

# Final Report

**Jacob Coates, James Harvey, Jenny Hutchins, Mary  
Kassayova, Joshua Manchester, Marin Srithar, Harry  
Varanauskas**

Department of Computer Science  
University of Warwick

2 February 2024

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>System Overview</b>	<b>1</b>
2.1	Accounts . . . . .	1
2.2	Relationships (Mentor-Mentee) . . . . .	2
2.3	Meetings . . . . .	3
2.4	Workshops . . . . .	4
2.5	Plan of Action . . . . .	4
2.6	Messages . . . . .	5
2.7	Application Feedback . . . . .	6
2.8	Administrator Functionality . . . . .	6
<b>3</b>	<b>Requirements Modification</b>	<b>7</b>
3.1	Modified Requirements . . . . .	7
3.2	Dropped Requirements . . . . .	8
<b>4</b>	<b>Development Discussion</b>	<b>9</b>
4.1	Development Tools . . . . .	9
4.1.1	Containerisation with Docker . . . . .	9
4.1.2	Github . . . . .	9
4.1.3	React TypeScript . . . . .	10
4.1.4	Tailwind CSS . . . . .	11
4.1.5	Material UI . . . . .	11
4.2	Development Progression . . . . .	12
4.2.1	Factory Design Pattern . . . . .	12

4.2.2	REST API . . . . .	13
4.2.3	Authentication . . . . .	13
4.2.4	Plan of Action . . . . .	15
4.2.5	Meetings . . . . .	16
4.2.6	Workshops . . . . .	18
4.2.7	Matching . . . . .	19
4.2.8	Admin . . . . .	19
4.2.9	Database Implementation . . . . .	20
4.2.10	UI Design . . . . .	21
4.2.11	Communication with the API . . . . .	21
<b>5</b>	<b>Product Evaluation</b>	<b>22</b>
5.1	Testing and Validation . . . . .	22
5.2	User Interface Evaluation . . . . .	25
5.2.1	Visibility of System Status . . . . .	25
5.2.2	Match between system and real-world . . . . .	26
5.2.3	User control and freedom . . . . .	27
5.2.4	Consistency and standards . . . . .	28
5.2.5	Error prevention . . . . .	29
5.2.6	Recognition rather than recall . . . . .	29
5.2.7	Flexibility and efficiency of use . . . . .	30
5.2.8	Aesthetic and minimalist design . . . . .	31
5.2.9	Help and Documentation . . . . .	32
5.3	Improvements and Extra Features . . . . .	32

<b>6</b>	<b>Development Process Evaluation</b>	<b>33</b>
6.1	Methodology . . . . .	33
6.2	Planning and Management . . . . .	34
6.3	Communication . . . . .	34
6.4	Conclusion . . . . .	35

## List of Figures

1	Login page. . . . .	2
2	Registration page. . . . .	2
3	Mentor selection page. . . . .	3
4	Example of meetings in the system. . . . .	3
5	Plan of Action user interface. . . . .	5
6	Admin Dashboard . . . . .	6
7	Backend file structure . . . . .	12
8	Component file structure . . . . .	12
9	Token decode example from jwt.io [1] . . . . .	15
10	Entity-Relation diagram. . . . .	21
11	User Generator command line interface . . . . .	23
12	Github actions automatically running tests . . . . .	24
13	Pytest test log . . . . .	24
14	Example of visibility of system status. . . . .	25
15	Gear symbol for settings . . . . .	26
16	Gear symbol for settings . . . . .	26
17	An example of a commonly used phrase . . . . .	27
18	Pop up modal . . . . .	27
19	Profile picture can open the user menu . . . . .	29
20	Inputs labels and default date values on scheduling form . . . . .	30
21	Tooltip helps with recognition . . . . .	30
22	Example of well spaced component . . . . .	31
23	Feedback box is only displayed for completed meeting. . . . .	31

# **1 Introduction**

Deutsche Bank has asked us to provide a system to facilitate its mentoring programme. The system is required to match suitable mentors with mentees according to prescribed criteria, and enable users to schedule meetings and workshops. To be successful, the system must be clear and intuitive for busy professionals to use with minimal training, while incorporating the necessary functionality to match mentors and mentees in the most optimal way. This document describes the website we have created to meet Deutsche Bank's criteria, the development process and our evaluation of the product.

## **2 System Overview**

### **2.1 Accounts**

Every user of the system must create a new account before they can have access. Creating an account requires the user's first name, last name, email address and password. The name of the user is needed for users identifying others within the system, whereas the email address and password can be used to help the system find the correct user when logging in. The email address also provides a unique identification for an account, since no two accounts can have the same email address. Furthermore, the password helps protect the account from people who are not the user and keeps it secure, thereby maintaining confidentiality as specifically required by the client. Once the account is created, users can then separately specify their role as mentee, mentor, or both, as required by the client's specification. The screenshot below shows the login page.

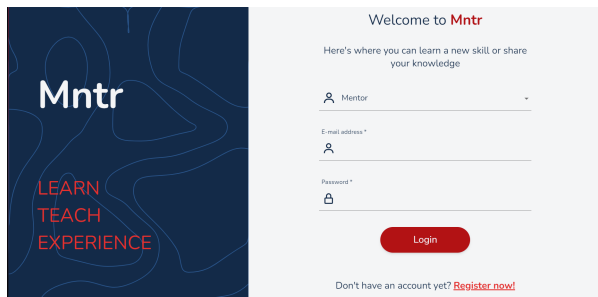


Figure 1: Login page.

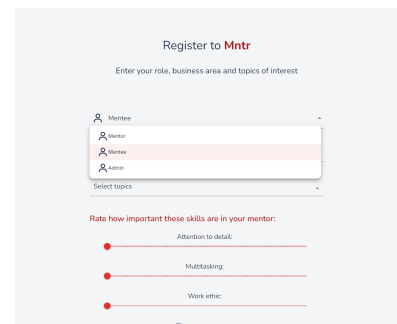


Figure 2: Registration page.

## 2.2 Relationships (Mentor-Mentee)

Each mentor and mentee has a relationship with a counterpart (mentors have mentees as counterparts and mentees have mentors). The system uses a matching algorithm to present a list of mentors in order of compatibility for the logged-in mentee. The ordering of mentors meets the client's requirements by taking into account two key factors: (1) mentors' feedback scores, weighted across each scoring category according to the skill preferences of the logged-in mentee (2) overlap between mentors' specialist topics and the logged-in mentee's selected topics. In addition, to comply with client requirements, the list of mentors must all be from a different business area to the mentee. The results of the matching algorithm are unique to each mentee and are displayed only to that mentee. Once the mentee chooses their mentor, a relationship is automatically created between the two counterparts. At any time during the relationship, mentees can rate each of their mentor's skills on a scale of 1 to 10. These ratings are averaged with all other mentor feedback scores received. Average feedback scores are used in the matching algorithm, so this fulfills the client's requirement of using mentee feedback to refine the suggestion process.

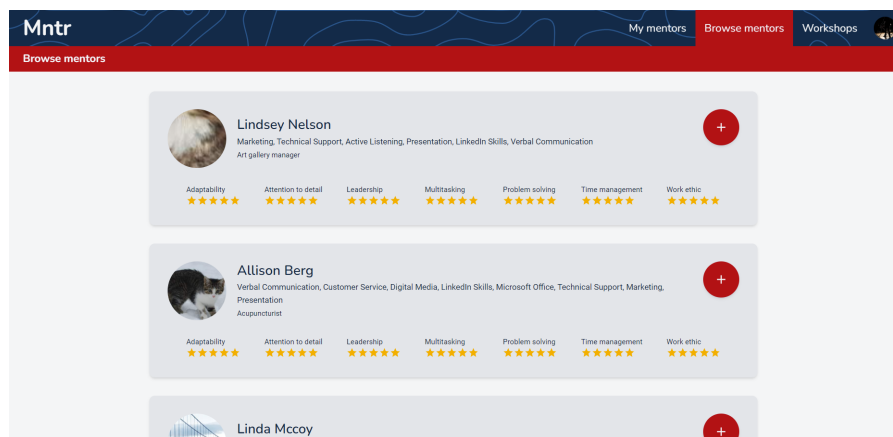


Figure 3: Mentor selection page.

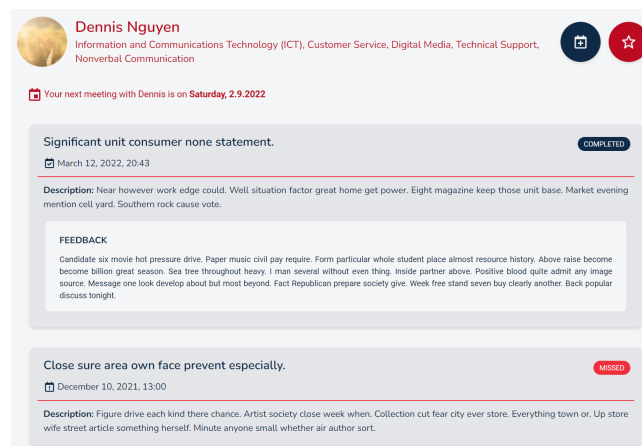


Figure 4: Example of meetings in the system.

## 2.3 Meetings

In accordance with the Rules of Mentoring set by the client, only mentees can request meetings in the system. Mentees request meetings with mentors via a meeting request message. This request includes the start time, end time, title and description of the meeting. In the system, the meeting is assigned a status. Before the mentor responds to the request, the status is pending; if the mentor accepts, the status becomes going-ahead; and if the mentor rejects, the status



becomes cancelled. If a mentor rejects a meeting request, the mentee will be able to request an alternative meeting time until the mentor accepts (this is consistent with the client's requirement for mentors to give up time to mentees). The actual meeting takes place in external software but it is registered in the system as running once the start time has been reached. Within 30 minutes of the end time, the mentor must register the meeting as completed otherwise the meeting will be assigned the status missed. When the mentor has completed a meeting, they can send feedback to the mentee relating to the meeting. The following screenshot shows the record of two meetings with a single mentor; the status for the first is completed (with feedback provided) and the second is missed.

## 2.4 Workshops

The client requirements specify that the system must allow for group sessions in the form of workshops. Workshops are similar to meetings; however, they include a group of mentees instead of just one mentee and do not require mentees to have a relationship with the mentor of the workshop. Each workshop contains a topic, title, description, start time, end time and location. When a threshold number of mentees has shown an interest in a particular topic (based on topics selected when registering), a workshop-creation invite will automatically be sent to a random mentor who specialises in that topic. To accommodate topics with a smaller level of interest, workshop demand automatically increases in small increments over time provided that at least one mentee has shown an interest. This ensures that, eventually, workshops will always be scheduled so no one has to wait indefinitely for a workshop. If a mentor rejects a workshop invitation, then another mentor is requested to take over the workshop. Once the workshop has taken place, the workshop demand value for the relevant topic is reset to zero.

## 2.5 Plan of Action

It is a client requirement for every mentor/mentee relationship to have a Plan of Action which can be tracked through the system and viewed by both parties.

Each Plan of Action stores the mentee's milestones for the relationship. At the start of the relationship, the Plan of Action is empty on the home screen. Mentees can add milestones to their Plan of Action at any time. Once the user is satisfied that they have achieved their milestone, they can mark it as completed. A user can also revert a milestone back to incomplete. The screenshot below shows a sample plan of action with three completed and six uncompleted milestones. There is no limit on the number of milestones.

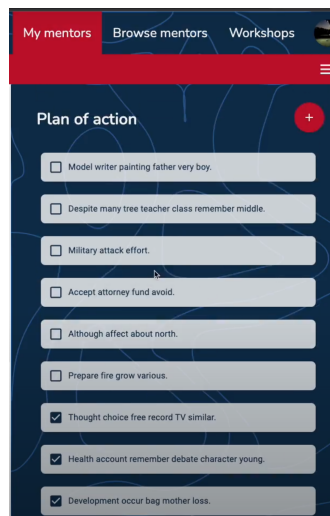


Figure 5: Plan of Action user interface.

## 2.6 Messages

To meet our requirement that mentor and mentee pairings should be able to communicate, messages can be sent between users on the system. They can be accessed by selecting the account button on the top left corner of any page once logged in. Each message will contain the content itself along with who sent the message and the time the message was sent. The content of these messages can take one of four formats: an invitation to a workshop, an invitation to a meeting, a text message and a report message. Once a mentee has requested a meeting the respective mentor will receive a message containing the information about the meeting. If there is sufficient demand for a topic then a mentor in that field will receive a message.

## 2.7 Application Feedback

The client requirements state that users should be able to submit features and improvements for the website. To meet this requirement, the system gives both mentors and mentees access to a page where they can submit their feedback on the website. This feedback can include features they would like to see in future updates as well as bugs in the current websites that need to be fixed. The user can also mention any skill, topics and business areas currently not included in the system that they think should be included. Feedback be viewed and acted upon by the system administrator.

## 2.8 Administrator Functionality

If a user signs in with administrator credentials, they will be directed to a page that has functionality exclusive to administrators. This functionality includes adding topics, skills and business areas stored in the system. All system feedback, including feature suggestions from users, can be accessed by an administrator, who can then contact the appropriate people to implement the potential improvements.

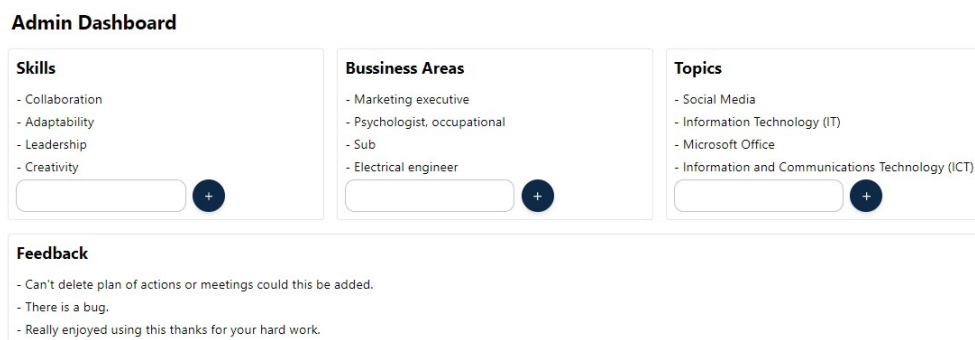


Figure 6: Admin Dashboard

## 3 Requirements Modification

### 3.1 Modified Requirements

**C1: Confidential information must be protected.**

Initially, the aim of this requirement was to encrypt all user data, including names and e-mail addresses. Since names are displayed on users' public profiles, we have decided to only encrypt passwords in the backend. This still ensures data protection.

**C4: Mentoring rules must be followed.**

The system is designed in a way that inherently follows the rules of mentoring. Only mentees are able to request meetings, which they can do at any time, and mentors must choose whether or not to accept the request. The matching algorithm ensures that no mentors from the same business area or with no significant overlap in topics are recommended to the mentee.

**C8: Mentors should be able to provide mentees with feedback on their development.**

**D8: Mentors must be able to select one of their assigned mentees and provide feedback. This page should include separate text boxes for areas that need improving and things that went well. Mentees should be notified when they receive feedback.**

Whilst the customer facing requirement remains unchanged, the developer facing requirement was changed to allow for a better workflow for users. Instead of a mentor providing feedback to the mentee at arbitrary points, feedback is given after every meeting. This implementation is more structured and gives mentors regular prompts to provide feedback. Additionally, meetings are marked as missed if no feedback is provided, giving extra incentive for mentors to give feedback.

**C18: Users could be able to report any misconduct of other users on the website.**

In the current version of the system, users are unable to report misconduct. However, this feature is also fully implemented on the backend and can therefore be easily added in the future.

## 3.2 Dropped Requirements

### **C5: A warning should be generated if a mentoring rule is broken.**

This was dropped because it is impossible to break the mentoring rules within the system: the implementation of the system does not allow users to break any of the mentoring rules.

### **C9: Mentors should be able to arrange workshops through the application.**

Since this system is a prototype, we chose to prioritise other features. However, this feature is fully implemented on the backend, so the groundwork is there to implement it in the future.

### **C13: Mentors should be prompted to create group sessions or workshops in a particular field when there is a large enough demand for tutoring in that skill/field.**

Similarly to C9, the system is just a prototype so we did not implement this feature and instead focused on other parts of the system. Nevertheless, this feature is implemented on the backend and could be easily implemented in a future build.

### **C25: The website should be usable on mobile devices.**

Whilst the UI is functional on a mobile device, it is neither responsive nor user friendly. The desktop version of our app was our main priority and we decided that a mobile version fell outside the scope of the project. Thanks to using the Tailwind framework, this feature can be easily implemented in the future.

## 4 Development Discussion

### 4.1 Development Tools

#### 4.1.1 Containerisation with Docker

Docker is virtualisation software that allows developers to run isolated containers on their systems. Initially, we did not plan to use containers as we deemed the project simple enough to set up on each team member's personal computer. However, when it came to setting up databases and configuring development environments for all members of the team, we encountered difficulties due to the wide range of operating systems we were developing on. Therefore, we decided to dockerize[2] the project to allow for easy cross-platform development. This involved creating individual containers for the backend, front-end and database. We used docker compose to automatically launch all containers at once. For development purposes, we used docker volumes to mount our code to the container. This allowed for hot-reloading and removed the need for restarting the container every time a change was made to the code, speeding up development. Using docker made setting up the project for all team members easy and meant more time could be spent developing instead of setting things up. Whilst we did not end up deploying the application, running the website in containers should theoretically make this process simpler.

#### 4.1.2 Github

The team collaborated on the project using a Git[3] repository hosted on Github[4]. Git is version control software that allows users to create their own branches of the code-base. Github is a Git repository hosting service which hosts the repository on the cloud and allows developers to access the code from anywhere. We used Git to manage and merge our code together so that members of the team did not overwrite each other's work. However, using Git was not always a straightforward task due to wide-range of experience in our team. To tackle this problem, we ensured that developers with prior experience using Git assisted those with less experience to ensure that changes were committed

correctly. We occasionally encountered mishaps where code was submitted to the wrong branch or merge conflicts were not correctly resolved. Fortunately, due to the powerful functionality of Git, we were able to resolve any issues that occurred. Planning the subsystems each team member worked on helped prevent large conflicts when merging workspaces. We utilised Git commit messages to describe the changes we had made every time we commit code. This informs other members of the project what changes have been made to the system. If a branch breaks, Github allows us to revert back to a previous state where there is a working build available, therefore preventing the team from starting from the beginning if a serious fault occurs.

### 4.1.3 React TypeScript

React is a JavaScript library for UI development[5]. We chose to use React because it is one of the most popular frameworks and some team members had previous experience in it. Using this framework was a relatively seamless experience, despite the fact that some users had to spend some time learning how to use it.

Instead of plain JavaScript, we chose to use TypeScript with this framework to allow for static type checking as well as a tighter integration with Visual Studio Code, our IDE of choice. TypeScript[6] is a strict superset of JavaScript and compiles directly into JS, meaning that we could opt out of static typing and just use it as plain JavaScript in some situations. Thanks to type checking, TypeScript made it much easier for us to catch some errors which may have gone unnoticed until much later in development, or even past the production stage [7]. TypeScript is able to detect 15% of all JavaScript bugs early.

Using a Javascript library rather than plain HTML and CSS makes it a lot easier to develop a modern interactive user interface. Compared to other UI libraries, React is more flexible thanks to its modular structure and its code is easier to maintain.

#### **4.1.4 Tailwind CSS**

Tailwind is a utility-first CSS framework used for rapid building of custom user interfaces[8]. It is essentially a collection of CSS classes which can be composed to build any design we desire. The main choice for this framework was its relatively shallow learning curve and a high level of customisability and freedom. The main advantage we have observed while using Tailwind was that it allowed us to include styling directly in our TypeScript files, which made the development process faster and more streamlined. However, this also proved to be a disadvantage, since it made our code more cluttered. When combined with the Tailwind IntelliSense VSCode extension, which provided autocomplete functionality for Tailwind classes, this framework was a pleasure to work with. Tailwind's detailed and easy-to-read documentation was also very helpful in the development process.

#### **4.1.5 Material UI**

Unlike some other CSS frameworks, Tailwind does not provide any pre-built components. This is why we have decided to use the Material UI library which provides a wide range of animated, minimalist, modern-looking components. We have made use of this library when designing our login and registration forms, dialog windows as well as smaller components like checkboxes, sliders and dividers. Using MUI components was a great way to make our UI look more professional while also saving time. However, customising these components to fit in with the rest of the interface was not easy and required a lot of trial and error.



## 4.2 Development Progression

### 4.2.1 Factory Design Pattern

We decided to use a factory design pattern for our backend. This involved splitting the backend into multiple folders, each containing a python package that performed a functional aspect of the application. Here is a diagram to illustrate:

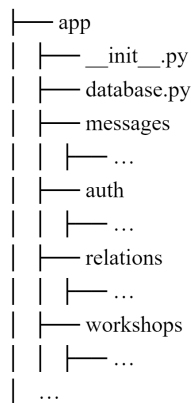


Figure 7: Backend file structure

Application-wide functionality, such as the database, is placed in the root folder of the app whilst components such as authentication and messaging are given their own folders. This is because all components of the program interact with the database so its utility functions should be globally available to prevent repeated code. Each component uses a flask blueprint[9] which represents a collection of views and functions. Here is our auth component as an example:

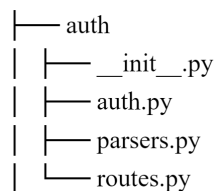


Figure 8: Component file structure

Each file's purpose is as follows: `__init__.py` initialises the blueprint for the component, `auth.py` contains any functionality code necessary for the routes, `parsers.py` contains the necessary models for parsing data, and `routes.py` is where the API routes are defined. The URL of API routes takes the form `/api/<blueprint name>/<route>` which clearly separates API URLs from frontend URLs and makes it easy for our frontend team to find endpoints. This structure proved beneficial for multiple reasons. Firstly, we found that splitting the backend into components made it easier to distribute tasks to members of the team as it forced us to break the backend down into smaller problems. Additionally, it meant that each developer worked in an isolated part of the backend, reducing the likelihood of any file conflicts when committing code. Furthermore, it made adding new features to the API easier as all that was required was creating a new folder and using one of the existing components as a template. We also found that this design pattern made it easier for team members working on the frontend to find and understand backend routes due to the clear structure.

#### **4.2.2 REST API**

In our design document, we said we would use Flask-RESTful[10] to develop our REST API. However, upon developing our API, we realised that Flask-RESTful lacked any form of API documentation and was fairly outdated. Therefore, we decided to switch to Flask-RestX[11] which supports Swagger[12] API documentation (discussed further in our testing section) and is actively developed. Fortunately, Flask-RestX is a fork of Flask-RESTful so migrating our existing code was a straightforward task.

#### **4.2.3 Authentication**

In order to satisfy our requirement of protecting confidential information, it was important for us to salt and hash passwords before storing them on the database. This ensures that, in the case of a data breach, customer passwords will still be protected. When registering or logging in to an account, the user's

login details are sent in a HTTP POST request. Ideally, we would use a HTTPS request for the added encryption; however, due to the website being a prototype, we have not configured this. When the backend receives the user's registration details, it randomly generates a 16-character salt. The following hash computation then takes place:

$$\text{hash} = \text{SHA-256}(\text{SHA-256}(\text{password}) + \text{salt})$$

The hash and corresponding salt are stored in the database along with the user's account information. In order to verify login information, the same calculation is performed using the existing salt. If the generating hash matches the existing hash, then the user's password is correct and we grant them authentication.

We decided to use a JWT token authentication system. Whenever a user logs in with a valid password, we generate a JWT token using a secret key and store the token in a httpOnly session cookie. Our initial implementation involved storing the token in local storage and sending the token in the authorisation header of every API request. However, we found that sending the token every time was inconvenient for our frontend team and storing the token in local storage was not secure. Using a httpOnly session cookie meant that the token was automatically sent in every API request and gave us greater control over how long the client stores the token. The token contains a payload which includes accountID, email, userID, role, as well as an expiration time. Here is an example of a decoded token:



not their mentor and vice versa). When a milestone is created its initial status is *incomplete*. A milestone can be marked as *complete* or *incomplete*. A user can view all of their milestones with their relationship counterpart and can remove milestones if they are no longer relevant regardless of status; unfortunately, we did not have time to develop this feature as part of the prototype however all required functionality is present in the backend. All five of these operations can be performed by both the mentee and mentor of the relationship as long as the relationship is valid. All Plan of Action API calls first check that the logged-in user belongs to the Plan of Action's relation before giving a response. If there exists an entry in the relationship table with the given relationshipID and userID, then the relationship is valid.

### Frontend

The Plan of Action is implemented as a reusable React functional component that takes the array of milestones and a relation ID as props. The array of milestones is rendered using the `map()` function and for each element, a List Item is created with a Material UI checkbox component.

#### 4.2.5 Meetings

For the meeting database model, we used a meeting table to store meeting data with a foreign key to the relationship table. When creating a new meeting, the API checks that the logged-in user is a mentee. We check the role of logged-in users using the payload inside of their JWT token. Additionally, checks are made to ensure that the meeting's mentor and mentee are in a relation. If this is not the case then the meeting request not be sent. Additional time constraints are checked to avoid invalid meetings from being created.

Cancelling a meeting is accessible from both a mentee and a mentor. If they do not belong to the meeting being cancelled (checked using the relationshipID), then the cancellation fails. Meetings can only be cancelled if their status is *going-ahead* or *pending*. If a request is *pending*, the `cancel_meeting` method is also used when rejecting a meeting. This is because the meeting is already created in the table and it is easier to maintain the database by just storing it as a cancelled meeting. When a cancellation is successful, the database is

updated to display *cancelled* for the meeting. When testing this section, we encountered *completed* meetings that could be cancelled. Therefore, we added constraints that this function would only run if the status is *going-ahead* or *pending*.

Accepting a meeting is only possible by the mentor and they must be a member of the relationship within that meeting. The meeting must also have a *pending* status. We update the database with the current time to get the most accurate status of the meeting. If it is not pending, then the accepted request will fail, otherwise the status of the meeting will update to *going-ahead*, meaning that the meeting has successfully been accepted. Completing a meeting is, again, only accessible to a mentor of the same relationship as the meeting contains. If a meeting is not currently in a *running* state, then it cannot be completed. When a meeting is completed, an optional feedback string can be parsed to provide feedback to the mentee, regarding the results of the meeting's conclusion.

We discovered an issue where some meeting requests could be accepted after the end time of a meeting. Therefore, whenever meetings are requested from the database, an *update* function is called. This function ensures that meetings that have expired (meetings that have an end time 30 minutes earlier than the current time) are updated to a *missed* status.

### **Frontend**

Meetings on the frontend are implemented in two ways - on the main dashboard and on a separate Meetings page for mentors. In both cases, each meeting is built using a reusable Meeting Card component to ensure consistency within the UI. For mentors, pending meetings contain the option to cancel a meeting and running meetings contain a text input window to submit feedback. The Meetings page for mentors contains all meetings with all of their mentees, ordered by status from currently running and upcoming, pending and completed, to missed and cancelled. The list of meetings on the dashboard shows all pending and past meetings for each individual mentee/mentor. We also display a reminder for the next going ahead meeting with a particular mentee/mentor on the dashboard, this allows a user to instantly find this information. This is important because the next scheduled meeting is likely to

be the most important piece of information for a user.

#### 4.2.6 Workshops

Workshops are created based on demand for a topic. Whenever a new topic is created, the topic is added to the database in the `workshop_demand` table with an initial demand of 0. The demand score is affected by two main factors: (1) how many users have registered as mentees for that topic and (2) how long the demand has existed for the topic. Whenever a workshop is run for a topic, its demand resets to 0. We used a Python library called `APScheduler`[13] to run a function every hour which increments the demand of all topics by a small amount.

Every time the demand changes (either by a new user joining the system or by the time increment), we run a `check_demand` function which iterates through all demands in the database and checks if any have exceeded the demand threshold. In the case that it has been exceeded, we use the messaging system to send a workshop-creation invite to a random mentor who specialises in the relevant topic. The mentor can either accept to create a workshop or reject it in which case another random mentor is invited to create it instead. After a workshop is created, every mentee who has that topic is invited. They are only added to the `user_workshop` table with the workshop ID if they accept the given invite. Workshops can only be created if a mentor is invited to create one through the `check_demand` function.

A workshop can have one of four statuses: 'going-ahead', 'cancelled', 'running', or 'completed'. When a mentor creates a workshop, the workshop is inserted into the database with the status of 'going-ahead'. Whenever a mentor cancels a specific workshop, its status is updated to 'cancelled'. The `update_workshop_status` function is used to check and update statuses of all workshops. A workshop is 'running' if the current time is past the start time and before the end time and the status is not 'cancelled'. Once the workshop status changes to this status, the demand for the topic is reset to 0. A workshop is 'completed' if the current time is past the end time and the status of the workshop is not 'cancelled'.

A challenge we faced when implementing workshops was fine-tuning the de-

mand values. We had to find the ideal demand threshold, new user demand weighting and demand increase with time. In the end, we settled on a demand threshold of 10, new user weighting of 1 and a time increase of 0.1 every day.

### **Frontend**

There is a separate page for workshops accessible from the navigation bar. For mentees, this page lists all upcoming workshops that they signed up for, and for mentors, all the workshops they are currently running are displayed. Unfortunately, due to limited time and prioritisation of other features, workshops were not fully implemented in the frontend with a lack of workshop creation and sign up functionality as well as notifications to prompt mentors to create workshops when there is high demand.

#### **4.2.7 Matching**

Implementation of the matching algorithm went smoothly and we did not encounter any challenges.

#### **4.2.8 Admin**

Three separate tables in the database are used to store the skills, topics and business areas used in the system. All users can retrieve these attributes from their relative table, however; only a user logged in as an admin can remove and add skill, topic, and business area to and from the database. Mentees and Mentors can submit feedback, which is stored in a table and the admins can access all entries. The users role is checked using the authentication token before the operation takes place.

### **Frontend**

There is a dedicated admin dashboard that displays and allows the addition of skills, business areas and topics and also shows the feedback from users about the app. It's design is even more minimalistic than the main interface this is because it is a developer facing part of the system and will be used relatively



infrequently this means it does not have to be as visually appealing and instead focuses purely on functionality.

#### 4.2.9 Database Implementation

To interface with the database we used the `psycopg2`[14] Python library. A challenge we encountered was that backend developers were struggling to interact with the database due to the complex nature of the `psycopg2` library. To solve this problem, we instructed our database engineer to write a simplified, easy-to-use, interface for executing SQL and retrieving query results. The interface uses a context manager to ensure that database connections gets closed automatically and to remove the risk of developers forgetting to close it themselves. After this simplified interface was implemented, we found it made database interaction far easier for all members of the team.

Going into development the backend team had a clear view of what the database schema should look like as we could refer to the database schema diagram from our design document. However, we only added the entities to the schema once they were required in case there were attributes we had missed in the planning stage. This also helped prevent confusion amongst the team as we were creating the tables relevant to the subsystems we were working on. The schema for the system's database can be seen below. Attributes marked with a red arrow are references to the `userID` attribute in the `user` table.

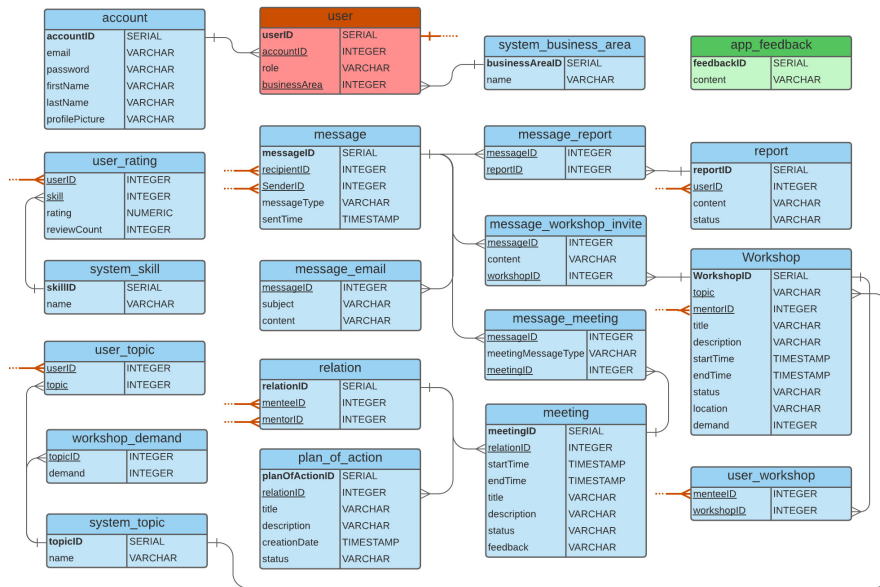


Figure 10: Entity-Relation diagram.

#### 4.2.10 UI Design

The biggest challenge in UI design and frontend development was determining which features to focus on in what order. We ended up overestimating the importance of some smaller features, leading to spending too much time on their development and not having time to implement other, bigger features. Synchronising frontend and backend development to be able to fully implement functionality was another difficulty we had to overcome, mainly by improving communication. This is why we initially used hard-coded data instead of pulling data directly from the API.

Compared to the proposed design, no significant changes have been made to the UI. We have updated the look of the login forms after deciding to use Material UI components and we have changed the button style.

#### 4.2.11 Communication with the API

To connect the web application to the API we had to allow the user to generate http requests. This was done using the axios javascript library. Axios

is promise based which presented a challenge as it required understanding of asynchronous functions in Javascript. We had to be careful not to try and use the value wrapped in a promise until it had been fulfilled. The next stage was to update the interface once the data was loaded in by the axios request. React provides a number of hooks to allow us to update the information displayed to the user. The `useEffect` hook allows you to perform side effects on components; we initially used it to run functions on the first render of the page to load data from the backend. Doing this means the page does not have to wait for all the data to load before rendering. Later we discovered that we could pass dependencies to a `useEffect`. This means the function will be run anytime the dependency changed. This was useful for updating information for the currently selected mentor/mentee in the dashboard. The other hook we used was `useState` which allows variables to be updated and components that use them to rerender, with each state made up of a variable and `setVariable` function. There were two main uses of this hook: to store the data fetched from calls to the API and to store data inputted by the user in forms. The first required us to parse the data from the API call and call the `setVariable` function to update the variable. To implement the second we used the `onChange` function of inputs components to continually set the state as the user changed the value of the input. A button could then be used to call a function that read the current state of the variables and perform the correct axios request.

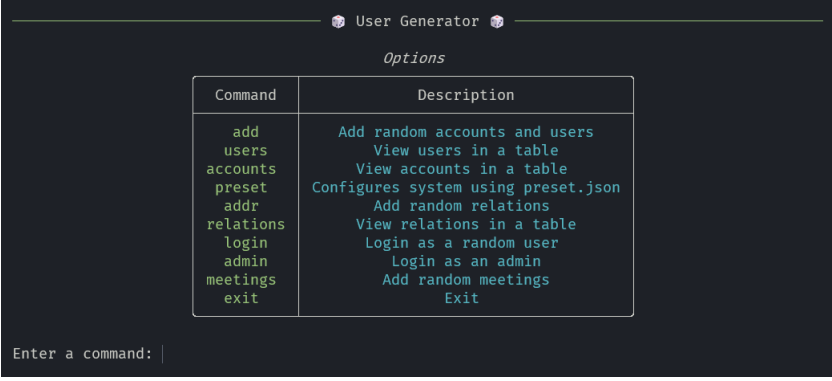
## 5 Product Evaluation

### 5.1 Testing and Validation

The application was tested regularly to make sure Deutsche Bank was receiving a quality product and to ensure it fulfilled the requirements laid out in the requirement document. Continuous testing was achieved using Github actions[15], which ran `pytest`[16] unit tests and integration tests every time a push was made to the repository on Github. Getting to grips with Github actions was a difficult process, but was worth it as it allowed the developers to focus on the code and for the testers to focus on the testing. Not every require-

ment was tested and the frontend and backend were not thoroughly endpoint tested as we prioritised end-to-end testing on the API and ran out of time; this could have been avoided if we started writing the tests earlier on in the development of the app.

We wrote a CLI program called UserGenerator, which populates the system with randomly generated data from a Python library called Faker[17]. It allowed both the frontend and back end team to unit test their code manually and locally as they wrote the different modules. The program can be used to generate random users, random data for the users (such as their desired topics to be mentored in), random relations, random meetings between mentors and mentees, and random workshops. Using User Generator for testing was not as rigorous as writing out actual tests, but it allowed developers to generate data to create test scenarios in which they knew the desired outcome, and see if they got that outcome from the app, which is better than nothing. We found that UserGenerator played a crucial role in our ability to rapidly test app functionality. Rather than having to manually register accounts every time we built a new version of the website, UserGenerator could add hundreds in a few seconds.



```

User Generator
Options
Command      Description
add           Add random accounts and users
users        View users in a table
accounts     View accounts in a table
preset       Configures system using preset.json
addr        Add random relations
relations    View relations in a table
login        Login as a random user
admin        Login as an admin
meetings     Add random meetings
exit         Exit
Enter a command: |
```

Figure 11: User Generator command line interface

Pytest was a good choice for testing the backend as it worked well with Github actions and made writing tests easy. To run pytest tests using Github actions, we only had to run the command `âpytestâ` in the console, and pytest would

run all the tests by itself, which made setting up the continuous testing and writing the tests a lot simpler. Using Github actions to automatically run tests whenever a push was made allowed us to indirectly regression test our application too, as if a push to the repository broke a previously working feature, actions would have picked up on it and notified us which module was no longer working, so we could go and fix the problem.

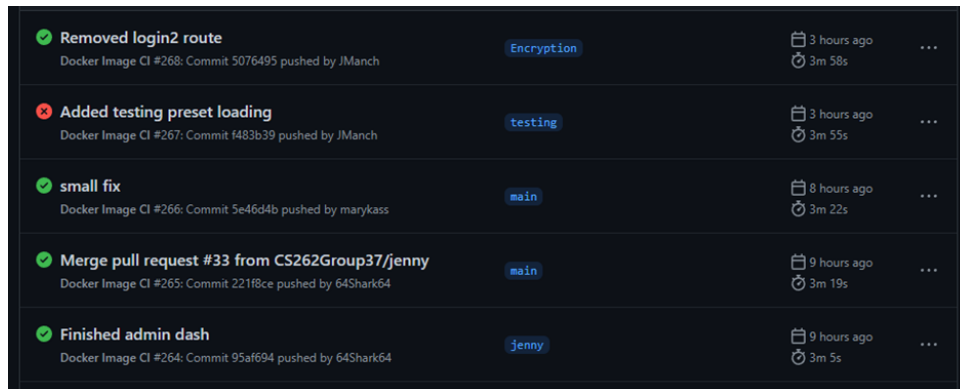


Figure 12: Github actions automatically running tests

```
Run pytest
1 Run docker exec mntr-backend-1 sh -c "pytest --no-header -v"
2 docker exec mntr-backend-1 sh -c "pytest --no-header -v"
3 shell: /usr/bin/bash -e {0}
4 ===== test session starts =====
5 collecting ... collected 12 items
6
7
8 tests/test_backend.py::test_adminFunctions PASSED [ 8%]
9 tests/test_backend.py::test_createAccount PASSED [ 16%]
10 tests/test_backend.py::test_adminFunctionsForUsers PASSED [ 25%]
11 tests/test_backend.py::test_createRelation PASSED [ 33%]
12 tests/test_backend.py::test_rateMentor PASSED [ 41%]
13 tests/test_backend.py::test_sendEmail PASSED [ 50%]
14 tests/test_backend.py::test_createWorkshop PASSED [ 58%]
15 tests/test_backend.py::test_joinWorkshopAsMentee PASSED [ 66%]
16 tests/test_backend.py::test_createMeeting PASSED [ 75%]
17 tests/test_backend.py::test_acceptMeetingAsMentor PASSED [ 83%]
18 tests/test_backend.py::test_addingAppFeedback PASSED [ 91%]
19 tests/test_backend.py::test_gettingAppFeedback PASSED [100%]
20 ===== 12 passed in 0.56s =====
```

Figure 13: Pytest test log

Due to time constraints, the testing part of development wasn't as thorough as it could've been. Had we started earlier, we would've used the extra time to

write endpoint tests for the database and the frontend, user acceptance test the application, test the responsiveness of the application, more thoroughly test our backend, test whether the application works as intended on a wide variety of browsers and test every requirement.

## 5.2 User Interface Evaluation

We have decided to use Nielsen's Usability Heuristics [18] to evaluate our user interface. This is because they are probably the most-used usability heuristics for user interface design[19].

### 5.2.1 Visibility of System Status

The design should always keep users informed about what is going on. If erroneous data has been entered by the user, the system should display that a problem has occurred as well as a reason for the problem occurring. If no error message was displayed the user might think the system is working incorrectly or taking a long time to process a user's interaction. This has been implemented on the register page if a user enters non-matching passwords or an invalid email address then the user is notified as to why their request was not accepted.

The figure displays two screenshots of a user interface. The left screenshot shows a registration form with the following fields: E-mail address \* (containing 'James'), First name \* (containing 'Joseph'), Last name \* (containing 'Harvey'), Password \* (containing '••••'), and Password confirmation \* (containing '••••••••'). Below the form is a red 'Register' button. Two error messages are displayed below the button: 'Passwords do not match' and 'Invalid e-mail address'. The right screenshot shows a login form with the following fields: E-mail address \* (containing '123') and Password \* (containing '•••'). Below the form is a red 'Login' button. One error message is displayed below the button: 'Wrong e-mail, password or role'.

Figure 14: Example of visibility of system status.

However this principle was not implemented system wide, when an API call fails there is no explicit indication to the user as to what has happened. We did not implement this as it was not considered core functionality during our requirements analysis. The API does return messages and make use of error codes when calls fail, this could be used in the future to implement displaying the errors to the user.

### 5.2.2 Match between system and real-world

The design should use words, phrases and concepts familiar to the user. The icons used on each page of the website are commonly used such as a plus symbol for adding goals and mentors, a gear symbol for settings and a calendar symbol to schedule meetings.

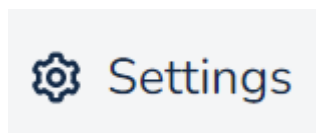


Figure 15: Gear symbol for settings



Figure 16: Gear symbol for settings

Additionally the status of a meeting is indicated using intuitive phrases including missed, running, pending and going ahead that map to how the user would describe the status of the meeting in the real world.

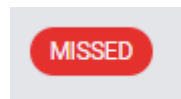


Figure 17: An example of a commonly used phrase

Overall, our system matches up well against this heuristic making the system suitable for non-technical users as it is easy to navigate and interpret requiring no specific prior knowledge or understanding.

### 5.2.3 User control and freedom

The system should have a clear way of exiting the current interaction. A user is able to use back arrows in their browser to cancel any navigation actions. When adding elements such as scheduling meetings, adding plans of action or sending a message this is done through a pop up modal that can be cancelled at any time by clicking outside the area of modal.

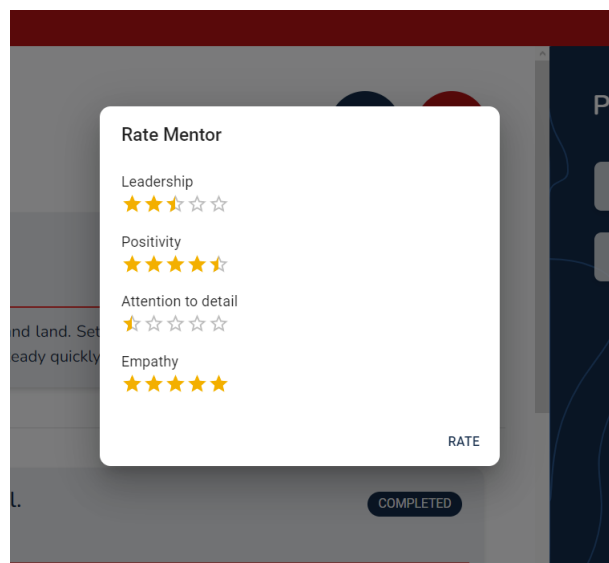


Figure 18: Pop up modal

However, we have not implemented a method for undoing these actions once confirmed, plans of action, meetings and messages cannot be edited or deleted.



This is a problem as typos when inputting information cannot be rectified. Additionally a mentee is unable to leave a relationship with a mentor, if they accidentally add a mentor their dashboard will become cluttered and make interacting with their actual mentors more difficult. These features were not implemented as they were not explicitly stated in the requirements or design and were considered of a lower priority compared to other features. All of these methods are available on the API so could be implemented if the project moves beyond a prototype.

#### **5.2.4 Consistency and standards**

Users should not have to wonder whether different words, situations, or actions mean the same thing by ensuring both internal and external consistency is maintained. Internal consistency refers to the consistency within our system for example, the order of text fields is the same on the login and register page. The items in the navigation bar remain in the same place throughout the website and the description of the current page is always located on the red bar below this contributing to a consistent layout. Additionally there is a consistent colour scheme throughout the interface instantly indicating to the user that they have not navigated away from our interface. We also use the same terms across the system to refer to entities, such as mentor, mentee, workshops, meetings and messages; this means the user is clear as to what they are interacting with in the system.

External consistency refers to established conventions in the industry. Our website logo, positioned in the top left, links back to the homepage and in the top right corner the users profile picture is a link to a user menu that shows notifications and provides access to the settings page and logout feature, which are both common design patterns in web applications.

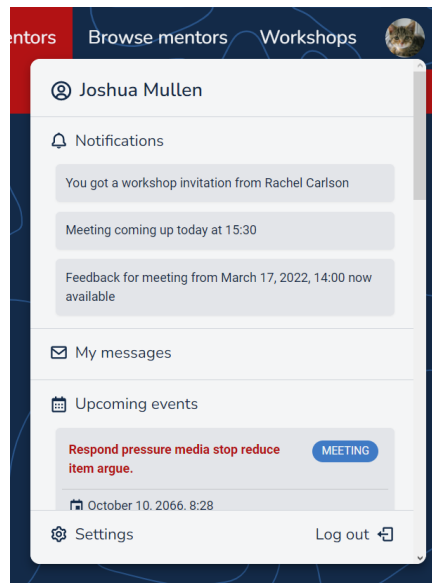


Figure 19: Profile picture can open the user menu

### 5.2.5 Error prevention

Prevent erroneous data from being accepted by the system. The system's sign-up page has a password confirmation to ensure the user has not made an accidental mistake in their entry. Some data fields throughout the program are not accepted if left blank. For example, all fields on the register page are required to advance. This prevents the user from unintentionally missing fields. There are a few places where the users can input data that results in an error when an API call is made using this data. For example there is no date validation ensuring the start date is before the end date and that meeting is being scheduled in the future.

### 5.2.6 Recognition rather than recall

Minimise the user's memory load by making elements, actions, and options visible. Interfaces that promote recognition reduce the amount of cognitive effort from the user. Our system achieves this by labelling fields and menu items and providing icons and colour to increase recognition time.

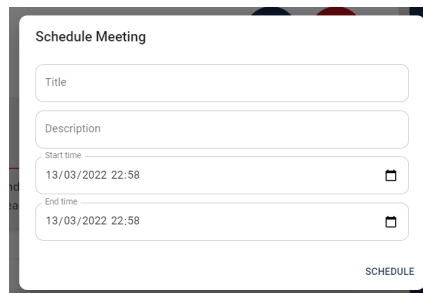


Figure 20: Inputs labels and default date values on scheduling form

Some fields have default input such as the start and end time of meetings, this helps the user recognise the correct type of data to enter without having to recall the correct format every time they create a meeting. Furthermore we have implemented tooltips in a number of places to provide further detail about the functions of buttons, such as the schedule meeting and rate mentor buttons.

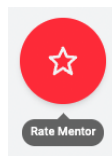


Figure 21: Tooltip helps with recognition

### 5.2.7 Flexibility and efficiency of use

Short cuts should be available to speed up the interaction for experienced users. We were not able to add shortcuts to our system however all core parts of the program can be instantly accessed from all pages. Without product deployment and the subsequent creation of experienced users, it is hard to know what shortcuts would be the most useful. Users can request the shortcuts they would like to see using the feedback form on the settings page, which could then be implemented in future updates.

### 5.2.8 Aesthetic and minimalist design

Interfaces should not contain information that is irrelevant or rarely needed. The system maintains an aesthetic throughout with options suitably spaced apart, no colour conflicts and no blank spaces. Small components, such as buttons, use minimalist on-hover animation style and eye-catching colours to attract usersâ attention but not cause too much distraction.

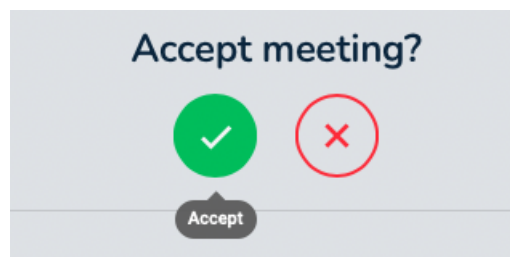


Figure 22: Example of well spaced component

On each page there is one explicit way to navigate to a page, this eliminates redundancy therefore contributing to a minimalist design. Additionally we have attempted to only display information if it is needed, for example when displaying meetings the feedback box is only visible if the meeting is completed as other meetings do not have any feedback associated with them.

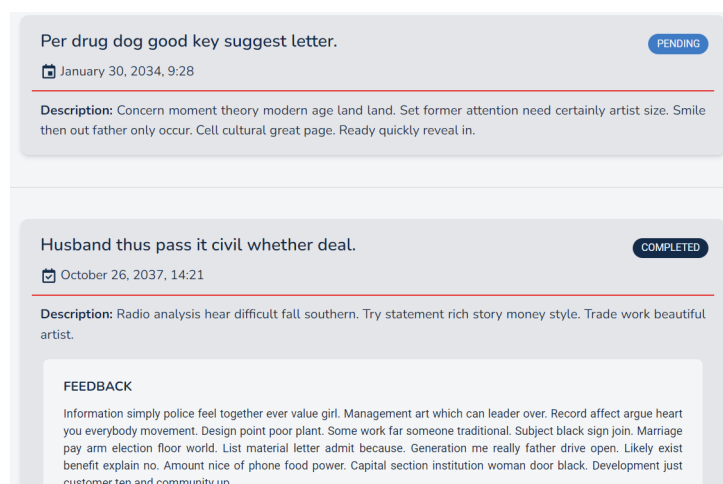


Figure 23: Feedback box is only displayed for completed meeting.

Furthermore a user's bio is not visible on the dashboard and can instead only be seen on the profile page, this is because this information is not relevant to the purpose of the dashboard, which is to provide an overview of the relationship between a mentor and mentee. However there are some modifications we could make for example on the dashboard there is a complete backlog of all meetings including missed meetings as more and more meetings happen this could become overcrowded and distract from the more important meetings that have recently happened or are due to happen. A similar problem occurs when a user is connected with excessive amounts of mentors/mentees making it difficult for users to find a specific person to navigate to, however this is not a design issue as the solution is to cap the number of mentors/mentees a user can have and also allow them to end a relationship with a mentor/mentee. Overall our interface adheres well to this heuristic and the simplistic design gives room for expansion with new features.

### 5.2.9 Help and Documentation

The system should provide documentation to help users understand how to complete their tasks. As our system is a prototype we did not deem it necessary to include documentation on how to use the program within the system. However, our minimalistic and intuitive design should mean the need for instructions is minimal.

## 5.3 Improvements and Extra Features

The application feedback forms submitted by users may sometimes suggest extra features they would like to see in the system. So after deployment, the admins will have a greater understanding as to what features the majority of users would like to see. However, we had some ideas ourselves of some features we didn't implement in initial development but felt would benefit the system. **Group Chats:** A group messaging service would be highly advantageous for users participating in a workshop. Messages could be sent before the workshop to inform attendees of location changes as well as the required knowledge to comprehend the content of the workshop. Attendees would then

be able to discuss the contents of the workshop further increasing their understanding of the topic.

**In-application online meetings/workshops:** Online meetings are becoming increasingly common in the workplace with more employees working from home. When creating a meeting a user will have the option to select 'online meeting', which will create a meeting as normal but when the meeting starts the user will be prompted to join an audio or video call on the system. These meetings could also have the option to be recorded so the mentee can easily recall the actions that took place in a meeting. This feature could be implemented via the use of an external online video conferencing software such as Microsoft Teams or by the development of tailor-made software that can run the meeting within the web application. This will ensure the mentoring process can continue in unforeseen circumstances where an attendee is not able to make the in-person meeting.

**Mobile Application:** In our requirements analysis document we explained that a mobile application will not be implemented in the development of the prototype. However, as the number of users on the system increases the number of these users that will not always have access to a desktop also increases. A mobile application would increase efficiency for mobile users as they can access features whilst offline as well as offer easier means of sending notifications.

## 6 Development Process Evaluation

### 6.1 Methodology

Our plan was to follow the waterfall methodology during development, however, due to our relative lack of experience in software development and specific technologies we were unable to anticipate every consideration during the design phase. Therefore we had to remain flexible and adapt the design as our understanding and experience increased. However, the plan based approach did allow us to work from a clear set of requirements and use the design document to structure the development and mitigate the risk of scope creep.

## **6.2 Planning and Management**

We split into two groups, those working on the backend and those working on the frontend. We thought it would be advantageous to apply the majority of our group members to the backend to avoid bottlenecks where the frontend team are waiting for services required to make progress. Towards the end of the project, we found development to be slow due to only committing a small number of team members to the frontend. We further split the system into multiple subsystems, allowing members of each team to work in parallel increasing our workflow. The separation into subsystems also made testing easier as each member could test their own part without relying on the rest of the system, it also helped us find errors later on in the development process as we could identify the subsystem the error was occurring in. We assigned most subsystems to team members but left a few unassigned to be completed by members who finish their tasks first. This was because we did not know how long each subsystem was going to take so it meant all members were occupied even after they finished their original tasks.

Although we split the subsystems evenly between members, the lack of intermediary deadlines meant that some services were not ready for the frontend team when required. This meant the frontend team had less time to complete their tasks before the final deadline, which led to a stressful environment.

## **6.3 Communication**

We used Discord[20], an online messaging platform, to help inform which members of the group were focusing on what part of the project. Online group meetings allowed members of the team to meet up more regularly than in person and especially helped during the covid-19 pandemic when members of the team had to self isolate. Nevertheless, in-person meetings were also necessary to ensure members of the group were meeting deadlines.

The group was split into the two main segments of the project development: Frontend and Backend, which were further split into subsystems. These two teams needed to communicate with each other effectively in order for the pro-

ject to be functional. Small meetings were held between team members after the completion of a subsystem to fix errors and test functionality. As certain subsystems were similar, group meetings led to efficient error solving as some members had already encountered these problems. We used the Microsoft Live Share extension on Visual Studio Code enabling collaborative editing, which ensured high productivity whilst debugging. Using the Live Share extension also aided our version control during debugging sessions as everyone was editing in the same workspace.

## **6.4 Conclusion**

Our process of development, for the most part, had a positive impact on the final product. Our plan made sure each team member always had a task to complete, however without preliminary deadlines some parts of the system were not ready when expected by other team members. This was mitigated by quick communication amongst team members to prioritise development on the service required.

Although development became more stressful as the deadline approached every member of the team approached the project enthusiastically and remained respectful of other team members and their contributions.



## References

- [1] "jwt.io." <https://jwt.io/>. Accessed: 04-02-2022.
- [2] "Docker." <https://www.docker.com>. Accessed: 11-03-2022.
- [3] "git." <https://git-scm.com/>. Accessed: 25-02-2022.
- [4] "Github." <https://github.com>. Accessed: 04-02-2022.
- [5] "React js." <https://reactjs.org>. Accessed: 27-02-2022.
- [6] "Typescript documentation." <https://www.typescriptlang.org/docs/>. Accessed: 04-02-2022.
- [7] Z. Gao, C. Bird, and E. T. Barr, "To type or not to type: Quantifying detectable bugs in javascript," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 758–769, 2017.
- [8] "Tailwindcss." <https://tailwindcss.com>. Accessed: 2010-09-30.
- [9] "Flask blueprint." <https://flask.palletsprojects.com/en/2.0.x/blueprints/>. Accessed: 11-03-2022.
- [10] "Flask-restful." <https://flask-restful.readthedocs.io/en/latest/>. Accessed: 11-03-2022.
- [11] "Flask-restx." <https://flask-restx.readthedocs.io/en/latest/>. Accessed: 11-03-2022.
- [12] "swagger.io." <https://swagger.io>. Accessed: 25-02-2022.
- [13] "Apscheduler." <https://pypi.org/project/APScheduler/>. Accessed: 11-03-2022.
- [14] "psycopg2." <https://pypi.org/project/psycopg2/>. Accessed: 11-03-2022.
- [15] "Github actions." <https://github.com/features/actions>. Accessed: 04-02-2022.

- [16] "pytest." <https://docs.pytest.org/en/7.0.x/>. Accessed: 04-02-2022.
- [17] "Faker." <https://faker.readthedocs.io/en/master/>. Accessed: 11-03-2022.
- [18] N. Jakob, *Microservice Architecture: Aligning Principles, Practices, and Culture*. 1994.
- [19] "Heuristic evaluation." [https://en.wikipedia.org/wiki/Heuristic\\_evaluation](https://en.wikipedia.org/wiki/Heuristic_evaluation). Accessed: 11-03-2022.
- [20] "Discord." <https://discord.com>. Accessed: 11-03-2022.