

April 23, 2023

DESIGN REPORT

# WETEAM

Nicolae Mihălache	s2553570
Nicolae Rusnac	s2467585
Sandu-Victor Mintuş	s2468034
Victor Horneţ	s2458543
Frank Bruggink	s1500430

**Faculty of Electrical Engineering, Mathematics and Computer Science  
(EEMCS)**

Supervisor:  
Y.d.C. Barrios Fleitas MSc

**UNIVERSITY OF TWENTE.**

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Background . . . . .	5
1.2	Current Solutions . . . . .	5
1.2.1	Matchmaking by students . . . . .	5
1.2.2	Matchmaking by teachers . . . . .	5
1.2.3	Creating a custom script . . . . .	6
1.3	Goal . . . . .	6
<b>2</b>	<b>Requirements</b>	<b>7</b>
2.1	User Stories . . . . .	7
<b>3</b>	<b>Global Design</b>	<b>9</b>
3.1	Proposed Flows . . . . .	9
3.1.1	Event organizer flow . . . . .	9
3.1.2	Participant flow . . . . .	10
3.2	Authentication . . . . .	10
3.3	User manuals . . . . .	12
3.3.1	Authentication . . . . .	12
3.3.2	Teacher Manual . . . . .	12
3.3.3	Student Manual . . . . .	14
<b>4</b>	<b>Backend Design</b>	<b>16</b>
4.1	Database design . . . . .	16
4.1.1	PostgreSQL . . . . .	16
4.1.2	Prisma . . . . .	17
4.1.3	Database Schema . . . . .	17
4.2	API . . . . .	17
4.3	Users and profiles . . . . .	18
4.4	Error Handling . . . . .	19
4.5	Matchmaker workflow . . . . .	19
4.5.1	Initial implementation . . . . .	19
4.5.2	Team Power . . . . .	21
4.5.3	Matching process . . . . .	23
4.6	Filters . . . . .	24
4.6.1	Profile filters . . . . .	24
4.6.2	Team Filters . . . . .	26
4.7	Notifications . . . . .	26

<b>5</b>	<b>Frontend Design</b>	<b>27</b>
5.1	Team/user cards . . . . .	27
5.2	Authentication . . . . .	28
5.3	Matching arena . . . . .	29
5.4	Forms . . . . .	29
5.5	Creator View . . . . .	31
5.6	Team menu . . . . .	31
5.7	Prompts . . . . .	32
5.8	Notifications . . . . .	32
<b>6</b>	<b>Testing</b>	<b>33</b>
6.1	Backend testing . . . . .	33
6.1.1	Postman . . . . .	33
6.2	Frontend testing . . . . .	33
6.2.1	Manual testing . . . . .	33
6.2.2	Usability testing . . . . .	34
<b>7</b>	<b>Future work</b>	<b>35</b>
7.1	Adding filters (frontend) . . . . .	35
7.2	Customizable criteria . . . . .	35
7.3	Chat . . . . .	35
7.4	Merging teams . . . . .	36
7.5	General improvements . . . . .	36
7.5.1	Styling . . . . .	36
7.5.2	Large-scale testing . . . . .	36
<b>8</b>	<b>Reflection</b>	<b>37</b>
8.1	Strengths . . . . .	37
8.1.1	effective project management . . . . .	37
8.1.2	Technical expertise . . . . .	37
8.2	Limitations . . . . .	37
8.2.1	Time . . . . .	37
8.2.2	Analysis paralysis . . . . .	38
<b>9</b>	<b>Appendices</b>	<b>39</b>
9.1	Appendix A. Work Distribution . . . . .	39
9.2	Appendix B. Development workflow . . . . .	40
9.2.1	Responsibility Areas . . . . .	40
9.2.2	Role distribution . . . . .	40
9.3	Appendix C. GitLab repositories . . . . .	40

# Abstract

The formation of effective teams is critical for completing a successful project, especially in an academic environment. However, this is a complex task because there are a lot of variables to take into consideration. Teachers have a difficult time forming the teams themselves, since often they do not possess the necessary information such as: background information about the students, preferences of students and knowledge about best practices for forming teams. Likewise, if students are given the opportunity to form the teams, they face the same challenges. This report describes an application that aims to facilitate the team formation process. It gives the teachers the opportunity to use a set of criteria that will be used to form the teams. Then, students complete a profile where they provide information about themselves, according to the set criteria. Our application suggests teams to the students based on their information, but also gives them the opportunity to form the team themselves. Hereby taking the burden off the teachers and giving the students more liberty to form the teams and providing them with the necessary information to do it.

# Chapter 1

## Introduction

### 1.1 Background

Forming teams in within an educational environment with a lot of people is not an easy task. Especially if the team formation is bound to criteria such as group size, diversity or expertise. This could be done manually by the teacher, but this will cost lots of work. Likewise, letting students make their own teams can also be problematic since often they do not possess enough background information about their peers in order to make informed decisions. Moreover, there are also no platforms where this could be easily arranged, so students often use applications such as discord or WhatsApp to arrange this themselves. As we can see, from both teacher and students perspective this is a very cumbersome process.

### 1.2 Current Solutions

The solutions that are currently there are as follows:

- The teachers lets the students form the teams themselves. This has several drawbacks
- The teacher forms the teams
- The teacher writes their own custom script

#### 1.2.1 Matchmaking by students

This is a well used approach especially with smaller groups and groups of students that already know each other. Student making teams themselves usually results in teams that the students are happy with and this process

#### 1.2.2 Matchmaking by teachers

Another approach is to have the teacher do the matchmaking by assigning the students themselves and making sure the criteria are met. Using this approach does

most likely ensure that the criteria are met but it will be a very time consuming process which will only worsen when the amount of students increase. This also takes away all agency from the students which usually is not preferred.

### **1.2.3 Creating a custom script**

For the more tech savvy teacher, creating a custom script to assign student at random while still conforming to certain criteria may certainly be a solution as this saves a lot of time over manually assigning students. The drawbacks, however, are that as the criteria change the script also has to be changed each time team need to be formed. The criteria may also include information the teacher does not have access to. Thus requiring them to create a custom survey and incorporating the results into the script. Additionally, as with the previous solution, this also takes all agency away from the students.

## **1.3 Goal**

The goal of this project is to create an application that provides smart matchmaking for students in order to make it easier and less burdensome for teachers to form teams for their course. Students will have enough information about their peers so they can make an informed decision and they are also provided with a friendly user interface where the matchmaking is taking place. The application should:

- Assist students in making informed decision when forming a team
- Improve the user experience of managing a team over the one on canvas
- Allow teachers to encourage team formation according to a specific criterion

# Chapter 2

## Requirements

Requirements elicitation was important in order to determine what does the client expect from the application. During the first meetings with our supervisor we have understood better what is expected from us and what is the purpose of the application. We have learned that during the previous design project there was another group with a similar project. Their purpose was to develop an application that would smartly generate teams based on some information about the students. Our project is quite similar to the previous one, however it also has some key differences. Our application should provide an environment where students could find possible teammates with enough information available so that they can make informed decisions. Moreover, ideally such an application could be extended so that it can be used for forming teams in any kind of environment. We have also discussed about what features should have the app and what should be the main focus. If we focus on university environment solely, then our main stakeholders are the teachers and the students. Teachers would use the application to create a matchmaking event and the students would join this event and browse their peers looking for possible teammates. Having done that, we have come up with a list of requirements in the form of user stories. We have written the user stories from 3 perspectives: General user of the application, a user who creates a matchmaking event (in a university environment these are the teachers) and a user who participate in the matchmaking event (in a university environment these are the students).

### 2.1 User Stories

<b>User</b>
As a user, I want to authenticate
As a user, want to use the application on both a computer and a mobile device

**Participant**

- As a participant, I want to be able to join a room with a code
- As a participant, I want to be able to see and access the rooms I joined
- As a participant, I want to create a profile containing relevant information
- As a participant, I want to see details about the other participants' profiles from the same room
- As a participant, I want the profile to have tags displaying relevant information
- As a participant, I want to filter other profiles based on tags
- As a participant, I want see a team's fitness rating when looking for teams
- As a participant, I want to request to join a team by sliding on a card
- As a participant, I want to filter out the teams which I cannot join
- As a participant, I want to see the restrictions that do not allow me to join a team
- As a participant, I want to become a team manager by starting a new team or joining a team
- As a participant, I want to leave my current team

**Event Organiser**

- As an event organiser, I want to be able to manage an event for a team-creation session
- As an event organiser, I want to specify the parameters used in determining team members, such as their skill with certain tools
- As an event organiser, I want the team to follow the specified restrictions, such as the amount of people of the same nationality
- As an event organiser, I want to be able to handle edge cases in case of leftover team members
- As an event organiser, I want to export the formed teams



# Chapter 3

## Global Design

### 3.1 Proposed Flows

To have a clear idea of the application we have made several flow diagrams to capture everything that should happen in our application. We have made the diagrams both from the perspective of an event organizer and from the perspective of a participant.

#### 3.1.1 Event organizer flow

An event organizer would login into the application. Then, they would create a new matchmaking event. For this they will set the matching criteria that will be used for matching and the team size. Afterwards, they will share the code with the students via some external tool. After the matching process has finished, they will export the formed teams to a csv file.

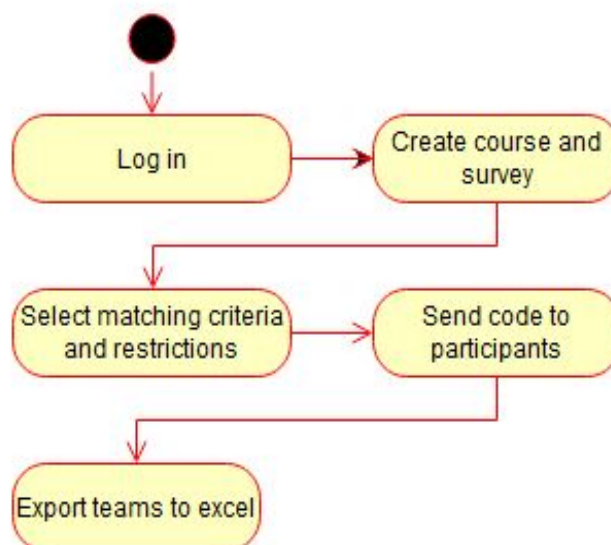


Figure 3.1: Event organizer workflow

### 3.1.2 Participant flow

The participant in a matchmaking event has a more complex workflow. Firstly, they authenticate into the app and use the code received from the organizer to join a room. They enter the code and join the event. Now they start the matchmaking process. They swipe left or right on their peers who may be already part of a team. There are 2 possibilities for a user to match with someone. If they match with someone who is not in a team, then they are paired together and form a new team of 2 members. Otherwise, they join the team that they matched with. Having joined a team, now the user also swipes either right or left on users until they either leave the team or the formed team is complete. You can see the diagram in figures 3.2 and 3.3

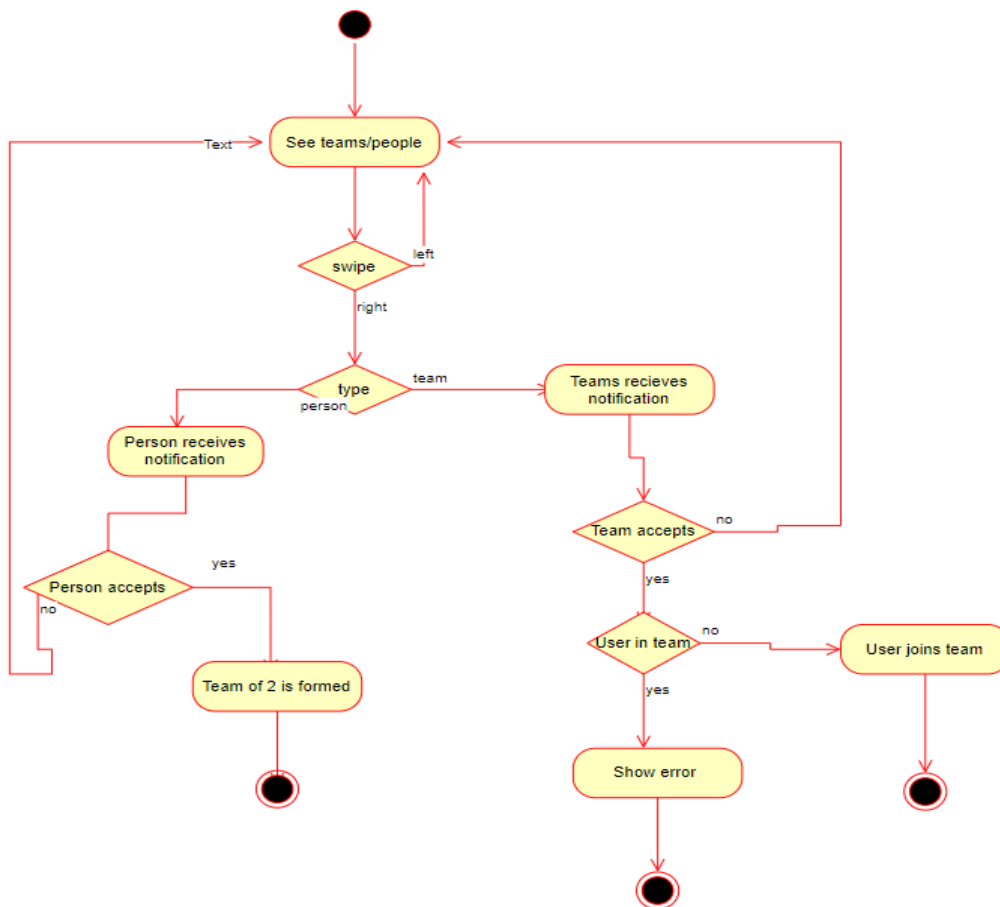


Figure 3.2: Person without team workflow

## 3.2 Authentication

Authentication design was very important for the security of our application. During the early meetings with our client we learned that there are no strong requirements in this regard. We have considered several options, such as using third-party accounts, such as Microsoft Google accounts. We decided that for the scope of this application it is not necessary to integrate these tools since it takes more time to

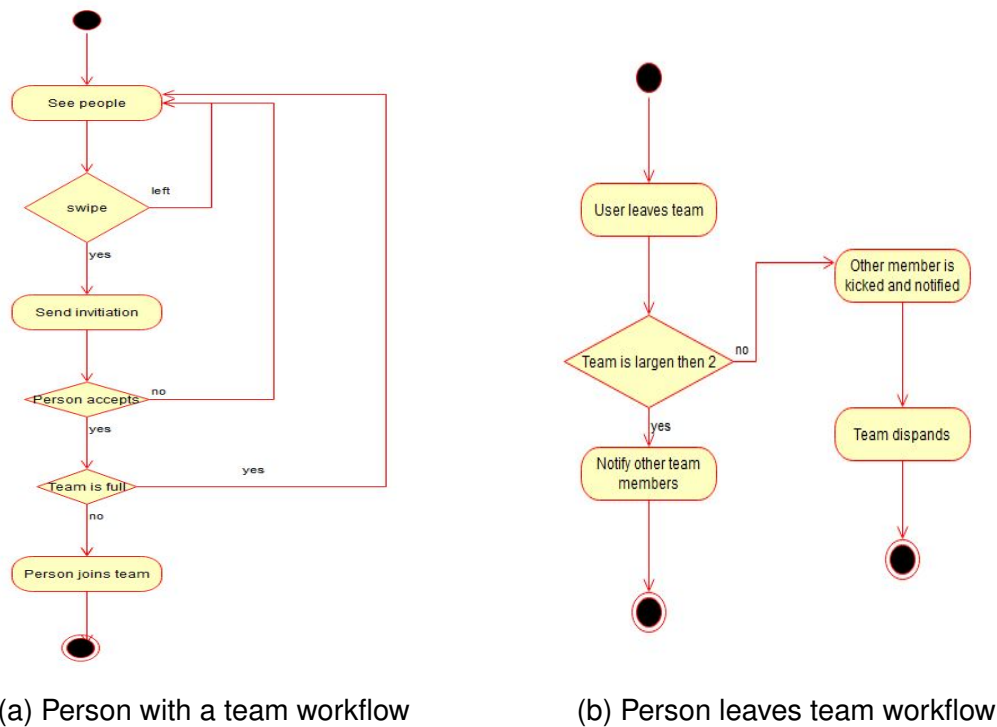


Figure 3.3: Participant workflow

implement and there are not that many benefits. Of course, it would have been easier for a user to login with an already existing account, but it would have made the development slower and we wanted to put more effort into other aspects of our application.

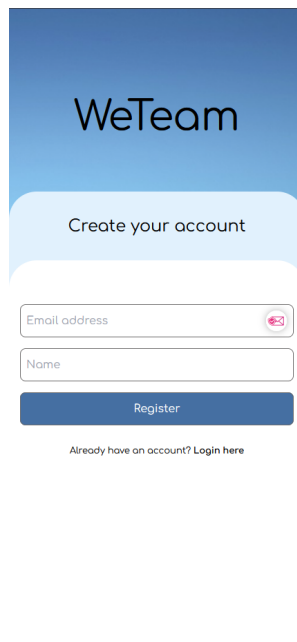
Therefore, we use a simple email and password combination. When a user registers for the first time, they enter their email and a password is generated and sent to their respective account. They can use this password to login into the application. If they wish to, they can do so via the "forgot password" button. Sending the password via email is a decision that comes with both, benefits and drawbacks. Potentially, it is prone to man-in-the-middle attacks when a hacker may intercept the email and get the password in plain text. However, this is unlikely and our application does not store any valuable information about a person, besides some specific information relevant for the team formation process. A benefit of this is that the user gets a strong password by default. Moreover, as we have mentioned, the user always has the opportunity to change their password. In order to send the emails we use a service called Mailgun. It provides APIs for sending, receiving and tracking emails. We chose it for its simplicity and reliability.

In order to persist the user session we use a JWT (JSON Web Token) which is generated each time the user logs in. The token is available for one hour, after which the user will have to log in again to generate a new token.

## 3.3 User manuals

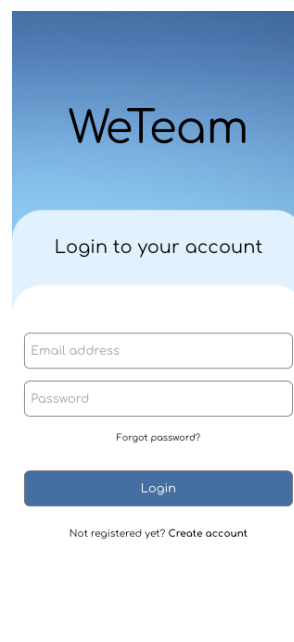
### 3.3.1 Authentication

Both teachers and students have the same authentication flow. To register, a user must provide their name and email, then they will receive a one time password with which they can log in.



The registration page features a blue header with the 'WeTeam' logo. Below the header is a light blue rounded rectangle containing the text 'Create your account'. The form consists of two input fields: 'Email address' with an email icon on the right, and 'Name'. A blue 'Register' button is positioned below the fields. At the bottom, there is a link that says 'Already have an account? Login here'.

Figure 3.4: Register page



The login page features a blue header with the 'WeTeam' logo. Below the header is a light blue rounded rectangle containing the text 'Login to your account'. The form consists of two input fields: 'Email address' and 'Password'. A blue 'Login' button is positioned below the fields. Above the button is a link that says 'Forgot password?'. Below the button is a link that says 'Not registered yet? Create account'.

Figure 3.5: Login page

### 3.3.2 Teacher Manual

#### 3.3.2.1 Room management

From the home page, teachers can create a new room by providing a name, description, expected team size and priority match criterion. Teachers can then see all their created rooms in the Arena tab, where they can also copy the code and share it with their students. Teachers can also select a room from the arena tab to see its details, such as the amount of teams, both complete and incomplete, as well as the number of students who are still not in a team.

#### 3.3.2.2 Exporting teams

Once the team formation period has ended, the teacher can access the room's page to export the formed teams as a csv file.

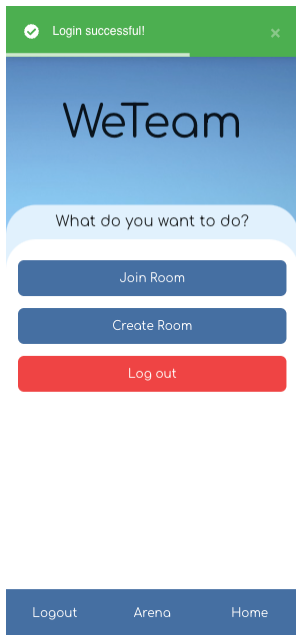


Figure 3.6: Home page

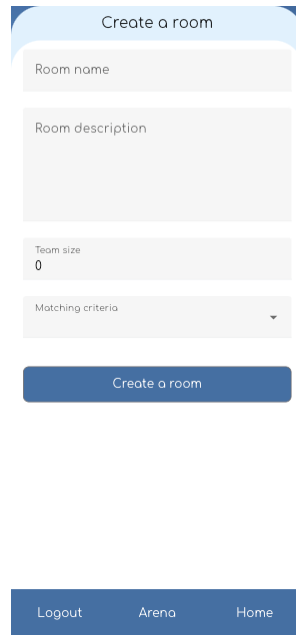


Figure 3.7: Create room page

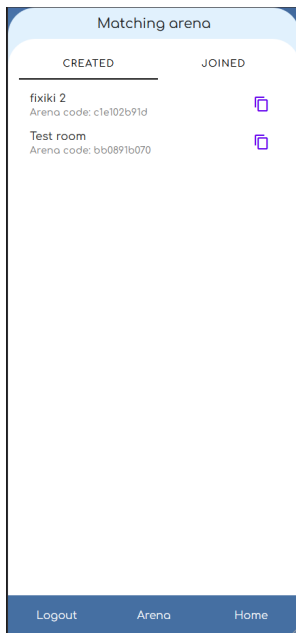


Figure 3.8: Created rooms page

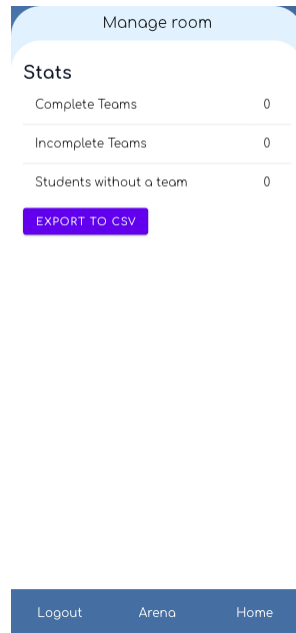


Figure 3.9: Room details page

### 3.3.3 Student Manual

#### 3.3.3.1 Room management

From the home page, students can join a room with the code given by their teacher. When joining a new room, students will be asked to fill their nationality, expected grade, known programming languages, belbin roles, preferred work location and a short description. This data will be used for computing the power of the teams. Students can then find all their joined rooms in the Arena tab, from which they can open the "Matching Arena" page.

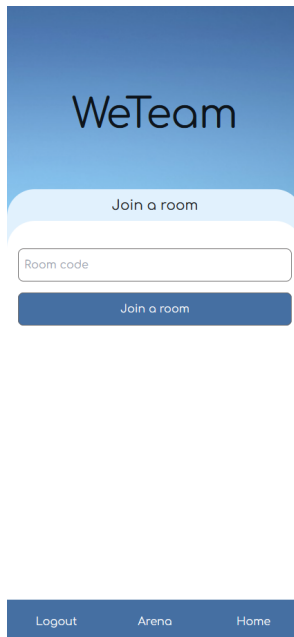


Figure 3.10: Join room page

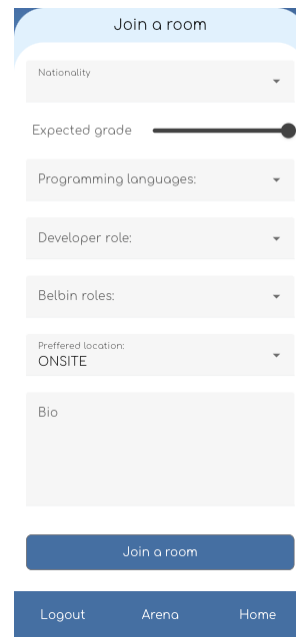


Figure 3.11: Profile form

#### 3.3.3.2 Matching process

In the "Matching Arena" page, students will be presented with a card of a potential match: either an incomplete team or a student without a team. Students will be able to swipe left to decline the match, or right to like the presented profile. If two students like each other, they will form a team. The same happens between a team and a solo student. Furthermore, when students swipe left on a match, the match will be added to their blacklist. This prevents the match to appear again until there are no more available matches. The blacklist will be cleared once student does not have any more available matches.

#### 3.3.3.3 Team management

Also in the "Matching Arena" page, students who joined a team can access the teams' details: its members and incoming requests. The incoming requests are solo students who want to join this team. From this dialog any team member can accept or decline an incoming request, which will either add the new member or

blacklist them. Furthermore, students can always see their current team members, as well as leave the team from the "My Team" dialog.

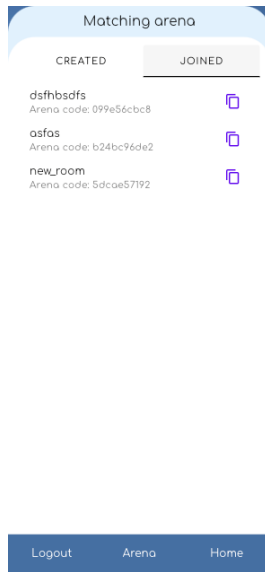


Figure 3.12: Joined rooms page

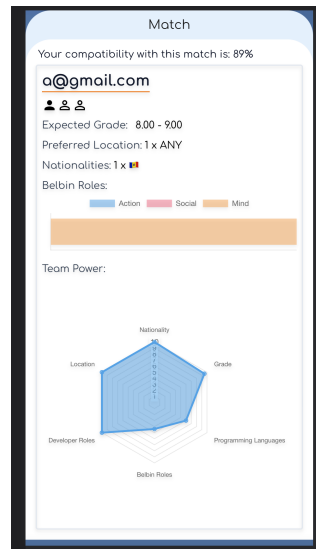


Figure 3.13: Match card

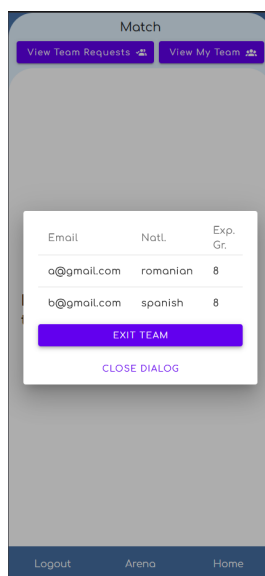


Figure 3.14: "My team" view

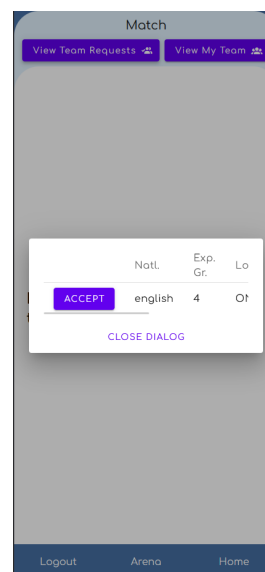


Figure 3.15: Incoming requests view

# Chapter 4

## Backend Design

The design of the back-end was a very important step for our application. We aimed to create a structure with a good separation of concerns, so that the code is kept clean and easy to maintain. It was also important to make a design that is easy to maintain and scale since our supervisor told us that he wants to add extra features on top of the application. The first part consisted of making a structure for our database. It was important to spend time on this so that we do not have to spend extra-time on making changes. In the end we did have to add some extra data structures in the database, because the application turned out to be more complex than we had originally thought. Having done that, we started working on the structure of code of the back-end. We have decided to use Nest.js for our back-end. It is a Node.js framework which adds several important features on top of it. First of all, it is built with Typescript, a super set of JavaScript which supports types variables and allows to write more reliable code. It also has a rigid, modular structure which improves the maintainability and the scalability of the application.

### 4.1 Database design

Database design was one the first steps of our development process. We have decided to use a relational database, because it allowed to have a clear structures for our tables. Likewise, our application is complex, therefore it was more optimal to have more rigid data structures.

#### 4.1.1 PostgreSQL

PostgreSQL is one the most popular choices in this regard. It is an open-source database system known for its ability to handle large amount of data and maintain high performance. Moreover, it is compatible with a large number of back-end systems, which increases the flexibility. Another important reason is that all of our team members have had experience working with PostgreSQL in the past, so for us it was a very convenient choice.



### 4.1.2 Prisma

We also decided to use an ORM(Object-Relational Mapping) for communication with the database. It provides a good abstraction by mapping data structures used in object-oriented programming languages to those used in relational databases. It allows us to avoid writing the SQL queries manually, which increases development productivity. It is also more secure, because it protects against attacks such as SQL injections and cross-site scripting. For our ORM we use Prisma, which has some useful extra features. It is strongly typed, which reduces the number of potential bugs. Moreover, it is compatible with several popular database, such as MySQL, PostgreSQL, and SQLite. This is crucial for scalability. If in the future the developers that will work on this application wish to use another database, they will not need to change the code that we use to communicate with it. Moreover, it provides a very nice visual editor called Prisma studio which allows to perform operations on the database directly from this UI.

### 4.1.3 Database Schema

The database Schema consists of nine data models. The User model stores general user data such as the email, name password and salt. It has an artificial key as a primary key. The room model contains details about the room, such as the creator of the room, the description, code and the required team size. The Team model stores data about the team, such as the room and the profiles. The Profile model stores user data for each individual room joined by the User. This consists of bio, nationality, expected grade, preferred working location, Belbin roles and programming languages. This model has many-to-one relation with the User table, because the same user can have multiple profiles, many-to-one relation with the Room model, because there are multiple profiles in one room, and a many-to-one relation with the Team model, because a team consists of several profiles, depending on the team size requirements. These four models represent the core of our application, since they store most of the important data. The main issue that we faced at this point is how to store the flow of the matching process. We had to represent scenarios when the user is in a team or not and when they either like or dislike someone. For this we decided to use a separate data structure for each separate case. We had thought about creating one data structure that would cover all the cases, but we decided that it would become cumbersome to operate with and overly complex. The Like model represents the case when a user swipes right another user. The Request model covers the situation when a user would like to join a formed team. The Invite model represents the case when a team swipes right on a user (any member of the team can do this). In order to represent a left swipe we use the BlackList model. In figure 4.1 you can see an overview of our database structure.

## 4.2 API

For our project we use a REST API. The Controller class is where we defined our endpoints. The Service class is where we have our main programming logic. The Mod-

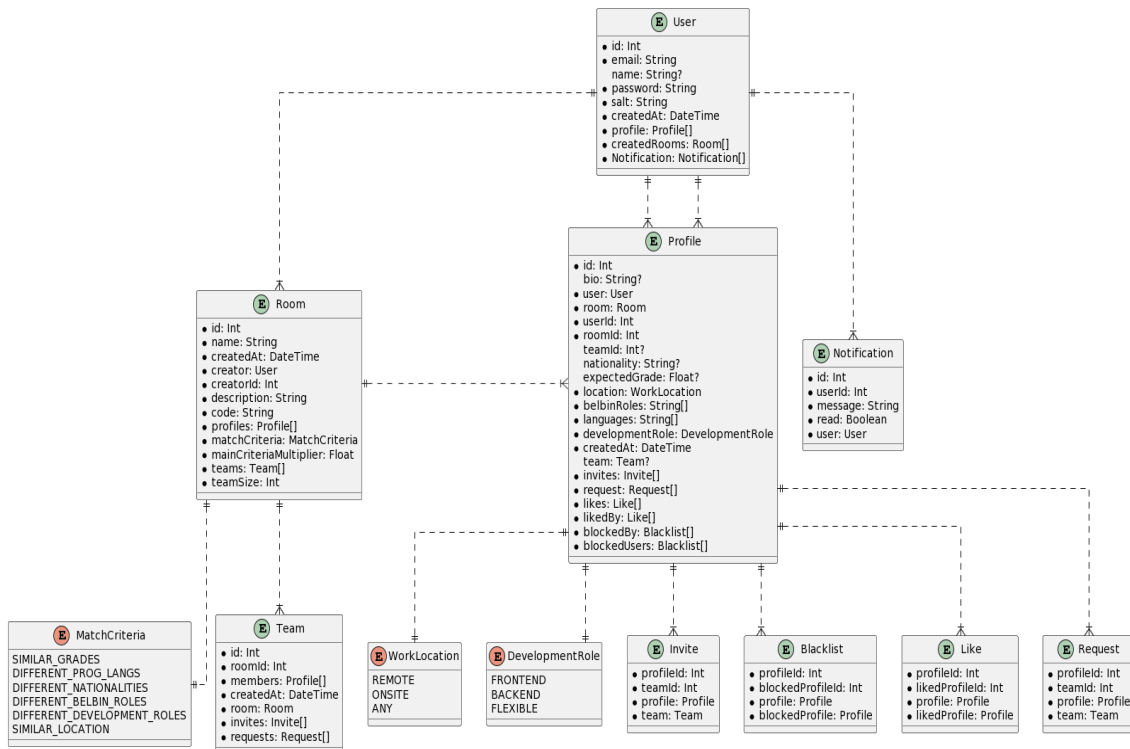


Figure 4.1: Database Schema

ule class is what bundles everything together nicely. This allows us to have a clear structure with a good separation of concerns. We have several separate Controllers for different logical structures. The authentication controller handles, as the name suggests, everything related to user authentication. It includes endpoints for registering, log in and password reset. The Rooms controller handles room manipulations. It allows the user to create a room and returns the generated code this room. A user can also join a created room by providing the code, delete a created room or leave a joined room. It also has additional endpoints for getting all the rooms created by the user and for extracting all the created teams in the database. For the Users we have a controller which handles basic CRUD(create,read,update,delete) database operations. The Matchmakers controller has two of the most important endpoints in the application. The first endpoint is a get request which returns the best possible match for a user. The second endpoint is a post request and represents the response of the user regarding the match that was provided. It can be either an accept, in case of a right swipe, or a decline, in case of a left swipe. The Notification Controller has one endpoint which is a get request that returns all the unread notifications of a user. Lastly, all the responses to the requests from the front-end are in JSON format, so that it is easily accessible and the data is kept consistent throughout the whole application.

### 4.3 Users and profiles

In order to store user data we have two separate data structures. User data stores general user data, while the Profile stores data specific for each matchmaking event.

Therefore, each user has multiple profiles, one per each room. This was necessary to allow a user to participate in multiple matchmaking events (possibly at the same time). Therefore, a room does not interact with the users themselves, but rather with an abstraction.

## 4.4 Error Handling

Error handling is crucial in order to avoid a crash of the server. We made sure to catch errors in all places where they could have happened, especially in the database queries, since they are very prone to errors. Moreover, we also return the error code so that the front-end knows what has happened and can display the appropriate message to the user.

## 4.5 Matchmaker workflow

### 4.5.1 Initial implementation

The primary focus of our initial implementation was to provide a tool similar to the canvas groups page, but with the added feature of being able to recommend teammates both to individuals and teams. What this means in terms of the workflow is that students, after completing their profile, were able to either create a new team or join an existing one. Students that are in a team could browse the profiles of their peers and invite them to join as well. Lastly, our initial implementation offered a recommendation system as well. Students which are in an incomplete team could ask our API for a "teammate suggestion", which would look at all the available students and return the combination of students which, upon being added to the team, would increase the team power the most. The team's members could then either invite them or request a new suggestion. Similarly, individuals who are not in a team, could ask our algorithm for a suggested team for them to join.

#### 4.5.1.1 Recommendation algorithm

For performance reason we have decided on implementing the recommendation algorithm through the use of a genetic algorithm. The first step of this algorithm is the selection of the initial population, which can be seen in listing 4.1. In this step, our algorithm takes the list of all available profile (i.e. profiles which are not in a team yet) and randomly shuffles them into small teams, the size of which is the remainder that the requesting team needs to be complete. For example, if a team has three out of five members when requesting a suggestion, the algorithm would shuffle the remaining profiles into teams of two people. Then, each of these newly formed teams would be appended to the initial team, forming multiple variations of a complete team. The newly formed complete teams would be ordered by performance according to their team power. Once each population's score has been determined, the algorithm moves over to the crossover step.

```
1 const population = MatchmakerService.assignRandomTeams(  
    availableTeammates,
```

```
3   recommendedTeamSize - team.length,  
   );
```

Listing 4.1: Selection of the initial population

For the crossover step, every generation, our algorithm takes the two best performing teams, splits them at the crossover point and then merges the obtained slices such that they form new children. Our goal was to ensure that the genetic algorithm returns a good enough recommendation with a quick response time. By only selecting the first two members of the population for this crossover step, we are certain that the algorithm will either always generate better results, or it will terminate early because its not possible to create any new population.

```
private static crossover<T>(
2   population: T[][] ,
   crossoverPoint: number ,
4 ): T[][] {
   if (crossoverPoint > 1 || crossoverPoint < 0) {
6     throw new HttpException(
       'Crossover point must be between 0 and 1',
8     HttpStatus.BAD_REQUEST,
   );
10  }

12  if (population.length < 2) {
     return [];
14  }

16  const newPopulation = [];
   const parent1 = population[0];
18  const parent2 = population[1];
   const crossoverIndex = Math.floor(crossoverPoint * parent1.length);
20  const child1 = [
     ...parent1.slice(0, crossoverIndex),
22  ...parent2.slice(crossoverIndex),
   ];
24  const child2 = [
     ...parent2.slice(0, crossoverIndex),
26  ...parent1.slice(crossoverIndex),
   ];
28  newPopulation.push(child1, child2);
   return newPopulation;
30 }
```

Listing 4.2: Implementation of the selection and crossover steps

As seen in listing 4.3, we have also implemented a mutation step in our implementation of the genetic algorithm. During each iteration of the algorithm, after the crossover step, there is a chance for our algorithm to mutate some of the teams. The mutation acts similarly to the crossover step: for each team, it first selects another member (team) of the population with a similar score, then it chooses a random crossover point and swaps the members which are at that index between the two teams. The resulting (mutated) team will have most of the members of the original team, but with one member being different.

```
private static mutate(population: any[][] , probability = 1) {
2   if (population.length < 2) {
```

```

    return population;
4  }
    const mutations = [];
6  for (let i = 0; i < population.length - 1; i++) {
    if (Math.random() < probability) {
8      const index = Math.floor(Math.random() * population[0].length);
    const mutation = population[0].slice();
10     mutation[index] = population[1][index];
    mutations.push(mutation);
12 }
    }
14 return [...population, ...mutations];
}

```

Listing 4.3: Implementation of the mutation step

#### 4.5.1.2 Findings

During our peer review meetings we have presented our implementation to one of our target audience, other students, and asked them for their views on it. Based on these sessions we came to the conclusion that most students can become unhappy when they are randomly assigned to a team. However, we also received positive feedback regarding the teammate recommendation aspect of our implementation. Therefore, we have decided to pivot to a different implementation, which would increase a user's autonomy when forming a team, while also allowing them to make more informed decision regarding the team's power.

### 4.5.2 Team Power

Our current implementation, however, adopts a different method of forming the teams, opting for a more manual approach on the users' part. Upon joining a room, a student can browse their peers' profiles and swipe left to discard them or right to *like* them. As this swiping process progresses, the students will eventually get grouped into a team, which the teacher can then export. To empower the students in making better decisions when building their team, the matchmaker computes the potential team's power and displays relevant insight for the user to consider. The algorithm considers six criteria when it determines the compatibility of a team: the similarity of their grades and preferred work location, the balance of the teams developer and belbin roles, and the diversity of their nationalities and known programming languages. We compute the power of a team by taking an aggregate of each of these scores, which is then used internally to determine the order in which potential matches are recommended to a user.

#### 4.5.2.1 Similar grades

The first criterion we are consider is the similiarity of the teams grades. Our reasoning for it is that students who expect a similar grade would . To get the grade score, We first compute the average distance of the team from the mean grade (4.1). Then we divide it by 10 — the maximum distance — and subtract it from 1.0 to get the

score as a percentage, as seen in equation 4.2.

$$AverageDistance(T) = \sum_{P \in Players(T)} \frac{|Grade(P) - MeanGrade(T)|}{Size(T)} \quad (4.1)$$

$$GradeScore(T) = 1 - \frac{AverageDistance(T)}{10} \quad (4.2)$$

#### 4.5.2.2 Similar preferred work location

When creating their profile, students must pick one of the ONSITE, REMOTE or ANY locations. As it can be seen in equation 4.3, the work location criterion scores higher when the majority of a team selects the same location out of REMOTE or ONSITE.

$$LocationScore(T) = \frac{|Onsite(T) - Remote(T)|}{Onsite(T) + Remote(T)} \quad (4.3)$$

$Onsite(T)$  – amount of people in team  $T$  which prefer to work on site

$Remote(T)$  – amount of people in team  $T$  which prefer to work remote

#### 4.5.2.3 Balanced developer roles

The developer roles criterion is used to determine the technical power of the team. When creating their profile, users have three options regarding their developer role: BACKEND, FRONTEND and FLEXIBLE. By choosing their developer role, a student shows their preference (or lack thereof) when it comes to the development process of their project. While the BACKEND and FRONTEND options are self-explanatory, the FLEXIBLE role is meant for the students who either don't have a preference about the development domain, or who would prefer to do the tasks which are not programming related. To compute the score of this criterion we use the equation 4.4, where  $Backend(T)$ ,  $Frontend(T)$  and  $Flexible(T)$  represent the amount of members of team  $T$  who chose the respective role. Our goal is to encourage teams with a balanced role distribution, so the function aims for a balance between the BACKEND and FRONTEND roles, using the FLEXIBLE role to fill in for the role with less members.

$$DeveloperRolesScore(T) = 1 - \frac{\max(|Backend(T) - Frontend(T)| - Flexible(T), 0)}{Size(T)} \quad (4.4)$$

#### 4.5.2.4 Balanced belbin roles

Furthermore, we are also encouraging teams to aim for a balanced distribution of belbin roles. We opted for a naive method of computing the belbin role score, as seen below (4.5), where  $UniqueRoles(T)$  represents the size of the set of unique belbin roles of a team. When making their profile, students can choose up to three belbin roles. When computing the score of a team, we simply look at the percentage of the belbin roles filled in a team.

$$BelbinRolesScore(T) = \frac{UniqueRoles(T)}{9} \quad (4.5)$$

#### 4.5.2.5 Diverse nationalities

Next, we are looking at the nationalities in a group to judge their potential power. Specifically, we aim to suggest teams with higher diversity in nationalities. We compute the nationality score according to formula 4.6, by taking the set of unique nationalities and dividing it by the amount of members in the team. Therefore a perfectly heterogeneous team will score a 1.0, which is the maximum possible score.

$$NationalityScore(T) = \frac{UniqueNationalities(T)}{Size(T)} \quad (4.6)$$

#### 4.5.2.6 Diverse programming languages

Just like the nationalities score, our algorithm uses the diversity of known programming language in a team as one of its matching criterion. The score is computed according to formula 4.7, where the  $UniqueLanguages(T)$  function returns the size of the set of unique languages of a team and  $TotalLanguages(T)$  is the total number of known programming languages by the team, including duplicates. Our reasoning for this is that by knowing different programming languages the students will have a more diverse skill set, which in turn would improve their chance of succeeding.

$$ProgrammingLanguageScore(T) = \frac{UniqueLanguages(T)}{TotalLanguages(T)} \quad (4.7)$$

#### 4.5.2.7 Priority matching criterion

When creating a team, teachers must select one of the six matching criteria to have an increased weight over the others. We call it the "priority" criterion. When computing the aggregate team power, the score of the priority criterion will have a weight of 50%, while the other scores will have a weight of 10% each. By implementing the scoring function this way, we allow teachers to select the most important matching criteria on a course-by-course basis.

### 4.5.3 Matching process

As we have mentioned in the database section, we have four main data structures in order to represent the matching process. A like is created when a user which is not in a team swipes right to another user which is not in a team. If the latter user swipes right on the first user as well, then the created like is deleted and a new team is formed. A request is created when a user swipes right on a team, If the team accepts the request, then the user becomes a member of that team and the request is deleted. An invite is created when a team swipes right on a user. If the user swipes right back on the team, then the invite is deleted and the user joins the team. Finally, if in any situation someone swipes left, then a blacklist is created. If a member of a team swipes left on a person, then that person will be in the blacklist of all the members. If a user swipes left on a person, then just that person is blacklisted. If a user swipes left on a team, then he blacklists all the members of that team. An important note is that when someone either joins or leaves a team, all the likes,requests, invites and blacklists of that profile are deleted.

## 4.6 Filters

On the back-end side, we have implemented a filtering feature in the API endpoints responsible for providing the user with a match. The two main categories of filters we have implemented are profile filters, which filter individuals based on the attributes considered by the matching algorithm and the team filters, which filter attributes of a team, such as its size and the members' profiles.

### 4.6.1 Profile filters

Profile filters can be used to filter the profiles. The main purpose of the profile filters is to reduce the pool of profiles considered during the matching process. Specifically, they reduce the size of the set of individual profiles who are not part of any team. Currently, we have implemented six profile filter categories, all of which have a respective matching criteria: the expected grade filter, the work location filter, the belbin roles filter, the developer roles filter, the nationality filter and the programming languages filter.

#### 4.6.1.1 Expected grade filter

The expected grade filter is represented by a range of grades, as defined in Listing 4.4. If a profile's expected grade falls outside the provided range, it would be filtered out of the pool of potential matches.

```
1 type ExpectedGradeFilter = {  
  criteria: 'expectedGrade';  
3   minValue: number;  
   maxValue: number;  
5 };
```

Listing 4.4: Type structure of a filter on the expected grade

#### 4.6.1.2 Work location filter

The location filter acts as a whitelist which includes all profiles with the same preferred working location as the *value* of the filter.

```
1 type LocationFilter = {  
  criteria: 'location';  
3   value: string;  
};
```

Listing 4.5: Type structure of a filter on the location

#### 4.6.1.3 Belbin roles filter

The belbin roles filter can be split into two filter types: exact or loose. In both cases, the filter works by including profiles with belbin roles which match the ones from the *values* of the filter. In the case of *exact* version, the filter triggers if the profile's roles match all the *values*. The *loose* filtering, meanwhile, gets triggered when the any of the profile's roles matches any of the filter's *values*.



```

1 type BelbinRolesFilter = {
  criteria: 'belbinRoles';
3  matching: 'exact' | 'loose';
  values: string[];
5 };

```

Listing 4.6: Type structure of a filter on the belbin roles

#### 4.6.1.4 Developer roles filter

The way in which the developer role filter works is similar to the location filter, except that instead of comparing the profile's preferred work location with the provided *value*, it filters the profile's development role.

```

2 type DeveloperRolesFilter = {
  criteria: 'developerRoles';
  value: string;
4 };

```

Listing 4.7: Type structure of a filter on the developer roles

#### 4.6.1.5 Nationality filter

The nationality filter can act either as a blacklist or whitelist on the profiles' nationalities. If the *include* action is present, the filter will include only the profiles which have any nationality from the *values* list. Otherwise, in the case of an *exclude* action, the filter will include only the profiles whose nationalities do not match any of the ones in the *values* list.

```

2 type NationalityFilter = {
  criteria: 'nationality';
  action: 'include' | 'exclude';
4  values: string[];
};

```

Listing 4.8: Type structure of a filter on the nationality

#### 4.6.1.6 Programming languages filter

Similar to the location and developer roles filters, the programming languages filter acts as a whitelist which only includes the profiles of users who know any of the languages provided in the *values* list.

```

2 type LanguagesFilter = {
  criteria: 'languages';
  values: string[];
4 };

```

Listing 4.9: Type structure of a filter on the programming languages

## 4.6.2 Team Filters

As the name implies, team filters are used to reduce the size of the pool of potential teams a profile might match with. Currently, we only have two categories of team filters: one which filters a team's size and one which filters the team's members.

### 4.6.2.1 Team size filter

The team size filter is represented by a range of numbers. If a team's size falls outside the provided range, it would be filtered out of the pool of potential matches.

```
2 type TeamSizeFilter = {  
3   criteria: 'teamSize';  
4   minValue: number;  
5   maxValue: number;  
6 };
```

Listing 4.10: Type structure of a filter on the team size

### 4.6.2.2 Team member filter

The team member filter reuses the already defined Profile Filters to filter based on the team's current members. Just like the belbin roles filter, this one also has two types: exact or loose. In the case of the *exact* matching, it will only trigger if the provided *filter* matches all the members of the team. On the other hand, a *loose* matching means that the *filter* may trigger for any member of the team.

```
1 type TeamMemberFilter = {  
2   criteria: 'teamMember';  
3   filter: ProfileFilter;  
4   matching: 'exact' | 'loose';  
5 };
```

Listing 4.11: Type structure of a filter on the team's members

## 4.7 Notifications

Notifications are important to keep the user updated with the most important new information, In our case, a user receives a notification is sent when one of a following happens: a user matched with another user and they formed a new team, a new member joined a team and a member left a team. Notifications are sent to all the users for which this information is relevant. Initially we had planned to implement the notifications using web sockets so that there is no need to overload the system with requests, however this turned out to be more complicated then we had originally thought and we did not have enough time to implement it like this. This is why we simply do a request to the database every 3 seconds to check for new notifications and if that is the case, these notifications are displayed to the user.

# Chapter 5

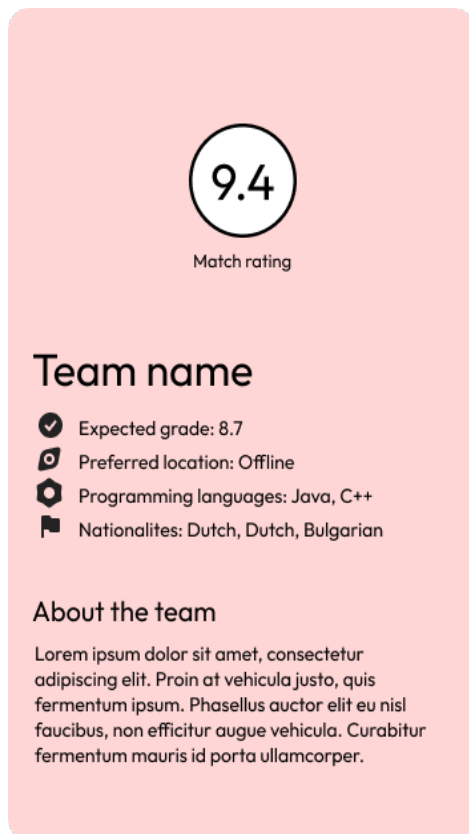
## Frontend Design

For building the application, the VUE framework was used since our team members were interested in learning it. VUE.js is a web-development framework that focuses on component composition. An important part of the design was creating reusable components to ensure good software practices. Precisely, we created the component 'BaseLayout' as a global component that would hold the contents of the application and 'AuthForm' to handle all authentication-related processes. In addition to these global components, a new component was created for each new page in the application. A router is used to link the views in the application to one another. The general design choices that were made in the front had the purpose of improving the user experience. The challenging part in designing a mobile application, in contrast to a regular app, was fitting all the information within the limited bounds of the mobile screen.

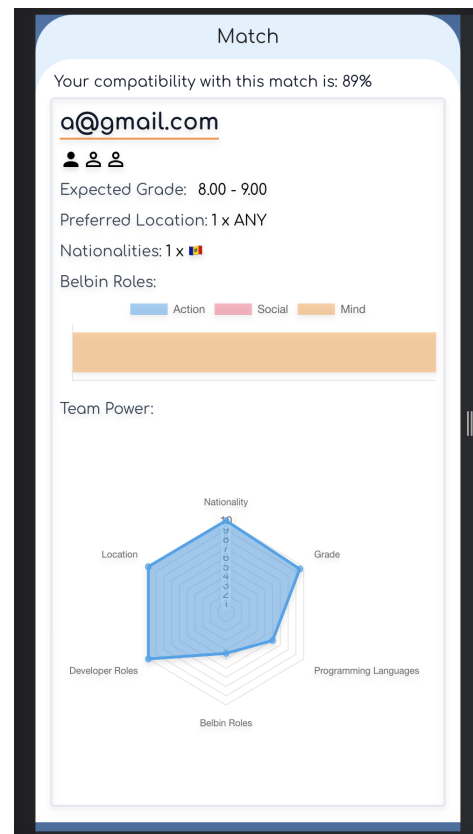
### 5.1 Team/user cards

The sliding cards are an important feature of the application, which was introduced to us by our supervisor. They provide a compact way of displaying relevant information about the user/team and an intuitive way for the user to express their preferences. It also benefits from the familiarity of most people with dating apps, such as Tinder. It makes the flow of the application more fun and 'gamifies' team formation in a way. The card contains a breakdown of the matching information. The 'Expected Grade', 'Preferred Location' and 'Nationalities' have been displayed in a straightforward manner by displaying the information by the side of the category name. For the 'Belbin Roles' a stacked horizontal bar chart was used and for the 'Team power' a radar chart. These kinds of graphs have been chosen to condense as much information as possible in a limited screen space without being redundant. This information is displayed to ensure that the users get the full picture of the score of the team/user they are matched with, rather than it being just a 'black box'. It allows the users to create better-informed decisions and form teams with a higher fitness level based on personal preferences or other criteria. The design of the card went through a series of changes. Initially, there was less focus on the criteria which composed the total score and more focus on the description of the team. As a result of discussions with our supervisor and some usability testing during the reflection meetings, we arrived at the conclusion it is better to fill the card with data related

to the score. In the prototype, there was a lot of unused space and the match rating used too much real estate. There were too few colours used, which we fixed by using a poly-chromatic design to draw the user's attention. In the final version, we used fewer words and more symbols to convey information in order to increase the speed by which the information is processed by the user. We also added icons to indicate the number of people that are part of a team, which is also important information the user should be aware of.



(a) Match card prototype



(b) Match card

Figure 5.1: Prototype and implementation

## 5.2 Authentication

For the login views the design was based on the practices used by existing applications to maximize the user experience. Basic validation is used for forms to minimize the errors that can occur. When the user submits the login form, they receive a token if the email/password combination is correct. The token is set as a variable in the local storage and used as a parameter in any subsequent requests. To guarantee a correct workflow in the application, the router checks the validity of the token before loading a view and redirects the user to the login page if it is not valid with an error being thrown. The logout functionality is currently implemented only locally, meaning that the token is not removed from the database, but a new one is generated when the user tries to log in again.

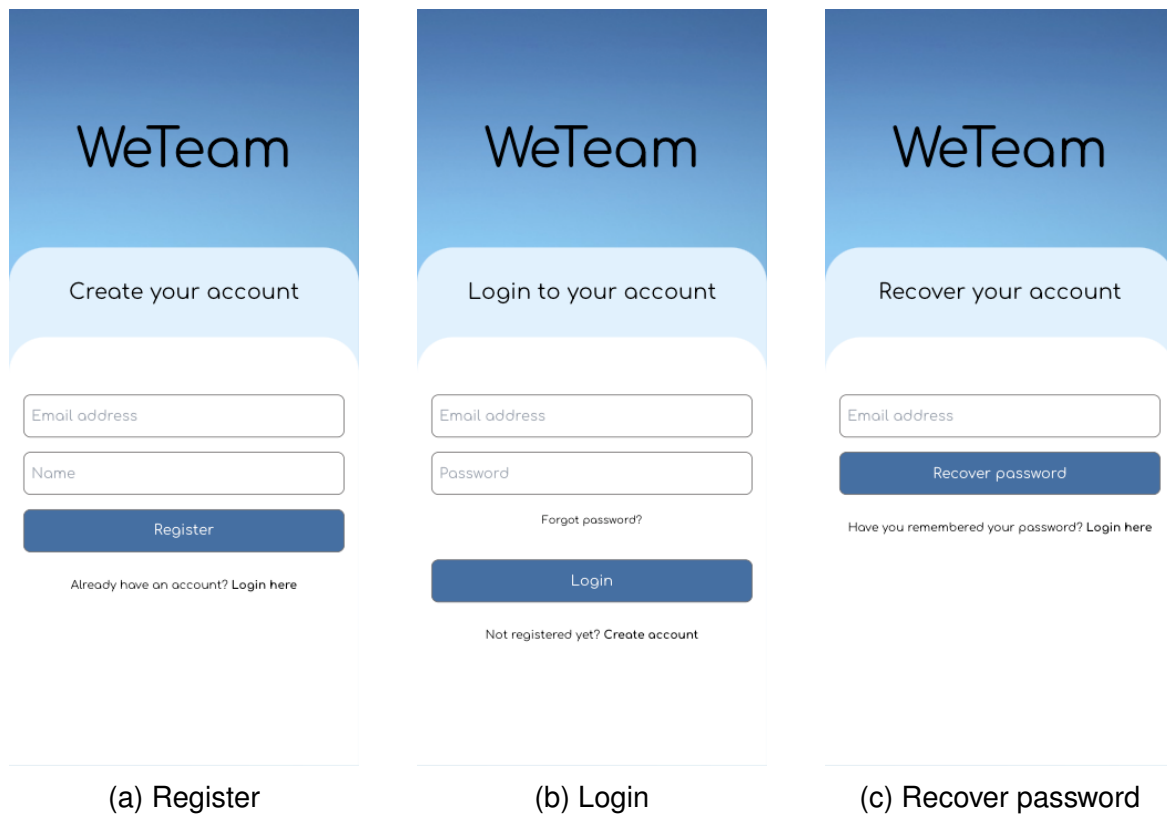


Figure 5.2: Authentication

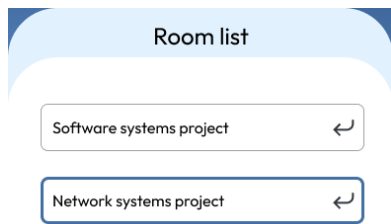
## 5.3 Matching arena

In the room, the user can see both the rooms they created and the ones in which they are a regular user. As a result of usability testing, it has undergone some changes.

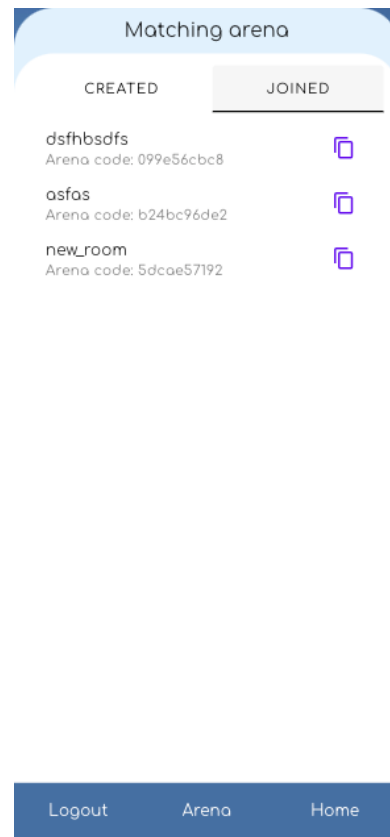
Initially, in the prototype, there was a button to enter the room, which you had to click after selecting the room. It was removed, as it added unnecessary steps, to the process. Now you only have to click the name of the room in order to join it. Initially, there was no way to differentiate between joined and created rooms, so two buttons were added to differentiate the room lists. Additionally, the code of the arenas was added and a button for copying the code to the clipboard, to make sharing the room code more convenient.

## 5.4 Forms

Both the user and the creator form collect information used by the matching algorithm to calculate the score. It uses drop-down menus, for the most part, to ensure that the entire form can fit in a compact way on the screen of a mobile device. For the form component, we used the 'Vuetify' library to improve the application's styling and mobile responsiveness. We use basic validation for the forms to ensure a better application flow.

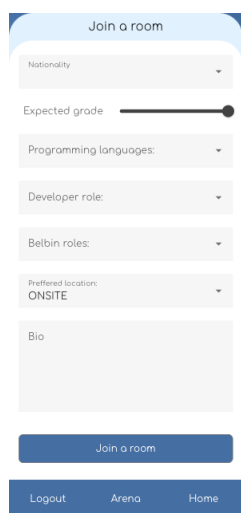
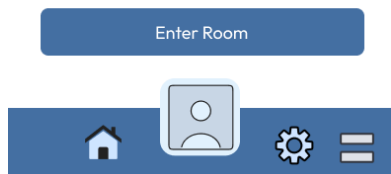


(a) Room list prototype

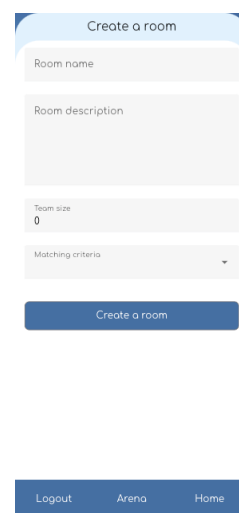


(b) Room list implementation

Figure 5.3: Matching arena



(a) Join Room



(b) Create Room

Figure 5.4: Overview of forms

## 5.5 Creator View

The creator of the room is able to see data about the current status of the team formation process. This allows the manager, to understand whether more time is needed, or if they can already export the teams, and handle the individual students that have not joined a complete team or any team at all. Additionally, they can export the teams to CSV, for further use, in other environments (e.g Canvas).

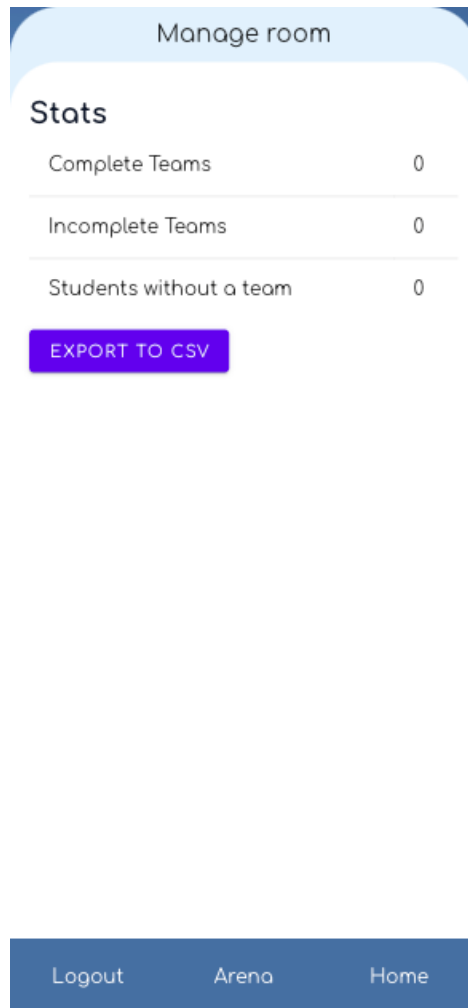


Figure 5.5: Creator view

## 5.6 Team menu

Whenever a user enters a team, the frontend expands and adds two more options for the user: the ability to see his team members and exit the team in case he is not happy by any reason and the ability to accept other peoples request to join the team. We decided to display the team information and the request information in a table since this proved to be the fastest way to display information to users, especially since they are already familiar with the data they are presented (since they had to complete the same data when joining the platform).

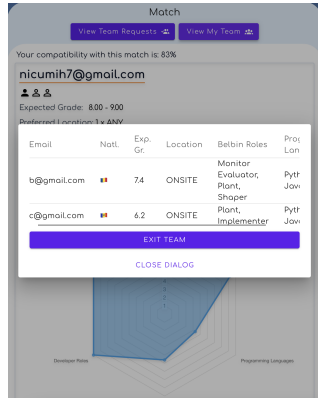


Figure 5.6: Team view

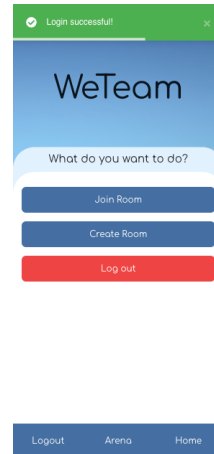


Figure 5.7: Prompt example

## 5.7 Prompts

For usability purposes, we use prompts throughout the workflow of the application. For generating the prompts, we have used the 'vue-toastification' library. It is mostly used in the form completion process to inform the user whether a form has been successfully validated and submitted and as feedback for JSON requests. To avoid being intrusive, they disappear after one second; they can be manually closed and are placed at the top of the page.

## 5.8 Notifications

For notifications we implemented a polling mechanism that retrieves each 3 seconds the new notifications from an endpoint, in order for this solution to scale we would improve it further and use HTTP 2 streams where each connection to the api can stream a bunch of data and poll the api in case of idle users.



# Chapter 6

## Testing

In order to ensure the reliability of our application we have performed rigorous testing throughout the development process. We have tested separately the back-end and the front-end and also made integration tests to see how all the components of the application interact with each other. We have also done several usability testing sessions to receive feedback from potential users of the application. Likewise, for the genetic matchmaking algorithms we also wrote unit tests to see if the results of the algorithm are correct.

### 6.1 Backend testing

#### 6.1.1 Postman

Postman is a powerful tool which allows to write and test APIs. We chose it because it is simple to use and also is very popular, which means that it has a very big community online where we could search for solutions to any problems that we encountered. For our project we used it to test our API endpoints. We made a common workspace for all the members of our team so that everyone has access to the tests. The testing flow was the following. We would login into the application with a dummy account and set the jwt token to the correct one. Then we would test the API endpoints with both correct and incorrect data to see how the application behaves in different situations, especially how it handles data that would create errors in either the database queries or in any back-end logic.

### 6.2 Frontend testing

#### 6.2.1 Manual testing

For the scope of our application, we decided, doing manual tests is good enough. Using a web testing framework, like Selenium would have been superfluous. To guide the testing process, the activity diagrams for the workflow of the application have been used for consistency.

In addition to more general system tests, whenever a new feature was added to the project, we tested its behaviour to work as intended, but also used the developer tools to analyze for possible performance issues (e.g. data leaks). Additionally, we

also tested the interaction with other relevant components to ensure style consistency and to prevent visual bugs or weird interactions between components.

### **6.2.2 Usability testing**

During the retrospective meetings we had and the meetings with our supervisor, we had opportunities to receive feedback from the perspective of the user. During these meetings, we asked our supervisor or our colleagues for feedback. We would then analyze the feedback and implement the relevant changes. Our application as a whole had undergone quite significant changes from a stylistic point of view, however to a lesser extent from a functional point of view.

# Chapter 7

## Future work

Our project, at this point, is functionally complete from the point of view of creating/managing teams. There are ways in which the project could be extended, by adding more functionalities or by improving the user experience and making the general flow of the application more efficient.

### 7.1 Adding filters (frontend)

The filters' functionality is already implemented in the backend as it has been mentioned in a previous section. A drop down menu could be added to the match view, where the user could select additional criteria. This would restrict the number of matches and save the user time, by showing relevant matches only.

### 7.2 Customizable criteria

Currently, the application is designed to be used only in a university environment. There is potential for generalizing the application even further to fit any context which utilizes team formation. Instead of the current fields used in the forms for the room creation/joining, one could create their own parameters which would be split into different types (e.g. quantitative/qualitative, continuous/discrete). Similar genetic algorithms used on the current parameters would be generalized to be used in this case as well. The downfall of this development path is that by making the application general, it starts to miss domain-specific details, which might make it less appealing. Nonetheless, it would be fun to see where you can get with this particular approach.

### 7.3 Chat

When the information on the cards is not enough, there should be a channel of communication where people could discuss with each other, or with other teams. Perhaps they want to make sure they are all on the same page or want some clarification regarding some attributes from the matching card. Currently, communication is only possible, only if you know the email of the user (members of the team/people

who want to join the team). There would be a panel where you could see all the people and the teams in the room, and where you could engage in conversation. Teams would be able to communicate with other users and teams and users with other users and other teams.

## **7.4 Merging teams**

Merging teams would be an improvement to the current workflow. Right now, if we have two different teams of two members, they cannot match. Implementing the feature of merging teams would allow these teams to merge and to create a team of four people. Adjusting the genetic algorithm for this feature would not be a difficult task. This would make the team formation process faster, since teams would not only be limited to user matches but also would be able to match other teams.

## **7.5 General improvements**

### **7.5.1 Styling**

When the application was created, no member of the team had much experience with creating a design in a user-friendly way. This led to some style inconsistencies throughout the application. To fix the inconsistencies a well-defined colour scheme should be created and general design principles should be applied. The responsiveness of the application is not fully consistent on all mobile devices. Some further investigation and improvements should be made on this front as well. At the beginning of the project, we were planning on working on a desktop version as well, but dropped the idea due to limited time constraints. This could also be a path of continuation.

### **7.5.2 Large-scale testing**

Despite the fact that the application has been tested manually on a lower scale, it has not been tested in a real project environment with hundreds of human users. There might be unpredictable bugs that could happen even if they should not from a theoretical point of view.

# Chapter 8

## Reflection

Overall, we are happy with the result of the project. We gathered together as a team, collaborated for the duration of a module and were able to create a functionally interesting working application from scratch. It was a useful experience in improving soft skills, but also technical ones. We were able to learn new technologies and practice already-know concepts. There were however some limitations.

### 8.1 Strengths

#### 8.1.1 effective project management

We were happy with the way in which tasks were split between team members. Everyone had their share of work and was able to carry it out appropriately. We were able to delegate task effectively, which resulted in few conflicts during the development process.

#### 8.1.2 Technical expertise

All the team members have been exposed to web development in the past. This made it easier to communicate with each other and collaborate. Even if we used a new tech stack, that was not familiar to everyone, it was easy to pick it up due to familiarity with similar frameworks and concepts.

### 8.2 Limitations

#### 8.2.1 Time

One of the more important limitations that we were faced with was time. Considering the application there were many features to be implemented and we had to prioritize what was most important in order to create a functional product. At times the progress was blocked by other team members due to the nature of the application, which prevented us from working concurrently. At all points, we tried to think critically and strategically and prioritised functionalities that would improve the product the most.

## **8.2.2 Analysis paralysis**

At the beginning of the project, we spent too much time designing the system without practical implementation. We were trying to figure out all the details in an abstract way. In retrospect, we shouldn't have done this. We would have had more time to work on the project if we just decided to go with the flow and figure out details on the way, rather than to create a theoretically perfect system.

# Chapter 9

## Appendices

### 9.1 Appendix A. Work Distribution

Member	Contributions
Nicolae Mihălache	on the frontend: notifications, teacher views, matching process (cards and sliding), prototypes for usability testing; on the backend: environment setup, authentication, users and profiles modules; final poster; demo during final presentation; initial proposal; section 5 of final report
Nicolae Rusnac	figma prototype; preliminary and final poster; final presentation; maintained the frontend; app layout, authentication, room management, forms, room manager, match card, team views (invite, members), prompts in application; initial proposal; sections 5, 6.2, 7, 8 of final report
Sandu-Victor Mintuș	database design; maintained the backend: notifications, swiping functionality of matchmaker (likes, requests, blacklist and invites); all peer review presentations; initial proposal; final presentation; poster, most of chapter 3.1, 3.2, 4.1-4.4, 4.7, 6.1 and cleanup of final report
Victor Horneț	database design; maintained the backend: team management and filtering modules, most of the matchmaker module (apart from likes and blacklist); most presentations, including final one; preliminary poster (from canvas); initial proposal; sections 3.3, 4.5, 4.6, 9.1 and cleanup of the final report
Frank Bruggink	supervisor meetings; some peer review presentations; final presentation; initial proposal; abstract, sections 1, 2 and 9.2 of the final report

Table 9.1: Work distribution

## 9.2 Appendix B. Development workflow

We distributed work according to the individual preferences and skills of each team member. We used the Belbin roles to better understand each other's strengths and weaknesses in a team. Furthermore, we designated a person responsible for different areas of the project; they acted as a tiebreaker for any decision in their area of interest.

### 9.2.1 Responsibility Areas

Scrum master	New person each sprint
UI/UX	Nicolae Rusnac
Backend logic	Sandu-Victor Mintuș
Frontend design	Nicolae Mihălache
Database	Victor Horneț

### 9.2.2 Role distribution

Nicolae Mihălache	Flexible
Nicolae Rusnac	Frontend Developer
Sandu-Victor Mintuș	Backend Developer
Victor Horneț	Backend Developer
Frank Bruggink	Frontend Developer

## 9.3 Appendix C. GitLab repositories

Here are attached hyperlinks to the project's university GitLab repositories.

- Group page
- Back-end repository (Internal Visibility)
- Front-end repository (Internal Visibility)