Lab Assignment 1: Bounded Context and Architecture

Software Engineering Methods

Group 30a

Markas Aišparas

Gayeon Jee

Maria Mihai

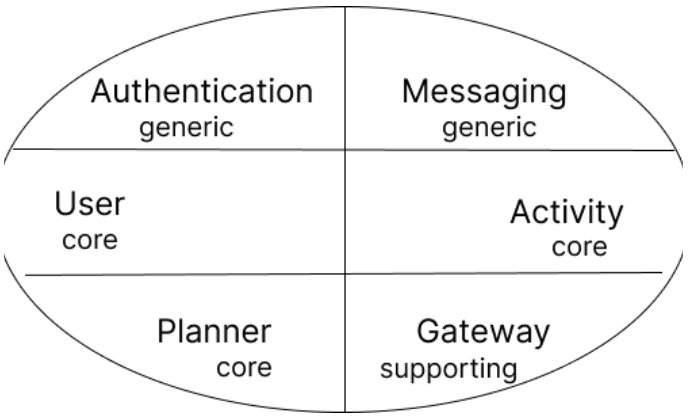
Jakub Patałuch

Zofia Rogacka-Trojak

Andrei Simionescu

Table of Contents

I. Bounded Context



We used Domain Driven Design when creating the architecture of the system, using the methodology described in Lecture 4.

We started by analysing the scenario and identifying domain related keywords: Training, Competition, Authentication, Security, Trainee, Account, Position, Availability, Gender, Skill Level, Certificate, Notification, Request.

After identifying the keywords, we started grouping them in related contexts:

Users, which contains all of the information about the users of the system: their account credentials, the positions that they can participate in and their skill levels, when they are available, the certificates they possess and their gender.

Activities, which contains all of the information about Training Sessions and Competitions. Common attributes are date and required roles, as well as required certificate when a Cox is needed. Competitions can have additional filters such as skill level or gender.

Planner is related to matching users to activities they are eligible for. It takes into account availabilities, roles and certificates, as well as the additional filters for competitions.

Messaging is responsible for informing users of the activities they have been selected for.

Authentication ensures that users can only access data they are authorized to access, and that users cannot impersonate other users or entities.

After further discussion, we identified the need for an additional bounded context: the **Gateway**, which is a supporting domain whose purpose is to facilitate the interactions between the system and its users.

II. Microservices

After identifying the bounded contexts, we needed to determine how to divide them into microservices. We began with the User microservice, which would store account information for rowing players so that we could match them. We then had to decide whether to include the activities in the same microservice as the users or to create a separate microservice for them. We decided that combining the User and Activity into a single microservice would make it too complex, so we kept them separate. The next step was deciding where the appointment scheduling (matching users and activities) would occur. We identified two options: it could be done within the Activity Microservice or in a separate microservice. The first option would reduce network use, but the second option would allow for better scaling, testing, and maintenance in the future, because the matching and the

activity management (such as creating, reading, updating, and deleting) would be handled by different entities. Therefore, we chose the second option and created the Planner microservice. The remaining bounded contexts were the generic Messaging and Authentication, as well as the Gateway microservice that supports them. We considered including messaging in the Planner microservice, but we decided to create a separate Messaging microservice to allow for independent evolution and scaling of these two. Regarding the authentication, there were two options: have each microservice implement it on its side, or have a global Authentication service. The first option has the following disadvantages: code duplication, taking time away from implementing the business logic, as well as future difficulties in maintaining the code. The second option would solve these problems, at the cost of increased latency. As latency is not an immediate concern for a rowing application, we decided to implement the second option. Finally, for the Gateway microservice, we could have allowed system users to place their requests directly with the respective microservices, but, for simplicity on the client side, we added the Gateway microservice.

All microservices communicate synchronously as when one microservice sends a request it waits for a response. There is no message queue keeping the requests and each microservice waits for a response, so the communication is not asynchronous. Moreover, all microservices have a hexagonal architecture to allow for loose coupling between them, and to make the components more easily interchangeable in the future, and to allow for better testing.

A. User Microservice

To improve the scalability and maintainability of our system, we have implemented a user bounded context within our domain driven design architecture. This context is encapsulated within a dedicated user microservice, which is responsible for managing, distributing, and persisting user data. To accomplish this, the microservice employs an interval service to handle CRUD operations and basic filtering requests on the user entity.

For example, when the Planner microservice sends a request for user IDs matching specific criteria, the user microservice will fulfill this request by providing the necessary information from the user database. This establishes a downstream relationship between the user and Planner microservices, as the user microservice supplies the data needed for the Planner to create matches. The user bounded context also has a downstream relationship with the Messaging bounded context, allowing for the retrieval of user-specific information such as email addresses and names.

Additionally, the user bounded context has an upstream relationship with the Gateway, as it must be able to receive authorized requests to create, modify, or delete user data. The Gateway may also request changes to a user's availability or role preferences, which will be handled exclusively within the user bounded context. This approach ensures that any modifications to the user entity are immediately reflected, without the need for the Planner microservice to update its own data.

To prevent unnecessary coupling, the user microservice does not communicate directly with the activity microservice. Instead, the user microservice focuses on efficiently managing and persisting user data, and providing relevant information to other microservices through its interface. This interface is designed to only return the specific data that other microservices require, rather than the entire user object, through the use of DTOs.

B. Activity Microservice:

One of the core components in our bounded context is Activity. We decided to distinguish activity as a separate microservice as it provides necessary functionally to manage activity entities. Its key role is to provide users with the possibility of creating, editing and deleting the activities.

The microservice communicates with others via DTOs. It is in an upstream relationship with Planner Microservice. It sends the planner information about all available activities in the databases so that it can create matches with the user preferences. Activity microservice provides an interface called Activity Manager, which enables authorized users to alter/modify existing activities, as well as to create or delete them.

The microservice has its own database, where it stores both types of activities, namely Training and Competition, with appropriate attributes. Training and Competition inherits from an abstract class Activity. It also has an Activity Service where all relevant endpoints to communicate with the database are located, and these functionalities are encapsulated via the Activity Controller class. Additionally, the authentication component authenticates all requests reaching the microservice.

C. Planner Microservice

The planner microservice provides the user the activities they are eligible for. It is the downstream component receiving information from the user because it provides matches depending on the user's information. Activity is also its upstream component since planner microservice depends on the activity microservices to provide the activities. All requests come through the API gateway. Thus, it is another upstream component of the planner. It communicates with other microservices through the planner manager interface, making use of DTOs.

There are the following components: database, planner service, and authentication. Database saves applicant's user id, match id, and status (one of revoked, pending, and accepted). The planner service implements functionality of the microservice. To explain briefly about its behavior, once the user requests for the eligible activities, it initially shows all available activities. Then, only when the user applies for it, it saves the match. Additionally, all requests arriving at the microservice should be authenticated and the token should be validated through the authentication component.

D. Authentication Microservice:

This microservice facilitates a secure use of the core components of the system by authenticating users. Its role is to provide users with a way to login and register via a combination of a username and password.

The authentication microservice has an upstream relationship with the API Gateway, providing it with a JWT token whenever a login operation is performed. This JWT can be used by other microservices to authorize requests to their core domain services, ensuring that only authenticated users can access the system's functionality.

To facilitate this authentication process, the authentication microservice maintains its own authentication database, separate from the user microservice. The authentication microservice has a downstream relationship with the user microservice. Whenever a register request is received, the authentication microservice sends the request to the user microservice to register the user. If the request is successful, the user id is returned from the

user microservice and stored in the authentication database along with the associated login information.

By decoupling the authentication logic from the user microservice and implementing it in its own authentication microservice, we enable the system to easily support alternative authentication methods in the future. For example, we could integrate with external Single Sign-On (SSO) providers to acquire JWT tokens and provide users with even more convenient and secure ways to access the system.

E. Messaging Microservice:

Messaging Microservice exposes an interface which enables other microservices (in our case just planning) to send messages of a given type to the user. That microservice will have an upstream relationship with the Planning one, providing functionality whenever it needs to send a given type of notification to the given user, but it won't be accessible straight through the API gateway.

The role is to provide other microservices with a way to notify user about certain events (e.g acceptance of the user for certain activity).It is designed using a strategy design pattern, a pattern which enables selecting an algorithm at runtime in our case the way of sending the notification (e-mail, sms, etc.). It doesn't have any persistent data so no DB is included, just controller, service, certain API connectivity libraries and authentication that doesn't take JWT but just restricts origin usage.

By separating the messaging logic away from planning microservices we not only enable other microservices to use that functionality in the future but also enable us to provide multiple custom ways of notifying the user by providing strategy design pattern as mentioned above.

F. Gateway Microservice:

The Gateway microservice's role is to facilitate and simplify interactions between the system and users. We chose to have this component instead of simply having system users directly query the microservices that correspond to the core bounded contexts for the following reasons: Firstly, to abstract away the different operations from the client side perspective. Secondly, to not expose the underlying implementation. Lastly, in order to allow adding new implementation without the risk that existing client side code would be broken.

It communicates with the Planner, Authentication, User and Activity services, to which it passes on the user requests. It therefore has an upstream communication with these microservice.

The Gateway microservice exposes the ApplicationAPI interface, which contains all of the actions that a system user can perform. To provide this functionality, the Gateway requires access to the UserManager, ActivityManager, and PlannerManager interfaces, as well as the AccessControl interface provided by the Authentication microservice. Since user queries are not persisted, the Gateway does not have a database of its own. Instead, it uses the facade design pattern to provide a unified interface for interacting with the underlying subsystems. This allows us to isolate clients from the complexities of the system, while providing a consistent experience for users.

Instead of relying on the gateway to authenticate the JWT, each microservice authenticates the JWT by itself to ensure that the token is valid even in the case where a malicious actor would try to bypass the gateway.

III. UML Component Diagram

The UML Component Diagram below reflects all of the described characteristics: