Assignment 1

Software architecture

We are developing a system where users can enroll for rowing activities. We have identified five bounded contexts; each being mapped to one microservice for the reasons listed below.

First of all, we considered security to be a crucial part of our application, hence our choice to create a microservice responsible only for that, the Authentication microservice. As for the Participant and Activity microservice, the choice came naturally from the context of rowing training and competitions. Since each activity has a number of users who partake, we decided that a Matcher microservice which handles these relations between activities and participants is necessary. Last but not least, in order to reduce the amount of business logic encapsulated in the Participant and Activity microservices, our team found a Notification microservice particularly useful, leaving it in charge of passing information between the owner of an activity and the users they choose.

Participant

The participant class will have the following attributes: NetID, Positions, Certificate, Gender, Organization, Level.

A participant has a **NetID** for identification. A **Positions** attribute is needed in order to express what roles they are able to fill (cox, coach, port side rower, starboard side rower, sculling rower). The **Certificate** attribute contains the most powerful certificate that the user possesses (in case the participant can fill the cox position). The **Gender** and **Organization** attributes are filled only if the user desires to participate in any competitions that require this, while the **Level** attribute is requested only for certain competitions. Participant microservice is the principal component of the system that acts like a gateway for the user. Most requests are sent to this microservice because we wanted to simulate the participant's communication with their account. (Whenever a request is sent to the user account, the token is retrieved from the credentials and passed to the next request.) A participant can add their preferences for the match in their account that will be saved in the ParticipantRepository. When a user requests a match, their preferences are retrieved from the database. A RequestMatch object is created, encapsulating a Participant instance and a list of timeslots that is sent to the Matcher. The participant will receive as response a List<TransferMatch> that represents the list of activities that the participant can attend. The participant chooses only one of them and

sends it back to the Matcher. The TransferMatch(the accepted activity) is sent to Matcher using ServerUtils. In order to accept a part of these users, the participant will send a List<TransferMatch> to the Matcher. The participant is also able to change its positions that will be updated in the ParticipantRepository.

Activity

The activity class has the following attributes: ActivityID, Name, TimeSlot, BoatType, Positions, Owner. The activity class is an abstract class extended by the subclasses Training and Competition. Competition has Organization, Gender and Competitive as attributes.

An activity should have an **ActivityID** for identification. **Name** stores the name of the activity chosen by the user. The **TimeSlot** attribute is used for storing the time and date the activity starts and ends. The **BoatType** determines which type of boat is used for the activity, this can be a C4, 4+ or 8+ in our system and is needed to determine which users can take the role of cox in the activity. In the **Positions** attribute we store the positions that need to be filled. The **Owner** attribute is the user that created the activity.

The subclass **Training** does not have any additional attributes, it is the simplest type of activity with no extra rules. The subclass **Competition** has the attributes **Organization** and **Gender** because all members of a competition team need to be of the same gender and organization. The **Competitive** attribute is used to state whether the competition is open to all rowers or to competitive rowers exclusively.

A user can send a request to the activity microservice to create, edit or cancel an activity. They can cancel an activity by **ActivityID**, which deletes this one specific activity, or by their **NetID**, which deletes all of the user's activities. The Activity microservice reflects all of the changes in the database. The Activity microservice authenticates users who want to edit or cancel an activity, as only the owner is allowed to make any changes to an already existing activity.

The Matcher microservice can request to get all activities from the activity database to later match it with the users. It can also request to only get all competitions, or only all trainings.

Matcher

The matcher class will have the following attributes: ActivityID , NetID and Position.

The matcher microservice functionality links enrolled Participants to each Activity and stores that information in the database. A matcher should have the **ActivityID** of the corresponding activity, the **NetID** of the participant that requested to participate in that activity and also the **Position** that they can fill to process the matching for each participant.

When the matcher microservice receives a request match, it filters the list of activities according to the details of that participant and sends back those activities as a list of TransferMatch. TransferMatch is a transfer object that contains the name of the activity, the position they are able to fill, the timeslot and also the NetID of the owner of that activity. As an answer for the list sent, the matcher microservice receives one of the transfer matches and creates a Match that will be added to the MatcherRepository, sending also a TransferMatch to the notification microservice. After the owner gets the list of pending participants, the matcher microservice receives a list of TransferMatch which represent the list of the accepted participants for that activity. It removes the matches that correspond to the name of the owner from the transfer matches and send to the notification microservice a list of TransferMatch that corresponds to the accepted Participants.

Notification

The notification class will have the following attributes: ActivityID, NetID, Message.

A notification should have an **ActivityID** corresponding to an activity in the Activity database and a **NetID** indicating which participant the notification will be sent to, along with a **Message** containing relevant information about the activity such as starting time and location.

The notification microservice notifies the owner of an activity about the users who want to participate in their activity by sending a List<TransferMatch> to the Participant microservice. Moreover, when the owner accepts a number of users, each accepted user is sent a TransferMatch containing the details of their admission.

Authentication

The authentication microservice is built around **Spring Security,** the library within Spring which generates JWT tokens for each **Participant** that registers and authenticates. Then, this JWT token is validated within each microservice by the class **JwtRequestFilter,** action made possible by the **SecurityContextHolder**.

Dataflow

- Authentication

First, the user needs to register to the app with the help of the Authentication microservice. After they register, the Authentication microservice will generate a JWT token, which is used to grant access to the user if they introduce the correct login details.

- Creation of activity

Participants create an activity by sending it to the Activity microservice. They need to provide the values for the attributes *Name, TimeSlot, BoatType* and *Positions*. In the case of a competition, they also need to provide the *Gender, Organization* and whether the rowers must be *Competitive*. The participant becomes the activity owner and their *NetID* is linked to the activity. They can edit any field of the created activity by calling the Activity microservice again or they can cancel the entire activity.

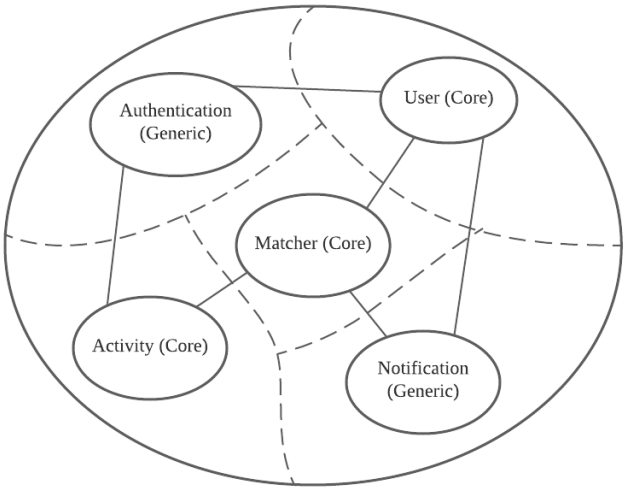
- Matching

When the user enrolls in activities, they fill up their available timeslots section and sends a request to the Matcher. The Matcher requests the available activities from the Activities microservice. After it receives them, the Matcher filters the ones that are available for the current user. Afterwards, the Matcher sends the filtered activities to the user. The user selects one of them and sends it back to the Matcher. The Matcher stores the User and the selected activity in its database, as a pending match.

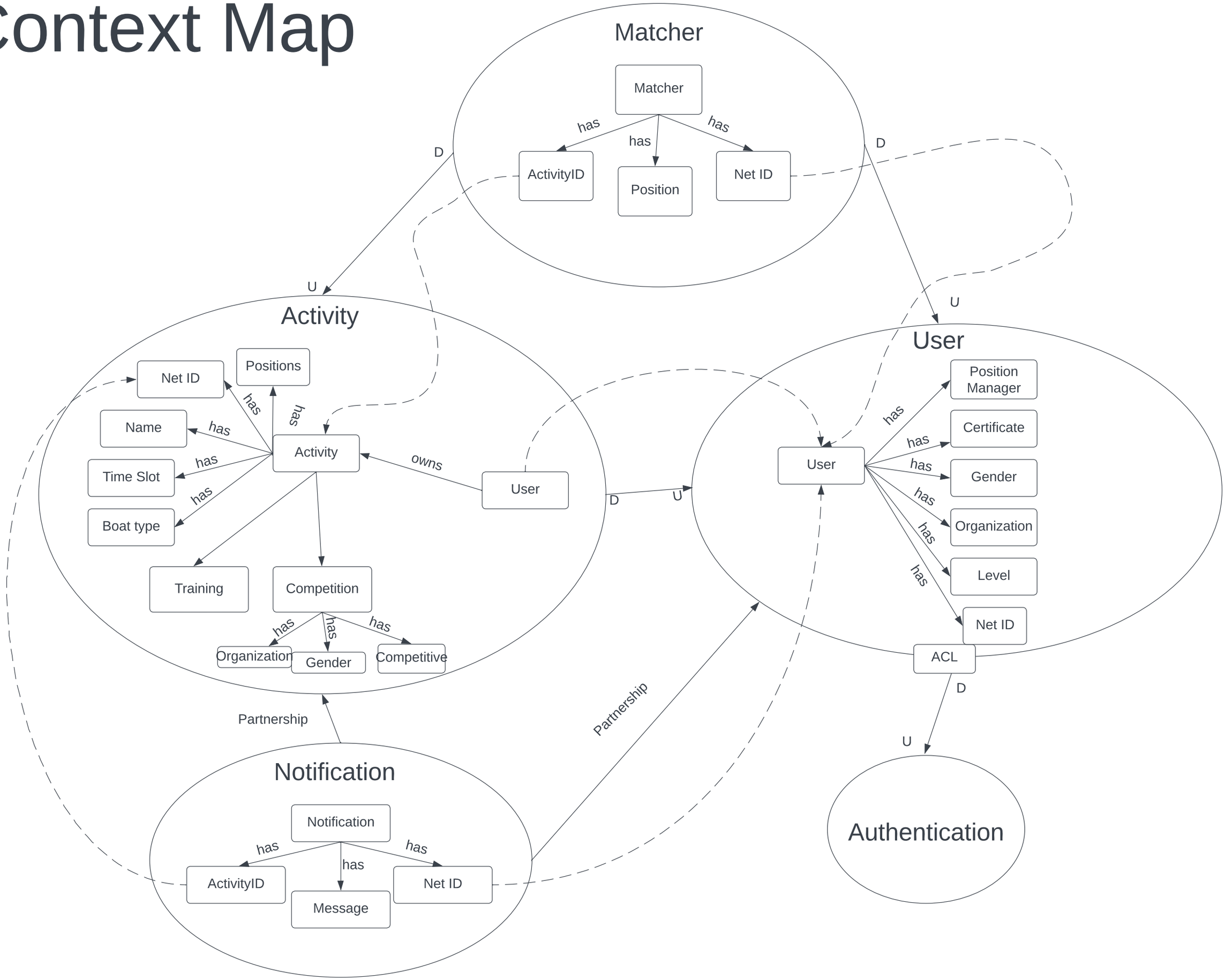
- Selection

Matcher sends the request of pending users to the notification microservice. The activity owner (user) gets a notification with the pending users for their activity. The activity owner selects users from the requests and sends them to the Matcher. Then, the Matcher deletes the pending matches from the MatcherRepository. Afterwards, accepted users are sent to the Notification microservice. Lastly, notification service sends a message to the accepted user to notify the selection.

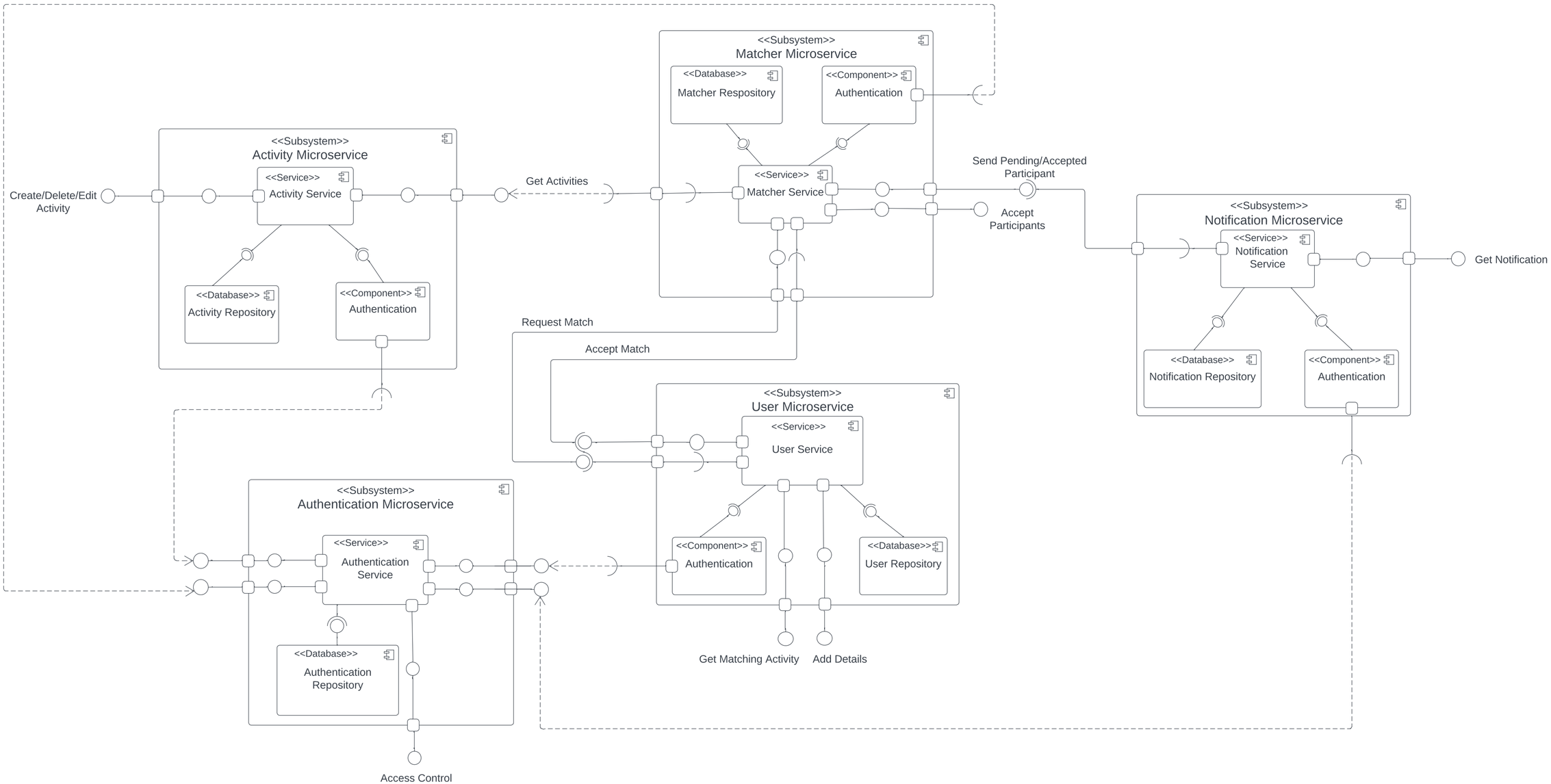
Bounded Context



Context Map


Matcher
Matcher
has
has
has
ActivityID
Position
Net ID
D
D
U
U
Activity
Net ID
Positions
has
has
Name
has
Activity
Time Slot
has
owns
User
has
Boat type
Training
Competition
has
has
has
Organization
Gender
Competitive
User
Position Manager
has
Certificate
has
User
has
Gender
has
Organization
has
Level
has
Net ID
ACL
D
U
Partnership
Partnership
Notification
Notification
has
has
has
ActivityID
Net ID
has
Message
Authentication

UML Diagram


<<Subsystem>>
Matcher Microservice
<<Database>>
Matcher Respository
<<Component>>
Authentication
<<Service>>
Matcher Service
<<Subsystem>>
Activity Microservice
<<Service>>
Activity Service
<<Database>>
Activity Repository
<<Component>>
Authentication
Create/Delete/Edit Activity
Get Activities
Send Pending/Accepted Participant
Accept Participants
<<Subsystem>>
Notification Microservice
<<Service>>
Notification Service
<<Database>>
Notification Repository
<<Component>>
Authentication
Get Notification
Request Match
Accept Match
<<Subsystem>>
User Microservice
<<Service>>
User Service
<<Component>>
Authentication
<<Database>>
User Repository
Get Matching Activity
Add Details
<<Subsystem>>
Authentication Microservice
<<Service>>
Authentication Service
<<Database>>
Authentication Repository
Access Control

Task 2: Design Patterns

Chain of Responsibility

1.1

The first design pattern we have chosen is “Chain of Responsibility” in the Matcher microservice because the activities have to go through 4 types of filtering before reaching the user.

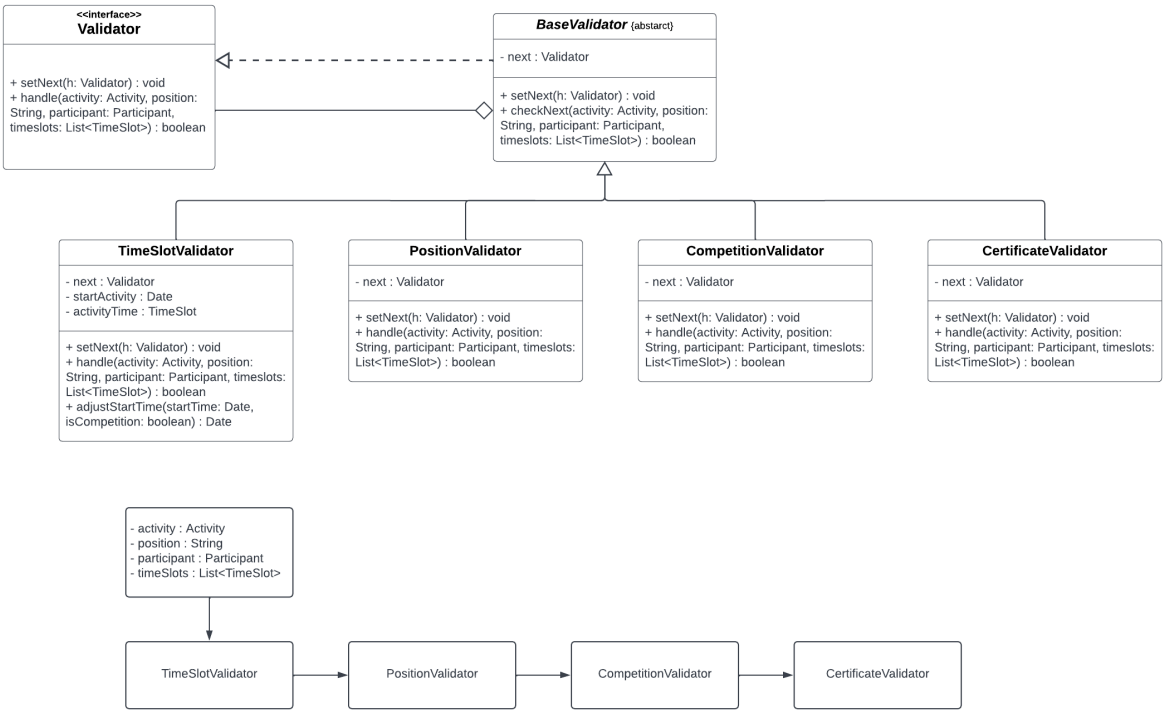
Firstly, we filter by TimeSlot with the TimeSlotValidator. The startDate of the activity is changed in order to meet the scenario requirements(-0.5 h for Training and –1 day for Competition) because the Participant needs time to reach the Activity location and staff has to be notified. If the startTime of the Participant is before or equal to the startTime of the activity and the endTime of the Participant is after or equal to the endTime of the activity the next validator will be called. Otherwise, false will be returned.

Secondly, we filter by position with the PositionValidator. If the Participant is able to fulfill the position passed to the method, the next validator will be checked. Otherwise, the chain stops.

Next, we filter by the Competitions's details with the CompetitionValidator. In case the Activity is a Training the next validator will be called. If not, Participant's gender, organization and competitiveness will be checked against Competition's specifications. In the circumstance that all of them match, the next validator will be analysed. Otherwise, false will be returned.

Finally, we filter by certificate with the CertificateValidator. If the current position is cox, this Validator will verify if the Participant's certificate is better than the boat used in the activity. If this is the case or the current position is not cox, then true will be returned because this is the last validator in the chain, else false will be returned.

1.2


<<interface>>
Validator
+ setNext(h: Validator) : void
+ handle(activity: Activity, position: String, participant: Participant, timeslots: List<TimeSlot>) : boolean
BaseValidator {abstarct}
- next : Validator
+ setNext(h: Validator) : void
+ checkNext(activity: Activity, position: String, participant: Participant, timeslots: List<TimeSlot>) : boolean
TimeSlotValidator
- next : Validator
- startActivity : Date
- activityTime : TimeSlot
+ setNext(h: Validator) : void
+ handle(activity: Activity, position: String, participant: Participant, timeslots: List<TimeSlot>) : boolean
+ adjustStartTime(startTime: Date, isCompetition: boolean) : Date
PositionValidator
- next : Validator
+ setNext(h: Validator) : void
+ handle(activity: Activity, position: String, participant: Participant, timeslots: List<TimeSlot>) : boolean
CompetitionValidator
- next : Validator
+ setNext(h: Validator) : void
+ handle(activity: Activity, position: String, participant: Participant, timeslots: List<TimeSlot>) : boolean
CertificateValidator
- next : Validator
+ setNext(h: Validator) : void
+ handle(activity: Activity, position: String, participant: Participant, timeslots: List<TimeSlot>) : boolean

- activity : Activity
- position : String
- participant : Participant
- timeSlots : List<TimeSlot>
TimeSlotValidator
PositionValidator
CompetitionValidator
CertificateValidator

Factory Method

2.1

The second design pattern we have chosen is “Factory Method”, its usage being evident in the Notification microservice, more specifically in NotificationController. This choice came to us naturally due to the implementation of TransferMatch, our DTO which is used to communicate between microservices.

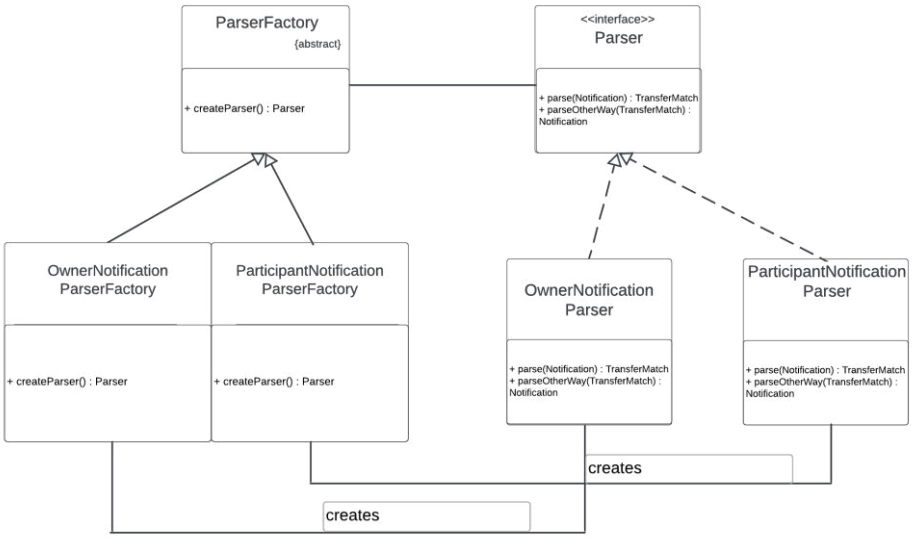
Firstly, when an owner accepts a list of users to partake in their activity, the matcher microservice sends a list of TransferMatch to the Notification microservice, which need to be parsed (since all attributes of TransferMatch are Strings), stored in the database as a Notification and sent to the accepted users. This action is being done by the “parseOtherWay” method in the ParticipantNotificationParser.

Secondly, when a user confirms that they would like to partake in an activity, the matcher microservice sends a TransferMatch to the Notification microservice, which will notify the owner of said activity about the participant. The transformation to a Notification object is done by the “parseOtherWay” method in the OwnerNotificationParser.

When an owner needs to be notified about the users that agreed to participate in their activity, the Notification microservice retrieves these Notifications from the database and they are converted to TransferMatches by the “parse” method in the OwnerNotificationParser.

Finally, when a user needs to be notified about the activities he has been accepted into, the Notification microservice retrieves these Notifications from the database and turns them into TransferMatches using the “parse” method in the ParticipantNotificationParser, DTOs which are being sent to the Participant microservice.

2.2


ParserFactory
{abstract}
+ createParser() : Parser
<<interface>>
Parser
+ parse(Notification) : TransferMatch
+ parseOtherWay(TransferMatch) : Notification
OwnerNotification ParserFactory
+ createParser() : Parser
ParticipantNotification ParserFactory
+ createParser() : Parser
OwnerNotification Parser
+ parse(Notification) : TransferMatch
+ parseOtherWay(TransferMatch) : Notification
ParticipantNotification Parser
+ parse(Notification) : TransferMatch
+ parseOtherWay(TransferMatch) : Notification
creates
creates