



## Dynamic Web Technologies

*Refresh App*

David MacDonald

**B00394201**

**20<sup>th</sup> April 2022**

## Introduction

This project has been built using React-Native and expo.

The Refresh application has been built to allow users to add second hand items to be sold. This has been achieved by allowing users to sign up, login and add listings. The fundamentals and advanced concepts used in React-Native have been utilized throughout this project.

## Fundamental Concepts

Core components and API's, React-Native libraries are full of components, and they are mostly fundamentals. The SafeAreaView is a View that is used specifically for iOS builds, this view lowers the contents of the screen to avoid the notch caused by the iPhones speaker and front facing camera. This view only works for iOS, to avoid issues with Android screens platform specific code has been implemented. This is achieved by importing the Platform module from react-native and dynamically setting the padding. This is done by using the platforms OS property, checking for the operating system, and returning the value based on that operating system.

## Advanced Concepts

### Image Picker

The image picker is installed in the terminal with `expo install expo-image-picker` whenever sensitive information is used the permissions should always be checked first. To get a user's permission the Effect hook is used. The Effect hook allows side effects to be performed in functional components, such as data fetching. To implement the effect hook it was imported from react.

The effect hook is called, and a function is added to it, the function gets executed every time the component is rendered. The `useEffect` function has an optional second parameter which is an array of dependencies. By passing an empty array it leaves the dependencies empty and calls the function only once. This stops the `useEffect` function from asking a user's permission every time they log into the app. The image picker object has useful properties and methods such as the `requestMediaLibraryPermissions`. The `request` method is an async method that returns a promise, this can't be passed within a `useEffect` function only a function that doesn't return a promise can be passed within a `useEffect` function. To solve the problem the logic was moved into a separate function to handle the promise and that function is called in the `useEffect` function. The `requestPermission` async function gives the result of the async method, the result object `granted` Boolean property was used to check on the value of the return. An alert is displayed asking the user to grant permissions, this alert is displayed only once. If the permissions are not granted, then an alert is displayed letting the user know that the app needs permissions to offer this feature. If the permissions are granted, then the app is launched with its full features and the user is never asked again. The props are destructured within this function and the props added are used to maintain state. Wherever an input of picker component is declared, that is where the state must be defined.

```
function ImageInput({ imageUrl, onChangeImage }) {  
  useEffect(() => {  
    requestPermission();//function passed to resolve promise  
  }, []);  
  
  const requestPermission = async () => {  
    const { granted } = await  
    ImagePicker.requestMediaLibraryPermissionsAsync();  
    if (!granted) alert("You need to enable permission to access  
the library.");  
  };  
}
```

The permissions API is another way to request permissions and is installed using `expo install expo-permissions`, this package provides all the objects needed when implementing

permissions. In the request permissions function the permissions .ask async is called, this allows for multiple permissions to be asked in the same function. The permissions.requestMediaLibrary and permissions.Location are passed in this function. This asks the user if the app can access these device API's and passes an alert for each permission before the app can be used. Once the user has granted these permissions, they can now access their media library and select an image to upload.

To select an image from the user's library a view is wrapped with the TouchableWithoutFeedback component that passes the handle press function through the onPress method. Material community icons are imported from the @expo/vector-icons library, these are used to add icons to objects like buttons or tabs. Within the view conditional rendering is used to check if an image has been selected, if the imageUri doesn't exist then the icon is rendered into the view.

```
return (
  <TouchableWithoutFeedback onPress={handlePress}>
    <View style={styles.container}>
      {!imageUri && (
        <MaterialCommunityIcons
          color={colours.medium}
          name="camera"
          size={40}
        />
      )}
      {imageUri && <Image source={{ uri: imageUri }}
style={styles.image} />}
    </View>
  </TouchableWithoutFeedback>
);
}
```

The handle press function checks if an image is not selected then it returns the select image function. If an image is selected and the user taps on that image and alert is passed asking if the user wants to delete the image. If the user selects "YES" then onChangeImage prop is set

to null, this removes the image from the container but doesn't delete it from the users media library.

```
const handlePress = () => {
  if (!imageUri) selectImage();
  else
    Alert.alert("Delete", "Are you sure you want to delete
this image?", [
      { text: "Yes", onPress: () => onChangeImage(null) },
      { text: "No" },
    ]);
};
```

Select image function is used to launch the image library and handle the image selected state. There are two configurational objects that are passed when the image library is launched, the media type is used to confirm an image is being selected and not a video. The second property is quality, which is a number of 0 and 1. One being the highest quality and zero being the lowest. The mid-range of 0.5 is used to prevent large uploads to the server. If the user cancels, then the prop of onChangeImage is called to notify the consumer of the component that the image has changed and pass the Uri of that image. Because this function returns a promise and there might be an error in selecting an image. A try catch block is added to the function to catch errors. This means the function will try to read the image selected or catch the error.

```
const selectImage = async () => {
  try {
    const result = await ImagePicker.launchImageLibraryAsync({
      mediaTypes: ImagePicker.MediaTypeOptions.Images,
      quality: 0.5, //add editing
    });
    if (!result.cancelled) onChangeImage(result.uri);
  } catch (error) {
    console.log("Error reading an image", error);
  }
};
```

## Form State and Validation

Formik is a library that handles form state management.

Formik install with `npm install formik`

By creating a forms folder in the components folder and adding the form components it has helped to keep the code tidy and easier to read. The form components are used to handle the form state. On a plain login screen the state requires variables to be set for each form input.

```
Function LoginScreen(props) {  
  Const [email, setEmail ] = useState();  
  Const [password, setPassword ] = useState();
```

This is the example of a form using two input fields, when the input fields grow the code becomes more complex. For every input field the state variable must be declared and then the code needs to be written to validate that variable. Formik takes care of form state and handles validation without all the complex code. Formik only works with React and React-Native. Formik is a named export so it must be wrapped in braces `import { Formik }` from `'formik'`; The formik component comes with props, initial values and onSubmit. The onSubmit prop handles what happens when the form is submitted. A function needs to be passed inside the Formik component, The argument of the function can be destructured to handle custom props and a jsx expression is returned.

## Yup

Formik has built in support for data validation with Yup, Yup is a JavaScript schema builder used for value parsing and validation.

Installation: `npm install yup`

The validation schema is built outside of the functional component to prevent it being redefined every time the function is re rendered. The validation schema is built and called inside the Formik components props.

The login screen Yup schema:

```
const validationSchema = Yup.object().shape({  
  
  email:Yup.string().required().email().label("Email").max(255),  
  password: Yup.string().required().min(4).label("Password")  
});
```

The object and shape (define the shape of the object) methods are called, and the object is passed. In the object the properties are passed that represent the input fields. The email is a string field, it's required before the form can submit. Setting it as email means it must be a valid email or it will not be accepted. The label is used when a validation error is called. The max and min can be set to limit how much or hoe little the character count is. That is the validation schema for the login form completed it now must be called in the Formik component props. The app form is the custom component created to handle the Formik

components. This allows them to be called multiple times in the app without having to re write the logic.

```
<AppForm
    initialValues={{ email: "", password: ""}}
    onSubmit={handleSubmit}
    validationSchema={validationSchema}
  >
```

To error message is also built as a custom component and called using the props of the Formik component

## Location API

Install: `expo install expo-location`

The latitude and longitude of the user has been logged in the listing edit screen. This console logs the longitude and latitude of the user when they make a new listing.

A hooks folder is created inside the App folder to store custom hooks, within this folder the `useLocation.js` file was created. Using a custom hook allows encapsulating some state and logic inside a single reusable function.

The `useEffect` is used here again to get the user location. The `useLocation` function is set outside of the `useEffect` as this can't return a promise. To get the User location the `getLocation` function is created with the permissions request. If the user doesn't grant permissions then the app just returns to the previous screen. Otherwise, the user's location is returned. The `getCurrentLocation` is called to get the result, the result object is structured to get the coordinates `{coords}` latitude and longitude `{coords: {latitude, longitude}}` and await the location. This is defined in a state variable to get the location and validate it. When the location is returned the two properties that are called are latitude and longitude.

The `useLocation` Hook:

```
export default useLocation = () => {
  const [location, setLocation] = useState();

  const getLocation = async () => {
    try {
      const { granted } = await
Location.requestForegroundPermissionsAsync();
      if (!granted) return;
      const {
```

```

        coords: { latitude, longitude },
      } = await Location.getCurrentPositionAsync();
      setLocation({ latitude, longitude });
      console.log('location', longitude, latitude);
    } catch (error) {
      console.log(error);
    }
  };

  useEffect(() => {
    getLocation();
  }, []);

  return location;
};

```

This is now a reusable function and can be called by adding a `const location = useLocation()` to the component using it.

## Navigation

For the navigation of the app react navigation was installed along with the third-party libraries, gesture-handler, native-screens, safe-area-context, and community/masked-view.


To implement the navigation flow for auth navigation a new folder called navigation was created in the app folder. This folder holds all the navigation code. The screens needed for the navigator are imported and the stack function was created with this the `authNavigator` component was created `<Stack.Navigator>` within this component the screens used for the auth navigation are imported and added to the stack, welcome screen, login, and register. These screens are what any user will see before authorized access.

## Caching data using async storage

By adding a cache layer to the app, I have been able to asynchronously store data while the app is offline. The cache layer serializes/deserializes objects so JSON.stringify or JSON.parse every single time. The cache layer also adds a time stamp to each item and finds out if they are expired or not. If it is expired, it automatically cleans up async storage to free up space. The `cache.js` file imports async storage and defines a function called `store` that takes a key and a value. Then calls async storage `set` item and passes the key. The key value is serialised as a string called `JSON.stringify`. This function returns a promise so must be marked as `async` and wrapped in a try catch block. The item is defined as an object instead of storing the value directly. This object contains two properties, one is the value of the item, and the other is the Timestamp. The timestamp is set to `date.now()`. The `get` function was then implemented with the value of a string. It was then parsed as a JSON object. The item is then checked to see if it exists or not. To calculate the expiry, I first used `moment.js` but after



discovering the import cost extension it shows that this library is extremely large, and I switched to dayjs. The size difference can be seen in the image below.

```
app > utility >  cache.js > ...  
1   import AsyncStorage from '@react-native-async-storage/async-storage';  
2   import moment from 'moment'; 61k (gzipped: 19.7k)  
3   import dayjs from 'dayjs'; 7.1k (gzipped: 3k)  
4
```

By calculating the time stamp based on 5 minutes it returns item expired. If the item was created less than 5 minutes ago then the item is fine. The expiry in minutes is stored as a constant at the top of the code. If the item is expired the is expired function returns null. This function is also used to clean the cache by calling AsyncStorage to remove the item. This breaks the command query separation principle; it is implemented to reduce the data stored in the storage and I am consciously breaking the rules. The item value is then returned if there are no expired items. The logic was then changed in the listings API to allow the listings to be stored in the cache if they have been uploaded from the server.

## Conclusion

The app was heavily based around front end design as that is what was asked. To show some of the features working properly I did add a backend API from a tutorial online. This consumes the listings and authorizes user login and registration. I enjoyed working with React-Native and for my portfolio will be adding my own backend to this project. I couldn't get the messages to work properly so this is something I will be working on after the project submission. In the learning process I built several apps that individually do the features included in the Refresh app. I have added a few extras like Formik and Yup and the offline capability caching.