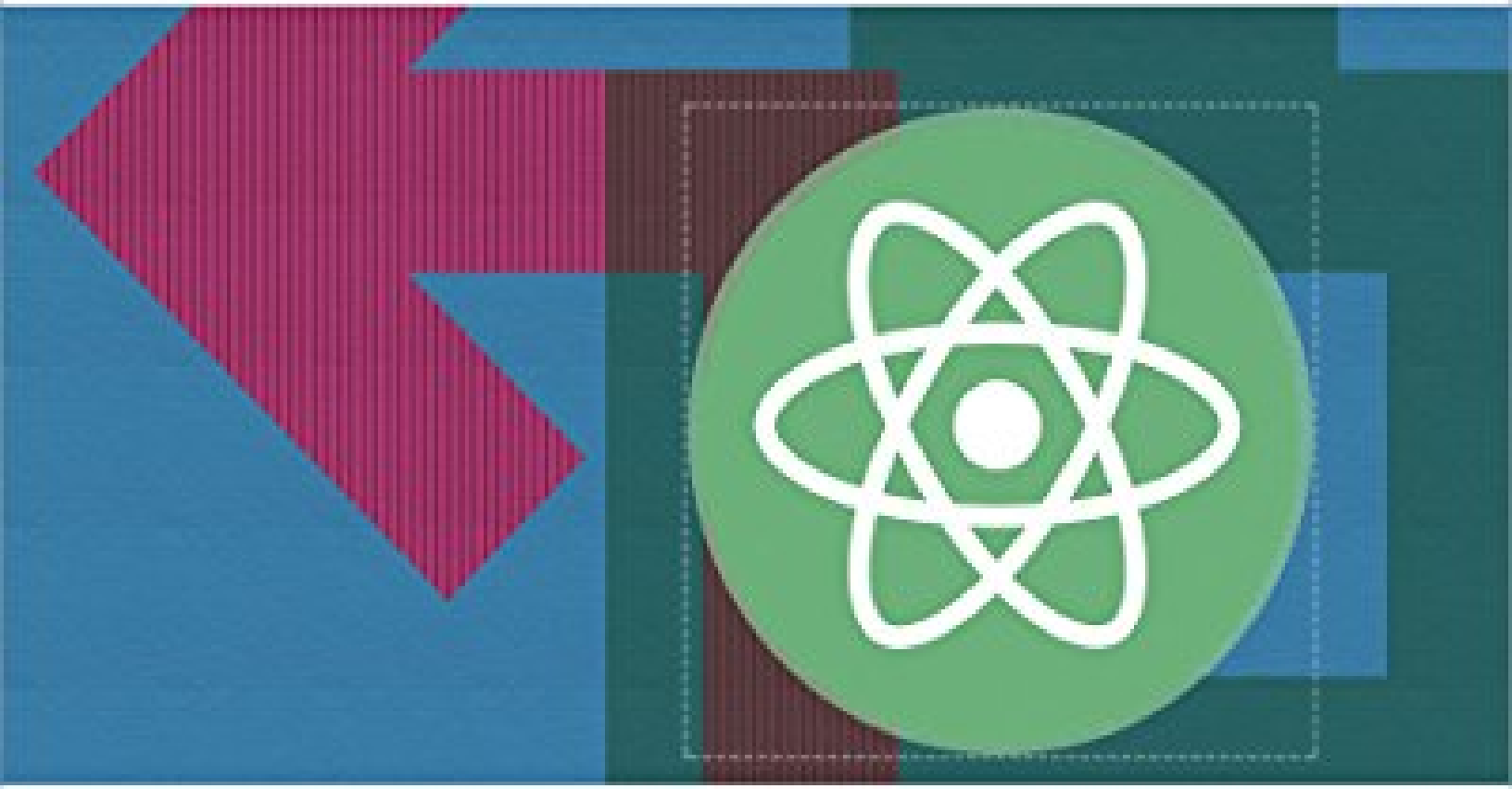
Tools

# React: Tools & Resources



sitepoint

# React: Tools & Resources



**Cover Design:** Alex Walker

## Notice of Rights



## Notice of Liability

# Trademark Notice

Rather than indicating every occurrence of a trademarked name as such, this book uses the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.


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 http://www.sitepoint.com/ to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, design, and more.

# Preface

This book is a collection of in-depth guides to some some of the tools and resources most used with React, such as Jest and React Router, as well as a discussion about how React works well with D3, and a look at Preact, a lightweight React alternative. These tutorials were all selected from SitePoint's React Hub.

# Who Should Read This Book

This book is for front-end developers with some React experience. If you're a novice, please read *Your First Week With React* before tackling this book.

# Conventions Used

## CODE SAMPLES

Code in this book is displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk in the park.
The birds were singing and the kids were all back
```

```
at school.</p>
```

Where existing code is required for context, rather than repeat all of it, ⋮ will be displayed:

```
function animate() {
    ⋮
new_variable = "Hello";
}
```

Some lines of code should be entered on one line, but we've had to wrap them because of page constraints. An ➥ indicates a line break that exists for formatting purposes only, and should be ignored:

```
URL.open("http://www.sitepoint.com/responsive-web-
➥design-real-user-testing/?responsive1");
```

You'll notice that we've used certain layout styles throughout this book to signify different types of information. Look out for the following items.

## TIPS, NOTES, AND WARNINGS

**Hey, You!**

Tips provide helpful little pointers.

### Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

### Make Sure You Always ...

... pay attention to these important points.

### Watch Out!

Warnings highlight any gotchas that are likely to trip you up along the way.

# Chapter 1: Getting Started with Redux

## BY MICHAEL WANYOIKE

A typical web application is usually composed of several UI components that share data. Often, multiple components are tasked with the responsibility of displaying different properties of the same object. This object represents state which can change at any time. Keeping state consistent among multiple components can be a nightmare, especially if there are multiple channels being used to update the same object.

Take, for example, a site with a shopping cart. At the top we have a UI component showing the number of items in the cart. We could also have another UI component that displays the total cost of items in the cart. If a user clicks the **Add to Cart** button, both of these components should update immediately with the correct figures. If the user decides to remove an item from the cart, change quantity, add a protection plan, use a coupon or change shipping location, then the relevant UI components should update to display the correct information. As you can see, a simple shopping cart can quickly become *difficult to keep in sync* as the scope of its features grows.

In this guide, I'll introduce you to a framework known as Redux, which can help you build complex projects in way that's easy to scale and maintain. To make learning easier, we'll use a simplified **shopping cart project** to learn how Redux works. You'll need to be at least familiar with the React library, as you'll later need to integrate it with Redux.

# Prerequisites

Before we get started, make sure you're familiar with the following topics:

- Functional JavaScript
- Object-oriented JavaScript
- ES6 JavaScript Syntax

Also, ensure you have the following setup on your machine:

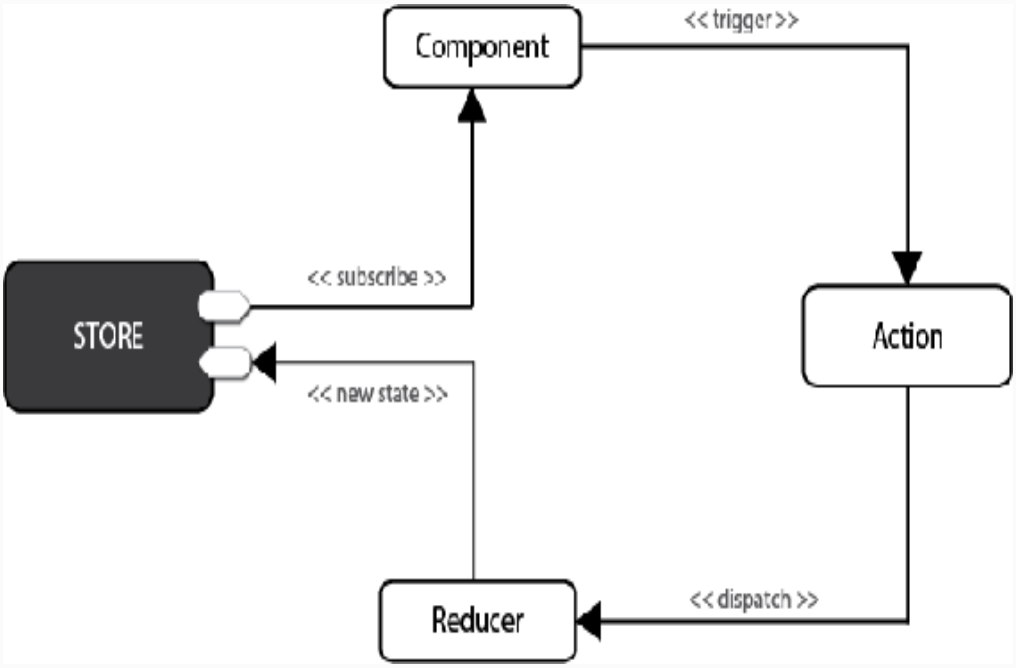
- a NodeJS environment
- a Yarn setup (recommended)

You can access the entire code used in this tutorial on GitHub.

# What is Redux?

Redux is a popular JavaScript framework that provides a predictable state container for applications. Redux is based on a simplified version of Flux, a framework developed by Facebook. Unlike standard MVC frameworks, where data can

flow between UI components and storage in both directions, Redux strictly allows data to flow in one direction only. See the below illustration:




In Redux, all data – i.e. **state** – is held in a container known as the **store**. There can only be one of these within an application. The store is essentially a state tree where states for all objects are kept. Any UI component can access the state of a particular object directly from the store. To change a state from a local or remote component, an **action** needs to be dispatched. **Dispatch** in this context means sending actionable information to the store. When a store receives an `action`, it delegates it to the relevant **reducer**. A `reducer` is simply a pure function that looks at the previous state, performs an action and returns a new state. To see all this in action, we need to start coding.

# Understand Immutability First

Before we start, I need you to first understand what **immutability** means in JavaScript. According to the Oxford English Dictionary, immutability means being **unchangeable**. In programming, we write code that changes the values of variables all the time. This is referred to as **mutability**. The way we do this can often cause unexpected bugs in our projects. If your code only deals with primitive data types (numbers, strings, booleans), then you don't need to worry. However, if you're working with Arrays and Objects, performing **mutable** operations on them can create unexpected bugs. To demonstrate this, open your terminal and launch the Node interactive shell:

```
node
```

Next, let's create an array, then later assign it to another variable:

```
> let a = [1,2,3]
> let b = a
> b.push(9)
> console.log(b)
[ 1, 2, 3, 9 ] // b output
> console.log(a)
[ 1, 2, 3, 9 ] // a output
```

As you can see, updating `array b` caused `array a` to change as well. This happens because Objects and Arrays are known **referential data types** – meaning that such data types don't actually hold values themselves, but are pointers to a

memory location where the values are stored. By assigning `a` to `b`, we merely created a second pointer that references the same location. To fix this, we need to copy the referenced values to a new location. In JavaScript, there are three different ways of achieving this:

1. using immutable data structures created by Immutable.js
2. using JavaScript libraries such as Underscore and Lodash to execute immutable operations
3. using native **ES6** functions to execute immutable operations.

For this article, we'll use the **ES6** way, since it's already available in the NodeJS environment. Inside your `NodeJS` terminal, execute the following:

```
> a = [1,2,3] // reset a
[ 1, 2, 3 ]
> b = Object.assign([],a) // copy array a to b
[ 1, 2, 3 ]
> b.push(8)
> console.log(b)
[ 1, 2, 3, 8 ] // b output
> console.log(a)
[ 1, 2, 3 ] // a output
```

In the above code example, array `b` can now be modified without affecting array `a`. We've used Object.assign() to create a new copy of values that variable `b` will now point to. We can also use the `rest operator(...)` to perform an immutable operation like this:

```
> a = [1,2,3]
[ 1, 2, 3 ]
> b = [...a, 4, 5, 6]
```

```
[ 1, 2, 3, 4, 5, 6 ]
> a
[ 1, 2, 3 ]
```

The rest operator works with object literals too! I won't go deep into this subject, but here are some additional ES6 functions that we'll use to perform immutable operations:

- spread syntax – useful in append operations
- map function – useful in an update operation
- filter function – useful in a delete operation

In case the documentation I've linked isn't useful, don't worry, as you'll see how they're used in practice. Let's start coding!

# Setting up Redux

The fastest way to set up a Redux development environment is to use the `create-react-app` tool. Before we begin, make sure you've installed and updated `nodejs`, `npm` and `yarn`. Let's set up a Redux project by generating a `redux-shopping-cart` project and installing the Redux package:

```
create-react-app redux-shopping-cart

cd redux-shopping-cart
yarn add redux # or npm install redux
```

Delete all files inside the `src` folder except `index.js`. Open the file and clear out all existing code. Type the following:

```
import { createStore } from "redux";

const reducer = function(state, action) {
  return state;
}

const store = createStore(reducer);
```

Let me explain what the above piece of code does:

- **1st statement**. We import a `createStore()` function from the Redux package.
- **2nd statement**. We create an empty function known as a **reducer**. The first argument, `state`, is current data held in the store. The second argument, `action`, is a container for:
    - **type** – a simple string constant e.g. `ADD`, `UPDATE`, `DELETE` etc.
    - **payload** – data for updating state
- **3rd statement**. We create a Redux store, which can only be constructed using a reducer as a parameter. The data kept in the Redux store can be accessed directly, but can only be updated via the supplied reducer.

You may have noticed I mentioned current data as if it already exists. Currently, our `state` is undefined or null. To remedy this, just assign a default value to state like this to make it an empty array:

```
const reducer = function(state=[], action) {
  return state;
}
```

Now, let's get practical. The reducer we created is generic. Its name doesn't describe what it's for. Then there's the issue of

how we work with multiple reducers. The answer is to use a `combineReducers` function that's supplied by the Redux package. Update your code as follows:

```
// src/index.js
…
import { combineReducers } from 'redux';

const productsReducer = function(state=[], action)
{
  return state;
}

const cartReducer = function(state=[], action) {
  return state;
}

const allReducers = {
  products: productsReducer,
  shoppingCart: cartReducer
}

const rootReducer = combineReducers(allReducers);

let store = createStore(rootReducer);
```

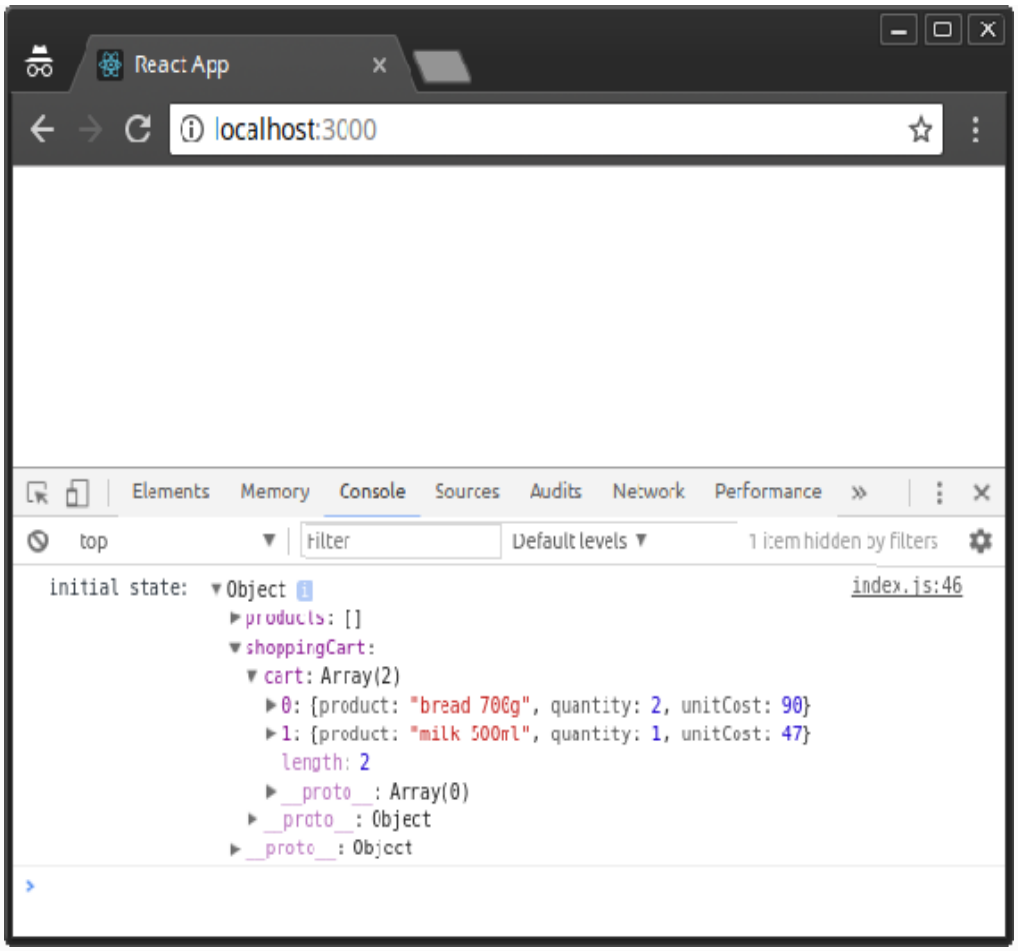
In the code above, we've renamed the generic reducer to `cartReducer`. There's also a new empty reducer named `productsReducer` that I've created just to show you how to combine multiple reducers within a single store using the `combineReducers` function.

Next, we'll look at how we can define some test data for our reducers. Update the code as follows:

```
// src/index.js
…
const initialState = {
```

```
  cart: [
    {
      product: 'bread 700g',
      quantity: 2,
      unitCost: 90
    },
    {
      product: 'milk 500ml',
      quantity: 1,
      unitCost: 47
    }
  ]
}

const cartReducer = function(state=initialState,
action) {
  return state;
}
…
let store = createStore(rootReducer);

console.log("initial state: ", store.getState());
```

Just to confirm that the store has some initial data, we use `store.getState()` to print out the current state in the console. You can run the dev server by executing `npm start` or `yarn start` in the console. Then press `Ctrl+Shift+I` to open the inspector tab in Chrome in order to view the console tab.




Currently, our `cartReducer` does nothing, yet it's supposed to manage the state of our shopping cart items within the Redux store. We need to define actions for adding, updating and deleting shopping cart items. Let's start by defining logic for a `ADD_TO_CART` action:

```
// src/index.js
…
const ADD_TO_CART = 'ADD_TO_CART';

const cartReducer = function(state=initialState,
action) {
  switch (action.type) {
    case ADD_TO_CART: {
      return {
        ...state,
```

```
          cart: [...state.cart, action.payload]
        }
      }

      default:
        return state;
    }
  }
  …
```

Take your time to analyze and understand the code. A reducer is expected to handle different action types, hence the need for a `SWITCH` statement. When an action of type `ADD_TO_CART` is dispatched anywhere in the application, the code defined here will handle it. As you can see, we're using the information provided in `action.payload` to combine to an existing state in order to create a new state.

Next, we'll define an `action`, which is needed as a parameter for `store.dispatch()`. **Actions** are simply JavaScript objects that must have `type` and an optional payload. Let's go ahead and define one right after the `cartReducer` function:

```
…
function addToCart(product, quantity, unitCost) {
  return {
    type: ADD_TO_CART,
    payload: { product, quantity, unitCost }
  }
}
…
```

Here, we've defined a function that returns a plain JavaScript object. Nothing fancy. Before we dispatch, let's add some code

that will allow us to listen to store event changes. Place this code right after the `console.log()` statement:

```
…
let unsubscribe = store.subscribe(() =>
  console.log(store.getState())
);

unsubscribe();
```

Next, let's add several items to the cart by dispatching actions to the store. Place this code before `unsubscribe()`:

```
…
store.dispatch(addToCart('Coffee 500gm', 1, 250));
store.dispatch(addToCart('Flour 1kg', 2, 110));
store.dispatch(addToCart('Juice 2L', 1, 250));
```

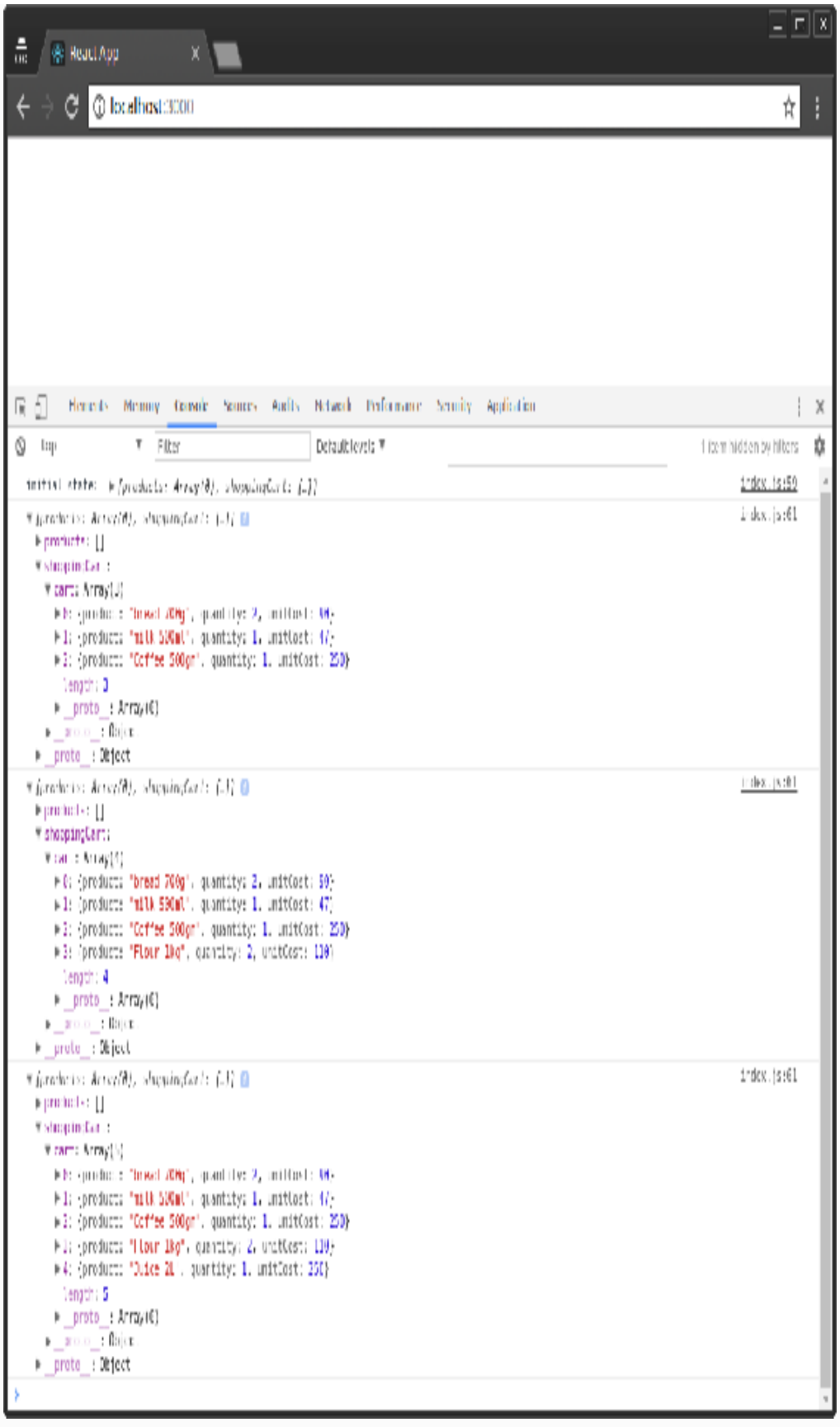
For clarification purposes, I'll illustrate below how the entire code should look after making all the above changes:

```
// src/index.js

import { createStore } from "redux";
import { combineReducers } from 'redux';

const productsReducer = function(state=[], action)
{
  return state;
}

const initialState = {
  cart: [
    {
      product: 'bread 700g',
      quantity: 2,
      unitCost: 90
    },
    {
```

```
      product: 'milk 500ml',
      quantity: 1,
      unitCost: 47
    }
  ]
}

const ADD_TO_CART = 'ADD_TO_CART';

const cartReducer = function(state=initialState,
action) {
  switch (action.type) {
    case ADD_TO_CART: {
      return {
        ...state,
        cart: [...state.cart, action.payload]
      }
    }

    default:
      return state;
  }
}

function addToCart(product, quantity, unitCost) {
  return {
    type: ADD_TO_CART,
    payload: {
      product,
      quantity,
      unitCost
    }
  }
}

const allReducers = {
  products: productsReducer,
  shoppingCart: cartReducer
}

const rootReducer = combineReducers(allReducers);

let store = createStore(rootReducer);

console.log("initial state: ", store.getState());

let unsubscribe = store.subscribe(() =>
  console.log(store.getState())
```

```
);

store.dispatch(addToCart('Coffee 500gm', 1, 250));
store.dispatch(addToCart('Flour 1kg', 2, 110));
store.dispatch(addToCart('Juice 2L', 1, 250));

unsubscribe();
```

After you've saved your code, Chrome should automatically refresh. Check the console tab to confirm that the new items have been added:


React App
localhost:3000
Elements
Memory
Console
Sources
Audits
Network
Performance
Security
Application
top
Filter
Default levels
initial state: {products: Array(0), shoppingCart: {…}}
index.js:50
index.js:61
products: []
shoppingCart:
cart: Array(3)
0: {product: "bread 700g", quantity: 2, unitCost: 90}
1: {product: "milk 500ml", quantity: 1, unitCost: 47}
2: {product: "Coffee 500gm", quantity: 1, unitCost: 250}
length: 3
__proto__: Array(0)
__proto__: Object
__proto__: Object
index.js:61
products: []
shoppingCart:
cart: Array(4)
0: {product: "bread 700g", quantity: 2, unitCost: 90}
1: {product: "milk 500ml", quantity: 1, unitCost: 47}
2: {product: "Coffee 500gm", quantity: 1, unitCost: 250}
3: {product: "Flour 1kg", quantity: 2, unitCost: 110}
length: 4
__proto__: Array(0)
__proto__: Object
__proto__: Object
index.js:61
products: []
shoppingCart:
cart: Array(5)
0: {product: "bread 700g", quantity: 2, unitCost: 90}
1: {product: "milk 500ml", quantity: 1, unitCost: 47}
2: {product: "Coffee 500gm", quantity: 1, unitCost: 250}
3: {product: "Flour 1kg", quantity: 2, unitCost: 110}
4: {product: "Juice 2L", quantity: 1, unitCost: 250}
length: 5
__proto__: Array(0)
__proto__: Object
__proto__: Object

# Organizing Redux Code

The `index.js` file has quickly grown large. This is not how Redux code is written. I've only done this to show you how simple Redux is. Let's look at how a Redux project should be organized. First, create the following folders and files within the `src` folder, as illustrated below:

```
src/
├── actions
│   └── cart-actions.js
├── index.js
├── reducers
│   ├── cart-reducer.js
│   ├── index.js
│   └── products-reducer.js
└── store.js
```

Next, let's start moving code from `index.js` to the relevant files:

```
// src/actions/cart-actions.js

export const ADD_TO_CART = 'ADD_TO_CART';

export function addToCart(product, quantity,
unitCost) {
  return {
    type: ADD_TO_CART,
    payload: { product, quantity, unitCost }
  }
}
```

```
// src/reducers/products-reducer.js

export default function(state=[], action) {
  return state;
}
```

```
// src/reducers/cart-reducer.js

import  { ADD_TO_CART }  from '../actions/cart-
actions';

const initialState = {
  cart: [
    {
      product: 'bread 700g',
      quantity: 2,
      unitCost: 90
    },
    {
      product: 'milk 500ml',
      quantity: 1,
      unitCost: 47
    }
  ]
}

export default function(state=initialState,
action) {
  switch (action.type) {
    case ADD_TO_CART: {
      return {
        ...state,
        cart: [...state.cart, action.payload]
      }
    }

    default:
      return state;
  }
}
```

```
// src/reducers/index.js

import { combineReducers } from 'redux';
import productsReducer from './products-reducer';
import cartReducer from './cart-reducer';

const allReducers = {
  products: productsReducer,
  shoppingCart: cartReducer
```

```
}

const rootReducer = combineReducers(allReducers);

export default rootReducer;
```

```
// src/store.js

import { createStore } from "redux";
import rootReducer from './reducers';

let store = createStore(rootReducer);

export default store;
```

```
// src/index.js

import store from './store.js';
import { addToCart }  from './actions/cart-
actions';

console.log("initial state: ", store.getState());

let unsubscribe = store.subscribe(() =>
  console.log(store.getState())
);

store.dispatch(addToCart('Coffee 500gm', 1, 250));
store.dispatch(addToCart('Flour 1kg', 2, 110));
store.dispatch(addToCart('Juice 2L', 1, 250));

unsubscribe();
```

After you've finished updating the code, the application should run as before now that it's better organized. Let's now look at how we can update and delete items from the shopping cart. Open `cart-reducer.js` and update the code as follows:

```
// src/reducers/cart-actions.js
…
```

```
export const UPDATE_CART = 'UPDATE_CART';
export const DELETE_FROM_CART =
'DELETE_FROM_CART';
…
export function updateCart(product, quantity,
unitCost) {
  return {
    type: UPDATE_CART,
    payload: {
      product,
      quantity,
      unitCost
    }
  }
}

export function deleteFromCart(product) {
  return {
    type: DELETE_FROM_CART,
    payload: {
      product
    }
  }
}
```

Next, update `cart-reducer.js` as follows:

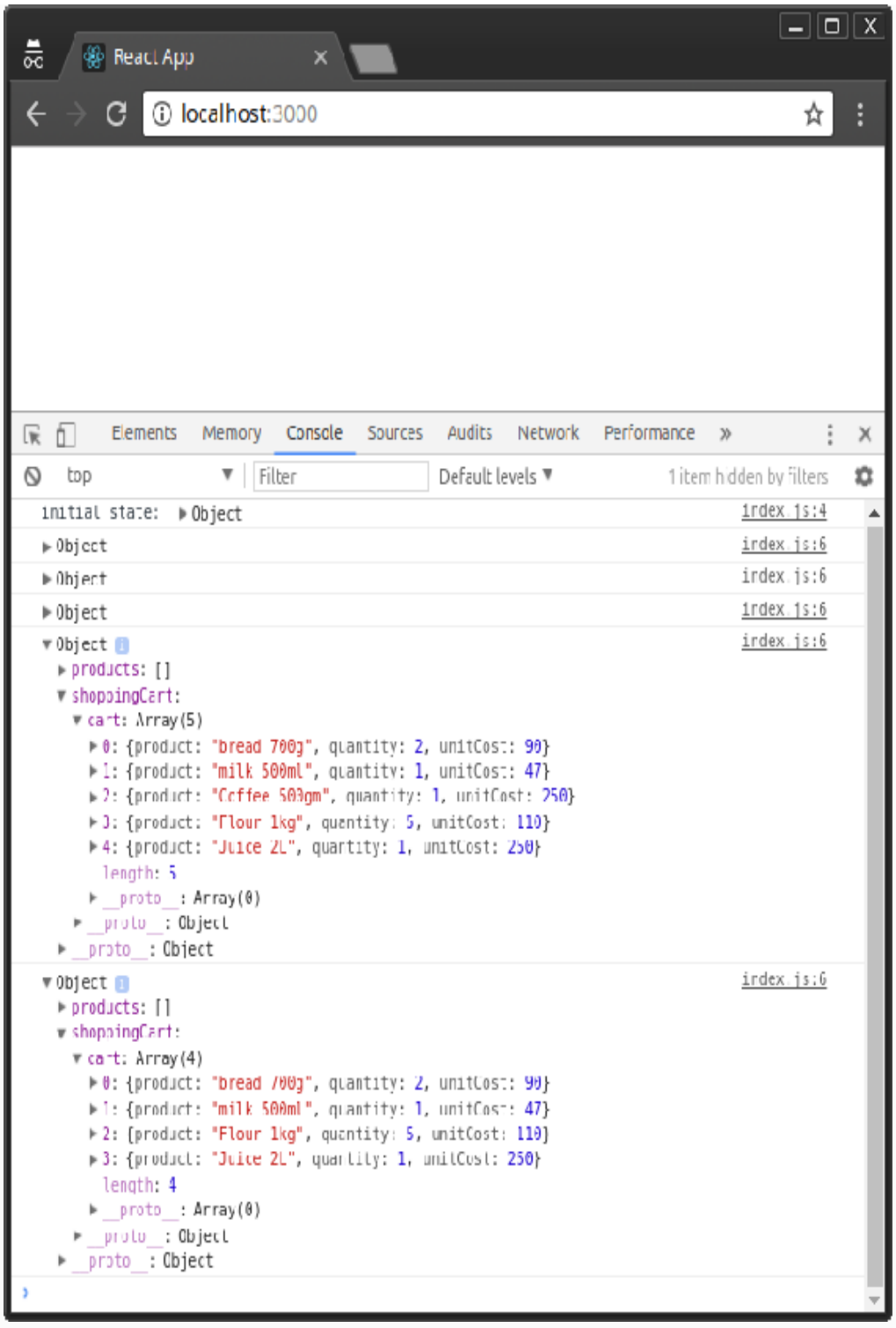
```
// src/reducers/cart-reducer.js
…
export default function(state=initialState,
action) {
  switch (action.type) {
    case ADD_TO_CART: {
      return {
        ...state,
        cart: [...state.cart, action.payload]
      }
    }

    case UPDATE_CART: {
      return {
        ...state,
        cart: state.cart.map(item => item.product
=== action.payload.product ?
        ➥action.payload : item)
```

```
      }
    }

    case DELETE_FROM_CART: {
      return {
        ...state,
        cart: state.cart.filter(item =>
item.product !== action.payload.product)
      }
    }

    default:
      return state;
  }
}
```

Finally, let's dispatch the `UPDATE_CART` and `DELETE_FROM_CART` actions in `index.js`:

```
// src/index.js
…
// Update Cart
store.dispatch(updateCart('Flour 1kg', 5, 110));

// Delete from Cart
store.dispatch(deleteFromCart('Coffee 500gm'));
…
```

Your browser should automatically refresh once you've saved all the changes. Check the console tab to confirm the results:

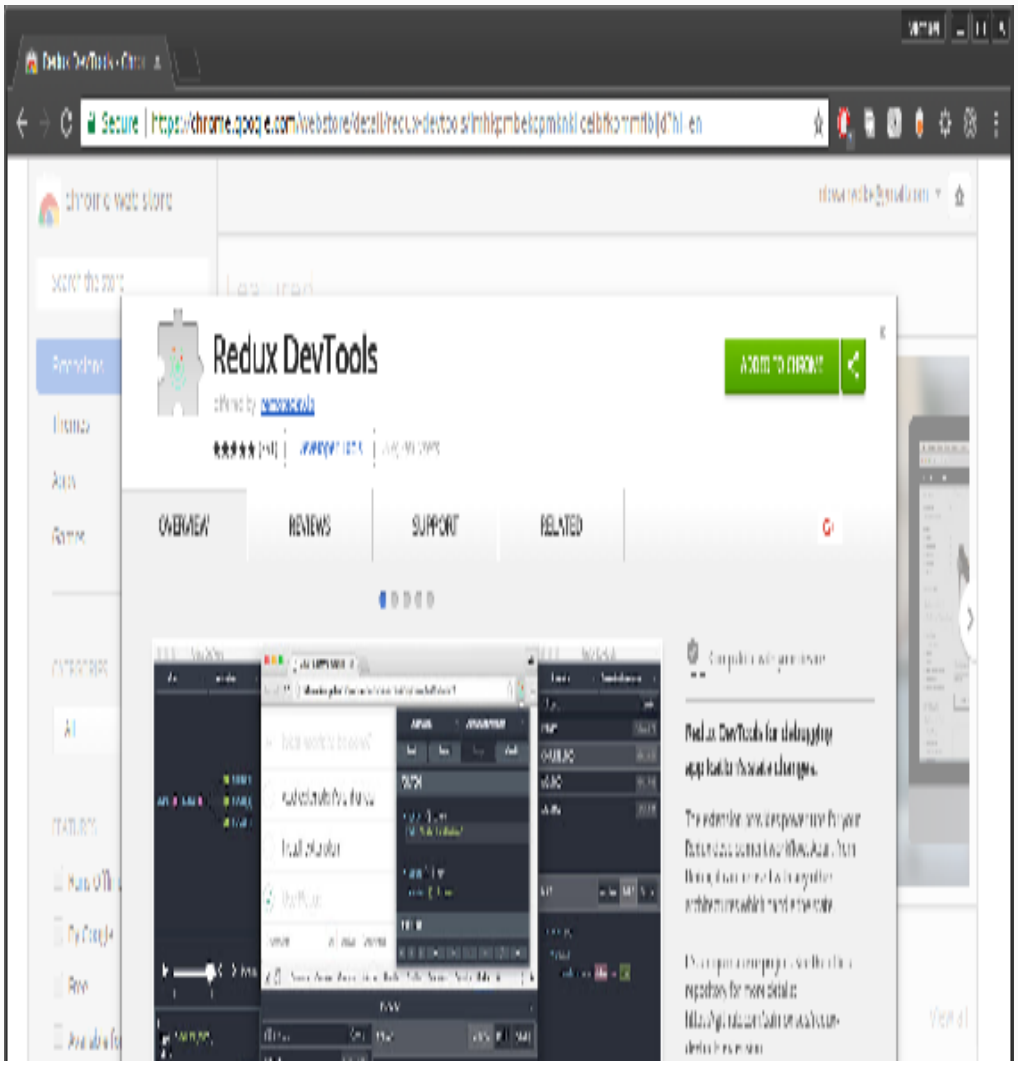



As confirmed, the quantity for 1kg of flour is updated from 2 to 5, while the the 500gm of coffee gets deleted from cart.

# Debugging with Redux tools

Now, if we've made a mistake in our code, how do we debug a Redux project?

Redux comes with a lot of third-party debugging tools we can use to analyze code behavior and fix bugs. Probably the most popular one is the **time-travelling tool**, otherwise known as redux-devtools-extension. Setting it up is a 3-step process. First, go to your Chrome browser and install the Redux Devtools extension.




Next, go to your terminal where your Redux application is running and press `Ctrl+C` to stop the development server. Next, use npm or yarn to install the redux-devtools-extension package. Personally, I prefer Yarn, since there's a `yarn.lock` file that I'd like to keep updated.

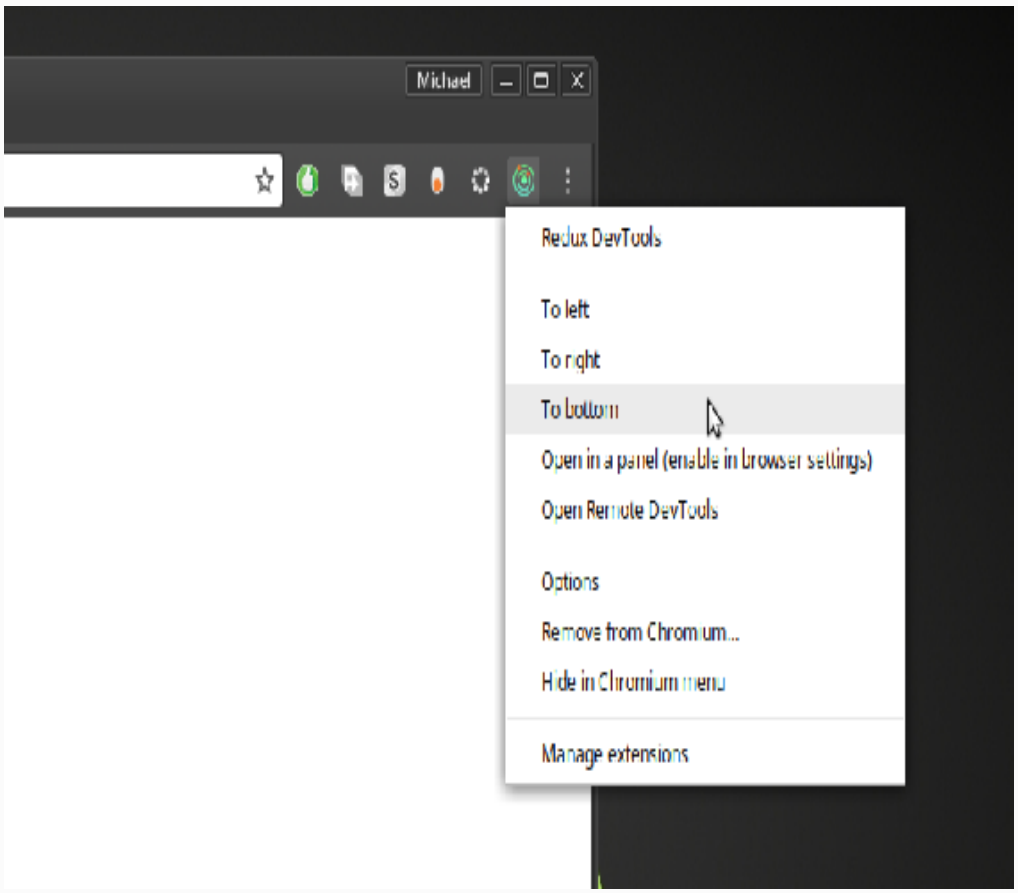
```
yarn add redux-devtools-extension
```

Once installation is complete, you can start the development server as we implement the final step of implementing the tool. Open `store.js` and replace the existing code as follows:

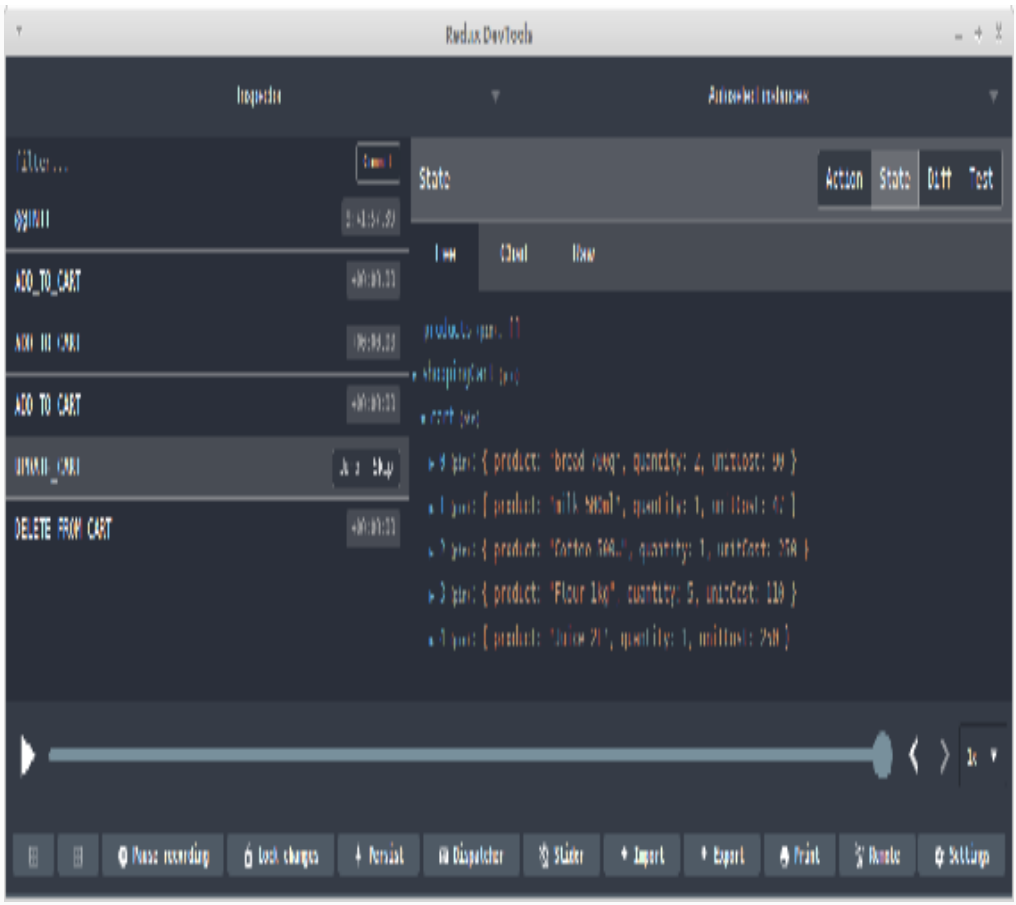
```
// src/store.js
import { createStore } from "redux";
import { composeWithDevTools } from 'redux-
devtools-extension';
import rootReducer from './reducers';
```

```
const store = createStore(rootReducer,
composeWithDevTools());

export default store;
```

Feel free to update `src/index.js` and remove all code related with logging to the console and subscribing to the store. This is no longer needed. Now, go back to Chrome and open the Redux DevTools panel by right-clicking the tool's icon:




In my case, I've selected to **To Bottom** option. Feel free to try out other options.




As you can see, the Redux Devtool is quite amazing. You can toggle between action, state and diff methods. Select actions on the left panel and observe how the state tree changes. You can also use the slider to play back the sequence of actions. You can even dispatch directly from the tool! Do check out the documentation to learn more on how you can further customize the tool to your needs.

## Integration with React

At the beginning of this tutorial, I mentioned Redux really pairs well with React. Well, you only need a few steps to setup the integration. Firstly, stop the development server, as we'll

need to install the react-redux package, the official Redux bindings for React:

```
yarn add react-redux
```

Next, update `index.js` to include some React code. We'll also use the `Provider` class to wrap the React application within the Redux container:

```
// src/index.js
…
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';

const App = <h1>Redux Shopping Cart</h1>;

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

Just like that, we've completed the first part of the integration. You can now start the server to see the result. The second part involves linking React's components with the Redux store and actions using a couple of functions provided by the `react-redux` package that we just installed. In addition, you'll need to set up an API using Express or a framework like Feathers. The API will provide our application with access to a database service.

In Redux, we'll also need to install further packages such as `axios` to perform API requests via Redux actions. Our React components state will then be handled by Redux, making sure that all components are in sync with the database API. To learn more on how to accomplish all this, do take a look at my other tutorial, "Build a CRUD App Using React, Redux and FeathersJS".

## Summary

I hope this guide has given you a useful introduction to Redux. There's still quite a bit more for you to learn, though. For example, you need to learn how to deal with async actions, authentication, logging, handling forms and so on. Now that you know what Redux is all about, you'll find it easier to try out other similar frameworks, such as Flux, Alt.js or Mobx. If you feel Redux is right for you, I highly recommend the following tutorials that will help you gain even more experience in Redux:

- Redux State Management in Vanilla JavaScript
- Redux Logging in Production with LogRocket
- Build a CRUD App Using React, Redux and FeathersJS
- Dealing with Asynchronous APIs in Server-rendered React

# Chapter 2: React Router v4: The Complete Guide

## BY MANJUNATH M

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 v4 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 the 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 in an application 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 – e.g. `www.example.com/products`.
- The browser's back and forward button should work as expected.
- The dynamically generated nested views should preferably have a URL of their own too – e.g. `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" component={About}/>
```

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

### React Router is a Third-party Library

A note before getting started. 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. If your requirements are limited to routers for navigation, you could implement a custom router from scratch without much hassle. However, understanding the basics of React Router will give you better insights into how a router should work.

# Overview

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

1. basic navigational routing
2. nested routing
3. nested routing with path parameters
4. 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. Once you're inside a particular demo directory, run `npm install` to install the dependencies. To serve the application on a development server, run `npm start` and head over to `http://localhost:3000/` to see the demo in action.

Let's get started!

# Setting up React Router

I assume you already have a development environment up and running. If not, head over to "Getting Started with React and JSX". Alternatively, you can use Create React App to generate the files required for creating a basic React project. This is the default directory structure generated by Create React App:

```
react-routing-demo-v4
   ├── .gitignore
   ├── package.json
   ├── public
   │   ├── favicon.ico
   │   ├── index.html
   │   └── manifest.json
   ├── README.md
   ├── src
   │   ├── App.css
   │   ├── App.js
   │   ├── App.test.js
   │   ├── index.css
   │   ├── index.js
   │   ├── logo.svg
   │   └── registerServiceWorker.js
   └── yarn.lock
```

The React Router library comprises three packages: `react-router`, `react-router-dom`, and `react-router-native`. `react-router` is the core package for the 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 on a mobile app development environment using React Native.

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

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

## React Router Basics

Here's an example of how our routes will look:

```
<Router>
  <Route exact path="/" component={Home}/>
  <Route path="/category" component={Category}/>
  <Route path="/login" component={Login}/>
  <Route path="/products" component={Products}/>
</Router>
```

## ROUTER

You need a router component and several route components to set up a basic route as exemplified above. Since we're building a browser-based application, we can use two types of routers from the React Router API:

1. `<BrowserRouter>`
2. `<HashRouter>`

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

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

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

The `<BrowserRouter>` is more popular amongst the two because it uses the HTML5 History API to keep track of your router history. The `<HashRouter>`, on the other hand, uses the hash portion of the URL (`window.location.hash`) to remember things. If you intend to support legacy browsers, you should stick with `<HashRouter>`.

Wrap the `<BrowserRouter>` component around the App component.

### index.js

```
/* Import statements */
import React from 'react';
import ReactDOM from 'react-dom';

/* App is the entry point to the React code.*/
import App from './App';

/* import BrowserRouter from 'react-router-dom' */
import { BrowserRouter } from 'react-router-dom';

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

#### A Router Component Can Only Have a Single Child Element

A router component can only have a single child element. The child element can be an HTML element – such as a div – or a React component.

For the React Router to work, you need to import the relevant API from the `react-router-dom` library. Here I've imported the `BrowserRouter` into `index.js`. I've also imported the `App` component from `App.js`. `App.js`, as you might have guessed, is the entry point to React components.

The above code creates an instance of history for our entire App component. Let me formally introduce you to history.

## HISTORY

`history` is a JavaScript library that lets you easily manage session history anywhere JavaScript runs. history provides a minimal API that lets you manage the history stack, navigate, confirm navigation, 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. `history.push()` is invoked when you click on a `<Link>` component, and `history.replace()` is called when you use `<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.

## LINKS AND ROUTES

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 pathname is matched with 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 browser 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 browser refresh.

We've covered everything you need to know to create a basic router. Let's build one.

## DEMO 1: BASIC ROUTING

### src/App.js

```
/* Import statements */
import React, { Component } from 'react';
import { Link, Route, Switch } from 'react-router-
dom';

/* Home component */
const Home = () => (
  <div>
    <h2>Home</h2>
  </div>
)

/* Category component */
const Category = () => (
  <div>
    <h2>Category</h2>
  </div>
)

/* Products component */
```

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

/* App component */
class App extends React.Component {
  render() {
    return (
      <div>
        <nav className="navbar navbar-light">
          <ul className="nav navbar-nav">

           /* Link components are used for linking
to other views */
            <li><Link to="/">Homes</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="/" component={Home}/>
           <Route path="/category" component=
{Category}/>
           <Route path="/products" component=
{Products}/>

      </div>
    )
  }
}
```

We've declared the components for Home, Category and Products inside `App.js`. Although this is okay for now, when the 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. The component that should be rendered is passed in as a second prop.

Here `/` matches both `/` and `/category`. Therefore, both the routes are matched and rendered. How do we avoid that? You should pass the `exact= {true}` props to the router with `path='/'`:

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

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

## Nested Routing

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

`<Route>` has three props that you can you use to define what gets rendered:

- **component**. We've already seen this in action. When the URL is matched, the router creates a React element from the given component using `React.createElement`.

- **render**. This is handy for inline rendering. The render prop expects a function that returns an element when the location matches the route's path.
- **children**. The children prop is similar to render in that it expects a function that returns a React element. However, children gets rendered regardless of whether the path is matched with the location or not.

## PATH AND MATCH

The ***path*** 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 and we call it the **match** object. The match object carries 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>`s
- `match.path`. A string that returns the route's path string – that is, `<Route path="">`. We'll be using this to build nested `<Route>`s.
- `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.

Now that we know all about `<Route>`s, let's build a router with nested routes.

## SWITCH COMPONENT

Before we head for 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="/" component={Home}/>
<Route path="/products" component={Products}/>
<Route path="/category" component={Category}/>
<Route path="/:id" render = {()=> (<p> I want this
text to show up for all routes
    ➥ other than '/', '/products' and '/category'
</p>)}/>
```

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.

## DEMO 2: NESTED ROUTING

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

**src/App.js**

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

class App extends Component {
  render() {

    return (
      <div>
        <nav className="navbar navbar-light">
          <ul className="nav navbar-nav">
            <li><Link to="/">Homes</Link></li>
            <li><Link
to="/category">Category</Link></li>
            <li><Link
to="/products">Products</Link></li>
          </ul>
       </nav>

    <Switch>
      <Route exact path="/" component={Home}/>
      <Route path="/category" component=
{Category}/>
       <Route path="/products" component=
{Products}/>
    </Switch>

    </div>
    );
  }
}
export default App;

/* Code for Home and Products component omitted
for brevity */
```

Unlike the earlier version of React Router, in version 4, the nested `<Route>`s should preferably go inside the parent component. That is, the Category component is the parent here, and we'll be declaring the routes for `category/:name` inside the parent component.

### src/Category.jsx

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

const Category = ({ match }) => {
return( <div> <ul>
    <li><Link to=
{`${match.url}/shoes`}>Shoes</Link></li>
    <li><Link to=
{`${match.url}/boots`}>Boots</Link></li>
    <li><Link to=
{`${match.url}/footwear`}>Footwear</Link></li>

  </ul>
  <Route path={`${match.path}/:name`} render=
{({match}) =>( <div> <h3>
  ➥ {match.params.name} </h3></div>)}/>
  </div>)
}
export default Category;
```

First, we've declared a couple of links for the nested routes. 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(match)` provides some useful information that might help to clarify it.

```
<Route path={`${match.path}/:name`}
  render= {({match}) =>( <div> <h3>
{match.params.name} </h3></div>)}/>
```

This is our first attempt at dynamic routing. Instead of hard-coding the routes, we've used a variable within the pathname. `:name` is a path parameter and catches everything after `category/` until another forward slash is encountered. So, a

pathname like `products/running-shoes` will create a `params` object as follows:

```
{
  name: 'running-shoes'
}
```

The captured data should be accessible at `match.params` or `props.match.params` depending on how the props are passed. The other interesting thing is that we've used a `render` prop. `render` props are pretty handy for inline functions that don't require a component of their own.

## DEMO 3: 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. Assume that we have the product data returned by a server API of the form below.

### src/Products.jsx

```
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'
},

];
```

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.

## src/Products.jsx

```
/* Import statements have been left out for code
brevity */

const Products = ({ match }) => {

   const productsData = [
    {
        id: 1,
```

```
        name: 'NIKE Liteforce Blue Sneakers',
        description: 'Lorem ipsum dolor sit amet,
consectetur adipiscing elit.
        ➥ Proin molestie.',
        status: 'Available'

    },

    //Rest of the data has been left out for code
brevity

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

    })

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

        <Route path={`${match.url}/:productId`}
            render={ (props) => <Product data=
{productsData} {...props} />}/>
        <Route exact path={match.url}
            render={() => (
            <div>Please select a product.</div>
            )}
        />
    </div>
  )
}
```

First, we created a list of `<Links>`s using the `productsData.id`s and stored it in `linkList`. The route takes a parameter in the path string which corresponds to that of the product id.

```
<Route path={`${match.url}/:productId`}
  render={ (props) => <Product data=
{productsData} {...props} />}/>
```

You may have expected `component = { Product }` instead of the inline render function. The problem is that we need to pass down `productsData` to the Product component along with all the existing props. Although there are other ways you can do this, I find this method to be the easiest. `{...props}` uses the ES6's spread syntax to pass the whole props object to the component.

Here's the code for Product component.

### src/Product.jsx

```
/* Import statements have been left out for code
brevity */

const Product = ({match,data}) => {
  var product= data.find(p => p.id ==
match.params.productId);
  var productData;

  if(product)
    productData = <div>
      <h3> {product.name} </h3>
      <p>{product.description}</p>
      <hr/>
      <h4>{product.status}</h4>  </div>;
  else
    productData = <h2> Sorry. Product doesnt exist
```

```
  </h2>;

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

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.

# Protecting Routes

For the final demo, we'll be discussing techniques concerned with protecting routes. So, if someone tries to access `/admin`, they'd be required to log in first. However, there are some things we need to cover before we can protect routes.

## REDIRECT

Like the server-side redirects, `<Redirect>` 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:
props.location}}}
```

So, if someone tries to access the `/admin` while logged out, they'll be redirected to the `/login` route. The information about the current location is passed via state, so that if the authentication is successful, the user can be redirected back to the original location. Inside the child component, you can access this information at `this.props.location.state`.

## CUSTOM ROUTES

A custom route is a fancy word for 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. Here's the custom route declared among other routes.

### src/App.js

```
/* Add the PrivateRoute component to the existing
Routes */
<Switch>
  <Route exact path="/" component={Home} data=
{data}/>
  <Route path="/category" component={Category}/>
  <Route path="/login" component={Login}/>
  <PrivateRoute authed={fakeAuth.isAuthenticated}
path='/products' component =
  ➥ {Products} />
</Switch>
```

`fakeAuth.isAuthenticated` returns true if the user is logged in and false otherwise.

Here's the definition for `PrivateRoute`:

### src/App.js

```
/* PrivateRoute component definition */
const PrivateRoute = ({component: Component,
authed, ...rest}) => {
  return (
    <Route
      {...rest}
      render={(props) => authed === true
        ? <Component {...props} />
        : <Redirect to={{pathname: '/login',
state: {from: props.
            ➥location}}} />} />
  )
}
```

The route renders the Admin component if the user is logged in. Otherwise, the user is redirected to `/login`. The good thing about this approach is that it is evidently more declarative and `PrivateRoute` is reusable.

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

### src/Login.jsx

```
import React from 'react';
import { Redirect } from 'react-router-dom';

class Login extends React.Component {

  constructor() {
    super();

    this.state = {
      redirectToReferrer: false
    }
    // binding 'this'
    this.login = this.login.bind(this);
  }

  login() {
```

```
    fakeAuth.authenticate(() => {
      this.setState({ redirectToReferrer: true })
    })
  }

  render() {
    const { from } = this.props.location.state ||
{ from: { pathname: '/' } }
    const { redirectToReferrer } = this.state;

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

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

/* A fake authentication function */
export const fakeAuth = {

  isAuthenticated: false,
  authenticate(cb) {
    this.isAuthenticated = true
     setTimeout(cb, 100)
  },
}
```

The line below demonstrates object destructuring, which is a part of the ES6 specification.

```
const { from } = this.props.location.state || {
from: { pathname: '/' } }
```

## DEMO 4: PROTECTING ROUTES

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

**CodeSandbox Example**

https://codesandbox.io/embed/nn8x24vm60

# Summary

As you've seen in this article, React Router is a powerful library that complements React for building better, declarative routes. Unlike the prior versions of React Router, in v4, 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 setup 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

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

# Chapter 3: How to Test React Components Using Jest

## BY JACK FRANKLIN

In this tutorial, we'll take a look at using Jest – a testing framework maintained by Facebook – to test our ReactJS components. We'll look at how we can use Jest first on plain JavaScript functions, before looking at some of the features it provides out of the box specifically aimed at making testing React apps easier. It's worth noting that Jest isn't aimed specifically at React: you can use it to test any JavaScript applications. However, a couple of the features it provides come in really handy for testing user interfaces, which is why it's a great fit with React.

## Sample Application

Before we can test anything, we need an application to test! Staying true to web development tradition, I've built a small todo application that we'll use as the starting point. You can find it, along with all the tests that we're about to write, on

GitHub. If you'd like to play with the application to get a feel for it, you can also find a live demo online.

The application is written in ES2015, compiled using Webpack with the Babel ES2015 and React presets. I won't go into the details of the build set up, but it's all in the GitHub repo if you'd like to check it out. You'll find full instructions in the README on how to get the app running locally. If you'd like to read more, the application is built using Webpack, and I recommend "A Beginner's guide to Webpack" as a good introduction to the tool.

The entry point of the application is `app/index.js`, which just renders the `Todos` component into the HTML:

```
render(
  <Todos />,
  document.getElementById('app')
);
```

The `Todos` component is the main hub of the application. It contains all the state (hard-coded data for this application, which in reality would likely come from an API or similar), and has code to render the two child components: `Todo`, which is rendered once for each todo in the state, and `AddTodo`, which is rendered once and provides the form for a user to add a new todo.

Because the `Todos` component contains all the state, it needs the `Todo` and `AddTodo` components to notify it whenever anything changes. Therefore, it passes functions down into

these components that they can call when some data changes, and `Todos` can update the state accordingly.

Finally, for now, you'll notice that all the business logic is contained in `app/state-functions.js`:

```
export function toggleDone(state, id) {…}

export function addTodo(state, todo) {…}

export function deleteTodo(state, id) {…}
```

These are all pure functions that take the state and some data, and return the new state. If you're unfamiliar with pure functions, they are functions that only reference data they are given and have no side effects. For more, you can read my article on A List Apart on pure functions and my article on SitePoint about pure functions and React.

If you're familiar with Redux, they're fairly similar to what Redux would call a reducer. In fact, if this application got much bigger I would consider moving into Redux for a more explicit, structured approach to data. But for this size application you'll often find that local component state and some well abstracted functions to be more than enough.

## To TDD or Not to TDD?

There have been many articles written on the pros and cons of **test-driven development**, where developers are expected to

write the tests first, before writing the code to fix the test. The idea behind this is that, by writing the test first, you have to think about the API that you're writing, and it can lead to a better design. For me, I find that this very much comes down to personal preference and also to the sort of thing I'm testing. I've found that, for React components, I like to write the components first and then add tests to the most important bits of functionality. However, if you find that writing tests first for your components fits your workflow, then you should do that. There's no hard rule here; do whatever feels best for you and your team.

> Note that this article will focus on testing front-end code. If you're looking for something focused on the back end, be sure to check out SitePoint's course Test-Driven Development in Node.js.

## Introducing Jest

Jest was first released in 2014, and although it initially garnered a lot of interest, the project was dormant for a while and not so actively worked on. However, Facebook has invested the last year into improving Jest, and recently published a few releases with impressive changes that make it worth reconsidering. The only resemblance of Jest compared to the initial open-source release is the name and the logo. Everything else has been changed and rewritten. If you'd like to find out more about this, you can read Christoph Pojer's comment, where he discusses the current state of the project.

If you've been frustrated by setting up Babel, React and JSX tests using another framework, then I definitely recommend giving Jest a try. If you've found your existing test setup to be slow, I also highly recommend Jest. It automatically runs tests in parallel, and its watch mode is able to run only tests relevant to the changed file, which is invaluable when you have a large suite of tests. It comes with JSDom configured, meaning you can write browser tests but run them through Node, can deal with asynchronous tests and has advanced features such as mocking, spies and stubs built in.

## Installing and Configuring Jest

To start with, we need to get Jest installed. Because we're also using Babel, we'll install another couple of modules that make Jest and Babel play nicely out of the box:

```
npm install --save-dev babel-jest babel-polyfill
babel-preset-es2015
        ➥ babel-preset-react jest
```

You also need to have a `.babelrc` file with Babel configured to use any presets and plugins you need. The sample project already has this file, which looks like so:

```
{
  "presets": ["es2015", "react"]
}
```

We won't install any React testing tools yet, because we're not going to start with testing our components, but our state functions.

Jest expects to find our tests in a `__tests__` folder, which has become a popular convention in the JavaScript community, and it's one we're going to stick to here. If you're not a fan of the `__tests__` setup, out of the box Jest also supports finding any `.test.js` and `.spec.js` files too.

As we'll be testing our state functions, go ahead and create `__tests__/state-functions.test.js`.

We'll write a proper test shortly, but for now, put in this dummy test, which will let us check everything's working correctly and we have Jest configured.

```
describe('Addition', () => {
  it('knows that 2 and 2 make 4', () => {
    expect(2 + 2).toBe(4);
  });
});
```

Now, head into your `package.json`. We need to set up `npm test` so that it runs Jest, and we can do that simply by setting the `test` script to run `jest`.

```
"scripts": {
  "test": "jest"
}
```

If you now run `npm test` locally, you should see your tests run, and pass!

```
PASS  __tests__/state-functions.test.js
  Addition
    ✓ knows that 2 and 2 make 4 (5ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 passed, 0 total
Time:        3.11s
```

If you've ever used Jasmine, or most testing frameworks, the above test code itself should be pretty familiar. Jest lets us use `describe` and `it` to nest tests as we need to. How much nesting you use is up to you; I like to nest mine so all the descriptive strings passed to `describe` and `it` read almost as a sentence.

When it comes to making actual assertions, you wrap the thing you want to test within an `expect()` call, before then calling an assertion on it. In this case, we've used `toBe`. You can find a list of all the available assertions in the Jest documentation. `toBe` checks that the given value matches the value under test, using `===` to do so. We'll meet a few of Jest's assertions through this tutorial.

## Testing Business Logic

Now we've seen Jest work on a dummy test, let's get it running on a real one! We're going to test the first of our state

functions, `toggleDone`. `toggleDone` takes the current state and the ID of a todo that we'd like to toggle. Each todo has a `done` property, and `toggleDone` should swap it from `true` to `false`, or vice-versa.

> If you're following along with this, make sure you've cloned the repo and have copied the `app` folder to the same directory that contains your `___tests__` folder. You'll also need to install the `shortid` package (`npm install shortid --save`), which is a dependency of the Todo app.

I'll start by importing the function from `app/state-functions.js`, and setting up the test's structure. Whilst Jest allows you to use `describe` and `it` to nest as deeply as you'd like to, you can also use `test`, which will often read better. `test` is just an alias to Jest's `it` function, but can sometimes make tests much easier to read and less nested.

For example, here's how I would write that test with nested `describe` and `it` calls:

```
import { toggleDone } from '../app/state-
functions';

describe('toggleDone', () => {
  describe('when given an incomplete todo', () =>
{
    it('marks the todo as completed', () => {
    });
  });
});
```

And here's how I would do it with `test`:

```
import { toggleDone } from '../app/state-
functions';

test('toggleDone completes an incomplete todo', ()
=> {
});
```

The test still reads nicely, but there's less indentation getting in the way now. This one is mainly down to personal preference; choose whichever style you're more comfortable with.

Now we can write the assertion. First we'll create our starting state, before passing it into `toggleDone`, along with the ID of the todo that we want to toggle. `toggleDone` will return our finish state, which we can then assert on:

```
const startState = {
  todos: [{ id: 1, done: false, name: 'Buy Milk'
}]
};

const finState = toggleDone(startState, 1);

expect(finState.todos).toEqual([
  { id: 1, done: true, name: 'Buy Milk' }
]);
```

Notice now that I use `toEqual` to make my assertion. You should use `toBe` on primitive values, such as strings and numbers, but `toEqual` on objects and arrays. `toEqual` is built to deal with arrays and objects, and will recursively check each field or item within the object given to ensure that it matches.

With that we can now run `npm test` and see our state function test pass:

```
PASS  __tests__/state-functions.test.js
  ✓ tooggleDone completes an incomplete todo (9ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 passed, 0 total
Time:        3.166s
```

## Rerunning Tests on Changes

It's a bit frustrating to make changes to a test file and then have to manually run `npm test` again. One of Jest's best features is its watch mode, which watches for file changes and runs tests accordingly. It can even figure out which subset of tests to run based on the file that changed. It's incredibly powerful and reliable, and you're able to run Jest in watch mode and leave it all day whilst you craft your code.

To run it in watch mode, you can run `npm test -- --watch`. Anything you pass to `npm test` after the first `--` will be passed straight through to the underlying command. This means that these two commands are effectively equivalent:

- `npm test -- --watch`
- `jest --watch`

I would recommend that you leave Jest running in another tab, or terminal window, for the rest of this tutorial.

Before moving onto testing the React components, we'll write one more test on another one of our state functions. In a real application I would write many more tests, but for the sake of the tutorial, I'll skip some of them. For now, let's write a test that ensures that our `deleteTodo` function is working. Before seeing how I've written it below, try writing it yourself and seeing how your test compares.

Remember that you will have to update the `import` statement at the top to import `deleteTodo` along with `toggleTodo`:

```
import { toggleTodo, deleteTodo } from
'../app/state-functions';
```

And here's how I've written the test:

```
test('deleteTodo deletes the todo it is given', ()
=&gt; {
    const startState = {
      todos: [{ id: 1, done: false, name: 'Buy
Milk' }]
    };

    const finState = deleteTodo(startState, 1);

    expect(finState.todos).toEqual([]);
  });
```

The test doesn't vary too much from the first: we set up our initial state, run our function and then assert on the finished

state. If you left Jest running in watch mode, notice how it picks up your new test and runs it, and how quick it is to do so! It's a great way to get instant feedback on your tests as you write them.

The tests above also demonstrate the perfect layout for a test, which is:

- set up
- execute the function under test
- assert on the results.

By keeping the tests laid out in this way, you'll find them easier to follow and work with.

Now we're happy testing our state functions, let's move on to React components.

## Testing React Components

It's worth noting that, by default, I would actually encourage you to not write too many tests on your React components. Anything that you want to test very thoroughly, such as business logic, should be pulled out of your components and sit in standalone functions, just like the state functions that we tested earlier. That said, it is useful at times to test some React interactions (making sure a specific function is called with the right arguments when the user clicks a button, for example). We'll start by testing that our React components render the

right data, and then look at testing interactions. Then we'll move on to snapshots, a feature of Jest that makes testing the output of React components much more convenient.

To do this, we'll need to make use of `react-addons-test-utils`, a library that provides functions for testing React. We'll also install Enzyme, a wrapper library written by AirBnB that makes testing React components much easier. We'll use this API throughout our tests. Enzyme is a fantastic library, and the React team even recommend it as the way to test React components.

```
npm install --save-dev react-addons-test-utils
enzyme
```

Let's test that the `Todo` component renders the text of its todo inside a paragraph. First we'll create `__tests__/todo.test.js`, and import our component:

```
import Todo from '../app/todo';
import React from 'react';
import { mount } from 'enzyme';

test('Todo component renders the text of the
todo', () => {
});
```

I also import `mount` from Enzyme. The `mount` function is used to render our component and then allow us to inspect the output and make assertions on it. Even though we're running our tests in Node, we can still write tests that require a DOM. This is because Jest configures jsdom, a library that

implements the DOM in Node. This is great because we can write DOM based tests without having to fire up a browser each time to test them.

We can use `mount` to create our `Todo`:

```
const todo = { id: 1, done: false, name: 'Buy
Milk' };
const wrapper = mount(
  <Todo todo={todo} />
);
```

And then we can call `wrapper.find`, giving it a CSS selector, to find the paragraph that we're expecting to contain the text of the Todo. This API might remind you of jQuery, and that's by design. It's a very intuitive API for searching rendered output to find the matching elements.

```
const p = wrapper.find('.toggle-todo');
```

And finally, we can assert that the text within it is `Buy Milk`:

```
expect(p.text()).toBe('Buy Milk');
```

Which leaves our entire test looking like so:

```
import Todo from '../app/todo';
import React from 'react';
import { mount } from 'enzyme';

test('TodoComponent renders the text inside it',
() => {
  const todo = { id: 1, done: false, name: 'Buy
Milk' };
```

```
  const wrapper = mount(
    <Todo todo={todo} />
  );
  const p = wrapper.find('.toggle-todo');
  expect(p.text()).toBe('Buy Milk');
});
```

Phew! You might think that was a lot of work and effort to check that "Buy Milk" gets placed onto the screen, and, well … you'd be correct. Hold your horses for now, though; in the next section we'll look at using Jest's snapshot ability to make this much easier.

In the meantime, let's look at how you can use Jest's spy functionality to assert that functions are called with specific arguments. This is useful in our case, because we have the `Todo` component which is given two functions as properties, which it should call when the user clicks a button or performs an interaction.

In this test we're going to assert that when the todo is clicked, the component will call the `doneChange` prop that it's given.

```
test('Todo calls doneChange when todo is clicked',
() => {
});
```

What we want to do is to have a function that we can keep track of its calls, and the arguments that it's called with. Then we can check that when the user clicks the todo, the `doneChange` function is called and also called with the correct arguments. Thankfully, Jest provides this out of the

box with spies. A **spy** is a function whose implementation you don't care about; you just care about when and how it's called. Think of it as you spying on the function. To create one, we call `jest.fn()`:

```
const doneChange = jest.fn();
```

This gives a function that we can spy on and make sure it's called correctly. Let's start by rendering our `Todo` with the right props:

```
const todo = { id: 1, done: false, name: 'Buy
Milk' };
const doneChange = jest.fn();
const wrapper = mount(
  <Todo todo={todo} doneChange={doneChange} />
);
```

Next, we can find our paragraph again, just like in the previous test:

```
const p =
TestUtils.findRenderedDOMComponentWithClass(render
ed, 'toggle-todo');
```

And then we can call `simulate` on it to simulate a user event, passing `click` as the argument:

```
p.simulate('click');
```

And all that's left to do is assert that our spy function has been called correctly. In this case, we're expecting it to be called

with the ID of the todo, which is `1`. We can use `expect(doneChange).toBeCalledWith(1)` to assert this, and with that we're done with our test!

```
test('TodoComponent calls doneChange when todo is
clicked', () => {
  const todo = { id: 1, done: false, name: 'Buy
Milk' };
  const doneChange = jest.fn();
  const wrapper = mount(
    <Todo todo={todo} doneChange={doneChange} />
  );

  const p = wrapper.find('.toggle-todo');
  p.simulate('click');
  expect(doneChange).toBeCalledWith(1);
});
```

## Better Component Testing with Snapshots

I mentioned above that this might feel like a lot of work to test React components, especially some of the more mundane functionalities (such as rendering the text). Rather than make a large amount of assertions on React components, Jest lets you run snapshot tests. These are not so useful for interactions (in which case I still prefer a test like we just wrote above), but for testing that the output of your component is correct, they're much easier.

When you run a snapshot test, Jest renders the React component under test and stores the result in a JSON file. Every time the test runs, Jest will check that the React

component still renders the same output as the snapshot. Then, when you change a component's behavior, Jest will tell you and either:

- you'll realize you made a mistake, and you can fix the component so it matches the snapshot again
- or, you made that change on purpose, and you can tell Jest to update the snapshot.

This way of testing means that:

- you don't have to write a lot of assertions to ensure your React components are behaving as expected
- you can never accidentally change a component's behavior, because Jest will realize.

You also don't have to snapshot all your components. In fact, I'd actively recommend against it. You should pick components with some functionality that you really need to ensure is working. Snapshotting all your components will just lead to slow tests that aren't useful. Remember, React is a very thoroughly tested framework, so we can be confident that it will behave as expected. Make sure you don't end up testing the framework, rather than your code!

To get started with snapshot testing, we need one more Node package. react-test-renderer is a package that's able to take a React component and render it as a pure JavaScript object. This means it can then be saved to a file, and this is what Jest uses to keep track of our snapshots.

```
npm install --save-dev react-test-renderer
```

Now, let's rewrite our first Todo component test to use a snapshot. For now, comment out the `TodoComponent calls doneChange when todo is clicked` test as well.

The first thing you need to do is import the `react-test-renderer`, and also remove the import for `mount`. They can't both be used; you either have to use one or the other. This is why we have commented the other test out for now.

```
import renderer from 'react-test-renderer';
```

Now I'll use the renderer we just imported to render the component, and assert that it matches the snapshot:

```
describe('Todo component renders the todo
correctly', () => {
  it('renders correctly', () => {
    const todo = { id: 1, done: false, name: 'Buy
Milk' };
    const rendered = renderer.create(
      <Todo todo={todo} />
    );
    expect(rendered.toJSON()).toMatchSnapshot();
  });
});
```

The first time you run this, Jest is clever enough to realize that there's no snapshot for this component, so it creates it. Let's take a look at `__tests__/__snapshots__/todo.test.js.snap`:

```
exports[`Todo component renders the todo correctly
renders correctly 1`] = `
<div
  className="todo  todo-1">
  <p
    className="toggle-todo"
    onClick={[Function]}>
    Buy Milk
  </p>
  <a
    className="delete-todo"
    href="#"
    onClick={[Function]}>
    Delete
  </a>
</div>
`;
```

You can see that Jest has saved the output for us, and now the next time we run this test it will check that the outputs are the same. To demonstrate this, I'll break the component by removing the paragraph that renders the text of the todo, meaning that I've removed this line from the `Todo` component:

```
<p className="toggle-todo" onClick={() =>
this.toggleDone() }>{ todo.name }</p>
```

Let's see what Jest says now:

```
FAIL  __tests__/todo.test.js
 ● Todo component renders the todo correctly ›
renders correctly

   expect(value).toMatchSnapshot()

   Received value does not match stored snapshot
1.
```

```
    - Snapshot
    + Received

      <div
        className="todo  todo-1">
    -   <p
    -     className="toggle-todo"
    -     onClick={[Function]}>
    -     Buy Milk
    -   </p>
        <a
          className="delete-todo"
          href="#"
          onClick={[Function]}>
          Delete
        </a>
      </div>

      at Object.<anonymous>
(__tests__/todo.test.js:21:31)
      at process._tickCallback
(internal/process/next_tick.js:103:7)
```

Jest realized that the snapshot doesn't match the new component, and lets us know in the output. If we think this change is correct, we can run jest with the `-u` flag, which will update the snapshot. In this case, though, I'll undo my change and Jest is happy once more.

Next we can look at how we might use snapshot testing to test interactions. You can have multiple snapshots per test, so you can test that the output after an interaction is as expected.

We can't actually test our Todo component interactions through Jest snapshots, because they don't control their own state but call the callback props they are given. What I've done here is move the snapshot test into a new file, todo.snapshot.test.js, and leave our toggling test in todo.test.js.

I've found it useful to separate the snapshot tests into a different file; it also means that you don't get conflicts between `react-test-renderer` and `react-addons-test-utils`.

Remember, you'll find all the code that I've written in this tutorial available on GitHub for you to check out and run locally.

## Conclusion

Facebook released Jest a long time ago, but in recent times it's been picked up and worked on excessively. It's fast become a favorite for JavaScript developers and it's only going to get better. If you've tried Jest in the past and not liked it, I can't encourage you enough to try it again, because it's practically a different framework now. It's quick, great at rerunning specs, gives fantastic error messages and tops it all off with its snapshot functionality.

If you have any questions please feel free to raise an issue on GitHub and I'll be happy to help. And please be sure check out Jest on GitHub and star the project; it helps the maintainers.

# Chapter 4: Building Animated Components, or How React Makes D3 Better

## BY SWIZEC TELLER

D3 is great. As the jQuery of the web data visualization world, it can do everything you can think of.

Many of the best data visualizations you've seen online use D3. It's a great library, and with the recent v4 update, it became more robust than ever.

Add React, and you can make D3 even better.

Just like jQuery, D3 is powerful but low level. The bigger your visualization, the harder your code becomes to work with, the more time you spend fixing bugs and pulling your hair out.

React can fix that.

You can read my book React+d3js ES6 for a deep insight, or keep reading for an overview of how to best integrate React

and D3. In a practical example, we'll see how to build declarative, transition-based animations.

A version of this article also exists as a D3 meetup talk on YouTube.

## Is React Worth It?

OK, React is big. It adds *a ton* of code to your payload, and it increases your dependency footprint. It's yet another library that you have to keep updated.

If you want to use it effectively, you'll need a build step. Something to turn JSX code into pure JavaScript.

Setting up Webpack and Babel is easy these days: just run `create-react-app`. It gives you JSX compilation, modern JavaScript features, linting, hot loading, and code minification for production builds. It's great.

Despite the size and tooling complexity, React *is* worth it, *especially* if you're serious about your visualization. If you're building a one-off that you'll never have to maintain, debug, or expand, stick to pure D3. If you're building something real, I encourage you to add React to the mix.

To me, the main benefit is that React **forces** strongly encourages you to componentize your code. The other benefits

are either symptoms of componentization, or made possible by it.

The main benefits of using React with your D3 code are:

- componentization
- easier testing and debugging
- smart DOM redraws
- hot loading

**Componentization** encourages you to build your code as a series of logical units – components. With JSX, you can use them like they were HTML elements: `<Histogram />`, `<Piechart />`, `<MyFancyThingThatIMade />`. We'll dive deeper into that in the next section.

Building your visualization as a series of components makes it **easier to test and debug**. You can focus on logical units one at a time. If a component works here, it will work over there as well. If it passes tests and looks nice, it will pass tests and look nice no matter how often you render it, no matter where you put it, and no matter who calls it.

**React understands the structure of your code**, so it knows how to redraw only the components that have changes. There’s no more hard work in deciding what to re-render and what to leave alone. Just **change and forget**. React can figure it out on its own. And yes, if you look at a profiling tool, you'll see that *only* the parts with changes are re-rendered.

Using `create-react-app` to configure your tooling, React can utilize **hot loading**. Let's say you're building a visualization of 30,000 datapoints. With pure D3, you have to refresh the page for every code change. Load the dataset, parse the dataset, render the dataset, click around to reach the state you're testing … yawn.

With React -> no reload, no waiting. Just immediate changes on the page. When I first saw it in action, it felt like eating ice cream while the crescendo of *1812 Overture* plays in the background. Mind = blown.

## Benefits of Componentization

Components this, components that. Blah blah blah. Why should you care? Your dataviz code already works. You build it, you ship it, you make people happy.

But does the code make *you* happy? With components, it can. Components make your life easier because they make your code:

- declarative
- reusable
- understandable
- organized

It's okay if that sounds like buzzword soup. Let me show you.

For instance, **declarative code** is the kind of code where you say *what* you want, not *how* you want it. Ever written HTML or CSS? You know how to write declarative code! Congratz!

React uses JSX to make your JavaScript look like HTML. But don't worry, it all compiles to pure JavaScript behind the scenes.

Try to guess what this code does:

```
render() {
  // ...
  return (
    <g transform={translate}>
      <Histogram data={this.props.data}
         value={(d) => d.base_salary}
         x={0}
         y={0}
         width={400}
         height={200}
         title="All" />
      <Histogram data={engineerData}
         value={(d) => d.base_salary}
         x={450}
         y={0}
         width={400}
         height={200}
         title="Engineer" />
      <Histogram data={programmerData}
         value={(d) => d.base_salary}
         x={0}
         y={220}
         width={400}
         height={200}
         title="Programmer"/>
      <Histogram data={developerData}
         value={(d) => d.base_salary}
         x={450}
         y={220}
         width={400}
         height={200}
```

```
            title="Developer" />
      </g>
    )
  }
```

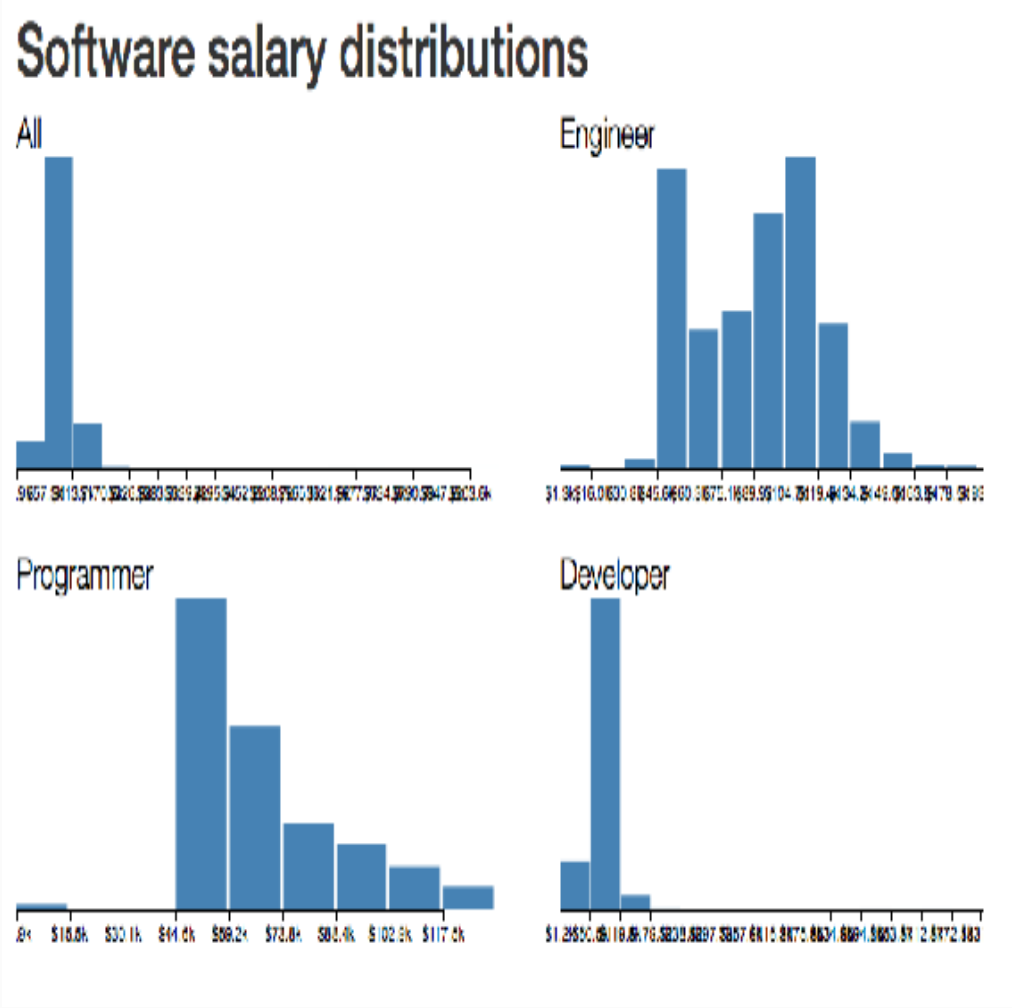
If you guessed *"Renders four histograms"*, you were right. Hooray.

After you create a Histogram component, you can use it like it was a normal piece of HTML. A histogram shows up anywhere you put `<Histogram />` with the right parameters.

In this case, the parameters are `x` and `y` coordinates, `width` and `height` sizing, the `title`, some `data`, and a `value` accessor. They can be anything your component needs.

Parameters look like HTML attributes, but can take any JavaScript object, even functions. It's like HTML on steroids.

With some boilerplate and the right dataset, that code above gives you a picture like this. A comparison of salary distributions for different types of people who write software.




Look at the code again. Notice how **reusable** components are? It's like `<Histogram />` was a function that created a histogram. Behind the scenes it *does* compile into a function call – `(new Histogram()).render()`, or something similar. `Histogram` becomes a class, and you call an instance's render function every time you use `<Histogram />`.

React components should follow the principles of good functional programming. No side effects, statelessness,

idempotency, comparability. Unless you really, really want to break the rules.

Unlike JavaScript functions, where following these principles requires deliberate effort, React makes it hard *not* to code that way. That's a win when you work in a team.

Declarativeness and reusability make your code **understandable** by default. If you've ever used HTML, you can read what that code does. You might not understand the details, but if you know some HTML and JavaScript, you know how to read JSX.

Complex components are made out of simpler components, which are made out of even simpler components, which are eventually made out of pure HTML elements. This keeps your code **organized**.

When you come back in six months, you can look at your code and think, *"Ah yes, four histograms. To tweak this, I should open the Histogram component and poke around."*

React takes the principles I've always loved about fancy-pants functional programming and makes them practical. I love that.

Let me show you an example – an animated alphabet.

## A Practical Example


We're going to build an animated alphabet. Not because it's the simplest example of using React and D3 together, but because it looks cool. When I show this at live talks, people always oooh and aaah, especially when I show proof that only the DOM elements with changes get redrawn.

This is a shortened version of a more in-depth article on React and D3 and transitions that I posted on my blog a few months ago. We're going to gloss over some details in this version to keep it short. You can dive into the full codebase in the GitHub repository.

The code is based on React 15 and D3 4.0.0. Some of the syntax I use, like class properties, is not in stable ES6 yet, but should work if you use `create-react-app` for your tooling setup.

To make an animated alphabet, we need two components:

- `Alphabet`, which creates random lists of letters every 1.5 seconds, then maps through them to render `Letter` components

- `Letter`, which renders an SVG text element, and takes care of its own enter/update/exit transitions.

We're going to use React to render SVG elements, and we'll use D3 for transitions, intervals, and some maths.

## THE ALPHABET COMPONENT

The `Alphabet` component holds the current list of letters in state and renders a collection of `Letter` components in a loop.

We start with a skeleton like this:

```
// src/components/Alphabet/index.jsx
import React, { Component } from 'react';
import ReactTransitionGroup from 'react-addons-
transition-group';
import * as d3 from 'd3';

require('./style.css');

import Letter from './Letter';

class Alphabet extends Component {
  static letters =
"abcdefghijklmnopqrstuvwxyz".split('');
  state = {alphabet: []}

  componentWillMount() {
    // starts an interval to update alphabet
  }

  render() {
    // spits out svg elements
  }
}

export default Alphabet;
```

We import our dependencies, add some styling, and define the `Alphabet` component. It holds a list of available letters in a static `letters` property and an empty `alphabet` in component state. We'll need a `componentWillMount` and a `render` method as well.

The best place to create a new alphabet every 1.5 seconds is in `componentWillMount`:

```
// src/components/Alphabet/index.jsx
  componentWillMount() {
    d3.interval(() => this.setState({
       alphabet: d3.shuffle(Alphabet.letters)
         .slice(0, Math.floor(Math.random() *
Alphabet.letters.length))
         .sort()
    }), 1500);
  }
```

We use `d3.interval( //.., 1500)` to call a function every 1.5 seconds. On each period, we shuffle the available letters, slice out a random amount, sort them, and update component state with `setState()`.

This ensures our alphabet is both random and in alphabetical order. `setState()` triggers a re-render.

Our declarative magic starts in the `render` method.

```
// src/components/Alphabet/index.jsx
render() {
  let transform = `translate(${this.props.x},
${this.props.y})`;
```

```
    return (
      <g transform={transform}>
        <ReactTransitionGroup component="g">
          {this.state.alphabet.map((d, i) => (
            <Letter d={d} i={i} key={`letter-${d}`}
/>
          ))}
        </ReactTransitionGroup>
      </g>
    );
  }
```

We use an SVG transformation to move our alphabet into the specified `(x, y)` position, then define a `ReactTransitionGroup` and map through `this.state.alphabet` to render a bunch of `Letter` components with wanton disregard.

Each `Letter` gets its current text, `d`, and index, `i`. The `key` attribute helps React recognize which component is which. Using `ReactTransitionGroup` gives us special component lifecycle methods that help with smooth transitions.

## ReactTransitionGroup

In addition to the normal lifecycle hooks that tell us when a component mounts, updates, and unmounts, ReactTransitionGroup gives us access to `componentWillEnter`, `componentWillLeave`, and a few others. Notice something familiar?

`componentWillEnter` is the same as D3's `.enter()`, `componentWillLeave` is the same as D3's `.exit()`, and `componentWillUpdate` is the same as D3's `.update()`.

"The same" is a strong concept; they're analogous. D3's hooks operate on entire selections – groups of components – while React's hooks operate on each component individually. In D3, an overlord is dictating what happens; in React, each component knows what to do.

That makes React code easier to understand. I think. ¯_(□)_/¯

`ReactTransitionGroup` gives us even more hooks, but these three are all we need. It's nice that in both `componentWillEnter` and `componentWillLeave`, we can use a callback to explicitly say *"The transition is done. React, back to you"*.

My thanks to Michelle Tilley for writing about `ReactTransitionGroup` on Stack Overflow.

## THE LETTER COMPONENT

Now we're ready for the cool stuff – a component that can transition itself into and out of a visualization declaratively.

The basic skeleton for our `Letter` component looks like this:

```
// src/components/Alphabet/Letter.jsx
```

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';
import * as d3 from 'd3';

class Letter extends Component {
    state = {
      y: -60,
      x: 0,
      className: 'enter',
      fillOpacity: 1e-6
    }
    transition = d3.transition()
                   .duration(750)
                   .ease(d3.easeCubicInOut);

    componentWillEnter(callback) {
      // start enter transition, then callback()
    }

    componentWillLeave(callback) {
      // start exit transition, then callback()
    }

    componentWillReceiveProps(nextProps) {
      if (this.props.i != nextProps.i) {
        // start update transition
      }
    }

    render() {
       // spit out a <text> element
    }
};

export default Letter;
```

We start with some dependencies and define a `Letter` component with a default state and a default transition. In most cases, you'd want to avoid using `state` for coordinates and other transient properties. That's what props are for. With transitions we use state because it helps us keep React's reality in sync with D3's reality.

That said, those magic default values could be default props. That would make our `Alphabet` more flexible.

### componentWillEnter

We put the enter transition in `componentWillEnter`.

```
// src/components/Alphabet/Letter.jsx
    componentWillEnter(callback) {
      let node =
d3.select(ReactDOM.findDOMNode(this));

      this.setState({x: this.props.i*32});

      node.transition(this.transition)
        .attr('y', 0)
        .style('fill-opacity', 1)
        .on('end', () => {
            this.setState({y: 0, fillOpacity: 1});
            callback()
        });
    }
```

We use `reactDOM.findDOMNode()` to get our DOM node and use `d3.select()` to turn it into a d3 selection. Now anything D3 can do, our component can do. Yessss!/p>

Then we update `this.state.x` using the current index and letter width. The width is a value that we Just Know™. Putting `x` in state helps us avoid jumpiness: The `i` prop changes on each update, but we want to delay when the `Letter` moves.

When a `Letter` first renders, it's invisible and 60 pixels above the baseline. To animate it moving down and becoming visible, we use a D3 transition.

We use `node.transition(this.transition)` to start a new transition with default settings from earlier. Any `.attr` and `.style` changes that we make happen over time directly on the DOM element itself.

This confuses React, because it assumes it's the lord and master of the DOM. So we have to sync React's reality with actual reality using a callback: `.on('end', …)`. We use `setState()` to update component state, and trigger the main `callback`. React now knows this letter is done appearing.

### componentWillLeave

The exit transition goes in `componentWillLeave()`. Same concept as above, just in reverse.

```
// src/components/Alphabet/
  componentWillLeave(callback) {
    let node =
d3.select(ReactDOM.findDOMNode(this));

    this.setState({className: 'exit'});

    node.transition(this.transition)
      .attr('y', 60)
      .style('fill-opacity', 1e-6)
      .on('end', () => {
          callback()
      });
  }
```

This time, we update state to change the `className` instead of `x`. That's because `x` doesn't change.

The exit transition itself is an inverse of the enter transition: letter moves down and becomes invisible. After the transition, we tell React it's okay to remove the component.

### componentWillReceiveProps

The update transition goes into `componentWillReceiveProps()`.

```
// src/components/Alphabet/Letter.jsx
  componentWillReceiveProps(nextProps) {
    if (this.props.i != nextProps.i) {
      let node =
d3.select(ReactDOM.findDOMNode(this));

      this.setState({className: 'update'});

      node.transition(this.transition)
        .attr('x', nextProps.i*32)
        .on('end', () => this.setState({x:
nextProps.i*32}));
    }
  }
```

You know the pattern by now, don't you? Update state, do transition, sync state with reality after transition.

In this case, we change the `className`, then move the letter into its new horizontal position.

### render

After all that transition magic, you might be thinking *"Holy cow, how do I render this!?"*. I don't blame ya!

But we did all the hard work. Rendering is straightforward:

```
// src/components/Alphabet/Letter.jsx
  render() {
    return (
      <text dy=".35em"
          y={this.state.y}
          x={this.state.x}
          className={this.state.className}
          style={{fillOpacity:
this.state.fillOpacity}}>
        {this.props.d}
      </text>
    );
  }
```

We return an SVG `<text>` element rendered at an `(x, y)` position with a `className` and a `fillOpacity`. It shows a single letter given by the `d` prop.

As mentioned: using state for `x`, `y`, `className`, and `fillOpacity` is wrong in theory. You'd normally use props for that. But state is the simplest way I can think of to communicate between the render and lifecycle methods.

## You Know the Basics!

Boom. That's it. You know how to build an animated declarative visualization. That's pretty cool if you ask me.

Here is what it looks like in action.

Such nice transitions, and all you had to do was loop through an array and render some `<Letter>` components. How cool is that?

# In Conclusion

You now understand React well enough to make technical decisions. You can look at project and decide: *"Yes, this is more than a throwaway toy. Components and debuggability will help me."*

For extra fun, you also know how to use React and D3 together to build declarative animations. A feat most difficult in the olden days.

To learn more about properly integrating React and D3 check out my book, React+d3js ES6.

# Chapter 5: Using Preact as a React Alternative

## BY AHMED BOUCHEFRA

Preact is an implementation of the virtual DOM component paradigm just like React and many other similar libraries. Unlike React, it's only 3KB in size, and it also outperforms it in terms of speed. It's created by Jason Miller and available under the well-known permissive and open-source MIT license.

## Why Use Preact?

Preact is a lightweight version of React. You may prefer to use Preact as a lightweight alternative if you like building views with React but performance, speed and size are a priority for you – for example, in the case of mobile web apps or progressive web apps.

Whether you're starting a new project or developing an existing one, Preact can save you a lot of time. You don't need to reinvent the wheel trying to learn a new library, since it's similar to, and compatible with, React – to the point that you

can use existing React packages with it with only some aliasing, thanks to the compatibility layer `preact-compat`.

## Pros and Cons

There are many differences between React and Preact that we can summarize in three points:

- **Features and API**: Preact includes only a subset of the React API, and not all available features in React.
- **Size**: Preact is much smaller than React.
- **Performance**: Preact is faster than React.

Every library out there has its own set of pros and cons, and only your priorities can help you decide which library is a good fit for your next project. In this section, I'll try to list the pros and cons of the two libraries.

## PREACT PROS

- Preact is lightweight, smaller (only 3KB in size when gzipped) and faster than React (see these tests). You can also run performance tests in your browser via this link.
- Preact is largely compatible with React, and has the same ES6 API as React, which makes it dead easy either to adopt Preact as a new library for building user interfaces in your project or to swap React with Preact for an existing project for performance reasons.
- It has good documentation and examples available from the official website.
- It has a powerful and official CLI for quickly creating new Preact projects, without the hassle of Webpack and Babel configuration.

- Many features are inspired by all the work already done on React.
- It has also its own set of advanced features independent from React, like Linked State.

# REACT PROS

- React supports one-way data binding.
- It's backed by a large company, Facebook.
- Good documentation, examples, and tutorials on the official website and the web.
- Large community.
- Used on Facebook's website, which has millions of visitors worldwide.
- Has its own official developer debugging tools extension for Chrome.
- It has the Create React App project boilerplate for quickly creating projects with zero configuration.
- It has a well-architectured and complex codebase.

# REACT CONS

- React has a relatively large size in comparison with Preact or other existing similar libraries. (React minified source file is around 136KB in size, or about 42KB when minified and gzipped.)
- It's slower than Preact.
- As a result of its complex codebase, it's harder for novice developers to contribute.

### License Concerns

Another con I listed while writing this article was that React had a grant patent clause paired with the BSD license, making it legally unsuitable for some use

cases. However, in September 2017, the React license switched to MIT, which resolved these license concerns.

## PREACT CONS

- Preact supports only stateless functional components and ES6 class-based component definition, so there's no **createClass**.
- No support for context.
- No support for React propTypes.
- Smaller community than React.

# Getting Started with Preact CLI

Preact CLI is a command line tool created by Preact's author, Jason Miller. It makes it very easy to create a new Preact project without getting bogged down with configuration complexities, so let's start by installing it.

Open your terminal (Linux or macOS) or command prompt (Windows), then run the following commands:

```
npm i -g preact-cli@latest
```

This will install the latest version of Preact CLI, assuming you have Node and NPM installed on your local development machine.

You can now create your project with this:

```
preact create my-app
```

Or with this, ff you want to create your app interactively:

```
preact init
```

Next, navigate inside your app's root folder and run this:

```
npm start
```

This will start a live-reload development server.

Finally, when you finish developing your app, you can build a production release using this:

```
npm run build
```

## Demystifying Your First Preact App

After successfully installing the Preact CLI and generating an app, let's try to understand the simple app generated with the Preact CLI.

The Preact CLI generates the following directory structure

```
├── node_modules
├── package.json
├── package-lock.json
```

```
└── src
    ├── assets
    ├── components
    │   ├── app.js
    │   └── header
    ├── index.js
    ├── lib
    ├── manifest.json
    ├── routes
    │   ├── home
    │   └── profile
    └── style
        └── index.css
```

The `components` folder holds Preact components, and the `routes` folder holds the page components used for each app's route. You can use the `lib` folder for any external libraries, the `style` folder for CSS styles, and the `assets` for icons and other graphics.

Note the `manifest.json` file, which is like `package.json` but for PWAs (progressive web apps). Thanks to the Preact CLI, you can have a perfect-score PWA out of the box.

Now, if you open your project's `package.json` file, you'll see that the main entry point is set to `src/index.js`. Here is the content of this file:

```
import './style';
import App from './components/app';

export default App;
```

As you can see, `index.js` imports styles, and `App` component from `./components/app**`, and then just exports it as the default.

Now, let's see what's inside `./components/app`:

```
import { h, Component } from 'preact';
import { Router } from 'preact-router';

import Header from './header';
import Home from '../routes/home';
import Profile from '../routes/profile';

export default class App extends Component {
    handleRoute = e => {
        this.currentUrl = e.url;
    };

    render() {
        return (
            <div id="app">
                <Header />
                <Router onChange=
{this.handleRoute}>
                    <Home path="/" />
                    <Profile path="/profile/"
user="me" />
                    <Profile path="/profile/:user"
/>
                </Router>
            </div>
        );
    }
}
```

This file exports a default class `App` which extends the `Component` class imported from the `preact` package. Every Preact component needs to extend the `Component` class.

`App` defines a `render` method, which returns a bunch of HTML elements and Preact components that render the app's main user interface.

Inside the `div` element, we have two Preact components, `Header` – which renders the app's header – and a `Router` component.

The Preact Router is similar to the latest version of React Router (version 4). You simply need to wrap the child components with a `<Router>` component, then specify the `path` prop for each component. Then, the router will take care of rendering the component, which has a path prop that matches the current browser's URL.

It's worth mentioning that Preact Router is very simple and, unlike React Router, it doesn't support advanced features such as nested routes and view composition. If you need these features, you have to use either the React Router v3 by aliasing `preact-compat`, or better yet use the latest React Router (version 4) which is more powerful than v3 and doesn't need any compatibility layer, because it works directly with Preact. (See this CodePen demo for an example.)

## Preact Compatibility Layer

The `preact-compat` module allows developers to switch from React to Preact without changing imports from React and

ReactDOM to Preact, or to use existing React packages with Preact.

Using `preact-compat` is easy. All you have to do is to first install it via npm:

```
npm i -S preact preact-compat
```

Then set up your build system to redirect imports or requires for `react` or `react-dom` to `preact-compat`. For example, in the case of Webpack, you just need to add the following configuration to `webpack.config.js`:

```
{
  "resolve": {
    "alias": {
      "react": "preact-compat",
      "react-dom": "preact-compat"
    }
  }
}
```

## Conclusion

Preact is a nice alternative to React. Its community is growing steadily, and more web apps are using it. So if you're building a web app with high-performance requirements, or a mobile app for slow 2G networks, then you should consider Preact – either as the first candidate view library for your project, or as a drop-in replacement for React.