MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

Back-end solution in the field of mental health care

Bachelor's Thesis

MARTIN OLIVER PITOŇÁK

Brno, Fall 2024

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Back-end solution in the field of mental health care

Bachelor's Thesis

MARTIN OLIVER PITOŇÁK

Advisor: RNDr. Jaroslav Ráček, Ph.D.

Department of Computer Systems and Communications

Brno, Fall 2024


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source. During the preparation of this thesis, I used the following AI tools:

- Grammarly for grammar checking (https://www.grammarly.com/),
- Github Copilot for code suggestions (https://github.com/features/copilot),
- ChatGPT to formulate the thesis structure (https://chatgpt.com/)

I declare that I used these tools in accordance with the principles of academic integrity. I checked the content and took full responsibility for it.

Martin Oliver Pitoňák

Advisor: RNDr. Jaroslav Ráček, Ph.D.

Acknowledgements

I would like to offer thanks to my advisor RNDr. Jaroslav Ráček for providing advice and guidance instrumantal to this theses. In addition to that, I would like to thank the testers of my application for granting their time to this cause.

Abstract

This thesis aims to implement a back-end REST API that is able to be utilized by both a React web dashboard, which is also developed by the thesis, and a mobile application tasked with providing mental health care to its users. The primary part of this thesis discusses the various approaches one can use to implement such back-end system alongside requirements the API, and web dashboard should abide by. The implementation not only focuses on the system's functionality, but also on its utility, ease-of-use and user satisfaction.

Keywords

back-end, Express, mental health, TypeScript, API, BaaS, REST

Contents

List of Figures

Introduction

As mental health challenges continue to affect a growing number of people, it is vital to ensure accessible and effective solutions. Mobile applications, with their widespread adoption and convenience, provide an excellent way for reaching individuals in need. These applications rely on robust back-end systems to deliver their functionalities seamlessly. This thesis explores the design, analysis, and implementation of such a back-end solution together with a companion web application for doctors and administrators.

The proposed system offers a comprehensive API that integrates with a mobile application and a companion web dashboard, ensuring both platforms have access to necessary data. The thesis begins by analyzing existing mental health applications, identifying successful features, and their common pitfalls. It then evaluates various approaches to back-end API implementation, detailing the choices that go into selecting the most suitable solution.

Later sections outline the requirements for the back-end system from the perspectives of both client-facing applications, emphasizing functionality, security, and user satisfaction. Key aspects of the design include a responsive and efficient dashboard user interface, as well as endpoints providing intuitive mobile application experience with a focus on essential features. The implementation highlights functionality unique to this system such as report generation and push notifications..

1 Analysis of Current Solutions

This chapter first goes through two of the most downloaded mental health applications, assesses their feature set, and looks at the users' reviews of these applications for possible improvements. Secondly, it will evaluate possible types of backend service for mobile applications. Lastly, it compares the application programming interface design of SOAP, REST, and GraphQL APIs.

1.1 Analysis of mobile applications focused on Mental Health

In order to understand the user use cases, possible functionalities that doctors should be able to use, and to figure out the possible data API endpoints, this thesis is going to analyze two applications available on the Google Play store: Daylio Journal and Voidpet Garden. Alongside the analysis of the applications, the reviews of their users are also going to be inspected to figure out any potential pitfalls the back-end API should avoid, or improvements it should incorporate.

1.1.1 Daylio Journal

Daylio Journal, as indicated by its name and description on the Play Store, is primarily a journaling app designed to replace traditional pen-and-paper journals with added benefits [1]. One of its most prominent features is mood tracking. By encouraging daily entries, the app collects data on your emotions—enabling it to create a personalized profile that helps you identify what brings you happiness. Additionally, it employs push notifications to remind you to stay consistent and to notify you about important dates you have previously entered.

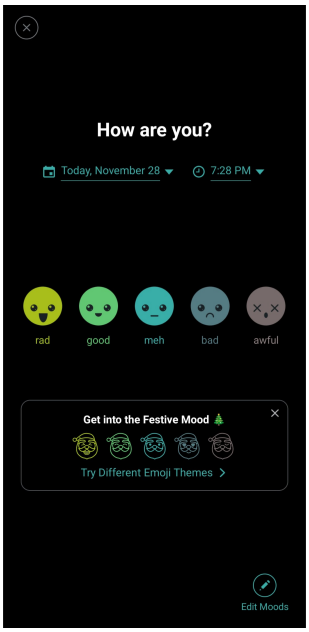

Figure 1.1: Mood recording feature in the Daylio application

1.1.2 Voidpet Garden: Mental Health

Voidpet takes a different route to help its users by gamifying the journaling experience [2]. Rather than recording users' moods and experiences in a journal like Daylio, Voidpet provides its users with a personalized pet to care for, similar to a real-life support animal. The animal's well-being should keep them returning to the application to continue to take care of it; this is achieved by completing various tasks the application gives them, such as answering various questions about their day or simply chatting with the pet. Finally, in the case of that failing, the reminders in the form of push notifications will try to lead the users back to the app.

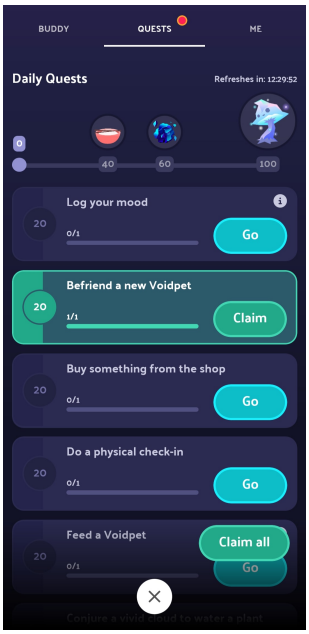

Figure 1.2: Voidpet tasks to help you take care of you pet

1.1.3 Users' reviews of the discussed applications

Google's review system in the Play Store can help us determine which ideas and functionalities users enjoy about the aforementioned apps, and warn us about their shortcomings or missing features. Daylio [1] users' positive reports generally praise the mood-tracking feature and often compare it to a streak that they should be keeping up. In terms of negative reviews, apart from app stability issues and payment structure, both of which are not concerning this thesis, the main shortcomings are a lack of customization in what stats are presented and the often non-functioning reminder system.

Voidpet [2] users applaud the simplicity of the app, numerous check-ins and reminders, and the refreshing usage of a pet system instead of only a simple journal. The main concerns of the community again are with the monetization system and the UX [1], which are, of no concern to us. Apart from those, the hidden difficulty of various

1. User experience

currencies and systems that need to be understood often drives users away from the application.

1.1.4 Takeaways

Users of these apps need a simple and intuitive experience to avoid being intimidated and driven away. We can achieve this by not overwhelming the user with many different systems and just providing the core concepts, such as journaling and mood tracking. Users of the Voidpet [2] application praised the innovative pet system. Instead of multiple different ways to keep your pet healthy and well, to simplify the experience, our users would only be required to journal and answer questions regarding their day to keep the pet happy.

Lastly, reminders and notifications appear to have a sizable impact on user happiness and retention. Doctors should have the option to directly interact with the patients by being able to remind them of their unfinished tasks with notifications. These changes would drastically simplify the user experience while still maintaining the core functionality of caring for their pet and not overwhelming them in the process.

1.2 Analysis of backend solutions

In this section, the thesis will compare different backend solutions, anging from a backend as a service to a fully custom implementation. After that, it will lay out the varying characteristics of API design that were considered when developing the API.

1.2.1 Backend-as-a-service

Backend-as-a-Service (BaaS) is a cloud service that lets you access predefined and predeveloped components to use for common app functionalities [3], such as:

- Authentication
- Database management
- Push Notifications

- Hosting
- Other platorm-specific features

As most of the back-end implementation is done by the Baa provider, the client can solely focus on the frontend side of their application, whether it is a web or a mobile app. On the one hand, the positives of taking this approach are the speed at which a solution can be developed, and the reduced developer overhead. On the other hand, the negatives, such as vendor locking, possible vendor outages, and often required payments, make this a challenging choice [4, 5].

Most often, developers utilizing BaaS will be required to pay for the service. The amount typically scales with the number of monthly active users the application is able to attract. The amount paid ranges from a few dollars a month to hundreds or even thousands in monthly payments to your BaaS provider [6]. The most used BaaS providers are [7]:

- AWS
- Firebase
- Supabase
- Netlify
- PocketBase

1.2.2 Custom backend

This approach is significantly more demanding on the application developers as all features required for the application to function, must be developed by the developers themselves. The main benefits of this approach is the flexibility in choosing the technology best suited for a project, and complete ownership of the whole technical stack and data. However, this solution does not completely eliminate the need to pay a provider for services, whether database hosting or a possible separate authentication service. Specialized talent that would not be necessary if a solution was being developed in conjunction with a BaaS provider [4] is an additional cost that needs to be considered.

1.2.3 Comparison of API designs

Frontend applications communicate with backend services through application programming interface (API) calls . This section will go through three primary standards [8]:

- REST
- SOAP
- GraphQL

SOAP, the oldest standard, was developed in 1998 based on extensible markup language (XML) that supports both HTTP and SMTP. Nowadays, this standard is used in the medical, banking, and enterprise sectors, mainly for its robustness and built-in error handling. When starting any new project, where one does not need to interface with outside SOAP services, it has been largely replaced by REST APIs, which do not use a large message size and do not include the XML overhead [9].

REST, meaning representational state transfer, is a more lightweight solution and is easier to use than SOAP. Utilizing HTTP methods like POST, CREATE, PUT, and DELETE, which neatly correspond with CRUD (Create, Read, Update, Delete) resource manipulation operations, it provides a loose binding between the two sides of the application, allowing for easy modification of the underlying functionality without changing the interface. In addition, it is inherently faster due to it switching XML for a more efficient JavaScript Object Notation (JSON) and its inherent statelessness To date, it is the most popular API standard [9, 10].

Lastly, GraphQL, developed by Facebook in 2012, allows for more efficient data gathering by being able to retrieve multiple resources with a single request, which alleviates both the problem of over-fetching (getting more data than required, thus requiring more bandwidth) and under-fetching (not getting enough data, thus having to make multiple requests to the backend service). However, it lacks proper implementation of error handling requiring its users to use often different third-party libraries, and due to its entirely different format, it might be challenging to grasp for those unfamiliar with it [9, 11].

1.2.4 Takeaways

Because we need custom business logic for processing data-related to tasks and moods, which patients will be able to interact with and record, we are required to implement custom backend logic, however we are able to make use of any backend-as-a-service provider for a simple database management and hosting, as well as our authentication, which is especially tricky to implement. Misconfiguring our authenticcation can have dire consequences on the whole system. For example, malicious actors being able to access patient data. Finally, REST API is a good choice for the communication between the backend and the client systems, as SOAP is mostly obsolete, and REST has more popularity and familiarity in the developer community compared to GraphQL.

2 System Requirements

This chapter goes over the required functionality of the system from the administrator, doctor, and patient side in the form of the user's use case, along with other non-functional requirements expected of the system. Additionally, it discusses the necessary endpoints for communication with the patient's mobile applications.

2.1 Admin and Doctor use cases

To build a helpful backend service and management dashboard, we first need to figure out what actions the involved parties are allowed to take. In this section, we go over the primary components for each use case, which are its primary actors, preconditions, triggers, and flows [12].

2.1.1 Sign In

It should be possible for the users to sign in into the dashboard to manage the user data.

- **Primary actors:** Administrator user, Doctor user
- **Precondition:** The user is currently not signed in
- **Postcondition:** The user should be logged in and on his home page
- **Trigger:** User clicks the sign in button
- **Basic flow:** The user will first input the required information. In our case, it will be his e-mail address and a password. Once completed, the user will click the sign-in button and be redirected to the admin web dashboard.
- **Alternative flow 1:** If the user fills in incorrect information, he shall be appropriately informed so that it is possible to fix the mistake.

- **Alternative flow 2:** If the sign-in process fails for any reason, even when the user fills in the correct information, that information should be presented with a request to try again or try again later.

2.1.2 Sign Out

Once the user is signed in, it should also be easy for the user to sign out from the system.

- **Primary actors:** Administrator user, Doctor user
- **Precondition:** The user is signed in to the system
- **Postcondition:** The user is signed out and on the sign-in screen
- **Trigger:** The user clicks the sign out button
- **Basic flow:** User will click the sign out button after which, he should be redirected back to the sign in page.

2.1.3 View statistics

Administrators shall be able to see various statistics of the whole system, such as the number of users in the system, whether doctors or patients, along with statistics about the patient's age, gender, and nationality distributions.

- **Primary actors:** Administrator user
- **Precondition:** The user is signed in
- **Postcondition:** It is possible for the user to see different statistics
- **Trigger:** User clicks the appropriate navigation button
- **Basic flow:** User clicks on a button that redirects him to a page displaying various statistics about the system.
- **Alternate flow:** The user clicks on a button that redirects him; however if a data fetching error occurs, the user should be informed about the issue.

2.1.4 Invite doctors

It would not be in the system's best interest for anyone to be able to create their account as a doctor; thus, this responsibility should be handled by the administrators, who will be able to invite doctors into the system.

- **Primary actors:** Administrator user
- **Precondition:** The user is signed in
- **Postcondition:** A new doctor has been invited into the system
- **Trigger:** The user clicks the button that sends the invitation
- **Basic flow:** After being presented with a form to invite the doctor, the administrator fills out the doctor's e-mail address and clicks the invitation button. Afterward, the user receives a message that the invitation has been sent.
- **Alternate flow:** After being presented with a form to invite the doctor, the administrator fills out the doctor's e-mail address and clicks the invitation button. When an error occurs while sending this invitation, the user shall be informed.

2.1.5 Accept invitation

When and administrator invites a new doctor into the system with the use case mentioned in 2.1.4, the invited doctor should be able to successfuly accept an invite and become a valid user.

- **Primary actors:** Doctor user
- **Precondition:** The user does not have an account in the system, and an invitation has been sent.
- **Postcondition:** The user does have an account in the system
- **Trigger:** New potential user clicks an invite link sent to his e-mail

- **Basic flow:** The invited user clicks an invite link delivered to him and gets redirected to a signup page where he fills in the required information. Consequently, he clicks the sign-in button and is again redirected to the doctor's home page.
- **Alternate flow 1:** The invited user clicks an invite link delivered to him and get redirected to a page informing him that the invite link has expired.
- **Alternate flow 2:** The invited user clicks an invite link delivered to him and gets redirected to a signup page where he fills in the incorrect information, and the signup fails. The user should be informed of the error and provided with ample information.

2.1.6 Display patients

Doctors should be able to view and search the patients assigned to them together with additional information regarding the patients, such as statistics about their moods in the last month.

- **Primary actors:** Doctor user
- **Precondition:** The user is signed in
- **Postcondition:** The user can see their patients
- **Trigger:** The user clicks the appropriate navigation button
- **Basic flow:** After the doctor clicks the navigation button responsible for navigating to the patient page, he gets redirected and after a short while, he can see the patients assigned to him.
- **Alternate flow:** After the doctor clicks the navigation button responsible for navigating to the patient page, he gets presented with an error message about data fetching failing.

2.1.7 View patient

When seeing the patients by utilizing use case 2.1.6, the doctor should be able to display further information regarding any patient assigned to him and visible in the 2.1.6 use case.

- **Primary actors:** Doctor user
- **Precondition:** The user is logged in, and the patient is present in the system while assigned to the doctor user performing this use case
- **Postcondition:** The user can see and managed the displayed patient
- **Trigger:** The user clicks on the patient he wants to manage
- **Basic flow:** User clicks on the desired patient when utilizing 2.1.6 and get redirected to a page dedicated wholly to the patient. There, the user can see information regarding the patient.
- **Alternate flow:** After clicking the desired patient, the data fails to load and the doctor get informed of the error in a non disrupting way.

2.1.8 Send reminder

When viewing detailed information about the patient by using 2.1.7, the doctor should have the option to send the patient a reminder that send a push notification to the patient's smartphone.

- **Primary actors:** Doctor user
- **Precondition:** The user is logged in, and the patient is present in the system while assigned to the doctor user performing this use case
- **Postcondition:** The chosen patient receives a reminder in the form of a push notification
- **Trigger:** The user clicks a button to send a reminder
- **Basic flow:** When on a page responsible for 2.1.7, the doctor clicks a button to send a reminder to the patient and gets informed of the action's success.
- **Alternate flow:** The user clicks the button responsible for sending the reminder; however, in this instance, the user is informed about an issue happening and the action not being successful.

2.1.9 Task management

Administrator users are able to manage tasks, which can be used by the doctors to either send them to the patient as they are or to be further edited by the doctors according to their specific needs. Administrators are able to create new tasks and, on top of that, browse the tasks already present in the system and edit or delete them at will. Additionally, doctors themselves can edit the predefined tasks from the administrator, and can edit or delete only their own tasks, so there is no interference in so-called "global" tasks created by administrators.

- **Primary actors:** Administrator user, Doctor user
- **Precondition:** The user is signed in
- **Postconditions:**
  - The administrator is able to view the tasks
  - A new task has been created
  - A task has been edited
  - A task has been deleted
- **Trigger:** User clicks a button to navigate to task management screen.
- **Task view flow:** User navigates to task management screen where he can browse the tasks in the system.
- **Task create flow:** The ser navigates to the task management screen. There, he is presented with a form that allows him to fill in task details. After doing so, the user clicks the button to create a new task and is informed of the success.
- **Task edit flow:** User navigates to task management screen and clicks the task that needs to be edited. By doing that, he is able to edit task data in a form similar to task creation. Lastly, the user clicks the submit button and after a moment is informed of the success.

- **Task delete flow:** The user navigates to the task management screen, and after selecting the task that is to be deleted, he clicks the delete button. After a short while, he will be informed of the result.
- **Task view alternate flow:** User navigates to task management screen where the tasks are unable to be loaded and is informed of the error.
- **Task create alternate flow:** After filling in the required information about the task, the user presses the appropriate button and after task creation fails, the user is informed of the issue.
- **Task edit alternate flow:** The user navigates to the task management screen and clicks the task that needs to be edited. After filling in the data and failing the edit action, the user is presented with the error statement.
- **Task delete flow:** The user navigates to the task management screen, and after selecting the task that is to be deleted, he clicks the delete button. Following that, the task deletion fails, and the user can see the error status.

2.1.10 Assign task

The doctor should be able to assign any task available to him to his patients.

- **Primary actors:** Doctor user
- **Precondition:** The user is logged in, and the patient is present in the system while assigned to the doctor user performing this use case
- **Postcondition:** The mentioned patient has been assigned a new task to complete
- **Trigger:** The user clicks the button to assign the task
- **Basic flow 1:** When on a page responsible for handling 2.1.7, the user clicks the button to assign a task to the specified patient and

gets shown a form requiring additional parameters regarding the assigned task. After filling in the information, the user clicks the button to submit the form, and thus, the task gets assigned to the patient. Finally, the user gets informed of the success. Finally, the user gets informed of the success.

- **Basic flow 2:** When on a page responsible for handling 2.1.9, the user fills in the required information, as well as the patient, to which the task is supposed to be assigned and does so by clicking the submit button. The user also gets informed about the success.
- **Alternate flow:** After the user clicks the button to assign the task to the patient, he gets informed of the assignment failing.

2.1.11 Manage patient's tasks

For the doctors in the system, it should be possible to view any task that was completed, is in progress, or has been assigned to a specific patient. When viewing the completed tasks, the doctors can further display the answers that the patient provided. Lastly, doctors can unassign any task not yet marked as being in progress or completed.

- **Primary actors:** Doctor user
- **Precondition:** The user is logged in, and the patient is present in the system while assigned to the doctor user performing this use case
- **Postcondition:**
  - The doctor can see the patient's assigned tasks
  - The doctor can see the patient's tasks which are in progress
  - The doctor can see tasks completed by the patient
  - The doctor can see answers to a completed task
  - A task has been unassigned
- **Trigger:** The doctor navigates to a page displaying information about a patient

- **View assigned tasks flow:** User navigates to the required page and presses a button to display tasks that have been assigned to a patient. After a short while, the user can browse these tasks
- **View in-progress tasks flow:** Similar to "View assigned tasks flow"; however, in this instance, the user selects to see in-progress tasks.
- **View completed tasks flow:** Analogous to previous view tasks flows, but now the user selects to see the tasks completed by the patient.
- **View task answers flow:** When viewing the tasks completed by the patient, the user clicks a display task to show the answers provided by the patient.
- **Unassign task flow:** When a user is viewing the tasks that are assigned to a certain patient, he clicks a button to unassign a specific task. He is informed of the success.
- **View tasks alternate flow:** When a user clicks to show a certain category of tasks, he is presented with information about an error occuring.
- **Unassign task alternate flow:** After clicking the button responsible for task unassignment, the user is shown information about the action's failure.

2.1.12 Report management

Administrators are able to generate anonymized data reports of the doctors or patients in the system. The newly generated reports as well as all those before it are available to be downloaded in a tabular CSV format.

- **Primary actors:** Administrator user
- **Precondition:** The user is signed in
- **Postcondition:**

  - Administrator is able to view previously generated reports
  - A new report has been generated

- **Triggers:** The user clicks a button to navigate to a report management screen.
- **Report view flow:** After navigating to the report management page, the administrator is able to view, filter, and download his previously generated reports.
- **Report generate flow:** After navigating to the report management page, the user is presented with an option to generate doctor or patient report. After choosing the desired filters, the administrator clicks the generate button, and after report generation is complete, is presented with an option to download the CSV file.
- **Report view alternate flow:** After navigating to the report management page, the administrator is informed of an error occurring while gathering required data.
- **Report generate flow:** After navigating to the report management page, and clicking the generated report button, the user is presented with an appropriate message about the generation failing.

2.2 Patient use cases

As it is not the responsibility of this thesis to develop an application for the patients, it cannot directly control how the patients are going interact with the application, the only way the back-end system can affect communication with the mobile application is through the API endpoints. This section analyzes the endpoints necessary to execute patient use cases.

2.2.1 Authentication

Due to patients being able to freely download the mobile application at will, an endpoint to handle the creation of a new patient user needs to

exist. It is not simply enough for users to be able to sign in, there must also be an option to log in and out of the application, therefore at least two additional endpoints are needed for handling that functionality. Lastly, it would be beneficial to provide an endpoint for retrieving information about the authenticated user for the application to use, together with an endpoint that handles patient deletion.

2.2.2 Pet management

The patients should be able to change the details of their pets, such as its name and color. For this reason, there should be an endpoint in place to handle requests for such changes. The pet creation process can be handled by the user creation endpoint mentioned in 2.2.1 because the only time pet creation endpoint would be required is after creating a new user. This change would eliminate one required API call, thus speeding up the potential application. The same thing can also be said about the pet deletion endpoint.

2.2.3 Task management

The main component of the application should handle tasks provided to the patient by their doctor. We can achieve this by creating at least two endpoints. The first one would handle requests for new tasks assigned by the doctor, while the other handles the delivery of the answers to specific tasks submitted by the patients.

2.2.4 Mood recording management

Another significant component of the application would be the recording of the patient's daily mood. The endpoints in this case are pretty simple, as we would simply need to provide basic Create, Read, Update, Delete (CRUD) functionality to the application. Additionally, a PUT endpoint for editing a day's mood record is quite beneficial if the patient decides to change his recording.

2.2.5 Diary management

The last significant component of the mobile application should be a simple diary where the patient can record their opinions and rec-

ollections of the day. Similarly to 2.2.4, a simple CRUD functionality reflected in the available endpoints should be enough for the application to accomplish the goal of recording diary entries.

2.3 Non-functional requirements

Non-functional requirements are there to describe the qualities of the system, in contrast to functional requirements, which depict how it should function and interact with the user [13]. This section highlights such requirements.

2.3.1 Responsive design

As any other modern application intended to be accessed through the web browser, the web dashboard should also respect contemporary responsive design that deals with different display sizes. Because the doctor and administrator dashboard is designed to be used solely on their respective work computers, it is not necessary for the dashboard to scale well for mobile devices. Nevertheless, it should still respond nicely to size changes of the browser window and not break in the process.

2.3.2 Performance

The web application should respond quickly to user input and not freeze for any amount of time. Furthermore, updates to any data should be reflected relatively quickly on the dashboard. Lastly, the responses of our API service should have minimal latency under the expected amount of load.

2.3.3 Usability

The dashboard should be simple to use and comprehend. Each page should be responsible for a two, at most three, pieces of functionality in order not to confuse the user. Besides that, the navigation between the pages should be intuitive and user-friendly. The application's design should follow a consistent style across all the pages.

3 Design Choices

The design choices chapter goes over the design of the data models in the Postgres database, together with technology choices used to achieve the final solution and the reasons for them.

3.1 Data models

3.1.1 User management

The `User` table represents all of the unique users of the application, whether it is the doctors or administrators interacting with the dashboard or patients with the potential mobile application. The records in this table are automatically inserted or deleted by database functions that respond to changes in the `users` table in Supabase's own protected `auth` schema, which is critical to the Supabase project's functionality and should thus be not interacted with. The `UserRole` model has a tight relation with the `User` model, as it directly defines a role for each user

The `Patient` and `Doctor` tables reference the user table and store data particular to that role. `Admin` table is not necessary as the application does not store any data unique to administrators. Therefore a record in the `User` table is sufficient

3.1.2 Bussiness data models

The `Pet` table stores data related to a patient's pet, such as name or mood. Alongside that, `Mood` and `Diary` tables are relatively simple to the point of being self-explanatory. In addition to mood data and diary records, they store additional metadata related to record's creation or updates. Complexity arises in tables `Task`, and `AssignedTask` as this is the heart of the application. The `Task` table represents task definitions that doctors and administrators manage. It stores data like the task's type, possible answers if it is a multiple-choice, task and the definition itself. Closely related, `AssignedTask` keeps information related to tasks already assigned to patients with data such as the assigned task's status, duration, user's ratings, and answers.

3.1.3 Notifications

Lastly, the `Notification` table saves all of the notifications sent to patient devices. The tokens that identify each one of the patients' devices are stored in the `FCMToken` table. Finally, `Messages` is a store for possible encouraging messages that can be sent to patients. This table is read from at random. A closer look at the implementation details can be found at 4.4.

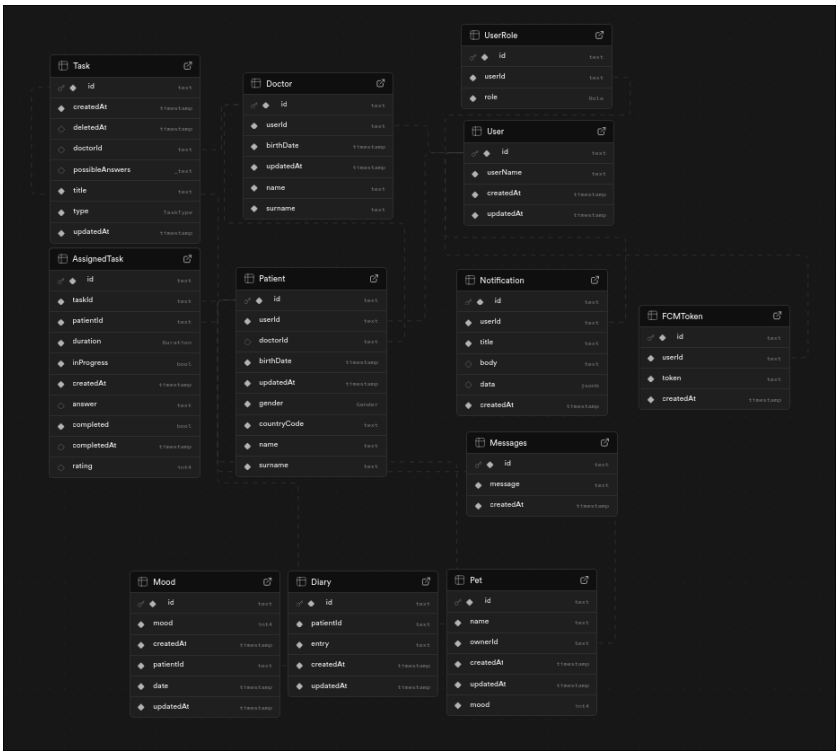

Figure 3.1: Visualization of the database schema

3.2 Technologies

This chapter will go through all the primary technologies this solution is built upon, and discuss the choices that went into choosing the employed technologies. Towards the end of the chapter, the thesis will also mention less impactful but still vital to the system's functioning.

3.2.1 BaaS provider

As mentioned in 1.2.4, this solution utilizes a backend-as-a-service provider for database hosting and authentication. The main choices for this solution consisted of the three most well-known platforms. These being, Firebase developed by Google, AWS Amplify, which is a part of Amazon's service AWS, and lastly, Supabase.

Firebase

Compared to the other two choices, Firebase is one provider we must use, even if we do not choose it as our primary BaaS provider. The reason for that is mobile push notifications functionality. Notifications are not just simply sent to the mobile client app; they must be sent through Google's Firebase Cloud Messaging (FCM). A more detailed description is in the implementation chapter and its subsection for notifications 4.4.

A significant positive to choosing Firebase is its generous no-cost pricing tier called Spark Plan [14], which provides 1GiB storage and 10GiB monthly network bandwidth. One thing it lacks in the free tier is cloud functions. All these positives would make it an easy choice, however Firebase is not providing us with an SQL database and instead opting for a custom-built document storage called Cloud Firestore [15]. As our data, consisting of relations between doctors, patients, and their tasks, are able to be effectively modeled in an SQL schema, this fact is a huge detractor from the possibility of using Google's Firebase platform.

AWS Amplify

This solution is more oriented towards projects that are greater on scope. This fact reflects in its free pricing tier, which is even better than the one Firebase offers [14, 16]. The Free Tier provides its users with an incredible 5GiB storage and 15GiB data transfer per month. Nevertheless, it suffers from the same issue as Firebase does 3.2.1, its NoSQL database [17]. There is an option to connect to a self-hosted SQL database, but that is exactly what this thesis aims to avoid with utilizing a BaaS provider; plus, it requires the use of GraphQL API [18].

Supabase

Supabase markets itself as an open-source alternative to Firebase that is easy to adapt. The sales pitch sounds great, so where is the catch? The catch is its weaker free tier. It scales with the application's monthly active users, so even if the application is relatively small but popular, the owner must still pay for the higher tier. On top of that, Supabase only provides 500MB of database space, 1GB of storage, and 5GB of network bandwidth [6]. These pricing options are somewhat weaker than the previously mentioned Firebase and AWS Amplify offerings. Despite all those downsides, it is still in contention, and that is because of its Postgres relational database [19]. Postgres is one of the most popular SQL database solutions; it scales well and offers various utility functions and extensions [20, 21].

With Supabase providing its users with an SQL database, excellent community support, authentication management, and even cloud functions, it is the best choice for this solution. While the free tier payment plan is a bit restrictive, it is more than enough for now. Added to that, if needed, the next step up on the pricing plan ladder is not aggressively expensive, starting at 25$ per month. With this choice, it is still required to set up a simple Firebase project to start using push notifications, as mentioned in 3.2.1.

3.2.2 ExpressJS

By its description, ExpressJS touts itself as fast, minimal, and unopinionated. This is precisely what this application needs [22]. Express is an excellent choice for this project due to its simplicity and extensibleness. As the back end of our system aims to have only a handful of endpoints, while almost all of them provide simple CRUD functionality, there is little need for a more robust framework. Built on extensible middleware modules [23], it will be easy to, for example, verify authenticated state by simply defining our own middleware logic. Lastly, the author's familiarity with the framework is also a positive feature.

3.2.3 React

React [24] is a library which helps you to build front-end web applications by creating so-called components that can be reused in different

parts of the application to simplify the markup and speed up the development of any solution. It provides an easy way to achieve performant app interactivity with its hooks [25] that provide you with options to store and use state, provide data to the whole application or handle transitions.

3.2.4 React Router

A type-safe routing library developed specially for React [26] will help us handle transitions and navigation between different routes in our application, whether the user is authenticated or not. Moreover, it also brings the ability to provide data between different routes to help us avoid unnecessary HTTP requests and simply pass the data we already have available.

3.2.5 Other technologies

Technologies mentioned here are less fundamental to the functioning of the application, but they still make the development easier with their benefits.

JWT

JSON Web Token (JWT) [27] is a unified way to transfer various claims and data between two parties securely. In our case, it will be used to verify valid user sessions to serve responses from our back-end API.

TurboRepo

Instead of maintaining two separate code repositories for this project, this thesis adopts TurborRepo [28] to manage, build, format, and lint code in a single repository.

Prisma

An object-relational mapping library [29] creates a type-safe abstraction layer between the database SQL queries and client code. It provides functions to use instead of writing raw SQL queries to retrieve data from a database.

Zod

A validation library is used mainly for validating user-submitted data [30]. Great to use for either validating submittable forms on the front end or incoming data in requests to the back end.

SwaggerUI

A tool to help write and host API endpoint documentation [31]. Instead of writing, for instance, a markdown document, which describes each endpoint in detail, one just needs to provide a configuration file documenting the endpoints, and SwaggerUI builds a website, where people can browse possible endpoints.

Tanstack Query

Tanstack Query [32] is the de facto standard way of fetching data for use in front-end applications. It provides automatic caching, refetching, request deduplication, and many more out-of-the-box nice to have features that help to decrease the amount of async fetching code you have to write.

TailwindCSS

An extensible cascading stylesheet(CSS) framework [33] which provides its users with a multitude of classes to use in their markup, therefore eliminating the need to write CSS files. It can be used with plain HTML as well as many popular frameworks such as Next.js, Vite, React or Angular [34].

Shadcn/ui

A collection of various reusable components [35] that allows you to create your own component library. Compared to real component libraries such as Chakra UI [36] or Material UI [37], he component files get copied to your code repository and allow you to change every detail about them.

4 REST API Implementation

This chapter goes through the details of the implementation of the REST API. Express APIs are constructed by attaching multiple Express Routers to handle endpoints on a specific route. Each of these routers may have different options and middleware [23] attached to them when they run.

4.1 Authentication

In order to verify that only valid users with an ongoing active session from either the web dashboard or mobile application are allowed to make requests to our API, the system needs to somehow authenticate them. This authentication is achieved by the client application passing a JWT (see 3.2.5) token in the Authorization header with every request. The token is required because REST is stateless, meaning the protocol does not retain any information about previous requests from the client. The client receives this JWT on every successful login through Supabase. As these tokens contain encoded information about the user, after validating, we can use this information while serving the request without the client having to provide additional data [38].

To verify this JWT, we can make use of the middleware modularity that Express [22, 23] offers us.

```
export const isAuthenticated = async (
  req: Request,
  res: Response,
  next: NextFunction
) => {
  const { authorization } = req.headers;
  if (!authorization) {
    return handleErrors(res, new
       UnauthorizedError());
  }

  const parts = authorization?.split(" ");
```

```
  if (parts[0] !== "Bearer" || parts.length !==
     2) {
    return handleErrors(res, new
       UnauthorizedError("Wrong token format"));
  }

  const { data, error } = await supabase.auth.
     getUser(parts[1]);

  if (error) {
    return handleErrors(res, new
       UnauthorizedError(error.message));
  }

  const jwt = jwtDecode(authorization!);
  res.locals.user = data.user;
  res.locals.user.appRole = jwt.user_role;

  return next();
};
```

Listing 4.1: Middleware to determine if the API received valid JWT

Each defined middleware function must take three arguments. The HTTP request and response objects that Express works with and a function conventionally called `next`. The `next` function represents the following middleware function to be called after this one is done executing [39]. This function first checks if the HTTP request contains an authorization header in the correct format If not, it returns a 401 Unauthorized error to the client. Next, it makes a call to Supabase to check if the JWT represents a valid user and its session. Again, if not, it returns a 401 error. If the check passes, it saves the decoded JWT data to a `res.locals` property so that the information can be accessed by a function handling the request.

Role-base Access Control

Additionally, in the aforementioned setup, it would make sense only to allow access to a particular endpoint based on the user's role; e.g.,

administrators can only access admin endpoints. We could simply add a role column to the `User` database schema and make a database request for the user's role on every API request. However, a much better way to do this is to embed this information into the user's JWT token that Supabase provides, and the API receives every request. The process of achieving this is described in Supabase's documentation under the Access and Security section [40]. We create a new database table that ties every user to its role by having two columns, one for the user's ID and the other one for the associated role. Then, we create an authentication hook in Supabase that runs just before a JWT token is issued. This hook is a Postgres database function written in their proprietary language called PL/pgSQL [41].

```
declare
    claims jsonb;
    user_role public."Role";
  begin
    -- Fetch the user role in the user_roles
       table
    select role into user_role from public."
       UserRole" where "userId" = (event->>'
       user_id');

    claims := event->'claims';

    if user_role is not null then
      -- Set the claim
      claims := jsonb_set(claims, '{user_role}',
          to_jsonb(user_role));
    else
      claims := jsonb_set(claims, '{user_role}',
          'null');
    end if;

    -- Update the 'claims' object in the
       original event
```

```
    event := jsonb_set(event, '{claims}', claims
      );

    -- Return the modified or original event
    return event;
  end;
```

Listing 4.2: Hook responsible for attaching role to user's JWT

This function first retrieves a role associated with the user who requested the JWT. After that, it retrieves claims. Claims are custom properties attached to users to give the application more information about them [42]. Then, the function adds the role information to the existing claims object or null if the user does not possess a role. Finally, we attach the new claims object to the event that triggered this function and return it. Doing this allows the back end to easily access the user's role from the JWT provided.

4.2 Schema definition

Because this application uses Prisma as the ORM of choice, it can just use Prisma Schema to define the database schema, perform migrations, and generate a client the application will use when accessing the database [29, 43, 44]. The schema is constructed by first providing options about which generator to use and information about the database.

```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider  = "postgresql"
  url       = env("DATABASE_URL")
  directUrl = env("DIRECT_URL")
}
```

Listing 4.3: Information in prisma schema, including database connection strings

Then, one can start declaring the database table by using the model keyword. The column definition consists of its name, type, and addition attributes, for example, if the column is the table's primary key, its potential default value, or its relation to another table.

```
model Messages {
  id        String   @id @default(uuid())
  message   String
  createdAt DateTime @default(now())
}
```

Listing 4.4: A Prisma Schema definition of Messages table with three columns

4.3 Endpoint definitions

One should follow some best practices when developing endpoints for REST API [45]. The most important of these are:

- **Use nouns in endpoint paths:** Paths inform API consumers about what resource is accessed or modified using that precise endpoint. The API should use correct and appropriate HTTP Methods to indicate what is being done with the resource. For instance, GET is used to retrieve a resource, POST is used to create a new record, and DELETE is utilized for deletion.
- **Use layered endpoint structure:** One should group related endpoints together in a thoughtful matter. For example, to access moods of a specific user, it is better to create a sub-path in the user endpoint for the moods, such as `/users/{id}/moods`, than to provide a completely new endpoint `/moods`.
- **Provide standard error and success codes:** The code returned from an endpoint should appropriately indicate what kind of result the endpoint is returning. That means returning proper error and success codes to the API consumer.

The fact that endpoints have a path can be used in the folder structure of this project. This way, every sub-path is a new folder with an Express Router, which contains either deeper Express Routers for

nested paths or endpoint definitions. The most intuitive way to split our endpoints would be by the type of user. That way, we have three endpoint paths `/admins`, `/patients`, and `/doctors`. Next, the `/tasks` endpoint would offer options to retrieve and modify task definitions, and `/notifications` would be used for notification management. It is further possible to split `/patients` endpoint into `/moods`, `/tasks`, and `/diary` for modifying each resource belonging to unique patients.

Figure 4.1: Paths of all API endpoints

The `{id}` path names signify variable paths that depend on the id of the requested resource. Additionally, because the API requires a JWT token in the `Authorization` header, as described in 4.1, it is possible to also expose `@me` endpoints to provide information about an authenticated user just from the information provided by the JWT token.

Controller pattern

In Express applications, the controller is the final middleware function, meaning this is the function where the API request handling resides. This piece of code handles calls to functions for request body validation, database manipulation, and response creation. It is best practice to group these controllers and access them a router. This massively simplifies the router file that is used to define specific endpoints.

```
taskRouter.get("/", taskController.getAll);
taskRouter.get("/:id", taskController.get);
taskRouter.post("/", taskController.create);
taskRouter.put("/:id", taskController.update);
taskRouter.delete("/:id", taskController.remove)
    ;
```

Listing 4.5: Definition of a router handling tasks endpoint

After defining the router, all that remains is to inform the Express app to use it on a distinct path, together with the authentication middleware function mentioned in 4.1.

```
app.use("/tasks", isAuthenticated, taskRouter);
```

Listing 4.6: Route being assigned a router

Request body validation

Per 3.2.5, this project uses Zod to validate data submitted in the body of the request. Validation is necessary because we cannot rely on the client to submit correct data or on the goodwill of bad actors. Additionally, by using this validation, we substantially simplify our controller logic due to the fact that we just need to call a single function to verify the incoming request instead of checking for every possibility.

```
export const updateMoodSchema = z.object({
  params: z
    .object({
      id: z.string().uuid(),
      moodId: z.string().uuid()
    })
    .strict(),
  body: z.object({
    mood: z.number().int().min(1).max(5)
  })
});
```

Listing 4.7: Zod object used for validating request for mood editing

In a single function call, we can verify that the request to update a patient's mood record contains a valid user's unique identifier, a legitimate identifier of a mood record, and a new value in a valid range.

```
const request = await parseRequest(
   updateMoodSchema, req, res);
  if (!request) return;

  try {
    const mood = await prisma.mood.update({
      where: {
        id: request.params.moodId
      },
      data: {
        mood: request.body.mood
      },
      select: {
        id: true,
        mood: true,
        date: true
      }
    });

    if (!mood) {
```

```
        return handleErrors(res, new InternalError
            ("Failed to update mood"));
    }

    return res.status(200).send({ value: mood })
        ;
```

Listing 4.8: Controller function handling the updates to patient's mood records

As can be seen in the code above, the first thing to happen is request validation in relation to a pre-defined schema. If it fails, the called function handles the API reply, and the controller just stops executing. After that, it is just a simple database update with error handling and a successful response if everything goes well.

4.4 Push notifications

Supabase documentation already includes an entry on how to set up push notifications [46]. To have a record of all notifications sent to mobile application users, there needs to be a database table for them. Instead of adding messages to this table after we send them, a better way of handling this is first creating records in the database, which then triggers a cloud function (a function which does not run on the system server, but is instead handled and distributed by Supabase themselves) [47] that sends the notification to the patient.

Because notifications are not sent directly to the user devices but instead handled through Google's Firebase Cloud Messaging [48] API, we need so-called FCM tokens that are used to identify user devices by FCM. The patients' mobile application provides these tokens, and our system must store them. To do that, Supabase recommends adding a column to a table representing users. This is not enough for this system because the patient could have multiple devices, and each device has a unique token. That is why the best way to handle this is to create a new table with a user's ID as a foreign key and a column for tokens.

Next, defining the cloud function that fires every time a record is inserted into the `Notifications` table is necessary.

```
const { data } = await supabase
  .from('FCMToken')
  .select('token')
  .eq('userId', payload.record.userId);

const accessToken = await getAccessToken({
  clientEmail: serviceAccount.client_email,
  privateKey: serviceAccount.private_key,
});

const fcmUrl = `https://fcm.googleapis.com/v1/
   projects/${serviceAccount.project_id}/
   messages:send`
const fcmPayloads = data.map((row: { token:
   string }) => (
  {
    payload: {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
        Authorization: `Bearer ${accessToken
           }`,
      },
      body: JSON.stringify({
        message: {
          token: row.token,
          notification: {
            title: payload.record.title,
            body: payload.record.body,
          },
          android: {
            priority: 'HIGH',
          },
          data: payload.record.data
        },
      }),
    },
```

```
      token: row.token,
    }
  ));
```

Listing 4.9: Edge function which handles push notifications

There is a need to modify the function provided by the Supabase documentation because this system stores multiple FCM tokens. Multiple messages must be constructed and sent to all of the patients' devices to send a notification. The need to set up a Firebase project, as mentioned in 3.2.1, is because we need the `serviceAccount` environment variables to get the access token to be able to use FCM. After sending these messages, the function waits for the replies from the FCM service and deletes any tokens in our database that FCM marks as `UNREGISTERED`. This can happen when a user re-installs the client application or deletes its data [49].

```
if (resp.ok === false
  && resp.error.code === 404
  && resp.error.details[0].errorCode === '
     UNREGISTERED'
) {
  await supabase
    .from('FCMToken')
    .delete()
    .eq('token', fcmPayload.token);
}
```

Listing 4.10: Handling of the UNREGISTERED tokens

Finally, the REST API exposes a POST `/notifications` endpoint, which selects a random message from the `Messages` table and sends a push notification to a patient based on a provided patient ID in the request body.

4.5 Reports

Alongside a Postgres database, Supabase also offers Storage [50], which is an object store meant for storing text, image and sound files. This way of storing the generated reports then allows administrators to

go over any previously generated reports and download them at will. Furthermore, it can be accessed through the Supabase's JavaScript SDK this application uses.

Report generation

To generate a report, `/admin/reports/patients` and `/admin/reports/doctors` POST endpoints are served by the back-end API, each to generate a report for its respective user group. These endpoints accept a few filtering parameters in the request body that are to be considered when retrieving data from the database for report generation, such as the birthday ranges of the users included or their gender. These filters are also added to the generated file's metadata.

After the data is retrieved, it is converted into a CSV string format by joining the values with commas into a row record, and the joining the records by including an end-of-line character between each one, in accordance with [51].

```
export const dataToCsv = (data: DoctorData[]):
   string => {
  const csvData = [COLUMNS];
  data.forEach((record) => {
    const row = [
      record.birthDate.toDateString(),
      record.user.createdAt.toDateString(),
      record._count.patients.toString()
    ];
    csvData.push(row);
  });

  return csvData.map((row) => row.join(",")).
     join("\n") + "\n";
};
```

Listing 4.11: Function converting database data into a CSV string

To upload the CSV file, a file path needs to be defined so that Supabase knows where to store it. Supabase's Storage operates with buckets. Each bucket has its own folders and files that store data [52]. After creating a `reports` bucket in the Supabase's web dash-

board, it is possible to access the functions to operate on it by calling `supabase.storage.from("reports")`. To organize the files, the `reports` bucket follows the following structure, where `{userId}` represents folders with IDs of users that own the reports within:

Figure 4.2: Reports bucket's folders structure

```
const BUCKET = supabase.storage.from("reports"
    );
const fileName = `report_${new Date().
    toISOString()}`;
const fullPath = `${type}/${userId}/${fileName
    }.csv`;

return BUCKET.upload(fullPath, fileContent, {
  metadata: metadata ? metadata : {},
  contentType: "text/csv"
});
```

Listing 4.12: Code handling report upload to a bucket

Each filename contains a timestamp to avoid potential name collisions. All that is required to upload a file is to pass the file's path and content to a bucket's upload function. Here, any filtering metadata is also passed together with a file mime type to ensure it is correctly handled when downloaded by the user [53, 54, 55].

Report downloads

When a user requests a report download, it would be a bad idea to download the file on the server and then pass the data back to the user, as it would drastically increase communication complexity and the data throughput needed to operate the service. Because the

reports are not hosted publicly to protect the data, there need to be two endpoints to generate this URL for each type of report and return it to the user. Supabase's SDK provides an option to achieve this by calling `createSignedUrl` function on a bucket where the file is stored [56]. After calling it, the function returns a URL to a file available for a certain amount of time. This time can be specified by providing the number of seconds required when calling a function; in this case, it is just sixty seconds. This kind of URL is also returned as a reply to a report generation request.

```
const url = await BUCKET.createSignedUrl(
   upload.data.path, 60);
if (url.error) {
  return handleErrors(res, url.error);
}
return res.status(200).send({ url: url.data.
   signedUrl });
```

Listing 4.13: Signed URL being return after generating a report

5 Dashboard Implementation

The chapter discusses various implementation details of the front-end dashboard made for both administrator and doctor users.

The dashboard is a single-page application (SPA), meaning that users have to load only a single HTML document, which is then updated with the help of JavaScript [57]. SPAs allow for more user interactivity with a trade-off in their difficulty implementing effective search engine optimization (SEO). This is a worthwhile trade-off for this application, as the targeted users are very specific and most likely instructed to use this precise application.

5.1 Authentication

Because this application utilizes Supabase Auth [58], it is possible leverage Supabase's JavaScript client SDK [1], which can handle session management automatically by storing it in the browser's local storage on sign-in and removing it on sign-out. This enhances the user experience noticeably due to not having to sign-in after each time the dashboard is reopened [59].

Authentication handling is defined at the top of the application, more specifically in the file called `App.tsx`

```
useEffect(() => {
  supabase.auth.getSession().then(({ data: {
      session } }) => {
    saveSession(session)
  })

  const {
    data: { subscription },
  } = supabase.auth.onAuthStateChange((
     _event, session) => {
    saveSession(session)
  })
```

1. Software development kit

```
    return () => subscription.unsubscribe()
}, [])
```

Listing 5.1: Authentication handling in a useEffect hook

The `useEffect` hook is a React hook [25] usually used for handling events made by external systems. It consists of a function definition that should run when the hook fires and an array of dependencies limiting when it can fire. Leaving the array empty means that the hook will run on application start. The hook function itself performs two actions. Retrieves a session object from the browser's local storage and sets up a listener that does the same on every authentication state change, such as sign-in, sign-out, or user updates. Lastly, we return an unsubscribe function, which stops the listener so as not to cause memory leaks.

Instead of storing the session similarly to Supabase's documentation [58], the application needs to do processing first. This is why the application defines a custom `saveSession` function.

```
function saveSession(
  session: Awaited<
    ReturnType<typeof supabase.auth.
       getSession>
  >["data"]["session"],
) {
  setSession(session);

  const currentUser = session?.user;
  if (session && currentUser) {
    const jwt = jwtDecode(session.
       access_token);
    currentUser.appRole = jwt.user_role;
  }

  setUser(currentUser ?? null);
  if (currentUser?.appRole === "DOCTOR") {
    axiosClient
      .get(`/doctors/@me`, {
        headers: {
```

```
            Authorization: `Bearer ${session!.
               access_token}`,
          },
        })
        .then(({ data }) => {
          setUserData(data.value);
        });
    }
  }
```

Listing 5.2: Function responsible for storing the user's session

Because the application makes use of Supabase's role-based access control [40], it also needs information about the user and not just the session. Therefore, after storing the session, this function then also stores the information about the user and requests additional data from the back end if the user has the role of a `DOCTOR` with the JWT 3.2.5 it receives from Supabase.

Finally, `App.tsx` provides the authentication data to the rest of the application by employing React's `useContext` hook [60].

```
  return (
    <UserContext.Provider
      value={{
        userData,
        user,
        session,
      }}
    >
      <RouterProvider router={router} />
    </UserContext.Provider>
  );
}
```

Listing 5.3: Context provider being passed the data about the user and the session

5.2 Routing

As mentioned in 3.2.4, the application uses React Router for its routing. Instead of serving multiple HTML documents (one for each path), React Router replaces the browser navigation and handles it independently. This means that our top application component (after the Context provider) is going to be a router provider (as seen in 5.3) with a browser router created by calling React Router's `createBrowserRouter` and pass in a list of objects containing the desired routes.

```
const router: Router = createBrowserRouter(
  routes);
```

Listing 5.4: Router creation

By default, the route object contains a TSX component to draw (TSX is a typed version of JSX [2]), its path, and possible children route objects. This design creates the layered path structure of the application. The array passed to the `createBrowserRouter` function contains the top-layer routes. Firstly, a `/` route for the whole application, secondly `/forbidden` route for error handling, and lastly a catch-all `*` wildcard route for when a user navigates to a non-existent route.

```
const routes: RouteObject[] = [
  {
    path: "/",
    children: rootRoutes,
  },
  {
    path: "/forbidden",
    element: <Forbidden />,
  },
  {
    path: "*",
    element: <NotFound />,
  },
];
```

Listing 5.5: Top level routes of the application

2. JavaScript syntax extension

Similarly, `rootRoutes` is yet another array containing the application's main routes. These again define the deeper routes of the application.

```
const rootRoutes: RouteObject[] = [
  {
    path: "doctor",
    element: (
      <DoctorRoute>
        <DoctorLayout />
      </DoctorRoute>
    ),
    children: doctorRoutes,
  },
  {
    path: "auth",
    children: authRoutes,
  },
  {
    path: "admin",
    element: (
      <AdminRoute>
        <AdminLayout />
      </AdminRoute>
    ),
    children: adminRoutes,
  },
];
```

Listing 5.6: Application's main routes

Protected routes

It is undesirable to let any user access any routes, as doctors should not be able to access administrator routes and vice-versa. The issue is solved by wrapping the top-level doctor and administrator routes in a `<DoctorRoute>` or `<AdminRoute>` element. These elements first check for an authenticated user by calling the `useContext` hook, as mentioned in 5.1. If it exists, those elements will return the children

passed into them; if not, the user gets redirected to the `/auth` page to perform authentication.

```
function DoctorRoute({ children }:
   DoctorRouteProps) {
  const { user } = useContext(UserContext);

  if (!user) {
    return <Navigate to="/auth" replace />;
  }

  if (user.appRole !== "DOCTOR") {
    return <Navigate to="/" replace />;
  }

  return children;
}
```

Listing 5.7: <DoctorRoute> element which protects the doctor routes

In addition to `<DoctorRoute>` and `<AdminRoute>` elements, there also exists a simple `<ProtectedRoute>` element that serves a similar function; the only difference is that it does not require a specific role to let the user through.

Handling redirects into the application

Handling redirects from outside the application is essential for doctors to be able to accept invites or for handling password resets. Route security cannot be accomplished by the flow described in the previous section, as users are redirected straight to a specific path, skipping the authentication page. Fortunately, Supabase adds a query parameter with the name of `token_hash` to the redirect links included in the sent emails [61]. After retrieving it, it is then possible to exchange this `token_hash` for a valid user session. To avoid unwanted errors, the token exchange must be performed before content is rendered. This is achieved by using React Router's route loader, which is a function that is meant to provide data to a route before it renders [62].

```
  {
```

```
    loader: async () => {
      const res = await supabase.auth.initialize
         ();
      if (res.error) {
        return redirect("/forbidden");
      }
      return null;
    },
    path: "reset-password",
    element: (
      <ProtectedRoute>
        <ResetPassword />
      </ProtectedRoute>
    ),
  },
```

Listing 5.8: Route definition for a reset password route

The token exchange itself is performed by calling `supabase.auth.initialize` function, which initializes a new session from the URL. Lastly, if anything goes wrong with the initialization, it is best to redirect the user to an error page.

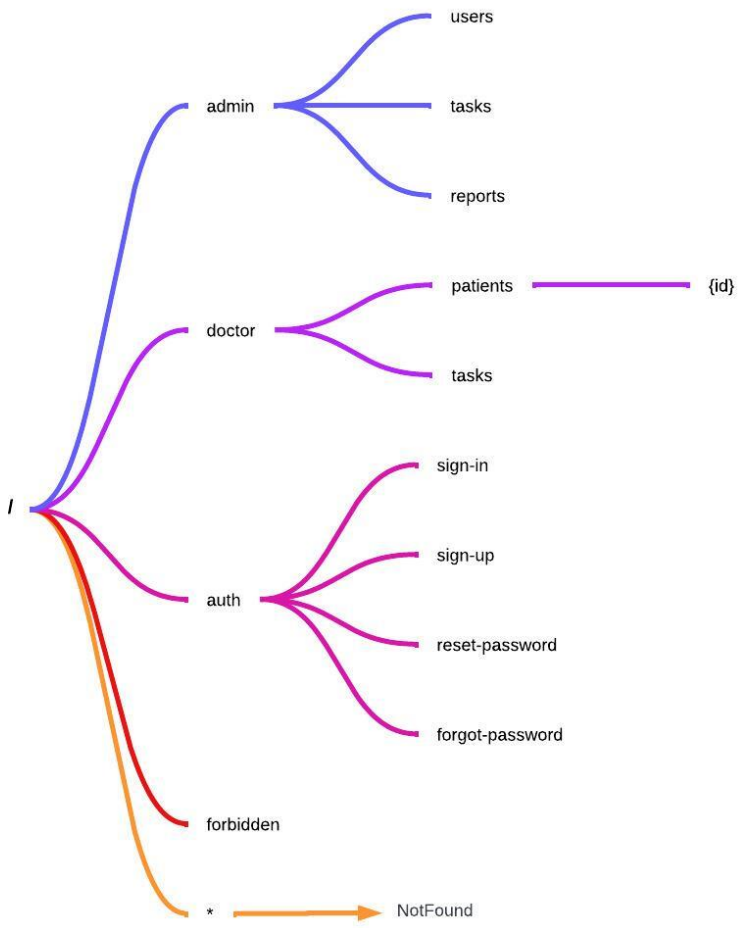

Figure 5.1: Dashboard routes

5.3 Pages

This section goes over and explains the design and functionality of the front-end dashboard pages. All of the pages that require the user to be authenticated (these are the `/admin` and `/doctor` pages) share a singular layout consisting of a navigation menu on the left, together

with a card-based main layout in order to improve user experience with less jarring transitions, as described in 2.3.3

Error pages

Error pages are made to be straightforward to understand. These screens are composed of a singular centered card with a potential short description of what went wrong in case of errors, with only a single button, which, when pressed, will redirect the user back to the home page. Depending on the authentication status the home page could either be the `/admin` and `/doctor` pages (depending on the type of user) or a sign-in page.

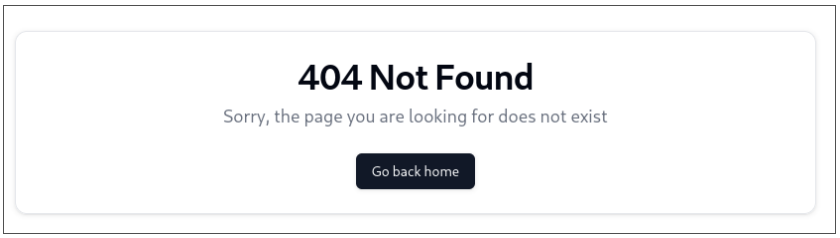

Figure 5.2: 404 error page

Auth pages

This application contains four auth pages in total. Similarly to error pages, they all (except the sign-up page) are again constructed with a singular centered card. The sign-in page and the page to request a password reset are free to be accessed by anyone, whereas the sign-up and password reset handling pages require a special redirect from outside the application to gain access.

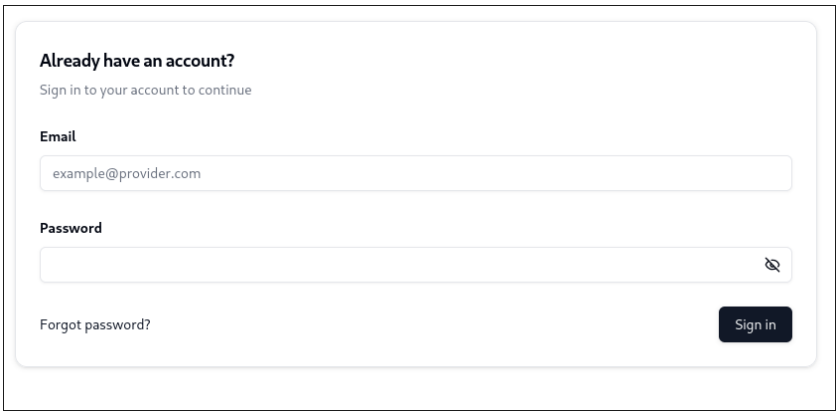

Figure 5.3: Dashboard login page

Administrator pages

The administrator dashboard consists of three separate pages. Firstly a page to "manage" users, which takes care of 2.1.3, 2.1.4 use cases by displaying information about all of the system's users and allowing the administrators to invite new doctors.

Secondly, there is a task management page consisting of various cards that allow the administrators to view, create, edit, and delete global tasks, thus handling 2.1.9.

And thirdly, the dashboard contains a report management page which handles 2.1.12 use-case by allowing administrators to browse, delete and generate new CSV reports.

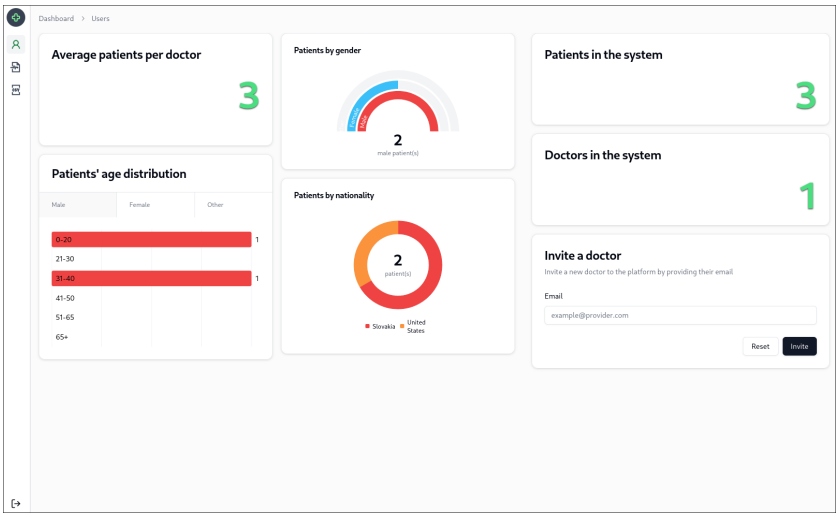

Figure 5.4: Administrator page displaying user statistics

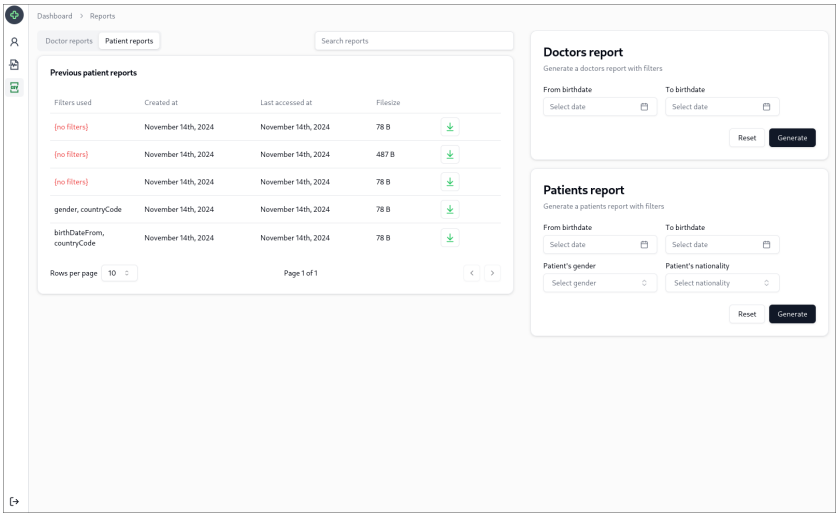

Figure 5.5: Page for report management

Doctor pages

The doctor dashboard also consists of three separate pages. The first one is visually quite similar to the administrator's task management

page, the main difference being that this page does not manage the global tasks but only the tasks relative to the signed-in doctor. Additionally, it allows doctors to assign tasks to their patients.

Next, doctors can browse through their patients, as well as see useful information about them on the patient management page that handles 2.1.6. By clicking on any patient displayed on this page, doctors are able to navigate to a single patient management page that handles 2.1.7, 2.1.10, 2.1.8 use-cases.

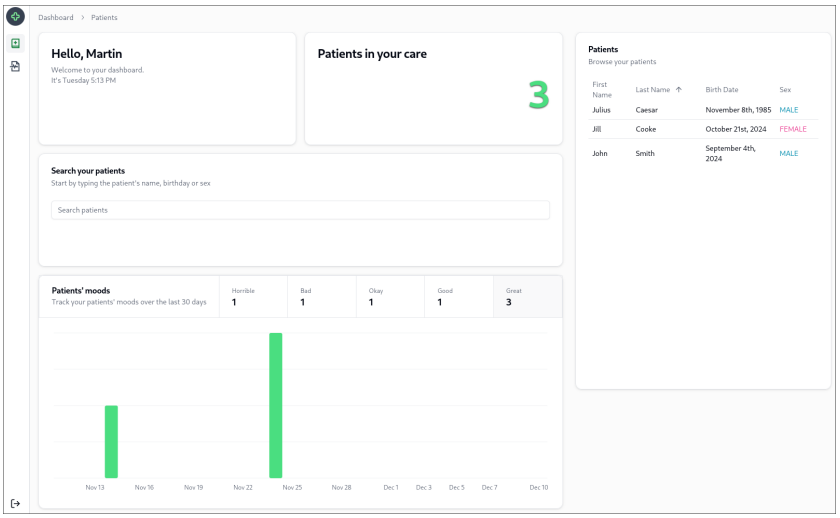

Figure 5.6: Doctor's page displaying patients

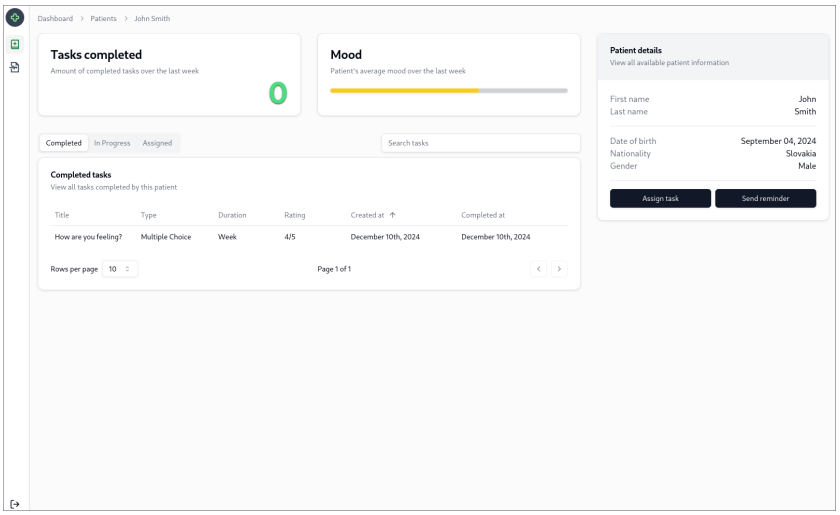

Figure 5.7: A single patient's details screen

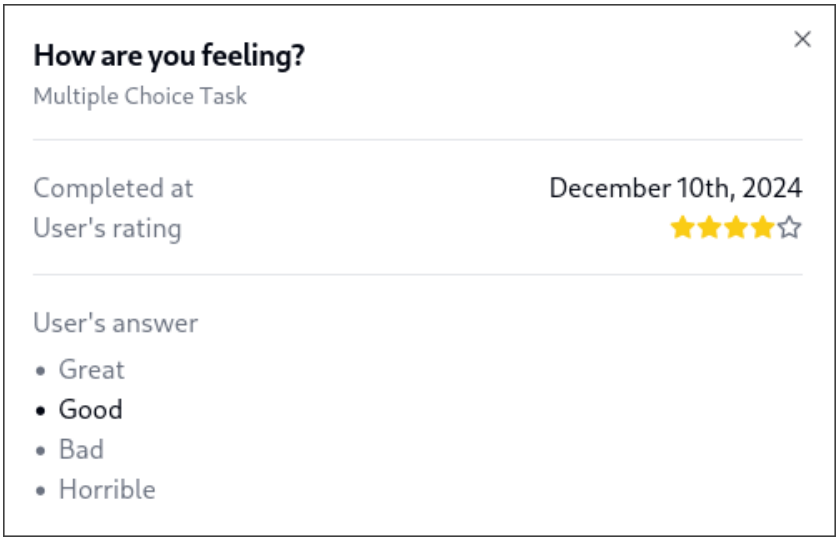

Figure 5.8: Dialog window showing patient's answer to a tasks

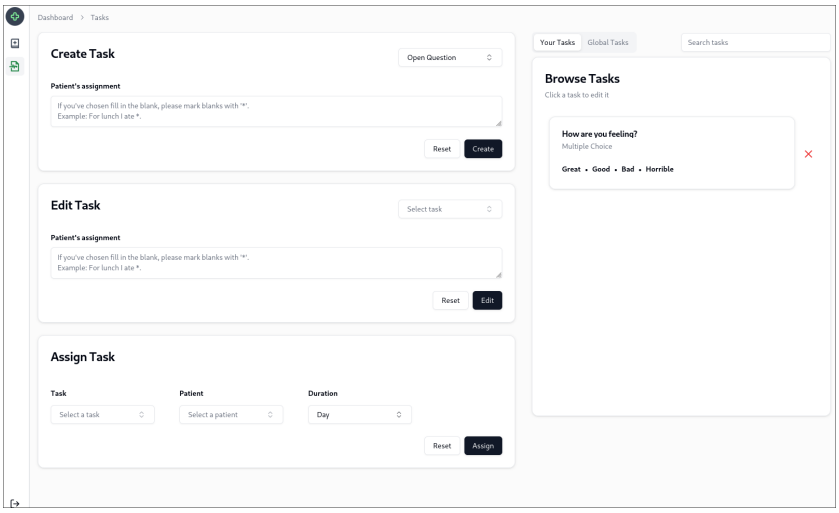

Figure 5.9: Tasks management page

5.4 Communication with the back-end API

Instead of only using a browser fetch API [63], the front-end application makes use of React Query discussed in 3.2.5. As mentioned before, it takes care of caching and re-fetching, which is massively beneficial for this application as it avoids repeated page loads and effectively speeds up the application. In addition to React Query, Axios [64] is used as a replacement for `fetch()`, as it manages automatic serialization and simplifies the code. Unfortunately, it is impossible to add an `Authorization` header at instantiation as the application does not possess the user's JWT token.

```
export const axiosClient = axios.create({
  baseURL: env.VITE_BACKEND_URL,
  headers: {
    "Content-Type": "application/json",
  },
});
export const queryClient = new QueryClient()
```

Listing 5.9: Instantioation of React Query and Axios clients

Queries made by React Query are required to contain a `queryKey` (a list of strings that uniquely identifies a query). This fact allows the application to easily update its state by fetching the new data when any mutation succeeds, by invalidating a query with a particular query key. This way, the data in the application always stays fresh.

```
const onTaskDelete = async (id: string) => {
  await deleteTask.mutateAsync(id, {
    onSuccess: () => {
      queryClient.invalidateQueries({ queryKey
         : ["tasks"] });
      toast.success("Task deleted successfully
         ");
    },
    onError: () => {
      toast.error("Failed to delete task");
    }
  });
}
```

Listing 5.10: Invalidation of query when mutation succeeds

In addition to invalidation queries, this application also uses the `onSuccess` and `onError` callbacks to display information to the user about the success of the data fetching or mutation by invoking a toast component [65].

5.5 Data visualization

For administrators and doctors to be able to quickly assess the state of the patients in the system, the data needs to be presented in a concise and straightforward way [66]. To achieve this, the application application uses a library called Recharts in conjunction with Shadcn to create visually pleasing and responsive charts [67, 35].

On the doctor's side of the application, a chart is used to display data about the moods of patients to allow doctors to see the overall mood trends of their patients over the last 30 days.

On the administrator's side, multiple charts are present to inform them about patient demographic splits. Charts can take on multiple

forms in order to accommodate different datasets. On the one hand, a chart with a precisely limited number of data points, such as a gender distribution, a simple radial graph will suffice. On the other hand, datasets such as nationality distribution, where a few substantial data points usually prevail among many others and users are rather interested in seeing the relation of the data points to each other than their sheer amount, is better suited for a pie chart [68].

6 Documentation and Testing

This chapter examines the documentation and testing done on both sides of the system, the back end and the dashboard, in order to enhance its usability.

6.1 Back-end documentation

Documentation is crucial when creating any API that expected to be used by other developers. It is necessary to document all API endpoints, the requests they accept, and the potential responses they return. This project uses Swagger UI [31] to achieve interactive API documentation. Swagger UI can generate an HTML document by using just a single JSON configuration file that contains details about the application's endpoints. These details include endpoint's request and response body, required and returned headers, and potential error codes. Other developers can then browse the site and even try the endpoints' functionality. This application hosts its documentation on a `/api-docs` route.

```
import swaggerUi from "swagger-ui-express";
import swaggerDoc from "./swagger.json";
app.use("/api-docs", swaggerUi.serve,
   swaggerUi.setup(swaggerDoc));
```

Listing 6.1: Configuration required to serve Swagger documentation

6.2 Front-end documentation

Creating usable documentation for web applications is a difficult. A study examining this problem suggests that users prefer to explore and make mistakes on their own and often use the application with only skimming, or not reading the manual at all [69]. As the dashboard contains a few pages with relatively straightforward functionality, it is best to avoid a manual altogether and focus more on the application's usability. This is achieved by documenting and describing every piece of functionality the site offers not in a manual, but the site itself.

That way, cards have descriptions specifying their utility, graphs have on-hover effects with further information, and input elements have placeholder values informing users about the desired purpose. In addition, on any action's failure, a message is shown to the user notifying them of the fact. Lastly, all forms contain potential error messages that communicate any issues to users.

Figure 6.1: Notification informing user of action's result

6.3 Back-end testing

In order to test the created API, this thesis utilizes a tool called Postman. Postman is used for managing, testing, documenting, and developing APIs [70]. It allows for thorough API testing by creating a collection of requests one can run to test the API against the contract set by the API's documentation described in 6.1. This way, keeping the API functionality tied to the documentation through the development process was possible.

6.4 Front-end testing

Four different people have tested the front-end dashboard, three of them acting as doctors and one administrator using accounts with predefined data. The testing aimed to find issues with the usability of the dashboard in relation to the application's intuitiveness, speed, and window responsiveness when performing requested tasks, such as task creation, report creation or task review. All of the testers described the application's speed being comparable to other web applications that they have used in the past. Issues in intuitiveness have been rectified by improving documentation in accordance with 6.2. Finally,

shortcomings in resizing responsiveness, such as elements moving out of the bounding box of the browser or undesirable scrolling, have been fixed by appropriate code changes.

7 Conclusion

This work aimed to create a working REST API that a mobile application could consume, focusing on a user's mental health. Alongside that, a web application was developed for doctors and administrators alike to manage the users of the potential mobile application. The system successfully implements the requirements presented in the chapter 2. The system has been tested and documented as described in 6. During the development process, several hurdles had to be overcome, one example being Supabase's changes to the default email provider [71], once again proving the downsides of using a BaaS provider, mentioned in 1.2.1. In spite of that, a successful implementation has been delivered.

7.1 Potential improvements

There are aspects of this solution that can definitely be improved in the future, both from the functional and non-functional sides.

Real-time chat functionality

A potential quality-of-life improvement for doctors and patients would be the ability to hold a real-time conversation. This feature would require implementation changes and additions to the both back-end API and the front-end application. The author believes it is a valuable piece of functionality to implement.

Server-side pagination

Paginated elements presented within the web application are currently paginated on the client side. This currently presents no issues as the application is relatively small and has a limited number of users. However, as the user base potentially expands, paginating excessive number of records directly in the web browser could prove too slow. By implementing this change, the potential issues in the future can be avoided.

Bibliography

1. *Daylio Journal - Mood Tracker*. Habitics s.r.o., 2015-08-17. Version 1.59.0. Available also from: https://play.google.com/store/apps/details?id=net.daylio.
2. *Voidpet Garden: Mental Health*. Voidpet, 2023-02-07. Version 3.7.10. Available also from: https://play.google.com/store/apps/details?id=com.voidpet.
3. CLOUDFLARE. *What is BaaS?* [online]. [visited on 2024-11-20]. Available from: https://www.cloudflare.com/learning/serverless/glossary/backend-as-a-service-baas.
4. ZALESKI, M. *What Is a Mobile App Backend and Does Your Mobile Application Need It?* [online]. 2024-01-08. [visited on 2024-11-20]. Available from: https://www.nomtek.com/blog/mobile-app-backend.
5. CHUCHAM, S. BACKEND-AS-A-SERVICE. *Medium* [online]. 2024 [visited on 2024-11-20]. Available from: https://medium.com/@6531503085/backend-as-a-service-c1165195e0a9.
6. SUPABASE. *Predictable pricing, designed to scale* [online]. [visited on 2024-11-20]. Available from: https://supabase.com/pricing.
7. DEL ALBA, L. *Best Backend as a Service (BaaS) in 2024* [online]. 2023-08-10. [visited on 2024-11-20]. Available from: https://www.sitepoint.com/best-backend-as-a-service-baas-in-2023/#topbaasprovidersin2023.
8. FOURRAGE, L. *How to integrate mobile apps with backend services?* [online]. 2024-07-06. [visited on 2024-11-20]. Available from: https://www.nucamp.co/blog/coding-bootcamp-full-stack-web-and-mobile-development-how-to-integrate-mobile-apps-with-backend-services.
9. MILLER, Ch. *From SOAP to REST: Tracing The History of APIs* [online]. 2023. [visited on 2024-11-20]. Available from: https://blog.treblle.com/from-soap-to-rest-tracing-the-history-of-apis/#apis-are-a-gateway-to-connectivity.

10. SHINDE, M. *REST API vs. SOAP API: Understanding the Key Differences for Your Project* [online]. 2024-08-16. [visited on 2024-11-20]. Available from: https://mayurashinde.medium.com/rest-api-vs-soap-api-understanding-the-key-differences-for-your-project-906073912ec9.

11. CONRARDY, R. *GraphQL: Pros & Cons* [online]. 2023-07-21. [visited on 2024-11-20]. Available from: https://medium.com/@conrardy/graphql-pros-cons-23f8995752eb.

12. FIGMA. *What is a use case?* [online]. [visited on 2024-11-21]. Available from: https://www.figma.com/resource-library/what-is-a-use-case/.

13. CHUNG, Lawrence; PRADO LEITE, Julio Cesar Sampaio do. On Non-Functional Requirements in Software Engineering. In: Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. ISBN 978-3-642-02463-4. Available also from: https://doi.org/10.1007/978-3-642-02463-4_19.

14. FIREBASE. *Pricing plans* [online]. [visited on 2024-11-25]. Available from: https://firebase.google.com/pricing.

15. FIREBASE. *Cloud Firestore* [online]. [visited on 2024-11-25]. Available from: https://firebase.google.com/docs/firestore.

16. AWS. *AWS Amplify Pricing* [online]. [visited on 2024-11-25]. Available from: https://aws.amazon.com/amplify/pricing.

17. AWS. *DynamoDB* [online]. [visited on 2024-11-25]. Available from: https://aws.amazon.com/dynamodb/.

18. AWS. *Connect API to existing MySQL or PostgreSQL database* [online]. [visited on 2024-11-25]. Available from: https://docs.amplify.aws/gen1/javascript/build-a-backend/graphqlapi/connect-api-to-existing-database/.

19. SUPABASE. *Open Source SQL Database*(*without the hassle*) [online]. [visited on 2024-11-25]. Available from: https://supabase.com/database.

20. POSTGRESQL. *Chapter 9. Functions and Operators* [online]. [visited on 2024-11-25]. Available from: https://www.postgresql.org/docs/17/functions.html.

21. POSTGRESQL. *Software Catalogue - PostgreSQL extensions* [online]. [visited on 2024-11-25]. Available from: https://www.postgresql.org/download/products/6-postgresql-extensions/.

22. EXPRESS. *Fast, unopinionated, minimalist web framework for Node.js* [online]. [visited on 2024-11-25]. Available from: https://expressjs.com/.

23. EXPRESS. *Fast, unopinionated, minimalist web framework for Node.js* [online]. [visited on 2024-11-25]. Available from: https://expressjs.com/en/resources/middleware.html.

24. REACT. *The library for web and native user interfaces* [online]. [visited on 2024-11-25]. Available from: https://react.dev/.

25. REACT. *Built-in React Hooks* [online]. [visited on 2024-11-25]. Available from: https://react.dev/reference/react/hooks.

26. ROUTER, React. *React Router* [online]. [visited on 2024-11-25]. Available from: https://reactrouter.com/.

27. JONES M. Bradley J., Sakimura N. *JSON Web Token (JWT)* [Internet Requests for Comments]. RFC Editor, 2015-05. RFC, 7519. RFC Editor. ISSN 2070-1721. Available also from: https://www.rfc-editor.org/rfc/rfc7519.txt.

28. VERCEL. *Make Ship Happen* [online]. [visited on 2024-11-25]. Available from: https://turbo.build/.

29. PRISMA. *PrismaORM* [online]. [visited on 2024-11-25]. Available from: https://www.prisma.io/.

30. ZOD. *TypeScript-first schema validation with static type inference* [online]. [visited on 2024-11-25]. Available from: https://zod.dev/?id=introduction.

31. SWAGGER. *SwaggerUI* [online]. [visited on 2024-11-25]. Available from: https://swagger.io/tools/swagger-ui/.

32. TANSTACK. *TanStack Query* [online]. [visited on 2024-11-25]. Available from: https://tanstack.com/query/latest.

33. TAILWINDCSS. *Rapidly build modern websites without ever leaving your HTML.* [online]. [visited on 2024-11-25]. Available from: https://tailwindcss.com/.

34. TAILWINDCSS. *Get started with Tailwind CSS* [online]. [visited on 2024-11-25]. Available from: https://tailwindcss.com/docs/installation/framework-guides.

35. SHADCN/UI. *Build your component library* [online]. [visited on 2024-11-25]. Available from: https://ui.shadcn.com/.

36. UI, Chakra. *Chakra UI is a component system for building products with speed* [online]. [visited on 2024-11-25]. Available from: https://www.chakra-ui.com/.

37. UI, Material. *Move faster with intuitive React UI tools* [online]. [visited on 2024-11-25]. Available from: https://mui.com/.

38. SUPABASE. *JWTs* [online]. [visited on 2024-11-26]. Available from: https://supabase.com/docs/guides/auth/jwts#jwts-in-supabase.

39. EXPRESS. *Writing middleware for use in Express apps* [online]. [visited on 2024-11-26]. Available from: https://expressjs.com/en/guide/writing-middleware.html.

40. SUPABASE. *Custom Claims & Role-based Access Control (RBAC)* [online]. [visited on 2024-11-26]. Available from: https://supabase.com/docs/guides/database/postgres/custom-claims-and-role-based-access-control-rbac.

41. POSTRESQL. *PL/pgSQL — SQL Procedural Language* [online]. [visited on 2024-11-26]. Available from: https://www.postgresql.org/docs/current/plpgsql-overview.html.

42. S., Mannava. *What are Claims? When do you use them?* [online]. [visited on 2024-11-26]. Available from: https://isriramkumarm.medium.com/what-are-claims-when-do-you-use-them-b15603d6fef2.

43. PRISMA. *Prisma Schema* [online]. [visited on 2024-11-26]. Available from: https://www.prisma.io/docs/orm/prisma-schema/overview.

44. PRISMA. *Generating Prisma Client* [online]. [visited on 2024-11-26]. Available from: https://www.prisma.io/docs/orm/prisma-client/setup-and-configuration/generating-prisma-client.

45. AU-YEUNG. *Best practices for REST API design* [online]. [visited on 2024-11-26]. Available from: https://stackoverflow.blog/2020/03/02/best-practices-for-rest-api-design/.

46. SUPABASE. *Sending Push Notifications* [online]. [visited on 2024-11-27]. Available from: https://supabase.com/docs/guides/functions/examples/push-notifications?queryGroups=platform&platform=fcm.

47. SUPABASE. *Deploy JavaScript globally in seconds* [online]. [visited on 2024-11-27]. Available from: https://supabase.com/edge-functions.

48. GOOGLE. *Firebase Cloud Messaging* [online]. [visited on 2024-11-27]. Available from: https://firebase.google.com/docs/cloud-messaging.

49. GOOGLE. *Detect invalid token responses from the FCM backend* [online]. [visited on 2024-11-27]. Available from: https://firebase.google.com/docs/cloud-messaging/manage-tokens#detect-invalid-token-responses-from-the-fcm-backend.

50. SUPABASE. *Store and serve any type of digital content* [online]. [visited on 2024-12-04]. Available from: https://supabase.com/storage.

51. Y., Shafranovich. *Common Format and MIME Type for Comma-Separated Values (CSV) Files* [Internet Requests for Comments]. RFC Editor, 2005-10. RFC, 4180. RFC Editor. Available also from: https://www.rfc-editor.org/rfc/rfc4180.txt.

52. SUPABASE. *Storage Quickstart* [online]. [visited on 2024-12-04]. Available from: https://supabase.com/docs/guides/storage/quickstart.

53. FREED N., Borenstein N. *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies* [Internet Requests for Comments]. RFC Editor, 1996-11. RFC, 2045. RFC Editor. Available also from: https://www.rfc-editor.org/rfc/rfc2045.txt.

54. FREED N., Borenstein N. *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types* [Internet Requests for Comments]. RFC Editor, 1996-11. RFC, 2046. RFC Editor. Available also from: https://www.rfc-editor.org/rfc/rfc2046.txt.

55. K., Moore. *MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text* [Internet Requests for Comments]. RFC Editor, 1996-11. RFC, 2047. RFC Editor. Available also from: https://www.rfc-editor.org/rfc/rfc2047.txt.

56. SUPABASE. *Create a signed URL* [online]. [visited on 2024-12-04]. Available from: https://supabase.com/docs/reference/javascript/storage-from-createsignedurl.

57. MOZILLA. *SPA (Single-page application)* [online]. [visited on 2024-12-01]. Available from: https://developer.mozilla.org/en-US/docs/Glossary/SPA.

58. SUPABASE. *Auth* [online]. [visited on 2024-12-01]. Available from: https://supabase.com/docs/guides/auth.

59. MOZILLA. *Window: localStorage property* [online]. [visited on 2024-12-01]. Available from: https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage.

60. REACT. *useContext* [online]. [visited on 2024-12-01]. Available from: https://react.dev/reference/react/useContext.

61. SUPABASE. *Passwordless email logins* [online]. [visited on 2024-12-02]. Available from: https://supabase.com/docs/guides/auth/auth-email-passwordless.

62. ROUTER, React. *Route Module* [online]. [visited on 2024-12-02]. Available from: https://reactrouter.com/start/framework/route-module#loader.

63. MOZILLA. *Fetch API* [online]. [visited on 2024-12-02]. Available from: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API.

64. AXIOS. *Promise based HTTP client for the browser and node.js* [online]. [visited on 2024-12-02]. Available from: https://axios-http.com/.

65. SHADCN/UI. *Sonner* [online]. [visited on 2024-12-02]. Available from: https://ui.shadcn.com/docs/components/sonner.

66. A., Morris. *What Is a Chart & Why Is It Important for Businesses?* [online]. [visited on 2024-12-02]. Available from: https://www.netsuite.com/portal/resource/articles/erp/chart.shtml.

67. RECHARTS. *A composable charting library built on React components* [online]. [visited on 2024-12-02]. Available from: https://recharts.org/en-US/.

68. YI M., Sapountzis M. *Essential chart types for data visualization* [online]. [visited on 2024-12-02]. Available from: https://www.atlassian.com/data/charts/essential-chart-types-for-data-visualization.

69. A., Blackler et al. Life Is Too Short to RTFM: How Users Relate to Documentation and Excess Features in Consumer Products. *Interacting with Computers*. 2016, vol. 28, no. 1, pp. 27–46.

70. POSTMAN. *AI is powered by APIs. APIs are powered by Postman.* [online]. [visited on 2024-12-05]. Available from: https://www.postman.com/.

71. KANGMINGTAY. *Supabase Auth: Changes to default email provider* [online]. 2024-09-18. [visited on 2024-12-06]. Available from: https://github.com/orgs/supabase/discussions/29370.