Assignment 1 - Task 1 Draft

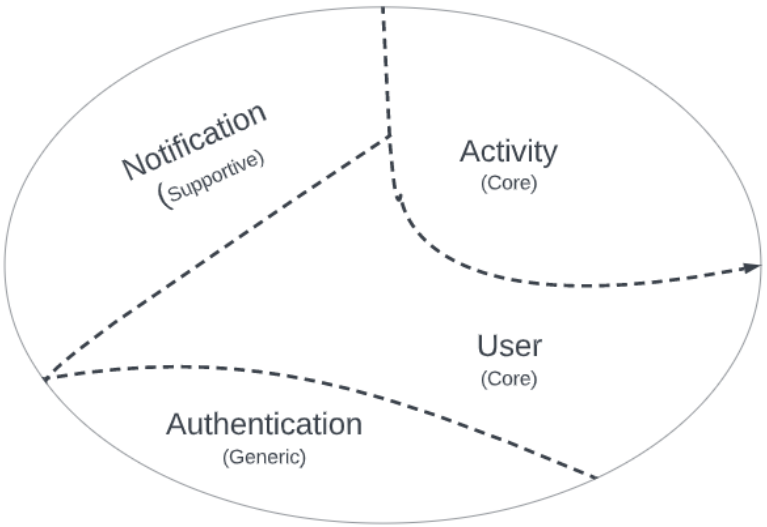
(CSE 2115 - Software Engineering Methods 2022 Project)

Bogdan-Andrei Bancuta
Alexandru Gabriel Cojocaru
Khalit Gulamov
Alexandru-Nicolae Ojica
Wing-Yan Joyce Sung
Efe Unluyurt

For the Software Engineering Methods 2022 course our team is going to implement an application to solve the issues with finding teams of the rowing associations in and surrounding Delft. The application will match available people to training occasions and competitions that still require participants. In order to make sure our application is modular and scalable we will be using a microservice architecture.

Bounded contexts

Firstly, in order to decide on the design of our application we have decided to apply Domain-Driven-Design. The first step is to establish the bounded contexts. The team started taking key elements from the assignment's description and drew 4 bounded contexts as follows : Activity, User, Authentication and Notification.

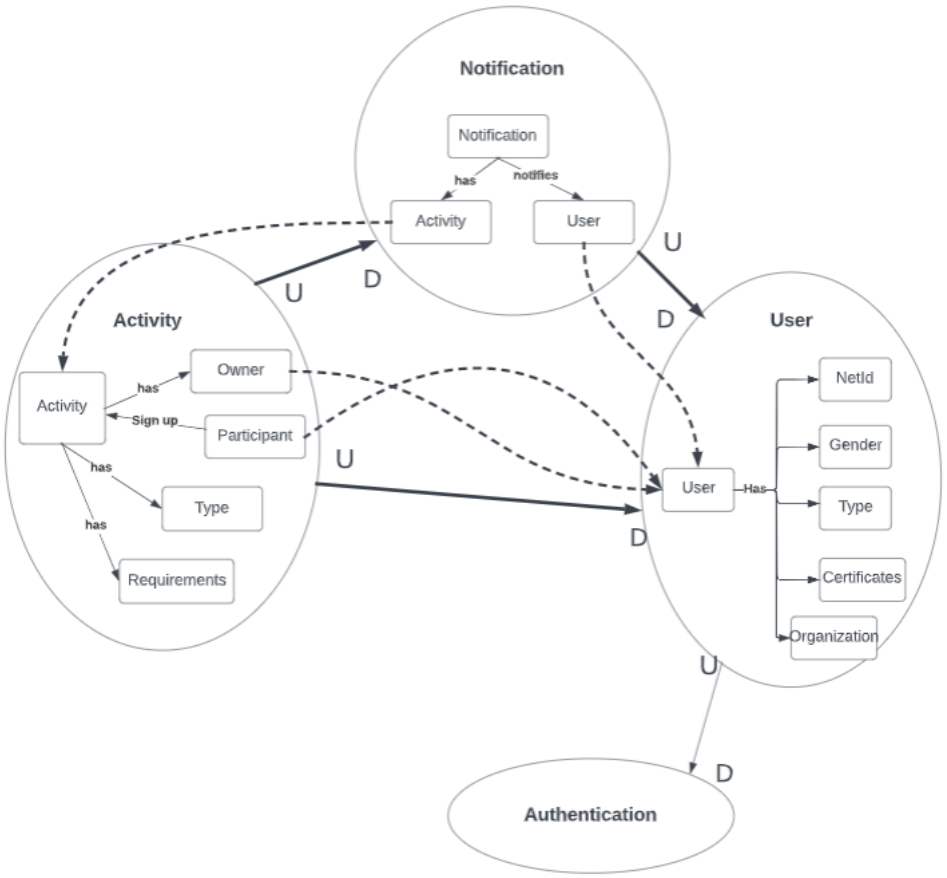


After identifying each context we assign one domain type out of the three: Core, Generic or Supportive. The core domains represent the most important part of the system. Generic domains are not core domains but the core depends on these services. Lastly, the Supporting domains are additional technical domains that help support the core but they are not critical. Our core domains are the User and Activity domain. These two will deal with most of the functionalities. Notification is our

supporting domain which is not a necessary feature for the application to work but it will be extremely useful in later developments. Authentication is our generic domain which is, even if is not a core domain, the pillar that supports the 2 core domains.

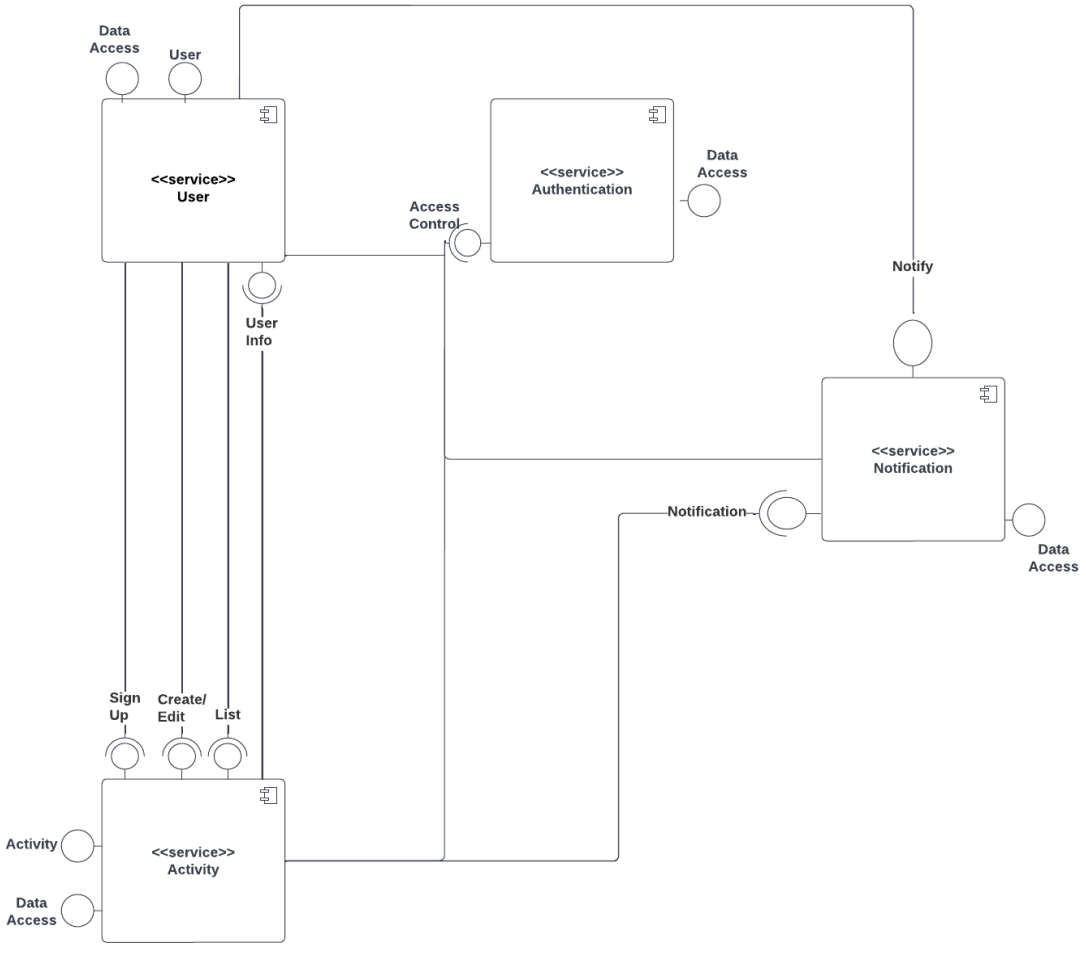
Microservices

After identifying the bounded contexts the team designed the context map in order to represent the interactions between each domain. From this we decided to make a separate microservice for each bounded context. We chose not to implement a separate microservice which handles the interaction between activities and users, since things that are changed in the activities are almost always facilitated by the User microservice. Therefore having a microservice that handles interactions between Activity and User, would make the Activity microservice redundant. Since these functionalities are now included in the Activity microservice, we decided to not also include the notifications functionality in the Activity microservice and instead keep that in a separate microservice, as we thought this would affect the modularity of our system.



This diagram showcases not only the way data traverses in our system from a microservice to another but also how each microservice interacts with the data inside of it.

The last step we take in order to get a good grasp on the design of our application is how different APIs interact with each other. The context map only shows interactions when considering the bounded contexts, but at this point we need to translate this information into a real world setting, so we started to draw a UML diagram. The diagram contains the 4 microservices relating to each bounding context. We use lollipops to illustrate bidirectional relations between the api endpoints and sockets to illustrate one-directional relations. Each endpoint either represents a functionality or is a data access socket where we showcase the microservice communicating with its internal database.



User microservice

The User microservice is responsible for keeping track of the user database, this includes the creation of new users and editing of attributes connected to the user.

The different attributes contained in the user database include a user ID for every user, which must be unique, since this ID will also be used for authentication. Other attributes of the user include the positions the user can fill (cox, coach, port side rower, starboard side rower or sculling rower), their availability and which cox certificates the user has. Additionally the attributes can be expanded to include the gender of the user, which rowing organisation they are part of and whether the user is an amateur or competitive rower, which is important for organising competitions in our applications.

A user must be in the user database and logged in to be able to use the application. When the user creates an account they will be added to the user database and the User microservice will interact with the Authentication microservice to secure the account with a password. Later users can log in at which point the User microservice will interact with the Authentication microservice to complete the log in. When users are logged in they must have access to their own details, but must not have access to details of other users in the user database. Personal details should be editable by the user using the User microservice, this will lead to permanent changes in the user database.

Activity microservice

The Activity microservice itself will be responsible for the Activity Database, where we will store the properties of an activity. Each activity will have an owner stored, who creates the activity and accepts/declines the sign-ups for an activity, or can kick the members. The users that are accepted, declined, or kicked out will be notified. A user can be an owner for more than one activity, and there can only be 1 owner for each activity. This owner can either participate in the activity (have a role) or not, since an activity owner can be someone who wants to own a team but not participate in the team itself. The owner will also be the only one to be able to edit the requirements for the activities that they own. There will be 5 different positions that can be filled in an activity, which are a cox, a coach, a port side rower, a starboard side rower, and a sculling rower. Everybody can apply for any role, except “cox”, since being a cox requires a certificate per boat type. An activity will also have a type stored, which is either a competition or a training session.

Depending on the activity type, different requirements will be applied such as all rowers in the boat needing to be of the same gender and same organisation. Every activity will also have a start time, since the users will only apply for the activities which start at the time that they are available. After this time, the activity will expire

and the users will not be able to apply for the positions.

After the owner creates an activity, the users are free to apply to the positions that they are eligible to. The users that are selected for a position will also be stored in the database. In order to keep the activities database consistent, the microservice will handle all the requests of updating, creating, deleting or retrieving information about a certain activity. This in turn, will ensure that no duplication or any out-of-date information will ever reach the database.

Moreover, in order for the system to be able to function correctly, the Activity microservice needs to be able to communicate with the other microservices we have designed. First of all, it should be able to receive requests from the user microservice about joining, editing or creating an activity and replicate the desired changes into the database. It should also be able to return a list of all the available activities back to the user if they requested so. Any request made to the Activity microservice by other microservices will be checked for validity by communicating with the Authentication microservice, which will check if the token of the user performing this action is valid.

Furthermore, in order to inform a user they have been accepted or kicked out of an activity, or any other changes to the details of it, a communication to the Notification microservice should be established.

Notification microservice

We created an additional microservice for notifications so that other microservices are only responsible for doing the job dedicated to them in order to preserve modularity. Notifications in our application are used to keep the users up-to-date about their status regarding the activity they signed up for. The microservice is responsible for sending notifications to the users after some of the actions made in other microservices such as being accepted to an activity. In the future we can include some other actions which can trigger notifications.

The Notification microservice receives information from the activity microservice such as information about the notification receiver and activity information with owner information to send to this receiver. It should also receive the token of the user who made the action leading to a notification, to check whether it is a valid user. For this reason the notification microservice has a bidirectional connection with the Authentication microservice, such that it can check whether the user token is valid. This is done to prevent any non-users from interacting with the Notification microservice to prevent requests from outside the system to happen. It has a one-directional connection with the User microservice only. This is used to send a user, which was indicated as a receiver from the activity information, a notification message. In further development stages we can change the functionality to possibly also be used for updating an user regarding their rejection in a certain activity or letting users know when a new activity suitable for them is created.

Authentication microservice

The Authentication microservice is responsible for verifying users' identity, keeping track of the logged in users and logging them out after a certain inactivity period. When a user logs in, the Authentication microservice will check the credentials by trying to find them in the database, and in case they are correct it generates a token and stores it as long as the user remains logged in or an inactivity timeout happens. Every time a request is made to one of the microservices it will include the token and the requested microservice will go through the AccessControl endpoint to check its validity with the Authentication microservice. After the validation is checked the Authentication microservice will send a status response back to let the other microservice know the operation can continue.