Table of Contents

Introduction 1.1

Monday 1.2

- Schedule 1.2.1
- Links 1.2.2
- Equipment 1.2.3
- Software 1.2.4

Tuesday A 1.3

- Expo 1.3.1
- React 1.3.2
  - Example component 1.3.2.1
  - Example state 1.3.2.2
- View 1.3.3
- Text 1.3.4
  - Example Text 1.3.4.1
- Styles 1.3.5
- Flexbox 1.3.6
  - Example Flexbox 1.3.6.1
- Exercise 1.3.7
  - Solution 1.3.7.1

Tuesday B 1.4

- TextInput 1.4.1
  - Example TextInput 1.4.1.1
- Touchables 1.4.2
  - Example Button 1.4.2.1
  - Example TouchableOpacity 1.4.2.2
- Exercise 1 1.4.3
  - Solution 1.4.3.1
- Image 1.4.4
  - Example remote image 1.4.4.1
  - Example local image 1.4.4.2
- Networking 1.4.5
  - Example fetch 1.4.5.1
- ScrollView 1.4.6
  - Example ScrollView 1.4.6.1
- FlatList 1.4.7
  - Example FlatList 1.4.7.1
- Exercise 2 1.4.8

Solution 1.4.8.1
Wednesday A 1.5
react-native init 1.5.1
Project structure 1.5.2
Bridge architecture 1.5.3
Troubleshooting 1.5.4
Debugging 1.5.5
Exercise 1.5.6
Wednesday B 1.6
AsyncStorage 1.6.1
Example AsyncStorage 1.6.1.1
React Navigation 1.6.2
Example React Navigation 1.6.2.1
Exercise 1.6.3

Intro

Welcome to Code in the Woods *Autumn 2018* brought to you by Barona Technologies, Forenom, and Andre Staltz!

This year we will learn how to build mobile apps for iOS or Android using React Native and JavaScript.

This booklet contains basic course material for all days during the camp, so you can follow at your own pace and not miss any important information. Feel free to make any questions to the teacher or the mentors at any point. :)

Monday

Today we will form teams (of 2 or 3 persons), learn basic informations about React Native and this course, and set up our computers with the necessary tools.

Course Schedule

The schedule for this code camp is divided into 5 parts:

- Monday
  - Teaming up
  - Why React Native
  - Goal for this camp
  - Intro to the schedule
  - Course material
  - Demo in Expo
  - Setting up computers
- Tuesday A
  - Expo, React, Views, Texts, Flexbox
  - *Exercise (colorful rectangles)*
- Tuesday B
  - TextInput, Touchable
  - *Exercise (submit new word)*
  - Image, fetch, ScrollView, FlatList
  - *Exercise (infinite list of images from jsonplaceholder)*
- Wednesday A
  - Eject, local project structure, bridge architecture, debugging tricks
  - *Exercise (compile yesterday's project locally)*
- Wednesday B
  - AsyncStorage
  - React Navigation, (bonus?) Native modules
  - *Exercise (build your own real app!)*

Links

The following links are useful for getting more information on the development tools.

Documentation

- React Native documentation
- React Native issues
- iOS Developer portal
- Android Developer portal
- React Navigation
- DevDocs.io (for general JavaScript, git, Node.js, npm)

Tools

- Xcode (for iOS)
- Android Studio
- Expo editor (Snack)
- React DevTools

Apps

- Expo client
- Recipe Explorer demo

Community resources

- Awesome React Native (libraries and components)
- JS.coach (curated React and React Native libraries)
- Public APIs (backends with open data)

Equipment

For this course, you will need your own **laptop**, **smartphone**, and **USB cable** to connect the smartphone to the laptop. The laptop can run either Windows or macOS or Linux.

Note that if you have an iPhone, you cannot develop apps for it unless you have Macbook!

If it is not possible to have the right combination of phone and computer, then pair up with another person in your team.

Software

This page describes the software and apps you will have to install to prepare your computer for developing apps with React Native. Try to follow the order given in this page.

Install Expo on your phone

Install the Expo app on your smartphone (either iOS or Android).

Test that Expo works

- Open snack.expo.io on your computer's browser
- Scan the QR code
  - In the website, press the **Run** button to open a QR code
  - Open Expo app on your smartphone, choose Scan, and aim it at the computer
- **OR** Insert the Device ID
  - In the website, press the **Run** button and select Device ID
  - Open Expo app on your smartphone, and look for the Device ID at the bottom
  - Input the Device ID into the website on your computer
- Check that the example app opens correctly

Basics in your terminal

We will be using the terminal a lot, so it's important to get familiar with terminals (also known as "console" or "Command Prompt" in Windows). Here's a quick guide to use `ls` and `cd` in the macOS Terminal.

Setup essentials

On macOS: install Brew and Chrome (will be used for debugging)

On Windows: install Chrome (will be used for debugging) and optionally install Cmder mini

On Linux: install Chrome (will be used for debugging)

Install git

On macOS:

```
brew install git
```

On Windows:

From the website.

On Linux:

From the website.

Install nvm

We recommend using nvm which manages multiple versions of Node.js.

On macOS or Linux:

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.33.11/install.sh | bash
```

On Windows:

From the nvm-windows website

Install Node.js and npm

```
nvm install v8.11
```

```
nvm use v8.11
```

Install a code editor

We recommend Visual Studio Code which is available for macOS, Windows, and Linux, unless you already have a preferred editor.

Install React Native tools

Follow these getting started instructions carefully. Choose **Building Projects with Native Code**, choose your **Development OS**, your **Target OS**, and follow the installation instructions.

Test that you can compile a project

To make sure that all the tools are working correctly for the course, try compiling the demo app (Recipe Explorer).

Download the project code:

```
git clone https://github.com/staltz/citw-demo.git
```

Go into the project folder:

```
cd citw-demo
```

Install dependencies:

```
npm install
```

Run the Android app:

Plug the phone to your computer using the USB cable, then run:

```
react-native run-android
```

Or **run the iOS app:**

Plug the iPhone to your Macbook using the USB cable, then run:

```
react-native run-ios
```

If the app shows on your phone with a pink header "Recipe Explorer" and a list of recipes, then you're all set!

Tuesday A

In this session, we are going to learn the basics of *React Native* using Expo, a tool that makes it very easy to quickly go from code to real app. We will learn about basic layout components, and styling.

Expo

Expo is a development environment that only requires a browser and a phone. It is easy to get started, and we will use it to learn the basics of React Native without getting into other kinds of compilation troubles.

If you haven't yet, read the Software installation guide to learn how to use Expo.

React

React Native uses React, a JavaScript framework for building user interfaces (UIs) on the Web or mobile.

Components

The center piece of React is the idea of a **component**, which is a reusable piece of UI code. You can think of a component as a tiny slice of the screen.

A component typically looks like this in JavaScript:

```
import {Component} from 'react';
import {Text} from 'react-native';

class Welcome extends Component {
  render() {
    return <Text>Hello, world</Text>;
  }
}
```

The most important part of a component is the **render** method. It returns a description of how that component should look like on the screen.

Nesting

Once you have created a React component, you can embed it inside the `render` method of another React component.

```
class FirstScreen extends Component {
  render() {
    return <Welcome />;
  }
}
```

Lifecycle

Because components are classes in JavaScript, it initially calls the `constructor`, but there are also more events along the life of a component.

In specific `componentDidMount` is a special method that will be called when the component was just recently inserted into the UI, and `componentWillUnmount` will be called when the component is about to be removed from the UI. You can use these moments to trigger special actions. In the example below, we display popups when the component enters or exits the screen.

```
class Welcome extends Component {
  componentDidMount() {
    alert('*Welcome* was inserted on the screen');
  }

  componentWillUnmount() {
    alert('*Welcome* will be removed from the screen');
  }

  render() {
    return <Text>Hello, world</Text>;
```

```
  }
}
```

Props

When nested, a child component can take arguments that allow us to customize how the child looks or behaves. These arguments are known as "props" in React jargon. In the example below, we modified `Welcome` so that it displays the name given to it by its parent `FirstScreen`.

```
import {Component} from 'react';
import {Text} from 'react-native';

class Welcome extends Component {
  render() {
    return <Text>Hello, {this.props.name}</Text>;
  }
}

class FirstScreen extends Component {
  render() {
    return <Welcome name={"Alicia"} />;
  }
}
```

Props also allow the parent component to control how the child component changes over time: if the parent passes a different prop value to the child, then the child will update itself to use the latest prop.

State

A component can also modify itself internally, by updating its own internal `state`. State is an object in every component, accessible via `this.state` and can be changed with the special method `this.setState(newStateObject)`.

In the example below, we make the Welcome component blink every second by updating a part of the `state`.

```
class Welcome extends Component {
  constructor(props) {
    super(props); // always needed if you have a constructor
    this.state = {show: true}; // Initialize the state!
  }

  // Once the component is inserted on the screen, begin a timer
  // to toggle the `show` boolean back and forth
  componentDidMount() {
    setInterval(() => {
      this.setState({show: !this.state.show});
    }, 500);
  }

  render() {
    return <Text>{this.state.show ? 'Hello, world' : ''}</Text>;
  }
}
```

Even though the code above works, it's wise to clean up the interval timer in case the Welcome component leaves the UI. We can do that in `componentWillUnmount`.

```
class Welcome extends Component {
  constructor(props) {
```

```
    super(props);
    this.state = {show: true};
  }

  componentDidMount() {
    // Keep a reference to the `interval`
    this.interval = setInterval(() => {
      this.setState({show: !this.state.show});
    }, 500);
  }

  componentWillUnmount() {
    // Clear the `interval`
    clearInterval(this.interval);
  }

  render() {
    return <Text>{this.state.show ? 'Hello, world' : ''}</Text>;
  }
}
```

Learn more

For more details about React, check the official docs, they have a good step-by-step guide there.

Example component

Here's a simple example of a hello world component that you can copy-paste into Expo:

```
import React, { Component } from 'react';
import { Text, View, StyleSheet } from 'react-native';
import { Constants } from 'expo';

export default class App extends Component {
  render() {
    return (
      <View style={styles.container}>
        <Text style={styles.paragraph}>Hello world</Text>
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    alignItems: 'center',
    justifyContent: 'center',
    paddingTop: Constants.statusBarHeight,
    backgroundColor: '#ecf0f1',
  },

  paragraph: {
    margin: 24,
    fontSize: 32,
    fontWeight: 'bold',
    textAlign: 'center',
    color: '#34495e',
  },
});
```

Example state component

Here's an example of a component with dynamic state that you can copy-paste into Expo:

```
import React, { Component } from 'react';
import { Text, View, StyleSheet } from 'react-native';
import { Constants } from 'expo';

class Countdown extends Component {
  constructor(props) {
    super(props);
    this.state = { count: props.count };
  }

  componentDidMount() {
    this.interval = setInterval(() => {
      this.setState(
        prevState =>
          prevState.count > 0 ? { count: prevState.count - 1 } : prevState,
      );
    }, 1000);
  }

  componentWillUnmount() {
    clearInterval(this.interval);
  }

  render() {
    return (
      <Text style={{ fontSize: 100, color: 'blue', ...this.props.style }}>
        {this.state.count}
      </Text>
    );
  }
}

export default class App extends Component {
  constructor(props) {
    super(props);
    this.state = { toggled: false };
  }

  componentDidMount() {
    setInterval(() => {
      this.setState(prev => ({ toggled: !prev.toggled }));
    }, 3000);
  }

  render() {
    return (
      <View style={styles.container}>
        <Text style={styles.paragraph}>Hello world</Text>
        <Countdown
          count={30}
          style={
            this.state.toggled
              ? { backgroundColor: 'red' }
              : { backgroundColor: 'yellow' }
          }
        />
      </View>
    );
  }
}
```

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    alignItems: 'center',
    justifyContent: 'center',
    paddingTop: Constants.statusBarHeight,
    backgroundColor: '#ecf0f1',
  },

  paragraph: {
    margin: 24,
    fontSize: 32,
    fontWeight: 'bold',
    textAlign: 'center',
    color: '#34495e',
  },
});
```

`<View>`

React Native comes with a dozen built-in components that you can easily import.

The `View` component is the most important one in React Native, it essentially represents a *portion of the screen*, and typically all other components need to use View. It is comparable to `<div>` on the web.

View has several props, but most of them are rarely used, except one: `style`. We will learn more about the `style` prop soon.

Read the API docs for View here

`<Text>`

The other very common component, besides `View`, is `Text`, which you can think of as similar to `<span>` on the web.

However, unlike the web, textual content can only be inside `Text`, not inside `View`.

```
// This works :)
<Text>Hello world</Text>
```

```
// This does NOT work :(
<View>Hello world</View>
```

Nesting

Text components can be nested, with the special exception that **View inside Text** does not work on Android (but it works on iOS).

```
// Works everywhere
class TextInANest extends Component {
  render() {
    return (
      <Text>
        <Text>Works</Text> everywhere
      </Text>
    );
  }
}
```

```
// Works ONLY on iOS
class BlueIsCool extends Component {
  render() {
    return (
      <Text>
        There is a blue square
        <View style={{width: 50, height: 50, backgroundColor: 'steelblue'}} />
        in between my text.
      </Text>
    );
  }
}
```

Props

Unlike on the web, this Text component has props specific to React Native and mobile apps, to name a few:

- `selectable: boolean`: allows or forbids selecting the textual content (e.g. for copying it to the clipboard)
- `numberOfLines` and `ellipsizeMode`: allow putting a maximum limit on the number of lines for the textual content, and how to display an ellipsis (`...`) when the limit is passed

Read the API docs for Text here

Example Text

Here's an example of nested Text components that you can copy-paste into Expo:

```
import React, { Component } from 'react';
import { Text, View, StyleSheet } from 'react-native';
import { Constants } from 'expo';

export default class App extends Component {
  render() {
    return (
      <View style={styles.container}>
        <Text
          style={styles.paragraph}
          numberOfLines={1}
          ellipsizeMode={'middle'}
        >
          Hello <Text style={styles.huge}>crazy weird</Text> world
        </Text>
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    alignItems: 'center',
    justifyContent: 'center',
    paddingTop: Constants.statusBarHeight,
    backgroundColor: '#ecf0f1',
  },

  paragraph: {
    fontSize: 32,
    fontWeight: 'bold',
    textAlign: 'center',
    color: '#34495e',
  },

  huge: {
    margin: 24,
    fontSize: 50,
    fontWeight: 'bold',
    color: '#ff0000',
  },
});
```

Styles

Most React Native components have a `style` prop, that allows you to pass an object with properties that customize how the component should look like. For example, here's how you can make a blue square in React Native.

```
class BlueSquare extends Component {
  render() {
    return <View style={{width: 50, height: 50, backgroundColor: 'blue'}}></View>;
  }
}
```

The properties of the style object are almost always similar to CSS, with the exception that React Native uses camelCase instead of snake-case. For example:

- `background-color` (CSS) becomes `backgroundColor` (React Native)
- `font-size` (CSS) becomes `fontSize` (React Native)

The allowed values are strings and numbers.

See the style properties allowed by `View` and the style properties allows by `Text`. Be careful not to mix View styles with Text styles, for example, `fontSize` on a View does not work, it works only on Text components.

Dimensions

Unlike CSS, React Native does not support `width: 50px` nor `width: 50%`. Whenever you need to specify a dimension, you just pass a number, and it will represent a "density-independent pixel".

For most purposes, you don't need to worry about `px` or `%` or `em`, just use a number and it will look approximately the same size on most devices. If you really need to have precise control over the actual pixel dimensions, use the APIs PixelRatio and Dimensions.

Stylesheets

The `style` prop can be given an object, like below:

```
class BlueSquare extends Component {
  render() {
    return <View style={{width: 50, height: 50, backgroundColor: 'blue'}}></View>;
  }
}
```

But React Native provides a nicer way, specially when there are many style properties, and that's the `Stylesheet` API. It allows you to create reusable styles that are named, for example:

```
import {Component} from 'react';
import {Stylesheet, View} from 'react-native';

const styles = StyleSheet.create({
  container: {
    width: 50,
    height: 50,
    backgroundColor: 'blue',
  }
});
```

```
class BlueSquare extends Component {
  render() {
    return <View style={styles.container}></View>;
  }
}
```

Learn more

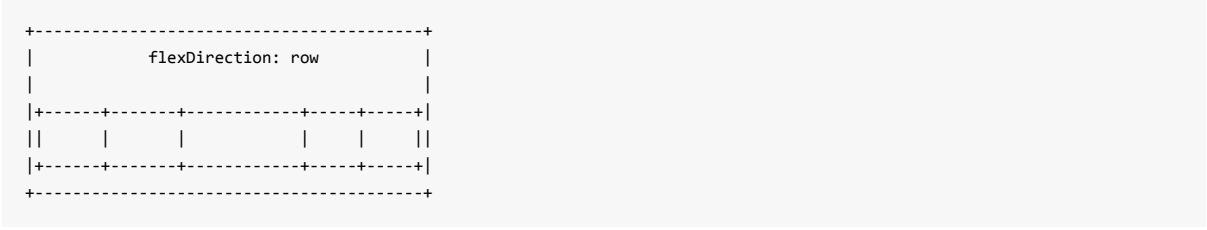
Learn more about Styles, dimensions, Flexbox, View style props, and Text style props in the official documentation.

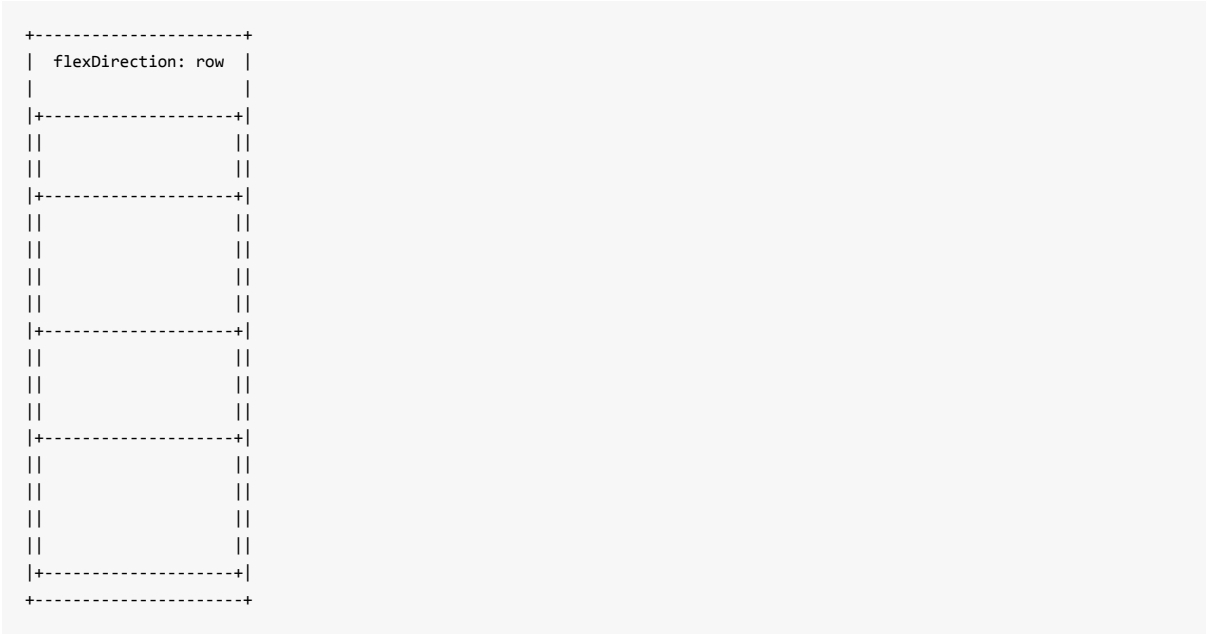
Flexbox

React Native's style system looks like CSS, but it has its differences. To begin with, there are no `display: inline`, no `display: inline-block`, no `width: 100%`, etc. Instead, React Native supports Flexbox, which turns out is enough to accomplish most layouts you wish, including 100% width and so forth. You can assume that by all components have `display: flex` by default, and in fact there is only `display: flex` and `display: none`.

flexDirection

With Flexbox, you need to first define the **direction** of a container: `row` (lay out children horizontally) or `column` (lay out children vertically). This direction is known as the "primary axis" of the Flexbox.

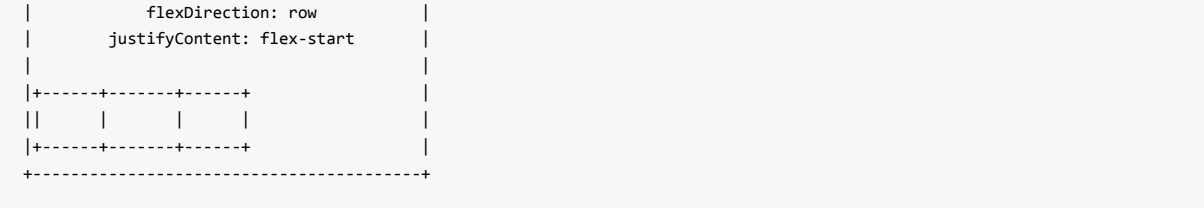


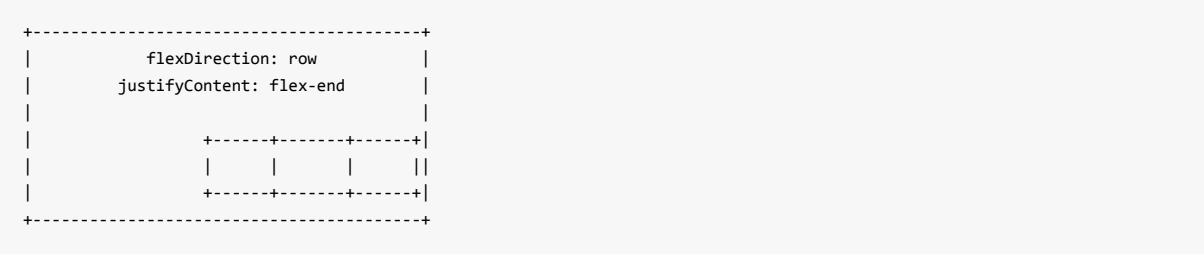


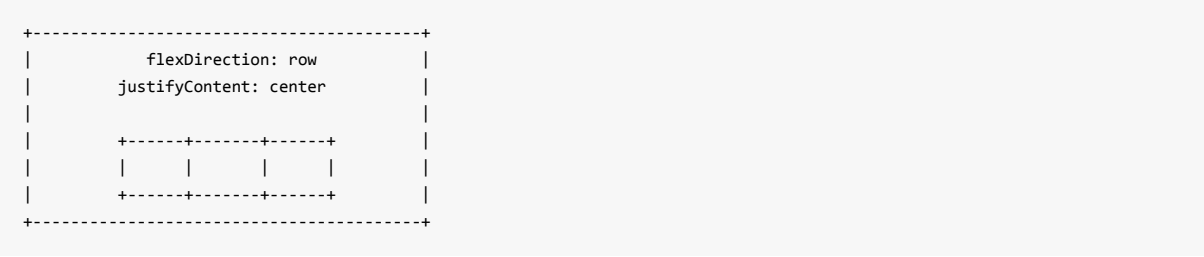
justifyContent

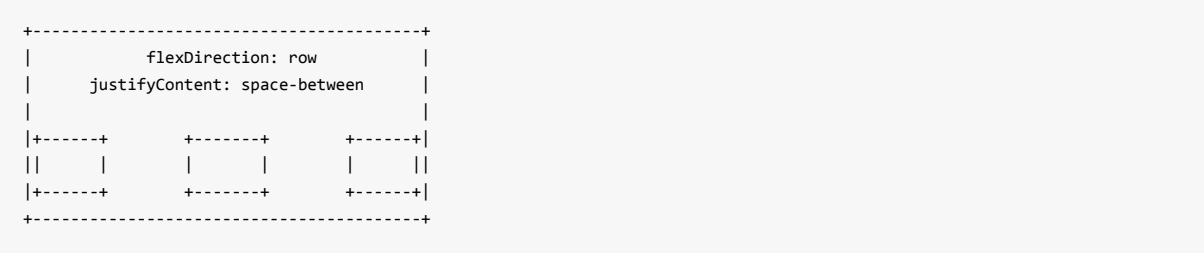
To change the positioning of the children along the flex direction (the primary axis), use justifyContent, which supports the values:

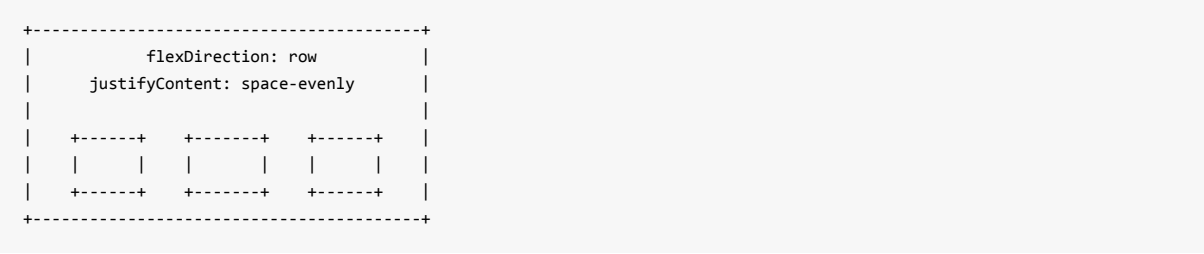
- `'flex-start'`: all children near the beginning of the parent
- `'center'`: all children grouped together in the center of the parent
- `'flex-end'`: all children near the end of the parent
- `'space-around'`, `'space-between'`, `'space-evenly'`: gradually distribute the children along the primary axis of the parent

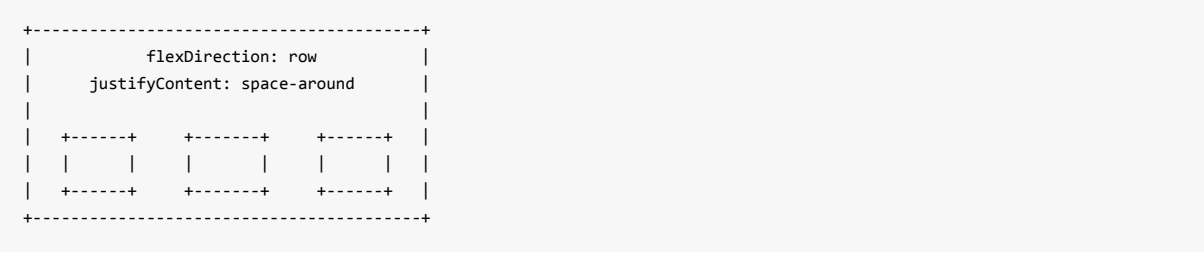












alignItems

To control the positioning of the *other* axis perpendicular to the primary axis, use alignItems, which supports the values:

- `'flex-start'` : all children near the beginning of the secondary axis of the parent

- `'center'` : all children near the center of the secondary axis of the parent
- `'flex-end'` : all children near the end of the secondary axis of the parent
- `'stretch'` : **instead of positioning the children on the secondary axis, *stretch* their size to fit the total secondary axis size**

```
+----------------------------------------+
|            flexDirection: row          |
|        justifyContent: flex-start      |
|          alignItems: flex-start        |
|+------+-------+------+                 |
||      |       |      |                 |
|+------+-------+------+                 |
|                                        |
|                                        |
|                                        |
|                                        |
+----------------------------------------+
```

```
+----------------------------------------+
|            flexDirection: row          |
|        justifyContent: flex-start      |
|            alignItems: center          |
|                                        |
|                                        |
|+------+-------+------+                 |
||      |       |      |                 |
|+------+-------+------+                 |
|                                        |
|                                        |
+----------------------------------------+
```

```
+----------------------------------------+
|            flexDirection: row          |
|        justifyContent: flex-start      |
|           alignItems: flex-end         |
|                                        |
|                                        |
|                                        |
|                                        |
|+------+-------+------+                 |
||      |       |      |                 |
|+------+-------+------+                 |
+----------------------------------------+
```

```
+----------------------------------------+
|            flexDirection: row          |
|        justifyContent: flex-start      |
|            alignItems: stretch         |
|+------+-------+------+                 |
||      |       |      |                 |
||      |       |      |                 |
||      |       |      |                 |
||      |       |      |                 |
||      |       |      |                 |
|+------+-------+------+                 |
+----------------------------------------+
```

Tips

width 100%: if you want to have the children essentially do `width: 100%`, then you probably need to use `alignItems: 'stretch'`, which in the case of width means you need to set these styles on the parent:

- `flexDirection: 'column'`
- `alignItems: 'stretch'`
- `justifyContent` doesn't make a difference

height 100%: you need to set these styles on the parent:

- `flexDirection: 'row'`
- `alignItems: 'stretch'`
- `justifyContent` doesn't make a difference

Fill the remaining space: if some children have fixed sizes but one child should fill all the remaining space, then you can use `flex: 1` which stretches that child along the **primary axis**, see diagram below:

```
+----------------------------------------+
|            flexDirection: row          |
|+------+------------------------+------+|
||      |         flex: 1        |      ||
|+------+------------------------+------+|
+----------------------------------------+
```

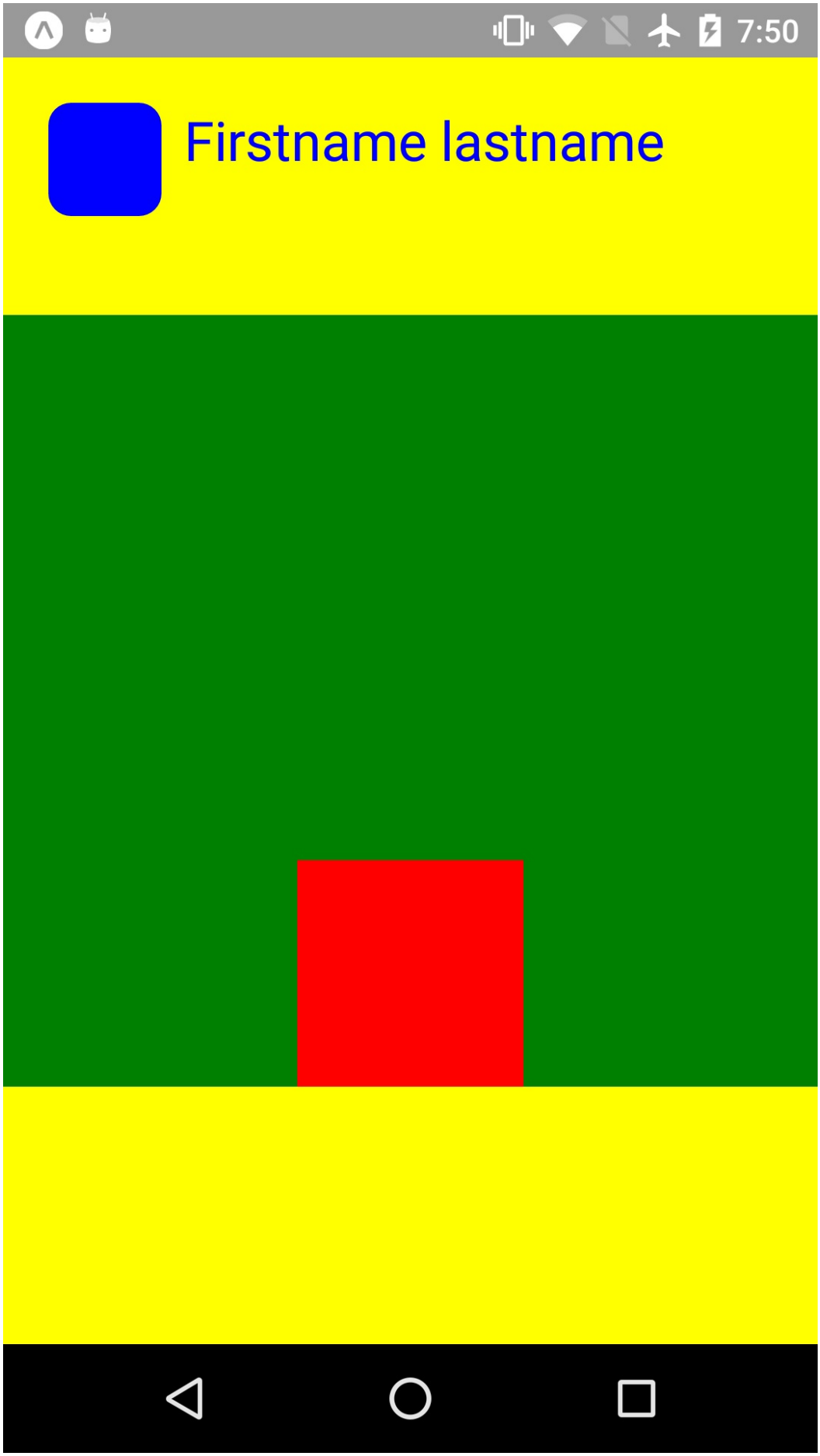
Example Flexbox

Here's an example of using Flexbox layout that you can copy-paste into Expo:

```
import React, { Component } from 'react';
import { View, StyleSheet } from 'react-native';
import { Constants } from 'expo';

export default class App extends Component {
  render() {
    return (
      <View style={styles.container}>
        <View style={styles.square1} />
        <View style={styles.square2} />
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: 'row',
    alignItems: 'center',
    justifyContent: 'center',
    paddingTop: Constants.statusBarHeight,
    backgroundColor: 'white',
  },

  square1: {
    backgroundColor: 'red',
    flex: 1,
    alignSelf: 'stretch',
  },

  square2: {
    backgroundColor: 'blue',
    flex: 1,
    height: 400,
  },
});
```

Exercise

In this exercise, use Flexbox and style props to create the following UI in Expo:


7:50
Firstname lastname

Solution

Here's the solution to the exercise, but try to solve it on your own before looking at this!

```
import React, { Component } from 'react';
import { View, Text, StyleSheet } from 'react-native';
import { Constants } from 'expo';

export default class App extends Component {
  render() {
    return (
      <View style={styles.container}>
        <View style={styles.header}>
          <View style={styles.avatar} />
          <Text style={styles.name}>Firstname lastname</Text>
        </View>
        <View style={styles.body}>
          <View style={styles.square} />
        </View>
        <View style={styles.footer} />
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: 'column',
    alignItems: 'center',
    justifyContent: 'center',
    paddingTop: Constants.statusBarHeight,
    backgroundColor: 'white',
  },

  header: {
    backgroundColor: 'yellow',
    flex: 1,
    flexDirection: 'row',
    alignSelf: 'stretch',
  },

  avatar: {
    backgroundColor: 'blue',
    marginTop: 20,
    marginLeft: 20,
    borderRadius: 10,
    width: 50,
    height: 50,
  },

  name: {
    marginTop: 20,
    marginLeft: 10,
    fontSize: 24,
    color: 'blue',
  },

  body: {
    backgroundColor: 'green',
    flex: 3,
    alignSelf: 'stretch',
    alignItems: 'center',
    justifyContent: 'flex-end',
  },
```

```
  square: {
    backgroundColor: 'red',
    width: 100,
    height: 100,
  },

  footer: {
    backgroundColor: 'yellow',
    flex: 1,
    flexDirection: 'row',
    alignSelf: 'stretch',
  },
});
```

Tuesday B

In this session, we are going to how to handle user interaction, how to display images, how to fetch data from the network, and display lists of components on the screen.

TextInput

Where the web has the element `<input type="text">`, React Native has the `<TextInput>` component, which allows the user to enter text into the component using the keyboard. Although it has the same purpose as `<input type="text">`, it has different props, methods, and event handlers.

Value prop

The most important prop is the `value`, allowing you to **set** the textual content of this component. In the example below, the TextInput will **always** have the contents `"Hello"`, even if the user tries to change it with their keyboard.

```
// The TextInput will always be "Hello"!
<TextInput value={"Hello"} />
```

This happens because the TextInput is a controlled React component, where the `value` prop has absolute control over what the user will see on the screen.

So, instead, we need to handle the user's keyboard events to update the state (with `setState`) and send the state into the `value` prop.

```
<TextInput
  onChangeText={(text) => this.setState({text})}
  value={this.state.text}
  />
```

Event handlers

The commonly used event handler props on the TextInput are:

- onChangeText: triggered when the user attempts to change the contents
- onFocus: triggered when the component gains focus
- onBlur: triggered when the component loses focus
- onSubmitEditing: triggered when the user presses a special "submit" button on their keyboard

Other props

Useful and common props on the TextInput include:

- autoFocus: if true, the TextInput will gain focus as soon as it is mounted
- multiline: allows the TextInput wrap the textual content to many lines
- placeholder: a string to display while there is no actual text content

Of course, this above is just a short list of the essentials. For more, read the API docs for View here

More form components

React Native comes with more components for making forms, such as Switch (a "checkbox"), Slider, and Picker (a "dropdown list").

Example TextInput

Here's an example of using TextInput to display the user's keypresses in uppercase, which you can copy-paste into Expo.

```
import React, { Component } from 'react';
import { View, Text, TextInput, StyleSheet } from 'react-native';
import { Constants } from 'expo';

export default class App extends Component {
  constructor(props) {
    super(props);
    this.state = { text: '' };
  }

  onChangeText(text) {
    this.setState(() => ({ text: text.toUpperCase() }));
  }

  render() {
    return (
      <View style={styles.container}>
        <TextInput
          onChangeText={text => this.onChangeText(text)}
          style={styles.input}
          placeholder={'Enter text here'}
        />
        <Text style={styles.text}>{this.state.text}</Text>
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: 'column',
    alignItems: 'center',
    justifyContent: 'flex-start',
    margin: 10,
    paddingTop: Constants.statusBarHeight,
    backgroundColor: 'white',
  },

  input: {
    height: 40,
    alignSelf: 'stretch',
    fontSize: 20,
  },

  text: {
    marginTop: 20,
    fontSize: 24,
  },
});
```

Touchables

The other kind of common user interaction in mobile apps, besides text input, is handling presses. React Native makes it possible to make any component handle user touches, using `Touchable*` components. They come in different flavors:

- TouchableOpacity
- TouchableHighlight
- TouchableNativeFeedback
- TouchableWithoutFeedback

But they all work the same way. These are **wrapper** components. They are not useful by themselves but they exist so you can put them around some existing component, for instance any `<View>`.

```
<TouchableOpacity onPress={() => alert('View was pressed!')}>
  <View style={styles.blueSquare} />
</TouchableOpacity>
```

All these Touchable variants have the `onPress` prop. The variants differ by how they look like when the user touches them:

- TouchableOpacity: when pressed, will make the child component slightly more transparent
- TouchableHighlight: like TouchableOpacity, but has an underlay color
- TouchableNativeFeedback (only on Android): when pressed, displays a ripple effect
- TouchableWithoutFeedback: when pressed, looks the same as when not pressed

Button

With Touchables, you can create your own button with a custom look-and-feel. However, if you want to just make a button that looks like standard iOS buttons or standard Android buttons, use the simple `<Button>` component, that has props:

- onPress: the event handler function
- title: the textual content in the button
- color: (on Android:) the background color for the button, or (on iOS:) the color of the text

Read the API docs for Button here

Example Button

Here's an example of using Button to change component state, which you can copy-paste into Expo.

```
import React, { Component } from 'react';
import { View, Text, Button, StyleSheet } from 'react-native';
import { Constants } from 'expo';

export default class App extends Component {
  constructor(props) {
    super(props);
    this.state = { text: '' };
  }

  onPressButton() {
    this.setState(() => ({ text: 'Hello!' }));
  }

  render() {
    return (
      <View style={styles.container}>
        <Button onPress={() => this.onPressButton()} title={'Press me'} />
        <Text style={styles.text}>{this.state.text}</Text>
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: 'column',
    alignItems: 'center',
    justifyContent: 'flex-start',
    margin: 10,
    paddingTop: Constants.statusBarHeight,
    backgroundColor: 'white',
  },

  text: {
    marginTop: 20,
    fontSize: 24,
  },
});
```

Example TouchableOpacity

Here's an example of using TouchableOpacity, similar to the previous Button example, which you can copy-paste into Expo.

```
import React, { Component } from 'react';
import { View, Text, TouchableOpacity, StyleSheet } from 'react-native';
import { Constants } from 'expo';

export default class App extends Component {
  constructor(props) {
    super(props);
    this.state = { text: '' };
  }

  onPressButton() {
    this.setState(() => ({ text: 'Hello!' }));
  }

  render() {
    return (
      <View style={styles.container}>
        <TouchableOpacity onPress={() => this.onPressButton()}>
          <View style={styles.square} />
        </TouchableOpacity>
        <Text style={styles.text}>{this.state.text}</Text>
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: 'column',
    alignItems: 'center',
    justifyContent: 'flex-start',
    margin: 10,
    paddingTop: Constants.statusBarHeight,
    backgroundColor: 'white',
  },

  square: {
    backgroundColor: '#0077ff',
    width: 100,
    height: 100,
    borderRadius: 20,
  },

  text: {
    marginTop: 20,
    fontSize: 24,
  },
});
```

Exercise

In this exercise, the goal is to display:

- A text input field
- A button: "Submit"
- A text label

Where the submit button will add the contents of the text input field to an array, and the text label will show all the words submitted so far. Every time the submit button is pressed, the text input is also cleared.

See this short video to understand the expected outcome.

Solution

Here's the solution to the exercise, but try to solve it on your own before looking at this!

```
import React, { Component } from 'react';
import { View, Text, TextInput, Button, StyleSheet } from 'react-native';
import { Constants } from 'expo';

export default class App extends Component {
  constructor(props) {
    super(props);
    this.state = { words: [] };
    this.inputRef = React.createRef();
    this.inputText = '';
  }

  onPressButton() {
    this.inputRef.value.clear();
    this.setState(prev => ({ words: prev.words.concat(this.inputText) }));
  }

  render() {
    return (
      <View style={styles.container}>
        <View style={styles.row}>
          <TextInput
            ref={this.inputRef}
            onChangeText={text => {
              this.inputText = text;
            }}
            style={styles.input}
            placeholder={'Enter word here'}
          />
          <Button title={'Submit'} onPress={this.onPressButton.bind(this)} />
        </View>
        <Text style={styles.text}>{this.state.words.join(' ')}</Text>
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: 'column',
    alignItems: 'center',
    justifyContent: 'flex-start',
    margin: 10,
    paddingTop: Constants.statusBarHeight,
    backgroundColor: 'white',
  },

  row: {
    flexDirection: 'row',
    alignSelf: 'stretch',
  },

  input: {
    flex: 1,
    height: 40,
    alignSelf: 'stretch',
    fontSize: 20,
  },

  text: {
```

```
    marginTop: 20,
    fontSize: 24,
  },
});
```

Image

Where the web has the element `<img src="cat.jpeg">`, React Native has the `<Image>` component. There are several differences between these two, worth highlighting.

The most visible difference is that in React Native the prop is `source`, not `src`. And the value for this prop is not a URL, it's usually an object:

```
<Image
  style={{width: 300, height: 200}}
  source={{uri: 'https://placekitten.com/408/287'}}
  />
```

Note the difference:

- `src="https://placekitten.com/408/287"` (HTML)
- `source={ {uri: 'https://placekitten.com/408/287'} }` (React Native)

Always fixed size

Unlike on the web, images in React Native **do not resize to fit the image content** after it is fetched. You must specify the size of the image before it is fetched. You can either do this with fixed numbers for `width` and `height`:

```
<Image
  style={{width: 300, height: 200}}
  source={{uri: 'https://placekitten.com/408/287'}}
  />
```

Or one of these can stretch to 100%, using Flexbox `alignSelf`:

```
// Suppose the parent component has `flexDirection: 'column'`
<Image
  style={{alignSelf: 'stretch', height: 200}}
  source={{uri: 'https://placekitten.com/408/287'}}
  />
```

Or:

```
// Suppose the parent component has `flexDirection: 'row'`
<Image
  style={{width: 300, alignSelf: 'stretch'}}
  source={{uri: 'https://placekitten.com/408/287'}}
  />
```

Local images

Besides images from the internet, you can also display local images from your project's folder, just "require" the file and pass it as the `source` prop:

```
<Image source={require('./my-icon.png')} />
```

Read more

Read a guide to using images here.

Read the API docs for the Image component here.

Example remote image

Here's an example of display an image from the internet using the Image component state, which you can copy-paste into Expo.

```
import React, { Component } from 'react';
import { View, Image, StyleSheet } from 'react-native';
import { Constants } from 'expo';

export default class App extends Component {
  constructor(props) {
    super(props);
  }

  render() {
    return (
      <View style={styles.container}>
        <Image
          source={{
            uri:
              'http://i890.photobucket.com/albums/ac107/lebert32/LuLu059.jpg',
          }}
          style={{ width: 300, height: 220 }}
        />
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: 'column',
    alignItems: 'center',
    justifyContent: 'flex-start',
    margin: 10,
    paddingTop: Constants.statusBarHeight,
    backgroundColor: 'white',
  },
});
```

Example local image

Here's an example of display a local image from your project using the Image component state, which you can copy-paste into Expo.

```
import React, { Component } from 'react';
import { View, Image, StyleSheet } from 'react-native';
import { Constants } from 'expo';
import asset from './assets/expo.symbol.white.png';

export default class App extends Component {
  constructor(props) {
    super(props);
  }

  render() {
    return (
      <View style={styles.container}>
        <Image source={asset} />
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: 'column',
    alignItems: 'center',
    justifyContent: 'flex-start',
    paddingTop: Constants.statusBarHeight,
    backgroundColor: 'blue',
  },
});
```

Networking

The JavaScript environment that runs in React Native is not like the browser, it's not like Node.js, but it has a little bit of both. One of the gladly convenient APIs available is `fetch`, which allows you to make network requests to servers on the internet.

Fetch

Just like in the browser, you can call `fetch(url)` and it will return a promise of the response, which you may have to chain with `.then()` call to pick either its textual data or its JSON data. For instance, for textual response:

```
fetch('https://example.com/name')
  .then(res => res.text())
  .then(text => alert('This is the response text: ' + text));
```

Or for JSON responses:

```
fetch('https://example.com/data.json')
  .then(res => res.json())
  .then(json => alert(json.field));
```

To practice with `fetch`, you can use the server jsonplaceholder.

Read more

Read more about networking APIs in React Native here

Example fetch

Here's an example of using `fetch` to get data from a server and display it in the app using `setState`, which you can copy-paste into Expo.

```
import React, { Component } from 'react';
import { View, Text, StyleSheet } from 'react-native';
import { Constants } from 'expo';

export default class App extends Component {
  constructor(props) {
    super(props);
    this.state = {};
  }

  componentDidMount() {
    fetch('https://jsonplaceholder.typicode.com/users/1')
      .then(res => res.json())
      .then(json => {
        this.setState(json);
      });
  }

  render() {
    return (
      <View style={styles.container}>
        {this.state.name ? <Text>{this.state.name}</Text> : null}
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: 'column',
    alignItems: 'center',
    justifyContent: 'flex-start',
    paddingTop: Constants.statusBarHeight,
    backgroundColor: 'white',
  },
});
```

ScrollView

Sometimes your components don't fit all in the same screen, and they may overflow the max height. On the web, this would automatically create the possibility to scroll. On React Native, we need to do this ourselves, by wrapping our components with a `<ScrollView>` parent.

```
<ScrollView>
  <Text>Lots and lots</Text>
  <Text>and lots and lots</Text>
  <Text>and lots and lots</Text>
  <Text>and lots and lots</Text>
  <Text>and lots and lots</Text>
  <Text>and lots and lots</Text>
  <Text>and lots and lots</Text>
  <Text>and lots and lots</Text>
  <Text>and lots and lots</Text>
  <Text>and lots and lots</Text>
  <Text>and lots and lots</Text>
  <Text>and lots and lots</Text>
  <Text>and lots and lots</Text>
  <Text>and lots and lots</Text>
  <Text>and lots and lots</Text>
  <Text>of text that goes beyond</Text>
  <Text>the maximum height</Text>
  <Text>of the phone's screen</Text>
</ScrollView>
```

That said, unlike on the web, ScrollView comes with many props to customize the scrolling experience: does it snap to a grid or is it fluid? Does it bounce when it reach the ends? Does the keyboard hide when we are scrolling? How does the deceleration feel like to the user? On mobile, scrolling is one of the core user experiences, so ScrollView allows controlling this user experience in details.

Read more

Read the API docs for ScrollView here

Example ScrollView

Here's an example that fetches many users' data from a server, and displays each user with a `<User>` component. Since there are multiple users, we use a ScrollView to allow scrolling through all of them.

In this example there are **two files**, `App.js` and `User.js`. You can copy-paste both of them as two different files into Expo.

File `App.js`:

```
import React, { Component } from 'react';
import { ScrollView, View, StyleSheet } from 'react-native';
import { Constants } from 'expo';
import User from './User';

export default class App extends Component {
  constructor(props) {
    super(props);
    this.state = { users: [] };
  }

  componentDidMount() {
    fetch('https://jsonplaceholder.typicode.com/users')
      .then(res => res.json())
      .then(users => {
        this.setState({ users });
      });
  }

  render() {
    return (
      <View style={styles.container}>
        <ScrollView>
          {this.state.users.map(user => <User data={user} />)}
        </ScrollView>
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    paddingTop: Constants.statusBarHeight,
    backgroundColor: 'white',
  },
});
```

File `User.js`:

```
import React, { Component } from 'react';
import { View, Text, StyleSheet } from 'react-native';

export default class User extends Component {
  render() {
    const { username, email, name, website } = this.props.data;

    return (
      <View style={styles.container}>
        <Text style={styles.name} numberOfLines={1}>
          {username} <Text style={styles.email}>{email}</Text>
        </Text>
        <Text style={styles.detail}>{name}</Text>
        <Text style={styles.detail}>
```

```
          {'Website: '}
          {website}
        </Text>
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: 'column',
    padding: 10,
    borderBottomWidth: 1,
    borderBottomColor: '#cccccc',
    alignSelf: 'stretch',
    alignItems: 'flex-start',
    justifyContent: 'flex-start',
  },

  name: {
    fontSize: 18,
  },

  email: {
    fontFamily: 'monospace',
    fontSize: 16,
    color: '#999999',
  },

  detail: {
    fontSize: 14,
  },
});
```

FlatList

One of the most common user experiences on mobile is "infinite scrolling" of a very large amount of data. It is unwise to download and display all of that data at once, because it might be too much for the phone to handle and compute.

So instead, we need a way of gradually fetching and gradually displaying new content, dynamically as the user is scrolling. This is a very challenging problem to solve, but gladly there is a built-in React Native component dedicated to just that, called **FlatList**.

renderItem

While ScrollView allows you to insert many children, FlatList does not have any children. Instead, you pass it an array of data in the prop called `data`, and then a prop called `renderItem` takes a function that knows how render a single item in that list. For this reason, FlatList is suitable when all the items in the list are of the same type.

```
<FlatList
  data={['Alice', 'Bob', 'Carla', 'David', 'Eve', 'Fred', 'Gloria']}
  renderItem={({ item }) => <Text>item</Text>}
  />
```

Note that the prop `renderItem` needs a function. This function should look like this:

- Input: an object `{item, index, ...etc}` which is provided to you
- Output: a React element, e.g. `<Text>` or `<View>` or `<MyComponent>`

keyExtractor

Because FlatList doesn't show all the items in the `data` array at the same time, and because the user is scrolling (maybe even scrolling very fast!), FlatList needs to have a way of detecting the true identity of each item. For this reason, you may have to provide the `keyExtractor` prop with a function:

```
<FlatList
  data={[{id: 28, name: 'Alice'}, {id: 12, name: 'Bob'}]}
  keyExtractor={(item, index) => item.id}
  renderItem={({ item }) => <Text>item.name</Text>}
  />
```

Other props

FlatList is actually a special type of ScrollView, so it supports all the props that ScrollView supports, but more! It allows advanced customization, such as:

- ListHeaderComponent: customize the header of the FlatList
- ListFooterComponent: customize the footer of the FlatList
- ItemSeparatorComponent: specify a component that is rendered between every two items
- onEndReached: event handler that is triggered when the user has scrolled to the bottom, this is usually a good place where to request more data and increase the size of the array

Read more

Read the API docs for FlatList here

Example FlatList

Here's an example of a FlatList that displays all the users from the server, which you can copy-paste into Expo. Note there are two files.

File `App.js` :

```
import React, { Component } from 'react';
import { FlatList, View, StyleSheet } from 'react-native';
import { Constants } from 'expo';
import User from './User';

export default class App extends Component {
  constructor(props) {
    super(props);
    this.state = { users: [] };
  }

  componentDidMount() {
    fetch('https://jsonplaceholder.typicode.com/users')
      .then(res => res.json())
      .then(users => {
        this.setState({ users });
      });
  }

  render() {
    return (
      <View style={styles.container}>
        <FlatList
          data={this.state.users}
          renderItem={({ item }) => <User data={item} />}
        />
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    paddingTop: Constants.statusBarHeight,
    backgroundColor: 'white',
  },
});
```

File `User.js` :

```
import React, { Component } from 'react';
import { View, Text, StyleSheet } from 'react-native';

export default class User extends Component {
  render() {
    const { username, email, name, website } = this.props.data;

    return (
      <View style={styles.container}>
        <Text style={styles.name} numberOfLines={1}>
          {username} <Text style={styles.email}>{email}</Text>
        </Text>
        <Text style={styles.detail}>{name}</Text>
        <Text style={styles.detail}>
          {'Website: '}
```

```
          {website}
        </Text>
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: 'column',
    padding: 10,
    borderBottomWidth: 1,
    borderBottomColor: '#cccccc',
    alignSelf: 'stretch',
    alignItems: 'flex-start',
    justifyContent: 'flex-start',
  },

  name: {
    fontSize: 18,
  },

  email: {
    fontFamily: 'monospace',
    fontSize: 16,
    color: '#999999',
  },

  detail: {
    fontSize: 14,
  },
});
```

Exercise

In this exercise, the goal is to display an **infinite scrolling list of images** from a server with images, for instance:

- https://jsonplaceholder.typicode.com/photos

See this short video to understand the expected outcome.

Solution

Here's the solution to the exercise, but try to solve it on your own before looking at this!

```
import React, { Component } from 'react';
import { FlatList, View, Image, StyleSheet } from 'react-native';
import { Constants } from 'expo';

export default class App extends Component {
  constructor(props) {
    super(props);
    this.state = { photos: [] };
  }

  componentDidMount() {
    fetch('https://jsonplaceholder.typicode.com/photos')
      .then(res => res.json())
      .then(photos => {
        this.setState({ photos });
      });
  }

  render() {
    return (
      <View style={styles.container}>
        <FlatList
          data={this.state.photos}
          renderItem={({ item }) => (
            <Image
              source={{ uri: item.url }}
              style={{ width: 200, height: 200 }}
            />
          )}
        />
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    paddingTop: Constants.statusBarHeight,
    backgroundColor: 'white',
  },
});
```

Wednesday A

In this session we are going to drop Expo, and instead, compile and run a dedicated app (with its own app name) for our project. This will require us to use Xcode or Android Studio (just a bit), but most of the time we will still be using JavaScript and React Native. In this process, we will learn about common errors, how to debug, and some troubleshooting tips.

react-native init

To create a local React Native project in your computer, we're going to use a command-line interface called `react-native`, you can install it with:

```
npm install --global react-native-cli
```

Then, `cd` into a folder of your choice and create your project:

```
react-native init MyProject
```

(You can choose any name to replace `MyProject`, but keep it one word, or camel case)

This will begin installing some libraries that React Native needs, and it will create a template project with everything set up for us to begin coding. Once the `init` is done, go into the project folder with `cd MyProject`.

Project structure

The `react-native init` tool created many folders and files for us. This is the typical project structure for React Native apps, and we shouldn't change it too much (if we want to make it easy to update React Native libraries in the future). The folder structure looks like this:

```
.
├── android
│   ├── app
│   ├── build.gradle
│   ├── gradle
│   ├── gradle.properties
│   ├── gradlew
│   ├── gradlew.bat
│   ├── keystores
│   └── settings.gradle
├── App.js
├── app.json
├── index.js
├── ios
│   ├── MyProject
│   ├── MyProjectTests
│   ├── MyProject-tvOS
│   ├── MyProject-tvOSTests
│   └── MyProject.xcodeproj
├── node_modules
│   └── ...
├── package.json
└── yarn.lock
```

There are a few important parts there.

- `android` : contains Java files and Gradle configs used by Android Studio to compile the Android app
- `ios` : contains ObjectiveC files and Xcode configs used by Xcode to compile the iOS app
- `App.js` **: this is your app's main JavaScript file where you will code**
- `index.js` : this is just an entry JavaScript file, don't edit it unless you know what you're doing
- `package.json` : an npm configuration file where we list the dependencies

There are also many dot files, such as `.babelrc` , `.buckconfig` , `.flowconfig` , `.gitattributes` , `.gitignore` , `.watchmanconfig` . These are important for compiling the app, so keep them.

Run it

To test if everything is working, connect your phone to the computer with a USB cable, and run this command:

```
react-native run-android
```

or this command:

```
react-native run-ios
```

If it works, congratulations! We can start coding in `App.js` . If it doesn't work, there are many possible reasons. First it's important to learn about the bridge architecture. Once you know about the architecture, it becomes easier to troubleshoot.

Bridge architecture

The way React Native works is complex and mixes multiple technologies, but in the big picture it can be understood as a marionette. There is a JavaScript runtime where you JavaScript code will run, and it is controlling the native parts (Java in the Android case, or ObjectiveC in the iOS case).

When you are developing the project, your JavaScript code will be hosted by a server, called the "React Packager". You can think of this as the same type of software that Webpack. The server can run on your computer, which means your phone will need to communicate with your computer! The computer will run the JavaScript and the phone will run the Java/ObjectiveC. See the diagram below:

```
  +---------------------------+
  |       Your JS code        |
  +---------------------------+
              ^
              |
              v
  +---------------------------+  // This part can run:
  |      React Packager       |  // - On your computer (terminal)
  |      "Metro Bundler"      |
  |       server (8081)       |  // - OR: on your computer (browser)
  +---------------------------+  // - OR: on your phone (for production)
       ^               ^
       |               |
-------|----bridge-----|---------
       |               |
       v               v
  +---------+    +------------+
  |  Java   |    | ObjectiveC |
  | Android |    |    iOS     |
  +---------+    +------------+
```

If your app doesn't work (e.g. it shows a red screen), we will have to double check if all these parts are correctly setup and running.

Troubleshooting

"Could not connect to development server"

If the app opens with a red screen saying that it couldn't find the app bundle, then it could be for one of these reasons:

- The server is not running on your computer yet
- The phone is not in the same Wi-Fi network as the computer is
- The phone does not know the correct IP address for the computer

So first make sure that the React packager server is running on your computer. It usually runs on port 8081, so you need to check if it's running, e.g. run the command `lsof -i :8081`. If lsof returns nothing, then there is no packager server.

To run the packager server, run `npm start`, and you should see this in the terminal:

```
┌──────────────────────────────────────────────────────────────────────────┐
│                                                                          │
│  Running Metro Bundler on port 8081.                                     │
│                                                                          │
│  Keep Metro running while developing on any JS projects. Feel free to    │
│  close this tab and run your own Metro instance if you prefer.           │
│                                                                          │
│  https://github.com/facebook/react-native                                │
│                                                                          │
└──────────────────────────────────────────────────────────────────────────┘
```

Then, make sure both phone and computer are on the same network, and:

- Discover your computer's IP address with `ifconfig` (macOS) or `ip addr` (Linux)
- Shake your phone (really, shake it!) until a menu shows
- Choose "Dev Settings", then choose "Debug server host & port for device" and insert your computer's IP address and port, for example `192.168.1.115:8081`
- Go back, and re-shake your phone, then choose "Reload"

Read more

The official documentation also has more troubleshooting instructions if you need.

Debugging

To solve issues with your code, you may need to gain more insight into what's going on. The usual `console.log` won't quite work in React Native, but here are your options, in increasing order of sophistication:

- `alert('hello')` works for very simple cases
- `console.warn('hello')` will show a yellow overlay in the corner of the screen
- Use Chrome to debug
  - Shake the phone
  - Choose "Debug JS Remotely"
  - Chrome should open in your computer
  - Open the console and it will show every `console.log` call
  - Breakpoints also work: add `debugger;` in a JavaScript line, and it will make Chrome pause there

DevTools

If you want to inspect the layout and component tree on the screen, the React DevTools work well together with React Native.

Read more

The official documentation also has more debugging tips if you need.

Exercise

In this exercise, get your favorite Expo example or exercise from yesterday and bring it into your local React Native project. Make sure it compiles and builds correctly, and can be used correctly.

Start warming up for your demo idea, do you already know what you want to build? In the next session we will get some final skills to build the whole thing. If you have extra time, start building it already. Check the list of open APIs and see if you can easily make requests with `fetch`.

Wednesday B

In this session, we're going to learn about a few final important things to know about: using the local database, and creating multiple screens and navigating between them. This will equip us with the essentials to build our own real app, and get creative!

AsyncStorage

Besides controlling the graphical user interface using React components, React Native also comes with built-in APIs to access some of the phone's special capabilities, for instance Geolocation, Clipboard, ImageStore, Vibration, etc. We are only going to learn one of these APIs today, and that's AsyncStorage. But now you know that the other APIs exist, in case you need them.

AsyncStorage provides a simple local database which you can use to store session data, store "bookmarks", and in general any kind of small amount of offline-available data on the phone. You can use it to store information that is only interesting to the user, not to the rest of the world.

Set

AsyncStorage only supports simple key-value pairs, where both key and value are strings. So, you cannot store objects directly, you have to first convert objects to strings using `JSON.stringify()`. Some examples:

```
AsyncStorage.setItem('myname', 'Andre');
```

```
AsyncStorage.setItem('userdata', JSON.stringify({age: 25, name: 'Alice', id: 8175062}));
```

This `setItem` API actually returns a Promise, so if you want to wait for it to complete before doing another action, you must use:

```
AsyncStorage.setItem(key, value).then(() => {
  // now we are sure that the item was saved in the database
});
```

Or with async-await:

```
async function myfunction() {
  await AsyncStorage.setItem(key, value);
  // now we are sure that the item was saved in the database
}
```

Get

To read from the database, provide the key to `getItem` and it will return the a Promise of the value. Remember, the value is a string, so you might need to `JSON.parse` it.

```
AsyncStorage.getItem('userdata').then(value => {
  const data = JSON.parse(value);
  console.log(data); // {age: 25, name: 'Alice', id: 817062}
});
```

Or with async-await:

```
async function myfunction() {
  const data = await AsyncStorage.getItem('userdata');
  console.log(data); // {age: 25, name: 'Alice', id: 817062}
}
```

Read more

Read all about AsyncStorage in the official docs.

Example – AsyncStorage

This example is the same as the TextInput exercise from Tuesday with a submit button , except we use AsyncStorage to save the submitted data in the local database with AsyncStorage, so it persists after the app is closed.

```
import React, { Component } from 'react';
import {
  View,
  Text,
  TextInput,
  Button,
  AsyncStorage,
  StyleSheet,
} from 'react-native';

export default class App extends Component {
  constructor(props) {
    super(props);
    this.state = { words: [] };
    this.inputRef = React.createRef();
    this.inputText = '';
  }

  componentDidMount() {
    AsyncStorage.getItem('words').then(words => {
      if (words) {
        this.setState({ words: JSON.parse(words) });
      }
    });
  }

  onPressButton() {
    this.inputRef.value.clear();
    this.setState(prev => {
      const words = prev.words.concat(this.inputText);
      AsyncStorage.setItem('words', JSON.stringify(words));
      return { words };
    });
  }

  render() {
    return (
      <View style={styles.container}>
        <View style={styles.row}>
          <TextInput
            ref={this.inputRef}
            onChangeText={text => {
              this.inputText = text;
            }}
            style={styles.input}
            placeholder={'Enter word here'}
          />
          <Button title={'Submit'} onPress={this.onPressButton.bind(this)} />
        </View>
        <Text style={styles.text}>{this.state.words.join(' ')}</Text>
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: 'column',
```

```
    alignItems: 'center',
    justifyContent: 'flex-start',
    margin: 10,
    paddingTop: 24,
    backgroundColor: 'white',
  },

  row: {
    flexDirection: 'row',
    alignSelf: 'stretch',
  },

  input: {
    flex: 1,
    height: 40,
    alignSelf: 'stretch',
    fontSize: 20,
  },

  text: {
    marginTop: 20,
    fontSize: 24,
  },
});
```

React Navigation

Finally, the exciting last part of this course: navigation. Most apps have multiple screens linked to each other, and with "back" buttons when they get stacked. The bad news is that React Native *does not* come with built-in solutions for this, they actually built something, but it wasn't great. The good news is that the developer community ended up building great libraries for this problem, and we're going to learn the most common one: React Navigation.

Install

First, we're going to have to install this library:

```
yarn add react-navigation
```

or:

```
npm install react-navigation
```

Choose a navigator

There are different ways of handling multiple screens, but the most common are:

- Tabs: at the bottom the user can select which screen they want to see
- Stack: there is only one screen shown at a time, but if you go to another screen, it will be "stacked on top" of the previous one, and you'll see a back button

Let's choose Stack because it's the simplest for now. The library will handle the stacking logic, the back button logic, and will also display a header with title and back button with default styles.

Import `createStackNavigator` from `react-navigation`:

```
import {createStackNavigator} from 'react-navigation';
```

When we call this function with a configuration object, it will return a "App" component, and that's what we need to use instead of the typical `App` component.

```
const App = createStackNavigator(configObject);

export default App;
```

Let's learn about the configObject.

Configure the screens

Each screen will be a React Native component, and will need a name. In this example we will have one home screen (codenamed `Home`) and a details screen (codenamed `Details`).

```
class HomeScreen extends React.Component {
  // ...
}
```

```
class DetailsScreen extends React.Component {
  // ...
}

export default createStackNavigator(
  // THIS IS THE CONFIG OBJECT
  // [codename]: {screen: MyComponent}
  {
    Home: {screen: HomeScreen},
    Details: {screen: DetailsScreen},
  }
);
```

The `createStackNavigator` also allows us to specify which of these screens is the initial one. Let's choose `Home`:

```
export default createStackNavigator(
  {
    Home: {screen: HomeScreen},
    Details: {screen: DetailsScreen},
  },
  {
    initialRouteName: 'Home',
  },
);
```

See full example

Open the next example to see the full code that uses React Navigation.

Read more

For all things React Navigation, check the official docs for detailed setup instructions and API explanations.

Example React Navigation

This example shows the basic usage of React Navigation, where the initial screen has one button that takes you to the second screen, and then you can go back. Notice that each screen is a normal React component, but we also have a `static navigationOptions` to configure the screen a bit.

`App.js` file:

```
import React, { Component } from 'react';
import { View, Text, Button } from 'react-native';
import { createStackNavigator } from 'react-navigation';

class HomeScreen extends React.Component {
  static navigationOptions = {
    title: 'Home',
  };

  render() {
    return (
      <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>
        <Text>Home Screen</Text>
        <Button
          title="Go to Details"
          onPress={() => this.props.navigation.navigate('Details')}
        />
      </View>
    );
  }
}

class DetailsScreen extends React.Component {
  static navigationOptions = {
    title: 'Details',
  };

  render() {
    return (
      <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>
        <Text>Details Screen</Text>
      </View>
    );
  }
}

// This is our "App" component
export default createStackNavigator(
  {
    Home: {
      screen: HomeScreen,
    },
    Details: {
      screen: DetailsScreen,
    },
  },
  {
    initialRouteName: 'Home',
  },
);
```

`index.js` file:

```
import {AppRegistry} from 'react-native';
```

```
import App from './App';
import {name as appName} from './app.json';

AppRegistry.registerComponent(appName, () => App);
```

Exercise

Now is your turn! Let's build something real with React Native. Pick an open data backend, invent a nice idea, then build a simple app that uses that data.

As a tip, most apps have:

- List screen: shows all the data from the server in an "infinitely scrolling" list
- Details screen: once the user presses an item on the list, it opens a details screen with more data on that item
- Something else?

But remember, if you have a different idea that doesn't fit this pattern, please feel free to do whatever you want! The best thing that could happen in this camp is that you end up with a real app that you could use in your daily life, so think about your own needs, or think about your friends' needs.

Feel free to ask the mentors for any help, either in code or feasibility or ideation. Have fun!