u-TAC

Degree Monitor for iOS and Android

Getting Started With React Native

Nicholas Reimherr

March 11, 2019

Contents

1 Syntax Hints **2**
1.1 Importing . . . 2
1.2 Comparison Operators . . . 2
1.3 Function Notation . . . 2
1.4 Arrays and Useful JS Functions . . . 3
1.5 JSX Syntax . . . 3

2 React & React Native **4**
2.1 React . . . 4
2.2 React Native . . . 4
2.3 NodeJS & Libraries . . . 4
2.3.1 NodeJS . . . 4
2.3.2 Libraries . . . 4

3 Building an Example GPACalculator App **5**
3.1 Getting Started . . . 5
3.1.1 Initializing the Project . . . 5
3.1.2 Setting Up App.js . . . 5
3.2 Creating a React Component . . . 6
3.2.1 Getting out of App.js . . . 6
3.2.2 Setting up State . . . 7
3.2.3 Updating States with setState . . . 8
3.2.4 Defining Functions in a Component . . . 8
3.2.5 Rendering our Courses . . . 9
3.2.6 Styling . . . 10
3.2.7 Making the GPA Field Update . . . 11

1 Syntax Hints

1.1 Importing

Default Imports

Every JavaScript file may declare as many as one ’default’ export. This component will be automatically imported regardless of whether or not the name matches the component itself if the import is NOT surrounded in curly braces.

Listing 1: Default Import

```
import A from './A'; // is the same as

import any_name from './A'; //Both of these imports will refer to the
    default component in A.js
```

Other Imports

A JavaScript file can contain any number of non-default exports. These exports can be components, classes, functions variables, etc... In order to import the components, you will need to use curly brace syntax with names that match the exports you need as shown in the following example.

Listing 2: Other Imports

```
import { a, b, c } from './A';
```

1.2 Comparison Operators

Avoid Double Equals

Using the == operator to check for equality can lead to some unexpected results because JavaScript will attempt to coerce the types before the comparison. The following statements are examples of potentially misleading results.

Listing 3: Unexpected Comparisons

```
'' == '0'             // false
0 == ''               // true
0 == '0'              // true

false == undefined    // false
false == null         // false
null == undefined     // true

' \t\r\n ' == 0       // true
```

Use Triple Equals

Instead, make use of the triple equals operators(===, !==) in order to avoid any type coercion. Using these operators will cause all of the previous examples to return false.

1.3 Function Notation

Arrow Functions

You can usually choose between different styles for declaring functions, however, for the sake of consistency Degree Monitor will use functions that use arrow notation.

Listing 4: Arrow Function

```
myFunction = (a, b, c) => { // parameters a, b, c
    // ...
}
```

1.4 Arrays and Useful JS Functions

//TODO...View list of JavaScript array functions

1.5 JSX Syntax

//TODO...View the documentation for JSX

2 React & React Native

2.1 React

Component Layout

The key concept behind react is the ability to dynamically refresh the browser on any changes to the variables defined in the state of a component. The following code shows an example of what the skeleton of a react component looks like.

Listing 5: Empty Component example

```
import React, { Component } from 'react';

//export keyword allows other JS files to import this component
export default class MyComponent extends Component {

    constructor(props) {
        super(props);

        this.state = {
            //State variables here
        }
    }
}
```

2.2 React Native

Explanation

React-Native will allow the team to build an iOS and Android app using JavaScript which gets compiled into native mobile code when deployed. It should be noted that React Native is not meant to be a "write-once" run anywhere framework. There will likely be some instances where code needs to be customized for the platform it is running on. On most projects, especially one on the scale of DegreeMonitor, code reuse exceeding 80% can be expected.

2.3 NodeJS & Libraries

2.3.1 NodeJS

What is NodeJS

NodeJS is a development environment that allows us to execute JavaScript code outside of our browsers. Node comes with a package manage npm, that will allow us to easily import and manage our dependencies. Local project dependiences will be install inside a node_modules folder and as a convention, we will not include the node_modules folder in our git repository. This means that when pulling the repository for the first time, you will not be able to start the application in expo using $ expo start until you run the following command inside the project directory:

Listing 6: Installing dependencies after pulling a git repository

```
$ npm install
```

2.3.2 Libraries

Native-Base

//TODO... View the Documentation for Native-Base

React-Navigation

//TODO... View the Documentation for React-Navigation

3 Building an Example GPACalculator App

3.1 Getting Started

3.1.1 Initializing the Project

Assuming you have already installed NodeJS and expo-cli, enter a directory in which you would like to place your project and run the following command: (NOTE: Choose the blank project option and any name you want)

Listing 7: Create Project

```
$ expo init GPACalculator
```

Now cd into this directory.
We will now need to install Native-Base with the following command which will give us a default set of components that will render to match the design paradigm of the mobile platform they are being run on.

Listing 8: Install Native-Base

```
$ npm install native-base --save
```

You will notice a message appear in the console after the installation is complete. We will use the suggested command in order to eject the theme and add it into our local project directory.

Listing 9: Eject Native-Base Theme

```
$ node node_modules/native-base/ejectTheme.js
```

This is going to create the file called native-base-theme which will allow us to customize some of the default colors of our native base components.

3.1.2 Setting Up App.js

The App.js is essentially the root of our application where we will begin to render some of the components of our project. In general, it is a best practice to write minimal code in App.js. Now, in order to actually make use of our theme customization's, we will need to make some modifications to our App.js file in the root of our project. We should begin by importing some of the files from our native-base-theme file.

Listing 10: App.js

```
import getTheme from './native-base-theme/components';
import variables from './native-base-theme/variables/commonColor';

import { StyleProvider, Container, Text } from 'native-base'; // These
    imports are from the native base library, not the theme files we just
    added
```

Next we need to modify what is currently being rendered inside the render function of the App class. Currently, you should see the following code:

Listing 11: App.js

```
export default class App extends React.Component {
    render() {
        return (
            <View style={styles.container}>
                <Text>Open up App.js to start working on your app!</Text>
            </View>
        );
    }
}
```

Lets go ahead and remove the View element and Text element and replace it with the following code:

Listing 12: App.js

```
export default class App extends React.Component {
    render() {
        return (
            //Applying a style to a Component requires use of the 'styles'
                prop
            //getTheme is a function we imported previously, along with the
                 variables parameter
            <StyleProvider style={getTheme(variables)}>
                <Container>
                    <Text>
                        Open up App.js to start working on your app!
                    </Text>
                </Container>
            </StyleProvider>
        );
    }
}
```

The View element was replaced with an empty native base container which is described as "a complete section of screen" per the native base documentation. We then wrapped the container element in a StyleProvider which allos us to apply the our custom theme variables to all of the components wrapped inside this tag. (Since App.js is the root of our component, this applies these styles to all subsequent components by default.) To set the style of a component, we can use the style prop. In this case, we are going to use the imported variables from our theme files as a parameter to the getTheme() function that was also imported.
For more information on native base theme customization, refer to the native base documentation.

3.2 Creating a React Component

3.2.1 Getting out of App.js

In order to keep App.js as clean as possible, lets create a *'src'* directory in the root of our project for the rest of our JavaScript files. Lets create a new file inside this directory called *Calculator.js*. Inside this file we can begin by adding the basic react component skeleton snippet from section *2.1*.

Listing 13: Calculator.js

```
import React, { Component } from 'react';

export default class Calculator extends Component {

    constructor(props) {
        super(props);

        this.state = {
            //State variables here
        }
    }
    render() {
        return ( ... )
    }
}
```

3.2.2 Setting up State

Now we need to go ahead and define a few state variables that will allow our page to be responsive.

Listing 14: Calculator.js

```
    ...
        this.state = {
            courses: [],
            newCourseName: '',
            newCourseGrade: '',
        }
    ...
```

We will use three variables, two for the new course information and an array to store the courses as we add them. Now we can start to modify the render function of our component. First, import the components we are going to be using.

Listing 15: Calculator.js

```
    import { StatusBar, StyleSheet } from 'react-native';

    import { Container, Content, Search, List, ListItem, Text, Button, Left
        , Right, Input, Form, Item } from 'native-base';

    ...
```

It is important to note that React expects to only see one component at the root of the render function so we will start with a Container and put the rest of our components inside. We will also go ahead and put our text component that will display the GPA.

Listing 16: Calculator.js

```
    render() {
        return (
            <Container >
                <Header transparent>
                    <Body>
                        <Text>
                            GPA Calculator
                        </Text>
                    </Body>
                </Header>
                <Text>
                    Your GPA: -
                </Text>
            </Container>
        );
    }
```

Before proceeding, return to App.js and import our new component then render it inside the render function.

Listing 17: Calculator.js

```
import Calculator from './src/Calculator';

...

render() {
    return (
        <StyleProvider style={getTheme(variables)}>
            <Calculator />
        </StyleProvider>
    );
}
```

Below the GPA text there will need to be a form to enter course codes, grade received and a button. We can use a native base Form component to accomplish this. We will do this right underneath the text component.

Listing 18: Calculator.js

```
    <Form>
        <Item rounded>
            <Input placeholder="Course Name" value={
                this.state.newCourseName}/>
        </Item>
        <Item rounded>
            <Input placeholder="Course Grade" value={
                this.state.newCourseGrade}/>
        </Item>
    </Form>
    <Button rounded color='error'>
       <Text>
            Add Course
        </Text>
    </Button>
```

Note that when we use a prop next to a component without setting any value, it is assumed to be a boolean that you are setting to true (<Item rounded> is the same as <Item rounded={true}>). Notice that the value prop of our Input fields are being set to the state variables that we declared previously. We need to access these variables by using this.state. before the name of the variable. Since these variables are declared in the state of the component, any changes to their values will automatically be re-rendered.

3.2.3 Updating States with setState

React does not allow us to modify these state variables directly, so instead we must use the setState() function. If you type in the Input fields right now, there will not be any changes to the text box. In order to make this happen, we need to give both of our Input components another prop called onTextChange.

Listing 19: Calculator.js

```
    <Form>
        <Item rounded>
            <Input placeholder="Course Name" value={
                this.state.newCourseName} onChangeText={
                this.handleNameChange}/>
        </Item>
        <Item rounded>
            <Input placeholder="Course Grade" value={
                this.state.newCourseGrade} onChangeText={
                this.handleGradeChange}/>
        </Item>
    </Form>
    <Button rounded color='error'>
       <Text>
            Add Course
        </Text>
    </Button>
```

We need to define a function inside of the onChangeText props which will get executed every time the value inside in Input field is updated. In this case we used, this.handleNameChange and this.handleGradeChange which we will now need to define.

3.2.4 Defining Functions in a Component

Lets move outside of the render function in order to define our new functions.

Listing 20: Calculator.js

```
    // underneath the constructor
    handleNameChange = (text) => {
        this.setState({newCourseName: text})
    }
    handleGradeChange = (text) => {
        this.setState({newCourseGrade: text})
    }
```

This is the first time we have used the setState() function. You can update multiple state variables in a single setState call (as we will see momentarily) **but is important to note that setState() is an asynchronous function.** This means you should not write code immediately after calling setState while assuming that the state variables have already been updated. Read the setState documentation for more details. So now that our text fields are updating properly and storing their values in their respective setState fields, we need to get our 'add course' button to work. Go back to your render function and update the Button component:

Listing 21: Calculator.js

```
<Button rounded color='error' onPress={() => {this.addCourse()}}>
   <Text>
        Add Course
    </Text>
</Button>
```

We are adding an onPress prop which expects a function that will get executed whenever the user presses the button. Lets go back outside of the render function and define our addCourse() function:

Listing 22: Calculator.js

```
    addCourse = () => {
        // This syntax allows us to refer to variables in our state by
            their names without having to begin with this.state every time.
        const { courses, newCourseGrade, newCourseName } = this.state;

        // We are create an object called newCourse which will be formatted
             in a style similar to JSON
        const newCourse = {
            name: newCourseName,
            grade: newCourseGrade
        }
        // Concatenate our new course to our list of existing courses and
            reset the Input fields to empty
        this.setState({
            courses: courses.concat([newCourse]),
            newCourseName: '',
            newCourseGrade: '',
        });
    }
```

So now we are able to add our courses into our state array but we still need a way to render these in some sort of list.

3.2.5 Rendering our Courses

Lets return to our components render function and find our where we are rendering our form. Above this lets use the native base <List> component. This component may have zero or more <ListItem> components that it will render. We want a list entry for each of the courses that have been added to our states courses array so we will user a JavaScript function called map to iterate through the array and render the correct number of <ListItem> elements programmatically.

Listing 23: Calculator.js

```
...
    <List>
    // To use javascript in conjunction with JSX you need to surround the
        javascript with braces.
    // Think of this map as a for each item loop
    // For each item in courses, course render a listitem with key equal to
         i the current index of the array
    // Generally when rendering components like this a unique key needs to
        be assigned to each element
    {this.state.courses.map((course, i) => (
        <ListItem key={i}>
            <Left>
                <Text>{course.name}</Text>
            </Left>
            <Right>
                <Text>Grade: {course.grade}</Text>
            </Right>
        </ListItem>
    ))}
    </List>
...
```

3.2.6 Styling

Styling is done in a manner very similarly to standard CSS. We will first need to import StyleSheet from react-native.

Listing 24: Calculator.js

```
import { StatusBar, StyleSheet } from 'react-native';
```

We need to declare our styles outside of our class. Create a const variable called styles and use StyleSheet.create() on the following styles:

Listing 25: Calculator.js

```
const styles = StyleSheet.create({
    headerBackground: {
        backgroundColor: '#8600b3',
    },
    headerText: {
        color: '#fff',
    },
    container: {
        justifyContent: 'center',
        alignItems: 'center'
    },
    button: {
        alignSelf: 'center',
        marginTop: 10,
    },
    formField: {
        marginLeft: 20,
        marginRight: 20,
        marginTop: 10,
    },
    text: {
        marginTop: 10,
        alignSelf: 'center',
        color: '#8600b3',
    }
});
```

Apply these styles simply requires the use of the style prop on the corresponding components. Update the following lines by adding the correct styles:

Listing 26: Calculator.js

```
// Also add the iosBarStyle={"light-content"} prop to make the text white
<Header transparent iosBarStyle={"light-content"} style={
    styles.headerBackground}>

<Text style={styles.text}>
    Your GPA: -
</Text>

<Text style={styles.text}>{course.name}</Text>

<Text style={styles.text}>Grade: {course.grade}</Text>

<Item rounded style={styles.formField}>
    <Input placeholder="Course Name" value={this.state.newCourseName}
        onChangeText={this.handleNameChange}/>
</Item>
<Item rounded style={styles.formField}>
    <Input placeholder="Course Grade" value={this.state.newCourseGrade}
        onChangeText={this.handleGradeChange}/>
</Item>

<Button rounded color='error' onPress={() => {
    this.addCourse()
}} style={styles.button}>
```

3.2.7 Making the GPA Field Update

// TODO...