COLLEGIO TECHNOLOGIES INC.

REACT NATIVE: FLEXBOX, FIREBASE AND REDUX

INTRODUCTION

- Now that we have built out a simple React Native program, let's build out what we know and see how it works with React Native!
- In addition, we'll be exploring some other React Native specific components.
- Let's start with exploring how we will style our apps using Flexbox.

INTRODUCTION TO FLEXBOX

- As mentioned in the last lesson, Flexbox is a method in which an element is positioned based off the parent element.
  - The parent element is commonly known as the **Flex Container** and the child elements are **Flex Items**.
- Flexbox is not exclusive to React Native; in fact, if you wanted to use a Flexbox layout inside of a standard web app, you're free to do so!
  - **Hint:** you use the **display: flex;** rule in CSS to turn an element into a flex container!
- Using the dimensions of the parent, you can align an element vertically or horizontally, evenly space sibling elements, and stretch elements to specific ratios.
- All components in a React Native project are by default flex containers already!
- Let's chat about the theory behind Flexbox:

THE FLEXBOX GRID SYSTEM

- When you render flex items in your React Native app, it is rendered across a graph containing two axes: **the main axis** and **the cross axis.**
  - The main axis is the primary axis depending on the flex direction. When set to ‘row’, the X axis is the main axis. When set to ‘column, the Y axis is the main axis. Alignment is set here with **justifyContent**.
  - The cross axis is the secondary axis. Alignment is set here with **alignItems**.
- You need to be mindful of the direction of each flex container in order to properly set up your vertical and horizontal alignment.

FLEXBOX RULES

- **flex -** this rule determines the ratio in which the parent window will be divided. All flex values within the parent element are gathered, and the ratio is determined from the overall total of flex values.
  - For instance, if we had 3 elements within a container, and each had a **flex: 1** set, each container would take up 1/3 of the screen. If one of the elements had a **flex: 2** set, that image would be half of the screen, and the other two components would be 1/4.
- **flexDirection -** set to *column* or *row*. Configures which direction the elements will be arranged in when rendered.
  - In addition, if you wanted components rendered in reverse order, you could use *row-reverse* or *column-reverse*.
- **alignSelf -** used to centre your container, and can also be used to stretch your container to the parent size.

JUSTIFYCONTENT AND ALIGNITEMS

- **alignSelf** is used to align a component based on the parent, but what if you wanted to align ***all*** elements in a specific fashion?
- Some of the most common transformations that you'll be doing with Flexbox involve aligning your elements within the parent container. We use **justifyContent** and **alignItems** to do this.
  - **justifyContent** aligns along the main axis (vertically for 'column', horizontally for 'row')
  - **alignItems** aligns along the cross axis (horizontally for 'column', vertically for 'row')
- Each of these items can have any of the following values:
  - **flex-start**: the (left or top) of the parent
  - **center:** the centre of the parent
  - **flex-end:** the (right or bottom) of the parent
  - **space-between:** evenly spaces out all elements, aligning them against the edge of the parent
  - **space-around:** evenly spaces out all elements, but adds space between the elements and the edge of the parent that is half of the space between elements
  - **space-evenly:** evenly spaces out all elements, but adds space around the edges that is equal to the space around the elements.
  - **stretch:** stretches out the element to consume the entire space

FLEXWRAP AND FLEXBOX CONCLUSION

- **flexWrap -** can be set to ‘wrap’ or ‘nowrap’. Used to toggle whether elements wrap or do not wrap around the parent container. Default is ‘nowrap’.

- That’s it! We now know enough about Flexbox to be dangerous! Let’s use this knowledge moving forward to create some cool styles inside of our app, and make it look like a mobile application!

- Keep in mind, if you are using Flexbox for web, you have some additional rules at your disposal, but we’ll leave that up to you to learn!

INTRODUCTION TO FIREBASE

WHAT IS FIREBASE?

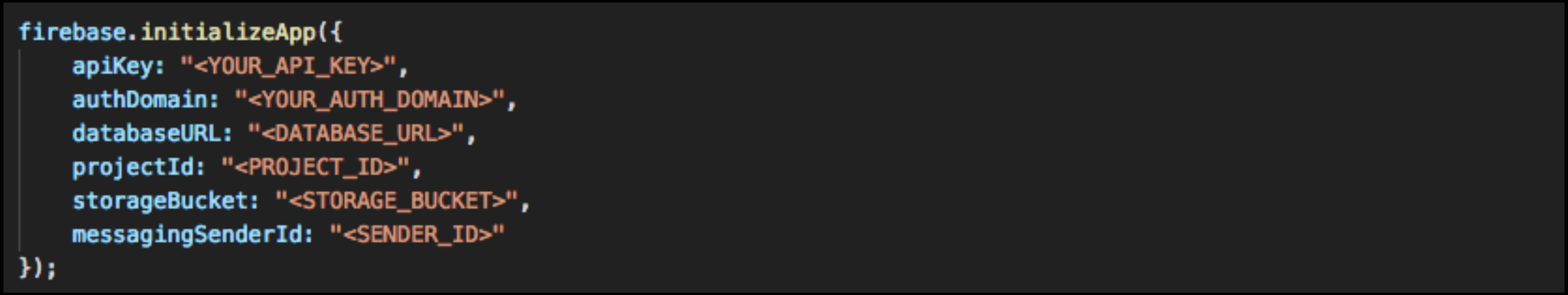
- Firebase is a service owned by Google that allows you to quickly create a back-end database, as well as perform other duties, such as login authentication and user tracking.
  - https://firebase.google.com/
- We can use firebase to automatically let us know when data has been updated.
- You need a Gmail account to use Firebase. By going to the firebase website, you can sign up and create a new project.
- When working with Firebase, you have 3 options: Firebase for iOS, Firebase for Android, or Firebase for web. For React Native, we’ll be using options from ‘Firebase for your web app’.

FIREBASE AUTHENTICATION

- In order to use authentication in our apps, we'll need to enable Auth in our Firebase project.
    - We can go to the Auth section by clicking on the link in the left sidebar.
- By selecting 'Set Up Sign-In Method', we can enable our method for signing into our app. For the time being, we'll only be using email/password.
    - At the moment, Firebase does not have great support for using OAuth sign-in from Google, Facebook etc., but we're hoping that will change soon!
- Moving forward, we'll be using some auth() specific functionality involving email and password with Firebase. You can review the official documentation for this at:
    - https://firebase.google.com/docs/auth/web/password-auth

FIREBASE SETUP

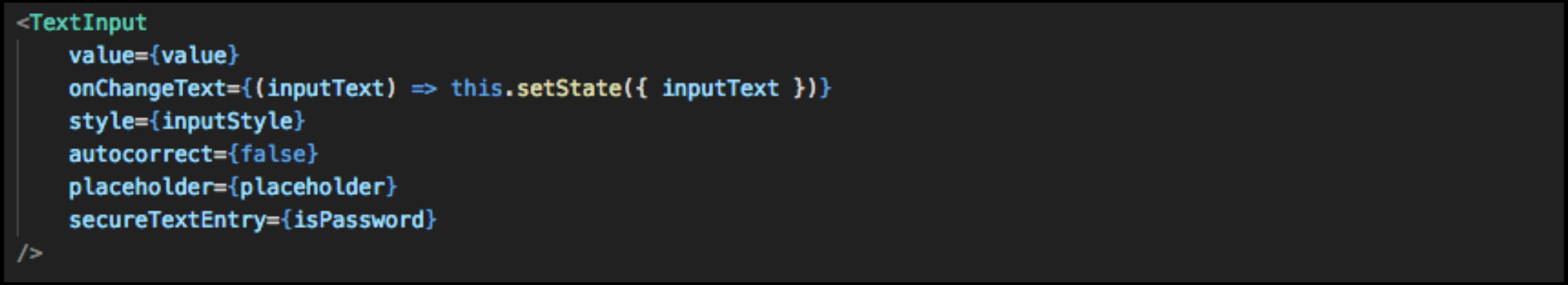
- Now that you have your account set up on Firebase's end, it's time to configure your project to use it. First, you'll need the Firebase Library.
  - **yarn add firebase@4.8.0**
  - **Note:** I had some issues with the newest version, and based on articles I read, people experiencing the same error reverted to this version and all was fine.
- You can now import firebase and add your credentials to initialize your app!
  - You can get your credentials inside of your firebase dashboard by clicking on the 'Add Firebase to your web app' button.



```
firebase.initializeApp({
    apiKey: "<YOUR_API_KEY>",
    authDomain: "<YOUR_AUTH_DOMAIN>",
    databaseURL: "<DATABASE_URL>",
    projectId: "<PROJECT_ID>",
    storageBucket: "<STORAGE_BUCKET>",
    messagingSenderId: "<SENDER_ID>"
});
```

REACT NATIVE TEXT INPUTS

- Now that we know how to work with Firebase and have set up the ability to login via email/password, let's set up a login form!
  - However, in order to do this, we're going to need a way for the user to enter input into our app…
- You add new text input components into your app by using the **<TextInput />** React Native component.
  - The **TextInput** component behaves very similarly to the **Image** component, in that you must specify a height and width in order to use it.
  - https://facebook.github.io/react-native/docs/textinput.html
- In order to use the **TextInput** component, you must specify an **onChangeText** attribute that contains an arrow function updating your state.
  - By doing this, you store the values of the input element inside of your state. Without doing this, there is no way to link the TextInput component to any logic.



```
<TextInput
    value={value}
    onChangeText={(inputText) => this.setState({ inputText })}
    style={inputStyle}
    autocorrect={false}
    placeholder={placeholder}
    secureTextEntry={isPassword}
/>
```

JAVASCRIPT PROMISES

- Almost all of the Firebase functions that we'll use are going to have to connect to the external server to complete, and as a result, we'll have to keep in consideration that these actions are **asynchronous**.
  - Therefore, we'll need to treat these calls as **promises**.
  - A promise is a structure that will eventually return a result, and needs a corresponding function to handle the returned data.

- We evaluate successful firebase calls by using the **.then()** function, and evaluate erroneous firebase calls by using the **.catch()** function.

EXERCISE: LOGIN SCREEN

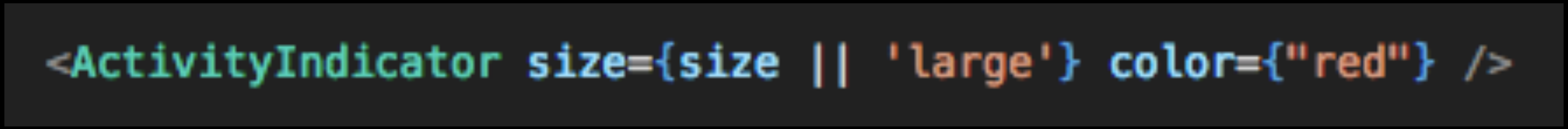
- Let's build a login screen that we'll use for the remainder of the project moving forward! The login screen will require the following:
  - TextInput for email
  - TextInput for password
  - Submit button
- When submitted, we will attempt to log in the user with email and password. If the user account isn't available, we're going to want to set the user up with a new account.
- Once a user is logged in, use firebase's **auth().onAuthStateChanged()** to determine if there is currently a logged in user.
  - If the user is not logged in, display the login form
  - If the user is logged in, show a logout button.

LOADING SCREEN SPINNER

- You may have noticed that the Login form isn't very responsive. This is because, although we are completing the functionality, we are never actually letting the user know that we are doing this by updating the state.

- Luckily, React Native comes with a default spinner that can be used for showing the user when some data is loading.
    - We do this by using the **ActivityIndicator** component.
    - This component will display a circulator loading indicator.

- In addition to any styles you declare, you can also specify the spinner size and color via props.



```
<ActivityIndicator size={size || 'large'} color={"red"} />
```

SETTING UP REDUX

INTRODUCTION

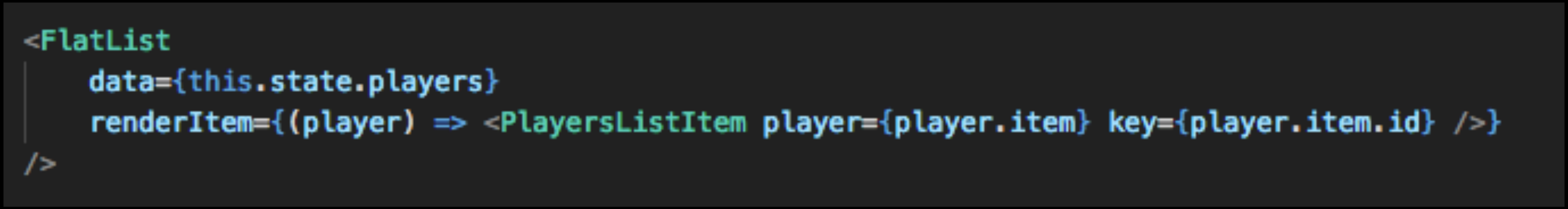
- Let’s look at what we now have available to us:
    - ***A listing of available free agents***
    - ***A login form that can log you into firebase***
- Even though we have both of these built out, we’re eventually going to need a way to combine everything up into one app.
- Let’s set up redux inside of our app, and create a store, reducers, actions and selectors for our free agents.
- By using redux with react, in addition to our players and filters, we’ll also be able to store auth information inside of a reducer.
- However, before we begin, let’s discuss how we can further optimize our app by using a ***list*** inside of our app.

LISTVIEW AND FLATLIST

- You may be asking the question: why would I need to declare a list when I can just use a **map()** function to display the components?
- This is due to a scaling issue. When you use the **map()** function, you are essentially rendering **every** in the list. If you had 1000 different items to display as components in the list, it would create 1000 different components. This will likely cause major performance issues.
  - The **FlatList** and **ListView** components, on the other hand, will only render the list items that are currently visible, and only update the viewable list items when necessary, swapping the data from any list items that go out of view with list items that come back into view.
- This causes immediate performance gains, and allows for much greater data scalability.
- **Note: ListView** is now depreciated, so we will solely using **FlatList**.
  - https://facebook.github.io/react-native/docs/flatlist.html

THE FLATLIST COMPONENT

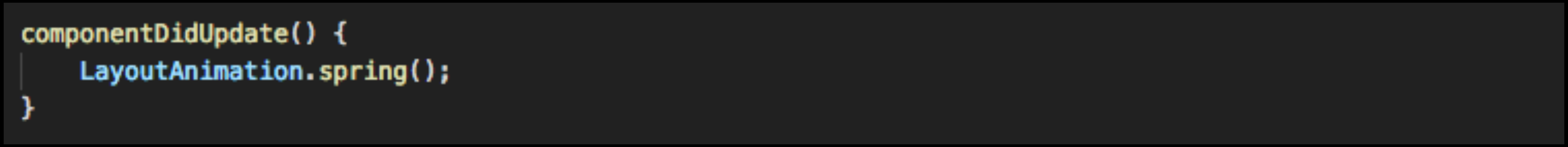
- The **FlatList** component requires 2 props:
  - **data**: the array of data to be displayed in the list
  - **renderItem**: a function that will render the display of each individual item.



```
<FlatList
    data={this.state.players}
    renderItem={(player) => <PlayersListItem player={player.item} key={player.item.id} />}
/>
```

LAYOUTANIMATION

- Right now, we have a working list of free agents, but what if we wanted to set up an effect where the quote and button are only shown when the player is touched?
- The **LayoutAnimation** React Native component can be used to create a bunch of neat pre-defined animations, such as **spring()**, to create a 'springy' effect whenever the element is clicked.
  - This should be placed within the **componentDidUpdate()** function.



```
componentDidUpdate() {
    LayoutAnimation.spring();
}
```

EXERCISE: SET UP REDUX, FLATLIST AND ANIMATION!

- We're going to finish the lesson by setting up redux for the next lesson, as well as some other cleanup!
  - We'll keep the LoginForm code, but we'll put it aside for now.

- Exercise Checklist:
  - Set up redux based on your previously built redux in the React Lesson (https://github.com/collegio/intro_react_redux2_completed)
  - Create your list of free agents as a **FlatList** component.
  - For each item in your list, only show the card header. When the header is clicked, show the message and button, as well as animate using **LayoutAnimation**.

RESOURCES

- **Flexbox:**
  - **React Native Official:** https://facebook.github.io/react-native/docs/flexbox.html
  - **React Native Layout Props:** https://facebook.github.io/react-native/docs/layout-props.html#flex
  - **Flexbox Tutorial (not React Native, but shows some good visual examples):** https://css-tricks.com/snippets/css/a-guide-to-flexbox/
  - **Styling Cheatsheet:** https://github.com/vhpoet/react-native-styling-cheat-sheet#flexbox
- **Firebase:**
  - **Official Site:** https://firebase.google.com/
  - **Firebase Auth:** https://firebase.google.com/docs/auth/web/password-auth
- **React Native:**
  - **TextInput:** https://facebook.github.io/react-native/docs/textinput.html
  - **ActivityIndicator:** https://facebook.github.io/react-native/docs/activityindicator.html
  - **FlatList:** https://facebook.github.io/react-native/docs/flatlist.html
  - **LayoutAnimation:** https://facebook.github.io/react-native/docs/layoutanimation.html

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.