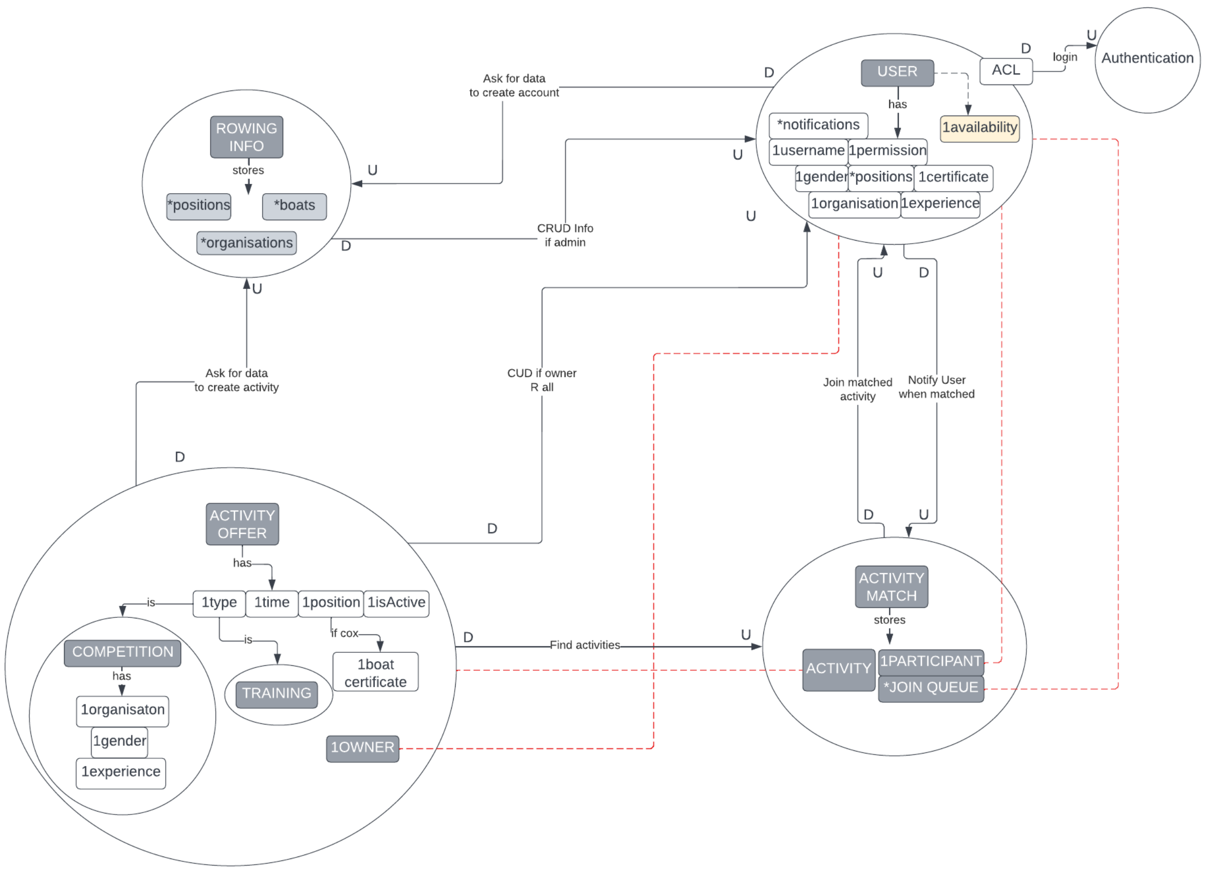
Software Engineering Methods 2021

Assignment 1

SEM group 34c

Detailed Bounded Context Map

After carefully studying the required product, we split the technical work that has to be done into a few core contexts.
We created the following detailed Bounded Context Map, to represent how we mapped out the bounded contexts and the relationships between them:



Bounded Contexts

Core domains: User, Activity Offer, Activity Match, Rowing Info

User is considered one of the core domains, since the user is the main entity that controls and interacts with the application. Whoever uses the application is considered to be a different user, and there are multiple types of users. We consider users in the following contexts:

- Activity: In the context of an Activity, the user can be the **Owner** of a rowing competition or training offer. The Owner creates the activity or competition offer and has the right to accept or refuse the other users that sign up. Whenever one creates that activity offer, the information about the available organizations, positions and boat, all this information is retrieved from the Rowing Info.

- User: Within the User context, everyone is a User
- Activity Match: In the context of an activity match, the user can be a **Participant**

Every user shall have a different username with a password. Also, when the account is created, the user shall specify his/her gender, the positions he/she could take in the boat, the certificates that he/she possesses (only applies for the cox position). Not only that, but also the organization and the experience (pro or amateur) as these two along with gender constitute essential criterias for participating in a competition. Another important aspect is the availability as it is crucial to know when the user might take part in an activity (so the application could provide the certain activities in the respective timeslot.)

Activity Offer is categorized as one of the core domains, as the software should allow the User to post offers for rowing training sessions and competitions. Having this functionality, posting offers for different Activities and getting matched with them, is the main purpose of our application. One Activity Offer is considered as a single "post", so when a user is in need of multiple people and positions for training, he has to post multiple offers: one for each person and position. The type of the Activity Offer determines if an Activity Offer is an offer for the training or competition.
We consider two different types of activities:

- **Training activity:** holds information about the User that created an offer (the Owner), the time that the training takes place, and the position which is needed for it. It also stores its state: whether it is active or expired.
- **Competition activity**: holds the same information as the training activity, plus the requirements of the Participant: their gender, organization and experience level.

The data in the Activity Offer can only be changed by the owner. Generally, this data is fixed and will not change - even when another User wants to become a Participant of this Activity (that is managed in the Activity Match context).

Activity Match
The Activity Match microservice is a core component of the system, responsible for matching users with offers based on a set of specified requirements. The microservice maintains two tables in its database: a queue of pending activity offers that users have applied for but have not yet been accepted by the activity owners, and a table of persistently stored activity matches. When a user sees an activity that they are interested in, whether it be a training or competition, they can make a call to the appropriate endpoint provided by the microservice (i.e. "joinQueue/competition" or "joinQueue/training") to express their interest. When the owner of the activity subsequently views their pending offers, they can make a call to "getPendingOffers" with the owner's NetId to retrieve all of the pending activities. Once the owner has decided on a winner for their activity, they can use the "setParticipant" endpoint to store the NetId of the owner, the NetId of the activity, and the NetId of the user that was accepted. A user can also use the "getUserMatches" endpoint to retrieve all of the activities that they are currently involved in. The data in the Activity Match microservice is dynamic, as it is updated whenever a user applies to join an activity or when an actual match occurs. The owner of an activity also has the ability to delete their activity, in which case a call is made from the Activity Offer microservice to the

Activity Match microservice's "deleteActivity" endpoint, resulting in the removal of the activity from both tables.

Rowing Info is also categorized as one of the core domains because it is a repository that the User and the Activity Offer microservices can access to query and update generic information about rowing associations in Delft. If a User has admin permissions, he can also create, edit, and delete entries in the repository. In addition, both the Activity Offer and User microservices have endpoints for checking the certificates and the organizations. (for instance the "/doesOrganisationExist" and "/doesCertificateExist"). Nonetheless, the rowing has 2 databases for storing the available organizations and certificates.
It encapsulates the following three collections:

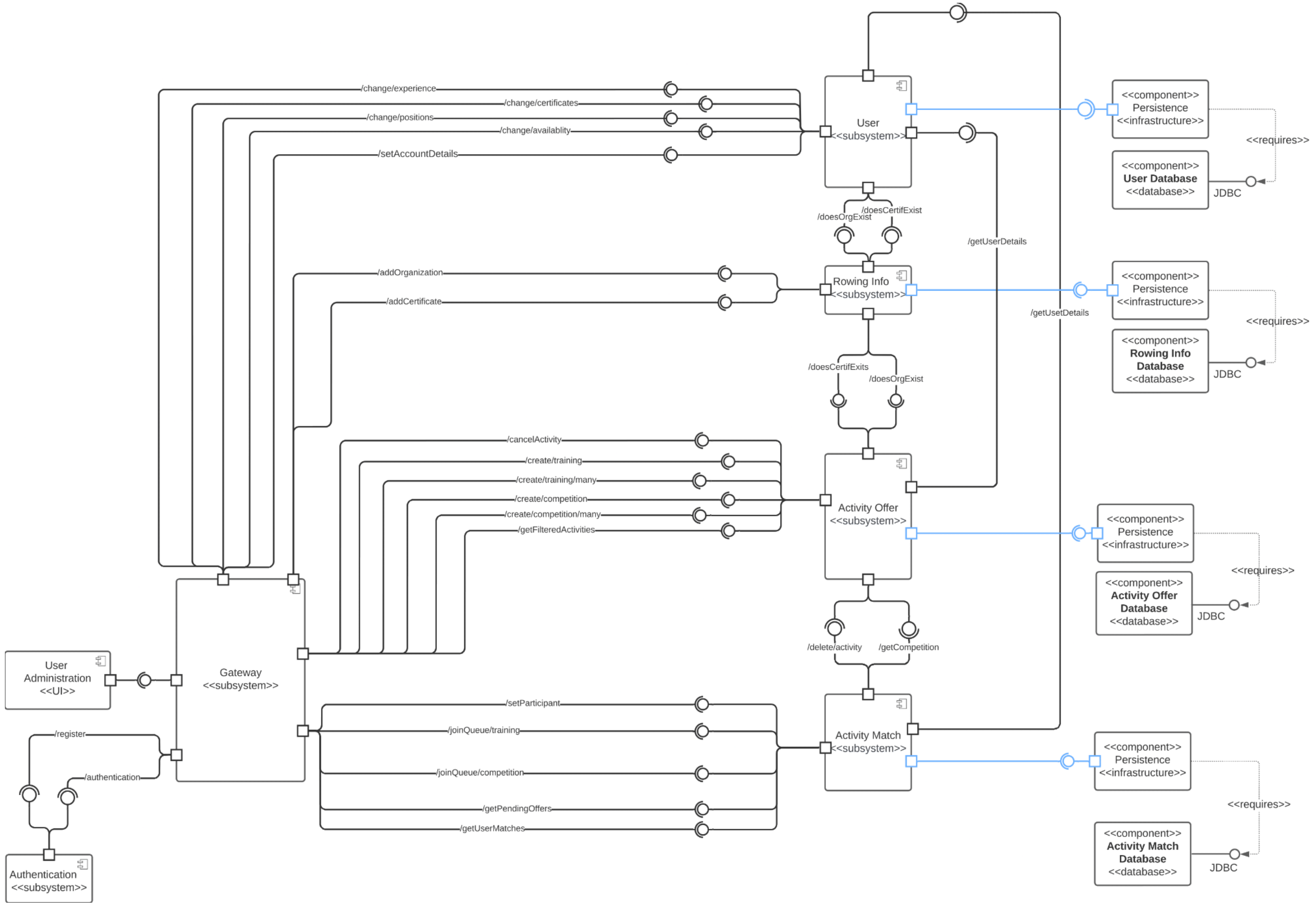
- **Positions**: All possible positions that a rower can select. By default, the following positions are supplied: Cox, Coach, Port side rower, Starboard side rower, Sculling rower.
- **Boat certificates**: A list of boat certificates (such as a C4, a 4+, or an 8+) needed to row boats used by rowing associations in Delft.
- **Organizations**: A list of names of all associations in Delft.

Generic domain: Authentication

Authentication is categorized as a generic domain in our system, as it is not specific to this system, and it allows the application to handle User login, signup, and permissions. The Anti-Corruption Layer (ACL) is a translation layer between the User context and the Authentication context, as they do not share the same semantics. Security is handled by Spring Security. Users log in or sign up with a unique string that identifies them, and with a password.

Component Diagram

We created the following component diagram, to model the high-level software components, and illustrate the interfaces to those components and how they interact: (see next page)


User <<subsystem>>
Rowing Info <<subsystem>>
Activity Offer <<subsystem>>
Activity Match <<subsystem>>
Gateway <<subsystem>>
User Administration <<UI>>
Authentication <<subsystem>>
<<component>> Persistence <<infrastructure>>
<<component>> User Database <<database>>
<<component>> Rowing Info Database <<database>>
<<component>> Activity Offer Database <<database>>
<<component>> Activity Match Database <<database>>
<<requires>>
JDBC
/change/experience
/change/certificates
/change/positions
/change/availablity
/setAccountDetails
/doesOrgExist
/doesCertifExist
/getUserDetails
/addOrganization
/addCertificate
/getUsetDetails
/doesCertifExits
/doesOrgExist
/cancelActivity
/create/training
/create/training/many
/create/competition
/create/competition/many
/getFilteredActivities
/delete/activity
/getCompetition
/setParticipant
/joinQueue/training
/joinQueue/competition
/getPendingOffers
/getUserMatches
/register
/authentication

UI client

User Administration <UI>

This is the front-end part of the application: a UI client that connects to all the microservices. There is only one UI client since there was not a need for multiple types of users. Apart from a small permission level (to create or edit data in Rowing Info), all users have the same functionalities and access. The user interface is not going to be implemented since the SEM project only focuses on the backend.

Microservice components

Each core domain in the bounded context has its own microservice. We named them as follows:

Authentication

This microservice is responsible for the security of our application. It is implemented using Spring Security access-control framework. The credentials are encrypted and stored safely in the User database. After the user creates his account with the corresponding endpoint, they can log in in the application through this microservice. This microservice uses the Chain of Responsibility design pattern. When the user logs in, the microservice provides a JSON Web Token (JWT). The user needs this unique token to access endpoints to their specific information and to filter and apply for activities.
We chose to include this microservice in our application because security is an important part of the functionality of the system. Moreover, we have made good use of the authentication template and we managed to include it to our system and adapt it fairly easily to our microservices like User microservice.

Gateway

This microservice is responsible for getting the request from the front-end, and then forwarding the request to the appropriate microservices. Those microservices are responsible for user authentication.
We chose to implement this microservice as it is very important for the overall structure of the application. It acts as the literal gate from the outside (the user part) to the inside of the system, to all the microservices. The users cannot access the other individual microservices, they access the gate that sends the request for them.

UserService

This microservice is responsible for creating the account of the user and storing information about them. It is an individual microservice, because the whole system is relying on the users. The database for this microservice stores the users' details which are key for the further matching and assigning users to the trainings and competitions.
Public methods contained in UserService are:

- Creating a new account by adding its data to the database

- Provide the data of the User like: organisation, certificate, positions that a user can fill, gender, username, is a user a professional rower and his availability. This data is consumed by both the ActivityOfferService as well as the ActivityMatchService.
- Change the details of the User like: availability, organisation, positions, level of experience, password
- Send the timeslot when the user is available
- Send accepted user(for offer) from User microservice to ActivityMatchMicroservice.
- Store the notification for the user if they have been selected for an offer (needed by ActivityMatchingService)

ActivityOfferService

We decided to have this microservice, because creating an offer is a main interaction that a User can take in our application, so it should be isolated from other parts in order to ensure good maintainability. This microservice has a database in which all of the already created Activities are stored, no matter if they are expired or not. We decided to also store outdated offers, because users might want to take a look at the past Activities. An admin can clear the database from outdated offers anytime, so the database will not be clogged. This microservice is used for creating offers for activities by Users. Public methods contained in it:

- Adding a new activity that is owned by the User that created it.
- Adding multiple activities in one request, so in the case that User needs multiple people to join, he can do it at one.
- Editing activities and marking them as expired by an Owner.
- Getting all of the created Activity Offers that are currently in the database.
- Getting a list of activities that match a specified filter. Needed by the ActivityMatchingService.
- Canceling an Activity Offer (only by an Owner of that Activity) that was created, but for some reasons is not valid anymore.

ActivityMatchingService

This microservice is responsible for matching Users to Activities, and sending out notifications to the Owner when someone applies and to the Participant in case the Owner accepts or rejects the User.
Public methods exposed by this microservice:

- Finding activities that match the filters specified by the User.
- Joining an activity.
- Accepting a User to join the activity.
- Reject a User to join the activity.
- Finding activities that a user was chosen for
- Finding activities all the activities owned by an Owner

RowingInfoService

This microservice is a wrapper around the repository of general information about rowing associations in Delft, and the sports of rowing in general. It exposes the following public methods:

- Getting a list of ***organizations*** that the user can join, the ***positions*** that a User can select, and the *boat* ***certificates*** that a user can hold when they update their user settings or when they create an activity.
- Creating, editing and deleting the ***organizations***, ***positions*** and ***certificates***.

We decided to have this microservice, because information stored in it is used in the entire application (in the user settings, creating an activity, and edited by the admin). So we isolate this repository into its own container.