CSE2115 Software Engineering Methods

Assignment 1 - Task 1

Authors

Bogdan-Andrei Bancuta - 5507340
Alexandru Gabriel Cojocaru - 5520967
Khalit Gulamov - 5491541
Alexandru-Nicolae Ojica - 5477484
Wing-Yan Joyce Sung - 5011825
Efe Unluyurt - 5452481

16 December 2022

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 (fig 1).

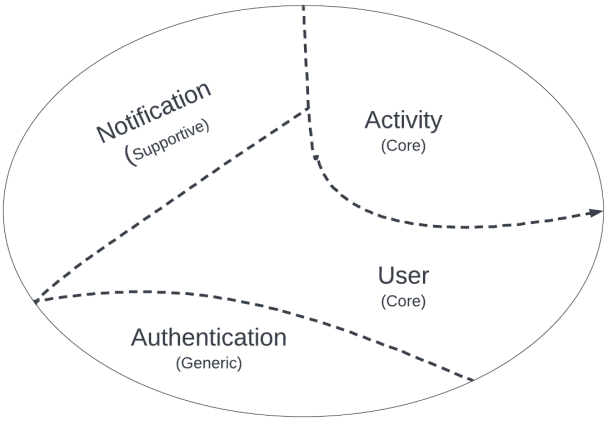

Figure 1: Bounded contexts

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.

The context map (fig 2) 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.

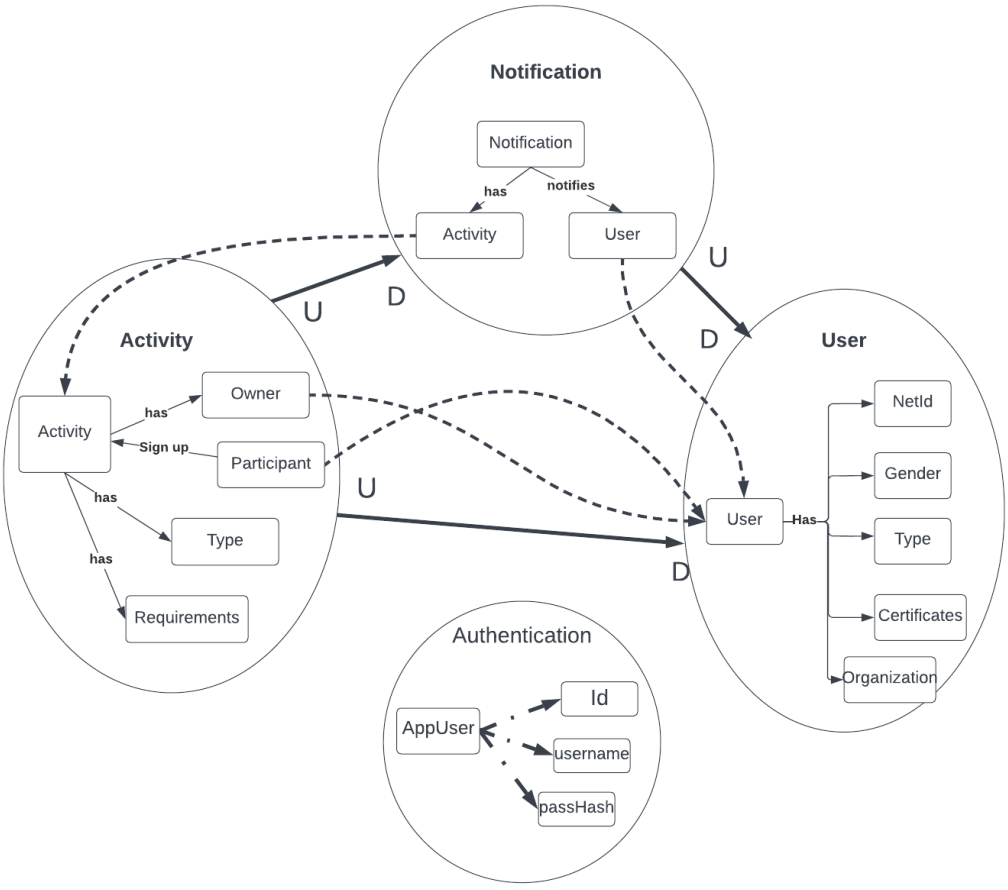

Figure 2: Context map

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 (fig 3) 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.

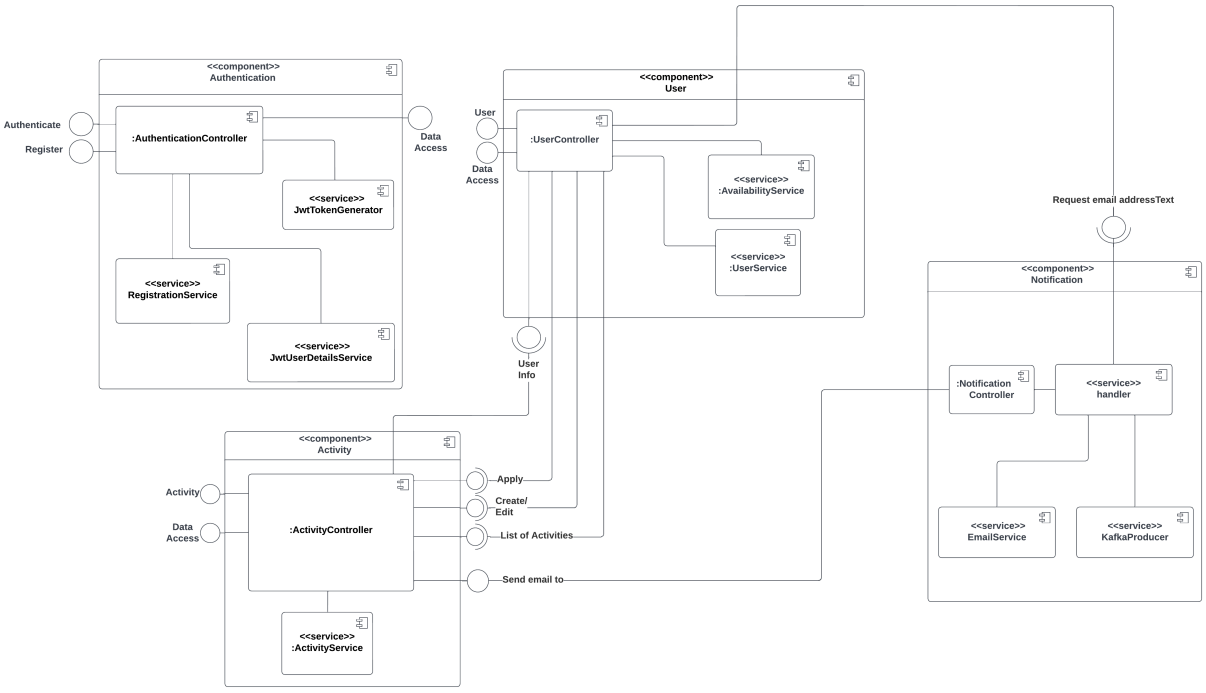

Figure 3: UML diagram of system architecture.

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. With the creating of an account the users gain the ability to create/edit their profile. 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 the role of 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 by validating the JWT token inside the header.

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 added a separate microservice for handling notifications to ensure that other microservices can focus on their specific tasks and maintain modularity. The notifications in our application are used to keep users informed about their status in relation to the activity they have signed up for. The notification microservice is responsible for sending notifications to users after certain actions are taken in other microservices, such as being accepted to an activity. In the future, we may also include additional actions that trigger notifications.

To send notifications, the notification microservice receives information from the activity microservice about the activity itself, as well as a user token to verify the identity of the user who initiated the action that triggered the notification. This is done to prevent unauthorized requests from outside the system. The notification microservice then uses the user ID included in the token to request the user's email address, which is used to send a notification message to the designated recipient. In future development, we may also update the functionality to include notifications for rejections, deletions from an activity, withdrawals from an activity, or notifications about new activities that may be of interest to the user. If the user's email is not found in the user database, the handler inside the notification microservice triggers the kafka producer service that communicates with a Kafka server instead of an email service. The kafka producer service sends a message to a kafka topic(this can be configured but by default it is the username of the user being notified). A motivation to why we chose to use this asynchronous message queue can be found under the Microservices section.

Authentication microservice

The Authentication microservice is responsible for verifying users identity and generating a token that allows the user to retrieve information and perform actions on the other microservice. When a user registers the Authentication microservice stores the credentials if no other account with the same username is found. 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 that has an expiration time. Each of the other microservices are able to decode the token and use the information stored inside it for each request the user makes.

Communication

Our design choice

We chose to implement both synchronous and asynchronous communication, but not for the same purpose. Let's start with the synchronous part.

The user interacts with the application using REST Api in a synchronous manner. More specifically, all requests to log in, register, modify/view profile details, manage activities and view/sign-up to activities are made in a synchronous manner, for each of them the user getting a response. The microservices also communicate with each other synchronously.

Synchronous communication between microservices can be useful in certain situations because it allows for real-time exchange of information between services. This can be particularly useful when the services need to coordinate their actions in order to complete a task or process.

One concrete example of synchronous communication between microservices in our project is when the Notification microservice needs to make a request to the User microservice to get the email address of a user and wait for a response before it can continue actually sending the email.

Synchronous communication can also be useful in situations where it is important to ensure that a specific sequence of events occurs in a particular order. For example, activity microservice needs to update a user's status for an application and then send a notification to the user, it may be important to ensure that the notification is not sent until after the status has been updated.

Now moving to the asynchronous part of our communication we need to go back to the Notifications microservice. What if a user doesn't have an email address? We still somehow need to notify them. Thus, we decided to use Apache Kafka, a highly scalable and powerful asynchronous message queue.

One advantage of using Kafka for sending notifications is that it can handle a large volume of messages without requiring the Notifications microservice to wait for a response from the user. This can be particularly useful in situations where the microservice needs to send a large number of notifications, as it allows the sender to continue processing requests without being blocked by the time it takes to send the notifications.

Another advantage of using Kafka for sending notifications is that it allows the microservice and user to operate independently of each other. The sender can publish messages to Kafka without needing to know the specific details of how the notifications will be delivered or consumed. This decoupling can make it easier to update or modify the sender or receiver without affecting the other component.

In addition, Kafka's use of message queues and consumer groups allows for scalable consumption of messages. This means that the receiver can process notifications at its own pace, potentially using multiple consumers to parallelize the work or just receive them on multiple devices, and the sender can continue to send notifications without being affected by the receiver's processing speed.