VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Duong Minh Chinh

A SMART TO-DO APPLICATION FOR PRODUCTIVITY AND ENTHUSIASTS WITH REACT NATIVE

Technology and Communication
2020

ACKNOWLEDGEMENT

Firstly, I would like to send my appreciation and many thanks to my teachers at Vaasa University of Applied Sciences, who inspired me on my journey of studying at VAMK. I appreciate the passion and the encouragement from Mr. Timo Kankaapää, my thesis supervisor, for being supportive and for his working opportunities for students. Timo did provide me many projects to improve my professional skills as a Software Engineer and I thank him for that.

I would like to give my special thanks to Mr. Seppo Mäkinen for being a great teacher throughout my academic years at VAMK. His earnest and care for students is highly appreciated, and it has affected my determination to be strict, careful and truthful on my career path.

Furthermore, I would like to thank the online community of developers including Stack Overflow, Github and Expo's forum of being helpful at my questions. Without the helps, this thesis might have not been accomplished in time.

I also would like to send my gratitude towards my teammates and especially my friend, Ha Nguyen, for providing ideas of the thesis, as well as supporting me along the way. Her dedicated management drove the project forward.

Finally, I would thank my parents for supporting me unconditionally on the path of completing my degree at VAMK. Their encouragement and love provided me confidence and belief in my chosen career.

Helsinki, 08.02.2020

Duong Minh Chinh

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Information Technology

ABSTRACT

Author	Duong Minh Chinh
Title	A Smart To-do Application for Productivity and Enthusiasts With React Native
Year	2020
Language	English
Pages	97
Name of Supervisor	Timo Kankaanpää

The thesis was done based on the urge of creating an innovative to-do mobile application with the goal of helping people to reach their best productivity performance. There are several major to-do mobile applications serving different kinds of planning tasks in the market, but none of them simplifies the methods (the way of planning) or helps users stay motivated enough. Quint, the name of the mobile application that this thesis worked on, defines all the matters and unifies useful features from each major application, as well as provides new ones in order to help users stay "hydrated".

Quint is a cross-platform mobile application. To be more concise, Quint includes two main parts to form a completely useable application. We separate them as Front-end and Back-end parts. The Front-end part was mainly built with React Native and Expo. The Back-end part was built with Nodejs (Express) and Firebase. This thesis focuses on the implementations of Quint in both Front-end and Back-end sides at the time Quint reached its beta testing phase.

Quint has reached it first milestone, which is to be released for beta testing in iOS (TestFlight). At the time writing the thesis, Quint has already received over 50 feedbacks from public testers with 100+ download times despite the shortness and the lack of advertisements.

Keywords Quint, React Native, Nodejs, Express, Firebase

CONTENTS

LIST OF FIGURES AND TABLES

LIST OF ABBREVIATIONS

URL	Uniform Resource Locator
CRUD	Create, Remove, Update, Delete
UI	User Interface
API	Application Programming Interface
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
SPA	Single-page Application
SQL	Structured Query Language
NoSQL	Not Only Structured Query Language

1 INTRODUCTION

1.1 Background and Motivations

Technology is growing in a fast pace nowadays. It facilitates a large part of people's burdens, as well as improves people's lives. Besides, with the growth of technology, there are more and more tech start-ups, which are founded by young people. These start-ups may have a short life, or they may last long and become big companies. Interestingly, the results depend on humans since although technology is great, it must be used efficiently and smartly by people. We all know that to run a start-up is not an easy job. It takes time and effort of many people to live up to their dreams. Thus, some may feel discouraged along the way. Quint is initiated with the idea of helping people work more productively by providing simplified planning methods and constructing self-reward systems so people will stay motivated doing what they plan to do.

Quint provides an importance priority matrix for managing tasks. The matrix consists of 4 levels of importance, which helps users define the ones that should be prioritized and be done first. In addition, Quint lets users decide their own rewards. Because there are no ways to identify proper rewards for all users, it is better to have users creating their own ones. When using Quint, users can set up points for each task so they can get those points when completing tasks. There is a shopping feature in Quint, where users define their wanted rewards and they can purchase those rewards with earned points. Quint also provides a purchase history so that users can easily trace bought items with 3 identifiers: what, when and how many. The idea of self-reward system came from daily needs of people. People momentarily purchase an item just because they have the urge of buying it, without a hesitance of re-thinking and considering its costs and benefits. In other words, people always want to treat themselves with rewards but normally those treats have bigger values than what people think. Quint makes people work for what they want as treats.

1.2 Objectives

The main goal of the thesis is to reach the first stage of Quint, which is beta testing for iOS via TestFlight app. At this stage, Quint is usable and can be downloaded by downloading via TestFlight app provided by Apple. When using TestFlight, simply enter Quint's TestFlight url to download Quint. After finishing downloading, Quint is available for usage.

There are two memberships, one is a Free plan and other is a Premium plan. Free plan has limited features while Premium plan has full access to all current and incoming features. At the first stage, to use Quint at its finest, users will be prompted a notification window that advices to register an account to test as a Premium member.

Although Quint is out for beta testing in iOS, it is still possible to develop and publish in Android. All it takes just a few changes due to different requirements in environments between iOS and Android. In the past, to develop a mobile application is expensive and cumbersome as there are 2 different developing teams working on 2 major mobile environments: Android (Java) and iOS (Swift, Objective C). Nowadays, the limitations are heavily reduced because of the appearances of cross-platform languages such as React Native and Flutter. With cross-platform technologies, it is possible to make "develop once, ship everywhere" mobile applications.

1.3 Scope and Limitations

The thesis contains a scope of the first stage version of Quint, which meets the major requirements:

- Users should be able to perform CRUD operations on tasks.
- Users should be able to register a new account and login/logout with intents or forces by the application.
- Users should be able to make an upgrade from Free plan to Premium plan through a subscription system.

- Users should be able to see their completion analytics in graphs.
- Users should be able to make their own rewards, to gain points from completing tasks and to purchase those rewards with earned points.
- Users should be able to see their purchase histories.
- The application should do double-check for receipts whenever users make an upgrade from Free plan to Premium plan.
- The application should check whether users have already logged in and whether their accounts are Free or Premium to determine the next steps when being opened (from background/inactive to foreground/active).
- The application should be able to store task data on disk so that whenever we push a new update on the merchant (iOS or Android), the application will not lose its old data. In other words, it should be able to preserve data over updates.

At the time writing the thesis, due to time constraints tests could not be carried out. All the testing results in this thesis were manual testing or laboratory testing. To test the app thoroughly, one should make attempts of using Jest, a testing framework made for JavaScript, especially for React and React Native, to unit-testing the application. Thus, we can use Jest to test in the Back-end side as well, since we use Nodejs (JavaScript) for controlling our server.

1.4 The Structure of the Thesis

In Chapter 2, we will discuss the relevant technologies and what knowledge is required to for specific features.

2 TECHNICAL BACKGROUND

2.1 The Front-end side

2.1.1 React and React Native

Firstly, to be able to understand about React Native more thoroughly, one should have enough knowledge about React.

React has been one of the most popular frameworks over recent years. It has been mentioned everywhere in the job markets for developers. Thus, developers are using it intensively for scaling and rapid developments. So, what is React? In short, React is a JavaScript framework for building UIs. Developers use React for building a web-based SPA. React is mostly known for its effectiveness, optimizations and controllable props, states. When using React, developers code for components. Those components can be re-rendered by updating their properties (props) or states. Therefore, it is easier for developers to control which components should update and when they should perform a re-render. This pattern leads to better performance and effortless debugging.

React Native powers mobile apps with its React-like syntax and easy-to-use APIs. Traditionally, building native mobile applications mean one should use either Xcode (Swift, Objective C) or Android Studio (Java). It makes building a mobile app time-consuming, costly, and more error-prone. With React Native, developers can use JavaScript to build native mobile apps and have them cross-platform seamlessly. This way of developing helps decreasing the cost, the time and the effort to build 2 separate platform applications.

Although the feature of "develop once, ship everywhere" is great, React Native still has drawbacks. Android and iOS are two completely different mobile platforms. Thus, React Native may have one API supported by iOS but not by Android and vice versa. When this kind of inconvenience happens, it is likely that there are external libraries fixing the issue. External libraries are open-source libraries, which are developed and maintained by developers and other developers can use them freely, or even contribute to the source code (for example: fixing a feature, pull some new refactor codes, etc.). However, external libraries are not always trustworthy to use since they can be abandoned by the authors so

the versions would not be compatible with the current version of React Native. Developers should take it into consideration to use external libraries since they have the responsibilities to persist the compatibilities of the libraries contained in their apps. Another drawback is due to using JavaScript, React Native may not be ideal for building heavily calculating applications compared to Java or Swift.

React Native is widely used todays. Here are some of popular apps that are powered by React Native:

Figure 1 Apps that use React Native

2.1.2 Expo

Expo is a framework and a platform built for React Native applications. It has a set of tools and services that enables developers to quickly develop, build, deploy and publish on iOS, Android and web apps.

Developers can choose whether to use or not to use Expo. This depends on the application requirements. There are two major workflows in Expo.

The first one is managed workflow. When creating a new React Native with Expo, developers will be recommended to use this workflow. The workflow eases the publishing

process for developers, which Expo will take care of automatically for both Android and iOS. Of course, to be able to publish an app to iOS, one should be publishing it via a MacOS system. Furthermore, using the managed workflow provides some useful APIs/components for developers to use. For example, there is one component called Splash Screen in a mobile app, which developers use for displaying the icon of the application while initializing the app with required data. This Splash Screen component cannot be easily achieved when using only React Native or native tools such as Android Studio or Xcode.

The other workflow is ejected workflow. This workflow is closer to apps created by React Native Command Line Interface, but it does provide useful components such as In App Purchases component, which I used in the thesis for making subscriptions. Another feature of Expo is that it is a platform, which gives developers a fast way to develop applications. While using React Native, ones must be using Android Studio or Xcode to perform simulations on real devices or emulations on virtual devices. Expo allows using only Node or any Expo integrated command line tools to do simulations that avoids using Xcode and Android Studio (Even though one can still do simulations and emulations on Xcode and Android Studio as normal with Expo).

The thesis will be discussing ejected workflow as Quint involves In App Purchases component from Expo.

2.1.3 Redux

The reason I picked up Redux and Redux Persist is that Redux has a similar way of handling data as state compared to React. To be frank, Redux is not a library customized for React. They accidentally have interchangeable ways of controlling, manipulating and processing data flows.

Now we know that React Native uses React as its core for syntax and APIs. Since React treats every UI element as a component and control their states to be able to update them, Redux can join the party and empower the process. Without Redux, components created

by React can only manage their own states, which means it will cause a big load of headache to easily update a component whenever we want. We shall examine the following diagram:

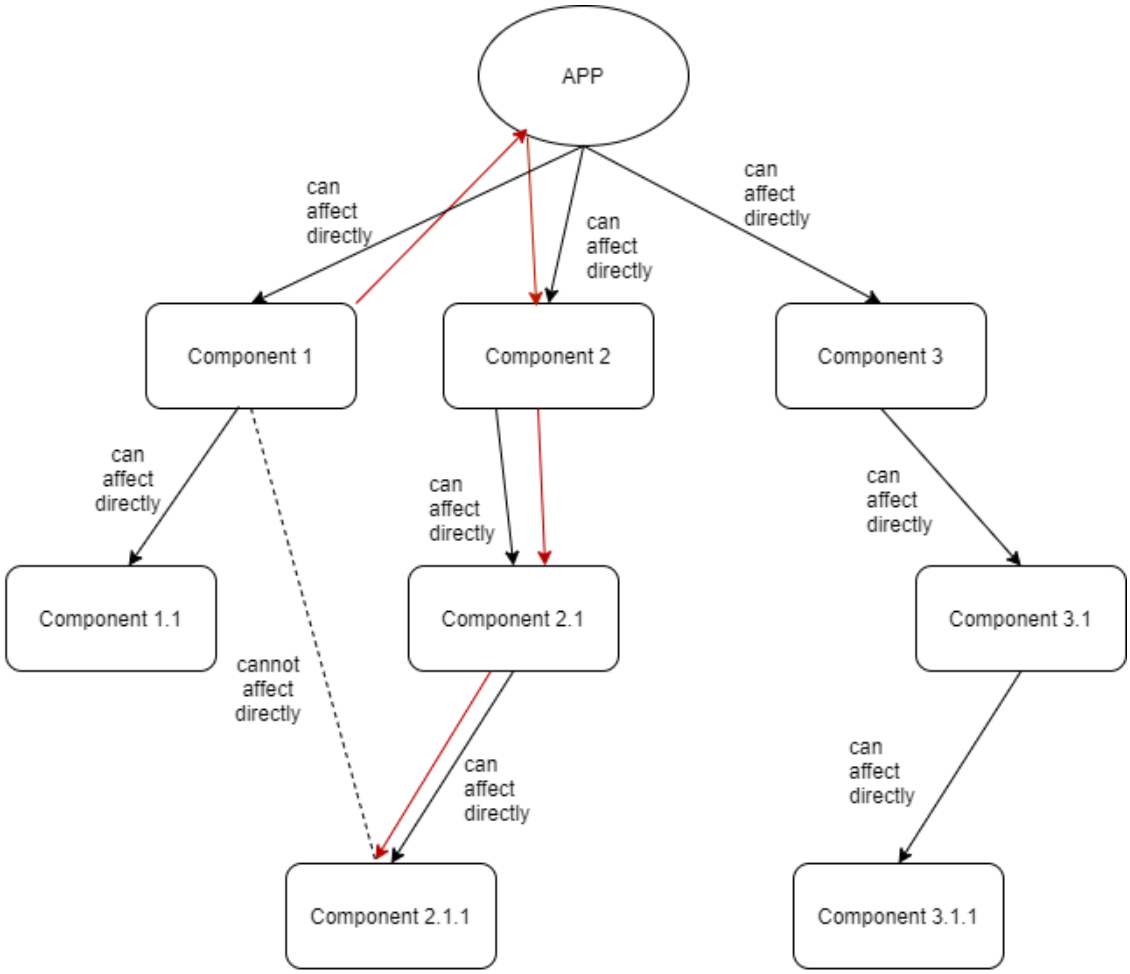


Figure 2 React without Redux updating indirect components

As the figure shows the relationship between components (parent-children relationship) in React, it is difficult for **Component 1** to update **Component 2.1.1**, which does not belong to **Component 1.** To be able to update **Component 2.1.1**, **Component 1** should make indirect updates via **APP** to **Component 2** to **Component 2.1** and to **Component**

2.1.1 (Following the red arrow's path)**.** Nowadays, React offers **useContext** for functional components (alternative for traditional class components) to easily achieve the above functionality.

Now let's take a look at the following figure with Redux integrated:

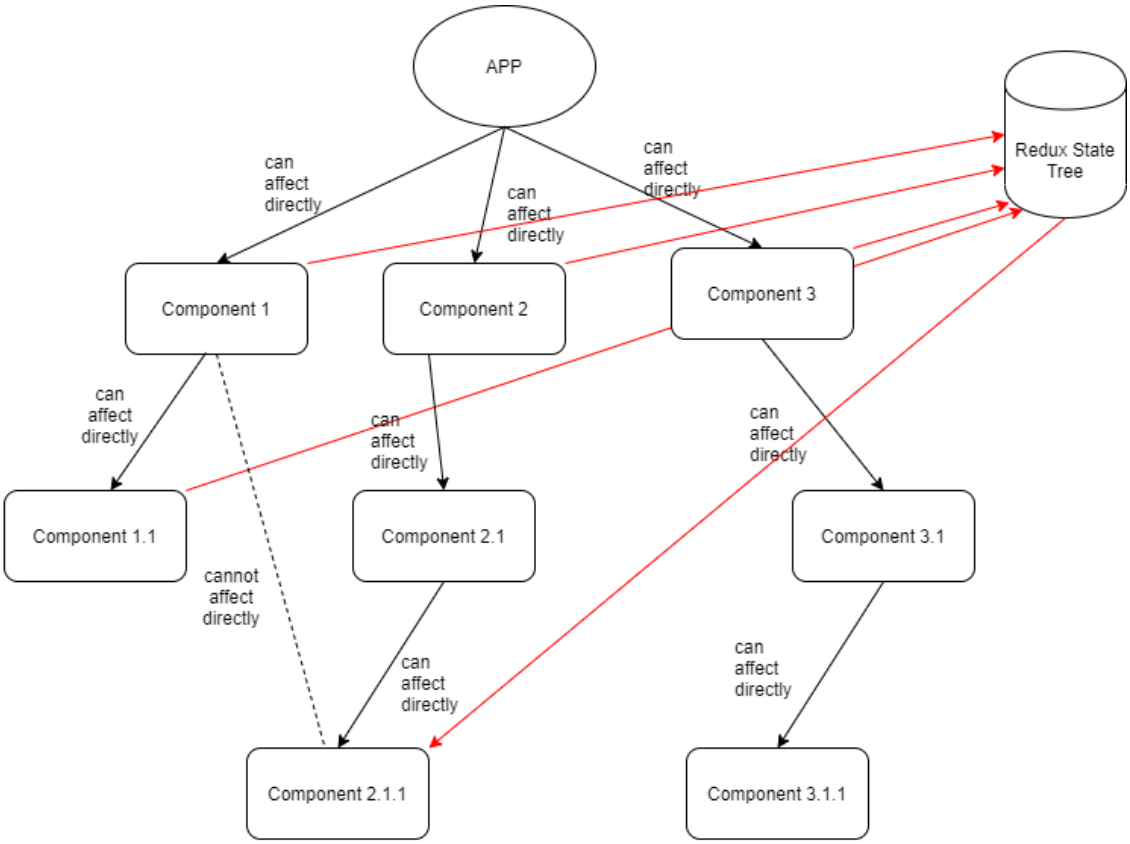


Figure 3 Redux with Redux updating indirect components

The red arrows indicate a more implementable way to handle updating indirect components. **Component 2.1.1** does not need to be updated by complex callings via **APP** or **Component 2** or **Component 2.1** anymore. Instead, **Redux State Tree** plays the role as a "global state" for the whole application and in case we want to update any components at anywhere in the app's structure, we can simply have those components listen to **Redux State Tree** so that they will be updated whenever **Redux State Tree** changes.

2.1.4 Redux Persist

When using Redux, it is common to mention about a Redux store. A redux store holds the whole state tree of the application. It means **Redux State Tree** in **Figure 1** and **Figure 2** is contained by a Redux store. In the application that the thesis is working on, Redux store is where the app stores all the task data as well as relevant user data. Basically, Redux store acts as a storage for the Front-end side.

The problem with a normal Redux store is that its data gets removed every time we close the application. This is a terrible user experience and as a to-do mobile application, we prioritize the app to be device-first, which means Quint should be working offline without the Internet connection. To overcome the issue, I used a third-party library called Redux Persist. Redux Persist helps persisting the Redux store so that the saved data remains at the disk. It makes pushing a new version of Quint into the App Store easier since old data stays as the same as before, which results in having a better user experience as every created task, reward or action that users made are reserved.

2.1.5 React Navigation

When using a mobile application, one should notice different screens provided by the app. One application may have as many screens as it wants. Quint is not exceptional. To be able to achieve the ability of creating separate screens, I chose a third-party library called React Navigation.

React Navigation provides easy-to-use APIs built for iOS and Android. Users using Quint should feel the smooth animations and gestures when navigating between screens. Although React Navigation does come with a default animation setting, developers can still completely customize its APIs to fit their own intents.

The following figure describes the basic setup for screens of Quint:

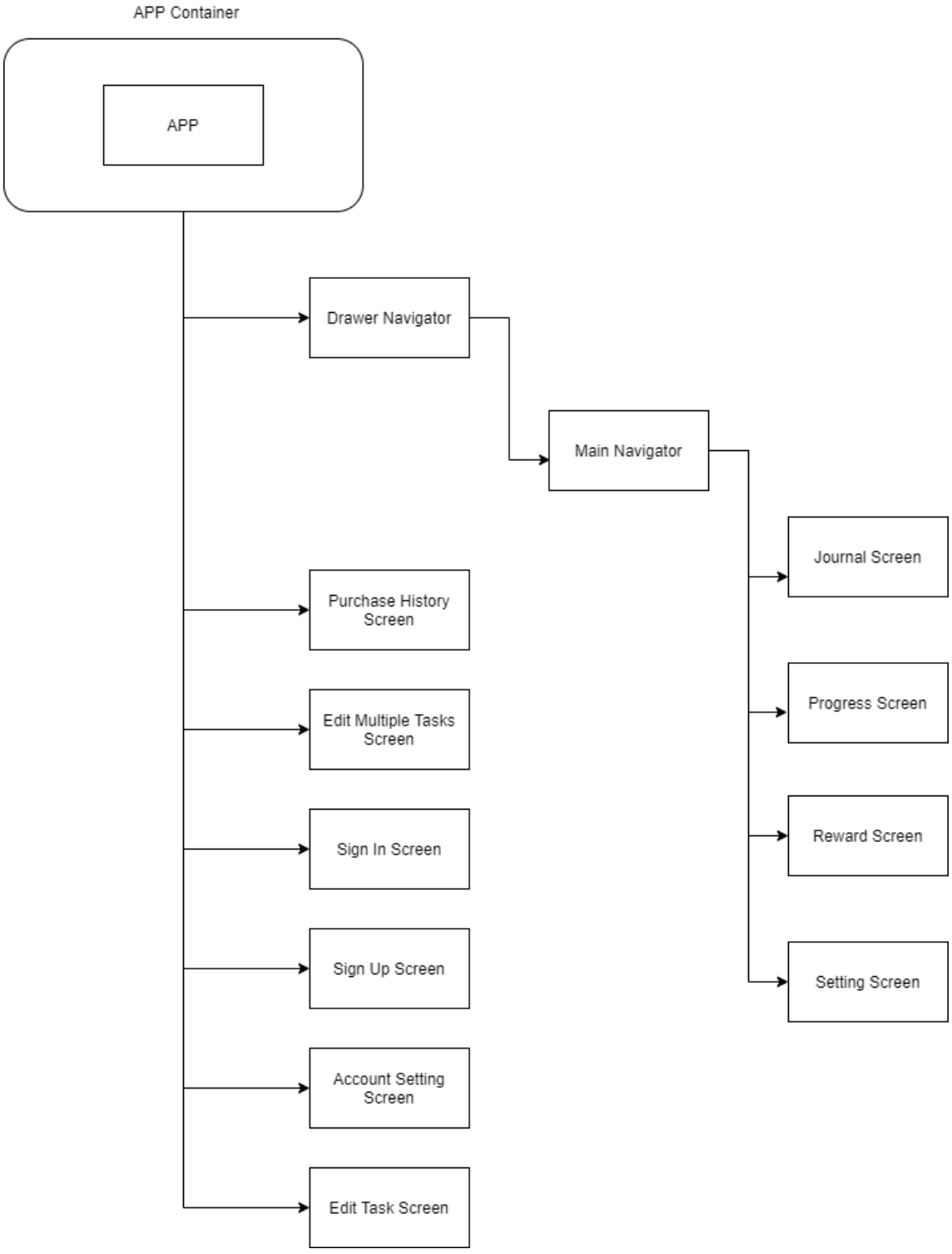


Figure 4 Quint's React Navigation basic setup

2.1.6 ImmutableJS

React Native is all about managing states. States play the most important role in the application. States define when and where the components should be updated. Thus, handling states in React Native is an essential for all developers to make their apps faster, more reliable and more error-proof.

State in React Native is an object. One should not mutate this object so that React Native knows whether state is changed. What is mutating? Mutating means changing an object's content but not its reference (or origin). Mutation is usually unwelcomed in React world since mutated objects cannot trigger React/React Native to re-render the intended components. Therefore, developers should always ensure their states remain immutable, which means every time they make an update of those states, they should return totally new ones. The reason behind this is because React/React Native can only do reference checks on the old and new states (Reference checks means checking their stored origin in the memory).

Why are we talking about states and mutations? Because to make objects immutable in JavaScript is no easy job. It may cause a tremendous drag-down in term of performance for applications. In addition, to immutate a nested object with several levels in React/React Native is such a boring, error-prone and time-consuming activity. Since there are scenarios that applications need their states to be totally immutable, which means every level of the states are immutable and developers have to do the processes.

I picked up ImmutableJS, a third-party library developed and maintained by Facebook, to deal with the situation. ImmutableJS provides easy-to-use APIs and shows to be drastically efficient in term of performance.

2.2 The Back-end side

2.2.1 Expressjs and Nodejs

I used Nodejs and Expressjs to build a server for Quint. Nowadays, there are many languages that can be used to develop a server such as PHP, Java, .Net, Python, and Golang. However, the reason I picked Nodejs is I have an adept knowledge in JavaScript and Nodejs offers a quick, scalable, and future-proof development.

Nodejs is a JavaScript runtime built on Chrome's V8 JavaScript engine, which is famously known for its efficiency for hosting JavaScript. On the other hand, Nodejs is an asynchronous event-driven JavaScript runtime and is designed to build scalable network applications.

When building a Nodejs server, one can simply use JavaScript and APIs that Nodejs provides or one can use extensional libraries built to support Nodejs further like Expressjs. Expressjs is a framework built especially for Nodejs. The reason it stands out is because of its minimal and flexible design, which provides a robust set of features for both web and mobile applications.

2.2.2 Firebase

Firebase, in short, is a SaaS cloud service provided by Google. Firebase is widely trusted and used by top applications at the time this thesis is being conducted. With Firebase's providing services, software applications can be speedily developed and thus, developers can focus more on the customer needs.

Firebase offers many robust functionalities to help developers build better apps. Some of the popular services provided by Firebase are:

- **Cloud Firestore:** is a real-time NoSQL database. We can use Cloud Firestore to store, retrieve and sync data between our apps with the database in the cloud. Cloud Firestore has proven its popularity by being used by thousands of apps due

to its scalability and performance. In this thesis we will be using Cloud Firestore instead of Realtime Database.

- **Cloud Storage:** with a similar concept to Cloud Firestore, Cloud Storage is where developers can store, retrieve and serve files. We can use Cloud Storage to store such as images, videos, and files under binary or blob files. Cloud Storage additionally provides the ability to download contents as well. We will be using the service in this thesis.
- **Realtime Database:** is a cloud-hosted NoSQL database that lets software applications store and sync data in realtime. Originally, Firebase Realtime Database is the first and original cloud-based database. Therefore, using Firebase Realtime Database will suffer some disadvantages compared from using Firebase Cloud Firestore as Firestore is a successor of Realtime Database. And since it is a successor, it will be improved and better than the origin.
- **Cloud Functions:** is a cloud-hosted service that allows building serverless software applications. Developers can run their backend code with Cloud Functions and so, the service will only execute the code whenever an event is triggered. Thus, the number of redundant networking requests will be reduced, which will avoid being error prone and server-overload. Because developers do not have to handle the backend (Google Cloud Functions will do it), there are so less work to do in term of maintaining the server. Another benefit of using Google Cloud Functions is that the backend code that we push will be securely kept and will be private from any outside intruders. This service is mentioned due to its popularity and, at this stage of the thesis, it will not be implemented.
- **Authentication:** provides an easy way to sign in the applications with any platform. The service supports end-to-end identity solution, email and password method, phone authentication, and Google, Twitter, Facebook, GitHub login, etc. At the time writing the thesis, there is a drawback of using Google Authentication, is that when using email and password method, we do not have a proper way to handle email verifications whenever users register new accounts. In order to surpass this issue, I combined using Google Authentication with SendGrid to accomplish the goal.

2.2.3 SendGrid

SendGrid is a popular email service nowadays. SendGrid is capable to provide many impactful things in term of marketing. In this thesis, SendGrid is used as an email provider to send email verifications to newly registered emails. In combined with Firebase Authentication, it allows us to form a complete process of registering new accounts for Quint.

3 APPLICATION DESCRIPTION

This section includes the general description of the thesis, as well as the quality requirements that the thesis holds. In addition, the section discusses different diagrams for different functionalities emerged in the thesis.

3.1 General Description

The main goal of the thesis is to build a to-do mobile application that can work on both major platforms: iOS and Android. However, the application, Quint, is differentiated from other to-do mobile applications such as TickTick, Todoist, or Microsoft To Do since it collects "good" functionalities from major apps and gets rid of "bad" functionalities as well. Furthermore, Quint provides its own innovative ways to motivate users so they can be interactive and motivated to complete their defined tasks.

In this thesis, the project is divided into two major parts: Front-end and Back-end implementations. Hence, we dive into each part and within each part, we discuss different crucial functionalities.

3.2 Quality Requirements

For each section: Front-end and Back-end side, the requirements are different. Additionally, each section contains three type of quality requirements based on the priorities, which are must-have, should-have and nice-to-have. The must-have functionalities, or features, are the core of the project, and hence, the major must-have features will be investigated closely in this thesis. The should-have features are important features, but not vital. The thesis will also discuss some of the should-have features. Finally, the nice-to-have features are features that either have been already implemented or will be implemented in the future. These features can be categorized as Future Work.

3.2.1 The Front-end side

Reference	Description	Priority

F1	Perform CRUD operations on to-do tasks.	1
F2	Perform CRUD operations on categories, which contain separate to-do tasks. Update a category results a change in each task about the category data of that task contained in the category	1
F3	Users can edit a single task.	1
F4	Users can edit multiple tasks.	1
F5	Perform complete/uncomplete operations on tasks.	1
F6	Tasks are classified into three types: Day, Week and Month.	1
F7	Tasks are put into correspondingly dates when users navigate the horizontal calendar in Journal View.	1
F8	Tasks with different priorities provide different points when completing/uncompleting.	1
F9	Users can delete a task partly, meaning at a specific date, and delete a task completely, meaning at every set date.	1
F10	Users can view the points they earn in each day of a specific month of a specific year in Progress View.	1
F11	Users can view the total Day, Week, Month tasks completed due to the selected month of the selected year.	1
F12	Users can view the stacked bar chart of completed tasks in a specific month of a specific year in term of priority.	1

F13	Perform CRUD rewards in Reward View. The Create and Update operations are up to the users.	1
F14	Users can view the history of earned rewards.	1
F15	Users can gain points to purchase rewards as products in a normal shop by completing defined tasks. In contrast, un-completing tasks result the point balance to be withdrew.	1
F16	Users can receive notifications at the start of a day, a week and a month about how many tasks there are to complete.	2
F17	Users can login, logout by using email and password method.	2
F18	Users can sync data when already logging in as a member.	2
F19	There should be a subscription model for users, who want to upgrade their plans so that they will get engaged more into the app. The benefits are included in the model.	2
F20	Users can use Google and Facebook sign-ins.	3
F21	The app should appear in the phone's widget so that users can easily complete their tasks.	3
F22	There should be more interactive animations for a better user experience.	3
F23	Users can drag-and-drop tasks and categories in their lik-ings.	3
F24	The app should have an on-board tutorial for new users.	3
F25	Users can rate the app and send feedbacks.	3

3.2.2 The Back-end side

Reference	Description	Priority
B1	Handle email and password authentication method using Firebase Authentication.	1
B2	Update the user information with Firebase Firestore and Firebase Cloud Storage.	1
B3	Register the subscription receipts of each subscription that users make to prevent frauds.	1
B4	Identify the free plan and premium plan users in order to provide appropriate functionalities.	1
B5	Handle push notifications so that the server can send a notification to multiple users.	2
B6	Handle Google and Facebook sign-ins. The server should act accordingly to each type of provider sign-ins so that the app can store proper user information.	2
B7	Integrate with SendGrid's APIs to notify users about new updates of the app by emails.	3
B8	Apply Stripe instead of using iOS and Android subscription systems.	3

3.3 Use Case Diagrams

In the last section, the thesis concludes that there are two primary parts of the project, which are the Front-end side and the Back-end side. The last section provides the must-

have and should-have requirements from each side, and thus, in this section, we discuss more about the Use Case diagrams of each side.

3.3.1 The Front-end side

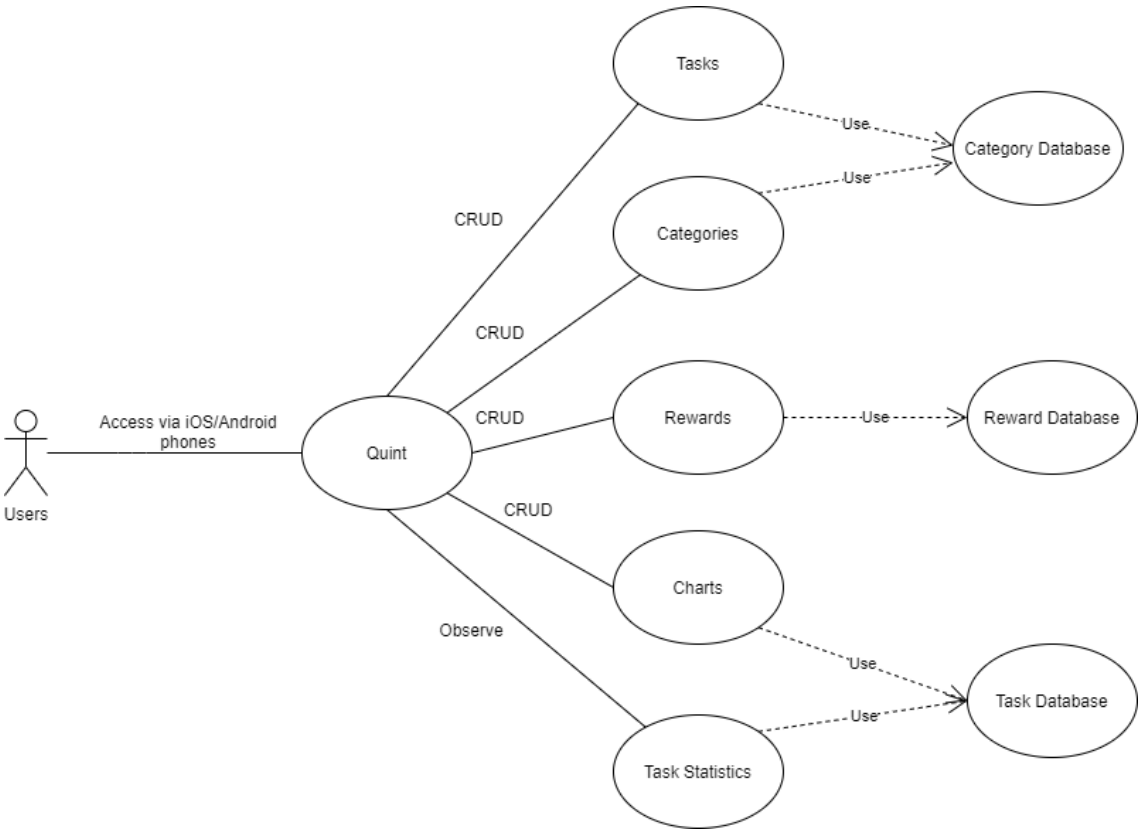


Figure 5 Using the App Use Case diagram

Figure 5 illustrates the relationship between a user and the application (Quint), the relationship between the application and key features and the relationship between key features and the data storages.

From the diagram, it shows that to be able to access to Quint, a user must be using a mobile phone or a tablet. At the time writing the thesis, Quint is supported in iOS phones (iPhones) and tablets (iPads). Furthermore, there are three databases used for three main tasks: relevant tasks about To-do Tasks, relevant tasks about Categories and relevant tasks about Rewards. In fact, the project contains more storages or databases such as storages for completed tasks to track the timestamp, points, task type in order to provide necessary

information for charts and statistics in Progress View. The User Case diagram above only displays the abstract of the project's implementation.

3.3.2 The Back-end side

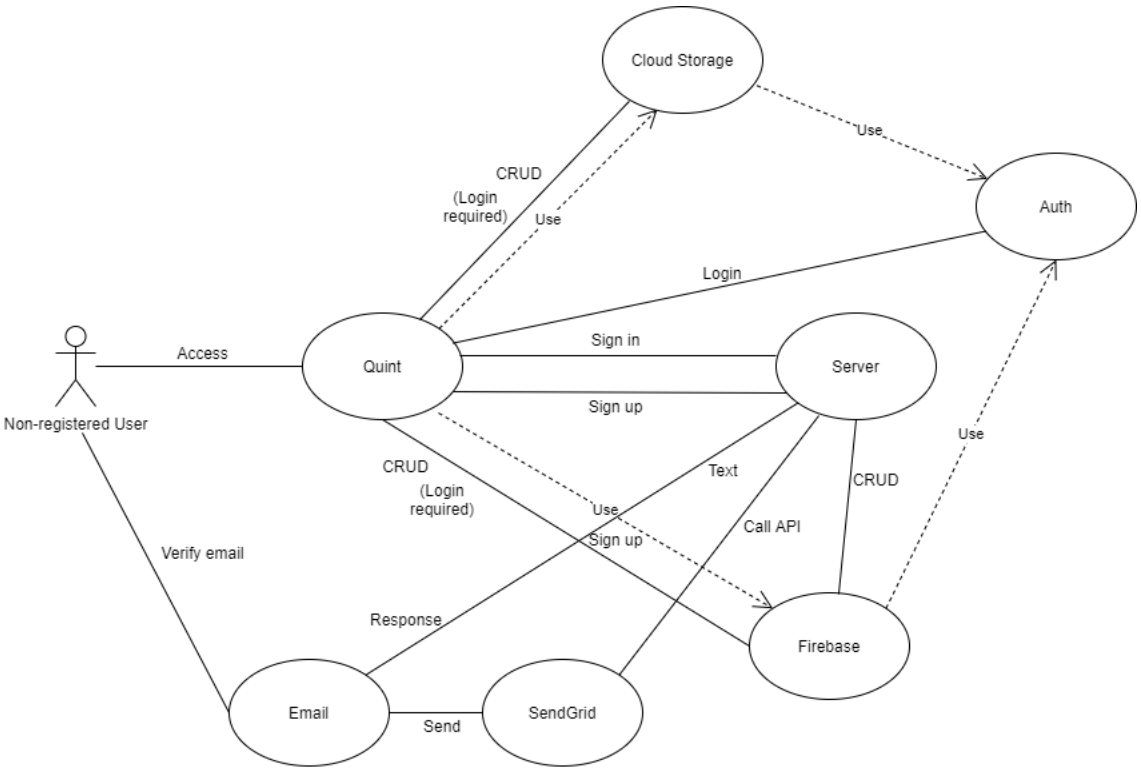


Figure 6 Managing Remote Requests User Case diagram

In the above figure, it is shown that the user can still access to Quint app via mobile phones and tablets. Note where there is no text of that connection displayed between the Actor and the Quint App, readers should consider the action is the same as in Figure 5.

A user can sign in into the app by using an existing account. Currently, the app supports email and password method and thus, in the near future, Quint will support Google and Facebook logins for a better user experience. After logging in, the user is allowed to perform changes involving directly the database and storage. Quint is using Firebase's client APIs to manipulate the interactions with Firestore and Cloud Storage, which benefits high performance and low-latency connections. The model does not need to be concerned with security issues since Firebase provides a safe method to protect the transactions between

the client and its services. In case the user does not have an existing account, a new account can be registered inside the app. When submitting the new account's information, the server will perform updates in Firebase with a unique register token, which is used to identify between accounts, and then call the SendGrid's APIs in order to send a email verification to the user. When the user verifies, the link in the email will trigger a request to the server and it completes the rest process of the account's registration. We will dive deeper into mentioned processes later.

3.4 Class Diagrams

Because of using JavaScript, a procedural programming language, which is not followed by Object Oriented Programming paradigm, the Class Diagrams shown in the thesis will be considered **Module** instead of **Class**.

Since the project's scope is big, there will be two types of Class (Module) Diagrams. The first type are diagrams to illustrate functionalities of the Front-end side and the second type are diagrams to illustrate functionalities of the Back-end side.

3.4.1 The Front-end side

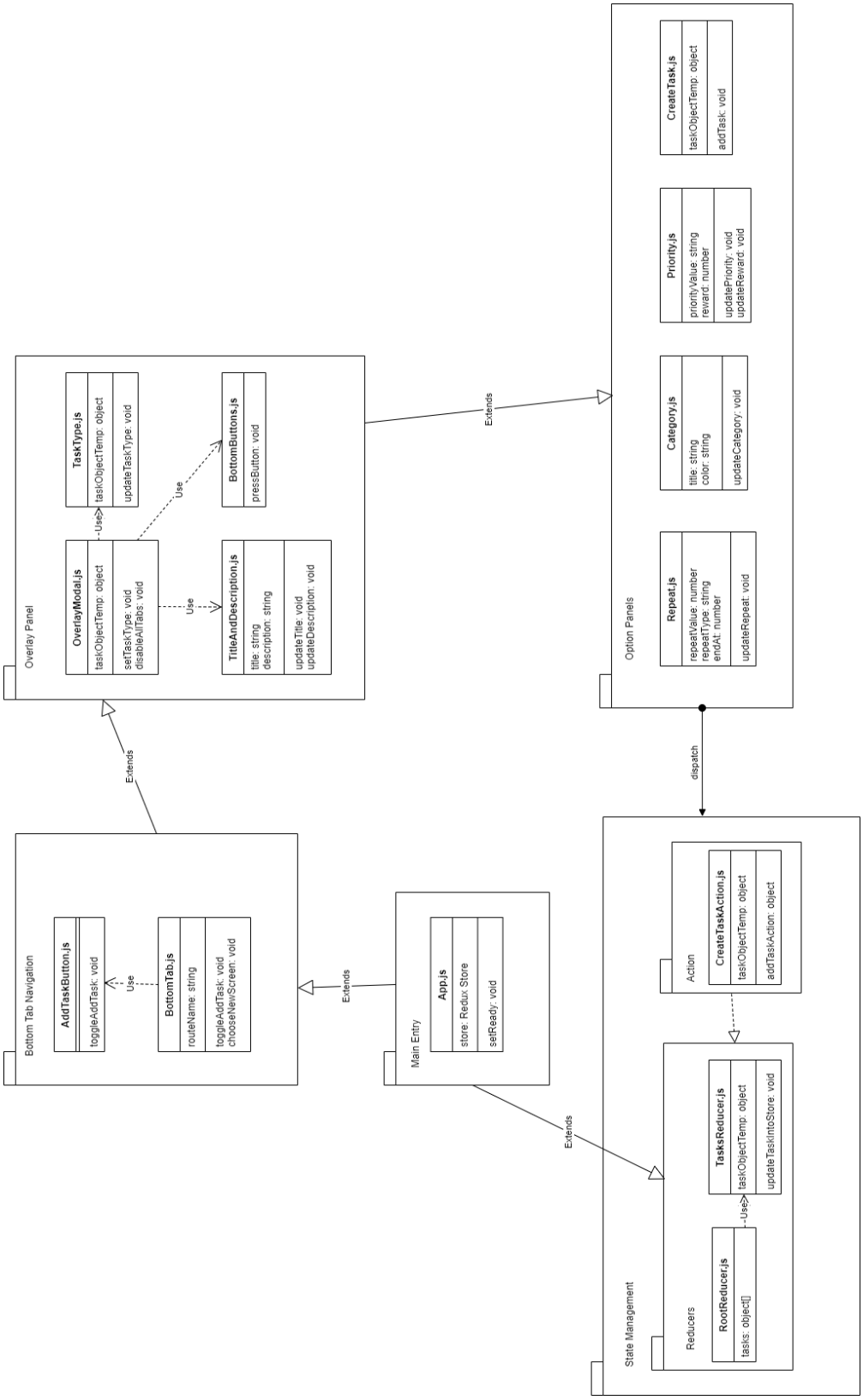


Figure 7 Class diagram for creating a new task

Figure 7 explains the relationships between classes (or modules) implemented in the project. **App.js** is the entry module, which serves the application and acts as the top-most parent module of the whole app. In this module, I handle the implementations of **Redux's** store so that the app can have a centralized state tree to use across components. Another important method is **setReady. setReady** method indicates the current state of the application, which decides what contents to firstly deliver to the users. For example, when users open the app, the app will then run some checks about user current subscriptions, or rehydrating (make available) the **Redux**'s store or even rescheduling some notifications properly.

BottomTab.js module is a child component of **App.js**. In fact, every other component is a child component of **App.js**. Each child component either renders views (visualizations) or contains exportable functions. In some cases, one may contain both aspects. **BottomTab.js** contains **AddTaskButton.js** and **OverlayModal.js**. Users can enable **OverlayModal.js,** which is a modal displaying input fields and option buttons for users to adjust their desire tasks, by pressing the button rendered in **AddTaskButton.js**.

When users open the modal, they can input tasks title and description, as well as make some changes of their repetitions, schedules, categories, priorities and rewards by pressing each defined button. All changes will be saved to a temporary object for the new task called **taskObjectTemp**. **taskObjectTemp** can be accessed globally since it is a state in the **Redux**'s store. When finishing tailoring desire tasks, users can press the confirm button provided by **CreateTask.js**. Then, **addTask** function in **CreateTask.js** module dispatches an action called **CreateTaskAction.js** to the **Redux**'s store.

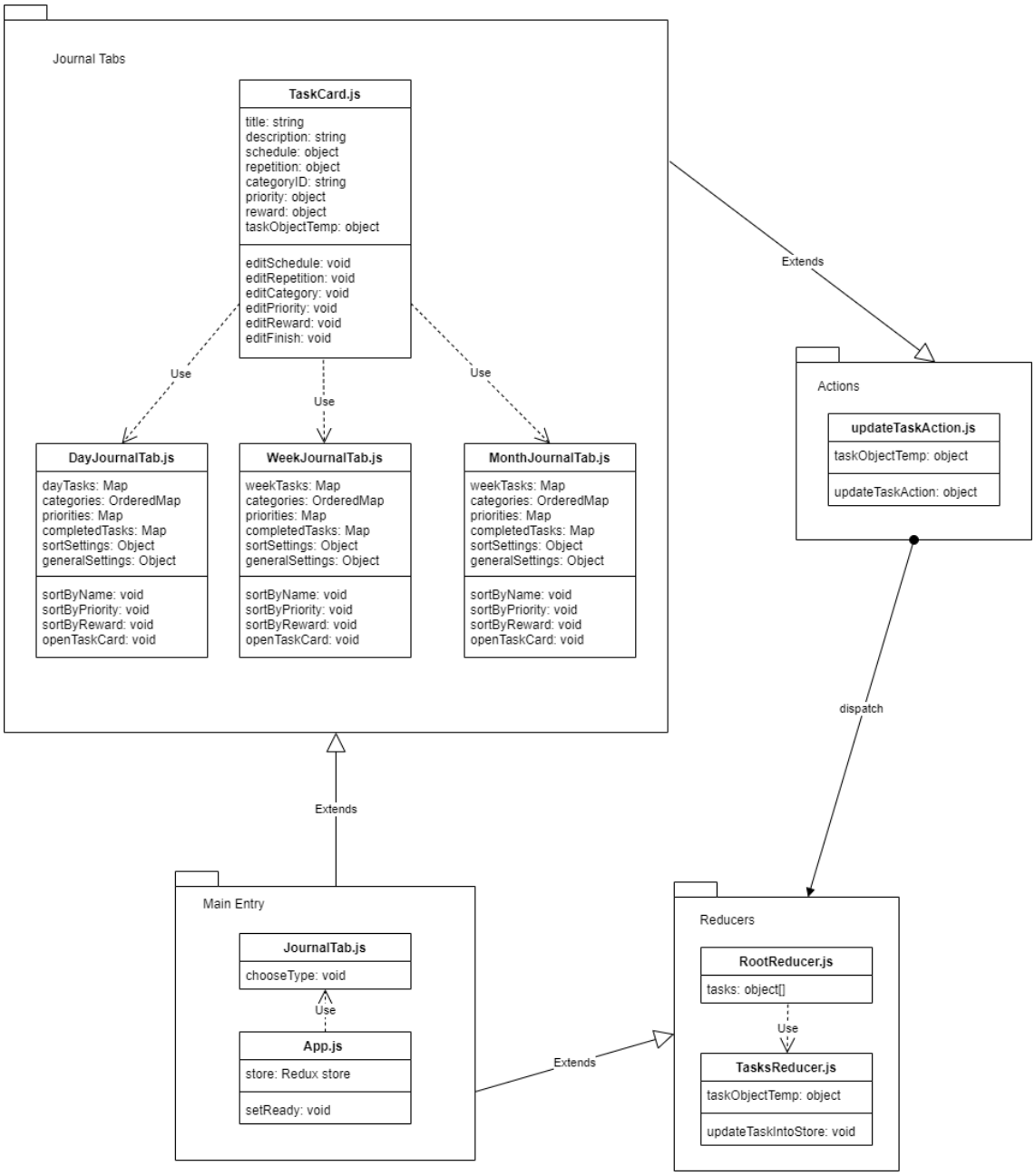


Figure 8 Class diagram for editing a task

Figure 8 illustrates the process in editing a task in Quint. As stated above, there is a **Redux's** store, which is the global state tree or the global store for every individual component to gain access to. Thus, each action emerged from the requirements are in fact, action to update the store.

Normally, **App.js** is the entry module of the application. **JournalTab.js** is a child module of **App.js**, which is a screen component. A screen component acts as a wrapper container for a group of relational features. In this case, **JournalTab.js** is the screen that contains components relevant to task managements, such as to edit, delete, create, sort or use horizontal calendars to observe specific tasks at specific dates. **JournalTab.js** has three child views: **DayJournalTab.js**, **WeekJournalTab.js**, **MonthJournalTab.js**. Each view is conditionally rendered based on the chosen of wanted viewing type of tasks of a user. When finishing choosing a view, users can then see related tasks existing in that view (users can choose a date to view date-related tasks as well). Clicking on the desire task will fire up the editing modal containing the information of that task, which allows users to modify wanted fields. Later, when finishing editing, the updated task is saved to the **Redux**'s store through a dispatch of action.

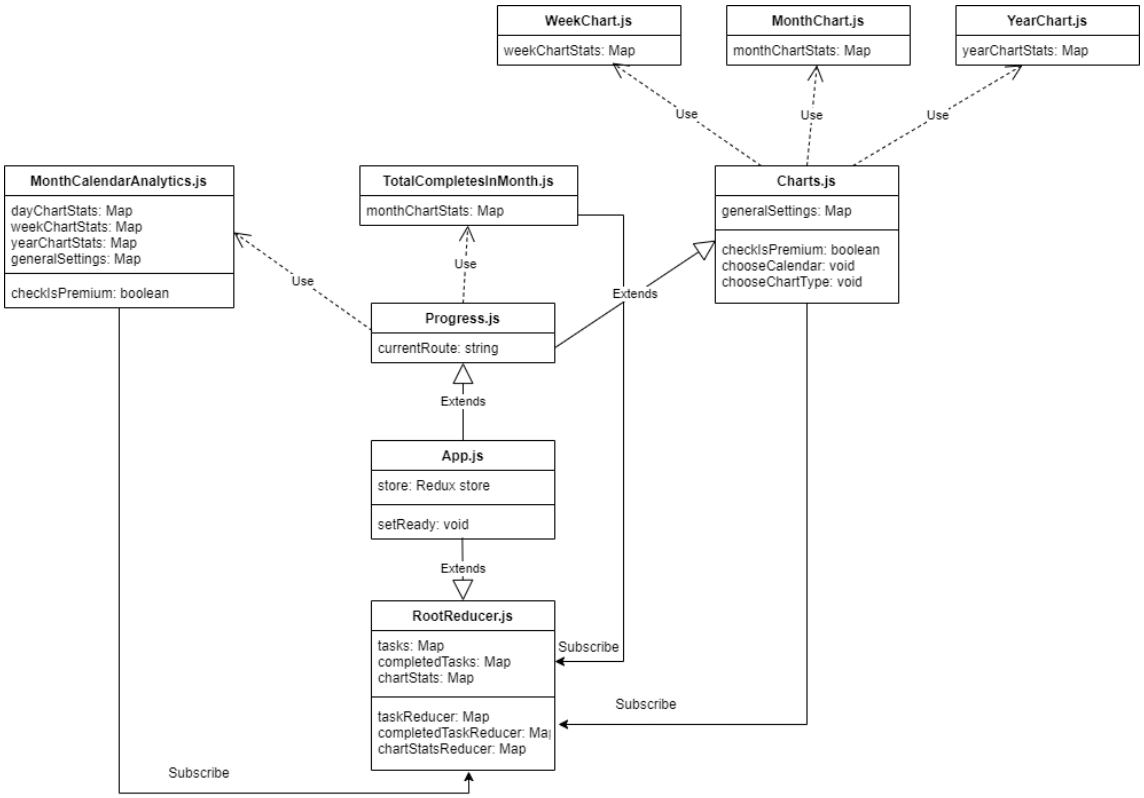


Figure 9 Class diagram for displaying statistics and charts

Progress.js module is the screen for monitoring the progress of user activities. By navigating to this screen, users can keep track of how productive they are and how performantly they have been completing their defined tasks. **Progress.js** consists of three components: **MonthCalendarAnalytics.js**, **TotalCompletesInMonth.js**, **Charts.js**. **MonthCalendarAnalytics.js** module displays a calendar, which users can manipulate to show different months. With each shown month, the app then displays according completed points in term of every day in that month, every week in that month and the points earned in total of that month. **TotalCompletesInMonth.js** module presents the total amounts of each type of completed tasks in the chosen month from the calendar of **MonthCalendarAnalytics.js** (Day, Week, Month). **Chart.js** has a method called **chooseChartType**, which allows users to choose what type of charts to display and a method called **chooseCalendar**, which will prompt a calendar accordingly to chosen type of chart when being invoked. Every mentioned component needs data to be able to update correctly. Therefore, those components subscribe to the **Redux's** store so that every time

the store is updated, those components are informed about the update and then perform appropriate actions.

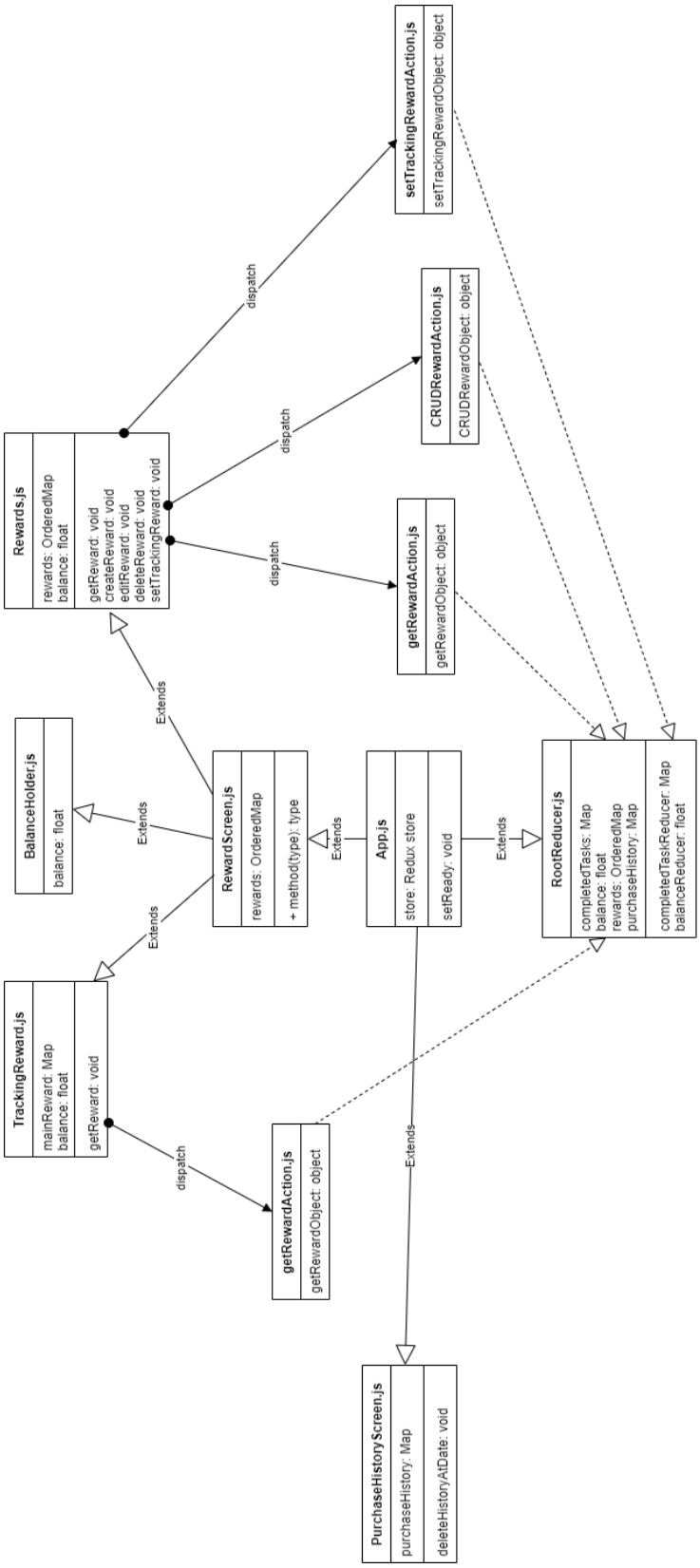


Figure 10 Class diagram for CRUD rewards and tracking main reward

The picture above shows the relationships between modules in Quint of performing CRUD reward operations and tracking the main reward in advance.

Firstly, for CRUD reward operations, **RewardScreen.js** module is the screen module that undertakes the wrapping functionality. The module contains three major components, which are **TrackingReward.js, BalanceHolder.js** and **Rewards.js**. Each component subscribes to the **Redux**'s store so that they will get notified when the store is updated. **TrackingReward.js** is where the app handles the work for the main reward. Users can track the wanted reward there to see the progress of the task so that they can focus even more. **BalanceHolder.js** is the component displaying the current amount of points that a user has. This component will be changed each time a user completes/uncompletes a task or purchases a reward. **Rewards.js** module handles the CRUD operations of rewards. Users can buy or get a defined reward from here. Earned rewards are displayed in the screen module **PurchaseHistoryScreen.js**. This module also subscribes to the store and has a function to delete a date containing purchased rewards within that date to free up space.

3.4.2 The Back-end side

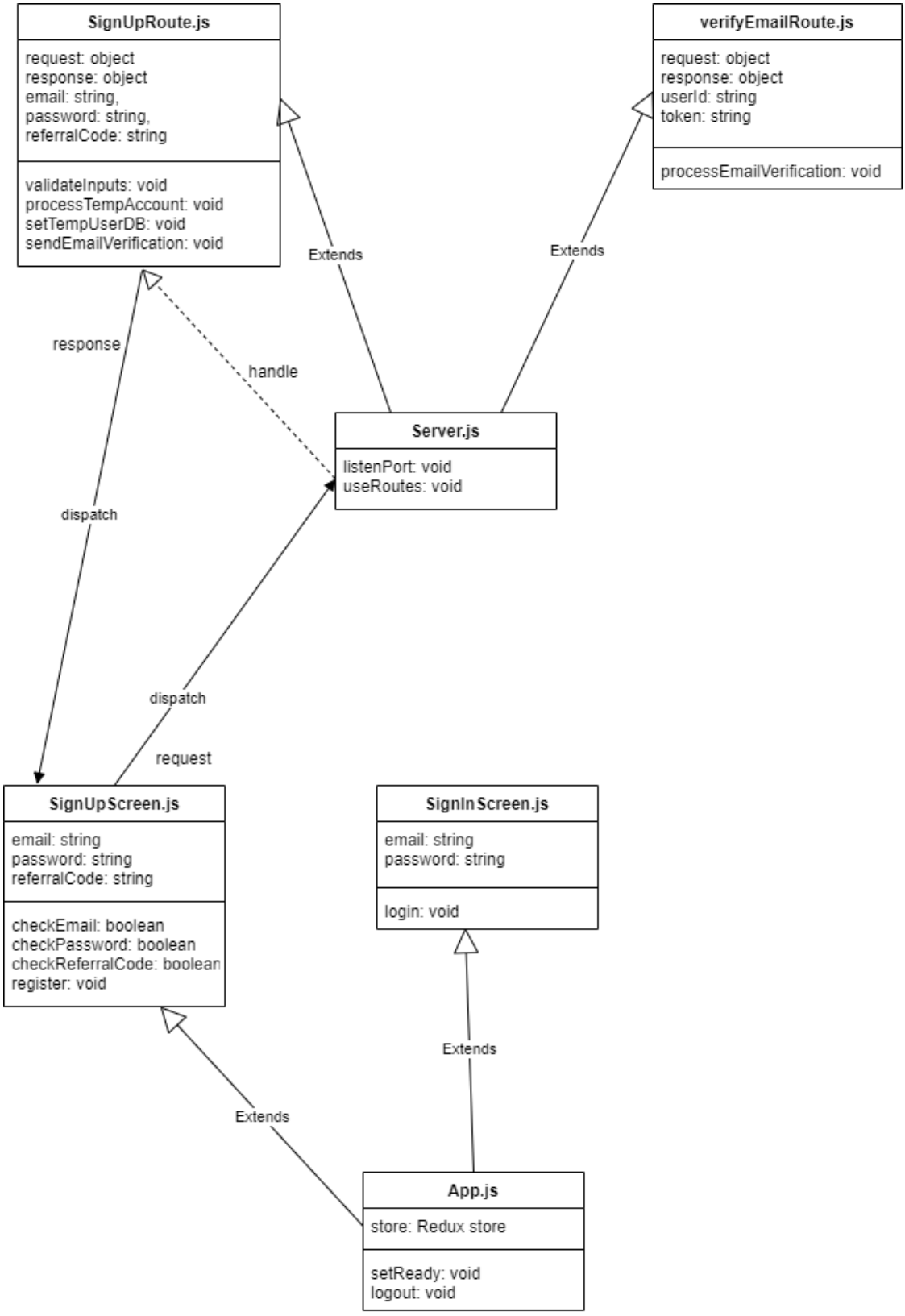


Figure 11 Class diagram for signing up a new account, sign in and sign out with email and password method

There are two main screens in Quint that are responsible for signing up accounts and logging in the system. First, **SignUpScreen.js** module is the screen component that allows users to create new accounts. At the time this thesis is being conducted, Quint requires two mandatory fields for new accounts, which are emails and passwords. **referralCode** is an input field for new accounts in order to earn a free month access of Premium package. To be able to have a referral code, users can achieve one via online events generated by the Quint team, or subscribe to new letters at the website, or get from a friend who already has a Quint account. Every time an account is created, it will gain a coupon code known as referral code. Account owner can give away that code to anyone in exchange of a free month Premium access if the code is used when a new account is created.

When hitting the button for registering an account, the app then sends a request to **Server,js**, which is the entry point of our server. Then, the server will find the proper route to handle the request. In this case, the request is processed in **SignInRoute.js** firstly. After running every method, the module sends an email verification to the registering email address. When clicking on the link shown in the email, a request is sent to **verifyEmailRoute.js** module and the app proceeds to finish the rest of the new account's registration. In case the verify link has not been clicked for 24 hours, the token will expire leading to the link becomes no longer valid. In such case, the server will schedule a cron job to clean up the temporarily created data in the database to free up space.

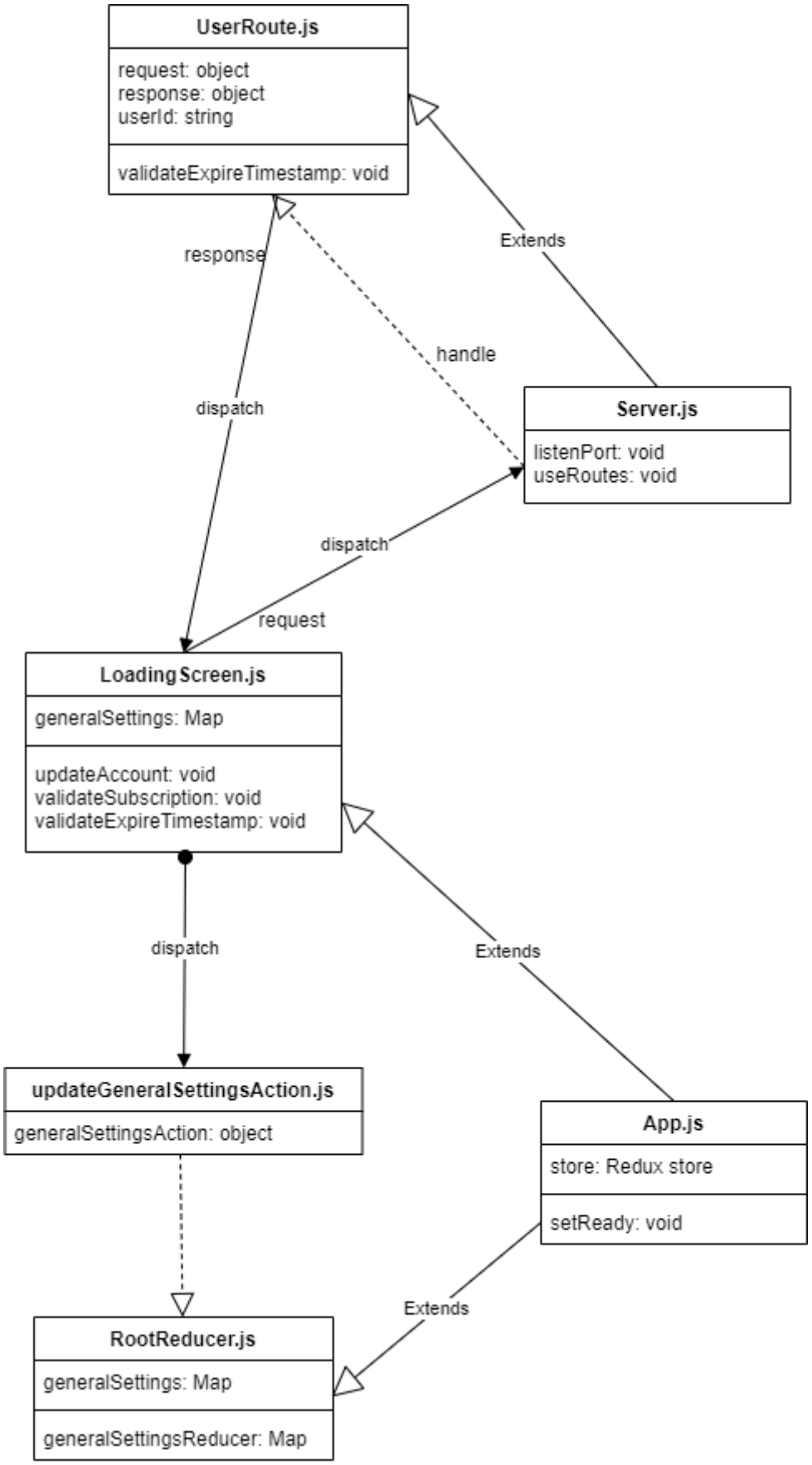

UserRoute.js
request: object
response: object
userId: string
validateExpireTimestamp: void
Extends
response
handle
dispatch
Server.js
listenPort: void
useRoutes: void
dispatch
request
LoadingScreen.js
generalSettings: Map
updateAccount: void
validateSubscription: void
validateExpireTimestamp: void
dispatch
Extends
updateGeneralSettingsAction.js
generalSettingsAction: object
App.js
store: Redux store
setReady: void
Extends
RootReducer.js
generalSettings: Map
generalSettingsReducer: Map

Figure 12 Class diagram for running initial checks

When a user opens the application, the app will run certain checks. It is important to note that those checks will be only ran when the application is initialized, which is not the case that the app comes to foreground (active) from background (inactive). When being opened, the app shows a loading screen, which is **LoadingScreen.js** module. In this phase, two functions are executed: **validateSubscription** and **validateExpireTimestamp**. Both methods involve communications with the server and the route, which is responsible for handling relevant requests, is located in **UserRoute.js** module. After receiving the responses from the server, **LoadingScreen.js** dispatches an action called **updateGeneralSettingsAction.js** to update the respective state in the **Redux's** store. Lastly, **App.js** calls the function **setReady** so that Quint is ready to serve.

3.5 Sequence Diagrams

In this section, the thesis discusses sequence diagrams of specific functionalities in Quint. For clarity, followed are two main sections containing respective sequence diagrams: the front-end sequence diagram and the back-end sequence diagram. With these diagrams, readers can have a clearer view over how functionalities in Quint work in general.

3.5.1 The Front-end side

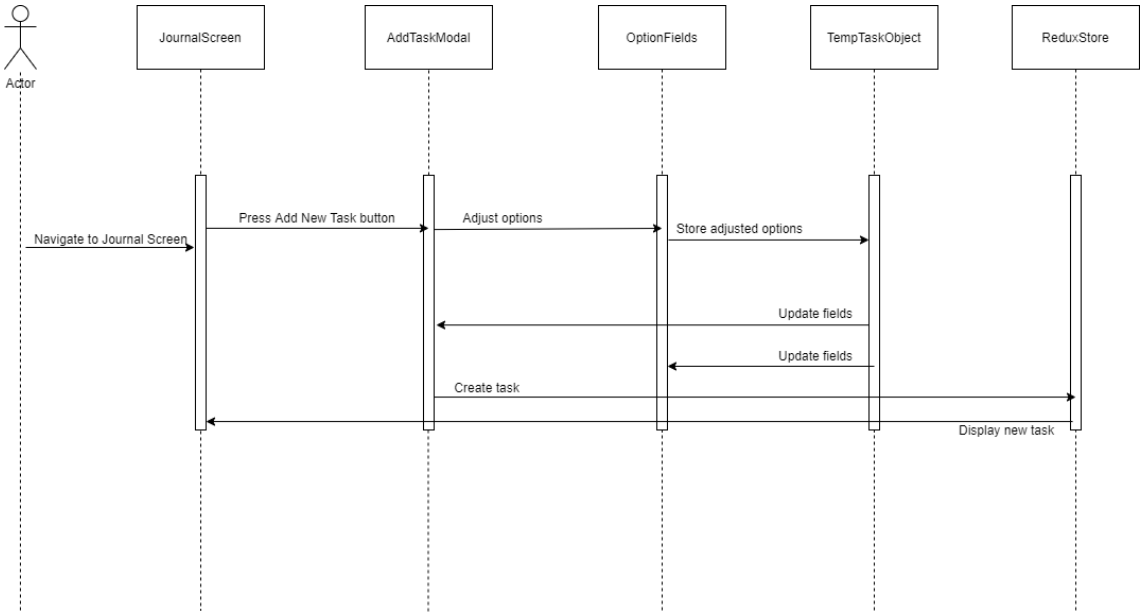


Figure 13 Sequence diagram for creating a task

Above is the figure showing the implementation of creating a single task in Quint. Basically, the app has four main screens: Journal, Progress, Rewards and Settings screen. Journal screen is where users can gain access and controls over their tasks. Firstly, to create a task, user must navigate to the Journal screen. The button for adding new tasks is only available when users are at that screen. When clicking on the button, a modal containing necessary fields and options for a task prompts up allowing users to modify their wanted tasks.

There are chances that users are creating a new task but then change their minds. Therefor, the app creates a temporary object for the ongoing process to save users inputs, which later loads those inputs again when users decide to continue with the process. Lastly, when finishing adjustments, an action will be dispatched to the Redux's store in order to update the state tree. At the same time, components, which subscribe to updated properties of the state tree (reducers), are sequentially re-rendered with new data.

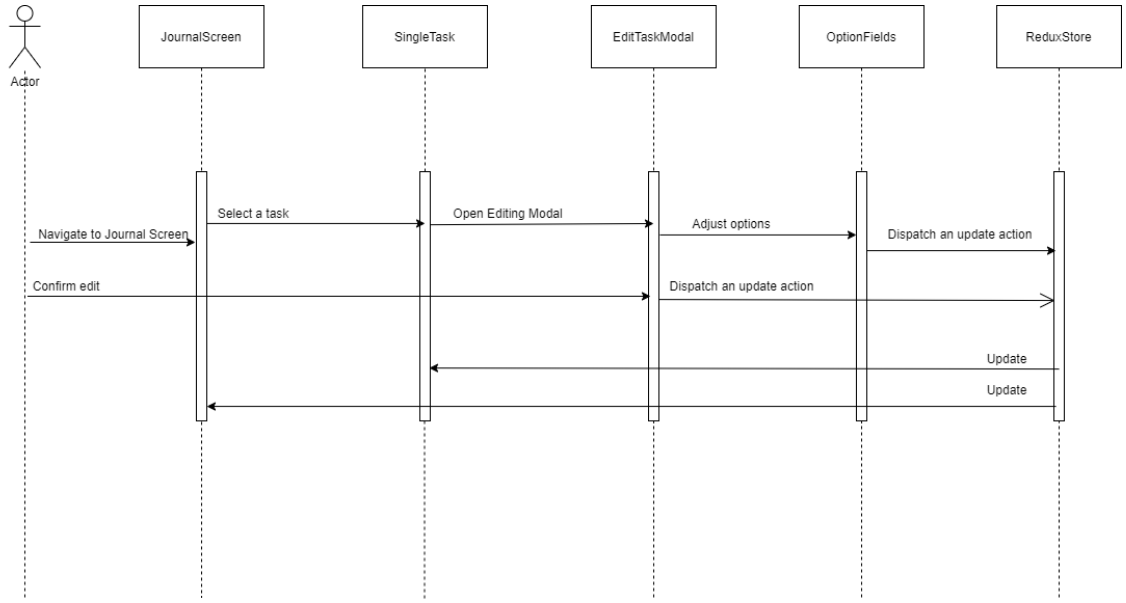


Figure 14 Sequence diagram for editing a task

The update operation for a single task in Quint is illustrated in Figure 14. When a task is created, it will be displayed in a list in Journal screen. To display a list of tasks, Quint depends on two criteria: date and category. For example, a user creates a task with the schedule option of 15th of May 2020 (15/05/2020) and the task belongs to the Work category. In addition, because Quint is a productivity app, each task is repeated indefinitely by default (with an interval of one day/week/month). Thus, if the user wants to view the created task with the type of Day, the repetition of default (every day), he/she must navigate to 15th of May 2020, or 16th of May 2020, or 17th of May 2020, etc while choosing the Work category from the Drawer.

After selecting the correct task to edit, a user can click on that task, which brings up a modal containing the task's information. He/she can directly edit the task's fields in the modal and those actions will be dispatched to the store to update the state tree.

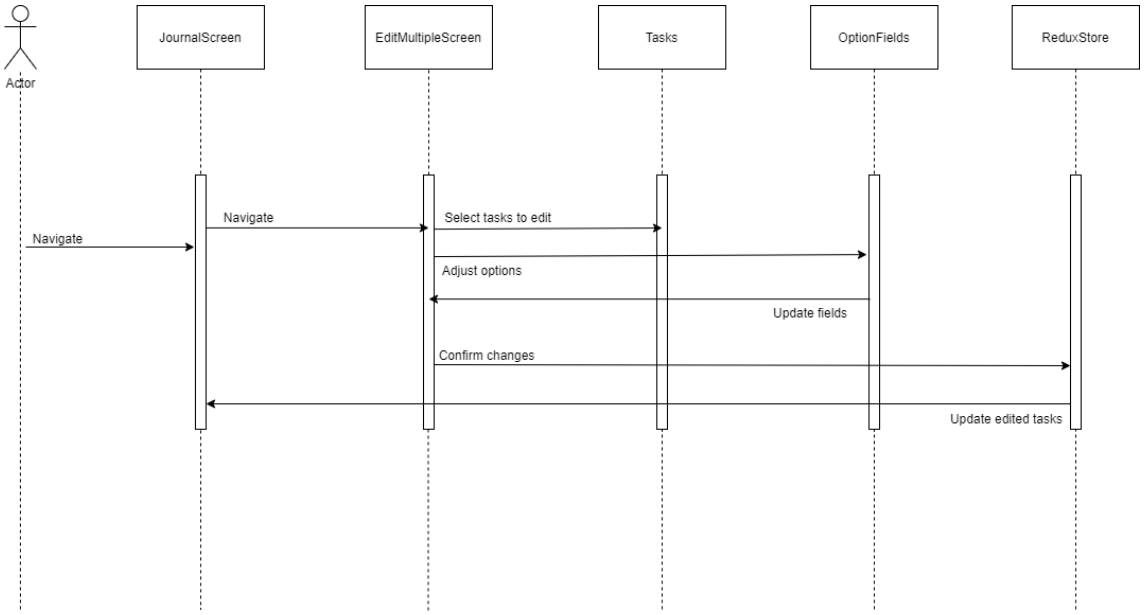


Figure 15 Sequence diagram for editing multiple tasks

In the above diagram, the workflow of updating multiple tasks is shown clearly. As being said, Journal screen is where a user can control his/her tasks. Quint supports editing a bulk of tasks at a time. By design, users can only update tasks that are in the same category. However, there is a category called “All tasks”, which basically is the common category containing all created tasks. Furthermore, there are a limit of options that users can edit when updating multiple tasks. Currently, users can edit the schedule and category options in update-in-bulk mode only. The reason behind it is that there are difficulties and unnecessary logics for users to understand, which also are not easy to implement in a developer’s perspective as well.

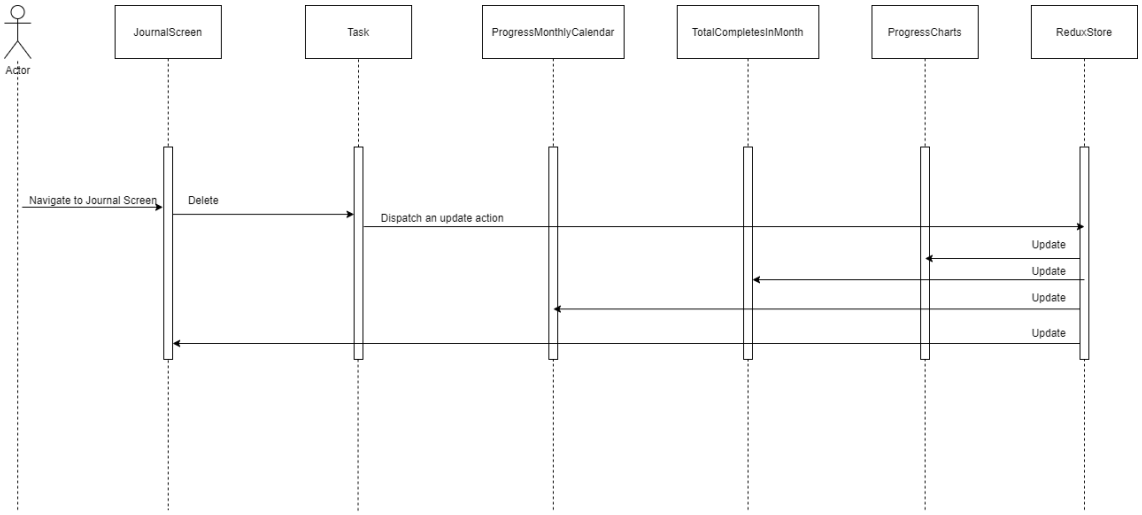


Figure 16 Sequence diagram for deleting a task

The diagram in Figure 16 illustrates the workflow of deleting a single task, which represents how Quint implements the deletion of tasks. To understand the work further, readers need to understand that deleting or updating a task results in a chain of required re-calculations of points, total task completions, completions of task types and completions based on date. All those calculations are mandatory to display correct statistics and charts in Progress screen. There are two types of deletions implemented in Quint. The first type is to delete a task at a specific date. The second one is to delete a task at its every date (completely removal). The first type of deletions recalculates the task's statistics in a specific date. For example, there is a Day task starting at 20th of May 2020 with the default repetition (repeat every day). A user can delete the instance of that task at 25th of May 2020, which results a deletion of statistics of May 25th, 2020 in Progress screen. On the other side, a complete removal of that task emerges a cut down of statistics of all date starting from 20th of May 2020 in Progress screen.

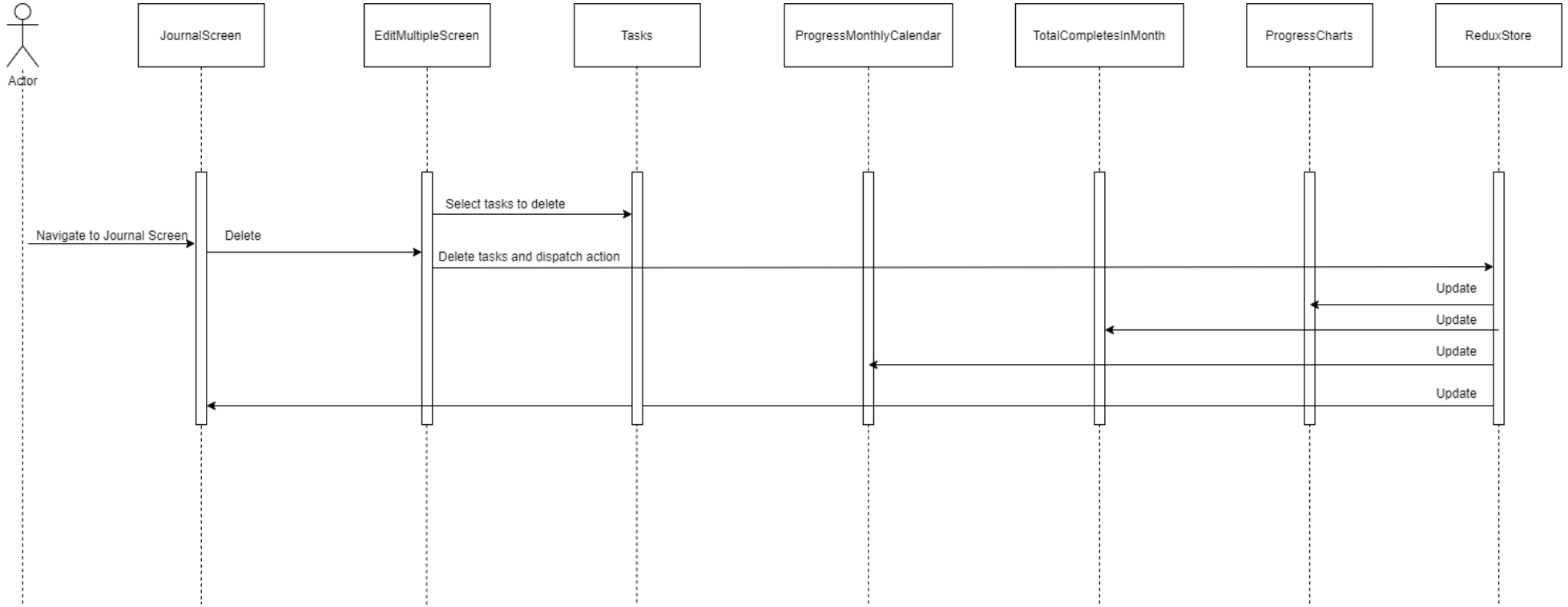

Actor
JournalScreen
EditMultipleScreen
Tasks
ProgressMonthlyCalendar
TotalCompletesInMonth
ProgressCharts
ReduxStore
Navigate to Journal Screen
Delete
Select tasks to delete
Delete tasks and dispatch action
Update
Update
Update
Update

Figure 17 Sequence diagram for deleting multiple tasks

The sequence diagram in Figure 17 illustrates the implementation of deleting multiple tasks in Quint. Like deleting a single task, there are two options for users to choose whether they want to delete at a specific date, or they want to remove selected tasks completely. By selecting tasks, there will be a temporary array to hold the selected task ids so that the system can perform correct changes upon those tasks. Two types of deletions in deleting multiple tasks work similarly compared to deleting a single task.

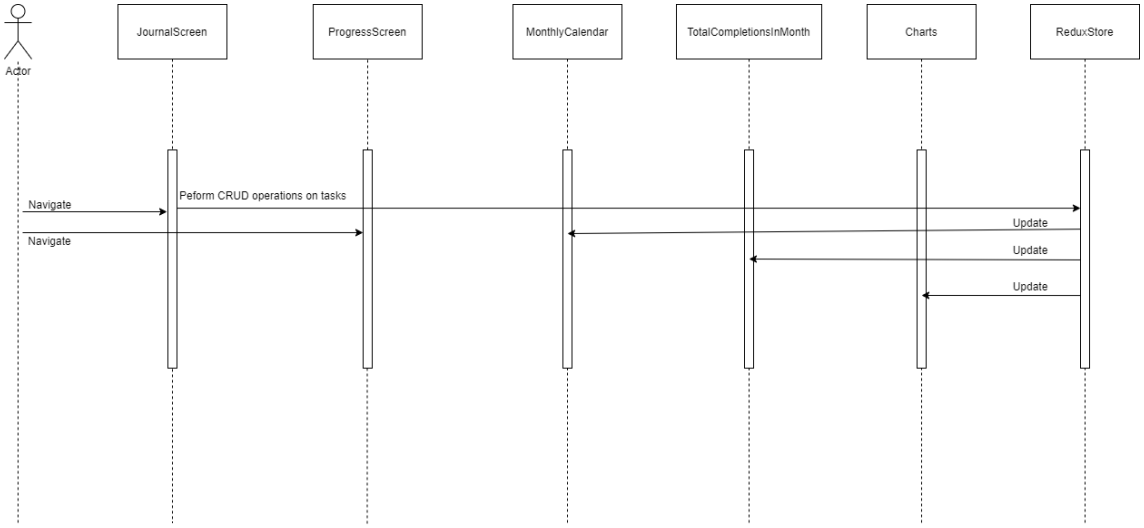


Figure 18 Sequence diagram for updating statistics and charts

As shown in Figure 18, the workflow of updating statistics and charts in Progress screen is quite straightforward. Normally, when a user performs CRUD operations of tasks in Journal screen, three main components in Progress screen will get updated accordingly. To be specific, when a task is marked completed, its defined points will be calculated and saved to the state tree. Not only saving the points, Quint also counts the completion of the task itself in several ways. For instance, **TotalComponentInMonth** component displays the total completions of each task type (Day, Week and Month). The completions of Day task type will increment by one if a Day task is completed. The logic is applied to Week and Month tasks as well.

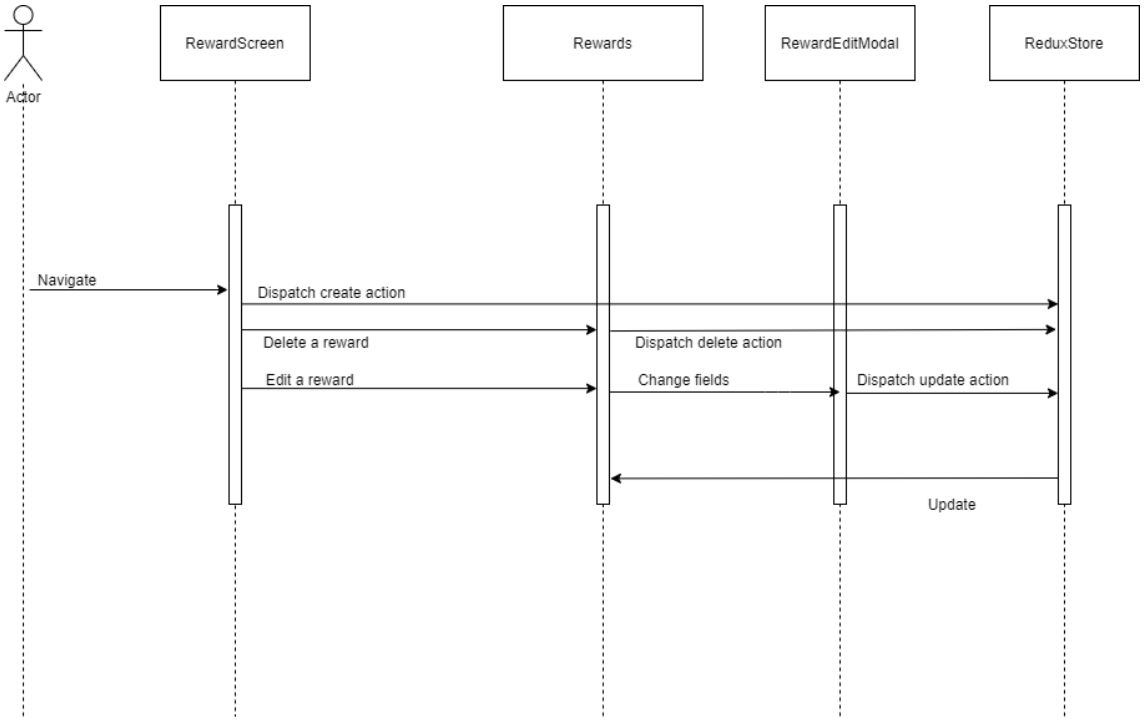


Figure 19 Sequence diagram for CRUD operations of rewards

The Reward screen allows users to perform CRUD operations on rewards. There is no limit of defining a reward. It is totally up to users to define their deserved rewards, as long as those rewards are reasonable and motivative. The CRUD operations are described in the sequence diagram in Figure 19. There is a list of rewards displayed in Reward screen, whose parent component subscribes to the Redux's store and gets updated whenever the subscribed property of the state tree changes. All operations eventually dispatch actions to the store in order to update the state tree, which in return updates the list of rewards.

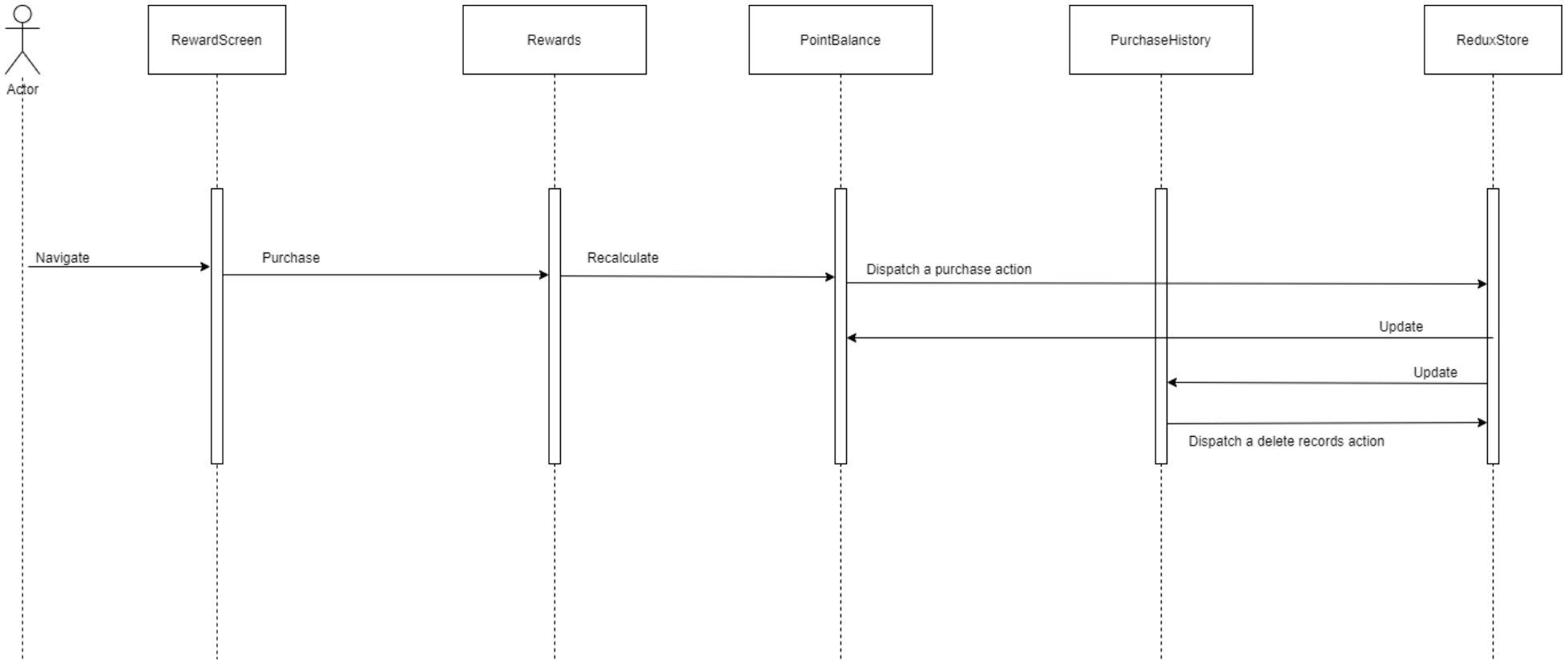

Actor
RewardScreen
Rewards
PointBalance
PurchaseHistory
ReduxStore
Navigate
Purchase
Recalculate
Dispatch a purchase action
Update
Update
Dispatch a delete records action

Figure 20 Sequence diagram for purchasing a reward

When completing defined tasks, users receive points. In Quint, points are the main currency to purchase goods, which in this case are defined rewards. Points are not fixed by the system. Users can change a task's points based on their desires. In most cases, Quint encourages users to relate to points as their in-used normal currencies such as euros, US dollars, Japanese yen. Having enough points allows users to get their wanted rewards for their productivities. Of course, to get a reward means in return a withdraw from the point balance of a user. The Reward system in Quint, indeed, works as a common grocery shop. Furthermore, users have the ability to review their purchased rewards so that they will not forget to actually treat themselves. Later when finishing treatments, users can delete a purchase record to clean the view.

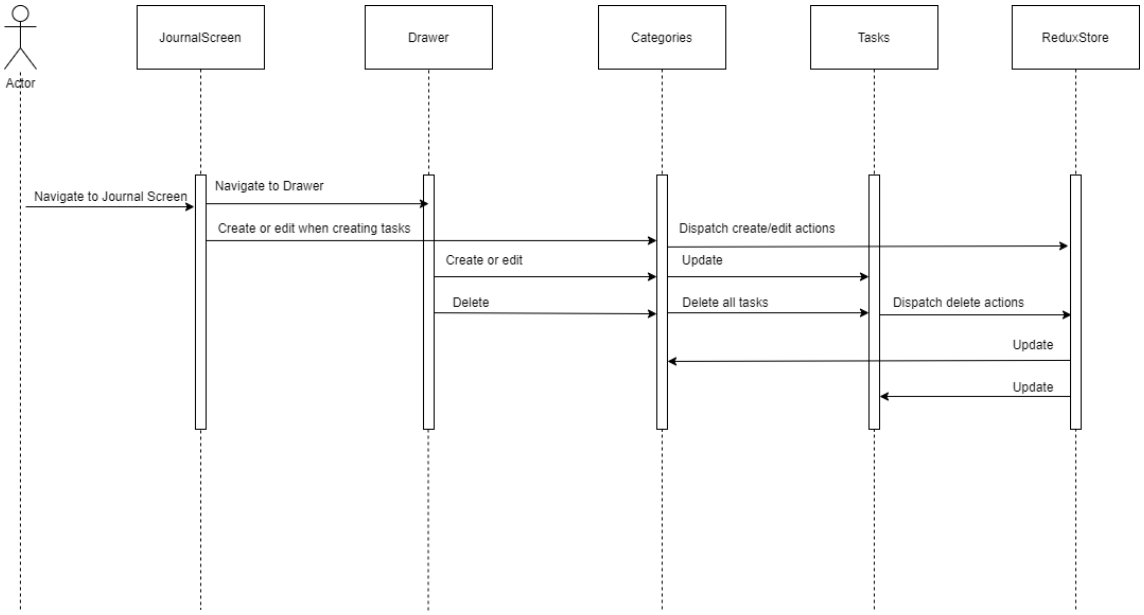


Figure 21 Sequence diagram of CRUD operations of categories

3.5.2 The Back-end side

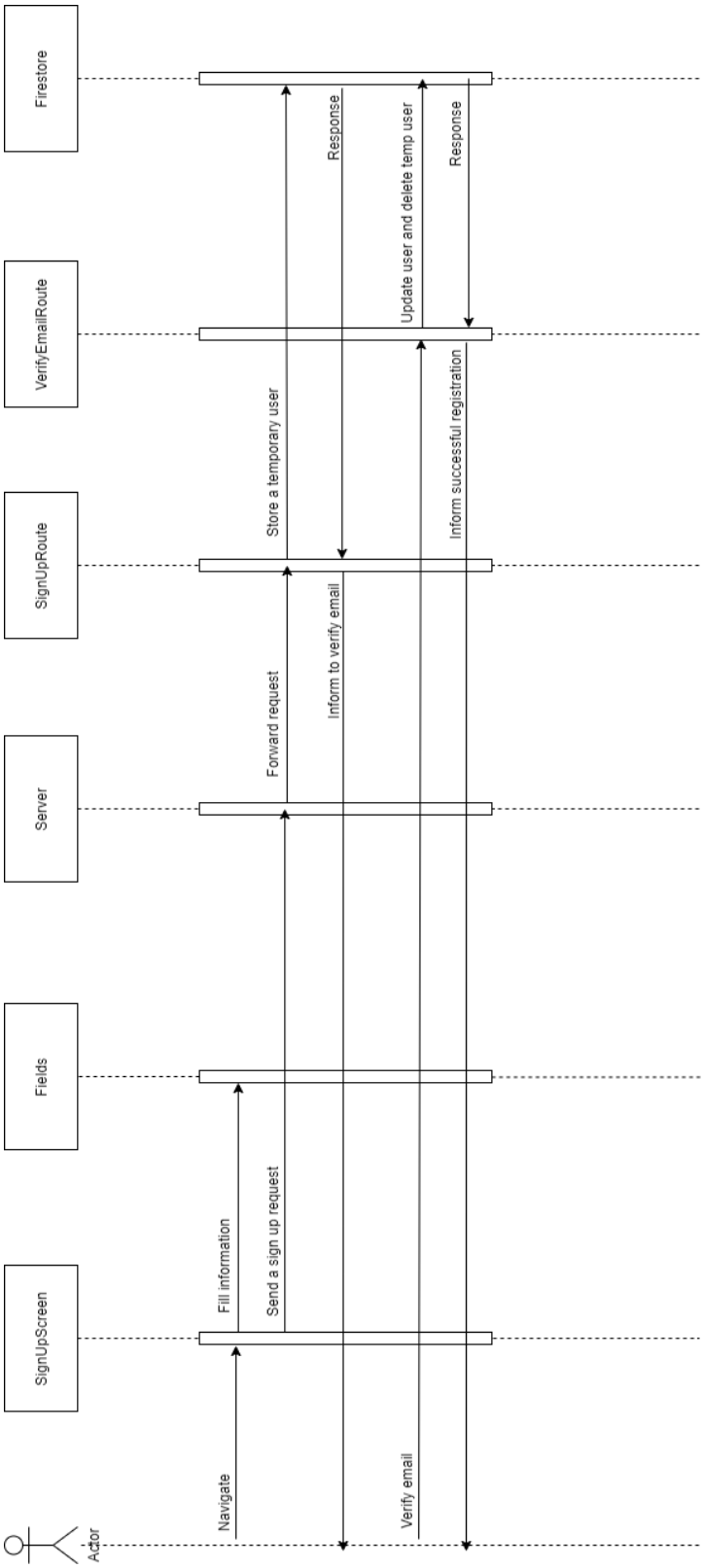


Figure 22 Sequence diagram for account registration

When a user wants to purchase the Premium package, or they want to sync data to the database (cloud), they need a Quint account to do so. The sequence diagram in Figure 22 explains the process of account registration in Quint. At the time writing the thesis, users can only create a new account within the app. This behaviour is certainly going to be changed in a short time.

To handle registering a new account, there are two implementations that developers need to put in mind. The first one is to handle the request sent from the form filled by a user in the app. This request contains important information such as email and password and is sent to Server, where it will be processed and forwarded to the right route handler. In this route, the server puts a temporary user data to the database with a unique registration token in order to validate the correct user when verifying email. Then if everything succeeds, Server responses back to the app and inform the user about a verification email sent to their registering email address. Step 2 is the last step, which handles the email verification and user update in Firestore.

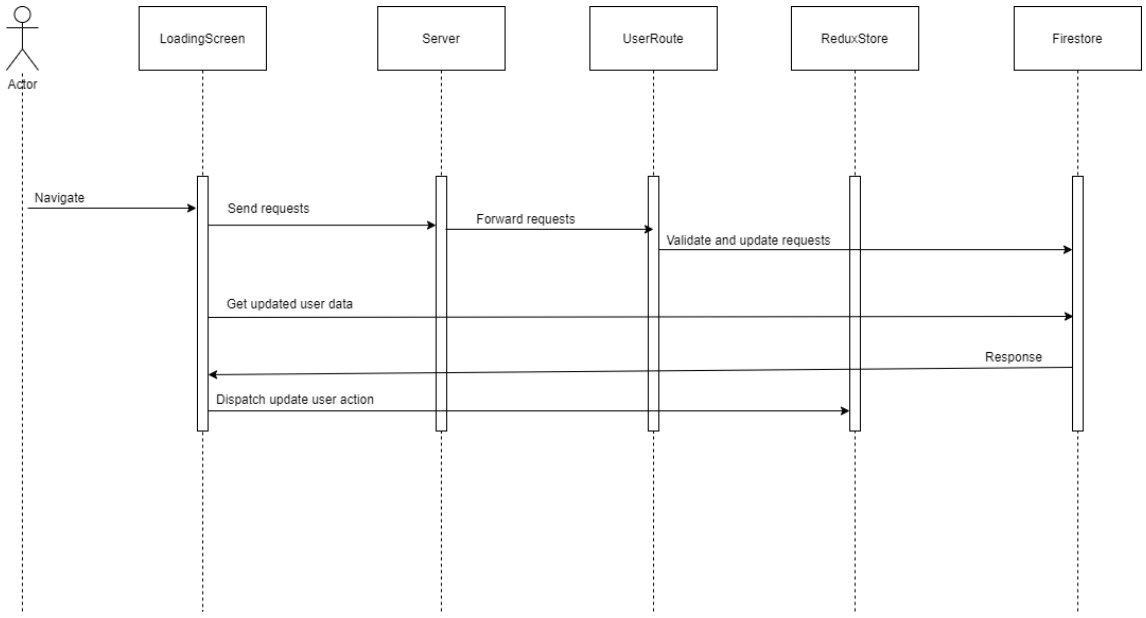


Figure 23 Sequence diagram for validating user subscription and expiration

When the app firstly initializes or wakes up from a cold start, this implementation will be run to make certain that the logged-in user has the up-to-date subscription status. The system sends two requests to the server if it is a Premium account. The first operation is

to check whether the account's Premium features are expired, which helps the app provide the most proper features to the user. The second operation is to check the validation of the user's subscription, which means to check whether the Premium package of the user wore off. Currently, subscription is implemented by using Apple's In-App-Purchase capability. For that reason, iOS subscription is what being validated in this process.

4 DATABASE DESIGNS & GUIS

In this chapter, the thesis discusses the GUI implementation of Quint, along with its database designs. The way of how this chapter works is for each database design, there will be explanation coming along with pictures of relevant GUI figures. In addition, this chapter is divided into two main sections: the Front-end side and the Back-end side as normal.

4.1 The Front-end side

It is important to remind that all the app's GUIs, colors and images are not designed by the author of the thesis. The credit goes to the design team, which is formed by close friends of the author. The design team is young in experience but still passionate about digitally UX & UI design in general. However, all animations of the app are implemented by the author despite of the fact that they are quite simple and plain.

Another important note is that in the Front-end side, Quint uses disk storage as the database. To implement the backbone of the database, the app uses ImmutableJS to performantly do general CRUD operations. There are multiple databases in Quint, such as a database for tasks, a database for completed tasks, a database for categories and a database for charts & statistics. Each database is a map created by ImmutableJS and rehydrated by Redux Persist. We will discuss this issue later in the thesis. The reason I chose using Map over List (Array) is that ImmutableJS provides a better querying performance with Map since Map does not require indexing (Quint does not need to index tasks in most cases so it is safe to use).

To keep this section short and readable, we are going to focus on the processes of Day tasks only as the processes of Week and Month tasks are pretty similar.

4.1.1 Task Database Design

There are three task types in Quint: Day, Week and Month. Each task type has a database in order to split up created tasks into smaller chunks.

```
interface dayTasksModel {
  taskId: {
    id: "taskId";
    title: string;
    description: string;
    category: "categoryId";
    repeat: {
      type: string;
      interval: {
        value: number;
        daysInWeek?: number[];
      };
    };
    goal: {
      max: number;
    };
    end: {
      type: string;
      endAt?: number;
      occurrence: number;
    };
    schedule: {
      year: number;
      month: number;
      day: number;
    };
    priority: {
      value: string;
    };
    reward: {
      value: number;
    };
  };
}

```

Figure 24 Database design for Day tasks

dayTasksModel can be considered as the name for the model of the database for Day tasks. Apparently, the database stores multiple Day tasks in key-value pairs. The key is a Day task's Id. By using this method, it is faster and more convenient to do CRUD operations on an intended task.

A Day task is also a Map, which unifies the idea of immutable programming across the application. To be exact, almost every object in Quint is converted to Map or OrderedMap, either is every array converted to List. There are common properties existing in Day, Week and Month tasks. Those properties are **id, title, description, category, goal, priority, reward**. Remained properties, which are **repeat, schedule, end**, will vary depending on the type of task, the type of repetition and the type of chosen ending option. Properties keys followed by **?** are optional (based on different chosen types).

In order to understand more about the issue, we will investigate a Day task created by default. The task has a title of “Work on some features”, a description of “Take a look at notes”, a priority of “Do first” (highest level), a reward of 5 points and a goal of completing 1 time a day. In addition, it belongs to the Inbox category, starts at 22nd March 2020 and ends at the same date as well.

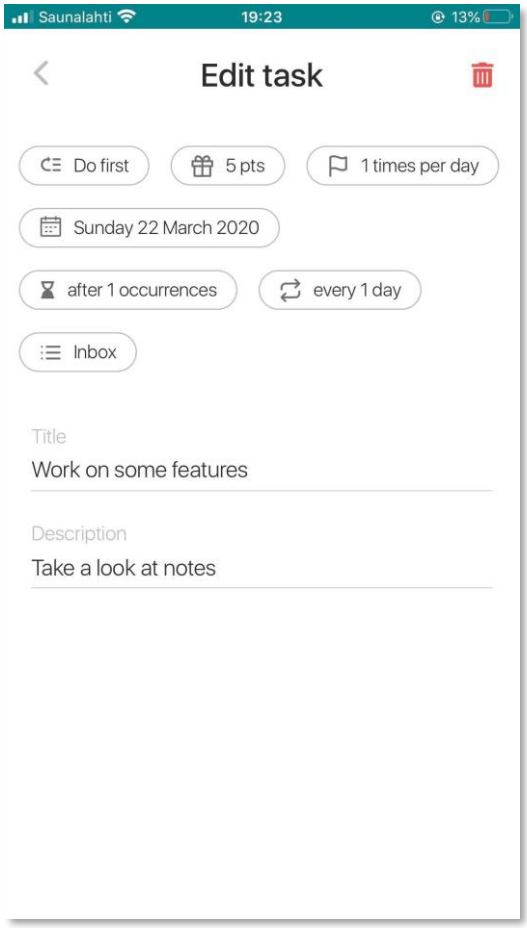


Figure 25 Example of a Day task's detail

As we can see, the Figure 25 displays the GUI of the Day task created with a title of “Work on some features”. Here comes the detailed implementation of the task.

```
const dayTasksModel = {
  taskId: {
    id: "taskId",
    title: "Work on some features",
    description: "Take a look at notes",
    category: "inbox",
    repeat: {
      type: "daily",
      interval: {
        value: 1
      }
    },
    goal: {
      max: 1
    },
    end: {
      type: "after",
      occurrence: 1
    },
    schedule: {
      year: 2020,
      month: 2,
      day: 22
    },
    priority: {
      value: "do_first"
    },
    reward: {
      value: 5
    }
  }
};

```

Figure 26 Detailed properties of the Day task example

This Day task was created by default, meaning I did not alter any option of it except for its title and description for a clearer recognition. It is important to know that **id** contains a cryptographically-strong random values generated by a package called **RFC4122 UUIDs**. By using such an identifier, it is certain that the app does not incidentally create a task with an existing id. **category** property contains a category's id, to which this task belongs. In this example, "inbox" is the id of category Inbox.

The property **repeat** has several variations based on the task type. Day task type has three repetition types, Week task type has two repetition types while Month task type has only one. For a Day task as the one in this example, there are “daily”, “weekly” and “monthly” repetition type. By default, Quint makes a Day task repeat daily. The property **interval’s value** indicates how often the task should happen daily, or weekly, or monthly. In our example, the task occurs every day. The property **goal’s max** presents how many times the task should be completed during the occurrence. The property **end** has three types in total: “never”, “on” and “after”. The “never” end type means the task goes on indefinitely. The “on” end type is followed by **endAt** property indicates the date that the task should stop. The “after” end type provides **occurrence** property to refer how many times the task should happen. For instance, the task in the example stops after showing up for one time, even when the repetition value is every day. The property **schedule** includes essential data about the starting date of the task. In this case, it encloses the starting year, month and the day-in-month.

There are four priorities in Quint and each priority implies a different emergent and important meaning. In the example, the task is prioritized as “Do first”, which has the most important indicator. The property **priority’s value** contains the “Do first” id. Lastly, the property **end’s value** displays the reward point that a user will get if he/she completes the task.

4.1.2 Category Database Design

```
69  interface categoriesModel {
70    categoryId: {
71      id: "categoryId";
72      name: string;
73      color: string;
74      taskQuantity: number;
75      plan: "free" | "paid";
76      createdAt: number;
77    };
78  }
79
```

Figure 27 Database design for categories

The model of a category is rather small. A category will contain its identifier, name, color and the number of tasks it possesses. The property **color** is to help people categorize the list of categories more easily since they can assign different colors to different categories.

In this section we discuss the default category which Quint sets up initially for users.

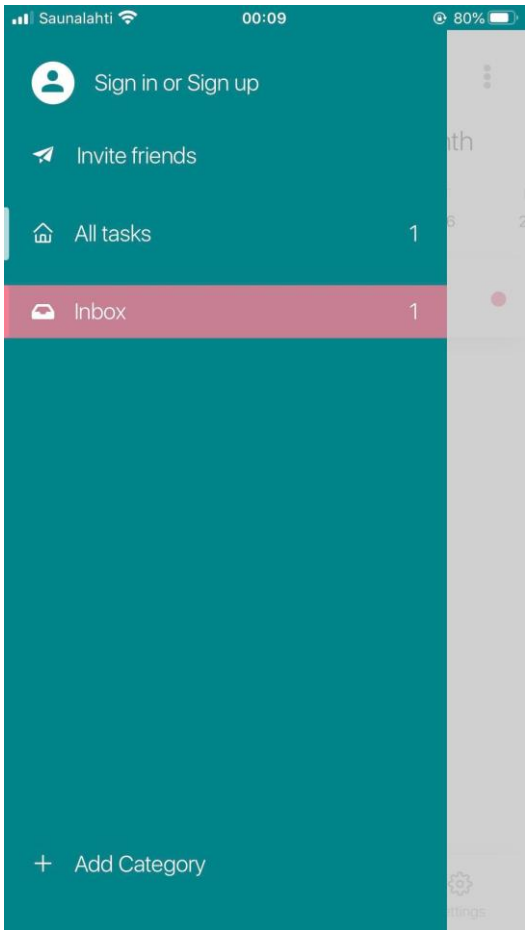


Figure 28 GUI example of the default category

As we can see in Figure 28, there is a category named Inbox. This is the mentioned default category. The property **id** of the category is defined as "cate_0" by default. For remained created category, their ids are uniquely generated similarly to task ids. The property **name** has a value of "Inbox" apparently. The property **color** has a value of "#F78096", which is displayed in hex value format. The property **taskQuantity** equals to 1 as there is only one task belonged to the category – the example task with the title of "Work on some features" in section 4.1.1. Currently, Quint has a limit for free users, which restrains them to create only 5 categories in total. By upgrading to Premium plan, users can create up to 99 categories. The property **plan** indicates whether this category was created during the "free" phase (meaning the first 5 categories) or during the "paid" phase (if users upgrade to Premium plan). The property **createdAt** refers to at what time the category was created.

4.1.3 Completed Task Database Design

```
interface dayCompletedTasksModel {
  taskId: {
    id: "taskId";
    category: string;
    completionTimestamp: {
      totalPoints: number;
      currentGoal: number;
      currentPriorityValue: string;
      completedPriorityArray: [number, number, number, number];
    };
  };
}
```

Figure 29 Database design for Day completed tasks

Because Quint displays tasks via dates, it is mandatory to keep track of a task's completion timestamp so that the app does know whether the task is completed or not. In Figure 29, the model for Day completed tasks are illustrated. It is a Map containing multiple Maps of completed tasks. The property **category** is the category's id that the task belonged at the time it is completed. This helps the app eliminate children task completion records when deleting a category. The property **completionTimestamp** indicates the completion timestamp of the task. The property **currentGoal** refers to the number of completion times that the user has made. This value does not exceed the defined goal value of the task. When the value equals to the defined goal value, meaning the task is completely accomplished. The property **totalPoints** is the total task reward points gained at the time the task is completed. The property **currentPriorityValue** tracks the latest priority level that the task is at. The property **completedPriorityArray** is a List containing 4 number-type items, which represents for the priority levels of Quint (Do first, Plan, Delay and Delegate). This List allows the app to update correct task-related completions in Progress screen, which will be shown later.

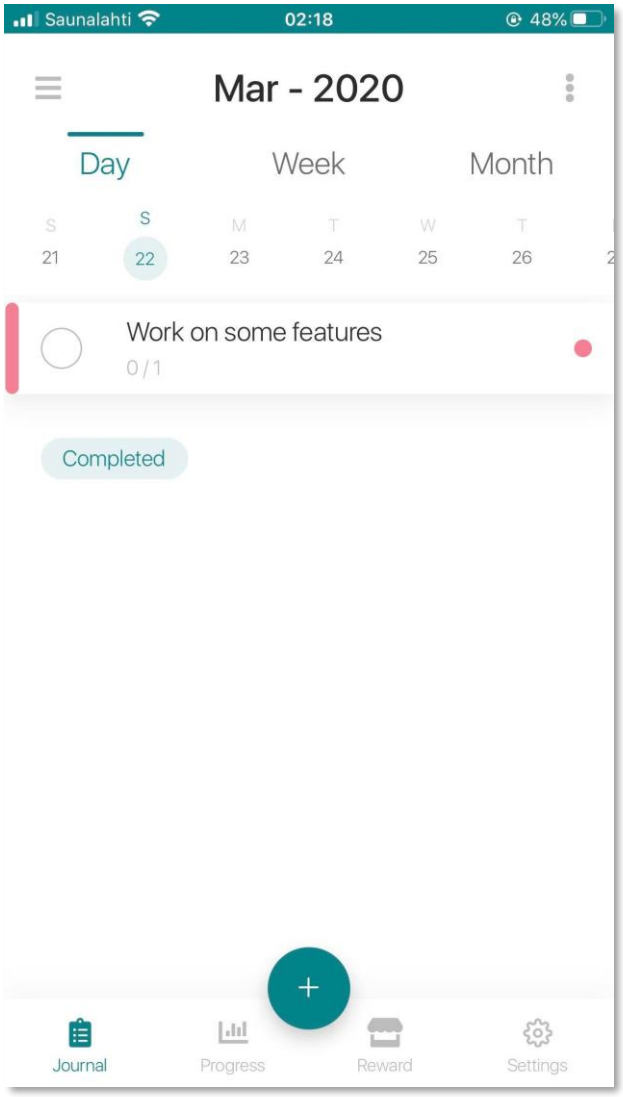


Figure 30 Uncompleted state of the example task

Figure 30 represents the uncompleted state of the example task used along this chapter. The task has a text of "0/1" saying that this task has a goal value of 1 and it has not been accomplished yet.

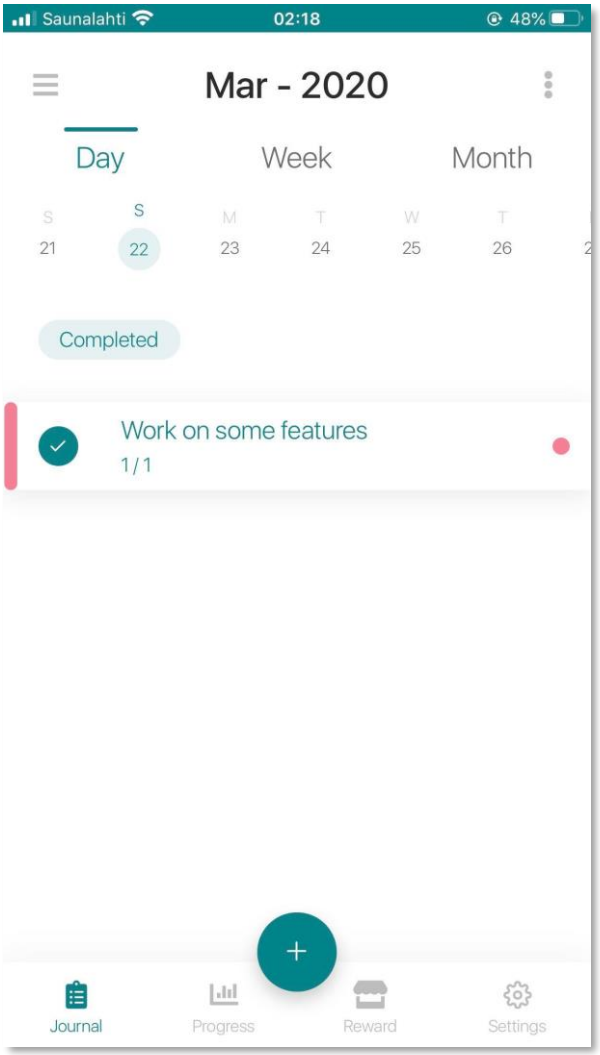


Figure 31 Completed state of the example task

By completing the task one time, the task now is completed since it only requires to be completed once.

dayCompletedTasksModel is updated with new data. A Map with the example task's id is added into the database. The property **completionTimestamp** is changed to the completion date in milliseconds. The property **currentGoal** is equal to 1. When the user uncompletes the task, the record remains and **currentGoal** is updated to 0. In the near future, uncompleting a task to its goal zero-value will result a deletion of the record in the database to free up space of the disk. The property **totalPoints** will be 5 as the reward value of the task is 5 and it is completed only one time. The property **currentPriorityValue** is

"Do first" priority's id. The property **completedPriorityArray** results in [1, 0, 0, 0] by the order of ["Do first", "Plan", "Delay", "Delegate"]. Furthermore, **completionTimestamp** of Day, Week and Month completed tasks database will be calculated based on the task type as following: Day task type results in calculating the milliseconds of the date including year, month and day that the completion occurs, Week task type results in calculating the milliseconds of the date including year, month, and the first day of the week that the completion occurs, Month task type results in calculating the milliseconds of the date including year, month, and the first day of the month that the completion occurs.

4.1.4 Database Designs of Day, Week, Month and Year Statistics

```
93  interface statisticsModel {
94    timestamp: {
95      totalPoints: number;
96      completedPriorityArray?: Array<number>;
97      taskTypeCompletion: [number, number, number];
98    };
99  }
100
```

Figure 32 Database design for general statistics

Figure 32 shows the general database design model in Quint. The app consists of 4 types of statistics databases representing for Day, Week, Month and Year as Progress screen requires detailed information about task completions. The property **timestamp** is similar to **completedTimestamp** mentioned in section 4.1.3, which is the completion date in milliseconds calculated based on the type of the database. The property **totalPoints** is similar to the **totalPoints** in section 4.1.3. The property **completedPriorityArray**'s existence depends on the type of the statistics database. If it is the Day statistics database, then there is no **completedPriorityArray**. The value of it also varies based on the type and the **timestamp** value. For example, **completedPriorityArray** in the Week statistics database is a list consisting of 7 values representing for 7 days in a week. On the other hand, it may consist of 30 values representing for 30 days in a month when for instance,

the timestamp is the first day of April. Every completed Day, Week, Month task updates all 4 databases in advance.

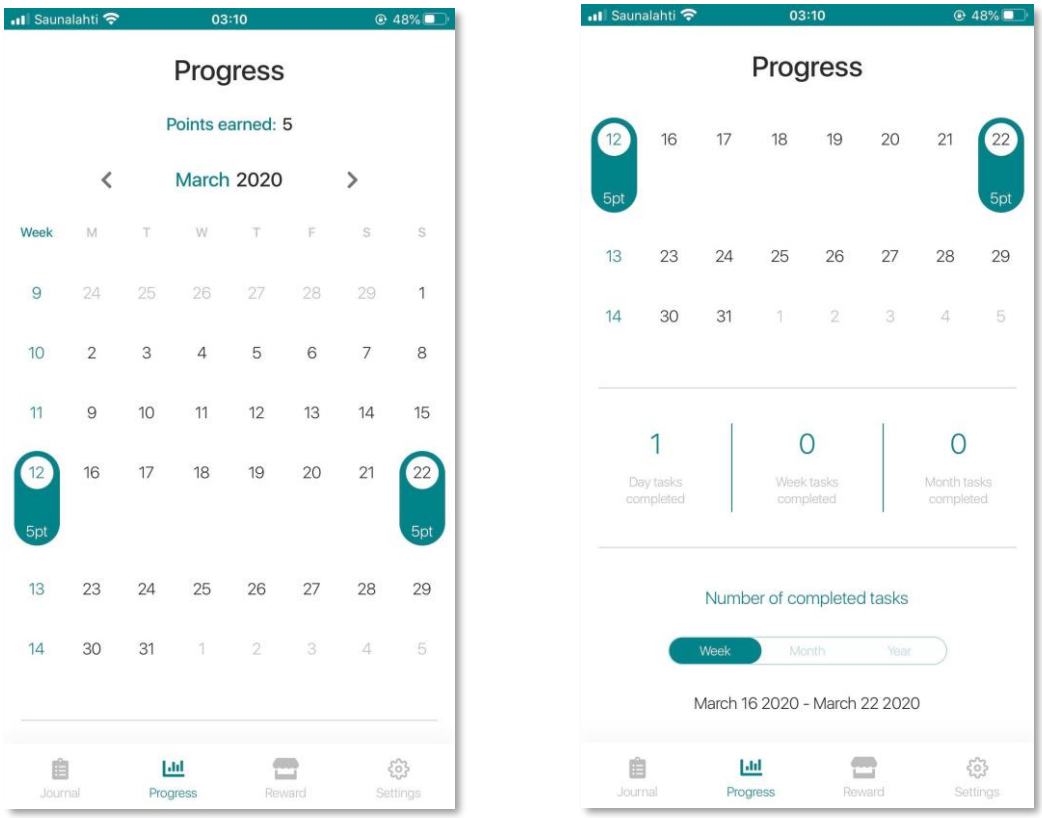


Figure 33 Monthly completion calendar and summary when completing the example task

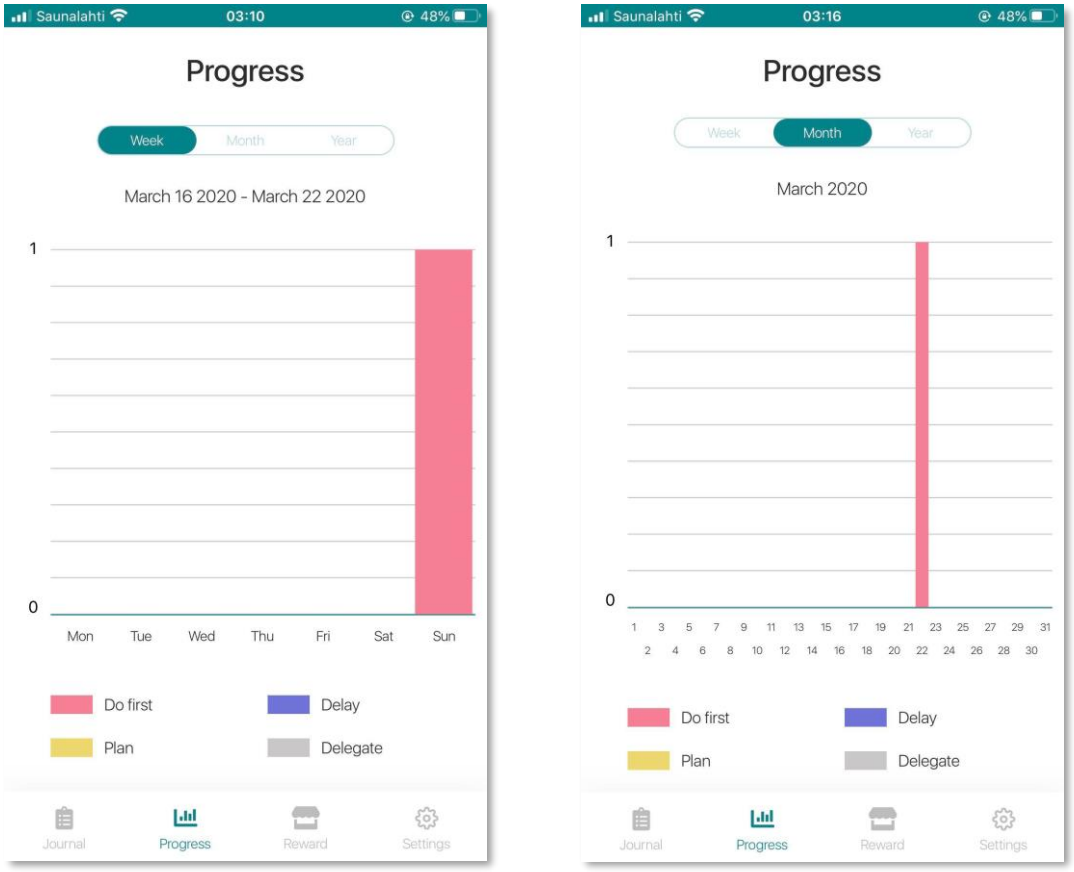


Figure 34 Week & Month Progess Chart when completing the example task

Figure 33 and 34 displays the sequential implementation of statistics and charts in Progress screen. The example task was completed at 22nd March 2020 with the goal value of one and the reward point of 5.

Above are crucial databases in Quint. Clearly there are many unmentioned databases. However, they are not mandatory to know to understand the project and to keep the thesis concrete and not too long, let move on to the next chapter.

4.2 The Back-end side

Quint uses Firestore to store user data, to validate subscriptions and expirations and as well as to sync data in the near future. In this chapter, I will provide database designs (collection and document designs in term of Firestore's methodology) of registration-related process and validation process.

4.2.1 User Database Design

In Quint, I use the User database for managing user-related processes such as new account registration, subscription validation and expiration validation.

```
interface usersModel {
  userId: {
    uuid: "userId";
    avatarUrl: string;
    createdAt: number;
    email: string;
    emailverified: boolean;
    expiryTimestamp: number;
    fullName: string;
    iosLatestReceipt: string;
    package: {
      billed: boolean;
      plan: "free" | "premium";
      renewalTimestamp: number;
    };
    referralCode: string;
    usedReferralCodeData: {
      createdAt: number;
      referUuid: string;
      value: string;
    };
  };
}
```

Figure 35 Database design for user-related data

Figure 35 displays the database design model of user-related data. The property **userId** is automatically generated by Firestore, which is definitely unique and easy for Firestore to index documents in the collection (rows in a table). The property **avatarUrl** holds a string exposing the url of the uploaded avatar if in case, a user updates his/her profile with an image. The uploading process will be handled with Firebase Cloud Storage and the url is retrieved from it as well. The property **emailVerified** indicates whether this account is validated or not. In case it is not, the server will schedule a cron job to delete it in order to free up space. The property **expiraryTimestamp** holds the date in milliseconds of the time that a user loses his/her Premium access. The property **iosLatestReceipt** has an encrypted string containing the receipt from the latest subscription that a user has made in iOS. By using this receipt, Quint knows whether the user's account is still subscribing to the plan or not. The property **package's renewalTimestamp** indicates the time that the

account should be charged again for extending the subscription. The property **referralCode** is the coupon granted when registering the account. The property **usedReferralCodeData's referUuid** indicates the **userId** property of another existing Quint account, which provides its referral code to the current account in order to gain a free month of Premium plan, if any.

4.2.2 Verification Token Database Design

```
125  interface verificationTokensModel {
126    tokenId: {
127      id: "tokenId",
128      tokenValue: string,
129      createdAt: number
130    }
131  }
132
```

Figure 36 Database design for verification tokens

The database is used to store tokens for verifying registered Quint accounts. The property **tokenId** is generated by default by Firestore. The property **tokenValue** is the exact value of the token, which will be compared once a user verifies his/her account. When clicking on the verifying url, the user is redirected to Quint's server email verification page, where the server should validate the verification token, as well as the user id. The property **createdAt** is to record the time in milliseconds that the token was created. The value is used to decide whether the verification link is expired. Normally, in Quint, the email verification link expires after 24 hours if there is no action performed, meaning the user does not click the link to verify the email. By doing this, the app certainly provides a strong security method to prevent false account registrations.

4.2.3 Referral Code Database Design

```
135 interface referralCodesModel {
136   referralCodeId: {
137     id: "referralCodeId",
138     value: string,
139     createdAt: number,
140     usedHistory: Array<Object>,
141   }
142 }
```

Figure 37 Database design for referral codes

At the time writing the thesis, Quint uses referral code marketing strategy to leverage the number of users. The way the referral codes works is that a referral code can be gained by participating online events arranged by the Quint team or when creating a new account. With this referral code, a user can grant a free month access of Premium plan to whomever used the coupon when signing up a new account. In return, the code owner also receives a free month of Premium plan as well. Certainly, there is no limit of using the referral code, meaning as long as the code is used to create verified Quint accounts, the owner receives a free month per account.

The property **createdAt** will be used in the future, when the campaign reaches its goal and the Quint team decides to limit the time of a referral code being used. The property **usedHistory** is an array containing objects about data of accounts registered along with the referral code such as their user ids and the times in milliseconds when the code was used (in other words, when the accounts were verified).

5 IMPLEMENTATIONS & RESULTS

Chapter 5 is where I address and analyze the implementations of critical requirements in Quint. For each explanation, there will be figures of its practical code, possible workflow diagrams and realistic results cropped from my Quint app in my device iPhone 7 Plus. As normal, the chapter will be broken down into two major sections: the Front-end side and the Back-end side. However, because this thesis is about to build a mobile application with React Native, so the Front-end side will be focused heavily.

5.1 The Front-end side

5.1.1 The application structure

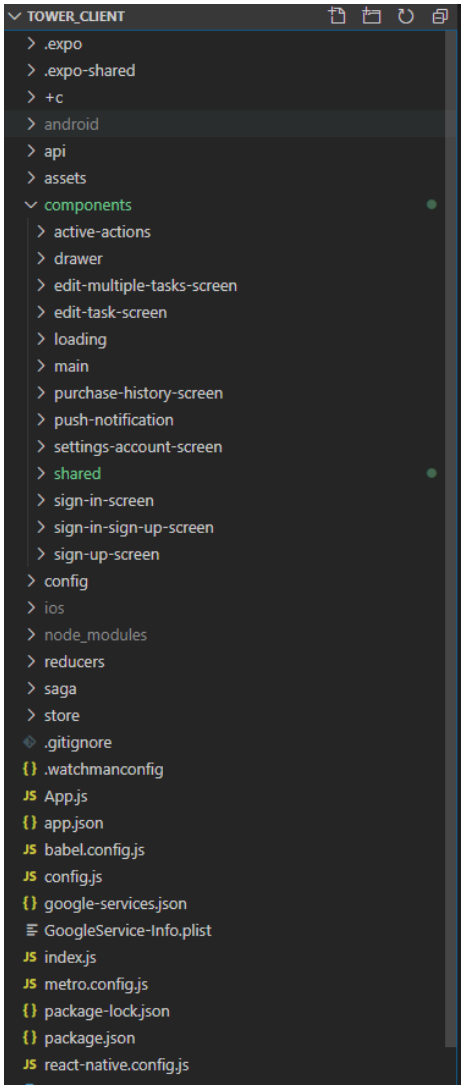
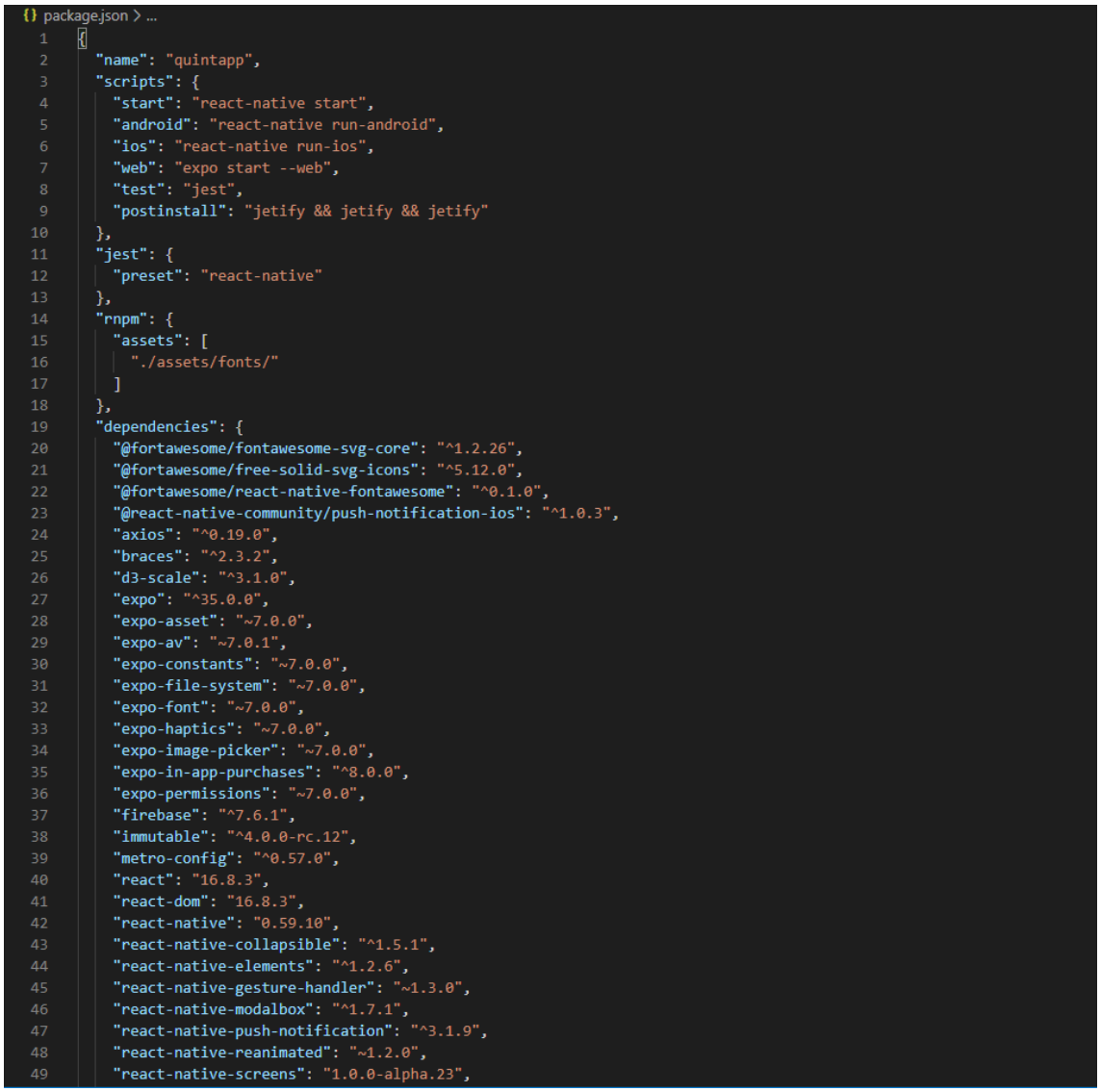


Figure 38 The structure of the client side

Above is the figure of the application structure used in the project. At the beginning, when starting the project using **Expo's Command Line Interface**, the project had a boilerplate for developing a React Native application. The default folders and files are such **App.js**, **index.js**, **node_modules**, **package.json** and **package-lock.json**. The **App.js** file is the entry file of the application, at where the highest level parent component is held. The

index.js file is the register file, which helps **Expo** and **React Native** recognize and run the application.



```
{} package.json > ...
{
  "name": "quintapp",
  "scripts": {
    "start": "react-native start",
    "android": "react-native run-android",
    "ios": "react-native run-ios",
    "web": "expo start --web",
    "test": "jest",
    "postinstall": "jetify && jetify && jetify"
  },
  "jest": {
    "preset": "react-native"
  },
  "rnpm": {
    "assets": [
      "./assets/fonts/"
    ]
  },
  "dependencies": {
    "@fortawesome/fontawesome-svg-core": "^1.2.26",
    "@fortawesome/free-solid-svg-icons": "^5.12.0",
    "@fortawesome/react-native-fontawesome": "^0.1.0",
    "@react-native-community/push-notification-ios": "^1.0.3",
    "axios": "^0.19.0",
    "braces": "^2.3.2",
    "d3-scale": "^3.1.0",
    "expo": "^35.0.0",
    "expo-asset": "~7.0.0",
    "expo-av": "~7.0.1",
    "expo-constants": "~7.0.0",
    "expo-file-system": "~7.0.0",
    "expo-font": "~7.0.0",
    "expo-haptics": "~7.0.0",
    "expo-image-picker": "~7.0.0",
    "expo-in-app-purchases": "^8.0.0",
    "expo-permissions": "~7.0.0",
    "firebase": "^7.6.1",
    "immutable": "^4.0.0-rc.12",
    "metro-config": "^0.57.0",
    "react": "16.8.3",
    "react-dom": "16.8.3",
    "react-native": "0.59.10",
    "react-native-collapsible": "^1.5.1",
    "react-native-elements": "^1.2.6",
    "react-native-gesture-handler": "~1.3.0",
    "react-native-modalbox": "^1.7.1",
    "react-native-push-notification": "^3.1.9",
    "react-native-reanimated": "~1.2.0",
    "react-native-screens": "1.0.0-alpha.23",
```

Figure 39 Image of package control file

Figure 39's content is about the content of the file **package.json**. As **React Native** uses JavaScript and requires a **Node** developing environment, it is mandatory that the project must have a file called **package.json** (the name is reserved) to control the project's information such as its name, author, version, to define command scripts in order to run in those in the CLI, and to manage installed packages (modules) and their package versions from the Node ecosystem. Installed packages must be compatible with React Native and

in most cases, they require additional set up steps to fully complete the installations. By having this file, it is easier for developers to keep track of currently existing packages in a application, as well as fixing compatibility problems when upgrading or downgrading package versions.

In the application structure, the **components** folder is where I keep all the business logic code. The folder contains child components of the app such as a screen, a navigator, a button. The **store** folder contains files belonged to **Redux's store.** The files are the configurations of the store. Lastly, the **reducer** folder contains **Redux reducers**, which are used in the app. The Redux store includes multiple reducers in order to help update the state tree.

5.1.2 **Redux store, reducers and actions**

```
import { createStore, applyMiddleware, compose } from "redux";
import thunk from "redux-thunk";
import createSagaMiddleware from "redux-saga";
import { batchDispatchMiddleware } from "redux-batched-actions";
import { persistReducer, persistStore } from "redux-persist";
import FSStorage from "redux-persist-expo-fs-storage";
import immutableTransform from "redux-persist-transform-immutable";
import rootReducer from "../reducers";
import rootSaga from "../saga";

const persistConfig = {
  transforms: [immutableTransform()],
  key: "root",
  storage: FSStorage(),
  blacklist: ["toggleEditMultipleTasks"]
};

const persistedReducer = persistReducer(persistConfig, rootReducer);

const sagaMiddleWare = createSagaMiddleware();

export const store = createStore(
  persistedReducer,
  applyMiddleware(batchDispatchMiddleware, thunk, sagaMiddleWare)
);

sagaMiddleWare.run(rootSaga);

export const persistor = persistStore(store);

```

Figure 40 Image of Redux's store configurations – store.js

For persisting the **Redux's state tree**, Quint is using **redux-persist** package. In the image above, the file exports two variables: **store** and **persistor**. The variable **store** holds the configurations for the **Redux's store**, which consists of middlewares. Middlewares are external packages used to achieved specific desires from developers. For example, a **thunk** middleware from **redux-thunk** allows developers to have an intermediate function before dispatching an action to the store. Meanwhile, **batchDispatchMiddleware** from **redux-batched-actions** provides the ability to batch or group multiple actions in one dispatch, which improves a lot in term of performance. The variable **persistor** is used to initialize the **Provider** wrapper component used in **App.js**, which enables the state tree's persistence. One last thing, the variable **rootReducer** is imported from the root file of the **reducers** folder, which contains all used reducers in the app. The root reducer is a function generated by using the method **combineReducers** provided by the package **redux.**

Let’s take a quick look into the reducer for updating Day tasks.

```
export const day_tasks = (state = Map(), action) => {
  switch (action.type) {
    case "UPDATE_DAY_TASK":
      return state.updateIn(action.keyPath, action.notSetValue, action.updater);

    case "DELETE_DAY_TASK":
      return state.delete(action.id);

    case "DELETE_ALL_DAY_TASKS_WITH_CATEGORY":
      return state.filterNot(task => Map(task).get("category") === action.id);

    case "RETURN_NEW_DAY_TASKS":
      return action.data.toMap();

    case "RESET_DAY_TASKS":
      return Map();

    default:
      return state;
  }
};

```

Figure 41 Reducer of Day tasks

The reducer for Day tasks is expressed as **day_tasks** function. The function takes two arguments: the current state of Day tasks (database of it) and the dispatched action from associated components. The state is initialized as a Map, as we discussed above that Quint is using Maps as databases to store data. The initial value of the state is, if converted to object, {}. The function will return the updated state in different cases. For example, when the action has a type of **UPDATE_DAY_TASK,** which is to add or edit a Day task, the reducer returns a new state (immutated one with a different origin in the memory) adding or editing that task. The method for updating is provided by **ImmutableJS**. To break down, **keyPath** is the task’s path in the Map, **notSetValue** is the value that will be set if there is no provided value in the updater, and **updater** is the function to update the Map.

```
import { batchActions } from "redux-batched-actions";
import {
  updateTitle,
  updateDescription,
  returnCorrespondCreatedTask,
  updateTaskTypeCreated
} from "../../../../../../../shared/actions/otherAction";
import {
  updateTask,
  resetNewTask
} from "../../../../../../../shared/actions/taskAction";
import { updatePriority } from "../../../../../../../shared/actions/priorityAction";
import { updateCategory } from "../../../../../../../shared/actions/categoryAction";

export const addTaskThunk = ({
  add_task_data,
  category_data,
  reset_new_task_type,
  priority_data,
  return_correspond_created_task,
  update_task_type_created
}) => (dispatch, getState) => {
  let actions_array = [
    updateTitle(""),
    updateDescription(""),
    resetNewTask(reset_new_task_type),
    updateTask(
      add_task_data.type,
      add_task_data.keyPath,
      add_task_data.notSetValue,
      add_task_data.updater
    ),
    updateCategory(
      category_data.keyPath,
      category_data.notSetValue,
      category_data.updater
    ),
    updatePriority(
      priority_data.keyPath,
      priority_data.notSetValue,
      priority_data.updater
    ),
    returnCorrespondCreatedTask(
      return_correspond_created_task.type,
      return_correspond_created_task.data
    ),
    updateTaskTypeCreated(update_task_type_created.data)
  ];

  dispatch(batchActions(actions_array));
};

```

Figure 42 Action of adding a task

As being said, Quint uses **thunk** middleware from **redux-thunk** to create an intermediate function, which will dispatch an action at the end, after executing some business logic. The function **addTaskThunk** is a thunk. At the end of the function, there is a method to dispatch action of **dispatch(batchActions(actions_array))**. The method **batchActions** from **redux-batched-actions** is in use due to the reason that the thunk has many actions to dispatch. If in case Quint does not use **batchActions**, each action will be dispatched separately and it will hurt the performance a lot since each dispatched action results a re-render of the connected component (imagine the component is very expensive, contains

multiple pictures and big modules to load). With the help from **redux-batched-actions**, the app just has to dispatch one action at the end. The method **batchActions** groups all defined actions in an array and send them when ready. This gives a great boost in term of performance since the app should re-render only once based on the actions-batched dispatch.

5.1.3 Add a task implementation

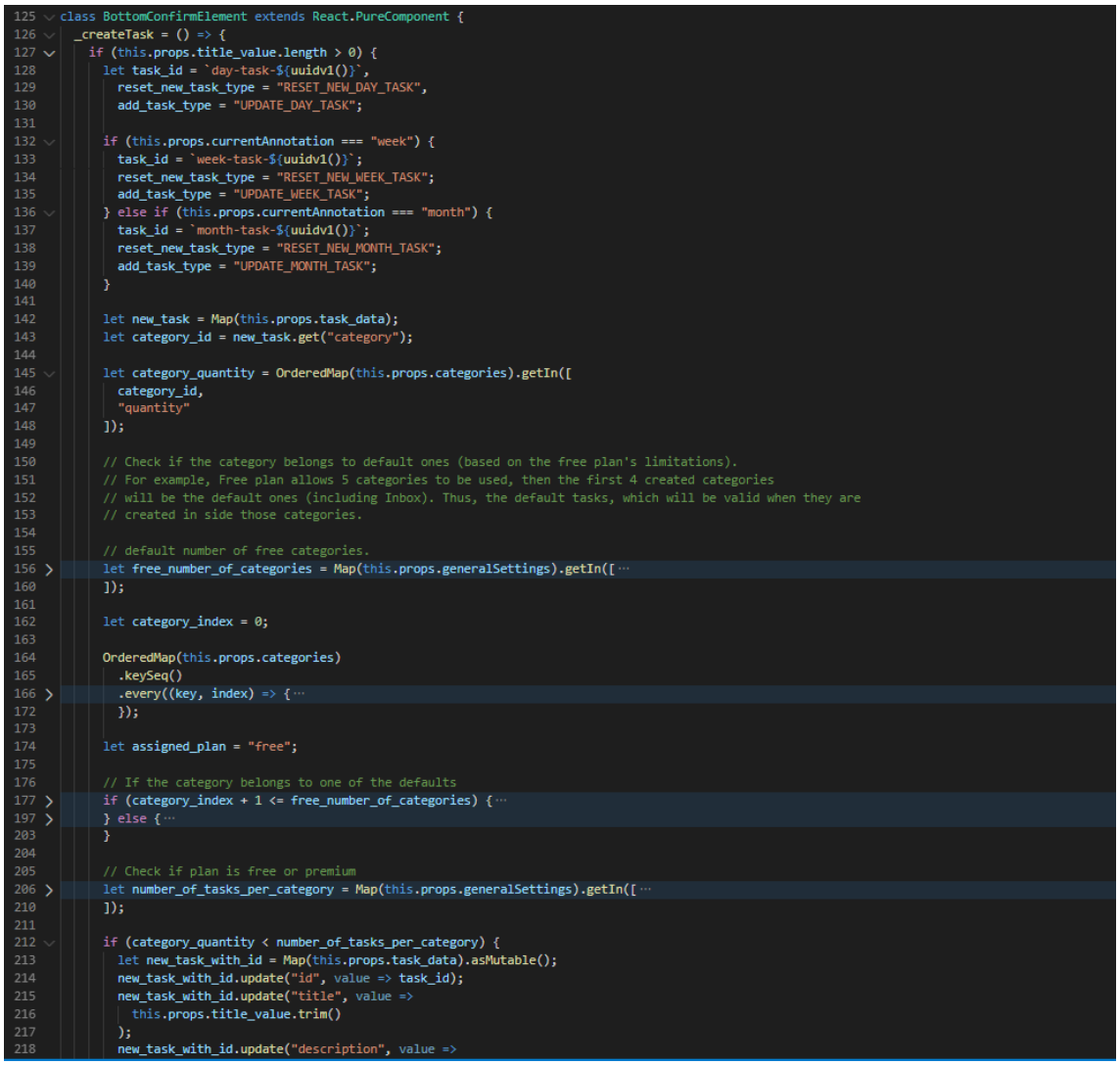
```
import { connect } from "react-redux";
import { addTaskThunk } from "./actions/addTaskThunk";

import BottomOptionsHolder from "./BottomOptionsHolder";

const mapStateToProps = (state, ownProps) => {
  if (ownProps.currentAnnotation === "day") {
    return {
      task_data: state["currentDayTask"],

      categories: state["categories"],
      priorities: state["priorities"],

      addTaskDescription: state["addTaskDescription"],
      addTaskTitle: state["addTaskTitle"],
      generalSettings: state["generalSettings"]
    };
  } else if (ownProps.currentAnnotation === "week") {
    return {
      task_data: state["currentWeekTask"],

      categories: state["categories"],
      priorities: state["priorities"],

      addTaskDescription: state["addTaskDescription"],
      addTaskTitle: state["addTaskTitle"],
      generalSettings: state["generalSettings"]
    };
  } else
    return {
      task_data: state["currentMonthTask"],

      categories: state["categories"],
      priorities: state["priorities"],

      addTaskDescription: state["addTaskDescription"],
      addTaskTitle: state["addTaskTitle"],
      generalSettings: state["generalSettings"]
    };
};

const mapDispatchToProps = (dispatch, ownProps) => ({
  addTaskThunk: data => {
    dispatch(addTaskThunk(data));
  }
});

export default connect(
  mapStateToProps,
  mapDispatchToProps
)(BottomOptionsHolder);

```

Figure 43 Connector of the component in charge of creating a new task

Above is the connecting component of the component which is responsible for creating a new task in Quint. The component being responsible for creating a new task and rendering the view is named **BottomOptionsHolder**, which will be shown below. In Quint, every component that wants to alter the global state tree must be connected or subscribed to the **Redux's store**.

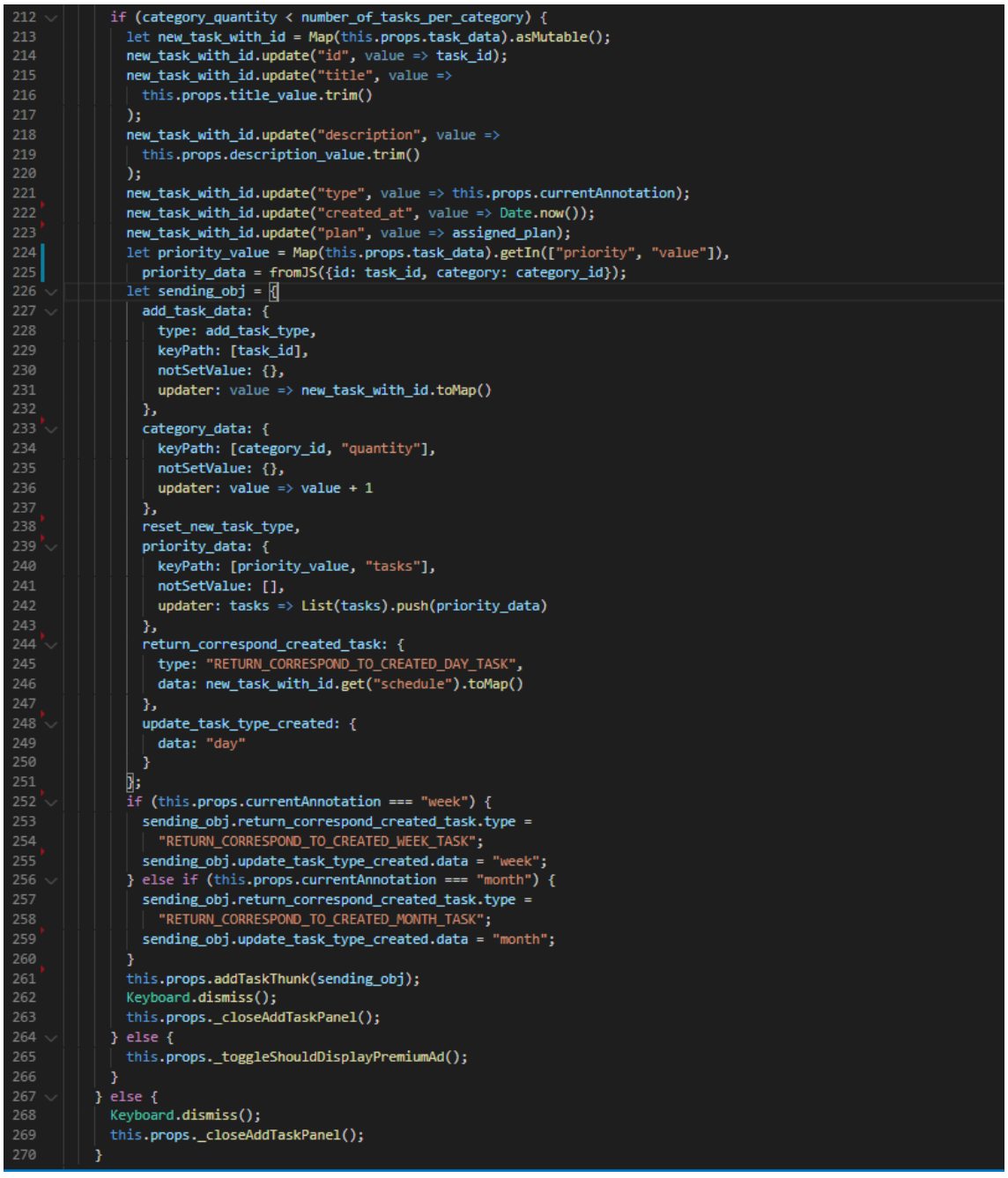
The connector consists of two major functions, which are **mapStateToProps** and **mapDispatchToProps**. The function **mapStateToProps** is the one getting necessary states from the state tree. By using the function, it allows the component re-renders when the state tree updates. The function **mapDispatchToProps** is the function used to pass the dispatch functions to the connected component. By using this function, the subscribed component can dispatch an action to the store in order to update the state tree. Using such the described connectors are helpful in preventing developers accidentally to dispatch an action directly to the store and to receive mass updates from it causing performance issues. Besides, it is tidy and easy to read in term of clean coding.

At the end, the connector is a High-Order React component, which receives the component **BottomOptionsHolder** as the input component and returns a new component providing a subscription to the store.



```
class BottomConfirmElement extends React.PureComponent {
  _createTask = () => {
    if (this.props.title_value.length > 0) {
      let task_id = `day-task-${uuidv1()}`,
        reset_new_task_type = "RESET_NEW_DAY_TASK",
        add_task_type = "UPDATE_DAY_TASK";

      if (this.props.currentAnnotation === "week") {
        task_id = `week-task-${uuidv1()}`;
        reset_new_task_type = "RESET_NEW_WEEK_TASK";
        add_task_type = "UPDATE_WEEK_TASK";
      } else if (this.props.currentAnnotation === "month") {
        task_id = `month-task-${uuidv1()}`;
        reset_new_task_type = "RESET_NEW_MONTH_TASK";
        add_task_type = "UPDATE_MONTH_TASK";
      }

      let new_task = Map(this.props.task_data);
      let category_id = new_task.get("category");

      let category_quantity = OrderedMap(this.props.categories).getIn([
        category_id,
        "quantity"
      ]);

      // Check if the category belongs to default ones (based on the free plan's limitations).
      // For example, Free plan allows 5 categories to be used, then the first 4 created categories
      // will be the default ones (including Inbox). Thus, the default tasks, which will be valid when they are
      // created in side those categories.

      // default number of free categories.
      let free_number_of_categories = Map(this.props.generalSettings).getIn([···
      ]);

      let category_index = 0;

      OrderedMap(this.props.categories)
        .keySeq()
        .every((key, index) => {···
        });

      let assigned_plan = "free";

      // If the category belongs to one of the defaults
      if (category_index + 1 <= free_number_of_categories) {···
      } else {···
      }

      // Check if plan is free or premium
      let number_of_tasks_per_category = Map(this.props.generalSettings).getIn([···
      ]);

      if (category_quantity < number_of_tasks_per_category) {
        let new_task_with_id = Map(this.props.task_data).asMutable();
        new_task_with_id.update("id", value => task_id);
        new_task_with_id.update("title", value =>
          this.props.title_value.trim()
        );
        new_task_with_id.update("description", value =>
```

Figure 44 Implementation of adding a task - part 1



```
if (category_quantity < number_of_tasks_per_category) {
  let new_task_with_id = Map(this.props.task_data).asMutable();
  new_task_with_id.update("id", value => task_id);
  new_task_with_id.update("title", value =>
    this.props.title_value.trim()
  );
  new_task_with_id.update("description", value =>
    this.props.description_value.trim()
  );
  new_task_with_id.update("type", value => this.props.currentAnnotation);
  new_task_with_id.update("created_at", value => Date.now());
  new_task_with_id.update("plan", value => assigned_plan);
  let priority_value = Map(this.props.task_data).getIn(["priority", "value"]),
    priority_data = fromJS({id: task_id, category: category_id});
  let sending_obj = {
    add_task_data: {
      type: add_task_type,
      keyPath: [task_id],
      notSetValue: {},
      updater: value => new_task_with_id.toMap()
    },
    category_data: {
      keyPath: [category_id, "quantity"],
      notSetValue: {},
      updater: value => value + 1
    },
    reset_new_task_type,
    priority_data: {
      keyPath: [priority_value, "tasks"],
      notSetValue: [],
      updater: tasks => List(tasks).push(priority_data)
    },
    return_correspond_created_task: {
      type: "RETURN_CORRESPOND_TO_CREATED_DAY_TASK",
      data: new_task_with_id.get("schedule").toMap()
    },
    update_task_type_created: {
      data: "day"
    }
  };
  if (this.props.currentAnnotation === "week") {
    sending_obj.return_correspond_created_task.type =
      "RETURN_CORRESPOND_TO_CREATED_WEEK_TASK";
    sending_obj.update_task_type_created.data = "week";
  } else if (this.props.currentAnnotation === "month") {
    sending_obj.return_correspond_created_task.type =
      "RETURN_CORRESPOND_TO_CREATED_MONTH_TASK";
    sending_obj.update_task_type_created.data = "month";
  }
  this.props.addTaskThunk(sending_obj);
  Keyboard.dismiss();
  this.props._closeAddTaskPanel();
} else {
  this.props._toggleShouldDisplayPremiumAd();
}
} else {
  Keyboard.dismiss();
  this.props._closeAddTaskPanel();
}
```

Figure 45 Implementation of adding a task - part 1

Creating a task in Quint is the most important requirement. Above are two figures containing the shorten code in action. To be able to examine the implementation clearly, I will explain the workflow and ideas in each line separately.

At the line **125**, this is the start of the class containing the button component for confirming a task creation in Quint. By following the syntax of defining a class, the app is using

the traditional way of **React**. At the time writing the thesis, React has introduced a new way of writing code with Hooks. The function **_createTask** at line **126** is the adding task function which will be invoked when a user clicks on the button. At the line **127**, the function starts with a conditional statement block in order to allow executing the task creation only when the task has a tittle (not an empty string). From the line **128** to the line **140**, the function determines the correct type of the creating task to be created. There are 3 task types in Quint, which are Day, Week and Month. Therefore, the function initializes necessary variables with the Day type, then if in case the creating task has a type of Week or Month, it then changes those variables accordingly. The variable **task_id** is the task's id using **uuidv1()** method to create a unique random string. The variable **reset_new_task_type** declares the action type of the resetting-temporary-creating-task-object action as every action in **Redux** needs a type so that associated reducers can distinguish an action in order to update the state tree correctly. The variable **add_task_type** is the action type of the action of creating a new task. The variable **new_task** in the line **142** holds the temporary task Map-converted data from the parental property received from the connector in Figure 43. The variable **category_id** is the category's id retrieved from the temporary task data. The variable **category_quantity** at the line **145** gets the quantity of belonged tasks of the mentioned category. This variable is used in order to increase the number of tasks in the category in case the function successfully creates the task. From the line **156** to **210**, the function implements the code for determining whether it should perform the task creation based on the usage of a user. The rest lines show how Quint performs the task creation process. To immutably add the task to the corresponding task database, a clone of the creating task (or temporary task) is created. The next lines of code are the process of updating that copied Map and finally the clone is the one to be added into the database.

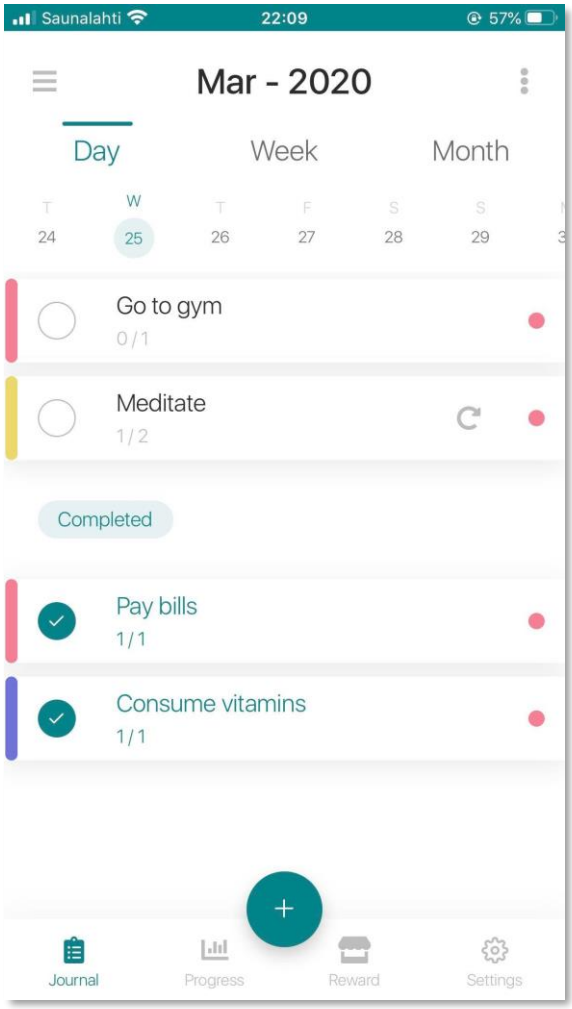


Figure 46 Result of adding Day tasks

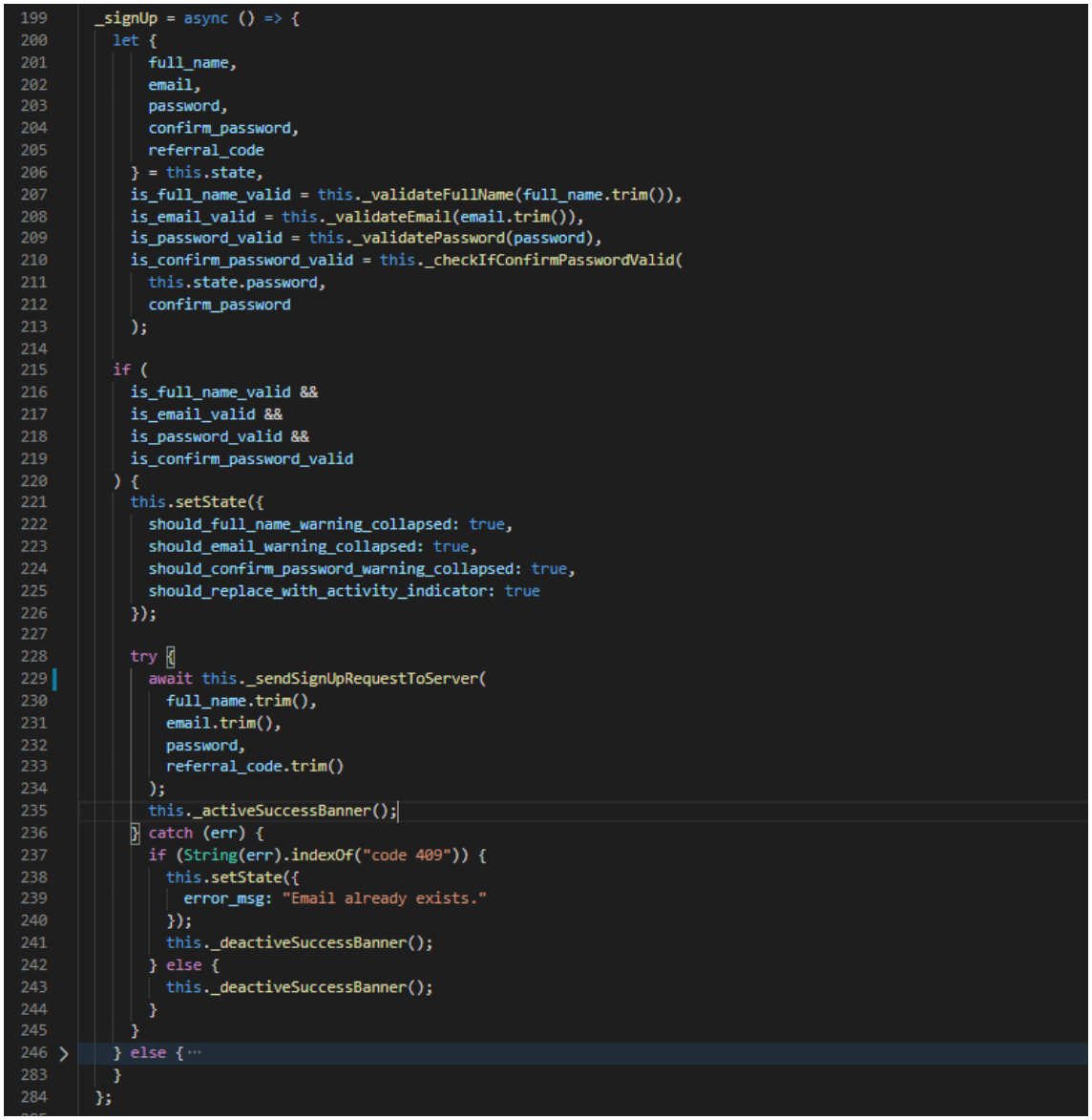
Above is the screenshot of adding tasks in Quint. I have added 4 Day tasks with different titles, goals, repetitions and priorities. On the right most side of each task, there is a red circle indicating the category that the task belongs to. In this case, all tasks belong to the default category “Inbox”. On the left most side of each task, there is a coloured vertical bar indicating the priority of each. The “Do first” priority is red, the “Plan” priority is yellow, the “Delay” priority is purple, and the “Delegate” priority is grey. Under each title there is the goal of the task. As you can see, the goal for the task “Mediate” is 2 and the task is completed once. Therefore, it is not counted as “completed” and there is an undo symbol near the category circle indicator on the left of the task.

Because the source code of each requirement's implementation has too many lines and requires a lot of words to explain, I will deliver the most crucial requirement in Quint which is the creating-a-task implementation shown above. Nonetheless, the remained requirements have the similar workflow and logic so to understand the implementation of creating a task is to understand almost all existing code in Quint.

5.2 The Back-end side

In this section, the thesis discusses about the sign-up implementation in Quint. There will be figures for in used code and the results of the work.

5.2.1 Client-side account registration

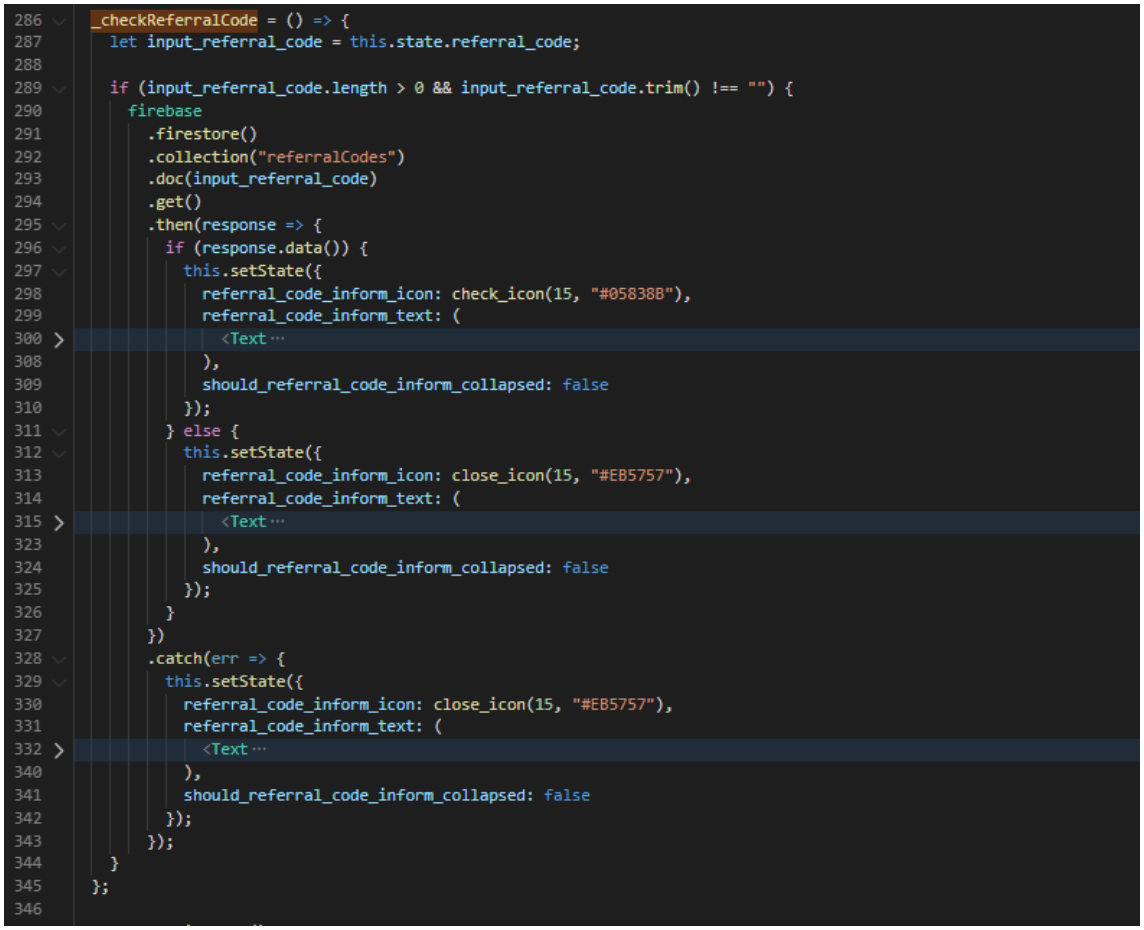


```
199  _signUp = async () => {
200    let {
201      full_name,
202      email,
203      password,
204      confirm_password,
205      referral_code
206    } = this.state,
207    is_full_name_valid = this._validateFullName(full_name.trim()),
208    is_email_valid = this._validateEmail(email.trim()),
209    is_password_valid = this._validatePassword(password),
210    is_confirm_password_valid = this._checkIfConfirmPasswordValid(
211      this.state.password,
212      confirm_password
213    );
214
215    if (
216      is_full_name_valid &&
217      is_email_valid &&
218      is_password_valid &&
219      is_confirm_password_valid
220    ) {
221      this.setState({
222        should_full_name_warning_collapsed: true,
223        should_email_warning_collapsed: true,
224        should_confirm_password_warning_collapsed: true,
225        should_replace_with_activity_indicator: true
226      });
227
228      try {
229        await this._sendSignUpRequestToServer(
230          full_name.trim(),
231          email.trim(),
232          password,
233          referral_code.trim()
234        );
235        this._activeSuccessBanner();
236      } catch (err) {
237        if (String(err).indexOf("code 409")) {
238          this.setState({
239            error_msg: "Email already exists."
240          });
241          this._deactiveSuccessBanner();
242        } else {
243          this._deactiveSuccessBanner();
244        }
245      }
246    } else {···
283    }
284  };
```

Figure 47 Code snippet of client-side signing up function

In the figure 47, the code snippet explains how the client-side collects user information and process them. At the start, the function **_signUp** is an asynchronous function, which has the ability to wait and execute related code (event-based) without blocking the JavaScript thread. By using this function, the app continues to execute other code in line but still resolves the incoming response from a http request. The function firstly stores user inputs such as full names, emails, and passwords from the line **200** to the line **206**. After

that, it runs several validations on the received full name, email and password from the line **207** to **213**. If all inputs are valid, the function sends a POST http request to the server containing all necessary user data. It is important to note that the server's url must be secured (https) in order to work in iOS and Android as the providers have updated their policies recently. From the line **228** to **245**, the function carries the request sending by applying a try-catch block in order to handle any possible errors. If the request is sent successfully the function will execute the method **_activeSuccessBanner**, which prompts an informative dialog about a verification email has been sent to the registered email. If the request fails, the function call **_deactiveSucessBanner** to inform the occurred error.



```
286  _checkReferralCode = () => {
287    let input_referral_code = this.state.referral_code;
288
289    if (input_referral_code.length > 0 && input_referral_code.trim() !== "") {
290      firebase
291        .firestore()
292        .collection("referralCodes")
293        .doc(input_referral_code)
294        .get()
295        .then(response => {
296          if (response.data()) {
297            this.setState({
298              referral_code_inform_icon: check_icon(15, "#05838B"),
299              referral_code_inform_text: (
300 >              <Text···
308              ),
309              should_referral_code_inform_collapsed: false
310            });
311          } else {
312            this.setState({
313              referral_code_inform_icon: close_icon(15, "#EB5757"),
314              referral_code_inform_text: (
315 >              <Text···
323              ),
324              should_referral_code_inform_collapsed: false
325            });
326          }
327        })
328        .catch(err => {
329          this.setState({
330            referral_code_inform_icon: close_icon(15, "#EB5757"),
331            referral_code_inform_text: (
332 >            <Text···
340            ),
341            should_referral_code_inform_collapsed: false
342          });
343        });
344    }
345  };
346
```

Figure 48 Code snippet for validating the input referral code

The function **_checkReferralCode** is used to validate the input coupon that a user gives. Normally, a user does not need to fill the field. He/she can leave the field empty and still proceeds the account registration. In case the user inputs some things, the function

_checkReferralCode will be invoked to validate the coupon with **Firestore**. From the line **290** to **295**, the client-side Firestore API is used to establish the connection with the database. Specifically, the database (or collection) has the name of "referralCodes". In the database, there are tables (or documents). Each table or document has an identifier to be a randomly unique generated referral code. The function calls the GET query from Firestore and Firestore returns a promise. From the line **295** to **343**, the code resolves the response from the promise if everything works out. Otherwise, it handles the rejection or error returned.

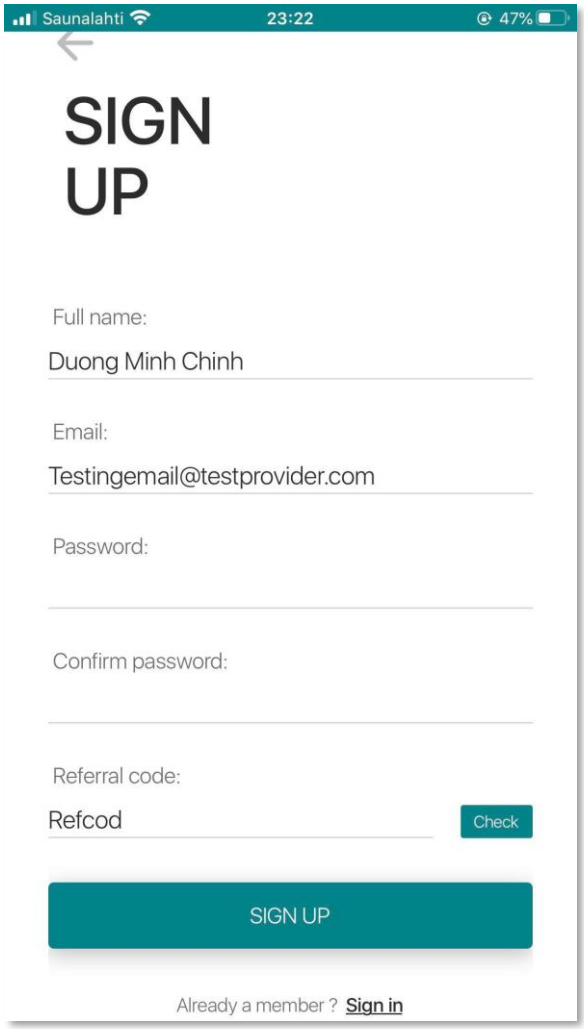


Figure 49 Screenshot of sign-up screen

The above figure displays the sign-up screen in Quint. Clearly shown in the picture, the user has filled all necessary information such as his name, email (fake one), password and

confirmed password. Because this screenshot was taken in the real device (iPhone 7 Plus) with is ability to capture the screen, the password-related fields are whitened out due to privacy policy of Apple. In addition, the user also inserted a referral code. Certainly, this code is false and used for the demonstration purpose only.

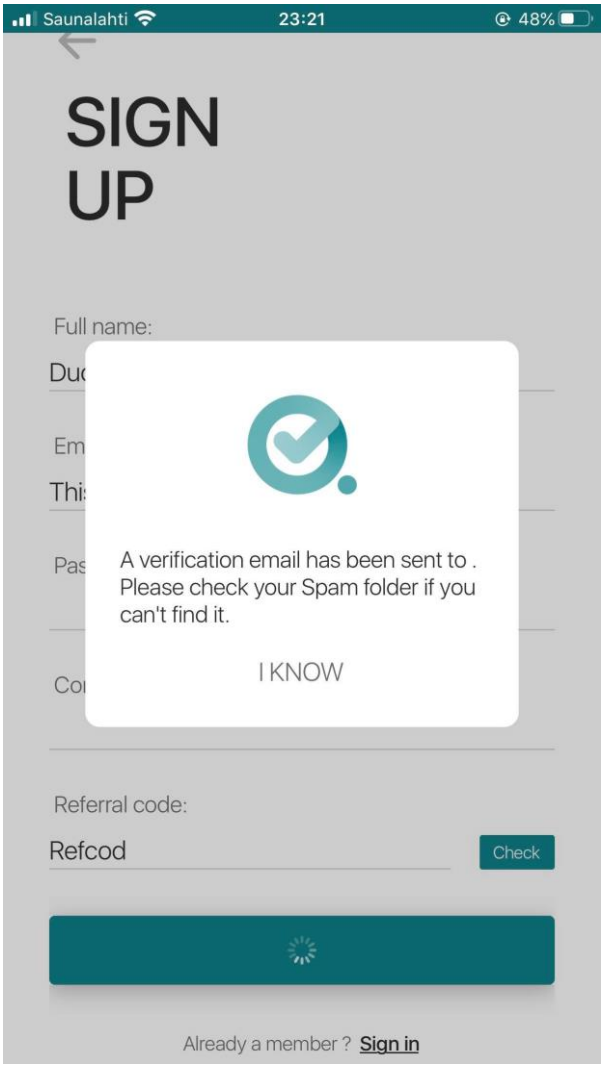


Figure 50 Email verification sent dialog

The dialog is shown because the account registration succeeded. Apparently, I used a made-up information to register an account. However, all register-related data will be processed in the server and the server will send an actual email verification to the registered email. Due to that action, the fake account cannot be used to login as it is not verified

by me (of course it is a fake one so I cannot activate the account) leading to the fact that it is a security method for ghosting or uncontrolled accounts in Quint.

5.2.2 Server application structure

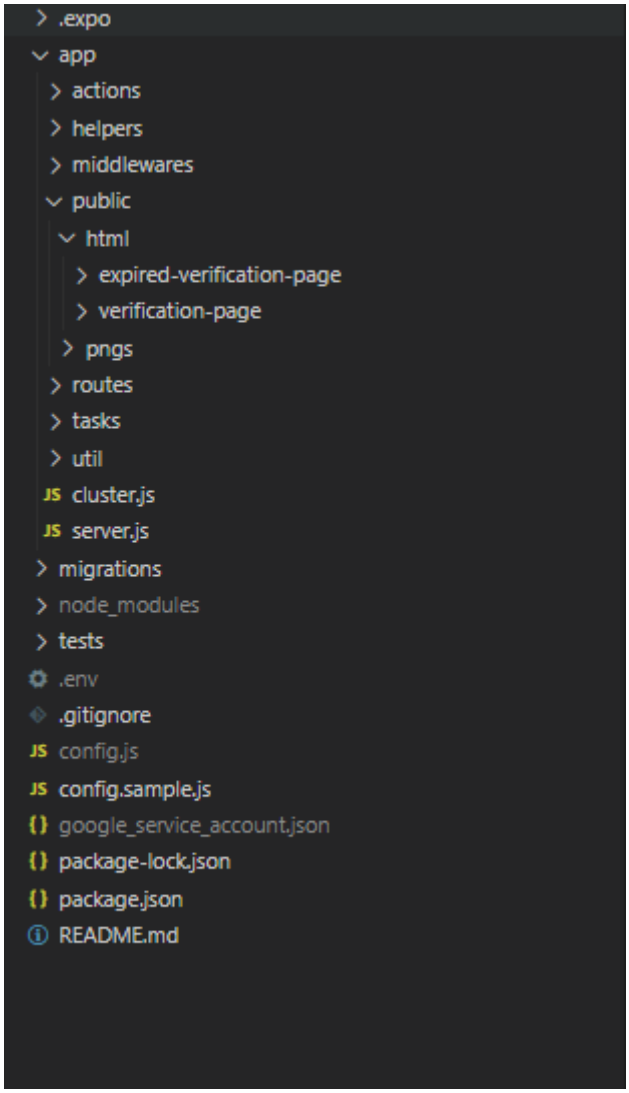


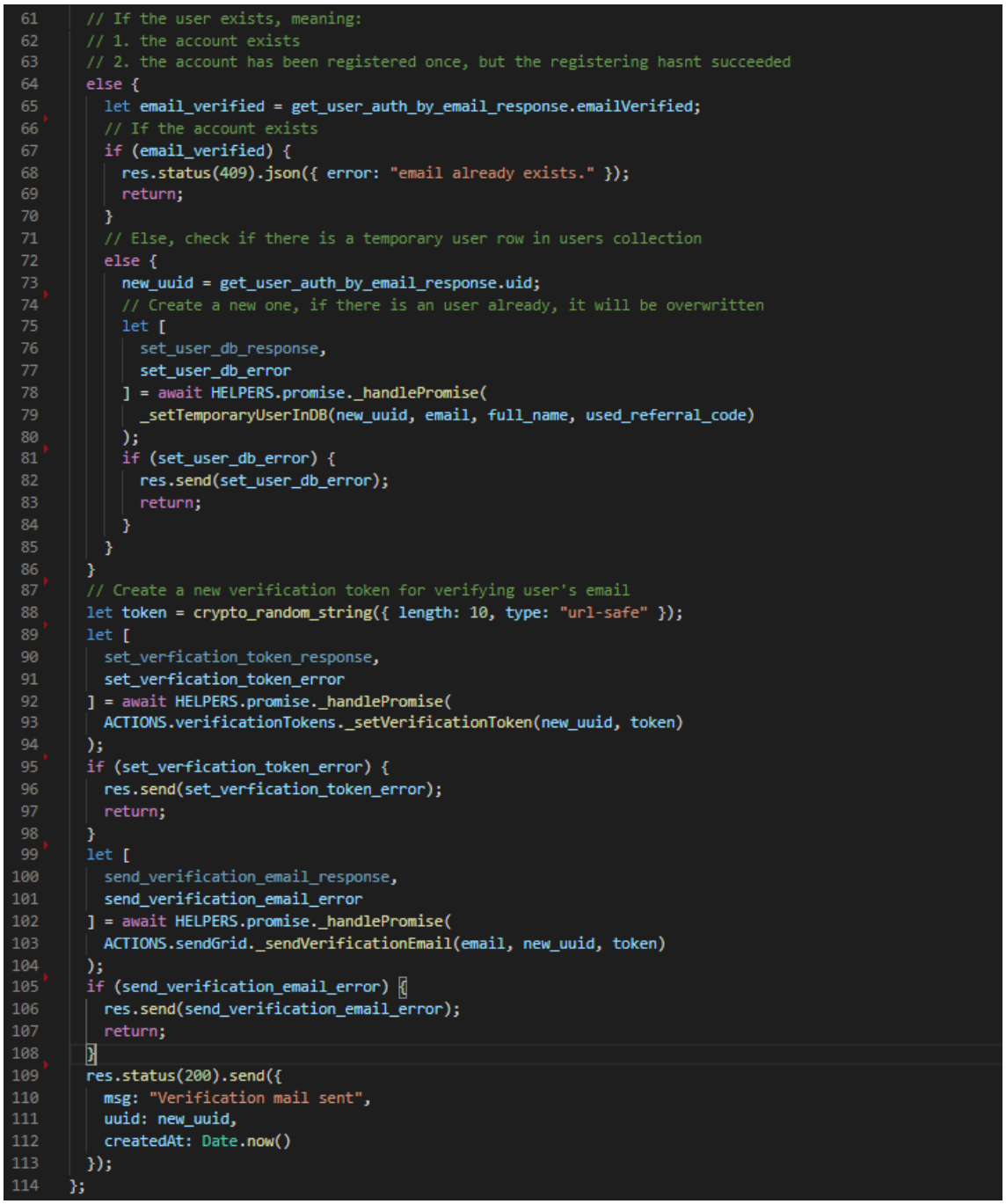
Figure 51 The structure of the server

The server's application structure follows a structuring paradigm called "Fractual" design. The structure helps the code base easier to read and look clean. It enables a quick on-boarding procedure for new developers since each feature has its own "Fractual" structure.

5.2.3 Send email verification

```
const _processTemporarySignUp = async (req, res, next) => {
  const { email, password, used_referral_code, full_name } = req.body;
  const [
    get_user_auth_by_email_response,
    get_user_auth_by_email_error
  ] = await HELPERS.promise._handlePromise(
    ACTIONS.auth._getUserAuthByEmail(email)
  );
  let new_uuid;
  // If the user doesnt exist, we create one in userpool and one in users collection.
  if (!get_user_auth_by_email_response) {
    // Create in userpool first to retrieve uuid
    let [
      create_user_auth_response,
      create_user_auth_error
    ] = await HELPERS.promise._handlePromise(
      ACTIONS.auth._createUserAuth(email, password)
    );
    if (create_user_auth_error) {
      res.end(create_user_auth_error);
      return;
    }
    new_uuid = create_user_auth_response.uid;
    // Create user in users collection
    let [
      set_user_db_response,
      set_user_db_error
    ] = await HELPERS.promise._handlePromise(
      _setTemporaryUserInDB(new_uuid, email, full_name, used_referral_code)
    );
    if (set_user_db_error) {
      res.end(set_user_db_error);
      return;
    }
  }
```

Figure 52 Code snippet for sign-up route handler – part 1



```
61    // If the user exists, meaning:
62    // 1. the account exists
63    // 2. the account has been registered once, but the registering hasnt succeeded
64    else {
65      let email_verified = get_user_auth_by_email_response.emailVerified;
66      // If the account exists
67      if (email_verified) {
68        res.status(409).json({ error: "email already exists." });
69        return;
70      }
71      // Else, check if there is a temporary user row in users collection
72      else {
73        new_uuid = get_user_auth_by_email_response.uid;
74        // Create a new one, if there is an user already, it will be overwritten
75        let [
76          set_user_db_response,
77          set_user_db_error
78        ] = await HELPERS.promise._handlePromise(
79          _setTemporaryUserInDB(new_uuid, email, full_name, used_referral_code)
80        );
81        if (set_user_db_error) {
82          res.send(set_user_db_error);
83          return;
84        }
85      }
86    }
87    // Create a new verification token for verifying user's email
88    let token = crypto_random_string({ length: 10, type: "url-safe" });
89    let [
90      set_verfication_token_response,
91      set_verfication_token_error
92    ] = await HELPERS.promise._handlePromise(
93      ACTIONS.verificationTokens._setVerificationToken(new_uuid, token)
94    );
95    if (set_verfication_token_error) {
96      res.send(set_verfication_token_error);
97      return;
98    }
99    let [
100     send_verification_email_response,
101     send_verification_email_error
102   ] = await HELPERS.promise._handlePromise(
103     ACTIONS.sendGrid._sendVerificationEmail(email, new_uuid, token)
104   );
105   if (send_verification_email_error) {
106     res.send(send_verification_email_error);
107     return;
108   }
109   res.status(200).send({
110     msg: "Verification mail sent",
111     uuid: new_uuid,
112     createdAt: Date.now()
113   });
114 };
```

Figure 53 Code snippet for sign-up route handler – part 2

Figure 52 and 53 shows the code implementation of how Quint handles the account registration sent from the client side. Firstly, from the line **27** to **86**, the function validates the existence of the registered email in Firestore. There are two cases. The first case is that the registered email does not exist in the database. The execution part of this case

starts from line **35** to **59**. In this situation, because of using **Firebase Authentication** to manage user accounts and **Firestore** to store user account data, it is mandatory to update those two places with provided user data. The method **ACTIONS.auth._createUserAuth** handles the user account creation in the Authentication user pool so that Firebase can gain controls over registered accounts. The method **_setTemporaryUserInDB** creates a document or a table containing requested user data in the collection/database "users" in Firestore. The second case is that whether the registered email already exists (verified) or it has been registered before but has not yet been verified. From the line **65** to **70**, the code handles the scenario that the email exists, in which the server responses an error request. From the line **72** to **84**, there is a repletion of the method **setTemporaryUserInDB** in order to create a moment user collection in the database for further usages. Moving next in the code, from the line **88** to **98**, the server creates a cryptographically strong random string as the verification token for the account by using the method **crypto_random_string**. As being said, this token will expire in 24 hours starting from the time it is saved to the database by the method **ACTIONS.verificationTokens._setVerificationToken**. The remained code invokes the function **_sendVerificationEmail** to send an email verification to the registered email. In addition, the route ends by send back to the client a response saying that the verification email has been sent.

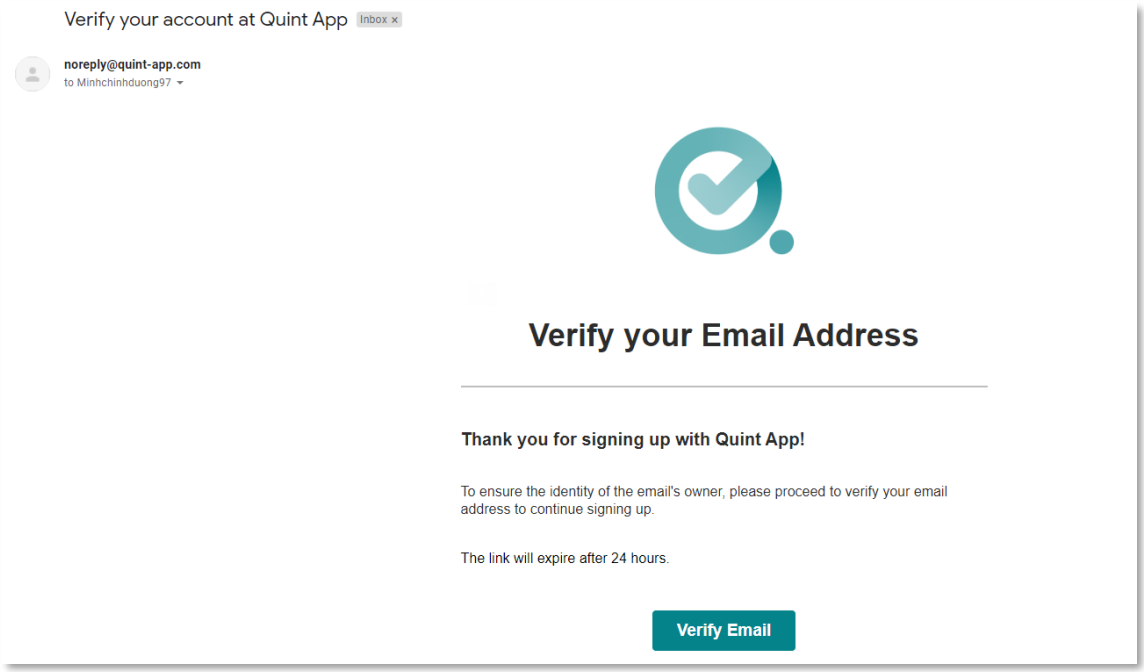


Figure 54 Screenshot of the account verification email

By clicking the “Verify Email” button, the registered account is verified and able to login into the Quint app. If in any cases that a user forgets to verify the account for over 24 hours, the email is still there but clicking the button results in a redirected website saying that the verification token is expired and he/she must register the account again.

6 CONCLUSION

The thesis was conducted with the goal of building a mobile application with React Native and providing an innovative to-do application that fits many levels of user dedications.

From a personal point of view, the designing phase with database schemas and diagrams was done well by spending time to research ideal approaches. Furthermore, the technology choices were proved to be working well since the author had firm experience of relevant tech stacks. There were many difficulties emerging in the project due to little experience and knowledge in mobile development. As the time went by, the difficulties were solved one by one with the help of online developer communities, as well as countless open source code in GitHub.

The thesis has many potentials and it can be improved greatly with future work. There were unimplemented features, which were planned at the beginning and along with the development time. The app can be applied with TypeScript in order to strongly type the variables leading to better developing experience. Tests were being applied partly in the application. However, to have automated testing implemented will be a must-have advantage in term of product quality management.

To summarize, the thesis has done explained, not all everything, but the most important material about Quint at its very first-stage of beta testing phase.

REFERENCES

React Native: https://reactnative.dev/

Expo: https://expo.io/

Firebase: https://firebase.google.com/

Stack Overflow: https://stackoverflow.com/

Heroku for hosting the server: https://www.heroku.com/

GitHub: https://github.com/

Apple Developer Documentation: https://developer.apple.com/documentation/

Google Developer Documentation: https://developer.android.com/docs

Font Awesome Icons: https://fontawesome.com/

Figma for UI frames: https://www.figma.com/

ExpressJS: https://expressjs.com/

NodeJS: https://nodejs.org/

Namespace Pollution Mechanism (NPM): https://www.npmjs.com/