Matching People through a Web Application Written in Pure Functional Languages

Internet Technology

Group:
SW707E18

Supervisor:
Florian Lorber

December 19, 2018

Department of Computer Science
Aalborg University
http://cs.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Matching People through a Web Application Written in Pure Functional Languages

Theme:
Internet Technology

Project Period:
Autumn 2018

Group:
SW707E18

Participants:
Anton Christensen
Henrik H. Sørensen
Jacob A. Svenningsen
Jonatan G. Frausing
Kasper D. Bargsteen
Theresa Krogh-Walker

Supervisor:
Florian Lorber

Pages:
50

Date of Completion:
December 19, 2018

Number of Copies:
1

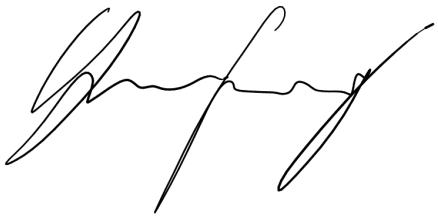
Abstract:

We design and implement a web application with a scalable architecture in the functional programming languages elm and Haskell, that matches users based on how much they agree on a set of statements.
We use collaborative filtering to predict responses to statements, and use correlations between every pair of statements to find compatible users, using data from the People Matching Project. The social platform is established so that people can create an account and respond to statements. We use their responses to calculate a score for how well they match with other users, and give them the possibility of chatting with them.
We test the usability of our system and the efficacy of the matching using anonymous participants. The evaluation is based on their responses to an online questionnaire, where they create users in our web application to interact with the web application.
The conclusion is that we successfully build a web application in the functional paradigm. We are able to predict responses to statements, but have difficulties evaluating whether the final matching is sufficient.

Anton Christensen
achri15@student.aau.dk

Henrik H. Sørensen
hsaren14@student.aau.dk

Jacob A. Svenningsen
jsvenn15@student.aau.dk



Jonatan G. Frausing
jfraus14@student.aau.dk

Kasper D. Bargsteen
kbargs15@student.aau.dk

Theresa Krogh-Walker
tkrogh15@student.aau.dk

Preface

Source Code

The code written in this project can all be found on GitHub at the following location `https://github.com/AAUSoftwareStudentGroup/P7-scalable`. There are multiple branches in the repository, but the final project is found on the *master* branch.

Special thanks

We would like to thank our Web Intelligence lecturer, Peter Dolog, who has been available for questions during the development of our Collaborative Filtering algorithm.

We would also like to thank our Data Intensive Systems lecturer, Christian Thomsen, for a discussion about scalability of MongoDB and PostgreSQL.

Thanks to the volunteers who have contributed as test users.

Finally, we would also like to thank our supervisor, Florian Lorber, for his assistance in the development of this project.

Abbreviations

The following is a list of abbreviations that are used multiple times throughout the report.

Original Phrase	**Abbreviation**
Application Programming Interface	API
Collaborative Filtering	CF
Cascading Style Sheets	CSS
Database Management System	DBMS
Document Store	DS
Document Object Model	DOM
Domain Name System	DNS
Hypertext Markup Language	HTML
Hypertext Transfer Protocol	HTTP
Hypertext Transfer Protocol Secure	HTTPS
Relational Database Management System	RDBMS
Internet Protocol	IP
Mean Squared Error	MSE
People Matching Project	PMP
Programming Paradigms	PP
Uniform Resource Locator	URL

Table 1: Abbreviations Table

Contents

1 Introduction **1**

2 Problem Analysis **2**

2.1 Online Dating Applications . . . 2
2.2 Recommendation Algorithms . . . 3
2.2.1 Collaborative Filtering . . . 3
2.2.2 Content-based filtering . . . 4
2.3 Avoidance of a Cold Start . . . 4
2.4 Scalability . . . 5
2.5 Programming Paradigms . . . 5
2.6 Problem Statement . . . 6

3 Design **7**

3.1 Architecture . . . 7
3.2 Presentation Layer . . . 8
3.2.1 User Interaction . . . 8
3.2.2 User Interface Considerations . . . 9
3.2.3 Site Map . . . 9
3.2.4 Chat Component . . . 10
3.2.5 Survey Component . . . 11
3.3 Business Layer . . . 11
3.3.1 API . . . 11
3.3.2 Matching Process . . . 11
3.3.3 Chat Component . . . 12
3.3.4 Security . . . 13
3.4 Database Layer . . . 13
3.4.1 Database Choices . . . 13
3.4.2 Database Model . . . 15

4 Implementation **17**

4.1 Architecture . . . 17
4.1.1 Docker . . . 17
4.1.2 Caddy . . . 18
4.1.3 Overall Architecture . . . 19
4.2 Presentation Layer . . . 20
4.2.1 Choice of Programming Language . . . 20
4.2.2 Creation of a Front-end with elm . . . 21
4.2.3 Styling with elm and SASS . . . 24
4.3 Business Layer . . . 25

4.3.1 Choice of Programming Language . . . 26
4.3.2 API . . . 27
4.3.3 Matching . . . 28
4.4 Database Layer . . . 33
4.4.1 Database Communication . . . 33
4.4.2 Schemaless Databases and Statically Typed Languages . . . 34

5 Testing **36**

5.1 Matching System . . . 36
5.1.1 Comparison of the Different Methods . . . 37
5.1.2 Results . . . 37
5.2 User Evaluation . . . 39
5.2.1 Results . . . 39
5.3 Testing of Code . . . 40
5.3.1 How to Test . . . 40
5.3.2 Our Tests . . . 41

6 Discussion **43**

6.1 Using Functional Programming . . . 43
6.1.1 Presentation Layer . . . 44
6.1.2 Business Layer . . . 44
6.2 The User Interface . . . 44
6.2.1 User Opinions . . . 46
6.3 Matching . . . 46
6.3.1 Asking the Right Questions . . . 46
6.3.2 Remove Importance of Possible Patterns in CF . . . 47
6.3.3 Possible Cultural Differences . . . 47

7 Conclusion **48**

7.1 Future Work . . . 48
7.1.1 Matching . . . 48
7.1.2 Scalability . . . 49
7.1.3 Extra Functionality for Users . . . 50
7.1.4 Usability Testing . . . 50

Bibliography **51**

Appendices **54**

A Centrality **55**

B Questionnaire **57**

Chapter 1

Introduction

In this project we explore the possibilities of creating a web application that can help its users meet new people. Additionally, we are interested in seeing if languages in the functional programming paradigm can be used for creating such an application. From this brief description of our goals, we formulate an initial problem statement.

Initial Problem Statement

How can we use languages from the functional programming paradigm to create a web application that can match users based on information about them?

In attempting to answer this question, we research several topics of relevance. In relation to matching, this includes an exploration of which services already exist that do something similar. Additionally, we gather information about which methods can be used for matching users and what information we need to obtain from users. When it comes to the functional programming paradigm, we analyse advantages and disadvantages and discuss which languages should be used for the different parts of the application, and how these behave at scale.

Chapter 2

Problem Analysis

In this chapter we discuss the different aspects that are needed for creating an online match-making website. This includes a short introduction to two large dating sites, algorithms for matching users on social media, and how to start up such a site with no initial information.

2.1 Online Dating Applications

In the following sections we analyse two dating services in order to obtain a better understanding of how other people have tried to solved similar problems before.

OKCupid

OKCupid creates matches by asking a user many questions about their preferences. Before a user is allowed to look at other profile, it is mandatory to answer 10 questions that pave the way for potential matches. As a part of their questions, the user has to choose what they would like a prospective partner to have. This can become very time consuming, as having to select your own answers, how much you care about the answer as well as a potential partner's answers. OKCupid gives a user percentage of how much they match with others based on the information that the user has given in the form of details and question answers. As well as these questions there is also a profile page where the user can write text about their likes/dislikes, their occupation and a small self-summary. This text allows a user to know whether the match OKCupid has made makes sense for them without having to actively engage with the other user.

OKCupid uses these questions to perform heuristics in order to find and connect people who share the same interests[1, 2].

Tinder

Another key-player currently on the market is Tinder. This mobile application allows you to find other single people within your geographic vicinity. Getting started with using tinder is easy compared to OKCupid, as the user only has to choose the gender, age range and maximum geographical distance of the people they are interested in[3]. The user is presented with a picture of a potential match and selects if they are interested in the person or not. It requires that both parties accept the connection to be able to communicate.

A study of 395 people between ages 18 and 34 finds that the main reason for people making connections is based on appearance[4]. The same study looks at the reason for people choosing Tinder over other similar applications. The most common reason (48.3% of all users) is that it is a popular application and therefore has a larger selection of users to match with. The next best reason is the ease of use at 14.7%.

Both dating applications have positive aspects, but there are certain aspects that make matches more precise. Tinder does not pay any attention to these, however, it could be argued that this is what makes Tinder so simple and thereby so popular. OKCupid tries to tailor its matches to users, but the way it is done can be very time-consuming and convoluted. Finding something in between where we have a simpler user experience than OKCupid, while maintaining a higher level of accuracy than Tinder is a good start.

2.2 Recommendation Algorithms

In this section we describe methods to suggest matches, similar to how the website OKCupid does it. We explain what collaborative filtering is and how it can be used in different contexts. For our purpose, making predictions based on a user's previous ratings is most relevant.

For a dating website, we expect the user's experience to be better if they expect that the next person they are going to connect with, is someone they will like. If we look at this as another similar problem, an example being a film site, we need to know how to best tailor film recommendations to the specific user.

2.2.1 Collaborative Filtering

Collaborative Filtering (CF) is used to make recommendations based on similarities and dissimilarities between users in a system[5]. The general idea is that if a group of users like the same things as a user u, then u is probably interested in the same things as the group. Table 2.1 shows five users who have each rated up to eight items, where the goal is to predict the unknown ratings. To solve this task, there are two forms of CF, a model-based and a memory-based method. The difference in the two categories is that the model-based approach uses a pre-trained model to do the predictions, where the memory-based approach does not.

	I1	**I2**	**I3**	**I4**	**I5**	**I6**	**I7**	**I8**
User 1	?	5	3	?	2	4	3	5
User 2	1	2	4	4	?	3	2	4
User 3	2	2	4	4	3	4	?	4
User 4	1	3	?	3	1	3	2	3
User 5	3	2	3	2	2	3	?	1

Table 2.1: Example dataset with responses 1 to 5

Memory-Based Methods

A memory-based collaborative filtering approach, relies on representing the entirety of the raw data set in memory. These methods can be further split in to two categories.

User-based CF The idea here is to find users, who have similar patterns to the user u and make recommendations for u by taking the weighted averages of this group of similar peers. This means that if user A and B have a similar way of rating in the past, it is then possible to use A's explicit rating to predict B's rating.

Item-based CF This approach is very similar to the first one. Where user-based CF finds similar users, and recommending items, that they liked, item-based CF finds sets of similar items and finds users that liked these.

Model-Based Methods

This approach uses a pre-trained model, to predict. Examples of these methods could be decision networks, Bayesian methods or latent-factor models[6]. These methods are used for recommender system for some streaming services.

Memory-based methods are simpler to implement than model-based and are fairly intuitive to understand. However, performance is drastically hindered with large data sets, and predictions are not as precise.

2.2.2 Content-based filtering

Another way to predict if a user will like an item, is to evaluate the item's textual content using content-based filtering[7]. This method is mostly used to suggest items, that has descriptions or ratings, which makes it difficult to use in a matching service. The profile description of other uses could be used as the textual content, but requires the users to write descriptive info about themselves.

In this section we have explained some algorithms that generalise and predict unknown information. With enough information about users, these algorithms can be used to make matching recommendations. We now need to know how this information can be acquired.

2.3 Avoidance of a Cold Start

Recommender systems must find a way of avoiding the issue of not having any data to base predictions on, colloquially this is know as the cold start problem. For matching purposes, we need information pertaining to individual user's personality and preferences. In this section we discuss how we can obtain such a data set before having a large user base.

People Matching Project (PMP) have asked almost 10.000 test subjects questions and statements regarding themselves. The test subjects answer in pairs and each have answered the same set of questions with values from 1-5[8]. The project has released answers to 606 questions, along with a correlation matrix describing the correlation between questions and friend pairs. Table 2.2 shows a select few statements from their 606 statements, where the correlation from friend's answers to the same question is shown. The correlation matrix has any combination of questions in the range between -1 and 1. The values describes how likely friends are to give the same answer, where positive value describes similar answers, and negative value describe dissimilar answers.

Due to the amount of data, it makes using their questions a viable option as introductory questions on a dating site for matching purposes. By using the data from the PMP, we can circumvent the issue of a cold start for any matching algorithms.

Question	**Correlation between Friends**
I am a feminist.	0.449
I have smoked a lot of marijuana in my life.	0.443
I support the United States.	0.43
I would be open to dating a transsexual person.	0.416
I like tattoos.	0.387
I use facebook every day.	0.386
I believe in evolution.	0.382
I make jokes about autism.	0.379
I believe in a universal power or God.	0.37
I would not want my children to be gay/lesbian.	0.365

Table 2.2: Shows correlation between friends for different statements

2.4 Scalability

A system is scalable if it is able to continue functioning without noticeable delay under heavy loads or if the system has to operate on increasingly larger datasets[9]. Possible ways of achieving this in a system include designing it in such a way that the load can be split onto several machines when it becomes impossible to add more resources to a single machine. Ideally, most parts of the system should scale sub-linearly in time and space complexity so as to prevent it from becoming necessary to add increasing amounts of resources to keep the system running. This be achieved through the use of caching, compression or similar methods.

2.5 Programming Paradigms

This semester's Programming Paradigms (PP) course has introduced the functional programming paradigm, that distinguishes itself from the imperative and object-oriented paradigms with pure functions and immutable objects. Knowledge of Functional programming provides alternative problem-solving skills that can be used in other paradigms as well[10]. It is used in the industry, an example of this, is how Facebook choose to a use functional programming languages to create their spam and malware filters in a way that performs better than their previous implementation[11, 12]. The nature of pure functions makes concurrent execution simpler, due to the fact that the absence of side effects makes the execution order inconsequential.

The main developer behind the functional programming language, Erlang, once said: *"Your Erlang program should just run N times faster on an N core processor"*[13]. While this statement is an exaggeration, as there is some overhead to creating an arbitrary number of threads, the basic idea that programs written using solely pure functions are much simpler to speed up with concurrency.

In the PP course we are introduced to the languages Scheme and Haskell, that are both functional, but have important dissimilarities. Scheme is in the Lisp family of languages. It is dynamically typed and is a good introduction to many functional programming constructs such as recursion, immutable objects and high order functions. Scheme is, however, not very well supported and it can be difficult to build larger applications[10]. Haskell is statically and strongly typed language and introduces many additional concepts. This makes Haskell very unlikely to have runtime errors since it helps the developer verify the code on compile time, which improves code maintainability

as the code base grows.

To investigate the interesting aspects that this paradigm offers, we choose to build our web service with languages that conform to the functional programming paradigm.

2.6 Problem Statement

Judging from the matchmaking services we have looked at, it seems that simplicity and familiarity are things users value. This presents us with a challenge when asking users to give us many details and answer questions about themselves.

For a website like OKCupid, users have to describe themselves, what they like and what they want their partner to like. However, we want to see if there is a way of creating a system where the user only has to describe themselves and let the system do the matching for them. This means we will focus on creating a matching system with more personal information in the form of questions about themselves, that does not require much extra effort from the user's side.

Since this semester includes the PP course, we also want to implement a matching system in the functional programming paradigm. This does not have an effect on what a user experiences, but we expect it to have an impact on the maintainability of the code base as the web application increases in scale.

Even though we analyse dating websites we do not want to focus on what makes a good romantic match, but rather matching on a platonic level. We make this decision, as we do not wish to focus on what makes a suitable romantic partner, but rather on if we can match people based on their responses to the PMP statements. Therefore, the main focus of this project will be based around the following problem statement:

Problem Statement

How can we create a web application in the functional paradigm that matches people based on their answers to statements from the PMP?

To help answer this problem, we will consider the following research questions:

- What are the advantages and disadvantages of making a web application in the functional programming languages we choose?
- How can we create a reliable matching system, using the data from the People Matching Project?

Chapter 3

Design

In this chapter we explore our choices with regards to the overall architecture of the project. The architecture encapsulates the different stages of this chapter. We explore the choices behind styling for the front-end, as well as the database design and what makes our website able to match users.

The main goal of this project is the creation of a web application which matches people with each other. Additionally, the application should offer a decent user experience. In order to determine which elements should be present in the application, we create a user story, which describes how a user interacts with the application. These features are explained in the following sections, where we begin with the overall architecture of the system.

3.1 Architecture

For this project we use a 3-layered architecture. Using this type of architecture for our solution benefits maintainability as we separate the presentation, functionality and the data, which makes each part more easily replaceable. In Figure 3.1 we have a simplified illustration of our architecture, which is explained more in depth in Chapter 4.

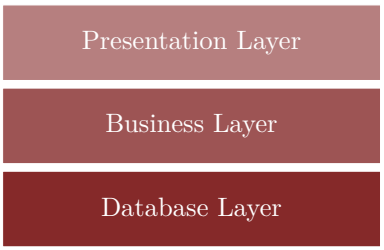


Figure 3.1: The 3-layered architecture

The decision to split the framework into layers is because it provides a clear division of responsibilities. This division enhances the readability of the framework while improving maintainability.

Presentation Layer This layer is what the user sees when using the web application. Therefore, it is important that the user is presented with something that is simple to use and with good usability. The components for this layer are described in Section 3.2.

Business Layer This is the binding link between the database and presentation layers and consists of several components, which is described in Section 3.3.1.

Database Layer This layer is where all our data is stored. An analysis of database types and how we then design the model for our database is described in Section 3.4.

3.2 Presentation Layer

This section describes our considerations regarding the interface our users interact with both in regards to how it looks and how it behaves when interacted with. The presentation layer is made up of several parts. We have the content of the website as the most significant part that leads us to the different components that are needed to make the content work, such as our chat system and how a user will interact with the matching system.

3.2.1 User Interaction

To explain how a user interacts with our web application, we present an example with a user named John. We assume that John wants to meet someone new. He chooses to use our website to do this. We describe John's use of the website through small user stories.

Signing up To do this, he needs to provide his email, username, password, geographical location, birth date, gender, a short text describing who he is, and finally a picture. After submitting this information, John has to answer ten questions. Once this has been done, John has created a user, which he can use to log in to the site with.

Logging in John returns to the website to look at his matches. He clicks on the log in button and types in his details. He now has access to the website and all of his personal information.

Viewing Matches By clicking on the Matches button, John can see a list of users he matches with. John wants to see if these matches can be changed by answering more questions.

Answering Questions As a part of the sign up phase, John has already answered 10 questions, but he wants more specific matches. He clicks on his profile page and clicks on the questions button. John can now answer as many questions as he wants.

Selecting a Match Since John has answered many questions, he feels like he is ready to find a good match. He clicks on the first person in his list of matches and looks at their profile. By looking at the profile, John can decide if he wants to start a conversation or not.

Chatting with a Match John has found Jane who he wants to talk to. He clicks on her profile and then on the chat button. Clicking on the chat button sends him to the messages page where he has a chat connection with Jane.

Mobile Chatting John likes talking to Jane and wants to keep talking to her even though he has to go out. He takes out his phone and accesses the website then goes through the same steps of logging in like with the desktop version. This time, he can go directly to messages to keep talking to Jane.

3.2.2 User Interface Considerations

While usability and user experience are not the main focus points of this project in the sense that we do not spend lots of resources on testing the site's usability, we still choose to spend time on designing a presentation layer that looks similar to existing social media platforms. We feel it is likely that a user will feel comfortable using a website that looks similar to what they may already be using. A polished design gives the impression of a more professional product. A selection of the most important choices relating to the user experience are discussed in the following sections.

Responsive Design

A significant portion of internet traffic in 2018 comes from devices with smaller screens, such as smart phones and tablets[14]. Given the popularity of *Tinder* and other matchmaking services and social media which primarily target smartphone users, it is reasonable to think that many of our prospective users would visit the website from their phones.

Therefore, a design that caters to these screen sizes is paramount for a modern website. We ensure this by changing the size and placement of important elements on the website depending on the width of the viewport.

Material Design

We choose to use Google's design language *Material Design* for several reasons. Simply using an already existing design guide is certainly beneficial, as it means someone has already put a lot of thought into how different components should interact and how to efficiently convey these things to the users. Additionally, using a widely used design language means that more people are already familiar with the way our components are presented. A comprehensive guide to using and implementing *Material Design* is freely available[15]. This guide includes several visual examples of different situations of components. This is very helpful when the experience we have with designing user interfaces is quite limited.

Animations

A part of the *Material Design* guideline we focus specifically on, is making it clear when interaction with a component is possible or when something is actively happening in the background, such as content being loaded. While newer research on the efficacy of the subject is sparse, the fact that modern technology giants such as Google and Apple use this extensively in their product design indicates that it is useful. Furthermore, this means it is a common aspect of web browsing that people are used to.

3.2.3 Site Map

The content that our web application consists of is described in this section. Figure 3.2 shows the sitemap. The following text is a description of each of the pages.

Home, not signed in The first page a user is met with is the front page. It consists of two buttons, one for sign-in and another for sign-up.

Sign up Here a user can create a profile by filling the presented sign up form.

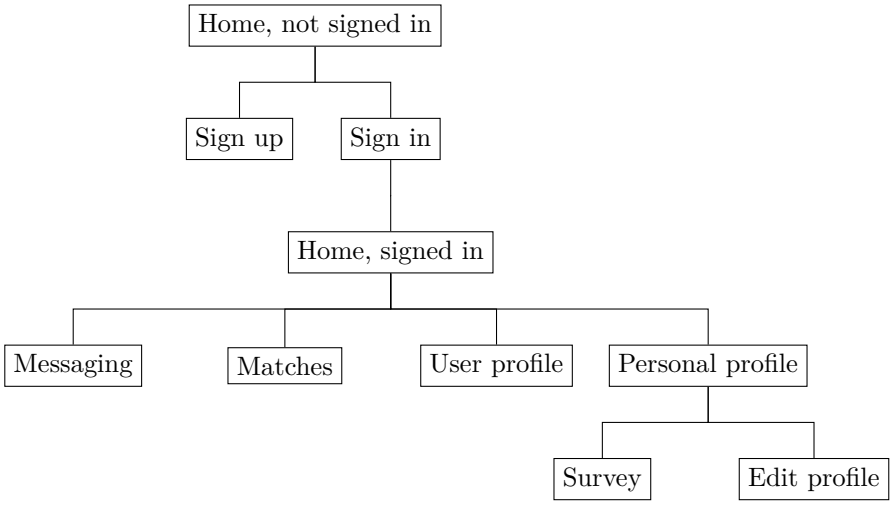


Figure 3.2: The navigation map which shows what pages are reachable from where

Sign in This is how a user enters the application and all of their personal details/messages by entering their log in details.

Home, signed in Here the user can get an overview of new matches and recent new messages in ongoing conversations.

Messaging This page includes a list of every profile the user has chatted with. To continue one of their chats, the user clicks on the name of the person to enter their personal chat history.

Matches Shows an ordered list of profiles based on their matching score calculated from how they answered the survey. To interact with one of these people, the user clicks on the desired profile. This leads them to the user's profile page, where there is an opportunity to start chatting with the person.

User profile This page is individual to each user. Here you can read about the other user, see their publicly available information and click a button to start a conversation with them on the messaging page.

Personal profile This is the logged in user's profile that is shown, along with an option to edit their profile information and an option for the user to go to the Survey page.

Survey This page provides the user with questions and allows the user to give a response from 1 to 5 on how much they agree with the current question.

Edit profile Here the logged in user can edit their information, such as password, bio and image.

3.2.4 Chat Component

A significant part of a website for meeting new people is the ability to communicate with them. Without it, making new friends is challenging. We think it is important that a user can access all past messages that are written between them and their friends, as is common in messaging services.

We want the design of this component to avoid confusing the users as much as possible. In order to accomplish this, it is important that messages sent by themselves and messages sent to them

can be easily distinguished. We draw inspiration from other popular messaging services, such as Facebook's Messenger and texting applications on modern smartphones. They differentiate who said what in a conversation by showing the user's self-authored messages on the right hand side of the screen, while the other person's messages are shown on the left.

3.2.5 Survey Component

As described in Section 2.3, we have obtained a number of questions that the answers of can be used to give an indication of how likely it is for two people to be friends. Since there are hundreds of them, the questions are only shown one at a time, so as to not overwhelm the users. We choose to not create a restriction on how many questions a user can answer if questions remain unanswered, as we think an arbitrary restriction would annoy users more than please them. However, since it would be impossible for us to display meaningful matches to a user who has not answered any questions, we ultimately choose to not display any matches for a user before they have answered at least ten questions. Instead we present them with a message explaining that they need to answer questions for us to find them matches along with a link to the survey page. Additionally, we make it easier to do so, by directing them to the survey as soon as they sign up, and tell them it is possible to take a look at their matches when they have answered ten questions.

3.3 Business Layer

In this section we describe the design of how information is passed between the front and back-end with the use of HTTP requests. There is an overview of how matching is accomplished with the use of CF. To conclude, we describe our chat component and discuss the importance of security and how we take this into account in our system.

3.3.1 API

HTTP requests are used to transmit data between the front-end and back-end through the Application Programming Interface (API) defined on the back-end. The API exposes a number of endpoints, which the front-end can send and retrieve information from. This information can be for a specific user, which is necessary for viewing a user's profile page. Additionally, the information on all the users that the logged in user can match with must be retrievable. We use primarily two types of HTTP requests on these endpoints, POST and GET. POST is used when a request changes the database, and GET is for retrieving data. We define the API as seen in Table 3.1.

3.3.2 Matching Process

Our matching process relies on the data and correlation matrix from PMP. Given a set of users with answers to all 606 questions, we can then produce an ordered list of matches for each user. It is, however, unlikely that users will want to answer all 606 questions before we can present them with viable matches. Instead, we let the user answer a subset of the questions and use CF to predict the rest.

Figure 3.3 shows the matching process in further detail. We are able to train a model for the questions in advance using CF, which lets us provide matches to new users almost instantly. Once a new user has answered enough questions for us to predict the rest, we run the matching algorithm and compare the predicted and actual answers with the rest of the user base. The result being an ordered list of recommended matches for the new user.

Type	Auth?	Address	Purpose
POST	False	/users	Register new user
GET	True	/users/{username}	Get single user
GET	True	/users/{offset}/{limit}	Get subset of users
GET	True	/match/{offset}/{limit}	Get subset of matches
GET	False	/users/exists/{username}	Check if username is available
POST	True	/edit	Edit user
POST	False	/login	Logs user in
POST	True	/logout	Logs user out
POST	True	/message/{username}	Send message to username
GET	True	/messages	Get all conversation previews
GET	True	/messages/{username}/{offset}/{limit}	Get subset of messages with username
GET	True	/questions	Get questions
POST	True	/questions	Submit answer to a question

Table 3.1: API design

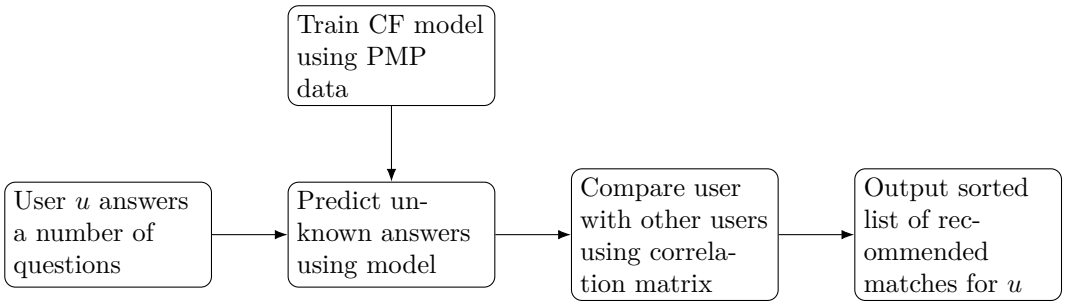


Figure 3.3: Matching process

3.3.3 Chat Component

Our chat service follows the common design patterns of websites like Facebook's Messenger. It has an overview over conversations a given user is part of, uses pagination for the messages, and, naturally, has the ability to send messages to other users. Pagination allows for faster page loads, and thereby a better user experience, since large data collections, such as messages, are split into multiple pieces that can be loaded as they are needed. The functionality is supported by the three `/messages` endpoints in the API, as seen in Table 3.1. The overview is created out of a collection of conversation previews. A conversation preview exist for each conversation a user is part of, and it contains the last message in the conversation with metadata so that we can show a list of a user's conversations. When a specific conversation is chosen, messages from it are fetched from the database in a paginated fashion, meaning that the front-end provides the parameters `limit` and `offset`. Additional messages can be fetched using the same endpoint, and the front-end can then concatenate the results into one long conversation.

3.3.4 Security

Awareness about security on the internet and in general has increased in recent years[16]. For our web application, two primary concerns exist. The transfer of data and storage of user passwords, which is handled by HTTPS and hashing, respectively.

Saving Passwords

When a user signs up they provide a username and password along with information about themselves. Whenever they want access to our service, they must provide the same username and password. We cannot, however, store passwords directly in the database. If someone found a way of accessing the database or back-end, they would have access to every user's password. The consequences of this, could potentially reach well beyond our own web service, as people have a tendency to use the same password on multiple sites[17].

Hashing can solve this problem for us. When a user signs up, we use a cryptographic hashing function on their password before saving it in the database. Any time someone logs in, we run the password they provide through the same hashing function and compare the result to the hash stored in the database.

Given a large enough user base, there are bound to be users who share the same passwords. Since the hashed versions are also equal knowing one user's password could mean knowing other's. In order to avoid this, we generate a random alphanumeric string called a salt when a user signs up. The salt is saved with the user and appended to their password before it is hashed. Whenever they log in, the salt belonging to their username is appended to the password they supply before being hashed. This means that users with the same password will not have the same hash stored in the database.

HTTPS

Whenever a user supplies valid credentials an authentication token is generated and saved on their user in the database. Any end point in the API that requires authentication must include a valid authentication token to succeed. HTTP does not prevent someone from intercepting packets containing this authentication token, or their password when the user signs in, while they are travelling between the user and the server. We use Hypertext Transfer Protocol Secure (HTTPS) to avoid this, as it encrypts any data before it leaves either the user or the server.

3.4 Database Layer

Before we can create a model for our database, we need to consider which type of database is best suited for this project. In the next section, we discuss different systems and how they compare with regards to our project idea. Finally, we propose a design for how we model the database.

3.4.1 Database Choices

The storage and management of data is fundamental for dynamic web applications. Because of this, the choice of database system needs consideration. NoSQL is an alternative to traditional Relational Database Management Systems (RDBMS). A NoSQL database is generally faster to set up, as there is no need to create a schema. RDBMS on the other hand has a lot of functionality that

ensures the database is used correctly. The following sections cover advantages and disadvantages of the two in a potential web service.

RDBMS

Relational Database Management Systems (RDBMS) are used to model relations between objects. In RDBMS, tables are created with a fixed structure with constraints that must be respected. This makes it well suited for data with known attributes that rarely change. An RDBMS is considered very robust, and comes with certain guarantees with regards to the data written and read. This is largely due to transactions following the *ACID* principles[18].

- **A**tomic: Everything inside a transaction is promised to either complete or fail entirely.
- **C**onsistency: A transaction must bring the database from one valid state to another valid state.
- **I**solation: Ensures that a transaction executed concurrently leaves the database in the same state as if it was done sequentially.
- **D**urability: Guarantees that a committed transaction will not be reverted in case of a system failure.

NoSQL

NoSQL is a general term describing several subcategories of databases. The most commonly used is known as Document Store (DS) and is the one we describe in this section. DS replaces the fixed structured entities in RDBMS with flexible documents. Instead of the ACID principles, NoSQL promises eventual consistency with the principles of BASE[19].

- **B**asically **A**vailable: The CAP theorem defines *Availability* as "A promise that a request receives a non-error message with the most recent write"[20]. *Basically* means that in most cases this is true, which is caused by the following two statements.
- **S**oft state: The system can change state without input. This is necessary, such that states can be merged across systems.
- **E**ventual consistency: If no system receives further input, then all systems will eventually be consistent.

RDBMS can be designed in the same way as DS such that joins are rarely needed which gives similar read-times between the two system types. In RDBMS writes must be atomic which means it is necessary to lock all tables involved in the transaction. In the DS NoSQL DBMS, MongoDB, writes are only atomic within a document, which reduces the overhead that is caused by locking[21].

Vertical vs. Horizontal scaling

Scaling vertically means giving the machine more computational power by adding system resources. At some point it becomes difficult to improve the hardware on a single machine. Horizontal scaling combines the computational power of multiple machines in order to further boost performance. When scaling horizontally, multiple machines are connected together and controlled by a central unit which assigns them tasks. Super computers are an example of horizontally-scaled systems.

Comparing RDBMS and NoSQL

RDBMS are suitable when the model is well defined and scales well with reads if joins are avoided. It puts responsibility on the database design since alterations should be avoided, which simultaneously ensures that the database is used correctly. NoSQL can be used regardless of how well defined the model is and scales similar to RDBMS regarding reads. Documents can be dynamically altered which allows greater flexibility, but can also make the model inconsistent. NoSQL has better write scalability because of the performance advantages of horizontal scalability. Since we want to build a scalable web application, we choose to use NoSQL due to aforementioned advantages they bring. Since MongoDB is one of the most popular databases available[22] with rising popularity and has extensive documentation[23], we choose to use MongoDB over other NoSQL solutions. An alternative is Google's Cloud Firestore which is still in beta release[24].

3.4.2 Database Model

As we discuss in the previous section, we need a database system to support our web application. An illustration of how we plan on storing all the information we need, is shown in Figure 3.4.

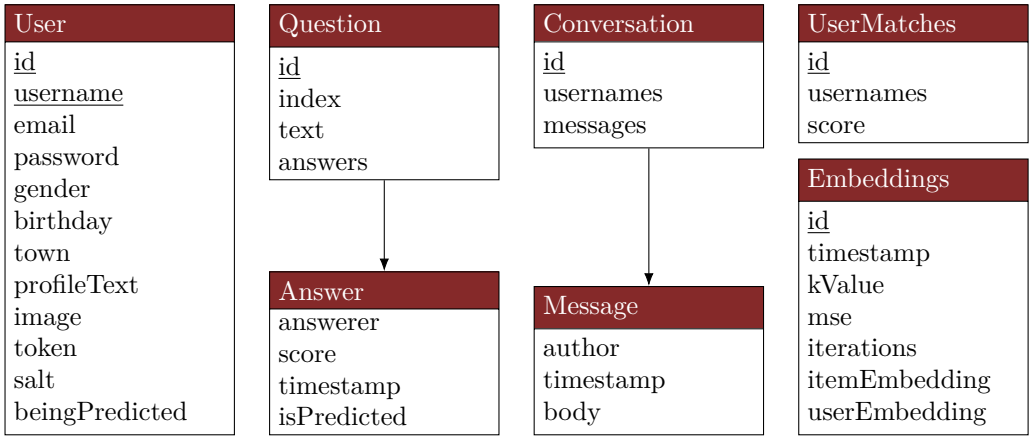


Figure 3.4: Model of all information stored in our database. Field names in plural are arrays and underlined fields have an index on them

It shows the information we have to save in order to provide the complete service we intend on implementing. The boxes without an arrow pointing to them shows our MongoDB collections. The `User` consists of documents describing everything we need to know about users of the system. The `Question` collections contain all questions including both predicted and actual answers given by users. We can tell which answers are real, and which are not, based on the `isPredicted` field for each `Answer` document. This allows us to populate the database with predicted answers based on how others have answered the questions, and how the user has answered other questions.

This allows us to create a single query to update an answer on a question document that will prevent our CF algorithm from changing answers we already know to be facts, and to prevent users from submitting answers to questions they have answered previously. In order to prevent checks here, to determine whether we need to append a new answer to the `answer` array, or change a predicted value, we populate the `Question` collection with dummy answers for a user when the user is created. As these values are being appended to the array while other modifications to this collection are all changing existing embedded documents, it does not matter that the collection is not locked during any of these modifications, as they do not affect each other.

For the `Conversation` collection, new messages are always appended to the array of messages and this is the only field that can be modified once a new conversation document has been inserted into the collection.

The last two collections, `UserMatches` and `Embeddings`, contain the data necessary for our matching system. The `UserMatches` collection contains all match pairs, while the `Embeddings` collection contains models used for our prediction. We choose to save a new `Embedding` document after a number of iterations when we train, as this allows us to stop and continue training later, should we decide to do so. When predicting, we can just choose to use the model with the lowest `mse`.

Summary

This concludes the design chapter. We present the three layers of our architecture: presentation, business and database. For each of these layers, we explain what decisions are taken with regards to the functionality for our web application.

For the presentation layer, we present a sitemap, showing what the user can do, and we discuss which significant components exists. We also mention which design principles we follow.

For the business layer, we present the API which allows the users of our web application to fetch and update information through the presentation layer. We also explain how we plan on predicting answers to questions users have not yet replied to and how we find matches for them. In the last part of this layer, we mention what security practices we choose to follow.

For the final layer, the database layer, we explain differences between RDBMS and NoSQL databases as well as what types of scaling there are. We compare these two forms of systems, and make a choice on which DBMS we choose to use. We end the chapter by designing how our model looks, presenting all the collections that are present and what each of these are used for.

Chapter 4

Implementation

In this chapter we describe how we implement our design choices for our web application. First, we describe how Docker is used to manage a number of containers for the services used in the application. Next, we explore the realisation of our design choices for each layer in our architecture.

4.1 Architecture

In this section we introduce Docker and explain how we use it to separate the services we use in our application. It also includes an exploration of one of these services, Caddy, which is central to our application. Finally, we explain how information flows through our application.

4.1.1 Docker

There are several technologies and software services that we need to develop and run our system. It is necessary for some of these to run on a publicly available server on the world wide web in order to allow users to use our system. Downloading, installing and configuring these services one by one on each computer we develop on is a cumbersome task. Docker is a service that can help with avoiding this[25].

Docker allows us to describe an application stack in the form of a `docker-compose.yml` file. Using a single command, Docker can create small virtual machines called containers for each of the services in the stack. These are connected in a virtual network where they can communicate with each other through specific ports. Additionally, some ports are exposed to the host machine's system.

The goal of using Docker is to ease the development process for each member in the group by providing a consistent environment on each machine. Thereby limiting inconsistencies between code run on different machines, including public servers.

An alternative to Docker is Vagrant, which uses full sized virtual machines instead of small containers[26]. This makes it significantly more resource intensive, as an entire operating system has to run on it. If we had to develop or run our application on a BSD system, then using Vagrant would be more sensible as Docker support on BSD systems is lacking.

The following is a list of the containers we use in Docker and what service they run.

MongoDB In Section 3.4.1 we decide to use MongoDB in the database layer. It is run in this container.

Haskell In Section 4.3.1 we decide to use Haskell in the business layer to serve as the API for our presentation layer as well as the matching algorithm and related systems. This container runs the build system Stack, which in turn runs the Haskell code. In addition, this container runs a script that keeps track of changes to the .hs files and runs the Haskell compiler whenever new changes are saved.

elm In Section 4.2.1 we decide to use elm to implement the front-end of our web service. This container runs a script, which keeps track of changes to .elm files and runs the elm compiler whenever a change occurs.

SASS In Section 4.2.3 we decide to use SASS to help with writing Cascading Style Sheets (CSS) styles. This container runs the SASS compiler which monitors a set of .scss files for changes and compiles them to CSS whenever a change occurs.

Caddy In Section 4.1.2 we decide to use Caddy as a webserver. This container runs the Caddy service.

4.1.2 Caddy

Caddy is a webserver much like Nginx[27, 28]. It has a number of features which are appropriate for smaller projects like ours. These features are described below.

Web server The basic feature of a web server is to return files when a URL is requested which points to that file. This is how we serve the compiled JavaScript application and CSS among other files that make up the front-end of our web application.

SSL Certificates An advantage of using Caddy over other similar services is that it negotiates free TLS certificates from Let's Encrypt as long as it is run on a publicly available domain[27, 29]. Having a TLS certificate means we can serve the web application via HTTPS as further explained in Section 3.3.4.

Reverse Proxy We use Caddy as a reverse proxy for the business layer. A reverse proxy is best described by explaining a regular proxy. A proxy is a server through which multiple clients route their internet traffic in order to hide themselves from the servers. A reverse proxy then, is the hiding of servers from users. How this is used in our application is explained in detail in Section 4.1.3.

Load Balancer Caddy's reverse proxy system can also be used to perform load balancing, meaning that if we have two or more servers running the API, we can split incoming requests between them as shown in Figure 4.1. Caddy offers different methods for deciding which instance of the API a new request should go to.

Nginx can perform the same actions, but in our experience, the configuration of Nginx tends to be more complex when compared to Caddy, and we are not alone in this: "The configuration is not that intuitive and you really need to get into the syntax and concepts to get an understanding of knobs to turn in order to achieve a certain goal. Its also much more fine-grained than necessary for the average user"[30]. Nginx performs significantly better and therefore has the ability to handle more requests at a time[30]. The trade-off we are making then, is between speed and simplicity. For scalability, Nginx is the better choice, but since it is easy to switch out at a later date, we choose to spend our resources elsewhere and go with the simpler setup of Caddy.

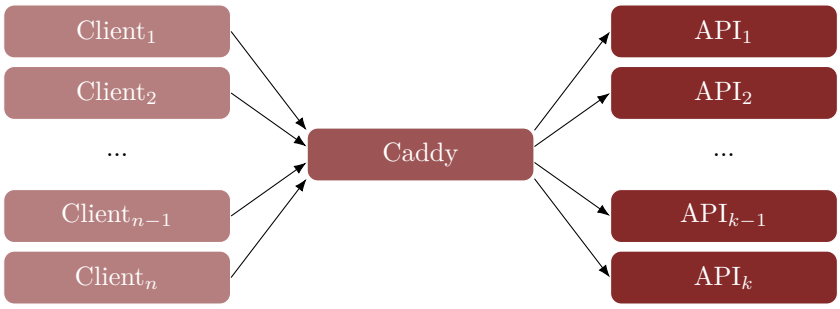


Figure 4.1: A load balancer

4.1.3 Overall Architecture

To show the flow of information through our application, the following example explains how each layer interacts and how Caddy is used in between.

A user visits the hypothetical site `https://friendr.com/login`, by entering the address into their web browser. A DNS lookup results in the URL pointing at an IP where Caddy is located. Caddy listens on ports 80 and 443, the latter of which matches the HTTPS protocol specified in the URL. The server is configured in such a way that if the URL does not match a file on it, request is redirected to the domain name alone, with the path appended as a GET parameter. In this case, the request would be redirected to `https://friendr.com/?path=/login`. This URL does match a file on the server, the `index.html` file. Once the user's browser receives and parses the `index.html` ↪ file, it finds that it needs to make requests for files like `/style/style.css` and `main.js` from `friendr.com`. Since the URL matches actual files on the server, Caddy just returns the files. The `main.js` file, is the front-end code compiled from elm to JavaScript. In this JavaScript file is code that uses the value in the `path` GET parameter to determine what HTML to render and how to handle user generated events on the page. For `/login` it starts by rendering the login page.

The user can now enter their username and password and sign into the web application. When this happens, the front-end code sends a request to `https://api.friendr.com/login` with the specified credentials. As previously mentioned and shown in Figure 4.2, Caddy acts as a reverse proxy, and sends all requests to `api.friendr.com` to the Haskell application through a locally opened port, which it is listening on.

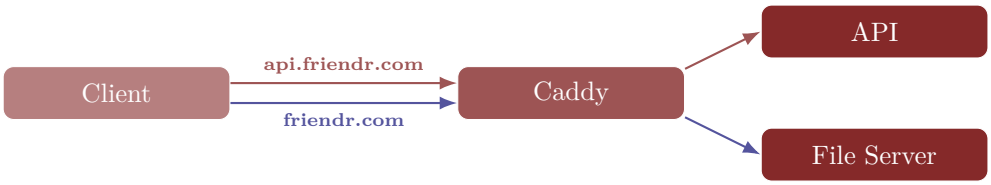


Figure 4.2: An example of how we use Caddy as a reverse proxy

In order to determine if the user can be logged in with the provided credentials, a number of steps have to be completed in the business layer. These steps are explained in detail in Section 3.3.4, but regardless of the authenticity of the credentials, the Haskell application has to communicate with the database layer. This is done through the virtual network created by Docker.

Finally, Haskell sends an answer back to the presentation layer elm application which depending on the result of the log-in, either redirects the user to a logged in page or shows an error.

In this section we have explored the overall architecture from an implementation standpoint. This includes explaining how and why we use Docker and Caddy, while also giving a detailed example of how information flows between the layers of our architecture.

4.2 Presentation Layer

For a user to be able to interact with our business layer and use the features we have implemented, there are certain considerations that need to be taken with regards to language, which we explore here. We begin with a discussion of our choice of language, then show how we use this language to create a user interface and finally how we choose between different approaches to controlling the visuals of the user interface.

4.2.1 Choice of Programming Language

For the implementation of our front-end, we choose the language elm. In accordance with our problem statement, it is a functional programming language. In this section we argue for our choice of elm, by discussing advantages and disadvantages between it and the Lisp-like ClojureScript. Both languages are compiled to JavaScript which allows it to run in most browsers.

The primary source for information about elm comes from the official elm guide. This acts as an introduction to the language and the architecture the language lends itself well to[31]. For ClojureScript the primary source of information is the official website, which contains code examples and a guide for people new to functional programming[32].

The syntax of elm is similar to that of Haskell. With the exception of type classes, their type systems are also quite similar. For example, all functions are automatically curried making partial function application simple to use. This is in contrast to ClojureScript where it is necessary to manually curry a specific number of arguments to allow the partial application functions. The syntax of ClojureScript is very different from elm and much closer to that of Lisp. While some argue that Lisp code can be written succinctly and elegantly, we find the readability of elm better. When working in a team, being able to quickly and easily understand each other's code is fundamental for efficiently working together.

Most new releases of elm break backwards compatibility. This happens because the elm ecosystem is a relatively young one, and several parts of it are still subject to significant changes to their functionality. Using elm then, could be a problem for a large scale project, but less so for us, since we do not plan to update to a new version at a later date. ClojureScript builds on the already established Clojure and is therefore more mature. However, a project such as this is the perfect opportunity to experiment with systems that are not yet mature and test out its capabilities.

elm prescribes a specific architecture that applications written in it should follow. This can be very beneficial for beginners, especially when working in a team as it makes it easier to predict where code with a specific functionality should be located. ClojureScript does not enforce a specific architecture, but rather lets the user choose how they want to accomplish their task. Which approach is best, comes down to what is being developed. A front-end that fits well into the elm architecture benefits from using elm, whereas one that does not will be hindered by it.

Unlike ClojureScript, elm is a typed language, which reduces the number of runtime errors that can occur. Additionally, it also disallows partial functions, which reduces this number further. This is a positive attribute for a front-end system to have, as unhandled errors should never reach the users of the system.

4.2.2 Creation of a Front-end with elm

In the following sections we discuss how we create the front-end of our web application. This entails an overview of the elm architecture and how we fit our application into it at two different levels. Additionally, we detail how we use elm's JavaScript interoperability in order to save data in the user's local storage.

The elm Architecture

As mentioned in Section 4.2.1, a significant portion of the elm guide is focused on introducing the elm architecture. At its most basic, the architecture can be described as the pattern of separating an elm application into three parts. The *model* is a data type which contains the state of the application, *update* is a function that details how to update the model, and finally the *view*, a function which generates HTML based on the model.

Additionally, the elm architecture is also heavily reliant on its Runtime System. The Runtime System is how elm handles the fact that writing a meaningful application with a user interface that uses nothing but pure functions is impossible: you need side-effects if you wish to show the user something that changes depending on their input. In functional programming, the common solution to this problem is to be wary whenever impure functions are needed, and generally try to keep them as separated from the pure functions as possible. Examples of these impure functions include manipulating the Document Object Model (DOM), generating random numbers and sending HTTP requests to servers. The separation between impure functions in the Runtime System and the pure functions from the developers is advantageous for two reasons: the impure code can be thoroughly tested and even proven to work by the compiler writers; the developers using elm can use a clean abstraction over side effects with strong guarantees of success.

The following listings show how we use the architecture in practice, by showing the individual parts of one of the smaller pages in our application, the login page. Listing 4.1 shows the model and relevant types. The model is simply a record that contains data relevant for the current page. As an example, the username and password contain whatever the user has typed in the username and password fields respectively. The type alias `Error` and data type `FormField` are used for validation of the input from users. The validation is rudimentary and simply tests if any data has been entered.

```
1  type alias Model =
2      { session   : Session
3      , title     : String
4      , errors    : List (Error)
5      , attemptedSubmission: Bool
6      , username  : String
7      , password  : String
8      }
9
10 type alias Error =
11     ( FormField, String )
12
13 type FormField
14     = Username
15     | Password
```

Listing 4.1: The model used for the log in page

The `Msg` data type is shown in Listing 4.2. It details the different kinds of updates that can occur on the login page. The first, `FormFieldChanged` is used whenever any of the input fields are changed. When this happens, the update function shown in Listing 4.3 is called and the model is updated with the new values and any errors that might arise from said values.

```
type Msg
  = FormFieldChanged FormField String
  | Submitted
  | HandleUserLogin (Result Http.Error UserInfo)
```

Listing 4.2: The data type Msg, which describes which kinds of updates can occur on the login page

```
update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
  case msg of
    FormFieldChanged field value ->
      let
        newModel = setField model field value
      in
      case Validate.validate modelValidator newModel of
        Ok validForm ->
          ( { newModel | errors = [] }
          , Cmd.none
          )
        Err errors ->
          ( { newModel | errors = errors }
          , Cmd.none
          )
    [...]
```

Listing 4.3: The update function on the login page for FormFieldChange

The second type of `Msg` is sent when the user submits the form. What happens in the update function when a message of this type occurs, is documented in Listing 4.4. The update function validates the username and password stored in the model. If these are both valid, a command is sent telling the Runtime System to send an HTTP request to the API with the valid credentials. The command also specifies that the update function should be called with the message `HandleUserLogin` when the HTTP request is resolved.

```
    Submitted ->
      case Validate.validate modelValidator model of
        Ok validForm ->
          ( { model | errors = [] }
          , sendLogin HandleUserLogin <| credsFromValidForm validForm
          )
        Err errors ->
          ( { model | errors = errors, attemptedSubmission = True }
          , Cmd.none
          )
```

Listing 4.4: The update function on the login page for Submitted

Listing 4.5 shows the case in the update function where the message has the value `HandleUserLogin`. As can be seen in Listing 4.2, this data constructor contains data of type `Result Http.Error UserInfo`. We use pattern matching to act differently depending on whether an error occurs or not. If the request succeeds, the authentication token and username returned from the API is used to send a command that logs the user in by saving their information to local storage. How this is accomplished is described later in this section. Otherwise a function handles the error by showing the user a notification which describes the problem.

```
    HandleUserLogin result ->
      case result of
        Ok userInfo ->
          ( model
          , Session.login userInfo
          )

        Err errResponse ->
          ( handleLoginError model errResponse
          , Cmd.none
          )
```

Listing 4.5: The update function on the login page for HandleUserLogin

While login is one of the simpler pages in our front-end, it is a good example of how the elm architecture is used in practice. All pages use the same general pattern, but the complexity increases when we introduce navigation between the pages, as discussed next.

The elm Architecture on a Larger Scale

In order to prevent sending new HTTP requests and loading elm in its entirety whenever users navigate between them, we create a Main elm application. The basic idea we follow is to let each individual page have its own *model*, *update* and *view*, while also having a main module with its own *model*, *update* and *view*. This way, the main module also follows the elm architecture. The main model maintains the state of the application as a whole, by keeping track of the currently activated page. The main update has a message type that corresponds to each individual page in the application, such that it can call the update function from the corresponding page. Much in the same way, the view in main module checks to see which page is currently active and calls the

view function from it.

Functionality Outside of elm

When a user logs in to our application, we want them to stay logged in as they navigate between pages. Additionally, we also want their logged-in state to carry over between tabs and windows in the same browser. The proper way to do this, is to use the browser's local storage, which is a storage-area that modern browsers come with. Anything stored here stays until the user decides to clear it, or if we change it. Unfortunately, elm does not offer any support, at the time of writing, for writing or reading data in local storage, which is an otherwise simple JavaScript operation. What elm does offer however, is a way for general JavaScript code and elm to operate together.

elm does this using what it calls ports. Ports specify a well-defined way for data to flow from elm to the outside JavaScript world and back. Any data that goes in to elm is encoded as JSON, which means it can be parsed to make sure it adheres to the format. This allows elm to reason about data coming from an untyped location in a typesafe way, as it simply will not enter elm code successfully if it does not adhere to the type specified for it.

```
1  type alias UserInfo =
2      { authToken   : Token
3      , username    : String
4      , firstLogIn  : Bool
5      }
6
7  [...]
8
9  port storeUserInfo : Maybe UserInfo -> Cmd msg
10
11 port onUserInfoChange : (Maybe UserInfo -> msg) -> Sub msg
```

Listing 4.6: The two ports used to save information about users to local storage

Listing 4.6 shows how we specify two ports related to local storage. The first is used to store information while the second is used to read. Notice how these continue to follow the elm architecture: storing user information through `storeUserInfo` is a command, which indicates there is a side effect related to it. Additionally, it shows that it is an outgoing port, which is in contrast to `onUserInfoChange`. Whenever JavaScript code sends data through it, elm can react to it, by modules subscribing to the port.

4.2.3 Styling with elm and SASS

elm does not define a specific way of generating HTML and CSS for a website. Instead, there are multiple libraries available to accomplish this. These generally mimic the structure of HTML in order to create the DOM while enforcing the rigorous type system of elm.

The following section details our experiences with two different approaches: HTML with CSS and elm-ui for creating the visual representation of our website, and based on our experience of using both, we ultimately make a choice on which of these two to use.

elm-ui, claims it is "a complete alternative to HTML and CSS"[33]. The library comes with functions for creating simple structural elements, such as paragraphs and form inputs. Most of the visible structure is built using the two functions *row* and *column*. Both functions take a list of child

elements as an argument and automatically visually arrange these children in rows and columns respectively. The idea is that this can greatly simplify the job of setting up a website. However, as the complexity of our website grows, this approach loses its appeal. Specifically, we can point to three distinct problems with using elm-ui.

Spacing Firstly, elm-ui replaces the job that is usually carried out by the CSS property *margin* with a function called *spacing*. Where *margin* works directly on the element it is applied to by creating a margin around it, *spacing* creates a margin around all child elements of the object it is applied to. This means we often find ourselves wrapping the elements we want to show in other elements, in order to position them correctly on the page. In practice, this makes it more difficult to create a general structure for each individual page and makes the code describing the DOM less readable.

Limited functions As elm-ui does not contain a full mapping of CSS properties to elm functions, there are certain styling options that are unavailable. An example of this, is transitions and animations, which as mentioned in Section 3.2.2 play a big role in enhancing the user experience of an application. elm-ui also does not contain the functionality for generating HTML forms, which are generally used whenever input from the user is needed. While it is possible to use input fields outside of forms, this foregoes some of the built-in functionality of modern browsers.

Responsiveness Finally, elm-ui does not seem to offer a reasonable way of handling responsiveness of our website, which is an important aspect of a modern web application, as mentioned in Section 3.2.2. elm itself allows us to subscribe to changes in screen size, which means we have the ability to serve different DOMs to users with different screen sizes. However, this effectively means creating all elements twice, which compared to the relative ease of using media queries in CSS, is a lot of work, which will certainly lead to a less maintainable code base.

Ultimately, we choose to not use elm-ui, but rather use the standard elm library *html*, which allows us to build the DOM using functions that directly map HTML tags available in HTML5 and applying classes to the elements, which are then styled in CSS. We choose not to use a library such as *elm-css* which could provide us with typesafe CSS[34]. We do not believe that type errors are a common source of mistakes in CSS, as the browser will highlight invalid CSS cases regardless. Instead we choose to use CSS through SASS, a language and accompanying compiler, which can simplify writing CSS for larger websites, by allowing for easier organising of CSS rules.

In this section we covered how we use elm to create the front-end of our application. This includes reasoning for why we choose elm, how we organise the application using the elm architecture, the usage of ports to make use of general JavaScript code in conjunction with elm. Finally we discuss different approaches to creating the visual part of the front-end and why we make the final choice we do.

4.3 Business Layer

This section describes how we implement the business layer. Firstly, we argue for our choice of programming language. We then describe how we create our API in Haskell, specifically by demonstrating how an endpoint and authentication works. Furthermore, we give a brief overview of the theory behind CF followed by an explanation of how it is implemented. Finally, we show how we use the results of CF to give pairs of users a score describing how good of a match they are.

4.3.1 Choice of Programming Language

In this section we explore our choice of programming language for the business layer of our web application. After comparing it to a few other functional programming languages, we end up using Haskell as the implementation language of our business layer.

As stated in our problem statement, we implement our system using the functional paradigm. There are a large variety of functional languages to choose from, but not all of them are of equal interest. First and foremost we want a language that is statically typed, as having the compiler helps us highlight type errors is a great help for keeping the code base maintainable. Additionally, the language needs to have sufficient libraries and frameworks for our needs. Clojure and Erlang would be two good candidates if not for the fact that neither includes a static type system. Clojure is in the Lisp language family, and we learn Scheme in the PP course, a dynamically typed language. Erlang is good for scalable web servers, but it is also dynamically typed. Both of these languages are multi-paradigm languages.

We choose to use Haskell because it include all these features. Additionally, according to the official website, it is a purely functional programming language with a strong and static type system which supports type inference. It is also lazily evaluated, easy to parallelise and has a vast number of libraries[35]. The following is a description of each feature.

Purely functional In Haskell there is a clear, compiler enforced, type level distinction between pure and impure code via the IO type. This has consequences for both the users of the language and the compiler's ability to optimise the code. For users, it is easy to reason about pure functions and their outputs, because the output *only* relies on the inputs. In languages where side-effects can occur anywhere, the compiler is not able to remove unused computational branches because their execution might alter the final result. The Haskell compiler has this possibility and uses it to optimise the code.

Laziness One of the most unique features of Haskell is laziness. In Haskell values are not computed until they are needed, enabling infinite lists and more, which aligns well with the declarative property.

Strong and Static Typing Haskell's type system is strongly typed, meaning that no implicit coercions can occur, and the possibility for a higher degree of safety exists. A catastrophic example of problems with coercions is the Ariane 5 rocket which exploded due to a coercion from a 64-bit floating point number to a 16-bit integer[36]. Its static type system checks the type validity of all functions and named values at compile time, which greatly reduces the risk of runtime errors. A language such as C# is also statically typed, but it gives less guarantees about runtime safety due to its inclusion of null and allowance of partial functions.

Type inference Everything in Haskell has a type, but you do not need to specify all of them. The compiler can infer the types for you, and even write them for you, should you want them as a specification for a given function. This can be especially useful when working with a new library, where you are unsure of the exact types of every function.

Concurrency Pure functions are by their very nature able to be executed concurrently, and Haskell supports concurrency with a number of libraries and easy to use primitives for concurrency[35].

Libraries There are free libraries available for Haskell on Hackage[35]. Including frameworks for creating RESTful API's and dealing with databases, which is one thing we need. A tool for finding libraries and functions is Hayoo, which is a Haskell-specific search engine, where you can find functions based on their names and types[37].

While each language feature is useful on its own, their interaction with each other is even more so. What you get is an expressive and type-safe language, which can be succinct and fast. In comparison with the other three languages, Scheme, Clojure and Erlang, Haskell is our choice.

4.3.2 API

In this section we explore the implementation of our API in Haskell, by describing how we have used Servant for its creation. Additionally, we clarify how authentication is handled in the business layer.

There are several viable frameworks in Haskell for creating and serving an API, such as Scotty, Yesod, and Servant[38, 39, 40]. For our API, and code in general, we want a high degree of type safety, avoid boilerplate code, and have code which is easy to maintain. Servant was built out of frustration with the mentioned existing frameworks and solutions to meet a superset of these requirements[41]. For this reason, we choose to use Servant.

An API created with Servant is made up of three primary parts:

- A specification of the API's endpoints, specified as types.
- A set of handler functions, each corresponding to a single endpoint, actualising the specified functionality
- A serving function, which combines the handlers with the specification and serves the API

The idea of specifying the endpoints as types is the primary difference between Servant and other frameworks, and while its underlying implementation uses advanced type level programming, it is simple to use. In Servant, each handler function returns the generic `Handler` type parametised with the type of a successful request. Listing 4.7 shows an example of a handler function, specifically the one for fetching a specific user is shown. Like many of the endpoints in our API, this one requires an authenticated user. Servant provides a solution for authentication checks. It requires specifying the type of authentication data, how to find and verify it, and what a successful authentication returns. We look for an authentication token stored in a specific header on incoming requests. The token is verified by checking if a user with that token exists in the database. If a user is found, their username is returned, else a response with status code 401 (unauthenticated) is returned. The username returned is then sent as the first parameter to the handler function along with the connection information to the database. For this handler, the user who made the request is not needed in order to respond, so the parameter is ignored. Rather, the handler attempts to fetch the requested user from the database. If found, it is returned, otherwise a response with error code 404 (not found), is returned.

```
1 fetchUserHandler :: MongoInfo -> Username -> Username -> Handler
    ↪ UserDTO
2 fetchUserHandler mongoInfo _ username = do
3   maybeUser <- liftIO $ DB.fetchUser mongoInfo username
4   case maybeUser of
5     Just user -> return user
6     Nothing -> Handler $ throwE $ err404 { errBody = "The user does not
        ↪  exist"}
```

Listing 4.7: fetchUserHandler function which either returns a user or an error in case the user is not found

In this section we have described how we use Servant to define our API in Haskell. We explain how authentication works in the API and give an example of a single handler function. The remaining handler functions are defined in a similar manner, only following their design specification from Section 3.3.1.

One endpoint of particular interest is the one fetching matches for a particular user, which we present in the next section.

4.3.3 Matching

We discuss the matching process shown in Figure 3.3, where we now go into detail for the implementation of the CF algorithm. In this section we describe matching, how we predict users' answers based on previous information and how we compare users with each other is also explained.

Collaborative Filtering

In this section we describe the theory for CF followed by our implementation. We use CF to predict what users would answer, such that they would not have to answer every question before we are able to find matches.

Theory

If we assume that there are x users and y questions, we would have a sparse matrix A with the dimensions (x, y), where each cell in this matrix represents a users answer of a question. This is called a utility matrix. The goal is to predict the cells that do not currently have information in them. This is done by estimating two matrices U and I (embedding matrices) of sizes (x, k) and (k, y), where k is a constant, and $U \times I$ approximate the known values of A. The estimation of the embedding matrices is done by an iterative update:

$$E = A - UI$$
$$U' = U - \alpha(-EI^T)$$
$$I' = I - \alpha(-A^T E)$$

where α is a small constant, called learning rate. The learning rate defines how much the embedding matrices are updated on each iteration. If it is set too high, then updates may make the embeddings worse, and too low will make the training longer.

The constant k must be set such that the embedding matrices can approximate the known values of A without overfitting. The values for α and k depend on the data set, and is chosen using trial

```
1  train :: Matrix -> EmbeddingPair -> Int -> LearningRate -> Maybe Double
       ↪  -> IO EmbeddingPair
2  train trainingMatrix embeddingPair iterations learningRate prevMSE =
3      [...]
4      train trainingMatrix embeddingPair' (iterations+1) learningRate' (
           ↪ Just trainingMSE)
5
6      where
7        guess = mkGuess embeddingPair
8        error = toTraining getError guess trainingMatrix
9
10       embeddingPair' = updateEmbeddings trainingMatrix False
             ↪ learningRate trainingGuess embeddingPair
11
12       toTraining :: Matrix -> Matrix
13       toTraining = (* targetHasValueMatrix)
14
15       [...]
```

Listing 4.8: Recursive training of embedding matrices

and error. With each update, the state of the model can be evaluated by taking the Mean Squared Error (MSE) of the error matrix, E. MSE is calculated by:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{d}_i - d_i)^2$$

Implementation

Our aim is to have $U \times I = A$, which means that the dimensions of the embeddings U and I must be defined from A. The embeddings are initialised with random values in the range [-1, 1]. After the embeddings have been initialised, we are ready to begin training, as seen in Listing 4.8.

The first step is to evaluate what the model would guess, which is done by $U \times I$ in line 7. After a guess is made, we find the error by taking the difference between our guess and the values in the training matrix. The function `toTraining` in lines 12-13 multiplies its input with `trainingHasValueMatrix`, which is a matrix that describes the values present in `trainingMatrix`. We use this function to remove the values from our error that are unknown in the trainingMatrix, such that these values are not considered. If this step is not included, the model would aim to guess 0 where we have no information. The last step is to update our embeddings before the next step of the recursion. This step is shown in Listing 4.9.

First we update the embedding for the users in line 5. The part '`tr' (mul itemEmb (tr' error))`' calculates the direction that we should update `userEmb`. We multiply this with `learningRate` to find how large the step should be, and finally we can apply it to `userEmb`. In lines 8-10 we update `itemEmb`, which is different in the sense that we must state if we update it. This is done so we can reuse this function when we predict and only want to update the user embedding. When we are training `isPredicting` is of course set to `False`. The tuple at the end of line 2 is the output of the function and consist of the updated embeddings.

During each iteration we can evaluate how low our error has become which is done by taking

```
1  updateEmbeddings :: Matrix -> Bool -> LearningRate -> EmbeddingPair ->
       ↪ EmbeddingPair
2  updateEmbeddings error isPredicting learningRate (userEmb, itemEmb) = (
       ↪ userEmb', itemEmb')
3    where
4      userEmb' :: EmbeddingMatrix
5      userEmb' = userEmb - tr' (mul itemEmb (tr' error)) * learningRate
6
7      itemEmb' :: EmbeddingMatrix
8      itemEmb' = if isPredicting
9                  then itemEmb
10                 else itemEmb - mul (tr' userEmb) error * learningRate
```

Listing 4.9: Update embeddings. `tr'` is the transpose function

the Mean Squared Error. This process enables us to generate two embedding matrices when the product approximates our input matrix. The embeddings are saved, along with the k-value and iteration-count, once every 100 iterations, such that we do not need to specify a condition to decide when to stop. We can then later evaluate the result and continue where we stopped.

The next step is the prediction after a user has answered some questions. The answers are represented as a sparse $(1, x)$ matrix. The method to predict this matrix is essentially the same as when we trained the embedding matrices. There are, however, a few differences:

- Instead of creating two embedding matrices, we only create one for the user. The embedding matrix for the questions must be loaded from the pre-trained model.
- Update embeddings is now called with `isPredicting` set to `True`.
- We do not evaluate how well we perform to increase prediction speed.
- Predicting has a fixed number of iterations before stopping.
- Instead of returning the embedding matrices, we now return our last guess, which is our prediction.

Both in training and prediction the learning rate must be set such that we find a good answer as quickly as possible. It can be a difficult process to find a fitting constant for this. Instead we have given the learning rate the ability to change depending on the process since last iteration. This is done by allowing the learning rate to grow when we improve our MSE, and shrink when we take a step that is too large that causes a worse MSE.

One example that we try can be seen in Listing 4.10. Here, we increase if we improve and decrease otherwise. We also make sure that we do not go below our initial learning rate. Another example is where the learning rate is reset to the initial value if we do not improve. The advantages of the one listed in Listing 4.10, is that we can start with a small learning rate, and then it dynamically finds a good fit.

This concludes the prediction for what a user would have answered for the remaining questions. The next step is to compare a user to other users.

```
1 learningRate' = if isSmaller trainingMSE prevMSE
2                 then learningRate + initialLearningRate'
3                 else maxLearningRate initialLearningRate' (learningRate
                     ↪  - initialLearningRate' * 4)
```

Listing 4.10: Updating learning rate

```
1 getScore :: Vector -> Vector -> Matrix -> Double
2 getScore user1 user2 matrix = sumElements scores
3     where
4         ansMat1     = toNByNMatrix user1
5         ansMat2     = tr' (toNByNMatrix user2)
6         difference  = abs (ansMat1 - ansMat2)
7         scores = -(difference - 2) * matrix
```

Listing 4.11: Finding the score between two users

Compute Match Scores

In this section we explain how a user match is evaluated. This can be done by finding similar users using a similarity measure, such as Pearson correlation or Cosine similarity. The data from the PMP provides us with a correlation matrix that describes the importance of each pair of statement. By using this data, we can get a weight for all pairs of statements. In this section we describe how we use this to gain a compatibility score instead of measuring similarity.

The values in the correlation matrix range from -1 to 1. Positive values mean that friends generally have similar answer to the pair of statements. Negative values mean that they give differing answers. The absolute value of the value determines the significance of the statement pair.

Listing 4.11 shows how we choose to calculate the score from two vectors of user answers. In lines four and five we convert the vectors into matrices. An example of this can be seen in Equation 4.1.

$$\begin{aligned} user1 &= \begin{bmatrix}3\\4\\1\end{bmatrix} \rightarrow \begin{bmatrix}3&3&3\\4&4&4\\1&1&1\end{bmatrix} \\ user2 &= \begin{bmatrix}5\\2\\1\end{bmatrix} \rightarrow \begin{bmatrix}5&5&5\\2&2&2\\1&1&1\end{bmatrix} \overset{\text{tr'}}{\rightarrow} \begin{bmatrix}5&2&1\\5&2&1\\5&2&1\end{bmatrix} \end{aligned} \tag{4.1}$$

By transposing the `user2` matrix, we can get the absolute values after subtracting the two matrices. This gives us the difference between each combination of answers. This is done in line 6 in Listing 4.11 and an example is shown in equation 4.2.

$$\begin{bmatrix}3-5&3-2&3-1\\4-5&4-2&4-1\\1-5&1-2&1-1\end{bmatrix} \overset{\text{abs}}{\rightarrow} \begin{bmatrix}2&1&2\\1&2&3\\4&1&0\end{bmatrix} \tag{4.2}$$

The matrix in line 6 `difference` has values with the range [0,4], where lower values mean closer answers. In line 7 this range is converted to [-2,2] and the matrix is element-wise multiplied with the correlation matrix. The computation on the running example is shown in Equation 4.3.

$$-\left(\underbrace{\begin{bmatrix}2 & 1 & 2\\ 1 & 2 & 3\\ 4 & 1 & 0\end{bmatrix}}_{difference} - 2\right) \cdot \underbrace{\begin{bmatrix}.2 & -.2 & .5\\ -.2 & .9 & -.6\\ .5 & -.6 & .5\end{bmatrix}}_{scores} = \begin{bmatrix}0 & -.2 & 0\\ -.2 & 0 & .6\\ -1 & -.6 & 1\end{bmatrix} \tag{4.3}$$

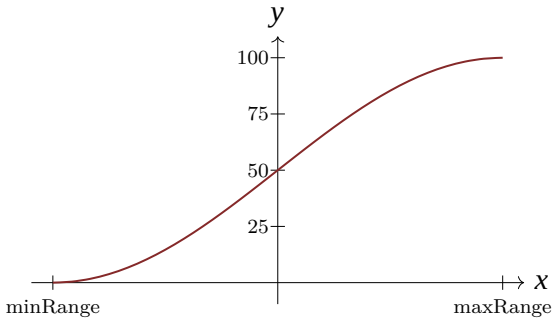
Since similar answers evaluate to a negative value after subtraction, and the correlation matrix has positive values where answers should be similar, then after multiplication, we see that smaller values are better. We negate this answer to follow the intuition that greater values are better. By summing each element in the matrix, we get a single value that describes a score between two users, where higher values are better. In line 2 we return the sum of the elements in the `scores` matrix. The example with `user1` and `user2` would thus give a score of:

$$0 - .2 + 0 - .2 + 0 + .6 - 1 - .6 + 1 = -.4$$

We can evaluate if a score is good or bad by checking if it is a positive or negative score, which makes this a bad match. We also know that higher values are better and can therfore compare scores to each other. We want to be able to know what the best and the worst score is. This is done by simulating a test of two perfect compatible users by:

$$\begin{aligned} maxScore &= \sum 2 \cdot abs(A_i) \\ minScore &= -maxScore \end{aligned} \tag{4.4}$$

What we are essentially doing is saying: The users agree if the correlation value is positive and they completely disagree otherwise. The score for a perfect match is computed to be 1825, but instead of saving a score that goes from -1825 to 1825, we map this to values from 0 to 100. Since the scores naturally will group together in the middle range, we use a function to stretch out these scores. The function does not change the order of matches, but spreads out the values across the whole range, which makes distinguishing between good, neutral and bad matches easier for users. This is done by mapping the range to a half sinus curve, as shown in Figure 4.3, where the x axis is the range [minScore, maxScore], and y is the range [0, 100].



$$f(x) = \frac{\sin\left(\frac{\pi}{\text{maxRange} \cdot 2 \cdot x}\right) + 1}{2} \cdot 100$$

Figure 4.3: Converting the computed score to a score between 0 and 100

```
1 fetchUser :: MongoConf -> Username -> IO (Maybe UserDTO)
2 fetchUser mongoConf username = runAction mongoConf fetchAction
3   where
4     fetchAction :: Action IO (Maybe UserDTO)
5     fetchAction = (fmap . fmap) userEntityToUserDTO (getBy (
        ↪ UniqueUsername username))
```

Listing 4.12: Function that fetches a single user and creates a UserDTO from the fetched user entity

This section has described the process of training a model, that we can use to predict a users unanswered questions. When all unanswered answers have been predicted we create a score using the correlation matrix provided by the PMP.

4.4 Database Layer

In this section, we describe how we realise our design choices relating to the database layer of our web application. We cover how our business layer communicates with the back-end using two libraries. We explain the differences between these two libraries and why we choose to use both of them. We end the section by explaining why we choose to create a schema for a schema-less database.

4.4.1 Database Communication

In this section, we explain how we use the Haskell framework Persistent for the mapping of objects between our business and database layers. Finally, we discuss challenges that can arise when mixing a schema-less database and a statically typed language.

We use the framework Persistent for communicating with and establishing the MongoDB database. It is an Object Relational Mapper for Haskell[42]. We use Persistent to create the schema in MongoDB and generate functions for marshalling and unmarshalling the data in a type-safe manner. Persistent can do this solely from the specification of a record type in Haskell, which means we avoid a lot of boiler-plate code.

An example of how Persistent is used is shown in Listing 4.12. In it, the function `fetchUser` is used. It attempts to fetch a single user from the database using the unique constraint on username with the function `getBy`. Since a requested username might not exist, the function returns a Maybe type. We are not interested in exposing all fields about a user to the API, for example the hashed password, so we extract the needed fields into the data transfer type UserDTO using the function `userEntityToUserDTO`.

It uses `fmap` to map a function on the contents of a structure. In this case we are dealing with a nested structure. We want to keep the structure, but alter the innermost content. This can be done by composing `fmap` with itself. Applying `fmap` composed with itself allows us to do so.

Since Persistent is a database-agnostic framework, it does not provide functionality for all queries possible in MongoDB. An example of this, is the slice operation, which makes it possible to only return a part of an array. Whenever we need functionality such as this, we use a less type-safe library specific to MongoDB, the aptly named mongoDB[43].

```
1  fetchConversationPreviews :: MongoInfo -> Username -> IO [
       ↪ ConversationPreviewDTO]
2  fetchConversationPreviews mongoConf ownUsername = runAction mongoConf
       ↪ fetchAction
3    where
4      fetchAction :: Action IO [ConversationPreviewDTO]
5      fetchAction = do
6        cursor <- Mongo.Query.find
7          ( ( Mongo.Query.select [ "members" =: (ownUsername::Text) ] "
               ↪ conversations")
8            { Mongo.Query.project = [ "messages" =: [ "$slice" =: (-1::
                 ↪ Int) ] ] }
9          )
10       docList <- Mongo.Query.rest cursor
11       return $
12         fmap (conversationEntityToConversationPreviewDTO ownUsername) .
              ↪  rights . fmap Persist.Mongo.docToEntityEither
13         $ docList
```

Listing 4.13: Fetching conversation previews by making use of raw MongoDB queries to only fetch the most recent message from each conversation

An example of this is that when fetching conversation previews, we only wish to return the most recent message sent. This is where we use the mongoDB library to write queries in a syntax similar to the one used in the MongoDB shell. These queries return documents, and as long as they have the exact same structure as the records defined for them, we can use a function from the the persistent-mongoDB library that allows us to convert them to the records we expect. This allows the rest of our code to stay type-safe. The code, which performs this operation can be seen in Listing 4.13.

The function finds all documents in the `conversation` collection where the current user is present in the `members` list. The `$slice` projection operator is used to retrieve the last embedded document in the `messages` array. The resulting documents are mapped to `Conversation` Entities using the `docToEntityEither` function, and finally converted to `ConversationPreviewDTO`s, as used by the front-end.

4.4.2 Schemaless Databases and Statically Typed Languages

While we use Persistent to create the schema for our MongoDB database, there is no notion of migrations built into MongoDB, since it is schema-less. This causes some problems when combined with the strong types of Haskell. This means all documents in a collection need to have the same format, otherwise it is impossible for Haskell to parse the contents of a collection. Whenever changes are made to the schema, we need to migrate the existing data ourselves. If it is simply the addition of a new field, a simple MongoDB query can fix it. Otherwise, we need to create a function in Haskell which maps the old data to the new format and saves it to the database again, before running the servers. Since these migration functions can be written solely using the Persistent library, they are type-safe and we do not run the risk of losing data.

Summary

In this chapter we cover the implementation of our web application as a whole. This includes a general overview of our architecture and how we use Docker to separate and manage the different services needed for the development and deployment of the application. We then explore how each layer in the application is implemented. For each of the two upper layers, this includes the choice of programming languages and how we have used them to implement each layer. The front-end is created using elm and the back-end in Haskell. The business layer also includes the algorithms used for matching users. The theory behind CF is presented along with examples of how we implement it. Finally, we discuss our choice of database and how communication happens between it and the business layer.

Chapter 5

Testing

In this chapter, we discuss the process of testing different parts of our system. We start with how we choose to test our rating system and what we have discovered. For the second part of this chapter we describe the test subjects' general user experience and what they say about our web application. This includes everything from the way it looks to whether it achieves its goal of creating suitable matches. Finally, we discuss testing of code.

5.1 Matching System

To test the efficacy of our matching system as a whole, we come up with a test procedure that allows us to gather data from real users and evaluate the results of our CF and matching on this data. In this section we discuss different methods for collecting the data and evaluating the results we can produce from it. After deciding on a method, we present our findings.

We present three distinct ways of obtaining information and using it with the CF and matching algorithms.

Individual match testing With this method, we collect answers to the statements from the PMP from a number of test subjects and feed these to our algorithm. This produces an ordered list of matches for each test subject. Each test subject must then spend a set amount of time interacting with their top matches. They then give feedback on how well they think they were matched with the person. The feedback can be a rating or something more qualitative such as an interview or a comment. We can then analyse this data and see how many matches rated each other highly.

Pairwise match testing Another option is to have test subjects participating in pairs with a person they have already matched with in real life, either a friend or romantic partner. Each friend gives answers to a number of statements as in the previous method, and specify who they participated with. We do not need further interactions with the test subjects, as we simply look at how well each pair matches.

Individual CF testing To see how well our CF implementation performs alone, we can ask a number of test subjects to supply answers to more statements than the previous two methods. We can then use some of these answers to train a model for the test subject and the remaining to test the accuracy of this model.

5.1.1 Comparison of the Different Methods

Individual match testing This method requires a large amount of test subjects to be available for several hours at a time. If there are too few subjects they are less likely to find appealing matches. Additionally, each matching pair needs to spend a considerable amount of time if they want to get to know each other enough for them to confidently rate the accuracy of the match.

Pairwise match testing While this method requires less time and energy from the test subjects, it might be difficult to find pairs of people who are willing to participate. Luckily, they do not have to do so at the same time. The entire testing can be done in less than 15 minutes.

Individual CF testing While this method tests less of our system, it is also simpler to find participants for, as it is more akin to a regular questionnaire that people are used to answer.

Seeing as **Individual match testing** has high demands for our test subjects, we find it too time consuming to find participants for it. Instead we chose **Pairwise match testing** for testing the efficacy of our matching algorithm as a whole, and ask our participants to answer as many questions as they want, in order to have data for **Individual CF testing**.

5.1.2 Results

The test results are split into two categories. The first category is the collaborative filtering and the second is the complete matching process.

Collaborative Filtering

When training the model, we use 80% of the provided data as training data and the remaining 20% as test data. By doing this we can use the MSE of the test data as a quality measure of the model (Table 5.1). This can be used to evaluate which k-value performs the best.

K	MSE
5	1.597
10	1.526
15	1.225
20	1.560
30	1.683
40	1.840
50	2.022

Table 5.1: MSE for different values of k

The aim is to find a k-value (as mentioned in Section 4.3.3), where the MSE is as low as possible. A too small k-value can not describe the complexity of the problem, and will result in a high MSE. With a too high k-value, we run into the issue of overfitting, which also causes a high MSE. The table shows that if we use a k-value of 15 we are able to create a model that can predict the answers in the data with an error of 1.225.

Of the 18 users that we have gathered answers from, 11 have answered 20 or more questions. In the same way, that we had a training set and a test set to build and evaluate the model, we can test how well we can predict our own users' answers. For these users, we have used 50% of their answers to create a prediction and the rest to evaluate how well we are predicting. The result of this test is shown in Table 5.2.

User	**Qustions answered**	**Training MSE**	**Prediction MSE**
1	439	1.110	1.155
2	286	1.357	1.830
3	111	1.395	1.504
4	73	0.888	1.138
5	73	0.783	1.799
6	72	1.179	1.307
7	68	0.714	1.534
8	52	1.040	1.354
9	41	0.834	1.317
10	20	0.070	1.523
11	20	0.088	1.389

Table 5.2: MSE for training set and predicted values

For the two users that have answered 20 questions, we are able to predict with an MSE around 1.5. The user, that has answered the most questions can we predict with an error of 1.2. The small difference between the prediction errors lets us conclude, that we have good predictions with few questions answered.

Matching Process

Users are asked to contribute with a good friend or romantic partner if possible, which 12 users do. We describe how well we match these users. We expect that each pair of test subjects match decently. Assume that the users u_1 and u_2 are a couple. Ideally no users have a match score that are higher than the one they share, but it is not a certainty.

To evaluate the accuracy of our matching process, we have done the following. For each pair of users, we have created an ordered list of all pairs that include either user in the pair. We then look at how many pairs are better matches than the original pair. The results of this, is shown in Table 5.3. The average can be read as: any given couple on average match worse than 54% of the other users. The second row describes the scores given.

Measure	**Average**	**Worst**	**Best**
Match	0.54	0.88	0.06
Score	74.5	60.9	90.1

Table 5.3: Scores for the complete matching process

These results are not very impressive. There could be many reasons why we are worse than random guessing, which would leave an average match value of 0.5. If we look at the scores gives for all

match combinations, we find that only six matches have a score below 50. What this tells us, is that almost every combinations are somewhat compatible. We need a larger user base to be able to decide the actual performance. The correlation matrix from PMP, that we use to match is based on friends answers, but all six user pairs contributed with their romantic partner. This can also explain why we are getting bad results.

In this section we describe how we test our matching method. We show that we have a model that can predict user answers, but the final matching has some issues. Testing using this method gives, despite many factors that are difficult to quantify, a unified measure of comparison for future implementations.

5.2 User Evaluation

In this section we describe how we evaluate the design of our user interface. Additionally, we describe the results of the evaluation.

As mentioned in Section 5.1, our matching test requires that we collect test subjects' answers to as many statements as possible. While we can certainly create an online questionnaire that includes all 606 questions from the PMP, we find it easier to use our final web application for data collection. This also allows us to receive feedback on the usability of the application as a whole from the participants we already have. We still create an online questionnaire where participants can write their feedback on the usability and visual design of the application. The questionnaire, including the questions asked within can be found in Appendix B. We use the same questionnaire to guide participants through the process of creating a user and supplying answers to statements in the web application. Additionally, for pairs of participants, we use it to record the usernames of both participants.

Generally, our questions are designed to give us a quick overview of how our test subjects feel about some parts of the user interface we imagine could be problematic, mainly the navigation and the design. In order to simplify the procedure for our test subjects, we prefer the use of quantitative questions, but we do include two boxes for additional comments regarding the navigation and design respectively.

5.2.1 Results

Our questionnaire is answered by a total of 20 participants. This is not a large enough sample size to draw statistically relevant conclusions from, but it gives us an idea of potential problems with our user interface.

The few comments we got on the design describe it as 'simple' and 'minimalistic'. Whether this is negative or not can be gleaned from the replies to the question "My overall impression of the website's design was positive". 60% of participants answer agree or strongly agree to this. We also ask the participants which pages in the application they visit, hoping we would be able to draw conclusions about which parts of the application make participants rate the overall design lower. The low number of participants makes this impossible. We can, however, see from the answers that about half the participants spend time exploring multiple pages in the application. The majority of participants show no problems navigating the application and creating a user. In production, however, having 20% of user's complain about the navigation or sign-up procedure of a web application is hardly acceptable.

```
1 main :: IO ()
2 main = hspec $
3   describe "sortBySndDesc" $
4   it "sorts a list of tuples by second element descendingly" $
5     sortBySndDesc [('c', 1), ('a', 3), ('z', 2)]
6     `shouldBe`
7     [('a', 3), ('z', 2), ('c', 1)]
```

Listing 5.1: Specification Testing in Haskell

5.3 Testing of Code

Non-trivial software should be tested in order to verify its correctness. There are several different ways of approaching testing, for example unit testing, specification testing, integration testing and end-to-end testing. Each has its advantages and disadvantages and are most commonly used for a specific subset of the tests needed to be written. One common element of them is, however that they can be run automatically. This gives several benefits, including the ability to do regression testing in which you test whether the addition of a new feature or correction of a bug has broken other pieces of the code[44]. The goal is to have a test suite, which helps to ensure the continued correctness and stability of your program, such that you can alter and improve the code swiftly and with confidence[44]. Some languages need more tests than others due to the nature of the language itself. In languages with some variation of *null* or without types, it is necessary to test for these specific scenarios. We have chosen to write in languages with strong type systems, namely Haskell and elm, and we can thereby avoid having to write a large number of tests that would be necessary in other more commonly used languages, such as C# or JavaScript[45]. But a strong type system cannot catch all logical errors, alas we still need to test our code.

5.3.1 How to Test

Specification Testing

In this section we will primarily look at testing in Haskell. While the specific libraries used for testing elm are different ones, the general approach is the same. In Haskell and elm two types of testing dominate. Specification testing and property testing. Hspec is a library for specification testing in Haskell[46]. The tests you write in it are mostly readable even to non-programmers, which is beneficial for teams consisting of a mixed background. It works by writing assertions about the expected results and then comparing the actual results with it. In Listing 5.1 an example of a test for `sortBySndDesc` is shown.

Property Testing

Property testing is an altogether different way of thinking about tests. Instead of specifying a number of test cases with specific values, you specify properties for your program, which must always hold. It is easier to understand with an example. Assume we want to test the associative property of multiplication. We can test it with a one or more specified cases, for example `7 * 11 = 11 * 7`, but it does not give us a great deal of confidence in its correctness. Instead we define the general property, and let a test framework generate a large number of cases for us. The property is

```
1 testUserCreationAndLogin = describe "users" $
2   it "creates a user" $ do
3     mongoInfo <- DB.fetchMongoInfo
4     createUserDTO <- generate arbitrary :: IO CreateUserDTO
5     eitherLoggedInDTO <- DB.createUser mongoInfo createUserDTO
6     isRight eitherLoggedInDTO `shouldBe` True
```

Listing 5.2: Integration test in Haskell

```
1 prop_fromDenseAndBack :: Matrix -> Bool
2 prop_fromDenseAndBack m = m == (toDense . fromDense) m
```

Listing 5.3: Property testing in Haskell

defined as a function in Haskell `multAssociative x y z = x * (y * z) == (x * y) * z`.

The test framework most commonly used for property testing in Haskell is QuickCheck[47]. QuickCheck provides the aptly named function `quickCheck`, which takes a predicate, generates test cases for it and shows whether all or only some of the cases succeeded. QuickCheck uses the typeclass `Arbitrary`, which each type you want to test must implement. Full control over the value domains exist, but often you just want any value of a given type. In such cases, writing an `Arbitrary` instance for a record type is simply a matter of using the `Arbitrary` instances for its parts. A record with the fields user and password, each of type String, can therefore be generated from two arbitrary strings.

5.3.2 Our Tests

As explained previously, fewer tests are needed in our languages of choice. We would however, have liked to implement more tests than we managed. Especially because Haskell and elm code naturally decomposes into small single purpose functions, which are easy to test. An example of an integration test in Haskell can be seen in Listing 5.2. It uses Hspec in combination with QuickCheck to check whether user creation is possible, and the `Right` option is returned from the `Either type`.

Listing 5.3 shows a perfect example of how to use property testing in our code. The matrix framework we used, HMatrix, had a function, `toDense` which takes a sparse matrix and returns a dense one. HMatrix was, however, lacking the complementary `fromDense`, which we needed and, therefore, wrote ourselves. The composition of `toDense` and `fromDense` should naturally give the original input matrix back, unchanged, which is the property described as a function in Haskell.

Summary

In this chapter, we discuss the different methods of testing. We chose to use pairwise match testing to test our matching algorithm as a whole. We also test our algorithm without user input, and show that we can predict user answers. To include users again, we asked them what they thought of the web application, but due to lack of respondents, it is difficult to make a conclusion. Finally, we covered why fewer tests are needed for languages with strong type systems such as elm and

Haskell. We explain how tests in functional languages can be created, with a focus on Haskell, and show a few examples of our own tests.

Chapter 6

Discussion

The goal of this project is to create a website that matches users based on information they give us about themselves. Before this is possible, we need to analyse what a dating website consists of and what users generally want from such a site. We choose to analyse two dating sites because our original idea was to develop a dating website, which we later change, as this requires much philosophical analysis in which traits make a good couple.

When looking back at this project, it is important to remember that as a part of our problem statement we want to be able to implement our system in the functional paradigm. We implement a system that shows and creates matches between users with two functional languages and within this chapter we overview the difficulties we face with these languages with regards to implementation. Creating matches without any initial data is also a challenge we have to circumvent, so we use the PMP data as a starting point. Finally, we discuss the status of the final result and show the outcome and results of our implementation.

6.1 Using Functional Programming

In the following two sections we discuss the results of using functional programming languages to create both the presentation layer and the business layer of our web service.

An interesting way to look at the advantages and disadvantages of using a functional programming language over an object oriented one, is the expression problem as described by Philip Wadler[48]. This is a description of a problem that is encountered in both programming languages. However, the way in which they look at the problem is different. In object oriented languages, it is easy to extend the code when adding new objects, but less so when adding functionality to existing objects. Here we do not mean objects as in instances of classes from object oriented programming, but instead a concept of the model without the inclusion of functionality. In functional programming languages the opposite is true: adding functionality to an already existing object is easy, but adding new objects is cumbersome. Because of this, one could argue that functional programming languages should be used for applications in which the types stay the same, but functionality is added. Conversely object oriented programming languages should be used for applications where functionality stays the same, but more types are added. Whether we made the right choice is discussed in the following sections.

6.1.1 Presentation Layer

Generally elm is good to work with when creating the presentation layer for our application. The type safety and complete removal of partial functions means the compiler can catch a lot of potential errors early on. Additionally, it provides helpful error messages when it does.

The elm architecture is a good starting point for creating an application, but the complexity increases, when adding more pages to the it. A significant portion of the code that is required to make this work is repetitive, in that the same functionality is required for each page and elm does not offer any language constructs that eases this process. This could improve in future versions of elm, as the newest version, (0.19), is where support for creating multi paged applications managed from one place is added.

It is difficult to argue whether a functional or an object oriented programming language is the right choice for our presentation layer without knowing what direction it takes in the future. During the project however, both functionality and objects are added to the code base at a steady rate, which means either choice is valid. However, if one starts out by designing the application around the objects it will include, it becomes much easier to add functions as it grows, and one could argue that our application contains significantly more functionality than objects, which lends itself well to the functional paradigm. Despite it being less simple to add more types, it is still relatively easy to do because of the excellent compiler errors in elm.

6.1.2 Business Layer

In Section 4.3.1 we describe our reasoning behind choosing Haskell for our business layer.

We use the Servant framework for creating our API. The framework overcomes the expression problem by using type level programming in the specification of the API. This makes it easy to add additional functionality and types without having to alter existing code. We experience no downsides in using Servant. The code is type-safe and succinct.

For communication between the business and database layers, we use the frameworks Persistent and mongoDB. Persistent makes marshalling and unmarshalling data easy, but it does not feature a full mapping of the MongoDB API, which is why we need the mongoDB driver. The mongoDB driver is not as type-safe as Persistent, meaning that, for example, incorrect field names cause problems that are hard to discover. Using the two frameworks in conjunction makes it harder to read the database code, because it requires understanding both.

Using a high level language such as Haskell makes it hard to do low level optimisations, because the exact details of the program are abstracted away. We use the library HMatrix for matrix operations, which relies on heavily optimised C-libraries. While each matrix operation is fast due to HMatrix, we use immutable data structures, which has a large impact on performance. It is possible to use mutable data structures in Haskell, but we choose not to use it, because of inexperience with it.

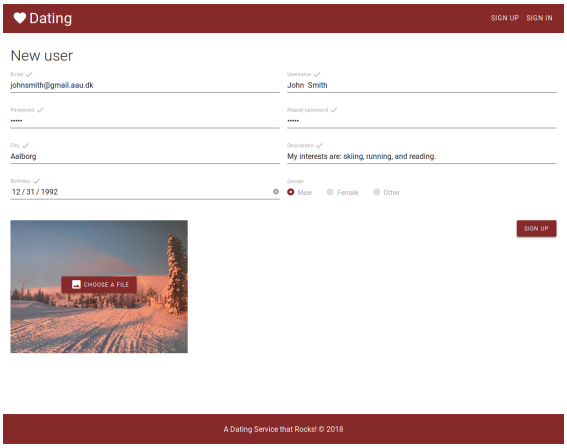
Overall, we are surprised by the succinctness of our Haskell codebase, and the power of the language, enabling us to achieve a lot of functionality with relatively few lines of code.

6.2 The User Interface

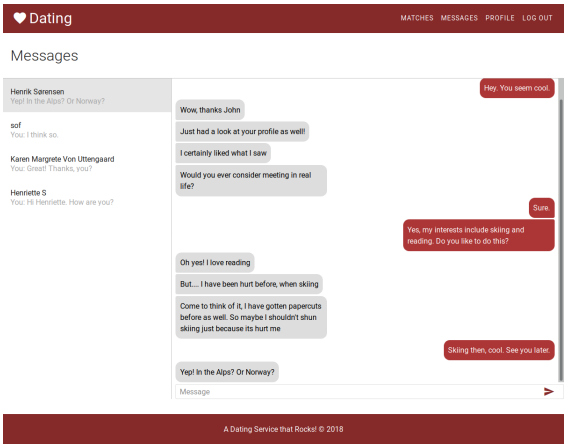
In Chapter 3, our design chapter, we describe which pages the website needs to have to be able to be functional for users. As our final product, we implement each page we set out to for our presentation layer. More energy has been spent on pages we feel are more important for a good user experience. This includes the sign-in page as this is what a user is first presented with. It also

includes the chat page, as we assume this is something users use most often on our site.

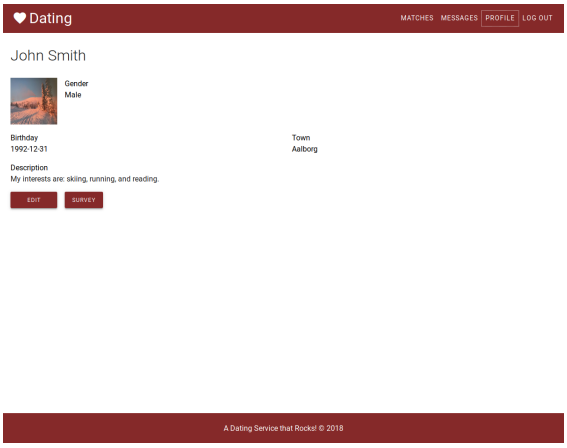
In Figure 6.1, there are four separate screenshots of the pages from the website. In Figure 6.1a, we show what a user fills out before being able to use the site. These are the criteria needed to create a user, before having to answer the first, random ten questions, where the user is forwarded to what is seen in Figure 6.1d. It is also possible for our website to connect users with each other via a chat service. How this looks to the user is seen in Figure 6.1b. Chat history is also saved, so a user can look at messages they receive in the same place without having to visit the specific user's profile page. The final page we show is seen in Figure 6.1c. This is a screenshot of a logged-in user's profile page, where they have the options of editing their profile or entering the survey page again so they can answer more questions.



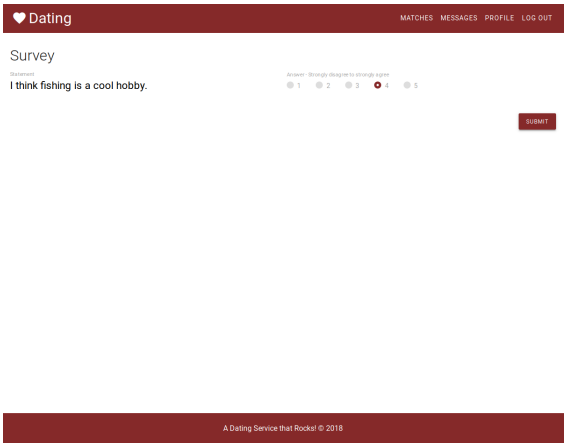
(a) A sign-up sheet filled out before sign-up



(b) Messages between two users



(c) A specific user's profile page



(d) One of the 606 questions

Figure 6.1: Screenshots of the current website

6.2.1 User Opinions

Even though usability is not a significant part of this project, we are still interested in what users have to say about the website to interact with the matching system. As mentioned in chapter 5, where we describe our evaluation from users, we do not have enough data to be able to conclude what users think of our website. However, from the data we do have, we can see that there is a positive trend. This may be because our sample size is small and mostly consists of people who are close to us.

6.3 Matching

In this section we discuss the matching process and the results thereof. We discuss the choice of using the data from PMP, that we have based our implementation on.

6.3.1 Asking the Right Questions

Questions are now selected at random, and it can be discussed if there exists better solutions to this. I could be interesting to investigate if the matching after a user has answered the initial 10 questions, could be improved by selecting the 10 question another way. We imagine three methods that could be used or combined to select the questions.

1: The score is based on the correlation values, which means that higher absolute values in the correlation matrix has higher impact on the score, thus making these questions important. Since the CF can never be perfect, we must trust that questions answered are more correct than the ones predicted. So to gain as much information as possible from the important questions, these must be answered by the users.

2: Another method could be to try to help the CF by answering the questions that has low correlation between them. This correlation should be based on single users only, instead of the original one that is based on friends answers. This new correlation matrix can then tell which questions are easy to guess using other answers. So if users often answer the same of a set questions, then it does not make sense to ask more than a few of these, compared to the questions that has a low correlation to them.

3: The third method is to ask questions that often are answered the same. A question that most people agree on provides very little information compared to questions that gets very distinct answers. These questions could be found by taking the standard deviation of their answers. A low value would mean that many users answer the same, whereas a high value would define questions that receive very distinct answers. An example can be seen in Figure 6.2, where the two black curves represent the standard deviations of questions where most answers are three and five respectively. The red curve represents a question with many distinct answers. These questions with high standard deviation should be investigated, to see if they could improve our matching results.

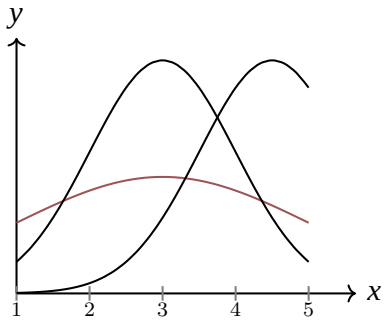


Figure 6.2: Standard deviation of three different questions

6.3.2 Remove Importance of Possible Patterns in CF

While the CF performs well, it still only gives partial correct predictions. It is possible that these small errors become a general pattern in predicted data, where some predictions are always high or always low, without it being accurate. This can make users with few answered questions more similar than if users have more question answered. This can be a problem, since the users with the most questions answered then become less attractive compared to the rest. A simple fix to this is to give a boost to the score corresponding to the amount of questions the two matching users have answered. The difficult part is selecting a scaling factor that fits, without causing the opposite effect, where users with many answered questions are more attractive.

6.3.3 Possible Cultural Differences

When looking at the results of our CF, it is interesting to see that our prediction errors for the test users are similar to those of the data used to train the model, because of the cultural differences between the subjects. Since we are unable to match the intended test subjects, it could be that the correlation matrix from PMP suffers from these cultural differences. Examples of these differences are interpretations of the questions: “I speak more than one language” and “I watch movies with subtitles.” In Denmark almost everybody speaks more than one language, whereas in America only about 20% are bilingual[49]. The other question is also true for many Danes, since many popular movies are produced in America. The other way around is more of a niche, and is comparable to having Danes rating the question: “I watch French or Spanish movies”. A method to improve on the cultural issues with the questions will be described in Section 7.1.

Chapter 7

Conclusion

Our problem statement is expressed as *"How can we create a web application in the functional paradigm that matches people based on their answers to statements from the PMP?"*

In order to answer this question, we look at what we have accomplished over the course of the project. We have successfully created a web application using the programming languages elm and Haskell, both of which are from the functional programming paradigm. Our user evaluations showed that there were problems with the usability of the finished application, especially the presentation layer of it. However, none of these stemmed from the choice of programming paradigm and could be alleviated given additional resources. The results from evaluating the matching system are not promising.

The collaborative filtering we used to predict users' answers to all statements from the PMP works well, but the subsequent attempt to match people based on their answers less so. Whether we had too few test subjects, if they were too different from the participants in the PMP or if our algorithm simply does not work is difficult to conclude. Given additional resources, we would like to evaluate it on a much larger set of people.

7.1 Future Work

The sections mentioned here are features or ideas we would like to have implemented if we did not have the time constraints that a semester project brings. The most significant part we would have liked to have worked on more is the matching algorithm, which will be explained in the first section. These things were considered during the course of the project, but were deemed not necessary and something that would be a good addition later on.

7.1.1 Matching

In this section we discuss the improvements that could be made to the matching algorithm. We start with different methods we could use to select questions. Other changes would be using a different data set and changing the algorithm in that we won't use regular gradient descent, but try stochastic gradient descent instead. These future improvements are described in the following sections.

Drop the PMP Data

The current matching algorithm is reliant on the static data from the PMP. This has been a necessity, since we do not have any actual user data. If the service receives a large user base, we could train our model without the start-up data. Additionally it would make sense to harvest the information from the usage of the service. By doing this we would be able to make correlations based on users of the site and their location, age, gender, etc. The messages between matched users, could be used to identify how successful a given match is. This would also allow us to ask new questions, since the evaluation would provide us with empirical data for it. Another reason for dropping the PMP data would be in the favour of having a data set that is closer to the userbase we want to cater to. An example of this being not having questions meant for an American audience, which would remove the issues with the cultural differences.

Stochastic Collaborative Filtering

The current state of our CF uses regular gradient decent. An alternative to this, is to use stochastic gradient decent, that uses a gradient descent of a fixed sized random selected subset of the problem during each iteration. This method is faster at each iteration, but requires more iterations to find a good solution. Usually a result is found faster, but at the cost of worse predictions. It should be investigated if this method can replace the existing one, or if a combination of the two can be used.

Alternative Matching Methods

As described in Section 2.2, can content-based filtering be used to find users similar other users that a person has liked. This requires many active users, but is definitely something that could be used to improve our matching. In Appendix A we describe how centrality measures can assign different scores to nodes in a network. This could be used to match users based on their popularity, which could also be used as a part of the matching process.

7.1.2 Scalability

If our web application is to be used in a production environment, we have to consider its scalability. We would have to be able to handle an increase in the number of concurrent requests to the business layer and increased size of the information stored in the database layer. With regards to database layer, we must consider what happens, when vertical scaling is no longer an option: sharding is a type of horizontal scaling that is used to distribute a database on multiple machines. Each instance of the database is called a shard and together they form a cluster that looks like a single entity to the outside world. When the database is queried, multiple machines work together to find the data simultaneously. As each shard only contains parts of the data, the index size is smaller, which generally results in faster computation. If the application has a large user base spread over a large geographic area, sharding also allows us to store data relevant to a specific user at a location that is close to them geographically. In the business layer, horizontal scaling is already possible as the API is stateless, all that is needed is some system configuration to allow the API to run on multiple systems at once without interfering with each other. The presentation layer can also be changed to allow for greater scalability. An example of this, is the chat component. In our finished application, we check for new messages every three seconds. This means opening a new connection to the API every time, which has a significant overhead. Instead, each new chat window opened by users, could maintain an open connection to the API through which any new messages are sent.

7.1.3 Extra Functionality for Users

This is not something that was fundamental for our project to work. Extra functionality being implemented would be purely for the benefit of users and not something that is required for them to be able to interact with the matching system. Examples of features we thought of during design was the feature of allowing a user to specify more specific traits in a partner, the most significant being gender.

We also considered the ability for a user to choose what it is they are looking for from the website, whether they want to meet new friends, casual dating or a serious relationship. This makes sense for when the website grows and has a larger userbase, but as of this point, it does not make sense to have. It would also filter out many users by choosing a specific purpose, so it does not look like a user has as many matches which could be disheartening.

7.1.4 Usability Testing

Testing the user interface more extensively is something we would do if there was more time for the project. This testing would consist of having users come to us and creating a user and using the website in front of us, whilst narrating what it is they are doing. Not only would this give us information for a more user friendly design with regards to navigation and functionality, but it could also give us feedback on what a user expects to see when using a matchmaking site.

To make it a more structured test. We would also give the user tasks to complete, to see how long it takes and if it is intuitive enough. An example of a task could be “Find a user and engage in conversation” We would then expect that a user would find the list of user page and then select the message icon to start a conversation. This is not necessarily how an actual user would see it. This is easier to find out by talking to actual users whilst they are using the site.

Bibliography

[1] OkCupid. *We use math to find your dates*. 2018. url: https://www.okcupid.com/about (visited on 12/03/2018).

[2] OkCupid. *Profiling and automated decision-making at OkCupid*. 2018. url: https://www.okcupid.com/legal/profiling (visited on 12/03/2018).

[3] Match Group. *Tinder - Get Started*. 2018. url: https://tinder.com/?utm_source=gotinder-nav-login (visited on 12/03/2018).

[4] Leah E. LeFebvre. "Swiping me off my feet: Explicating relationship initiation on Tinder". In: *Journal of Social and Personal Relationships* 35.9 (2018), pp. 1205–1229. doi: 10.1177/0265407517706419.

[5] Prince Grover. *Various Implementations of Collaborative Filtering*. 2017. url: https://towardsdatascience.com/various-implementations-of-collaborative-filtering-100385c6dfe0 (visited on 09/22/2018).

[6] Charu C. Aggarwal. *An Introduction to Recommender Systems*. Springer International Publishing, 2016. isbn: 978-3-319-29659-3. doi: 10.1007/978-3-319-29659-3_1. url: https://doi.org/10.1007/978-3-319-29659-3_1.

[7] Peter Dolog. *Recommender Systems: Introduction & Content-based Filtering*. 2018. url: https://www.moodle.aau.dk/pluginfile.php/1375776/mod_folder/content/0/WI_9_RecSys_Content2018.pdf?forcedownload=1 (visited on 11/28/2018).

[8] Martin J. Osborne. *People Matching Project*. Apr. 17, 2017. url: https://research.peoplematching.org/ (visited on 10/17/2018).

[9] André B. Bondi. "Characteristics of Scalability and Their Impact on Performance". In: *Proceedings of the 2Nd International Workshop on Software and Performance*. WOSP '00. Ottawa, Ontario, Canada: ACM, 2000, pp. 195–203. isbn: 1-58113-195-X. doi: 10.1145/350391.350432. url: http://doi.acm.org/10.1145/350391.350432.

[10] et al. Gordon Sommers. *Why Study Functional Programming?* url: https://acm.wustl.edu/functional/whyfp.php (visited on 12/10/2018).

[11] Simon Marlow. *Fighting spam with Haskell*. 2015. url: https://code.fb.com/security/fighting-spam-with-haskell/ (visited on 11/02/2018).

[12] Jon Prudy Simon Marlow. *Open sorcing Haxl, a library for Haskell*. 2015. url: https://code.fb.com/web/open-sourcing-haxl-a-library-for-haskell/ (visited on 11/02/2018).

[13] Joe Armstrong. *What's all the fuss about Erlang?* 2007. url: https://pragprog.com/articles/erlang (visited on 11/02/2018).

[14] StatCounter. *Percentage of mobile device website traffic worldwide from 1st quarter 2015 to 3rd quarter 2018*. 2018. url: https://www.statista.com/statistics/277125/share-of-website-traffic-coming-from-mobile-devices/ (visited on 11/19/2018).

[15] Google. *Material Design Guidelines*. 2018. url: https://material.io/design/ (visited on 11/29/2018).

[16] EU. *GDPR*. 2018. url: https://eugdpr.org (visited on 12/14/2018).

[17] Darren Guccione. *What the Most Common Passwords of 2016 List Reveals [Research Study]*. Jan. 2017. url: https://keepersecurity.com/blog/2017/01/13/most-common-passwords-of-2016-research-study/ (visited on 12/14/2018).

[18] Database Guide. *What is ACID in databases*. 2016. url: https://database.guide/what-is-acid-in-databases/ (visited on 11/01/2018).

[19] Dan Pritchett. "BASE: An ACID Alternative". In: (2008). doi: 10.1145/1394127.1394128.

[20] Julian Browne. *Brewers CAP theorem*. 2009. url: http://www.julianbrowne.com/article/brewers-cap-theorem (visited on 11/01/2018).

[21] MongoDB. *Atomicity and Transactions*. 2018. url: https://docs.mongodb.com/manual/core/write-operations-atomicity/ (visited on 10/31/2018).

[22] DB-engines. *DB-Engines Ranking*. Dec. 2018. url: https://db-engines.com/en/ranking (visited on 12/12/2018).

[23] MongoDB. *The MongoDB 4.0 Manual*. 2018. url: https://docs.mongodb.com/manual/ (visited on 12/12/2018).

[24] Google. *Google's Cloud Firestore*. 2018. url: https://firebase.google.com/docs/firestore/ (visited on 12/18/2018).

[25] docker. url: https://www.docker.com (visited on 12/14/2018).

[26] HashiCorp. url: https://www.vagrantup.com/ (visited on 12/13/2018).

[27] Light Code Labs. url: https://caddyserver.com/ (visited on 12/12/2018).

[28] NGINX Inc. url: https://www.nginx.com/ (visited on 12/12/2018).

[29] Internet Security Research Group. url: https://letsencrypt.org/ (visited on 12/13/2018).

[30] Ferdinand Mütsch. url: https://ferdinand-muetsch.de/caddy-a-modern-web-server-vs-nginx.html (visited on 12/12/2018).

[31] Evan Czaplicki. *The Official elm Guide*. elm. url: https://guide.elm-lang.org/ (visited on 11/12/2018).

[32] ClojureScript. *ClojureScript Guide*. ClojureScript. url: https://clojurescript.org/guides/ (visited on 11/12/2018).

[33] Matthew Griffith. *elm-ui*. elm packages. url: https://package.elm-lang.org/packages/mdgriffith/elm-ui/latest/ (visited on 10/12/2018).

[34] Richard Feldman. *elm-css*. elm packages. url: https://package.elm-lang.org/packages/rtfeldman/elm-css/latest (visited on 10/31/2018).

[35] The Haskell Community. *Haskell*. 2018. url: https://haskell.org (visited on 12/14/2018).

[36] Douglas N. Arnold. *Ariane 5*. Aug. 2000. url: http://www-users.math.umn.edu/~arnold/disasters/ariane.html (visited on 12/14/2018).

[37] Sebastian Philipp. *Hayoo! - Haskell API Search*. 2017. url: http://hayoo.fh-wedel.de/ (visited on 12/12/2018).

[38] Scotty. *Scotty*. 2018. url: https://hackage.haskell.org/package/scotty (visited on 12/18/2018).

[39] Yesod. *Yesod*. 2018. url: https://www.yesodweb.com/ (visited on 12/18/2018).

[40] Oleg Grenrus. *servant-server: A family of combinators for defining webservices APIs and serving them*. Oct. 2018. url: http://hackage.haskell.org/package/servant-server-0.14.1 (visited on 12/12/2018).

[41] Alp Mestanogullari. *Why Servant*. July 2018. url: https://haskell-servant.github.io/posts/2018-07-12-servant-dsl-typelevel.html (visited on 12/18/2018).

[42] Michael Snoyman. *persistent: Type-safe, multi-backend data serialization*. Oct. 2018. url: http://hackage.haskell.org/package/persistent-2.9.0 (visited on 12/12/2018).

[43] Victor Denisov. *mongoDB: Driver (client) for MongoDB, a free, scalable, fast, document DBMS*. May 2018. url: http://hackage.haskell.org/package/mongoDB-2.4.0.0 (visited on 12/12/2018).

[44] Smartbear. *What is Regression Testing*. 2018. url: https://smartbear.com/learn/automated-testing/what-is-regression-testing (visited on 12/18/2018).

[45] Bryan O'Sullivan, Don Stewart, and John Goerzen. *Real World Haskell: Testing and quality assurance*. 2008. url: http://book.realworldhaskell.org/read/testing-and-quality-assurance.html (visited on 12/18/2018).

[46] Simon Hengel. *Hspec: A Testing Framework for Haskell*. 2018. url: http://hspec.github.io (visited on 12/18/2018).

[47] Koen Claessen and Nick Smallbone. *QuickCheck*. 2018. url: http://hackage.haskell.org/package/QuickCheck (visited on 12/18/2018).

[48] Philip Wadler. *The Expression Problem*. Nov. 1998. url: http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt (visited on 12/13/2018).

[49] Raymond Watson. *Bilingual America*. 2017. url: https://www.puertoricoreport.com/bilingual-america/#.XBE15S5AqV4 (visited on 12/12/2018).

[50] Andrew Disney. *Social Network Analysis*. Cambridge. url: https://cambridge-intelligence.com/keylines-faqs-social-network-analysis/ (visited on 10/12/2018).

[51] Lawrence Page Sergey Brin. "The Anatomy of a Large-Scale Hypertextual Web Search Engine". In: (1997).

[52] Leo Katz. "A new status index derived from sociometric analysis". In: (1953).

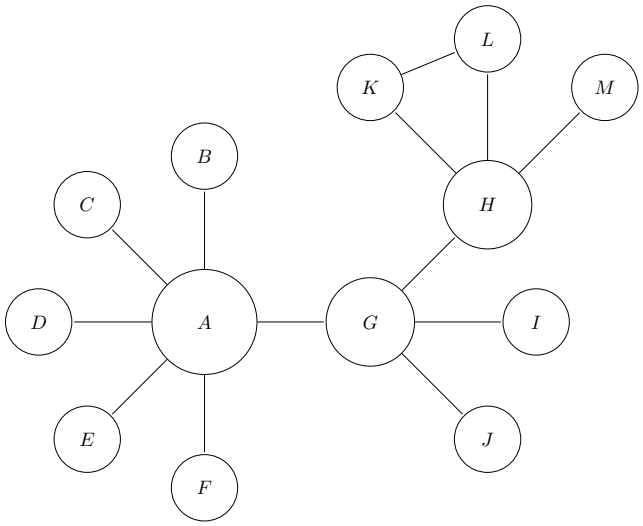
[53] Phuoc Quang Nguyen et. al. "PageRank vs. Katz Status Index, a Theoretical Approach". In: (2009). doi: 10.1109/ICCIT.2009.272.

Appendices

Appendix A

Centrality

A term used in network analysis that describes the importance of each vertex is centrality[50]. This section introduces the concept of Centrality and how it can be used. There are different algorithms that tries to compute this, which each describe a different form of importance. Centrality is used to find important people in a social network or websites with high credibility. For such services it is important to be able to validate if two people are similar in terms of popularity if we are to find people with similar personalities.



(a) An undirected network

Degree Centrality (DC) This method is a simple way of finding importance in a network, and can be used to find popular nodes. Node score is based on the amount of its own edges. If the graph is directed, this score is divided into an in-degree and an out-degree. In figure A.1a the value of A will have a score of 6; and G and H will have a score of 4.

Betweeness Centrality (BC) This node score is based on number of shortest paths within the network that the node is used in. It can be used to find individuals who have a high communication flow importance. A and G are each part of 45 shortest paths while H is only part of 29. The reason that G is more important than H is that G connects H to the larger part of the network.

Closeness Centrality (CC) The sum of the length of all shortest paths defines the score for each node. This measure can be used for finding the nodes which can quickest influence the network. Using this method, G will score better (20) than A (21), since it has a closer connection to all other nodes.

Eigenvector Centrality (EC) Each node is given a score similar to DC, based on each node's own edges. In addition to this, EC also recursively considers the importance of its neighbours and their neighbours. A gets a score of 0.16, G scores 0.14 and H scores 0.12. The equation to calculate the value for a given node is given by:

$$x_i = \frac{1}{\lambda} \sum_{k=1}^{n} a_{k,i}\, x_k \tag{A.1}$$

Where x_i is a node, λ is a constant, and A is the adjacency matrix such that $a_{k,i}$ is 1 if k is a neighbour of i and 0 otherwise. This can be converted into the following matrix multiplication:

$$\lambda x = xA \tag{A.2}$$

PageRank and Katz Centrality (KC) Similar to EC, both PageRank and KC consider the relation to important nodes[51, 52]. The difference is that PageRank and KC also considers the direction of the edges. These algorithms both measure how likely random walks are to end on a given page, which in practice makes them very similar[53].

$$Katz(i) = \sum_{k=1}^{\infty} \sum_{j=1}^{n} a^k A_{ji}^k \tag{A.3}$$

Where a is a constant between 0-1, A^k is the kth adjencency matrix of A such that A_{ji}^k is 1 if there is a path from j to i in exactly k steps.

$$PR(i) = (1-d) + d \sum_{k \in M_i} \frac{PR(k)}{C(k)} \tag{A.4}$$

Where PR is the PageRank of a node, d is a constant between 0 and 1, M is the set of neighbours pointing to i, and $C(k) = \frac{1}{o}$, where o is the amount of outgoing edges from k.

The summation in KC and the recursion in PageRank causes both algorithms to not have a natural stopping point, which makes computation on large-scale networks complicated.

We can use the centrality measures to gather information of how important each person is. In-degree shows who receives the most attention, while out-degree shows who gives the most. BC and CC are both measures of how close each individual is to the rest of the network. The high importance nodes that these algorithms find are nodes that will broadcast well, which will not be an immediate feature for our solution. EC, KC and PageRank are the only methods that assigns scores. EC only works on undirected graphs, while the other two can be used on directed graphs as well. If our network is undirected this method can be applied since its complexity is better than that of PageRank and KC. If, however, we can assign meaningful direction in our model, then PageRank and KC should be a consideration.

Appendix B

Questionnaire

Functional dating - Introduction

In relation to our seventh semester project, we have created a dating website. We are very interested in seeing if we are able to use algorithms to match people with other users in the system, with whom they have a lot in common with.

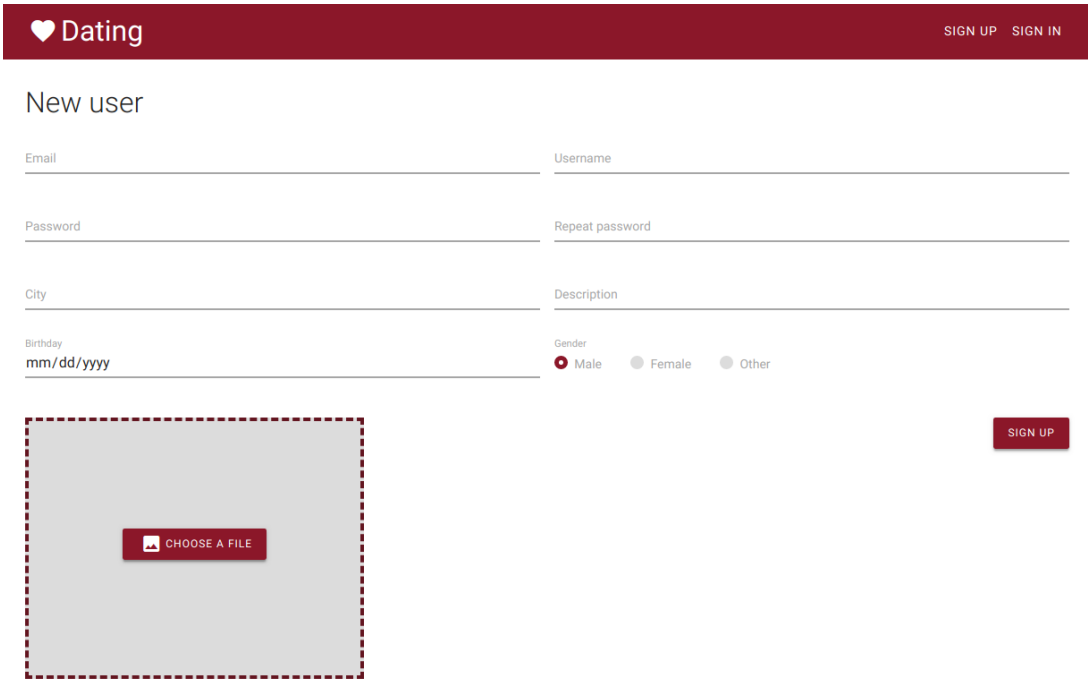
What we need from you and your partner or best friend is truthful answers to at least ten questions (but the more, the better). Preferably you shouldn't talk about the questions, as we want you to answer as if you did not know each other.
We would like for you to answer the questions through our website. After signing up, you should be taken to a page where you can answer questions. The same page can be found through the Profile link in the header.

While signing up requires an email, a username, an image, etc. it is not important that you use your real personal information - we just need to know the username you and your significant other sign up with, so we can test the efficacy of our algorithm.

The website can be found at
dating.antonchristensen.net

* Required

This is the sign up page we want you to use. All fields are required, but we do not care if you fill any of it out properly.



1. **What is the other person you did the survey with, to you?** *
   Mark only one oval.
   - ◯ Romantic partner
   - ◯ Friend
   - ◯ None - I did the survey alone

2. **What username did you use when signing up to dating.antonchristensen.net?** *
   Please read the introduction text, if this question stumps you.

   _____

3. **What username did your significant other use when they signed up to dating.antonchristensen.net?**
   Please read the introduction text, if this question stumps you.

   _____

4. **Did you answer at least ten questions after signing up?** *
   Mark only one oval.

   - Yes *Skip to question 5.*
   - No *Start this form over.*

Skip to question 5.

User experience of the site

Thank you for answering some questions about yourself.
We would greatly appreciate it if you spent five more minutes rating your experience using the website.

Please answer truthfully.

5. **I used the website on a mobile phone** *
   Mark only one oval.

   - Yes
   - No

6. **I had difficulties signing up** *
   Mark only one oval.

	1	2	3	4	5	
Strongly disagree						Strongly agree

7. **I had difficulties navigating the website** *
   Mark only one oval.

	1	2	3	4	5	
Strongly disagree						Strongly agree

8. **I looked at these parts of the website** *
   Check all that apply.

   - [ ] Matches (a list of all users)
   - [ ] My profile
   - [ ] Another user's profile
   - [ ] Messaging
   - [ ] Editing a profile
   - [ ] Sign in page

9. **My overall impression of the website's design was positive** *
Mark only one oval.

	1	2	3	4	5	
Strongly disagree	◯	◯	◯	◯	◯	Strongly agree

10. **General comments regarding the navigation**

11. **General comments regarding the design**