Delft University of Technology

Software Engineering Methods: Assignment 1

CSE2115 Group 31b

Alexandra Darie	5511100
Elena Dumitrescu	5527236
Robert Vadastreanu	5508096
Sander Vermeulen	5482127
Xingyu Han	5343755
Zoya van Meel	5114470

December 16, 2022

1 Introduction

Our task is to implement a scheduling system for the rowing associations in Delft, in order to facilitate the process of finding training sessions and competitions. In order to design our architecture, we have used a Domain-Driven Design approach, following the steps presented in the Software Engineering Methods lectures.

The starting point was gathering the requirements from the given scenario. Afterwards, we constructed a Context Map and a Component Diagram to showcase the architecture. In the following sections, we explain in more detail the process of designing these models.

2 Context Map

The first step in designing our architecture was identifying the bounded contexts from the domain specifications. Our current model contains four domains in total. The User, the Event and the Activity Scheduler are our three core domains, while the Authentication is a generic domain. Based on these, we have constructed the Context Map which can be seen in Figure 1.

2.1 Users

The first domain we identified was the User domain, which handles all data and functionality related to the users of our application. As mentioned in the Rowing Scenario, *the users have certain attributes that will influence whether or not they will be accepted in the events.* Thus, we needed a domain that ensures the communication between the actual person that interacts with the app and the other domains, handling how the user input is interpreted and storing the permanent attributes for future interactions.

Since gender, organization, certificates, availability, and positions are necessary for each request, we decided to keep the permanent parts such as gender, organization and certificates in the database, the others being filled by the application user each time they want to find an event. The events that are suitable for their request are then presented to them such that they can afterward pick their final choice. Then, moving over to the user's relationship with events, we know that "*the user is able to create events*", so we decided to enable this domain to send event models directly to the event domain.

2.2 Authentication

Since the users have accounts within the application, they need to be able to authenticate using their unique IDs and passwords as requested in the scenario: "*All users of the system need to authenticate themselves to determine who they are and what they can do on the platform*". Therefore, we have identified Authentication as our second domain, which uses the Spring Security framework to fulfill its responsibilities as stated in the scenario. Due to the fact that logging and signing in will take place within the aforementioned part, this domain will also store the pairs of passwords and their corresponding IDs.

2.3 Events

The third domain we identified was the Event domain. We have chosen to integrate this domain since the Rowing Scenario highlights the different types of events (activities) a user can participate in, which all have different restrictions: "*However, not everybody can join every training although they might be available, the restrictions can be found in Domain specifications*", "*Important is that this system should also be functioning for competitions*". This domain then handles the storing of events, which can be either training sessions or competitions. These types of events will have different entry requirements respectively, which will be stored inside the database of this service. Alongside storing the type and requirements of an event, the date and location will also be stored, so that the Scheduler service can

use this information to correctly match a user with an event. Ultimately, the owner of the event is stored, which is used to make sure that no unauthorized users are able to alter the event's details in some way, which is also stated in the Rowing Scenario: "*A user should not be able to edit or cancel someone else's activity.*".

All of these details are specified when creating the event by the owner. After the event has already been created, the owner is still allowed to change some of the details, or cancel the event entirely. This is also according to the Rowing Scenario, wherein it states: "*If I manage to find someone outside of the application I also want to be able to cancel or edit the activity on the platform*" Finally, the stored information is used by the Scheduler when matching users to events.

2.4 Activity Scheduler

The fourth domain we found was the Activity Scheduler. Since we did not want to overload the current domains with functionality, we found it appropriate to integrate another domain, whose main purpose is to mediate the communication between the User and the Event. Our decision is also motivated by the Rowing Scenario, in which it is stated that the basic flow of the application is to match users with appropriate events: "*Application that will match available people to trainings and competitions that still require extra people*", functionality that does not necessarily belong to any of the before defined domains. The Activity Scheduler will store these matches in order to know in which events the users are enrolled.

Since the Activity Scheduler maintains most of the communication between the other domains, we have decided to assign one more purpose to it, which is handling notifications. The need for notification functionality was again highlighted in the Rowing Scenario: "*In this case the activity owner can accept or decline their request to join the training. After accepting the trainee should be notified that they have been accepted for the training.*". The requests are, therefore, sent between the participating users and the event owners. After a user submits their request to join an event, a notification for the owner will be created, which will ask for their decision regarding the user's request. If the owner approves, the scheduler will create another notification for that user, confirming that they can join the event. These two types of notifications will again be stored by the Activity Scheduler since we want the users to be able to access their notifications at all times.

3 Component Diagram

To continue our design process, we have constructed a UML Component Diagram based on the previous Context Map, in which we mapped the bounded contexts into microservices and modeled in a more extensive way the relationship between them. The UML Diagram is shown in Figure 2.

All microservices follow the **Ports-and-Adapters Architecture**, in order to improve maintainability and testability. If there is a microservice that has to be exchanged or becomes obsolete, it can easily be disconnected from the other microservices and replaced. For example, if the authentication is going to be handled differently, we only have to change the ports of the connected microservices with the authentication. The domain layer of each microservice contains repositories for abstracting the database access, as well as Data Transfer Objects for facilitating the data transmission between the microservices. The database contents are defined via the entities we use for our core domain (Users, Accounts, Events, Matches, Notifications). In addition, the microservices make use of service interfaces along with their implementations (the controllers only use interfaces to allow for loose coupling). All communication between microservices is synchronous.

3.1 User

The User microservice interacts not only with the other microservices, but also with the users themselves. More extensively, this microservice serves as a gateway for the users to interact with the system and is the only microservice that directly communicates with the user (in a real application, it would

connect with the UI). The communication includes personal information, but it can also be information regarding events the users have created or applied to. Therefore, the User microservice's main **role** is to communicate what a user wants from the application to the rest of the microservices. The other microservices handle these requests and send the proper information or data back to the User microservice. The User microservice is **responsible** for relaying what the other microservices have done back to the user.

Personal information, like a user's gender, organization, and certificate are stored in the User database. The communication with the User database is handled by the User Repository infrastructure. The User microservice interacts with the Event microservice when a user wants to create or edit an event. This interaction is handled by the User Controller. The User Controller takes a user's input and relays it to the Event microservice in the proper format.

The interaction with Scheduler microservice is done by the User Service. This service makes it so that the User Controller only has to relay User choices to the User Service and get Notifications from the User Service. The User Service makes it so that the functionality of the User microservice is further decoupled. This decoupling leads to code that is easier to maintain and adapt over time The User Service can post choices of the user to the Scheduler microservice. These choices can come from an event owner (someone who created an event) or from an applicant to an event. The other function of the User Service is handling the notifications that come from the Scheduler microservice. These notifications can be for both normal users and event owners.

The User service makes it so that the User Controller does not have to know what type of notification has to be outputted to the user, just that there are notifications. This is because an event owner can also be an applicant themselves. They might receive notifications about people that applied to their event, but also whether they themselves were accepted to an event they applied to.

3.2 Authentication

The Authentication microservice is **responsible** for handling user verification and registration. As shown in Figure 2, the microservice consists of two components, namely the Auth Manager and Account Repository. The Auth Manager's **role** is handling the user input. This occurs only when a user is registering for the first time or logging in for a session. The Account Repository acts as the persistence layer for the microservice's connection to the Authentication database. The Authentication Repository infrastructure handles all interactions with the database.

The Authentication microservice is only connected to one other microservice directly, namely the User microservice. With the bearer token that Auth Manager generates, a user can access all the functionality of our system. All interaction of the user with the microservices requires a valid bearer token that came from the Auth Manager.

3.3 Events

The Event microservice's main **role** is to store all the existing Events and provide this to the other services. The microservice has two main functionalities: one is creating, editing and deleting events based on input from the User service and the other is providing the Events to the ActivityScheduler. Therefore, the **responsibility** of the Event service is to correctly store and provide Events.

The adding of events is done via the "addEvent" port, which requires all of the information to be sent via a Post request, which is the main input of this service. The editing of events is done with the "editEvent" port, which requires new information that will overwrite existing information to be sent via a Patch request. Finally, the "deleteEvent" port solemnly needs an event id, sent by a Delete request.

The stored events are then sent as output to the ActivityScheduler when required. The retrieval of events is done via the "getAllEvents" port, a Get request which returns a list of all events currently stored in the Event service's database. The Scheduler service can also solemnly ask for the owner of

an event, according to the events' "eventId", which is done by a Get request to the "getEventOwner" port.

3.4 Activity Scheduler

As we have decided before, the ActivityScheduler's main **role** is to maintain the communication between the User and the Event microservices. This microservice has two important pieces of functionality: matching the users with appropriate events and sending notifications. The ActivityScheduler is then **responsible** for correctly passing requests and responses to the User and Event microservices, acting like a man in the middle. It retrieves, parses, and filters events, as well as user requests. It is also responsible for storing, updating, and deleting matches and notifications.

The matching process has two steps, highlighted by the "matchUserWithEvent" and the "getUserChoice" ports of the Component Diagram. First, we receive the User request as a Post request, which contains their general information (certificates, gender, organization, whether they are a professional or not), as well as their preferred positions and availability. The Scheduler then sends a request to the Event microservice, retrieving all events, which are afterward filtered to match the user's request. All these filtered events are then sent to the User microservice as output so that the user can choose their preferred event out of those. Processing this choice represents the second step of the matching, in which the match is saved in the database as "pending" and a notification for the owner is constructed and saved in the database.

The notification process is pull-based: in order to maintain the sychronous flow of the application, the notifications are stored in their database via the repository and can be retrieved at all times. All notifications of a specific user can be retrieved from the scheduler via the "getNotifications" port, which is accessed by the User microservice. Retrieving the owner's decision regarding a request ("getOwnerDecision") is another part of the notification functionality, in which a notification for the requesting user is constructed if the decision is positive and the match is updated as "not pending" in the database. If the owner rejects the user's request, the match is removed from the database altogether.

A Figures

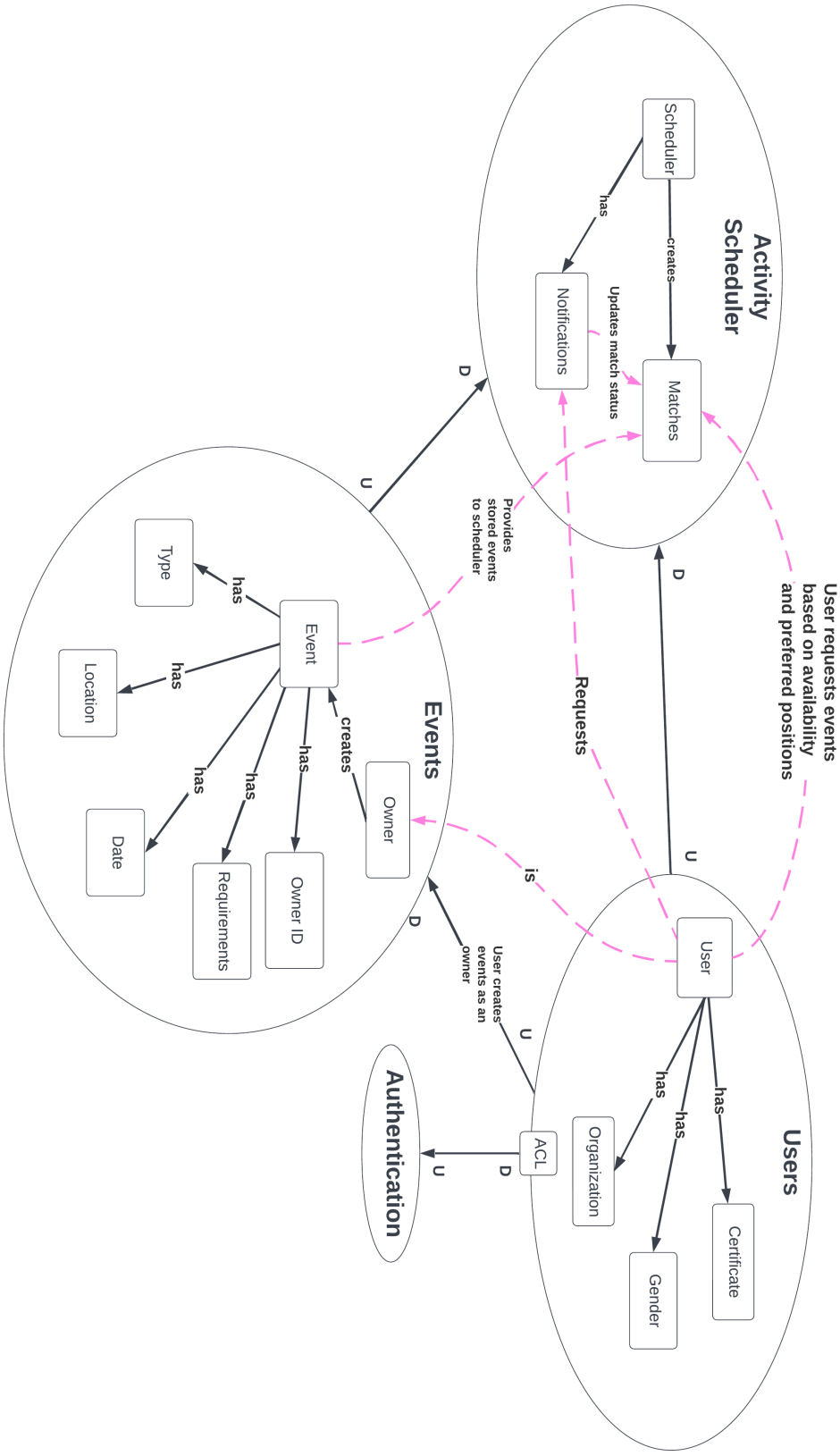

Figure 1: Context Map for the rowing application.

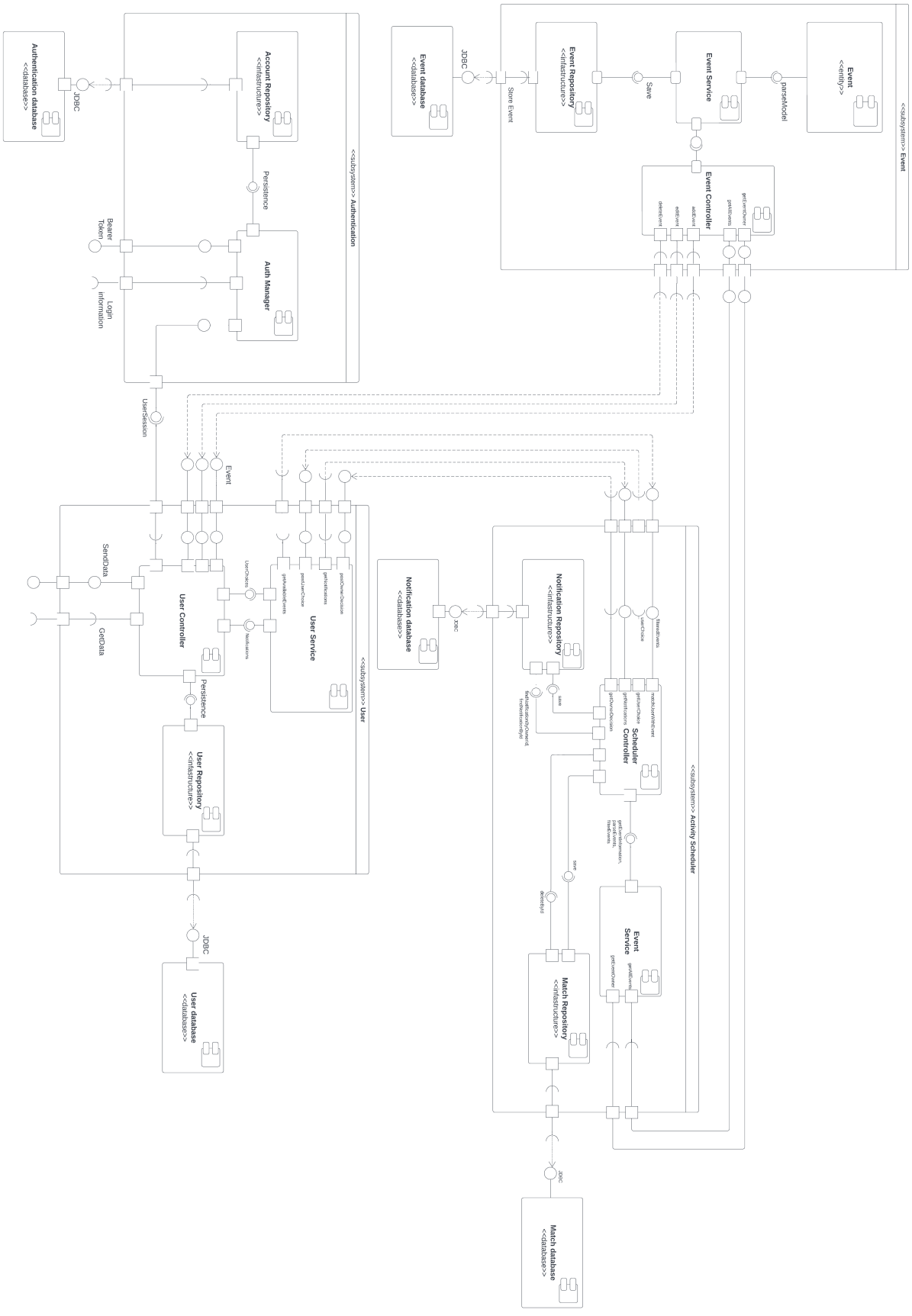

Figure 2: UML Diagram for the rowing application.