Software Architecture

This document provides a comprehensive architectural overview of our (Group 34a) Rowing Association platform, using several different architectural views to depict different aspects of the system. It is intended to capture and convey the significant architectural decisions which have been made on the system.

The bounded context of our implementation is based on the relationship between users, competitions, boats, and certifications. In order to achieve the client's demands, we should divide our implementation of the application into four parts: Rower/Administrator (User), Competition/Training (Event), Certificate, and Authentication.

A User is the operator of the system. They have attributes such as ***id***, ***(hashed) password***, ***name***, ***gender***, ***experience level***, ***role***, ***available days***, ***boat preference,*** and ***event preference.*** The user is also the one that can ***join***, ***create***, ***edit*** or ***cancel*** an event. If they are the creator of the specific event, they should also be able to ***manage*** other users in their own events, being able to remove or add them.

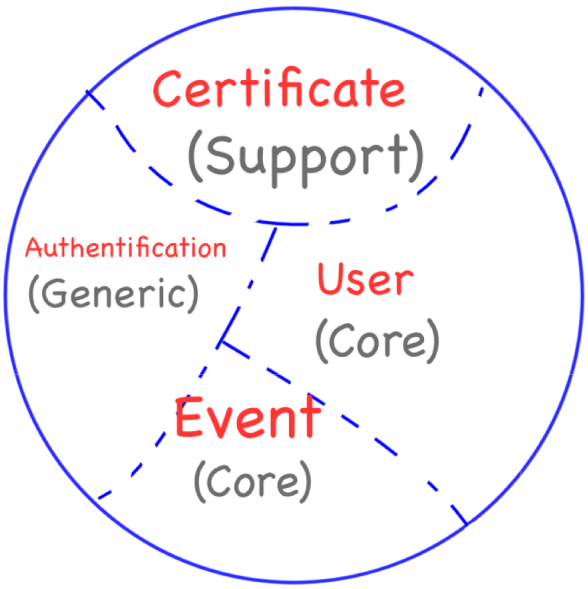
The event can be a ***COMPETITION*** or a ***TRAINING SESSION***. Based on that, each event has different requirements:

- A ***COMPETITION*** should be able to select users based on role and experience level. As such, every event has a different set of rules that it requires the users to abide by, for example, that all team members need to be from the same gender or that they all need to be from the same organization. If a rower is not eligible to be a cox on a certain boat type because of missing certificates for the used boat type, the ***COMPETITION*** has to reject them on the cox position from the get-go.
- A ***TRAINING SESSION*** is similar but it has less restrictions, for example different genders can train together. If it only allows competitive players

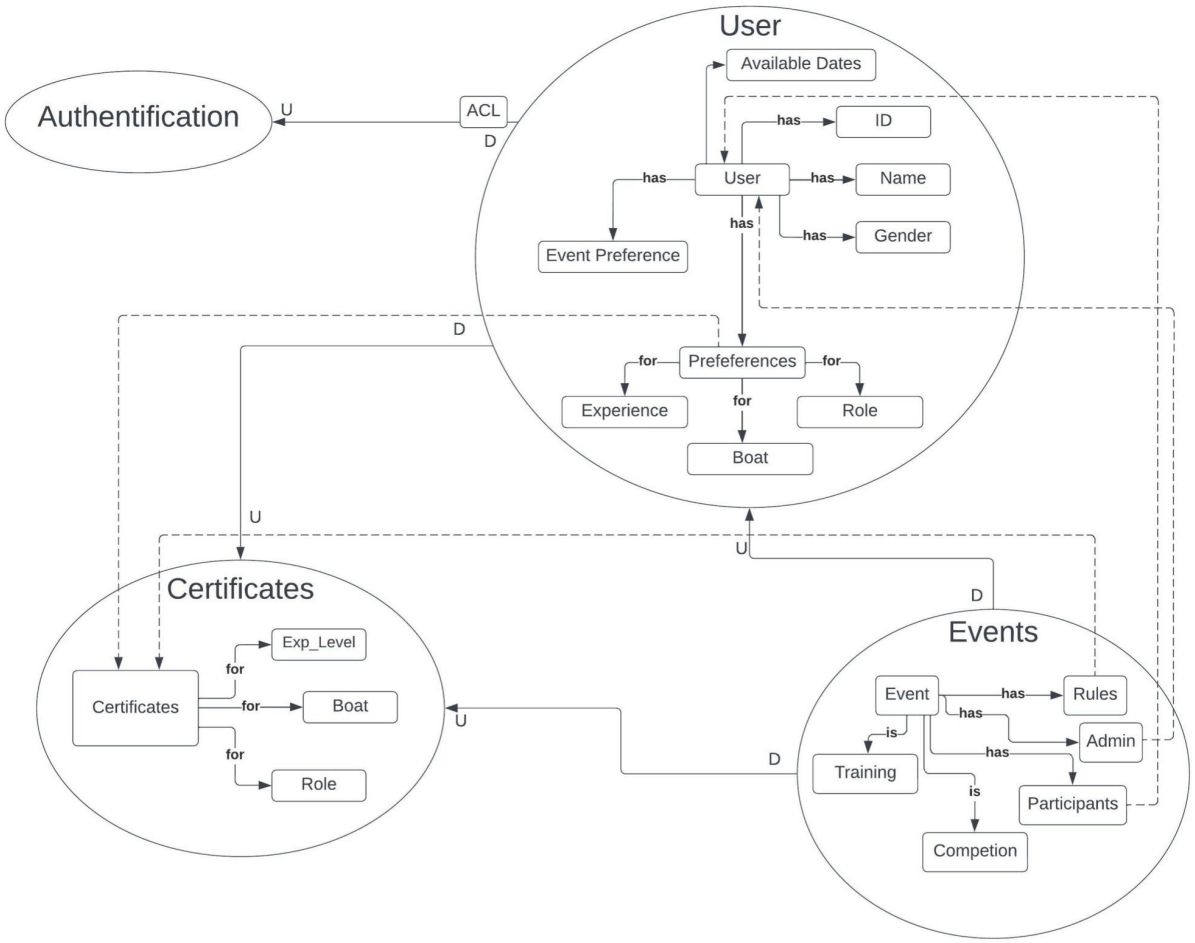
to train with each other, the system should pre-select the Users based on these criteria.

Because of the reasons mentioned above, we decided that it's best to include a ***role***, ***boat preference,*** and ***experience level*** separately from the User, in Certificate. This microservice encapsulates the concept of validating accurate User-Event match, by storing hash-based indices corresponding to the combination of boat certificate, position preference, and pro/amature. Users set out their certifications and preferences, and Events also decide the specific rules they implement based on certificates. Furthermore, a user cannot be accepted for placement in an event, when the ***COMPETITION*** starts within less than a day, or the ***TRAINING SESSION*** starts within less than half an hour. Finally, the user has to be notified when they are accepted for an event they wanted to participate in.

Authentication takes care of validating Users when ***selecting*** and ***managing*** Events, but also Certificate. This extra layer makes sure that no malicious party can impersonate an administrator, falsify a certificate, or any other unintended activity.



The **Users** and the **Events** have an Upstream-Downstream relationship, where **Users** is the Upstream part. The decision was made because **Events** not only depend on the rowers that apply for them, but also on the admins that manage them. **Certificate** has an open-host protocol, used by **Users** and **Events**. This is done because both these domains need to access a list of certificates in order to validate preferences for the **User**, but also rules for the **Event**. This is also done to expand the scalability/availability of the app. Finally, **Authentication** is an anti-corruption layer for the **User**.



As discussed above, it was decided to split the functionality of the system into four different services of which the **User** and **Event** services are the core domains.

The **User** microservice holds the data for all users and processes the requests to create new users and edit the data of existing users. The users are stored here with all the personal information needed by the services that implement the rowing application, which makes it possible to access and edit user data when other services are down.

The **Events** service handles the events that are created by the users, and the requests to join an event by other users. This means that the events service is very much dependent on the user service because the user data is needed for it to function correctly, though data can still be requested when the user service is not working correctly.

The **Authentication** service handles the security of the whole system by verifying the identity of the users. This directly makes sure that other users can not edit or change the personal data of other users nor can they edit the events made by others. This service is not directly implemented into the user service with the reason that the authentication is scalable and can be expanded to other services and not just the system for this rowing service.

Finally, the **Certificate** service is used to verify and store the certificates of users for certain positions and boat types. Certificates can be added per user, and the events service can request them to check if the certificate is valid for the applied position. By isolating this from the user service, the amount of data and validation between the event and user services is reduced a lot, which means less overhead and faster processing, because certificates are only requested for cox positions, while all other positions people can apply for do not require a certificate.

All of this together creates a highly scalable system that can be expanded whenever new features, requirements, or even whole new applications are needed to be implemented.