Assignment 1 – Group 29A

"Rowing" is a platform that allows people to join rowing events such as trainings and competitions for their team. This program will help organisations fill up open positions with members that have signed up on the platform without having to ask around for available people.

MICROSERVICE DESIGN

Our team has decided to use 5 microservices that will each handle a part of the program: authentication, user, event, gateway and notification microservices. This means that, in the event of a malfunctioning system, some actions can still be performed by users. Each microservice will follow a **hexagonal design**, which will not have the presentation layer implemented due to the client's request (there is no user interface that interacts with them). Splitting the work between more microservices would be redundant and would create more dependencies between tasks than we would need. Having fewer microservices would scale the app vertically and thus reduce the performance of our code.

In order to realise the implementation, we have broken down the idea into smaller tasks. The main components of the program will be the **authentication system**, the **user**, the **notifications** and the **events** (competitions and trainings). These will make the bounded contexts of the architecture.

The **gateway microservice** will permit all users to communicate with the application using the same port. This will aid verify if a user is authenticated before performing any action. All API calls will go through this gateway that will map the requests to adequate microservices for future handling. We chose to implement a gateway because it offers multiple benefits. Firstly, it allows for a better scalability of the service, being able to direct traffic to the suitable microservice. Secondly, by being the single entry point into the system, it will remove the problem of determining the location of each microservice and also reduce the number of requests as the gateway enables clients to call multiple services with a single request and, thus, improving the user experience.

The authentication system will be a generic domain because we need to have a functional scheduling platform in which multiple people can join. Accounts will store the data needed to match a user with an event, as well as a unique id to distinguish between different users. Using this system, we prevent users from wrongfully modifying data or accessing resources they are not supposed to have access to.

The **authentication microservice** will use the data passed by a user as an infrastructure layer, but also access data from the *user* database through the *userService*. The domain layer contains only the *appUser* which will contain only the needed information for authenticating (email, id & password) and hashing methods for the password. The third layer application layer will be responsible for the logic of the microservice i.e. hashing passwords and validating an email.

Users are one of the core domains of the application. They will be the main triggers of actions throughout the whole system of microservices. They will interact with every entity there exists. Without users, the purpose of the platform would not exist.

The **user microservice** will contain an infrastructure layer, that represents the database which will contain every user that has joined the platform and information relevant to signing them up to an event. The domain layer will contain every entity and auxiliary data from the *shared* package. The application layer will work on creating new users, editing their information and deleting them if wanted and performing enqueueing for an event.

We decided to split authentication from the user microservice for a number of reasons. Firstly, it offers flexibility in how we can structure our program, embracing the single responsibility principle as each microservice will only handle single functionality. Secondly, it enhances the security of our program as we do not store all the data in one place. Moreover, authentication in microservices can lead to a variety of complex situations which can make it hard to manage. This means that having them separated will make it easier to implement across all microservices

Events are also a core domain. They are the main object that users interact with. There will be 2 types of events: trainings and competitions. Users join events by asking the system to enrol in the ones they chose. After being enqueued, the event owner will be able to select whether they will accept them in the team or not.
The **event microservice** will also contain an infrastructure layer that will store every pending event and all related information. Users will be able to access it in order to view matching events. The domain layer will contain every entity from the *shared* package. The application layer will include implementations regarding the creation, editing, and deletion of events (the latter 2 being performed only by the creator). It will also include logic needed for enqueueing for an event.

The notifications are a supporting domain as they will inform users about the outcome of their requests to join events. Users will be able to see a list of the received notifications on the platform, but also via email..

The **notification microservice** will not have a separate infrastructure layer, but it can access data both from the *user* & *event* databases using their services. The domain layer will concentrate on the design pattern that it will use to give the ability of having multiple ways of sending notifications: by email and by using the platform. It will contain the different strategies that it uses (which permit scalability if more options are conceptualised), but it will also access the *event* entity. The application layer mainly implements the structure of the messages and changes a user's data if necessary (we will not be using an auxiliary emailing system, thus not implementing the actual action of sending email).

With this architecture, the different objects interact minimally, but still provide the intended actions to the users' requests, without causing any damage to the already enrolled members' experience if there are system failures.

UML DIAGRAM

The description of the system architecture might seem complicated when trying to put it in words, which is why we have drawn the following diagram to illustrate how the components of the software interact (see *Picture 1*).

MICROSERVICES

Gateway microservice:

This microservice provides all functionality related to connecting all created subsystems. The subsystem ensures that the implementation will not have code duplication and will create a smooth connection between all microservices. More specifically, it responds to the following types of requests:

- *Redirect* - given a specific request it calls a microservice responsible for handling the data.

Authentication:

The purpose of this microservice is to authenticate the users and distribute tokens. It is necessary in order to only allow verified users to access the microservices. Each user receives a token after logging in or registering. Those two actions are done by rest API calls including the details such as password and username.

After receiving a token the client will use it to access each microservice. We decided to perform a security check on each microservice instead of using a gateway because we believe that in the future when the number of users would increase, the gateway would be a bottleneck in our system. Furthermore, we think that we would be putting a lot of pressure on the gateway.

- *registering* - parsing the details of the user and creating an instance of a user in the user database
- *Logging in* - inputting the logging details and returning a token

User microservice:

This microservice provides all functionality related to creating, editing and deleting users. Every account must have all information filled in before performing any actions. The system will use **synchronous** communication in order to make sure all information is appended to the database and avoid errors in later dependencies.

- *Editing/ adding /deleting user information* - the microservice will handle all the operations regarding the user's data i. e. editing, adding and deleting information.
- *Requesting latest notifications* - the microservice will fetch all new accept/reject notifications belonging to the given user

Events microservice:

Events will be accessible to users using this microservice. This will administrate every single detail about them and will permit actions that do changes to the *event* database. This microservice calls the *Notification* microservice **asynchronously** when the creator of an event accepts/rejects a user from the event's queue. Other than that, all communication supported by this will be **synchronously**, as the client will receive an answer before another action is being started.

- *Creation* - a user can create an event by filling up all needs

- *Editing* - the creator can edit information about the event (except for the time if it's a competition)
- *Deletion* - the creator can delete the whole event
- *Matching* - match a user to all the events that they can join to using the information from the time that the request was done (not going to update on eventual updates of events/profile)
- *Enqueue* - the user can enqueue to an event and be added to the list
- *Accept/reject request to join event* - Accept or recent a user to the event

Notification microservice:

Notifications are being sent as soon as the creator of the event accepts/rejects a user to their event. This one will not have a separate infrastructure layer, but will make use of the event and user databases using the fitting services and APIs. The *User* microservice triggers the run of this microservice that will do modifications to a user's data (append a notification), meaning that it will communicate **asynchronously** with the user. This system will implement:

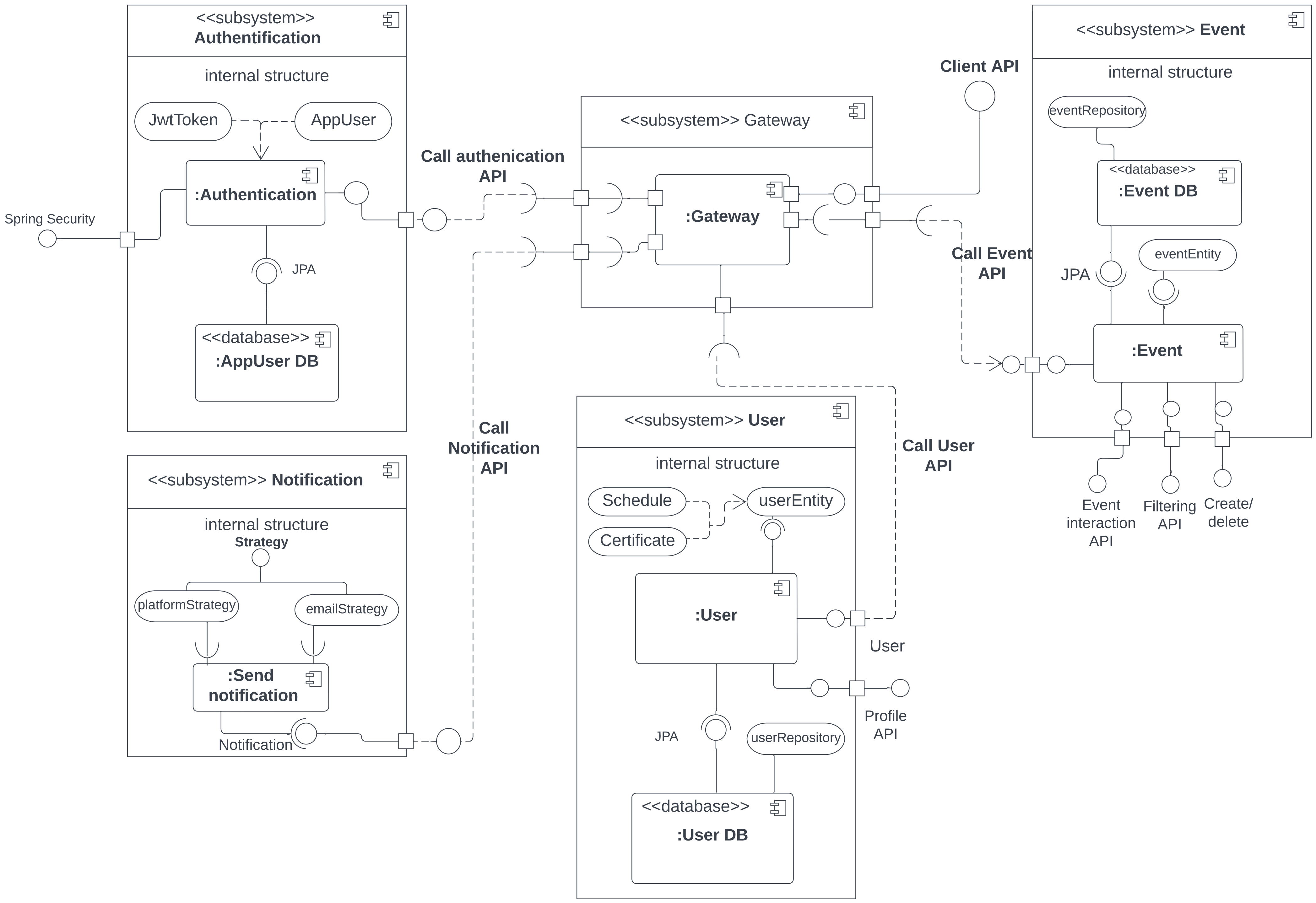
- *Send notification email* - a user will receive a notification email with their queue status
- *Send notification on the platform* - a user will have a notification appended on their platform and can see their queue status without the need for a mailing system

DESIGN PATTERNS

Due to the fact that we have decided to utilise 2 ways of sending notifications, in order to minimise code duplication, we have opted for the *Strategy* design pattern, that will design when the system should add the notification to the platform and when it should send the message to the registered email. This decision will also allow further changes if more notification sending options are wanted in the future.

The *notification* entity will be able to change strategies, meaning that it will transform the way it deals with notifications. In our idea of implementation, both options will be used. Both strategies will have the same idea of implementation, only with slight changes.

Picture 1 - UML Diagram



<<subsystem>> Authentification
internal structure
JwtToken
AppUser
:Authentication
Spring Security
JPA
<<database>>
:AppUser DB
Call authenication API
<<subsystem>> Gateway
:Gateway
Client API
<<subsystem>> Event
internal structure
eventRepository
<<database>>
:Event DB
eventEntity
JPA
:Event
Call Event API
Event interaction API
Filtering API
Create/ delete
<<subsystem>> Notification
internal structure
Strategy
platformStrategy
emailStrategy
:Send notification
Notification
Call Notification API
<<subsystem>> User
internal structure
Schedule
Certificate
userEntity
:User
User
Profile API
JPA
userRepository
<<database>>
:User DB
Call User API