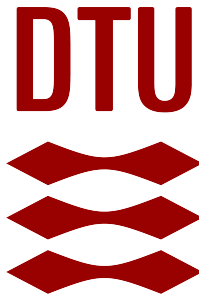


TECHNICAL UNIVERSITY OF DENMARK



02122 SOFTWARE TECHNOLOGY PROJECT

Group 1 - FullStack Development

Thursday the 23th of June

[Tobias Collin	s183713]
[Jacob Hartmann Martens	s204443]
[Simon Peter Sundt Poulsen	s204445]
[Christopher Zwinge	s204459]

Abstract (Everyone)

This report describes the process of designing and creating a fullstack application that allows diabetes patients and their medical health professionals to monitor the patients' glucose levels. It explores using the Vue framework to make a frontend, Java Spring Boot to make a backend, MySQL Database to store user data, and Swagger RESTful API to define the interaction between the other services.

Glossary of Terms

- **Patient** : A user of the program that has diabetes
- **Doctor** : A user of the program who is a healthcare professional
- **Admin** : The user of the program with administrative rights
- **Artificial/Bionic pancreas** : A medical device that monitors blood glucose levels and injects
- **User** : The people and systems that interact with the application
- **SPA** : A Single Page Application refers to a webapp that dynamically changes what it displays instead of moving between different webpages
- **Measurement** : A json object containing glucoselevels, bolus, basal and carbohydrates
- **ERD** : Entity Relation Diagram
- **Mount** : A part of the Vue lifecycle right after initial rendering. Appendix B
- **DOM** : Document object model
- **Model** : Context dependent, but may refer to a class that models a database schema, or a model in the model-view-controller principle.
- **Wrapper** : Used to denote an object or a method that changes the behavior or style of the underlying object.
- **View/Page** : The object that shows content on a website
- **Container** : Object containing the styling for elements on a page
- **Component** : Object which is used on Pages to display additional content
- **Parent** : A view or component that displays another component is the parent of that component
- **Child** : A component that is used in another View or component is the child of that component

- **Props** : Information that is sent to a component from a parent View or component
- **Emits** : Information that is sent from a child component to a parent View or component
- **Router** : Vue router module which is used to display different content on a page
- **Routing** : Sending information to the router to change the content of the page
- **The App** : App.vue component
- **User data** : Data such as the users first and last name as well as an id and password
- **User type** : The user's type which is either patient, doctor or admin
- **Glucose level/concentration** : Blood sugar level
- **GET** : HTTP entity for getting a json object which has been designed in springboot
- **Weak Entity** : In ER diagrams: An entity that takes a foreign key as its id
- **Total participation** : In ER diagrams: E.g. an Entity requiring another Entity to exist.
- **MockMvc** : Springboots model view controller test support.

Contents

Abstract (Everyone)	i
Glossary of Terms	ii
Contents	iv
1 Introduction (Collin)	1
2 Problem analysis (Martens)	2
3 Goals and success criteria (Zwinge)	3
3.1 Goals	3
4 Design process and considerations	5
4.1 Use case diagram (Poulsen)	5
5 Design architecture	7
5.1 Database design (Poulsen)	7
5.2 Agile Framework (Collin)	9
5.3 Material Design (Collin)	10
5.4 Program Structure	10
5.5 Development Tools (Collin)	12
6 Implementation	14
6.1 Backend (Java Springboot) (Poulsen)	14
6.2 Frontend (JavaScript Vue)	15
7 Testing (Collin)	23
7.1 Springboot direct testing	23
7.2 Manual testing	23
7.3 Postman	24
7.4 Vue testing	24
8 Project management Tools (Collin)	26
8.1 Time problems	28

8.2	Work distribution	29
9	Personal Development	30
9.1	Tobias Collin	30
9.2	Jacob Martens	30
9.3	Christopher Zwinge	30
9.4	Simon Poulsen	31
10	Discussion	32
10.1	Frontend refactoring (Martens)	32
10.2	Models and controllers (Martens)	32
10.3	What could be done better	32
11	Conclusion	34
A	User login system	35
A.1	Previous login implementation	35
A.2	Current login system	37
B	Lifecycle of Vue	41
C	Design sketches	42
D	Use Case Diagram	49
E	Class Diagram	50

CHAPTER 1

Introduction (Collin)

Diabetes has become a more common medical condition in the world. A diabetic person's body is unable to regulate the blood sugar by itself, which is why diabetic people need to inject insulin on a daily basis to maintain a stable blood sugar concentration.

This has to be measured compared to the amount of glucose in the blood. Which is further complicated by the ingestion of food which has glucose in it. Individuals with type 1 diabetes often have a glucose sensor that can measure the glucose concentration in the body, as well as an artificial pancreas (See glossary) that can regulate the concentration by injecting insulin when the blood glucose levels gets too low.

In order to inject the right amount of insulin at the right time., it is necessary to observe and analyse the data generated by the glucose sensor.

People with diabetes and healthcare professionals need a tool to view and manage the data related to the chronic health condition and the doctors need an efficient way to handle the patients who suffer from it.

CHAPTER 2

Problem analysis (Martens)

This project will be a full stack system for healthcare professionals (later referred to as doctors) and people with diabetes (later referred to as patients) which aims to create a web application for patients and doctors. The patients should be able to view the data in visually friendly graphs, and for the doctors, the goal is to be able to manage their own patients and view the patients' data.

This gives rise to many complications which need to be handled correctly. The data has to be visually detailed yet simple enough for a non-professional (patient) to look at and interpret.

The data will also need to be checked for correctness to avoid faulty data. This has to be handled before loading the data into the application. The data will also need to be permanently stored for later use, and the same applies to the user credentials of the doctors and patients.

This project addresses the problem with a user-friendly GUI with data visualisation and user management tools, which will be coded with JavaScript, HTML and CSS by using the Vue 3 framework. The data will be stored in a MySQL database and made accessible to the frontend through the Java framework Spring Boot. The interface that will be used to build the data bridge between the frontend and backend will be a Swagger RESTful API which protocolises the transfer of the data to ensure that the data will be sent as intended.

The primary users of this system will be patients and doctors. The doctor should have access to the data of the patients, and the patients, whom the project primarily focuses on, should be able to view their current glucose levels visually on graphs, and the patient should be able to register when the patient eats/has eaten a meal to allow the doctor and patient to get ahead of handling the fast increasing glucose level which the patient will likely experience after consuming a meal. The application will also need a way manage the user system; that is to add, edit and remove patients and doctors.

CHAPTER 3

Goals and success criteria (Zwinge)

3.1 Goals

The goals of this project can be split into two categories; the program goals and the learning goals. The program goals are the goals for the program that will be the result of the project. The learning goals are the goals for learning to use new programs, programming languages and other software.

3.1.1 Program goals

MoSCoW

To specify the specific goals of the program we made a MoSCoW table. The table is made by splitting the possible features of the program into must-haves, should, could and won't. By splitting the features into these categories, we get several of the features the program must have to be called a success as well as many optional ones which would make it better.

MVP

From the MoSCoW table a simple description of the Minimum Viable Product(MVP) could then be made. The program must have some manner of permanent data storage, it must be able to show a glucose level graph for a patient and it must have a dashboard. These are the basic criteria for the MVP. On top of that we should implement most if not all of the features from the should column of the MoSCoW table and some of the extra features from the could. The success criteria for the program goals is therefore to have a working MVP.

3.1.2 Learning goals

The learning goals for the project is to learn to write JavaScript, HTML and CSS for use in webapp development, specifically to use with the Vue framework. Furthermore another goal is to learn to use MySQL to setup databases for permanent storage and

Must	Should	Could	Won't
Permanent Data Storage	Doctor info & Patient info	Bluetooth/NFC	Mobile app
Glucose level graph	Login	Notifications (doctor to patient or app to patient)	Food picture scanner
Dashboard	Bar chart for glucose levels	Chat	Password encryption in the server
	Average levels of glucose over a (month)	User inputs and registrations (of meals)	
		Lookup tables for food products	
		Dark/light mode switch	
		Exercise registration	
		Customizable background (upload image)	
		System administration/admin (add and update doctors)	
		Emergency function	

Table 3.1: MoSCoW model for the project

to learn to use the Springboot framework to access the database. The success criteria for these goals is whether or not the MVP is finished. This is because we'll need to have learned at least to a certain degree all of the goals to be able to finish the MVP.

CHAPTER 4

Design process and considerations

4.1 Use case diagram (Poulsen)

In order to model how the user would interact with the application, a use case diagram was made early in the development (see Appendix E):

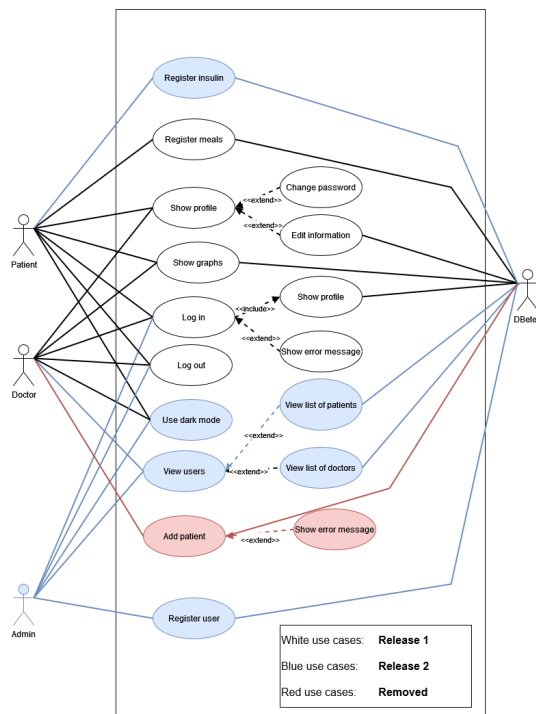


Figure 4.1: Use cases

The diagram shows the features to be implemented and shows the changes that were added as development progressed. There are three primary actors: The patient, the doctor and the admin. The outgoing lines from actors show association; which features the actors may interact with. For example the patient should be able to log in and out, but not register users. The secondary actor DBetes is the database, that only provides the data when requested. E.g. when the patient requests to view its profile information the database should provide that data only when requested to do so.

4.1.1 Design drafts (Poulsen)

After making the use case diagram different designs for the GUI were drafted and discussed (see Appendix C). This is when the idea of having a sidebar and topbar for ease of navigation was conceived. The sidebar was considered particularly useful for navigating between pages because of its extensibility; a new button could be added when a new page was added.

4.1.2 Integration of the admin user (Martens)

Initially, the application could only be used by doctors and patients such that the doctor would be responsible for adding patients and other doctors to the system. (REF. Use case diagram release 1) This seemed like an acceptable solution at first, since the main focus of the project wasn't the administration of the users but rather the administration of the diabetes data.

Nevertheless, as more administrative features were added to the application, the main focus of the doctor role, namely to administer patients, was diminished. Hence, the admin user was added to assume responsibility for adding and editing users. (REF. Use case diagram release 2)

CHAPTER 5

Design architecture

5.1 Database design (Poulsen)

The database was designed by an ERD (Entity Relationship Diagram) and went through multiple iterations, the most prominent of which are shown in **Figure 5.1** and **Figure 5.2**.

The final diagram consists of three entities: The Patient, Doctor and Measurement entity. The entities have various attributes. For example the patient has the attributes: id, being its primary key, first and last name and password, age being an inferred attribute signified by the dotted lines around the attribute name.

The Patient-Doctor relationship was decided to be a N:1 relationship. It was discussed whether this should be a N:M relationship instead, to allow a patient to have multiple doctors, but ultimately the simpler N:1 relationship was chosen as this better models the real world patient-GP (general practitioner) relationship.

As for the Patient-Measurement relationship, the 1:N relationship was chosen, as a patient can have many measurements. The measurement entity is a weak entity and is identified by the patient id and a timestamp for when the measurement was taken. In the final ER diagram the decision was made to cut the exercise attribute for simplicity's sake, as this attribute would have the same functionality as carbohydrates, but it could easily be added in future iterations of the application.

During the final stages of the design process, the decision was made to cut the alert entity of the ERD (see **Figure 5.1**), as this was deemed of lower priority according to the MoSCoW described in section 3. At the same time, it was decided to remove the total participation of the Patient in the Patient-Doctor relationship to allow for patients to be created without a doctor at first and to be assigned a doctor at a later time. This decision was made to account for situations where a doctor might be removed from the database, in which case the patient should remain.

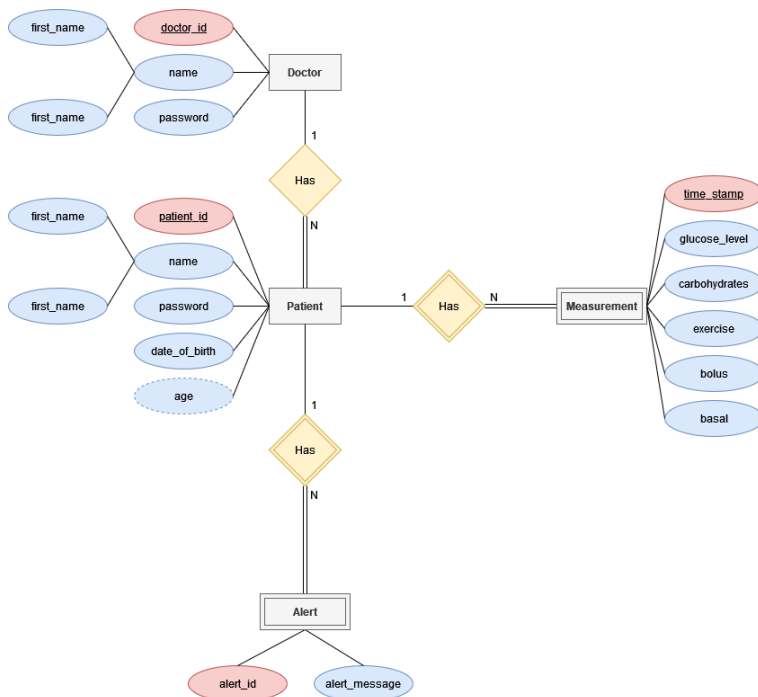


Figure 5.1: Draft of ER diagram

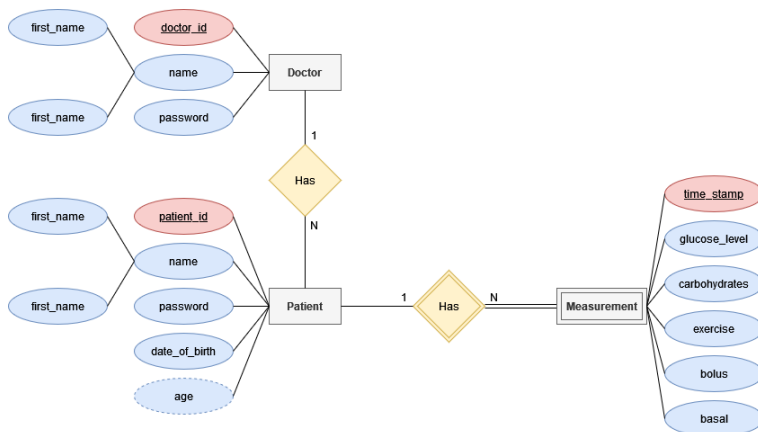


Figure 5.2: Final ER diagram

5.3 Material Design (Collin)

A major component of looking at a nice webapp is understanding how it's done. A good guideline of how to make a webapp comes in the form of material design ¹. While the main focus of the source is mobile apps, there is a subsection on websites which shows good ideas to make a presentable website.

5.3.1 Colors

A color rule called 60-30-10 ² was used as an guideline into how much area each of the visual colors should take. This helps give an overall good color scheme for once website. This made us have 3 main colors to work with, with a few extra colors for highlighting and text.

5.3.2 Layout

The idea from material design which leant most into the design was of all interfaces being in a container with a different color so it's easier for people to know which parts should be looked at and which are only a part of the background.

5.4 Program Structure

5.4.1 Model View Controller (Collin)

The program can be split into a model view and controller part. The model for the fullstack program would be the backend except for the controller classes which are part of the controller. That is both Java Spring Boot and the database written in MySQL. The view is the frontend of the application which in this case is JavaScript, HTML and CSS with the Vue framework. The controller between these two applications is the API, which is JSON formatted requests which can be sent and received by the model and view to send information between the different parts. The user can see the frontend page, send request when needed through the JSON to manipulate the database. From there the backend is a model and controller, while the frontend of the app is it's own design.

5.4.1.1 Frontend

The frontend has been split into views and components. Views are pages and their functionalities. Some of the views uses components which are normally large components with many functions or a single component which is needed many times. The

¹<https://material.io/>

²<https://www.thespruce.com/timeless-color-rule-797859>

frontend uses the API inputs from JSON request in some components to control the data and the show it to the user. At the same time, the user can input in the page to send an request back to the backend.

5.4.1.2 Backend

In the backend the springboot handles the keeping of data and how the data is represented as the model of the backend. The controller and repository classes act as the controlles of the backend. Controlling that the view interacts correctly and specifically with the model and updates to the user the data of the model. The API calls then go out from the springboot application to the frontend.

5.4.2 Frontend structure (Martens)

The frontend source code has been structured such that each file belongs to a category. The categories are apparent in the folder structure, as each folder represents a category.

The assets folder contain the non-scripts, which, in this project, is exclusively .svg-files used for the icons and images in the App.

The components and views folders contain all the .vue-scripts which are responsible for all the visual aspects of the application. The only exception is the App.vue script, which is the top level container for the application, and hence it is placed directly in the src-folder together with the main.js script which is responsible for mounting (REF. Mount) the app in the DOM (REF. DOM)

The model folder and the services folder contains the frontend models (REF. model) and frontend "controllers" respectively. Note that controllers in this context is used to denote the scripts that controls some of the general behavior of the frontend. Only the userController.js and backend.js scripts are behaving as actual controllers as defined by the MVC principle.

5.4.2.1 Patient / Admin / Doctor structure (Martens)

As the size of the frontend increased and more pages were added, the need for a better structure became essential. Thus, the views folder was split into a patient, doctor and admin folder which would contain the pages solely related to their respective type. Only the scripts available for all three user types (patient, doctor and admin) remained in the top level views folder; namely the landing page and the "404 page not found" page.

5.4.2.2 Login (Zwinge)

Since the views had now been completely split it became apparent that some kind of login would be needed to determine which type of user was using the app. Furthermore the login would be necessary to have multiple different patients and doctors and be

able to display the relevant data for those. Therefore login pages were added to the design, one for each of the different type of users.

5.4.2.3 Components and views (Martens)

The patient/admin/doctor restructuring of the views folder disallowed the use of the same page for different user types. E.g the patientUserPage and doctorUserPage both display information about the user currently logged in. Although this seems to be a disadvantage, it is in reality considered a good code of conduct as it makes the code more extensible and thus easier to maintain. This is because the pages are simply used as containers and wrappers for the content. The contents of the pages are primarily the components which can be reused and mounted on any page, and the pages also allow for the use of multiple components without having to change the behavior of the individual components. That being the case, the frontend becomes more flexible and easier to extend.

5.5 Development Tools (Collin)

5.5.1 Frontend tools

Vue

Most of the frontend was made in the frameworking tool Vue, more specifically Vue 3.0. The main reason Vue 3 was taken compared to Vue 2 is that Vue 3 was promised as a fast, easy to run website. The main tradeoff is that Vue 3 is quite new as the programming language came out at the start of the year. That means that many packages have not been updated to Vue 3 and therefor not usable. Though as a focus the group decided this as a learning experience of how to learn Vue, and not its many extensions. Therefore it was decided to try and ignore as many as possible of the extensions. This made the overall progress slower but the understanding of the fundamentals better.

Extensions to Vue

In the frontend to receive API request the tool axios was used. Axios was chosen since it has plenty of examples and some documentation to go with it. This made it ideal as a starting tool. It's main competitor was API Fetch, which seems less used and not as well described, this lead to the use of axios instead.

The packages chart.js and vue-chart.js are working together to help show the graphs in the program. They are made to make it easier to display visual data. With Vue-charts.js being a wrapper for chart.js. This is because chart.js does not work natively with Vue.

The last special package that was used was called Vue router. This help by segregating the views into distinct pages that can be routed to. This had the effect of allowing a segregated view and component style of program. This allows us to have a Single Page Application (SPA)

5.5.2 Backend tools

Spring Boot

Spring Boot handles the backend. It is a java extension that allows spring applications. Spring application work in in the idea of database management by allowing java to handle the request coming from an API and handle them as specified in the code to either store, retrieve or search through.

Scheduling

Since the data from a diabetes insulin pump would be in intervals of 5 minutes. A mockup system has been made that uses scheduling. This allows data from a sample file to be sent at an interval just like it would be from a insulin pump, to simulate said pump.

MySQL

MySQL is used as the programs database since it's used for entity-relation databases. Beyond that the tool is also one of the most widely known database tools. The last benefit is that springboot is already build to handle a potential MySQL database connected with it.

Swagger API

Swagger is an online API which allows you to send JSON request to and from a recipient. In this case, that would be the backend Spring Boot application. The main usage is manual testing and manual usage of the API.

5.5.3 RESTful API

An Application Programming Interface or API is making a connection between a webserver or computer to a web application. A RESTful API has one main advantage compared to a normal API: RESTful API's make the frontend and the backend independent of each other and therefore makes an abstraction layer which protects each part from interference by the other. So no manipulation can happen to the webserver from the webapp.

CHAPTER 6

Implementation

6.1 Backend (Java Springboot) (Poulsen)

The final database design (see **Figure 5.2**) was implemented using Java Spring Boot. The backend consists of three parts: the models, the repositories and the controllers. The models consist of classes annotated by `@Entity` which maps the class to a database table. For example, there's a patient class with fields: id, first-Name, lastName and doctorId annotated by `@Column` specifying that these fields are mapped to the patient-table's columns in the database. The JPA repositories work as an automatic part for storage, retrieval and search of an emulation of a collection of objects. Which in this case is the model and the controller. This has the effect in Spring Boot of being the glue that makes the other parts communicate. Compared to CRUD repositories, JPA has some extra features which could be called to make use of Spring Boot testing and other services which are not used. The controllers respond to incoming API requests by making the models available to the frontend. Below is a class diagram showcasing the general structure of the backend (see appendix E):

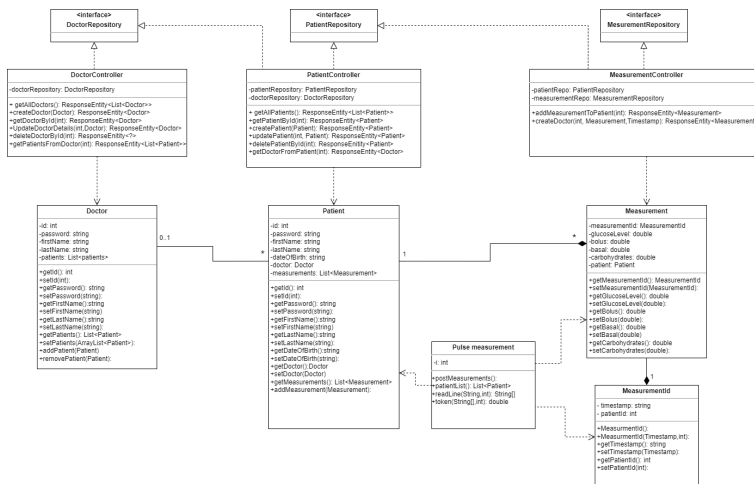


Figure 6.1: Backend UML Class diagram

Bidirectional Patient-Doctor relationship (Poulsen)

The Patient model has a Doctor field annotated by `@JsonBackReference`¹. This is to prevent infinite recursion since the Doctor model has a list of patients. To illustrate the problem, consider the following example of a Doctor object with a list of Patients:

```
Doctor: [Patient: Doctor: [Patient: Doctor: [Patient: Doctor[...]]]
```

The issue with this annotation, however, is that it prevents getting a doctor from a patient. In these cases a solution could be implemented by either searching the doctor's list of patients for a patient with matching id, or by replacing the Doctor object in the Patient model with a doctor id, and then searching the list of all doctors for a matching id. Neither of these changes were made however, due to time constraints, as the problem was discovered late in the development process, when trying to populate the patient profile in the frontend with its information including the doctor's information.

Scheduling (Collin)

The way the program uploads at a specific time interval is through a schedule made by a springboot component that is automatically run when needed and calling on the right controllers to run its code. This is all handled automatically by the springboot application. The way it works is through an cron object which takes the pc's time and uses it to know when it needs to update. This happens in this programs case every time the minute clock can be divided evenly with 5, making for a 5 minute interval.

The data it uses are loaded from a csv file, which has a lot of example measurements. These measurements are divided by row and column. Each row is used by a specific patient id and each column used as how far along the data the program has gotten.

6.2 Frontend (JavaScript Vue)

6.2.1 Navigation with Vue router (Zwinge)

To be able to navigate between pages without having to enter a different path for each page or having to change the entire page Vue router extension is used. Vue router is assigned a path and a name for each of the different views. It can then be accessed in each view and through the use of buttons or commands a new path can be pushed to the router to change the currently displayed content. This allows the app to change pages without having to refresh the page. Without this feature it would have been necessary to implement some kind of cookie or other storage for the browser to keep track of the user session.

¹Jackson - Bidirectional Relationships:

<https://www.baeldung.com/jackson-bidirectional-relationships-and-infinite-recursion>, visited June 23 2022 15:00.

6.2.1.1 Topbar and Sidebars (Zwinge)

A topbar and a sidebar was implemented to make the navigation in the app as quick and simple as possible. The sidebar was made to navigate between the specific pages that each user type had available. For the patient, the sidebar has a graph page and a meals page (Also acts as an insulin submission page) and the sidebar therefore has those two symbols. (see **Figure 6.2**) The doctor only needs to be able to see its patients and therefore doesn't have a sidebar. The Admin mainly needs four pages, one for viewing the patients, one for viewing the doctors, and another two for adding patients and doctors to the system. These four pages are shown in the sidebar as a dropdown menu for patients and a dropdown for doctors which then show the view and pages. (see **Figure 6.3**) The topbar is the same for the Patient and the Doctor having a home button which is the app's icon as well as a darkmode toggle switch and a profile button which opens a dropdown. The dropdown allows the user to go to the profile page, the change password page and a logout button. The admin topbar is slightly different as it doesn't need the dropdown so the profile button is instead replaced with a logout button. (see **Figure 6.4**)



Figure 6.2: The patient sidebar buttons

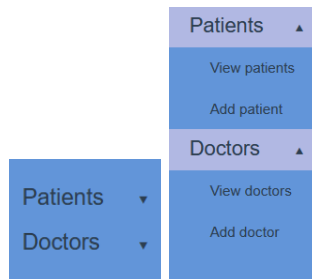


Figure 6.3: The admin sidebar dropdown



Figure 6.4: The two different topbar's. *Left is for patient/doctor and right is for admin*

6.2.1.2 Login Control (Zwinge)

The topbar and sidebars once more reiterated that it was necessary to keep track of if the user was logged in and which type of user that was using the program. It was also important to keep track of the user's Id when logged in, to be able to display the content specific to that user such as the differing sidebars and topbars. Therefore the login pages for each user had to be created along with a login control system. For the first iteration we implemented these properties in our main app page and updated them through emitting the login from each login page. For this system the user submits a id and a password and those are then checked with the database to see whether they match. If they matched we would directly route them to the next page and emit the login information back to the main app. This worked for a simple login but made the user data difficult to access as it would have to be sent from the main app to whichever page needed it. Furthermore anyone was still able to access any page by writing the page path because the router couldn't access the login status from the main app. Therefore a separate file was created which held the information and which the main app updated whenever the data was updated. (See implementation in Appendix A.1)

6.2.1.3 Navigation guards in router (Zwinge)

When the router became able to access the user status, navigation guards were introduced in the router. Navigation guards simply mean implementing some logic to reroute whoever tries to access a page if they don't have the right credentials. To check whether a page required login to access we implemented a meta value for each of the possible routings in the router. From then on whenever someone without credentials tried to access pages that required login we simply routed them back to our landing page. This also works in the opposite direction, if a user that is logged in tries to access a page that doesn't require login they are redirected to the main page for that user type.

6.2.1.4 User Table (Poulsen)

The doctor's site differs from the patient's, due to the lack of a sidebar. This being replaced with a table component. This decision was made due to the doctor and patient having different requirements for what information they should be able to access. Since the doctor only needs to be able to navigate between his patients, a table is used for navigation. The table component is made to be as reusable as possible, also being used for the admin's patient list and doctor list, by taking props from the pages that let the component know which columns to display and it emits

a patient when clicking on a row to be able to redirect to the correct patient's page. The table also has support for basic searching and sorting for the application user's ease of navigation.

6.2.2 Meal & insulin page (Collin)

The page sends information provided by the patient to the database. This comes in three values: Carbohydrates, bolus & basal. The patient then has the option of choosing a time of the day to update with new information. The main special part of the implementation is that a get mapping is first used to get the old values. Then if the patient does not set some values, they won't be set to zero in the update to the time. This is mainly only a problem because the new input values is controlled through the drop down menus, which set's them at default of zero.

However, due to time constraints the data registered is not shown in a graph. The idea was to see when and where the patient had inputted these values.

6.2.3 Graphs (Poulsen)

The blood glucose level graph adds points to the graph for a week after which it begins to calculate the confidence intervals (see **Figure 6.6**. The confidence intervals can be calculated in the following way:

$$\bar{x} \pm 96 \cdot \frac{\sigma}{\sqrt{n}}$$

Where \bar{x} is the mean, σ is the standard deviation.

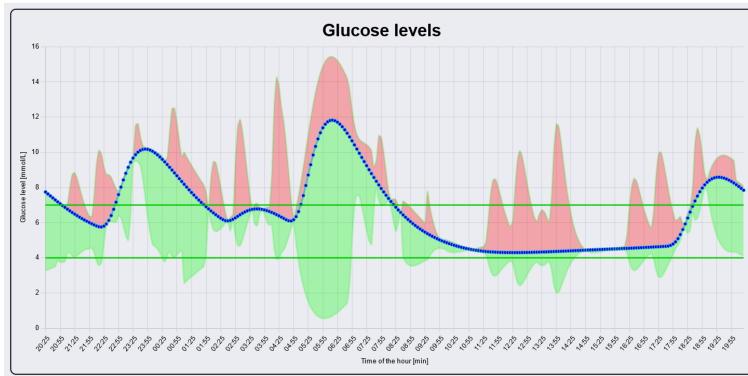


Figure 6.5: Glucose level graph

6.2.4 Darkmode (Collin)

Darkmode is a switch which uses an onclick function to change from light theme to dark theme or vice versa, beyond this it also intercepts at the app startup if the theme is a dark-theme on your browser and immediately makes it set to that theme. This comes at the cost of the system being able to access the main page, which has left the code in the main app. The program then sets the variables in a CSS component to a hashcode for a color, which is then used by all other components by calling said component, therefore making a theme. A snippet can be seen in the following for setting a property in the style part.

```

20 /*
21  * Author: Tobias Collin
22  */
23 document.getElementById('app').style.setProperty("--primary-color", '#424242');
24 document.getElementById('app').style.setProperty("--secondary-color", '#212121'
25 );
26 document.getElementById('app').style.setProperty("--accent-color", '#747474');
27 document.getElementById('app').style.setProperty("--highlight-color", '#B4B4B4'
28 );
29 document.getElementById('app').style.setProperty("--strong-text-color", '#911
30 did');
31 document.getElementById('app').style.setProperty("--text-color", '#DDDDDD');
```

Listing 6.1: Setting darkmode

These variables are then changed to another set of values when the switch is clicked. A note to make is that the variables are never declared in the style part of the Vue component, they become declared in the script part. The variables can be called upon as seen in Listing 6.2

6.2.4.1 Page wrapper (Martens)

Since most of the application is designed and styled in a similar fashion where the content would be centered around the middle of the page, and the topbar and sidebar would be placed in fixed positions, it made sense to declare a styling that would apply to all views from a single point in the program. This was achieved by declaring CSS classes globally in App.vue which could be applied to the outer most <div> element in all the views. (See Listing 6.2.)

```

63 .page-container {
64     position: absolute;
65     top: var(--topbar-height);
66     left: var(--sidebar-max-width);
67     width: 100%;
68     height: 100%;
69 }
70 .page-wrapper {
71     position: relative;
72     display: flex;
73     flex-direction: column;
```

```

74 |     justify-content: center;
75 |     align-items: center;
76 |     width: calc(100vw - var(--sidebar-max-width));
77 |     height: calc(100vh - var(--topbar-height));
78 | }

```

Listing 6.2: page-wrapper and page-container from App.vue

In order to avoid writing a `<div>` wrapper and container in all the page-scripts, the `<div>`'s could be placed around the `<router-view>` components as this would be the location where the router module would mount the pages when prompted. In the App.vue script, only the page-wrapper is used as this centers the content. (See Listing 6.3.)

```

2 | <div class="page-wrapper">
3 |   <router-view />
4 | </div>

```

App.vue

Listing 6.3: page-wrapper used in App.vue

And in the patientSite.vue, doctorSite.vue and adminSite.vue scripts, both the page-wrapper and page-container are used as they, respectively, center the content and limits the size of the pages to be within the borders of the sidebar and topbar. (See Listing 6.4.)

```

3 | <div class="page-container">
4 |   <div class="page-wrapper">
5 |     <router-view />
6 |   </div>
7 | </div>

```

patientSite.vue

Listing 6.4: page-wrapper and page-container used in patientSite.vue

6.2.5 User model and -controller (Martens)

While coding the frontend and implementing multiple login systems for the admin, doctor and patient, it became a necessity to save the user data (REF. user data) and user type (REF. user type) after logging in as a means to use the user data in other components.

The initial login system, which was explained in Section 6.2.1.2, was replaced by a user model and a user controller. The user model holds the loggedIn-status of the user, the type of user and the user data. Hence, the user model creates an abstraction for the three user types as the same instance can be used for all three types. Meanwhile, the user controller instantiates a user from the user model class, and the user controller class contains methods for retrieving (getters) and replacing (setters) the fields of the instantiated user. The user controller is made globally available to all Vue components by instantiating it as one of the App's global properties (in main.js)


which means that all components can get direct access to the `logIn` and `logOut` methods by importing the controller as opposed to the previous system where logging in and out was controlled by the App.

By using this controller, the attributes of the user model are not directly exposed to the application, which provides a layer of safety when accessing the user data as it can only be accessed through the methods defined in the user controller. Thus, the user model can safely be extended with both private and public attributes and methods without risking an erroneous leak of data to the application. (See implementation in Appendix A.2).

6.2.6 Dynamic requirements directive (Martens)

For some of the input fields where certain requirements had to be met in order to proceed with an action, the requirements are shown as red text to let the user know that some requirements are not satisfied. (REF. `patientForm`, maybe insert image)

These requirements are dynamically updating by using a computed property (REF.



The image shows a 'Patient Submission Form' with a light blue background. At the top, it says '*All fields must be filled out.' in red. Below that, it says '*Password must be at least 6 characters long and contain at least 1 uppercase letter and 1 special character or number.' in red. The form contains the following fields: 'First name:' with a text input, 'Last name:' with a text input, 'Password:' with a text input, 'Date of birth:' with a date picker showing '23 / 06 / 2022' and a close button, 'Doctor:' with a dropdown menu showing '--Select a doctor--', and a 'Submit' button at the bottom.

Figure 6.6: Patien submission form

computed) that checks when a dependency updates; in this case the data variables that the input fields model. The computed property works as intended, but the code

also takes up a lot of space in the script, and the same code has been reused in multiple components to display dynamic requirements. To avoid this inconvenience, the idea was to create a custom requirement directive. The Vue framework allows custom directives to be declared and used in the App, which makes it possible to easily apply behavior and styling to DOM elements (REF: custom directives). The custom directive is currently written as a simple styling directive which applies a specific font to the text in the HTML element where it is used.

```
2 const requirement = {
3   mounted(el) {
4     el.style.fontFamily = 'verdana';
5     el.style.fontSize = '12px';
6   }
7 }
```

directives.js

Listing 6.5: Custom requirement directive in directives.js

The idea for the more advanced directive which would apply all the behavior needed to replace the computed property is as follows:

- The directive would be applied to input fields
- The value passed to the directive would be the id of the HTML tag, where the requirements would be shown
- The text explaining the unsatisfied requirement could be passed as a dynamic argument
- To verify when the text in the input field satisfied a requirement, modifiers could have been used (a modifier for each type of requirement, e.g. NoNumbersAllowed)

(REF: <https://vuejs.org/guide/reusability/custom-directives.html#hook-arguments>) The implementation of this directive was however deprioritised to save time for working on more important features.

CHAPTER 7

Testing (Collin)

7.1 Springboot direct testing

We first started with using springboot's inbuilt testing framework. These tests used the inbuilt function to find and make the objects with a response, which could be tested on. The main problem with these tests was that they were in a random order and they needed extra code in the controllers to handle the request directly in the backend. This made it unattractive.

```
1  @Autowired
2  private MockMvc mvc;
3  @Test
4  public void getaDoctor() throws Exception{
5      createTestPatient("john", "doe", "password", "22.12.21");
6      mvc.perform(MockMvcRequestBuilders.get("/api/v1/patients")
7          .accept(MediaType.APPLICATION_JSON))
8          .andDo(print())
9          .andExpect(status().isOk())
10         .andExpect(MockMvcResultMatchers.jsonPath("$.firstName").value("
11             john"))
12         .andExpect(...) // shorten down
```

Listing 7.1: Springboot Test

As can be seen in this test, a custom function to make a test person is needed if none exist and you need to make a MockMvc that has special functions to test and to append in a Media type. All these things for making a test and then if there are multiple tests. The run order of the test was in an none too obvious order of starting put mappings before post mappings and so on. Which facilitates the need for already having data in the server which is the same as in the test, for proper testing of the API.

7.2 Manual testing

From there an assortment of manual tests of posting deleting and so forth in a specific order to check for bugs, this led to some errors showing up as the program was near-

ing it's final stages since not all cases were covered, leading to some heavy technical debt's. ¹. When that was not sufficient anymore a new testing suite was looked at.

7.3 Postman

The new suite of test came from the program postman which is a program for handling api's, which also had a test suite which was used in the later parts of the project to test API's and how they were handled. The main benefit of postman was that it did not interact with the backend, but instead with the Json API commands instead, making an abstraction layer between the backend and the test. This made sure they would not impede each other. From there some assessments were used to make sure the api's handled as predicted, which test on some get and post and put methods.

From here you would get a failed or passed on the test, where you could then manually check the body of in this case the get method to see if it was operating as claimed, beyond sending the right response.

7.4 Vue testing

When it came to testing the front end, with it's different components, we deemed it not needed since how interconnected it was with the view, which is best tested manually by looking and observing how the program handles itself, to see if the components and views are placed correctly or not.

¹https://en.wikipedia.org/wiki/Technical_debt

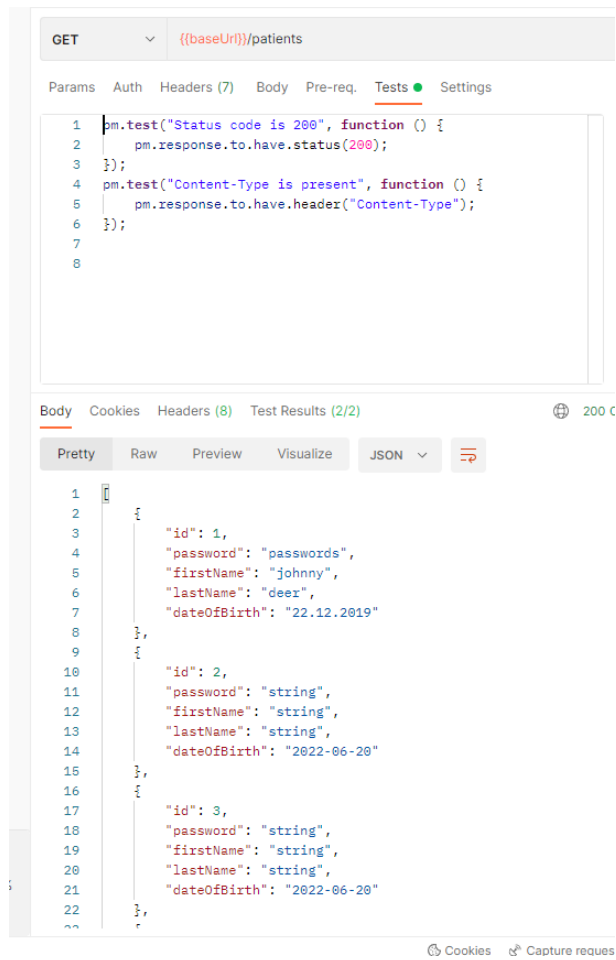


Figure 7.1: postman test get all patients

CHAPTER 8

Project management Tools (Collin)

8.0.1 Gantt chart

The group started by using a gantt chart to keep control over when and where the group progress had been made, this came down to how many days where planned to be needed to make the design, implementation and so on. This was a general idea to see how far we had made it, compared to how much was left to be worked upon.

A large section at the start of the project was dedicated to learning and understanding the many nuances of full stack development, all from frontend to backend development.

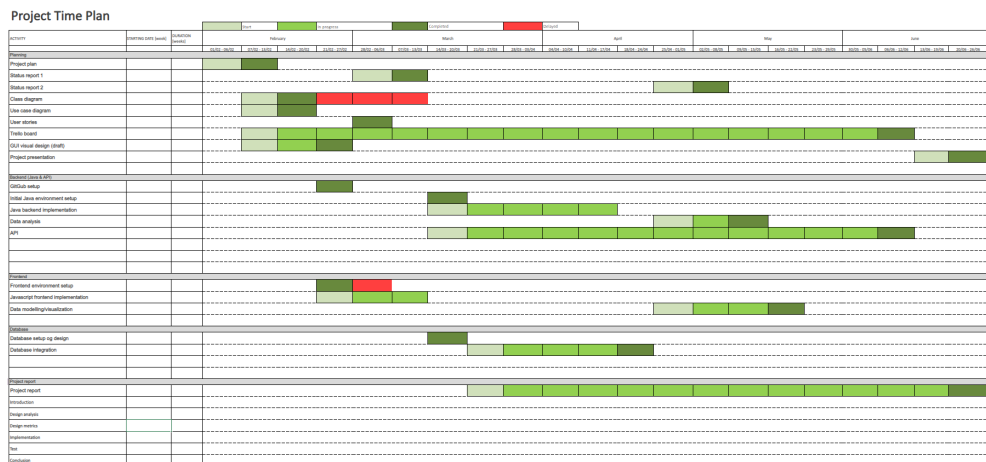


Figure 8.1: Rough timeplan made in gantt

As can be seen in the top of the figure different colors where for if we were ahead, behind or on time.

8.0.2 Worklog

As we started upon implementing different components a worklog was made to keep track on when a specific task was done, and to see if people were keeping up with the implementations needed to fulfill the timeplan as laid out in the gantt chart.

Date	Work done	Participants
Sun 13/03	Group status meeting.	All
Wed 16/03	Status meeting with Paul.	All
	<ul style="list-style-type: none"> - Implemented router functionality to sidebar buttons. - Implemented a simple <u>topbar</u> layout. - Made the sidebar available to all sub-pages except the login-page. 	Jacob + Zwinge
	<ul style="list-style-type: none"> - Started the API interface - Setup of Spring Boot - Made a version 1 of the E/R diagram 	Tobias
	<ul style="list-style-type: none"> - Made a version 1 of the E/R diagram 	Simon
Wed 23/03	Status meeting with Tobias and Sarah.	All
	<ul style="list-style-type: none"> - Changed sidebar component to include buttons with icons instead of text 	Jacob + Zwinge
	<ul style="list-style-type: none"> - Further Development of the API - First iteration of the database - Visual design of the database corresponding to 2NF normalization rules. 	Tobias
	<ul style="list-style-type: none"> - SQL 	Simon
Wed 30/03	<ul style="list-style-type: none"> - Visualized the initial design and layout of the app 	All
	<ul style="list-style-type: none"> - Finished initial layout and functionality of sidebar (Added 4 initial pages mapped to the sidebar buttons) ((Done since last week)) - Implemented a simple container for settings window 	Jacob
	<ul style="list-style-type: none"> - Finished initial layout and functionality of <u>topbar</u> - 	Zwinge
	<ul style="list-style-type: none"> - ER diagram ver. 2 - Base implementation of <u>mySQL</u> database - Design <u>draft</u> for patient list (Sat 03/04) 	Simon
	<ul style="list-style-type: none"> - API for many to many - More API calls - <u>Biglookup</u> for the authentication system and to check if your logged in. 	Tobias

Figure 8.2: Worklog of 3 weeks at the start

As can be seen in the figure, it is separated into 3 parts. The date, what work

has been done this week and lastly who had done it. This was to get a general feel of what people had done.

8.0.3 Trello

Trello board was made for the last intensive 3 weeks, where in the task where all written up and a time added for when it should be finished, this way we could keep track of if we ever fell behind the work schedule, also people were assigned to these task so the workload could be split as evenly as possible. As the project developed certain task were reassigned or worked on together to make the workload more even.

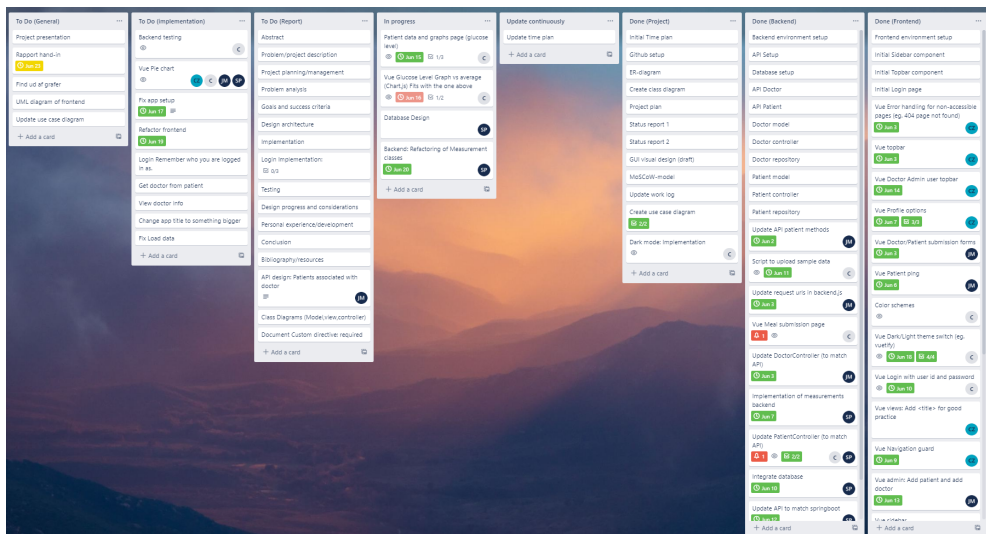


Figure 8.3: Trello

The first group is what is still being worked on, while the second part is what has been made. Some of them have people who are responsible for getting it done with a deadline added for when the task could be expected to be finished.

8.1 Time problems

The main problem was our inexperience with the medium of fullstack development which lead to some late changes that took time and manpower. These changes did make an overall better product. However the plan was flawed from it full inception. More experience was needed at the start to have a better idea of how to start it. Also

that the initial API structure and testing was not good enough meant some rework and bugs cropped up in the later parts.

Admin edit function

Due to time constraints, it was not possible to implement a component for the admin to edit patients and doctors. The tables redirect to an empty page. This component could easily be implemented at a later time, since the redirecting is already handled.

8.2 Work distribution

The main idea was to make it so every member had a part in the frontend, backend and API. So every member was responsible and could understand all parts of the program. Some people specialised in certain parts of the program, but the main idea was still used.

CHAPTER 9

Personal Development

9.1 Tobias Collin

The main reason this topic was chosen, was that i had little to no experience in fullstack development, so it was a good way to dip my toes in how it works. Since the group and I insisted on keeping it basic, it lent to learning a lot of the foundation which could then possibly be used when i graduated from university and began working at a future company. Beyond that it gave me an insight into how to make databases, operate them and make a visual interface that could be accessed through a web browser. Beyond that how to handle server request.

9.2 Jacob Martens

To be working on a large project such as this was a challenge at several levels. It was difficult to manage the project and make preparations for the implementation without any idea of how long it would take or how difficult it would be to code each part of the application. I especially learned how important it is to distribute the work and give each (or at least a few) group member(s) a role such that different people are responsible for making sure that the report is updated continuously, the code is structured properly, everyone has something to work on, etc.

I didn't learn much about setting up the database, only how to decide and model the relationships between the tables, but I did learn a lot about setting up the backend using SpringBoot, writing a matching API, and I especially went in depth with the frontend environment, learning HTML, CSS, JavaScript and the Vue framework.

9.3 Christopher Zwinge

We chose this project mainly because none of us had any experience in fullstack development and it seemed interesting to learn something new. Us having little to no experience made the project difficult to properly plan and make the project but we learned a lot about how to make large application. I personally learned a lot of HTML,CSS, and JavaScript and only some things about API's and MySql as there simply wasn't time to learn everything.

9.4 Simon Poulsen

This project has been a good experience to get acquainted with full stack technologies. I had not worked with frontend development, APIs or database technologies before, so there were many new concepts to grasp before implementation could begin. While there was not time to become very proficient in everything, it was enough to get an understanding how one might go about creating a large webapplication with many different components.

I also learned the importance of planning a software project of this scale, as the first few weeks could have been more efficient due to lack of experience with large group projects like this.

CHAPTER 10

Discussion

10.1 Frontend refactoring (Martens)

Had there been more time, the frontend would have been refactored to make it more extensible. Primarily the structure of the views (REF. view) and components (REF. component) could have been refactored so that the views would only contain components and wrappers, and components could have been made more extensible by allowing props (REF. props) or emits (REF. emits) to change simple behavior and design aspects which would have decreased the amount of reused code sections.

10.2 Models and controllers (Martens)

Some components contain a lot of similar data which could be modelled as a single object to limit the amount of different functionalities in a single component/script. A great example of this is the glucose data in the GlucoseGraph component. The GlucoseGraph component is not only responsible for displaying the graph but also for fetching the data from the database, loading the data into arrays for the points on the graph, calculating the confidence intervals and updating all this every fifth minute. Most of this could have been moved to a measurement model which, just like the user model, could have been instantiated on a measurement controller such that the GlucoseGraph component could simply call the right methods in the controller to load the data into the graph.

10.3 What could be done better

10.3.1 Data generation (Poulsen)

One way the data could have been loaded into the database was using statistical bootstrapping¹ to randomly generate new data based on the example data from the CSV file containing glucose level measurements provided for the project rather than

¹Bootstrapping (Statistics):
[https://en.wikipedia.org/wiki/Bootstrapping_\(statistic\)](https://en.wikipedia.org/wiki/Bootstrapping_(statistic)),
visited June 23 2022 16 : 00.

loading the data chronologically when Spring Boot is launched. This might have been possible using the language R, but it would have required more time to set up than was left when this idea was discussed. Furthermore, the work required to do so might not have been worth it since the data generation is purely for demoing the data visualization component of the application; it *would* make the confidence intervals look more realistic, but that would not matter, as in a real-world scenario the data would come from some device interfacing with the application anyway.

10.3.2 View component structure

A more rigorous splitting of views and components could be done. A good example is the login pages which still has functionality in the view page itself instead of in a component which has not been made. However doing this make it so a view page would hold nothing but a component. So the main drawback is a less well defined border between a view and a component, instead only big or reused part are made into components. Which makes some views handle data while in theory they should not.

CHAPTER 11

Conclusion

(Everyone) Over the course of the project the group has learned how to use many different languages and tools for fullstack development, such as making a webpage with the Vue framework, HTML and CSS, defining a RESTful API and making a database. The end result of the project is a working webpage which contains a database which gives it permanent data storage, a frontend which can display several different pages for the three user types. This includes a glucose level graph for each patient and a dashboard. Furthermore a backend and an API which connect the frontend and the database. This already fulfills our MVP. On top of that, there is a patient and doctor page, a login system for each user type, a dark and a light mode with a switch, system administration in in form of an admin that can add patients and doctors. Together all these features create a simple but useful website and more than fulfills our success criteria.

APPENDIX A

User login system

A.1 Previous login implementation

The login implementation before the user model and user controller was implemented was made as shown in the following code sections.

The 'logged in' status could be globally set in the variables.js script.

```
1  /*                                                                 variables.js (old)
2  * Responsible author: Christopher Zwinge
3  * Contributors:
4  */
5  const loggedInStatus = {
6    isLoggedIn: false,
7
8    get getStatus() {
9      return this.isLoggedIn
10    },
11
12    set setLoggedIn(val) {
13      this.isLoggedIn = val
14    }
15  }
16
17  export { loggedInStatus }
```

Listing A.1: Previous login system in variables.js

The App would be responsible for showing the right components depending which user was logged in.

```
1  <!--                                                                 App.vue (old)
2  * Author of login implementation: Christopher Zwinge
3  -->
4  <template>
5    <div id="appcontainer">
6      <div id="appLayout" v-if="app.loggedIn||app.loggedInAsDoctor||app.
7        loggedInAsAdmin">
8        <Topbar @showDropdown="showDropdown.isVisible=!showDropdown.isVisible"
9          @darkMode="toggleDarkmode()" />
10        <Sidebar v-if="app.loggedIn" />
11        <AdminSidebar v-if="app.loggedInAsAdmin"/>
```

```

10     </div>
11
12     <div id="app">
13         <router-link to="/"></router-link>
14         <router-view :User="User" @logIn="logIn()" @logInasDoctor="logInDoctor()"
15             @logInasAdmin="logInAdmin()" />
16     </div>
17
18     <div id=showDropdown v-if="showDropdown.isVisible">
19         <ProfileDropdown :app="app" @logOut="logOut()" />
20     </div>
21 </template>

```

Listing A.2: Previous login system in App.vue (template)

The App would keep local variables to hold the 'logged in' status, as well as provide the methods for logging in as each of the different user types and log out.

```

45 /*                                                                 App.vue (old)
46 * Author: Christopher Zwinge
47 */
48 app:{
49     loggedIn: loggedInStatus.getUserStatus,
50     loggedInasDoctor: loggedInStatus.getDoctorStatus,
51     loggedInasAdmin: loggedInStatus.getAdminStatus
52 },
53 User:{
54     id: 0,
55     Name: "Full name",
56     Doctor: "Doctor name",
57 },
58 darktheme: null,
59 }
60 },
61 methods:{
62     logIn(){
63         this.app.loggedIn = true
64         loggedInStatus.setUserLoggedIn = true;
65     },
66     logInDoctor(){
67         this.app.loggedInasDoctor = true
68         loggedInStatus.setDoctorLoggedIn = true;
69     },
70     logInAdmin(){
71         this.app.loggedInasAdmin = true
72         loggedInStatus.setAdminLoggedIn = true;
73     },
74     logOut(){
75         this.showDropdown.isVisible = false;
76         this.app.loggedIn = false;
77         loggedInStatus.setUserLoggedIn = false;
78         this.app.loggedInasDoctor = false
79         loggedInStatus.setDoctorLoggedIn = false;
80         this.app.loggedInasAdmin = false

```

```
81     loggedInStatus.setAdminLoggedIn = false;
82   },
```

Listing A.3: Previous login system in App.vue (script)

A.2 Current login system

In the current login system a user is created for use in the userController and it has the necessary attributes to know if the user is logged in, it's user type and it's data, it also has a field for our darkmode.

```
2  /*                                                                 user.js
3  * Responsible author: Jacob Martens
4  * Contributors:
5  */
6  constructor() {
7      this.isLoggedIn = false
8      this.type = ""
9      this.darkTheme = null
10     this.data = null
11 }
12
13 getType() {
14     return this.type
15 }
16
17 setType(type) {
18     this.type = type
19 }
20
21 getData() {
22     return this.data
23 }
24
25 setData(data) {
26     this.data = data
27 }
```

Listing A.4: Current login system in user.js (script)

In the usercontroller we can check each of the users attributes and set them by calling the functions within.

```
6  /*                                                                 userController.js
7  * Responsible author: Jacob Martens
8  * Contributors: Christopher Zwing
9  */
10 static isUserLoggedIn() {
11     return user.isLoggedIn
12 }
13
14 static getDarkTheme() {
```

```

15     return user.darkTheme
16 }
17
18 static setDarkTheme(val) {
19     user.darkTheme = val
20 }
21
22 static login(userType, userData) {
23     user.isLoggedIn = true
24     user.setType(userType)
25     user.setData(userData)
26 }
27
28 static logout() {
29     user.isLoggedIn = false
30     user.setType("")
31     user.setData(null)
32 }
33
34 static changePassword(password) {
35     user.data.password = password
36 }
37
38 static getUserType() {
39     return user.getType()
40 }
41
42 static getUserData() {
43     return user.getData()
44 }

```

Listing A.5: Current login system in userController.js (script)

In the individual login pages the user's attributes are updated through the userController.

```

1  /*                                                                 doctorLogin.vue (new)
2  * Responsible author: Tobias Collin
3  * Contributors: Christopher Zwinge for adding the user information to the
   userController
4  */
5  data() {
6      return {
7          input: {
8              id: null,
9              password: "",
10             wrongInput: false,
11             wrongInputString: ""
12         }
13     },
14     methods: {
15         async login() {
16             if(this.input.id != null || this.input.password != "") {
17                 let data = await this.getDoctorData()

```

```

19     if(data != null && this.isLoginValid(data.password)){
20         console.log("Logged in succesfully")
21         this.$router.push("doctorSite");
22         this.$UserController.login("doctor", data) /* HUSK AT SÆTTE USER
                DATA */
23     }else{
24         this.input.wrongInput = true,
25         this.input.wrongInputString = "Username and/or password was wrong";
26         console.log("Username and/or password was wrong");
27     }
28     } else {
29         this.input.wrongInput = true,
30         this.input.wrongInputString = "Username and/or password was empty";
31         console.log("Username and/or password was empty");
32     }
33 },
34 async getDoctorData() {
35     let data;
36     await this.$axios.get(this.$backend.getUrlGetDoctorById(this.input.id))
37         .then(response =>{
38             data = response.data
39         }).catch(() =>
40             this.input.wrongInput = true,
41             console.log("hello")
42         );
43     return data
44 },
45 isLoginValid(password){
46     console.log(password);
47     return password == this.input.password
48 },
49 back(){
50     this.$router.push({name: "landing"});
51 }

```

Listing A.6: Current login system in doctorLogin.vue (script)

The site shown no longer checks if the user is logged in instead only the content necessary for the current type of user is displayed.

```

1  <!--                                     doctorSite.vue (new)
2  * Responsible author: Jacob Martens
3  * Contributors:
4  -->
5  <template>
6      <title> Doctor site </title>
7      <div class="page-container">
8          <div class="page-wrapper">
9              <router-view />
10             </div>
11         </div>
12         <Topbar />
13     </template>

```

Listing A.7: Current login system in doctorSite.vue (template)

Instead Navigation guards have been introduced to redirect away from content that requires login.

```
97 /*                                                                 router.vue (new)
98 * Author of code snippet: Christopher Zwinge
99 *
100 */
101 {
102     path: '/doctor',
103     redirect: { name: 'patients' },
104     component: doctorSite,
105     name: 'doctorSite',
106     meta: {
107         requiredLoggedIn: true
108     },
109     .....
```

```
183 .....                                                                 router.vue (new)
184 router.beforeEach((to) => {
185     if (to.matched.some(route => route.meta.requiredLoggedIn)) {
186         if (!UserController.isUserLoggedIn()) {
187             return { name: 'landing' }
188         }
189     } else if (to.matched.some(route => !route.meta.requiredLoggedIn)) {
190         if (UserController.getUserType() == "patient") {
191             return { name: 'patientSite' }
192         }
193         else if (UserController.getUserType() == "doctor") {
194             return { name: 'doctorSite' }
195         }
196         else if (UserController.getUserType() == "admin") {
197             return { name: 'adminSite' }
198         }
199     }
200 })
```

Listing A.8: Current login system in router.vue (template)

APPENDIX B

Lifecycle of Vue

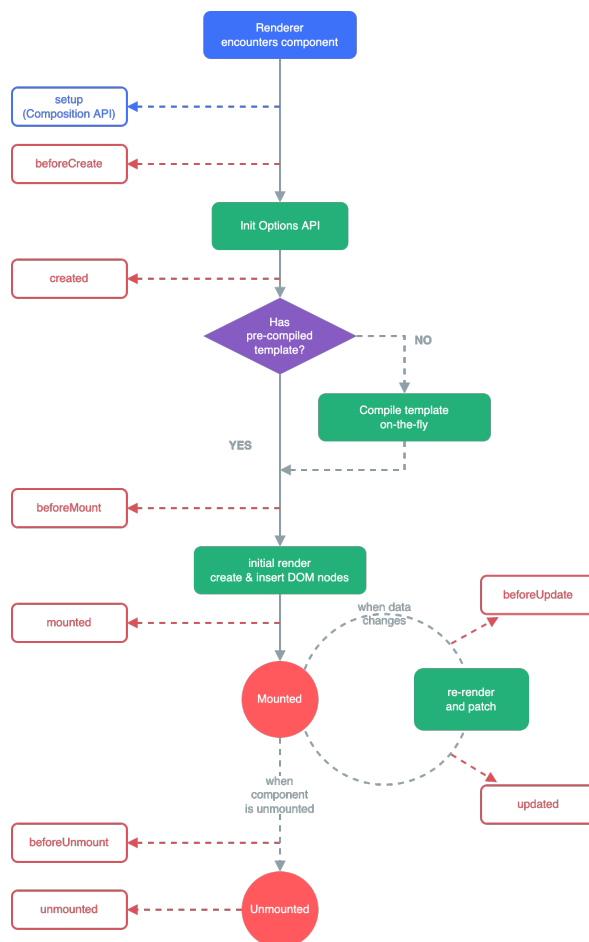


Figure B.1: Lifecycle of Vue

Image borrowed from: vuejs.org/guide/essentials/lifecycle.html#lifecycle-diagram

APPENDIX C

Design sketches

Login

User id

Password

Figure C.1: App sketch 1 Christopher Zwinge

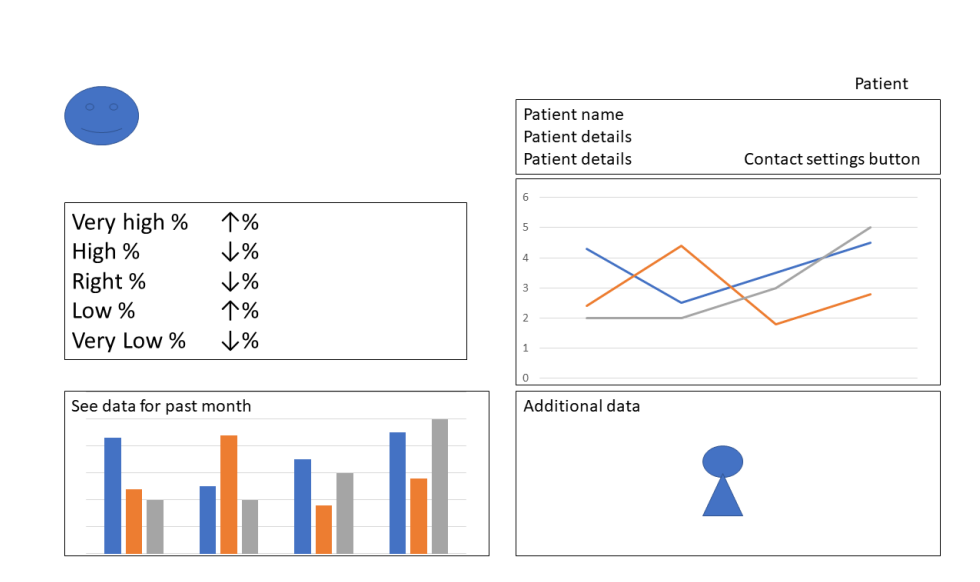


Figure C.2: App sketch 2 Christopher Zwinge

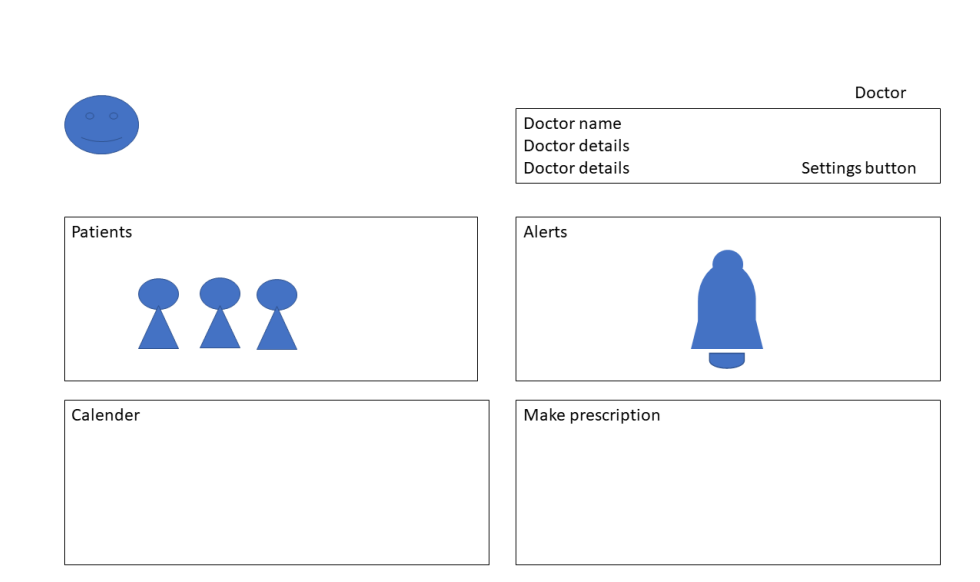


Figure C.3: App sketch 3 Christopher Zwinge

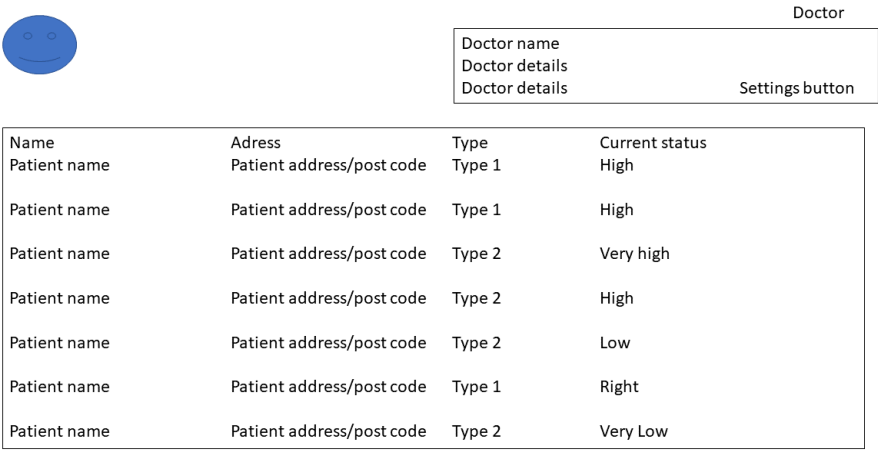


Figure C.4: App sketch 4 Christopher Zwinge

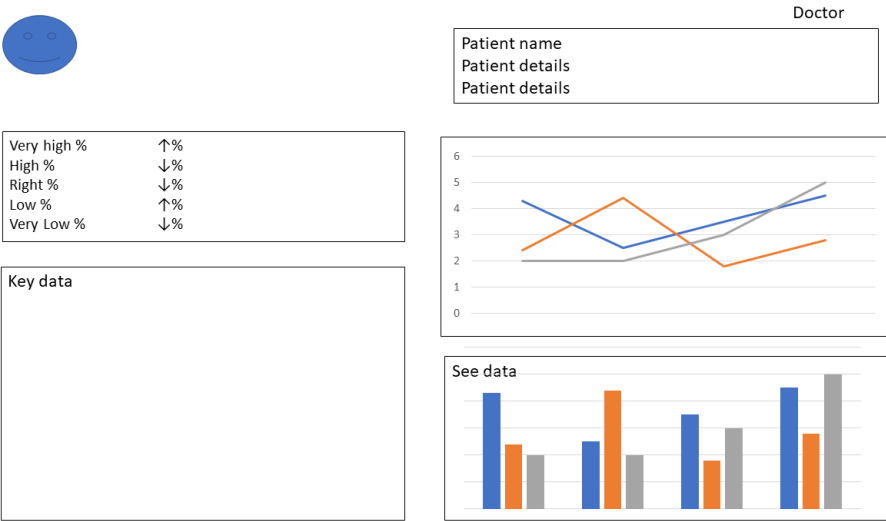


Figure C.5: App sketch 5 Christopher Zwinge

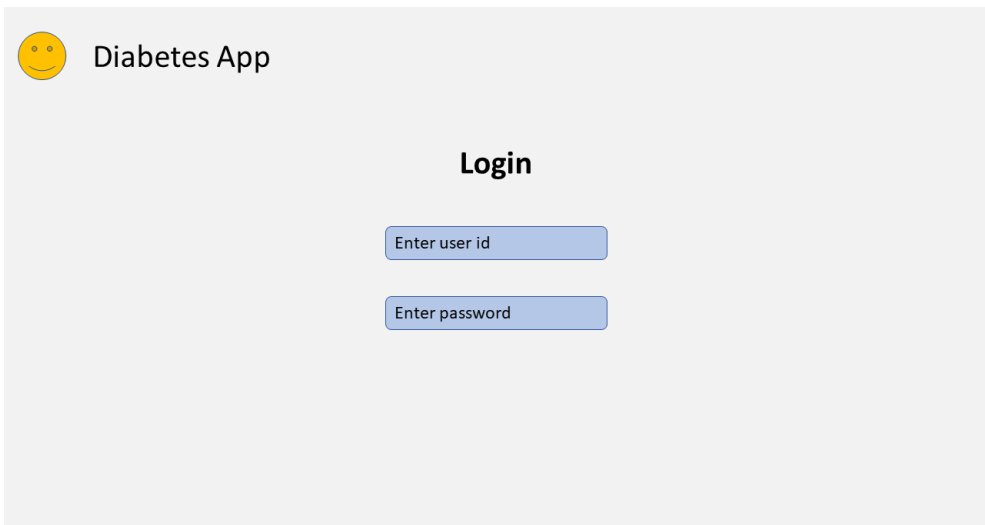


Figure C.6: Gui draft 1 Christopher Zwinge

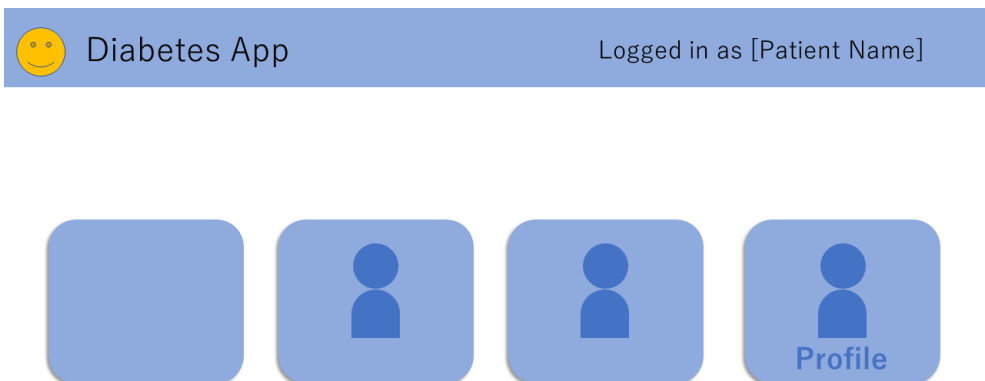


Figure C.7: Gui draft 2 Christopher Zwinge

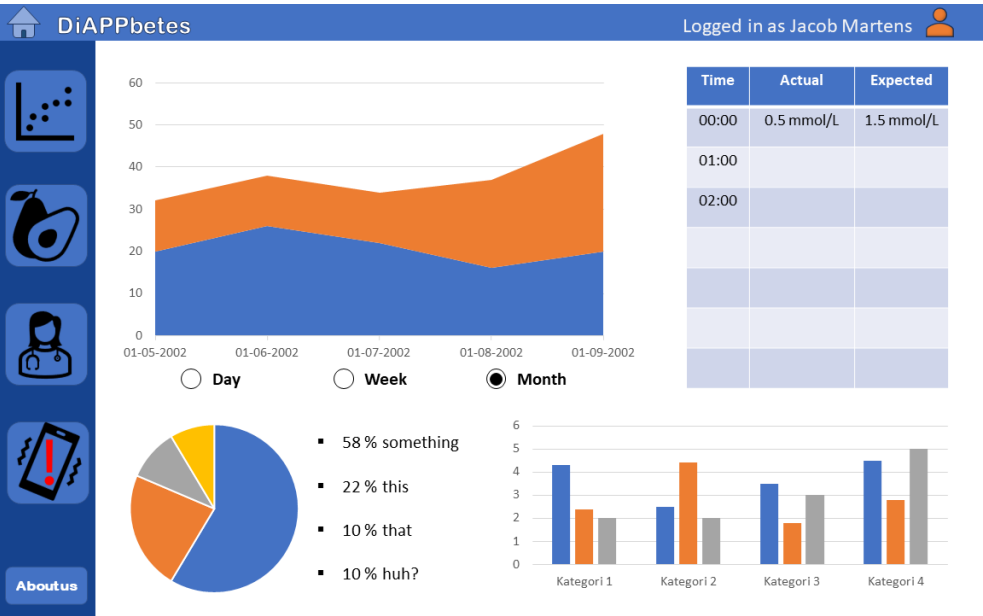


Figure C.8: App sketch 1 Jacob Martens

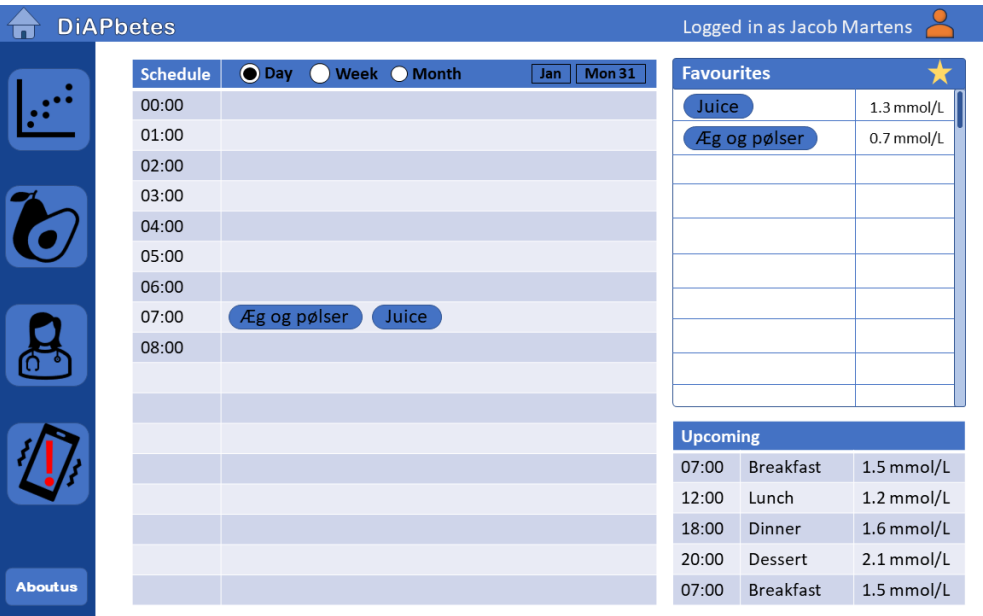


Figure C.9: App sketch 2 Jacob Martens



Figure C.10: App sketch 3 Jacob Martens

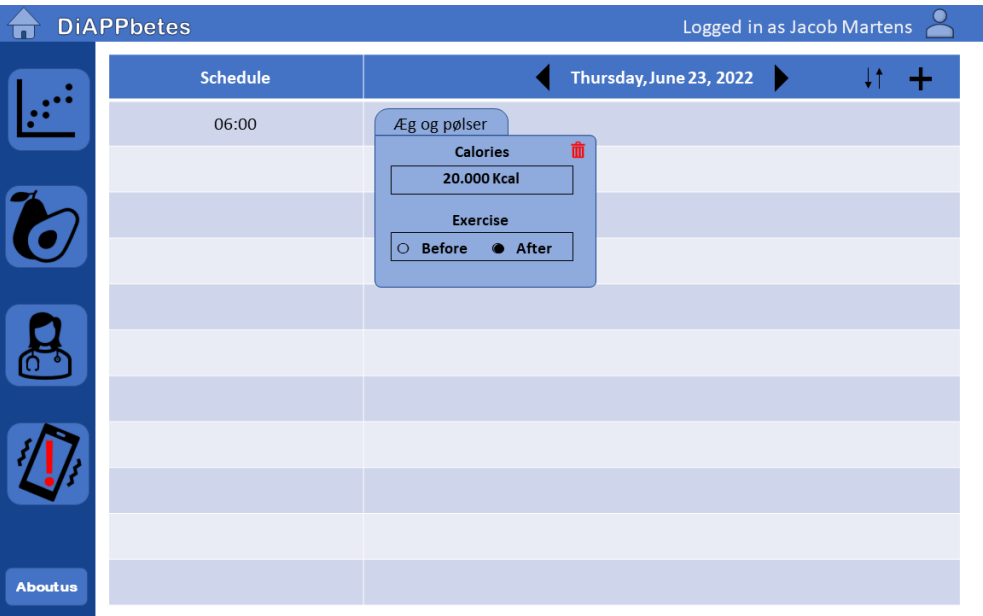


Figure C.11: App sketch 4 Jacob Martens



Figure C.12: App sketch 5 Jacob Martens

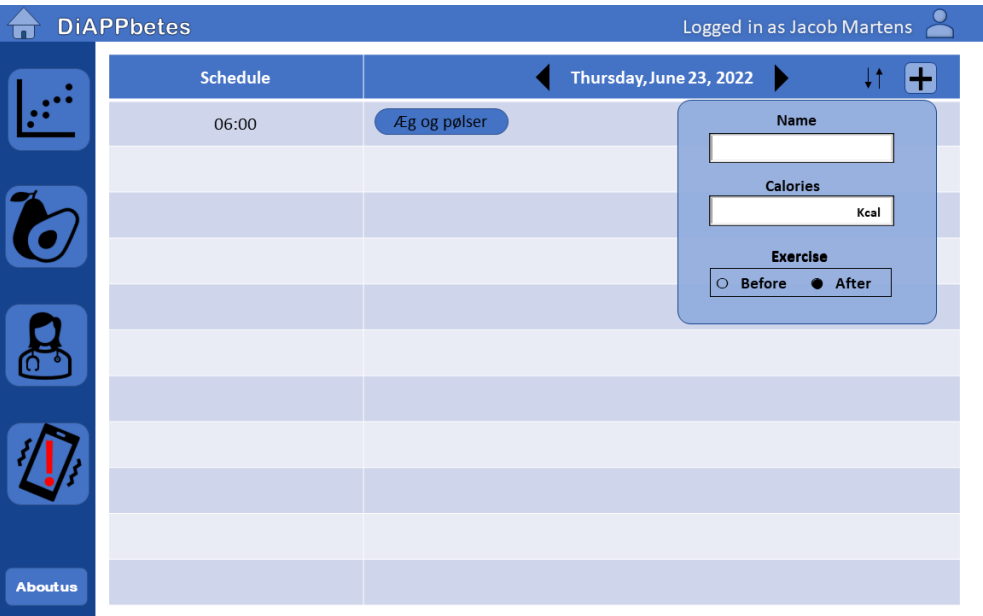


Figure C.13: App sketch 6 Jacob Martens

APPENDIX **D**

Use Case Diagram

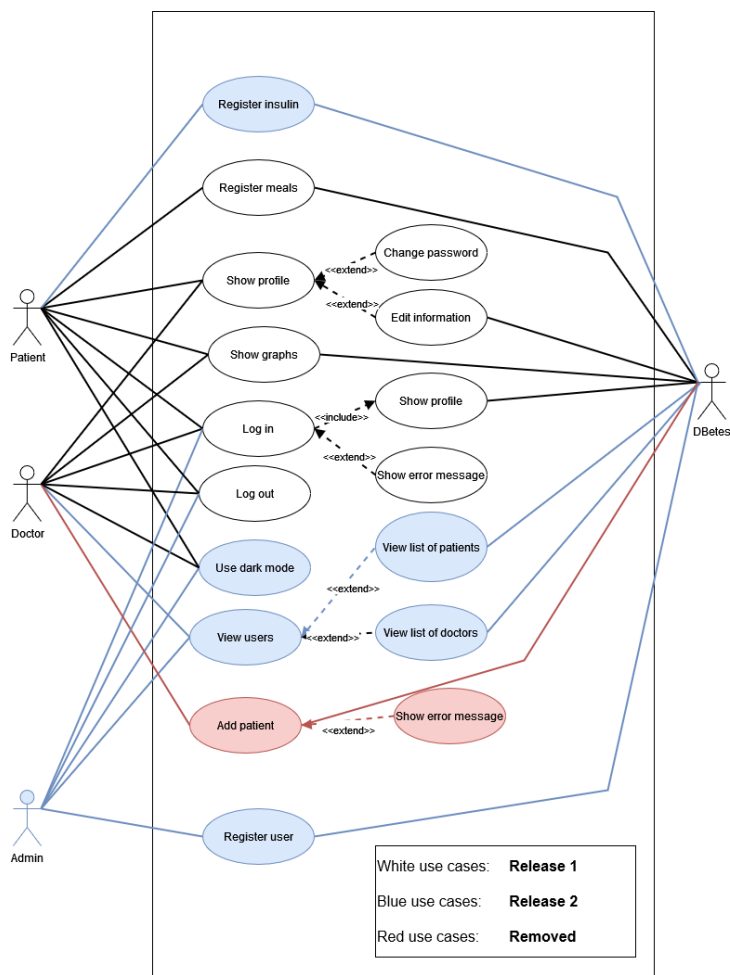


Figure D.1: Use Cases

Class Diagram

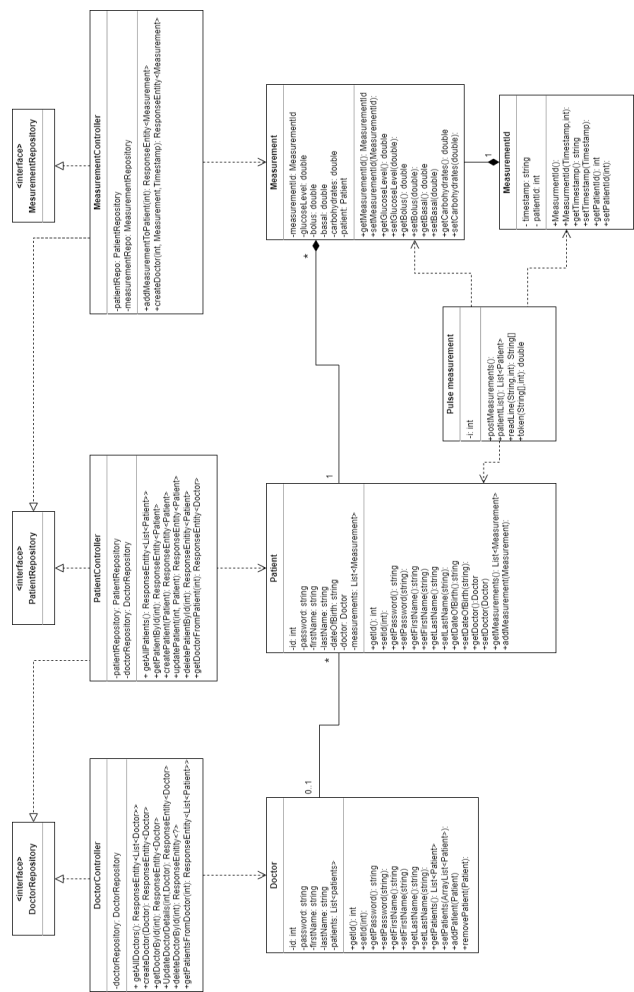


Figure E.1: Class diagram

