Tampere University of Applied Sciences

MealBudget App

Developing a mobile application for calculating the cost of a meal with React Native and Expo

Juska Kellokumpu

BACHELOR'S THESIS
June 2025

Software Engineering

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in Software Engineering

KELLOKUMPU, JUSKA:
MealBudget App
Developing a mobile application for calculating the cost of a meal with React Native and Expo

Bachelor's thesis 46 pages
June 2025

The objective of this practice-based thesis was to create a mobile application for calculating costs of meals. The technologies used in the project were React Native, Expo, Node.js, Express, and PostgreSQL. The theory part covers these technologies, and the practical part goes through the development process of the application. The application was named as MealBudget.

In the beginning of the development the minimum viable product was defined. The application should at least have a pantry section for saving ingredients, a recipes section for saving recipes and a way to link ingredients to recipes. After the ingredients would be linked to the recipe, the total cost and a cost of a portion would be calculated for the recipe based on the cost of the ingredients. Nice-to-have features were also defined. They included a grocery list, nutrition info, saving recipes from the internet and price history for ingredients. After the design phase, the programming of the application was carried out.

The result is a working mobile application that is able to calculate the cost of the recipe based on the linked ingredients. The goal of a minimum viable application was reached, yet the nice-to-have features were not implemented. Due to time constraints the focus was placed on enhancing the existing features instead of adding new ones. Finishing tasks and fixing issues took more time than intended.

A number of suggestions were made for future development. The current cost calculation is a simple one so it should be changed to account for different densities in ingredients. Besides the nice-to-have features mentioned above, the application could benefit from more pleasing styling and small quality of life improvements that make the user experience smoother. Adding proper testing would ensure that the functionality of the application stays good when new features are added. After the application is improved, it could be deployed to the public.

Key words: mobile application development, react native, expo

CONTENTS

1 INTRODUCTION 5
2 MOBILE APPLICATION DEVELOPMENT 6
2.1 React Native 7
2.2 React fundamentals 8
2.2.1 Core concepts 9
2.2.2 Hooks 10
2.3 Expo 10
2.3.1 Expo Go and development build 11
2.3.2 Expo Router 12
3 BACKEND 14
3.1 Node.js and Express 14
3.2 Database 15
3.2.1 Comparing PostgreSQL and MongoDB 16
4 APP DEVELOPMENT 18
4.1 The idea 18
4.2 Market research 18
4.3 Designing the app 20
4.3.1 Frontend design 20
4.3.2 Backend design 22
4.4 Setting up the environment 24
4.5 The development process 24
4.5.1 Creating basic pantry and recipes sections 24
4.5.2 Connecting ingredients and recipes 28
4.5.3 Recipe calculation 30
4.5.4 Enhancing the app 31
5 SHOWCASING THE APP 34
6 DISCUSSION 42
REFERENCES 45

GLOSSARY

SDK	Software Development Kit
UI	User Interface
DOM	Document Object Model
HTML	Hypertext Markup Language
CSS	Cascading Style Sheets
JSX	JavaScript XML
API	Application Programming Interface
REST	Representational State Transfer
SQL	Structured Query Language
CRUD	Create, Read, Update, Delete
NoSQL	Not Only SQL
BSON	Binary JSON
JSON	JavaScript Object Notation
ACID	Atomicity, Consistency, Isolation, Durability
MVP	Minimum Viable Product
ER	Entity Relationship

1 INTRODUCTION

This thesis focuses on the development of a mobile application for calculating meal costs. The aim is to create a mobile application that allows the user to save ingredients with their prices and then link the ingredients to recipes to calculate how much one meal actually costs. This will help the user to be aware of how much money is going to food and what kind of meals are cheaper. Most recipe web sites and applications do not include the costs in their recipes.

The name of the application is MealBudget. The project is made with React Native, Expo, Node.js and PostgreSQL. The theory section of the thesis will focus on explaining the technologies. During the thesis process the development of the app is started and a working version is made. The development process is explained in the practical part of the thesis.

The minimum working version should have at least a pantry section for saving ingredients, a recipes section for saving recipes and a way to link ingredients and recipes together to display the cost. Other features would be nice to have, such as a grocery list, nutritional information, and price history for the ingredients. Due to time constraints some of the features might not be implemented. However, there are plans to continue the development of the app after the thesis is complete.

2 MOBILE APPLICATION DEVELOPMENT

Mobile application development means the process of developing software applications that run on mobile devices, like smartphones and tablets. It includes creating software bundles that can be installed on mobile devices, implementing backend services, and testing the application on the actual devices. The two dominant device platforms are iOS from Apple Inc. and Android from Google. iOS is only available for Apple devices, while Android operating system is used by Google devices and many other mobile device manufacturers. Developing applications for iOS and Android require different SDKs and toolchains. (Amazon, n.d.-a.) Many development teams choose Android first, since around 70 percent of smartphones run Android. Additionally, Apple App Store has more restrictions so developers can find it easier to publish their apps on the Google Play Store instead. (IBM, n.d.)

There are four different types of mobile applications. They are called native apps, cross-platform apps, hybrid apps, and progressive web apps. Native apps run directly on the operating system, and they are written in the programming language and frameworks that are provided by the platform. Native apps have full access to the hardware and features of the devices and the best runtime performance. Codebases are required for each platform. Cross-platform apps can be written in a desired language, and the code is compiled for each operating system to run natively on the device. Even if the app is developed for multiple platforms a single codebase can be used but bridging the code requires more work and the app can have performance limitations. (Amazon n.d.-a; Microsoft n.d.)

Hybrid apps are built with standard web technologies. The web application is encapsulated in a container that allows the application to act like a native application. Same codebase can be shared between web and mobile applications, but the performance is lower. Progressive web apps are web applications that can provide a mobile app like user experience while utilizing browser capabilities. They are accessible through a URL, so app store delivery and installation are not needed. However, there is limited support for native device features. (Amazon n.d.; Microsoft n.d.)

2.1 React Native

React Native is an open-source mobile app framework which is based on JavaScript. It can be used to build native iOS and Android applications, meaning it can be used for cross-platform development. React Native provides UI components that are platform agnostic. Those components map directly to a platform's native UI components. Meta released React Native in 2015. In 2018 it had the 2nd highest number of contributors in GitHub. It is used in many popular apps, such as Facebook, Discord, and Microsoft Teams. (React Native, n.d.)

Benefits of using React Native include code reusability, fast refresh, large community, and cost efficiency. Reusing the code for multiple platforms and being able to see changes to the app without building it speeds up development time. Since React Native is open-source, developers can contribute to the framework's development and create their own tools and libraries. Large community also means that help and guides are easily available. Developers can use the same code to build apps for multiple platforms, which means that one team can handle developing for both iOS and Android. Developers don't necessarily need to learn platform specific languages. (Budziński, n.d.)

Potential drawbacks also exist. There is a lack of some custom modules, which might require teams to have multiple separate codebases instead of one. Package compatibility and debugging issues can slow down the development speed. Most of the time React Native can scale well but some companies have changed from React Native to native development when faced with scalability issues. (Budziński, n.d.)

A view is the fundamental part of UI in Android and iOS development. Views can contain other views, display text and images, or be interactive. In Android development views are written in Java or Kotlin, while in iOS development either Swift or Objective-C is used. With React Native one can use JavaScript by using React components. React Native will create the corresponding native views at runtime. These platform-backed components are called Native Components. React Native also includes Core Components, a set of ready-to-use Native Components that

can be used to build an app straight away. Common ones are shown in Figure 1. You can also build your own Native Components or use community-contributed components which are available at React Native Directory website (https://react-native.directory/). React Native uses the same API structure as React components, so it's important to understand how React component APIs work. (React Native, 2025a.)

React Native UI Component	Android View	iOS View	Web Analog	Description
`<View>`	`<ViewGroup>`	`<UIView>`	A non-scrolling `<div>`	A container that supports layout with flexbox, style, some touch handling, and accessibility controls
`<Text>`	`<TextView>`	`<UITextView>`	`<p>`	Displays, styles, and nests strings of text and even handles touch events
`<Image>`	`<ImageView>`	`<UIImageView>`	`<img>`	Displays different types of images
`<ScrollView>`	`<ScrollView>`	`<UIScrollView>`	`<div>`	A generic scrolling container that can contain multiple components and views
`<TextInput>`	`<EditText>`	`<UITextField>`	`<input type="text">`	Allows the user to enter text

Figure 1. Common Core Components and their corresponding components in Android and iOS with web analog as seen in React Native's documentation (2025a).

React Native is based on React, a JavaScript library for building single-page web applications, also developed by Meta. While the API structure is the same, there are some differences between them. React is used to build UI for web applications, while React Native is used for mobile applications. React uses HTML and CSS, whereas React Native utilizes native UI components and stylesheets. React can use HTML-like components, e.g. <p> and but React Native must use components like <Text> and <View> instead. While React Native uses Native API for rendering components. React uses a virtual DOM to render browser code. DOM is a programming interface that represents a web page document and the HTML content inside of it. Virtual DOM is a virtual representation of the real DOM. This allows for a quickly updating dynamic UI with enhanced performance. (Panchasara, 2024.)

2.2 React fundamentals

2.2.1 Core concepts

According to React Native's documentation (2025b) core concepts behind React includes components, JSX, props and state. In the past, HTML was responsible for content, CSS for design and JavaScript for logic, and they were often placed in separate files. As web pages became more interactive, logic was more tightly related to content and JavaScript oversaw the HTML. For this reason, rendering logic and markup exist in same components in React. This ensures that they stay in sync. A React component is a JavaScript function that might contain some markup that React will render. JSX is a syntax extension that is used in React components to represent markup. It looks like HTML, but with stricter rules and it can display dynamic information. (React, n.d.-e) Both React and React Native use JSX (React Native, 2025b).

Props, i.e. properties, are used by components to communicate with each other. Parent components can pass some data to their child components. Props are passed to a JSX tag, and they can include any JavaScript value, e.g. objects, arrays and functions. In ready-made components props are predefined, but you can pass any props to your own custom components. A props object is the only argument React component functions accept. Usually, the props object is destructured into individual props, which is done by adding curly brackets when declaring props. Props are immutable but if a component needs to change its props, the parent component will pass it a new object, and the component can render again with the new data. (React, n.d.-c.)

The memory of the component is called state. User interaction should change what components render to the screen, so components need to remember things such as the current input value in a form or the current image displayed. Changes on the screen require a component to render again. Using local variables wouldn't work since they don't persist between renders and they wouldn't even trigger renders when they change. For that, a useState Hook is required. It consists of a state variable that holds data between renders, and a state setter function that will update the variable and tell React to re-render the component. Initial value can be set when declaring a variable with useState. State is local to the component. If the same component is rendered twice on a screen, both components will

have their own state. State is also private, as the parent component does not know anything about a child component's state and cannot change it. (React, n.d.-d.)

2.2.2 Hooks

In React, Hooks are functions whose names start with word "use". They are special functions that allow hooking into different React features. They are only available while React is rendering and thus can only be called at the top level of components or in custom Hooks. Hooks were added to React version 16.8. and they allowed using state and other features without writing a class. They provide a more direct API to React features, such as state, props, and context. With Hooks, it's possible to reuse stateful logic between components. (React, n.d.-b; React, n.d.-d.)

There are several built-in Hooks in React besides useState. The useContext hook allows even a deeply nested component to receive information from distant parents without needing to pass it as props. You could use it to pass the current UI theme to all components. The useRef hook declares a ref that can hold any data that is not used for rendering. Usually, it's used for DOM nodes or timeout IDs. The useEffect hook can be used for connecting and synchronizing with external systems, like network, browser DOM and animations. The useMemo and useCallback hooks are used to optimize re-rendering performance and to skip calculations. The useMemo hook is used to cache expensive calculations and the useCallback can be used to cache a function definition. (React, n.d.-a.)

2.3 Expo

Expo is an open-source framework for building mobile applications with React Native. It offers a multitude of tools and features, which can be installed in almost any React Native project. Expo allows features like developing complex apps entirely in JavaScript, developing mobile apps without Android Studio or Xcode, and installing natively compatible libraries from the command line. These features are not available in a bare React Native project. (Expo, 2024.)

2.3.1 Expo Go and development build

Expo also offers an app called Expo Go. It is a sandbox environment that is specifically designed for running and testing Expo projects. If you only used React Native to develop a project, you would need higher level APIs that allow for features such as routing, navigation and push notifications. It would take some time to install them, and the app must be rebuilt each time a new dependency is added. Testing a bare React Native project on a real device requires running the app through Android Studio or Xcode and connecting your phone to the computer. (Moedano, 2024.)

Prototyping and developing a mobile app is a lot quicker and simpler with the Expo Go app. The app comes with pre-installed common libraries. Your phone doesn't need to be connected to the computer, since you can download the Expo Go app on your phone and use it to scan a QR code of your app. This also makes it possible to develop an iOS application on a Windows machine if you use an iPhone. (Moedano, 2024.)

There are some limitations with Expo Go. Only libraries that are bundled in Expo Go can be used with it. Native code in Expo Go is fixed once it is built, so a package containing native code that is not built into Expo Go cannot be used. On the other hand, JavaScript-only libraries work fine. Another limitation is unexpected behavior between development and production. Expo Go does not accurately simulate features like OAuth, notifications, analytics, maps, or features that require passing an API key to native code. (Moedano, 2024.)

Expo Go is not the only way to develop apps with Expo. Another way is to use a development build. A React Native app consists of the native app bundle installed on a physical device and the JavaScript bundle that runs inside it. Expo Go app serves as the native app bundle, but with a development build you can create the native app bundle by yourself. A development build allows you to modify project configurations, install native libraries and write your own native code, giving you the full control over native runtime. Live reloading is possible with a development build, so there is no need to rebuild the app after each change unless the underlying native code is modified or a new dependency that requires native code is

installed. It is recommended to use a development build if you are going to build a real-world app. Using it gives a better idea of how the app will perform in production. (Moedano, 2024.)

2.3.2 Expo Router

One of the libraries included in Expo is Expo Router. It is a file-based router that allows managing navigation between screens, and it uses the same components on multiple platforms. An alternative router is React Navigation, which is not file-based. Using a file-based routing is beneficial, since it is a well-known concept that is easy to understand. Scaling and refactoring are easier, since files can be moved around without having to update imports. (Expo, n.d.-c.)

Expo Router has a few rules that should be followed in order to create a good file-based router. All screens or pages are files inside the “app” directory and each file has a default export that defines a distinct page. Directories inside the “app” directory can define groups of stacks or tabs. All pages have a URL that can be navigated to, which means that Expo Router supports universal deep linking. The initial route should be named as “index” as Expo Router will try to match that for the “/” URL. Files named “_layout” are rendered before any routes, and they can contain any initialization code. Components that are not routes should not be placed inside the “app” directory. Expo Router will treat every file inside the “app" directory as a route. Expo Router is built on top of React Navigation, which means that React Navigation’s documentation can be referred to for configuring and styling the navigation. (Expo, n.d.-a.)

Expo Router uses a file-based routing notation to define different navigation patterns. A simple name with no notation defines a static route, so the URL matches exactly as it appears in the file tree. Square brackets denote a dynamic route, which can be used in a file or a directory. For example, “[userName]” parameter could match different usernames in the application. Parentheses are used in directories to group up routes without affecting the URL. A file named “app/(tabs)/home” would have “/home” as its URL without the “tabs”. Route groups can be used for organization or defining complex relationships between

routes. Files named as “index” are the default routes for the directories they reside in. The “_layout” files define how routes inside a directory relate to each other. They can be arranged as a stack or tabs. Routes that start with a plus sign are used for specific purposes. One commonly used route is “+not-found”, which will catch requests that don’t match any route in the app. In the Figure 2 below, you can see one example of the file structure and notation. (Expo, n.d.-b.)

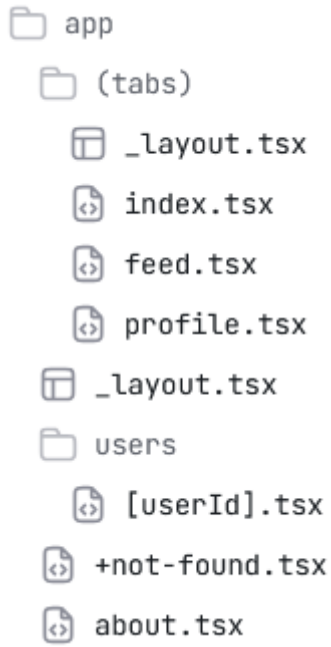

Figure 2. Example of Expo Router notation (Expo, n.d.-b).

3 BACKEND

3.1 Node.js and Express

Node.js is an asynchronous JavaScript runtime environment, which is designed for building scalable network applications. Node.js was created in 2009 by Ryan Dahl. Node.js can handle many operations concurrently since it uses an event-driven and non-blocking model. This is different from traditional server-side platforms that process requests synchronously. In a synchronous system a request will wait for a data, which can block the thread. Node.js can use modern constructs like “async/await” to manage asynchronous operations and continue processing other tasks without blocking the main thread. The event loop mechanism in Node.js is responsible for executing JavaScript code. It also handles asynchronous events. Node.js runs on a single thread but with the event loop it can handle multiple asynchronous tasks by queueing callbacks for events. (Parody, 2025.)

The advantages of using Node.js include high performance, since its event-driven architecture can handle multiple concurrent operations. The default package manager called npm includes millions of different open-source packages that add functionality to applications and simplify development. Node.js offers great scalability with its lightweight architecture and it is easy to run in containers. Because Node.js uses JavaScript, it is possible to use a single language for both client-side and server-side code. (Parody, 2025.) In mobile development you could use React Native and Node.js to build an app for both iOS and Android solely in JavaScript.

Node.js does not come without some challenges. Due to Node.js’s single-threaded event loop it can struggle with CPU-intensive tasks like heavy data processing. Asynchronous programming uses a lot of callbacks which can lead to a deeply nested code. This kind of code can be hard to read and maintain. Different libraries evolve quickly, which can introduce deprecations and breaking changes to a project. This requires developers to keep track of updates in the different libraries they use. (Parody, 2025.)

Node.js is versatile and it works well with applications that require high performance and scalability. Common use cases include real-time applications, like chat apps, streaming applications, like video and music services, APIs and microservices, IoT applications, and single-page applications, where Node.js can seamlessly integrate with frameworks like React. (Parody, 2025.)

Express is a commonly used minimalistic backend web framework for Node.js. Express provides features for handling HTTP requests, routing, middleware and error handling. It simplifies building server-side applications and APIs. Express is a lightweight layer on top of Node.js that helps in processing requests and sending responses. Express listens to an incoming request using the appropriate HTTP method. If the request matches a route defined in the code, the request is processed. Middleware, such as logging or parsing the data, can be used before the response is sent. After the request is processed, the application sends a response to the client. Custom error handling can be added with Express, in case something goes wrong during the process. (Codeacademy, n.d.)

The pros of Express are its minimalistic design and flexibility. It doesn't have unnecessary features, which makes it fast. It is also easy to customize if you want to add more features. Flexibility allows the code to be organized however you want as it doesn't enforce a specific structure or architecture. Express is perfect for RESTful APIs with its clean routing system and middleware support. From another perspective these pros can be considered as cons. The minimalistic approach means that many features, for instance authentication or validation, must be added manually. While Express is flexible, this can make it harder to define a project structure, which can lead to inconsistent code organization. Express might not be suitable for complex applications with strict architectures. (Codeacademy, n.d.)

3.2 Database

A database is used to store and organize persistent data. Most modern applications use databases. Common types are relational databases and non-relational databases. Relational databases commonly use SQL for CRUD operations: creating, reading, updating, and deleting data. Data is stored in tables consisting of

columns and rows. Tables can be joined together with foreign keys. Examples of relational databases are Microsoft SQL Server, MySQL and PostgreSQL. Non-relational databases are usually referred as NoSQL databases. These databases do not use a relational model. Their data model is different. Four kinds of NoSQL databases exist: document databases, column-oriented databases, key-value stores, and graph databases. MongoDB is one example of a NoSQL database that stores data flexibly in BSON documents. (MongoDB, n.d.)

3.2.1 Comparing PostgreSQL and MongoDB

PostgreSQL and MongoDB are different types of databases with their own distinct data models. PostgreSQL is an object-relational database management system where data is stored in tables. PostgreSQL offers scalability, concurrency, and data integrity. MongoDB is a NoSQL database where all types of data can be stores as JSON documents. It allows for a fast retrieval, analysis and replication. (Amazon, n.d.-b.)

PostgreSQL combines relational database with object-oriented features. In a table, every column represents the type of information, and every row contains individual data points. Different data types are supported. PostgreSQL uses a predefined schema which is stricter but makes the data integrity stronger. MongoDB stores data in key-value pairs, which are stores in JSON documents. Documents can hold various types of data. BSON provides additional datatypes and efficient data processing. The data model is more flexible in MongoDB, and it allows storing unstructured and dynamic data. (Amazon, n.d.-b.)

PostgreSQL and MongoDB have a few architectural differences. PostgreSQL's basic storage unit is a row, which is called a tuple. It can hold a single record under a specific data type that is specified by the column. In MongoDB, the basic unit is a serialized JSON document. The document contains key-value pairs, in which the keys are strings, and the values are types of data. PostgreSQL uses Postgres SQL as its query language, which is similar to SQL but with additional features, like functions. It is also compatible with standard SQL. On the other hand, MongoDB uses MongoDB Query Language, which is rich in features and

allows for text searches, aggregation frameworks and pipelines, and document querying. (Amazon, n.d.-b.)

Other key differences also exist between the two. PostgreSQL is ACID compliant. This means that PostgreSQL ensures that every transaction is atomic, consistent, isolated and durable. It keeps the data consistency at a high level. In MongoDB, however, ACID compliant transactions can only be used in a few limited cases. PostgreSQL defines relationships between tables with foreign keys. This allows for complicated joins, so you can query data from multiple tables at the same time. In MongoDB, predefined relationships between document collections are not used. Denormalization, embedding related data within documents, is used instead. It helps to optimize read operations, since all the needed data is present within the queried document. It also minimizes the need to join data together. (Amazon, n.d.-b.)

The application needs and the data used determine which database you should use. PostgreSQL has a structured system, which works best for data warehousing, ecommerce and web applications. Applications that need to handle complex relationships with data and that require data to be consistent. MongoDB’s flexible data model and high performance is good for content management systems and stream analysis. For example, content management applications can have a lot of unstructured data, such as videos and images. High performance works well for high-traffic applications. (Amazon, n.d.-b.)

4 APP DEVELOPMENT

4.1 The idea

When prices are increasing, it can be hard to figure out where to save money. If you live on a tight budget, doing grocery shopping and making meals by yourself instead of relying on restaurants and takeouts can save you a lot of money. Nonetheless, it can be hard to picture how much money goes to self-made meals and how much different meals cost. It would also be nice to see how much different ingredients cost.

The solution is a mobile app where you can input price of the ingredients, connect those ingredients to recipes and let the app calculate the cost per recipe and meal. Saving the prices of the ingredients can also help you see if its price has changed at the store, which is especially useful for seasonal products like fruits and vegetables, which tend to vary in price depending on the season.

This app idea has been on my mind for some time and using it as my thesis project finally gave me the push to start developing it. I usually cook for myself, and I use different recipes. I'm also quite cost-conscious, so it would be interesting to know how much money I spend on food. Originally the idea was more focused on the price of food items themselves instead of meals. I wanted to create an app where you could save the price of an item and compare it between stores to see where it was the cheapest. I wasn't quite sure where to go with that, so I shifted focus on the cost of recipes.

4.2 Market research

For the first step of the app development, I decided to do some market research. I looked up some apps and websites to see if there were any similar ones to my idea. I checked some recipe websites but usually the cost of the recipe or the ingredients were not listed. One exception to this was a website called BudgetBytes (https://www.budgetbytes.com/). In their site, they show the cost of each ingredient in the recipe and calculate a cost for the whole recipe and for one

portion. However, they are based in the US, which means that the costs are shown in dollars instead of euros. Ingredient prices can also vary a lot across the world.

A mobile application called Paprika Recipe Manager 3 is probably the closest to my idea, albeit without the cost calculation. It is a recipe manager app, but I have mostly used it to keep track of items in my pantry. The app allows you to save ingredients to a pantry, saving recipes manually or from the internet, making grocery lists and meal plans. Using the grocery list will save the bought items to the pantry. Cost of the ingredients or recipes are not supported. Paprika 3 has a premium version for unlimited recipes and cloud sync. (Paprika App, n.d.)

Kesko's K-Ruoka mobile app has a section for recipes. The recipes section looks nice, and each recipe usually displays an image, name, time to make, cost per portion and the rating. It's also possible to favorite the recipe. Recipes can be searched for, and they are shown in categories. In an individual recipe page, more details are seen, like the description and suitable diets. Each recipe contains the ingredients and instructions steps. The recipe also contains a buy section where you can save the ingredients to a grocery list or add them to a shopping cart. Some ingredients can be swapped in the buy section. Servings and the cost per serving is shown. Servings can be changed which will scale the recipe's ingredients, but this has no effect on the cost per serving. The buy section and swapping the ingredients also doesn't affect it. No cost is shown next to the ingredients so it's a bit unclear how the cost of the recipe is calculated. K-ruoka also exists as a website (https://www.k-ruoka.fi/reseptit) but the costs of the recipes are not visible there.

I also checked an app called Lose It!, which is a mobile app for tracking calories and nutrients. It was not as relevant as the others, but I was interested in its log view. The log view is a daily view where you can add food or meals you've eaten, which then shows the amount of calories consumed. I thought this feature could also be nice in my app but instead of calories it would show the amount of money you've consumed during the day. In Lose It! users can add foods that other users can also use for logging. I thought this could also be a nice idea for my app when I was more focused on the cost of food items. Users could have reported how

much some food items cost in some store and then others could see where to buy the items the cheapest. However, I moved away from that idea.

4.3 Designing the app

Before I started programming, I made some design decisions. I decided on the minimum viable product (MVP), nice-to-haves and a wish list. MVP is the version of the app with the minimum required functionality. For the MVP I decided to have

- a pantry section for saving ingredients
- a recipes section for saving recipes
- ingredients should include name, amount, unit, and cost per kg data
- recipes should include title, servings, ingredients, and instructions data
- ingredients from the pantry can be connected to a recipe and the cost of the recipe is calculated based on the ingredients.

As for the nice-to-haves I wanted to include a grocery list, nutritional info, price history, searching and sorting, and the ability to save recipes from the internet. For the wish list I placed functionality that I was quite sure I would not have time to do but maybe I could do them in the future. It included a daily view for meals eaten, meal plan section, sharing info between users and possibly adding a quick meal cost calculator, which would allow the user to add ingredients together without needing to have a proper recipe. After setting these goals I made a simple schedule for the tasks.

As I looked at different recipe apps, I came across an API called Spoonacular API. It is an API that can be used get data for ingredients, recipes, products and menu items. It includes price data for ingredients and cost breakdown for recipes. (Spoonacular, n.d.) I could have used it in this project, but I decided to go with a bare approach without it. One of the reasons was because it’s based in the US so the price data might not work for Finland.

4.3.1 Frontend design

I decided to try out Figma to design the frontend for the app. I used it to figure out what screens are needed and what they could look like. The design is displayed

in Figure 3. It resembled the Paprika 3 app's UI, at least with the way the ingredients and recipes are listed. I wanted to show an ingredient's details on a modal that would show different information about the ingredient. It would have included a price graph that would have been based on the ingredient's price history.

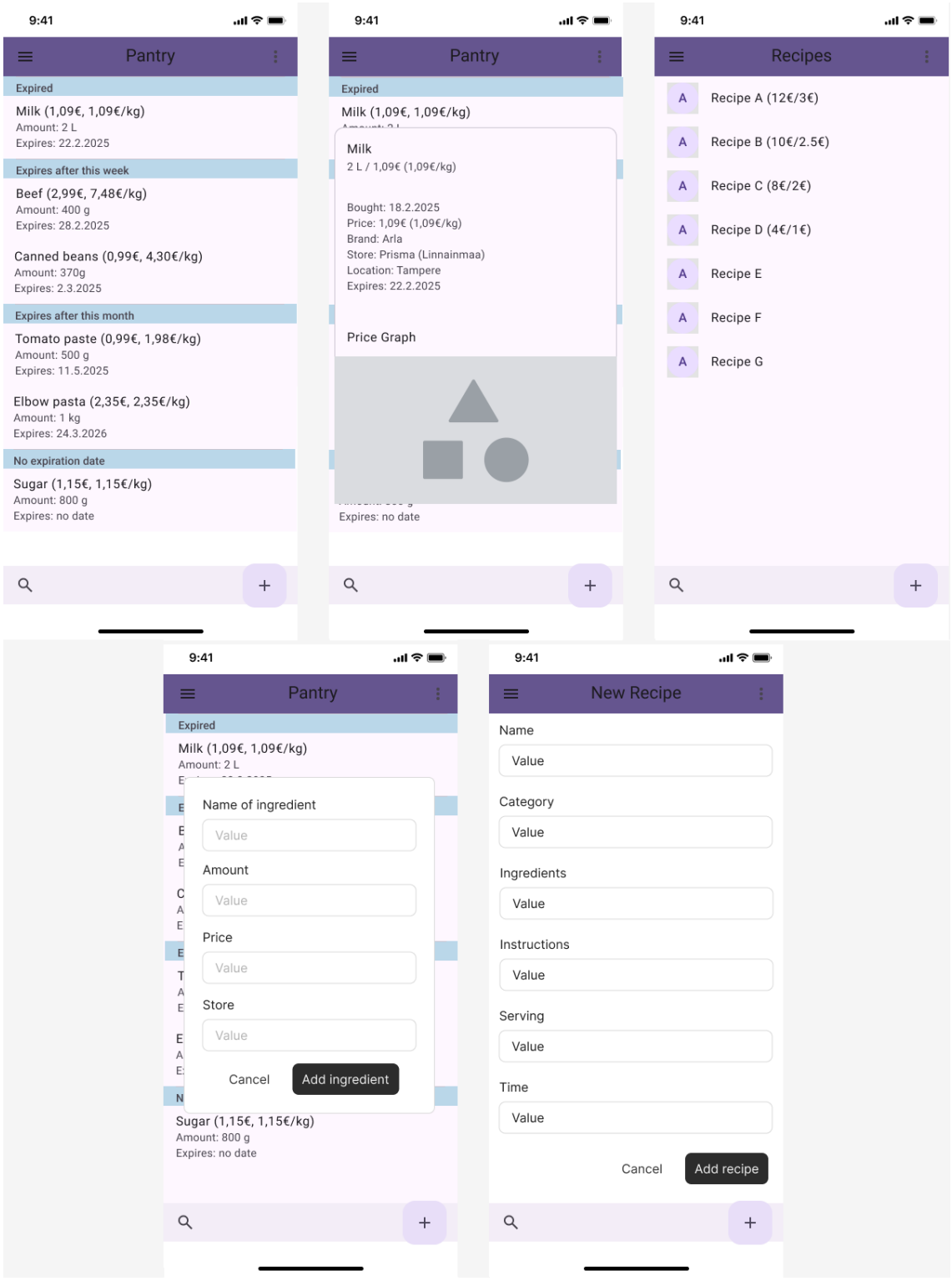

Figure 3. UI prototype made in Figma

4.3.2 Backend design

For the backend I decided to go with Node.js and Express since they are familiar to me and they’re simple to set up. At first, I wasn’t sure which database to choose. Since ingredients would have a close relationship with recipes I decided to go with PostgreSQL, since it allows for complex joins and the data integrity is high.

I made an Entity Relationship (ER) diagram to design the relational database for the app, which is visible in Figure 4. An ER diagram is a flowchart that can be used to show how different entities, e.g. people, concepts or objects, relate to each other in a system. They are commonly used in software engineering to design or debug relational databases. Different symbols depict the relationships and attributes of the entities. (Lucidchart, n.d.)

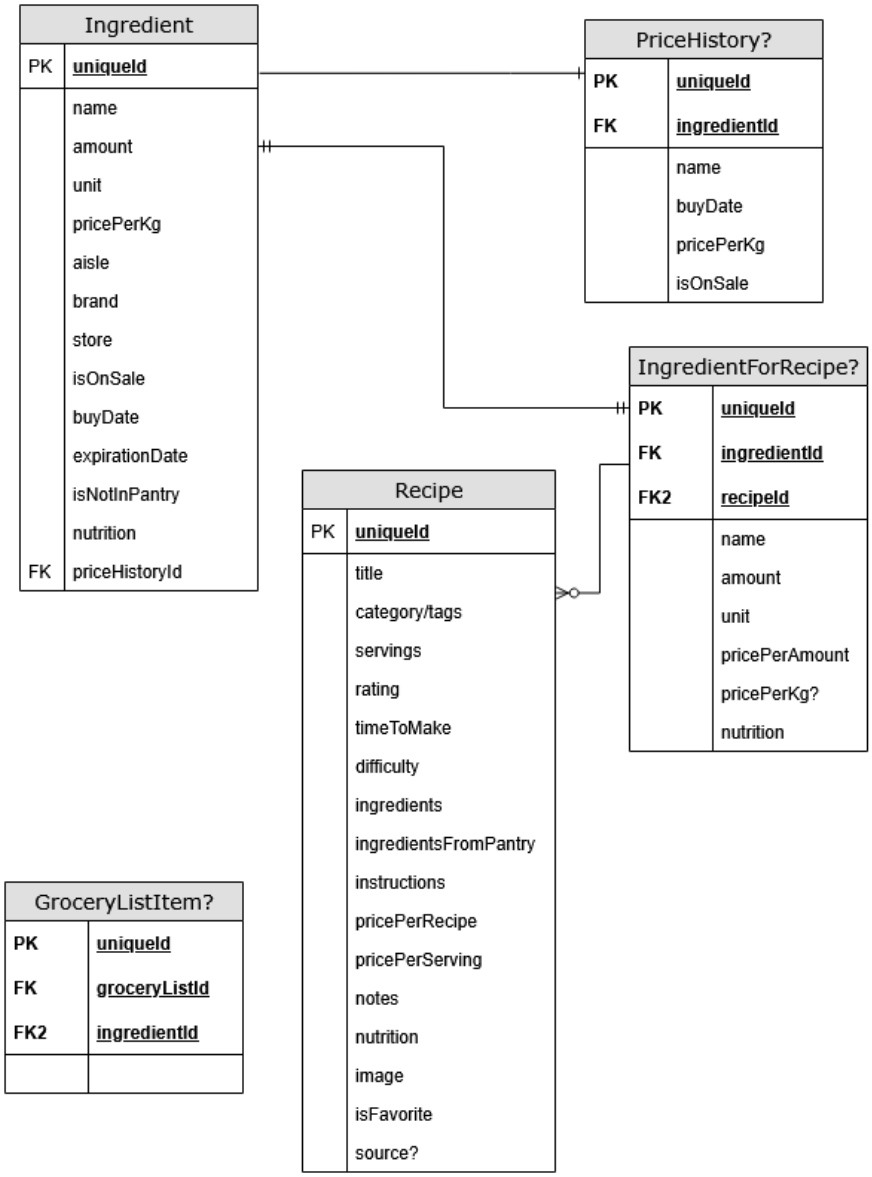

Figure 4. ER diagram for a database design

I was sure that ingredients and recipes would require their own tables and I listed the possible attributes that they might have, e.g. aisle and isOnSale for ingredients and difficulty and rating for recipes. For connecting an ingredient to a recipe I assumed that I needed a separate table, since an ingredient can belong to multiple recipes and a recipe can be related to multiple ingredients. Having a many-to-many relationship in PostgreSQL requires using a join table (Hasura, n.d). In the previous figure the relationship is denoted wrong though. You can refer to different relationship notations in ER diagrams in Figure 5. In the ER diagram the ingredient is depicted as having a one-to-one relationship with the join table and recipe is depicted as having zero-or-many relationship with the join table. In the design phase I wasn't quite sure yet what is the best way to connect ingredients

with recipes. The ER diagram also includes a price history entity that would be connected to an ingredient, and a grocery list that would be connected to an ingredient also. In this stage I was unsure about them and did not give them much thought.

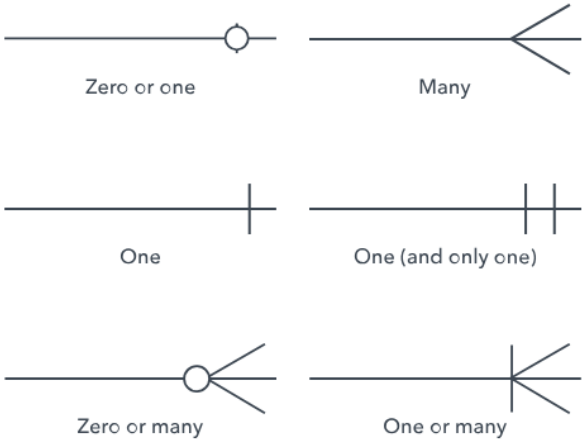

Figure 5. Screenshot of ER diagram relationship notation (Lucidchart, n.d.).

4.4 Setting up the environment

After the design phase I chose my technologies and set them up. For the frontend I decided to go with React Native and Expo. I created an example project with “npx create-expo-app@latest” command in Visual Studio Code. I downloaded Expo Go on my Android phone and tested the example project. After verifying that it works, I ran “npm run reset-project” command to remove the boilerplate code, which gave me a fresh project I could use. I set up a simple backend with Node.js and Express. I utilized pgAdmin and Postman for database and backend operations. I also set up Eslint for consistent coding style and error warnings. In GitHub I made separate projects for the frontend and the backend and I wrote tasks in GitHub Issues. I thought about using Docker but abandoned the idea due to time constraints, since setting it up would take some time.

4.5 The development process

4.5.1 Creating basic pantry and recipes sections

First, I started working on the pantry section in the backend. I set up a simple database table “ingredients” that only had four columns: name, amount, unit, and price_per_kg. All of them were required fields. In the backend code I set up routes for the ingredient for handling requests between database and the backend. I used Postman to send requests to the routes and made necessary changes until reading, adding, deleting and updating ingredient data to the database worked.

On the frontend side I first looked through Expo tutorials to see good examples of the code. I went with Expo Router and made tabs for different views, that were Home, Pantry and Recipes. In the Pantry tab I made a list that displays the ingredient data from the database. I made a modal and placed a form for adding ingredients inside of it. Then I made it possible to go to a detailed view of an ingredient with a dynamic route using the ingredient ID. In that view I added an edit button that displayed a form for editing in a modal, and a delete button that allowed for removing the specific ingredient. The initial pantry page and ingredient edit forms are visible in Figure 6 and Figure 7.

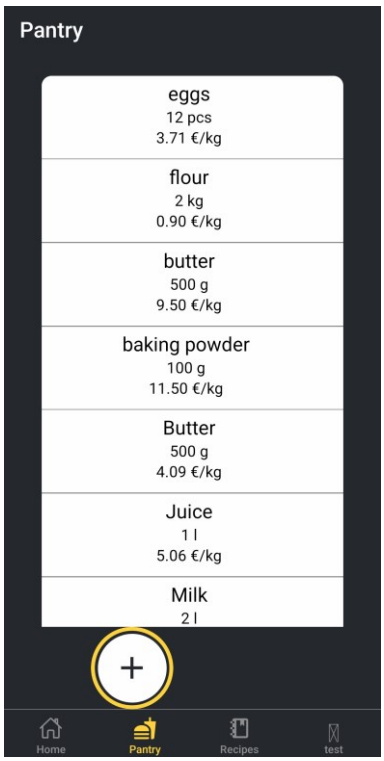

Figure 6. The initial pantry section.

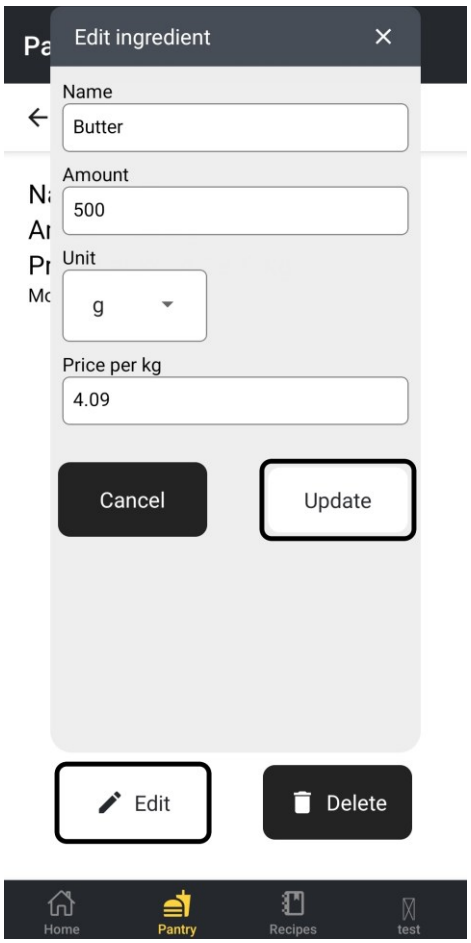

Figure 7. The initial ingredient edit form.

After the basic pantry section was done, I moved onto the recipes section. I was able to reuse a lot of the previous code. In database I made a simple table for recipes, which included required columns of title, servings, ingredients and instructions. Ingredients data type was an array of JSON objects, which consisted of key-value pairs of name, amount and unit. Instructions was an array of text type. I also added some optional columns at this stage. Programming was done in the same way as previously, by adding CRUD operations on the backend and frontend. The initial recipes section is in Figure 8 and the initial recipe edit form is in Figure 9.

Figure 8. The initial recipes section.

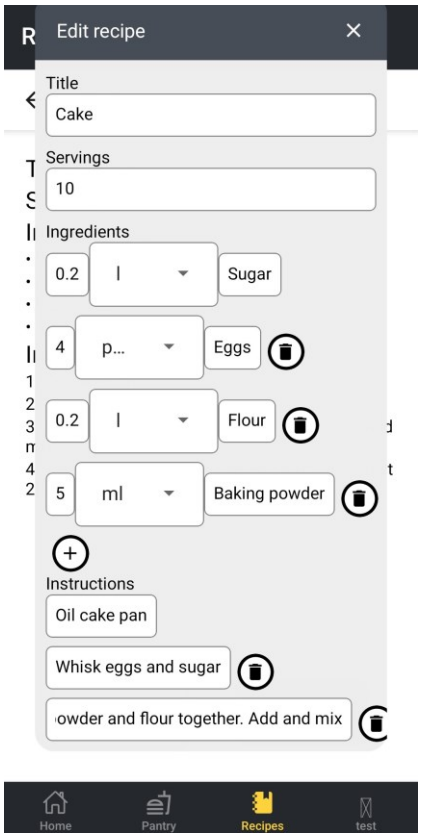

Figure 9. The initial recipe edit form.

4.5.2 Connecting ingredients and recipes

After the base for Pantry and Recipes was done, it was time to connect them. After some reading, I found out that the best way was to use join tables, where I can place foreign keys for both the recipe and the ingredient. In that way ingredients and recipes can have many-to-many relationship. I created the join table with following columns: recipe_id, ingredient_id, amount, unit, price_per_kg. Later in the project some of the columns changed into ingredient_amount, ingredient_unit and ingredient_cost, for better naming convention and since price_per_kg was not necessary and could be fetched from the ingredients table. The current code for creating the join table in the backend is visible in Figure 10. In the backend I used join queries to combine data, for example displaying the connected ingredient's name in the recipe. I added linking and unlinking an ingredient to the backend.

```
const createRecipesIngredients = `
CREATE TABLE IF NOT EXISTS recipes_ingredients (
  recipe_id INT REFERENCES recipes (recipe_id) ON UPDATE CASCADE ON DELETE CASCADE,
  ingredient_id INT REFERENCES ingredients (ingredient_id) ON UPDATE CASCADE ON DELETE CASCADE,
  CONSTRAINT recipe_ingredient_pkey PRIMARY KEY (recipe_id, ingredient_id),
  ingredient_amount NUMERIC(6, 2) NOT NULL,
  ingredient_unit VARCHAR(20) NOT NULL,
  ingredient_cost NUMERIC(6, 2)
);
`
```

Figure 10. Code for creating the join table for ingredients and recipes.

I added buttons to the recipe page for linking and unlinking an ingredient. If the ingredient is linked, the linked version of the ingredient is shown instead of the plain ingredient from the recipe. After that I made a simple placeholder calculation function that gives the cost of the ingredient if its unit is in kilograms. Then the cost of the ingredient is displayed next to its name. Figure 11 shows the linked ingredients and how in the beginning only the cost of those with kilogram units were calculated.

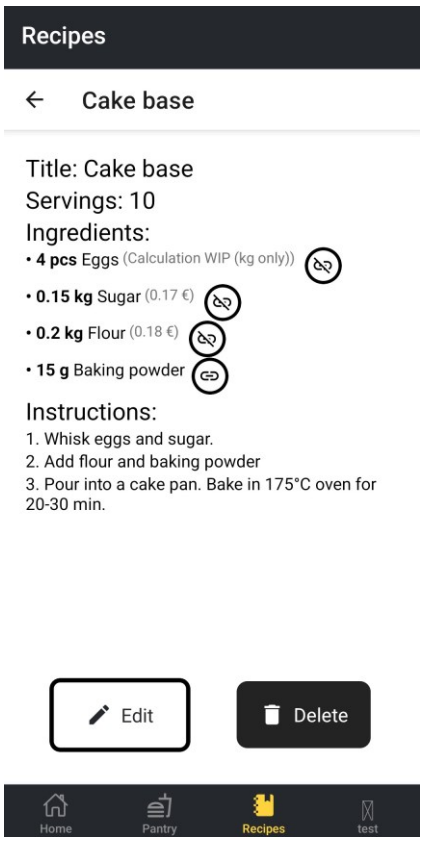

Figure 11. Recipe page with linked ingredients and their costs.

When the user presses the link button, the ingredient's name, amount and unit that are in the recipe are sent to the backend with the recipe's ID. The backend checks if the ingredient's name is found in the pantry and returns the first result. This result includes the ingredient's ID, which will be added to the join table along with recipe's ID and ingredient's amount and unit from the recipe. With that the linking is complete. Unlinking requires recipe and ingredient's IDs. If a match is found in the join table, it is deleted. Both ingredient and recipe's IDs have cascade update and delete functions in the database. If an ingredient or a recipe is deleted, their entry in the join table is also deleted.

Currently basing the linking on the name could cause some problems, if there are ingredient entries with duplicate names in the pantry or the names don't match exactly. I think a better way would be to give the user more control in choosing which ingredient to link. In the linking process the app could show the user possible options from the pantry that are similar in the ingredient name and the user could choose the best one. This is one of the features I would like to add later.

I tried adding the linking of an ingredient to recipe's add and edit forms but had some difficulties making it work. In the add form the linking fails since the recipe does not yet exist in the database and thus has no ID. Meanwhile in the edit form, if user adds a new ingredient, i.e. adds a new field to the form for the ingredients, the fields have not yet passed data to the database, so linking fails. For now, I decided that linking from the individual recipe page is enough.

4.5.3 Recipe calculation

When linking worked, I focused on the cost calculation function. Sometimes calculating the cost is not that simple, especially if the unit is a volume. Due to density the masses can vary. For example, a cup of granulated sugar weighs almost 200 grams while a cup of all-purpose flour weighs only 120 grams (Cohen & Matley, 2025).

At first, I made the cost calculation function in the frontend but later moved it to the backend. I went with a simple function that does not yet account for density. The function calculates the cost of an ingredient for the recipe, and it's based on the ingredient's price per kg, and its amount and unit in the recipe. It converts the amount into kilograms and then multiplies that with cost per kg. The cost calculation returns the cost of the ingredient for the recipe, which is saved in the join table.

For the recipes table I added cost_per_serving and total_cost columns. Cost_per_serving is a generated field, which is automatically generated when total_cost is updated. It divides the total cost by servings to get the cost per serving. Total_cost is saved for the recipe if all ingredients are linked and they all have a calculated cost. The total cost is calculated in the frontend, and it is then sent to the backend to be saved to the recipe table. After that the frontend will fetch the updated recipe and is able to display recipe's total cost and cost per serving.

During the last stages of the project, I added calculation for the unit pieces. I added columns weight_per_piece and cost_per_piece for the ingredients table, so that calculating a cost per piece would be possible. First, I wasn't sure which one to use for calculation, but I gravitated towards using weight_per_piece, since

for a user it might be easier to know. Using the weight actually worked well with the existing function, since it just changes the weight into kilograms. I added a simple field in the ingredient forms for the weight per piece that I would like the user to fill if they want to use pieces in recipes. The functionality is not fully tested yet, so some problems could appear.

4.5.4 Enhancing the app

Now that the MVP was done, it was time to enhance the app and add more features. I added more optional fields for ingredients and recipes tables and added those fields to the forms. For the recipe I added expiration and buy dates. I added a datetime picker component to the form but had some troubles. After some time, I was able to get it to work with the form component. I also added an image field to recipes and created an image picker component to be used in the forms. User can choose a picture from the gallery, take a picture with a camera or paste an image URL to the form.

In the detailed views of an ingredient and a recipe I tried to add the edit and delete buttons to the page header so that they would be more accessible. However, weirdly enough the buttons did not work, at least not in Expo Go. The buttons' onPress functions do not always fire when pressing them. It could be possible that the buttons would have worked in a development build, but I did not try that. I did not find a solution for this problem, so I left the buttons out of the header.

At this point I added backend operations for updating and deleting a linked ingredient. Before, if the ingredient was linked in the recipe but the recipe was edited, e.g. the ingredient amount was changed, this change was not reflected in the app. If the linked ingredient was removed in the edit form, it was not actually deleted, and the cost of the recipe couldn't be displayed anymore. In the edit form I added a function to compare the current array of ingredients and the new edited array of ingredients. If they are different, frontend knows to send a request to update the linked ingredients. If a linked ingredient is deleted in the edit form, the current array will still have the ingredient while the edited array will not. In that case frontend knows to send that ingredient's ID for deletion from the join table.

I added users and authentication to the app. The users table in the database is a simple one with just an ID, username and a hashed password. In the frontend I created a login and a sign-up form. I made a useAuth hook to handle authentication. I kept the authentication simple, for now a token is created in the frontend when user is logged in. Later, I would like to move the token creation to the backend. The token is saved to local memory. In Expo Router I made authenticated routes. If a user is not signed in, they are redirected to the login page. Access to the rest of the app is allowed if the token exists. Recipes and ingredients tables have foreign keys for user's ID. Backend operations were changed to include the user's ID and user can only see their own entries. I added a settings page, where user can logout or delete their account.

In the beginning of the app, I didn't pay much attention to the styling of the app and kept it as it happened to be. At this stage I did some small styling changes but nothing big. I did not follow the UI design I made in the beginning as I did not want to use a lot of time for changing up the style. Seems like if you don't aim for the look in the beginning, it is harder to change it later.

I noticed that the forms were kind of annoying to use. I wanted to add a next button on the keyboard, so that the user can just press that to go to the next text input. I learned that there was a way to do that by assigning refs to the text inputs. However, I was using React Hook Form's dynamic field arrays. This means that the user can append or remove the fields. This caused some issues, and I did not figure out a way to do this quickly, so I gave up on the feature and moved on since I did not have much time left. In the future I want to add this feature to increase user experience. Surely there is a way to dynamically assign refs to the text inputs. Another way, which I find more enticing, is to remove the field arrays altogether and use a multiline text input and regex to separate the ingredient information. Using one text input for the ingredients could also be better for the user experience since tapping on a new text input is more disruptive than just writing it all on the same input.

I noticed that the keyboard covered the latter text inputs in the form. I tried using KeyboardAvoidingView and KeyboardAwareScrollView components but they did

not work. I came across information that in Android the keyboard should automatically avoid text inputs. I think the reason it did not work was because the form was inside a modal component and not on its own page. I decided to move the forms into their own pages, which ate up some time as I refactored the forms and navigation. It worked and the keyboard no longer covers text inputs.

At the end of the project, I cleaned up the code and did some small fixes here and there. I did also add a nutrition table to the database but was not able to implement this to the project. Unfortunately, the time ran out and I was not able to implement many of the features I would have liked to.

5 SHOWCASING THE APP

When launching the app for the first time the user is shown the login page as seen in Figure 12. The user can switch between login and sign-up forms which both ask for a username and a password. In Figure 13 the file-based routing of the app is shown. The initial page is actually “app/(authorized)/(tabs)/index.jsx” but the “_layout.jsx” inside “(authorized)” directory checks for a token before the routes are rendered. If no token exists, the user is redirected to “login/index.jsx” page. When the user logs in a token is created and saved in local memory. User can now access the rest of the app.

Figure 12. Login page of the app.

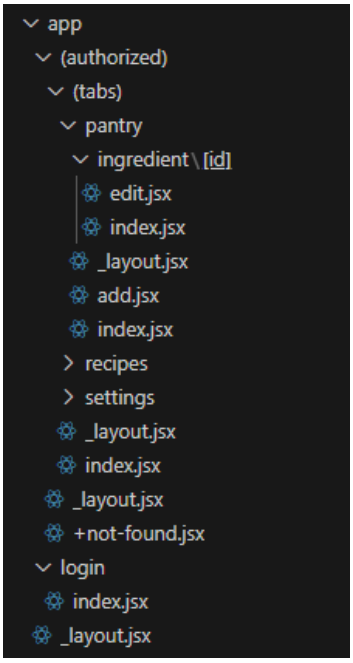

Figure 13. Screenshot of the routing in the app.

The home page is just a simple screen explaining how to use the app, visible in Figure 14. The routes are visible as tabs on the bottom of the screen, named Home, Pantry, Recipes and Settings.

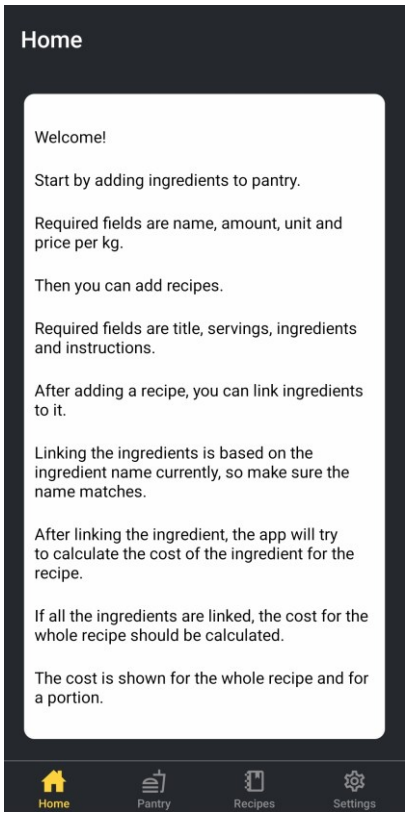

Figure 14. Home page of the app.

The pantry page contains a list of the added ingredients in Figure 15. For a new user the list would be empty. User can press the button “Add an ingredient” to navigate to a form or they can press one ingredient to navigate to that ingredient’s individual page, which uses dynamic navigation. Ingredients in the list display their names, amounts, units, cost per kg, and optionally an expiration date. In Figure 16 you can see the individual page of an ingredient. Currently it just lists the details of the ingredient. With buttons at the bottom the user can edit or delete the ingredient.

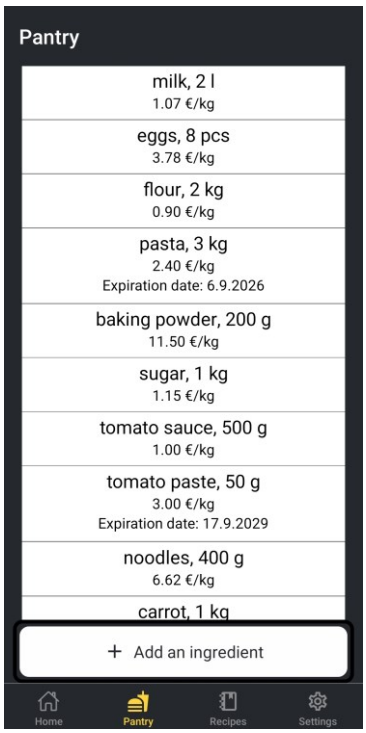

Figure 15. The pantry page of the app.

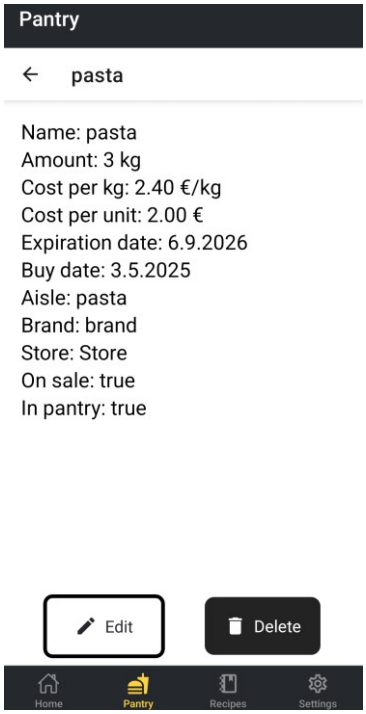

Figure 16. The individual ingredient page.

The add form and the edit form for an ingredient are in Figure 17 and Figure 18 respectively. The forms are similar but edit form's fields are filled with the ingredient's data. Mandatory fields are name, amount, unit and cost per kg. Optional fields are cost per package, expiration and buy date, aisle, brand, store, weight

per piece and checkboxes if the ingredient is in pantry and if the ingredient was on sale. Forms have validation and changing values in the edit form will warn the user of unsaved changes.

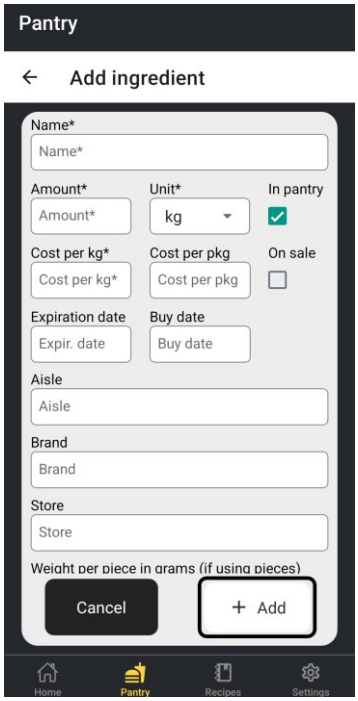

Figure 17. Form for adding an ingredient.

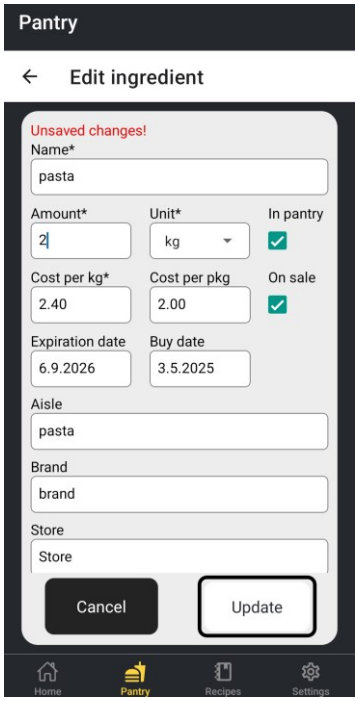

Figure 18. Form for editing an ingredient.

Next up is the recipes page in Figure 19. It's similar to the pantry page. Recipes are shown in a list. Each recipe displays a title, the amount of servings and an image. If no image was provided, a placeholder image is used. The rating, a heart for a favorite and the cost of the recipe is also shown if they're available. The cost shows the total cost of the recipe and the portion cost inside parentheses. User can press the add button to add a recipe or check out a recipe by tapping one.

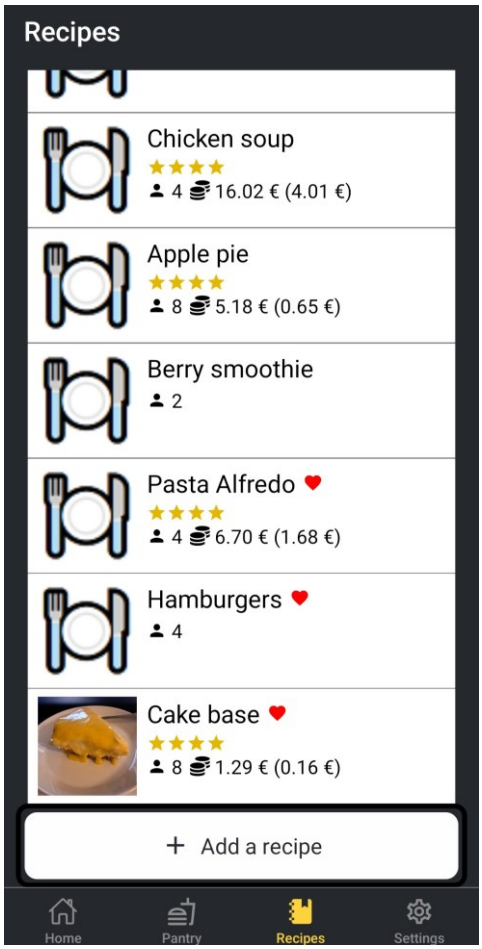

Figure 19. The recipes page of the app.

In Figure 20 the individual recipe page is shown. It displays more details and styling is used to separate parts of the recipe. Link buttons are next to the ingredients. In this case all the ingredients are linked and the costs of the ingredients and the recipe are available. The buttons can be tapped to link or unlink the ingredients. Edit and delete buttons are on the bottom.

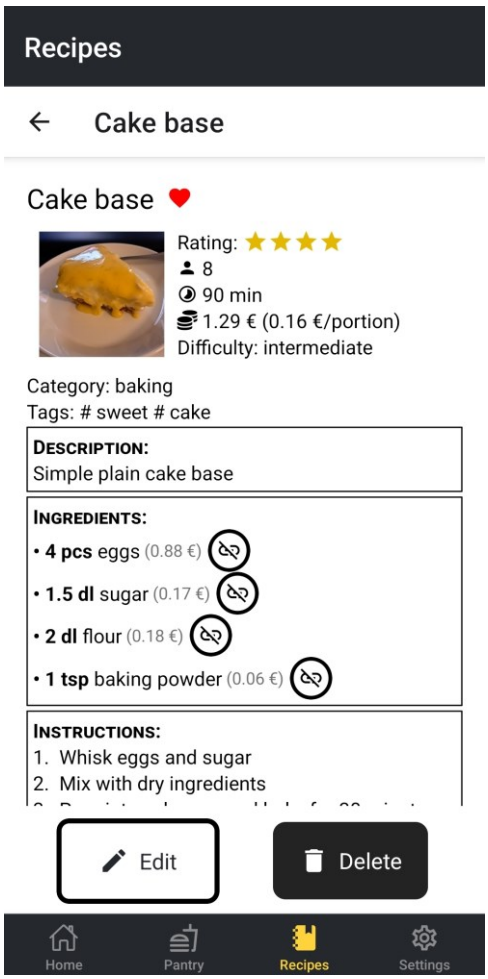

Figure 20. The individual recipe page.

The add form for the recipe is in Figure 21. It shows the mandatory fields, which are title, servings, ingredients and instructions, and the user can press the arrow button to expand the form and see the optional fields. Ingredients and instructions fields are field arrays. For ingredients there are separate fields for amount, unit and name. With the add button user can append more fields so they can add as many ingredients and instructions as they want. At least one ingredient and one instruction are required. If the user adds more fields, a remove button is added next to fields so the user can remove unneeded ones. Optional fields include description, category, tags, difficulty, minutes to make, notes, image, rating and a checkbox for setting the recipe as a favorite. The edit form is in Figure 22, which is similar to the add form, but filled with existing recipe details. User can update the recipe if needed.

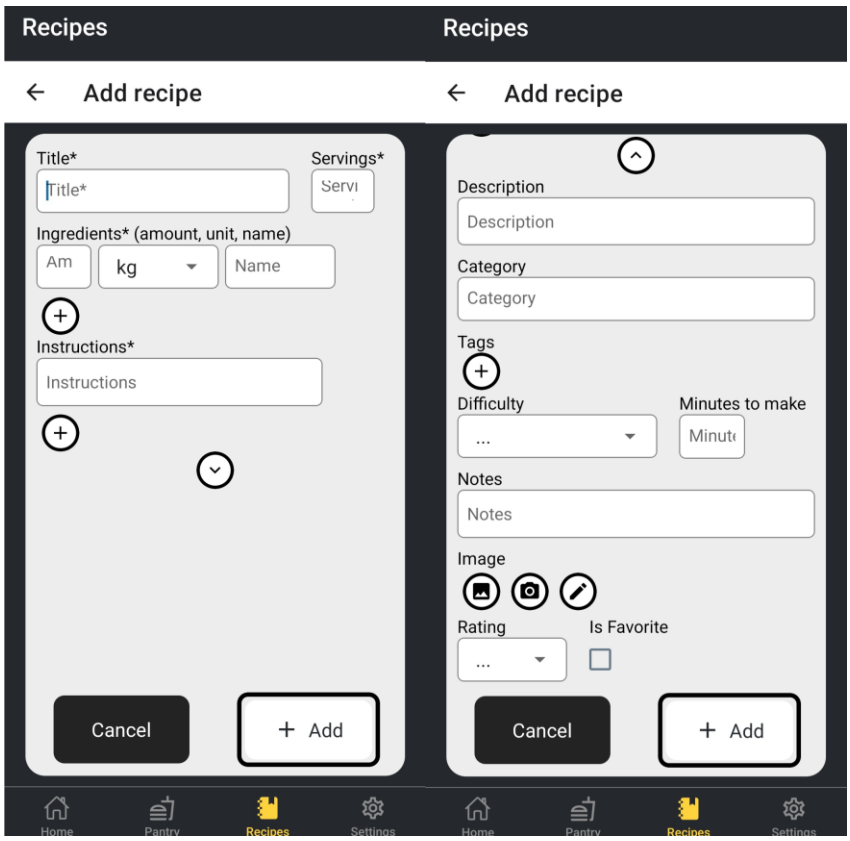

Figure 21. The add form for a recipe.

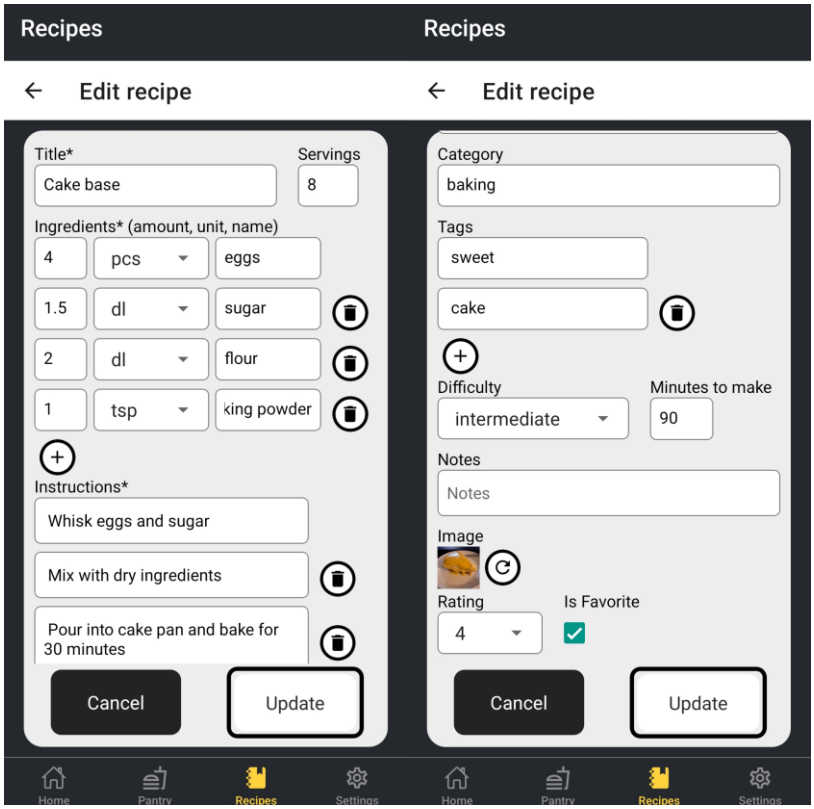

Figure 22. The edit form of a recipe.

The last page is the settings page. As seen in Figure 23 it is a simple page with only two buttons. The logout button allows the user to sign out of the app. Second button allows the user to delete their account. After pressing the button, the user has to confirm that they really want to delete their account. If they confirm, the user is signed out, the account is deleted and all entries connected to that account are also deleted.

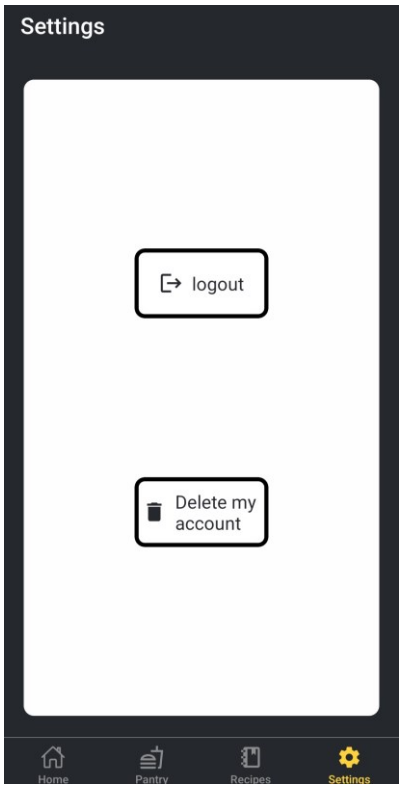

Figure 23. The settings page of the app.

6 DISCUSSION

Finishing the MVP version of the app was successful. CRUD operations for the ingredients and recipes work and ingredients can be linked to recipes. The cost of the recipe and a meal is displayed automatically when all the ingredients are linked in a recipe and updating the linked ingredients in the edit recipe form is reflected on the app. During the process I learned more about PostgreSQL and Expo. I had used Expo Go once before, but this time I focused learning more about Expo and I used a few Expo libraries in the project.

On the other hand, many nice-to-have features were not implemented. Most of the time went to enhancing and fixing the existing features, such as adding more columns to ingredients and recipes and ensuring that linking an ingredient and forms work properly. Work was also quite slow in the beginning as the imminent threat of a deadline was not there to motivate. I did create a schedule for the tasks, but I quickly fell behind it as implementing some features took longer as I thought and a few issues regularly popped up that needed fixing. I would have really liked to have a price history for the ingredients and a grocery list that would have added ingredients automatically to the pantry. I did create a nutrition table in the database but wasn't able to implement that in the project yet. In the design phase I was quite sure that I might not have time to implement these features, so I was prepared to just leave them out. I suppose it's a good idea to double or even quadruple the amount of time you think a task is going to take.

I faced multiple problems with the forms. I used React Hook Form library because I wanted to have the ability to dynamically append and remove fields for the ingredients and instructions for the recipe. It took me some time to make a datetime picker component to work with the form. I wanted to add better user experience by allowing the user to traverse the text inputs with keyboard's next button, but I wasn't able to quickly figure out a way to add refs to the dynamic fields. The keyboard was covering the text inputs. I spent some time trying to fix this problem and then I figured out that placing the forms inside modals was creating the issue. I had to move the forms to their own pages, and I had to refactor the forms and navigation. Lastly, I noticed a bug. If the user has added something to an optional

field but later tries to clear that out in the edit form, the validation throws an error. I used Yup validation, and I changed the validation to allow for optional values, but it did not work. After some reading, I came across a comment, where someone said that making the values optional did not work with React Hook Form. Since this was at the end of the project, I didn't have time to fix it. These problems made me reconsider using React Hook Forms, so in the future I might switch to another form library.

I have quite many ideas for future suggestions and features for the application. I think currently the app is quite plain and a lot of the optional fields for ingredients and recipes don't really do much. The existing features can be made better. I would like to change the FlatList components to SectionLists and add sorting, for example sort ingredients based on alphabetical order, by expiration date or by last updated. The cost calculation still doesn't account for density so it should be changed. Alternatively, the Spoonacular API could be used for calculations. The nutrition info should be added to the app. Recipes could also have an option to calculate the nutrition info from the ingredients.

There also should be new features for the app. The features I listed in the design phase in nice-to-haves and wish list would be preferable. A price history table would save the prices of the ingredients and display the prices on a graph on the ingredient view. A grocery list could be used to add ingredients to the pantry if user marks them as bought. Recipes could allow user to add selected ingredients to the grocery list. A daily view could show the meals eaten and show how much money was used on food during the day. Calories and nutrition information could also be shown for the daily view. If user marks a recipe as done, it could automatically remove the amount of the ingredients from the pantry if desired. A meal plan section could allow user to add recipes to it so the user can plan weekly meals. I think one big feature would be saving recipes from the internet, which is possible in the Paprika 3 app. Then the user wouldn't have to type everything manually. Currently the app is quite strict with ingredient names and other data types, so it would need some flexibility.

Other suggestions would be adding a proper testing to the app, so that we can quickly see if parts of the app work correctly when adding new features and

changing existing ones. Proper testing and finding bugs are important. The ability to import and export data would be good for users, especially if they want to bring recipes from another app. A price unit conversion would allow the app to be used anywhere in the world. Currently the app assumes the user uses euros. User experience should be enhanced, and the styling could be more pleasing. The last suggestion is building and distributing the app.

Overall, I'm satisfied with the results given the short time frame. I wish I was quicker or more efficient in the project since I really wanted the app to have more features, but I can keep working on it after completing the thesis. I'm kind of making the app for myself since *I* am the one who wants to use it but I think other people could be interested in it too, so the app has potential. The code is available on GitHub (https://github.com/jkellok/MealBudget).

REFERENCES

Amazon. N.d.-a. What is Mobile Application Development? Read on 27.5.2025. https://aws.amazon.com/mobile/mobile-application-development/

Amazon. N.d.-b. What's the Difference Between MongoDB and PostgreSQL? Read on 28.5.2025. https://aws.amazon.com/compare/the-difference-between-mongodb-and-postgresql/

Budziński, M. N.d. What Is React Native? Complex Guide for 2024. Netguru. Read on 27.5.2025. https://www.netguru.com/glossary/react-native

Codeacademy. N.d. Express.js Explained: What it is and How to Use it in Your JavaScript Project. Read on 28.5.2025. https://www.codecademy.com/article/what-is-express-js

Cohen, C. & Matley, H. Ingredient Weight Chart: Common Food Conversions. Updated on 28.2.2025. Instacart. Read on 29.5.2025. https://www.instacart.com/company/ideas/ingredient-weight-chart/

Expo. 2024. Core concepts. Updated on 1.7.2024. Read on 28.5.2025. https://docs.expo.dev/core-concepts/

Expo. N.d.-a. Core concepts of file-based routing. Read on 28.5.2025. https://docs.expo.dev/router/basics/core-concepts/

Expo. N.d.-b. Expo Router notation. Read on 28.5.2025. https://docs.expo.dev/router/basics/notation/

Expo. N.d.-c. Introduction to Expo Router. Read on 28.5.2025. https://docs.expo.dev/router/introduction/

Hasura. N.d. Relationships. Read on 29.5.2025. https://hasura.io/learn/database/postgresql/core-concepts/6-postgresql-relationships/

IBM. N.d. What is mobile application development? Read on 27.5.2025. https://www.ibm.com/think/topics/mobile-application-development

Lucidchart. N.d. What is an Entity Relationship Diagram (ERD)? Read on 29.5.2025. https://www.lucidchart.com/pages/er-diagrams

Microsoft. N.d. What is mobile application development? Read on 27.5.2025. https://azure.microsoft.com/en-in/resources/cloud-computing-dictionary/what-is-mobile-app-development/

Moedano, B. 2024. Expo Go vs Development Builds: Which should you use? Released on 12.9.2024. Expo. Read on 28.5.2025. https://expo.dev/blog/expo-go-vs-development-builds

MongoDB. N.d. Types of Databases. Read on 28.5.2025. https://www.mongodb.com/resources/basics/databases/types

Panchasara, M. 2024. React vs React Native: Which One to Choose and Why? Updated on 30.8.2024. Radixweb. Read on 27.5.2025. https://radixweb.com/blog/react-vs-react-native

Paprika App. N.d. Paprika User Guide for Android. Read on 24.2.2025. https://www.paprikaapp.com/help/android/

Parody, L. 2025. How Node.js Works: A Comprehensive Guide in 2025. Released on 28.1.2025. NodeSource. Read on 28.5.2025. https://nodesource.com/blog/how-nodejs-works

React. N.d.-a. Built-in React Hooks. Read on 28.5.2025. https://react.dev/reference/react/hooks

React. N.d.-b. Introducing Hooks. Read on 28.5.2025. https://legacy.reactjs.org/docs/hooks-intro.html

React. N.d.-c. Passing Props to a Component. Read on 27.5.2025. https://react.dev/learn/passing-props-to-a-component

React. N.d.-d. State: A Component's Memory. Read on 27.5.2025. https://react.dev/learn/state-a-components-memory

React. N.d.-e. Writing Markup with JSX. Read on 27.5.2025. https://react.dev/learn/writing-markup-with-jsx

React Native. N.d. React Native. Read on 27.5.2025. https://reactnative.dev/

React Native. 2025a. Core Components and Native Components. Updated on 14.4.2025. Read on 27.5.2025. https://reactnative.dev/docs/intro-react-native-components

React Native. 2025b. React Fundamentals. Updated on 14.4.2025. Read on 27.5.2025. https://reactnative.dev/docs/intro-react

Spoonacular. N.d. Spoonacular API Main Page. Read on 29.5.2025. https://spoonacular.com/food-api