CS261 Coursework Design Document

Group 17: *Ben Lewis, Dan Risk, Edmund Goodman, Jay Re Ng, John-Loong Gao, Rahul Vanmali, Tomás Chapmann Fromm*

1 Introduction

Deutsche Bank requires a system prototype to support the mentoring process for employees. The principal purpose of this document is to broadly record the process documentation and design choices for said system prototype. For process documentation, this document will detail development methodologies in addition to workflow and project organisation. Design choices include a prototype user interface design and technical diagrams demonstrating user interactions between multiple users and the system. Furthermore, our testing strategy and technical descriptions such as choice of technologies used and system architecture will be covered in depth to illustrate a fuller idea of the final product.

2 Process documentation and planning

2.1 Development methodology

We opted for an agile development methodology based on the Scrum model. The development timeline will be broken down into a large initial planning phase and a series of week-long sprints. We chose to elect our project manager as scrum master, since this role is similar to that of the project manager in organising the team and improving workflow. We will use Trello to create and manage a scrum board, which will allow for easy management of the product backlog. At the start of each sprint, we will meet to discuss the items from the product backlog to be completed in that sprint - this will usually be decided according to the project timeline below, however some small adaptations may need to be made. Each day within the sprint, the team will discuss progress and bring to attention any challenges they are facing. At the end of the sprint, we will carry out a sprint review, and each developer can showcase parts of the system which have been completed. The sprint cycle is completed with a sprint retrospective, where improvements can be made to the development process before the next sprint.

An agile methodology is optimal for a short timeframe as it allows rapid product development with concurrent implementation and testing so we can make effective use of the time available. It also allows for more flexibility if we face unexpected delays during development, which is likely with an inexperienced team. Producing a thorough plan before beginning the agile development process is suitable since deliverable deadlines are fixed, so ensuring work is completed on time is critical. Additionally, contact with the client is limited so requirements are unlikely to change and the project will not deviate much from the initial plan.

2.2 Project organisation and communication

We decided on a flat team hierarchy since we have similar levels of software development experience, self-managed but with well defined responsibilities. To communicate between meetings, we decided to use Discord, as it is already a popular choice within the team and is easy to learn for new users. It offers both text and voice chat as well as a number of features such as GitHub integration, so we are notified via Discord when pull requests are made, for example.

Discord can be used for online meetings however we aim to convene in-person at least twice a week, as this enables us to share ideas much more easily and maintain focus during meetings. At these meetings we can compare progress on the system with our plan, and assign tasks based on documentation or what tasks need doing in order to stay on schedule. We will use Trello to manage the product backlog, and see which tasks are currently in progress, to ensure the project will be completed by the deadline.

2.3 Project schedule

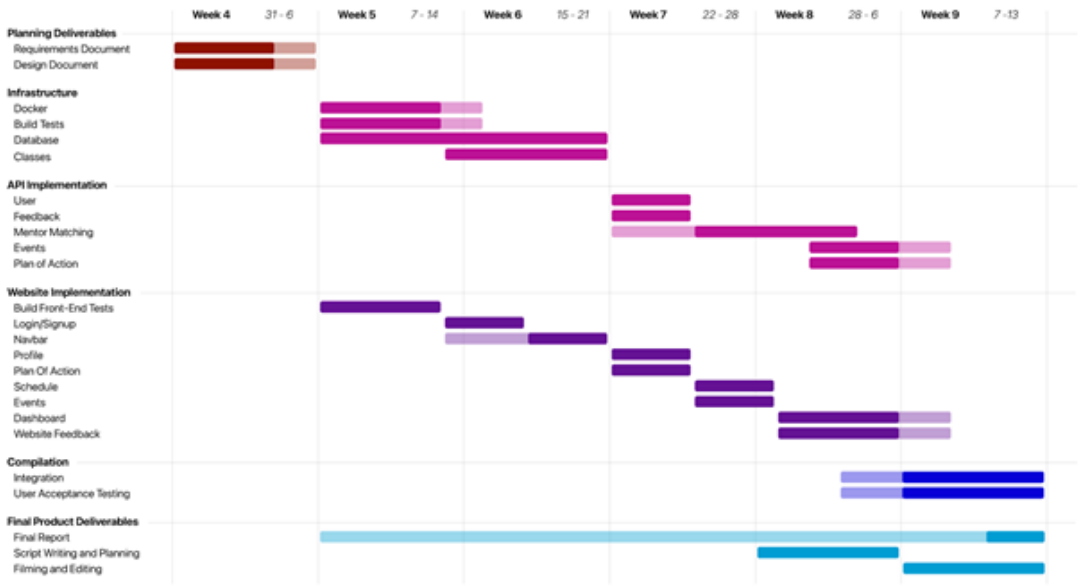

Figure 1: Gantt chart of the project schedule.

The timeline outlines which features of the system will be implemented in each weekly sprint, divided according to the team which is working on them, with final product documentation separated as this is a collaborative effort by the whole team. As we are following a Scrum methodology, task order may be adjusted weekly, ie. in each sprint, so that the project remains on track, so which features are completed in each week will most likely change. It also acts as a reminder that the final report should be completed, when time is available, concurrently with the project itself, so it doesn't need to be written in its entirety in the few days before submission.

2.4 Risk management

Risk	Impact	Likelihood	Severity	Mitigation	Contingency	Residual
Team members unable to work	Smaller team until the member can resume work	2	7	Team members take precautions	Reduce scope and reallocate work across team	2
Task longer than expected	Project delayed until task completed	5	4	Allocate buffer time to allow overrunning tasks	Re-assign team members to load balance	4
Poor code quality	System fails testing or linting procedures	5	4	Code review and automated testing through CI/CD	Re-assign team memebrs to fix pair program	3
Requirements change	Design and existing code must be changed	3	8	Use an agile methodology to facilitate requirements change	Update project plan and proceed with new one	3
Code lost	Code must be rewritten	1	7	Use 'git' as version control, and remote backup to GitHub	Restore code from local or remote backups	1
Problems with dependencies	Component of project fails external library or technology	2	4	Choose technologies and dependencies carefully	Find replacement library or technology	2
Scope creep	Addition of unnecessary features causing growth of project scale	7	4	Include extra features in project timeline, and stick to it	Drop lowest priority features	5

3 Technical description

3.1 Scope

Our system must satisfy a complex specification, so its development must be carefully planned and executed. However, the goal is to create a prototype, not a finished product. This means that some features that would be required in production, but are either do not fit in the timescale for the prototype, e.g. , or would require integration from external systems, e.g. company calendar applications, should not be implemented. Our requirements analysis fully defines the scope of what we will implement.

3.2 System attributes

This section will communicate the different qualities needed by software, what is within our scope, how we have addressed them, and what we are not doing for this prototype.
The end goal of the software engineering lifecycle is to end with satisfied customers. As such, it is crucial that usability of the software is given important consideration. Our goal is to ensure that the website is intuitive to use, has little ambiguity and can be learnt ideally without assistance even for non-technical users. There will be a wide coverage of acceptance testing carried out to ensure this is achieved. Along with this, our prototype aims, if possible, to make our system accessible to people with disabilities with features such as enlarging text and higher contrast colours.
Compatibility and portability are important points to consider since we want to make our system as portable as it needs to be, and compatible with systems used by most of our users. Being a web app, makes it portable since every user will have some access to a device able to run a browser. Extreme cases such as legacy browsers will not be prioritised since it is unlikely to be used. The prototype will not include desktop and mobile applications since it is out of the scope and not needed with a web app, this early in the mentoring system.
Security is a relevant quality to consider, especially since we do not want to create issues for any relevant parties involved. Complete and effective considerations to security for most areas of vulnerability is out of scope for a prototype implementing its core functionalities. Important security details such as account access protection through salted/hashed stored passwords are to be implemented. Data will only be accessed through the REST API, allowing us to ensure only required data is sent, and adds a layer of security since our database is not directly interacted with. Allows us to also validate any user information before storing it in the database.

3.3 Technologies used

A fundamental decision in the project is deciding which technologies should be used to compose the so-called "tech stack". Since this is such an important set of decisions, we carefully considered a number of options with respect to a set of criteria which assess the quality of a component technology choice. These criteria include:

- Suitability, how well does the technology match the problem our system solves?
- Documentation and popularity, does the choice have strong documentation and an active community likely to have already answered problems we may encounter?
- Consistency, how well the choice meshes with the other component technologies?
- Performance, will the technology run sufficiently fast, and can it be scaled later needing to switch to something else more performant?
- Experience, how much prior knowledge does our team have of the technology?

We decided to implement the project as a web application, as opposed to a mobile or desktop only one. This is because it is generic and can be accessed by almost all users, as essentially every modern device has a web browser. However, this design choice must later be taken into consideration when ensuring the UI is suitable for mobile. This choice was made as the system must be widely accessible by users with many different types of devices, and a web application was the only suitable choice for this.

We chose Django for our backend framework, which runs on top of Python. This is suitable for our system as it facilitates fast development, which is required to finish our prototype in the very short turnaround of around five weeks, with very short code sprints, and it inherently supports our non-functional requirements of security and scalability. Furthermore, it has robust documentation and is widely used with an active community. Additionally, Python is so ubiquitous that it can interface consistently with almost any other technology choice, for example persistent data storage and sentiment analysis. Finally, our team has a lot of experience with Python development, and additionally testing techniques for Python, mitigating learning curves and ensuring bug-free code.

Since we are making a web application, we will use HTML/CSS/JS. We decided to use the Vue.JS framework for the frontend JS, as it facilitates writing responsive websites, but has a much less steep learning curve than other frameworks such as React, and is known to integrate well with Django, the backend framework we propose using. Furthermore, we are going to use bootstrap as the CSS framework as it allows us to quickly develop a professional looking website with little hassle by providing building block CSS classes.

We chose PostgreSQL to store our persistent data in the form of a relational database. This is because the type of data recorded by the system is well suited to storage in a relational schema. For example, storing a set of users, each of whom

have various properties and relationships with other users is a very good match for this model. Furthermore, postgresql has very good documentation in comparison to other relational databases, and it meshes well into our tech stack as django and postgresql are commonly used in tandem. Finally, all of our team has experience with it due to the CS258 databases module.

We chose the python library NLTK (natural language toolkit) to implement our sentiment analysis. This is because it is compatible with the rest of the backend framework, which is already written in python, and it is the most popular and widely used library within its field.

We will use JSON for the REST API, as it is incredibly ubiquitous, and links easily into most web frameworks, as it is a subset of javascript. Additionally, both our backend and frontend team members have experience with it, which is crucial as it will form the link between the two sides of the technology stack,

We chose docker for our containerisation. This supports all of the rest of the tech stack, and allows "lift-and-shift", which is a very useful property for prototype systems. Additionally, it is by far the most widely used choice for containerisation, and has good documentation. Finally, none of our team has experience with any form of containerisation, so choices cannot be compared by this metric.

We chose git with GitHub for our version control. This is because it is the industry standard for version control, every member of our team had at least rudimentary experience with it from CS133, and some members of our team had experience with advanced features for CI/CD such as GitHub Actions.

3.4 System architecture

Our system architecture follows the MVC (Model View Controller) architecture which is a well-known model to be successful in creating a system that presents data obtained through a controller that interacts with the database. This is reflected in the docker containerisation figure with our website being the view, the Django system being the controller, and Postgresql as the model. Being modularised to 3 separate components, allows easy distribution of development, with each focusing on a subset of skill sets. One downfall of this architecture is that the system is tightly connected in that if one system fails, the rest falls - however we believe that this is outweighed by the benefits of this architecture. Furthermore, the fact that it is a trusted design pattern suggests it is not an inherently problematic approach. A REST API will be used to serve data to the user, allowing both static site pages and dynamic JSON data to be served from the Django backend.

We decided to containerise our model, as the specification stated that Deutsche Bank already uses containers for this type of application, so it will provide modularity within their existing systems. Furthermore, it gives the property of easy "lift-and-shift", meaning that all the dependencies can be handled within the containers, which themselves are portable and compatible across machines.

We decided against using microservices, instead opting for a monolithic architecture. This was predominantly for two reasons: Django has a higher overhead than other backend frameworks like Flask, so making multiple microservices can reduce performance; and there is not a clean way nor a good reason to split the internal logical model into separate microservices.

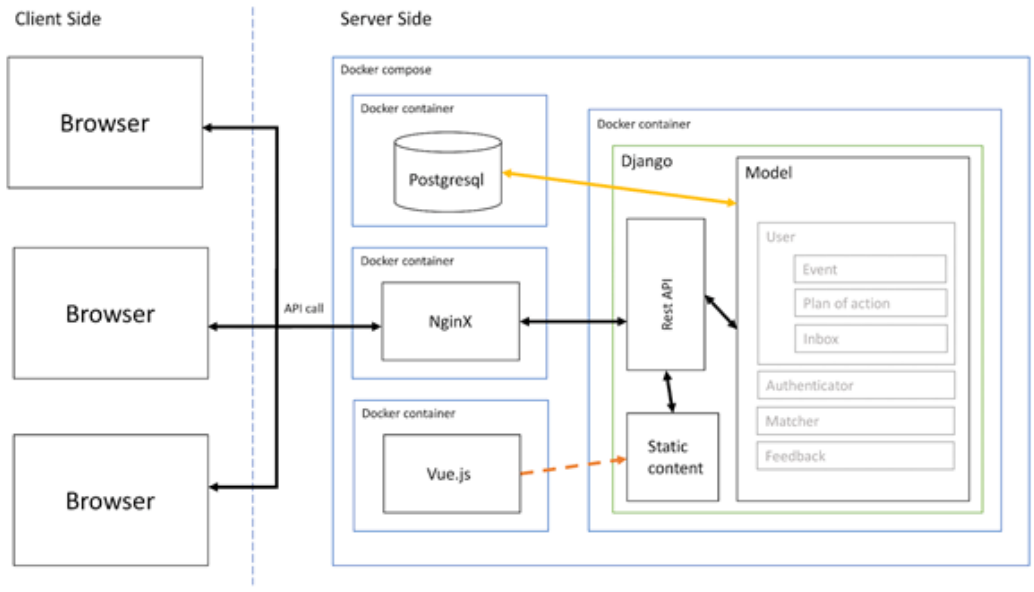

Figure 2: A system diagram of our system architecture.

The containerisation has two distinct parts. The first is the three containers dynamically interact with each other based on user requests, with the NginX container serving requests into the Django backend container, which runs the internal logic, and can look up data from the postgresql database container. The second is the final container which generates the static site content from Vue.js to pure HTML, CSS, and Javascript. This is then copied into the Django container so it only needs to be generated once, not on a request-by-request basis.

3.4.1 Database Design

Figure 3: A entity relationship diagram of our database structure.

3.4.2 Object Structure

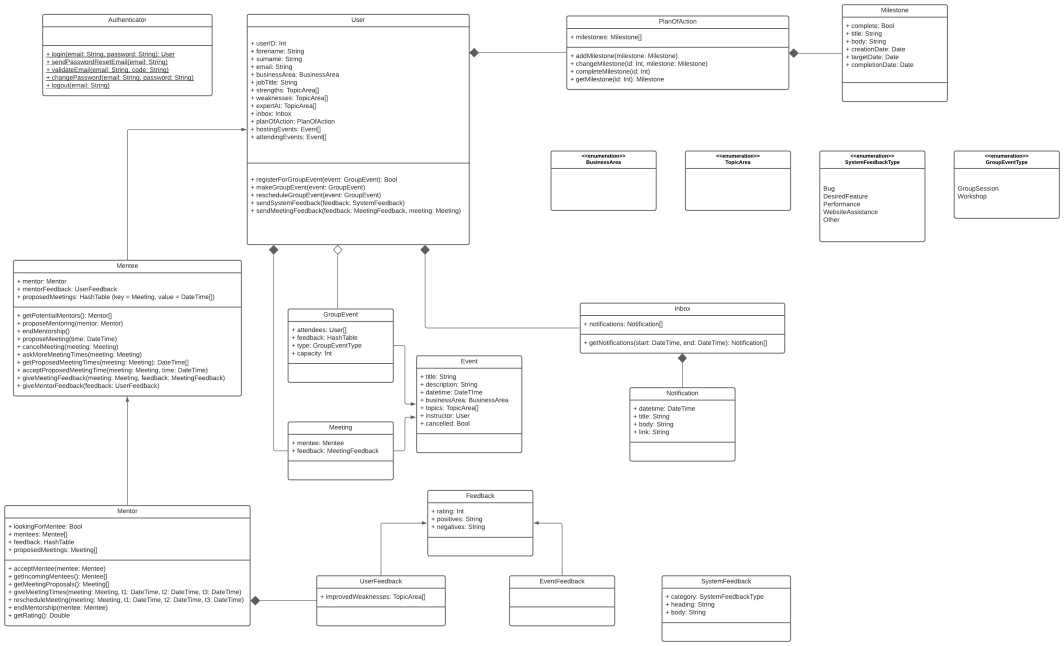

Figure 4: A UML class diagram of our object structure.

3.4.3 API design

We have planned to design an API which will be used by the front-end to interface with the core back-end components of the system. We've decided to settle on a REST API implementation, as it emphasises client-server decoupling, which will make it easier to separate front-end and back-end tasks between our developers—something which is of the utmost importance when working in a smaller, less experienced team. The REST API system will give the front-end developers access to fixed API endpoints, which will act as a bridge between the front-end and back-end of the application. The front-end developers won't need to know how these endpoints are implemented, they only need to know what functionality they provide. This

will save development time as developers can now focus on their specialties. This modular way of designing the API allows for back-end code to be modified without having to consult the front-end developers about potential consequences. As long as the specification is followed, a back-end modification should have no effect on how the front-end developers interact with the system. Overall, this will save time as unnecessary communication between developers will be eliminated.

4 UI/UX design

4.1 Page Hierarchy

The page hierarchy below displays the sites that our webpage comprises of. Here is a brief outline of what each page will consist of:

- Login Screen: The landing page of the website that allows the user to login to go to the dashboard or the sign up page to create an account.
- Sign Up: Allows the user to create an account.
- Dashboard: The central hub for the user where all pages can be accessed from and displays useful information that may be useful (look at page designs for more detail).
- Profile: This page has 3 purposes: resetting the users password, seeing the users data and modifying the users data.
- Plans of Action: Displays the users plan of action and their mentees plans of actions too if they have any.
- Individual POA: If the user clicks on a POA, then they come to this page where the POA can be viewed and modified.
- Schedule: Shows the users schedule, allows them to request meetings and has a list of feedback and meeting requests that the user should respond to.
- Group Events: Allows the user to search for workshops and group sessions within the system that they can attend.
- Individual Event: A page that details an event (meeting or group event) and allows the owner of an event to modify or delete the event.

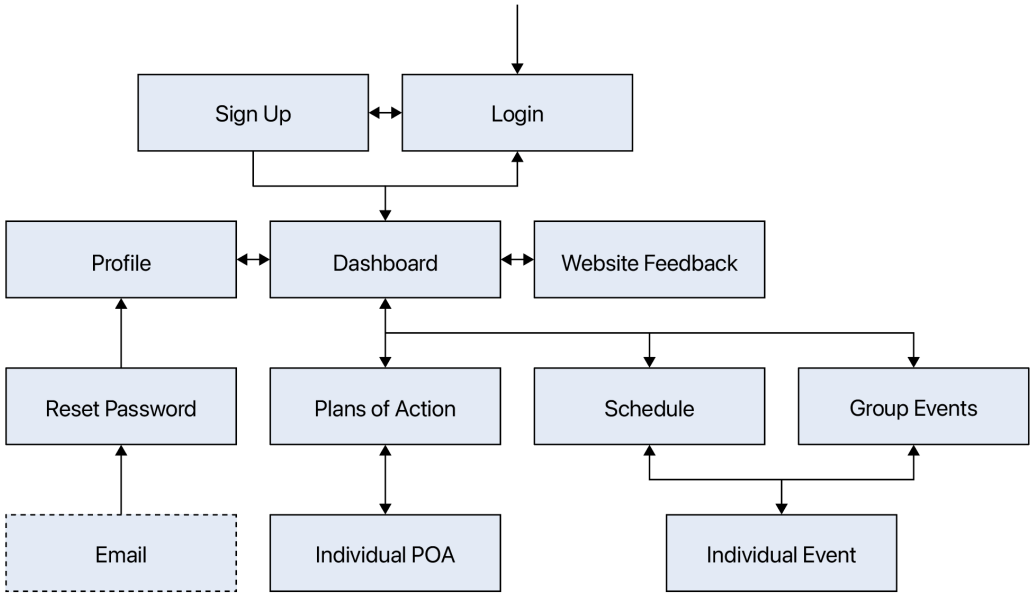

Figure 5: A diagram of website page hierarchy.

4.2 User-system interaction

The specification was also very vague about how to match up mentors and mentees, therefore this diagram clarifies all the ambiguities around this.

- The user can only have 1 mentor at a time
- The system can give the user a list of potential other users who are currently looking for mentees
- The user then picks one of these users and a mentor mentee request is sent to the potential mentor
- If the potential mentor accepts, then a relationship is formed

- Otherwise, the mentee needs to choose another mentor from the list

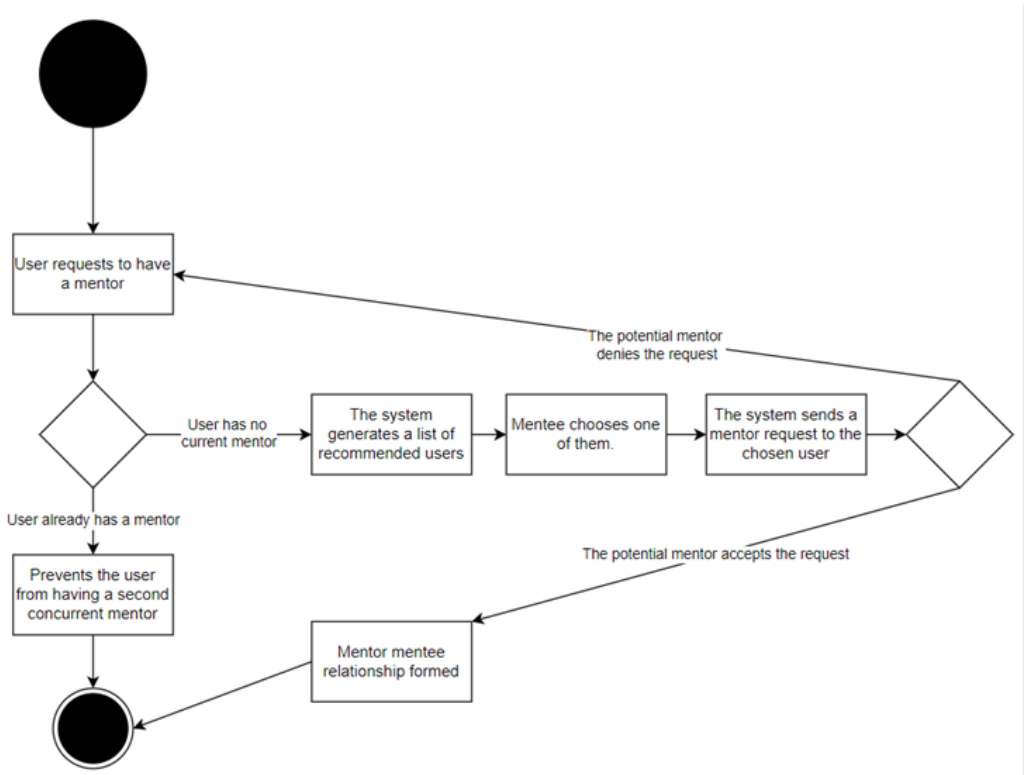

Figure 6: A diagram of the states required to match a mentor and a mentee.

The diagram below displays what functions the Schedule page has. This handles a lot of ambiguities around how the meeting and feedback system will work within our system. The important points from this diagram is that:

- Mentee's can cancel meetings, but mentors can only reschedule
- Mentee's can request a meeting, but mentors cannot not
- For a meeting to be officially approved, both parties need to agree on the time and date otherwise they go into this loop of suggesting times to each other
- The user will be prompted on the schedule page to give feedback to recent events they have gone to

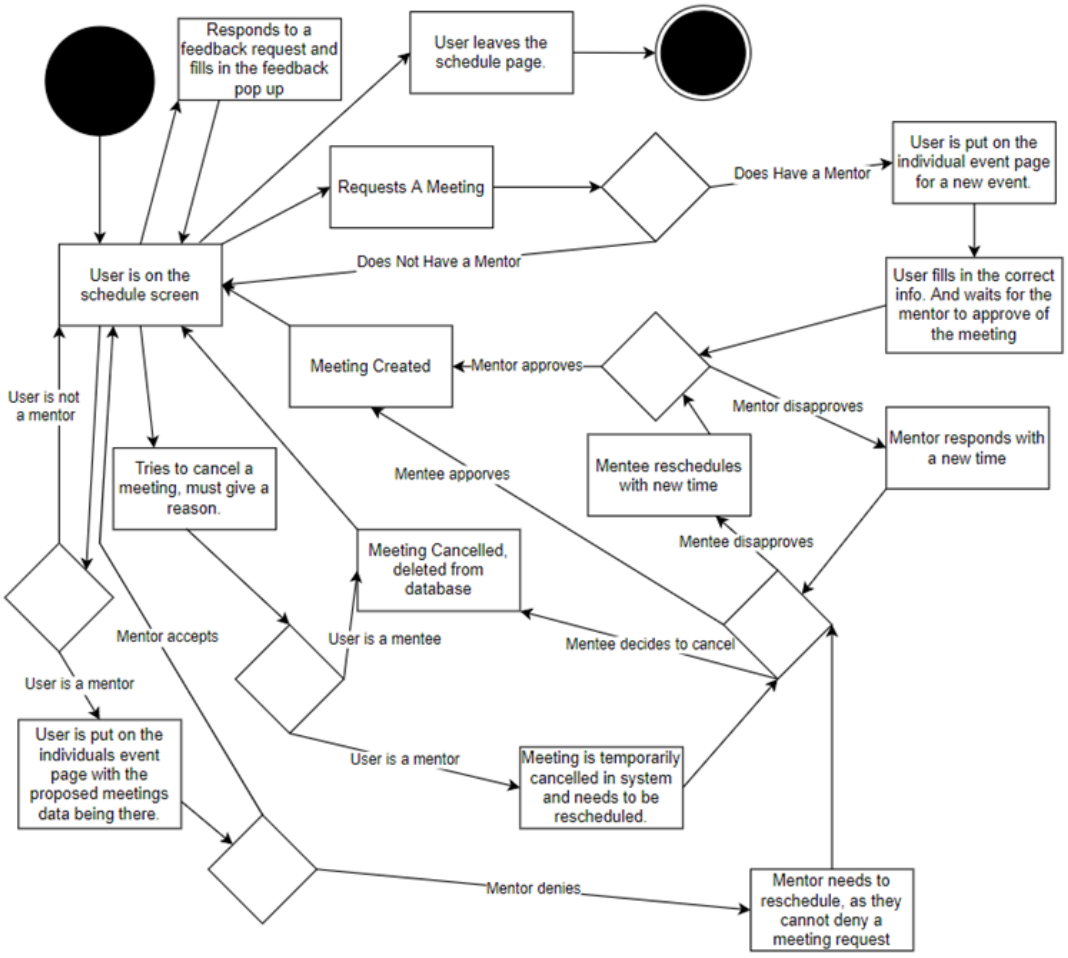

Figure 7: A diagram of the states required to schedule a meeting.

4.3 Page Design

The dashboard is the central page for users that displays all the useful information for the user, such as their upcoming meetings, their current plan of action and allows them to get a new mentee or mentor. The navigation bar at the top of the page will be on most of the pages and will allow the user to access all the other pages of the website as well as their notifications. The notifications inform the user about any recent events that they may be interested in such as giving feedback about a recent event or a mentee/mentor meeting coming up. The drop down button next to the notification bell will link to the user's profile page, the website feedback form and a sign out button.

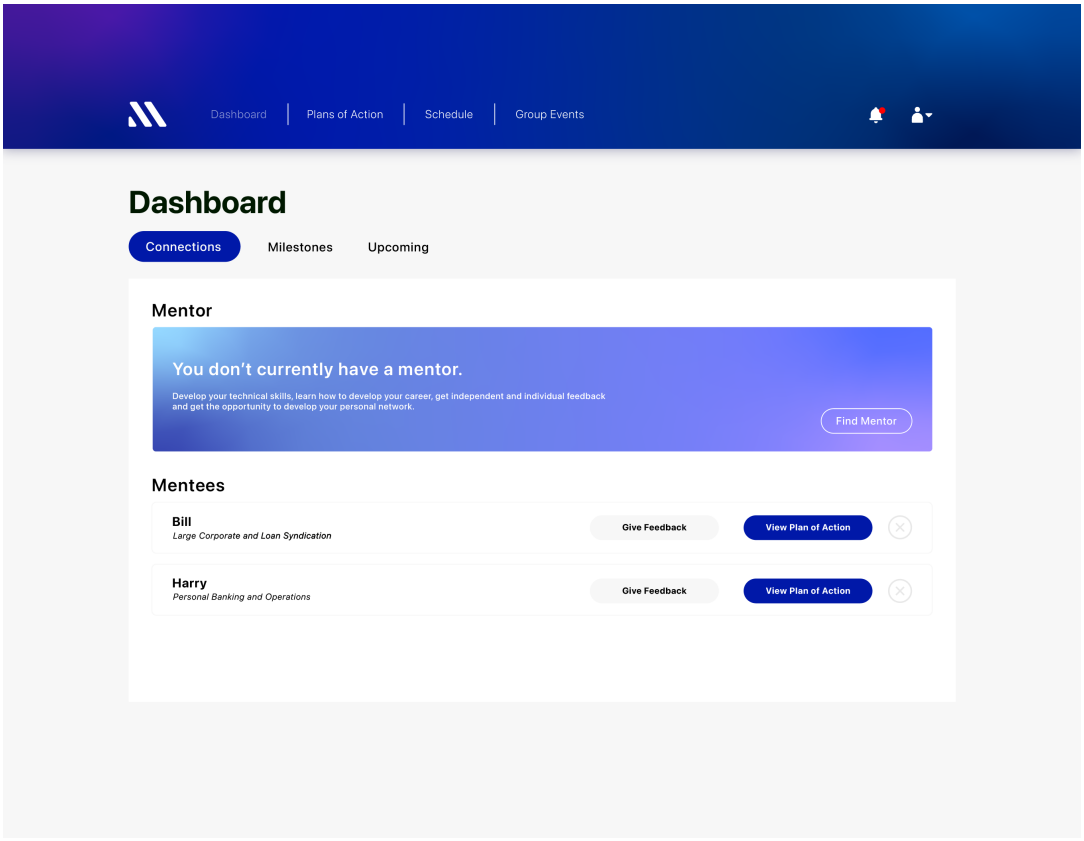

Figure 8: The initial design of our system dashboard.

The schedule page will have the users calender and all the requests the user has (meeting and feedback). This user is not a mentee so the request meeting button at the bottom is currently missing, but if the user was a mentee then there would be a request meeting button at the bottom. Additionally, clicking on a meeting will take the user to the individual event page for the meeting which will have the feedback log for the meeting.

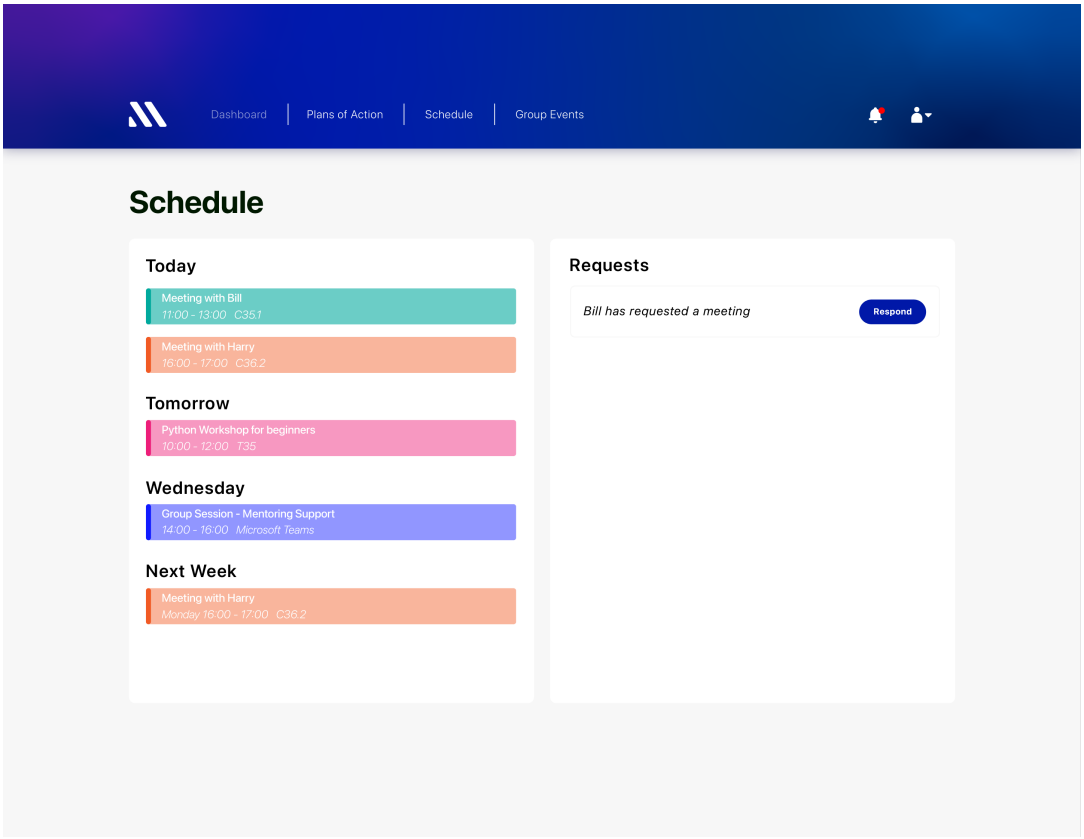

Figure 9: The initial design of our system schedule page.

5 Testing

5.1 Test cases

5.2 Unit and integration testing

Tests will form an integral part of our development cycle, as we plan to follow the agile test driven development ideologies. We will implement this by creating unit tests for all of the above test cases, allowing us to ensure all the required properties of the system are met. These can then be run throughout the development process. All of the tests will be automatically run on a pull request to the GitHub repository, and must pass in order for the code to be merged into the main branch, to ensure its validity. This is an implementation of "continuous integration" from CI/CD, and will be implemented using GitHub Actions.

There are many tools for unit testing which can be used. Since we are using a python backend framework, we will use 'PyTest' for unit tests on the backend code, along with 'codecov' to assess the coverage of the tests, and 'pylint' to ensure that stylistic code is being written. Since we are using a javascript frontend framework, we will use 'Jest' for unit tests on the frontend code and UI, along with 'ESLint' to check that the Javascript is correct. Finally, we will use Postman to test that the API serves the correct data. There are relatively few tools for integration testing, and it will likely mostly have to be done by hand. We will use the 'Puppeteer' tool to automate this task as far as possible. The combination of all of these tools allows us to create a testing suite which will fully cover our system.