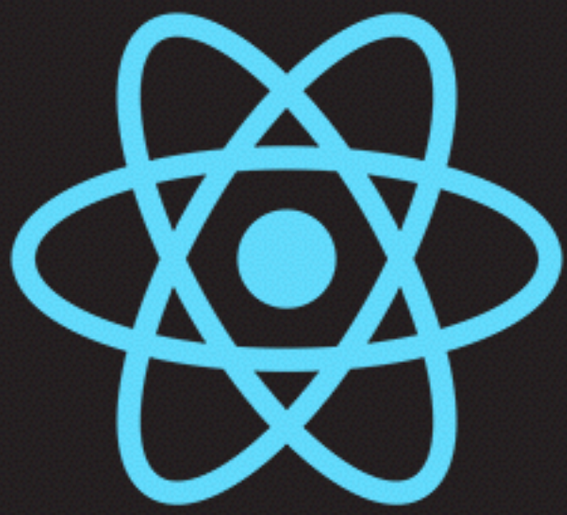
COLLEGIO TECHNOLOGIES INC.

REACT NATIVE: REDUX, ROUTING AND REUSING COMPONENTS



INTRODUCTION

- We're ready to finish the core of this app off! Now that we have Redux loaded and ready to go, it's time to connect it to Firebase, as well as provide some slick routing so we can add to our base feature set!

- Let's begin by seeing how we can connect our Login Form to redux, and how we can set up synchronous and asynchronous Action Generators.

SYNCHRONOUS AND ASYNCHRONOUS ACTION GENERATORS

- So far, whenever we've worked with redux, any actions that we've generated have been synchronous actions; that is, they are performed, and then a value is returned, but the system waits until it is returned.

- However, in order to use firebase's auth commands, which are asynchronous and return promises, we need to have a way in which to handle whatever is eventually sent back from the action generator call.

- In order to handle asynchronous actions in redux, we're going to use another library, called **Redux Thunk**.
  - **npm install --save redux-thunk**

REDUX THUNK

- Redux Thunk allows us to bend the rules of action generators by allowing us to return functions from our action generators, which by default should only return actions.
  - The returned function will then be called with **dispatch()**, allowing us to make calls to other actions.
- Redux Thunk is what's known as **middleware** for redux, and in order to use it, we'll also need to include the **applyMiddleware** function from redux.
  - **applyMiddleware** is used as the third argument inside of the **createStore** function.

```
const store = createStore(
    combineReducers({
        players: playersReducer,
        filters: filtersReducer,
        selectedPlayer: selectedPlayerReducer,
        auth: authReducer
    }),
    {},
    applyMiddleware(ReduxThunk)
);
```

REDUX THUNK

- Once you have Redux Thunk set up in your project, you can use it within your action generators by returning a function instead of an object containing an action. The function will contain a **dispatch** argument, that is what you will use for calling other actions.
    - The **dispatch** function should be used within the **.then()** or **.catch()** functions returned from the promise.

- One very cool feature of using Redux Thunk: we're not just limited to one dispatch action, and in fact can perform multiple dispatches from the same function!

```
export const loginUser = ({ email, password }) => {
    return (dispatch) => {
        authLoading(dispatch);

        firebase.auth().signInWithEmailAndPassword(email, password)
            .then((user) => {
                loginUserSuccess(dispatch, user);
            })
            .catch(() => {
                firebase.auth().createUserWithEmailAndPassword(email, password)
                    .then((user) => {
                        loginUserSuccess(dispatch, user);
                    })
                    .catch((err) => {
                        loginUserError(dispatch, err.message);
                    })
            });
        };
};
```

HANDLING ERRORS INSIDE OF YOUR REDUX THUNK ACTIONS

- Remember, when we are working with Redux Thunk, we are returning a function instead of an object.
  - If your function contains a promise, and the promise code happens to create an error, there is a chance that the error can happen silently.
    - This is because, if you have a catch function attached to your promise, any errors with the AJAX call **or** the associated callback will default to the **.catch()** function.

- If you are custom handling your error, adding a **console.log** function call during development to print the error can provide you with better details of the error context.

REDUX THUNK CONCLUSION

- We'll be using Redux Thunk moving forward to perform any asynchronous calls. By using it, we can pass functions instead of objects.
  - This is something that you'll be using often when developing React apps (how many apps do you know of that don't reach out to an external server?)

- Finally, now that we know how to handle async API calls, how can we set up our app so that we can route specific calls to different views?

REACT NATIVE ROUTING

INTRODUCTION

- We'll be using **React Native Router Flux** for setting up routing inside of our React Native project
  - **npm install - -save react-native-router-flux**
  - https://github.com/aksonov/react-native-router-flux
- This library will break out app up into distinct **scenes**, where a scene is a component that the user can navigate to.
  - When we used React with the DOM, we used **<Route>** components, where in React Native we'll use **<Scene>** components.
- Each scene will accept two properties at minimum:
  - A **key** containing unique information to identify the scene and is used for navigation. This is completely different from the key prop that we use for identifying array items.
  - A **component** which will contain the specific top-level component to display
- In addition, the following are also important properties:
  - The **title** property declares the page title that will be used in the navbar
  - The **initial** property states whether the scene is the initial scene to display. Basically, like the 'homepage' of your app.

REACT NATIVE ROUTER FLUX

- When creating scenes inside of your app, you start by wrapping them inside of a **<Router>** tag.
  - In addition, all specific page **<Scene>** tags must be wrapped inside of a **<Scene>** tag of their own.

```
const MainRouter = () => {
    return (
        <Router>
            <Scene key="main">
                <Scene key="login" component={LoginForm} title="Login" initial />
                <Scene key="players" component={PlayersList} title="Free Agents" />
            </Scene>
        </Router>
    );
};
```

ROUTER NAVIGATION

- Automatic navigation between routes can be performed by using the **Actions** import from **react-native-router-flux**.
  - Once **Actions** has been imported, you can use the **Actions.<route-key>()**, where **<route-key>** is the supplied key for the route that you wish to navigate to.

```
export const loginUserSuccess = (dispatch, user) => {
    dispatch({
        type: 'LOGIN_USER_SUCCESS',
        user
    });

    Actions.playerList();   // sends the user to the scene with key 'playerList'
};
```

GROUPING SCENES

- There are going to be cases where you will want to make use of certain features of react-native-router-flux, such as the automatic back button.
    - Then again, there are also going to be times when you don't want this, such as when you log in (you don't want to see a back button to the login screen if you've already done so).

- You can group each scene by wrapping it inside of another **<Scene>** component. Once you do this, you can change scenes by using the **Actions** class and navigating to the scene grouping's name.

```
<Router>
    <Scene key="root" hideNavBar>
        <Scene key="auth">
            <Scene key="login" component={LoginForm} title="Login" initial />
        </Scene>

        <Scene key="main">
            <Scene
                key="players"
                component={PlayersList}
                title="Free Agents"
                initial
            />

            <Scene
                key="playerCreate"
                title="Become a Free Agent!"
                component={PlayerCreate}
            />
        </Scene>
    </Scene>
</Router>
```

ADDING NAVBAR BUTTONS

- You can also add buttons to the navbar by setting the **'rightTitle'** and **'onRight'** Scene properties.
  - **rightTitle** is used to specify the title that will be displayed.
  - **onRight** is the callback that performs whatever you want. This is an arrow function.

```
<Router>
    <Scene key="root" hideNavBar>
        <Scene key="auth">
            <Scene key="login" component={LoginForm} title="Login" initial />
        </Scene>

        <Scene key="main">
            <Scene
                key="players"
                component={PlayersList}
                title="Free Agents"
                rightTitle="Join Up!"
                onRight={() => { Actions.playerCreate() }}
                initial
            />

            <Scene
                key="playerCreate"
                title="Become a Free Agent!"
                component={PlayerCreate}
            />
        </Scene>
    </Scene>
</Router>
```

EXAMPLE: SETTING UP YOUR ROUTES

- Now that we know some basic routing, let's go ahead and set up our routes for the app. To begin, let's set up routes for:
  - logging in
  - viewing the list of players
  - adding a new player
- You should group your two player routes inside of the same route container for easy routing, and link to the create player component from the list players component.
- For now, we can set up the form elements inside of the add player form to all be text input elements.

THE PICKER COMPONENT

- The picker component in React Native is used to render a form picker inside of your app, which is similar to a <select> element in web.
  - https://facebook.github.io/react-native/docs/picker.html

- You create a new picker by declaring a parent **<Picker>** component with an initial selected value, and several child **<Picker.Item>** components which will hold the different options.

  - Each **<Picker.Item>** component has a **label** and a **value** prop to house the label and option value.

THE MODAL COMPONENT

- Unlike React for DOM, where we have to use a router as a third party package, the **<Modal>** component comes packaged with React Native, and works similarly to the react-modal package for React DOM.

- You need to set a few props for the modal to work:
  - **visible** - determines whether the modal is visible or not.
  - **onRequestClose** - accepts a function, and is called when the modal window is closed. You ***must*** include this for modal windows to work with Android.

- In addition, you can set additional features for your modal, such as show/hide animation and background transparency.

FIREBASE DATA

INTRODUCTION TO WORKING WITH FIREBASE DATA

- We've used Firebase to work with authentication data by using the **auth()** function already, but how do we actually work with data?
  - We'll use the **database()** function to work with stored data in firebase.

- We'll use the **database()** function to read, write, update, and delete data.
  - This is commonly known as ***CRUD (Create, Read, Update, Delete)*** operations.

- The **database()** function will chain into a **.ref()** function, that is where your data store is located. For instance, if you are storing players inside of firebase, it would make sense to use **'/players'** as your database ref location.

- Once you selected function is performed on the database, you can handle any required functionality after the call completion by using the **.then()** function.

READING FIREBASE DATA

- If you are reading firebase data into your database and want the dataset to dynamically update with each operation performed, you want to use the **.on()** function.

- This function accepts two arguments:
  - What you want to monitor (in our case, we set this to 'value' to get database values)
  - A callback function that is triggered whenever the data is updated.

- In addition, if you wanted to just retrieve the data once, you can use the **.once()** function and pass it the same value as the first argument above.

```
firebase.database().ref('/players')
    .on('value', (snapshot) => {
        dispatch({
            type: 'FETCH_PLAYERS',
            players: snapshot.val()
        });
    });
```

WRITING DATA TO FIREBASE

- You can add new data to firebase by chaining the **.push()** function to your **database().ref()** function.
  - This function will be passed an object containing the data that you want to store. Please note, firebase will automatically create an ID for the database item.

```
firebase.database().ref('/players')
    .push({ name, gender, skill_level, sport_type, message, user_id: currentUser.uid })
    .then(() => {
        dispatch({
            type: 'RESET_PLAYER_FORM'
        });

        // NOTE: using:
        // Actions.main({ type: 'reset' });
        // would have worked too!

        Actions.pop();
    });
```

UPDATING DATA IN FIREBASE

- You can update records by using the **.set()** function, which is passed an object containing the new values that you want to set. Please note, when setting the **.ref()** function, you need to set the item ID that you want to update.

```
firebase.database().ref('/players/'+uid)
    .set({ name, gender, skill_level, sport_type, message, user_id: currentUser.uid })
    .then(() => {
        dispatch({
            type: 'RESET_PLAYER_FORM'
        });
        Actions.main({ type: 'reset' });
    });
```

DELETING DATA IN FIREBASE

- You can delete records by using the **.remove()** function. Please note, when setting the **.ref()** function, you need to set the item ID that you want to delete.

```
firebase.database().ref('/players/'+uid)
    .remove()
    .then(() => {
        Actions.main({ type: 'reset' });
    });
```

EXERCISE: THE FREE AGENT TRACKER

- Time to level up the free agent tracker! The goals of this exercise are to perform the following:
  - Set up redux to work with both authentication and database CRUD operations by using Redux Thunk.
  - Set up routing to serve the login, list players, edit player and create player routes.
  - Use a modal window to confirm deletion.
  - Use **React Communications** to set up texting functionality.
- By the end of this example, we should have a fully functioning app that we can use!

RESOURCES

- **Redux Thunk:**
  - **Official Site:** https://github.com/gaearon/redux-thunk
- **React Native Router Redux:**
  - **Official Site:** https://github.com/aksonov/react-native-router-flux
- **React Native:**
  - **Picker:** https://facebook.github.io/react-native/docs/picker.html
  - **Modal:** https://facebook.github.io/react-native/docs/modal.html
- **Lodash:**
  - **Official Site:** https://lodash.com/
  - **Github Repo:**
- **Firebase:**
  - **Reading and Writing Data:** https://firebase.google.com/docs/database/web/read-and-write
  - **Working with Lists of Data:** https://firebase.google.com/docs/database/web/lists-of-data

DISCLAIMER

- Copyright 2018 Collegio Technologies Inc.
- All content within this presentation has been compiled by Collegio Technologies Inc. All material has been sourced through open source resources and source code and as such conforms to Canadian copyright laws and regulations. All rights reserved.