sitepoint

REACT:
TOOLS & SKILLS

2ND EDITION

THE SITEPOINT REACT SERIES

React: Tools & Skills, 2nd Edition

- **Author:** Manjunath M, Michael Wanyoike, Camilo Reyes, JavaScript Joe, Pavels Jelisejev, Wern Ancheta, Nilson Jacques, Craig Buckler
- **Cover Design:** Alex Walker

Notice of Rights

Notice of Liability

Trademark Notice

Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood
VIC Australia 3066
Web: www.sitepoint.com
Email: books@sitepoint.com

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit sitepoint.com to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, Ruby, mobile development, design, and more.

Chapter

1

React Router v5: The Complete Guide

Manjunath M, Michael Wanyoike

React Router is the de facto standard routing library for React. When you need to navigate through a React application with multiple views, you'll need a router to manage the URLs. React Router takes care of that, keeping your application UI and the URL in sync.

This tutorial introduces you to React Router v5 and a whole lot of things you can do with it.

Introduction

React is a popular library for creating single-page applications (SPAs) that are rendered on the client side. An SPA might have multiple **views** (aka **pages**), and unlike conventional multi-page apps, navigating through these views shouldn't result in the entire page being reloaded. Instead, we want the views to be rendered inline within the current page. The end user, who's accustomed to multi-page apps, expects the following features to be present in an SPA:

- Each view should have a URL that uniquely specifies that view. This is so that the user can bookmark the URL for reference at a later time. For example, `www.example.com/products`.
- The browser's back and forward button should work as expected.
- Dynamically generated nested views should preferably have a URL of their own too—such as `example.com/products/shoes/101`, where 101 is the product ID.

Routing is the process of keeping the browser URL in sync with what's being rendered on the page. React Router lets you handle routing *declaratively*. The declarative routing approach allows you to control the data flow in your application, by saying "the route should look like this":

```
<Route path="/about">
  <About />
</Route>
```

You can place your `<Route>` component anywhere you want your route to be rendered. Since `<Route>`, `<Link>` and all the other React Router APIs that we'll be dealing with are just components, you can easily get up and running with routing in React.

Not from Facebook

There's a common misconception that React Router is an official routing solution developed by Facebook. In reality, it's a third-party library that's widely popular for its design and simplicity.

Overview

This tutorial is divided into different sections. First, we'll set up React and React Router using npm. Then we'll jump right into some React Router basics. You'll find different code demonstrations of React Router in action. The examples covered in this tutorial include:

- basic navigational routing
- nested routing
- nested routing with path parameters
- protected routing

All the concepts connected with building these routes will be discussed along the way.

The entire code for the project is available on this GitHub repo.

Let's get started!

Setting up React Router

To follow along with this tutorial, you'll need a recent version of Node installed on your PC. If this isn't the case, then head over to the Node home page and download the correct binaries for your system. Alternatively, you might consider using a version manager to install Node. We have a tutorial on using a version manager here.

Node comes bundled with npm, a package manager for JavaScript, with which we're going to install some of the libraries we'll be using. You can learn more about using npm here.

You can check that both are installed correctly by issuing the following commands from the command line:

```
node -v
> 12.19.0

npm -v
> 6.14.8
```

With that done, let's start off by creating a new React project with the Create React App tool. You can either install this globally, or use `npx`, like so:

```
npx create-react-app react-router-demo
```

When this has finished, change into the newly created directory:

```
cd react-router-demo
```

The React Router library comprises three packages: react-router, react-router-dom, and react-router-native. The core package for the router is `react-router`, whereas the other two are environment specific. You should use `react-router-dom` if you're building a website, and `react-router-native` if you're in a mobile app development environment using React Native.

Use npm to install `react-router-dom`:

```
npm install react-router-dom
```

Then start the development server with this:

```
npm run start
```

Congratulations! You now have a working React app with React Router installed. You can view the app running at http://localhost:3000/.

React Router Basics

Now let's familiarize ourselves with a basic React Router setup. To do this, we'll make an app with three separate views: Home, Category and Products.

The `Router` Component

The first thing we'll need to do is to wrap our `<App>` component in a `<Router>` component (provided by React Router). Since we're building a browser-based application, we can use two types of router from the React Router API:

- BrowserRouter
- HashRouter

The primary difference between them is evident in the URLs they create:

```
// <BrowserRouter>
http://example.com/about
```

```
// <HashRouter>
http://example.com/#/about
```

The `<BrowserRouter>` is the more popular of the two because it uses the HTML5 History API to keep your UI in sync with the URL, whereas the `<HashRouter>` uses the hash portion of the URL (`window.location.hash`). If you need to support legacy browsers that don't support the History API, you should use `<HashRouter>` . Otherwise `<BrowserRouter>` is the better choice for most use cases. You can read more about the differences here.

So, let's import the `BrowserRouter` component and wrap it around the `App` component:

```
// src/index.js

import React from "react";
import ReactDOM from "react-dom";
import App from "./App";
import { BrowserRouter } from "react-router-dom";

ReactDOM.render(
  <BrowserRouter>
    <App />
  </BrowserRouter>,
  document.getElementById("root")
);
```

The above code creates a `history` instance for our entire `<App>` component. Let's look at what that means.

A Little Bit of History

> *The `history` library lets you easily manage session history anywhere JavaScript runs. A `history` object abstracts away the differences in various environments and provides a minimal API that lets you manage the history stack, navigate, and persist state between sessions. — React Training docs*

Each `<Router>` component creates a `history` object that keeps track of the current location (`history.location`) and also the previous locations in a stack. When the current location changes, the view is re-rendered and you get a sense of navigation. How does the current location change? The history object has methods such as `history.push` and `history.replace` to take care of that. The `history.push` method is invoked when you click on a `<Link>`

component, and `history.replace` is called when you use a `<Redirect>`. Other methods—such as `history.goBack` and `history.goForward`—are used to navigate through the history stack by going back or forward a page.

Moving on, we have Links and Routes.

`Link` and `Route` Components

The `<Route>` component is the most important component in React Router. It renders some UI if the current location matches the route's path. Ideally, a `<Route>` component should have a prop named `path`, and if the path name matches the current location, it gets rendered.

The `<Link>` component, on the other hand, is used to navigate between pages. It's comparable to the HTML anchor element. However, using anchor links would result in a full page refresh, which we don't want. So instead, we can use `<Link>` to navigate to a particular URL and have the view re-rendered without a refresh.

Now we've covered everything you need to make our app work. Update `src/App.js` as follows:

```
import React from "react";
import { Link, Route, Switch } from "react-router-dom";

const Home = () => (
  <div>
    <h2>Home</h2>
  </div>
);

const Category = () => (
  <div>
    <h2>Category</h2>
  </div>
);

const Products = () => (
  <div>
    <h2>Products</h2>
  </div>
);

export default function App() {
  return (
    <div>
      <nav className="navbar navbar-light">
```

```
        <ul className="nav navbar-nav">
          <li>
            <Link to="/">Home</Link>
          </li>
          <li>
            <Link to="/category">Category</Link>
          </li>
          <li>
            <Link to="/products">Products</Link>
          </li>
        </ul>
      </nav>

      { /* Route components are rendered if the path prop matches the current URL */}
      <Route path="/"><Home /></Route>
      <Route path="/category"><Category /></Route>
      <Route path="/products"><Products /></Route>
    </div>
  );
}
```

Here, we've declared the components for `Home`, `Category` and `Products` inside `App.js`. Although this is okay for now, when a component starts to grow bigger, it's better to have a separate file for each component. As a rule of thumb, I usually create a new file for a component if it occupies more than 10 lines of code. Starting from the second demo, I'll be creating a separate file for components that have grown too big to fit inside the `App.js` file.

Inside the `App` component, we've written the logic for routing. The `<Route>`'s path is matched with the current location and a component gets rendered. Previously, the component that should be rendered was passed in as a second prop. However, recent versions of React Router have introduced a new route rendering pattern, whereby the component(s) to be rendered are children of the `<Route>`.

Here `/` matches both `/` and `/category`. Therefore, both the routes are matched and rendered. How do we avoid that? You should pass the `exact` prop to the `<Route>` with `path='/'`:

```
<Route exact path="/">
  <Home />
</Route>
```

If you want a route to be rendered only if the paths are exactly the same, you should use the exact prop.

Nested Routing

To create nested routes, we need to have a better understanding of how `<Route>` works. Let's look at that now.

As you can read on the React Router docs, the recommended method of rendering something with a `<Route>` is to use `children` elements, as shown above. There are, however, a few other methods you can use to render something with a `<Route>`. These are provided mostly for supporting apps that were built with earlier versions of the router before hooks were introduced:

- `component`: when the URL is matched, the router creates a React element from the given component using `React.createElement`.
- `render`: handy for inline rendering. The `render` prop expects a function that returns an element when the location matches the route's path.
- `children`: this is similar to `render`, in that it expects a function that returns a React component. However, `children` gets rendered regardless of whether the path is matched with the location or not.

Path and Match

The `path` prop is used to identify the portion of the URL that the router should match. It uses the Path-to-RegExp library to turn a path string into a regular expression. It will then be matched against the current location.

If the router's path and the location are successfully matched, an object is created which is called a match object. The `match` object contains more information about the URL and the path. This information is accessible through its properties, listed below:

- `match.url`: a string that returns the matched portion of the URL. This is particularly useful for building nested `<Link>` components.
- `match.path`: a string that returns the route's path string—that is, `<Route path="">`. We'll be using this to build nested `<Route>` components.
- `match.isExact`: a Boolean that returns true if the match was exact (without any trailing characters).
- `match.params`: an object containing key/value pairs from the URL parsed by the Path-to-RegExp package.

Implicit Passing of Props

Note that when using the `component` prop to render a route, the `match`, `location` and `history`

route props are implicitly passed to the component. When using the newer route rendering pattern, this is not the case.

For example, take this component:

```
const Home = (props) => {
  console.log(props);

  return (
    <div>
      <h2>Home</h2>
    </div>
  );
};
```

Now render the route like so:

```
<Route exact path="/" component={Home} />
```

This will log the following:

```
{
  history: { ... }
  location: { ... }
  match: { ... }
}
```

But now instead render the route like so:

```
<Route exact path="/"><Home /></Route>
```

This will log the following:

```
{}
```

This might seem disadvantageous at first, but worry not! React v5.1 introduced several hooks to help you access what you need, where you need it. These hooks give us new ways to manage our router's state and go quite some way to tidying up our components.

I'll be using some of these hooks throughout this tutorial, but if you'd like a more in-depth look, check out the React Router v5.1 release announcement. Please also note that hooks were introduced in version 16.8 of React, so you'll need to be on at least that version to use them.

The `Switch` Component

Before we head to the demo code, I want to introduce you to the Switch component. When multiple `<Route>`s are used together, all the routes that match are rendered inclusively. Consider this code from demo 1. I've added a new route to demonstrate why `<Switch>` is useful:

```
<Route exact path="/"><Home /></Route>
<Route path="/category"><Category /></Route>
<Route path="/products"><Products /></Route>
<Route path="/:id">
  <p>This text will render for any route other than '/'</p>
</Route>
```

If the URL is `/products`, all the routes that match the location `/products` are rendered. So, the `<Route>` with path `/:id` gets rendered along with the `<Products>` component. This is by design. However, if this is not the behavior you're expecting, you should add the `<Switch>` component to your routes. With `<Switch>`, only the first child `<Route>` that matches the location gets rendered:

```
<Switch>
  <Route exact path="/"><Home /></Route>
  <Route path="/category"><Category /></Route>
  <Route path="/products"><Products /></Route>
  <Route path="/:id">
    <p>This text will render for any route other than those defined above</p>
  </Route>
</Switch>
```

The `:id` part of `path` is used for dynamic routing. It will match anything after the slash and make this value available in the component. We'll see an example of this at work in the next section.

Now that we know all about the `<Route>` and `<Switch>` components, let's add nested routes to our demo.

Dynamic Nested Routing

Earlier on, we created routes for `/`, `/category` and `/products`. But what if we wanted a URL in the form of `/category/shoes`?

Let's start by updating `src/App.js` as follows:

```
import React from "react";
import { Link, Route, Switch } from "react-router-dom";
import Category from "./Category";

const Home = () => (
  <div>
    <h2>Home</h2>
  </div>
);

const Products = () => (
  <div>
    <h2>Products</h2>
  </div>
);

export default function App() {
  return (
    <div>
      <nav className="navbar navbar-light">
        <ul className="nav navbar-nav">
          <li>
            <Link to="/">Home</Link>
          </li>
          <li>
            <Link to="/category">Category</Link>
          </li>
          <li>
            <Link to="/products">Products</Link>
          </li>
        </ul>
      </nav>

      <Switch>
        <Route path="/"><Home /></Route>
        <Route path="/category"><Category /></Route>
        <Route path="/products"><Products /></Route>
      </Switch>
    </div>
  );
}
```

You'll notice that we've moved `Category` into its own component. This is where our nested routes should go.

Let's create `Category.js` now:

```
// src/Category.js

import React from "react";
import { Link, Route, useParams, useRouteMatch } from "react-router-dom";

const Item = () => {
  const { name } = useParams();

  return (
    <div>
      <h3>{name}</h3>
    </div>
  );
}

const Category = () => {
  const { url, path } = useRouteMatch();

  return (
    <div>
      <ul>
        <li>
          <Link to={`${url}/shoes`}>Shoes</Link>
        </li>
        <li>
          <Link to={`${url}/boots`}>Boots</Link>
        </li>
        <li>
          <Link to={`${url}/footwear`}>Footwear</Link>
        </li>
      </ul>
      <Route path={`${path}/:name`}>
        <Item />
      </Route>
    </div>
  );
};

export default Category;
```

Here, we're using the useRouteMatch hook to gain access to the `match` object. As previously mentioned, `match.url` will be used for building nested links and `match.path` for nested routes. If you're having trouble understanding the concept of match, `console.log(useRouteMatch())` provides some useful information that might help to clarify it.

```
<Route path={`${path}/:name`}>
  <Item />
```

```
</Route>
```

This is our first proper attempt at dynamic routing. Instead of hard-coding the routes, we've used a variable within the `path` prop. `:name` is a path parameter and catches everything after `category/` until another forward slash is encountered. So, a path name like `products/running-shoes` will create a `params` object as follows:

```
{
  name: "running-shoes";
}
```

To access this value within the `<Item>` component, we're using the useParams hook, which returns an object of key/value pairs of URL parameters.

Try this out in your browser. The Category section should now have three sub-sections, each with their own route.

Nested Routing with Path Parameters

Let's complicate things a bit more, shall we? A real-world router will have to deal with data and display it dynamically. Let's assume we have some product data returned by an API in the following format:

```
const productData = [
  {
    id: 1,
    name: "NIKE Liteforce Blue Sneakers",
    description:
      "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Proin molestie.",
    status: "Available",
  },
  {
    id: 2,
    name: "Stylised Flip Flops and Slippers",
    description:
      "Mauris finibus, massa eu tempor volutpat, magna dolor euismod dolor.",
    status: "Out of Stock",
  },
  {
    id: 3,
    name: "ADIDAS Adispree Running Shoes",
    description:
      "Maecenas condimentum porttitor auctor. Maecenas viverra fringilla felis, eu pretium.",
    status: "Available",
```

```
  },
  {
    id: 4,
    name: "ADIDAS Mid Sneakers",
    description:
      "Ut hendrerit venenatis lacus, vel lacinia ipsum fermentum vel. Cras.",
    status: "Out of Stock",
  },
];
```

Let's also assume that we need to create routes for the following paths:

- `/products` : this should display a list of products.
- `/products/:productId` : if a product with the `:productId` exists, it should display the product data, and if not, it should display an error message.

Create a new file `src/Products.js` and add the following (making sure to copy in the product data from above):

```
import React from "react";
import { Link, Route, useRouteMatch } from "react-router-dom";
import Product from "./Product";

const Products = ({ match }) => {
  const productData = [ ... ];
  const { url } = useRouteMatch();

  /* Create an array of `<li>` items for each product */
  const linkList = productData.map((product) => {
    return (
      <li key={product.id}>
        <Link to={`${url}/${product.id}`}>{product.name}</Link>
      </li>
    );
  });

  return (
    <div>
      <div>
        <div>
          <h3>Products</h3>
          <ul>{linkList}</ul>
        </div>
      </div>

      <Route path={`${url}/:productId`}>
```

```
        <Product data={productData} />
      </Route>
      <Route exact path={url}>
        <p>Please select a product.</p>
      </Route>
    </div>
  );
};

export default Products;
```

First, we use the `useRouteMatch` hook to grab the URL from the `match` object. Then we build a list of `<Links>` components using the `id` property from each of our products, which we store in a `linkList` variable.

The first route uses a variable in the `path` prop which corresponds to that of the product ID. When it matches, we render out the `<Product>` component (which we'll define in a minute), passing it our product data:

```
<Route path={`${url}/:productId`}>
  <Product data={productData} />
</Route>
```

The second route has an `exact` prop, so will only render when the URL is `/products` and nothing is selected.

Now, here's the code for the `<Product>` component. You'll need to create this file at `src/Product.js`:

```
import React from "react";
import { useParams } from "react-router-dom";

const Product = ({ data }) => {
  const { productId } = useParams();
  const product = data.find(p => p.id === Number(productId));
  let productData;

  if (product) {
    productData = (
      <div>
        <h3> {product.name} </h3>
        <p>{product.description}</p>
        <hr />
        <h4>{product.status}</h4>
```

```
      </div>
    );
  } else {
    productData = <h2> Sorry. Product doesn't exist </h2>;
  }

  return (
    <div>
      <div>{productData}</div>
    </div>
  );
};

export default Product;
```

The `find` method is used to search the array for an object with an ID property that equals `match.params.productId`. If the product exists, the `productData` is displayed. If not, a "Product doesn't exist" message is rendered.

Finally, update your `<App>` component as follows:

```
import React from "react";
import { Link, Route, Switch } from "react-router-dom";
import Category from "./Category";
import Products from "./Products";

const Home = () => (
  <div>
    <h2>Home</h2>
  </div>
);

export default function App() {
  return (
    <div>
      <nav className="navbar navbar-light">
        <ul className="nav navbar-nav">
          <li>
            <Link to="/">Home</Link>
          </li>
          <li>
            <Link to="/category">Category</Link>
          </li>
          <li>
            <Link to="/products">Products</Link>
          </li>
        </ul>
```

```
      </nav>

      <Switch>
        <Route exact path="/"><Home /></Route>
        <Route path="/category"><Category /></Route>
        <Route path="/products"><Products /></Route>
      </Switch>
    </div>
  );
}
```

Now when you visit the application in the browser and select "Products", you'll see a sub-menu rendered, which in turn displays the product data.

Have a play around with the demo. Assure yourself that everything works and that you understand what's happening in the code.

Protecting Routes

A common requirement for many modern web apps is to ensure that only logged-in users can access certain parts of the site. In this next section, we'll look at how to implement a protected route, so that if someone tries to access `/admin`, they'll be required to log in.

However, there are a couple of aspects of React Router that we need to cover first.

The `Redirect` Component

As with server-side redirects, React Router's Redirect component will replace the current location in the history stack with a new location. The new location is specified by the `to` prop. Here's how we'll be using `<Redirect>`:

```
<Redirect to={{pathname: '/login', state: { from: location }}}
```

So, if someone tries to access the `/admin` route while logged out, they'll be redirected to the `/login` route. The information about the current location is passed via the `state` prop, so that if the authentication is successful, the user can be redirected back to the page they were originally trying to access.

Custom Routes

A custom route is a fancy way of describing a route nested inside a component. If we need to make a decision whether a route should be rendered or not, writing a custom route is the way to

go.

Create a new file `PrivateRoute.js` in the `src` directory and add the following content:

```
import React from "react";
import { Redirect, Route, useLocation } from "react-router-dom";
import { fakeAuth } from './Login';

const PrivateRoute = ({ component: Component, ...rest }) => {
  const location = useLocation();

  return (
    <Route {...rest}>
      {fakeAuth.isAuthenticated === true ?
        <Component />
      :
        <Redirect to={{ pathname: "/login", state: { from: location } }} />
      }
    </Route>
  );
};

export default PrivateRoute;
```

As you can see, in the function definition we're destructuring the props we receive into a `Component` prop and a `rest` prop. The `Component` prop will contain whichever component our `<PrivateRoute>` is protecting (in our case, `Admin`). The `rest` prop will contain any other props we've been passed.

We then return a `<Route>` component, which renders either the protected component or redirects us to our `/login` route, depending on whether or not the user is logged in. This is determined here by a `fakeAuth.isAuthenticated` property, which is imported from the `<Login>` component.

The good thing about this approach is that it's evidently more declarative and `<PrivateRoute>` is reusable.

Important Security Notice

In a real-world app, *you need to validate any request for a protected resource on your server*. This is because anything that runs in the client can potentially be reverse engineered and tampered with. For example, in the above code one can just open React's dev tools and change the value of `isAuthenticated`, thus gaining access to the protected area.

Authentication in a React app is worthy of a tutorial of its own, but one way to implement it would be using JSON Web Tokens. For example, you could have an endpoint on your server which accepts a username and password combination. When it receives these (via Ajax), it checks to see if the credentials are valid. If so, it responds with a JWT, which the React app saves (for example in `sessionStorage`), and if not, it sends a `401 Unauthorized` response back to the client.

Assuming a successful login, the client would then send the JWT as a header along with any request for a protected resource. This would then be validated by the server before it sent a response.

When storing passwords, *the server would not store them in plaintext*. Rather, it would encrypt them—for example, using bcryptjs.

Implementing the Protected Route

Now let's implement our protected route. Alter `src/App.js` like so:

```
import React from "react";
import { Link, Route, Switch } from "react-router-dom";
import Category from "./Category";
import Products from "./Products";
import Login from './Login';
import PrivateRoute from "./PrivateRoute";

const Home = () => (
  <div>
    <h2>Home</h2>
  </div>
);

const Admin = () => (
  <div>
    <h2>Welcome admin!</h2>
  </div>
);

export default function App() {
  return (
    <div>
      <nav className="navbar navbar-light">
        <ul className="nav navbar-nav">
          <li>
            <Link to="/">Home</Link>
          </li>
          <li>
```

```
            <Link to="/category">Category</Link>
          </li>
          <li>
            <Link to="/products">Products</Link>
          </li>
          <li>
            <Link to="/admin">Admin area</Link>
          </li>
        </ul>
      </nav>

      <Switch>
        <Route exact path="/"><Home /></Route>
        <Route path="/category"><Category /></Route>
        <Route path="/products"><Products /></Route>
        <Route path="/login"><Login /></Route>
        <PrivateRoute path="/admin" component={Admin} />
      </Switch>
    </div>
  );
}
```

As you can see, we've added an `<Admin>` component to the top of the file and are including our `<PrivateRoute>` within the `<Switch>` component. As mentioned previously, this custom route renders the `<Admin>` component if the user is logged in. Otherwise, the user is redirected to `/login`.

Finally, here's the code for the Login component:

```
import React, { useState } from "react";
import { Redirect, useLocation } from "react-router-dom";

export default function Login() {
  const { state } = useLocation();
  const { from } = state || { from: { pathname: "/" } };
  const [redirectToReferrer, setRedirectToReferrer] = useState(false);

  const login = () => {
    fakeAuth.authenticate(() => {
      setRedirectToReferrer(true);
    });
  };

  if (redirectToReferrer) {
    return <Redirect to={from} />;
  }
```

```
  return (
    <div>
      <p>You must log in to view the page at {from.pathname}</p>
      <button onClick={login}>Log in</button>
    </div>
  );
}

/* A fake authentication function */
export const fakeAuth = {
  isAuthenticated: false,
  authenticate(cb) {
    this.isAuthenticated = true;
    setTimeout(cb, 100);
  }
};
```

By now, there's hopefully nothing too tricky going on here. We use the useLocation hook to access the router's `location` prop, from which we grab the `state` property. We then use object destructuring to get a value for the URL the user was trying to access before being asked to log in. If this isn't present, we set it to `{ pathname: "/" }`.

We then use React's `useState` hook to initialize a `redirectToReferrer` property to `false`. Depending on the value of this property, the user is either redirected to where they were going (that is, the user is logged in), or the user is presented with a button to log them in.

Once the button is clicked, the `fakeAuth.authenticate` method is executed, which sets `fakeAuth.isAuthenticated` to `true` and (in a callback function) updates the value of `redirectToReferrer` to `true`. This causes the component to re-render and the user to be redirected.

Working Demo

Let's fit the puzzle pieces together, shall we? Here's the final demo of the application we built using React router.

Summary

As you've seen in this guide, React Router is a powerful library that complements React for building better, declarative routes. Unlike the prior versions of React Router, in v5, everything is "just components". Moreover, the new design pattern perfectly fits into the React way of doing things.

In this tutorial, we learned:

- how to set up and install React Router
- the basics of routing and some essential components such as `<Router>`, `<Route>` and `<Link>`
- how to create a minimal router for navigation and nested routes
- how to build dynamic routes with path parameters
- how to work with React Router's hooks and its newer route rendering pattern

Finally, we learned some advanced routing techniques for creating the final demo for protected routes.

Chapter

20 Essential React Tools

2

Camilo Reyes

The React ecosystem has evolved into a growing list of dev tools and libraries. The plethora of tools is a true testament to React's popularity. For devs, it can be a dizzying exercise to navigate this maze that changes at neck-breaking speed. To help navigate your course, below is a list of essential React tools, techniques and skills for 2020 and beyond.

Hooks

- Website: reactjs.org/docs/hooks-intro.html
- Repository: github.com/facebook/react
- GitHub stars: 157,000+
- Developer: Facebook
- Current version: 16.14.0
- Contributors: 1,500+

While not strictly a tool, any developer working with React in the 2020s *needs* to be familiar with hooks. These are a new addition to React as of version 16.8 which unlock useful features in function components. For example, the `useState` hook allows a function component to have its own state, whereas `useEffect` allows you to perform side effects after the initial render—for example, manipulating the DOM or data fetching. Hooks can be used to replicate lifecycle methods in functional components and allow you to share code between components.

The following basic hooks are available:

- useState: for mutating state in a function component without lifecycle methods
- useEffect: for executing functions post-render, useful for firing Ajax requests
- useContext: for accessing component context data, even outside component props

Pros:

- mitigates state management complexity
- supports function components
- encourages separation of concerns

Cons:

- context data switching can increase cognitive load

If you'd like to find out more about hooks, then check out our tutorial, "React Hooks: How to Get Started & Build Your Own".

Function Components

- Website: reactjs.org/docs/components-and-props.html
- Repository: github.com/facebook/react
- GitHub stars: 157,000+
- Developer: Facebook
- Current version: 16.14.0
- Contributors: 1,500+

With the advent of hooks, function components—a declarative way to create JSX markup without using a class—are becoming more popular than ever. They embrace the functional paradigm because they don't manage state in lifecycle methods. This emphasizes focus on the UI markup without much logic. Because the component relies on props, it becomes easier to test. Props have a *one-to-one* relationship with the rendered output.

This is what a functional component looks like in React:

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

Pros:

- focuses on the UI
- testable component
- less cognitive load when thinking about the component

Cons:

- no lifecycle methods

Create React App

- Website: create-react-app.dev
- Repository: github.com/facebook/create-react-app
- GitHub stars: 82,000+
- Developer: Facebook
- Current version: 3.4.1
- Contributors: 800+

Create React App is the quintessential tool for firing up a new React project. It manages all React

dependencies via a single npm package. No more dealing with Babel, webpack, and the rest. All it takes is one command to set up a local development environment, with React, JSX, and ES6 support. But that's not all. Create React App also offers hot module reloading (your changes are immediately reflected in the browser when developing), automatic code linting, a test runner and a build script to bundle JS, CSS, and images for production.

It's easy to get started:

```
npx create-react-app my-killer-app
```

And it's even easier to upgrade later. The entire dependency tool chain gets upgraded with `react-scripts` in `package.json`:

```
npm i react-scripts@latest
```

Pros:

- easy to get started
- easy to upgrade
- single meta-dependency

Cons:

- no server-side rendering, but allows for integration

If you'd like to find out more about using Create React App, please consult our tutorial, "Create React App: Get React Projects Ready Fast".

Proxy Server

- Website: create-react-app.dev/docs/proxying-api-requests-in-development
- Repository: github.com/facebook/create-react-app
- GitHub stars: 82,000+
- Developer: Facebook
- Current version: 3.4.1
- Contributors: 800+

Starting from version `react-scripts@0.2.3` or higher, it's possible to proxy API requests. This allows the back-end API and local Create React App project to co-exist. From the client side, making a request to `/my-killer-api/get-data` routes the request through the proxy server. This

seamless integration works both in local dev and post-build. If local dev runs on port `localhost:3000`, then API requests go through the proxy server. Once you deploy static assets, it goes through whatever back end hosts these assets.

To set a proxy server in `package.json`:

```
"proxy": "http://localhost/my-killer-api-base-url"
```

If the back-end API is hosted with a relative path, set the home page:

```
"homepage": "/relative-path"
```

Pros:

- seamless integration with back-end API
- eliminates CORS issues
- easy set up

Con

- might need a server-side proxy layer with multiple APIs

PropTypes

- Website: npmjs.com/package/prop-types
- Repository: github.com/facebook/prop-types
- GitHub stars: 3,600+
- Developer: Facebook
- Current version: 15.7.2
- Contributors: 45+

PropTypes declares the type intended for the React component and documents its intent. This shows a warning in local dev if the types don't match. It supports all JavaScript primitives such as `Boolean`, `Number`, and `String`. It can document which props are required via `isRequired`.

For example:

```
import PropTypes;

MyComponent.propTypes = {
```

```
  boolProperty: PropTypes.bool,
  numberProperty: PropTypes.number,
  requiredProperty: PropTypes.string.isRequired
};
```

Pros:

- documents component's intent
- shows warnings in local dev
- supports all JavaScript primitives

Cons:

- no compile type checking

TypeScript

- Website: typescriptlang.org
- Repository: github.com/Microsoft/TypeScript
- GitHub stars: 65,000+
- Developer: Microsoft
- Current version: 4.0.3
- Contributors: 530+

JavaScript that scales for React projects with compile type checking. This supports all React libraries and tools with type declarations. It's a superset of JavaScript, so it's possible to opt out of the type checker. This both documents intent and fails the build when it doesn't match. In Create React App projects, turn it on by passing in `--template typescript` when creating your app. TypeScript support is available starting from version `react-script@2.1.0`.

To declare a prop type:

```
interface MyComponentProps {
  boolProp?: boolean; // optional
  numberProp?: number; // optional
  requiredProp: string;
}
```

Pros:

- compile type checking
- supports all React tools and libraries, including Create React App

- nice way to up your JavaScript skills

Cons:

- has a learning curve, but opt out is possible

If you'd like to find out more about using TypeScript with React, check out "React with TypeScript: Best Practices".

Redux

- Website: redux.js.org/
- Repository: github.com/reduxjs/redux
- GitHub stars: 54,000+
- Developers: Dan Abramov and Andrew Clark
- Current version: 4.0.5
- Contributors: 800+

Predictable state management container for JavaScript apps. This tool comes with a store that manages state data. State mutation is only possible via a dispatch message. The message object contains a type that signals to the reducer which mutation to fire. The recommendation is to keep everything in the app in a single store. Redux supports multiple reducers in a single store. Reducers have a one-to-one relationship between input parameters and output state. This makes reducers pure functions.

A typical reducer that mutates state might look like this:

```
const simpleReducer = (state = {}, action) => {
  switch (action.type) {
    case 'SIMPLE_UPDATE_DATA':
      return {...state, data: action.payload};

    default:
      return state;
  }
};
```

Pros:

- predictable state management
- multiple reducers in a single store
- reducers are pure functions

Cons:

- set up from scratch can be a bit painful

React-Redux

- Website: react-redux.js.org
- Repository: github.com/reduxjs/redux
- GitHub stars: 18,500+
- Developer: Redux team
- Current version: 7.2.1
- Contributors: 220+

If you want to use Redux in your React apps, you'll soon discover the official React bindings for Redux. This comes in two main modules: `Provider` and `connect`. The `Provider` is a React component with a `store` prop. This prop is how a single store hooks up to the JSX markup. The `connect` function takes in two parameters: `mapStateToProps`, and `mapDispatchToProps`. This is where state management from Redux ties into component props. As state mutates, or dispatches fire, bindings take care of setting state in React.

This is how a connect might look:

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

const mapStateToProps = (state) => state.simple;
const mapDispatchToProps = (dispatch) =>
  bindActionCreators({() => ({type: 'SIMPLE_UPDATE_DATA'})}, dispatch);

connect(mapStateToProps, mapDispatchToProps)(SimpleComponent);
```

Pros:

- official React bindings for Redux
- binds with JSX markup
- connects components to a single store

Cons:

- learning curve is somewhat steep

It should also be noted that, with the introduction of hooks and React's Context API, it's possible

to replace Redux in some React applications. You can read more about that in "How to Replace Redux with React Hooks and the Context API".

React Router

- Website: reactrouter.com
- Repository: github.com/ReactTraining/react-router
- GitHub stars: 42,000+
- Developer: React Training
- Current version: 5.2.0
- Contributors: 640+

React Router is the de facto standard routing library for React. When you need to navigate through a React application with multiple views, you'll need a router to manage the URLs. React Router takes care of that, keeping your application UI and the URL in sync.

The library comprises three packages: react-router, react-router-dom, and react-router-native. The core package for the router is `react-router`, whereas the other two are environment specific. You should use `react-router-dom` if you're building a website, and `react-router-native` if you're building a React Native app.

Recent versions of React Router have introduced hooks, which let you access the state of the router and perform navigation from inside your components, as well as a newer route rendering pattern:

```
<Route path="/">
  <Home />
</Route>
```

If you'd like to find out more about what React Router can do, please see "React Router v5: The Complete Guide".

Pros:

- routing between components is fast
- animations and transitions can be easily implemented
- connects components to a single store

Cons:

- without additional configuration, data is downloaded for views a user might not visit

- client-side routing (whereby JavaScript is converted to HTML) has SEO implications

ESLint

- Website: eslint.org
- Repository: github.com/eslint/eslint
- GitHub stars: 17,000+
- Developer: Nicholas C. Zakas and the ESLint team
- Current version: 7.11.0
- Contributors: 820+

ESLint is a linting tool that can be used to keep your code style consistent, enforce code quality and spot potential errors and bad patterns ahead of time. It offers a plugin with React-specific linting rules and is often used in conjunction with Airbnb's React style guide.

Although ESLint can be run via the command line, it pays dividends to spend some time integrating it into your editor of choice. Many of the problems it finds can be automatically fixed and, coupled with a tool like Prettier, ESLint can seriously help improve the quality of your code, as well as your overall developer experience.

Anyone using Create React App will notice that it ships with ESLint already enabled and provides a minimal set of rules intended to find common mistakes.

Pros:

- flexible: any rule can be toggled, and many rules have extra settings that can be tweaked
- extensible: many plugins available
- support for JSX and TypeScript

Cons:

- editor integration can potentially prove bothersome
- can potentially introduce several new dependencies to a project

If you'd like to learn more about ESLint, please consult our tutorial, "Up and Running with ESLint — the Pluggable JavaScript Linter".

Lodash

- Website: lodash.com
- Repository: github.com/lodash/lodash

- GitHub stars: 46,500+
- Developer: John-David Dalton and Lodash team
- Current version: 4.17.20
- Contributors: 300+

Lodash is a modern JavaScript utility library useful for React components. For example, React form input events like `onChange` fire once per keystroke. If the component gets data from a back-end API, it fires requests once per keystroke. This spams the back-end API and causes issues when many people use the UI. Lodash comes with debounced events, which fires one API request with many keystrokes.

To set up `onChange` debounced events:

```
onChange={(e) => debounce(updateDataValue(e.target.value), 250)}
```

Pros:

- modular dependency
- plays well with code-splitting
- easy to use

Cons:

- knowing when to debounce events is not immediately obvious

Axios

- Website: npmjs.com/package/axios
- Repository: github.com/axios/axios
- GitHub stars: 77,500+
- Developer: axios team
- Current version: 0.20.0
- Contributors: 250+

Making HTTP requests to fetch or save data is one of the most common tasks a client-side JavaScript application will need to do. And there is arguably no library better suited to the task than axios, a Promise-based HTTP client with an easy-to-use API. The tool supports `async` ... `await` syntax to make Ajax requests from the browser. It supports error handling in case there are errors via `catch`. The tool's API supports HTTP requests such as GET, DELETE, POST, PUT, and PATCH. This also plays well with Promise API calls like `Promise.all()` to send HTTP

requests in parallel.

Similar to jQuery's `$.ajax` function, you can make any kind of HTTP request by passing an options object to axios:

```
axios({
  method: 'post',
  url: '/login',
  data: {
    user: 'camilo',
    lastName: 'reyes'
  }
});
```

Pros:

- promise based
- supports async/await
- supports error handling

Cons:

- it can't get any more awesome

If you'd like to learn more about using axios in your projects, see "Introducing Axios, a Popular, Promise-based HTTP Client".

Jest

- Website: jestjs.io
- Repository: github.com/facebook/jest
- GitHub stars: 32,500+
- Developer: Facebook
- Current version: 26.5.3
- Contributors: 1,000+

Jest is a testing framework with a focus on simplicity for JavaScript projects. The good news is it comes built-in with Create React App. It works with projects that use Babel, TypeScript, and Node. There's no configuration on most React projects. Tests can run in watch mode, which keeps track of code changes and reruns tests. The API contains `it`, and `expect` to quickly get started.

A sanity check to make sure tests execute is:

```
it('says true is true', () => {
  expect(true).toBe(true);
});
```

Pros:

- easy set up with Create React App
- fluent API
- runs in watch mode

Cons:

- too bare bones to render React components

If you'd like to find out how you can use Jest in your React projects, please see "How to Test React Components Using Jest".

Enzyme

- Website: enzymejs.github.io/enzyme/
- Repository: github.com/enzymejs/enzyme
- GitHub stars: 19,000+
- Developer: Airbnb
- Current version: 3.11.0
- Contributors: 350+

Enzyme is a JavaScript testing utility for React that makes it easier to test components. The API is meant to be as intuitive as jQuery for component traversal. To get Enzyme, it needs two packages: `enzyme`, and a separate adapter. The adapter must be compatible with the version of React. For example, `enzyme-adapter-react-16` for React `^16.4.0`, `enzyme-adapter-react-16.3` for `~16.3.0`, so on and so forth. The adapter needs a configuration file `setupTest.js` to integrate with Jest.

When using React 16, install Enzyme with:

```
npm i --save-dev enzyme enzyme-adapter-react-16
```

Pros:

- supports React components
- supports Jest test framework
- intuitive API

Cons:

- kind of painful to set up an adapter in Jest

Shallow Renderer

- Website: enzymejs.github.io/enzyme/docs/api/shallow.html
- Repository: github.com/airbnb/enzyme
- GitHub stars: 19,000+
- Developer: Airbnb
- Current version: 3.11.0
- Contributors: 350+

This is shallow rendering useful for limiting tests to one level deep. It renders the parent component without affecting its children in a tree hierarchy. This isolates the test and makes assertions more robust. Shallow rendering supports a good chunk of the Enzyme API for traversing components. The `shallow` API does call lifecycle methods like `componentDidMount` and `componentDidUpdate` during render. With hooks, the shallow renderer does not call `useEffect`. One tip is do `console.log(component.debug())` to inspect what the shallow renderer sees.

To test a React component using the shallow renderer:

```
const component = shallow(<ParentComponent data={"Dave"} />);
expect(component.find('p').at(0).text()).toBe('Dave');
```

Pros:

- isolates test
- full featured API
- allows quick debugging

Cons:

- must navigate the sea of options in Enzyme's API to find this diamond in the rough

Storybook

- Website: storybook.js.org
- Repository: github.com/storybookjs/storybook
- GitHub stars: 54,000+
- Developer: Storybook
- Current version: 6.0.26
- Contributors: 1100+

This is an open-source tool for manual testing of React components in isolation. Storybook provides a sandbox to build components to get into hard to reach edge cases. It allows mocking so it can render components in key states that are hard to reproduce. Setup is automatic with Create React App when using `react-scripts`. Each story in Storybook can target a single component with many states. The story files have a convention like `component.stories.js` so one can quickly find them.

To get started with Storybook:

```
npx -p @storybook/cli sb init
```

Pros:

- covers hard-to-reach edge cases
- renders components in sandbox
- integrates with Create React App

Cons:

- hard to automate tests

You can find out more about Storybook in our guide, "React Storybook: Develop Beautiful User Interfaces with Ease".

React Bootstrap

- Website: react-bootstrap.github.io
- Repository: github.com/react-bootstrap/react-bootstrap
- GitHub stars: 18,400+
- Developer: react-bootstrap
- Current version: 1.3.0

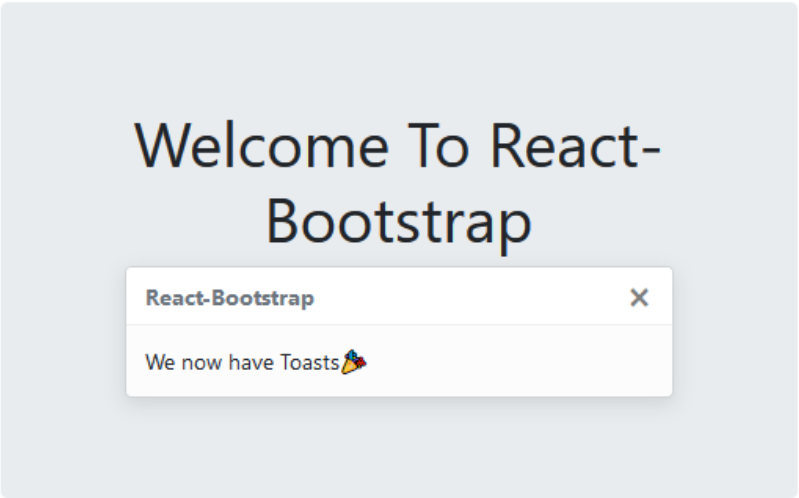
- Contributors: 300+

This is the most popular front-end framework rebuilt for React. Every Bootstrap component is built from scratch as a React component. This replaces Bootstrap JavaScript and nukes dependencies like jQuery. The latest version supports Bootstrap 4.5. React Bootstrap works with the thousands of Bootstrap themes already found in version 4. Each component has accessibility in mind and is accessible by default. It supports Create React App out of the box, and custom themes are also supported.

To fire up React Bootstrap in a React project:

```
npm install react-bootstrap bootstrap
```

This is what the result might look like:



Pros:

- rebuilt from scratch with React components
- accessibility in mind
- supports Create React App

Cons:

- custom themes can be tricky in Create React App

Material-UI

- Website: material-ui.com
- Repository: github.com/mui-org/material-ui
- GitHub stars: 54,500+
- Developer: Material-UI
- Current version: 4.9.3
- Contributors: 1,500+

Material-UI offers popular React components for faster and easier web development. It allows building your own design system or starting with Material Design. There are templates and themes available, both premium and free. Premium themes have a price tag depending on functionality. Material-UI comes via an npm package for quick installation.

To get started with Material-UI:

```
npm install @material-ui/core
```

This is what the result might look like:

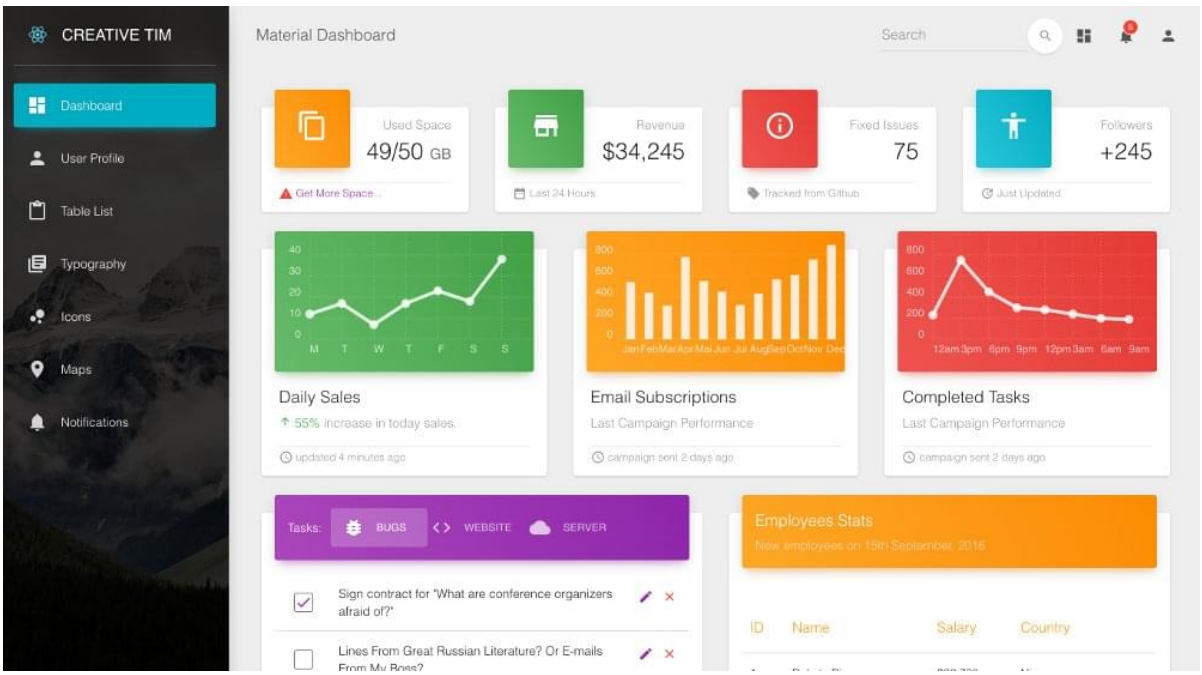


Image source

Pros:

- build a powerful UI with little effort
- offers many components
- offers many templates

Cons:

- some premium templates do cost, but might be worth the money

React DevTools

- Website: reactjs.org/blog/2019/08/15/new-react-devtools.html
- Repository: github.com/facebook/react
- GitHub stars: 157,000+
- Developer: Facebook
- Current version: 16.14.0
- Contributors: 1,500+

One of the most important tools in any React developer's toolbelt should be the React Developer Tools—a browser extension for both Chrome and Firefox. This allows you to easily inspect a React tree, including the component hierarchy, props, state, and more.

Once installed, the dev tools will give you two new tabs in your browser console—**Components** and **Profiler**. Clicking the former will show you all of the components in the current tree and allow you to filter them by name, whereas the latter allows you to record performance information about your React app.

This is a must-have tool when it comes to debugging a React app that consists of more than a handful of components.

Pros

- helps you understand what's happening in your React app
- makes debugging considerably less painful
- see which sites are using React in production

Cons

- slight learning curve

Awesome React

- Repository: github.com/enaqx/awesome-react
- GitHub stars: 39,500+
- Developer: n/a
- Current version: n/a
- Contributors: 550+

Let's round this list off with … another list! This time awesome-react—a GitHub repo containing a collection of awesome things relating to the React ecosystem.

The repo has a Tools section that contains many of the tools listed in this guide (and a whole lot more besides), as well as sections on *Tutorials*, *Demos*, *Videos*, *Conference Talks*, *ReactNative*, *GraphQL* and more. Whatever you're looking for in the world of React, this is a great place to start.

Pros

- something for everyone
- covers a wide variety of resources
- actively kept up to date

Cons

- too much choice

Conclusion

As shown, React's ecosystem has exploded within the last few years. It's the tool of choice for enterprise wanting to reuse React components in a consistent suite. Each tool is standalone with few interdependencies. My recommendation is to give these tools at least a try.

Chapter

3

React with TypeScript: Best Practices

JavaScript Joe

React and TypeScript are two awesome technologies used by a lot of developers these days. Knowing how to do things can get tricky, and sometimes it's hard to find the right answer. Not to worry. We've put together the best practices along with examples to clarify any doubts you may have.

Let's dive in!

How React and TypeScript Work Together

Before we begin, let's revisit how React and TypeScript work together. React is a "JavaScript library for building user interfaces", while TypeScript is a "typed superset of JavaScript that compiles to plain JavaScript." By using them together, we essentially build our UIs using a typed version of JavaScript.

The reason you might use them together would be to get the benefits of a statically typed language (TypeScript) for your UI. This means more safety and fewer bugs shipping to the front end.

Does TypeScript Compile My React Code?

A common question that's always good to review is whether TypeScript compiles your React code. The way TypeScript works is similar to this interaction:

TS: "Hey, is this all your UI code?"
React: "Yup!"
TS: "Cool! I'm going to compile it and make sure you didn't miss anything."
React: "Sounds good to me!"

So the answer is yes, it does! But later, when we cover the `tsconfig.json` settings, most of the time you'll want to use `"noEmit": true`. What this means is TypeScript *will not* emit JavaScript out after compilation. This is because typically, we're just utilizing TS to do our TypeScript.

The output is handled, in a CRA setting, by `react-scripts`. We run `yarn build` and `react-scripts` bundles the output for production.

To recap, TypeScript compiles your React code to type-check your code. It doesn't emit any JavaScript output (in most scenarios). The output is still similar to a non-TypeScript React project.

Can TypeScript Work with React and webpack?

Yes, TypeScript can work with React and webpack. Lucky for you, the webpack documentation has a guide on that.

Hopefully, that gives you a gentle refresher on how the two work together. Now, on to best practices!

Best Practices

We've researched the most common questions and put together this handy list of the most common use cases for React with TypeScript. This way, you can use this guide as a reference in your own projects.

Configuration

One of the least fun, yet most important parts of development is configuration. How can we set things up in the shortest amount of time that will provide maximum efficiency and productivity? We'll discuss project setup including:

- `tsconfig.json`
- ESLint
- Prettier
- VS Code extensions and settings.

Project Setup

The quickest way to start a React/TypeScript app is by using `create-react-app` with the TypeScript template. You can do this by running:

```
npx create-react-app my-app --template typescript
```

This will get you the bare minimum to start writing React with TypeScript. A few noticeable differences are:

- the `.tsx` file extension
- the `tsconfig.json`
- the `react-app-env.d.ts`

The `tsx` is for "TypeScript JSX". The `tsconfig.json` is the TypeScript configuration file, which

has some defaults set. The `react-app-env.d.ts` references the types of `react-scripts`, and helps with things like allowing for SVG imports.

tsconfig.json

Lucky for us, the latest React/TypeScript template generates `tsconfig.json` for us. However, they add the bare minimum to get started. We suggest you modify yours to match the one below. We've added comments to explain the purpose of each option as well:

```
{
  "compilerOptions": {
    "target": "es5", // Specify ECMAScript target version
    "lib": [
      "dom",
      "dom.iterable",
      "esnext"
    ], // List of library files to be included in the compilation
    "allowJs": true, // Allow JavaScript files to be compiled
    "skipLibCheck": true, // Skip type checking of all declaration files
    "esModuleInterop": true, // Disables namespace imports  and enables CJS/AMD/UMD style imports
    "allowSyntheticDefaultImports": true, // Allow default imports from modules with no default export
    "strict": true, // Enable all strict type checking options
    "forceConsistentCasingInFileNames": true, // Disallow inconsistently-cased file references
    "module": "esnext", // Specify module code generation
    "moduleResolution": "node", // Resolve modules using Node.js style
    "isolatedModules": true, // Unconditionally emit imports for unresolved files
    "resolveJsonModule": true, // Include modules imported with .json extension
    "isolatedModules": true, // Transpile each file as a separate module
    "noEmit": true, // Do not emit output (meaning do not compile code, only perform type checking)
    "jsx": "react" // Support JSX in .tsx files
    "sourceMap": true, // Generate corresponding .map file
    "declaration": true, // Generate corresponding .d.ts file
    "noUnusedLocals": true, // Report errors on unused locals
    "noUnusedParameters": true, //  Report errors on unused parameters
    "incremental": true // Enable incremental compilation
    "noFallthroughCasesInSwitch": true // Report errors for fallthrough cases in switch statement
  },
  "include": [
    "src/**/*" // The files TypeScript should type check
  ],
  "exclude": ["node_modules", "build"] // The files to not type check
}
```

The additional recommendations come from the react-typescript-cheatsheet community and the explanations come from the Compiler Options docs in the Official TypeScript Handbook. This is a wonderful resource if you want to learn about other options and what they do.

ESLint/Prettier

In order to ensure that your code follows the rules of the project or your team, and the style is consistent, it's recommended you set up ESLint and Prettier. To get them to play nicely, follow these steps to set it up.

1. Install the required dev dependencies:

```
yarn add eslint @typescript-eslint/parser @typescript-eslint/eslint-plugin eslint-plugin-react --dev
```

2. Create a `.eslintrc.js` file at the root and add the following:

```
module.exports = {
  parser: '@typescript-eslint/parser', // Specifies the ESLint parser
  extends: [
    'plugin:react/recommended',
    'plugin:@typescript-eslint/recommended',
  ],
  parserOptions: {
  ecmaVersion: 2018, // Allows for the parsing of modern ECMAScript features
  sourceType: 'module', // Allows for the use of imports
  ecmaFeatures: {
    jsx: true, // Allows for the parsing of JSX
  },
  },
  rules: {
    // Place to specify ESLint rules. Can be used to overwrite rules specified from the
    // extended configs
    // e.g. "@typescript-eslint/explicit-function-return-type": "off",
  },
  settings: {
    react: {
      version: 'detect', // Tells eslint to automatically detect the version of React
    },
  },
};
```

3. Add Prettier dependencies:

```
yarn add prettier eslint-config-prettier eslint-plugin-prettier --dev
```

4 Create a `.prettierrc.js` file at the root and add the following:

```
module.exports = {
  semi: true,
  trailingComma: 'all',
  singleQuote: true,
  printWidth: 120,
  tabWidth: 4,
};
```

5 Update the `.eslintrc.js` file:

```
module.exports = {
  parser: '@typescript-eslint/parser', // Specifies the ESLint parser
  extends: [
    'plugin:react/recommended',
    'plugin:@typescript-eslint/recommended',
+   'prettier/@typescript-eslint',
+   'plugin:prettier/recommended',
...
```

These recommendations come from a community resource written called "Using ESLint and Prettier in a TypeScript Project", by Robert Cooper. If you visit this resource, you can read more about the "why" behind these rules and configurations.

VS Code Extensions and Settings

We've added ESLint and Prettier and the next step to improve our DX is to automatically fix/ prettify our code on save.

First, install the ESLint extension for VS Code. This will allow ESLint to integrate with your editor seamlessly.

Next, update your Workspace settings by adding the following to your `.vscode/settings.json` :

```
{
  "eslint.autoFixOnSave": true,
  "eslint.validate": [
    "javascript",
    "javascriptreact",
    { "language": "typescript", "autoFix": true },
```

```
    { "language": "typescriptreact", "autoFix": true }
  ],
  "editor.formatOnSave": true,
  "[javascript]": {
    "editor.formatOnSave": false
  },
  "[javascriptreact]": {
    "editor.formatOnSave": false
  },
  "[typescript]": {
    "editor.formatOnSave": false
  },
  "[typescriptreact]": {
    "editor.formatOnSave": false
  }
}
```

This will allow VS Code to work its magic and fix your code when you save. It's beautiful!

These suggestions also come from the previously linked article "Using ESLint and Prettier in a TypeScript Project", by Robert Cooper.

Components

One of the core concepts of React is components. Here, we'll be referring to standard components as of React v16.8, meaning ones that use hooks as opposed to classes.

In general, there's much to be concerned with for basic components. Let's look at an example:

```
import React from 'react'

// Written as a function declaration
function Heading(): React.ReactNode {
  return <h1>My Website Heading</h1>
}

// Written as a function expression
const OtherHeading: React.FC = () => <h1>My Website Heading</h1>
```

Notice the key difference here. In the first example, we're writing our function as a *function declaration*. We annotate the *return type* with `React.Node` because that's what it returns. In contrast, the second example uses a *function expression*. Because the second instance returns a function, instead of a value or expression, we annotate the *function type* with `React.FC` for React "Function Component".

It can be confusing to remember the two. It's mostly a matter of design choice. Whichever you choose to use in your project, use it consistently.

More on React.FC and React.ReactNode

To read more about `React.FC`, look here, and read here for more on `React.ReactNode`.

Props

The next core concept we'll cover is props. You can define your props using either an interface or a type. Let's look at another example:

```
import React from 'react'

interface Props {
  name: string;
  color: string;
}

type OtherProps = {
  name: string;
  color: string;
}

// Notice here we're using the function declaration with the interface Props
function Heading({ name, color }: Props): React.ReactNode {
  return <h1>My Website Heading</h1>
}

// Notice here we're using the function expression with the type OtherProps
const OtherHeading: React.FC<OtherProps> = ({ name, color }) =>
  <h1>My Website Heading</h1>
```

When it comes to types or interfaces, we suggest following the guidelines presented by the `react-typescript-cheatsheet` community:

- "Always use interface for public API's definition when authoring a library or 3rd-party ambient type definitions."
- "Consider using type for your React Component Props and State, because it is more constrained."

You can read more about the discussion and see a handy table comparing types vs interfaces here.

Let's look at one more example so we can see something a little bit more practical:

```
import React from 'react'

type Props = {
  /** color to use for the background */
  color?: string;
  /** standard children prop: accepts any valid React Node */
  children: React.ReactNode;
  /** callback function passed to the onClick handler*/
  onClick: ()  => void;
}

const Button: React.FC<Props> = ({ children, color = 'tomato', onClick }) => {
   return <button style={{ backgroundColor: color }} onClick={onClick}>{children}</button>
}
```

In this `<Button />` component, we use a type for our props. Each prop has a short description listed above it to provide more context to other developers. The `?` after the prop named `color` indicates that it's optional. The `children` prop takes a `React.ReactNode` because it accepts everything that's a valid return value of a component (read more here). To account for our optional `color` prop, we use a default value when destructuring it. This example should cover the basics and show you have to write types for your props and use both optional and default values.

In general, keep these things in mind when writing your props in a React and TypeScript project:

- Always add descriptive comments to your props using the TSDoc notation `/** comment */`.
- Whether you use types or interfaces for your component props, use them consistently.
- When props are optional, handle appropriately or use default values.

Hooks

Luckily, the TypeScript type inference works well when using hooks. This means you don't have much to worry about. For instance, take this example:

```
// `value` is inferred as a string
// `setValue` is inferred as (newValue: string) => void
const [value, setValue] = useState('')
```

This is one area in React and TypeScript that shines. It means you don't have to do much. Things *just work* and it's beautiful.

On the rare occasions where you need to initialize a hook with a null-ish value, you can make use of a generic and pass a union to correctly type your hook. See this instance:

```
type User = {
  email: string;
  id: string;
}

// the generic is the < >
// the union is the User | null
// together, TypeScript knows, "Ah, user can be User or null".
const [user, setUser] = useState<User | null>(null);
```

The other place where TypeScript shines with Hooks is with `userReducer`, where you can take advantage of discriminated unions. Here's a useful example:

```
type AppState = {};
type Action =
  | { type: "SET_ONE"; payload: string }
  | { type: "SET_TWO"; payload: number };

export function reducer(state: AppState, action: Action): AppState {
  switch (action.type) {
    case "SET_ONE":
      return {
        ...state,
        one: action.payload // `payload` is string
      };
    case "SET_TWO":
      return {
        ...state,
        two: action.payload // `payload` is number
      };
    default:
      return state;
  }
}
```

Source: `react-typescript-cheatsheet` *Hooks section*

The beauty here lies in the usefulness of discriminated unions. Notice how `Action` has a union of two similar-looking objects. The property `type` is a string literal. The difference between this and a type `string` is that the value must match the *literal* string defined in the type. This means your program is extra safe because a developer can only call an action that has a `type` key set to `"SET_ONE"` or `"SET_TWO"`.

As you can see, Hooks don't add much complexity to the nature of a React and TypeScript project. If anything, they lend themselves well to the duo.

Common Use Cases

This section is to cover the most common use cases where people stumble when using TypeScript with React. We hope by sharing this, you'll avoid the pitfalls and even share this knowledge with others.

Handling Form Events

One of the most common cases is correctly typing the `onChange` used on an input field in a form. Here's an example:

```
import React from 'react'

const MyInput = () => {
  const [value, setValue] = React.useState('')

  // The event type is a "ChangeEvent"
  // We pass in "HTMLInputElement" to the input
  function onChange(e: React.ChangeEvent<HTMLInputElement>) {
    setValue(e.target.value)
  }

  return <input value={value} onChange={onChange} id="input-example"/>
}
```

Extending Component Props

Sometimes you want to take component props declared for one component and extend them to use them on another component. But you might want to modify one or two. Well, remember how we looked at the two ways to type component props, types or interfaces? Depending on which you used determines how you extend the component props. Let's first look at the way using `type`:

```
import React from 'react';

type ButtonProps = {
    /** the background color of the button */
    color: string;
    /** the text to show inside the button */
    text: string;
```

```
}

type ContainerProps = ButtonProps & {
    /** the height of the container (value used with 'px') */
    height: number;
}

const Container: React.FC<ContainerProps> = ({ color, height, width, text }) => {
  return <div style={{ backgroundColor: color, height: `${height}px` }}>{text}</div>
}
```

If you declared your props using an `interface`, then we can use the keyword `extends` to essentially "extend" that interface but make a modification or two:

```
import React from 'react';

interface ButtonProps {
    /** the background color of the button */
    color: string;
    /** the text to show inside the button */
    text: string;
}

interface ContainerProps extends ButtonProps {
    /** the height of the container (value used with 'px') */
    height: number;
}

const Container: React.FC<ContainerProps> = ({ color, height, width, text }) => {
  return <div style={{ backgroundColor: color, height: `${height}px` }}>{text}</div>
}
```

Both methods solve the problem. It's up to you to decide which to use. Personally, extending an interface feels more readable, but ultimately, it's up to you and your team.

You can read more about both concepts in the TypeScript Handbook:

- Intersection Types
- Extending Interfaces

Third-party Libraries

Whether it's for a GraphQL client like Apollo or for testing with something like React Testing Library, we often find ourselves using third-party libraries in React and TypeScript projects. When this happens, the first thing you want to do is see if there's a `@types` package with the

TypeScript type definitions. You can do so by running:

```
#yarn
yarn add @types/<package-name>

#npm
npm install @types/<package-name>
```

For instance, if you're using Jest, you can do this by running:

```
#yarn
yarn add @types/jest

#npm
npm install @types/jest
```

This would then give you added type-safety whenever you're using Jest in your project.

The `@types` namespace is reserved for package type definitions. They live in a repository called DefinitelyTyped, which is partially maintained by the TypeScript team and partially the community.

Should these be saved as `dependencies` or `devDependencies` in my `package.json` ?

The short answer is "it depends". Most of the time, they can go under `devDependencies` if you're building a web application. However, if you're writing a React library in TypeScript, you may want to include them as `dependencies`.

There are a few answers to this on Stack Overflow, which you may check out for further information.

What happens if they don't have a @types package?

If you don't find a `@types` package on npm, then you essentially have two options:

1. Add a basic declaration file
2. Add a thorough declaration file

The first option means you create a file based on the package name and put it at the root. If, for instance, we needed types for our package `banana-js`, then we could create a basic declaration file called `banana-js.d.ts` at the root:

```
declare module 'banana-js';
```

This won't provide you type safety but it will unblock you.

A more thorough declaration file would be where you add types for the library/package:

```
declare namespace bananaJs {
    function getBanana(): string;
    function addBanana(n: number) void;
    function removeBanana(n: number) void;
}
```

If you've never written a declaration file, then we suggest you take a look at the guide in the official TypeScript Handbook.

Summary

Using React and TypeScript together in the best way takes a bit of learning due to the amount of information, but the benefits pay off immensely in the long run. In this guide, we covered configuration, components, props, hooks, common use cases, and third-party libraries. Although we could dive deeper into a lot of individual areas, this should cover the 80% needed to help you follow best practices.

If you'd like to get in touch, share feedback on this guide or chat about using the two technologies together, you can reach me on Twitter @jsjoeio.

Further Reading

If you'd like to dive deeper, here are some resources we suggest:

react-typescript-cheatsheet

A lot of these recommendations came straight from the react-typescript-cheatsheet. If you're looking for specific examples or details on anything React-TypeScript, this is the place to go. We welcome contributions as well!

Official TypeScript Handbook

Another fantastic resource is the TypeScript Handbook. This is kept up to date by the TypeScript team and provides examples and an in-depth explanation behind the inner workings of the language.

TypeScript Playground

Did you know you can test out React with TypeScript code right in the browser? All you have to do is import React. Here's a link to get you started.

Chapter

4

React vs Angular: An In-depth Comparison

Pavels Jelisejevs

Should I choose Angular or React? Each framework has a lot to offer and it's not easy to choose between them. Whether you're a newcomer trying to figure out where to start, a freelancer picking a framework for your next project, or an enterprise-grade architect planning a strategic vision for your company, you're likely to benefit from having an educated view on this topic.

To save you some time, let me tell you something up front: this guide won't give a clear answer on which framework is better. But neither will hundreds of other guides with similar titles. I can't tell you that, because the answer depends on a wide range of factors which make a particular technology more or less suitable for your environment and use case.

Since we can't answer the question directly, we'll attempt something else. We'll compare Angular and React, to demonstrate how you can approach the problem of comparing any two frameworks in a structured manner on your own and tailor it to your environment. You know, the old "teach a man to fish" approach. That way, when both are replaced by a BetterFramework.js in a year's time, you'll be able to re-create the same train of thought once more.

Where to Start?

Before you pick any tool, you need to answer two simple questions: "Is this a good tool per se?" and "Will it work well for my use case?" Neither of them mean anything on their own, so you always need to keep both of them in mind. All right, the questions might not be that simple, so we'll try to break them down into smaller ones.

Questions on the tool itself:

- How mature is it and who's behind it?
- What kind of features does it have?
- What architecture, development paradigms, and patterns does it employ?
- What is the ecosystem around it?

Questions for self-reflection:

- Will I and my colleagues be able to learn this tool with ease?
- Does it fit well with my project?
- What is the developer experience like?

Using this set of questions, you can start your assessment of any tool, and we'll base our comparison of React and Angular on them as well.

There's another thing we need to take into account. Strictly speaking, it's not exactly fair to compare Angular to React, since Angular is a full-blown, feature-rich framework, while React just a UI component library. To even the odds, we'll talk about React in conjunction with some of the libraries often used with it.

Maturity

An important part of being a skilled developer is being able to keep the balance between established, time-proven approaches and evaluating new bleeding-edge tech. As a general rule, you should be careful when adopting tools that haven't yet matured due to certain risks:

- The tool may be buggy and unstable.
- It might be unexpectedly abandoned by the vendor.
- There might not be a large knowledge base or community available in case you need help.
- Both React and Angular come from good families, so it seems that we can be confident in this regard.

React

React is developed and maintained by Facebook and used in their products, including Instagram and WhatsApp. It has been around for around since 2013, so it's not exactly new. It's also one of the most popular projects on GitHub, with more than 150,000 stars at the time of writing. Some of the other notable companies using React are Airbnb, Uber, Netflix, Dropbox, and Atlassian. Sounds good to me.

Angular

Angular has been around since 2016, making it slightly younger than React, but it's also not a new kid on the block. It's maintained by Google and, as mentioned by Igor Minar, even in 2018 was used in more than 600 hundred applications in Google such as Firebase Console, Google Analytics, Google Express, Google Cloud Platform and more. Outside of Google, Angular is used by Forbes, Upwork, VMWare, and others.

Features

Like I mentioned earlier, Angular has more features out of the box than React. This can be both a good and a bad thing, depending on how you look at it.

Both frameworks share some key features in common: components, data binding, and platform-agnostic rendering.

Angular

Angular provides a lot of the features required for a modern web application out of the box. Some of the standard features are:

- dependency injection
- templates, based on an extended version of HTML
- class-based components with lifecycle hooks
- routing, provided by `@angular/router`
- Ajax requests using `@angular/common/http`
- `@angular/forms` for building forms
- component CSS encapsulation
- XSS protection
- code splitting and lazy loading
- test runner, framework and utilities for unit-testing.

Some of these features are built into the core of the framework and you don't have an option not to use them. This requires developers to be familiar with features such as dependency injection to build even a small Angular application. Other features such as the HTTP client or forms are completely optional and can be added on an as-needed basis.

React

With React, you're starting with a more minimalistic approach. If we're looking at just React,

here's what we have:

- instead of classic templates, it has JSX, an XML-like language built on top of JavaScript
- class-based components with lifecycle hooks or simpler functional components
- state management using setState and hooks.
- XSS protection
- code splitting and lazy loading
- error handling boundaries
- utilities for unit-testing components

Out of the box, React does not provide anything for dependency injection, routing, HTTP calls, or advanced form handling. You are expected to choose whatever additional libraries to add based on your needs which can be both a good and a bad thing depending on how experienced you are with these technologies. Some of the popular libraries that are often used together with React are:

- React-router for routing
- Fetch (or axios) for HTTP requests
- a wide variety of techniques for CSS encapsulation
- Enzyme or React Testing Library for additional unit-testing utilities

The teams I've worked with have found the freedom of choosing your libraries liberating. This gives us the ability to tailor our stack to particular requirements of each project, and we haven't found the cost of learning new libraries that high.

Languages, Paradigms, and Patterns

Taking a step back from the features of each framework, let's see what kind of high-level concepts are popular with both frameworks.

React

Several important things come to mind when thinking about React: JSX, components, and hooks.

JSX

In contrast to most frameworks, React doesn't have a separate templating language. Instead of following a classical approach of separating markup and logic, React decided to combine them within components using JSX, an XML-like language that allows you to write markup directly in your JavaScript code.

While the merits of mixing markup with JavaScript might be debatable, it has an indisputable benefit: static analysis. If you make an error in your JSX markup, the compiler will emit an error instead of continuing in silence. This helps by instantly catching typos and other silly errors. Using JSX caught on in different projects—such as MDX, which allows using JSX in markdown files.

Components

In React you can define components using functions and classes.

Class components allow you to write your code using ES classes and structure the component logic into methods. They also allow you to use React's traditional lifecycle methods to run custom logic when a component is mounted, updated, unmounted, and so on. Even though this notation is easier to understand for people familiar with OOP programming, you need to be aware of all the subtle nuances that JS has—for example, how `this` works, and not forgetting to bind event handlers.

Functional components are defined as simple functions. They are often pure and provide a clear mapping between the input props and the rendered output. Functional code is usually less coupled and easier to reuse and test. Before the introduction of hooks, functional components could not be stateful and did not have an alternative to the lifecycle methods.

There's a trend among React developers to ditch class components in favor of simpler functional components, but with hooks being a newer feature, you'll usually see a mix of both approaches in larger React projects.

Hooks

Hooks are a new feature of React introduced in version 16.8. They are functions that allow you to class component state and lifecycle method features in functional components. There are two hooks provided by React: `useState` for managing the state, and `useEffect` for creating side effects—such as loading data or manually editing the DOM.

Hooks have been introduced to make functional components simpler and more composable. You can now split large functions into smaller atomic parts, allowing you to divide up related pieces of functionality—separating them from the rendering logic and reusing it in different components. Hooks are a cleaner alternative to using class components and other patterns, such as render functions and higher-order components—which can quickly get overly complicated.

React provides ways of structuring your application without involving a lot of complicated

abstraction layers. Utilizing functional components together with hooks allows you to write code that is simpler, more atomic, and reusable. Even though the notion of combining the code and templates might seem controversial, separating the presentation and application logic into different functions allows you to achieve similar results.

Angular

Angular has a few interesting things up its sleeve as well, starting from the basic abstractions such as components, services, and modules, to TypeScript, RxJS, and Angular Elements, as well as its approach to state management.

Main Concepts

Angular has a higher abstraction level than React, thus introducing more fundamental concepts to get familiar with. The main ones are:

- **components**: defined as specially decorated ES classes that are responsible for executing the application logic and rendering the template
- **services**: classes responsible for the implementation of business and application logic, used by components
- **modules**: essentially DI containers for wiring related components, services, pipes, and other entities together.

Angular makes heavy use of classes as well as concepts such as DI, which are less popular in the world of front-end development, but should be familiar to anyone with back-end development experience.

TypeScript

TypeScript is a new language built on top of JavaScript and developed by Microsoft. It's a superset of JavaScript ES2015 and includes features from newer versions of the language. You can use it instead of Babel to write state-of-the-art JavaScript. It also features an extremely powerful typing system that can statically analyze your code by using a combination of annotations and type inference.

There's also a more subtle benefit. TypeScript has been heavily influenced by Java and .NET, so if your developers have a background in one of these languages, they're likely to find TypeScript easier to learn than plain JavaScript (notice how we switched from the tool to your personal environment). Although Angular was the first major framework to actively adopt TypeScript, it's now getting traction in a lot of other projects as well, such as Deno (a TypeScript native runtime), Puppeteer, and TypeORM.

It's also possible (and wise) to use TypeScript together with React.

RxJS

RxJS is a reactive programming library that allows for more flexible handling of asynchronous operations and events. It's a combination of the Observer and Iterator patterns blended with functional programming. RxJS allows you to treat anything like a continuous stream of values and perform various operations on it such as mapping, filtering, splitting, or merging.

The library has been adopted by Angular in its HTTP module as well for some internal use. When you perform an HTTP request, it returns an Observable instead of the usual Promise. This approach opens the door for interesting possibilities, such as the ability to cancel a request, retry it multiple times, or work with continuous data streams such as WebSockets. But this is just the surface. To master RxJS, you'll need to know your way around different types of Observables, Subjects, as well as around a hundred methods and operators.

State Management

Similar to React, Angular components have a concept of a component state. Components can store data in their class properties and bind the values to their templates. If you want to share the state across the application, you can move it to a stateful service that can later be injected into the components. Since reactive programming and RxJS is a first-class citizen in Angular, it's common to make use of observables to recalculate parts of the state based on some input. This, however, can get tricky in larger applications since changing some variables can trigger a multi-directional cascade of updates that's difficult to follow. There are libraries for Angular that allow you to simplify state management at scale. We'll have a closer look at them later on.

Angular Elements

Angular elements provide a way to package Angular components as custom elements. Also known as web components, custom elements are a framework-agnostic standardized way to create custom HTML elements that are controlled by your JavaScript code. Once you define such an element and add it to the browser registry, it will automatically be rendered everywhere it's referenced in the HTML. Angular elements provide an API that creates the necessary wrapper to implement the custom component API and make it work with Angular's change detection mechanism. This mechanism can be used to embed other components or whole Angular applications into your host application, potentially written in a different framework with a different development cycle.

We've found TypeScript to be a great tool for improving the maintainability of our projects, especially those with a large codebase or complex domain/business logic. Code written in

TypeScript is more descriptive, and easier to follow and refactor. Even if you're not going with Angular, we suggest you consider it for your next JavaScript project. RxJS introduces new ways of managing data flow in your project, but does require you to have a good grasp of the subject. Otherwise, it can bring unwanted complexity to your project. Angular elements have the potential for reusing Angular components, and it's interesting to see how this plays out in the future.

Ecosystem

The great thing about open source frameworks is the number of tools created around them. Sometimes, these tools are even more helpful than the framework itself. Let's have a look at some of the most popular tools and libraries associated with each framework.

Angular

Angular CLI

A popular trend with modern frameworks is having a CLI tool that helps you bootstrap your project without having to configure the build yourself. Angular has Angular CLI for that. It allows you to generate and run a project with just a couple of commands. All of the scripts responsible for building the application, starting a development server, and running tests are hidden away from you in `node_modules`. You can also use it to generate new code during development and install dependencies.

Angular introduces an interesting new way of managing dependencies to your project. When using `ng add`, you can install a dependency and it will automatically be configured for usage. For example, when you run `ng add @angular/material`, Angular CLI downloads Angular Material from the npm registry and runs its install script that automatically configures your application to use Angular Material. This is done using Angular schematics. Schematics is a workflow tool that allows the libraries to make changes to your codebase. This means that the library authors can provide automatic ways of resolving backward-incompatible issues you might face when installing a new version.

Component Libraries

An important thing in using any JavaScript framework is being able to integrate them with a component library of your choice, to avoid having to build everything from scratch. Angular offers integrations with most of the popular component libraries as well as native libraries of its own. For example:

- ng-bootstrap for using Bootstrap widgets
- Material UI, for Google's Material Design components
- NG-ZORRO, a library of components implementing the Ant Design specification
- Onsen UI for Angular, a library of components for mobile applications
- PrimeNG, a collection of rich Angular components

State Management Libraries

If the native state management capabilities are not enough for you, there are several popular third-party libraries available in this area.

The most popular one is NgRx. It's inspired by React's Redux but also makes use of RxJS to watch and recalculate data in the state. Using NgRx can help you enforce an understandable unidirectional data flow, as well as reduce coupling in your code.

NGXS is another state management library inspired by Redux. In contrast to NgRx, NGXS strives to reduce boilerplate code by using modern TypeScript features and improving the learning curve and overall development experience.

Akita is a newer kid on the block, which allows us to keep the state in multiple stores, apply immutable updates, and use RxJS to query and stream the values.

Ionic Framework

Ionic is a popular framework for developing hybrid mobile applications. It provides a Cordova container that's nicely integrated with Angular and a pretty material component library. Using it, you can easily set up and build a mobile application. If you prefer a hybrid app over a native one, this is a good choice.

Angular universal

Angular universal is a project that bundles different tools to enable server-side rendering for Angular applications. It's integrated with Angular CLI and supports several Node.js frameworks, such as Express and Hapi, as well as with .NET core.

Augury

Augury is a browser extension for Chrome and Firefox that helps to debug Angular applications running in development mode. You can use it to explore your component tree, monitor change detection, and optimize performance issues.

Compodoc

Compodoc is a static documentation generator for Angular. Similar to other documentation generators, it can create static HTML documentation based on the TSDoc comments in your code. Compodoc, however, comes with convenient features specifically for Angular, such as browsing your module structure, routes, and classifying classes into components, services, and so on.

Ngx-admin

Ngx-admin is a popular framework for creating custom dashboards with Angular and using either Nebular or Angular Material as the component libraries.

There are plenty of other libraries and tools available in the Awesome Angular list.

React

Create React App

Create React App is a CLI utility for React to quickly set up new projects. Similar to Angular CLI, it allows you to generate a new project, run the app in development mode, or create a production bundle. It uses Jest for unit testing, supports application profiling using environment variables, back-end proxies for local development, TypeScript, Sass, PostCSS, and many other features.

Component Libraries

Similar to Angular, React has a wide variety of component libraries to choose from:

- ant-design
- Material UI
- react-bootstrap
- Semantic UI
- Onsen UI, optimized for mobile applications
- Blueprint, for creating desktop applications

State Management Libraries

The introduction of hooks has certainly shaken up state management in React. There are ongoing discussions if there even is a need for a third-party state management library. Even though hooks address the immediate need for working with the state, other libraries can still push this further by allowing us to use time-tested implementation patterns, lots of additional

libraries, and development tools.

Redux is a state management library inspired by Flux, but with some simplifications. The key idea of Redux is that the whole state of the application is represented by a single object, which is mutated by functions called reducers. Reducers themselves are pure functions and are implemented separately from the components. This enables better separation of concerns and testability.

MobX is an alternative library for managing the state of an application. Instead of keeping the state in a single immutable store, as Redux does, it encourages you to store only the minimal required state and derives the rest from it. It provides a set of decorators to define observables and observers and introduce reactive logic to your state.

Styling Libraries

Unlike Angular, React does not provide native CSS encapsulation capabilities, so you need to look for third-party solutions. There are numerous solutions to this problem, with no clear leader amongst them. Some of the popular ones are:

- Styled Components, a library that allows you to create React components with your styling applied, as well as style your components
- CSS Modules, which allows you to import CSS files and generate unique isolated class names to reference the styles
- Emotion, which combines the approaches of Styled Components and CSS Modules into a single library

PropTypes

PropTypes is an optional feature of React that allows you to introduce component runtime prop validation. In contrast to using static type checking with TypeScript, PropTypes will perform type checks when your application is actually running. This comes in especially handy when developing libraries when you can't be sure your clients are using TypeScript, even if you are. Since React 15.5, prop types have been moved to a separate prop-types library and are now completely optional. Considering its benefits, we advise you to use it to improve the reliability of your application.

React Native

React Native is a platform developed by Facebook for creating native mobile applications using React. Unlike Ionic, which produces a hybrid application, React Native produces a truly native UI. It provides a set of standard React components that are bound to their native counterparts. It

also allows you to create your components and bind them to native code written in Objective-C, Java, or Swift.

Next.js

Next.js is a framework for the server-side rendering of React applications. It provides a flexible way to completely or partially render your application on the server, return the result to the client, and continue in the browser. It tries to make the complex task of creating universal applications easier, so the setup is designed to be as simple as possible, with a minimal amount of new primitives and requirements for the structure of your project.

React Admin

React-admin is a framework for building CRUD-style SPA applications on top of existing REST or GraphQL APIs. It comes with handy features, such as a UI built with Material Design, internationalization, theming, data validation, and more.

UI Development Environments

A major trend in front-end development for the last couple of years has been the boom of development tools that allow you to develop, test, and document your component interactively and separately from the application. Storybook has established itself as one of the leaders in this area, with support for both React and Angular. However, there are other alternatives for React.

React Styleguidist, similarly to Storybook, allows you to create interactive documentation of your components. In contrast to Storybook, the generated UI looks more like an interactive readme than a separate set of stories. While Storybook shines as a development environment, Styleguidist is more a documentation tool.

We've also mentioned MDX in this guide. It allows you to spice up your Markdown files by adding interactive JSX components.

Testing Helpers

Testing UI components usually involves having to render them in a sandbox environment, simulate user interaction, and validate the output results. These routine tasks can be simplified by using the appropriate testing helpers. For Angular, this is the built-in TestBed. For React, there are two popular candidates: Enzyme and Testing Library.

Enzyme is the de-facto standard choice. It allows you to render your components into a full or shallow DOM as well as interact with the rendered component. It mostly follows a white box

testing approach, where your tests can reference some of the internals of the component like its children components, props, or state.

Testing Library follows a different approach and pushes you to interact with your components as a user would, without knowing the technical implementation. Tests created this way are usually less brittle and easier to maintain. Although it's most popular with React, the Testing Library is also available for Angular.

Gatsby

Gatsby is a static website generator that uses React.js. It allows you to use GraphQL to query the data for your websites defined in Markdown, YAML, JSON, external APIs, and popular content management systems.

React 360

React 360 is a library for creating virtual reality applications for browsers. It provides a declarative React API that's built on top of the WebGL and WebVR browser APIs, thus making it easier to create 360 VR experiences.

React Developer Tools

React Dev Tools is a browser extension for Chrome for debugging React applications that allows you to traverse the React component tree and see their props and state.

There are plenty of other libraries and tools available in the Awesome React list.

Adoption, Learning Curve and Development Experience

An important criterion for choosing a new technology is how easy it is to learn. Of course, the answer depends on a wide range of factors, such as your previous experience and a general familiarity with the related concepts and patterns. However, we can still try to assess the number of new things you'll need to learn to get started with a given framework. Now, if we assume that you already know ES6+, build tools, and all of that, let's see what else you'll need to understand.

React

With React, the first thing you'll encounter is JSX. It does seem awkward to write for some developers. However, it doesn't add that much complexity: just expressions, which are JavaScript, and special HTML-like syntax. You'll also need to learn how to write components, use props for configuration, and manage internal state. You don't need to learn a new template

syntax, since all of this is plain JavaScript. While React supports class-based components, with the introduction of hooks, functional development is getting more popular. This will require you to understand some basic functional development patterns.

The official tutorial is an excellent place to start learning React. Once you're done with that, get familiar with the router. The React Router might be slightly complex and unconventional, but nothing to worry about. Depending on the size, complexity, and requirements of your project, you'll need to find and learn some additional libraries, and this might be the tricky part. But after that, everything should be smooth sailing.

We were genuinely surprised at how easy it was to get started using React. Even people with a back-end development background and limited experience in front-end development were able to catch up quickly. The error messages you might encounter along the way are usually clear and provide explanations on how to resolve the underlying problem.

The downside is that you'll need to invest time into choosing the libraries to support your development activities. Given how many of them are there, this can pose a challenge. But this can be done along with your development activities after you've gotten comfortable with the framework itself.

Although TypeScript is not required for React, we strongly recommend you assess and adopt it in your projects.

Angular

Learning Angular will introduce you to more new concepts than React. First of all, you'll need to get comfortable with TypeScript. For developers with experience in statically typed languages such as Java or .NET, this might be easier to understand than JavaScript, but for pure JavaScript developers, this might require some effort. To start your journey, we can recommend the Tour of Heroes tutorial.

The framework itself is rich in topics to learn, starting from basic ones such as modules, dependency injection, decorators, components, services, pipes, templates, and directives, to more advanced topics such as change detection, zones, AoT compilation, and Rx.js. These are all covered in the documentation. Rx.js is a heavy topic on its own and is described in much detail on the official website. While relatively easy to use on a basic level, it gets more complicated when moving on to advanced topics.

All in all, we noticed that the entry barrier for Angular is higher than for React. The sheer number of new concepts may be overwhelming to newcomers. And even after you've started, the experience might be a bit rough, since you need to keep in mind things like Rx.js subscription

management, change detection performance, and bananas in a box (yes, this is a piece of actual advice from the documentation). We often encountered error messages that are too cryptic to understand, so we had to google them and pray for an exact match.

It might seem that we favor React here, and we definitely do. We've had experience onboarding new developers to both Angular and React projects of comparable size and complexity, and somehow with React it always went smoother. But, as I said earlier, this depends on a broad range of factors and might work differently for you.

Popularity and Community Feedback

Both frameworks are highly popular and have communities of followers and advocates. There are also numerous articles suggesting you use one or the other technology, mostly highlighting their positive sides. Let's see if we can find a more objective way to represent the popularity of each project and developer satisfaction.

The npm download statistics shows almost five times more downloads for React than Angular. Google trends also reports more searches for React worldwide.

The 2019 State of JavaScript report lists React as the most popular front-end framework, with Angular being second to last, preceding only Ember. 71% of the participants said that they've used React and would use it again. Only 21% of the participants said the same about Angular, and 35% said that they've used Angular and would *not* use it again (only 8% said that about React).

The Hacker News Hiring Trends for 2019 lists React as the most wanted technology amongst employees for more than 31 months in a row.

Stack Overflow lists React as the second most popular framework after jQuery. Angular is listed as the third one. Their Most Loved, Dreaded, and Wanted Web Frameworks report paints a similar picture to the others.

The State of Developer Ecosystem 2020 report by Jet Brains confirms React's position as the most popular front-end framework.

Making a Decision

You might have already noted that each framework has its own set of capabilities, both with their good and bad sides. But this analysis has been done outside of any particular context and thus doesn't provide an answer on which framework should you choose. To decide on that, you'll need to review it from the perspective of your project. This is something you'll need to do on your own.

To get started, try answering these questions about your project and when you do, match the answers against what you've learned about the two frameworks. This list might not be complete, but should be enough to get you started:

- How big is the project?
- How long is it going to be maintained?
- Is all of the functionality clearly defined in advance or are you expected to be flexible?
- If all of the features are already defined, what capabilities do you need?
- Are the domain model and business logic complex?
- What platforms are you targeting? Web, mobile, desktop?
- Do you need server-side rendering? Is SEO important?
- Will you be handling a lot of real-time event streams?
- How big is your team?
- How experienced are your developers and what is their background?
- Are there any ready-made component libraries that you would like to use?

If you're starting a big project and you'd like to minimize the risk of making a bad choice, consider creating a proof-of-concept product first. Pick some of the key features of the projects and try to implement them in a simplistic manner using one of the frameworks. PoCs usually don't take a lot of time to build, but they'll give you some valuable personal experience on working with the framework and allow you to validate the key technical requirements. If you're satisfied with the results, you can continue with full-blown development. If not, failing fast will save you a lot of headaches in the long run.

One Framework to Rule Them All?

Once you've picked a framework for one project, you'll get tempted to use the exact same tech stack for your upcoming projects. Don't. Even though it's a good idea to keep your tech stack consistent, don't blindly use the same approach every time. Before starting each project, take a moment to answer the same questions once more. Maybe for the next project, the answers will be different or the landscape will change. Also, if you have the luxury of doing a small project with a non-familiar tech stack, go for it. Such experiments will provide you with invaluable experience. Keep your mind open and learn from your mistakes. At some point, a certain technology will just feel natural and right.

Chapter

Getting Started with React Native

5

Wern Ancheta

With the ever-increasing popularity of smartphones, developers are looking into solutions for building mobile applications. For developers with a web background, frameworks such as Cordova and Ionic, React Native, NativeScript, and Flutter allow us to create mobile apps with languages we're already familiar with: HTML, XML, CSS, and JavaScript.

In this guide, we'll take a closer look at React Native. You'll learn the absolute basics of getting started with it. Specifically, we'll cover the following:

- what React Native is
- what Expo is
- how to set up an React Native development environment using Expo
- how to create an app with React Native

Prerequisites

This tutorial assumes that you're coming from a web development background. The minimum requirement for you to be able to confidently follow this tutorial is to know HTML, CSS, and JavaScript. You should also know how to install software on your operating system and work with the command line. We'll also be using some ES6 syntax, so it would help if you know basic ES6 syntax as well. Knowledge of React is helpful but not required.

What is React Native?

React Native is a framework for building apps that work on both Android and iOS. It allows you to create real native apps using JavaScript and React. This differs from frameworks like Cordova, where you use HTML to build the UI, which will then just be displayed within the device's integrated mobile browser (WebView). React Native has built-in components which are compiled to native UI components, while your JavaScript code is executed through a virtual machine. This makes React Native more performant than Cordova.

Another advantage of React Native is its ability to access native device features. There are many plugins which you can use to access native device features, such as the camera and various device sensors. If you're in need of a platform-specific feature that hasn't been implemented yet, you can also build your own native modules—although that will require you to have considerable knowledge of the native platform you want to support (Java or Kotlin for Android, and Objective C or Swift for iOS).

If you're coming here and you're new to React, you might be wondering what it is. React is a JavaScript library for the Web for building user interfaces. If you're familiar with MVC, it's basically the View in MVC. React's main purpose is to allow developers to build reusable UI components. Examples of these components include buttons, sliders, and cards. React Native

took the idea of building reusable UI components and brought it into mobile app development.

What is Expo?

Before coming here, you might have heard of Expo. It's even mentioned in the official React Native docs, so you might be wondering what it is.

In simple terms, Expo allows you to build React Native apps without the initial headache that comes with setting up your development environment. It only requires you to have Node installed on your machine, and the Expo client app on your device or emulator.

But that's just how Expo is initially sold. In reality, it's much more than that. Expo is actually a platform that gives you access to tools, libraries and services for building Android and iOS apps faster with React Native. Expo comes with an SDK which includes most of the APIs you can ask for in a mobile app development platform:

- Camera
- ImagePicker
- Facebook
- GoogleSignIn
- Location
- MapView
- Permissions
- Push Notifications
- Video

Those are just few of the APIs you get access to out of the box if you start building React Native apps with Expo. Of course, these APIs are available to you as well via native modules if you develop your app using the standard React Native setup.

Plain React Native or Expo?

The real question is which one to pick—plain React Native or Expo? There's really no right or wrong answer. It all depends on the context and what your needs are. But I guess it's safe to assume that you're reading this tutorial because you want to quickly get started with React Native. So I'll go ahead and recommend that you start out with Expo. It's fast, simple, and easy to set up. You can dive right into tinkering with React Native code and get a feel of what it has to offer in just a couple of hours.

But as you begin to grasp the different concepts, and as the need for different native features arises, you might find that Expo is kind of limiting. Yes, it has a lot of native features available, but

not all the native modules that are available to standard React Native projects are supported.

Projects Closing the Gap

Projects like unimodules are beginning to close the gap between standard React Native projects and Expo projects, as it allows developers to create native modules that work for both React Native and ExpoKit.

Setting Up the React Native Development Environment

To quickly get started with React Native, the recommended method is to set up Expo.

The only prerequisite of setting up Expo is that you need to have Node.js installed in your machine. To do this, you can either head to the official Node download page and grab the relevant binaries for your system, or you can use a version manager, which allows you to install multiple versions of Node and switch between them at will.

Once you have Node.js installed, install the Expo CLI. This is used for creating, serving, packaging, and publishing projects:

```
npm install -g expo-cli
```

Next, install Yarn, the preferred package manager for Expo:

```
npm install -g yarn
```

That's really all there is to it! The next step is to download the Expo client App for Android or iOS. Note that this is the only way you can run Expo apps while you're still in development. When you're ready to ship the app, you can follow this guide to create standalone binaries for iOS and Android which can be submitted to the Apple App Store and Google Play Store.

What We'll Be Building

Now that your development environment is set up, we can look at the app we're going to create—a Pokémon search app. It will allow the user to type the name of a Pokémon into an input box, before fetching the Pokémon's details from an external API and displaying them to the user.

Here's what the finished thing will look like:

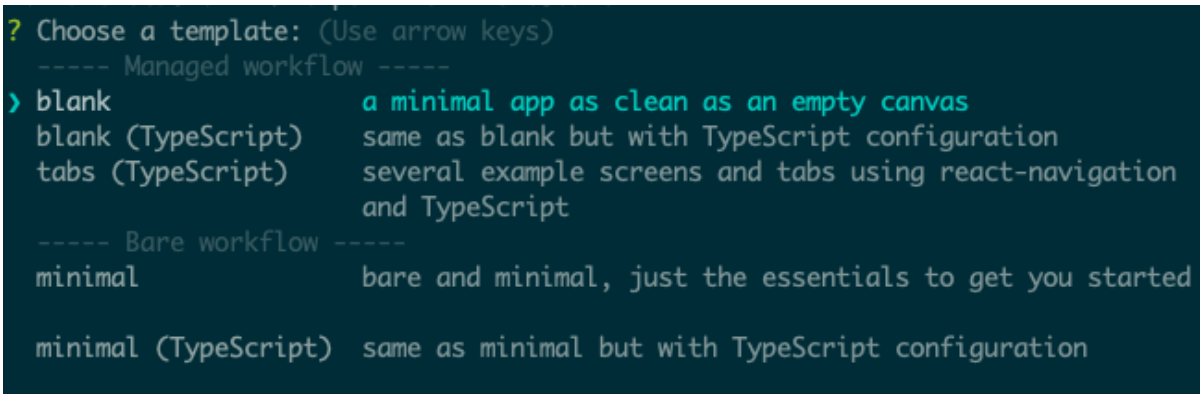
As ever, you can find the source code for this in our GitHub repo.

Bootstrapping the App

On your terminal, execute the following command to create a new React Native project using Expo:

```
expo init RNPokeSearch
```

Under **Managed Workflow**, select **blank**. By default, this will install the dependencies using Yarn.



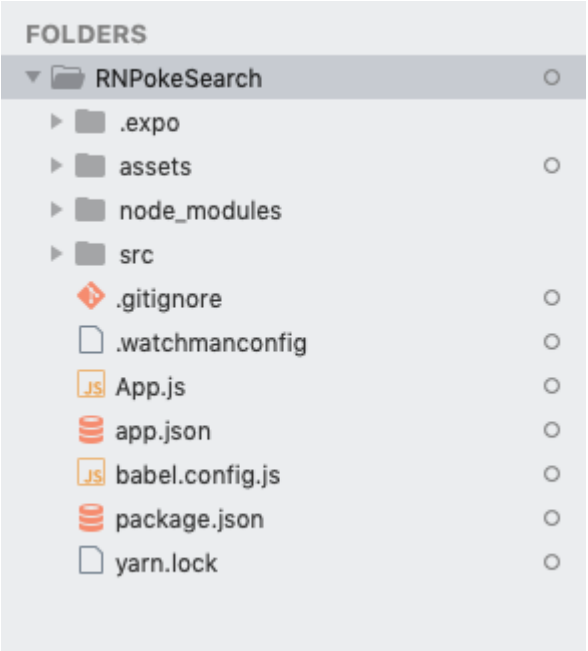
You might be asking what this **Managed workflow** and **Bare workflow** is. These are the two types of workflows that Expo supports. With a managed workflow, you only have to deal with JavaScript and Expo manages everything for you. While in Bare workflow, you have full control over the native code. It gives you the same freedom as the React Native CLI, but with the added bonus of Expo's libraries and services. You can visit this managed vs bare intro page if you want to learn more about workflows in Expo.

Just like in a web environment, you can install libraries to easily implement different kinds of functionality in React Native. Once the project is created, we need to install a couple of dependencies: pokemon and axios. The former is used for verifying if the text entered in the search box is a real Pokémon name, while axios is used to make an HTTP request to the API that we're using, namely the PokeAPI:

```
yarn add pokemon axios
```

React Native Project Directory Structure

Before we proceed to coding, let's first take a look at the directory structure of a React Native project created with Expo:



Here's a breakdown of the most important files and folders that you need to remember:

- `App.js` : the main project file. This is where you'll start developing your app. Any changes you make to this file will be reflected on the screen.
- `src` : acts as the main folder which stores all the source code related to the app itself. Note that this isn't included in the default project created by Expo CLI. The name of this folder can be anything. Some people use `app` as well.
- `assets` : this is where the app assets such as icons and splash screens are stored.
- `package.json` : where the name and versions of the libraries you installed for this project are added.
- `node_modules` : where the libraries you installed are stored. Note that this already contains a lot of folders before you installed the two libraries earlier. This is because React Native also has its own dependencies. The same is true for all the other libraries you install.

Don't mind the rest of the folders and files for now, as we won't be needing them when just getting started.

Running the App

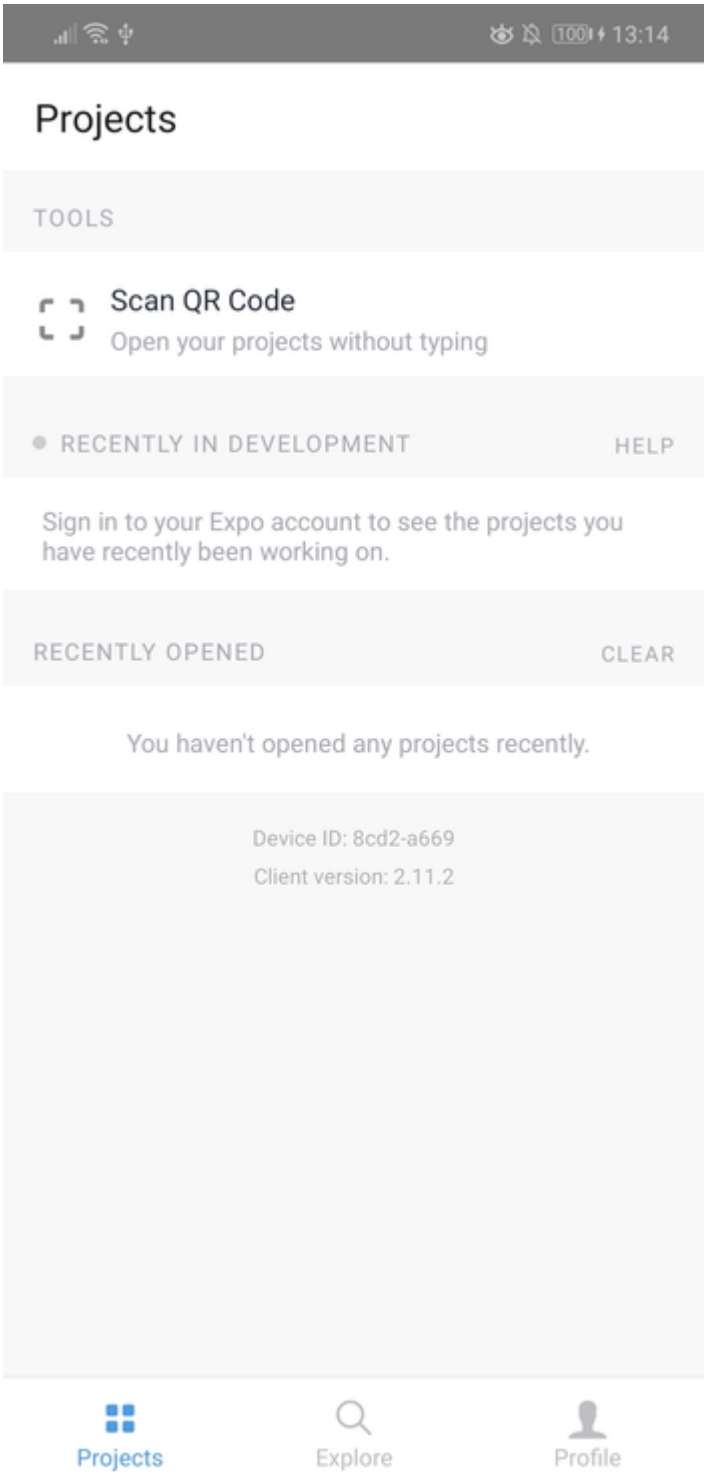
At this point, you can now run the app by executing the command below. Make sure that you've already installed the corresponding Expo client (Android or iOS) for your phone and that it's connected to the same network as your computer before doing so. If you don't have an Android or iOS device you can test with, you can use the Android Studio Emulator or the iOS simulator so

you can run the app on your machine:

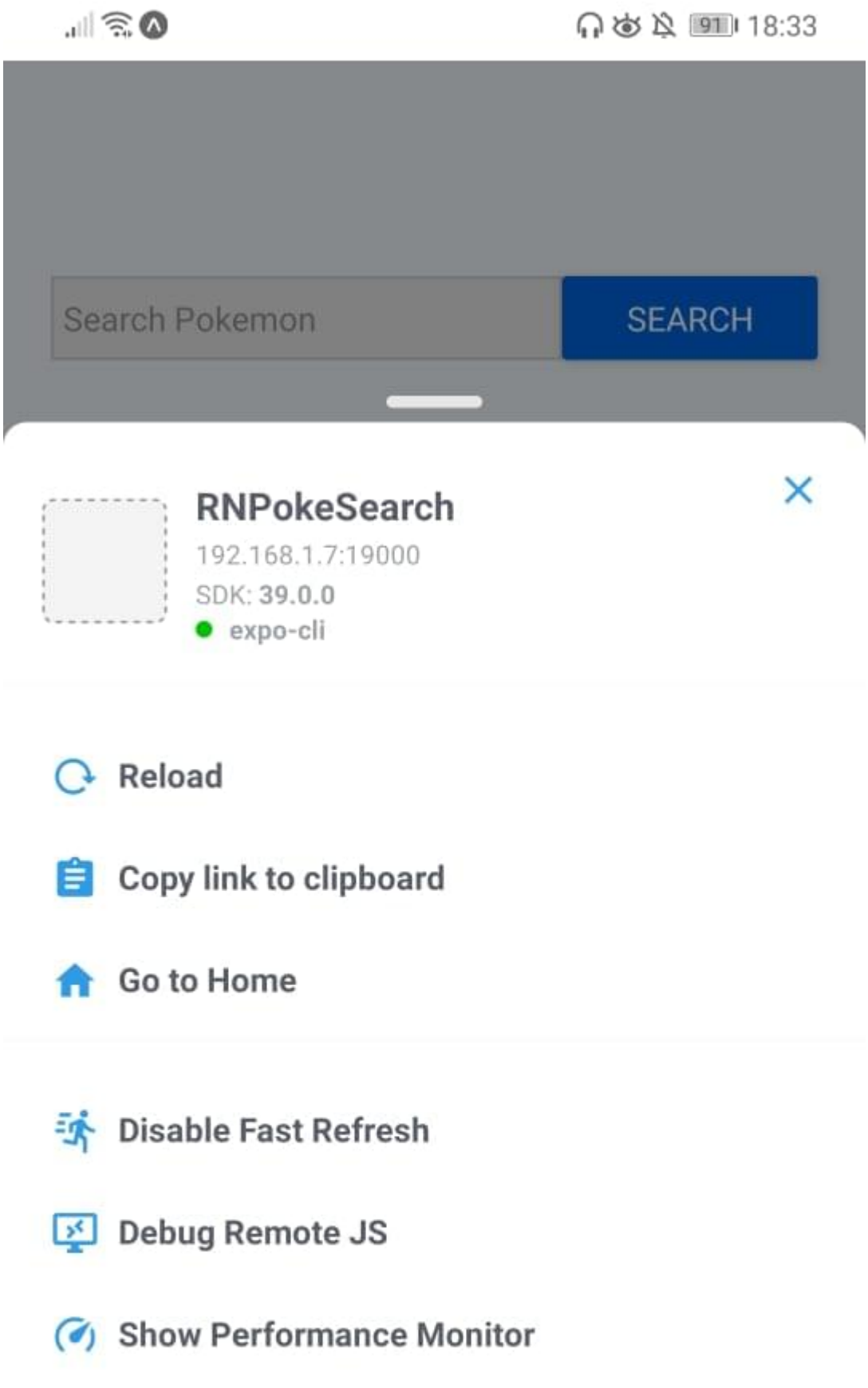
```
yarn start
```

Once it's running, it will display a QR code:

Open your Expo client app, and in the projects tab click on **Scan QR Code**. This will open the app on your Android or iOS device. If you have an emulator running, you can either press `i` to run it on the iOS simulator or `a` to run it on the Android emulator.



If you're testing on a real device, shake it so the developer menu will show up.


18:33
Search Pokemon
SEARCH
RNPokeSearch
192.168.1.7:19000
SDK: 39.0.0
expo-cli
Reload
Copy link to clipboard
Go to Home
Disable Fast Refresh
Debug Remote JS
Show Performance Monitor

Make sure that **Fast Refresh** is enabled. This allows you to automatically reload the changes that you make on your components.

Coding the App

Expo has many built-in components which you can use to accomplish what you want. Simply dig through the API documentation and you'll find information on how to implement what you need. In most cases, you either need a specific UI component or an SDK which works with a service you plan on using. More often than not, here's what your workflow is going to look like:

1. Look for an existing package which implements what you want.
2. Install it.
3. Link it. This is only necessary if you're on Expo's bare workflow and the package you've installed has a corresponding native dependency.
4. Use it in your project.

Now that you've set up your environment and learned a bit about the workflow, we're ready to start coding the app.

First, let's scaffold out the files we'll need. These are `src/Main.js`, as well as `src/components/Pokemon.js`. The `Main` component will hold the code to display the search input and query the API, whereas the `Pokemon` component will be used to display the returned Pokémon data:

```
mkdir -p src/components
touch src/Main.js
touch src/components/Pokemon.js
```

Add some dummy content to both files:

```
// src/Main.js
import React, { Component } from 'react';

export default class Main extends Component {
  render() {
    return null;
  }
}
```

```
// src/components/Pokemon.js
```

```
import  React  from  'react';

const  Pokemon  =  ()  =>  null;
```

Next, replace the contents of the `App.js` file with the following code:

```
import React from 'react';
import Main from './src/Main';

function App() {
  return <Main />;
}

export default App;
```

The first line in the code above code imports React. You need to import this class any time you want create a component.

The second line is where we import the custom `Main` component. We'll fill this in later, but for now, know that this is where we'll put the majority of our code.

After that, we create the component by creating a new function. All this function does is return the `Main` component.

Lastly, we export the class so that it can be imported and rendered by Expo.

Next, in `src/Main.js` file and add the following:

```
// src/Main.js
import React, { Component } from 'react';
import {
  SafeAreaView,
  View,
  Text,
  TextInput,
  Button,
  Alert,
  StyleSheet,
  ActivityIndicator,
} from 'react-native';
```

The second line imports the components that are built into React Native. Here's what each one does:

- `SafeAreaView` : for rendering content within the safe area boundaries of a device. This automatically adds a padding that wraps its content so that it won't be rendered on camera notches and sensor housing area of a device.
- `View` : a fundamental UI building block. This is mainly used as a wrapper for all the other components so they're structured in such a way that you can style them with ease. Think of it as the equivalent of `<div>` . If you want to use Flexbox, you have to use this component.
- `Text` : for displaying text.
- `TextInput` : the UI component for inputting text. This text can be plain text, email, password, or a number pad.
- `Button` : for showing a platform-specific button. This component looks different based on the platform it runs on. If it's Android, it uses Material Design. If it's iOS, it uses Cupertino.
- `Alert` : for showing alerts and prompts.
- `ActivityIndicator` : for showing a loading animation indicator.
- `StyleSheet` : for defining the component styles.

Next, import the libraries we installed earlier:

```
import axios from 'axios';
import pokemon from 'pokemon';
```

As well as the custom `Pokemon` component used for displaying Pokémon data:

```
import Pokemon from "./components/Pokemon";
```

Resolving Components

If Expo is unable to resolve the `Pokemon` (or any other) component, try restarting the server.

Because getting the required Pokémon data involves making two API requests, we have to set the API's base URL as a constant:

```
const POKE_API_BASE_URL = 'https://pokeapi.co/api/v2';
```

Next, define the component class and initialize its state:

```
export default class Main extends Component {
  constructor(props) {
```

```
    super(props)

    this.state = {
      isLoading: false, // decides whether to show the activity indicator or not
      searchInput: '', // the currently input text
      name: '', // Pokémon name
      pic: '', // Pokémon image URL
      types: [], // Pokémon types array
      desc: '', // Pokémon description
    };
  }

  render() {
    return null;
  }
}
```

In the code above, we're defining the main component of the app. You can do this by defining an ES6 class and having it extend React's `Component` class. This is another way of defining a component in React. In the `App.js` file, we created a *functional component*. This time we're creating a *class-based component*.

The main difference between the two is that **functional components** are used for presentation purposes only. Functional components have no need to keep their own state because all the data they require is just passed to them via props. On the other hand, **class-based components** maintain their own state and they're usually the ones passing data to functional components. Note that this is the traditional way of creating components in React. A more modern approach would be to stick with a functional component and use the state hook to manage the state—though in this tutorial we're just going to keep things simple and stick with a class-based component.

If you want to learn more about the difference between functional and class-based components, read the tutorial "Functional vs Class-Components in React".

Going back to the code, we're initializing the state inside our component. You define it as a plain JavaScript object. Any data that goes into the state should be responsible for changing what's rendered by the component. In this case, we put in `isLoading` to control the visibility of the activity indicator and `searchInput` to keep track of the input value in the search box.

This is an important concept to remember. React Native's built-in components, and even the custom components you create, accept properties that control the following:

- what's displayed on the screen (data source)

- how they present it (structure)
- what it looks like (styles)
- what actions to perform when user interacts with it (functions)

We'll go through those properties in more detail in the next section. For now, know that the values of those properties are usually updated through the state.

The rest of the state values are for the Pokémon data. It's a good practice to set the initial value with the same type of data you're expecting to store later on—as this serves as documentation as well.

Structuring and Styling Components

Let's return to the component class definition. When you extend React's `Component` class, you have to define a `render()` method. This contains the code for returning the component's UI and it's made up of the React Native components we imported earlier.

Each component has its own set of props. These are basically attributes that you pass to the component to control a specific aspect of it. In the code below, most of them have the `style` prop, which is used to modify the styles of a component. You can pass any data type as a prop. For example, the `onChangeText` prop of the `TextInput` is a function, while the `types` prop in the `Pokemon` is an array of objects. Later on in the `Pokemon` component, you'll see how the props will be used.

Replace the `render()` method in `Main.js` with the following:

```
render() {
  const { name, pic, types, desc, searchInput, isLoading } = this.state; // extract the Pokémon data

  return (
    <SafeAreaView style={styles.wrapper}>
      <View style={styles.container}>
        <View style={styles.headContainer}>
          <View style={styles.textInputContainer}>
            <TextInput
              style={styles.textInput}
              onChangeText={(searchInput) => this.setState({ searchInput })}
              value={this.state.searchInput}
              placeholder="Search Pokémon"
            />
          </View>
          <View style={styles.buttonContainer}>
            <Button
```

```
              onPress={this.searchPokemon}
              title="Search"
              color="#0064e1"
            />
          </View>
        </View>

        <View style={styles.mainContainer}>
          {isLoading && <ActivityIndicator size="large" color="#0064e1" />}

          {!isLoading && (
            <Pokemon name={name} pic={pic} types={types} desc={desc} />
          )}
        </View>
      </View>
    </SafeAreaView>
  );
}
```

Breaking down the code above, we first extract the state data:

```
const { name, pic, types, desc, searchInput, isLoading } = this.state;
```

Next, we return the component's UI, which follows this structure:

```
SafeAreaView.wrapper;
  View.container;
    View.headContainer;
      View.textInputContainer;
        TextInput;
      View.buttonContainer;
        Button;
    View.mainContainer;
      ActivityIndicator;
        Pokemon;
```

The above structure is optimized for using Flexbox. Go ahead and define the component styles at the bottom of the file:

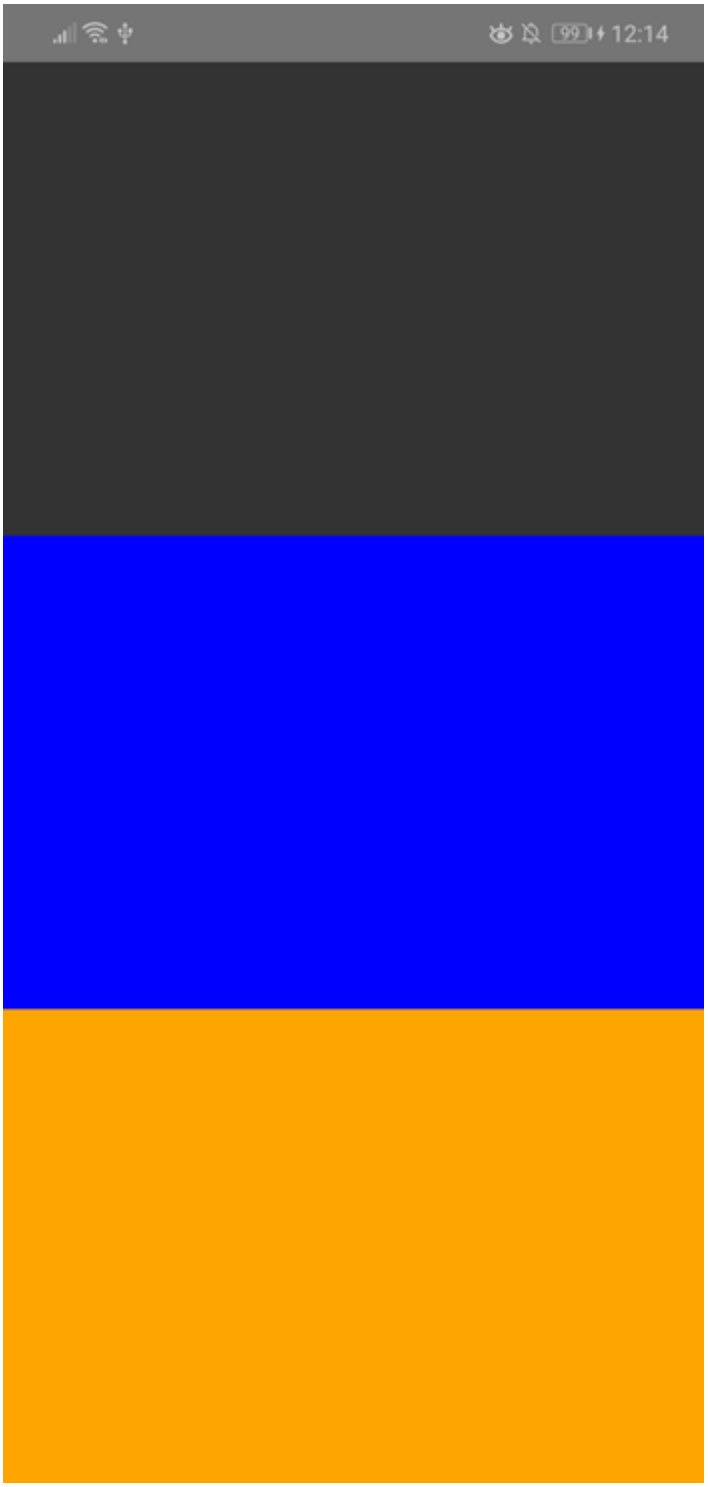
```
const styles = StyleSheet.create({
  wrapper: {
    flex: 1,
  },
  container: {
    flex: 1,
    padding: 20,
```

```
    backgroundColor: '#F5FCFF',
  },
  headContainer: {
    flex: 1,
    flexDirection: 'row',
    marginTop: 100,
  },
  textInputContainer: {
    flex: 2,
  },
  buttonContainer: {
    flex: 1,
  },
  mainContainer: {
    flex: 9,
  },
  textInput: {
    height: 35,
    marginBottom: 10,
    borderColor: '#ccc',
    borderWidth: 1,
    backgroundColor: '#eaeaea',
    padding: 5,
  },
});
```

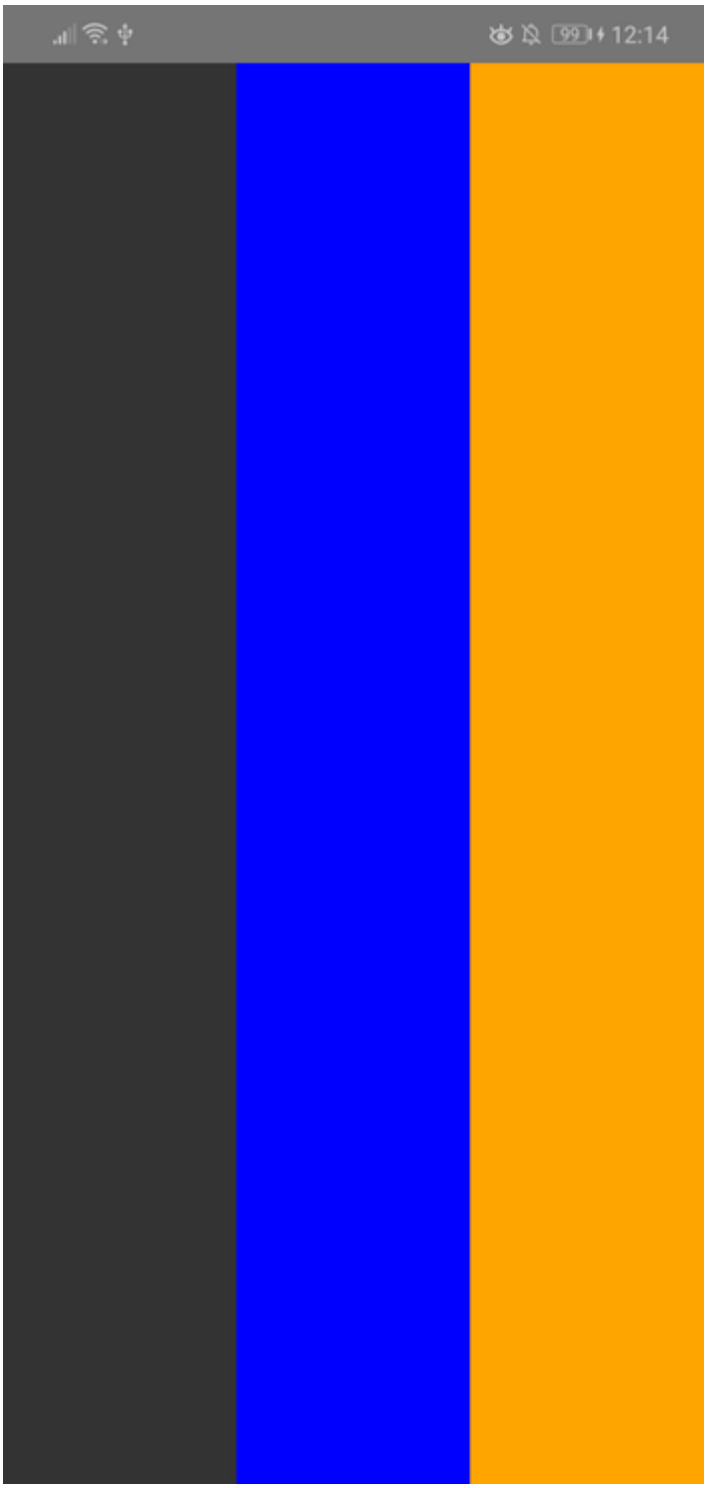
In React Native, you define styles by using `StyleSheet.create()` and passing in the object that contains your styles. These style definitions are basically JavaScript objects, and they follow the same structure as your usual CSS styles:

```
element: {
  property: value;
}
```

The `wrapper` and `container` is set to `flex: 1`, which means it will occupy the entirety of the available space because they have no siblings. React Native defaults to `flexDirection: 'column'`, which means it will lay out the flex items vertically.



In contrast, (`flexDirection: 'row'`) lays out items horizontally.



It's different for `headContainer`, because even though it's set to `flex: 1`, it has `mainContainer` as its sibling. This means that `headContainer` and `mainContainer` will both share the same

space. `mainContainer` is set to `flex: 9` so it will occupy the majority of the available space (around 90%), while `headContainer` will only occupy about 10%.

Let's move on to the contents of `headContainer`. It has `textInputContainer` and `buttonContainer` as its children. It's set to `flexDirection: 'row'`, so that its children will be laid out horizontally. The same principle applies when it comes to space sharing: `textInputContainer` occupies two thirds of the available horizontal space, while `buttonContainer` only occupies one third.

The rest of the styles are pretty self explanatory when you have a CSS background. Just remember to omit `-` and set the following character to uppercase. For example, if you want to set `background-color`, the React Native equivalent is `backgroundColor`.

CSS Supported in React Native

Not all CSS properties that are available on the Web are supported in React Native. For example, things like floats or table properties aren't supported. You can find the list of supported CSS properties in the docs for View and Text components. Someone has also compiled a React Native Styling Cheat Sheet, *and* there's a style section in the documentation for a specific React Native component that you want to use. For example, here are the style properties that you can use for the Image component.

Event Handling and Updating the State

Let's now break down the code for the `TextInput` and `Button` components. In this section, we'll talk about event handling, making HTTP requests, and updating the state in React Native.

Let's start by examining the code for `TextInput`:

```
<TextInput
  style={styles.textInput}
  onChangeText={(searchInput) => this.setState({ searchInput })}
  value={this.state.searchInput}
  placeholder="Search Pokémon"
/>
```

In the above code, we're setting the function to execute when the user inputs something in the component. Handling events like this are similar to how they're handled in the DOM: you simply pass the event name as a prop and set its value to the function you wish to execute. In this case, we're inlining it because we're just updating the state. The value input by the user is automatically

passed as an argument to the function you supply, so all you have to do is update the state with that value. Don't forget to set the value of the `TextInput` to that of the state variable. Otherwise, the value input by the user won't show as they type.

Next, we move on to the `Button` component. Here, we're listening for the `onPress` event:

```
<Button onPress={this.searchPokemon} title="Search" color="#0064e1" />
```

Once pressed, the `searchPokemon()` function is executed. Add this function right below the `render()` method. This function uses the `async...await` pattern because performing an HTTP request is an asynchronous operation. You can also use Promises, but to keep our code concise, we'll stick with async/await instead. If you're not familiar with this tecnique, be sure to read "Flow Control in Modern JS".

```
// src/Main.js
import React, { Component } from 'react';
...
export default class Main extends Component {
  ...

  render() { ... }

  searchPokemon = async () => {
    try {
      const pokemonID = pokemon.getId(this.state.searchInput); // check if the Pokémon is valid

      this.setState({
        isLoading: true, // show the loader while request is being performed
      });

      const { data: pokemonData } = await axios.get(
        `${POKE_API_BASE_URL}/pokemon/${pokemonID}`
      );
      const { data: pokemonSpecieData } = await axios.get(
        `${POKE_API_BASE_URL}/pokemon-species/${pokemonID}`
      );

      const { name, sprites, types } = pokemonData;
      const { flavor_text_entries } = pokemonSpecieData;

      this.setState({
        name,
        pic: sprites.front_default,
        types: this.getTypes(types),
        desc: this.getDescription(flavor_text_entries),
        isLoading: false, // hide loader
```

```
      });
    } catch (err) {
      Alert.alert('Error', 'Pokémon not found');
    }
  };
}

const styles = StyleSheet.create({ ... });
```

Breaking down the code above, we first check if the entered Pokémon name is valid. If it's valid, the National Pokedex ID (if you open the link, that's the number on top of the Pokémon name) is returned and we supply it as a parameter for the HTTP request. The request is made using axios' `get()` method, which corresponds to an HTTP GET request. Once the data is available, we extract what we need and update the state.

Here's the `getTypes()` function. All it does is reassign the `slot` and `type` properties of the Pokémon types to `id` and `name`:

```
getTypes = (types) =>
  types.map(({ slot, type }) => ({
    id: slot,
    name: type.name,
  }));
```

Here's the `getDescription()` function. This finds the first English version of the `flavor_text`:

```
getDescription = (entries) =>
    entries.find((item) => item.language.name === 'en').flavor_text;
```

Add them after the `searchPokemon` function, like so:

```
import React, { Component } from 'react';
...
export default class Main extends Component {
  ...

  render() { ... }

  searchPokemon = async () => { ... };
  getTypes = (types) => types.map( ... );
  getDescription = (entries) => entries.find( ... );
}

const styles = StyleSheet.create({ ... });
```

Pokémon Component

Now that our app is fetching data from the API, it's time to expand the `Pokemon` component we stubbed out earlier, so that we can display said data. Open the `src/components/Pokemon.js` file and replace the contents with the following:

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

const Pokemon = ({ name, pic, types, desc }) => {
  if (!name) {
    return null;
  }

  return (
    <View style={styles.mainDetails}>
      <Image source={{ uri: pic }} style={styles.image} resizeMode="contain" />
      <Text style={styles.mainText}>{name}</Text>

      <FlatList
        columnWrapperStyle={styles.types}
        data={types}
        numColumns={2}
        keyExtractor={(item) => item.id.toString()}
        renderItem={({ item }) => (
          <View style={[styles[item.name], styles.type]}>
            <Text style={styles.typeText}>{item.name}</Text>
          </View>
        )}
      />

      <View style={styles.description}>
        <Text>{desc}</Text>
      </View>
    </View>
  );
};

//
const styles = StyleSheet.create({
  mainDetails: {
    padding: 30,
    alignItems: 'center',
  },
  image: {
    width: 100,
    height: 100,
  },
```

```
mainText: {
  fontSize: 25,
  fontWeight: 'bold',
  textAlign: 'center',
},
description: {
  marginTop: 20,
},
types: {
  flexDirection: 'row',
  marginTop: 20,
},
type: {
  padding: 5,
  width: 100,
  alignItems: 'center',
},
typeText: {
  color: '#fff',
},
normal: {
  backgroundColor: '#8a8a59',
},
fire: {
  backgroundColor: '#f08030',
},
water: {
  backgroundColor: '#6890f0',
},
electric: {
  backgroundColor: '#f8d030',
},
grass: {
  backgroundColor: '#78c850',
},
ice: {
  backgroundColor: '#98d8d8',
},
fighting: {
  backgroundColor: '#c03028',
},
poison: {
  backgroundColor: '#a040a0',
},
ground: {
  backgroundColor: '#e0c068',
},
flying: {
  backgroundColor: '#a890f0',
```

```
  },
  psychic: {
    backgroundColor: '#f85888',
  },
  bug: {
    backgroundColor: '#a8b820',
  },
  rock: {
    backgroundColor: '#b8a038',
  },
  ghost: {
    backgroundColor: '#705898',
  },
  dragon: {
    backgroundColor: '#7038f8',
  },
  dark: {
    backgroundColor: '#705848',
  },
  steel: {
    backgroundColor: '#b8b8d0',
  },
  fairy: {
    backgroundColor: '#e898e8',
  },
});

export default Pokemon;
```

In the code above, we first checked if the `name` has a falsy value. If it has, we simply return `null`, as there's nothing to render.

We're also using two new, built-in React Native components:

- `Image`: used for displaying images from the Internet or from the file system
- `FlatList`: used for displaying lists

As we saw earlier, we're passing in the Pokémon data as prop for this component. We can extract those props the same way we extract individual properties from an object:

```
const Pokemon = ({ name, pic, types, desc }) => {
    // ..
};
```

The `Image` component requires the `source` to be passed to it. The `source` can either be an image from the file system, or, in this case, an image from the Internet. The former requires the

image to be included using `require()`, while the latter requires the image URL to be used as the value of the `uri` property of the object you pass to it.

`resizeMode` allows you to control how the image will be resized based on its container. We used `contain`, which means it will resize the image so that it fits within its container while still maintaining its aspect ratio. Note that the container is the `Image` component itself. We've set its `width` and `height` to `100`, so the image will be resized to those dimensions. If the original image has a wider width than its height, a `width` of `100` will be used, while the `height` will adjust accordingly to maintain the aspect ratio. If the original image dimension is smaller, it will simply maintain its original size:

```
<Image source={{ uri: pic }} style={styles.image} resizeMode={"contain"} />
```

Next is the `FlatList` component. It's used for rendering a list of items. In this case, we're using it to render the types of the Pokémon. This requires the `data`, which is an array containing the items you want to render, and the `renderItem`, which is the function responsible for rendering each item on the list. The item in the current iteration can be accessed the same way props are accessed in a functional component:

```
<FlatList
  columnWrapperStyle={styles.types}
  data={types}
  numColumns={2}
  keyExtractor={(item) => item.id.toString()}
  renderItem={({ item }) => (
    <View style={[styles[item.name], styles.type]}>
      <Text style={styles.typeText}>{item.name}</Text>
    </View>
  )}
/>
```

In the code above, we also supplied the following props:

- `columnWrapperStyle`: used for specifying the styles for each column. In this case, we want to render each list item inline, so we've specified `flexDirection: 'row'`.
- `numColumns`: the maximum number of columns you want to render for each row on the list. In this case, we've specified `2`, because a Pokemon can only have two types at most.
- `keyExtractor`: the function to use for extracting the keys for each item. You can actually omit this one if you pass a `key` prop to the outer-most component of each of the list items.

At this point, you can now test the app on your device or emulator:

```
yarn start
```

While on the terminal, you can press `a` if you want to run the app on the Android emulator or `i` if you want to run it on the iOS simulator.

Please also note that the names of the Pokémon must start with a capital letter—for example, “Pikachu”, not “pikachu”.

Conclusion and Next Steps

That’s it! In this tutorial, you’ve learned how to set up the React Native development environment using Expo. You’ve also learned how to create your very first React Native app.

To learn more, check out these resources:

- Official React Native docs
- Official Expo docs
- Awesome React Native
- Mastering React Native

And don’t forget, you can find the source code used in this tutorial on this GitHub repo.

Chapter

6

15 React Interview Questions with Solutions

Nilson Jacques

React's popularity shows no sign of waning, with the demand for developers still outstripping the supply in many cities around the world. For less-experienced developers (or those who've been out of the job market for a while), demonstrating your knowledge at the interview stage can be daunting.

In this guide, we'll look at fifteen questions covering a range of knowledge that's central to understanding and working effectively with React. For each question, I'll summarize the answer and give links to additional resources where you can find out more.

1. What's the virtual DOM?

Answer

The virtual DOM is an in-memory representation of the actual HTML elements that make up your application's UI. When a component is re-rendered, the virtual DOM compares the changes to its model of the DOM in order to create a list of updates to be applied. The main advantage is that it's highly efficient, only making the minimum necessary changes to the actual DOM, rather than having to re-render large chunks.

Further reading

- Understanding the Virtual DOM
- Virtual DOM Explained

2. What's JSX?

Answer

JSX is an extension to JavaScript syntax that allows for writing code that looks like HTML. It compiles down to regular JavaScript function calls, providing a nicer way to create the markup for your components.

Take this JSX:

```
<div className="sidebar" />
```

It translates to the following JavaScript:

```
React.createElement(
  'div',
  {className: 'sidebar'}
)
```

Further reading

- An introduction to JSX
- JSX in Depth

3. What's the difference between a class component and a functional one?

Answer

Prior to React 16.8 (the introduction of hooks), class-based components were used to create components that needed to maintain internal state, or utilize lifecycle methods (i.e. `componentDidMount` and `shouldComponentUpdate`). A class-based component is an ES6 class that extends React's `Component` class and, at minimum, implements a `render()` method.

Class component:

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

Functional components are stateless (again, < React 16.8) and return the output to be rendered. They are preferred for rendering UI that only depends on props, as they're simpler and more performant than class-based components.

Functional component:

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

The Introduction of Hooks

The introduction of hooks in React 16.8 means that these distinctions no longer apply (see questions 14 and 15).

Further reading

- Functional Components vs Class Components in React
- Functional vs Class-Components in React

4. What are keys used for?

Answer

When rendering out collections in React, adding a key to each repeated element is important to help React track the association between elements and data. The key should be a unique ID, ideally a UUID or other unique string from the collection item:

```
<ul>
  {todos.map((todo) =>
    <li key={todo.id}>
      {todo.text}
    </li>
  )};
</ul>
```

Not using a key, or using an index as a key, can result in strange behavior when adding and removing items from the collection.

Further reading

- Lists and Keys
- Understanding React's key prop

5. What's the difference between state and props?

Answer

props are data that are passed into a component from its parent. They should not be mutated,

but rather only displayed or used to calculate other values. State is a component's internal data that can be modified during the lifetime of the component, and is maintained between re-renders.

Further reading

- props vs state

6. Why call setState instead of directly mutating state?

Answer

If you try to mutate a component's state directly, React has no way of knowing that it needs to re-render the component. By using the `setState()` method, React can update the component's UI.

Bonus

As a bonus, you can also talk about how state updates are not guaranteed to be synchronous. If you need to update a component's state based on another piece of state (or props), pass a function to `setState()` that takes `state` and `props` as its two arguments:

```
this.setState((state, props) => ({
  counter: state.counter + props.increment
}));
```

Further reading

- Using State Correctly

7. How do you restrict the type of value passed as a prop, or make it required?

Answer

In order to type-check a component's props, you can use the `prop-types` package (previously included as part of React, prior to 15.5) to declare the type of value that's expected and whether

the prop is required or not:

```
import PropTypes from 'prop-types';

class Greeting extends React.Component {
  render() {
    return (
      <h1>Hello, {this.props.name}</h1>
    );
  }
}

Greeting.propTypes = {
  name: PropTypes.string
};
```

Further reading

- Typechecking with proptypes

8. What's prop drilling, and how can you avoid it?

Answer

Prop drilling is what happens when you need to pass data from a parent component down to a component lower in the hierarchy, "drilling" through other components that have no need for the props themselves other than to pass them on.

Sometimes prop drilling can be avoided by refactoring your components, avoiding prematurely breaking out components into smaller ones, and keeping common state in the closest common parent. Where you need to share state between components that are deep/far apart in your component tree, you can use React's Context API, or a dedicated state management library like Redux.

Further reading

- Prop Drilling

9. What's React context?

Answer

The context API is provided by React to solve the issue of sharing state between multiple components within an app. Before context was introduced, the only option was to bring in a separate state management library, such as Redux. However, many developers feel that Redux introduces a lot of unnecessary complexity, especially for smaller applications.

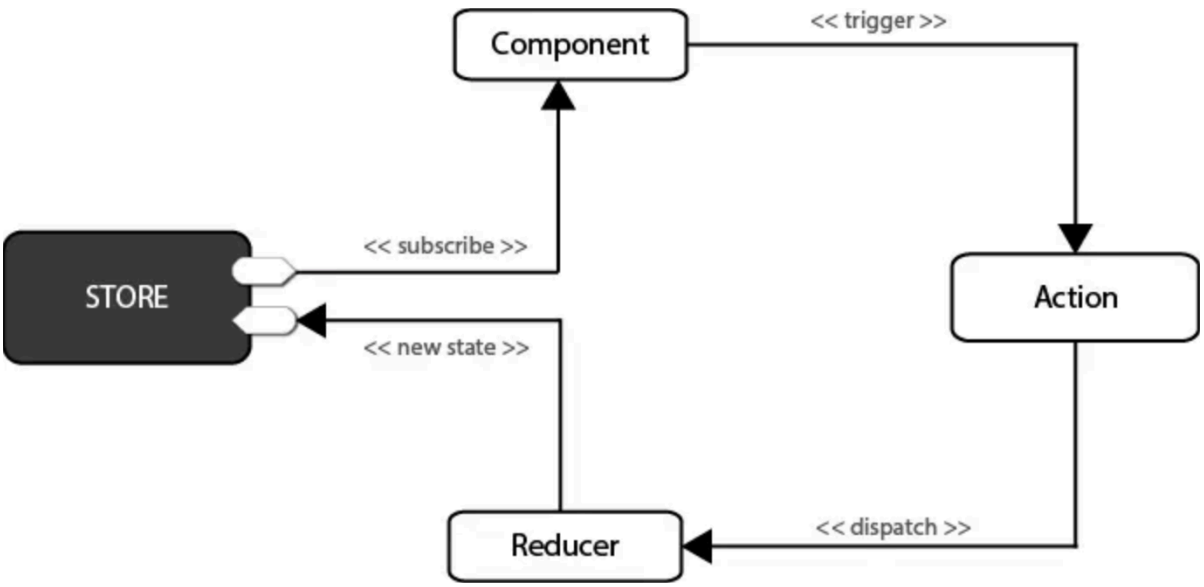
Further reading

- Context (React Docs)
- How to Replace Redux with React Hooks and the Context API

10. What's Redux?

Answer

Redux is a third-party state management library for React, created before the context API existed. It's based around the concept of a state container, called the store, that components can receive data from as props. The only way to update the store is to dispatch an action to the store, which is passed into a reducer. The reducer receives the action and the current state, and returns a new state, triggering subscribed components to re-render.



Further reading

- Getting Started with Redux
- A Deep Dive into Redux

11. What are the most common approaches for styling a React application?

Answer

There are various approaches to styling React components, each with pros and cons. The main ones to mention are:

- **Inline styling**: great for prototyping, but has limitations (e.g. no styling of pseudo-classes)
- **Class-based CSS styles**: more performant than inline styling, and familiar to developers new to React
- **CSS-in-JS styling**: there are a variety of libraries that allow for styles to be declared as JavaScript within components, treating them more like code.

Further reading

- How to Style React Components

12. What's the difference between a controlled and an uncontrolled component?

Answer

In an HTML document, many form elements (e.g. `<select>`, `<textarea>`, `<input>`) maintain their own state. An uncontrolled component treats the DOM as the source of truth for the state of these inputs. In a controlled component, the internal state is used to keep track of the element value. When the value of the input changes, React re-renders the input.

Uncontrolled components can be useful when integrating with non-React code (e.g if you need to support some kind of jQuery form plugin).

Further reading

- Controlled vs Uncontrolled Inputs
- Controlled Components (React Docs)
- Uncontrolled Components (React Docs)

13. What are the lifecycle methods?

Answer

Class-based components can declare special methods that are called at certain points in its lifecycle, such as when it's mounted (rendered into the DOM) and when it's about to be unmounted. These are useful for setting up and tearing down things a component might need, setting up timers or binding to browser events, for example.

The following lifecycle methods are available to implement in your components:

- **componentWillMount**: called after the component is created, but before it's rendered into the DOM
- **componentDidMount**: called after the first render; the component's DOM element is now available
- **componentWillReceiveProps**: called when a prop updates
- **shouldComponentUpdate**: when new props are received, this method can prevent a re-render to optimize performance
- **componentWillUpdate**: called when new props are received *and* `shouldComponentUpdate` returns `true`
- **componentDidUpdate**: called after the component has updated
- **componentWillUnmount**: called before the component is removed from the DOM, allowing you to clean up things like event listeners.

When dealing with functional components, the `useEffect` hook can be used to replicate lifecycle behavior.

Further reading

- React Lifecycle Methods Diagram
- The Component Lifecycle API

14. What are React hooks?

Answer

Hooks are React's attempt to bring the advantages of class-based components (i.e. internal state and lifecycle methods) to functional components.

Further reading

- Learn React Hooks in 5 minutes
- React Hooks: How to Get Started & Build Your Own

15. What are the advantages of React hooks?

Answer

There are several stated benefits of introducing hooks to React:

- Removing the need for class-based components, lifecycle hooks, and `this` keyword shenanigans
- Making it easier to reuse logic, by abstracting common functionality into custom hooks
- More readable, testable code by being able to separate out logic from the components themselves

Further reading

- Benefits of React Hooks
- React Hooks—advantages and comparison to older reusable logic approaches in short

Wrapping Up

Although by no means a comprehensive list (React is constantly evolving), these questions cover a lot of ground. Understanding these topics will give you a good working knowledge of the library, along with some of its most recent changes. Following up with the suggested further reading will help you cement your understanding, so you can demonstrate an in-depth knowledge.

Good luck!

Chapter

7

Five Awesome React UI Libraries for Your Next Project

Michael Wanyoike

In this guide, I'll compare and contrast five popular UI libraries for React. I'll look at their strengths and weaknesses, as well as provide you with a runnable demo for each library, to give you a sense of whether or not it would be a good fit for your next project.

What Makes a Good User Interface?

Before we get into the libraries themselves, I'd like to take a minute to consider the problem these libraries are there to solve.

In short, a good user interface:

- is based on *human-centered design*
- meets *accessibility standards*

If you're new to **human-centered design**, you should watch this six minute "bad doors are everywhere" Vox clip. It's the best introduction on the subject I've come across. You'll come to understand that there's an expectation about how we interact with physical and virtual objects. Failing to observe these principles of design in your user interface will frustrate your users every time they use your application.

Accessibility standards ensure that interfaces are usable by people with the widest range of abilities possible. They encompass assistive technologies, such as screen readers, and they ensure that text and graphic elements have the correct contrast ratio for good readability to assist those with visual impairments. These standards are good for everyone: I've come across a few sites with poor legibility even for users with 20/20 vision.

Fortunately for us front-end developers, we have access to open-source design systems and UI libraries that have handled these exact concerns. All we have to do is install and use the provided UI components based on the documentation and guidance given.

What Else Do I Need to Know?

In order to use these UI libraries effectively, you'll need to have some experience in using modern CSS. Specifically, you should know how to work with either Flexbox or Grid layouts. Some of the UI libraries we'll look at in this article are not perfect, and you might need to leverage your CSS skills to fix issues with layout styling.

When building interfaces with a UI library, your strategy should be to use the components like *Lego blocks*. You may have to write some custom CSS to direct the layout of each component in relation to the others, but you should avoid using plain CSS to change the internal styling of a component. That is an uphill task. It's much easier to build your own UI component from scratch.

Most UI library components are configurable via props that allow you to change attributes, such as:

- color presets
- color schemes—such as dark, light, or inverted
- size
- direction

Some libraries do provide additional utilities for further component customization. Such utilities usually target:

- alignment
- spacing
- typography
- display

… and many other things.

As you can see, there is a learning curve involved if you want to be able to utilize a UI library effectively. Picking an extensive UI library that you can use and rely on in multiple projects is highly recommended, as you won't waste time switching and re-learning a new UI library.

Currently, there are over 25 UI libraries made for React that are in various stages of development. Most are free, but some require a commercial license. For this guide, we'll focus on UI libraries that are:

- open-source
- production-ready
- regularly maintained
- well documented
- contain an extensive list of components
- meet accessibility standards

In the upcoming sections, we'll look at a handful of React UI libraries, starting with the simplest and ending with some of the most advanced.

I'd also like to mention that, at the time of writing, React 17 had recently been released. You may come across some migration issues with some of the projects. However, I believe they should be resolved in the near future.

With that said, let's get started.

React Strap

If you're already familiar with Bootstrap, you're going to quickly get acquainted with React Strap. Setting it up is pretty straightforward. All you have to do is execute the following in your React application:

1. Install the package dependency:

```
yarn add bootstrap reactstrap
```

2. Import the Bootstrap CSS framework (for example, in `src/index.js`):

```
import 'bootstrap/dist/css/bootstrap.min.css';
```

3. Import and use React Strap components as follows:

```
import React from 'react';
import { Button } from 'reactstrap';

export default (props) => {
  return (
    <Button color="danger">Danger!</Button>
  );
};
```

You'll find that React Strap provides React components for the majority of Bootstrap classes, such as:

- alerts
- buttons
- forms
- carousels
- navbars
- tables (basic)

Here is a quick mockup I created using ReactStrap UI library.

Here are other Bootstrap-related React UI libraries:

- Shards React: a free, beautiful and modern React UI kit based on Shards (a Bootstrap 4 UI kit).

- React Bootstrap: has better documentation and twice the number of GitHub stars. However, I came across a bug using one of the components, which is why it wasn't the main focus here.
- React MDB Bootstrap: combines Bootstrap and Google's Material Design. The free versions are open-source under MIT License. The Pro versions are proprietary and can have an EULA license. At the time of writing, the free version of this library hadn't been updated for some two months.

Bootstrap-based UI libraries are great for those who are already familiar with Bootstrap and want to build a simple user interface. Otherwise, if you're new to this, I'd recommend you look to other UI libraries that offer more components.

Below, I've provided links for the documentation and source code:

- Components Documentation
- Source code: 9.6k GitHub stars

React Suite

React Suite, as the name suggests, is a library of React components for "middle platform and back-end products". It supports all modern browsers, as well as IE back to v10. Do note that the IE 10 experience has been significantly downgraded with simpler styles and animations due to the limitations of such a legacy browser.

React suite is a desktop-based system. It doesn't support responsive or mobile screens. It requires a minimum of **React v16+** to run. Below is a quick run-down of the development environments it can be used in:

- Typescript
- Electron
- Reason

It also has guides for use with:

- Create React App
- Next.js (it supports server-side rendering)

It supports theme customization and internationalization. React Suite uses Less as its styling language, so you may need to install a Less compiler in your project.

React Suite provides developers with a massive list of advanced UI components, such as:

- Layout
- Loader
- Placeholder
- Popover, Tooltip, Alert, Notification
- Badge
- Drawer
- Divider
- TagPicker, SelectPicker, TreePicker, DatePicker
- Cascader
- Navbar
- Sidenav
- Steps
- Very Advanced Table (virtualized)
- Tree, CheckTree
- Timeline
- Full Calendar
- Custom validation via Schema models

Here's how you install it:

1. Install the package dependency:

```
yarn add rsuite
```

2. Update `src/index.js`:

```
import 'rsuite/dist/styles/rsuite-default.css';
```

or in `src/styles.less`:

```
@import '~rsuite/lib/styles/themes/default/index.less';
```

3. Import and use a React Suite component as follows:

```
import React from 'react';
import { Button } from 'rsuite';

export default (props) => {
  return (
```

```
    <Button appearance="primary">Hello world</Button>
  );
};
```

React Suite also provides a dark-theme version. All you have to do is swap out the current style for this one in your `src/styles.less` :

```
@import '~rsuite/lib/styles/themes/dark/index.less';
```

Alternatively, you can programmatically allow users to switch between the two themes, light and dark mode. Here is a quick mockup I created using the library.

Below I've provided links for the documentation and source code:

- Documentation
- Source: 4.8k GitHub stars

Ant Design

Ant Design refers to itself as a design language and framework. It features enterprise-class UI components, internationalization support and theme customization. It requires a minimum React version of 16.9 to operate and supports all modern browsers, as well as IE 11.

The UI library contains an extensive list of advanced UI components, which include:

- Advance table
- Drawer
- Popup over
- Popup Messages
- Confirmation Popups
- Notification box
- Toggle Switch
- Double column transfer choice box
- Tree Select
- Upload interfaces
- Full calendar
- Skeleton

It also provides a wider color palette system containing a total of 120 colors, which is made up of 12 primary colors and their derivative colors.

Implementing Ant Design is your project is quite simple. All you have to do is:

1. Install Ant Design package:

```
yarn add antd
```

2. Import stylesheet— `src/index.js` :

```
import 'antd/dist/antd.css';
```

3. Import and use components:

```
import React from 'react';
import { DatePicker } from 'antd';

export default (props) => {
  return (
    <DatePicker />
  );
};
```

Here is a quick mockup I created using Ant Design UI library.

Below are links to the documentation and source code:

- Documentation
- Source: 68k GitHub stars

Material UI

Material Design is a comprehensive design specification created by Google in 2014. It's considered by many developers to be the gold standard for user interfaces, as it covers many areas of design such as typography, grids, space, scale, color, and imagery. It's also customizable and supports dark mode, which is an essential requirement for many brands.

The best way to implement Google's Material Design in your React project is to use Material UI. Other implementations you can look at include MaterializeCSS and React MDB Bootstrap, which is an amalgamation of Bootstrap and Material design. Material UI is more popular, and it features a vast number of useful, advanced components, such as:

- Snackbar
- Progress
- Speed dial
- Stepper
- Timeline
- Datagrid

Do note that some components have pro versions of the basic ones. The pro versions are commercial and often quite pricey, such as the advanced version of the Datagrid.

Implementing Material UI in your project is quite simple. All you have to do is:

1. Install the Material UI package:

```
npm install @material-ui/core
```

2. Load the default Roboto font— `src/index.html` (optional):

```
<link rel="stylesheet"
href="https://fonts.googleapis.com/css?family=Roboto:300,400,500,700&display=swap" />
```

3. Import and use components:

```
import React from 'react';
import { Button } from '@material-ui/core';

function App() {
  return <Button color="primary">Hello World</Button>;
}
```

Here is a quick mockup I created using Material UI library.

Below are links to the documentation and source code:

- Documentation
- GitHub: 62K GitHub stars

Fluent UI React

Fluent UI is a design system by Microsoft that aims to provide a collection of UX components

that share code, design and behavior across multiple platforms. The platforms targeted are:

- web
- Windows
- iOS
- Android

Fluent UI React was formerly known as UI Fabric. Be aware that there is office-ui-fabric-react, which refers to the same library we'll be discussing here.

Implementing Fluent UI in your project is quite simple. All you have to do is:

1. Install the Fluent UI package:

```
yarn add @fluentui/react
```

2. Import CSS stylesheets for Fabric Core in `src/index.html` (optional):

```
<link rel="stylesheet" href="https://static2.sharepointonline.com/
files/fabric/office-ui-fabric-core/11.0.0/css/fabric.min.css" />
```

3. Import and use components:

```
import React from 'react';
import { PrimaryButton } from '@fluentui/react';

function App() {
  return <PrimaryButton>Hello World</PrimaryButton>;
}
```

Here is a mockup I created using the Fluent UI library.

I wasn't able to completely dig into the library, as parts of the documentation were incomplete and a number of components were throwing errors in the console. There are over 1,000 reported issues on the main repository, but as the project is still in active development, we hope to see a more polished version of this library soon from the company that brought us Visual Studio Code.

Below are links to the documentation and source code:

- Documentation

- GitHub: 9.7K GitHub stars

Bonus

Before we end this guide, I'd like us to consider a certain scenario. Let's assume you have a UX designer on your team and you're in charge of the final look of the front-end interface. Most probably, the design you'll be given will be unique and original. Unfortunately, trying to implement a custom UI design with an existing UI library can be very difficult.

An alternative approach is to use a utility-based CSS framework such as TailwindCSS. There are already a few React TailwindCSS UI libraries that are available:

- Tailwind React UI
- Windmill React UI

Here's a demo using TailwindCSS, in order to demonstrate the flexibility of building custom UI components from scratch.

This approach is good if there's only a handful of components you have to build. However, if the project requires a significant number of components, a better approach is for the UX designer to use a Figma/Sketch design kit for the UI library you want to use. Here are a few examples:

- Prime Design System
- Material Design Theming UI Kit
- Bootstrap Design Kit

Summary

The UI libraries I've discussed are some of the most popular on the market. If you need to get up and running quickly, give your app an attractive look and feel, and ensure cross-browser and cross-device compatibility, then any of them will serve you well.

However, as popular as these libraries are, they're not the only players in town. There are many great projects out there that I haven't been able to cover here, such as Prime Faces React and Blueprint.js, both of which deserve an honorable mention.

And if you're charged with evaluating a React UI library, remember (at the very least) to check its:

- maintenance/commit interval
- documentation quality
- list of UI components

Chapter

8

Getting Started with Next.js

Craig Buckler

Next.js is a Node.js application framework created by Vercel. You can use it to create static websites or blogs, but its scope is unlimited. Applications such as ecommerce stores, classified advertising, social networks, APIs, and anything else *should* be possible.

How Does Next.js Use React?

The Next.js home page claims it's "The React Framework for Production".

I'm not convinced this helps! Most developers know React as a browser UI library and would be forgiven for thinking Next.js is another client-side framework.

Next.js uses React to render pages and components—*but it occurs server-side*. Next.js is primarily a Node.js server-side framework. It happens to use React, but that's a fairly small part of the overall developer experience.

How Does Next.js Work?

Next.js combines the speed of client-side single-page applications (SPAs) with the robustness of server-side rendering (SSR).

When you initially access a Next.js-powered site, the server will return the page's HTML with all its assets (CSS, images, and so on). Where possible, page components will have been statically generated at build time. The server can render other components at request time, so content can be customized according to the user's credentials or other criteria.

The response is fast because no client-side JavaScript processing has been required at this point. Unlike a *pure* React app, Next.js pages are search-engine friendly and accessible to everyone even when JavaScript fails.

If all goes well, the application effectively becomes an SPA after the initial page load. Links to internal pages trigger an Ajax request which downloads the appropriate content, caches it, and renders without a full-page refresh. Code-splitting and bundling techniques are used to ensure the minimum of data is downloaded on demand.

Finally, when client-side processing is necessary, Next.js will have already returned a component's initial state from the server. That component is then **hydrated** so it's fully interactive in the browser.

The following features may tempt you to try Next.js for your next project …

Quick Setup and Zero Configuration

Updating Node

To follow along with this tutorial, you'll need a recent version of Node installed on your PC. If this isn't the case, head over to the Node home page and download the correct binaries for your system. Alternatively, you can install Node using a version manager.

Next.js provides a `create-next-app` CLI tool to quickly start building an application:

```
npx create-next-app
```

You can also create an app from scratch by initiating a new Node.js project and installing Next.js and React modules:

```
mkdir myapp
cd myapp
npm init
npm i --save next react react-dom
```

A `"scripts"` section can then be added to your `package.json` file to start development and build processes:

```
"scripts": {
  "dev": "next dev",
  "build": "next build",
  "start": "next start"
},
```

Next.js uses webpack, Babel, TypeScript, Sass, CSS Modules, PostCSS, hot reloading and more, but you shouldn't require any configuration if you follow the conventions.

Start the Next.js server in development mode using this:

```
npm run dev
```

Or you can use this:

```
npx next dev
```

Then access your application by navigating to http://localhost:3000/ in a web browser.

You can normally keep the server running; hot reloading ensures components and assets are refreshed as you develop. If you don't get the result you expect, it may be necessary to stop the server with `Ctrl`|`Cmd` + `C` and restart again.

File-based Page Routing

Next.js has a file-system–based router. Any React component file created within the project's *`pages`* directory and sub-directories is automatically available as a route and rendered accordingly. For example, a component at:

- *`pages/index.js`* renders the root home `/` page
- *`pages/contact.js`* renders a */contact* page
- *`pages/about.js`* renders an */about* page
- *`pages/about/privacy.js`* renders an */about/privacy* page

Open *`pages/index.js`* (or create it, if you installed the dependencies manually) and add the following code. This exports a functional React component which returns JSX code:

```
export default function Home() {
  return (
    <>
      <h1>Next.js app</h1>
      <p>This is a demonstration!</p>
    </>
  );
}
```

Returning JSX

JSX must be returned within a single containing element such as a * *. The *<>* *... </>* notation defines a document fragment, so no additional container is required.

Your browser should automatically reload the page when the file is saved. You may also notice a lightning icon at the bottom of the page, because Next.js has determined it can be pre-rendered at build time.

Client-side Page Transitions

Standard HTML `<a>` tags used in JSX create a hyperlink to another page and the browser will initiate a full-page refresh. Client-side transitions for pages within the same Next.js site can be implemented using using the `<Link>` component from `next/link`. For example:

```
import Link from 'next/link';
export default function Home() {
  return (
    <>
      <h1>Next.js app</h1>
      <p><Link href="/about">About us</Link></p>
    </>
  );
}
```

When the `About us` link is clicked, Next.js will request the page data using Ajax, cache it, and render it within the page.

Dev Mode

This may be less evident when running in development mode, since pages are re-rendered on demand.

Static Assets

Any file placed in the project's `public` directory and sub-directories is presumed to be a static asset such as an image, icon, `robots.txt`, or any other non-changing file. They can be referenced using standard URLs. For example:

```
<img src="/images/logo.svg" alt="logo" />
```

This loads the image from the file at `/public/images/logo.svg`.

Template Components

Next.js uses React components to implement templating. You can place these anywhere, but a new `components` directory is practical. For example, a `<Layout>` component could be defined in `components/layout.js`:

```
import Link from 'next/link';

export default function Layout({ children }) {
  return (
    <>
      <header>
        <img src="/images/logo.svg" alt="logo" />
        <nav>
          <ul>
            <li><Link href="/">home</Link></li>
            <li><Link href="/about">about</Link></li>
          </ul>
        </nav>
      </header>

      <main>{ children }</main>

      <footer>
        <p><Link href="/terms">terms and conditions</Link></p>
      </footer>
    </>
  );
}
```

Any page can use this component. Its content is passed to `<Layout>` in a `children` property of the `props` object, such as `pages/index.js` :

```
import Layout from '../components/layout';
import Link from 'next/link';

export default function Home() {
  return (
    <Layout>
      <h1>Next.js app</h1>
      <p><Link href="/about">About us</Link></p>
    </Layout>
  );
}
```

Static Generation

Static generation occurs once at build time. Where possible, Next.js will generate static HTML which requires no further processing on each request.

If you export an `async` function named `getStaticProps()` , Next.js will render the page at build time using the `props` returned from that function. The following example at `/pages/joke.js`

calls the API at https://official-joke-api.appspot.com/jokes/random to fetch a random joke (Next.js provides the *node-fetch* module):

```
import Layout from '../components/layout';

// fetch a random joke at build time
export async function getStaticProps() {
  const res = await fetch('https://official-joke-api.appspot.com/jokes/random');

  return {
    props: {
      data: await res.json()
    }
  }
}

// use joke data
export default function Joke({ data }) {
  return (
    <Layout>
      <h1>A random { data.type } joke</h1>
      <p className="setup">{ data.setup }...</p>
      <p className="punchline">{ data.punchline }</p>
    </Layout>
  );
}
```

On a production server, the page rendered at http://localhost:3000/joke shows the same joke on each refresh. *(You may see different jokes during development, since static content can be re-rendered.)*

Server-side Rendering

Server-side rendering (SSR) occurs on every user request. This may be necessary if content needs to change according to a user's account settings, items in their shopping cart, their geographical location, the browser they're using, the time of day, or any other factor.

The *getServerSideProps()* function is identical to *getStaticProps()*, except that Next.js will call it on every request. The function in the page above (in the Static Generation section) could be renamed accordingly.

```
// fetch a random joke at request time
export async function getServerSideProps() {
  const res = await fetch('https://official-joke-api.appspot.com/jokes/random');
```

```
  return {
    props: {
      data: await res.json()
    }
  }
}
```

On a production server, the page rendered at http://localhost:3000/joke shows a different joke on each refresh.

Calling a Third-party API

Calling a third-party API on every page request is not a good idea: there may be access restrictions or performance issues. You could improve this by caching responses in a file or database so data is available locally.

Dynamic Routes

Adding `[` square brackets `]` to a page's file name creates a dynamic route which calls the page on any matching URL. For example, the file at `pages/post/[id].js` is called for routes such as `post/1`, `post/abc`, and so on:

```
import { useRouter } from 'next/router'

export default function Post() {
  const router = useRouter();
  const { id } = router.query;

  return <p>Post ID: {id}</p>;
}
```

Naming the file `pages/post/[...id].js` defines a catch-all route, which calls the page on routes such as `post/a/1`, `posts/a/b/c`, and so on. The returned `id` is an array containing the associated paths—such as `['a', 'b', 'c']`.

If a page has dynamic routes and uses `getStaticProps`, it needs to define a list of paths to be statically rendered at build time (see the Static Generation section above). This can be done by exporting an `async` function named `getStaticPaths()`. For example, a file named `[id].js` must have `id` values returned:

```
export async function getStaticPaths() {
  return {
    paths: [
      { params: { id: 'abc' } },
      { params: { id: 'xyz' } },
      { params: { id: '123' } }
    ],
    fallback: false
  };
}
```

False Fallback

Using a *`fallback`* value of *`false`* results in a 404 page for any undefined route.

The *`getStaticProps()`* function can be exported to return data for each *`id`*, where *`getData()`* is called three times and passed *`'abc'`*, *`'xyz'`*, and *`'123'`* to return an object associated with that *`id`*:

```
// dynamic route content
export async function getStaticProps({ params }) {
  return {
    props: {
      data: await getData(params.id)
    }
  }
}
```

Client-side Rendering

Next.js will render React components on the server. However, any component using a hook such as React.useState() or React.useEffect() will be hydrated as a client-side component after the page has loaded. For example, this component at */components/clicker.js* shows a button with a number that increments on every click:

```
import React from 'react';

export default function Clicker() {
  const [count, setCount] = React.useState(0);

  return (
```

```
    <button onClick={() => setCount(count+1)}>
      I was clicked { count } times
    </button>
  );
}
```

Be wary that some properties may be available on the server but not the client, and vice-versa. For example, the following code will fail to render because the `window` object is not available server-side:

```
// THIS WILL FAIL!
const [href, setHref] = React.useState(window.location.href);
```

To fix this issue, an initial state of `null` can be defined which is initialized within a `useEffect` hook:

```
const [href, setHref] = React.useState(null);

React.useEffect(() => {
  if (!href) {
    setHref(window.location.href);
  }
}, [href])
```

This works because `useEffect` doesn't run on the server, so by the time we hit that block, we must be on the client.

API Routes

Any file inside the `pages/api/` directory is mapped to an `/api/` route and served as an API endpoint rather than as a page.

The file must export a default function which is passed these:

1. `req` : an incoming request object
2. `res` : an outgoing response object

For example, take this code at `pages/api/hello.js` :

```
export default (req, res) => {
```

```
  const name = req.query.name || 'World';
  res.status(200).json({ text: `Hello ${ name }!` })
}
```

This returns the following:

- `{ "text": "Hello World!" }` when `/api/hello` is called
- `{ "text": "Hello SitePoint!" }` when `api/hello?name=SitePoint` is called

You can either try this out in your browser, or by using cURL:

```
curl http://localhost:3000/api/hello
curl http://localhost:3000/api/hello\?name\=SitePoint
```

Like pages, APIs can also use dynamic route file notations. For example, `pages/api/user/[id].js` is called when `/api/user/123` is loaded. The `'123'` parameter can be determined from `req.query.id`.

CSS Options

Next.js has built-in support for CSS options, including Styled JSX, CSS modules, Sass, Less, and more.

Using the joke page at `/pages/joke.js` as an example, styled JSX can be added, which scopes CSS styles to the component. In other words, the `.setup` and `.punchline` styles will not apply elsewhere, even if the same selectors are used:

```
export default function Joke({ data }) {
  return (
    <Layout>
      <h1>A random { data.type } joke</h1>
      <p className="setup">{ data.setup }...</p>
      <p className="punchline">{ data.punchline }</p>

      <style jsx>{`
        .setup { font-weight: bold; }
        .punchline { font-style: italic; }
      `}</style>
    </Layout>
  );
}
```

Alternatively, CSS modules can be used by creating a file named `jokes.module.css` and

importing it as a `styles` object which references specific selectors:

```
import styles from './jokes.module.css';

export default function Joke({ data }) {
  return (
    <Layout>
      <h1>A random { data.type } joke</h1>
      <p className={ styles.setup }>{ data.setup }...</p>
      <p className={ styles.punchline }>{ data.punchline }</p>
    </Layout>
  );
}
```

Global CSS

Global CSS can be defined in a `styles` directory, such as `styles/main.scss`.

Using SCSS

To use SCSS files, install Sass with `npm install sass` first.

Global stylesheets *must* be imported in a file named `pages/_app.js` so styles are applied to all pages and components:

```
import '../styles/globals.css'

function MyApp({ Component, pageProps }) {
  return <Component {...pageProps} />
}

export default MyApp
```

Next.js Application Deployment

Once you're happy with your Next.js application, you can build a production-level site without source maps and other development options. Stop the Next.js server using `Ctrl`|`Cmd` + `C`, then run the following:

```
npm run build
```

(Alternatively, use `npx next build`). This builds all server files to the `.next` directory.

The Next.js server can be started on any production server with Node.js v10 or above using `npm run start` or `npx next start`.

Vercel offers automated deployment and hosting options for Next.js apps.

Export a Next.js Site to Static HTML

Next.js can also export a site to static HTML pages. This is similar to using a static-site generator such as Eleventy, since the resulting pages can be viewed without server-side technologies such as Node.js. There are several caveats, and the export process will fail if any page uses server-side rendering.

The export must be run immediately after a build:

```
npx next build
npx next export
```

The resulting HTML and asset files are output to an `out` directory, which can be copied to any web server directory.

Your Next(.js) Project

Next.js is a framework with the flexibility to be adapted for any type of web project. It can be used to create fast, React-based SPAs with robust server-side rendering. It's easy to get started, the documentation is great, and the developer experience is enjoyable.

Chapter

9

React vs Vue: A Side-by-side Comparison

Michael Wanyoike

“Should I learn React or Vue?” is a question many developers can struggle with when starting out in front-end development. Back in the day, React was a solid choice, as it didn’t have a strong competitor (other than maybe AngularJS). However, Vue has grown rapidly these last few years, and has now become a serious contender for building apps of all sizes.

I’ve been using React in multiple projects for the past couple of years and I love it. However, recently I’ve also had the chance to get to grips with Vue. In this guide, I want to look at how the two stack up against each other and hopefully provide a starting point for those who have to decide which way to jump.

Criteria for the Decision

For many developers, the decision over which technology to use has already been made at their place of work. This guide is written for those who are in a position to make this choice for themselves.

But before making a decision, let’s first ask ourselves *why* we’re asking this question. What is it that we’re really looking for in a front-end framework?

In short, we’re all looking for something that:

- is easy to learn
- has excellent documentation
- provides developer support through forums, Slack, Discord channels etc.
- allows easy integration with other platforms
- is stable and regularly maintained
- offers new job opportunities

We’ll go through each of these points in this guide.

Before we move on, you may be wondering why I haven’t included Angular in this comparison. In my experience, Angular is the hardest JavaScript framework to learn. I’ve also found that most releases have a number of significant breaking changes that make the majority of past documentation and tutorials outdated, even if they were only published in the last year.

Upgrading your Angular projects to the latest major release can also be a time consuming process, as you have to re-write certain sections of your code to comply with the new requirements. You’ll also need a solid test suite to ensure all the features you had implemented are still working after the migration.

With that out of the way, let’s dive in to the core of our comparison of React and Vue.

A High-level Overview of React vs Vue

React is a view library with hardly any opinion on how you should structure your code. Vue, on the other hand, is a framework made up of core libraries and software patterns. Both are designed to help developers build user interfaces.

If you want a quick answer to which front-end framework to go with, I'll tell you right now: *if you're looking for a job, go with React. If you're looking to build your own software product or business, go with Vue.*

Feel free to stop right here. By making a fast decision for yourself, you won't deal with the fatigue from struggling to make a choice later on. But if you're not sure which is the best fit for you, or you just want to see what things look like on the other side of the fence, keep reading.

Why Am I Enthusiastic About Vue?

Although I'll try to remain unbiased and objective throughout this guide, let me tell you what got me enthusiastic about Vue.

Recently, I migrated a project that was still in development from a React framework (Next.js), to a Vue framework (Nuxt.js). Making the switch was like a breath of fresh air. Many of the issues I had been struggling with were gone in an instant. For example:

- Next.js lacks a modular/plugin ecosystem. Hence integration with a platform or a library such as TailwindCSS is manual. With Nuxt, all you have do is install a plugin, and the integration works with very minimal configuration.

- Next.js is a framework that gives you a bare-bones project structure. You have to research and set up additional tools inside your project workspace to ensure formatting, linting and line endings are consistent in each code file. With Nuxt.js, you're given the option to install all these additional tools and configurations during the project creation stage.

- Next uses `.js` files for everything, including library and React code. This means if you enable linting of React on `.js` files, you'll get a lot of false positives on utility JavaScript files that don't contain any React code. With Nuxt, it uses a `.vue` extension, which has its own linting rules.

None of these issues were real show stoppers, and had more to do with the frameworks themselves than with React or Vue. But nonetheless, this was my entry ticket to the Vue ecosystem.

And please don't think I'm implying that React is a terrible framework. It isn't. React took us from a terrible, buggy front-end environment to a much better one. Vue takes us even further to a better, more efficient development workflow. It's still early days, but that has been the main difference I've encountered when working with React and Vue.

Ease of Use

While working with Vue code, you always have the documentation on hand that explains the software pattern and practice you should use to solve a particular problem. Sometimes I feel like the creators of Vue are reading my mind whenever they have a well documented solution to a problem I just encountered.

With React, you're provided the tools without clear instructions on how you should use them. Hence, you have to research, reading tutorials and articles on how to solve a particular problem using React. Quite often, you'll have to install additional libraries to solve certain problems.

A good example of such a problem I solved instantly in Vue was when I was trying to figure out how to toggle a class name programmatically. I simply searched for "Vue dynamic class" and straight away found this page, which explained exactly how to do it. All I had to do was simply use the `:class` attribute.

With React, after you Google the search term, you'll have to go through Stack Overflow and a few more articles. Eventually you'll come across the classnames package, which you have to install and import into your React file to get the equivalent feature of Vue's `:class` attribute.

With Vue, you'll find that the core library provides the majority of the common features required in any project. So you won't need as many third-party packages to implement a feature.

Upgrading and Backwards Compatibility

Both React and Vue are excellent in terms of backward compatibility. 99% of the time, upgrading an older React project goes smoothly without any issues. I haven't used Vue for very long, so I can't yet say the same about Vue. The Vue team has just released their latest major release—Vue 3—which has been in the making for a long time. It's a rewrite of Vue 2 that's taken advantage of new JavaScript language features and also addressed architectural issues in the current codebase.

The latest major release of Vue has successfully been implemented in a way that doesn't inflict huge migration costs to users—which would also cause fragmentation in the ecosystem. Here's a migration guide where they've listed their breaking changes. As a new Vue developer, I haven't

been affected at all and am quite happy to benefit from all the improvements and changes they've introduced in Vue 3.

Coding Comparison

React is about a year older than Vue. It brought significant and disruptive changes to front-end web development, pushing predecessors like jQuery towards obsolescence. As a result, it has the largest user base. Vue, on the other hand, has a much smaller community that's been slowly growing over time. Though smaller, the community is still sufficient, and you can quickly get support from its members.

When it comes to performance, both React and Vue are very fast and pretty much on the same level. Vue 3's architecture has been re-written for improved performance. There's a third-party benchmark you can view to compare performance results. Unfortunately, these results are for older versions of React and Vue. Still, the results were quite good back then and most definitely they should have improved in the most recent versions.

Let's now get to the developer experience differences between React and Vue. When it comes to local state management and lifecycle methods, both are on par. But I do prefer Vue's naming convention of lifecycle methods—for example, React's `componentWillUnmount()` vs Vue's `beforeUnmount()`. However, that's a minor issue, and there's a number of situations where I feel Vue is superior.

Let's go over a few of them.

JSX vs Vue Templates

React requires the use of JSX to render HTML content. JSX is a syntax that allows you to mix JavaScript and HTML elements. There are many developers who love this style. However, there are limitations, which I'll cover shortly. Here's a JSX component example:

```
import React from 'react'

const Hello = ({ name }) => {
  return (
    <div className="py-4 bg-red-100 border border-red-200">
      <p className="italic text-center text-red-700">
        Hello, { name }
      </p>
    </div>
  )
}
```

```
export default Hello
```

Vue does support JSX, but I prefer the classic HTML syntax. Here's a Vue template component example:

```
<template>
  <div class="py-4 bg-red-100 border border-red-200">
    <p class="italic text-center text-red-700">
      Hello, {{ name }}
    </p>
  </div>
</template>

export default {
  props: ['name']
}
```

To me, the separation of HTML and JavaScript code is preferble. It's much cleaner. Novice Vue developers can leverage their existing HTML skills without having to learn anything new. With JSX, you'll have to rename reserved keywords such as `class` to `className`, which is something I often forget. In Vue, it's much easier to copy HTML code from theme templates and design kits and use it as is. React devs have to put in a little more work while copying HTML code to JSX. Fortunately for them, there are several React theme templates with JSX syntax on the market.

CSS Component Scoping

CSS scoping ensures the styles you implement in a component don't affect the rest of the application. There are many ways you can implement CSS scoping in both React and Vue. You can import a component CSS file or use inline styling. You can also put your component styles in the global style sheet and use the BEM naming convention to provide scoping.

Vue supports scoped styling right inside the component. And it looks like this:

```
<template>
  <div class="hero">
    <h1> Hello World </h1>
  </div>
</template>

<style scoped>
.hero {
  @ apply bg-gray-900 text-gray-100 text-4xl;
```

```
}
</style>
```

With Vue, you can use the `@apply` directive to inline any existing utility classes into your own custom CSS—something that's quite handy when using a CSS framework like TailWindCSS. This isn't possible in React unless you implement it in an external stylesheet. There are CSS libraries such as styled components and emotion that make styling in React more manageable. I've tried using them, but I'm not convinced of their benefits. However, that's just me, as there are many React and Vue developers who love using them.

Conditional Rendering

Conditional rendering is basically an `if-statement` for HTML code. There's a couple of ways you can do it in React.

React classic approach:

```
function Greeting({ isLoggedIn }) {
  if (isLoggedIn) {
    return <UserGreeting />;
  }
  return <GuestGreeting />;
}
```

React inline approach:

```
function Mailbox({ unreadMessages }) (
  <div>
    <h1>Hello!</h1>
    {unreadMessages.length > 0 && (
      <h2> You have {unreadMessages.length} unread messages.</h2>
    )}
  </div>
)
```

With the inline approach, I find it a bit hard to read the code. This is how Vue handles conditional rendering:

```
<template>
  <div>
    <h1>Hello!</h1>
    <h2 v-if="unreadMessages.length > 0">
      You have {{ unreadMessages.length }} unread messages.
```

```
    </h2>
  </div>
</template>
```

To me, the Vue version is more readable. You can also add an “else” block using `v-else` like this:

```
<template>
  <div>
    <h1>Hello!</h1>
    <h2 v-if="unreadMessages.length > 0">
      You have {{ unreadMessages.length }} unread messages.
    </h2>
    <h2 v-else>You have read all your messages</h2>
  </div>
</template>
```

With React, you can do inline if-else like this:

```
render() {
  const isLoggedIn = this.state.isLoggedIn;

  return (
    <div>
      {isLoggedIn
        ? <LogoutButton onClick={this.handleLogoutClick} />
        : <LoginButton onClick={this.handleLoginClick} />
      }
    </div>
  );
}
```

As you can see, it’s not quite obvious what’s going on here. With Vue, you can use `v-else-if` to stack multiple conditions like this:

```
<template>
  <div v-if="type === 'A'">
    A
  </div>
  <div v-else-if="type === 'B'">
    B
  </div>
  <div v-else-if="type === 'C'">
    C
  </div>
  <div v-else>
    Not A/B/C
```

```
  </div>
</template>
```

Vue also provides a v-show directive that's preferred if you need to toggle something very often. `v-if` is recommended if the condition is unlikely to change at runtime.

List Rendering

To render lists in React, you'll need to use the JavaScript `map()` function to generate an HTML element for each item. Here's an example of how that might look:

```
render() {
  const numbers = [1, 2, 3, 4, 5];
  const listItems = numbers.map((number, index) =>  <li key={index}>{number}</li>);

  return (
    <div>
      <ul>{listItems}</ul>
    </div>
  );
}
```

Keys are required to uniquely identify each element in order for React to track which items have changed, been added or removed. This requirement also applies to Vue. However, Vue uses directives to perform list rendering like this:

```
<template>
  <div>
    <ul v-for="(number, index) in numbers" :key={index}>
      <li>{{ number }}</li>
    </ul>
  </div>
</template>
```

I strongly prefer the way Vue handles list rendering. In this simple example, you won't see much of a difference. However, the difference will become clearer when the code starts to become bigger. With React, you have to keep looking up and down to see how the code works. With the Vue code, you can read continuously from top to bottom and quickly understand the code without having to refer back up again.

Let's Build an App

Now let's turn our minds towards building something in both React and Vue.

In order to evaluate if a particular library or framework is a good fit for a project we want to work on, we need to identify potential issues early on, before they become a challenge. As an example, I've decided to focus on ecommerce. Traditionally, all you needed to build an ecommerce site was to implement a monolithic software solution such as Magento, Prestashop or WooCommerce (WordPress). These applications gave business owners powerful back ends but with slow front ends. The solution was, of course, to implement caching and add more server resources, which can quickly become costly.

The latest trend for ecommerce businesses is to replace the front end with a static website built with a static-site generator. Static websites are blazing fast and greatly improve the online shopping experience for visitors. Businesses that fail to catch on will soon have their "Kodak moment". The main benefits of static-site generation include:

- blazing-fast page speeds
- improved SEO ranking and conversions
- significantly improved security: no surface to attack
- reduced server costs
- scalability
- better developer experience: front- and back-end concerns are separated

To implement this kind of functionality in React and Vue, it makes sense to turn to an existing framework such as Gatsby (for React) and Gridsome (for Vue).

Both of these are well suited to static-site generation for ecommerce applications due to a feature called **hydration**. This is a process of shipping HTML to a browser, then when JavaScript can run, converting this HTML into a single-page application to make it interactive. This is an extremely useful feature that is not natively possible with static-site generators such as Hugo. They both also provide native support for GraphQL, which greatly simplifies the task of querying data from back-end providers.

Speaking of back-end providers, we need to ensure that the framework we choose can easily connect to the back end of our choice. The back ends you'll likely consider are:

- Traditional back ends such as WordPress, Magento, and Prestashop
- Headless CMS providers such as Contentful, Prismic, Sanity, Storyblok, and ButterCMS
- Modern ecommerce back ends such as Shopify, and Snipcart

You'll also need to consider essential ecommerce services, which can be implemented by integrating with platforms that provide:

- Authentication—such as Auth0, Okta

- Search capabilities—such as Algolia
- Reviews—such as Trustpilot, Yotpo
- Serverless functions—such as Firebase, Vercel, Netlify, and AWS Lambda
- User tracking and metrics—such as Google Analytics, and Mixpanel

If you were building a blog, you would consider a different set of platforms to integrate with, such as Disqus for comments. I hope you get my point on the evaluation strategy I'm trying to drive at. Once you've identified all the systems and features you need to work with, go through the documentation of each platform and evaluate the level of support they provide for each framework. Fortunately, you'll find either an official plugin or a community one that allows fast and easy integration with a particular platform/service.

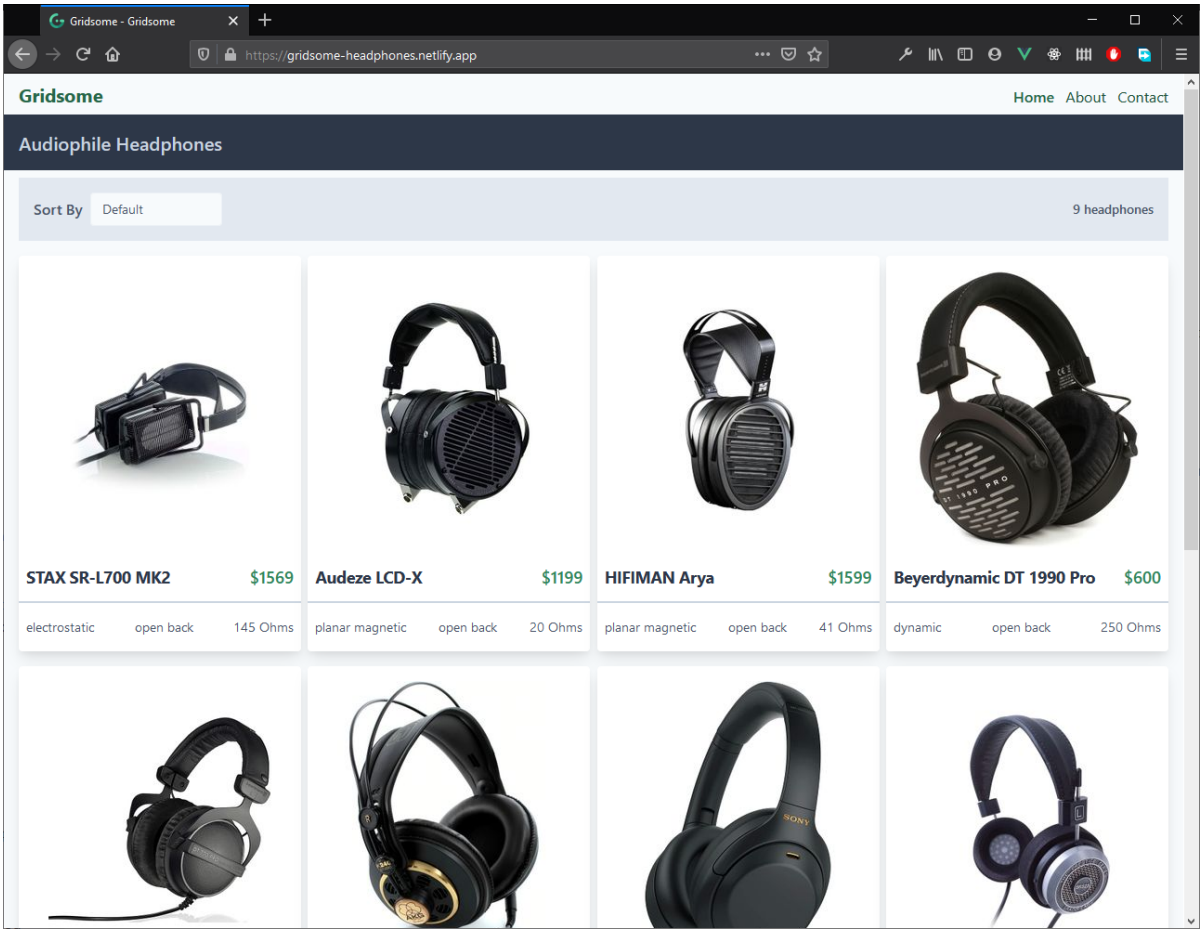
Project Breakdown

I've built two identical headphone ecommerce projects each with a different framework:

- Gatsby — React
- Gridsome — Vue

Both projects have access to the same back end that I've built and populated using Storyblok, a headless CMS service. I've also used TailwindCSS, a utility-based CSS framework, in both frameworks.

Here's a screenshot of the Gridsome version. The Gatsby version looks identical:



The source code for the Gatsby version is here, and you can find the Gridsome source code here. I've included my (read-only) Storyblok token in the repo, so you should be able to clone either repository, install the dependencies and have the complete site run on your machine. I recommend that you take some time to explore the source code and get a feel for how React and Vue do things.

Both Gatsby and Gridsome can be used as static-site generators. They have the same architecture and share the same concepts. Gatsby was the first to launch, and it uses React. Unlike other static-site generators that run on non-JavaScript languages, Gatsby provides developers with the unique hydration feature. As explained above, this means that, when a static page is loaded on the browser from the server, React takes over and starts handling rendering of the page. Thus it becomes a single-page application, but with fast initial load times and SEO support.

Gridsome is the Vue alternative to Gatsby. It's highly inspired by Gatsby, and as you'll see, the architecture and concepts are identical. These include:

- performance optimization using Google's PRPL pattern
- automatic routing
- a plugin ecosystem
- simple integration with data from different sources such as file system, CMS, APIs, SaaS platforms and databases
- a GraphQL data layer
- PWA support

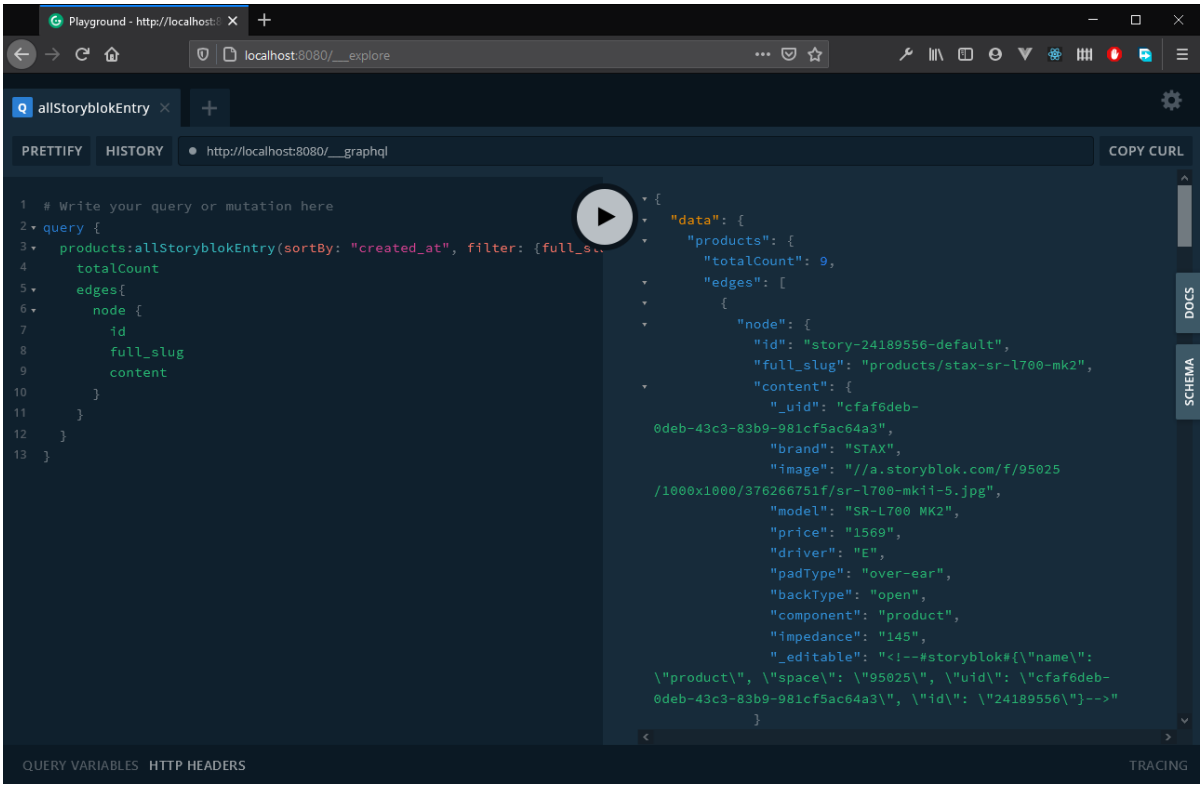
CMS Integration: GraphQL

Both Gridsome and Gatsby have a plugin that supports easy integration with Storyblok source via the GraphQL layer:

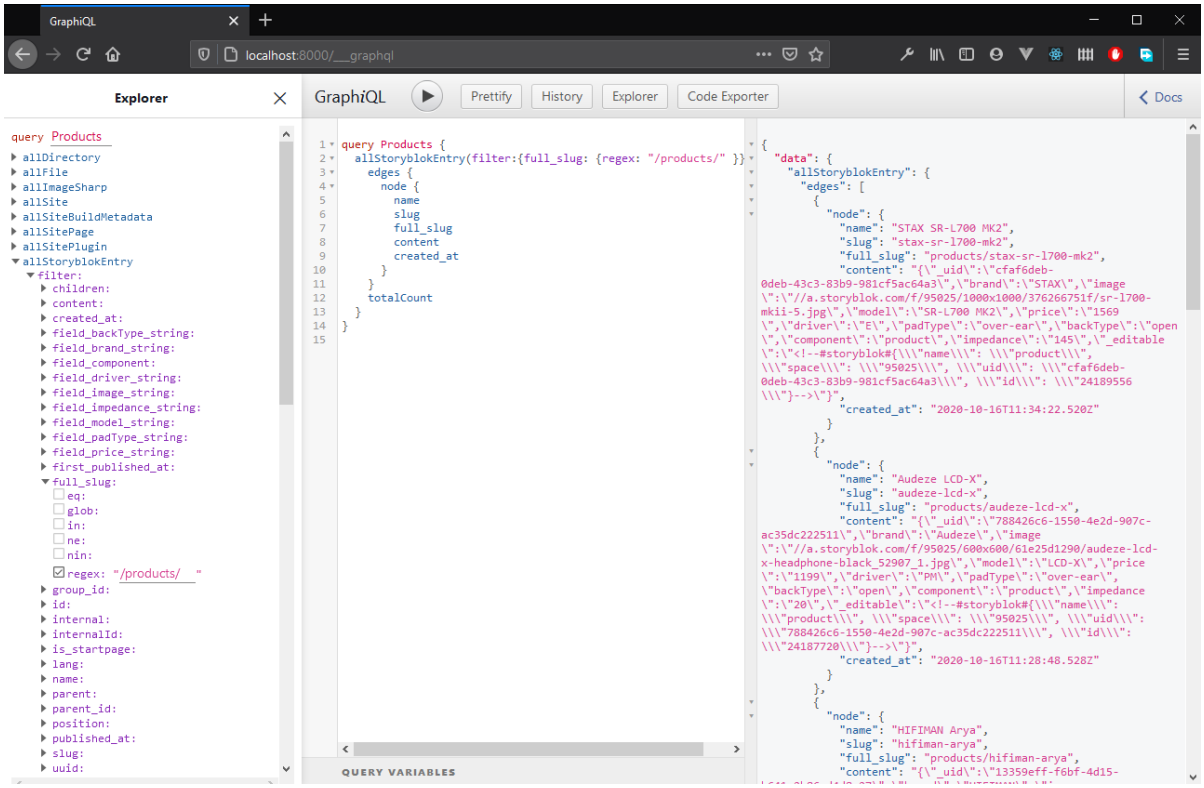
- gridsome-source-storyblok
- gatsby-source-storyblok

The plugins are well documented, with clear instructions on how to use them. Storyblok also provides first-class documentation and technical support to both site generators. They also have excellent GraphQL interfaces with excellent documentation.

Gridsome GraphQL:



Gatsby GraphQL:



Configuring the Secret API Token

In order to connect to Storyblok, you need to provide an API token that's made available via the settings dashboard. It's important that this API token is not saved in your source code. The recommended place to keep is in a `.env` file. This file needs to be excluded using `.gitignore`. Here's an example in Gridsome of how the token is accessed in `gridsome-config.js`:

```
{
  plugins: [
    {
      use: 'gridsome-source-storyblok',
      options: {
        client: {
          accessToken: process.env.STORYBLOK_API,
        },
      },
    },
    {
      use: 'gridsome-plugin-tailwindcss',
    },
  ],
}
```

Quite simple, right? You probably should expect that it's the same process with Gatsby, but, unfortunately, no. It's an unusually long process. The technical reason why it can't be done the simple way is beyond me. Thanks to Storyblok's technical support, I was guided on how to set it up. This is the process.

First, install the following packages:

```
yarn add dotenv crossenv
```

At the top of gatsby-node.js, add this code to suppress the `fs` error messages that crop up when using a `dotenv` package:

```
exports.onCreateWebpackConfig = ({ actions }) => {
  actions.setWebpackConfig({
    node: {
      fs: 'empty',
    },
  })
}
```

Next, add this at the top of `gatsby-config.js`:

```
require('dotenv').config({
  path: `.env.${process.env.NODE_ENV}`,
})
```

You'll need to store your API token in the `.env.development` file. Make sure to exclude it in your `.gitignore` file. Next, you'll have to access the API token in `gatsby-config.js` as follows:

```
{
  plugins: [
  {
    resolve: 'gatsby-source-storyblok',
    options: {
      accessToken: `${process.env.STORYBLOK_TOKEN}`,
      homeSlug: 'home',
      version: process.env.NODE_ENV === 'production' ? 'published' : 'draft',
    },
  },
]
}
```

Take note of the tilde syntax used to wrap `process.env.STORYBLOK_TOKEN`. Finally, you'll need to

modify your `develop` script in `package.json`. In Linux, you don't need `cross-env`, but it's important to use it since it's needed on Windows:

```
{
  "scripts": {
    "develop": "cross-env NODE_ENV=development && gatsby develop"
  }
}
```

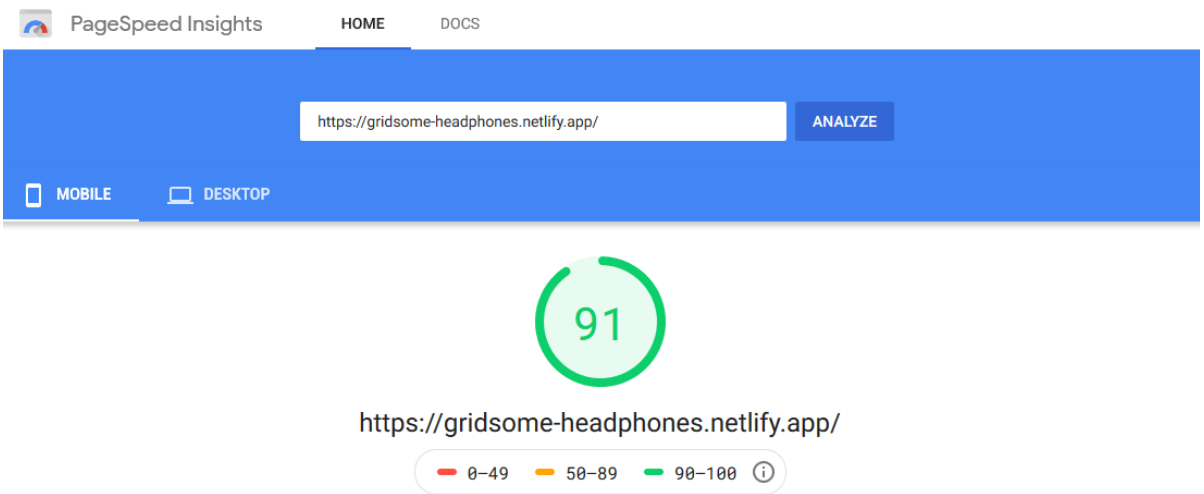
With all that set, Gatsby should connect to your Storyblok API safely via the secret `.env` token. It took me two days to resolve this. If you were to follow this Stack Overflow answer, you'd be stuck like I was. There are quite a number of dead-end answers online regarding this topic. I'm quite grateful to Storyblok's support team for helping me resolve this issue.

Luckily for you, I've saved you hours of pain, as I've documented the whole procedure for you.

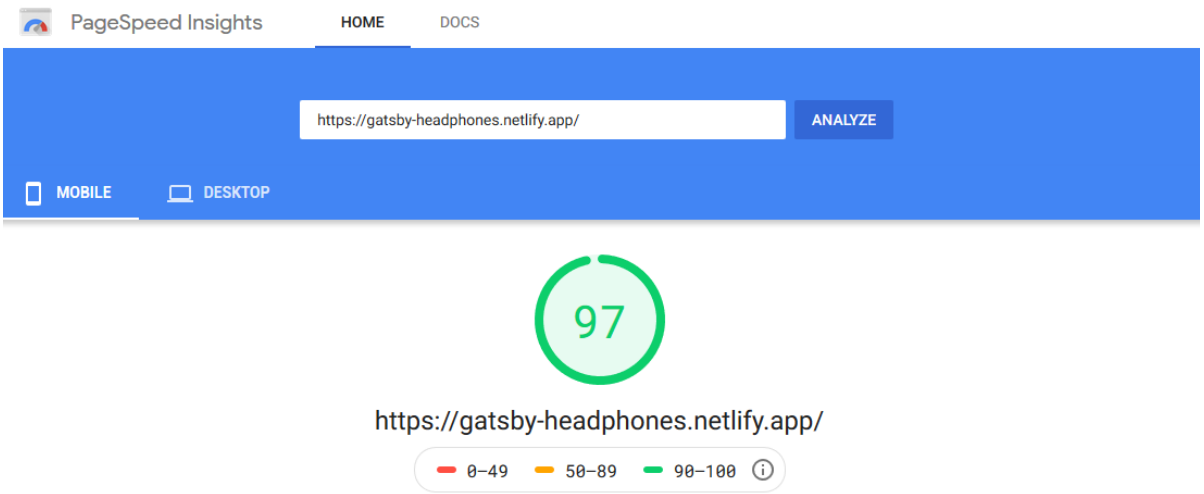
Page Speed Scores

Both frameworks are excellent at achieving very high Chrome Lighthouse scores with zero effort. For a monolithic CMS such as WordPress, achieving good scores is extremely hard even with the best caching plugins and hosting. With the headphone projects I've built, both scored 100% on the desktop page speed test. For mobile, the Gridsome version got 91% while the Gatsby version got 97%.

Gridsome mobile score:



Gatsby Mobile score:



Technically, Gatsby is the winner here. However, the difference in scores is marginal and it's almost impossible for end users to tell the difference. According to Google:

- a score below 50 is considered poor
- below 90 requires improvement
- 90 and above is considered good

Image Optimization

Image optimization for today's computing devices is important! In the past, smartphones used to have lower screen resolutions, so you only needed to provide a lower resolution image. Nowadays, most smartphones—even entry level ones—have high resolution screens that are capable of showing detailed images.

There are also no standards on the size of screen resolution being used on different devices. Trying to support them all is near impossible and impractical. The best solution is to generate multiple resolutions of the same image and let the browser select the optimum image resolution for its screen. This is done using the srcset attribute of the `img` tag.

With Gatsby, this work is done for you by gatsby-image. Gridsome's version is called g-image.

In both versions, I used Storyblok's image optimization service to optimize the product images. Both Gatsby and Gridsome have internal tools for image optimization which, due to lack of time, I wasn't able to use.

In Gatsby's case, I should have used gatsby-storyblok-image, a plugin that allows use of `gatsby-image` outside of GraphQL. I believe if I had completed that step, I'd have achieved 100% mobile

page speed for Gatsby.

In Gridsome's case, there's unfortunately no similar plugin. The `g-image` component (see below) only optimizes images loaded from the file system. Fortunately, the `gridsome-source-storyblok` allows downloading of images to the local file system. It also updates content references to reflect the new location of the images. If I'd set this up, I believe I'd have achieved a higher score in the Gridsome page speed test.

If I were to choose a winner here, I'd say Gatsby has better image optimization support for Storyblok.

Dependency Package Size

The Gatsby project `node_modules` folder consumed about 464MB of disk space. It relies on quite a number of dependencies for it to work:

```
{
   "dependencies": {
    "tailwindcss": "^1.9.4",
    "@tailwindcss/typography": "^0.2.0",
    "gatsby": "^2.24.67",
    "gatsby-env-variables": "^2.0.0",
    "gatsby-image": "^2.4.20",
    "gatsby-plugin-manifest": "^2.4.33",
    "gatsby-plugin-offline": "^3.2.30",
    "gatsby-plugin-postcss": "^3.0.3",
    "gatsby-plugin-react-helmet": "^3.3.12",
    "gatsby-plugin-sharp": "^2.6.38",
    "gatsby-source-filesystem": "^2.3.32",
    "gatsby-source-storyblok": "^1.1.1",
    "gatsby-transformer-sharp": "^2.5.16",
    "postcss": "^8.1.2",
    "prop-types": "^15.7.2",
    "react": "^16.12.0",
    "react-dom": "^16.12.0",
    "react-helmet": "^6.1.0",
    "react-icons": "^3.11.0",
    "storyblok-react": "^0.1.1",
    "storyblok-rich-text-react-renderer": "^2.0.0"
  },
  "devDependencies": {
    "cross-env": "^7.0.2",
    "dotenv": "^8.2.0",
    "prettier": "2.1.2",
  },
```

```
}
```

Gridsome, on the hand, only consumed 256 MB of `node_modules` space—about half. As a result, running the dev server for Gridsome was faster, especially on my 5400 rpm hard drive. Its package dependencies are surprisingly small:

```
{
  "dependencies": {
    "gridsome": "^0.7.0",
    "gridsome-source-storyblok": "^1.1.0",
    "gridsome-plugin-tailwindcss": "^3.0.1",
    "@tailwindcss/typography": "^0.2.0",
    "vue-awesome": "^4.1.0"
  },
  "devDependencies": {
    "prettier": "^2.1.2"
  }
}
```

I must say it's incredible how one can built an identical site in Gridsome with half the dependencies.

Query Syntax

GraphQL page queries are used to generate pages. However, sometimes we need to make a component fetch its own data. For that, we need to use static queries. Below is a simplified version of the ProductsList.jsx component:

```
import React, { useState } from 'react'
import { useStaticQuery, graphql } from 'gatsby'

import ProductCard from './ProductCard'

const ProductList = () => {
  const data = useStaticQuery(graphql`
    query Products {
      allStoryblokEntry(filter: { full_slug: { regex: "/products/" } }) {
        edges {
          node {
            name
            slug
            full_slug
            content
            created_at
          }
```

```
        }
        totalCount
      }
    }
  `)

  const products = data.allStoryblokEntry.edges.map(edge => {
    return JSON.parse(edge.node.content)
  })

  const productCards = products().map(product => (
    <ProductCard product={product} key={product._uid} />
  ))

  return (
    <div className="container mx-auto mt-2">
      <div className="flex flex-wrap justify-center px-4 mt-4 md:justify-between">
        {productCards}
      </div>
    </div>
  )
}

export default ProductList
```

With Gatsby, there's the `useStaticQuery` hook used to write the GraphQL code. Let's have a look at the Gridsome version—ProductsList.vue—below:

```
<template>
  <div class="container mx-auto mt-2">
    <div class="flex flex-wrap justify-center px-4 mt-4 md:justify-between">
      <div v-for="product in sortedProducts" :key="product.id">
        <ProductCard :product="product" />
      </div>
    </div>
  </div>
</template>

<static-query>
  query {
    products: allStoryblokEntry(sortBy: "created_at", filter: {full_slug: {regex:"products" }}) {
      totalCount
      edges{
        node {
          id
          name
          full_slug
          content
```

```
          created_at
        }
      }
    }
  }
</static-query>

<script>
import ProductCard from './ProductCard'

export default {
  components: {
    ProductCard,
  },
  data() {
    return {
      products: null,
    }
  },
  created() {
    // Extract content & edge data into products
    this.products = this.$static.products.edges.map((edge) => {
      return edge.node.content
    })
  },

}
</script>
```

What you'll notice is that the Gridsome version uses the `<static-query>` tag to separate GraphQL code from the JavaScript section. I prefer this version, since you can have clean, readable code with separate sections for HTML, JavaScript, GraphQL and CSS.

With Gatsby projects, you can easily come across code that's a mix of all four in one code block, which can be messy and unreadable.

Component Management

Modern headless CMSs such as Prismic and Storyblok use custom components to organize and structure content. A full-blown website can easily have 100 components for rendering different types of content.

Vue provides us with a feature known as dynamic components to help manage component code. This feature allows the switching of components during runtime depending on the content provided. Here's an example of how I've used it in Gridsome:

```
<template>
  <Layout>
    <component
      v-if="story.content.component"
      :key="story.content._uid"
      :blok="story.content"
      :is="story.content.component"
    />
  </Layout>
</template>
```

Gridsome supports the automatic import of components, so we don't have to do it manually.

In React, there's no equivalent feature, so you have to implement the above solution using classic JavaScript code like this:

```
import Hero from './Hero'
import Page from './Page'
import Product from './Product'
import ProductList from './ProductList'
import PageContent from './PageContent'
import Placeholder from './Placeholder'

const ComponentList = {
  page: Page,
  hero: Hero,
  'product-list': ProductList,
  product: Product,
  'page-content': PageContent,
  placeholder: Placeholder,
}

const Components = type => {
  if (typeof ComponentList[type] === 'undefined') {
    return Placeholder
  }
  return ComponentList[type]
}
```

With React, you need to import each new content component you create. The code works, but you'll have more code to maintain as your project grows.

Clearly, Gridsome wins here.

Job Opportunities

We're almost at the end of this guide, and the only thing I haven't covered from my original list is job opportunities. Since this has turned into something of a Gatsby vs Gridsome comparison, let's start there.

I think it's fair to say that Gatsby takes the cake here, as it's way more popular (it recently raised $28 million in funding, for example). There are some Gridsome opportunities you can find on freelancer sites, although in my experience, Nuxt is more popular and has plenty of contract job opportunities.

Other than that, at the time of writing, a quick search on indeed.com reveals almost 4,000 positions working with Vue, compared to 55,000+ positions listed for React developers. So basically, if you're a developer looking for job opportunities, it's better to focus on React and integration with other platforms.

However, if you're a developer looking to start your own project, a Vue framework is a better fit, since you can implement new features at a fast pace. Plus, in my experience, you'll find that there are more Vue developers than there are Vue jobs available. A React developer can also start working on a Vue project within just a few hours of reading Vue's documentation.

Summary

If you were tasked with migrating an existing site with thousands of posts, my recommendation is to go with Gatsby. Gatsby is the only JavaScript static-site generator that supports incremental builds. This feature is available via Gatsby Cloud, which requires a subscription. Incremental builds are suitable if you have multiple authors publishing content multiple times a day. Gatsby also has a better approach to image optimization when it comes to integrating it with the Storyblok image optimization service.

Gridsome, on the other hand, is better for starting projects from scratch where the requirements are not fully known. Gridsome allows you to implement new features fast, as it's well documented on software patterns and has a rich plugin ecosystem as well. It supports a cached build for faster generation, but it's not as fast as incremental builds. The incremental build feature is currently planned on Gridsome's roadmap.

As for whether React or Vue is a better fit for your next project, that's for you to decide. But I hope I've provided you with plenty of jumping off points to base your decision on.