



# *CT5118 Software Design and Development*

## *Project*

Mobile Study Application built using React Native

***Students:*** Ethan Kane and Richard O'Connell

***Supervisor:*** Karen Young

***1SD1:*** Higher Diploma in Software Design and Development

# Table of Contents

1. Introduction .....	3
1.1 Background .....	3
1.2 Project/App goals.....	3
1.3 Planned Application Program Flow.....	4
1.4 Functional and Non-Functional Requirements .....	4
1.4.1 Functional requirements.....	4
1.4.2 Non-Functional Requirements.....	5
1.4.3 Stakeholders .....	5
1.5 Use Case Diagram .....	5
2. Technologies .....	6
2.1 React .....	6
2.1.1 React Components.....	6
2.1.2 The Virtual DOM .....	7
2.1.3 JSX .....	7
2.2 React Native .....	8
2.2.1 React Native concepts.....	10
2.2.2 Styling in React Native .....	12
2.2.3 React Native Navigation.....	13
2.2.4 Redux .....	14
2.3 Expo.....	16
2.4 NodeJS and NPM.....	16
2.5 Firebase .....	17
3. Application/Project Implementation .....	19
3.1 Setting up development environment.....	19
3.2 Detailed System Design and Implementation .....	21
4. Testing.....	50
4.1 Black Box Testing .....	51
5. Challenges .....	67
6. Conclusion.....	68
References .....	69

# 1. Introduction

## 1.1 Background

In the modern world, the ubiquitous access to information through the use of mobile applications is one of the leading sectors within the information technology sector. According to Statistica, there are currently 3.5 billion smartphone users worldwide which equates to nearly 45% of the world population access to a smartphone [1]. As of March 2020, Android's app marketplace (Google play store) has around 2.8 million applications, whereas the iOS app store (Apple app store) has roughly 2.2million applications. All of this shows the relevance of mobile development in today's modern digital society. The software industry has adopted a mobile first approach to software development whereby the development of high-quality user interfaces is primarily focused on mobile devices and the user experience on these devices. In software today a poor mobile experience can have significant negative impacts on stakeholders' brand.

Over the past 10 years, new technologies have arisen to help deliver projects that do not have adequate resources or requirements to maintain and develop applications for separate platforms, primarily Android and iOS. Cross-platform technologies enable the ability to use one environment or programming language with a shared code base that can run on several platforms whether that is Android, iOS or the web. The single shared code base can facilitate the development of business logic and maintenance of the app.

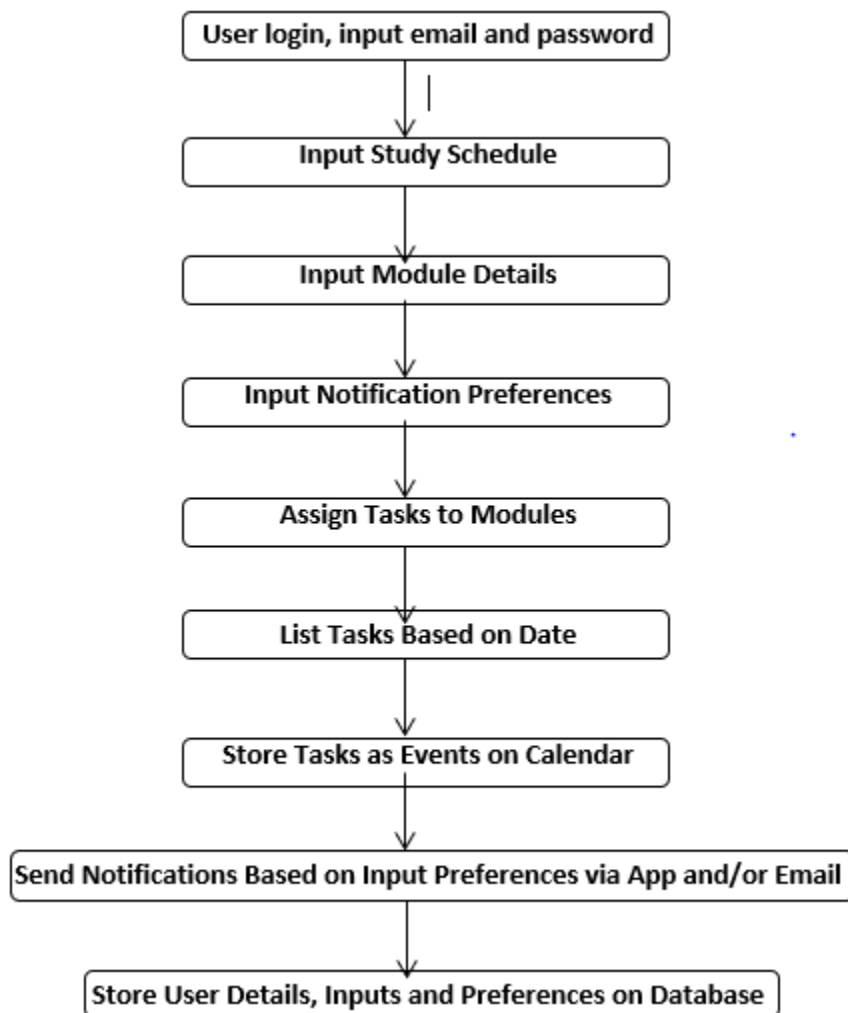
One of the most popular technologies that can assist in achieving cross-platform development is React-Native, which is an extension of React which is a web development library framework also referred to as React.js or ReactJS. React Native is written in JavaScript and aims to allow the use of native functionalities and integration across android and iOS platforms. React native is continuously evolving and establishing itself to be one of the best options for software developers and engineers with experience in react or JavaScript and would like to delve into native mobile application development [2].

## 1.2 Project/App goals

The aim of this project is to develop a mobile application using the React Native framework that could help users manage their university study schedule and have the ability to customise notifications and store documents as required. The goal was to create a simplified process for students to store and manage their schedule as best suits their needs.

By developing using React Native we are enabling the ability for the application to be used on either the android or iOS platform.

### 1.3 Planned Application Program Flow



### 1.4 Functional and Non-Functional Requirements

#### *1.4.1 Functional requirements*

The App must load when opened

The app must allow a user to create a profile

The app must allow the user to sign in after they have created a profile

The app must allow the user to upload an assignment

The user must be able to view the assignment and it must persist even after the user has signed out

The user must be able to set up push notifications to remind them to study

The user must be able to set specific times when a push notification is sent

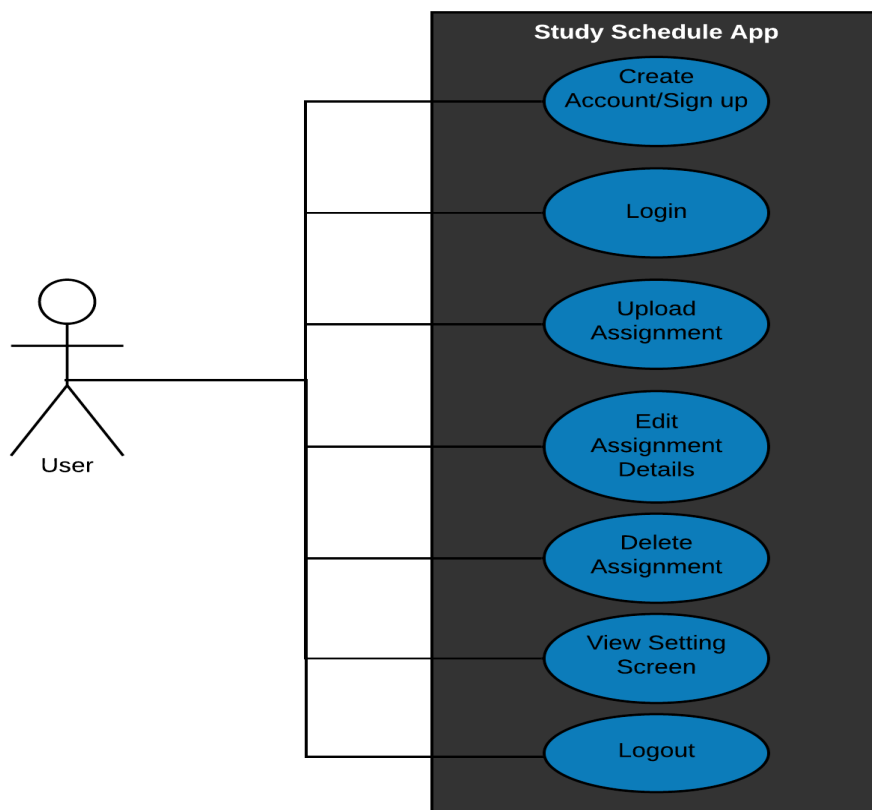
#### ***1.4.2 Non-Functional Requirements***

The app must have a clean easy to navigate User experience

#### ***1.4.3 Stakeholders***

Students, professors, teaching assistants, University Admin, software professionals, domain experts

### **1.5 Use Case Diagram**



## 2. Technologies

### 2.1 React

There are multiple strategies available at present in contemporary web development for handling data representations on an architectural level. One of the most recognized method is the model view controller (MVC), which associates a data model with a view that is being handled by a controller. As Soon as the model data changes, the controller re-renders the view with the new updated data [3]. React is an open source web framework/library which was developed by Facebook to resolve challenges routinely encountered when developing sizable user interfaces on the web that change over time. These large-scale interfaces are very prominent across the web today with the increasing popularity of social networking sites with a lot of moving parts such as Facebook and Instagram [2]. React takes its stance in controlling the view layer for example in a MVC written application and the user interfaces are designed such that they are constructed out of numerous, declarative components that each have their own state [4]. To be precise, the UI of the application is assembled by these components that each have access to their own data state supplied by the model. In conjunction with the virtual DOM, the components allow React to listen for changes and re-render a particular part of the UI instead of the entire view [5].

#### **2.1.1 React Components**

So as to build a scalable UI, React views are composed of multiple reusable components [6]. Through encapsulating functionality up into distinct components, the splitting up of concerns becomes much easier which facilitates code reusability, easier testing and for the combination of various components into brand-new ones [4]. The advantages and reasoning for increasing modularity of applications was summed up in the book, *The Art of Unix Programming* by Eric Raymond where he wrote,

*“The only way to write complex software that won’t fall on its face is to hold its global complexity down, to build it out of simple parts connected by well-defined interfaces, so that most problems are local and you can have some hope of upgrading a part without breaking the whole”*

This is specifically the tactic React uses in resolving the troubles associated with intricate user interfaces. When Facebook was developing ReactJS, they did not build a full MVC architecture to displace existing frameworks. Each React component has their own `render()` function which in turn classifies the UI of that particular component. In Addition, each component has access to its individual state which specifies what data is currently, being supplied by the model. As a result of this tightly coupled relation between the `render()` function and state, it allows React components to detect changes in data and prompt re-rendering of instance data [5].

### *2.1.2 The Virtual DOM*

The Document Object Model (DOM), is a tree like structure for the display of a web page, which allows one to traverse the tree and update or modify any node of the tree. For instance, normally these updates on the web are made using JavaScript. Yet, lots of modern web pages are developed as single page applications or pages which dynamically adjust without sending the user to another URL, hence making these trees considerably larger. These dynamic web apps also require the DOM tree to be updated frequently and promptly which can become tricky as the tree grows [2,5]. For example, scrolling through a web app such as Twitter or Facebook users can simply fill the DOM tree with thousands of nodes. In order to remove one of these nodes it would now mean we would have to traverse the whole tree to locate that specific node. To deal with these issues related to the DOM, React utilizes something known as the Virtual DOM which is a collection of libraries, techniques and algorithms that are used to effectively re-render selected elements of a tree structure through generating a lightweight JavaScript copy of the DOM [4,7]. As an Alternative to working with the DOM directly, an abstract version (Virtual DOM) of it is created and all subsequent alterations are made to this lighter weight copy. As changes are made to the Virtual DOM it is then compared to the initial DOM to locate the differences between nodes, and these nodes can be carefully chosen and updated. React is able to determine the variations between nodes using a specific algorithm along with batching the DOM read and write operations, enabling the sub-trees to be updated effectively [7]. Through applying this technique, the rendering speed is significantly increased, as for each time an update needs to be made, the diffing-algorithm determines the minimum amount of changes required to update the concrete DOM [7].

### *2.1.3 JSX*

JSX is a HTML/XML like syntax extension to JavaScript that extends and enhances ECMAScript standard which is used by React. It is intended to be utilized by preprocessors (i.e. transpilers such as Babel) to convert HTML/XML text found in JavaScript files into traditional JS objects that can be parsed by a JS engine [8]. JSX is made out of components and elements. Components are references to corresponding React components and elements are the JSX equivalent to ordinary HTML tags. Essentially, by employing JSX it enables the ability to write concise HTML/XML like structures such as DOM like tree structures in the same file as you write your JS code [8]. JSX helps with enabling Reacts code to maintain a high readability and reusability.

In React Native, just like in React, views are written in JSX, merging markup and the JavaScript that governs it into the one file. JSX was met with strong reaction by developers when React launched. For example, a lot of web developers are accustomed to keeping the separation of files based on certain technologies is

the norm whereas you keep the HTML, CSS and JavaScript files separate. The suggestion of mixing HTML markup, styling and control logic into a unified language can seem confusing. However, JSX prioritises the separation of concerns ahead of the separation of technologies. With React Native, this is much more strictly enforced. In an environment with no browser, it makes much more sense to merge our markup, styles and behavior into one file for every component. Hence, the JS files in React Native applications are in fact JSX files [9].

## 2.2 React Native

Following a positive experience with developing React, Facebook pioneered React Native to facilitate mobile development. In 2015, Facebook released React Native touting the philosophy of a library/framework that enables developers to learn once, write anywhere. This declaration implies that any developer that is able to create a web application with React will be able to build a cross platform mobile application without any additional learning. React Native boasts a large community with a lot of developers and third-party libraries readily available. From the statistics on Stack Overflow for 2019, it shows two cross platform mobile frameworks, React Native and Xamarin in the top 10 most used technologies [10]. However, React Native is almost twice as popular and in demand over Xamarin in terms of its users, commits and contributes on GitHub [11]. Furthermore, React Native is the second most contributed repository on GitHub since 2017 [12]. Thus, React Native is at the forefront of all the other frameworks in regard to community and support. Two studies [13,14], implies that applications created using React Native are hardly noticeably different from the native application. One explanation for this is that React Native utilizes the same underlying UI elements as native applications and developers put these elements all together using JavaScript to create a mobile application for both iOS and Android. As mentioned previously, plain old vanilla JavaScript is not actually used by developers to create the application UI, however instead the UI is made out of JSX which is an HTML/XML like syntax, that is converted into plain JavaScript at runtime. When the native UI elements are utilized and retrieved by React Native the outcome is a genuine look and feel native app. React Native applies this interpreted methodology, so that JavaScript/JSX is translated and interpreted to the systems native code at runtime. Despite the fact the compilation/translation happens by use of JIT (Just In Time), the native platform widgets are available and every UI component in RN is mapped to a native widget. Access to these native UI widgets, which is provided by a bridge, called the React Native Bridge which is shown in Figure !!! . The requests to the native platform APIs and their subsequent responses go across the bridge back and forth and is accessed up to 60 times per second to make communication seamless. Even Though, the notion



behind the bridge is fascinating, this remains to be considered one of the major problems with using React Native, as the bridge is a bottleneck and can lead to some performance issues [15].

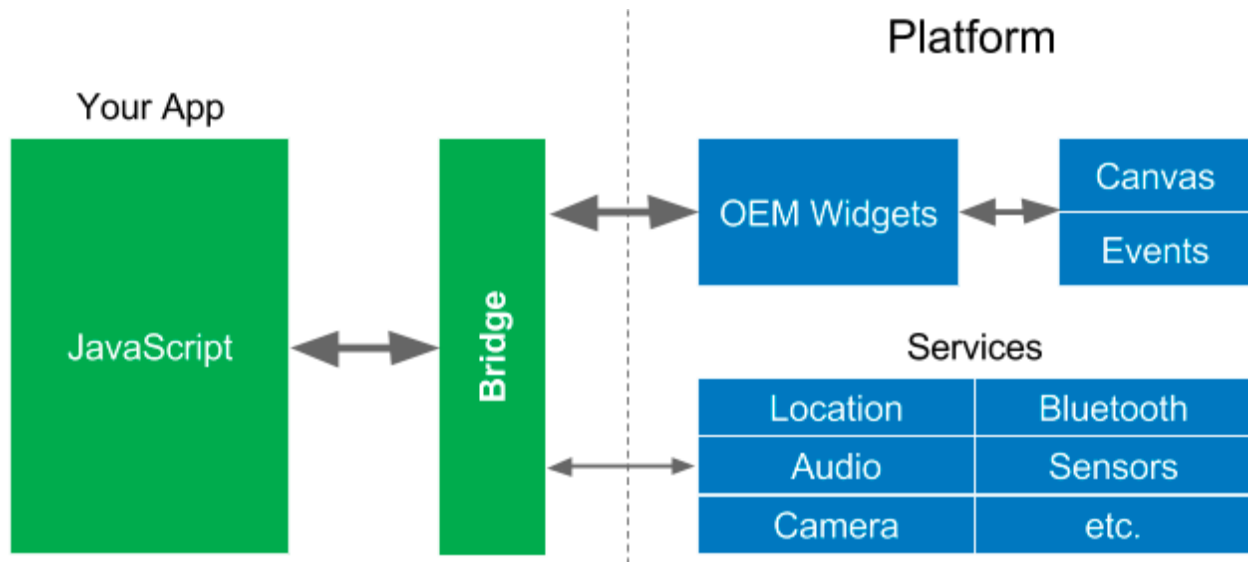


Figure 1 React Native Bridge Architecture [32]

As Figure 1 indicates, there are two separate regions in the application programmed with React Native. The first is the native region where all of the UI and views are rendered, the second is the JavaScript region where the constraints are put on the UI involving the application logic and styling. These two regions are linked via the bridge. React Native's architecture in this regard is quite similar to a Client-Server model where one could imagine these two regions occurring on different computers (mobile and desktop) and the bridge acts as the WebSocket between the two. Performance issues tend to occur when moving from the JavaScript region to the native region over the bridge. The native platform widgets are accessed via the bridge 60 times per second. To enable and maintain decent performance within React Native applications, the developer ought to keep the passes across the bridge to a bare minimum which requires a significant amount of effort. Using Virtual DOM method helps with minimizing the quantity of data that needs to be sent across the bridge, nonetheless, it is still not completely sufficient, signifying that developers should continuously be aware and optimize their code regularly in order to achieve and maintain acceptable performance. Navigation between screens, Animations, swipeable containers, and any additional user-interaction behavior is susceptible to performance bottlenecks. At times it remains extremely difficult to resolve these issues and there are articles available to address these concerns and

to help developers improve their code and to optimize performance when developing React Native apps. in order to improve React Native apps performance [15].

### *2.2.1 React Native concepts*

The data within React Components is handled by state and props. One of the ways in which data is generated and controlled within a React Native component is by manipulating state. Component state is asserted once the component is created, and its composition is a simple JavaScript object. State can be updated inside the component by using the `setState` function [16].

Alternatively, data can also be controlled by using props. Props, short for properties, are passed down as parameters once the component is built. However, unlike state, they cannot be updated inside the component.

State is a set of values that a component handles. React Native interprets UIs as simple state machines. As soon as the state of a component is changed by calling the `setState` function, React Native re-renders the component. If some of the child components are inheriting this state as props, subsequently all of the child components are re-rendered also. When creating an application using React Native, recognizing how state functions is essential for the reason that state governs how stateful components render and behave. A Components state is what enables developers to build components that are interactive and dynamic. The most important part to recognize when distinguishing between state and props is that state is mutable, while props are immutable [16].

Props are a component's inherited values that get passed down from a parent component. Props can either be static or dynamic values as soon as they are declared and initialized, except when they are inherited, they are immutable. The only way props can be modified, is by modifying the original values at the top level where they are first declared and passed down. In the official documentation from React, "Thinking in React" it explains the perception of props as "a way of passing data from parent to child." [17]. Table 1 illustrates several differences and similarities to differentiate between props and state [16].

Props	State
External data	Internal data
Immutable	Mutable
Inherited from a parent	Created in the component
Can be changed by a parent component	Can only be updated in the component
Can be passed down as props	Can be passed down as props
Can't change inside the component	Can change inside the component

*Table 1 Props vs. State*

Figure 2 illustrates how React Native functions and renders depending on the specific platform. In place of rendering to the browser DOM, React Native invokes Java APIs to render to Android components, or Objective-C APIs to render to iOS components. This is what differentiates React Native from other cross-platform application development options, which mostly take the approach of rendering web-based views. This is made possible because of the bridge, that provides an interface into the specific host platforms native UI elements. React components return JSX markup as of their render function, which defines the way they should appear. When using React for web development, this translates precisely to the browser's DOM. In React Native, this JSX markup is translated to conform to the host platform, so a `<VIEW>` could be converted into an iOS-specific UI View [9].

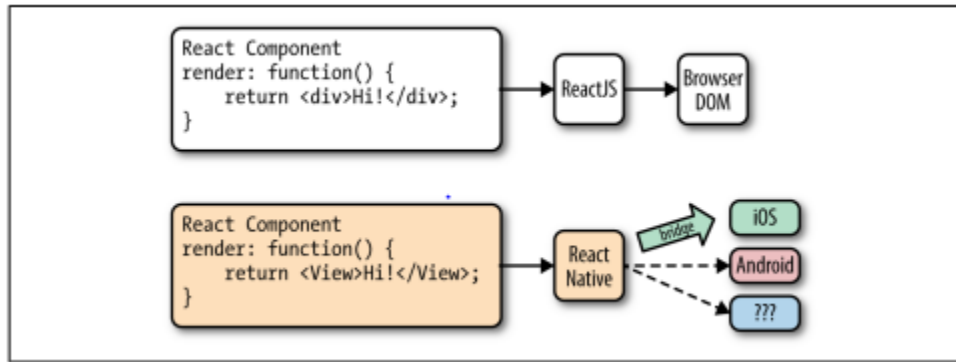


Figure 2 Example of React rendering to multiple platforms [9]

Table 2 shows some of the commonly used components which essentially create the UI of the application.

Component	Description
View	View is the most fundamental component in React Native for building user interfaces. View works as a container. It is often used to encapsulate other elements or components. To place elements in a specific area we place the components inside “View” tag.
Text	Text is another fundamental component in React Native. It is used for displaying text.
Image	Image is a component for displaying images. The image can be fetched from Uniform Resource Identifier or local source.
TextInput	TextInput is a component used for making text input field that let user input into the app through keyboards.
ScrollView	ScrollView is a scrolling container that helps users scroll the content if the content overflows the container.
FlatList	FlatList is an enhanced version of ScrollView. FlatList only renders the items that currently visible in the screen. Thus, FlatList makes a better performance than ScrollView and is a better choice if you need to render a long list.
StyleSheet	StyleSheet works similar to CSS. StyleSheet provides an abstraction layer that defines the styles of a user interface component. It’s how the component looks like.

Table 2 React Native basic components

### 2.2.2 Styling in React Native

With styling, React Native takes a unique approach, bringing styles exclusively into the realm of JavaScript and making developers link style objects clearly to components. This attitude managed to elicit a strong

reaction from developers, as it represented a major withdrawal from the CSS based styling norm that has been around since 1996 [18]. To better comprehend the model of React Native styling, first we ought to contemplate several of the problems related to the traditional CSS stylesheet approach [19].

Every one of CSS class names and rules are global in scope, signifying that imposing some styling on one component can simply break another if the developer is not cautious. In React Native, there is the opportunity to retain the advantageous elements of CSS, but additionally the liberty for significant divergence. React Native applies a subsection of the existing CSS styles, concentrating on retaining the narrow styling API whilst still being highly expressive. As a substitute to stylesheets, React Native uses JavaScript based style objects/components. A major strength of React Native is that it forces developers to keep their JavaScript code i.e. their components - modular. With styles being brought into the JavaScript realm, React Native makes developers write modular styles, as well [9].

Most React Native styling revolves around the use of `StyleSheet.create`. This styling function is a little example of syntactic sugar with a variety of additional perks. These `StyleSheets` are immutable and by creating them, rather than passing around raw JavaScript objects, it decreases the amount of allocations, thus helping the applications performance. It also encourages developers to organize and keep their code clean [9]. The styling is appended in the React Native JavaScript code with a call to `StyleSheet.create`.

An alternative way to mimic the “cascading” behavior of CSS, is through passing styles props/properties on a component. This pattern can be used to create more extensible components that be more efficiently controlled and styled through their parents [9].

Another big change and of significant importance in mobile app development, and styling React Native applications is positioning. CSS upholds an abundance of positioning methods such as absolute positioning, float, block layout, table and much more. React Native has a much more focused approach to positioning, relying heavily on flexbox in addition to absolute positioning, and properties such as padding and margin. Unlike existing positioning modes like block and inline, flexbox grants developers the ability to create direction-agnostic layouts [9, 20].

### ***2.2.3 React Native Navigation***

One of the most fundamental aspects of a fully functional mobile application is focused around navigation. Mobile apps normally are composed of several screens and allow for transitions between them. React Native does not include any navigation libraries to efficiently implement multiple screens and a routing mechanism. However, there are many third-party navigation libraries to control those transitions and

enable developers to express the relationships between screens. The most popular navigation library is React Navigation which is available from the React community GitHub project [9].

The React Navigation library is JavaScript based and all the controls and transitions are controlled by JavaScript. There are typically three main types of navigation used for mobile app development such as a stack, tab and/or drawer type of navigator [16].

The fundamentals of stack navigators, functions around moving from one screen to the next, substituting the current screen and typically applies an animated transition. This enables users to navigate forwards or backwards within the stack. Stack navigation can be thought of like an array of screen components. For example, pushing a screen component into the array brings the user to the new screen component. Whilst to go back to the previous screen, you pop off the last screen component from the stack [16].

The drawer navigator, functions as a side menu that typically is accessible from the right side of the screen and reveals a list of options. When an option is selected, the drawer closes, and the user is taken to the new screen [16].

Creating routes in React Native are built around components. The component is shown or loaded depending on the navigator that is being used, whether it be a stack, drawer or tab type or a combination of each, the routing will also vary. State and data need to also be managed throughout these routes. Typically, the Redux state management library is used to provide methods to control and manage the data. Along with using Redux it is best practice to use AsyncStorage to persevere the state so that if a user refreshes or closes the application, the data is still available [16].

#### **2.2.4 Redux**

When developing mobile applications in React Native, the data layer can become very complex and unmanageable if is not controlled correctly and purposefully. One of the most widely adopted techniques for handling data within a React Native app is through the Redux data architecture library. Redux was developed and is maintained by Facebook and from the official documentation, the library is portrayed as *“predictable state container for JavaScript apps”* [21].

Redux is essentially a state management library that is used for managing the flow of data within an application [9]. Redux creates a hierarchy for the applications state. The main advantage of Redux is the manner in which it controls data. It maintains a unidirectional data flow, which gives the developer a transparent and easy way to develop and debug applications. Redux is fundamentally a global state object that is the single source of truth within an application. In an React Native app each component has two

forms of data that controls it, state and props. The Redux global state object is received as props into React Native components and any point in time a portion of data is altered in the Redux state, the entire application takes this new data as props. Redux streamlines the app state by shifting it all into one place called a store. This allows developers, when they require the value of an object, they are able to know precisely where to look and can anticipate the same value to be accessible and up to date somewhere else in the app [16].

Figure 3 illustrates the Redux store and demonstrates its accessibility from all elements of the application and provides excellent visibility on just how the data in the application is structured [22].

As soon as a change occurs in the view (UI), an action is dispatched. Once actions have been dispatched the values in the redux store will change over. Actions include a payload object that is controlled by the reducer to decide how to update the state as a result. This change to the state is subsequently shown in the view, typically with a visual change to the components.

To summarize, Redux enables developers the ability to pass data and properties to children within the application without having to overtly pass the properties to each individual child. The Redux store holds the application state and can be changed via actions, which return action object from the container components. Actions are created to update the store and change the data being passed into a reducer. Reducers function within the store and do not modify their own arguments, they acknowledge the current state of the app and an action, that has been dispatched. They subsequently return the new state of the app, which can be a clone of the current state but in no way be a altered current state. The reducers are state machines which tell which state follows the state you are in, if the given action has been dispatched [16].

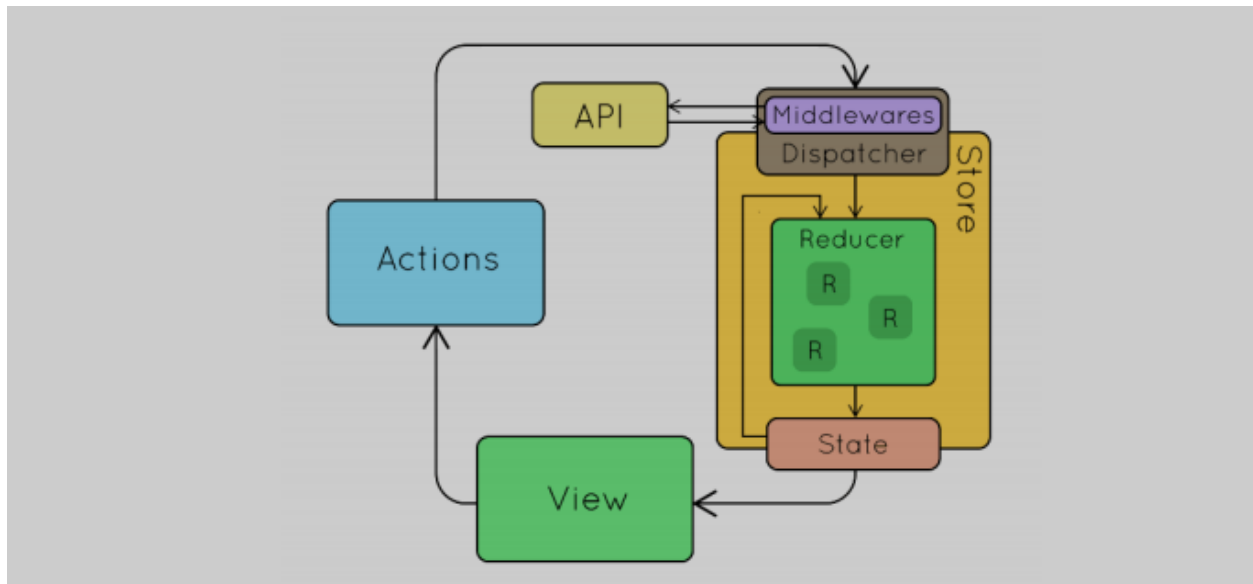


Figure 3 React Redux flow diagram [22]

## 2.3 Expo

Expo is a piece of software composed of libraries, tools and services that facilitate building native mobile application such as iOS and Android project using React Native and JavaScript. The applications that are written in Expo remain React-Native apps which additionally include the Expo Software Development Kit (SDK). The SDK is native code, and the library offers the ability to access the device's system specific functionality such as contacts, camera, GPS, local storage etc. That signifies that the use of Android Studio or XCode or having to write any native code is not required. The way in which Expo works is that it runs the JSX code and serves it up over a WebSocket so that the React Native application fetches the bundle and executes it, while at the same time also running the native code which is associated to the specific platform. With React Native applications being developed entirely in JavaScript/JSX, they can be ported and run on any native environment containing the Expo SDK. Additionally, it also offers User Interface (UI) components which carry out a range of use-cases that are not offered in the React Native core CLI projects such as the icon packs and blur view. Lastly, the Expo SDK gives access to services which are relatively difficult to manage by the majority of apps such as push notifications, managing assets and constructing native binaries that are ready to be published and deployed to the app store [16, 23].

## 2.4 NodeJS and NPM

NodeJS is cross platform, open source asynchronous JavaScript runtime environment that executes JavaScript on the backend outside of the web browser and is aimed at creating scalable network applications and applications with many input/output operations [24]. One of the major advantages that



has come from the development of Node.js is the built-in support for package management from the NPM package manager [25]. NPM offers the developer a set of reusable components which can easily be installed and used in React Native projects. It is one of the most prominent open source libraries available [25]. React Native utilizes NPM to handle project dependencies. The NPM registry contains packages for all kinds of JavaScript projects, not just Node. NPM utilizes a file called package.json to hold metadata about the project, as well as the list of all dependencies [9].

## 2.5 Firebase

Firebase offers real-time database that was developed and is cloud hosted by Google. This cloud database stores the data as JSON objects and is synchronized in real-time to all of the connected client (Android, iOS, web). With this real time database there is no requirement for an intermediary server application, it can be accessed directly from the client [26].

Data is additionally able to persist locally even if the client application is offline, which permits real-time events to be continuously generated, to signify the updated state of the data to the client. As soon as, a client device regains connection, Firebase syncs the changes in the local data with the remote updates that transpired whilst the client was offline, blending any differences automatically [26]. Firebase is a NoSQL database, developing a suitably configured database in Firebase involves planning on exactly how the data is going to be stored and later on retrieved, in order to benefit from the optimizations that Firebase provides for non-structured data. The real-time database API that is available in Firebase can be used to only allow operations that can be executed rapidly [26,27].

Figure 4 shows an example of Firebases data structure, with it being a NoSQL database that stores the client data as JSON objects. With this NoSQL database, there are no tables/records, but instead data segments converts into JSON nodes in a predefined structure that is able be retrieved through a key, which can be for example, user IDs ,semantic names, or proxy keys created by Firebase [28] .

Firebases storage architecture could be considered as a cloud hosted JSON tree [28]. When accessing data, it is feasible to traverse through this JSON hierarchy and request certain child nodes of importance that require speedy updates when additional clients alter the data [27].

```
{
  "users": {
    "alovelace": {
      "name": "Ada Lovelace",
      "contacts": { "ghopper": true },
    },
    "ghopper": { ... },
    "eclarke": { ... }
  }
}
```

*Figure 4 Firebase Real Time database data structure [28]*

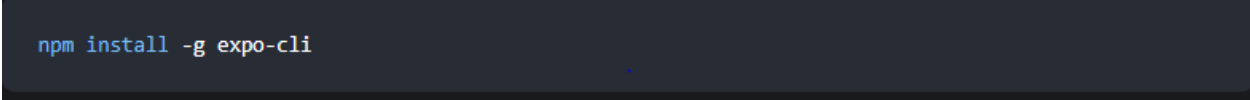
Firebase permits developers the ability to store nested data up to 32 levels. While devising the data structure for the database, it must be carefully considered that querying data for any node, will automatically require that all of its child nodes data be fetched also [28]. It is important and good practice when designing the data structure for Firebase, that developers aim to keep it as flat as possible, as when read/write access permission is granted at a particular node within the database, access to all the underlying nodes is also granted [28].

However, Firebase can store more than just data, it can also cater for storage of various file types such as images, video and audio. Firebase has additional features and services such as a built-in user authentication system that lets developers implement a straightforward registration and login scheme. It provides a hassle-free method of linking social accounts for the purpose of user authentication. Another feature that is available through Firebase is Analytics, where developers can obtain comprehensive information regarding how the various users have engaged with their application. There are also quality control features, which allow the developer the ability to view crash reports and see the overall performance of their application. A notification system can be set up through Firebase Cloud Messaging (FCM), that can notify client applications or send data messages [29].

### 3. Application/Project Implementation

#### 3.1 Setting up development environment

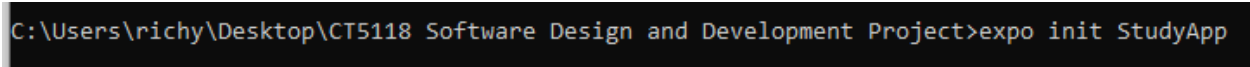
To get started creating a React Native application, an up to date version of Node.js needs to be installed on one's system. This automatically enables the use of NPM and the ability to install additional dependencies, tools and libraries for your application. Once node has been installed, NPM is used to install the EXPO CLI command line utility, see Figure 5.



```
npm install -g expo-cli
```

*Figure 5 NPM installation of expo [30]*

Then using windows command prompt (CMD) terminal, navigate to where the project is to be created and using the “*expo init ProjectName*” CLI, Figure 6, which is a project generator that is maintained by the Expo team, in the React Native community GitHub repository.



```
C:\Users\richy\Desktop\CT5118 Software Design and Development Project>expo init StudyApp
```

*Figure 6 CMD console of NPM installation of the Create React Native project generator*

Figure 6 illustrates the command line code that will use NPM to install the react native library and it is all that is necessary to get underway. Then using the command “npm start” within the project directory, this will generate a QR code which can be scanned via the Expo client on the browser or a mobile device (Android or iOS) as long as both computer and mobile device are connected to the same network [16].

In order to use the Android emulator on windows systems, it is required to have installed the Java SE Development Kit (JDK) and Android Studio in addition to node and the react native cli. Setting up Android Studio can be cumbersome and there are several pieces of software that need to be installed through Android Studio to enable the emulator to work such as:

- Android SDK
- Android SDK platform
- Performance (Intel HAXM)
- Android Virtual Device

Once all prerequisites have been installed, the desired SDK platform for the emulator is chosen from the SDK manager within Android Studio. This emulator can be used via Android Studio through the Android Virtual Device Manager or through the expo windows client [30].

Figure 7 shows the list of dependencies installed using NPM for the project.

```
{
  "main": "node_modules/expo/AppEntry.js",
  "scripts": {
    "start": "expo start",
    "android": "expo start --android",
    "ios": "expo start --ios",
    "web": "expo start --web",
    "eject": "expo eject"
  },
  "dependencies": {
    "@expo/vector-icons": "^10.0.3",
    "expo": "^34.0.0",
    "expo-linear-gradient": "^6.0.0",
    "moment": "^2.24.0",
    "react": "16.8.3",
    "react-dom": "^16.8.6",
    "react-native": "https://github.com/expo/react-native/archive/sdk-34.0.0.tar.gz",
    "react-native-web": "^0.11.4",
    "react-navigation": "^3.11.1",
    "react-native-gesture-handler": "^1.3.0",
    "react-native-reanimated": "^1.1.0",
    "react-navigation-header-buttons": "^3.0.1",
    "react-redux": "^7.1.0",
    "redux": "^4.0.4",
    "redux-thunk": "^2.3.0"
  },
  "devDependencies": {
    "babel-preset-expo": "^5.1.1",
    "redux-devtools-extension": "^2.13.8"
  },
  "private": true
}
```

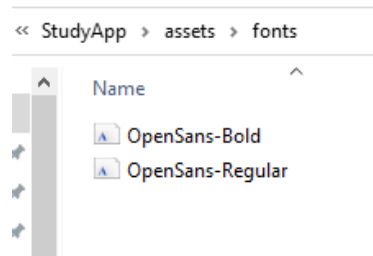
Figure 7 List of dependencies installed on the project, package.json file.

### 3.2 Detailed System Design and Implementation

Name	Date modified	Type	Size
.expo	13/04/2020 14:08	File folder	
assets	13/04/2020 12:40	File folder	
components	13/04/2020 13:07	File folder	
constants	02/08/2019 10:44	File folder	
models	13/04/2020 12:00	File folder	
navigation	12/04/2020 22:46	File folder	
node_modules	11/04/2020 13:45	File folder	
screens	12/04/2020 22:10	File folder	
store	02/08/2019 10:44	File folder	
.gitignore	01/08/2019 13:30	Text Document	1 KB
.watchmanconfig	26/10/1985 09:15	WATCHMANCON...	1 KB
App	13/04/2020 13:21	JS File	2 KB
app	13/04/2020 00:22	JSON File	1 KB
babel.config	26/10/1985 09:15	JS File	1 KB
package	05/08/2019 06:55	JSON File	1 KB

Seen above is the final folder layout of our project, we followed best practice of whenever a component was reusable, we would save it in a separate .js file and import it when it was needed.

In the assets folder, we have a fonts folder which contains two fonts, Open sans and open sans bold:



Both of which were imported into the entire app by placing them in the App.js folder which wraps the entire app and is always rendered.

```
const getFonts = () => {
  return Font.loadAsync({
    'open-sans': require('./assets/fonts/OpenSans-Regular.ttf'),
    'open-sans-bold': require('./assets/fonts/OpenSans-Bold.ttf')
  });
};
```

These fonts can then be used anywhere within the app by either inline styling or by declaring them in the stylesheet:

```
const styles = StyleSheet.create({
  center: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center'
  },
  text: {
    fontFamily: 'open-sans-bold'
  }
});
```

The assets container also contains the icon and splash screen PNG files. The icon file is the image that the app has an icon after it has been downloaded onto a user's phone and the splash screen is what the user sees while it is booting up on their phone.

The components folder contains four .js files, which we all reused several times throughout the project.

AssignmentItem.js was the format we used whenever a user added an assignment it would have this styling and layout:

```
return (
  <Template style={styles.assignment}>
    <View style={styles.touched}>
      <TouchableCmp onPress={props.onSelect} useForeground>
        <View>
          <View style={styles.details}>
            <Text style={styles.title}>{props.title}</Text>
            <Text style={styles.description}>Due Date: {props.dueDate}</Text>
            <Text style={styles.description}>Description: {props.description}</Text>
          </View>
          <View style={styles.actions}>
            {props.children}
          </View>
        </View>
      </TouchableCmp>
    </View>
  </Template>
);
```

[EDIT](#)**UML assignment**[DELETE](#)

Due Date: 26.08.2020, 23.59

Description: Use case diagrams

CustomButton.js was a button that could accept different symbols as props therefore, it would always render the same way and could display different symbols:

```
const CustomButton = props => {  
  return (  
    <HeaderButton  
      {...props}  
      IconComponent={Icons}  
      iconSize={23}  
      color={Platform.OS === 'android' ? 'white' : Colors.first}  
    />  
  );  
};  
  
export default CustomButton;
```

CustomButton was used several times throughout the project, here is an example from the UserAssignmentScreen:

```
headerLeft: (  
  <HeaderButtons HeaderButtonComponent={CustomButton}>  
    <Item  
      title="Menu"  
      iconName={Platform.OS === 'android' ? 'md-menu' : 'ios-menu'}  
      onPress={() => {  
        navData.navigation.toggleDrawer();  
      }}  
    />  
  </HeaderButtons>  
) ,  
headerRight: (  
  <HeaderButtons HeaderButtonComponent={CustomButton}>  
    <Item  
      title="Add"  
      iconName={Platform.OS === 'android' ? 'md-add' : 'ios-add'}  
      onPress={() => {  
        navData.navigation.navigate('EditAssignment');  
      }}  
    />  
  </HeaderButtons>  
)  
)
```

And it's corresponding output on the simulator:



Input.js was a simple validation file where we used reducers to check if info typed in on forms or in the authentication process was valid or not, these were the validations we checked.

```
const textChange = text => {
  const email = /^[^<>()\\[\]\\\.,;:\s@"]+(\.[^<>()\\[\]\\\.,;:\s@"]+)*)(\".+\"))@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\])|(([a-zA-Z\-0-9]+\.)+[a-zA-Z]{2,}))$/;
  let isValid = true;
  if (props.required && text.trim().length === 0) {
    isValid = false;
  }
  if ([props.email && !email].test(text.toLowerCase())) {
    isValid = false;
  }
  dispatch({ type: INPUT_CHANGE, value: text, isValid: isValid });
};
```

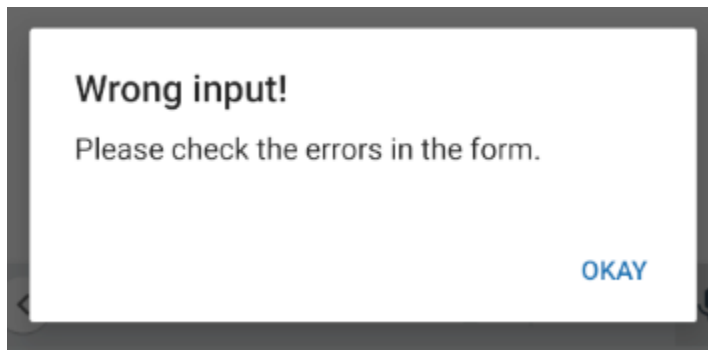
Here is what it returned:

```
return (
  <View style={styles.formControl}>
    <Text style={styles.label}>{props.label}</Text>
    <TextInput
      {...props}
      style={styles.input}
      value={inputState.value ? String(inputState.value) : null}
      onChangeText={textChange}
      onBlur={lostFocus}
    />
    {!inputState.isValid && inputState.touched && (
      <View style={styles.errorContainer}>
        <Text style={styles.errorText}>{props.errorText}</Text>
      </View>
    )}
  </View>
);
```



This is it being used to validate the assignment description submission in the EditAssignmentScreen

```
Input
  id="description"
  label="Description"
  errorText="Please enter a valid description!"
  keyboardType="default"
  autoCapitalize="sentences"
  autoCorrect
  onChange={inputChange}
  initialValue={editedAssignment ? editedAssignment.description : ''}
  initiallyValid={!editedAssignment}
  required
  minLength={5}
```



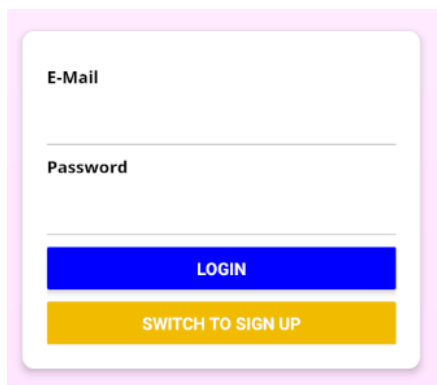
Template.js is a very simple .js file that gives a lifted look to whatever implements it:

```
const Template = props => {
  return <View style={{...styles.Template, ...props.style}}>{props.children}</View>;
};

const styles = StyleSheet.create({
  Template: {
    shadowColor: 'black',
    shadowOpacity: 0.26,
    shadowOffset: { width: 0, height: 2 },
    shadowRadius: 8,
    elevation: 5,
    borderRadius: 10,
    backgroundColor: 'white'
  }
});
```

it also receives whatever props are given to it so styles can be layered on top of it as seen in the AuthScreen:

```
<Template style={styles.authent}>
  <ScrollView>
    <Input
      id="email"
      label="E-Mail"
      keyboardType="email-address"
      required
      email
      autoCapitalize="none"
      errorText="Please enter a valid email address."
      onChange={inputChange}
      initialValue=""
    />
    <Input
      id="password"
      label="Password"
      keyboardType="default"
      secureTextEntry
      required
      minLength={5}
      autoCapitalize="none"
      errorText="Please enter a valid password."
      onChange={inputChange}
      initialValue=""
    />
  </ScrollView>
</Template>
```



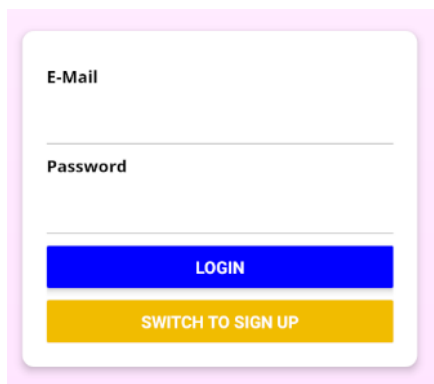
The constants folder only contains one file and that's Colors.js.

It exports two colours, a first blue shade and a second complimentary yellow, both are used throughout the app to make it more uniform.

```
constants > JS Colors.js > [e] default
1  export default {
2    first: '#0000ff',
3    second: '#F1BC0D'
4  };
```

Colors.js being used to style the buttons in the AuthScreen:

```
<View style={styles.buttons}>
  {loading ? (
    <ActivityIndicator size="small" color={Colors.first} />
  ) : (
    <Button
      title={signup ? 'Sign Up' : 'Login'}
      color={Colors.first}
      onPress={authenticator}
    />
  )}
</View>
<View style={styles.buttons}>
  <Button
    title={`Switch to ${signup ? 'Login' : 'Sign Up'}`}
    color={Colors.second}
    onPress={() => {
      setSignup(prevState => !prevState);
    }}
  />
</View>
```



E-Mail

Password

LOGIN

SWITCH TO SIGN UP

The models folder contains the Assignment.js which is a regular JavaScript class that gives us the basic setup of our Assignment model and all its variables:

```
class Assignment {
  constructor(id, ownerId, title, description, dueDate) {
    this.id = id;
    this.ownerId = ownerId;
    this.title = title;
    this.description = description;
    this.dueDate = dueDate;
  }
}

export default Assignment;
```

In the navigation folder we have two .js file directly relating to navigation:

NavigationContainer.js is a wrapper container used solely to wrap the entire app and redirects the user back to the AuthScreen to log in again once their authentication token which is timed, logs them out:

```
import { NavigationActions } from 'react-navigation';

import Navigator from './Navigator';

const NavigationContainer = props => {
  const navReference = useRef();
  const isAuthenticated = useSelector(state => !!state.auth.token);

  useEffect(() => {
    if (!isAuthenticated) {
      navReference.current.dispatch(
        NavigationActions.navigate({ routeName: 'Auth' })
      );
    }
  }, [isAuthenticated]);

  return <Navigator ref={navReference} />;
};

export default NavigationContainer;
```

The NavigationContainer.js wraps around the main navigation file Navigator.js.

Most of the React Navigation data is held in Navigator.js. This starts by importing all the screens:

```
import SettingsScreen from '../screens/SettingsScreen';
import UserAssignmentsScreen from '../screens/UserAssignmentsScreen';
import EditAssignmentScreen from '../screens/EditAssignmentScreen';
import AuthScreen from '../screens/AuthScreen';
import StartupScreen from '../screens/StartupScreen';
import Colors from '../constants/Colors';
import * as authActions from '../store/actions/auth';
```

Setting up default navigation style, which is made into a function so all the screens can share the same header settings:

```
const navOptions = {
  headerStyle: {
    backgroundColor: Platform.OS === 'android' ? Colors.first : ''
  },
  headerTitleStyle: {
    fontFamily: 'open-sans-bold'
  },
  headerBackTitleStyle: {
    fontFamily: 'open-sans'
  },
  headerTintColor: Platform.OS === 'android' ? 'white' : Colors.first
};
```

The above is only used in this file which is why it was not made into a component by itself.

Then each area of the app is set up as a stack navigation, which means that all of the screens stack on top of one another, it's identifying name and it's configuration for the drawer navigation is also set up, giving it a unique icon, size and colour.

Settings was supposed to be more robust but sadly is just a dummy screen:

```
const SettingsNavigator = createStackNavigator(  
  {  
    Setup: SettingsScreen  
  },  
  {  
    navigationOptions: {  
      drawerIcon: drawerConfig => (  
        <Ionicons  
          name={Platform.OS === 'android' ? 'md-settings' : 'ios-settings'}  
          size={23}  
          color={drawerConfig.tintColor}  
        />  
      )  
    },  
    defaultNavigationOptions: navOptions  
  }  
);
```

The add assignment section was completed and is the main stage of the app and is where the user is navigated to after they log in:

```
const AddAssignment = createStackNavigator(  
  {  
    UserAssignments: UserAssignmentsScreen,  
    EditAssignment: EditAssignmentScreen  
  },  
  {  
    navigationOptions: {  
      drawerIcon: drawerConfig => (  
        <Ionicons  
          name={Platform.OS === 'android' ? 'md-list' : 'ios-list'}  
          size={23}  
          color={drawerConfig.tintColor}  
        />  
      )  
    },  
    defaultNavigationOptions: navOptions  
  }  
);
```

It contains two Screens, UserAssignmentScreen, where the users see what assignments they have added and EditAssignmentScreen, which is navigated to when a user is adding a new assignment or editing an old one.

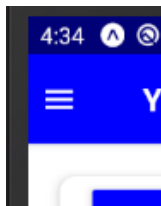
The authentication section does not need a drawer configuration because the logout button will serve the same purpose as it logs the user out thus bringing them to the primary screen which in this case is the StartUpScreen, which checks to see if the current user has any authentication details which they would not be given if they just pressed logout, it would then direct them to the AuthScreen

```
const AuthenticateNavigator = createStackNavigator(  
  {  
    Auth: AuthScreen  
  },  
  {  
    defaultNavigationOptions: navOptions  
  }  
);
```

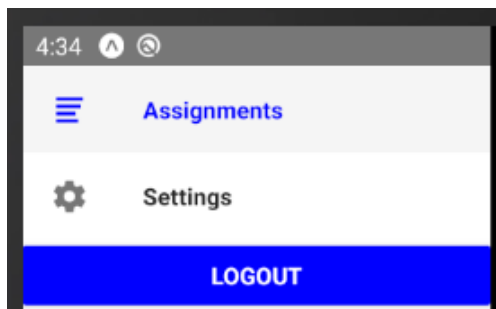
The drawer configuration has three options: Assignments which leads to the AddAssignmentStack above. Settings which leads to the Settings Stack and a Button which logs the current user out of their current session:

```
const Navigator = createDrawerNavigator({
  {
    Assignments: AddAssignment,
    Settings: SettingsNavigator,
  },
  {
    contentOptions: {
      activeTintColor: Colors.first
    },
    contentComponent: props => {
      const dispatch = useDispatch();
      return (
        <View style={{ flex: 1, paddingTop: 20 }}>
          <SafeAreaView forceInset={{ top: 'always', horizontal: 'never' }}>
            <DrawerItems {...props} />
            <Button
              title="Logout"
              color={Colors.first}
              onPress={() => {
                dispatch(authActions.logout());
              }}
            />
          </SafeAreaView>
        </View>
      );
    }
  }
});
```

The drawer on the simulator before it is pressed:



And it's display after it is pressed:



Lastly a Switch navigator is created, the difference between a switch and a stack navigator is that with a switch it does not record the user's history so they cannot return to the page they were just on. This is primarily used for authentication flows because once the user is signed in they should not be able to return to the sign in page unless they log out.

The sequence is also ordered, Startup is first to check if the user has a valid authentication token, if they do then it navigates to Main and if they do not then it navigates to Auth.

```
const MainNavigator = createSwitchNavigator({
  Startup: StartupScreen,
  Auth: AuthenticateNavigator,
  Main: Navigator
});

export default createAppContainer(MainNavigator);
```

Then everything is wrapped in createAppContainer at the root of the app and exported.



In the screens folder are all the .js files that are individual screens that the user can navigate to:

StartupScreen.js returns an activity indicator which is a spinning circle to suggest loading, it is doing this while it checks to see if the current user has a valid token i.e. has signed in recently. It also checks if the current user has ever signed in or if the token they have has expired. If the user fails any of these checks they are returned to the AuthScreen.

```
useEffect(() => {
  const login = async () => {
    const userData = await AsyncStorage.getItem('userData');
    if (!userData) {
      props.navigation.navigate('Auth');
      return;
    }
    const convertedData = JSON.parse(userData);
    const { token, userId, expiryDate } = convertedData;
    const expirationDate = new Date(expiryDate);

    if (expirationDate <= new Date() || !token || !userId) {
      props.navigation.navigate('Auth');
      return;
    }

    const expirationTime = expirationDate.getTime() - new Date().getTime();

    props.navigation.navigate('Main');
    dispatch(authActions.authenticate(userId, token, expirationTime));
  };

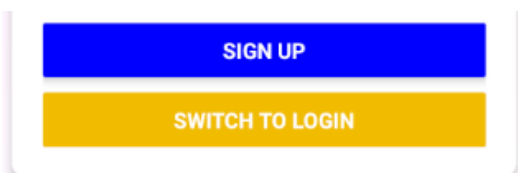
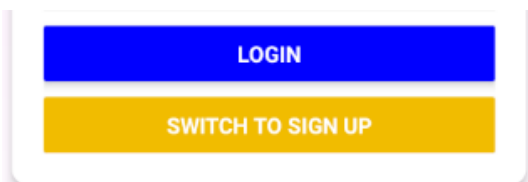
  login();
}, [dispatch]);

return (
  <View style={styles.fullScreen}>
    <ActivityIndicator size="large" color={Colors.first} />
  </View>
);
```

The AuthScreen.js returns two of the previous components, both the Template.js and the Input.js to show a login form which takes an email and a password, above two buttons which can switch titles and actions depending on which are clicked.

```
let action;
if (signup) {
  action = authActions.signup(
    formState.inputValues.email,
    formState.inputValues.password
  );
} else {
  action = authActions.login(
    formState.inputValues.email,
    formState.inputValues.password
  );
}
```

```
<View style={styles.buttons}>
  {loading ? (
    <ActivityIndicator size="small" color={Colors.first} />
  ) : (
    <Button
      title={signup ? 'Sign Up' : 'Login'}
      color={Colors.first}
      onPress={authenticator}
    />
  )}
</View>
<View style={styles.buttons}>
  <Button
    title={`Switch to ${signup ? 'Login' : 'Sign Up'}`}
    color={Colors.second}
    onPress={() => {
      setSignup(prevState => !prevState);
    }}
  />
</View>
```











If the user has an email and password stored on the firebase server they will be directed to the main page.




```
try {  
  await dispatch(action);  
  props.navigation.navigate('Main');  
} catch (err) {  
  setError(err.message);  
  setLoading(false);  
}  
};
```

## Authentication ?

[Users](#) [Sign-in method](#) [Templates](#) [Usage](#)

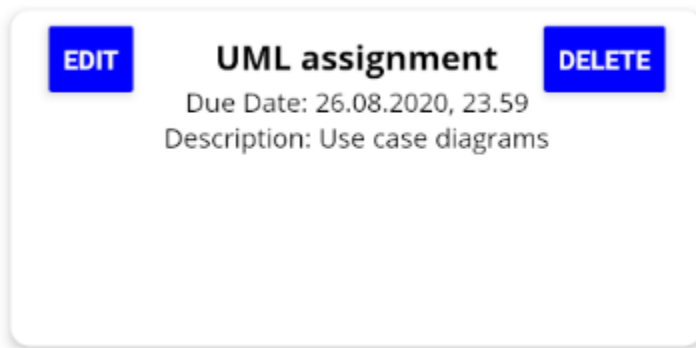
[Add user](#)  

Identifier	Providers	Created	Signed In	User UID 
test@gmail.com		Apr...	Apr...	42IWqDgJYoOHb...
ethankane92@gmail.com		Feb...	Apr...	6ZN1qfRjJDObly...
e.kane4@nuigalway.ie		Feb...	Apr...	Ep5bu7hsvlQ3dP...
richyoconnell@gmail.com		Apr...	Apr...	gfVuNbH93rZK8i...
rocon92@gmail.com		Apr...	Apr...	svv0Qx5jCdZ2xa...

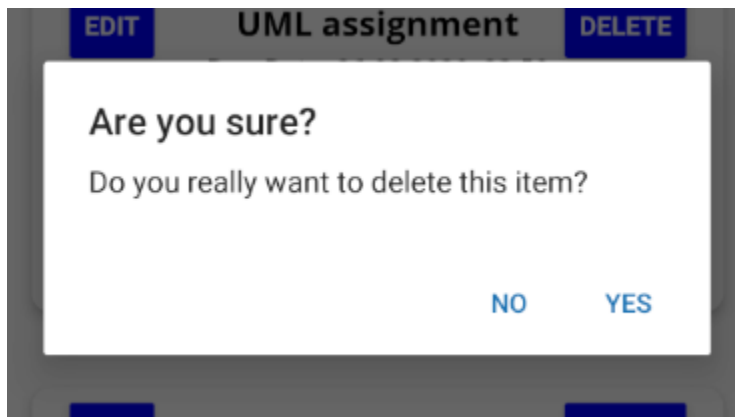
Rows per page: 50  1-5 of 5  

If they do not, they will have to sign up, providing an email and password.

Once signed up and after a successful login then the user will be directed to the UserAssignmentScreen.js which returns a FlatList of assignments using the AssignmentItem.js component. Each item has two buttons which say “EDIT” and “DELETE”.



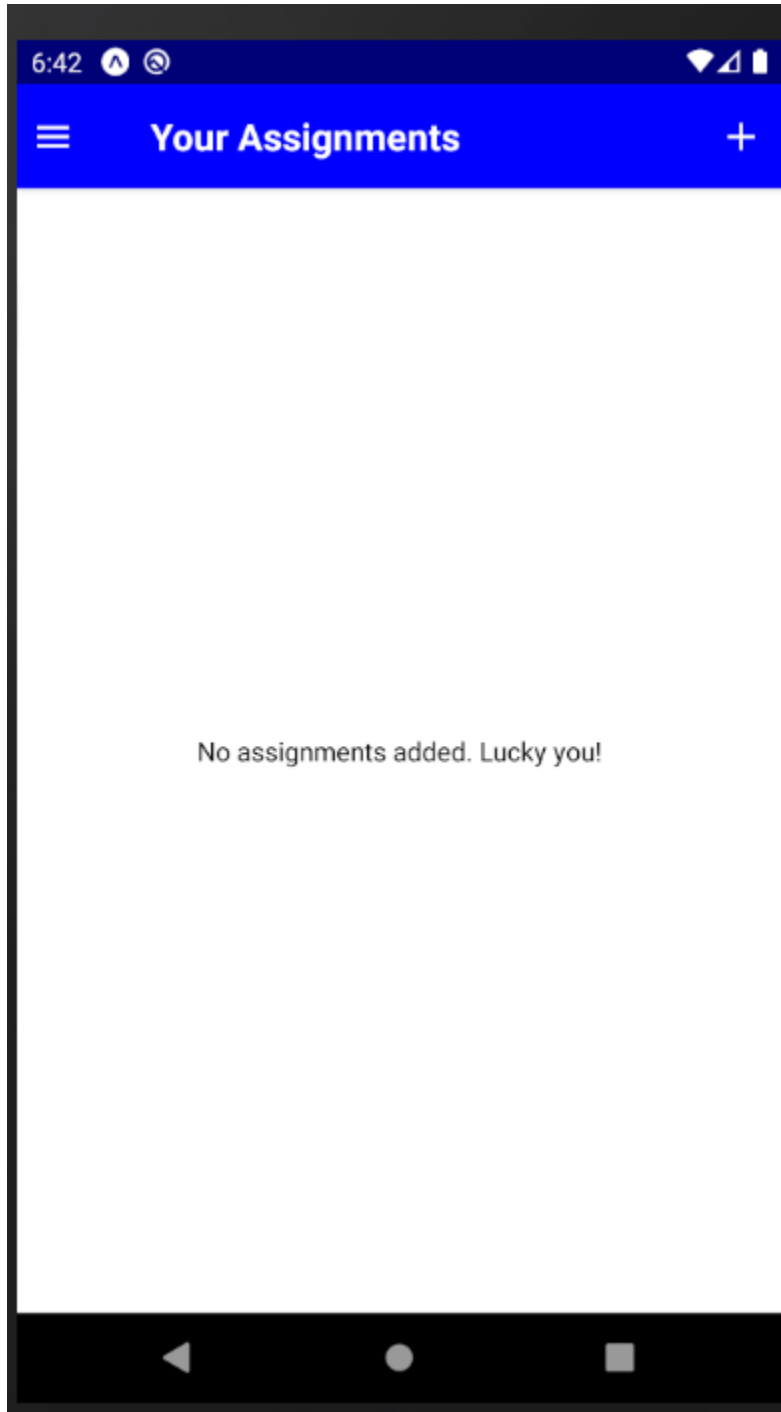
If the user clicks “DELETE” they will be prompted with an Alert asking if they are sure.



```
const deleter = id => {  
  Alert.alert('Are you sure?', 'Do you really want to delete this item?', [  
    { text: 'No', style: 'default' },  
    {  
      text: 'Yes',  
      style: 'destructive',  
      onPress: () => {  
        dispatch(assignmentsActions.deleteAssignment(id));  
      }  
    }  
  ]);  
};
```

This will remove the assignment from the list and from the database.

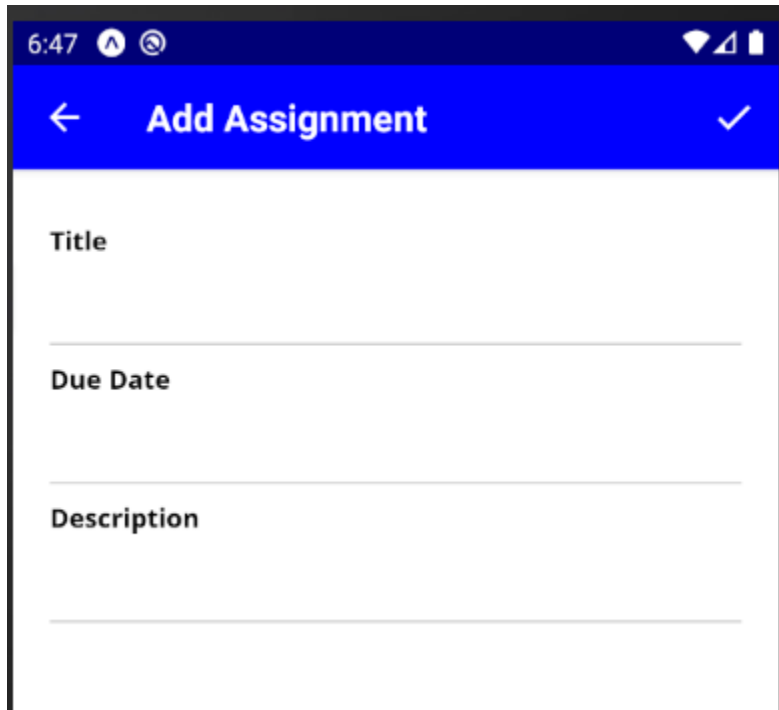
If the user has no assignments added, then they will see the default screen:



If on one of the items, the user clicks the button “EDIT” or they click the Plus button in the top right of the screen then the user will be brought to the EditAssignmentScreen.

EditAssignmentScreen.js returns a scrollable view of three options that the user can edit.

They are blank if they are adding a new assignment:



6:47

← Add Assignment ✓

**Title**

---

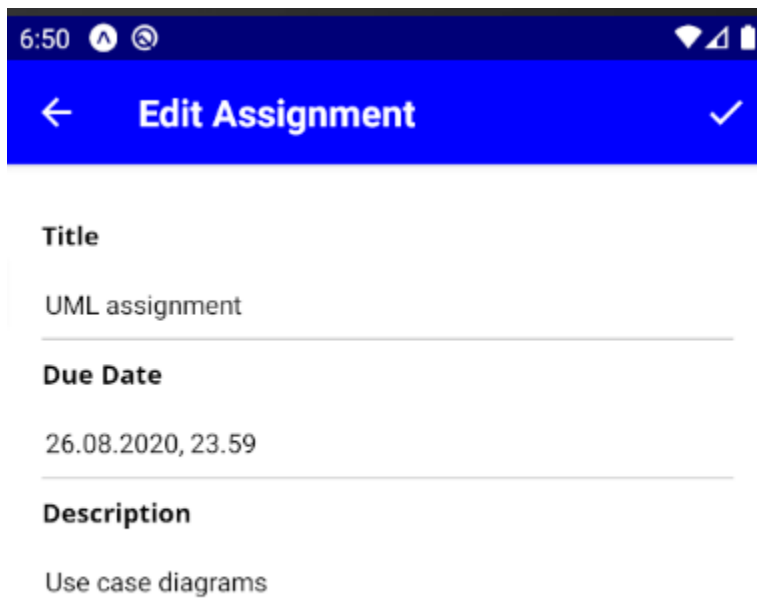
**Due Date**

---

**Description**

---

If they are editing an assignment, they will have the details of the assignment the user clicked on to edit:



6:50

← Edit Assignment ✓

**Title**

UML assignment

---

**Due Date**

26.08.2020, 23.59

---

**Description**

Use case diagrams

---

This is managed by a useReducer hook:

```
const [formState, dispatchFormState] = useReducer(formReducer, {
  inputValues: {
    title: editedAssignment ? editedAssignment.title : '',
    description: editedAssignment ? editedAssignment.description : '',
    dueDate: editedAssignment ? editedAssignment.dueDate : '',
  },
  inputValidities: {
    title: editedAssignment ? true : false,
    description: editedAssignment ? true : false,
    dueDate: editedAssignment ? true : false
  },
  formIsValid: editedAssignment ? true : false
});
```

Validities are also checked by using the Input component:

```
<Input
  id="title"
  label="Title"
  errorText="Please enter a valid title!"
  keyboardType="default"
  autoCapitalize="sentences"
  autoComplete
  returnKeyType="next"
  onChange={inputChange}
  initialValue={editedAssignment ? editedAssignment.title : ''}
  initiallyValid={!editedAssignment}
  required
/>
```

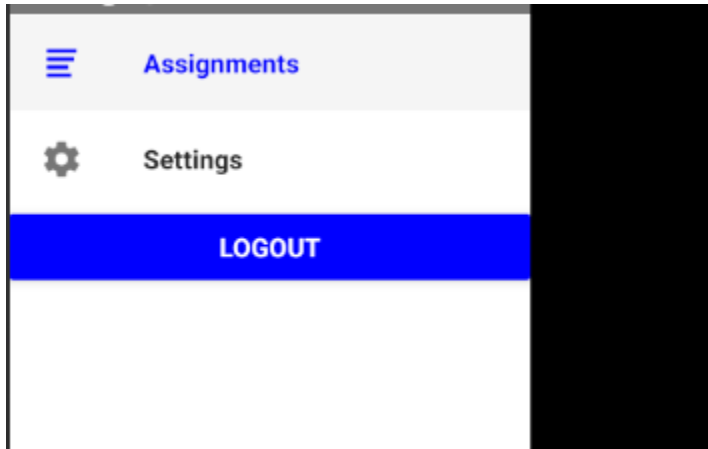
Once the user is satisfied with the details they entered or updated, they click the button in the top right corner with a tick, this will check the validity of the text, if they clicked the “EDIT” button it will trigger the updateAssignment action and if they clicked the plus button it will trigger the createAssignment action. If the form is not valid, they will get an alert asking them to recheck their details. Once either updateAssignment or createAssignment has been completed the page will navigate back to UserAssignmentScreen with the new assignment displayed there.

```
const submitter = useCallback(async () => {
  if (!formState.formIsValid) {
    Alert.alert('Wrong input!', 'Please check the errors in the form.', [
      { text: 'Okay' }
    ]);
    return;
  }
  setError(null);
  setLoading(true);
  try {
    if (editedAssignment) {
      await dispatch(
        assignmentsActions.updateAssignment(
          assId,
          formState.inputValues.title,
          formState.inputValues.description,
          formState.inputValues.dueDate
        )
      );
    } else {
      await dispatch(
        assignmentsActions.createAssignment(
          formState.inputValues.title,
          formState.inputValues.description,
          formState.inputValues.dueDate
        )
      );
    }
    props.navigation.goBack();
  } catch (err) {
    setError(err.message);
  }

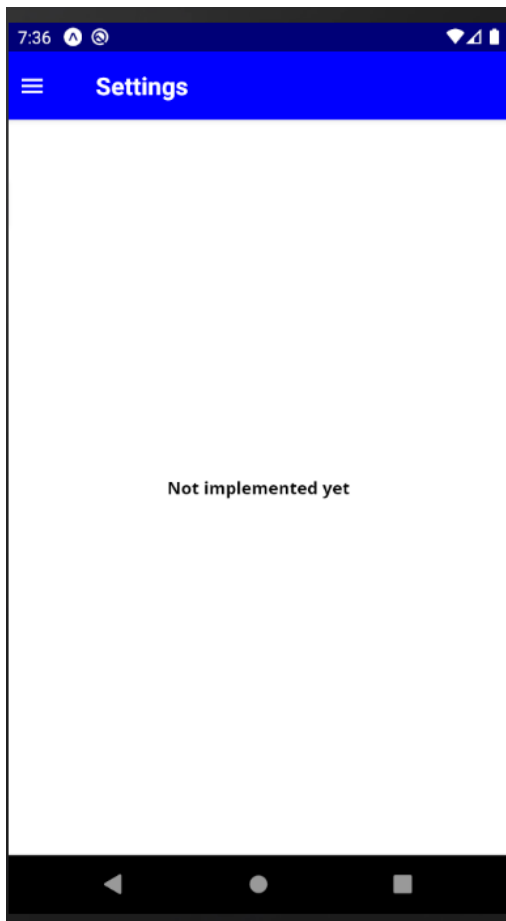
  setLoading(false);
}, [dispatch, assId, formState]);
```



In the drawer there is also an option for the SettingsScreen:



SettingsScreen.js only returns a text element that reads “Not implemented yet”. It was envisioned as where users could set their notification settings for the reminders about assignments as they drew closer, but we did not have enough time to implement it.



The app.js file in the root level file that is the first to be executed when the app starts.

It fetches custom fonts, loads up navigation, combines reducers, applies redux thunk and wraps the rest of the app in the Provider so that everything in the app even if there are multiple components has access to the redux store.

```
const rooter = combineReducers({
  assignments: assignmentsReducer,
  auth: authReducer
});

const store = createStore(rooter, applyMiddleware(Thunk));

const getFonts = () => {
  return Font.loadAsync({
    'open-sans': require('./assets/fonts/OpenSans-Regular.ttf'),
    'open-sans-bold': require('./assets/fonts/OpenSans-Bold.ttf')
  });
};

export default function App() {
  const [fontLoad, setFontLoad] = useState(false);

  if (!fontLoad) {
    return (
      <AppLoading
        startAsync={getFonts}
        onFinish={() => {
          setFontLoad(true);
        }}
      />
    );
  }
  return (
    <Provider store={store}>
      <NavigationContainer />
    </Provider>
  );
}
```

All essential to be loaded at the launch of the app.

The store folder has two folders, actions and reducers. They both contain js files of the same names auth.js and assignment.js.

When a component dispatches an action, it reaches a reducer which then updates the apps state.

The initial state in the assignment reducer is an empty array:

```
const initialState = {  
  userAssignments: []  
};
```

Then a component like in EditAssignmentScreen which has imported the actions--

```
import * as assignmentsActions from '../store/actions/assignments';
```

--will call an action from the assignmentActions it has imported:

```
assignmentsActions.createAssignment(  
  formState.inputValues.title,  
  formState.inputValues.description,  
  formState.inputValues.dueDate  
)
```

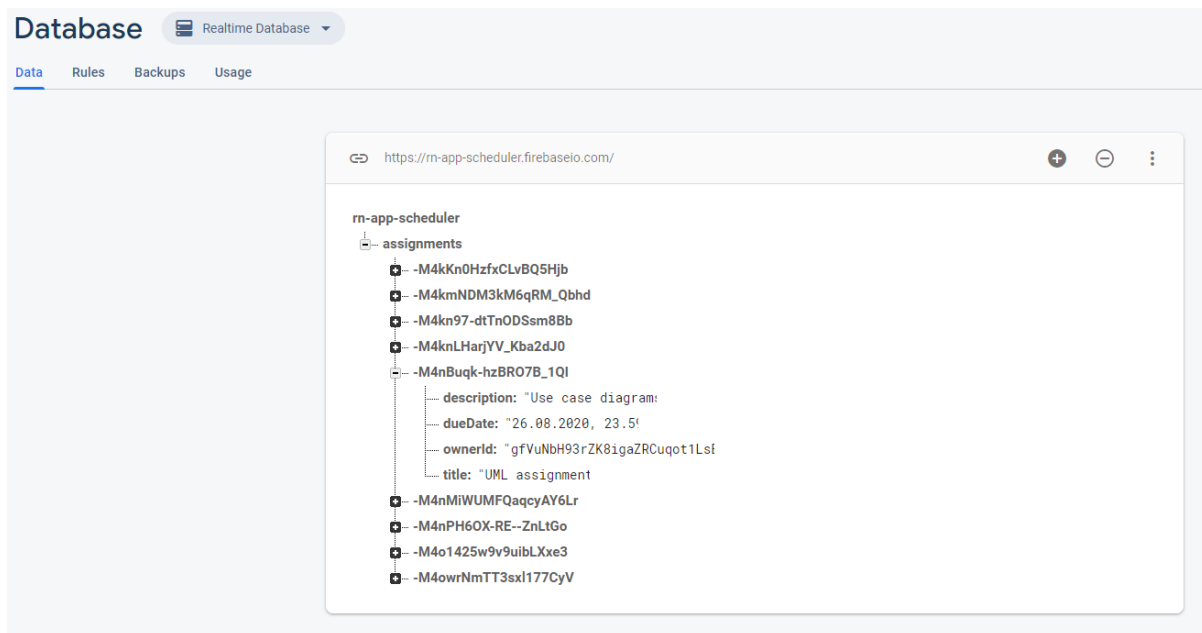
Which will dispatch a function in the assignments.js in the actions folder:

```
export const createAssignment = (title, description, dueDate) => {
  return async (dispatch, getState) => {
    const token = getState().auth.token;
    const userId = getState().auth.userId;
    const response = await fetch(
      `https://rn-app-scheduler.firebaseio.com/assignments.json?auth=${token}`,
      {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json'
        },
        body: JSON.stringify({
          title,
          description,
          dueDate,
          ownerId: userId
        })
      }
    );

    const resData = await response.json();

    dispatch({
      type: CREATE_ASSIGNMENT,
      assignmentData: {
        id: resData.name,
        title,
        description,
        dueDate,
        ownerId: userId
      }
    });
  });
};
```

The above function will also run some async code which will post the object to the database on firebase:



This will reach a reducer in the assignments.js in the reducers folder, which has also imported the actions:

```
store > reducers > JS assignments.js > ...  
1  import {  
2    DELETE_ASSIGNMENT,  
3    CREATE_ASSIGNMENT,  
4    UPDATE_ASSIGNMENT,  
5    SET_ASSIGNMENTS  
6  } from '../actions/assignments';
```

The reducer will in turn update the initial state in this case adding a new assignment into the userAssignments array updating our initial state:

```
case CREATE_ASSIGNMENT:
  const newAssignment = new Assignment(
    action.assignmentData.id,
    action.assignmentData.ownerId,
    action.assignmentData.title,
    action.assignmentData.description,
    action.assignmentData.dueDate
  );
  return {
    ...state,
    userAssignments: state.userAssignments.concat(newAssignment)
  };

```

There are similar corresponding actions and reducers to delete update and return our assignment objects.

There are similar actions and reducers for user authentication:

The initial state is two null values:

```
const initialState = {
  token: null,
  userId: null
};

```

Then a component in the Authscreen calls one of the actions it has imported:

```
import * as authActions from '../store/actions/auth';

```

```
action = authActions.signup(
  formState.inputValues.email,
  formState.inputValues.password
);
} else {
  action = authActions.login(
    formState.inputValues.email,
    formState.inputValues.password
  );
}



```







This dispatches a function in the auth.js in the actions folder, for logging in, it will run async code which will find the user's email and password authentication details on firebase. It also has some basic error handling if the email is not found in the database. It will also keep track of how much time is left on the token before the user will have to sign in again, the latter is saved locally.




```
export const login = (email, password) => {  
  return async dispatch => {  
    const response = await fetch(  
      'https://identitytoolkit.googleapis.com/v1/accounts:signInWithPassword?key=AIzaSyD8WG0UuGiOzybIAqGB491ZoT9Jxb83Y4',  
    )  
    {  
      method: 'POST',  
      headers: {  
        'Content-Type': 'application/json'  
      },  
      body: JSON.stringify({  
        email: email,  
        password: password,  
        returnSecureToken: true  
      })  
    }  
  )  
};  
  
if (!response.ok) {  
  const errorResData = await response.json();  
  const errorId = errorResData.error.message;  
  let message = 'Something went wrong!';  
  if (errorId === 'EMAIL_NOT_FOUND') {  
    message = 'This email could not be found!';  
  } else if (errorId === 'INVALID_PASSWORD') {  
    message = 'This password is not valid!';  
  }  
  throw new Error(message);  
}  
  
const resData = await response.json();  
dispatch(  
  authenticate(  
    resData.localId,  
    resData.idToken,  
    parseInt(resData.expiresIn) * 1000  
  )  
);  
const expirationDate = new Date(  
  new Date().getTime() + parseInt(resData.expiresIn) * 1000  
);  
saveDataToStorage(resData.idToken, resData.localId, expirationDate);  
};  
};
```

## Authentication

[Users](#) [Sign-in method](#) [Templates](#) [Usage](#)

[Add user](#)  

Identifier	Providers	Created	Signed In	User UID 
test@gmail.com		Apr 13, 2020	Apr 13, 2020	42IWqDgJyoOHbgS0gW6uWwyXj...
ethankane92@gmail.com		Feb 27, 2020	Apr 8, 2020	6ZN1qfRjJDOBlyg1V4qdKceMvdC2
e.kane4@nuigalway.ie		Feb 27, 2020	Apr 3, 2020	Ep5bu7hsvlQ3dPA3iZBuQvTVGv1
richyoconnell@gmail.com		Apr 1, 2020	Apr 13, 2020	gfVuNbH93rZK8igaZRCuqot1LsE3
rocon92@gmail.com		Apr 12, 2020	Apr 13, 2020	svv0Qx5jCdZ2xauXJxgmbxinsBD3

Rows per page: 50  1-5 of 5  

Thus reaching the reducer in the auth.js in the reducers folder and updating the new values:

```
case AUTHENTICATE:
  return {
    token: action.token,
    userId: action.userId
  };
```



Similar actions and reducers take place when a user clicks the button titled “LOGOUT” on the side drawer in Navigation.js:

```
<DrawerItems {...props} />
<Button
  title="Logout"
  color={Colors.first}
  onPress={() => {
    dispatch(authActions.logout());
  }}
/>
```

Which dispatches this function in auth.js in the actions folder:

```
export const logout = () => {
  clearLogoutTimer();
  AsyncStorage.removeItem('userData');
  return { type: LOGOUT };
};

const clearLogoutTimer = () => {
  if (timer) {
    clearTimeout(timer);
  }
};

const setLogoutTimer = expirationTime => {
  return dispatch => {
    timer = setTimeout(() => {
      dispatch(logout());
    }, expirationTime);
  };
};
```

Reaching the reducer in the auth.js in the reducers folder and returning the state to it's initial empty values therefore logging the user out.

```
case LOGOUT:
  return initialState;
default:
  return state;
```

As both of us developed different aspects of the App at different times we also employed a sandwich testing style, as each of us built a module or a screen or connected the database we tested it sometimes with dummy buttons or navigators to see if it yielded the correct results, these stubs were then replaced as we built further [31].

## 4.1 Black Box Testing

We also employed black box testing at the very end of the process, When the app had gotten to the point where we could do no more work on it given the deadlines we decided to implement some rather simple black box functional test cases to see if we had fulfilled some of our requirements.

We started with 7 functional requirements;

### **Test Case 1**

Requirement: The App must load when opened:

We first did this on android studio's built in emulator where we had done most of the coding

npm

Microsoft Windows [Version 10.0.18363.752]

(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\richy\Desktop\CT5118 Software Design and Development Project\StudyApp>npm start

> @ start C:\Users\richy\Desktop\CT5118 Software Design and Development Project\StudyApp

> expo start

There is a new version of expo-cli available (3.17.24).  
You are currently using expo-cli 3.17.11  
Install expo-cli globally using the package manager of your choice;  
for example: `npm install -g expo-cli` to get the latest version

Starting project at C:\Users\richy\Desktop\CT5118 Software Design and Development Project\StudyApp

Expo DevTools is running at <http://localhost:19002>

Opening DevTools in the browser... (press shift-d to disable)

Starting Metro Bundler on port 19001.

Successfully ran `adb reverse`. Localhost URLs should work on the connected Android device.

Tunnel ready.

<exp://192.168.0.165:19000>



To run the app with live reloading, choose one of:

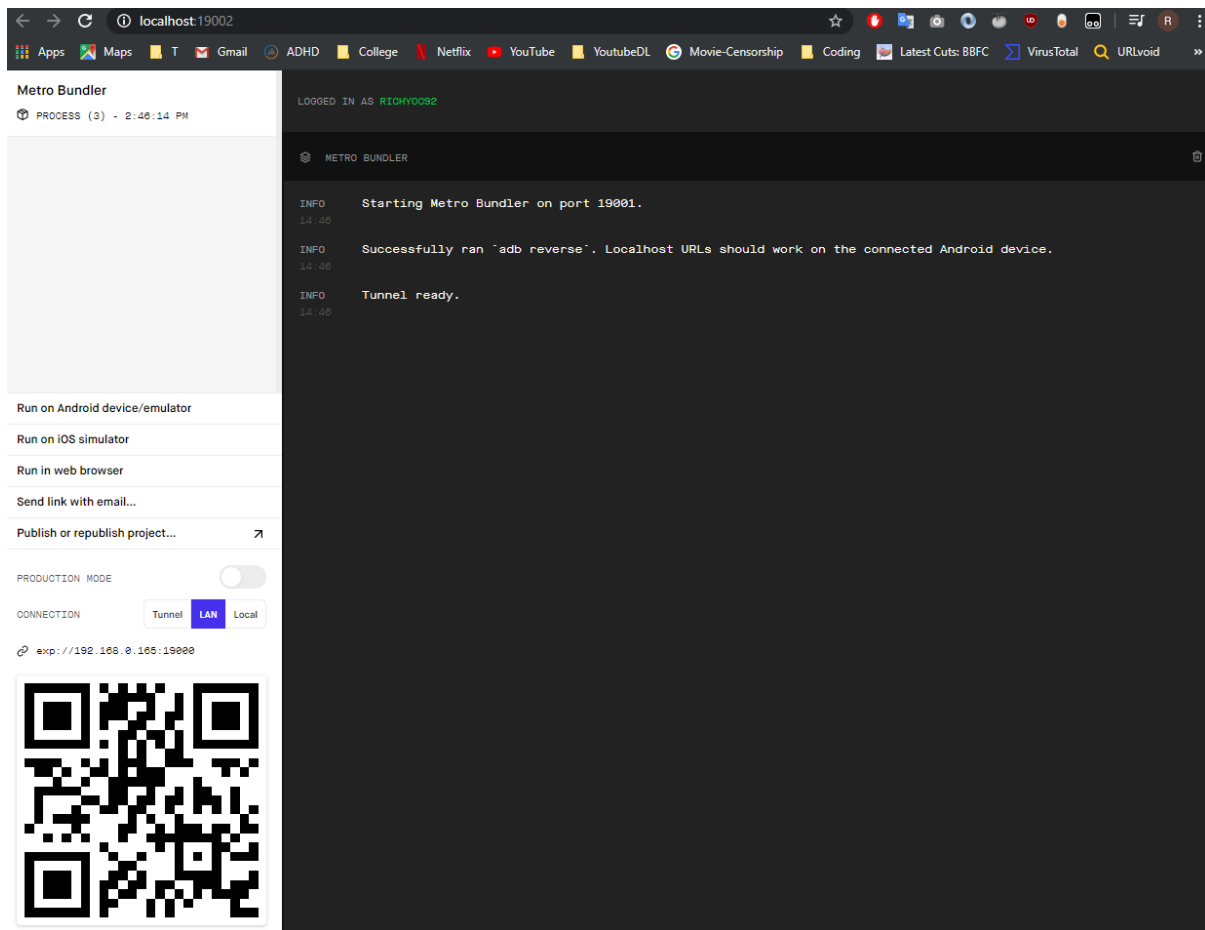
- Sign in as @richyoc92 in Expo client on Android or iOS. Your projects will automatically appear in the "Projects" tab.
- Scan the QR code above with the Expo app (Android) or the Camera app (iOS).
- Press a for Android emulator, or w to run on web.
- Press e to send a link to your phone with email.

Expo Press ? to show a list of all available commands.

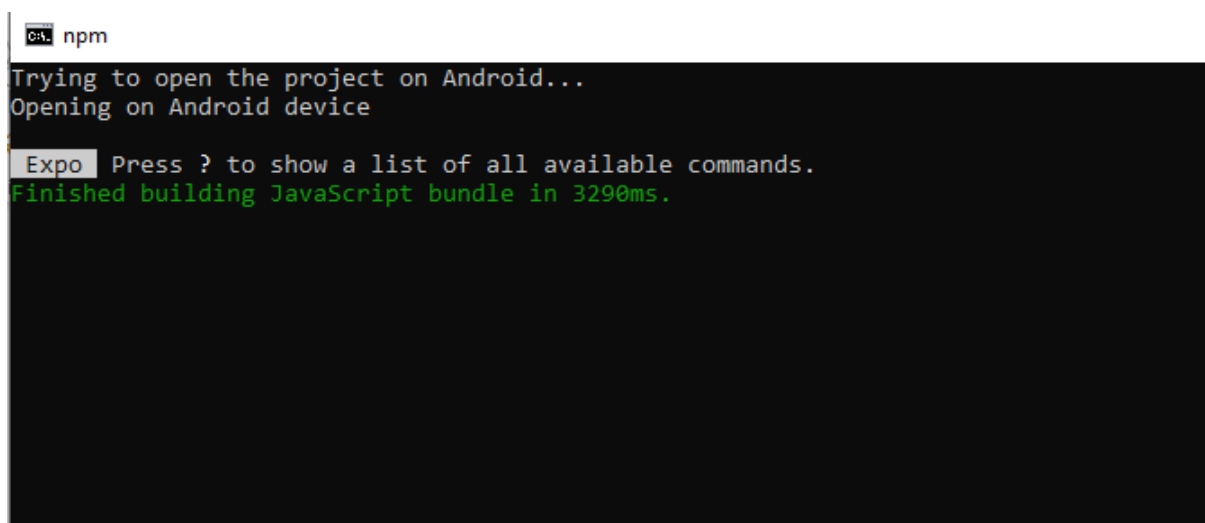
Logs for your project will appear below. Press Ctrl+C to exit.

-

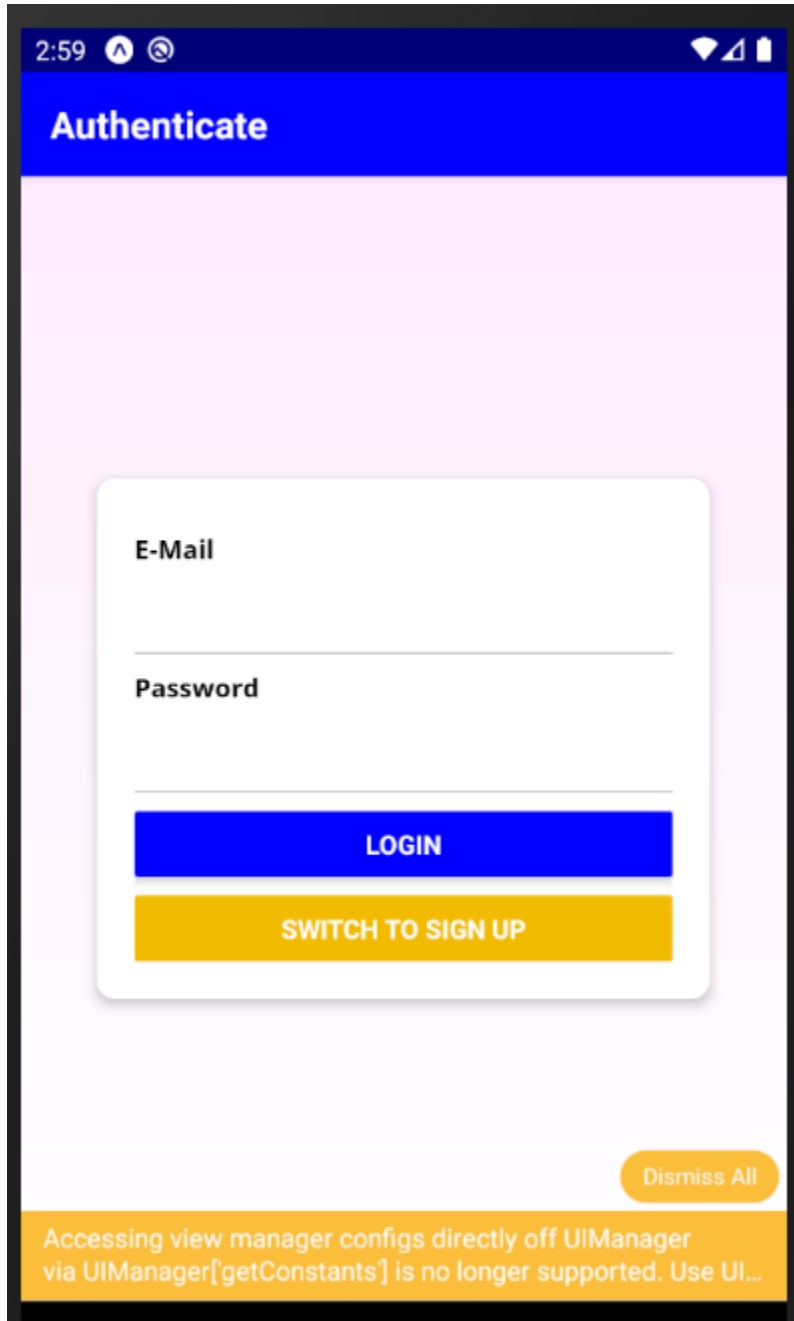
This opened up the expo on our localhost:



From there we could choose the “Run on Android device/emulator” option once we had a working emulator open:

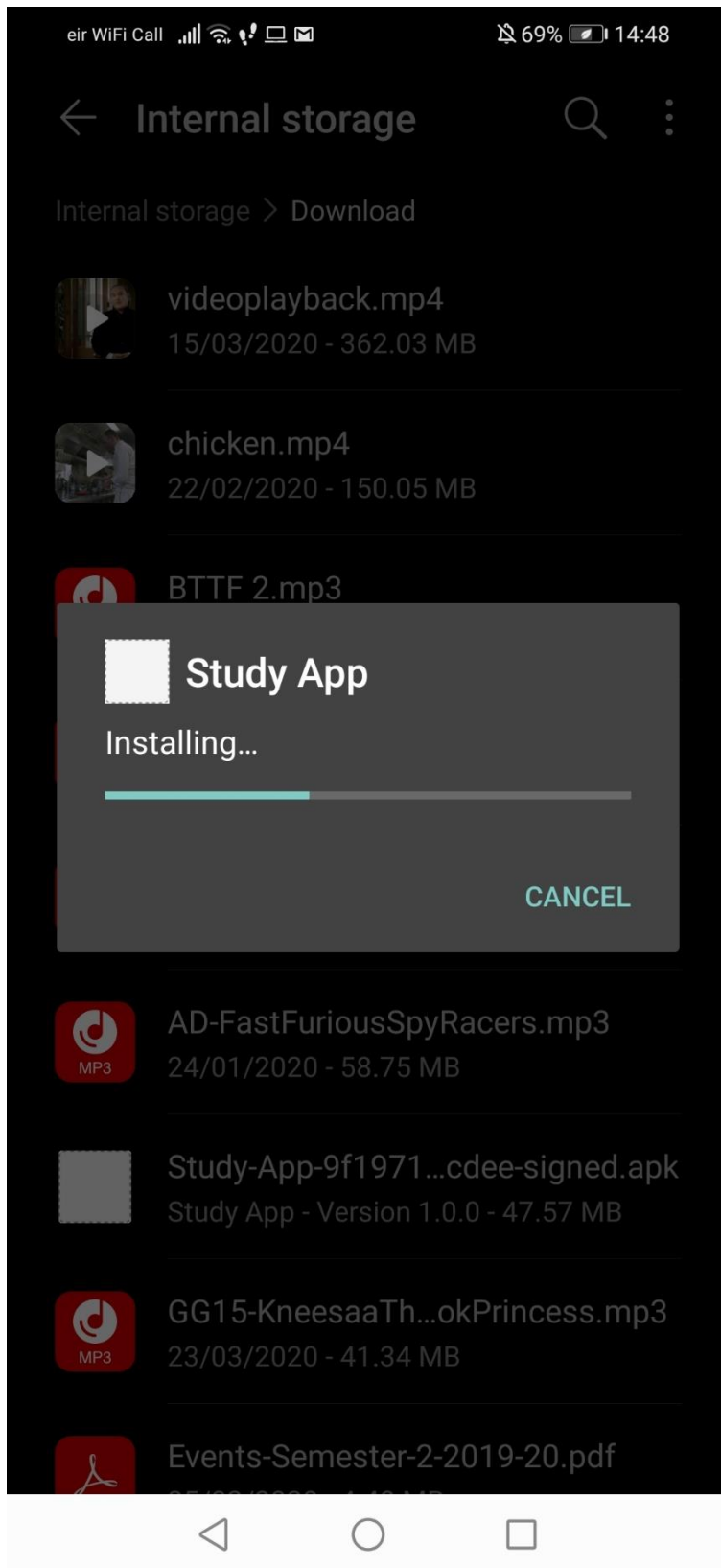


This opened the app on our emulator:

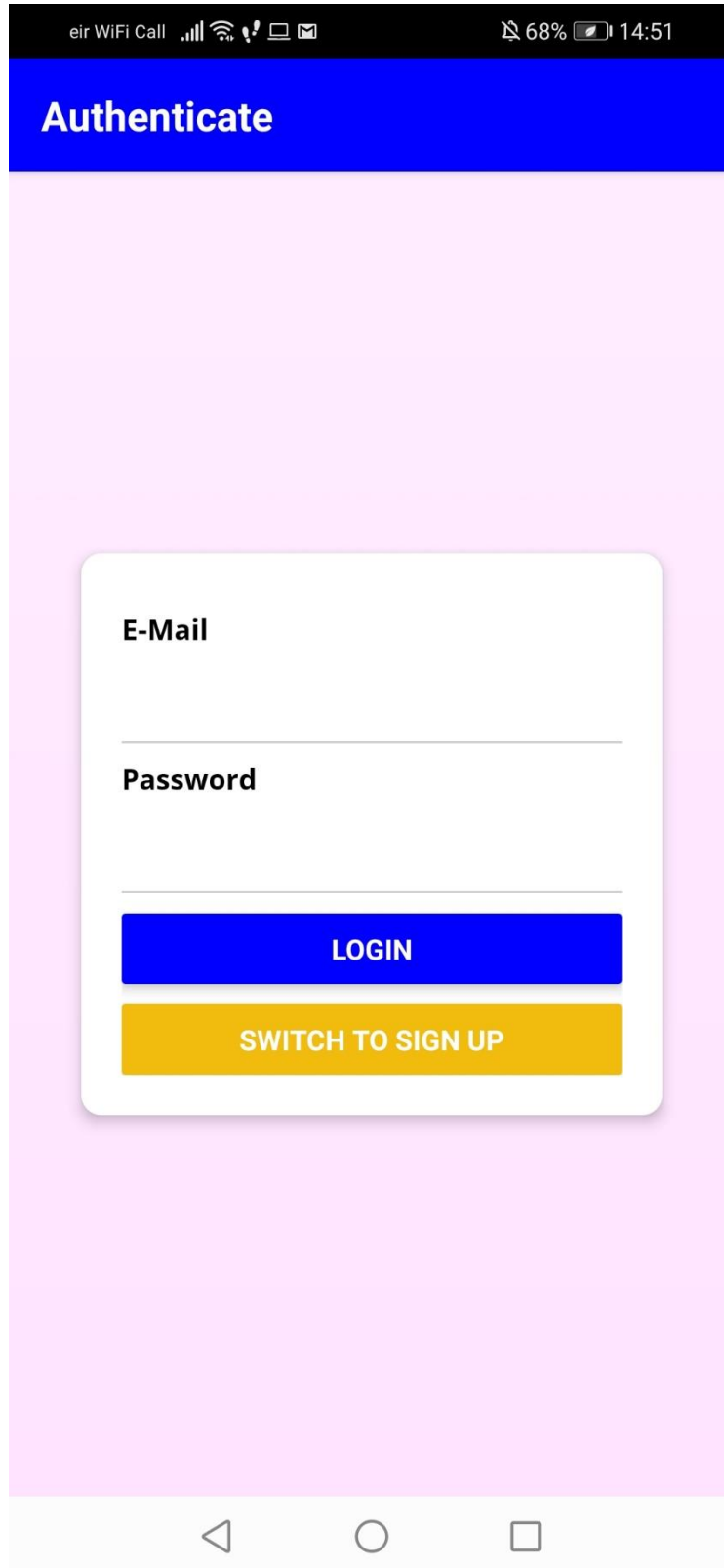


Although there was a warning around an outdated dependency the app still opened as expected.

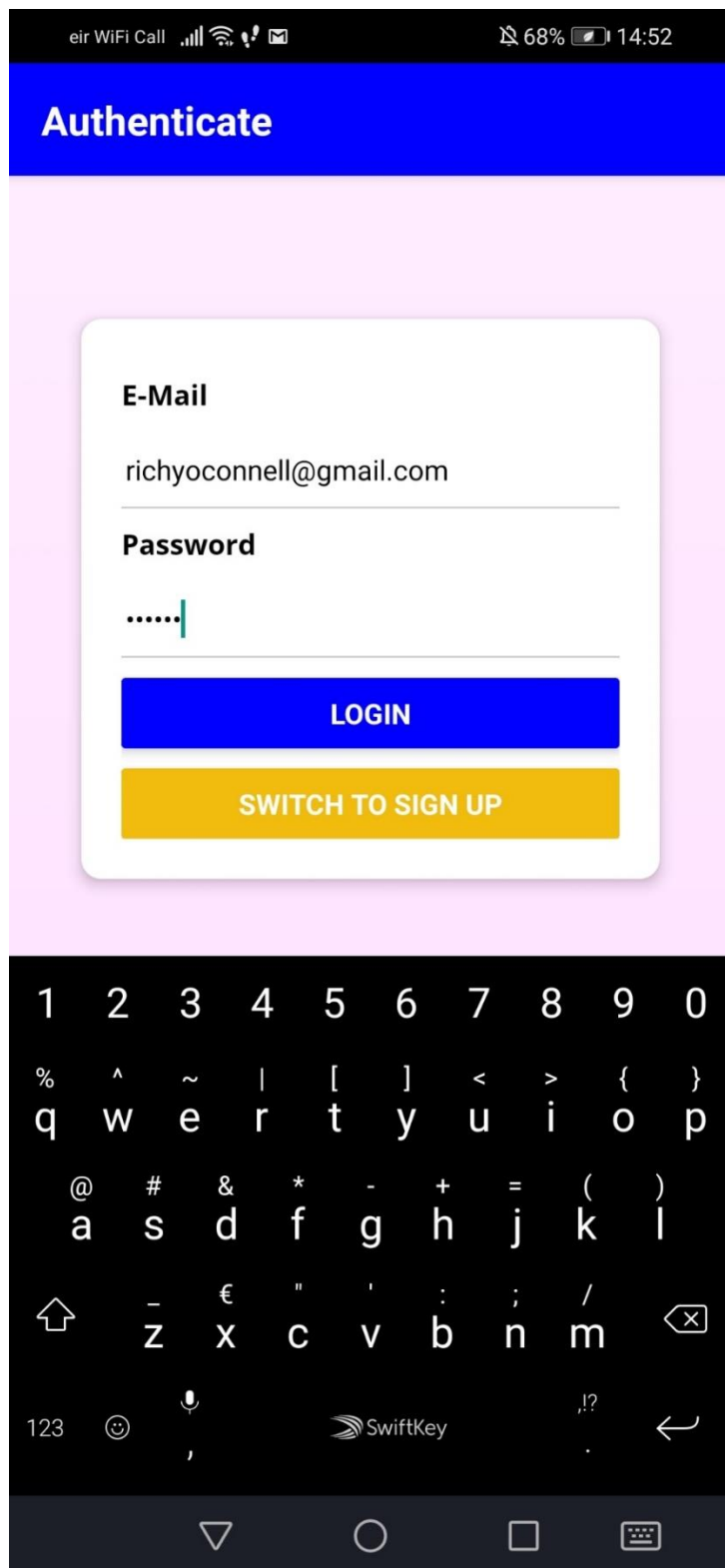
We also exported it from expo as an apk and installed it on an android phone:



The app also opened on an android phone without any errors:







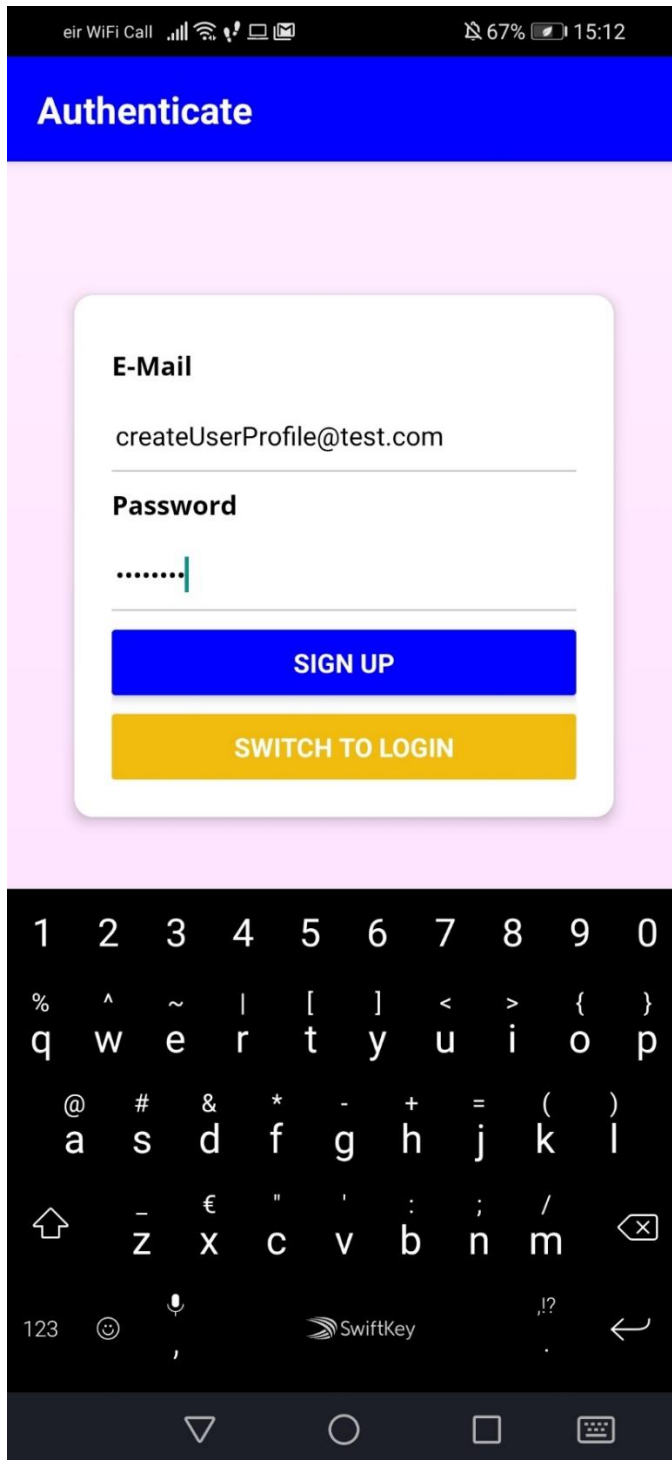
Requirement: **Fulfilled**

Test Case 1: **Passed**

## Test Case 2

Requirement: The app must allow a user to create a profile

Using the mobile phone, we entered a dummy email and password into the sign up option:



It was then confirmed as being added on the firebase authenticated users:

Authentication

Users Sign-in method Templates Usage

Search by email address, phone number, or user UID Add user

Identifier	Providers	Created	Signed in	User UID ↑
createuserprofile@test.com	📧	Apr 14, 2020	Apr 14, 2020	20EB4HYfJz9ruJ8pwLol1b1TP6z1

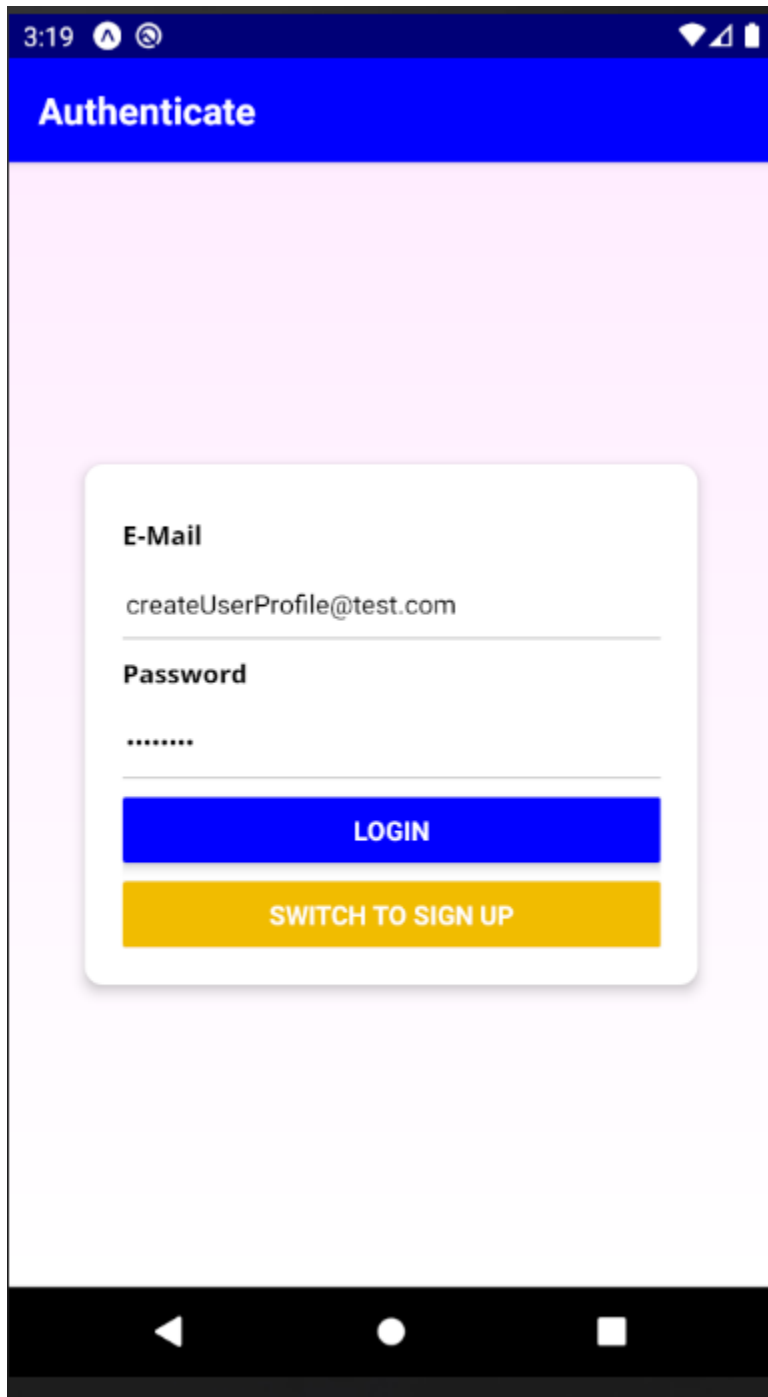
Requirement: **Fulfilled**

Test Case 2: **Passed**

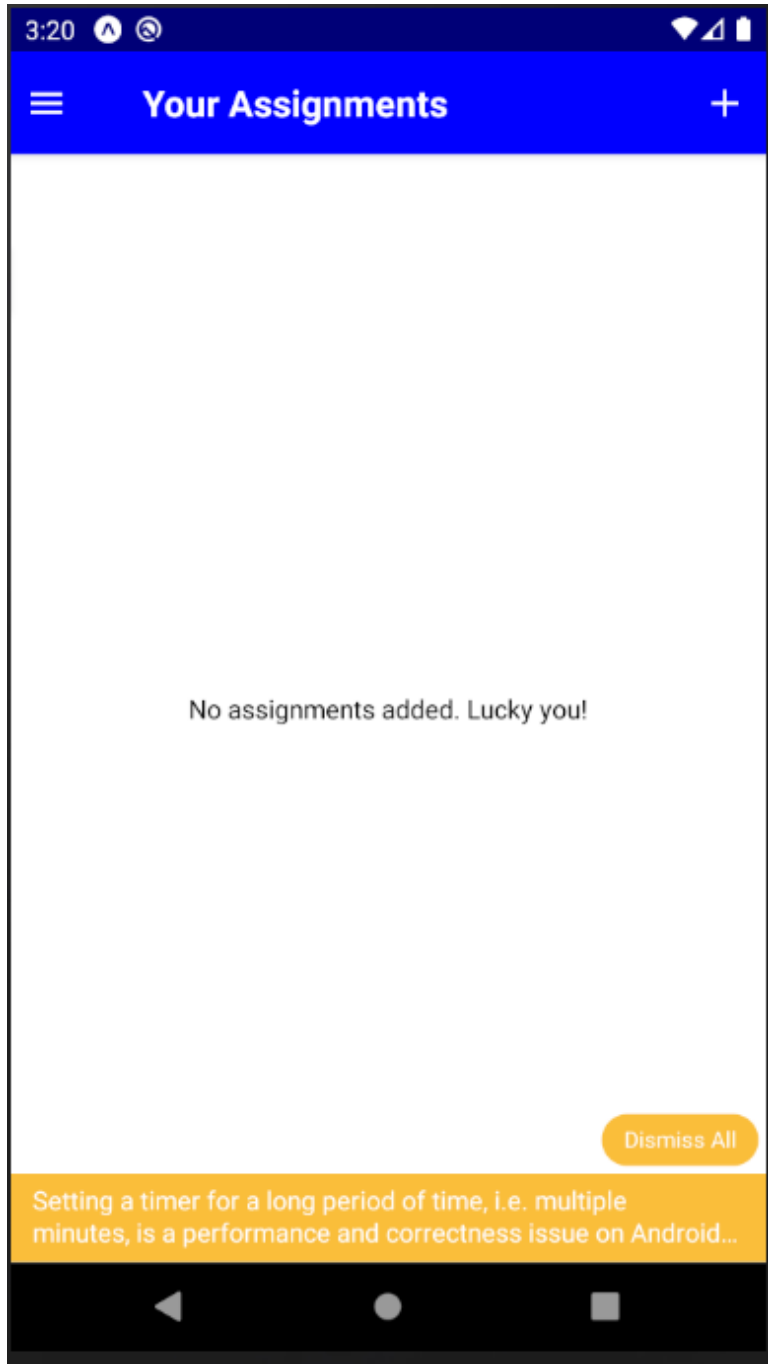
### Test case 3

Requirement: The app must allow the user to sign in after they have created a profile

We used the emulator to test this so as to make sure it was not being stored locally:



This signed me into the new user profile which featured no assignments added:



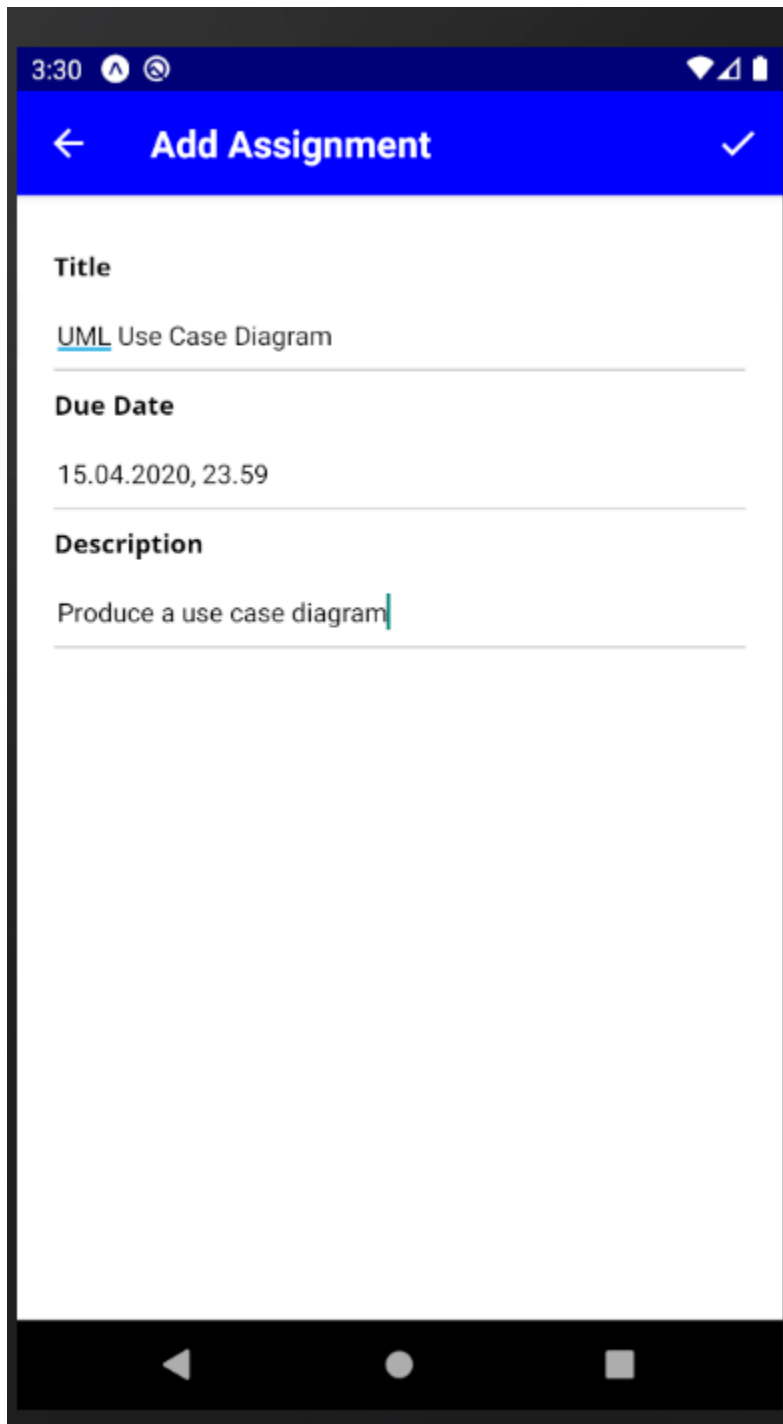
A warning was received about performance issues using a long timer for android phones, but this was anticipated as it was a timer for authentication and not something that would cause performance issues.

Requirement: **Fulfilled**

Test Case 3: **Passed**

#### Test case 4

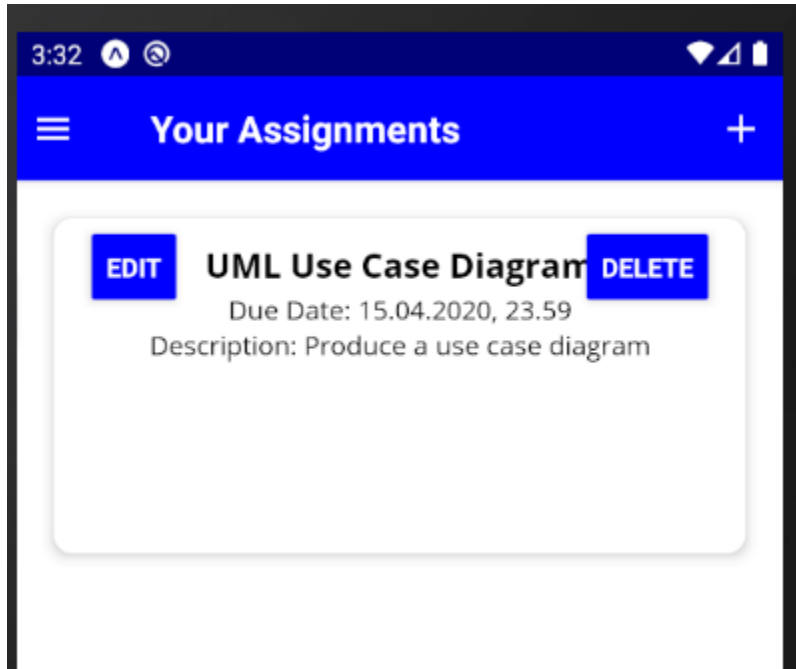
Requirement: The app must allow the user to upload an assignment:



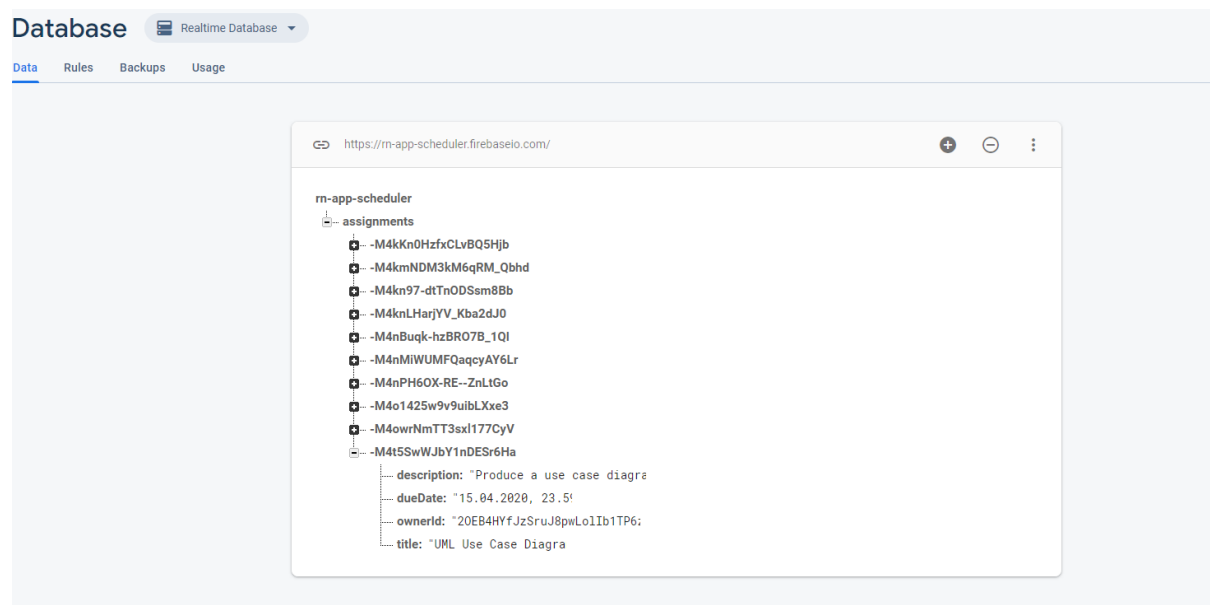
The screenshot shows a mobile application interface for adding an assignment. At the top, there is a blue header bar with a back arrow on the left, the text "Add Assignment" in the center, and a checkmark icon on the right. Below the header, the form consists of three sections: "Title" with the text "UML Use Case Diagram", "Due Date" with the text "15.04.2020, 23.59", and "Description" with the text "Produce a use case diagram". Each section is separated by a horizontal line. The bottom of the screen shows the standard Android navigation bar with a back arrow, a home circle, and a recent apps square.

We then clicked the tick to submit:

It showed up on the assignments page:



And on the database:



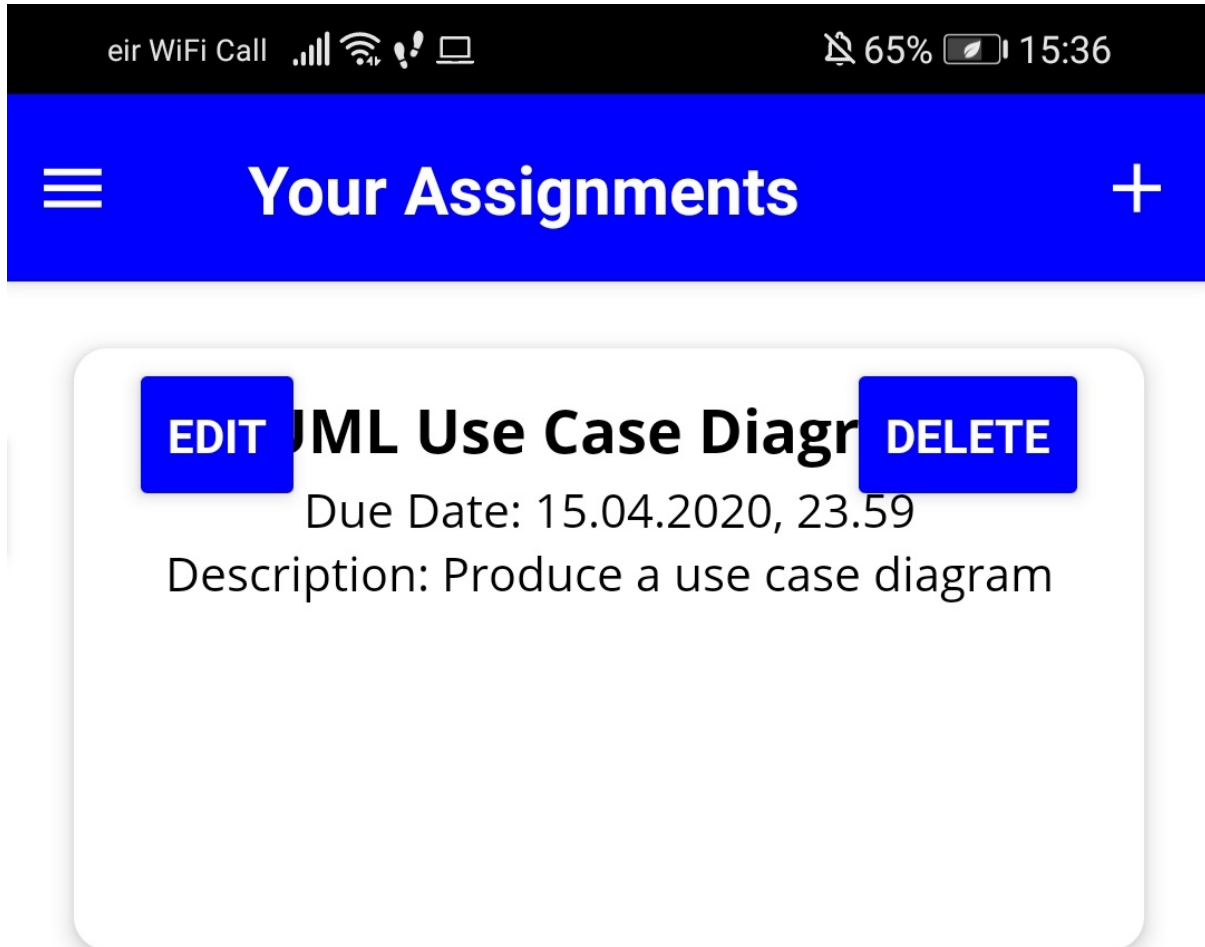
Requirement: **Fulfilled**

Test Case 4: **Passed**

### Test case 5

Requirement: The user must be able to view the assignment and it must persist even after the user has signed out

For this we signed out of the emulator and signed back into the app on the phone:



Styling issues aside, the Assignment object persists.

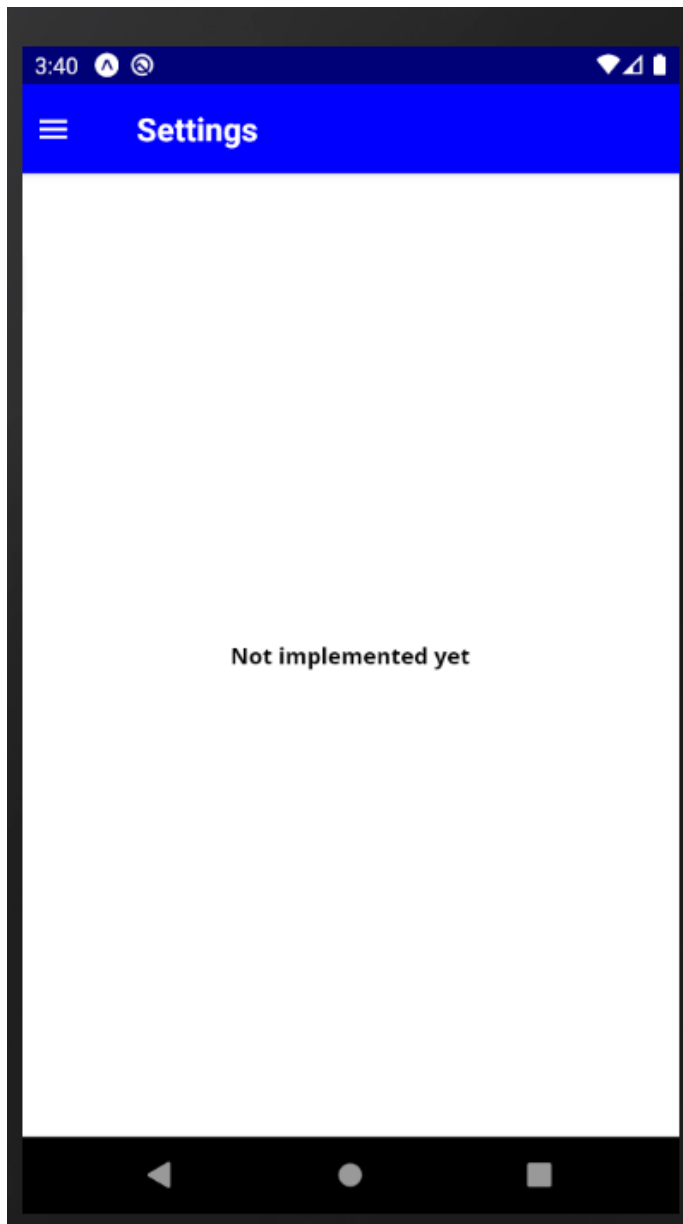
Requirement: **Fulfilled**

Test Case 5: **Passed**



**Test case 6 (not applicable)**

Requirement: The user must be able to set up push notifications to remind them to study:



Sadly, we ran out of time before we got around to implementing this requirement.

Requirement: **Unfulfilled**

Test Case 6: **Failed**

**Test Case 7 (not applicable)**

Requirement: The user must be able to set specific times when a push notification is sent

As with the last case, this was never implemented.

Requirement: **Unfulfilled**

Test Case 7: **Failed**

## 5. Challenges

There was a myriad of challenges involved in this project, but the two major limiting factors were related to scale of the project and lack of knowledge. As we are equally new to software development with no substantial experience in the realm of web development or mobile app development. In hindsight our pitch to our supervisor for this project seemed plausible and welcome challenge for us to be able to accomplish. As we took a step back and got fully into the development process, gaps in our knowledge couldn't be overlooked. React Native is habitually taken up by developers who have prior experience with React development on the web.

A long time was spent on simply getting the development environment set up. React Native is still young and growing, with lots of third-party contributions such as react navigation and various NPM packages, which meant we had a crash course in matching dependencies all by itself. Having emulators run the code, setting path variables, installing, uninstalling and reinstalling packages took up a large portion of the time which was not accounted for by ourselves.

Learning the fundamentals behind React Navigation was extremely time consuming and during development there was a major update to version 5, which depreciated what we had implemented with version 4 and triggered numerous bugs. These bugs arose through version mismatching between React Navigation and various NPM packages previously installed. We solved this by reverting to the well documented React navigation 3. Luckily we made some of that time back when we discovered we could use Firebase as our backend server and save ourselves writing multiple APIs but by that point we did not have enough time to implement the push notifications we had initially hoped would be a key feature.

Attempts were made in using the NPM package "React-Native-Document-Picker", to allow us to extract pdf files stored on our mobiles by extract the URI from the device and passing the information onto our assignment form to be uploaded to firebase. Unfortunately, we were unable to make this package to work to our liking and had to remove from the application.

Another goal of ours was to truly test the cross-platform capabilities but unfortunately neither of us had an apple/mac computer, so testing on the iOS platform was not viable. Arrangements were made with colleagues that had iOS compatible systems for us to conduct and test our app, but sadly the lockdown happened before we could get a chance to access such machinery so the app was only ever tested on windows and android platforms.

## 6. Conclusion

This project has thought us a huge amount, one of them sadly was the old adage of “you can have it fast, cheap or good”. Our project, what we have completed of it, is well constructed as we followed the documentation on best practices as outline in the official documentation, to produced a strong, well designed if albeit unfinished app.

Over the course of the project, we both learned new skills and technologies. Configuring dependencies is one of the most important aspects that we came away with. We also both became more efficient with JavaScript and the MVC design pattern.

Even though all of our desired functionality was unable to be implemented, it has given us a good knowledgebase and taste of what mobile app development fully entails. We are both proud and happy with what we have been able to produce. The entire experience and knowledge we have gained from undertaking this project is invaluable.

## References

- [1] Statista. 2020. *Smartphone Users Worldwide 2020* / Statista. [online] Available at: <<https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/>>
- [2] Hunt, P., 2013. *Why Did We Build React? – React Blog*. [online] Reactjs.org. Available at: <<https://reactjs.org/blog/2013/06/05/why-react.html>>
- [3] Coplien, T., 2009. *The DCI Architecture: A New Vision Of Object-Oriented Programming*. [online] Artima.com. Available at: <[https://www.artima.com/articles/dci\\_vision.html](https://www.artima.com/articles/dci_vision.html)>
- [4] Gackenhaimer, C., 2015. *Introduction To React*. Berkeley, CA: Apress.
- [5] Gackenhaimer, C., 2015. *Introduction To React*. New York: Apress, p.Using React to Build Scalable and Efficient User Interfaces.
- [6] GitHub. 2020. *Facebook/React*. [online] Available at: <<https://github.com/facebook/react>>
- [7] Antonio, C., 2015. *Pro React*. Apress, p. Inside the DOM Abstraction.
- [8] Search Engine Land. n.d. *What Is JSX*. [online] Available at: <<https://www.reactenlightenment.com/react-jsx/5.1.html>>
- [9] Eisenman, B., 2017. *Learning React Native, 2Nd Edition*. 2nd ed. [Place of publication not identified]: O'Reilly Media, Inc.
- [10] Stack Overflow. 2019. *Stack Overflow Developer Survey 2019*. [online] Available at: <<https://insights.stackoverflow.com/survey/2019>>
- [11] Noel, Y., 2018. *Xamarin Vs React Native: 2018 Comparison Guide For Developers* / Codementor. [online] Codementor.io. Available at: <<https://www.codementor.io/@stalky00/xamarin-vs-react-native-2018-comparison-guide-for-developers-o6oswmj88>>
- [12] GitHub Octoverse 2017. 2017. *Github Octoverse 2017*. [online] Available at: <<https://octoverse.github.com/2017/>>
- [13] Hansson, N. and Vidhall, T., 2016. Effects on performance and usability for cross-platform application development using React Native.

- [14] Axelsson, O. and Carlström, F., 2016. Evaluation targeting react native in comparison to native mobile development.
- [15] Kol, T., 2016. *Performance Limitations Of React Native And How To Overcome Them*. [online] Medium. Available at: <<https://medium.com/@talkol/performance-limitations-of-react-native-and-how-to-overcome-them-947630d7f440>>
- [16] Dabit, N., 2018. *React Native In Action*. New York: Manning Publications.
- [17] reactjs.org. (n.d.). *Thinking in React – React*. [online] Available at: <https://reactjs.org/docs/thinking-in-react.html>
- [18] Wikipedia. 2020. *Cascading Style Sheets*. [online] Available at: <[https://simple.wikipedia.org/wiki/Cascading\\_Style\\_Sheets](https://simple.wikipedia.org/wiki/Cascading_Style_Sheets)>
- [19] Speaker Deck. 2014. *React: CSS In JS*. [online] Available at: <<https://speakerdeck.com/vjeux/react-css-in-js>>
- [20] Docs.expo.io. 2020. *Layout With Flexbox - Expo Documentation*. [online] Available at: <<https://docs.expo.io/versions/latest/react-native/flexbox/>>
- [21] Redux.js.org. n.d. *Redux*. [online] Available at: <<https://redux.js.org/introduction/getting-started>>
- [22] Slides. n.d. *Redux. From Twitter Hype To Production*. [online] Available at: <<http://slides.com/jenyaterpil/redux-from-twitter-hype-to-production#/>>
- [23] Expo.io. n.d. [online] Available at: <<https://expo.io/>>
- [24] En.wikipedia.org. n.d. *Node.Js*. [online] Available at: <<https://en.wikipedia.org/wiki/Node.js>>
- [25] Node.js. n.d. *Docs / Node.Js*. [online] Available at: <<https://nodejs.org/en/docs/>>
- [26] Firebase. n.d. *Firebase Realtime Database*. [online] Available at: <[https://firebase.google.com/docs/database/?gclid=CjwKCAjwvtX0BRAFEiwAGWJyZNv5TpJwOmnYFxWajWwSzUmaJi\\_RxH4b5I9uMpTsPDhNAaQU05EZxxoCaPoQAvD\\_BwE](https://firebase.google.com/docs/database/?gclid=CjwKCAjwvtX0BRAFEiwAGWJyZNv5TpJwOmnYFxWajWwSzUmaJi_RxH4b5I9uMpTsPDhNAaQU05EZxxoCaPoQAvD_BwE)>
- [27] Wingerath, W., Gessert, F., Friedrich, S., Witt, E. and Ritter, N., 2017. The Case For Change Notifications in Pull-Based Databases. *Datenbanksysteme für Business, Technologie und Web (BTW 2017)-Workshopband*.

[28] Firebase. n.d. *Structure Your Database | Firebase Realtime Database*. [online] Available at: <<https://firebase.google.com/docs/database/web/structure-data>>

[29] Firebase. n.d. *Firebase Cloud Messaging*. [online] Available at: <[https://firebase.google.com/docs/cloud-messaging/?gclid=CjwKCAjwvtX0BRAFEiwAGWJyZOz\\_RZV1zNm0AZeYBD4qYP-n0giRW9KBFyrsK5NeMDh3WgqFNjvMexoCifYQAvD\\_BwE](https://firebase.google.com/docs/cloud-messaging/?gclid=CjwKCAjwvtX0BRAFEiwAGWJyZOz_RZV1zNm0AZeYBD4qYP-n0giRW9KBFyrsK5NeMDh3WgqFNjvMexoCifYQAvD_BwE)>

[30] Setting up the development environment · React Native. 2020. Setting up the development environment · React Native. [ONLINE] Available at: <https://reactnative.dev/docs/environment-setup>

[31] GeeksforGeeks. 2020. Sandwich Testing | Software Testing - GeeksforGeeks. [ONLINE] Available at: <<https://www.geeksforgeeks.org/sandwich-testing-software-testing/>>

[32] František Gažo. 2020. Native apps with Flutter and React Native? - INLOOPX - Medium. [ONLINE] Available at: <<https://medium.com/inloopx/native-apps-with-flutter-and-react-native-8d300805b3c6>>