Learning React Native by Formini

React Native is a popular framework for building native applications that run on iOS and Android devices. You build these applications using JavaScript. These applications are fast, render smoothly, and truly feel like native applications. This is because unlike the hybrid frameworks of the past, React Native does not use the web view. It uses the multi-threaded native components and APIs that ship with your device.

- it is based on react codebase
- same consepts like state, props, JSX...
- renders the UI with native components...
- ...instead of the DOM

> if you know react then you're almost there! 💯

Getting started:

- There is two ways to settup RN's envirenment.
- *Expo CLI* which is a set of tools built for RN
- React Native CLI requires *Xcode* or *Android Studio*

```
npm install -g expo-cliexpo init ProjectNamecd ProjectNamenpm start
```

> Expo is an open-source platform for making universal native apps for Android, iOS, and the web with JavaScript and React.

Getting started:

```
import React from 'react';
import { Text, View } from 'react-native';
function App() {
  return (
    <View style={{
      flex: 1,
      backgroundColor: '#fff',
      alignItems: 'center',
      justifyContent: 'center'}}>
      <Text>Tempor nulla eiusmod minim occaecat ea.</Text>
    </View>
  )
}
export default App
```

Components are functions that return JSX

- JSX looks like HTML but is actually JavaScript
- Inside of JSX use curly braces to contain JS
- A lot of people go “YUCK!” when they see HTML mixed into JS, but JSX is actually one of the things that makes React awesome 🦄

```
// regular function
function Greeting() {
  const name = "Jamal"
  return ( // Return JSX
      <View>
        <Text>Hello, {name}</Text> {/* Anything inside {} will execute as JS */}
      </View>
  )
}
```

After defining a function component

> you can use it within another component - it is a “custom component”

```
// define the component Greeting
function Greeting() {
  const name = "Jamal"
  return (
      <View>
        <Text>Hello! {name}</Text>
      </View>
  )
}

// Use Greeting component in another component
function App() {
    return (
        <View>
            <Greeting/>
            <Text>Duis laboris sunt est commodo reprehenderit ea aliquip cupidatat aliqua nostrud anim velit.</Text>
        </View>
    )
}
```

Data that comes in from the outside of a component is called > “props” (properties)

This can be passed > from a parent to a child through JSX attributes

Props come into function components as > the first argument to the function.

```
// Props as the first argument to the function
function Greeting(props) {
  return (
      <View>
        <Text>Hello, {props.name}</Text>
      </View>
  )
}

// Set a prop on a child component by passing an attribute
export default function App() {
    return (
        <View>
            <Greeting name="Jamal" />
            <Text>Minim id sint do labore.</Text>
        </View>
    )
}
```

A bit more about JSX

Consider this variable declaration:

```
const element = <h1>Hello, world!</h1>;
```

This funny tag syntax is neither a string nor HTML.

It is called JSX, and it is a syntax extension to JavaScript. We recommend using it with React to describe what the UI should look like. JSX may remind you of a template language, but it comes with the full power of JavaScript.

JSX produces React "elements". We will explore rendering them to the DOM in the next section. Below, you can find the basics of JSX necessary to get you started.

Why JSX?

React embraces the fact that rendering logic is inherently coupled with other UI logic: how events are handled, how the state changes over time, and how the data is prepared for display.

Instead of artificially separating *technologies* by putting markup and logic in separate files, React separates *concerns* with loosely coupled units called "components" that contain both. We will come back to components in a further section, but if you're not yet comfortable putting markup in JS, this talk might convince you otherwise.

React doesn't require using JSX, but most people find it helpful as a visual aid when working with UI inside the JavaScript code. It also allows React to show more useful error and warning messages.

With that out of the way, let's get started!

Embedding Expressions in JSX

In the example below, we declare a variable called `name` and then use it inside JSX by wrapping it in curly braces:

```
const name = 'Josh Perez';
const element = <h1>Hello, {name}</h1>;
ReactDOM.render( element, document.getElementById('root')
);
```

You can put any valid JavaScript expression inside the curly braces in JSX. For example, `2 + 2`, `user.firstName`, or `formatName(user)` are all valid JavaScript expressions.

In the example below, we embed the result of calling a JavaScript function, `formatName(user)`, into an `<h1>` element.

```
function formatName(user) { return user.firstName + ' ' + user.lastName;
}
const user = { firstName: 'Harper', lastName: 'Perez'
};
const element = ( <h1> Hello, {formatName(user)}! </h1>);
ReactDOM.render( element, document.getElementById('root')
);
```

Try it on CodePen

We split JSX over multiple lines for readability. While it isn't required, when doing this, we also recommend wrapping it in parentheses to avoid the pitfalls of automatic semicolon insertion.

JSX is an Expression Too

After compilation, JSX expressions become regular JavaScript function calls and evaluate to JavaScript objects.

This means that you can use JSX inside of `if` statements and `for` loops, assign it to variables, accept it as arguments, and return it from functions:

```
function getGreeting(user) { if (user) { return <h1>Hello, {formatName(user)}!</h1>; } return <h1>Hello, Stranger.</h1>;
}
```

Specifying Attributes with JSX

You may use quotes to specify string literals as attributes:

```
const element = <div tabIndex="0"></div>;
```

You may also use curly braces to embed a JavaScript expression in an attribute:

```
const element = <img src={user.avatarUrl}></img>;
```

Don't put quotes around curly braces when embedding a JavaScript expression in an attribute. You should either use quotes (for string values) or curly braces (for expressions), but not both in the same attribute.

> Warning:
> Since JSX is closer to JavaScript than to HTML, React DOM uses camelCase property naming convention instead of HTML attribute names.
>
> For example, class becomes className in JSX, and tabindex becomes tabIndex.

Specifying Children with JSX

If a tag is empty, you may close it immediately with `/>`, like XML:

```
const element = <img src={user.avatarUrl} />;
```

JSX tags may contain children:

```
const element = ( <div> <h1>Hello!</h1> <h2>Good to see you here.</h2> </div>);
```

JSX Prevents Injection Attacks

It is safe to embed user input in JSX:

```
const title = response.potentiallyMaliciousInput;
// This is safe:
const element = <h1>{title}</h1>;
```

By default, React DOM escapes any values embedded in JSX before rendering them. Thus it ensures that you can never inject anything that's not explicitly written in your application. Everything is converted to a string before being rendered. This helps prevent XSS (cross-site-scripting) attacks.

JSX Represents Objects

Babel compiles JSX down to `React.createElement()` calls.

These two examples are identical:

```
const element = ( <h1 className="greeting"> Hello, world! </h1>);
```

```
const element = React.createElement( 'h1', {className: 'greeting'}, 'Hello, world!'
);
```

`React.createElement()` performs a few checks to help you write bug-free code but essentially it creates an object like this:

```
// Note: this structure is simplified
const element = { type: 'h1', props: { className: 'greeting', children: 'Hello, world!' }
};
```

These objects are called "React elements". You can think of them as descriptions of what you want to see on the screen. React reads these objects and uses them to construct the DOM and keep it up to date.

React hooks

Internal, *changeable* data is called "state".

- State is defined by the `useState` function
- Which returns the data,
- and a function to change that data (in an array).

NEVER set the state variable directly > always use that function

---

When state or props change, your component updates AUTOMATICALLY 🎉

✨ This is the magic of React! ✨

```
// import useState from React
import React, { useState } from 'react'
import { Text, View, Button } from 'react-native';

export default (props) => {
  // call useState and pass a default value
  const [count, setCount] = useState(1)

  // useState returns the current value and and update function
  const incrementCount = () => {
    setCount(count + 1)
  }

  return (
      <View>
        <Text>Hello, {props.name}</Text>
        <Text>You've clicked  {count} times</Text> {/* Display state in {} */}
        <Button
          onPress={incrementCount}
          title="Click me"
        />
      </View>
  )
}
```

A bit more about hooks

Hooks are a new addition in React 16.8. They let you use state and other React features without writing a class.

Equivalent Class Example

If you used classes in React before, this code should look familiar:

```
class Example extends React.Component { constructor(props) { super(props); this.state = { count: 0 }; } render() { return ( <div> <p>You clicked {this.state.count} times</p> <button onClick={() => this.setState({ count: this.state.count + 1 })}> Click me </button> </div>); }
}
```

The state starts as `{ count: 0 }`, and we increment `state.count` when the user clicks a button by calling `this.setState()`. We'll use snippets from this class throughout the page.

> Note
>
> You might be wondering why we're using a counter here instead of a more realistic example. This is to help us focus on the API while we're still making our first steps with Hooks.

Hooks and Function Components

As a reminder, function components in React look like this:

```
const Example = (props) => { // You can use Hooks here! return <div />;
}
```

or this:

```
function Example(props) { // You can use Hooks here! return <div />;
}
```

You might have previously known these as "stateless components". We're now introducing the ability to use React state from these, so we prefer the name "function components".

Hooks **don't** work inside classes. But you can use them instead of writing classes.

What's a Hook?

Our new example starts by importing the `useState` Hook from React:

```
import React, { useState } from 'react';
function Example() { // ...
}
```

What is a Hook? A Hook is a special function that lets you “hook into” React features. For example, `useState` is a Hook that lets you add React state to function components. We’ll learn other Hooks later.

When would I use a Hook? If you write a function component and realize you need to add some state to it, previously you had to convert it to a class. Now you can use a Hook inside the existing function component. We’re going to do that right now!

> Note:
>
> There are some special rules about where you can and can’t use Hooks within a component. We’ll learn them in Rules of Hooks.

Declaring a State Variable

In a class, we initialize the `count` state to `0` by setting `this.state` to `{ count: 0 }` in the constructor:

```
class Example extends React.Component { constructor(props) { super(props); this.state = { count: 0 }; }
```

In a function component, we have no `this`, so we can’t assign or read `this.state`. Instead, we call the `useState` Hook directly inside our component:

```
import React, { useState } from 'react';
function Example() { // Declare a new state variable, which we'll call "count" const [count, setCount] = useState(0);
```

What does calling `useState` do? It declares a “state variable”. Our variable is called `count` but we could call it anything else, like `banana`. This is a way to “preserve” some values between the function calls — `useState` is a new way to use the exact same capabilities that `this.state` provides in a class. Normally, variables “disappear” when the function exits but state variables are preserved by React.

What do we pass to `useState` as an argument? The only argument to the `useState()` Hook is the initial state. Unlike with classes, the state doesn’t have to be an object. We can keep a number or a string if that’s all we need. In our example, we just want a number for how many times the user clicked, so pass `0` as initial state for our variable. (If we wanted to store two different values in state, we would call `useState()` twice.)

What does `useState` return? It returns a pair of values: the current state and a function that updates it. This is why we write `const [count, setCount] = useState()`. This is similar to `this.state.count` and `this.setState` in a class, except you get them in a pair. If you’re not familiar with the syntax we used, we’ll come back to it at the bottom of this page.

Now that we know what the `useState` Hook does, our example should make more sense:

```
import React, { useState } from 'react';
function Example() { // Declare a new state variable, which we'll call "count" const [count, setCount] = useState(0);
```

We declare a state variable called `count`, and set it to `0`. React will remember its current value between re-renders, and provide the most recent one to our function. If we want to update the current `count`, we can call `setCount`.

> Note
>
> You might be wondering: why is useState not named createState instead?
>
> “Create” wouldn’t be quite accurate because the state is only created the first time our component renders. During the next renders, useState gives us the current state. Otherwise it wouldn’t be “state” at all! There’s also a reason why Hook names always start with use. We’ll learn why later in the Rules of Hooks.

Reading State

When we want to display the current count in a class, we read `this.state.count`:

```
<p>You clicked {this.state.count} times</p>
```

In a function, we can use `count` directly:

```
<p>You clicked {count} times</p>
```

Updating State

In a class, we need to call `this.setState()` to update the `count` state:

```
<button onClick={() => this.setState({ count: this.state.count + 1 })}> Click me </button>
```

In a function, we already have `setCount` and `count` as variables so we don't need `this`:

```
<button onClick={() => setCount(count + 1)}> Click me </button>
```

Recap

Let's now **recap what we learned line by line** and check our understanding.

```
1: import React, { useState } from 'react'; 2: 3: function Example() { 4: const [count, setCount] = useState(0); 5: 6: return (
7: <div> 8: <p>You clicked {count} times</p> 9: <button onClick={() => setCount(count + 1)}>
10: Click me
11: </button>12: </div>13: );
14: }
```

- **Line 1:** We import the `useState` Hook from React. It lets us keep local state in a function component.
- **Line 4:** Inside the `Example` component, we declare a new state variable by calling the `useState` Hook. It returns a pair of values, to which we give names. We're calling our variable `count` because it holds the number of button clicks. We initialize it to zero by passing `0` as the only `useState` argument. The second returned item is itself a function. It lets us update the `count` so we'll name it `setCount`.
- **Line 9:** When the user clicks, we call `setCount` with a new value. React will then re-render the `Example` component, passing the new `count` value to it.

This might seem like a lot to take in at first. Don't rush it! If you're lost in the explanation, look at the code above again and try to read it from top to bottom. We promise that once you try to "forget" how state works in classes, and look at this code with fresh eyes, it will make sense.

Tip: What Do Square Brackets Mean?

You might have noticed the square brackets when we declare a state variable:

```
const [count, setCount] = useState(0);
```

The names on the left aren't a part of the React API. You can name your own state variables:

```
const [fruit, setFruit] = useState('banana');
```

This JavaScript syntax is called "array destructuring". It means that we're making two new variables `fruit` and `setFruit`, where `fruit` is set to the first value returned by `useState`, and `setFruit` is the second. It is equivalent to this code:

```
var fruitStateVariable = useState('banana'); // Returns a pair var fruit = fruitStateVariable[0]; // First item in a pair var
setFruit = fruitStateVariable[1]; // Second item in a pair
```

When we declare a state variable with `useState`, it returns a pair — an array with two items. The first item is the current value, and the second is a function that lets us update it. Using `[0]` and `[1]` to access them is a bit confusing because they have a specific meaning. This is why we use array destructuring instead.

> Note
>
> You might be curious how React knows which component useState corresponds to since we're not passing anything like this back to React. We'll answer this question and many others in the FAQ section.

Tip: Using Multiple State Variables

Declaring state variables as a pair of `[something, setSomething]` is also handy because it lets us give *different* names to different state variables if we want to use more than one:

```
function ExampleWithManyStates() { // Declare multiple state variables! const [age, setAge] = useState(42); const [fruit,
setFruit] = useState('banana'); const [todos, setTodos] = useState([{ text: 'Learn Hooks' }]);
```

In the above component, we have `age`, `fruit`, and `todos` as local variables, and we can update them individually:

```
function handleOrangeClick() { // Similar to this.setState({ fruit: 'orange' }) setFruit('orange'); }
```

You **don't have to** use many state variables. State variables can hold objects and arrays just fine, so you can still group related data together. However, unlike `this.setState` in a class, updating a state variable always *replaces* it instead of merging it.

Make lists of things by looping over an array of data with `map`

Return an element from each loop iteration

Provide a unique `key` to each element in the list to ensure best performance

```
const DATA = [
  { id: 4, title: 'A New Hope' },
  { id: 5, title: 'The Empire Strikes Back' },
  { id: 6, title: 'Return of the Jedi' },
]

function MyList() {
  return (
    <View>
    { // Wrap entire JS in {}
      DATA.map(
        (item) => <Text key={item.id}>{item.title}</Text>
      )
    }
    </View>
  )
}

export default function App() {
  return <MyList items={DATA} />
}
```

Style

- you style your application using JavaScript.
- All of the core components accept a prop named style.
- It's CSS, except names are written using camel casing...
- ...e.g. *backgroundColor* rather than background-color.
- It's cleaner to use StyleSheet.create

1. Define a styles object using *StyleSheet.create*

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

const styles = StyleSheet.create({
  bigBlue: {
    color: 'blue',
    fontWeight: 'bold',
    fontSize: 30,
  },
  red: {
    color: 'red',
  },
});
```

1. Set use it with `style={ styles.bigBlue }`

```
export default () => {
  return (
    <View>
      <Text style={styles.red}>just red</Text>
      <Text style={styles.bigBlue}>just bigBlue</Text>
      <Text style={[styles.bigBlue, styles.red]}>bigBlue, then red</Text>
      <Text style={[styles.red, styles.bigBlue]}>red, then bigBlue</Text>
    </View>
  );
}
```

Layout #1

1. We can specify withs and heights

```
export default class FixedDimensionsBasics extends Component {
  render() {
    return (
      <View>
        <View style={{width: 50, height: 50, backgroundColor: 'powderblue'}} />
        <View style={{width: 100, height: 100, backgroundColor: 'skyblue'}} />
        <View style={{width: 150, height: 150, backgroundColor: 'steelblue'}} />
      </View>
    );
  }
}
```

1. Or make the most of Flexbox

```
export default class FlexDimensionsBasics extends Component {
  render() {
    return (
      // Try removing the `flex: 1` on the parent View.
      // The parent will not have dimensions, so the children can't expand.
      // What if you add `height: 300` instead of `flex: 1`?
      <View style={{flex: 1}}>
        <View style={{flex: 1, backgroundColor: 'powderblue'}} />
        <View style={{flex: 2, backgroundColor: 'skyblue'}} />
        <View style={{flex: 3, backgroundColor: 'steelblue'}} />
      </View>
    );
  }
}
```

Layout #2 Flexbox

```
{
  flex: 1,
  flexDirection: 'row', //column
  justifyContent: 'space-between',
  alignItems: 'center'
}
```

```
<View style={{
  flex: 1,
  flexDirection: 'column',
  justifyContent: 'center', //space-between, space-around, start, end
  alignItems: 'center' //start, end, stretch
}}>
  <Text>Kuri</Text>
  <Text>Login</Text>
</View>
```

Common UI components

1. View

```
<View style={styles.container}>
  {/* Any valid Components */}
</View>
```

2. Text

```
<View style={styles.container}>
  <Text>Mollit non proident nulla incididunt proident ex dolore irure.</Text>
</View>
```

3. Buttons

```
<View style={styles.container}>
  <View style={styles.buttonContainer}>
    <Button
      onPress={this._onPressButton}
      title="Press Me"
    />
  </View>
  <View style={styles.buttonContainer}>
    <Button
      onPress={this._onPressButton}
      title="Press Me"
      color="#841584"
    />
  </View>
  <View style={styles.alternativeLayoutButtonContainer}>
    <Button
      onPress={this._onPressButton}
      title="This looks great!"
    />
    <Button
      onPress={this._onPressButton}
      title="OK!"
      color="#841584"
    />
  </View>
  </View>
```

4. ScrollView

```
<ScrollView>
  {/* Any valid Components */}
</<ScrollView>
```

5. Image

```
<Image
  source={{
    uri: "https://facebook.github.io/react-native/img/tiny_logo.png",
    width: 64,
    height: 64
  }}
/>
```

6. Flatlist

```
<FlatList
  data={[
    {key: 'Devin'},
    {key: 'Dan'},
    {key: 'Dominic'},
    {key: 'Jackson'},
    {key: 'James'},
    {key: 'Joel'},
    {key: 'John'},
    {key: 'Jillian'},
    {key: 'Jimmy'},
    {key: 'Julie'},
  ]}
  renderItem={({item}) => <Text style={styles.item} key={item.key}>{item.key}</Text>}
/>
```

7. To make any section clickable: TouchableOpacity

```
<TouchableOpacity
  onPress={anyfunction}
>
  {/* any components/views */}
</TouchableOpacity>
```

Perform Async functions and side effects inside of `useEffect`

(takes a callback)

The second argument is an array of dependencies.

Include any variable the useEffect uses that might change, or an empty array if there are none.

```
// import useEffect from React
import React, {useState, useEffect} from 'react'

function TrainignsPage(props) {
    const [trainings, setTrainigs] = useState([])

    // Call useEffect with a function as the first argument
    useEffect(() => {
        // Do async actions like data fetching inside the callback
        fetch("https://api.formini.dz/trainings")
            .then(response => response.json())
            .then($trainigns => setTrainings($trainigns))
    }, [props.id])
    // the second argument is an array of dependencies.
    // ⚠️ don't forget this!

    return (
        <View>
        {
            trainings.map(
                training => <Text key={training.id}>{training.Title}</Text>
            )
        }
        </View>
    )
}
```

A bit more about the Effect Hook

Hooks are a new addition in React 16.8. They let you use state and other React features without writing a class.

The *Effect Hook* lets you perform side effects in function components:

```
import React, { useState, useEffect } from 'react';
function Example() { const [count, setCount] = useState(0); // Similar to componentDidMount and componentDidUpdate: useEffect(() => { // Update the document title using the browser API document.title = `You clicked ${count} times`; }); return ( <div> <p>You clicked {count} times</p> <button onClick={() => setCount(count + 1)}> Click me </button> </div>);
}
```

This snippet is based on the counter example from the previous page, but we added a new feature to it: we set the document title to a custom message including the number of clicks.

Data fetching, setting up a subscription, and manually changing the DOM in React components are all examples of side effects. Whether or not you're used to calling these operations "side effects" (or just "effects"), you've likely performed them in your components before.

> Tip
>
> If you're familiar with React class lifecycle methods, you can think of useEffect Hook as componentDidMount, componentDidUpdate, and componentWillUnmount combined.

There are two common kinds of side effects in React components: those that don't require cleanup, and those that do. Let's look at this distinction in more detail.

Effects Without Cleanup

Sometimes, we want to **run some additional code after React has updated the DOM.** Network requests, manual DOM mutations, and logging are common examples of effects that don't require a cleanup. We say that because we can run them and immediately forget about them. Let's compare how classes and Hooks let us express such side effects.

Example Using Classes

In React class components, the `render` method itself shouldn't cause side effects. It would be too early — we typically want to perform our effects *after* React has updated the DOM.

This is why in React classes, we put side effects into `componentDidMount` and `componentDidUpdate`. Coming back to our example, here is a React counter class component that updates the document title right after React makes changes to the DOM:

```
class Example extends React.Component { constructor(props) { super(props); this.state = { count: 0 }; } componentDidMount() { document.title = `You clicked ${this.state.count} times`; } componentDidUpdate() { document.title = `You clicked ${this.state.count} times`; } render() { return ( <div> <p>You clicked {this.state.count} times</p> <button onClick={() => this.setState({ count: this.state.count + 1 })}> Click me </button> </div>); }
}
```

Note how **we have to duplicate the code between these two lifecycle methods in class.**

This is because in many cases we want to perform the same side effect regardless of whether the component just mounted, or if it has been updated. Conceptually, we want it to happen after every render — but React class components don't have a method like this. We could extract a separate method but we would still have to call it in two places.

Now let's see how we can do the same with the `useEffect` Hook.

Example Using Hooks

We've already seen this example at the top of this page, but let's take a closer look at it:

```
import React, { useState, useEffect } from 'react';
function Example() { const [count, setCount] = useState(0); useEffect(() => { document.title = `You clicked ${count} times`; });
return ( <div> <p>You clicked {count} times</p> <button onClick={() => setCount(count + 1)}> Click me </button> </div>);
}
```

What does `useEffect` do? By using this Hook, you tell React that your component needs to do something after render. React will remember the function you passed (we'll refer to it as our "effect"), and call it later after performing the DOM updates. In this effect, we set the document title, but we could also perform data fetching or call some other imperative API.

Why is `useEffect` called inside a component? Placing `useEffect` inside the component lets us access the `count` state variable (or any props) right from the effect. We don't need a special API to read it — it's already in the function scope. Hooks embrace JavaScript closures and avoid introducing React-specific APIs where JavaScript already provides a solution.

Does `useEffect` run after every render? Yes! By default, it runs both after the first render *and* after every update. (We will later talk about how to customize this.) Instead of thinking in terms of "mounting" and "updating", you might find it easier to think that effects happen "after render". React guarantees the DOM has been updated by the time it runs the effects.

Detailed Explanation

Now that we know more about effects, these lines should make sense:

```
function Example() { const [count, setCount] = useState(0); useEffect(() => { document.title = `You clicked ${count} times`; });
}
```

We declare the `count` state variable, and then we tell React we need to use an effect. We pass a function to the `useEffect` Hook. This function we pass *is* our effect. Inside our effect, we set the document title using the

`document.title` browser API. We can read the latest `count` inside the effect because it's in the scope of our function. When React renders our component, it will remember the effect we used, and then run our effect after updating the DOM. This happens for every render, including the first one.

Experienced JavaScript developers might notice that the function passed to `useEffect` is going to be different on every render. This is intentional. In fact, this is what lets us read the `count` value from inside the effect without worrying about it getting stale. Every time we re-render, we schedule a *different* effect, replacing the previous one. In a way, this makes the effects behave more like a part of the render result — each effect "belongs" to a particular render. We will see more clearly why this is useful later on this page.

> Tip
>
> Unlike componentDidMount or componentDidUpdate, effects scheduled with useEffect don't block the browser from updating the screen. This makes your app feel more responsive. The majority of effects don't need to happen synchronously. In the uncommon cases where they do (such as measuring the layout), there is a separate useLayoutEffect Hook with an API identical to useEffect.

Effects with Cleanup

Earlier, we looked at how to express side effects that don't require any cleanup. However, some effects do. For example, **we might want to set up a subscription** to some external data source. In that case, it is important to clean up so that we don't introduce a memory leak! Let's compare how we can do it with classes and with Hooks.

Example Using Classes

In a React class, you would typically set up a subscription in `componentDidMount`, and clean it up in `componentWillUnmount`. For example, let's say we have a `ChatAPI` module that lets us subscribe to a friend's online status. Here's how we might subscribe and display that status using a class:

```
class FriendStatus extends React.Component { constructor(props) { super(props); this.state = { isOnline: null }; this.handleStatusChange = this.handleStatusChange.bind(this); } componentDidMount() { ChatAPI.subscribeToFriendStatus( this.props.friend.id, this.handleStatusChange ); } componentWillUnmount() { ChatAPI.unsubscribeFromFriendStatus( this.props.friend.id, this.handleStatusChange ); } handleStatusChange(status) { this.setState({ isOnline: status.isOnline }); } render() { if (this.state.isOnline === null) { return 'Loading...'; } return this.state.isOnline ? 'Online' : 'Offline'; }
}
```

Notice how `componentDidMount` and `componentWillUnmount` need to mirror each other. Lifecycle methods force us to split this logic even though conceptually code in both of them is related to the same effect.

> Note
>
> Eagle-eyed readers may notice that this example also needs a componentDidUpdate method to be fully correct. We'll ignore this for now but will come back to it in a later section of this page.

Example Using Hooks

Let's see how we could write this component with Hooks.

You might be thinking that we'd need a separate effect to perform the cleanup. But code for adding and removing a subscription is so tightly related that `useEffect` is designed to keep it together. If your effect returns a function, React will run it when it is time to clean up:

```
import React, { useState, useEffect } from 'react';
function FriendStatus(props) { const [isOnline, setIsOnline] = useState(null); useEffect(() => { function handleStatusChange(status) { setIsOnline(status.isOnline); } ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange); // Specify how to clean up after this effect: return function cleanup() { ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange); }; }); if (isOnline === null) { return 'Loading...'; } return isOnline ? 'Online' : 'Offline';
}
```

Why did we return a function from our effect? This is the optional cleanup mechanism for effects. Every effect may return a function that cleans up after it. This lets us keep the logic for adding and removing subscriptions close to each other. They're part of the same effect!

When exactly does React clean up an effect? React performs the cleanup when the component unmounts. However, as we learned earlier, effects run for every render and not just once. This is why React *also* cleans up effects from the previous render before running the effects next time. We'll discuss why this helps avoid bugs and how to opt out of this behavior in case it creates performance issues later below.

> Note
>
> We don't have to return a named function from the effect. We called it cleanup here to clarify its purpose, but you could return an arrow function or call it something different.

Recap

We've learned that `useEffect` lets us express different kinds of side effects after a component renders. Some effects might require cleanup so they return a function:

```
useEffect(() => { function handleStatusChange(status) { setIsOnline(status.isOnline); } ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange); return () => { ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange); }; });
```

Other effects might not have a cleanup phase, and don't return anything.

```
useEffect(() => { document.title = `You clicked ${count} times`; });
```

The Effect Hook unifies both use cases with a single API.

If you feel like you have a decent grasp on how the Effect Hook works, or if you feel overwhelmed, you can jump to the next page about Rules of Hooks now.

Tips for Using Effects

We'll continue this page with an in-depth look at some aspects of `useEffect` that experienced React users will likely be curious about. Don't feel obligated to dig into them now. You can always come back to this page to learn more details about the Effect Hook.

Tip: Use Multiple Effects to Separate Concerns

One of the problems we outlined in the Motivation for Hooks is that class lifecycle methods often contain unrelated logic, but related logic gets broken up into several methods. Here is a component that combines the counter and the friend status indicator logic from the previous examples:

```
class FriendStatusWithCounter extends React.Component { constructor(props) { super(props); this.state = { count: 0, isOnline: null }; this.handleStatusChange = this.handleStatusChange.bind(this); } componentDidMount() { document.title = `You clicked ${this.state.count} times`; ChatAPI.subscribeToFriendStatus( this.props.friend.id, this.handleStatusChange ); } componentDidUpdate() { document.title = `You clicked ${this.state.count} times`; } componentWillUnmount() { ChatAPI.unsubscribeFromFriendStatus( this.props.friend.id, this.handleStatusChange ); } handleStatusChange(status) { this.setState({ isOnline: status.isOnline }); } // ...
```

Note how the logic that sets `document.title` is split between `componentDidMount` and `componentDidUpdate`. The subscription logic is also spread between `componentDidMount` and `componentWillUnmount`. And `componentDidMount` contains code for both tasks.

So, how can Hooks solve this problem? Just like you can use the *State* Hook more than once, you can also use several effects. This lets us separate unrelated logic into different effects:

```
function FriendStatusWithCounter(props) { const [count, setCount] = useState(0); useEffect(() => { document.title = `You clicked ${count} times`; }); const [isOnline, setIsOnline] = useState(null); useEffect(() => { function handleStatusChange(status) { setIsOnline(status.isOnline); } ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange); return () => { ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange); }; }); // ...
}
```

Hooks let us split the code based on what it is doing rather than a lifecycle method name. React will apply *every* effect used by the component, in the order they were specified.

Explanation: Why Effects Run on Each Update

If you're used to classes, you might be wondering why the effect cleanup phase happens after every re-render, and not just once during unmounting. Let's look at a practical example to see why this design helps us create components with fewer bugs.

Earlier on this page, we introduced an example `FriendStatus` component that displays whether a friend is online or not. Our class reads `friend.id` from `this.props`, subscribes to the friend status after the component mounts, and

unsubscribes during unmounting:

```
componentDidMount() { ChatAPI.subscribeToFriendStatus( this.props.friend.id, this.handleStatusChange ); } componentWillUnmount() { ChatAPI.unsubscribeFromFriendStatus( this.props.friend.id, this.handleStatusChange ); }
```

But what happens if the `friend` prop changes while the component is on the screen? Our component would continue displaying the online status of a different friend. This is a bug. We would also cause a memory leak or crash when unmounting since the unsubscribe call would use the wrong friend ID.

In a class component, we would need to add `componentDidUpdate` to handle this case:

```
componentDidMount() { ChatAPI.subscribeToFriendStatus( this.props.friend.id, this.handleStatusChange ); } componentDidUpdate(prevProps) { // Unsubscribe from the previous friend.id ChatAPI.unsubscribeFromFriendStatus( prevProps.friend.id, this.handleStatusChange ); // Subscribe to the next friend.id ChatAPI.subscribeToFriendStatus( this.props.friend.id, this.handleStatusChange ); } componentWillUnmount() { ChatAPI.unsubscribeFromFriendStatus( this.props.friend.id, this.handleStatusChange ); }
```

Forgetting to handle `componentDidUpdate` properly is a common source of bugs in React applications.

Now consider the version of this component that uses Hooks:

```
function FriendStatus(props) { // ... useEffect(() => { // ... ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange); return () => { ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange); }; });
```

It doesn't suffer from this bug. (But we also didn't make any changes to it.)

There is no special code for handling updates because `useEffect` handles them *by default*. It cleans up the previous effects before applying the next effects. To illustrate this, here is a sequence of subscribe and unsubscribe calls that this component could produce over time:

```
// Mount with { friend: { id: 100 } } props
ChatAPI.subscribeToFriendStatus(100, handleStatusChange); // Run first effect
// Update with { friend: { id: 200 } } props
ChatAPI.unsubscribeFromFriendStatus(100, handleStatusChange); // Clean up previous effect
ChatAPI.subscribeToFriendStatus(200, handleStatusChange); // Run next effect
// Update with { friend: { id: 300 } } props
ChatAPI.unsubscribeFromFriendStatus(200, handleStatusChange); // Clean up previous effect
ChatAPI.subscribeToFriendStatus(300, handleStatusChange); // Run next effect
// Unmount
ChatAPI.unsubscribeFromFriendStatus(300, handleStatusChange); // Clean up last effect
```

This behavior ensures consistency by default and prevents bugs that are common in class components due to missing update logic.

Tip: Optimizing Performance by Skipping Effects

In some cases, cleaning up or applying the effect after every render might create a performance problem. In class components, we can solve this by writing an extra comparison with `prevProps` or `prevState` inside `componentDidUpdate`:

```
componentDidUpdate(prevProps, prevState) { if (prevState.count !== this.state.count) { document.title = `You clicked ${this.state.count} times`; }
}
```

This requirement is common enough that it is built into the `useEffect` Hook API. You can tell React to *skip* applying an effect if certain values haven't changed between re-renders. To do so, pass an array as an optional second argument to `useEffect`:

```
useEffect(() => { document.title = `You clicked ${count} times`;
}, [count]); // Only re-run the effect if count changes
```

In the example above, we pass `[count]` as the second argument. What does this mean? If the `count` is `5`, and then our component re-renders with `count` still equal to `5`, React will compare `[5]` from the previous render and `[5]` from the next render. Because all items in the array are the same (`5 === 5`), React would skip the effect. That's our optimization.

When we render with `count` updated to `6`, React will compare the items in the `[5]` array from the previous render to items in the `[6]` array from the next render. This time, React will re-apply the effect because `5 !== 6`. If there are multiple items in the array, React will re-run the effect even if just one of them is different.

This also works for effects that have a cleanup phase:

```
useEffect(() => { function handleStatusChange(status) { setIsOnline(status.isOnline); }
ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange); return () => {
ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange); };
}, [props.friend.id]); // Only re-subscribe if props.friend.id changes
```

In the future, the second argument might get added automatically by a build-time transformation.

> Note
>
> If you use this optimization, make sure the array includes all values from the component

> scope (such as props and state) that change over time and that are used by the effect. Otherwise, your code will reference stale values from previous renders. Learn more about how to deal with functions and what to do when the array changes too often.
>
> If you want to run an effect and clean it up only once (on mount and unmount), you can pass an empty array ([]) as a second argument. This tells React that your effect doesn't depend on any values from props or state, so it never needs to re-run. This isn't handled as a special case — it follows directly from how the dependencies array always works.
>
> If you pass an empty array ([]), the props and state inside the effect will always have their initial values. While passing [] as the second argument is closer to the familiar componentDidMount and componentWillUnmount mental model, there are usually better solutions to avoid re-running effects too often. Also, don't forget that React defers running useEffect until after the browser has painted, so doing extra work is less of a problem.
>
> We recommend using the exhaustive-deps rule as part of our eslint-plugin-react-hooks package. It warns when dependencies are specified incorrectly and suggests a fix.

Images

1. From assets (local images)

```
<Image source={{ require('./images/logo.png') }} />
```

2. From network

```
<Image
  source={{uri: 'https://facebook.github.io/react/logo-og.png'}}
  style={{width: 400, height: 400}}
/>
```

3. Uri Data Images

```
<Image
  source={{ uri: '
/>
```

Images #2

```
<ImageBackground source={...} style={{width: '100%', height: '100%'}}>
  <Text>Inside</Text>
</ImageBackground>
```

Example with local images

```
return (
  <ImageBackground
    source={require('./images/background.png')}
    style={{width: '100%', height: '100%'}}
  >
    <Text>Futura Kuri</Text>
  </ImageBackground>
);
```

Navigation

1. Installation of dependencies

```
npm i react-navigation
npm i react-native-gesture-handler
npm i react-navigation-stack
```

2. Configuration of the main navigator

```
import { createAppContainer } from 'react-navigation';
import { createStackNavigator } from 'react-navigation-stack';
import DetailsScreen from './screens/details'
import HomeScreen from './screens/home'

const MainNavigator = createStackNavigator({
  // Home: { screen: HomeScreen },
  Details: { screen: DetailsScreen },
});
const App = createAppContainer(MainNavigator);
export default App;
```

3. Configuration of headers

```
const HomeScreen = () => JSX
HomeScreen.navigationOptions = {
  title: 'Writings'
}
export default HomeScreen
```

4. Other header common options

```
AScreen.navigationOptions = {
  title: 'Home',
  headerStyle: {
    backgroundColor: '#000000',
  },
  headerTitleStyle: {
    color: 'white',
    fontWeight: 'bold',
  },
};
```

To remove the default header

```
AScreen.navigationOptions = {
  header: null
};
```

Navigation #2

1. Navigation from A to B

page A

```
const HomeScreen = (props) => {
  const { navigate } = props.navigation;
  const details = () => {
    navigate('Details')
  }
  return (
    <Button
      title="Go to details page"
      onPress={details}
    />
  );
}
HomeScreen.navigationOptions = {
  title: 'Home',
};
```

page B

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

const DetailsScreen = () => {
  return (
    <View>
      <Text>DETAILS PAGE</Text>
    </View>
  )
}
export default DetailsScreen
```

2. Passing data from A to B

```
<Button
  title="Go to details page"
  onPress={() => navigate('Details', { mydata: ['alpha', 'beta'], name: 'jamal' })}
/>
```

3. Reading data from the previous page

```
const ProfileScreen = (props) => {
  const { getParam } = props.navigation
  const mydata = getParam('mydata', [])
  return (
    <View>
    {
      mydata.map((item) =><Text>{item}</Text>)
    }
    </View>
  )
}
```

hint: desctructuring

```
// lists:
const [ a, b ] = [1, 2, 3, 4];

// objects
const { navigate } = {item: any, navigate: any}
```

Tab navigation

Possibly the most common style of navigation in mobile apps is tab-based navigation. This can be tabs on the bottom of the screen or on the top below the header (or even instead of a header).

This guide covers `createBottomTabNavigator`. You may also use `createMaterialBottomTabNavigator` and `createMaterialTopTabNavigator` to add tabs to your application.

Minimal example of tab-based navigation

```
import React from 'react';
import { Text, View } from 'react-native';
import { createAppContainer } from 'react-navigation';
import { createBottomTabNavigator } from 'react-navigation-tabs';

class HomeScreen extends React.Component {
  render() {
    return (
      <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}><Text>Home!</Text></View>);
  }
}

class SettingsScreen extends React.Component {
  render() {
    return (
      <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}><Text>Settings!</Text></View>);
  }
}

const TabNavigator = createBottomTabNavigator({
  Home: HomeScreen,
  Settings: SettingsScreen,
});

export default createAppContainer(TabNavigator);
```

→ Run this code

Customizing the appearance

This is similar to how you would customize a stack navigator — there are some properties that are set when you initialize the tab navigator and others that can be customized per-screen in `navigationOptions`.

```
// You can import Ionicons from @expo/vector-icons if you use Expo or
// react-native-vector-icons/Ionicons otherwise.
import Ionicons from 'react-native-vector-icons/Ionicons';
import { createAppContainer } from 'react-navigation';
import { createBottomTabNavigator } from 'react-navigation-tabs';

export default createBottomTabNavigator(
  {
    Home: HomeScreen,
    Settings: SettingsScreen,
  },
  {
    defaultNavigationOptions: ({ navigation }) => ({
      tabBarIcon: ({ focused, horizontal, tintColor }) => {
        const { routeName } = navigation.state;
        let IconComponent = Ionicons;
        let iconName;
        if (routeName === 'Home') {
          iconName = `ios-information-circle${focused ? '' : '-outline'}`;
          // Sometimes we want to add badges to some icons.
          // You can check the implementation below.
          IconComponent = HomeIconWithBadge;
        } else if (routeName === 'Settings') {
          iconName = `ios-options`;
        }

        // You can return any component that you like here!
        return <IconComponent name={iconName} size={25} color={tintColor} />;
      },
    }),
    tabBarOptions: {
```

```
      activeTintColor: 'tomato',
      inactiveTintColor: 'gray',
    },
  }
);
```

→ Run this code

Let's dissect this:

- `tabBarIcon` is a property on `navigationOptions`, so we know we can use it on our screen components, but in this case chose to put it in the `createBottomTabNavigator` configuration in order to centralize the icon configuration for convenience.
- `tabBarIcon` is a function that is given the `focused` state, `tintColor`, and `horizontal` param, which is a boolean. If you take a peek further down in the configuration you will see `tabBarOptions` and `activeTintColor` and `inactiveTintColor`. These default to the iOS platform defaults, but you can change them here. The `tintColor` that is passed through to the `tabBarIcon` is either the active or inactive one, depending on the `focused` state (focused is active). The orientation state `horizontal` is `true` when the device is in landscape, otherwise is `false` for portrait.
- Read the full API reference for further information on `createBottomTabNavigator` configuration options.

Add badges to icons

Sometimes we want to add badges to some icons. A common way is to use an extra view container and style the badge element with absolute positioning.

```
export default class IconWithBadge extends React.Component {
  render() {
    const { name, badgeCount, color, size } = this.props;
    return (
      <View style={{ width: 24, height: 24, margin: 5 }}><Ionicons name={name} size={size} color={color} />{badgeCount > 0 && (
          <Viewstyle={{
              // If you're using react-native < 0.57 overflow outside of parent
              // will not work on Android, see https://git.io/fhLJ8
              position: 'absolute',
              right: -6,
              top: -3,
              backgroundColor: 'red',
              borderRadius: 6,
              width: 12,
              height: 12,
              justifyContent: 'center',
              alignItems: 'center',
            }}><Text style={{ color: 'white', fontSize: 10, fontWeight: 'bold' }}>{badgeCount}
            </Text></View>)}
      </View>);
  }
}
```

From UI perspective this component is ready to use, but you still need to find some way to pass down the badge count properly from somewhere else, like using React Context, Redux, MobX or event emitters.

```
const HomeIconWithBadge = props => {
  // You should pass down the badgeCount in some other ways like React Context API, Redux, MobX or event emitters.
  return <IconWithBadge {...props} badgeCount={3} />;
};
export default HomeIconWithBadge;
```

Jumping between tabs

Switching from one tab to another has a familiar API — `this.props.navigation.navigate`.

```
import { Button, Text, View } from 'react-native';

class HomeScreen extends React.Component {
  render() {
    return (
      <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}><Text>Home!</Text><Buttontitle="Go to Settings"
  }
}
```

```
class SettingsScreen extends React.Component {
  render() {
    return (
      <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}><Text>Settings!</Text><Buttontitle="Go to Home"
  }
}
```

→ Run this code

A stack navigator for each tab

Usually tabs don't just display one screen — for example, on your Twitter feed, you can tap on a tweet and it brings you to a new screen within that tab with all of the replies. You can think of this as there being separate navigation stacks within each tab, and that's exactly how we will model it in React Navigation.

```
import { createAppContainer } from 'react-navigation';
import { createStackNavigator } from 'react-navigation-stack';
import { createBottomTabNavigator } from 'react-navigation-tabs';

class DetailsScreen extends React.Component {
  render() {
    return (
      <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}><Text>Details!</Text></View>);
  }
}

class HomeScreen extends React.Component {
  render() {
    return (
      <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>{/* other code from before here */}
        <Buttontitle="Go to Details"onPress={() => this.props.navigation.navigate('Details')}/></View>);
  }
}

class SettingsScreen extends React.Component {
  render() {
    return (
      <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>{/* other code from before here */}
        <Buttontitle="Go to Details"onPress={() => this.props.navigation.navigate('Details')}/></View>);
  }
}

const HomeStack = createStackNavigator({
  Home: HomeScreen,
  Details: DetailsScreen,
});

const SettingsStack = createStackNavigator({
  Settings: SettingsScreen,
  Details: DetailsScreen,
});

export default createAppContainer(
  createBottomTabNavigator(
    {
      Home: HomeStack,
      Settings: SettingsStack,
    },
    {
      /* Other configuration remains unchanged */
    }
  )
);
```

→ Run this code

Why do we need a TabNavigator instead of TabBarIOS or some other component?

It's common to attempt to use a standalone tab bar component without integrating it into the navigation library you use in your app. In some cases, this works fine! You should be warned, however, that you may run into some frustrating unanticipated issues when doing this.

For example, React Navigation's `TabNavigator` takes care of handling the Android back button for you, while standalone components typically do not. Additionally, it is more difficult for you (as the developer) to perform actions such as "jump to this tab and then go to this screen" if you need to call into two distinct APIs for it. Lastly, mobile user

interfaces have numerous small design details that require that certain components are aware of the layout or presence of other components — for example, if you have a translucent tab bar, content should scroll underneath it and the scroll view should have an inset on the bottom equal to the height of the tab bar so you can see all of the content. Double tapping the tab bar should make the active navigation stack pop to the top of the stack, and doing it again should scroll the active scroll view in that stack scroll to the top. While not all of these behaviors are implemented out of the box yet with React Navigation, they will be and you will not get any of this if you use a standalone tab view component.

Text Inputs

```
import { TextInput, ... } from 'react-native';

const MyInputsComponent = (props) => {
  const [title, setTitle] = useState('')

  return (
    <View style={{padding: 10}}>
      <TextInput
        style={{height: 40}}
        placeholder="ex: Learning React!"
        onChangeText={(text) => setTitle(text)}
        value={title}
      />
      <Text style={{padding: 10, fontSize: 42}}>
        {title.toUpperCase()}
      </Text>
    </View>
  );
}
```

More about inputs

1. TextInput properties

   ```
   <TextInput
     {...props} // Inherit any props passed to it; e.g., multiline, numberOfLines below
     multiline
     numberOfLines={4}
     editable
     maxLength={40}
   />
   ```

2. Switch

> Renders a boolean input.

```
<Switch
  onValueChange = {(value) => {setValue(!value)}}
  value = {_value}
/>
```

Icons

import { Ionicons } from '@expo/vector-icons';

```
<Ionicons name="md-checkmark-circle" size={32} color="green" />
```

List of available icons

https://expo.github.io/vector-icons/

GraphQL

GraphQL is an open-source data query and manipulation language for APIs, and a runtime for fulfilling queries with existing data. GraphQL was developed internally by Facebook in 2012 Before being publicly released in 2015.

```
query {
  breakingNewses {
    id
    title
  }
  importantNewses {
    id
    title
  }
}
```

GraphQL

Fetching data from GraphQL endpoints can be done... directly using the fetch API

```
const graphqlQuery = `
query {
  breakingNewses {
    id
    title
  }
  importantNewses {
    id
    title
  }
}
`
const options = {
  method: "post",
  headers: {
    "Content-Type": "application/json"
  },
  body: JSON.stringify({
    query: graphqlQuery
  })
};

fetch(`url`, options)
```

A bit more about GraphQL

> Learn about GraphQL, how it works, and how to use it in this series of articles. Looking for documentation on how to build a GraphQL service? There are libraries to help you implement GraphQL in many different languages. For an in-depth learning experience with

> practical tutorials, visit the How to GraphQL fullstack tutorial website. We have also partnered with edX to create a free online course, Exploring GraphQL: A Query Language for APIs.

GraphQL is a query language for your API, and a server-side runtime for executing queries by using a type system you define for your data. GraphQL isn't tied to any specific database or storage engine and is instead backed by your existing code and data.

A GraphQL service is created by defining types and fields on those types, then providing functions for each field on each type. For example, a GraphQL service that tells us who the logged in user is (`me`) as well as that user's name might look something like this:

```
type Query {
  me: User
}type User {
  id: ID
  name: String
}
```

Along with functions for each field on each type:

```
function Query_me(request) {
  return request.auth.user;
}

function User_name(user) {
  return user.getName();
}
```

Once a GraphQL service is running (typically at a URL on a web service), it can be sent GraphQL queries to validate and execute. A received query is first checked to ensure it only refers to the types and fields defined, then runs the provided functions to produce a result.

For example the query:

```
{
  me {
    name
  }
}
```

Could produce the JSON result:

```
{
  "me": {
    "name": "Luke Skywalker"
  }
}
```

What is a Headless CMS (GraphQL CMS, Contentful, Strapi...)

To understand what a headless CMS is, it helps to first look at the traditional content management system and what it was designed to do. Traditional CMSes have been around since the early days of web development. Platforms like Wordpress and Sitecore were designed to store and present content elements like text, images and video on websites.

But digital platforms have evolved, creating the need for more flexible solutions. Now, enterprises are developing websites, mobile sites, apps, digital displays, conversational interfaces and more. Meanwhile, the traditional CMS has failed to keep pace. Why? Because a CMS organizes content in webpage-oriented frameworks, making it impossible for the same content to adapt to other digital platforms.

Headless CMS decouples the content from the presentation

One way to solve this problem is a “headless” CMS — if the presentation layer of a website is the “head” of a CMS, then cutting off that presentation layer creates a headless CMS.

A headless CMS is any type of content management system where the content repository “body” is separated or decoupled from the presentation layer head. Some traditional CMS platforms offer an API that allows you to send content to a separate presentation layer. They call this “headless” because the presentation layer is separated from the body.

While this type of headless CMS enables you to choose an appropriate presentation layer for a digital platform, it doesn’t solve an underlying problem: structuring content so that it can be reused across different platforms and channels.

Why a headless CMS only solves part of the problem

The traditional CMS approach to managing content put everything in one big bucket — content, images, HTML, CSS. This made it impossible to reuse the content because it was commingled with code.

Some CMS platforms offer a “headless” or “decoupled” option through an API that connects the content repository to a separate presentation layer. But the problem of managing the content itself remains. How can content be organized so it’s easy to reuse across digital platforms? How can it be stored separately from code, so the code doesn’t hinder a variety of digital applications?

Content infrastructure solves the problem of organizing content

Content infrastructure is a type of headless content management system, but it doesn’t take the traditional approach of organizing content around pages. Instead, it starts with a content model — a framework for organizing types of content and defining how each type relates to another.

In this type of headless CMS, a content model is custom-built for each organization, so that content creators aren’t stuck with the preprogrammed models offered by a traditional CMS. The content model breaks down content into individual elements, such as a blog post headline or the copy on a call to action button. You can define how each element relates to others, creating a flexible model that can fit any digital container.

Why enterprises are embracing structured content

With the proliferation of digital platforms, enterprises are often crippled by a proliferation of CMS instances — dozens, or even hundreds. As a result, they have to duplicate content from a website CMS to an app CMS and then to a digital display CMS.

Content infrastructure eliminates endless copy and pasting work, enabling organizations to unify all content in a single headless content hub. This makes editing way easier — change the copy or image in one place, and that change applies everywhere the content is located. Unification improves brand consistency and compliance, and enables editors to nimbly update content across all channels, making campaigns a breeze.

Content infrastructure also enables simultaneous collaboration, replacing the slow waterfall approach to development in favor of an agile framework where teams can work in parallel. This type of headless CMS offers a competitive advantage for enterprises that need to rapidly spin up new software, landing pages and microsites.

Lastly, content infrastructure makes reusability a breeze, a key component for making the most of resources spent on content creation. When all content is accessible for use on any digital platform, brands can make the most of features like personalization and localization.

See how headless content infrastructure can work for your digital teams

---

Redux

Redux is a predictable state container for JavaScript apps.

1. Installation

```
npm install –save redux
```

2. Import redux

```
import { createStore } from 'redux'
```

3. Create the reducer

```
function counter(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1 // function 1
    case 'DECREMENT':
      return state - 1 // function 1
    default:
      return state // function 1
  }
}
```

4. Create the store

```
let store = createStore(counter)
```

5. Render the store

```
<Provider store={store}>
  <App />
</Provider>,
```

State management

1. Create the store folder

2. Create a state.js file with the following

```
import React, { createContext, useReducer, useContext } from 'react';

// 1. creation de context
export const StateContext = createContext();

// 2. creation de stateProvider
export const StateProvider = ({ reducer, initialState, children }) => (
  <StateContext.Provider value={useReducer(reducer, initialState)}>
    {children}
  </StateContext.Provider>
);

// 3. creation de useStore
export const useStore = () => useContext(StateContext);
```

State management #2

0. create store/index.js

1. import StateProvider

```
import { StateProvider } from ‘./state’;
import React from ‘react’;
```

2. export useStore

```
export { useStore } from ‘./state’
```

3. create the StoreProvider

```
const StoreProvider = ({ children }) => {

  // 3.1 define initialState
  const initialState = {
    trainings: [],
    title: 'Kuri',
  }

  // 3.2 define the reducer
  const reducer = (state, action) => {
    switch (action.type) {
      case 'setTrainings':
        return {
          ...state,
          trainings: action.payload
        }
      case 'setTitle':
        return {
          ...state,
          title: action.payload
        }
      default:
        return state
    }
  }

  //4. return JSX
  return (
    <StateProvider initialState={initialState} reducer={reducer}>
      {children}
    </StateProvider>
  );
}
export default StoreProvider;
```

State management

1. Add StoreProvider as a root element

```
import StoreProvider from './store'
// ...
<StoreProvider>
  // children
</StoreProvider>
```

2. Create Store and dispatch from useStore

```
import { useStore } from '../store';

const AComponent = () => {
  ...
  const [{ news }, dispatch] = useStore();
  ...
  return ...
    data={news}
  ...
}
```

3. Create Store and dispatch from useStore

```
dispatch({
  type:'toggleExpand'
})
```

```
dispatch({
  type:'reservations',
  payload: []
})

store.reservations
```

Redux with React

From the very beginning, we need to stress that Redux has no relation to React. You can write Redux apps with React, Angular, Ember, jQuery, or vanilla JavaScript.

That said, Redux works especially well with libraries like React and Deku because they let you describe UI as a function of state, and Redux emits state updates in response to actions.

We will use React to build our simple todo app, and cover the basics of how to use React with Redux.

> Note: see the official React-Redux docs at https://react-redux.js.org for a complete guide on how to use Redux and React together.

Installing React Redux

React bindings are not included in Redux by default. You need to install them explicitly:

```
npm install --save react-reduxCopy
```

If you don't use npm, you may grab the latest UMD build from unpkg (either a development or a production build). The UMD build exports a global called `window.ReactRedux` if you add it to your page via a `<script>` tag.

Presentational and Container Components

React bindings for Redux separate *presentational* components from *container* components. This approach can make your app easier to understand and allow you to more easily reuse components. Here's a summary of the differences between presentational and container components (but if you're unfamiliar, we recommend that you also read Dan Abramov's original article describing the concept of presentational and container components):

Copy of Untitled

Property	Presentational Components	Container Components
Purpose	How things look (markup, styles)	How things work (data fetching, state updates)
Aware of Redux	No	Yes
To read data	Read data from props	Subscribe to Redux state
To change data	Invoke callbacks from props	Dispatch Redux actions
Are written	By hand	Usually generated by React Redux

Most of the components we'll write will be presentational, but we'll need to generate a few container components to connect them to the Redux store. This and the design brief below do not imply container components must be near the top of the component tree. If a container component becomes too complex (i.e. it has heavily nested presentational components with countless callbacks being passed down), introduce another container within the component tree as noted in the FAQ.

Technically you could write the container components by hand using `store.subscribe()`. We don't advise you to do this because React Redux makes many performance optimizations that are hard to do by hand. For this reason, rather than write container components, we will generate them using the `connect()` function provided by React Redux, as you will see below.

Designing Component Hierarchy

Remember how we designed the shape of the root state object? It's time we design the UI hierarchy to match it. This is not a Redux-specific task. Thinking in React is a great tutorial that explains the process.

Our design brief is simple. We want to show a list of todo items. On click, a todo item is crossed out as completed. We want to show a field where the user may add a new todo. In the footer, we want to show a toggle to show all, only completed, or only active todos.

Designing Presentational Components

I see the following presentational components and their props emerge from this brief:

- `TodoList` is a list showing visible todos.
  - `todos: Array` is an array of todo items with `{ id, text, completed }` shape.
  - `onTodoClick(id: number)` is a callback to invoke when a todo is clicked.
- `Todo` is a single todo item.
  - `text: string` is the text to show.
  - `completed: boolean` is whether the todo should appear crossed out.
  - `onClick()` is a callback to invoke when the todo is clicked.
- `Link` is a link with a callback.
  - `onClick()` is a callback to invoke when the link is clicked.
- `Footer` is where we let the user change currently visible todos.
- `App` is the root component that renders everything else.

They describe the *look* but don't know *where* the data comes from, or *how* to change it. They only render what's given to them. If you migrate from Redux to something else, you'll be able to keep all these components exactly the same. They have no dependency on Redux.

Designing Container Components

We will also need some container components to connect the presentational components to Redux. For example, the presentational `TodoList` component needs a container like `VisibleTodoList` that subscribes to the Redux store and knows how to apply the current visibility filter. To change the visibility filter, we will provide a `FilterLink` container component that renders a `Link` that dispatches an appropriate action on click:

- `VisibleTodoList` filters the todos according to the current visibility filter and renders a `TodoList`.
- `FilterLink` gets the current visibility filter and renders a `Link`.
  - `filter: string` is the visibility filter it represents.

Designing Other Components

Sometimes it's hard to tell if some component should be a presentational component or a container. For example, sometimes form and function are really coupled together, such as in the case of this tiny component:

- `AddTodo` is an input field with an "Add" button

Technically we could split it into two components but it might be too early at this stage. It's fine to mix presentation and logic in a component that is very small. As it grows, it will be more obvious how to split it, so we'll leave it mixed.

Implementing Components

Let's write the components! We begin with the presentational components so we don't need to think about binding to Redux yet.

Implementing Presentational Components

These are all normal React components, so we won't examine them in detail. We write functional stateless components unless we need to use local state or the lifecycle methods. This doesn't mean that presentational components *have to* be functions—it's just easier to define them this way. If and when you need to add local state, lifecycle methods, or performance optimizations, you can convert them to classes.

`components/Todo.js`

```
import React from 'react'
import PropTypes from 'prop-types'

const Todo = ({ onClick, completed, text }) => (
  <li
    onClick={onClick}
    style={{
      textDecoration: completed ? 'line-through' : 'none'
    }}
  >
    {text}
  </li>
)

Todo.propTypes = {
  onClick: PropTypes.func.isRequired,
  completed: PropTypes.bool.isRequired,
  text: PropTypes.string.isRequired
}

export default Todo
```

`components/TodoList.js`

```
import React from 'react'
import PropTypes from 'prop-types'
import Todo from './Todo'

const TodoList = ({ todos, onTodoClick }) => (
  <ul>
    {todos.map((todo, index) => (
      <Todo key={index} {...todo} onClick={() => onTodoClick(index)} />
    ))}
  </ul>
)

TodoList.propTypes = {
  todos: PropTypes.arrayOf(
    PropTypes.shape({
      id: PropTypes.number.isRequired,
      completed: PropTypes.bool.isRequired,
      text: PropTypes.string.isRequired
    }).isRequired
  ).isRequired,
  onTodoClick: PropTypes.func.isRequired
}

export default TodoList
```

`components/Link.js`

```
import React from 'react'
import PropTypes from 'prop-types'

const Link = ({ active, children, onClick }) => {
  if (active) {
    return <span>{children}</span>
  }

  return (
    <a
      href=""
      onClick={e => {
        e.preventDefault()
        onClick()
      }}
    >
      {children}
    </a>
  )
}

Link.propTypes = {
  active: PropTypes.bool.isRequired,
  children: PropTypes.node.isRequired,
  onClick: PropTypes.func.isRequired
}

export default Link
```

`components/Footer.js`

```
import React from 'react'
import FilterLink from '../containers/FilterLink'
import { VisibilityFilters } from '../actions'

const Footer = () => (
  <p>
    Show: <FilterLink filter={VisibilityFilters.SHOW_ALL}>All</FilterLink>
    {', '}
    <FilterLink filter={VisibilityFilters.SHOW_ACTIVE}>Active</FilterLink>
    {', '}
    <FilterLink filter={VisibilityFilters.SHOW_COMPLETED}>Completed</FilterLink>
  </p>
)

export default Footer
```

Implementing Container Components

Now it's time to hook up those presentational components to Redux by creating some containers. Technically, a container component is just a React component that uses `store.subscribe()` to read a part of the Redux state tree and supply props to a presentational component it renders. You could write a container component by hand, but we suggest instead generating container components with the React Redux library's `connect()` function, which provides many useful optimizations to prevent unnecessary re-renders. (One result of this is that you shouldn't have to worry about the React performance suggestion of implementing `shouldComponentUpdate` yourself.)

To use `connect()`, you need to define a special function called `mapStateToProps` that describes how to transform the current Redux store state into the props you want to pass to a presentational component you are wrapping. For example, `VisibleTodoList` needs to calculate `todos` to pass to the `TodoList`, so we define a function that filters the `state.todos` according to the `state.visibilityFilter`, and use it in its `mapStateToProps`:

```
const getVisibleTodos = (todos, filter) => {
  switch (filter) {
    case 'SHOW_COMPLETED':
      return todos.filter(t => t.completed)
    case 'SHOW_ACTIVE':
      return todos.filter(t => !t.completed)
    case 'SHOW_ALL':
    default:
      return todos
  }
}

const mapStateToProps = state => {
  return {
    todos: getVisibleTodos(state.todos, state.visibilityFilter)
  }
}
```

In addition to reading the state, container components can dispatch actions. In a similar fashion, you can define a function called `mapDispatchToProps()` that receives the `dispatch()` method and returns callback props that you want to inject into the presentational component. For example, we want the `VisibleTodoList` to inject a prop called `onTodoClick` into the `TodoList` component, and we want `onTodoClick` to dispatch a `TOGGLE_TODO` action:

```
const mapDispatchToProps = dispatch => {
  return {
    onTodoClick: id => {
      dispatch(toggleTodo(id))
    }
  }
}
```

Finally, we create the `VisibleTodoList` by calling `connect()` and passing these two functions:

```
import { connect } from 'react-redux'

const VisibleTodoList = connect(
  mapStateToProps,
  mapDispatchToProps
)(TodoList)
```

```
export default VisibleTodoList
```

These are the basics of the React Redux API, but there are a few shortcuts and power options so we encourage you to check out its documentation in detail. In case you are worried about `mapStateToProps` creating new objects too often, you might want to learn about computing derived data with reselect.

Find the rest of the container components defined below:

`containers/FilterLink.js`

```
import { connect } from 'react-redux'
import { setVisibilityFilter } from '../actions'
import Link from '../components/Link'

const mapStateToProps = (state, ownProps) => {
  return {
    active: ownProps.filter === state.visibilityFilter
  }
}

const mapDispatchToProps = (dispatch, ownProps) => {
  return {
    onClick: () => {
      dispatch(setVisibilityFilter(ownProps.filter))
    }
  }
}

const FilterLink = connect(
  mapStateToProps,
  mapDispatchToProps
)(Link)

export default FilterLink
```

`containers/VisibleTodoList.js`

```
import { connect } from 'react-redux'
import { toggleTodo } from '../actions'
import TodoList from '../components/TodoList'

const getVisibleTodos = (todos, filter) => {
  switch (filter) {
    case 'SHOW_ALL':
      return todos
    case 'SHOW_COMPLETED':
      return todos.filter(t => t.completed)
    case 'SHOW_ACTIVE':
      return todos.filter(t => !t.completed)
  }
}

const mapStateToProps = state => {
  return {
    todos: getVisibleTodos(state.todos, state.visibilityFilter)
  }
}

const mapDispatchToProps = dispatch => {
  return {
    onTodoClick: id => {
      dispatch(toggleTodo(id))
    }
  }
}

const VisibleTodoList = connect(
  mapStateToProps,
  mapDispatchToProps
)(TodoList)

export default VisibleTodoList
```

Implementing Other Components

`containers/AddTodo.js`

Recall as mentioned previously, both the presentation and logic for the `AddTodo` component are mixed into a single definition.

```
import React from 'react'
import { connect } from 'react-redux'
import { addTodo } from '../actions'

let AddTodo = ({ dispatch }) => {
  let input

  return (
    <div>
      <form
        onSubmit={e => {
          e.preventDefault()
          if (!input.value.trim()) {
            return
          }
          dispatch(addTodo(input.value))
          input.value = ''
        }}
      >
        <input
          ref={node => {
            input = node
          }}
        />
        <button type="submit">Add Todo</button>
      </form>
    </div>
  )
}
AddTodo = connect()(AddTodo)

export default AddTodo
```

If you are unfamiliar with the `ref` attribute, please read this documentation to familiarize yourself with the recommended use of this attribute.

Tying the containers together within a component

`components/App.js`

```
import React from 'react'
import Footer from './Footer'
import AddTodo from '../containers/AddTodo'
import VisibleTodoList from '../containers/VisibleTodoList'

const App = () => (
  <div>
    <AddTodo />
    <VisibleTodoList />
    <Footer />
  </div>
)

export default App
```

Passing the Store

All container components need access to the Redux store so they can subscribe to it. One option would be to pass it as a prop to every container component. However it gets tedious, as you have to wire `store` even through presentational components just because they happen to render a container deep in the component tree.

The option we recommend is to use a special React Redux component called `<Provider>` to magically make the store available to all container components in the application without passing it explicitly. You only need to use it once when you render the root component:

`index.js`

```
import React from 'react'
import { render } from 'react-dom'
import { Provider } from 'react-redux'
import { createStore } from 'redux'
```

```
import todoApp from './reducers'
import App from './components/App'

const store = createStore(todoApp)

render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
)
```

Extras: imports and exports

```
// file1
export AFunction
export CFunction
export DObject
export { AFunction, CFunction, DObject }
export default BFunction

// file2
import BFunction, { Afunction, CFunction, DObject } from './file1'
import BFunction from './file1'

// file3
export { AFunction } from './file1'
```

Extras: destructuring usecases

```
const AComponent = ({aParam, bParam}) => {
  return <Text>{aParam} {bParam}</Text>
}
```

Deep Destructuring
```JSX
const AScreen = ({navigation: { navigate, getParam }}) => {
  ...
  navigate('ARoute')
  ...
  return <View> {/* any views */} </View>
}
```

Extras: preferable project structure

```
--|screens (here we create a file per UI screen)
  |--home.js
  |--details.js
  |--contact.js
  |...
--|elements (put the extracted UI components)
  |--articles.js
  |--footer.js
  |...
--|network (group networks calls per type)
```

```
  |--articles.js
  |--news.js
  |...
--|store (state management store)
  |--state.js
  |--index.js
--|utils (additional features)
  |--camera.js
  |--share.js
  |--validation.js
  |...
```

UI kits

A set of UI components that helps building applications UIs

1. `Native Base` (https://nativebase.io/)
2. `React Native Elements` (https://react-native-training.github.io/react-native-elements/)
3. `Shoutem` (https://shoutem.github.io/ui/)
4. `UI Kitten` (https://akveo.github.io/react-native-ui-kitten/)
5. `React Native Material UI` (https://github.com/xotahal/react-native-material-ui)

NativeBase

An open source framework to build React Native apps over a single JavaScript codebase for Android and iOS.

1. installation of NativeBase > npm i –save native-base
2. install expo fonts > expo install expo-font
3. load fonts (within useEffect of the root element ex: App.js)

```
import * as Font from 'expo-font';
import { Ionicons } from '@expo/vector-icons';

useEffect(async () => {
  await Font.loadAsync({
    Roboto: require('native-base/Fonts/Roboto.ttf'),
    Roboto_medium: require('native-base/Fonts/Roboto_medium.ttf'),
    ...Ionicons.font,
  })
}, [])
```

NativeBase

Using NativeBase's components is almost identical to `react-native` components

1. import the ui components

```
import {
  Container,
  Header,
  Title,
  // ...
} from 'native-base'
```

The most used components are: Container, Header, Title, Body, Icon, Text, Button, Card, Form, Input...

NativeBase

With `native-base` we use `Container` instead of `View`

1. Header

```
<Header>
  <Left>
    <Button transparent>
      <Icon name='menu' />
    </Button>
  </Left>
  <Body>
    <Title>Header</Title>
  </Body>
  <Right />
</Header>
```

2. Button

There's a wide rang of options to create standard based buttons

[full, block, bordered, primary, success, info, warning, dark...]

```
<Button full>
  <Text>Click Me!</Text>
</Button>
```

3. Card

```
<Card>
  <CardItem>
    <Body>
      <Text>
          //Your text here
      </Text>
    </Body>
  </CardItem>
</Card>
```

4. Fab

```
<Fab
  style={{ backgroundColor: '#5067FF' }}
  position="bottomRight"
  onPress={() => { }}>
  <Icon name="home" />
</Fab>
```

5. Form

```
<Form>
  <Item>
    <Input placeholder="Username" />
  </Item>
  <Item last>
    <Input placeholder="Password" />
  </Item>
</Form>
```

6. List

```
<List>
  <ListItem>
    <Text>Simon Mignolet</Text>
  </ListItem>
  <ListItem>
    <Text>Nathaniel Clyne</Text>
  </ListItem>
  <ListItem icon>
    <Left>
      <Button style={{ backgroundColor: "#FF9501" }}>
        <Icon active name="airplane" />
      </Button>
    </Left>
    <Body>
      <Text>Airplane Mode</Text>
    </Body>
    <Right>
      <Switch value={false} />
    </Right>
  </ListItem>
</List>
```

Layouts with NativeBase

To compose layouts we can use `Cols` and `Rows` instead of using Views with flex

```
import { Col, Row } from 'react-native-easy-grid';
// flexible heights
<Row>
  <Col style={{ backgroundColor: '#635DB7' }}></Col>
  <Col>
    <Col style={{ backgroundColor: '#635DB7' }}></Col>
    <Col style={{ backgroundColor: '#00CE9F' }}></Col>
  </Col>
</Row>
// fixed heights
<Row>
  <Col style={{ backgroundColor: '#635DB7', height: 200 }}></Col>
  <Col style={{ backgroundColor: '#00CE9F', height: 200 }}></Col>
</Row>
```

Native modules

Here is the expo way to integrate native code (camera, network, browser...)

(https://docs.expo.io/versions/latest/bare/unimodules-full-list/)

Audio

(https://docs.expo.io/versions/latest/sdk/audio/)

1. install the Audio SDK > expo install expo-av

2. create the audioObject

```
import { Audio } from 'expo-av';
const soundObject = new Audio.Sound();
```

3. create a play Function

```
const play = async () => {
  try {
    // 2.1 load local audio file from assets
    await soundObject.loadAsync(require('../assets/sample.mp3'));
    // 2.2 play audio
    await soundObject.playAsync();
  } catch (error) {
    // 2.3 handle errors
  }
}
```

4. create a stop Function

```
const stop = async () => {
  try {
    // 3.1 stop audio
    await soundObject.stopAsync();
    // 3.2 unload audio file
    soundObject.unloadAsync();
  } catch (error) {
    // 3.3 handle errors
  }
}
```

Network

(https://docs.expo.io/versions/latest/sdk/network/)

1. install the Network SDK

```
expo install expo-av
```

2. import the Network SDK

```
import * as Network from 'expo-network';
```

3. get network's status

```
const networkStatus = await Network.getNetworkStateAsync();
// {
//   type: NetworkStateType.CELLULAR,
//   isConnected: true,
//   isInternetReachable: true,
// }
```

1. get the ip address

```
const ipAddress = await Network.getIpAddressAsync();
// "92.168.32.44"
```

1. get the mac address

```
const macAddress = await Network.getMacAddressAsync();
// "E5:12:D8:E5:69:97"
```

1. detect airplain-mode status

```
const isAirplaneModeEnabled = await Network.isAirplaneModeEnabledAsync();
// false
```

FileSystem

(https://docs.expo.io/versions/latest/sdk/filesystem/)

1. install the Sharing SDK

```
expo install expo-file-system
```

2. import the Sharing SDK

```
import * as FileSystem from 'expo-file-system';
```

3. start sharing (uri is a link to a file)

```
const image_source = 'https://images.unsplash.com/photo-1508138221679-760a23a2285b?ixlib=rb-1.2.1&ixid=eyJhcHBfaWQiOjEyMDd9&w=1
const { uri } = await FileSystem.downloadAsync(
  image_source,
  FileSystem.documentDirectory + '.jpeg'
)
```

Sharing

(https://docs.expo.io/versions/latest/sdk/sharing/)

1. install the Sharing SDK

```
expo install expo-sharing
```

2. import the Sharing SDK

```
import * as Sharing from 'expo-sharing';
```

3. start sharing (uri is a link to a file)

```
Sharing.shareAsync(uri)
```

Animations

Animations are very important to create a great user experience. Stationary objects must overcome inertia as they start moving. Objects in motion have momentum and rarely come to a stop immediately. Animations allow you to convey physically believable motion in your interface.

React Native provides two complementary animation systems: `Animated` for granular and interactive control of specific values, and `LayoutAnimation` for animated global layout transactions.

`Animated` API

The `Animated` API is designed to concisely express a wide variety of interesting animation and interaction patterns in a very performant way. `Animated` focuses on declarative relationships between inputs and outputs, with configurable transforms in between, and `start` / `stop` methods to control time-based animation execution.

`Animated` exports six animatable component types: `View`, `Text`, `Image`, `ScrollView`, `FlatList` and `SectionList`, but you can also create your own using `Animated.createAnimatedComponent()`.

For example, a container view that fades in when it is mounted may look like this:

Let's break down what's happening here. In the `FadeInView` constructor, a new `Animated.Value` called `fadeAnim` is initialized as part of `state`. The opacity property on the `View` is mapped to this animated value. Behind the scenes, the numeric value is extracted and used to set opacity.

When the component mounts, the opacity is set to 0. Then, an easing animation is started on the `fadeAnim` animated value, which will update all of its dependent mappings (in this case, only the opacity) on each frame as the value animates to the final value of 1.

This is done in an optimized way that is faster than calling `setState` and re-rendering. Because the entire configuration is declarative, we will be able to implement further optimizations that serialize the configuration and runs the animation on a high-priority thread.

Configuring animations

Animations are heavily configurable. Custom and predefined easing functions, delays, durations, decay factors, spring constants, and more can all be tweaked depending on the type of animation.

`Animated` provides several animation types, the most commonly used one being `Animated.timing()`. It supports animating a value over time using one of various predefined easing functions, or you can use your own. Easing functions are typically used in animation to convey gradual acceleration and deceleration of objects.

By default, `timing` will use a easeInOut curve that conveys gradual acceleration to full speed and concludes by gradually decelerating to a stop. You can specify a different easing function by passing a `easing` parameter. Custom `duration` or even a `delay` before the animation starts is also supported.

For example, if we want to create a 2-second long animation of an object that slightly backs up before moving to its final position:

```
Animated.timing(this.state.xPosition, {
  toValue: 100,
  easing: Easing.back(),
  duration: 2000,
}).start();
```

Take a look at the Configuring animations section of the `Animated` API reference to learn more about all the config parameters supported by the built-in animations.

Composing animations

Animations can be combined and played in sequence or in parallel. Sequential animations can play immediately after the previous animation has finished, or they can start after a specified delay. The `Animated` API provides several methods, such as `sequence()` and `delay()`, each of which take an array of animations to execute and automatically calls `start()` / `stop()` as needed.

For example, the following animation coasts to a stop, then it springs back while twirling in parallel:

```
Animated.sequence([
  // decay, then spring to start and twirl
  Animated.decay(position, {
    // coast to a stop
    velocity: {x: gestureState.vx, y: gestureState.vy}, // velocity from gesture release
    deceleration: 0.997,
  }),
  Animated.parallel([
    // after decay, in parallel:
    Animated.spring(position, {
      toValue: {x: 0, y: 0}, // return to start
    }),
    Animated.timing(twirl, {
      // and twirl
```

```
      toValue: 360,
    }),
  ]),
]).start(); // start the sequence group
```

If one animation is stopped or interrupted, then all other animations in the group are also stopped. `Animated.parallel` has a `stopTogether` option that can be set to `false` to disable this.

You can find a full list of composition methods in the Composing animations section of the `Animated` API reference.

Combining animated values

You can combine two animated values via addition, multiplication, division, or modulo to make a new animated value.

There are some cases where an animated value needs to invert another animated value for calculation. An example is inverting a scale (2x → 0.5x):

```
const a = new Animated.Value(1);
const b = Animated.divide(1, a);

Animated.spring(a, {
  toValue: 2,
}).start();
```

Interpolation

Each property can be run through an interpolation first. An interpolation maps input ranges to output ranges, typically using a linear interpolation but also supports easing functions. By default, it will extrapolate the curve beyond the ranges given, but you can also have it clamp the output value.

A mapping to convert a 0-1 range to a 0-100 range would be:

```
value.interpolate({
  inputRange: [0, 1],
  outputRange: [0, 100],
});
```

For example, you may want to think about your `Animated.Value` as going from 0 to 1, but animate the position from 150px to 0px and the opacity from 0 to 1. This can be done by modifying `style` from the example above like so:

```
style={{
    opacity: this.state.fadeAnim, // Binds directly
    transform: [{
      translateY: this.state.fadeAnim.interpolate({
        inputRange: [0, 1],
        outputRange: [150, 0]  // 0 : 150, 0.5 : 75, 1 : 0
      }),
    }],
  }}
```

`interpolate()` supports multiple range segments as well, which is handy for defining dead zones and other handy tricks. For example, to get a negation relationship at -300 that goes to 0 at -100, then back up to 1 at 0, and then back down to zero at 100 followed by a dead-zone that remains at 0 for everything beyond that, you could do:

```
value.interpolate({
  inputRange: [-300, -100, 0, 100, 101],
  outputRange: [300, 0, 1, 0, 0],
});
```

Which would map like so:

```
Input | Output
------|-------
  -400|    450
  -300|    300
  -200|    150
  -100|      0
   -50|    0.5
```

```
   0|    1
  50|  0.5
 100|    0
 101|    0
 200|    0
```

`interpolate()` also supports mapping to strings, allowing you to animate colors as well as values with units. For example, if you wanted to animate a rotation you could do:

```
value.interpolate({
  inputRange: [0, 360],
  outputRange: ['0deg', '360deg'],
});
```

`interpolate()` also supports arbitrary easing functions, many of which are already implemented in the `Easing` module. `interpolate()` also has configurable behavior for extrapolating the `outputRange`. You can set the extrapolation by setting the `extrapolate`, `extrapolateLeft`, or `extrapolateRight` options. The default value is `extend` but you can use `clamp` to prevent the output value from exceeding `outputRange`.

Tracking dynamic values

Animated values can also track other values. Set the `toValue` of an animation to another animated value instead of a plain number. For example, a "Chat Heads" animation like the one used by Messenger on Android could be implemented with a `spring()` pinned on another animated value, or with `timing()` and a `duration` of 0 for rigid tracking. They can also be composed with interpolations:

```
Animated.spring(follower, {toValue: leader}).start();
Animated.timing(opacity, {
  toValue: pan.x.interpolate({
    inputRange: [0, 300],
    outputRange: [1, 0],
  }),
}).start();
```

The `leader` and `follower` animated values would be implemented using `Animated.ValueXY()`. `ValueXY` is a handy way to deal with 2D interactions, such as panning or dragging. It is a wrapper that contains two `Animated.Value` instances and some helper functions that call through to them, making `ValueXY` a drop-in replacement for `Value` in many cases. It allows us to track both x and y values in the example above.

Tracking gestures

Gestures, like panning or scrolling, and other events can map directly to animated values using `Animated.event`. This is done with a structured map syntax so that values can be extracted from complex event objects. The first level is an array to allow mapping across multiple args, and that array contains nested objects.

For example, when working with horizontal scrolling gestures, you would do the following in order to map `event.nativeEvent.contentOffset.x` to `scrollX` (an `Animated.Value`):

```
onScroll={Animated.event(
   // scrollX = e.nativeEvent.contentOffset.x
   [{ nativeEvent: {
        contentOffset: {
          x: scrollX
        }
      }
    }]
 )}
```

When using `PanResponder`, you could use the following code to extract the x and y positions from `gestureState.dx` and `gestureState.dy`. We use a `null` in the first position of the array, as we are only interested in the second argument passed to the `PanResponder` handler, which is the `gestureState`.

```
onPanResponderMove={Animated.event(
  [null, // ignore the native event
  // extract dx and dy from gestureState
  // like 'pan.x = gestureState.dx, pan.y = gestureState.dy'
```

```
  {dx: pan.x, dy: pan.y}
])}
```

Responding to the current animation value

You may notice that there is no clear way to read the current value while animating. This is because the value may only be known in the native runtime due to optimizations. If you need to run JavaScript in response to the current value, there are two approaches:

- `spring.stopAnimation(callback)` will stop the animation and invoke `callback` with the final value. This is useful when making gesture transitions.
- `spring.addListener(callback)` will invoke `callback` asynchronously while the animation is running, providing a recent value. This is useful for triggering state changes, for example snapping a bobble to a new option as the user drags it closer, because these larger state changes are less sensitive to a few frames of lag compared to continuous gestures like panning which need to run at 60 fps.

`Animated` is designed to be fully serializable so that animations can be run in a high performance way, independent of the normal JavaScript event loop. This does influence the API, so keep that in mind when it seems a little trickier to do something compared to a fully synchronous system. Check out `Animated.Value.addListener` as a way to work around some of these limitations, but use it sparingly since it might have performance implications in the future.

Using the native driver

The `Animated` API is designed to be serializable. By using the native driver, we send everything about the animation to native before starting the animation, allowing native code to perform the animation on the UI thread without having to go through the bridge on every frame. Once the animation has started, the JS thread can be blocked without affecting the animation.

To use the native driver for normal animations add `useNativeDriver: true` to the animation config when starting it.

```
Animated.timing(this.state.animatedValue, {
  toValue: 1,
  duration: 500,
  useNativeDriver: true, // <-- Add this
}).start();
```

Animated values are only compatible with one driver so if you use native driver when starting an animation on a value, make sure every animation on that value also uses the native driver.

The native driver also works with `Animated.event`. This is especially useful for animations that follow the scroll position as without the native driver, the animation will always run a frame behind the gesture due to the async nature of React Native.

```
<Animated.ScrollView // <-- Use the Animated ScrollView wrapper
  scrollEventThrottle={1} // <-- Use 1 here to make sure no events are ever missed
  onScroll={Animated.event(
    [
      {
        nativeEvent: {
          contentOffset: {y: this.state.animatedValue},
        },
      },
    ],
    {useNativeDriver: true}, // <-- Add this
  )}>
  {content}
</Animated.ScrollView>
```

You can see the native driver in action by running the RNTester app, then loading the Native Animated Example. You can also take a look at the source code to learn how these examples were produced.

Caveats

Not everything you can do with `Animated` is currently supported by the native driver. The main limitation is that you can only animate non-layout properties: things like `transform` and `opacity` will work, but flexbox and position properties will not. When using `Animated.event`, it will only work with direct events and not bubbling events. This means it does not work with `PanResponder` but does work with things like `ScrollView#onScroll`.

When an animation is running, it can prevent `VirtualizedList` components from rendering more rows. If you need to run a long or looping animation while the user is scrolling through a list, you can use `isInteraction: false` in your animation's config to prevent this issue.

Bear in mind

While using transform styles such as `rotateY`, `rotateX`, and others ensure the transform style `perspective` is in place. At this time some animations may not render on Android without it. Example below.

```
<Animated.Viewstyle={{
    transform: [
      {scale: this.state.scale},
      {rotateY: this.state.rotateY},
      {perspective: 1000}, // without this line this Animation will not render on Android while working fine on iOS
    ],
  }}/>
```

Additional examples

The RNTester app has various examples of `Animated` in use:

- AnimatedGratuitousApp
- NativeAnimationsExample

`LayoutAnimation` API

`LayoutAnimation` allows you to globally configure `create` and `update` animations that will be used for all views in the next render/layout cycle. This is useful for doing flexbox layout updates without bothering to measure or calculate specific properties in order to animate them directly, and is especially useful when layout changes may affect ancestors, for example a "see more" expansion that also increases the size of the parent and pushes down the row below which would otherwise require explicit coordination between the components in order to animate them all in sync.

Note that although `LayoutAnimation` is very powerful and can be quite useful, it provides much less control than `Animated` and other animation libraries, so you may need to use another approach if you can't get `LayoutAnimation` to do what you want.

Note that in order to get this to work on **Android** you need to set the following flags via `UIManager`:

```
UIManager.setLayoutAnimationEnabledExperimental &&
  UIManager.setLayoutAnimationEnabledExperimental(true);
```

This example uses a preset value, you can customize the animations as you need, see LayoutAnimation.js for more information.

Additional notes

`requestAnimationFrame`

`requestAnimationFrame` is a polyfill from the browser that you might be familiar with. It accepts a function as its only argument and calls that function before the next repaint. It is an essential building block for animations that underlies all of the JavaScript-based animation APIs. In general, you shouldn't need to call this yourself - the animation APIs will manage frame updates for you.

`setNativeProps`

As mentioned in the Direct Manipulation section, `setNativeProps` allows us to modify properties of native-backed components (components that are actually backed by native views, unlike composite components) directly, without having to `setState` and re-render the component hierarchy.

We could use this in the Rebound example to update the scale - this might be helpful if the component that we are updating is deeply nested and hasn't been optimized with `shouldComponentUpdate`.

If you find your animations with dropping frames (performing below 60 frames per second), look into using `setNativeProps` or `shouldComponentUpdate` to optimize them. Or you could run the animations on the UI thread rather than the JavaScript thread with the useNativeDriver option. You may also want to defer any computationally intensive

work until after animations are complete, using the InteractionManager. You can monitor the frame rate by using the In-App Developer Menu “FPS Monitor” tool.

Extras: VSCode theming and extensions

Themes: [Cobalt2 Theme Official, **Night Owl** or Night Owl Light]

Font-families: [**Dank Mono** or Fira Code] with "editor.fontLigatures": true

Extensions:

```
2gua.rainbow-brackets-0.0.6                    mechatroner.rainbow-csv-1.3.1
aaron-bond.better-comments-2.0.5               mikestead.dotenv-1.0.1
asvetliakov.snapshot-tools-0.4.0               mrmlnc.vscode-scss-0.8.1
attilabuti.vscode-mjml-1.6.0                   ms-azuretools.vscode-azureappservice-0.16.2
be5invis.vscode-custom-css-3.0.9               ms-vscode.azure-account-0.8.7
britesnow.vscode-toggle-quotes-0.3.2           ms-vscode.cpptools-0.26.2
christian-kohler.npm-intellisense-1.3.0        msjsdiag.debugger-for-chrome-4.12.3
christian-kohler.path-intellisense-1.4.2       naumovs.color-highlight-2.3.0
cmstead.jsrefactor-2.20.5                      orta.vscode-jest-3.0.2
coenraads.bracket-pair-colorizer-1.0.61        pflannery.vscode-versionlens-0.24.0
dart-code.dart-code-3.7.0                      pnp.polacode-0.3.4
dart-code.flutter-3.7.0                        prisma.vscode-graphql-0.2.2
dbaeumer.vscode-eslint-1.9.1                   ritwickdey.liveserver-5.6.1
digitalstreamio.mjml-syntax-0.0.1              saviorisdead.theme-githubcleanwhite-0.0.3
donjayamanne.githistory-0.4.11                 sdras.night-owl-1.1.3
dsznajder.es7-react-js-snippets-2.4.4          siegebell.scope-info-0.2.0
eamodio.gitlens-10.2.0                         silvenon.mdx-0.1.0
file-icons.file-icons-1.0.19                   sleistner.vscode-fileutils-3.0.0
formulahendry.auto-close-tag-0.5.6             streetsidesoftware.code-spell-checker-1.7.20
formulahendry.auto-rename-tag-0.1.1            sysoev.language-stylus-1.11.0
foxundermoon.next-js-0.0.2                     techer.open-in-browser-2.0.0
ikuyadeu.r-1.1.8                               tyriar.lorem-ipsum-1.2.0
jpoissonnier.vscode-styled-components-0.0.27   wakatime.vscode-wakatime-2.2.1
k--kato.intellij-idea-keybindings-0.2.37       wesbos.theme-cobalt2-2.1.6
kkapsner.duplicateselection-1.0.1              wix.vscode-import-cost-2.12.0
kumar-harsh.graphql-for-vscode-1.15.3          zhouronghui.propertylist-0.0.2
```

Extras: Resources

React Native Official Documentation (https://facebook.github.io/react-native/docs/getting-started)

React Official Documentation (https://reactjs.org/docs/getting-started.html)

Expo Documentation (https://docs.expo.io/versions/latest/)

NativeBase Documentation (https://docs.nativebase.io/)

Awesome React (https://github.com/enaqx/awesome-react)

Awesome React Native (https://github.com/jondot/awesome-react-native)

GraphQL (https://graphql.org/learn/)

Some good solutions for building GraphQL based backend are: [

- GraphCMS (https://graphcms.com/)
- Contentful (https://www.contentful.com/)
- Appollo (NodeJS) (https://apollographql.com/docs/apollo-server/)

]