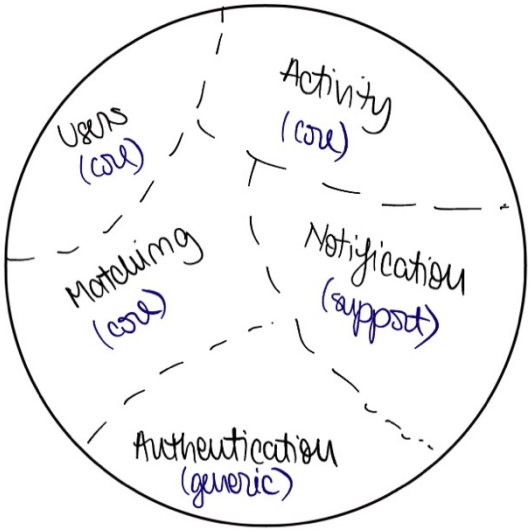
Software Architecture

By conducting *Domain-Driven Design (DDD)* on the given ***'Rowing' scenario*** we have identified the following keywords: **user, match, activity, training, competition, notification, member, account, ID, authentication, password, positions, availability** which helped us concretely divide the bounding contexts. Thus, we started by comparing these terms and trying to find similarities and assess the coupling between the domains given. Consequently, we came up with the below shown bounding contexts:



We mapped the **user, ID, member** to the **Users core** domain as it is vital to keep records of the users that are registered. Next, the keywords: **activity, training, competition, positions** were encapsulated in the **Activity core** domain so that we could be able to express the main functionality of the application. The **Matching core** domain is derived from the **match and availability** keywords that, even though they could be part of the Activity and Users domain, they also have a different functionality as they represent the connection between the aforementioned domains. Since the whole purpose of the application is to match the user with activities, the matching microservice represents the core functionality of the application. The **Authentication** domain is a **generic** one, mapped by the **account, authentication and password** keywords which do not encapsulate the core functionality but sustain them. Lastly, the remaining **notification** keyword was mapped to the **Notification support** domain as it supports the action of sending the emails when matching a user to an activity, but it is not essential.

Note that the communication between microservices will be done **synchronously**, as explained in the lecture, however we have included as a won't have feature a shift to using asynchronous communication with Kafka.

Users microservice

As the system is concerned with matching people to activities with respect to the users information, it's critical for the system to be able to discern and remember different users. Representing a core domain, the ***'Users' microservice*** is responsible for persisting users' unique ID, gender, competitiveness, certificates and organization in a user database. These are the attributes of a User entity. The microservice will be able to handle missing data for all attributes excluding ID. There will only be one certificate entry stored per user as certificates supersede each other.

After the client authenticates themselves, they will be able to include their ID in a request to the ***'Users' microservice*** to create their own account in the system, where their data will be persisted. The microservice

then offers an interface allowing authenticated clients to enter and update their attributes (excluding the ID) if they, for example, gain a new certificate or change organization. The microservice is also responsible for filtering these organization names, to allow for optimal matching. Internally in the micro service, we utilise **The ports and adapters architecture** to increase testability and flexibility, making the service open for extension as specified by the requirements. This also allows us to focus on the user business logic disconnected from the implementation details. The microservice also needs to be able to provide all the persisted data for a user, identified by their ID, to the ***'Matching' microservice*** when requested.

We believe this is enough functionality to justify decoupling 'Users' as its own lightweight microservice. An alternative would be only maintaining a user database and distributing the functionality across other microservices, but this would result in less modularity and more difficulty extending the system with user-related functionalities. The microservice is not too coupled as it is only relied upon by the ***'Matching' microservice***. There is also a high degree of cohesion within the microservice as all operations involve the fundamental User entities, meaning no further split of responsibilities is required.

Activity microservice

The main goal of this service is to allow the user to create activities which, later on, will be matched with the 'User' entities. The core ***'Activity' microservice*** will be the sole provider of the functionality for interacting with the activity database. The 'Activity' entities will have such attributes as the owner (the 'User' who published it), it's timeslot, the certificate, type, and the positions that need to be filled.

Originally, the service is only connected to a single other microservice, namely, the ***'Matching' service,*** however, it will implement the communication with the client and the ***'Activity Repository'*** as well. The main interfaces it contains are for the client manipulating the activities, ***'Matching' service*** querying the activities for matches and the 'Activity Repository' for persisting an activity. As for the interfaces it requires, there is the extraction of activities from the repository, the ability to update the activities in matchings, and finally an interface from the client to update the information. With the presence of these interfaces that communicate with other services, the repository connected to the database and the other classes encapsulated within the domain layer, **the Hexagonal Architecture** is the first option as a low-level structure of this subsystem.

The key reason that we chose to implement it this way was to keep the microservices lightweight. Instead of handling the activity entities' manipulation in the same service as the matchings, we opted for decoupling them to separate the concerns of manipulating the activities and using them to pair with the users. Regarding the option of dividing it into smaller parts, we decided that it is as atomic as it could be and doing so would introduce more communication overhead.

Matching microservice

Being responsible for matching the availabilities specified by users with already existing activities in the repository, the ***'Matching' microservice*** is meant to emulate the interaction between the other Users and Activity services. The initial idea that the service evolves from is, in fact, the basic interaction of the application: "A user should be able to mention its availability and then should be matched to activities (trainings or competitions which require extra people". Nonetheless, the next question would be using an API Gateway rather than the ***'Matching' microservice*** we aim to implement. However, this service comes in handy when, based on availability, a user can be matched to an activity that is overlapping with an already matched one. Thus, we require a database to modify, save and filter the previously done matches in order to manipulate the overall ***'Matching' process.*** Another possibility would have been implementing this

functionality either in ***'User'*** or ***'Activity' microservices***. However, doing so would cause a decrease in cohesion, maintainability and modularity of the overall software architecture.
For the low-level architecture we opted for **ports and adapters** since it complies with the requirements in the optimal way. Thus, in the application layer we included three components: Activity/Users/Notification Communication which handle the communication of the matching microservice to the others and the Matching Controller for the client requests.

The overall flow of the following interactions: (see also Context Map – bottom of the document)

1. ***Client publishes availability***

The client submits their availability timeslot and positions they would like to fill through an API request to the ***'Matching' microservice.*** This in turn, retrieves matched activities by communicating with the ***'Activity' microservice*** services, which are then returned to the client as an API response.

2. ***Client chooses matched activity***

From the list that was returned in the last step, the client creates a new API request specifying the chosen activity. The ***'Matching' system*** is then responsible for preparing a notification for the owner of the chosen activity.

3. ***Owner of activity accepts or denies a participant***

The client in the role of the owner can make a request to this microservice in order to retrieve the pending requests from the ***'MatchingRepository'***. From the given list, the owner can send back a request API for each pending participant containing the acceptance or denial. Using this decision, the system can persist the match between the accepted participant and the chosen activity, and use the other services to notify back the user with the owner's decision. Otherwise, it can remove the pending request, and send a notification as before.

Through all of these interactions, it is clear that the ***'Matching' service*** is the main component handling the communication between other services and sits at the centre of the software architecture created to comply with the ***'Rowing' scenario.***

Authentication microservice

Security is a vital part of any modern piece of software. In our application the area of security has mainly to do with the requests coming in from the users. It is important for the correct functioning of the entire system that **1.** the users are who they say they are and **2.** that they have the required permissions for the actions they request the system to perform.

The ***'Authentication' microservice*** will handle the verification of the user's credentials and their permissions. Whenever a user wants the system to process a request, they first need to login. This is done by sending a request to the authentication microservice containing their user ID and password. The ***'Authentication' service*** will then process this request, validate the users credentials and, if the user is verified, return a token. With this token the user can make requests to the other microservices without having to provide their credentials each time. The other microservices only need to validate the token, which is easier to do and more secure, because the users don't have to send over their password each time. Furthermore, the token has a limited lifetime so that it can only be used for one session. Note that the validation and registration are the main part of the whole ***'Authentication' microservice*** and will be implemented using **The Chain of Responsibility** design pattern, making the whole process and interactions explained above a great fit for the **Hexagonal low-level architecture**.

The advantage of having a separate ***'Authentication' microservice*** is that all the other services don't have to implement their own user verification, thus greatly reducing the vulnerability of the system. The authentication service is thus not part of the core functionality of the application, but a generic service that is still vital to the functioning of the system.

This service will also handle the creating of new user accounts. The password will not be stored in the database as plain text but will be hashed to an encrypted string. This to ensure that in the event of a database breach the users' passwords will not be leaked.

Notification microservice

The feature within our project which allows direct communication between the application and its registered users is the ***'Notification' system***, which is capable of delivering emails regarding announcements about rowing matches – a ***'Notification' microservice*** is therefore an appropriate design choice for this functionality. The lower-level architecture of this microservice has a **layered structure**: the 'NotificationController' contains the methods which handle the HTTP requests directly, the *'Director'* decides which type of 'Notification' to create and send, and the *'NotificationBuilder'* separates all business logic from the rest of the microservice. This LLD decision has the following advantages:

1. ***Flexibility and maintainability*** - any layer can be changed without affecting other layers; in our case the format of notification messages can change quite often without damaging the email sending procedure;
2. ***Testability*** - we can test each layer in isolation (using mocks and stubs).

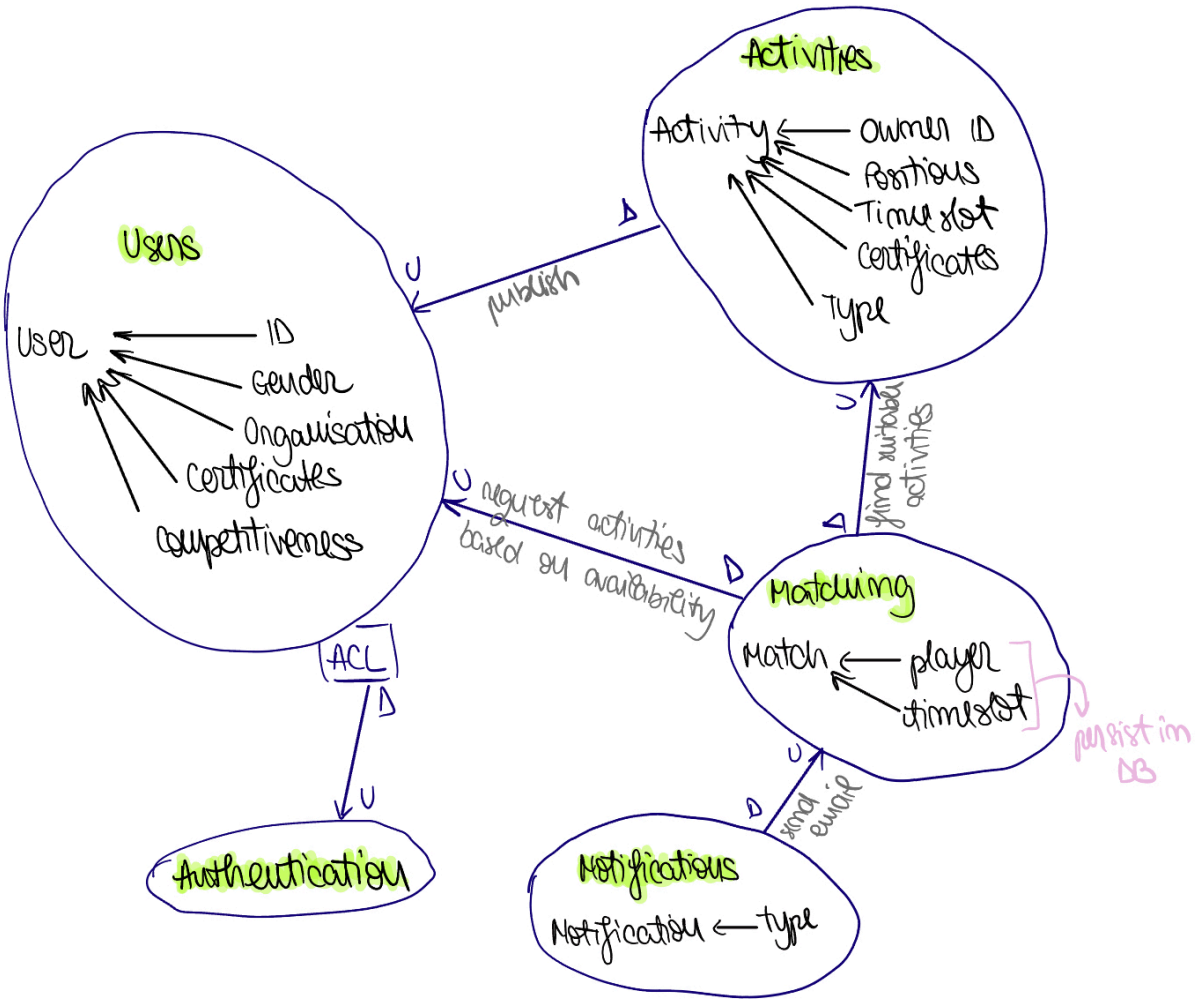
A disadvantage would be increased complexity because of the additional abstraction, but this is needed for a microservice which provides diverse email communication with the users (communication explained in more detail below).

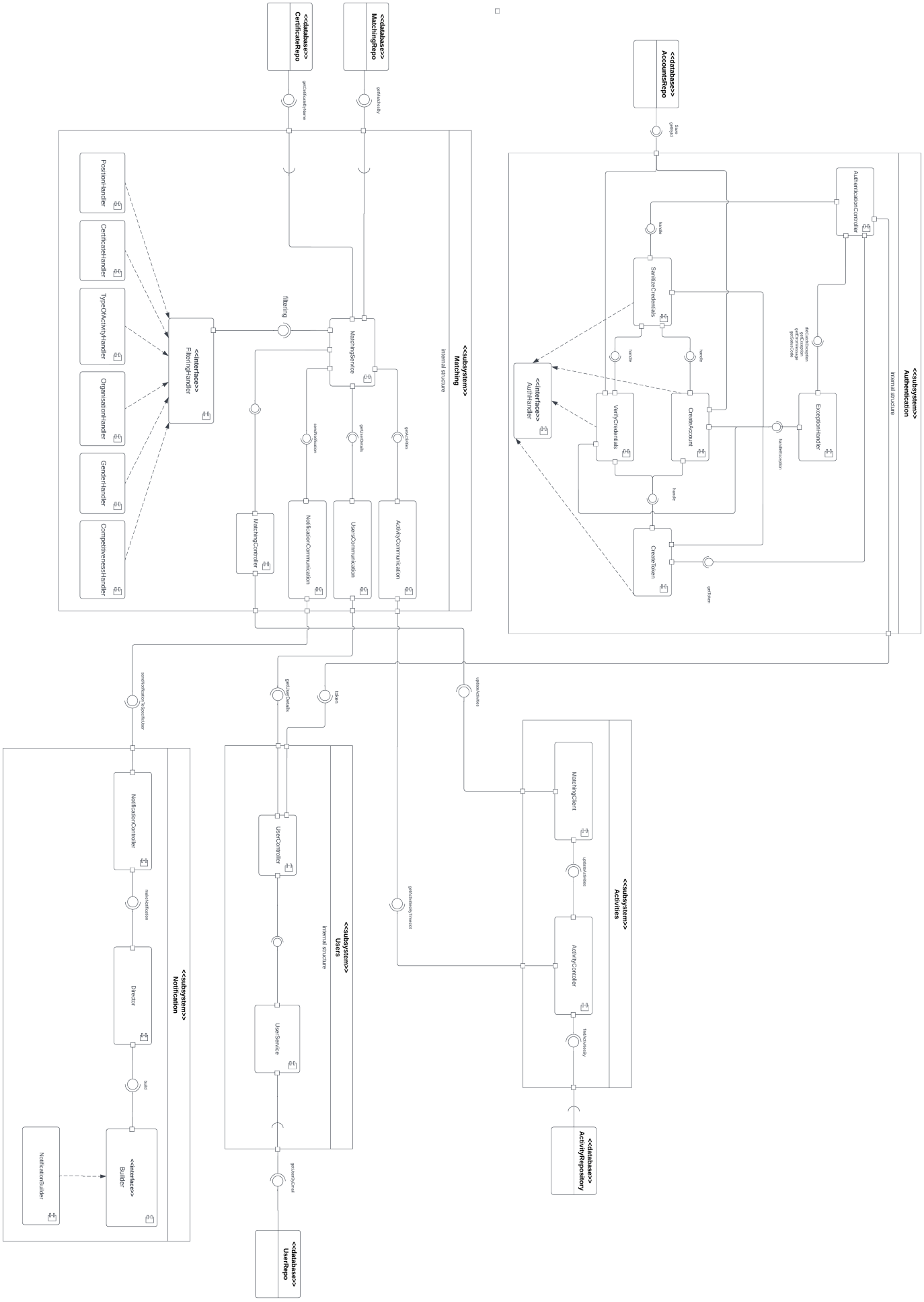
There are three main issues covered by the notification service, all being in a close relationship with the matching service, since direct messages are needed only in case of additions or changes committed to the matching history. These changes are based on decisions made by rowers and by the publishers of a certain training or competition. More precisely:

1. The system shall **notify the publishers by email** about users that want to participate in their training session. After a user chooses an activity, and the request is saved in the matching database with a status of 'pending', the notification service sends an email to the publisher of that activity. This represents a suggestion for him to give permission to the player or deny his participation as soon as possible.
2. The system shall **notify the users by email** whether the publisher of the training they selected accepted their request or not. When the status of the participation request changes to 'accepted' or 'denied' based on the answer of the publisher, the user which submitted the request is announced of this decision by email and the flow of the application continues with other requests.
3. The system shall **notify participants in case of modifications / cancellations made to already accepted activities**. When a publisher updates an activity, the player who already had a match with the activity is notified of its deletion / modification and is instructed to start over the matching process.

In conclusion, the ***'Notification' microservice*** accounts for an important part of the rowing contest application flow and provides a pleasant user experience through email communication.

Context Map


Activities
Activity
owner ID
Position
Time slot
certificates
Type
Users
User
ID
Gender
Organisation
certificates
competitiveness
publish
find suitable activities
request activities based on availability
Matching
Match
player
timeslot
persist in DB
ACL
Authentication
Notifications
Notification
type
send email


<<subsystem>>
Authentication
internal structure
AuthenticationController
SanitizeCredentials
VerifyCredentials
CreateAccount
CreateToken
ExceptionHandler
<<interface>>
AuthHandler
<<database>>
AccountsRepo
<<subsystem>>
Matching
internal structure
MatchingService
FilteringHandler
MatchingController
NotificationCommunication
UserCommunication
ActivityCommunication
PositionHandler
CertificateHandler
TypeOfActivityHandler
OrganisationHandler
GenderHandler
CompetitivenessHandler
<<database>>
MatchingRepo
<<database>>
CertificateRepo
<<subsystem>>
Activities
MatchingClient
ActivityController
<<database>>
ActivityRepository
<<subsystem>>
Users
internal structure
UserController
UserService
<<database>>
UserRepo
<<subsystem>>
Notification
NotificationController
Director
<<interface>>
Builder
NotificationBuilder