



**NUI Galway**  
**OÉ Gaillimh**

# **Fitness Meal Plan Generator**

**Oisin McNally**

**14376701**

**Electronic & Computer Engineering**

**Final Year Project Thesis**

**2017 / 2018**

**Supervisor: Dr. Des Chambers**

## Abstract

This report discusses information related to the development of a Fitness Meal Plan Generator mobile application. It contains background research on some of the technologies and frameworks used in the project. The report continues with the tool chain used for the application and where each technology and framework is implemented. The implementation details describe how the project requirements were implemented in the finished product. The testing and evaluation section details the tests that were carried out during the development of the application, while the possibility for extension gives details on future possible development ideas if someone was to continue this project. The report ends with the conclusions where I give my own opinion on the project.

## Table of Contents

Abstract .....	2
Introduction .....	5
Health & Safety and Broader Societal Impact .....	6
Architecture Overview.....	7
Literature and Technology Overview .....	8
React Native .....	8
Managing Application Level State with Redux .....	10
Related Applications .....	11
MyFitnessPal.....	11
MyDietician .....	11
Fitness Meal Planner .....	12
Calculating Maintenance Calories and TDEE (Total Daily Expenditure).....	13
Tool chain .....	16
Front end .....	16
Back end .....	17
Project Requirements.....	18
Front end .....	18
Back end .....	18
Implementation Details .....	19
Login & Sign Up.....	19
Specification .....	19
Design .....	20
Main Menu.....	23
Specification .....	23
Design .....	24
Navigation.....	25
Question Form.....	27
Specification .....	27
Design .....	28
Firebase Fetch .....	30
Firebase Save.....	31
Food Preferences .....	32
Specification .....	32
Design .....	33
Add Food.....	35

Delete Food .....	37
Meal Plan & Meal .....	38
Specification .....	38
Design – Meal Plan .....	39
Design – Meal .....	42
Weight Statistics .....	44
Specification .....	44
Design .....	45
Group Manager .....	47
Specification .....	47
Design .....	48
Spring Boot RESTful Web Service .....	50
Introduction.....	50
Food Controller .....	53
Meal Controller.....	54
Recommender System.....	55
Meal Plan Controller.....	57
Evaluation & Testing .....	59
Possibility for Extension .....	62
Conclusion .....	63
Table of References .....	64

## Introduction

The project I have decided to pursue for my final year project is my self-proposed idea the “Fitness Meal Plan Generator”. The aim of this project is to create a mobile application that generates a tailored meal plan for an individual based on their personal information, goals and foods they enjoy. Their progress is then monitored and the feedback is returned to the user. The idea is to be able to emulate the role of a dietician or nutritionist. The front end of the application uses React Native [1] (JavaScript based framework by Facebook) as the application can be written in a single language and be compiled for Android and iOS. The back end of the application uses a Spring Boot RESTful web service to handle the more advanced application logic and retrieve information from databases.

## Health & Safety and Broader Societal Impact

This project does not carry any health and safety risk as there are no dangers such as machinery or high voltages involved. If this project formed a real world product I think there could be a risk of life and limb. If there was an option to create a meal plan for someone with diabetes the information would need to be correct or it may cause damage to their health which the developer could be at fault for. I also believe there would be a societal impact as this app tries to emulate the work of a dietician and therefore may help more people with their diet goals due to it being more affordable and easily accessible on a smartphone.

## Architecture Overview

### Front-end

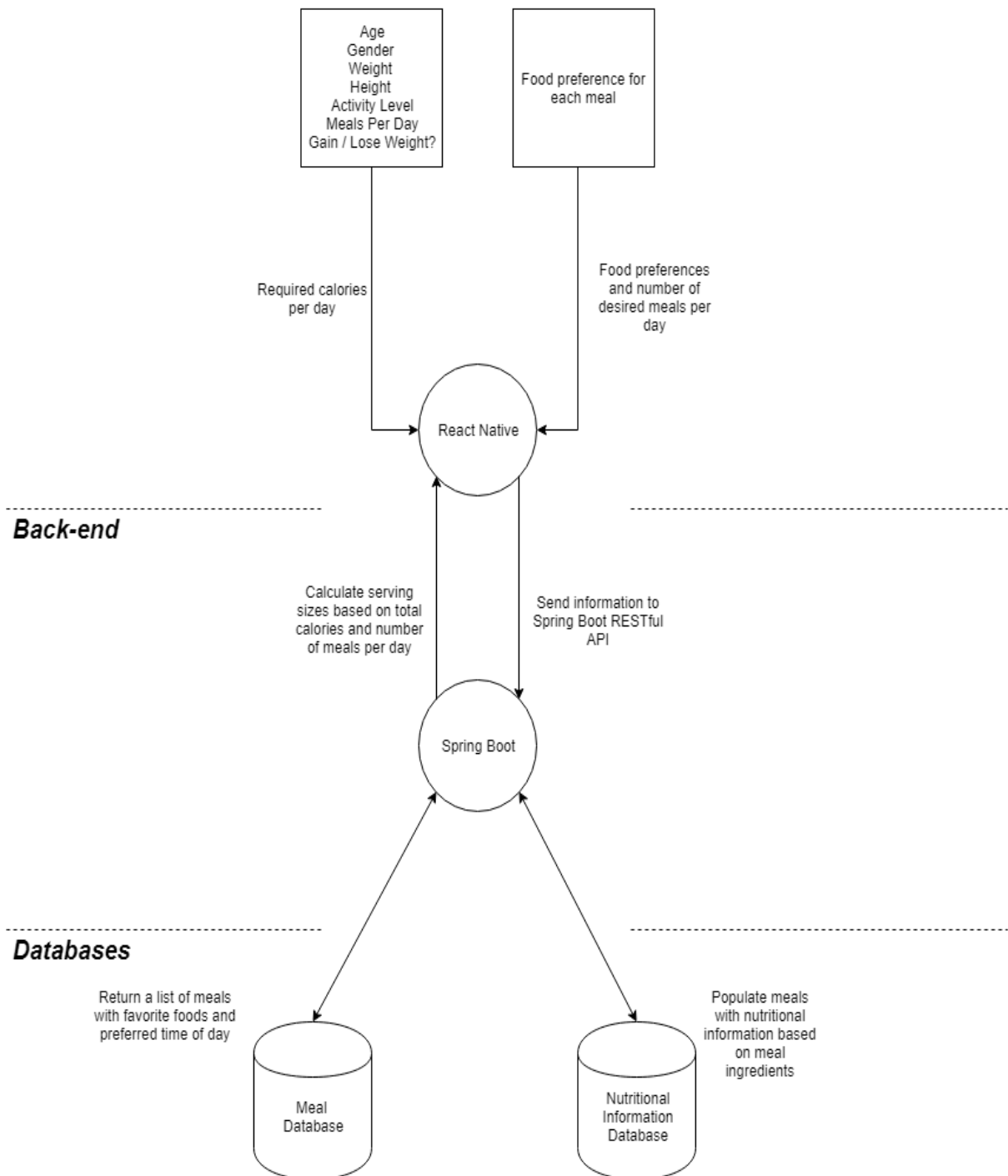


Figure 1 - Architecture overview diagram

## Literature and Technology Overview

### React Native

React Native is a JavaScript based framework used to develop mobile applications for both Android and iOS devices. React Native allows developers to develop an application in a single language known as JSX that can then be rendered on either platform. It achieves this by using a bridge to invoke the native UI components on either platform so the application uses real mobile UI components and not just web views. This bridge acts as an API between the JSX code and Objective-C for iOS or Java for Android which allows access to device features like the camera, Bluetooth and messaging services.

The main advantage of React Native is that one language can be used to develop a cross platform application which reduces the learning curve and development time. Native is not as resource intensive as it works separately from the main UI thread. A Native application can be composed of many views which will render if the state or properties of a component change.

Since a React Native application is composed of JavaScript files, live reloading can be used to see instant changes in the development process. With live reloading enabled, by simply hitting save in the code editor any changes to a file are acknowledged and updated in real time on the application. This means we do not have to wait for the entire application to rebuild to see changes recently implemented. This quick visual response to changes made is great for reducing development time.

React Native can be debugged through Google Chromes or Safari developer tools. This allows console logging of different objects or outputs from the application.

Apple allows JavaScript based changes to be implemented to an existing application with no additional review process which allows developers to make instant changes to their applications.



One of the disadvantages of React Native is that it is a relatively new framework that was released in 2015. Not all of the features of Android and iOS are supported which means you may need to use a 3<sup>rd</sup> party solution to some of the problems you encounter.

Overall React Native is an interesting new framework with huge benefits by being able to use a single language to develop a cross platform application without sacrificing quality or user experience. There may be a degree of risk as the framework is quite new and still being improved but the advantages greatly outweigh the disadvantages. [2]

## Managing Application Level State with Redux

Redux [3] is a predictable state container for JavaScript applications. It provides a stable way to manage application level state.

The state of the application is held inside a container called a store. To change any aspect of the application state an action with a command name must be created. This action is sent to a reducer which will change the application state based on the command in the action that was sent.

Asynchronous actions can be handled with the middleware Redux Thunk [4]. For example, if information was requested from a website, a promise would be instantly returned. The application state can then be updated differently based upon the promise being fulfilled (request was successful) or not (request failed). Therefore the state of the application does not have to be updated until the outcome of the event is known.

Because of the way that the state can only be changed through actions, there is a predictable way of manipulating it. There is also a history of how the state has been changed which can be used to revert back to a previous application state.

## Related Applications

### MyFitnessPal

MyFitnessPal [5] is a free calorie tracking app available on Android and iOS. It allows users to set a goal and provides a recommendation on the amount of calories that should be consumed per day. The user can add foods to categories such as breakfast, lunch, dinner and snacks by searching the MyFitnessPal food database or by scanning a barcode.

Users get a breakdown of the nutritional information of different foods and receive notifications if foods are high in protein, low in salt, high in saturated fat, etc. Statistics are provided on the amount of nutrients you have consumed in the day and through a weight tracking graph. The app can also log and track your exercise and steps with connectivity support for other apps and smart devices.

### MyDietician

MyDietician [6] is an app that connects the user with an actual registered dietician to evaluate their diet and provide feedback so the user can make dietary changes. A user will send a picture of the meal with some other information such as how much of the portion was consumed, pre and post meal hunger levels and stress levels at the time. A registered dietician then evaluates their weekly diet habits and will message them with appropriate advice.

### **Fitness Meal Planner**

Fitness Meal Planner [7] would be the most similar app to the one being developed for the final year project. Basic information about the user is taken such as age, height, weight, and gender to calculate the daily calorie requirements. Preferences such as amount of meals per day and allergies or food restrictions are then used to the calculated daily calories amongst your preferred number of meals and calculate the appropriate meals and serving sizes.

After a meal plan for the day has been generated there is an option to generate a shopping list with the ingredients needed for the day.

There are no options to track user statistics or input the food items you would like in each meal so the meal plan is not very user customisable. I also found some of the meals generated to have unusual combinations of ingredients such as eggs and rice cakes or soybeans with kidney beans and Brussels sprouts.

## Calculating Maintenance Calories and TDEE (Total Daily Expenditure)

The word macros (short for macronutrients) is used to explain the protein, carbohydrate and fat content of food. Macros and calories have a close relationship where 1g of protein or carbs is 4 calories and 1g of fat is 9 calories. Weight loss or weight gain comes down to first calculating how much calories your body needs to maintain its weight, or enough to provide energy for your daily activities. To gain weight you eat more calories than your maintenance amount and to lose weight you eat fewer calories than your maintenance amount.

However due to the uniqueness of each person where genetics, metabolism and other factors must be accounted for, calculating your maintenance calories using a formula should give you an accurate estimation of what they are likely to be but may need to be tweaked slightly to suit each person.

First we must calculate your REE (Resting Energy Expenditure) which is the amount of calories needed to maintain your bodyweight. This is then used to calculate your TDEE (Total Daily Energy Expenditure) based on your REE and your activity level.

To do this we can use “The Mifflin, M.D., St Jeor formula” [8] which is a popular and well respected formula to calculate your TDEE.

### For males

$$10 \times \text{weight (kg)} + 6.25 \times \text{height (cm)} - 5 \times \text{age (years)} + 5 = \text{REE}$$

### For females

$$10 \times \text{weight (kg)} + 6.25 \times \text{height (cm)} - 5 \times \text{age (years)} - 161 = \text{REE}$$

This gives us the amount of calories that your body needs on a daily basis to provide energy for all of its functions. However since most people have some level of activity in their life, we now go on to calculate a more realistic value which is the TDEE based upon your activity level.

### Sedentary

Just normal everyday activity like a little walking, a couple flights of stairs, eating, talking etc.  
(REE X 1.2)

**Light activity**

Any activity that burns an additional 200-400 calories for females or 250-500 calories for males more than your sedentary amount. (REE x 1.375)

**Moderate activity**

Any activity that burns an additional 400-650 calories for females or 500-800 calories for males more than your sedentary amount. (REE x 1.55)

**Very Active**

Any activity that burns more than about 650 calories for females or more than 800 calories for males in addition to your sedentary amount. (REE x 1.725)

The next step is to calculate how much protein, carbohydrates and fats the calories should be composed of. For this example we will use some common ratios.

- 1g of protein per pound of bodyweight
- 25% of TDEE calories for fats
- Remainder of calories used for carbohydrates.

If we have a 180lbs male with 3500Kcals TDEE the calculation would be as follows.

- Protein:  $1g \times 180lbs = 180g$  ( $180g \times 4kcal \text{ per gram} = 720kcal$ )
- Fat:  $3500kcal \times 25\% = 875kcal$  ( $9kcal \text{ per gram so } 875/9 = 97g$ )
- Carbohydrate:  $3500kcal - 720kcal - 875kcal = 1905kcal$  ( $4kcal \text{ per gram so } 1905/4 = 476g$ )

Therefore the concluding macros would be

- Protein: 180g / 720kcal
- Fat: 97g / 875kcal
- Carbohydrates: 476g / 1905kcal
- **Total TDEE: 3500kcal**

To gain weight we would have to eat more than this to an amount no greater than 20% which would be  $3500 \times 0.2 = 700$  and  $3500 + 700 = 4200\text{kcal}$ .

To lose weight this is applied in the reverse direction and we would have to eat less than this to an amount no greater than 20% which would be  $3500 - 700 = 2800\text{kcal}$ . [9]

## Tool chain

- React Native
- Redux (state management)
- Moment.js (JavaScript date manipulation) [10]
- Victory-native (graphs and charts) [11]
- Axios (JavaScript HTTP requests) [12]
- React-native-vector-icons (icon library) [13]
- react-native-router-flux (navigation) [14]
- Firebase (user authentication and statistics) [15]
- Spring Boot (RESTful web service) [16]
- MongoDB (JSON database) [17]
- Heroku (cloud hosting platform as a service) [18]

## Front end

I decided to use React Native to build the front-end of the mobile application. As mentioned earlier, React Native is a JavaScript based framework for building cross platform native mobile applications. There are alternatives to this such as Ionic, Apache Cordova and Phone Gap.

I opted for React Native due to it being an up and coming framework which is about 2 years old and backed by Facebook. I had some experience with React from my work placement where other developers and a software architect recommended the framework to me. Native also has a live reloading option allowing developers to see the effects of code changes on an application very quickly.

While React Native is used to build the individual views for the front end, Redux is used to manage the state of the application. Redux is a framework that is commonly used with React Native to predictably manage application state.

Navigation will be handled by react-native-router-flux. This is a package installed through npmJS which allows easy navigation between different views. Each view is assigned a key in a Router.js file and then `Actions.{key}` can be called anywhere in the project to load the view.



Firebase which is a cloud API backed by Google handles user authentication. It allows authentication through e-mail and common social media platforms like Facebook and Twitter and is easily integrated with React Native. User objects are stored in a JSON database. Statistics are also displayed with information about users such as gender, age, device OS and country.

React-native-axios will be used to handle HTTP requests to the RESTful API backend. It is installed through npmJS [19] and converts request and response data to JSON automatically. It has support for the Promise API where a HTTP request returns a promise object and the promise object is then fulfilled when the asynchronous request is complete.

There are many other packages listed above that are used to provide functionality for other aspects of the application such as date manipulation, graphs & charts and icons.

## Back end

The backend of the application is a RESTful web service that will be built using Spring Boot. While on work placement I primarily worked on the development of a RESTful web service using Spring Boot with my team so there will not be much of a learning curve involved to get familiar with it again. Spring Boot is an opinionated version of Spring where development can be started with very little configuration. There is an embedded Tomcat so WAR files don't need to be deployed separately.

A database will be created with MongoDB to store data in different repositories. The database and web service will be deployed on the cloud hosting site Heroku and connected by placing a URI link to the database in the properties file for the web service.

## Project Requirements

### Front end

- Login screen that authenticates a user using Firebase.
- Sign Up screen to create a user and store their details in the Firebase database.
- A questionnaire screen to calculate the user's daily calorie requirements based on their personal information, goals and how many meals they want per day.
- A screen with a container for each meal where desired ingredients can be added and removed separately to each one.
- Homepage view containing buttons to navigate the user to the different views of the application.
- View / edit meal plan screen where a meal plan with portion sizes is generated. Meals can be replaced and portion sizes can be edited.
- User statistics screen shows how well a user has tracked to their goals.
- Manager screen to track groups of users and see statistics about the group.

### Back end

- Develop RESTful API endpoints to fulfil tasks that are difficult for the front end. Endpoints are stored in a controller class that receive / return data in JSON.
- Endpoint and service development to handle retrieval of food items and meals from database.
- Endpoint and service development to store user feedback in the MongoDB database for use in the recommender system.
- MongoDB database to store meal, food and recommender system data.

## Implementation Details

### Login & Sign Up

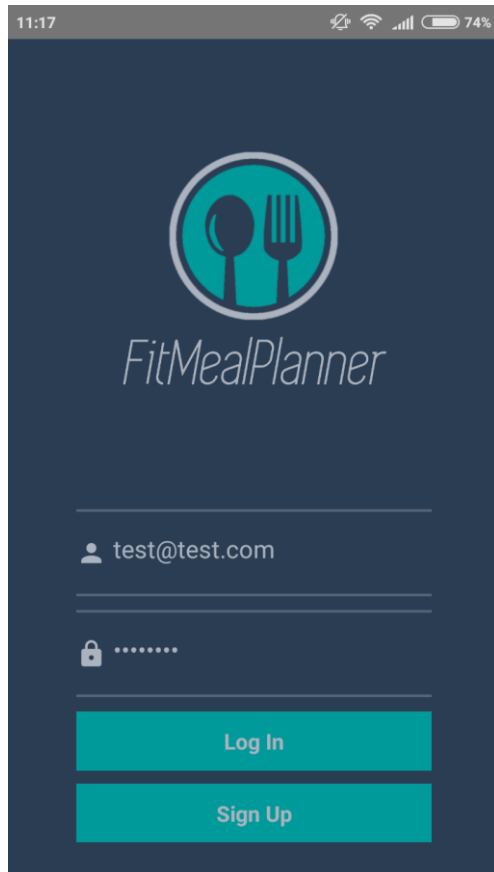


Figure 2 - Login Screen

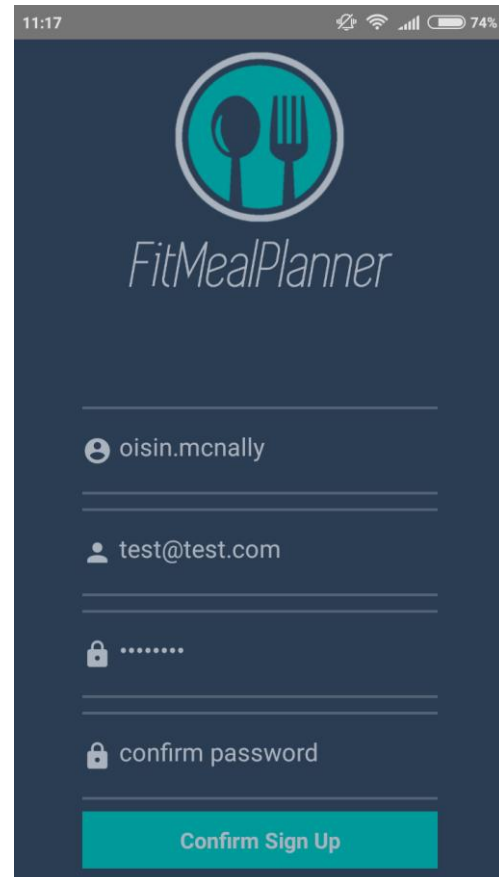


Figure 3 - Sign Up Screen

### Specification

The goal of the login and sign up screen is to authenticate a user with the Firebase server or create an account if the user is not already signed up. The user enters their account email and password which is sent to the Firebase server on press of the “Log In” button. By pressing the “Sign Up” button a new screen will open. The user completes a form with their username, email and password which is used to create a new account on the Firebase server on press of the “Confirm Sign Up” button.

## Design

Using React Native, each component is built inside of a view. With the login screen there are 2 sub views inside of the main view. The size of these views is configured using CSS Flexbox where each sub view occupies a ratio of the main view.

```
const styles = {  
  logoContainer: {  
    marginTop: 50,  
    flex: 1,  
    justifyContent: 'center'  
  },  
}
```

Figure 4 - CSS Similar Style for Logo View

In this example each view is assigned a flex of 1. Therefore the total flex of the screen is sum of the flex of each sub view ( $1 + 1 = 2$ ). Each sub view then occupies a ratio of  $\frac{1}{2}$  of the space for the main view. If a device with a bigger screen were to run the application the components would scale up to match the screen size and also maintain their aspect ratio.

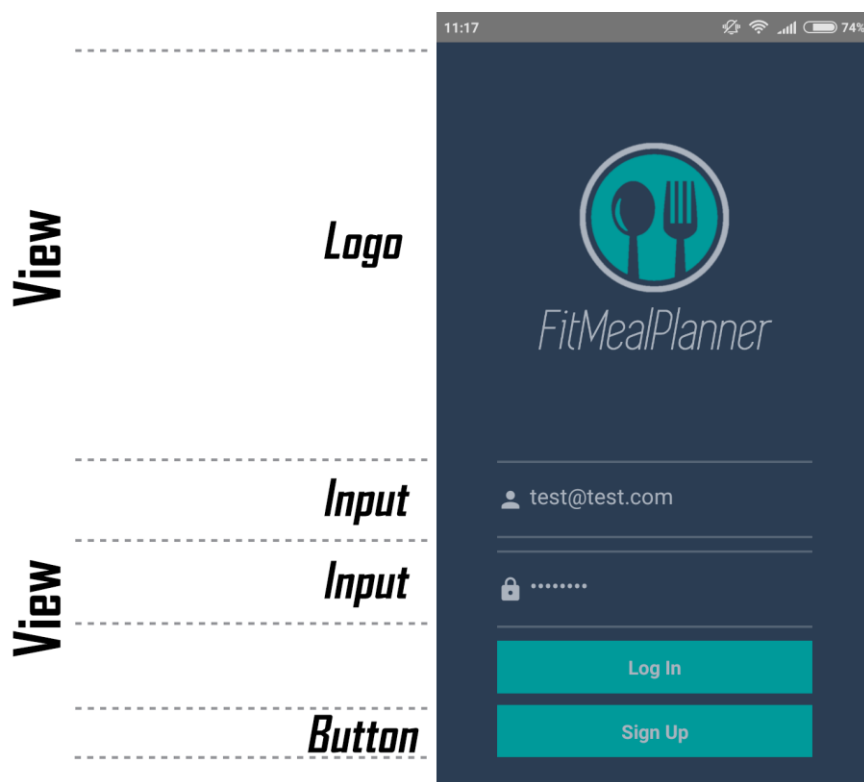


Figure 5 - Component layout of Login screen

The state of the component is managed with Redux and the decision process of the login component can be seen below.

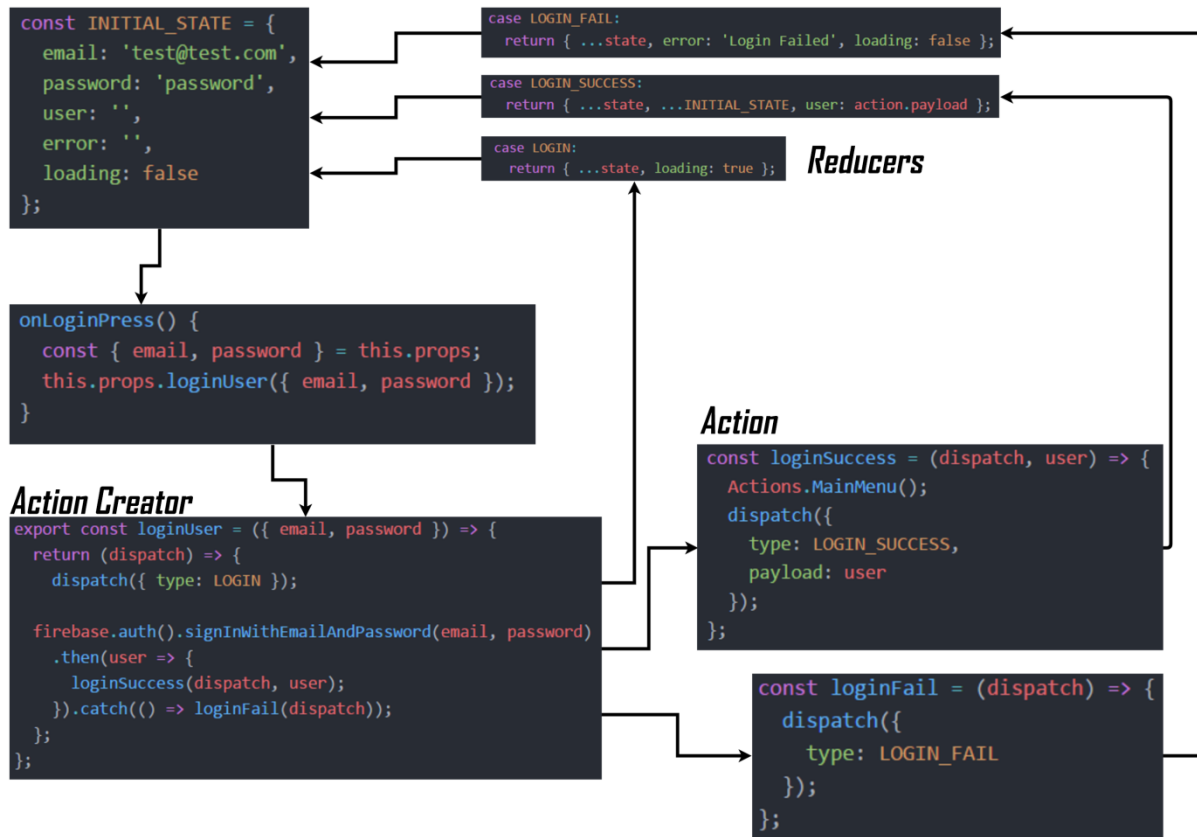


Figure 6 – State change process with Redux for user login

The state of the authentication component is managed by a Redux store. Each time a change is made to the email and password field the state is updated with the text in each input box.

When the login button is pressed an onPress handler for the button will call the function “onLoginPress()”. The username and password are retrieved from the store and sent to the action creator.

The login process is asynchronous meaning an immediate response is not available and the application must wait on the response of the login attempt from Firebase. An action of “type: LOGIN” is dispatched which updates the state to set loading to true. This causes a spinner to appear on the login screen.

If the login is successful, “*Actions.MainMenu()*” will cause the main menu screen to open. An action is dispatched to update the user element of the state to store the info related to the user currently logged in. If the login fails then the error piece of state is updated with a message and the loading is set to false indicating the request is complete but was not a success.

User sign up follows the same process except instead of calling the Firebase method

“*signInWithEmailAndPassword()*” the method “*createUserWithEmailAndPassword()*” is called instead.

```
firebase.auth().createUserWithEmailAndPassword(email, password)
```

Figure 7 - Firebase method to create a new user

## Main Menu

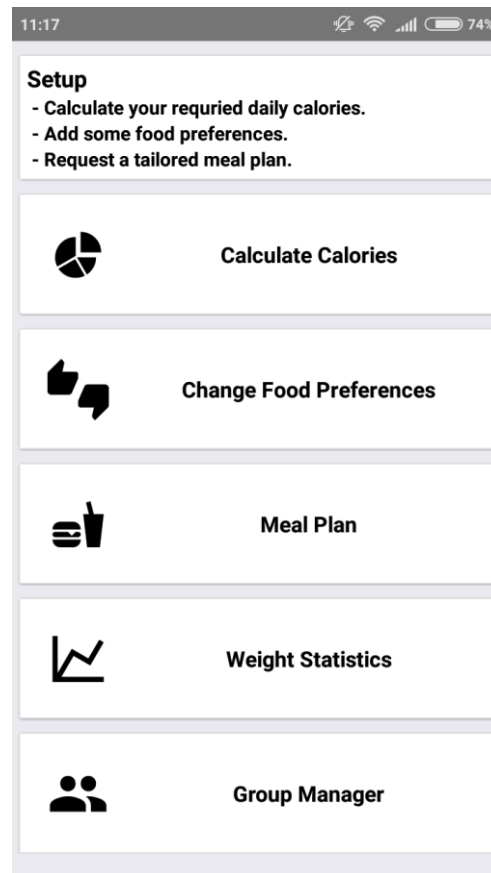


Figure 8 - Main Menu Screen

## Specification

To build a main menu that is displayed upon a successful login and show the user a small tutorial on how to get started with the application. There is a button displayed for each menu option that will navigate the user to their desired screen when pressed.

## Design

The design is very simple and the main objective is to allow the user to navigate between the different components of the app.

The layout of this screen uses Card and CardSection components. A Card is an elevated section with a bordering edge to give the appearance of a Card. CardSections can then be used inside of the Card to house components. Each button is located inside a CardSection with a flex of 1. This means that the button will occupy the entire space of the CardSection.

Components such as the Card, CardSection and Button are fundamental building blocks and are re-used throughout application. They are designed in such a way to be customisable so they can be re-used instead of being re-created for each separate component.

```
<Card>
  <CardSection>
    <Icon style={styles.iconStyle} name='chart-pie' size={45} color='black' />
    <Button onPress={this.openQuestionForm.bind(this)}>Calculate Calories</Button>
  </CardSection>
</Card>
<Card>
  <CardSection>
    <Icon style={styles.iconStyle} name='thumbs-up-down' size={45} color='black' />
    <Button onPress={this.openFoodPreferenceList.bind(this)}>Change Food Preferences</Button>
  </CardSection>
</Card>
```

Figure 9 - JSX Code to render the top 2 buttons of the menu



## Navigation

The main menu is where most of the applications navigation takes place. Navigation is handled using a library called *react-native-router-flux*. Each button has an `onPress` handler which will call a function to navigate the user to a new screen.

```
openQuestionForm() {  
  Actions.QuestionForm();  
}  
  
openFoodPreferenceList() {  
  Actions.FoodPreferenceList();  
}
```

Figure 10 – Functions called by the `onPress` handler for buttons on the main menu

Each component is registered to a Scene and added to a Stack. The `LoginForm` is at the top of the stack and will be the first component displayed when the application opens. By calling the method “`Actions.{key}`” e.g. “`Actions.QuestionForm()`” the `QuestionForm` component is added to the top of the stack and the components screen is displayed to the user. “`Actions.pop()`” can be called to pop the component from the stack and return to the previous component.

```
const RouterComponent = () => {  
  return (  
    <Router>  
      <Stack key="root">  
        <Scene key="LoginForm" component={LoginForm} hideNavBar />  
        <Scene key="SignUpForm" component={SignUpForm} hideNavBar />  
        <Scene key="MainMenu" component={MainMenu} title="Main Menu" hideNavBar />  
        <Scene key="QuestionForm" component={QuestionForm} title="Calculate Calories" />  
        <Scene key="FoodPreferenceList" component={FoodPreferenceList} title="Food Preferences" />  
        <Scene key="AddFoodList" component={AddFoodList} title="Add Food List" />  
        <Scene key="DeleteFoodList" component={DeleteFoodList} title="Delete Food List" />  
        <Scene key="MealPlan" component={MealPlan} title="Meal Plan" />  
        <Scene key="ListOfMeals" component={ListOfMeals} />  
        <Scene key="EditMeal" component={EditMeal} title="Edit Meal" />  
        <Scene key="WeightStats" component={WeightStats} title="Weight Statistics" />  
        <Scene key="GroupManager" component={GroupManager} title="Group Manager" />  
        <Scene key="UserInfo" component={UserInfo} title="User Info" />  
      </Stack>  
    </Router>  
  );  
};
```

Figure 11 - JSX code assigning each component to a Scene for navigation

The title attribute is used to display a text title on top of the screen and the hideNavBar attribute prevents a back button from appearing. This is used for component like the MainMenu where we do not want to allow the user to return to the login screen after they have logged in.

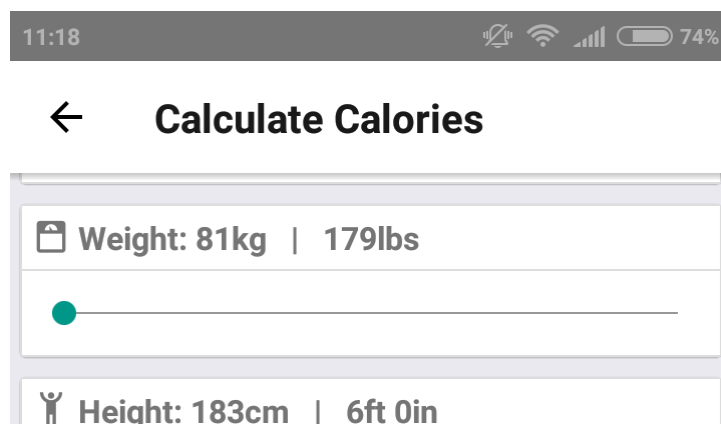


Figure 12 - Example of title attribute being used with a Scene

## Question Form

The screenshot shows a mobile app interface for calculating calories. The title bar at the top is grey with a back arrow and the text 'Calculate Calories'. The status bar above it shows the time 11:18, signal strength, Wi-Fi, and 74% battery. The main content area is a light grey card with several sections. On the left, a vertical dashed box labeled 'Card' encompasses the entire card. Two horizontal dashed boxes labeled 'CardSection' are positioned over the top two sections of the card. On the right, various component labels are placed next to their corresponding UI elements: 'Text' for the weight input, 'Slider' for the weight slider, 'Text' for the height input, 'Slider' for the height slider, 'Picker' for the activity level dropdown, 'Picker' for the goal dropdown, 'Text' for the calories result, 'Button' for the 'Calculate Calories' button, and 'Button' for the 'Save' button.

Component	UI Element
Text	Weight: 81kg   179lbs
Slider	Weight slider
Text	Height: 183cm   6ft 0in
Slider	Height slider
Picker	Activity Level: Very active (6-7 days/wk)
Picker	Goal: Gain weight (light)
Text	Calories: 3300Kcal
Button	Calculate Calories
Button	Save

Figure 13 - Component layout of QuestionForm screen

## Specification

Build a screen to collect information from the user. This information is used in a formula to determine the daily calories a user needs to achieve their goals. This value is required to determine the portion sizes in the user's meal plan.

## Design

Components are usually built inside a single view as seen with the login screen. In this case a ScrollView is used to allow the user to scroll between the different parts of the form.

Values such as height and weight are rendered in multiple metrics for ease of understanding. This is achieved by rendering the value in one format and also rendering the result of a function where the value is converted to a new metric and returned.

```
cmToFt(num) {  
  const ft = ((num * 0.393700) / 12);  
  const roundFt = Math.floor(ft);  
  const inches = Math.round((ft - roundFt) * 12);  
  return `${roundFt}ft ${inches}in`;  
}
```

Figure 14 - JavaScript function to convert centimetres to feet

The component has its own Redux store to keep track of all values entered by the user. When a Slider is moved or a new Picker option is selected the state will update to reflect this change. The value of each piece of information in state is retrieved and displayed alongside the text.

```
<Text>Weight (kg): {this.props.weight}</Text>
```

Figure 15 - JSX code to render the users weight from the Redux store

This is the first time we have encountered props. Props refers to properties which can be passed to a component as an argument when it is instantiated. Redux works by mapping the state of attributes saved in the store of the component to the component props. This allows any piece of state in the components Redux store to be accessed by calling this.props.{nameOfAttribute} e.g. this.props.weight.

Each Slider or Picker has an `onValueChange` listener so that when a value is changed a function can be called with the new value.

```
onValueChange={weight => this.props.updateQuestionForm({ prop: 'weight', value: weight })}
```

Figure 16 - `onValueChange` listener for a Slider to update the stored value for a user's weight

Each time the state of a component is changed it must be re-rendered to reflect the change and show the updated value to the user. The cycle of updating the state can be seen below.

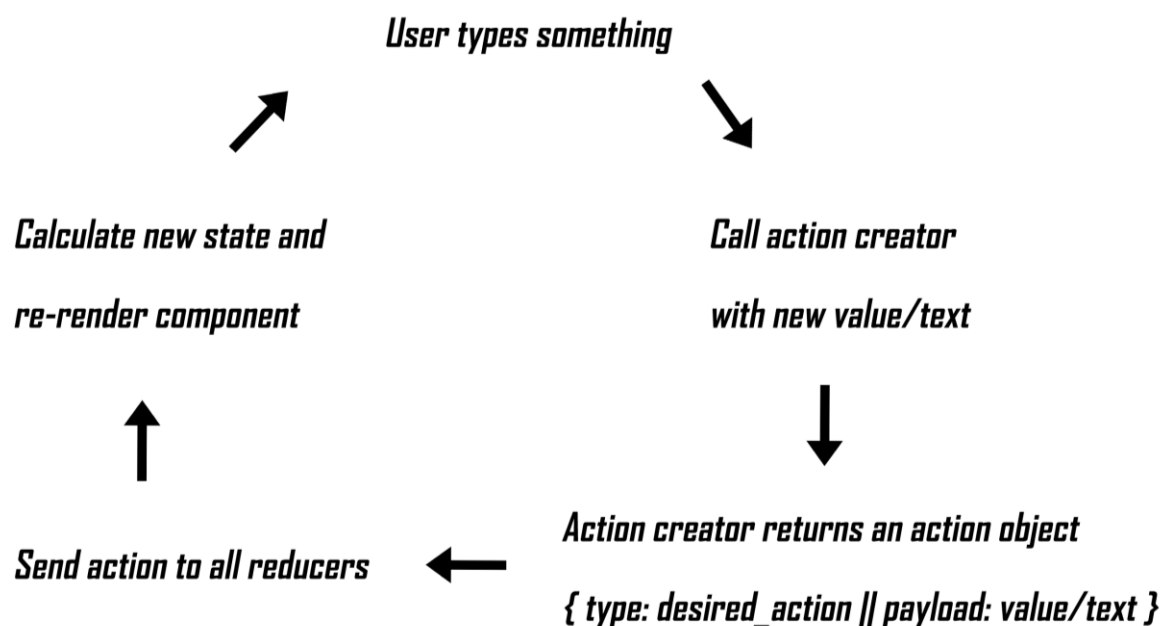


Figure 17 - Lifecycle of component state change

## Firestore Fetch

### Redux State

```
const INITIAL_STATE = {
  error: '',
  loading: false,
  // QuestionForm
  questionForm: {
    gender: '',
    age: '',
    weight: '',
    height: '',
    activityLevel: '',
    goal: '',
    maintenanceCalories: '',
    goalCalories: ''
  }
};
```

```
case FETCH_QUESTION_FORM:
  return { ...state, questionForm: action.payload, loading: false };
case FETCH_QUESTION_FORM_ERROR:
  return { ...state, error: action.payload, loading: false };
```

### Reducer

### ComponentWillMount()

```
componentWillMount() {
  this.props.questionFormSetLoading();
  this.props.fetchQuestionForm();
}
```

### Action Creator

```
export const fetchQuestionForm = () => {
  const { currentUser } = firebase.auth();
  return (dispatch) => {
    firebase.database().ref(`users/${currentUser.uid}/questionForm`)
      .on('value', snapshot => {
        if (snapshot.val() !== null) {
          dispatch({ type: FETCH_QUESTION_FORM, payload: snapshot.val() });
        } else {
          dispatch({ type: FETCH_QUESTION_FORM_ERROR, payload: 'Firestore Error! No entry exists...' });
        }
      });
  };
};
```

### Firestore Entry

```
finalyearproject-979ce
├── users
│   └── KbfNNS6aBealWAmGJx4KtG4FcPx2
│       └── foodPreferences
│           └── questionForm
│               ├── activityLevel: "1.725"
│               ├── age: 21
│               ├── gender: "Male"
│               ├── goal: "500"
│               ├── goalCalories: 3500
│               ├── height: 170
│               ├── maintenanceCalories: 3000
│               └── weight: 78.7
```

Figure 18 - Process flow when fetching data from Firestore

React lifecycle methods will be called automatically depending on certain changes to the component. `ComponentWillMount` is a React lifecycle method that is called before the component first mounts to the screen. The loading piece of state is set to true and a Spinner is displayed. An action is called that attempts to fetch the users information from the Firestore database.

If there is a Firestore entry with the information it is returned as a JavaScript object and the state is updated with the fetched values. If there is not a Firestore entry an error is returned alerting the user that no entry exists.

## Firestore Save

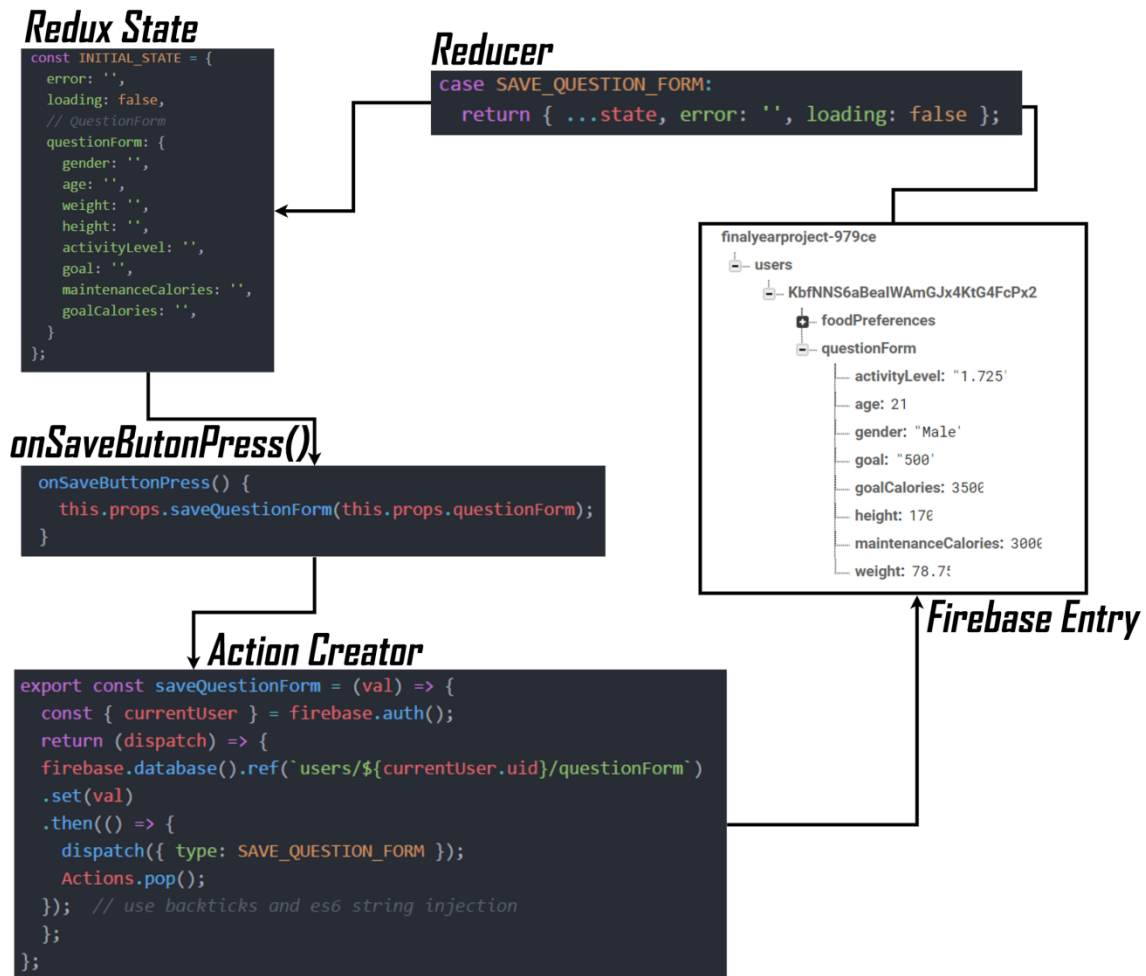


Figure 19 - Process flow when saving data to Firestore

When the save button is pressed the `onPress` handler for the button is called. This calls the function “`onSaveButtonPress()`” which calls an action creator with the state of the `QuestionForm` component as an argument.

The action creator will attempt to save the state of the `QuestionForm` component to Firestore and then call “`Actions.pop()`” which will cause the application to return to the main menu. The state of the component is then reset where `loading` is set to `false` and any error text is cleared.

## Food Preferences

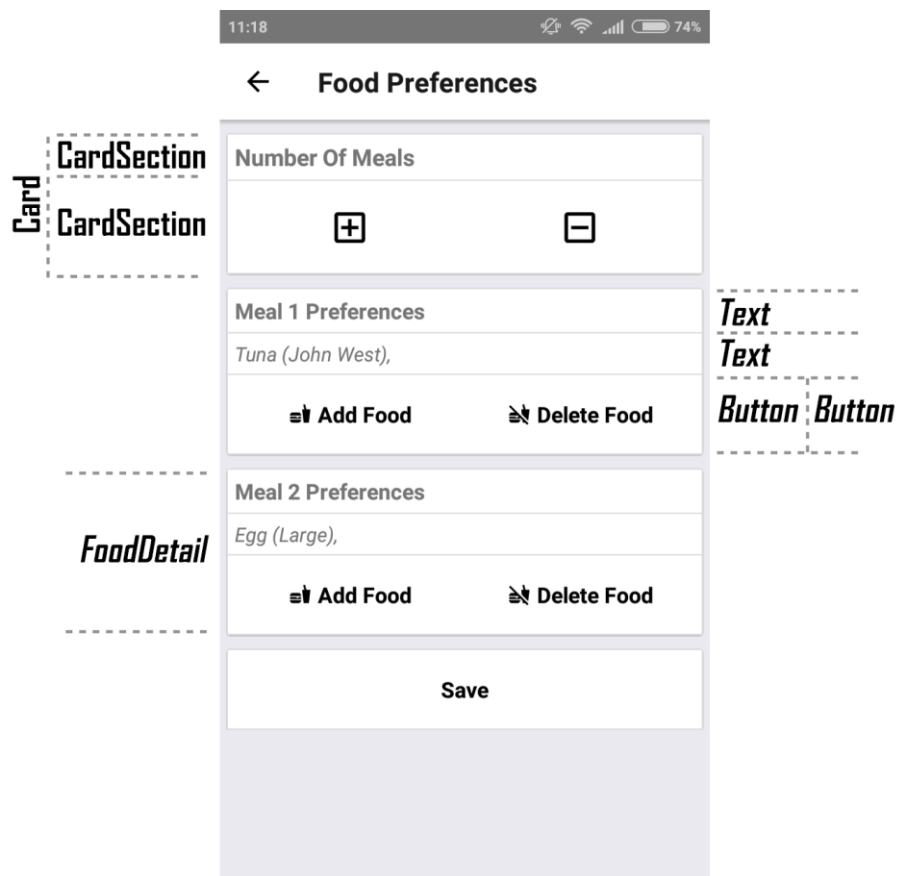


Figure 20 - Component layout of FoodPreferences screen

## Specification

Design a component that allows the user to specify a desired food for each meal. Buttons with a + and – icon are used to create / delete new instances of a FoodDetail component. The FoodDetail component allows users to add / remove foods from the list of preferences for the meal and renders the preferences previously selected.



## Design

All sub components are rendered within a ScrollView allowing the user to scroll on the screen as the number of components increases. The buttons with the + and – icons are used to add and delete meal preference boxes on the screen. When the user presses a + or - button the value for numberOfMeals in the Redux store is updated. The number of FoodDetail components rendered is dependent on this value.

```
renderFoodDetails() {  
  const foodDetails = [];  
  for (let i = 0; i < this.props.numberOfMeals; i++) {  
    foodDetails.push(<FoodDetail key={i} mealNumber={i + 1} />);  
  }  
  return foodDetails;  
}
```

Figure 21 - JSX code to create / delete FoodDetail components

This component is similar to the QuestionForm screen where a Firebase fetch is performed before the component mounts using the React lifecycle method `ComponentWillMount`. If the user has previously saved their meal preferences it will be retrieved from Firebase and displayed. When the save button is pressed the user's changes stored in the Redux store will be saved to Firebase under their id of the logged in user.

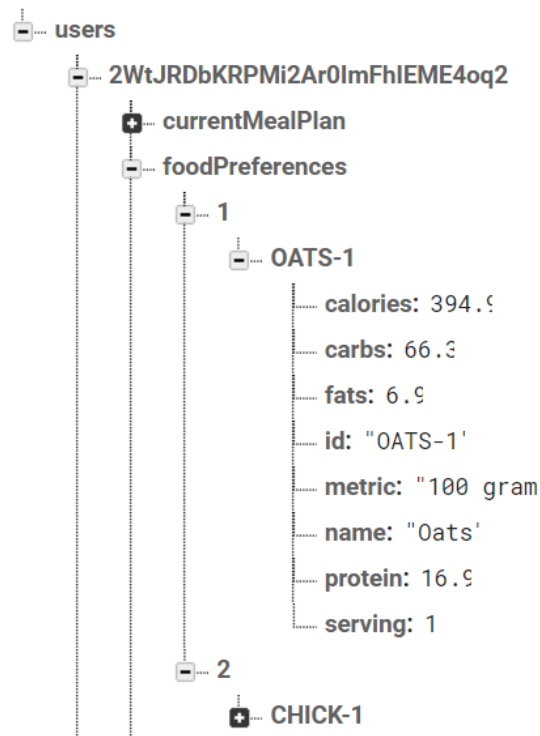


Figure 22 - Example of food preferences save in the Firebase Database

## Add Food

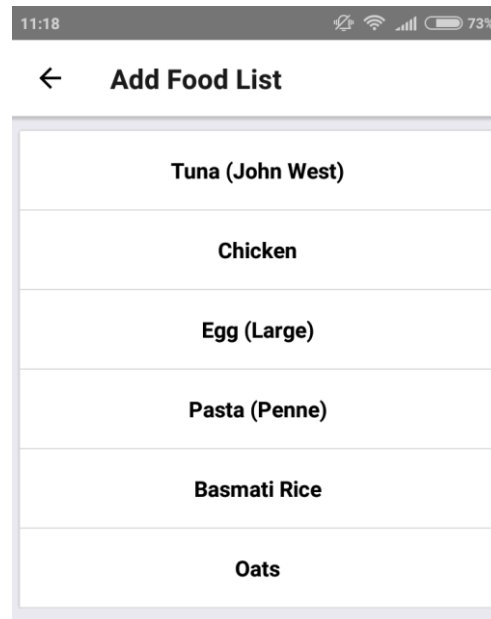


Figure 23 - Add Food screen

When the “Add Food” button is pressed an onPress handler calls a function that navigates the user to the AddFoodList component. This component calls a function before it is mounted using the React lifecycle method ComponentWillMount that will send a HTTP GET request to the Spring Boot RESTful web service back end and return a list of all foods from the food inventory table of the MongoDB database.

```
export const requestFoods = () => {  
  return (dispatch) => {  
    dispatch({ type: FOOD_PREFERENCES_SET_LOADING });  
    axios.get(`${Config.BASE_URL}/foods/all`)  
    .then(response => {  
      console.log(response);  
      dispatch({ type: FETCH_FOODS_DB, payload: response.data });  
    })  
    .catch(error => {  
      console.log(error);  
      dispatch({ type: FETCH_FOOD_PREFERENCES_ERROR, payload: 'API Error! Could not complete request...' });  
    });  
  };  
};
```

Figure 24 - Method to send HTTP GET request to return food inventory

The payload of the HTTP response is in JSON format and the JavaScript map function is used to render a button for each element in the JSON response.

```
renderFoods() {  
  return (  
    Object.values(this.props.foods).map(food =>  
      <CardSection>  
        <Button onPress={this.updateSelectedFoods.bind(this, food)}>{food.name}</Button>  
      </CardSection>  
    )  
  );  
}
```

Figure 25 - Method to render a button for each element in the JSON response

Each button has an onPress handler that calls the method “*updateSelectedFoods()*”. Depending on which button is pressed the appropriate food object will be passed as a parameter to the function and stored in the component state. When the state is updated the component re-renders and the selected food is displayed on the screen to the user.

## Delete Food

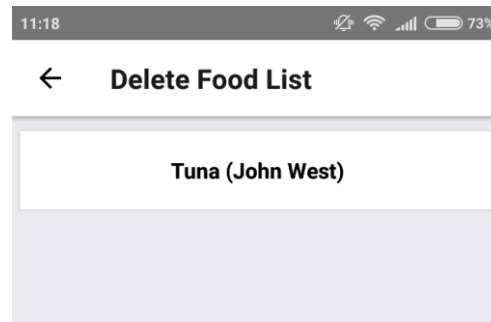


Figure 26 - Delete Food screen

When the “Delete Food” button is pressed an onPress handler calls a function that navigates the user to the DeleteFoodList component. This component follows the same method of operation as the AddFoodList component except a list of already selected foods is rendered as button components instead of performing a HTTP GET and rendering all foods from the food inventory table of the MongoDB database. This method is also called automatically before the component mounts using the React lifecycle method ComponentWillMount.

When a button with the food name is pressed a function is called with the appropriate food object as a parameter and it is deleted from the list of selected foods. The component is re-rendered and the screen is updated to reflect the change to the user.

## Meal Plan & Meal

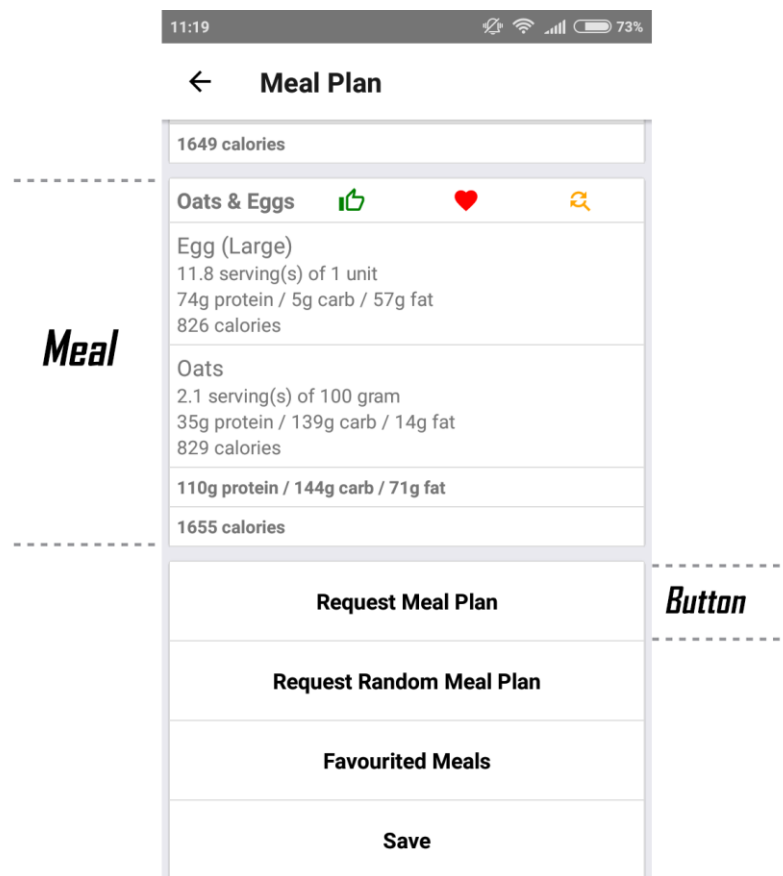


Figure 27 - Component layout of MealPlan screen

## Specification

To design a component allowing a user to request a meal plan based on their required calories, number of meals per day and food preferences for each meal. It should display each meal in a separate container with buttons allowing a user to like, favourite and find similar. When a meal is pressed on, a user should be able to change the serving sizes or swap it with one of their favourites.

## Design – Meal Plan

A meal plan is composed of a list of Meal objects. Before the component mounts to the screen a function is called to retrieve the current meal plan from the Firebase database within the React lifecycle method `ComponentWillMount`. If no meal plan is found on Firebase then an error is shown to the user.

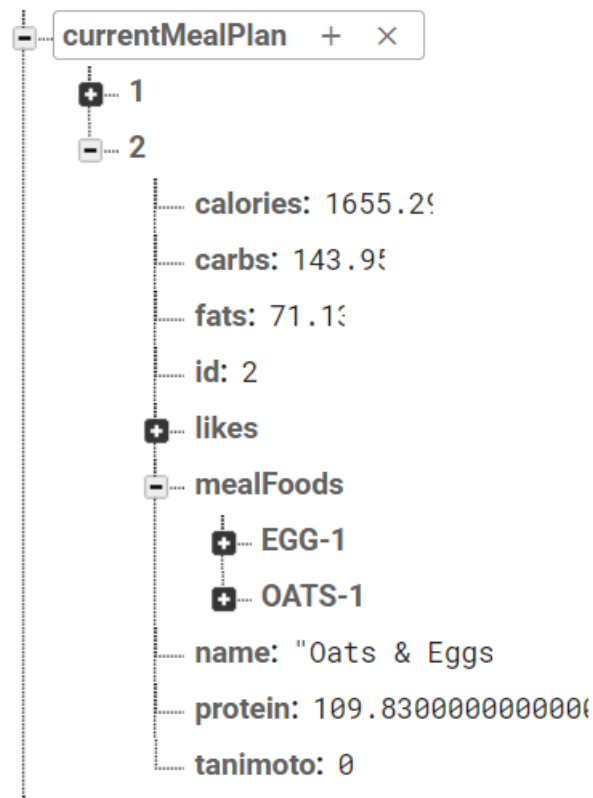


Figure 28 - Meal Plan entry from the Firebase database with the 'Oats and Eggs' meal object expanded

If a meal plan is returned the JSON snapshot is saved to the components Redux store and the JavaScript map function is used to iterate over each JSON entry and render it as a Meal component. There are many variations of the Meal component that could be rendered. If the component was able to be edited (touchable) or had been added to favourites then a slightly altered version of the Meal component would be rendered instead.

```
return (  
  <TouchableOpacity key={meal.id} onPress={this.onMealPress.bind(this, index)}>  
    <Meal meal={meal} favourited favouritedList={favouritedList} admin={admin} />  
  </TouchableOpacity>  
);
```

Figure 29 - JSX code to render a touchable (editable) Meal component

The “Request Meal Plan” button is used to construct a meal plan from meals in the MongoDB database. A HTTP POST is sent to a URL endpoint in the Spring Boot web service with the desired daily calories and number of meals in the URL as path variables. The food preferences for each meal are attached as a JSON payload. The meal plan is returned in JSON format, saved in the components Redux store and rendered on the screen as a list of Meal components. If the user presses the “Save” button the meal plan is saved to the Firebase database, else it is discarded.

```
/**  
 * Make an API request for a meal plan  
 * Must have port forwarding set up on chrome if web service is on localhost  
 */  
export const requestMealPlan = ({ numberOfMeals, calories, foodPreferences }) => {  
  return (dispatch) => {  
    dispatch({ type: MP_START_LOADING });  
    axios.post(`${Config.BASE_URL}/mealplan/${calories}/${numberOfMeals}`, foodPreferences)  
      .then(response => {  
        dispatch({ type: MP_SET_MEAL_PLAN, payload: response.data.mealPlan });  
      })  
      .catch(error => {  
        console.log(error);  
        dispatch({ type: MP_SET_ERROR, payload: 'API Error! Could not complete request...' });  
      });  
  };  
};
```

Figure 30 - Function to perform a HTTP POST requesting a meal plan from the web service



The “Request Random Meal Plan” button performs the same operation as described above but uses a HTTP GET which requests the web service to construct a random meal plan and return it in JSON format. This is because there is no need to send the food preferences for each meal as a JSON payload so a HTTP POST is not required.

The “Favourited Meals” button navigates the user to a screen displaying a list of meals that they have added to their favourites. By pressing the heart logo on the header of the meal it will fill red if the meal is in the users favourites or has no fill if it is not.

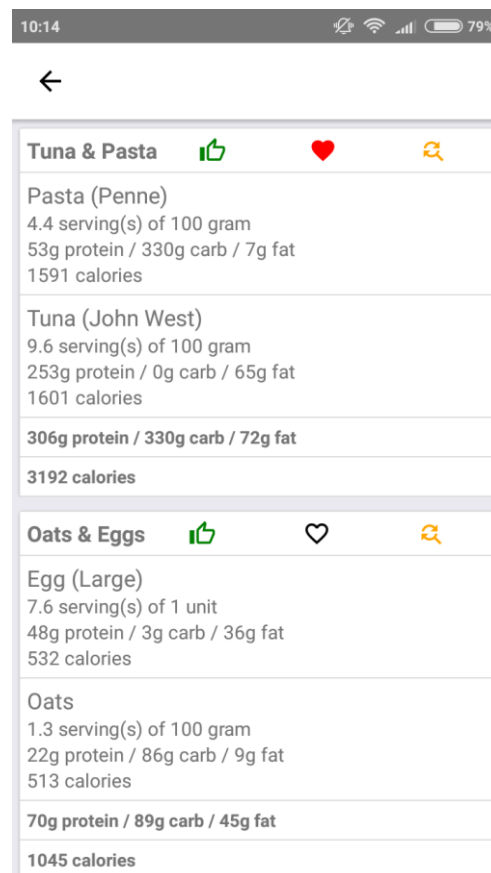


Figure 31 - Showing the difference between meal added to favourites and not added to favourites

## Design – Meal

A Meal component is composed of a list of foods that are rendered in rows with their total macronutrient breakdown displayed at the bottom of the component.




<b>Oats &amp; Eggs</b>			
<b>Egg (Large)</b> 11.8 serving(s) of 1 unit 74g protein / 5g carb / 57g fat 826 calories			
<b>Oats</b> 2.1 serving(s) of 100 gram 35g protein / 139g carb / 14g fat 829 calories			
<b>110g protein / 144g carb / 71g fat</b>			
<b>1655 calories</b>			

Figure 32 - Example of a Meal component

The header for the meal contains 3 buttons. The green “Thumbs Up” button is used to collect user feedback on the meal. If a user likes a meal their user id is saved in a list attached to the meal on the MongoDB database. This data is used in the recommender system which can be activated by pressing the yellow “Magnifying Glass” button.

Pressing this button on a meal uses a HTTP GET request with the web service to return a list of similar meals with their similarity rating.

<b>Chicken &amp; Rice</b>
<b>29% also liked</b>
<b>Chicken</b> 1 serving(s) of 100 gram 28g protein / 0g carb / 2g fat 128 calories
<b>Basmati Rice</b> 1 serving(s) of 100 gram 7g protein / 79g carb / 1g fat 352 calories
<b>35g protein / 79g carb / 2g fat</b>
<b>479 calories</b>

Figure 33 - Meal returned with similarity rating

The meal may be touchable depending on where it is accessed from which allows the serving sizes to be changed or the meal can be swapped with one from the users favourites. A Regex pattern is used to prevent the user from entering invalid characters for a serving size.

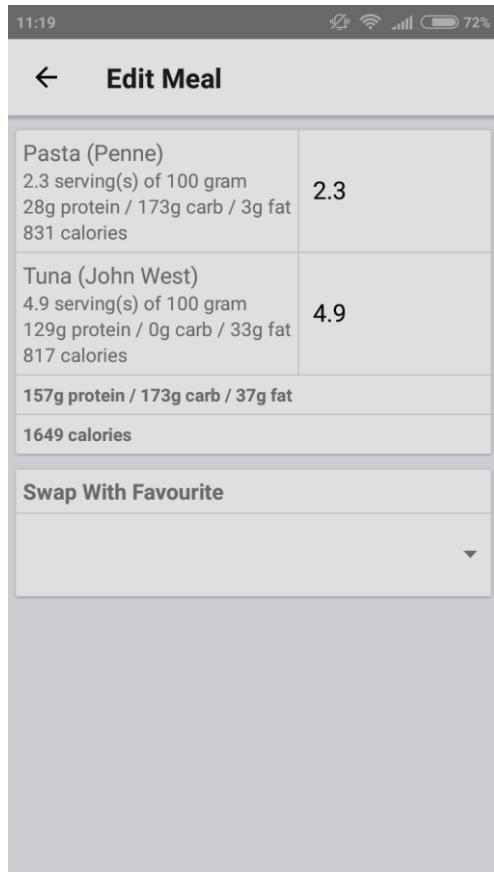


Figure 34 - Edit Meal screen

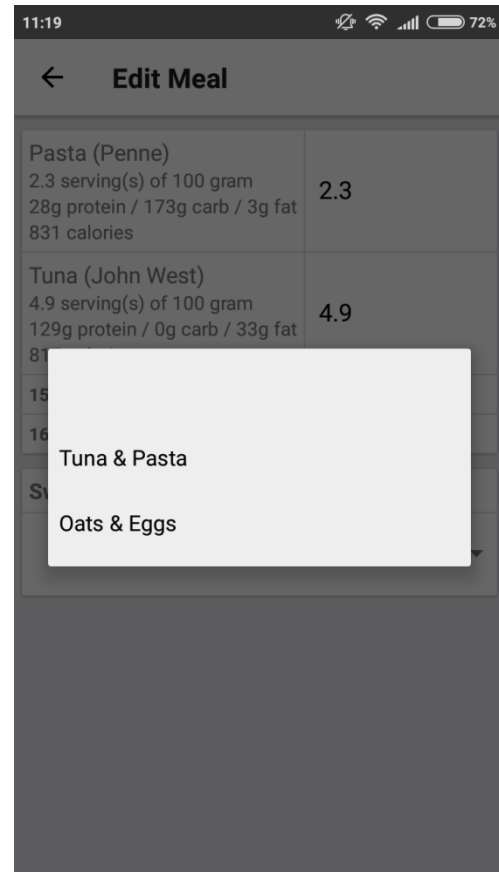


Figure 35 - Picker opened displaying meals the user has added to their favourites

## Weight Statistics



Figure 36 - Weight Statistics screen with KG metric selected



Figure 37 - Weight Statistics screen with lbs. metric selected

## Specification

Design a component allowing a user to monitor their progress by keeping track of their weight. A user should be able to view their weight in history on a graph that can be altered by timeline or by metric. A formula should also calculate the average weight gain or loss of the user over the selected timeline and be displayed in the selected metric.

## Design

Before the component is mounted to the screen the previous user weights (if any) are fetched from the Firebase database. This JSON snapshot is then iterated over and a row is rendered for each entry with the date and value in descending order from the most recent date as seen in the screenshots above.

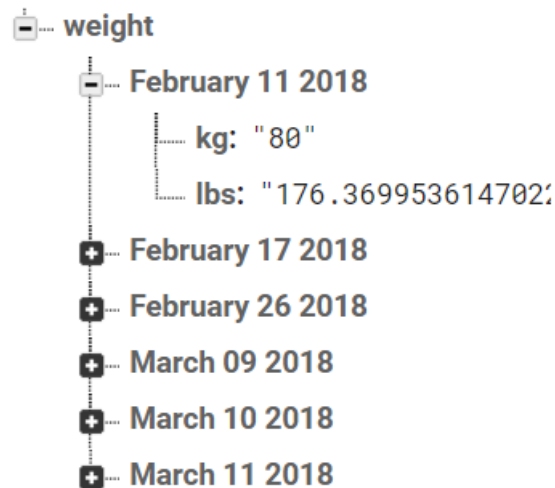


Figure 38 - Weight entries stored in the Firebase database

A library called *victory-native* which is a JavaScript graph and chart library by Victory Labs is used to generate the graph. A library called *Moment.js* is used for operations on dates and provides methods to easily separate the entire history of weight entries into a desired timeframe e.g. past month, past 6 months, etc. This is achieved by rewinding a certain amount of days from the present (e.g. past month would rewind 31 days) and adding the entries to an array used by the graph.

```
for (let i = numRewindDays; i >= 0; i--) {
  const dateKey = String(moment().subtract(i, 'days').format('MMM DD YYYY'));
  if (allWeights[dateKey] !== undefined) {
    displayData[dateKey] = allWeights[dateKey];
  }
}
```

Figure 39 - Code to collect dates within a specified timeframe with Moment.js

The metric and data for the graph is stored in the components Redux store. The graph is animated and will move and change axis values if the user changes the metric or timeline option from the picker. This is because when there is change to the component state the component knows to re-render.

A user can press the “Update Weight” button and a Modal will be displayed. A weight can be entered and is checked by a Regex to only allow valid characters to be entered. If the user enters their weight in kg it will also be converted to lbs. (and vice-versa) and both entries will be saved under the date in the Firebase database.

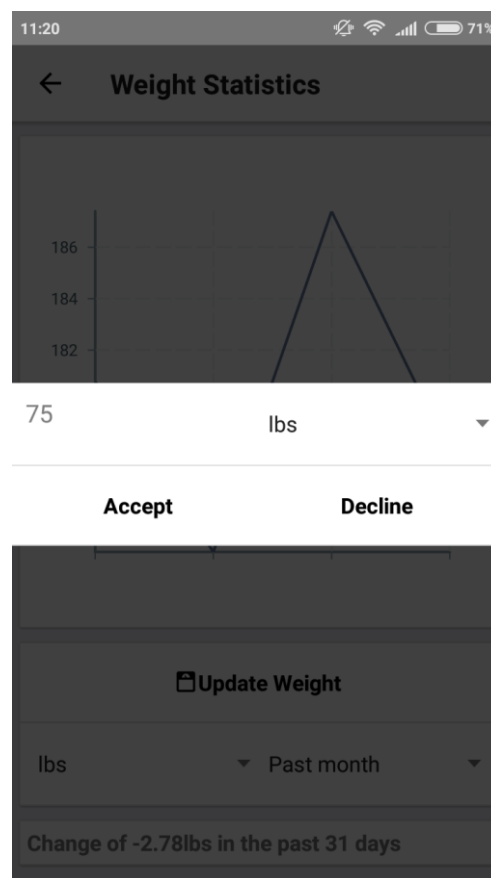


Figure 40 - Modal popup to update weight

## Group Manager

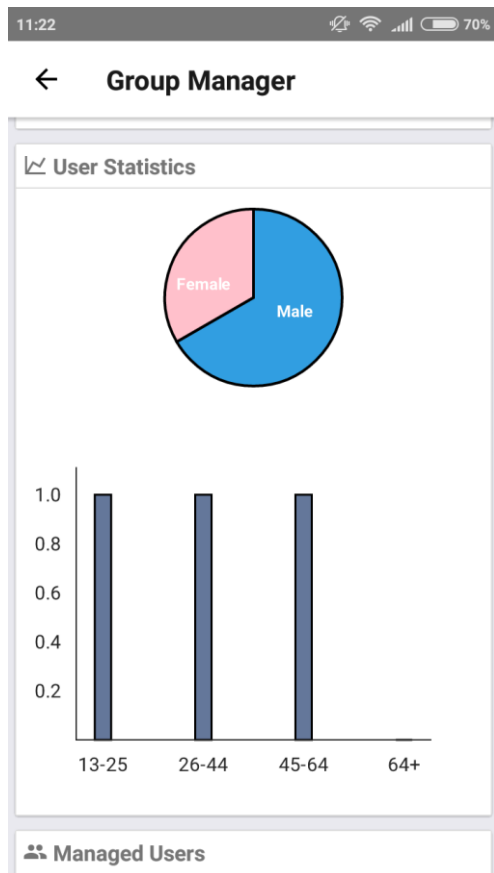


Figure 41 - Group Manager Screen (1)

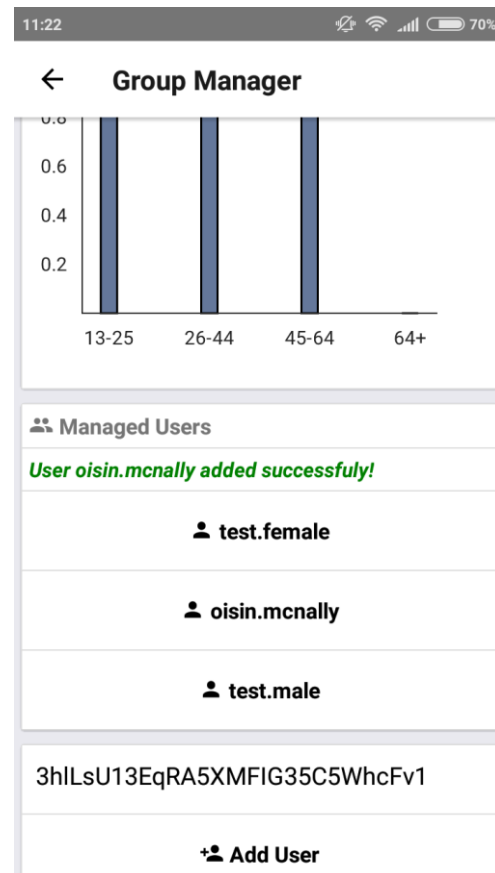


Figure 42 - Group Manager Screen (2)

## Specification

Design a component used to track a group of users. This could be used in a situation where a team manager would like to track their players by being able to see their meal plans, a history of their weigh ins and display some information about the user. There should also be a view showing statistics about the group.

## Design

The graphs seen under the “User Statistics” heading are generated using the *victory-native* library by Victory Labs. Before the component mounts this data is retrieved from the Firebase database by calling a function in the React lifecycle method `ComponentWillMount`.

The top chart is a pie chart showing the distribution of the gender of users in the group while the bottom is a bar chart showing the distribution of the ages of users in the group. The bar chart is animated where the bars grow from the bottom up when the screen is opened.

A user can be added to the group by entering their id in the text box above the “Add User” button. If a user with the id exists a green message is displayed indicating they have been successfully added, else a red error message is displayed. The user id string is quite long so a button was implemented to allow the id to be shared on a platform the user has installed on their device.

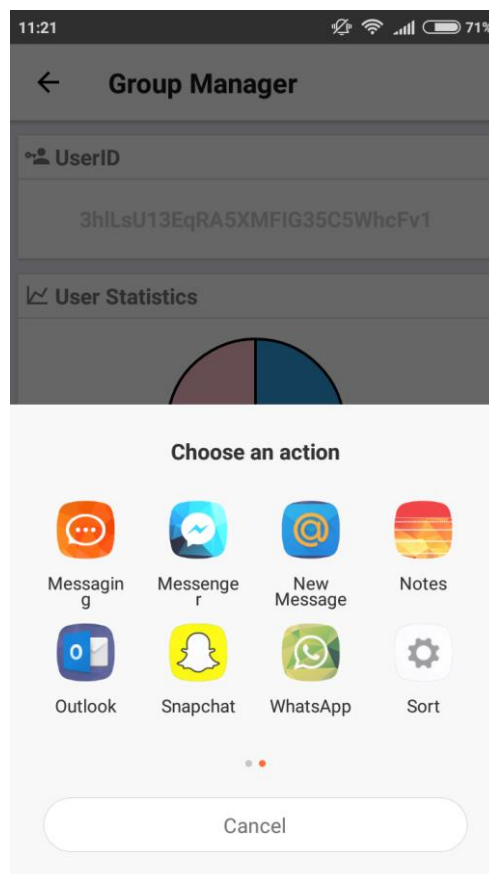


Figure 43 - Pressing the user id allows it to be shared



By pressing on a user's name under the "Managed Users" header a "User Info" screen is opened. This screen displays personal information about the user, their current meal plan and also displays a graph with their weigh ins that can be altered by timeline and by metric as seen with the previous WeightStatistics component.

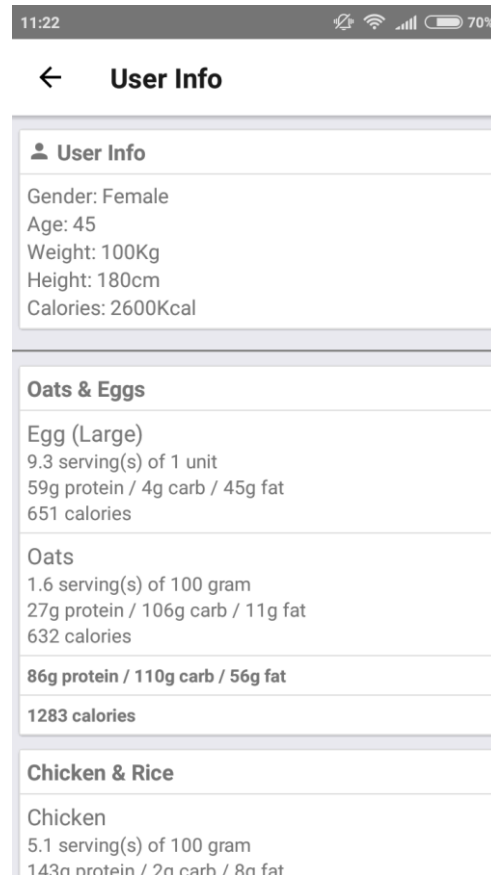


Figure 44 - Press on the name of a managed user to open their user info

## Spring Boot RESTful Web Service

### Introduction

The Spring Boot RESTful web service is used to fulfil requests requiring complex code that would not be easily developed if written in JavaScript. It also is used to communicate with the MongoDB database to retrieve meal and food inventory for the front end. A three tier architecture is used for the web service which can be seen in the table below.

Level	Name	Description
Tier 1	Controller	Maps URL endpoints to a method and extracts information from the HTTP request.
Tier 2	Service	Performs operations on database data and returns the desired response to the Controller.
Tier 3	DAO (Data Access Object)	Retrieve information from the MongoDB database.

Figure 45 - Three tier web service architecture

The web service and database were hosted online using Heroku which is a PaaS (Platform as a Service). The mLab MongoDB add-on was used to host the database and provide a link to connect the web service and database together by placing the link into an application.properties file. The web service was pushed to a specific repository where Heroku would deploy it and provide the base URL of the web service.

```
Oisin@Oisin MINGW64 /c/dev/spring/fyp-web-service (master)
$ git push heroku master
Counting objects: 24, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (19/19), done.
Writing objects: 100% (24/24), 5.42 KiB | 0 bytes/s, done.
Total 24 (delta 13), reused 0 (delta 0)
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Gradle app detected
remote: -----> Spring Boot detected
remote: -----> Installing JDK 1.8... done
remote: -----> Building Gradle app...
remote: -----> executing ./gradlew build -x test
remote:      :compileJava
remote:      :processResources
remote:      :classes
remote:      :findMainClass
remote:      :jar
remote:      :bootRepackage
remote:      :assemble
remote:      :check
remote:      :build
remote:
remote: BUILD SUCCESSFUL
remote:
remote: Total time: 9.253 secs
remote: -----> Discovering process types
remote: Procfile declares types      -> (none)
remote: Default types for buildpack -> web
remote:
remote: -----> Compressing...
remote: Done: 66M
remote: -----> Launching...
remote: Released v9
remote: https://calm-everglades-30372.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/calm-everglades-30372.git
d013787..3d9824d master -> master
```

Figure 46 - Pushing the web service project to the Heroku repo using Git

MongoDB is a JSON database so there must be a way to convert the JSON to a Java object model. The Jackson Mapper is used to convert the JSON object to a defined Java Object model automatically by Spring Boot.

```
public Food(String id,
            String name,
            double serving,
            String metric,
            double protein,
            double carbs,
            double fats) {
    this.id = id;
    this.name = name;
    this.serving = serving;
    this.metric = metric;
    this.protein = protein * serving;
    this.carbs = carbs * serving;
    this.fats = fats * serving;
    this.calories = (protein*4) + (carbs*4) + (fats*9);
}
```

Figure 47 - Food Object (Java Model)

```
_id: "TUNA-1"
_class: "com.fyp.Entity.Food"
name: "Tuna (John West)"
metric: "100 gram"
protein: 26.4
carbs: 0
fats: 6.8
calories: 166.79999999999998
serving: 1
```

Figure 48 - Food Object (MongoDB JSON Model)

Data was pushed to the MongoDB database by creating a list of Food and Meal Java objects and using a method to save them to their respective repositories.

```
@Bean
CommandLineRunner commandLineRunner(FoodRepository foodRepository) {
    return args -> {
        // save foods to repository
        FakeData fakeData = new FakeData();
        for (Map.Entry<String, Food> entry : fakeData.getFoods().entrySet()) {
            foodRepository.save(entry.getValue());
        }
    };
}
```

Figure 49 - Java method to push Food objects to FoodRepository in MongoDB

## Food Controller

The controller is annotated as a RestController and assigned the mapping “/foods”.

```
@RestController
@RequestMapping("/foods")
public class FoodController {
```

Figure 50 - Food Controller configuration

The food controller contains methods to retrieve a Map of all foods from the database. This is used in the front end when a user wants to add foods to their preferences for each meal. HTTP GET requests are used and the response is a *Map<String, Food>* that is converted to JSON format for the front end. The map key (String) is the food id while the map value (Food) is the Java Food object.

A test endpoint was included that did not need to connect to the database so that the web service could be unit tested as an entity by itself.

```
/**
 * Get foods from the FakeData object
 * No connection to MongoDB
 * @return
 */
@RequestMapping(value = "/all/offline", method = RequestMethod.GET)
public Map<String,Food> getFoodFromOffline() { return foodService.getFoodFromOffline(); }

/**
 * Returns all food documents from the food collection
 * on MongoDB
 * @return
 */
@RequestMapping(value = "/all", method = RequestMethod.GET)
public Map<String,Food> getAllFoodFromDatabase() { return foodService.getAllFoodFromDatabase(); }
```

Figure 51 - Food Controller URL endpoints

## Meal Controller

The meal controller is assigned the mapping “/meal” and is used to call methods that are only associated with a meal.

```
@RestController
@RequestMapping("/meal")
public class MealController {
```

Figure 52 - Meal Controller configuration

The endpoints in the meal controller are used if a user pressed the like button for a meal which adds their user id to a set associated with that meal. The other endpoint is used when a user presses a button to find similar meals.

```
@RequestMapping(value = "{id}/similar", method = RequestMethod.GET)
public List<Meal> getSimilarMeals(@PathVariable("id") Integer mealId) {
    return mealService.getSimilarMeals(mealId);
}
```

Figure 53 - Meal Controller endpoint to find similar meals

## Recommender System

The recommender system uses item-item collaborative filtering which is a common algorithm used in other recommender systems. It firstly involves collecting the ids of users that like each meal and then collecting the likes under a HashSet in the MongoDB database. The user can then send a request from the front end to find a meal similar to the one they have requested.

The meal id is sent in the URL endpoint as a path variable where it is extracted and used in a method to find similar meals based on the likes collected from other users.

```
/**
 * Return a list of meals similar to meal indexed by mealId
 * @param mealId
 * @return
 */
@RequestMapping(value = "{id}/similar", method = RequestMethod.GET)
public List<Meal> getSimilarMeals(@PathVariable("id") Integer mealId) {
    return mealService.getSimilarMeals(mealId);
}
```

Figure 54 - Web service method mapped to recommender system URL endpoint

All meals are retrieved from the database initially instead of performing multiple database requests to reduce latency. The meal that the user is requesting to find one similar to is also retrieved.

The users that like each meal from the database are compared to the users that like the meal that we want to find similar to. If there is greater than a 20% correlation between both meals it is added to a list of similar meals that is returned to the front end.

This operation is search intensive which is why a HashSet was used. A HashSet has a Big O notation look-up complexity of  $O(1)$  which is the lowest complexity available. This was used to keep the algorithm operation time to a minimum.

```
public List<Meal> getSimilarMeals(Integer mealId) {  
    List<Meal> allMeals = mealDao.getAllMealsFromDatabase();  
    Meal userInputMeal = mealDao.findById(mealId);  
    List<Meal> similarMeals = new ArrayList<>();  
  
    for (Meal meal : allMeals) {  
        Set<String> likesMealA = userInputMeal.getLikes();  
        Set<String> likesMealB = meal.getLikes();  
        List<String> likesBoth = new ArrayList<>(likesMealA);  
        likesBoth.retainAll(likesMealB); // retain all userId that are common in both lists  
        double tanimotoCoefficient = (double) likesBoth.size()  
            / (double) ((likesMealA.size() + likesMealB.size()) - likesBoth.size());  
  
        // If they are >= 20% in similarity  
        if (tanimotoCoefficient >= 0.20) {  
            meal.setTanimoto(tanimotoCoefficient);  
            similarMeals.add(meal);  
        }  
    }  
    return similarMeals;  
}
```

Figure 55 - Java method for Item-Item based Collaborative Filtering



## Meal Plan Controller

The meal plan controller is given the request mapping “/mealplan” and is used for operations involving the generation of a meal plan for a user.

```
@RestController
@RequestMapping("/mealplan")
public class MealPlanController {
```

Figure 56 - Meal Plan controller configuration

This controller contains endpoints for unit testing of the service as seen with the food controller, generation of a random meal plan and also for generation of a custom meal plan. The main endpoint in this controller is to generate a custom meal plan for the user which can be seen in the snippet below.

```
@RequestMapping(value = "/{calories}/{numberOfMeals}", method = RequestMethod.POST)
public MealPlan getMealPlanFromDatabase(
    @PathVariable("calories") int calories,
    @PathVariable("numberOfMeals") int numberOfMeals,
    @RequestBody List<ConcurrentHashMap<String, Food>> foodPreferencesArray) {
    return mealService.getMealPlanFromDatabase(calories, foodPreferencesArray);
}
```

Figure 57 - Meal Plan controller endpoint to request a custom meal plan

The food preferences sent by the user in the payload of the HTTP POST request are mapped to a ConcurrentHashMap as it does not allow null entries which caused problems with the functionality of the algorithm.

The method to construct a custom meal plan for a user consists of the following steps;

- Retrieve a list of all meals from the database
- Get the first meal from the list
- Get the list of foods for that meal
- Does the food list for the meal contain a food from the user's preferences?
- If so add the meal to a list, if not go to the next meal from the database

Once the meal plan has been constructed the serving sizes for each meal must be scaled appropriately to suite the users daily calorie requirements. The steps to scale the serving sizes for the meal are outline below;

- Get the number of meals in the meal plan
- Calculate the calories for each meal
- Divide these calories evenly amongst each ingredient in the meal
- Calculate the increase in serving size based on the increase in calories
- Repeat for each meal in the meal plan

```
public static void scaleServingSizes(int calories, Map<Integer, Meal> mealsForMealPlan) {  
    int numOfMeals = mealsForMealPlan.size();  
    double calPerMeal = calories / numOfMeals;  
  
    for (Map.Entry<Integer, Meal> mealEntry : mealsForMealPlan.entrySet()) {  
        double calPerFood = calPerMeal / (mealEntry.getValue().getMealFoods().size());  
        for (Map.Entry<String, Food> foodEntry : mealEntry.getValue().getMealFoods().entrySet()) {  
            double servingSize = (calPerFood / foodEntry.getValue().getCalories());  
            foodEntry.getValue().setServing(helperMethods.roundDoubleTo(servingSize, decimalPlaces: 1));  
        }  
        // Calculate the new totals  
        mealEntry.getValue().calculateTotals();  
    }  
}
```

Figure 58 - Method to scale serving sizes for meal plan

The meal plan is then returned to the meal plan controller where it is mapped to JSON using the Jackson Mapper and returned to the front end to be rendered.

## Evaluation & Testing

With React Native, continuous feedback is provided when designing the front end of the application. Options to enable hot and live reloading cause the application to re-render the screen each time a change to the application source code is saved. Each React Native component is designed in a JavaScript file which allows the application to quickly show developer changes without the need to recompile the entire application.

The Spring Boot RESTful web service was deployed and tested on localhost before being deployed on cloud hosting. The web service was deployed on the localhost of a PC while the application was running on an Android device. The localhost of the PC was not the same as the localhost of the Android device so port forwarding was setup through Google Chrome to ensure that the same localhost would also be accessible on the Android device when connected to the PC through a USB.

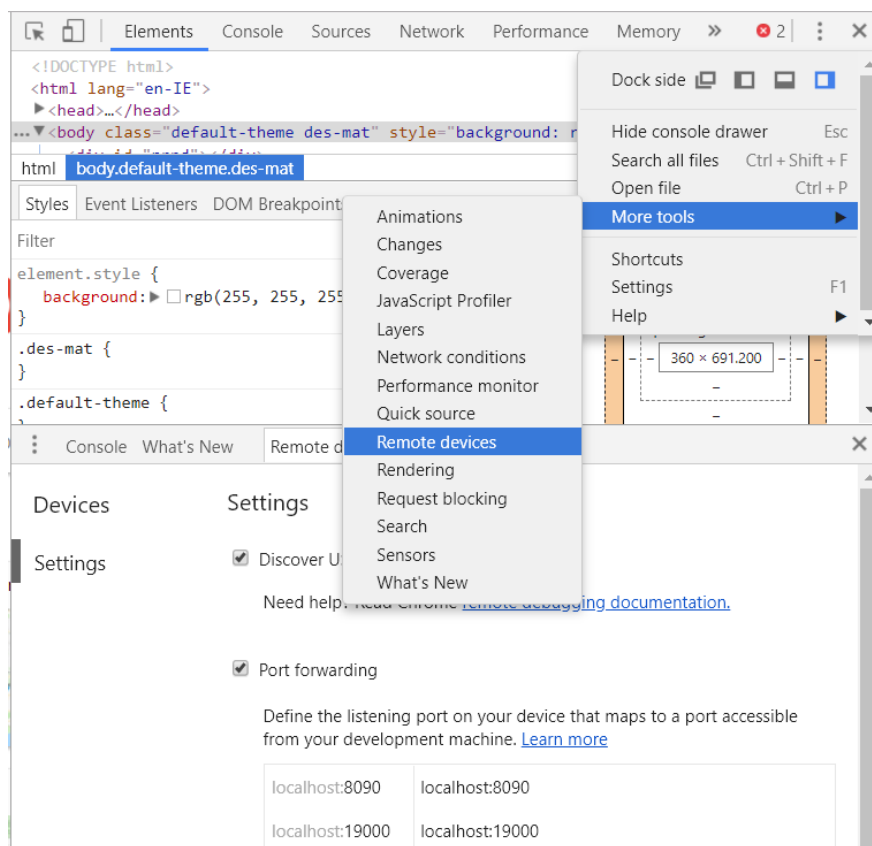


Figure 59 - Port forwarding localhost port 8090 to an Android remote device using Google Chrome

After the web service had been started the endpoints were tested using Postman [20] on PC and using the REST Client for Android [21] application on mobile. This was used to confirm that the endpoints were up and running and working as expected on both PC and mobile.

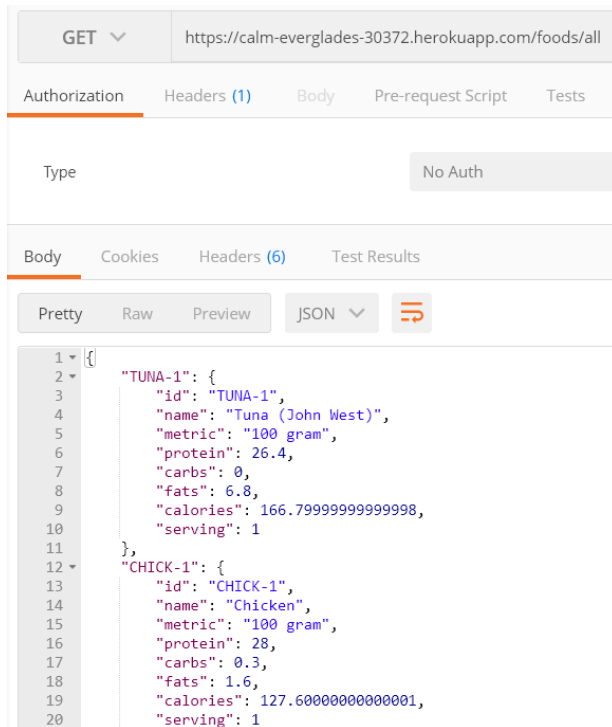


Figure 60 - GET response on Postman (PC)

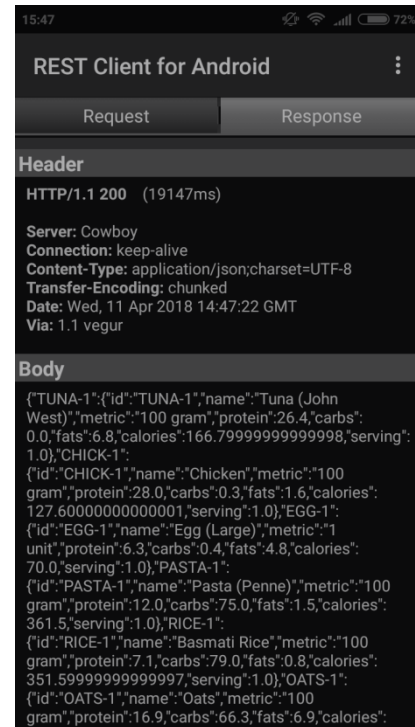


Figure 61 - GET response on REST Client for Android (Android)

A configuration file was used for the front end React Native application so that the web service URL could be changed in one place and then this change would be effective over the entire application. The web service URL was assigned to a static variable and then could be changed between localhost and the cloud hosted URL when testing was finished.

```
class Config {
  static BASE_URL = 'https://calm-everglades-30372.herokuapp.com';
  static APP_NAME = 'FitMealPlanner';
  // http://localhost:8090
  // https://calm-everglades-30372.herokuapp.com
}

export default Config;
```

Figure 62 - React Native configuration file where BASE\_URL can be changed to localhost for testing

The web service also contained endpoints that returned static data and did not need to connect to the database. This was so the web service could be unit tested in the sense that if there was a problem with either the web service or the database, if the endpoint separated from MongoDB was working then the web service was deployed correctly and the error may be with the deployment of the database instead.

```
/**
 * Get foods from the FakeData object
 * No connection to MongoDB
 * @return
 */
@RequestMapping(value = "/all/offline", method = RequestMethod.GET)
public Map<String, Food> getFoodFromOffline() { return foodService.getFoodFromOffline(); }
```

Figure 63 - Example of endpoint used for unit testing the web service

## Possibility for Extension

This application has many possibilities for extension if it were to be deployed to a large group of users. I think that the addition of taking more extensive dietary options for meals such as gluten free, dairy free and vegan options would be important as there are many people that require strict dietary requirements.

The functionality of scaling meals by also taking macronutrients into consideration (e.g. protein, carbohydrates and fats) would allow users a more fine grained control over their meal plan requirements.

One of the most important aspects of an application is collecting large amount of data from users based on their behaviour and then use this information for marketing and sales. The implementation of functionality such as *“Find this meal near you”* could be used to provide users with a map of where a meal may be purchased and possibly generate an affiliate revenue income from purchases.

Affiliate links could be used in conjunction with Google AdSense to generate income by displaying offers for products such as dietary supplements, fitness equipment and gym memberships that would generate income based on page leads and sales commission.

The group manager functionality could also be more extensive and sold as an in-app purchase to teams.

## Conclusion

This application provides users with a quick and easy platform that emulates the service provided by a dietician or nutritionist. Functionality is provided to create a tailored meal plan, track their progress and manage a group from a single application.

The implementations of a level of intelligence where users can be recommended meals based on their feedback or see an average description of their progress attempts to add unique functionality to this application.

This project incorporates a wide range of frameworks delivered in a cross-platform mobile application and provided an insight into the role of a Full-Stack Developer.

## Table of References

- [1] Facebook.github.io. (2018). *React Native · A framework for building native apps using React*.  
[online] Available at: <https://facebook.github.io/react-native/>
- [2] EISENMAN, BONNIE. "Chapter 1: What Is React Native?" *LEARNING REACT NATIVE: Building Native Mobile Apps with Javascript*, O'REILLY MEDIA, 2017
- [3] Redux.js.org. (2018). *Read Me - Redux*. [online] Available at: <https://redux.js.org/>
- [4] GitHub. (2018). *gaearon/redux-thunk*. [online] Available at: <https://github.com/gaearon/redux-thunk>
- [5] Play.google.com. (2018). [online] Available at:  
<https://play.google.com/store/apps/details?id=com.myfitnesspal.android>
- [6] Anon, (2018). [online] Available at: <https://www.mydietitian.com>
- [7] Play.google.com. (2018). [online] Available at:  
[https://play.google.com/store/apps/details?id=com.fitnessmeals.fitnessmealplanner&hl=en\\_GB](https://play.google.com/store/apps/details?id=com.fitnessmeals.fitnessmealplanner&hl=en_GB)
- [8] Mifflin MD, e. (2018). *A new predictive equation for resting energy expenditure in healthy individuals*. - PubMed - NCBI. [online] Ncbi.nlm.nih.gov. Available at:  
<https://www.ncbi.nlm.nih.gov/pubmed/2305711>
- [9] Healthy Eater. (2018). *How to Calculate Your Macros to Transform Your Body*. [online] Available at: <https://healthyeater.com/how-to-calculate-your-macros>
- [10] Momentjs.com. (2018). *Moment.js | Home*. [online] Available at: <https://momentjs.com/>
- [11] GitHub. (2018). *FormidableLabs/victory-native*. [online] Available at:  
<https://github.com/FormidableLabs/victory-native>
- [12] GitHub. (2018). *axios/axios*. [online] Available at: <https://github.com/axios/axios>
- [13] GitHub. (2018). *oblador/react-native-vector-icons*. [online] Available at:  
<https://github.com/oblador/react-native-vector-icons>



- [14] GitHub. (2018). *aksonov/react-native-router-flux*. [online] Available at: <https://github.com/aksonov/react-native-router-flux>
- [15] Firebase. (2018). *Firebase*. [online] Available at: <https://firebase.google.com/>
- [16] Projects.spring.io. (2018). *Spring Boot*. [online] Available at: <https://projects.spring.io/spring-boot/>
- [17] MongoDB. (2018). *MongoDB for GIANT Ideas*. [online] Available at: <https://www.mongodb.com/>
- [18] Heroku.com. (2018). *Cloud Application Platform | Heroku*. [online] Available at: <https://www.heroku.com/>
- [19] Npmjs.com. (2018). *npm*. [online] Available at: <https://www.npmjs.com/>
- [20] Postman. (2018). *Postman*. [online] Available at: <https://www.getpostman.com/>
- [21] Play.google.com. (2018). [online] Available at: <https://play.google.com/store/apps/details?id=com.sn.restandroid>

