of the other queues to avoid collisions.

With this architecture, the different objects interact minimally, without causing any damage to the already enrolled members' experience if there are system faulties. Also, we will be using much less memory due to the fact that we will store only the essential data that must be accessible to every member of the platform.

## *MICROSERVICES*

**User microservice:**

- *Editing/ adding /deleting user information* - the microservice will handle all the operations regarding the user's data i. e. editing, adding and deleting information.
- *Requesting latest notifications* - the microservice will fetch all new accept/reject notifications belonging to the given user

**Events microservice:**

- *Creation* - a user can create an event by filling up all needs
- *Editing* - the creator can edit information about the event (except for the time if it's a competition)
- *Deletion* - the creator can delete the whole event

**Authentication:**

- *Logging in* - the whole process of logging in
- *Logging out* - the whole process of logging out

**Listing microservice:**

- *Match to all potential events after filling up all profile data* - match a user to all the events that they can join to using the information from the time that the request was done (not going to update on eventual updates of events/profile)

**Enqueuing microservice:**

- *Request join event* - a user can request to join an event if they meet all the specified requirements
- *Dequeue* - a user can dequeue from an event
- *Accept/reject & send notification* - Handling accepting or declining the user as a creator of the event, appending a notification about the decision.