# SEM Assignment 1 - Task 1 - group 33A

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

December 16, 2022

## Bounded context split explanation

We have used Domain-Driven Design (DDD) on the scenario we were assigned (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 to make the coupling of our system as loose as possible, while also keeping 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 establishes 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 1: The Bounded Contexts diagram

From the low-level structure perspective, we have decided to implement each subsystem (microservice) using ***Hexagonal Architecture***, which allows for achieving this loosely coupled system. To follow this ports and adapters pattern, we made use of internal interfaces that establish the communication between our microservices. Moreover, each subsystem contains a repository that directly communicates with the database and all the other classes encapsulated within the domain layer.

## 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 follows:

1. The "Matching" component receives a request from a user, which has just specified their 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. 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 3. 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 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 the "User" component, as well as provide a way of API calls authentication between the microservices, which opens the door for performing authorization.

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 inputting their credentials. These consist of an ID - userID attribute - stored as a String, and a password - password attribute - stored as an encrypted String. The values of these attributes are forwarded later on to the "User" subsystem. Subsequently, the credentials are stored in the corresponding database. The password encryption and data flow safety are achieved with the 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 authorization success. Otherwise, the "User" is notified of the failure. Both mentioned possibilities are achieved through API calls.

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

Authentication:

- Encryption
- Identity verification

Authorization:

- Access grant by means of a JWT token

### 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 the "Matching" subsystem, thus providing nice isolation for the microservice pattern.

The "Notification" microservice notifies either the owner of the activity that a new application for the slot has been created or the applicant that they have received the slot. The basic flow of a notification is comprised of:

1. "Matching" needs to notify the user about the developments related to the requests to join activities. This is done either by email or by saving notifications to the database so the user could later request them.
   - The "Notification" microservice persists a database entry containing some information about the user that applied, the target user that should receive it, and the type of notification: request permission (notifyOwner) or confirmation (notifyUser). Persistence 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.
   - If email is chosen rather than persistence, then notifications are not saved and are sent to the target, provided the target has a valid email address.
2. When the user is online and queries the "Notification" service endpoint for messages by providing their ID, the service returns notifications parsed to basic text that have been accumulated for that specific user.

### 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 stores all the necessary information about the user and communicates this through the APIs of the other microservices. The role of the microservice itself and its logical flow is:

1. 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. If the authentication succeeds, the "User" microservice receives its token, which allows him to take further actions.
2. The "User" microservice can request to receive all the notifications that are related to him from the "Notification" subsystem.

3. The ”User” microservice can also make a call to the API of the ”Activity” microservice to create a new activity.

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 possible match the ”User” subsystem communicates with the ”Matching” microservice to start the matching procedure based on his/her information.
   - The API of ”Matching” would then send the available activities the user matches with. Afterward, he/she can then decide which activities to take part in. This information is then sent back to the ”Matching” component.
   - Whenever another user has matched with an activity in the ”Matching” microservice, the owner of that activity can decide whether to accept or decline 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 APIs of ”User” and ”Matching” microservices such that the distinct activities will have the right users.

”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 competition creation, the *gender* and *organization* 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 checking for available activities, the ”Matching” subsystem makes a call to the ”Activity”’s API to provide it with the available activities for which that user can enroll, based on the time interval correlation.

3. If a User has chosen to enroll in a certain event, the ”Matching” component will announce the ”Activity” microservice. Furthermore, ”Activity” obtains that User’s data to verify if that person is eligible for that event. If so, the previously available position will be removed from the according activity.

4. If a User has chosen to unroll from an event, the ”Activity” microservice will add back the reserved position to the corresponding activity.

## Diagrams

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

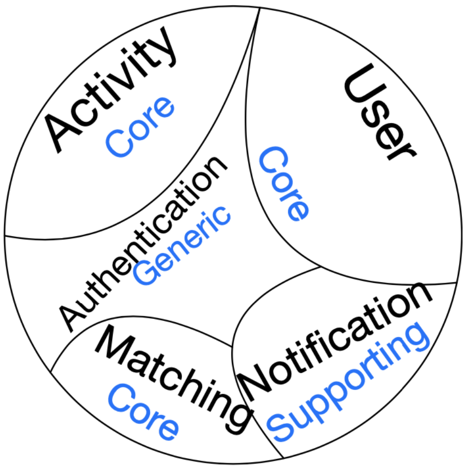

Figure 1: The Bounded Contexts diagram

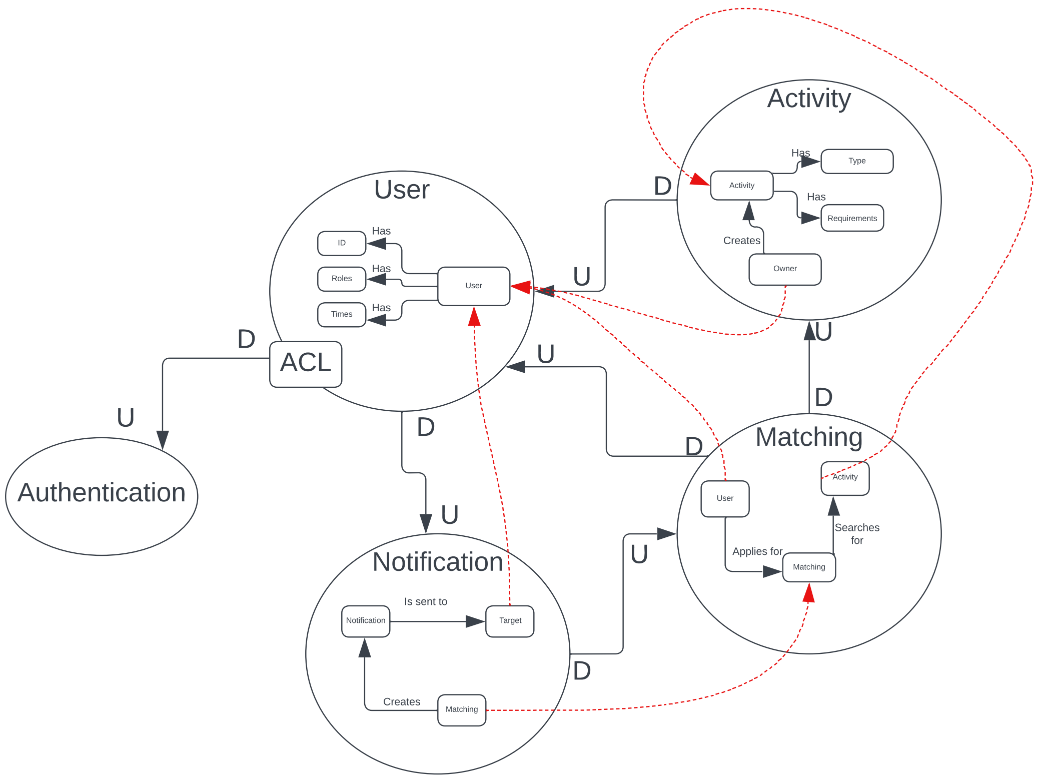

Figure 2: The Context Map

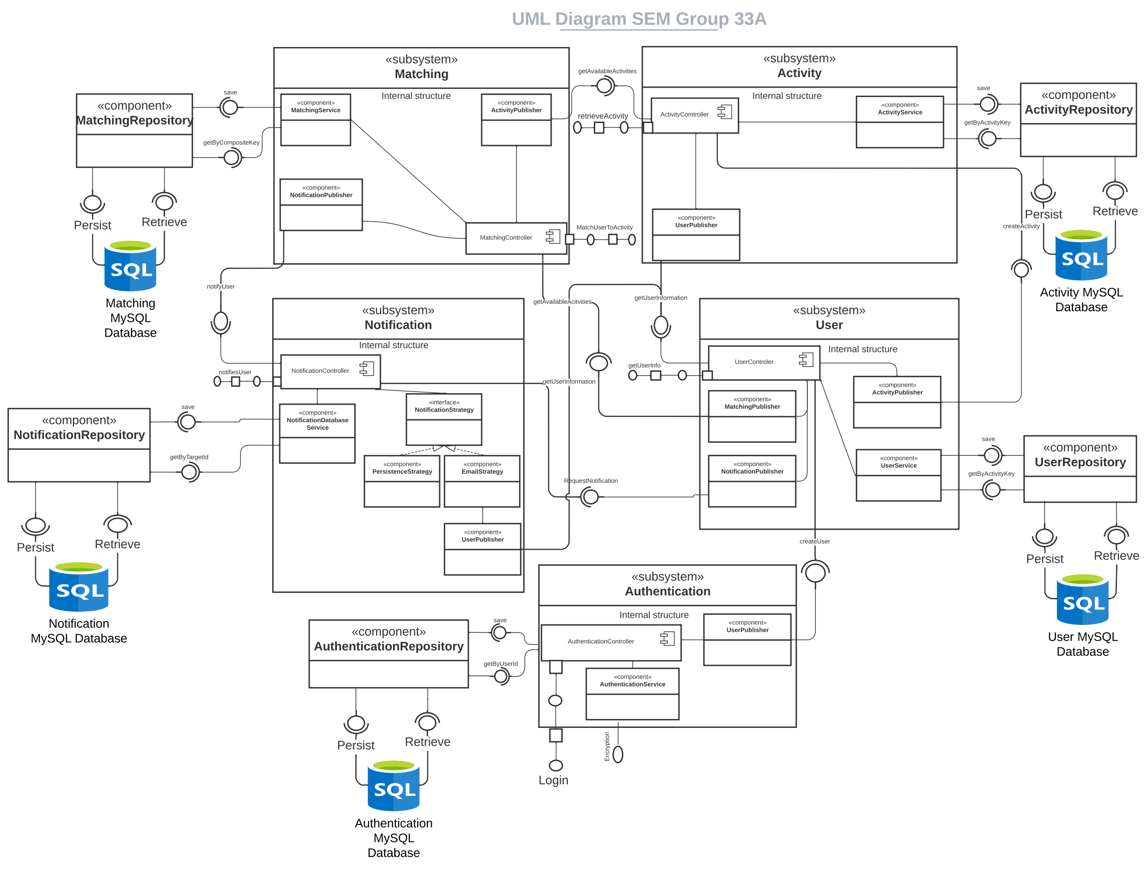

Figure 3: Component diagram in UML notation