# Mobile Development (CM3050)

**Course Notes**

Felipe Balbi

January 12, 2022

# Contents

# Week 1

Key Concepts

- Understand the limitations and advantages of different platforms
- Discuss the elements of apps you enjoy
- Understand the course structure

## 1.006 Getting started on this module

React Native is an Open Source framework created by Facebook which lets us JavaScript to produce native code for different operating systems. Some of the benefits are:

1. Single language for multiple platforms
2. Ease and flexibility of JavaScript
3. Near-native performance
4. Heavily used in industry

## 1.103 The pathway from developer to consumer

The process to moving an app from developer to consumer is roughly laid out below.

1. Ideation

   The idea for an app has to come from somewhere

2. Storyboarding

   Drawing an idea is usually the simplest form of conveying that idea

3. Prototyping

   Helps us understanding challenges that may arise from implementing the idea

4. Feedback and Production

   Using our prototype, we can gather feedback to understand if the app fulfills its purpose

5. Testing

   Automated testing and manual testing is an important part of guaranteeing stability and quality of the app

6. Approval and Release

   After testing, apps go through an approval process by e.g. Google and Apple before they can be released to end users

7. On sale

   Apps are finally available in the App Store and can be purchased by anyone

## 1.201 Multiple codebases

There are two major mobile OSes:

- iOS ($\approx 26.5\%$ market share)
- Android ($\approx 72.9\%$ market share)

This means that developers usually have to write the same app twice — once for iOS with Switch of Objective-C, and once for Android using Kotlin or Java —; needless to say this creates the possibility of minor incompatibilities between the two versions of the app.

The alternative to writing two apps is to write a single app in a common language that can be compiled for both platforms.

**Native Apps** Written specifically for a single OS

**Hybriid Apps** Written in a common language

## 1.203 Native apps vs hybrid apps

React Native, while using a common language for all platforms, is much closer to a native application. During compilation of the project, React Native renderas our views with native code on the target platform thus giving us the best of both worlds.

In table 1 we list pros and cons of Native Apps. Similarly, table 2 shows the same comparison for Hybrid Apps.

Table 1: Native Apps Pros & Cons

| **Pros** | **Cons** |
|---|---|
| Fast | Lengthy and Complex to build |
| Device-level APIs | No cross-compatibility |
| Native GUI Elements | |

Table 2: Hybrid Apps Pros & Cons

| **Pros** | **Cons** |
|---|---|
| Cross-compatibility | Slower perforance |
| Larger market reach | No device-level APIs |
| Faster to write | |

## 1.301 What is React Native?

React Native is a framework written in JavaScript that allows us to build code for multiple platforms.

## 1.302 What is Expo?

Expo helps manage a React Native project.

## 1.305 Javascript arrow notation

A traditional JavaScript function looks like this:

```
function (a) {
  return a * 2;
}
```

We can convert it Arrow Notation like so:

```
(a) => {
  return a * 2;
}
```

Single line functions can be further simplified by removing braces and the `return` keyword:

```
(a) => a * 2;
```

Next, the parenthesis around `(a)` can also be removed:

```
a => return a * 2;
```

Functions without arguments or with more than one argument must keep parenthesis around arguments:

```
() => 42;
(a, b) => a**2 + 2*a*b + b**2;
```

## 1.306 Creating a new React Native project and overview of the file structure

Create a new project with:

```
$ expo init --npm
```

Choose a name for the application and choose the `blank` template. Hitting `ENTER` will create a new directory with the name of the application and all dependencies already installed. When it completes, we can change into the new application's directory and run:

```
$ expo start
```

# Week 2

Key Concepts

- Understand the limitations and advantages of different platforms
- Discuss the elements of apps you enjoy
- Understand the course structure

## 1.401 JSX and props

JSX is a key part of React Native. It allows us to create components and use them as if they were regular HTML tags. Component names must start with a capital letter.

Props can be passed in an HTML `attribute=value` format.

## 1.501 Testing on simulators and devices

Testing the application on a device while under development can help us uncover issues early. The simulator is another option which works well.

## 1.601 Marketplaces and conforming to the rules

Reasons for rejection

1. App doesn't work
2. Use of copyrighted material
3. Breach of safety guidelines
4. Safeguarding issues
5. Circumventing the app stores
6. Low-quality design

# Week 3

Key Concepts

- Understand the need for wireframing
- Discuss the link between psychology and design decisions
- Understand and identify different design styles and replicate them using code

## 2.001 Skeuomorphism, minimalism and neumorphism

**Skeuomorphism** emulating the look of physical objects in your application

**Minimalism** prioritization of essential elements

**Neumorphism** the middle-ground between skeuomorphism and minimalism

## 2.003 Colour palettes and mood psychology

**Blue** calmness, serenity, stability, reliability

**Red & Yellow** speed, immediacy, increase appetite

**White** simplicity, calmness, professionalism

## 2.006 Design ideology

- Apple Human Interface Guidelines
- Google Material Design Guidelines

## 2.101 What are dark patterns?

Dark Patterns is the term given to design choices that are crafted to make you do things you don't want to do.

## 2.103 Dark patterns

- Dark Patterns website

## 2.201 Introduction to wireframing

Wireframes allow us to design layouts in order to demonstrate an idea and check if it's likely to be practical.

## 2.203 Creating a wireframe for a shopping app

Before wireframing, it's a good to create a User Flow Diagram. After we can start working on our wireframe.

# Week 4

Key Concepts

- Understand the need for wireframing
- Discuss the link between psychology and design decisions
- Understand and identify different design styles and replicate them using code

## 2.401 How to style elements in React Native

Styles, in `React`, are created with `StyleSheet.create()` method. We can use them by setting the `style` property of the element to the relevant object. Like so:

```
import { StatusBar } from 'expo-status-bar';
import React from 'react';
import { StyleSheet, Text, View } from 'react-native';

export default function App() {
  return (
      <View style={styles.container}>
        <Text>Open up App.js to start working on your app!</Text>
        <StatusBar style="auto" />
      </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  }
});
```

In this case, the `<Text>` and `<StatusBar>` elements are children of the `<View>` element and will, therefore, *inherit* its styling.

## 2.501 What is responsive design?

With so many different models of devices in the market, each with a different pixel density, screen sizes, and resolution, designing a single app that works on all form factors is a rather large task.

Responsive design is a technique where the application's UI adapts to the screen dimensions. We do this by building apps designed for proportions, rather than specific sizes.

In ReactNative, Flexbox is used to provide a consistent layout on different screens.

## 2.504 React Native flex

- React Documentation: Flexbox

## 2.505 Apple developer guidelines

- Apple Developer Guidelines: Adaptivity And Layout

## 2.601 Design with accessibility in mind

When building an application, four categories should be kept in mind:

1. Vision

   Blindness, color blindness and all forms of vision impairment

2. Hearing

   Hearing loss and other hearing disabilities

3. Physical and motor

   People with physical and motors impairment, may experience difficulties holding or manipulating their devices

4. Literacy and learning

   These include difficulty speaking, reading, managing complexity and maintaining attention or focus.

ReactNative wraps Android and iOS specific accessibility features.

`accessible` **Prop** When true, indicates the view is an accessibility element.

`accessibilityLabel` **Prop** Allows us to assign descriptive label to an element.

`accessibilityHint` **Prop** Describes what will happen when the user performs the action on the element.

**Accessibility Actions** Allow assistive technologies to programmatically perform actions.

## 2.603 Accessibility in computational technology

- Antona, M. and C. Stephanidis (eds) Universal Access in Human-Computer Interaction. Design Approaches and Supporting Technologies. 14th International Conference, UAHCI 2020, Held as Part of the 22nd HCI International Conference, HCII 2020. (Springer International Publishing, 2020).

# Week 5

Key Concepts

- Use JSX to create basic elements on the screen
- Understand the need for pagination
- Understand and be able to use UI elements in React Native

## 3.001 UI elements and the OS

`ActivityIndicator` Shows background activity

`Button` Default UI Button element

`Image` An element that renders an image

`ImageBackground` Fills a space with an image, resizing it where applicable

`Switch` Regular toggle switch element

`Text` Renders text in the screen

`TextInput` Lets user input text into a field

`TouchableOpacity` Area capable of receiving touch events

`View` A basic *container* element. Can be thought of as sheets of paper

## 3.004 React Native component documentation

- React: Core Components And APIs

# Week 6

Key Concepts

- Use JSX to create basic elements on the screen
- Understand the need for pagination
- Understand and be able to use UI elements in React Native

## 3.301 Why pages are important

Navigation and Pages create a clear separation of our content. It turns our content into easily understandable chunks of information.

## 3.303 Navigation in React Native

In order to use navigation within a React Native app, we must first install and import the React Navigation library.

```
$ npm install @react-navigation/native
$ expo install react-native-screens react-native-safe-area-context
$ npm install @react-navigation/native-stack
```

After installing dependencies we can import with:

```
import { NavigationContainer } from '@react-navigation/native';
import { createStackNavigator } from '@react-navigation/stack';
```

We can use the `NavigationContainer` as the root element of our app with a `Stack.Navigator` inside. Each *view* in the stack can be modeled as a function that returns the required *view*. The full setup will look like so:

```
import { StatusBar } from 'expo-status-bar';
import React from 'react';
import { StyleSheet, Text, View } from 'react-native';
import { NavigationContainer } from '@react-navigation/native';
import { createStackNavigator } from '@react-navigation/stack';

function HomeScreen() {
```

```
  return (
    <View>
      <Text>Home Screen</Text>
    </View>
  );
}

const Stack = createStackNavigator();

export default function App() {
  return (
    <NavigationContainer>
      <Stack.Navigator>
        <Stack.Screen name="Home" component={HomeScreen} />
      </Stack.Navigator>
    </NavigationContainer>
  );
}
```

## 3.305 How to programatically navigate

```
import { StatusBar } from 'expo-status-bar';
import React from 'react';
import { StyleSheet, Text, View, Button } from 'react-native';
import { NavigationContainer } from '@react-navigation/native';
import { createStackNavigator } from '@react-navigation/stack';

function HomeScreen({navigation}) {
  return (
    <View>
      <Text>Home Screen</Text>
      <Button title="To details"
              onPress={() => navigation.navigate("Details")}/>
    </View>
  );
}

function DetailsScreen() {
  return (
    <View>
      <Text>Details Screen</Text>
      <Button title="Back Home"
              onPress={() => navigation.navigate("Home")}/>
```

```
23      </View>
24    );
25  }
26
27  const Stack = createStackNavigator();
28
29  export default function App() {
30    return (
31      <NavigationContainer>
32        <Stack.Navigator>
33          <Stack.Screen name="Home" component={HomeScreen} />
34          <Stack.Screen name="Details" component={DetailsScreen} />
35        </Stack.Navigator>
36      </NavigationContainer>
37    );
38  }
```

## 3.307 Navigation in React Native guide

- React Navigation

# Week 7

Key Concepts

- To develop a stronger understanding of advanced techniques
- Use advanced methods of interaction
- Understand and program animations

## 4.101 What are table views?

ReactNative does not provide native table view components. There are several repositories providing the functionality. We will use react-native-tableview-simple.

To install it we use:

```
npm i react-native-tableview-simple
```

And use it by importing:

```
import { Cell, Section, TableView} from 'react-native-tableview-simple';
```

Table views are comprised of *sections* and *cells*. A *section* is a group of *cells*. While *sections* are not required for a table view, it's encouraged to use them to separate content into categories.

The basic table view looks like so:

```
<TableView>
  <Section>
    <Cell title="Cell 1"/>
    <Cell title="Cell 2"/>
  </Section>
</TableView>
```

We can also create cells programmatically:

```
let items = ["Cell 1", "Cell 2"];

return (
  // ...
  { items.map((element, i) => (
```

```
      <Cell title={element}/>
    ))}
);
```

Mapping over an array of objects is more idiomatic:

```
let items = [{"title": "Cell 1"}, {"title": "Cell 2"}];

return (
  // ...
  { items.map((element, i) => (
    <Cell title={element.title}/>
  ))}
);
```

Custom cells can be created with the help of the `cellContentView` prop.

```
<Cell
  cellContentView = {
    <View>
      <Text>Testing!</Text>
    </View>
  }
/>
```

These can also be turned into static components which can be referenced later:

```
const CellVariant = (props) => (
  <Cell
    {...props}
    cellContentView = {
      <View>
        <Text>Testing!</Text>
      </View>
    }
  />
)

// ...

return (<CellVariant/>)
```

## 4.201 What are scroll views and flat lists?

Scroll Views allow us to display content that wouldn't normally fit the screen. `ScrollView` will render every child, even the ones that can't be seen until scrolled into view.

A `FlatList` is the perfect alternative to `ScrollView` because it will lazily load items.

## 4.202 Performance monitoring in React Native

When building the application in debug mode, we can shake the device in order to display a react native debug menu, which includes performance monitoring.

# Week 8

Key Concepts

- To develop a stronger understanding of advanced techniques
- Use advanced methods of interaction
- Understand and program animations

## 4.401 What are animations?

Animations serve to provide feedback to actions taken by the user. There are lots of animations built into the mobile OS. Some of which are launcher effects, *elastic banding* with scrolling, and sleep/wake animations. We can also make custom animations.

ReactNative's Animated API allows us to build simple time-based animations. It can be used with the following components:

1. `View`
2. `Text`
3. `Image`
4. `ScrollView`
5. `FlatList`
6. `SectionList`

To use animations with a components **not** on the list above, we can do so with `Animated.createAnimatedComponent()`.

```
const AnimatedText = (props) => {
  return (
    <Animated.Text
      style = {{
        ..props.style,
        opacity: textOpacity,
      }}
    >
      TEXT
```

```
    </Animated.Text>
  )
}

const textOpacity = useRef(new Animated.Value(0)).current;
React.useEffect(() => {
  Animated.timing(textOpacity,
                  {
                    toValue: 1,
                    duration: 3000,
                  }
                 ).start();
}, [textOpacity])
```

## 4.403 Composing more complex animations

```
const AnimatedText = (props) => {
  const textOpacity = useRef(new Animated.Value(0)).current;

  React.useEffect(() => {
    Animated.timing(
      textOpacity,
      {
        toValue: 1,
        duration: 3000,
        /* useNativeDriver: true */
      }
    ).start();
  }, [textOpacity]);

  return (
    <Animated.Text
      style={{
        ...props.style,
        opacity: textOpacity
      }}
    >
      <props.children}
    </Animated.Text>
  )
}

export default function App() {
```

```
28    return (
29      <View style={styles.container}>
30        <AnimatedText>
31          Test!
32        </AnimatedText>
33      </View>
34    )
35  }
```

## 4.405 React native animation docs

- React Native Animated Documentation

## 4.501 What are gestures?

Gestures are user input recognized as more than a regular tap. Examples are:

- Double tap
- Scrolling
- Sliding
- Pinch to zoom

The *Gesture Responder System* manages the lifecycle of gestures in our app. React Native provides the `TouchableOpacity` element for our touch processing. Another useful element is the `PanResponder`, which provides a wrapper for the responder handlers.

Table 3 below summarizes the `gestureState` object's properties:

Table 3: `gestureState` Object Properties

| Property | Description |
|---|---|
| `stateID` | Unique identifier |
| `moveX`, `moveY` | Last touch coordinates |
| `x0`, `y0` | Screen coordinates of responder grant |
| `dx`, `dy` | Accumulated traveled distance since first touch down |
| `vx`, `vy` | Gesture velocity |
| `numberActiveTouches` | Number of the touches currently on screen |

## 4.601 What are alerts?

Alerts serve to notify the user of something that needs immediate attention. To use alerts we must first import `Alert` from `react-native`. The simplest alert can be called using:

```
Alert.alert("message");
```

The full list of possible parameters are shown below:

```
alert(title, message?, buttons?, options?);
```

where:

`title` The dialog's title. Pass `null` to hide

`message` optional message that appears below the title

`buttons` optional array with button configurations

`options` **Android-specific** configuration

Here's a sample `button` array:

```
[
  {
    text: "Ok",
    onPress: () => console.log("Ok pressed"),
  },
  {
    text: "Cancel",
    onPress: () => console.log("Cancel pressed"),
    style: "cancel",
  },
]
```

There are three `AlertButtonStyle` enums:

`default` The default button style

`cancel` Often represented as red text

`destructive` Often represented as *permanent* action, e.g. a solid red background

On iOS we can define as many buttons as we'd like, but on Android there is a hard-limit of three buttons.

## 4.701 What are timers?

Timers allow us to execute code after a set delay.

# Week 9

Key Concepts

- Learn advanced Javascript techniques
- Understand and implement testing
- To discuss what makes efficient programming

## 5.001 Efficient programming

Clean code makes it easier to understand and maintain. Some features of clean code are:

**Comments** Comments serve as documentation for the intention of the code

**Modularization** Modular code separates concerns and makes it easy to focus on a small part of the code at a time

**Multiple Files** By breaking different components to multiple files, we can focus on a single at a time without all the extra clutter

`console.log()` Only use it for debugging

**Naming** Making use of clear, descriptive names helps with code readability

## 5.101 What is automated testing?

Testing helps ensure that a change to the codebase doesn't introduce any regressions. Modularized code allows for testing small modules in isolation.

To structure tests, we follow *AAA* (Arrange, Act, Assert). The test definition below:

> given a date in the past, `colorForDueDate()` returns red.

Translates into the following test definition:

```
1 it('given a date in the past, colorForDueDate() returns red',
2    () => {
3      expect(colorForDueDate('2000-10-20')).toBe('red');
4    });
```

Unit test cover small units of code, e.g. individual functions or classes.

Mocking allows us to override dependencies in order to provide a predictable, testable state.

Integration testing will verify that multiple functions or classes work well together.

Component testing verifies that a components behaves correctly when interacted with and that it renders as expected.

## 5.201 Functional components vs Class components

A Functional Component is simply a component created using a function:

```
function FunctionalComponent() {
  return (
    <View>
      <Text>Testing</Text>
    </View>
  )
}
```

A Class component needs a few additions:

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

class ClassComponent extends Component {
  render() {
    return (
      <View>
        <Text>Testing</Text>
      </View>
    )
  }
}
```

Functional components usually require less code to achieve the same functionality.

## 5.203 React Native documentation: Class components

- React Native Class Components Documentation

# Week 10

Key Concepts

- Learn advanced Javascript techniques
- Understand and implement testing
- To discuss what makes efficient programming

## 5.301 Syntax transformers

### Block Scoping

`var` creates variables with function scope while `let` obeys block scoping.

```
function whoAmI() {
  if (true) {
    var name = "foo"
    let location = "home";
  }

  console.log(name); /* foo */
  console.log(location); /* undefined */
}

whoAmI();
```

### Destructuring

Destructuring lets us *unpack* the values inside an object or array into distinct variables.

```
let person = {
  name: "Foo",
  location: "Home",
  device: "Phone"
}
let {name, device} = person;

let bar = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
let [first, second, ...rest] = bar;
```

```
10
11 console.log(name); /* Foo */
12 console.log(device); /* Phone */
13 console.log(first); /* 1 */
14 console.log(second); /* 2 */
15 console.log(rest); /* [3, 4, 5, 6, 7, 8, 9, 10] */
```

for of

```
1  let nums = [1, 2, 3, 4];
2
3  for (let num of nums) {
4    console.log(num);
5  }
6
7  /* 1
8   * 2
9   * 3
10  * 4
11  */
```

## 5.303 React Native documentation: Syntax transformers

- JavaScript Environment

## 5.304 Performance enhancement

The `InteractionManager` allows us to schedule tasks to run at a later time when there are no interactions:

```
1 InteractionManager.runAfterInteractions(() => {
2   /* Expensive computation here */
3 });
```

# Week 11

Key Concepts

- To handle and manipulate data
- To understand the importance of data ethics
- To understand what data sources includes

## 6.001 Data sources

Apps usually communicate with different data sources (web servers, API endpoints, local storage, ...) in order to gather data to display to the users.

## 6.003 Ethics

In summary, don't collect user data unless it's essential for the functionality of the app. If we do collect user data, we must protect it.

## Red Hat API Security

- API Security Blog post

## 6.101 What is JSON and XML?

JSON and XML are very common data exchange formats.

## 6.103 Loading JSON from a file

```
1  import { StatusBar } from 'expo-status-bar';
2  import React from 'react';
3  import { StyleSheet, Text, View } from 'react-native';
4  import JSONdata from './test.json'; /* parses the string automatically */
5
6  export default function App() {
7    return (
8      <View style={styles.container}>
```

```
        <Text>Hello {JSONdata.name}</Text>
        <StatusBar style="auto" />
      </View>
    );
}

const styles = Stylesheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
});
```

## 6.201 Saving data

`AsyncStorage` is a recommended library to handle saving data **unencrypted** data to phone storage. It can be installed with the command below:

```
expo install @react-native-async-storage/async-storage
```

Here's a primer on how to use it:

```
import AsyncStorage from '@react-native-async-storage/async-storage';

/* Saving */
AsyncStorage.setItem('@testKey', "test").then(() => {
  console.log("Saved");
}).catch((err) => {
  console.log(`ERROR: ${err}`);
});

/* Retrieving */
AsyncStorage.getItem('@testKey').then((value) => {
  console.log(value);
}).catch((err) => {
  console.log(`ERROR: ${err}`);
});

```

For encrypted data, we can `SecureStore`. There is a limit of 2048 bytes for encrypted data. Install it with:

```
expo install expo-secure-store
```

Here's a primer on how to use it:

```
import * as SecureStore from 'expo-secure-store';

/* Saving */
SecureStore.setItemAsync("testKey", "test").then(() => {
  console.log("Saved");
}).catch((err) => {
  console.log(`ERROR: ${err}`);
});

/* Retrieving */
SecureStore.getItemAsync("testKey").then((value) => {
  console.log(value);
}).catch((err) => {
  console.log(`ERROR: ${err}`);
});
```

## 6.203 AsyncStorage and SecureStore documentation

- AsyncStorage Documentation
- SecureStore Documentation

## 6.204 Saving data in React Native

```
import { StatusBar } from 'expo-status-bar';
import React from 'react';
import { StyleSheet, Text, View, TextInput } from 'react-native';
import AsyncStorage from '@react-native-async-storage/async-storage';

export default function App() {
  const [text, onChangeText] = useState("");

  if (text == "") {
    AsyncStorage.getItem('@textInput').then((value) => {
      onChangeText(text);
    }).catch((err) => {
      console.log(`ERROR: ${err}`);
    });
  }

  return (
    <View style={styles.container}>
```

```
19        <TextInput
20          style={styles.input}
21          onChangeText={async (text) => {
22            await AsyncStorage.setItem('@textInput', text);
23            onChangeText(text);
24          }}
25          value={text}
26          placeHolder={"Value"}
27        />
28        <StatusBar style="auto" />
29      </View>
30    );
31  }
32
33  const styles = Stylesheet.create({
34    container: {
35      flex: 1,
36      backgroundColor: '#fff',
37      alignItems: 'center',
38      justifyContent: 'center',
39    },
40  });
```

# Week 12

Key Concepts

- To start work on the final project
- To give feedback on others ideas
- To develop an idea for the final project