SEM Assignment 1 - draft - group 33A

Laimonas Lipinskas (5559375)
Lotte Kremer (4861957)
Nicolae Radu (5527104)
Razvan Loghin (5480620)
Dawid Plona (5205018)
Vlad Nitu (5529557)

December 2, 2022

Bounded context split explanation

We have used Domain-Driven Design (DDD) on the scenario we were assigned to (Rowing) in order to identify different bounded contexts. Firstly, we came up with some keywords that helped us in separating the domain, such as: *user, activity, training, competition, authentication, account, password, time slot, position, member, availability* and *ID*. Consequently, we used these keywords in order to make the coupling of our system as loose as possible, while also keep in mind that we should maximize the cohesiveness of each separate sub-domain. Thus, the "User" microservice is a Core domain mapped by *user, member, time slot and ID* because our system could not exist without persisting the users' accounts credentials. The same goes for "Activity", which is mapped by *activity, training and competition* keywords, as it is the second entity without which we could not persist possible activities in the system. Moreover, the "Matching" microservice establish the pairing between users and activities they will take part in, so it is also vital for our logic flow. It is mapped by availability. We chose to model "Notification" as support domain, as it is responsible for notifying the users when they are found a matching activity, which is not crucial. Lastly, the "Authentication" component is mapped by the following keywords: *authentication, account, password* and we consider it a generic domain, as the User (which is a core) would not be able to interact with the system if it was not for creating an account and then log in. See Figure 2: The Bounded Contexts diagram

Matching

The "Matching" represents one subpart of the bounded context and it was designed by us to comply with the microservices architectural pattern and to decouple the communication between different components of our system, such as "User", "Activity", and "Notification". It also persists the pending matches that are still flowing through the network and all the future activities that will be approved.

The "Matching" microservice acts as an API Gateway and has the responsibility of administrating the communication between users, activities, and notifications. The logic flow of our system is as following:

1. The "Matching" component receives a request from a user, which has just specified his available time slots and personal details. Afterward, the "Matching" filters the time slots that it can pair the user to such that a new activity will not overlap an already existing one. Another duty of the filtering that is being driven by our central microservice is to ensure that users would not be informed about activities that they cannot participate in. Then, it forwards the request to the "Activity" microservice.

2. Possible activities that would match our user's availability are returned to our main microservice, which are then passed to the user to select the ones that he/she would like to take part in, and for which they are eligible for. Subsequently, for each activity selected by the user, a new instance will be created in the Matching database through an API call of the "Persistence" method (see Figure 4. Component Diagram), provided by "Matching Repository". This instance will have the following attributes: userID, activityID, position, and a boolean flag "pending" set to True, meaning that a matching has not been approved yet.

3. After we persist a pending matching, an API call to the "Notification" microservice will be made to announce the owner about the possible matching that has been made to his

activity.

4. An owner can accept or decline a possible matching. If it accepts it, then the "Matching" microservice will:

   - set the "pending" flag to False
   - make a call through the "takeAvailableSpot" Activity's API endpoint to mark this specific activity instance as occupied, and to ensure that no overlaps will occur.
   - talk through another API call with the "Notification" microservice in order to announce the user that the owner has given consent for his participation in this specific activity.

   If it declines, then the pending entity gets deleted from the database, as we no longer need to consider this user-activity pair.

Authentication

The "Authentication" represents another subpart of the bounded context identified by the group. It is designed to implement the microservices architectural pattern, as well as clearly separate and extract the authorisation and authentication process, its logic and storage, from the remaining parts of the system. It is used to allow, facilitate and secure the communication and data flow with "User" component.

The "Authentication" microservice acts as an API endpoint for the access requests made by users and encompasses all the necessary, underlying logic of the process of logging into the system. The overall logic flow of this part of the system is as follows:

1. New user registers in the system for the first time by creating their credentials. These consist of an ID - userID attribute - stored in the "Authentication" storage as a String, and a password - password attribute - stored in the "Authentication" storage as an encrypted String. The values of these attributes are passed to "Authentication" as a POST request. Subsequently, the credentials are stored in the corresponding database. The password encryption and data flow safety is achieved with use of Spring Security within the microservice's internal logic.

2. Existing user provides their userID and password by means of a POST request to "Authentication". The provided userID, and password encrypted within the internal logic, are compared against the currently stored credentials' list. If a match is achieved, the "User" is notified of the authorisation success. Otherwise, "User" is notified of the authorisation failure. Both mentioned possibilities are achieved by means of an API call to "User" and differ by the passed parameters.

The process governed by "Authentication" may be further broken down in the following way:

Authentication:

- Encryption
- Identity verification

Authorisation:

- Access level determination
- Access grant

Notification

The Notification bounded context is used for acting as a messenger from "Matching" to the user about certain matching developments. The intention is to separate this communication away from Matching in order to have it focus more on the matching part, thus providing nice isolation for the microservice pattern.

The "Notification" microservice acts as an API Gateway which is called from "Matching" to notify either the owner of the activity that a new application for the slot has been created, or the applicant that that they have received the slot. The basic flow of a notification is comprised of:

1. "Matching" needs to notify the user of the developments regarding the requests to join activities.
    - If there has been a new application for an activity by some user, matching requests a notification to be sent to the target and the "Notification" microservice persists a database entry containing these attributes: userID which belongs to the applicant, targetId which belongs to the owner of the activity and the activityId so that the owner could know which activity this request relates to.
    - if matching requests a notification of type application acceptance all of the fields are the same, the only difference being that both userID and targetID are the same.

    Persistance is needed because if the service were to send the message as soon as it receives it and the owner was offline, the notification would be lost.
2. When the user is online and queries the "Notification" service endpoint for messages (possibly with short polling) by providing their Id, the service returns the notifications that have been accumulated for that specific user.
    - If the userID is not equal to targetID it means that the notification type is of a request to join the activity that belongs to activityID and the owner has to resolve it with matching.
    - Otherwise it is a notification of an accepted application.

User

One of the other core domains in our Bounded Contexts Diagram is the "User". It was decided that this should be a core domain as the whole system depends on a functional "User". The "User" micro-service that we designed out of this stores all the information necessary about the user and communicates this through the APIs of the other microservices. All the information stored in the "User" microservice can also be found in Figure 1, the UML Diagram. In the following paragraph the role of the microservice itself and its logical flow is:

1. First of all before the "User" microservice can communicate and receive requests from the "Activity", "Notification" and "Matching" microservices, it should make a call to the

"Authentication" microservice. Only when this microservice gives a message to the API of the User microservice that the authentication was successful, the "User" microservice can undergo further actions.

2. The API of the "User" microservice can now receive the messages from the "Notification" microservice, if there are any.

3. The "User" microservice can also make a call to the API of the "Activity" microservice to create a new activity of which it will then be the owner. *We are aware that this is not yet soon in Figure 4 and this will be added next week.*

4. The "User" can also communicate with the API of the microservice "Matching" in multiple ways.

   - Whenever a user wants to look for a new match the "User" API first sends the necessary information about this user to find available matches through getAvailableActivities. This included information about the user's free timeslots, the positions it can take in rowing and its personal information such as gender.
   - The API of "Matching" would then sendAvailableActivities. The user can then decide through the "User" microservice which activity it wants to choose. This information is then send back to the "Matching" component.
   - Whenever another user has matched with an activity in the "Matching" microservice, the owner of that activity can "decideMatch": by accepting or declining it.

Activity

"Activity" is the third and last core domain of our Bounded Context Diagram, since the whole scenario depends on the users being able to create activities. The "Activity" microservice stores information about different types of activities and communicates with the API's of "User" and "Matching" microservices in such that the distinct activities will have the right users.

As it may be seen in Figure 1, "Activity" is an abstract class that defines the two types of possible events: Training and Competition. A Training event would be just a basic activity needing: *an activity ID, the ID of the owner of the activity, the time interval, all the available positions and the certificates needed for those positions.* For the creation of a competition the *gender* and *organisation* need to be specified.

The following paragraph will clarify the interactions and logical flow of this microservice with the others:

1. Every user can be the owner of an activity by creating one, for which it specifies a time-slot and the entry requirements.

2. When a user is looking for available activities, the "Matching" microservice makes a call to the "Activity"'s API to provide it with the available activities for which that User can enroll. This exchange of data is done through *getAvailableActivities* and *sendAvailableActivities.*

3. If a User has chosen to enroll in a certain event, the "Matching" component will announce the "Activity" microservice through *takeAvailableSpot*, so that the *availablePositions* will be changed.

Diagrams

On the following pages we have displayed all the different diagrams that were used by us in order to make the architecture for the assignment.

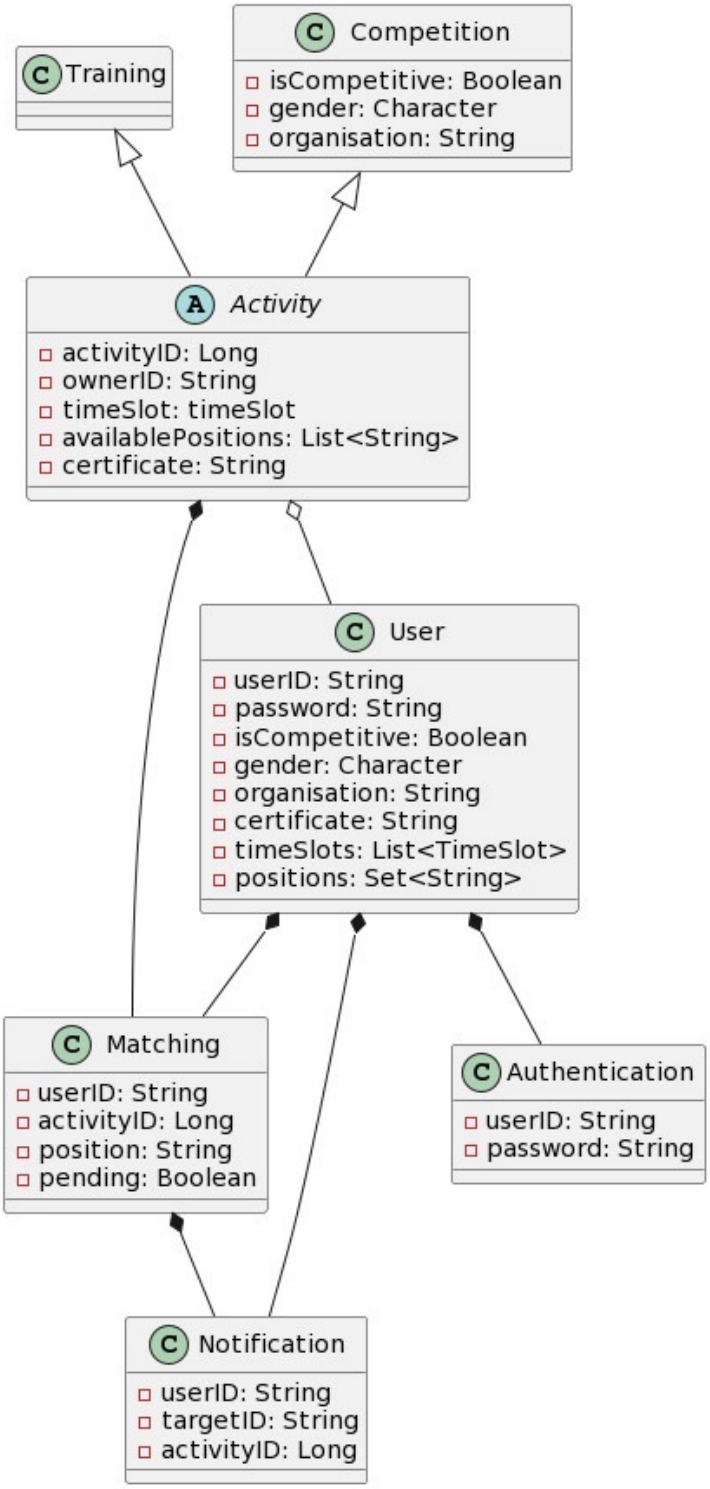

Figure 1: UML diagram

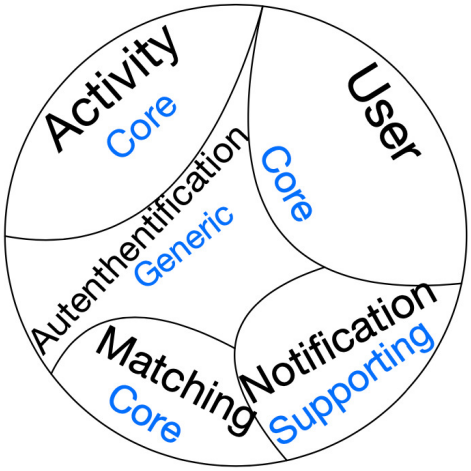

Figure 2: The Bounded Contexts diagram

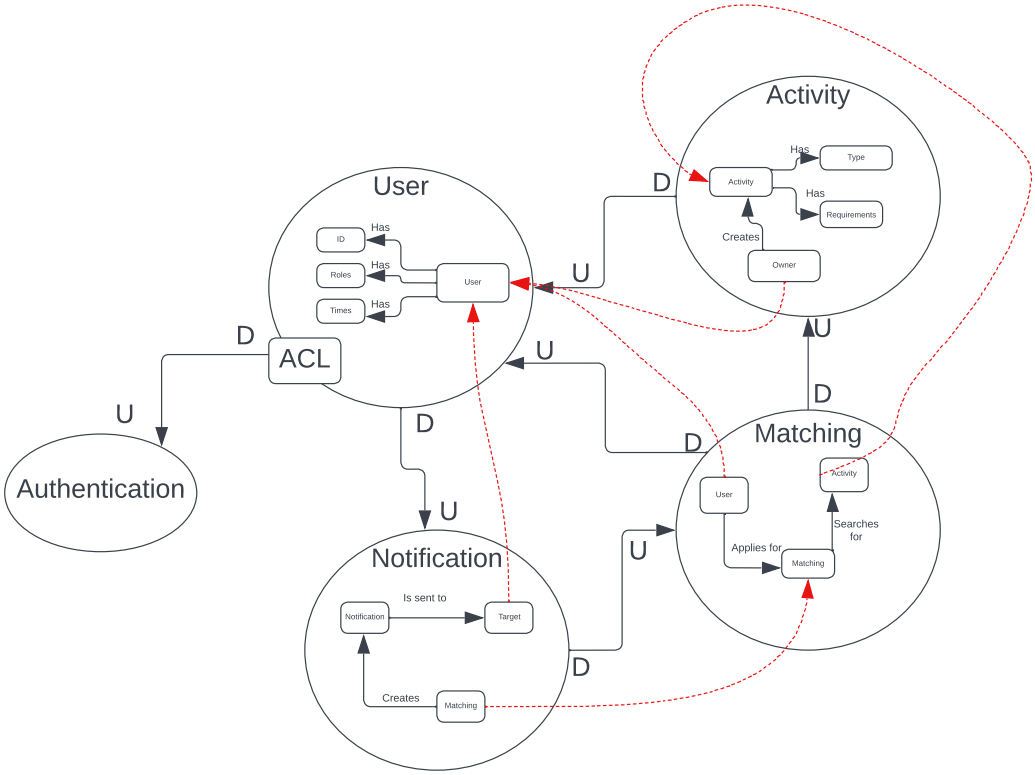

Figure 3: The Context Map

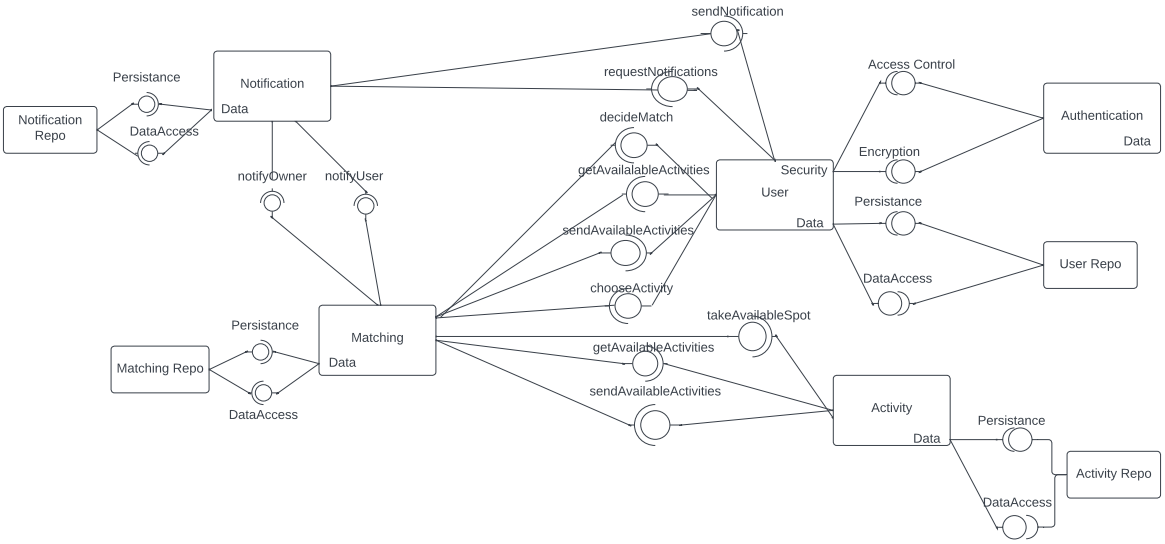

Figure 4: Component Diagram (which uses the Lollipop notation)