# Routing

## Building the components of react-router

### The completed app

All the example code for this chapter is inside the folder routing in the code download. We’ll start off with the basics app:

$ cd routing/basics

Taking a look inside this directory, we see that this app is powered by create-react-app:

$ dir

README.md nightwatch.json package.json public/

src/

tests/

Our React app lives inside src/:

$ dir src

App.css

App.js SelectableApp.js complete/

index-complete.js index.css index.js

logo.svg

complete/ contains the completed version of App.js. The folder also contains each iteration of App.js that we build up throughout this section.

Install the npm packages:

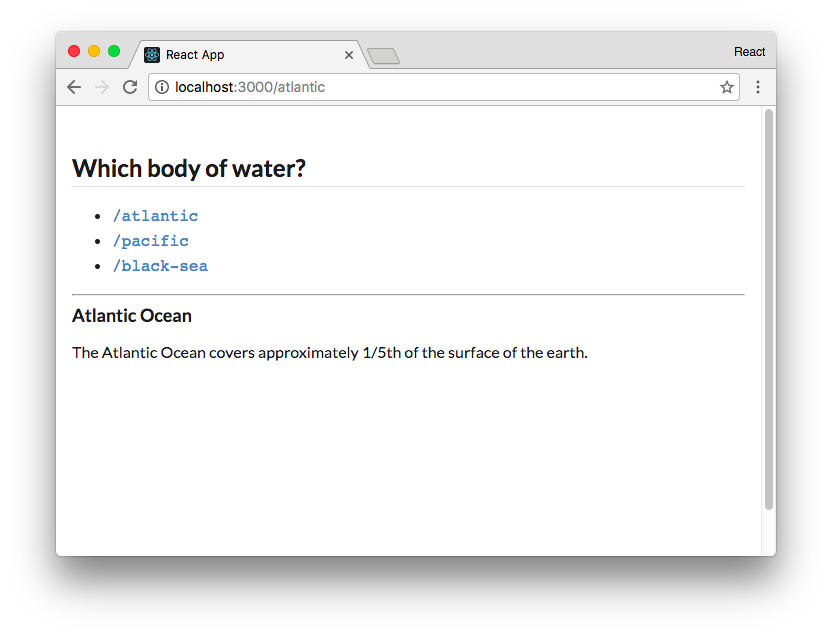
$ npm i

At the moment, index.js is loading index-complete.js. index-complete.js uses SelectableApp to give us the ability to toggle between the app’s various iterations. SelectableApp is just for demo purposes.

If we boot the app, we’ll see the completed version:

$ npm start

The app consists of three links. Clicking on a link displays a blurb about the selected body of water below the application:



The completed app

Notice that clicking on a link changes the location of the app.

Clicking on the link /atlantic updates the URL to /atlantic. Importantly, the browser does not make a request when we click on a link. The blurb about the Atlantic Ocean appears and the browser’s URL bar updates to /atlantic instantly.

Clicking on the link /black-sea displays a countdown. When the countdown finishes, the app redirects the browser to /.

The routing in this app is powered by the react-router library. We’ll build a version of the app ourselves by constructing our own React Router components. We’ll be working inside the file App.js throughout this section.

### Building Route

We’ll start off by building React Router’s Route component. We’ll see what it does shortly. Let’s open the file src/App.js. Inside is a skeletal version of App. Below the import statement for React, we define a simple App component with two <a> tag links:

**routing/basics/src/App.js**

**class** App **extends** React.Component {

render() { **return** (

<div className='ui text container'

>

<h2 className='ui dividing header'> Which body **of** water?

</h2>

<ul>

<li>

<a href='/atlantic'>

<code>/atlantic</code>

</a>

</li>

<li>

<a href='/pacific'>

<code>/pacific</code>

</a>

</li>

</ul>

<hr />

{*/\* We'll insert the Route components here \*/*} </div>

);

}

}

We have two regular HTML anchor tags pointing to the paths /atlantic and /pacific.

Below App are two stateless functional components:

**routing/basics/src/App.js**

**const** Atlantic = () => (

<div>

<h3>Atlantic Ocean</h3>

<p>

The Atlantic Ocean covers approximately 1/5th **of** the surface **of** the earth.

</p>

</div>

);

**const** Pacific = () => (

<div>

<h3>Pacific Ocean</h3>

<p>

Ferdinand Magellan, a Portuguese explorer, named the ocean 'mar pacifico' **in** 1521, which means peaceful sea. </p>

</div>

);

These components render some facts about the two oceans. Eventually, we want to render these components inside App. We want to have App render Atlantic when the browser’s location is /atlantic and Pacific when the location is /pacific.

Recall that index.js is currently deferring to index-complete.js to load the completed version of the app to the DOM. Before we can take a look at the app so far, we need to ensure index.js mounts the App component we’re working on here in ./App.js instead.

Open up index.js. First, comment out the line that imports index-complete:

// [STEP 1] Comment out this line: **// import "./index-complete";**

As in other create-react-app apps, the mounting of the React app to the DOM will take place here in index.js. Let’s un-comment the line that mounts App:

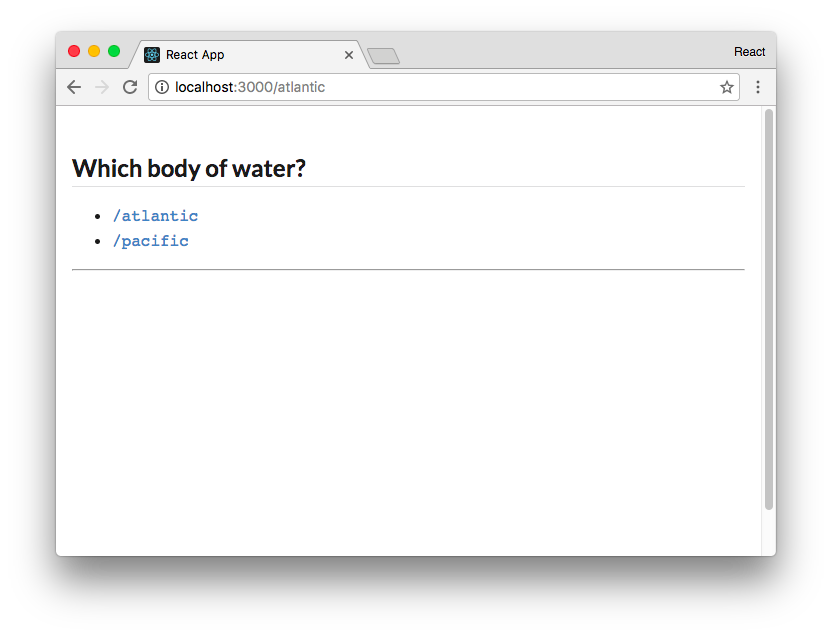
// [STEP 2] Un-comment this line:

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

From the root of the project’s folder, we can boot the app with the start command:

$ npm start

We see the two links rendered on the page. We can click on them and note the browser makes a page request. The URL bar is updated but nothing in the app changes:



We see neither Atlantic nor Pacific rendered, which makes sense because we haven’t yet included them in App. Despite this, it’s interesting that at the moment our app doesn’t care about the state of the pathname. No matter what path the browser requests from our server, the server will return the same index.html with the same exact JavaScript bundle.

This is a desirable foundation. We want our browser to load React in the same way in each location and defer to React on what to do at each location.

Let’s have our app render the appropriate component, Atlantic or Pacific, based on the location of the app (/atlantic or /pacific). To implement this behavior, we’ll write and use a Route component.

In React Router, Route is a component that determines whether or not to render a specified component based on the app’s location. We’ll need to supply Route with two arguments as props:

* The path to match against the location
* The component to render when the location matches path

Let’s look at how we might use this component before we write it. In the render() function of our App component, we’ll use Route like so:

**routing/basics/src/complete/App-1.js**

<ul>

<li>

<a href='/atlantic'>

<code>/atlantic</code>

</a>

</li>

<li>

<a href='/pacific'>

<code>/pacific</code>

</a>

</li>

</ul>

<hr />

**<Route path='/atlantic' component={Atlantic} />**

**<Route path='/pacific' component={Pacific} />** </div>

);

**Route**, like everything else in React Router, is a component. The supplied path prop is matched against the browser’s location. If it matches, Route will return the component. If not, Route will return null, rendering nothing.

At the top of the file above App, let’s write the Route component as a stateless function. We’ll take a look at the code then break it down:

**routing/basics/src/complete/App-1.js**

import React from 'react';

**const Route = ({ path, component }) => {**

**const pathname = window.location.pathname; if (pathname.match(path)) {**

**return (**

**React.createElement(component)**

**);**

**} else {**

**return null;**

**}**

**};**

class App extends React.Component {

We use the ES6 destructuring syntax to extract our two props, path and component, from the arguments:

**routing/basics/src/complete/App-1.js**

**const** Route = ({ path, component }) => {

Next, we instantiate the pathname variable:

**routing/basics/src/complete/App-1.js**

**const** pathname = window.location.pathname;

Inside a browser environment, window.location is a special object containing the properties of the browser’s current location. We grab the pathname from this object which is the path of the URL.

Last, if the path supplied to Route matches the pathname, we return the component. Otherwise, we return null:

**routing/basics/src/complete/App-1.js**

**if** (pathname.match(path)) {

**return** (

React.createElement(component)

);

} **else** {

**return null**;

}

While the Route that ships with React Router is more complex, this is the component’s heart. The component matches path against the app’s location to determine whether or not the specified component should be rendered.

Let’s take a look at the app at this stage.

 You can also render components passed as props like this:

**const** Route = ({ pattern, component: Component }) => {

**const** pathname = window.location.pathname; **if** (pathname.match(pattern)) {

**return** (

<Component />

);

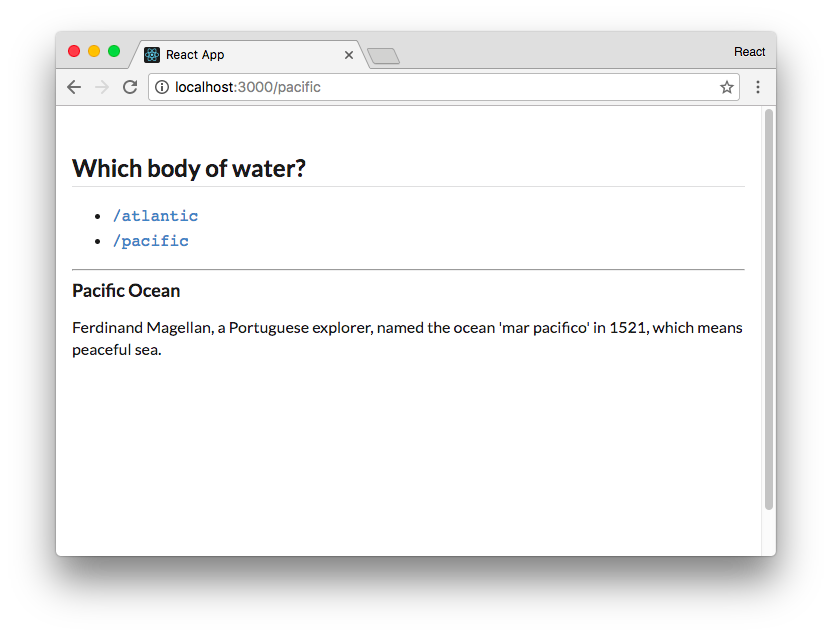
It’s imperative when you do this that the component name is capitalized, which is why we extract the component as Component in the arguments. But when a component class is a dynamic variable as it is here, oftentimes React developers prefer to just use React.createElement() as opposed to JSX.

###### **Try it out**

Save App.js. Ensure the Webpack development server is running if it isn’t already:

$ npm start

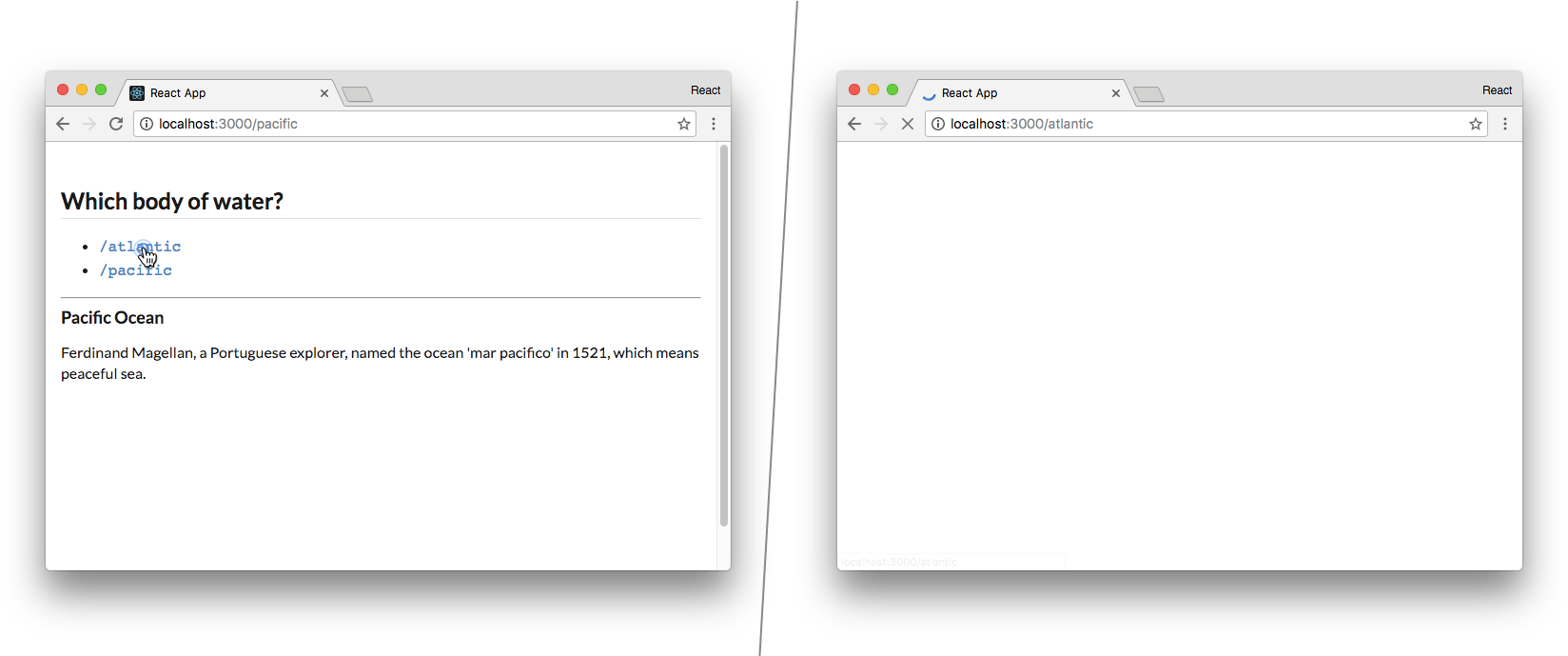
Head to the browser and visit the app. Notice that we’re now rendering the appropriate component when we visit each location:



**/pacific** now renders **Pacific**

Our app is responding to some external state, the location of the browser. Each Route determines whether its component should be displayed based on the app’s location. Note that when the browser visits /, neither component matches. The space both Route components occupy is left empty.

When we click on a link, we see that the browser is doing a full page load:



Clicking on **/atlantic** triggers a full page load

By default, our browser makes a fresh request to the Webpack development server every time we click a link. The server returns the index.html and our browser needs to perform the work of mounting the React app again.

As highlighted in the intro, this cycle is unnecessary. When switching between /pacific and /atlantic, there’s no need to involve the server. Our client app already has all the components loaded and ready to go. We just need to swap in the Atlantic component for the Pacific one when clicking on the /atlantic link.

What we’d like clicking the links to do is change the location of the browser without making a web request. With the location updated, we can re-render our React app and rely on Route to appropriately determine which components to render.

To do so, we’ll build our own version of another component that ships with React Router.

### **Building Link**

In web interfaces, we use HTML <a> tags to create links. What we want here is a special type of <a> tag. When the user clicks on this tag, we’ll want the browser to skip its default routine of making a web request to fetch the next page. Instead, we just want to manually update the browser’s location.

Most browsers supply an API for managing the history of the current session, window.history. We encourage trying it out in a JavaScript console inside the browser. It has methods like history.back() and history.forward() that allow you to navigate the history stack. Of immediate interest, it has a method history.pushState() which allows you to navigate the browser to a desired location.

The history API received some updates with HTML5. To maximize compatibility across browsers, react-router interfaces with this API using a library called History.js. This history package is already included in this project’s package.json:

**routing/basics/package.json**

"history": "4.3.0",

Let’s update our App.js file and import the createBrowserHistory function from the history library. We’ll use this function to create an object, called history, which we’ll use to interact with the browser’s history API:

**routing/basics/src/complete/App-2.js**

import React from 'react';

**import createHistory from 'history/createBrowserHistory'; const history = createHistory();**

const Route = ({ path, component }) => {

Let’s compose a Link component that produces an <a> tag with a special onClick binding. When the user clicks on the Link component, we’ll want to prevent the browser from making a request. Instead, we’ll use the history API to update the browser’s location.

Just like we did with the Route component, let’s see how we’ll use this component before we implement it.

Inside of the render() function of our App component, let’s replace the <a> tags with our upcoming Link component. Rather than using the href attribute, we’ll specify the desired location of the link with the to prop:

**routing/basics/src/complete/App-2.js**

<ul>

<li>

**<Link to='/atlantic'>**

<code>/atlantic</code>

**</Link>**

</li>

<li>

**<Link to='/pacific'>**

<code>/pacific</code>

**</Link>**

</li>

</ul>

Our Link component will be a stateless function that renders an <a> tag with an onClick handler attribute. Let’s see the component in its entirety and then walk through it:

**routing/basics/src/complete/App-2.js**

**const Link = ({ to, children }) => (**

**<a**

**onClick={(e) => {**

**e.preventDefault(); history.push(to);**

**}} href={to}**

**>**

**{children}**

**</a> );**

class App extends React.Component {

Stepping through this:

**onClick**

The onClick handler for the <a> tag first calls preventDefault() on the event object. Recall that the first argument passed to an onClick handler is always the event object. Calling preventDefault() prevents the browser from making a web request for the new location.

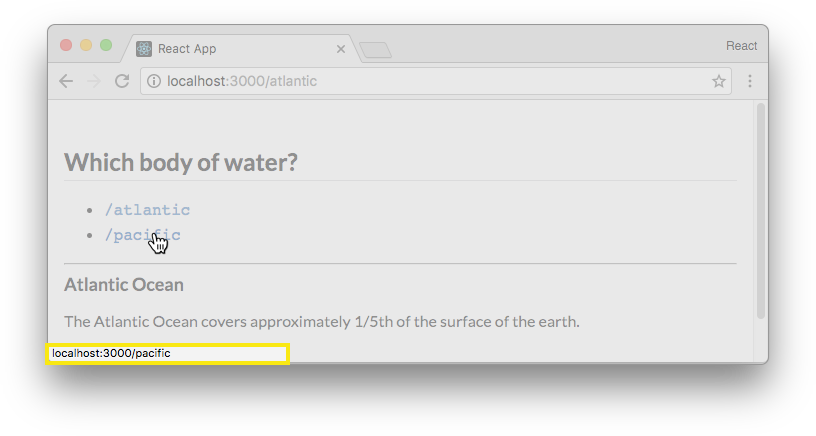
Using the history.push() API, we’re “pushing” the new location onto the browser’s history stack.

Doing so will update the location of the app. This will be reflected in the URL bar.

**href**

We set the href attribute on the <a> tag to the value of the to prop.

When a user clicks a traditional <a> tag, the browser uses href to determine the next location to visit. As we’re changing the location manually in our onClick handler, the href isn’t strictly necessary. However, we should always set it anyway. It enables a user to hover over our links and see where they lead or open up links in new tabs:



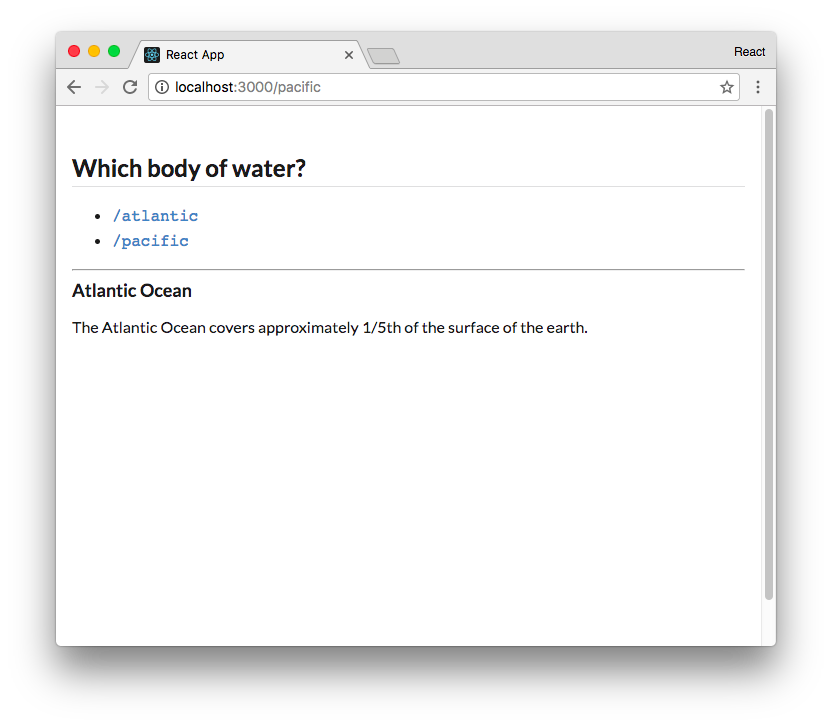
Hovering over a link

**children**

Inside of the <a> tag we render the prop children. As covered in the chapter “Advanced Component Configuration,” the children prop is a special prop. It is a reference to all React elements contained inside of our Link component. This is the text or HTML that we’re turning into a link. In our case, this will be either <code>/atlantic</code> or <code>/pacific</code>.

With our app using our Link component instead of vanilla <a> tags, we’re modifying the location of the browser without performing a web request whenever the user clicks on one of our links.

If we save and run the app now, we’ll see that the functionality isn’t quite working as we expect. We can click on the links and the URL bar will update with the new location without refreshing the page, yet our app does not respond to the change:



After clicking the link **/pacific**, URL bar says **/pacific** but we still see **Atlantic**

While **Link** is updating the location of the browser, our React app is not alerted of the change. We’ll need to trigger our React app to re-render whenever the location changes.

The history object provides a listen() function which we can use here. We can pass listen() a function that it will invoke every time the history stack is modified. We can set up the listen() handler inside of componentDidMount(), subscribing to history with a function that calls forceUpdate():

**routing/basics/src/complete/App-2.js**

class App extends React.Component {

**componentDidMount() {**

**history.listen(() => this.forceUpdate());**

**}**

render() {

When the browser’s location changes, this listening function will be invoked, re-rendering App. Our Route components will then re-render, matching against the latest URL.

###### **Try it out**

Let’s save our updated App.js and visit the app in the browser. Notice that the browser doesn’t perform any full page loads as we navigate between the two routes /pacific and /atlantic!

Even with the tiny size of our app we can enjoy a noticeable performance improvement. Avoiding a full page load saves hundreds of milliseconds and prevents our app from “blinking” during the page change. Given this is a superior user experience now, it’s easy to imagine how these benefits scale as the size and complexity of our app does.

Between the Link and Route components, we’re getting an understanding of how we can use a component-driven routing paradigm to make updates to the browser’s location and have our app respond to this state change.

We still have two more components to cover: Redirect and Switch. These will give us even more control over the routing in our app.

Before building these components, however, we’ll build a basic version of React Router’s Router component. react-router supplies a Router component which is the top-most component in every react-router app. As we’ll see, it’s responsible for triggering re-renders whenever the location changes. It also supplies all of React Router’s other components with APIs they can use to both read and modify the browser’s location.

### **Building Router**

Our basic version of Router should do two things:

1. Supply its children with context for both location and history
2. Re-render the app whenever the history changes

Regarding the first requirement, at the moment our Route and Link components are using two external APIs directly. Route uses window.location to read the location and Link uses history to modify the location. Redirect will need to access the same APIs. The Router supplied by reactrouter makes these APIs available to child components via context. This is a cleaner pattern and means you can easily inject your own location or history object into your app for testing purposes.

Regarding the second requirement, right now App subscribes to history in componentDidMount(). We’ll move this responsibility up to Router, which will be our app’s top-most component.

Let’s use Router inside App before building it. Because we will no longer need to use componentDidMount() in App, we can turn the component into a stateless function.

At the top of App, we’ll convert the component to a function, remove componentDidMount(), and add the opening tag for <Router>:

**routing/basics/src/complete/App-3.js**

**const App = () => (**

**<Router>**

<div

className='ui text container'

>

And close both off at the bottom:

**routing/basics/src/complete/App-3.js**

<Route path='/atlantic' component={Atlantic} />

<Route path='/pacific' component={Pacific} /> </div>

**</Router>**

**);**

We’ll declare Router above App. Let’s see what it looks like in full before walking through the component:

**routing/basics/src/complete/App-3.js**

**class Router extends React.Component {**

**static childContextTypes = { history: PropTypes.object, location: PropTypes.object,**

**};**

**constructor(props) {**

**super(props);**

**this.history = createHistory(); this.history.listen(() => this.forceUpdate());**

**}**

**getChildContext() {**

**return {**

**history: this.history, location: window.location,**

**};**

**}**

**render() {**

**return this.props.children;**

**}**

**}**

Subscribing to **history**

Inside of the constructor for our new Router component, we initialize this.history. We then subscribe to changes, which is the same thing we did inside the App component:

**routing/basics/src/complete/App-3.js**

constructor(props) {

**super**(props);

**this**.history = createHistory();

**this**.history.listen(() => **this**.forceUpdate()); }

Exposing context

As we mentioned earlier, we want Router to expose two properties to its child components. We can use the context feature of React components. Let’s add the two properties we want to pass down, history and location, to the child context.

In order to expose context to children, we must specify the type of each context. We do that by defining childContextTypes.

First, we import the prop-types package at the top of the file:

**routing/basics/src/complete/App-3.js**

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

Then we can define childContextTypes:

**class** Router **extends** React.Component {

**static** childContextTypes = { history: PropTypes.object, location: PropTypes.object,

};

Then, in getChildContext(), we return the context object:

**routing/basics/src/complete/App-3.js**

getChildContext() {

**return** {

history: **this**.history, location: window.location,

};

}

Finally, we render the children wrapped by our new Router component in the render() function:

**routing/basics/src/complete/App-3.js**

render() {

**return this**.props.children;

}

Because we’re initializing history inside of Router, we can remove the declaration that we had at the top of the file:

**routing/basics/src/complete/App-3.js**

import React from 'react';

import createHistory from 'history/createBrowserHistory';

~~const history = createHistory();~~

Since we now have a Router component that is passing the history and location in the context, we can update our Route and Link components to use these variables from our context.

Let’s first tackle the Route component. The second argument passed to a stateless functional component is the context object. Rather than using the location on window.location, we’ll grab location from that context object in the arguments of the component:

**routing/basics/src/complete/App-3.js**

**const Route = ({ path, component }, { location }) => {**

**const pathname = location.pathname;** if (pathname.match(path)) {

return (

React.createElement(component)

);

} else {

return null;

}

};

**Route.contextTypes = { location: PropTypes.object,**

**};**

Below Route, we set the property contextTypes. Remember, to receive context a component must white-list which parts of the context it should receive.

Let’s also update our Link component in a similar manner. Link can use the history property from the context object:

**routing/basics/src/complete/App-3.js**

**const Link = ({ to, children }, { history }) => (**

<a onClick={(e) => {

e.preventDefault(); history.push(to);

}} href={to}

>

{children}

</a>

);

**Link.contextTypes = { history: PropTypes.object,**

**};**

Our app is now wrapped in a Router component. While it lacks lots of the features provided by the actual Router supplied by react-router, it gives us an idea of how the Router component works: It supplies location-management APIs to child components and forces the app to re-render when the location changes.

Let’s save our updated App.js and head to the app in our browser. We see that everything is working exactly as before.

With our Router in place, we can now roll our own Redirect component that uses history from context to manipulate the browser’s location.

### Building Redirect

The Redirect component is a cousin of Link. Whereas Link produces a link that the user can click on to modify the location, Redirect will immediately modify the location whenever it is rendered.

Like Link, we’ll expect this component to be supplied with a to prop. And, like Link, we’ll grab history from context and use that object to modify the browser’s location.

However, where we do this is different.

Above Router, let’s write the Redirect component and see how it works:

**routing/basics/src/complete/App-4.js**

**class Redirect extends React.Component {**

**static contextTypes = {**

**history: PropTypes.object,**

**}**

**componentDidMount() {**

**const history = this.context.history; const to = this.props.to; history.push(to);**

**}**

**render() {**

**return null;**

**}**

**}**

class Router extends React.Component {

We’ve placed the history.push() inside componentDidMount()! The moment this component is mounted to the page, it calls out to the history API to modify the app’s location.

If you’re familiar with routing paradigms from other web development frameworks, the Redirect component might appear particularly curious. Most developers are used to things like an imperative routing table to handle redirects.

Instead, react-router furnishes a declarative paradigm consisting of composable components. Here, a Redirect is represented as nothing more than a React component. Want to redirect? Just render a Redirect component.

Because we’re defining Redirect as a JavaScript class, we can define contextTypes inside the class declaration with static.

In the completed version of the app, we saw a third route, black-sea. When this location was visited, the app displayed a countdown timer before redirecting to /. Let’s build this now.

First, we’ll add a new Link and Route for the component that we’ll soon define, BlackSea:

**routing/basics/src/complete/App-4.js**

<ul>

<li>

<Link to='/atlantic'>

<code>/atlantic</code>

</Link>

</li>

<li>

<Link to='/pacific'>

<code>/pacific</code>

</Link>

</li>

**<li>**

**<Link to='/black-sea'>**

**<code>/black-sea</code>**

**</Link>**

**</li>**

</ul>

<hr />

<Route path='/atlantic' component={Atlantic} />

<Route path='/pacific' component={Pacific} />

**<Route path='/black-sea' component={BlackSea} />** </div>

</Router>

Let’s go ahead and define BlackSea at the bottom of App.js.

First, let’s implement the counting logic. We’ll initialize state.counter to 3. Then, inside componentDidMount(), we’ll perform the countdown using JavaScript’s built-in setInterval() function:

**routing/basics/src/complete/App-4.js**

**class BlackSea extends React.Component {**

**state = {**

**counter: 3,**

**};**

**componentDidMount() {**

**this.interval = setInterval(() => (**

**this.setState(prevState => {**

**return {**

**counter: prevState.counter - 1,**

**};**

**}**

**)), 1000);**

**}**

The setInterval() function will decrease state.counter by one every second.

We have to remember to clear the interval when the component is unmounted. This is the same strategy we used in the timers app in the second chapter:

**routing/basics/src/complete/App-4.js**

**componentWillUnmount() {**

**clearInterval(this.interval);**

**}**

Last, let’s focus on the redirect logic. We’ll handle the redirect logic inside our render() function. When the render() function is called, we’ll want to check if the counter is less than 1. If it is, we want to perform a redirect. We do this by including the Redirect component in our render output:

**routing/basics/src/complete/App-4.js**

**render() { return (**

**<div>**

**<h3>Black Sea</h3>**

**<p>Nothing to sea [sic] here ...</p>**

**<p>Redirecting in {this.state.counter}...</p>**

**{**

**(this.state.counter < 1) ? (**

**<Redirect to='/' />**

**) : null**

**}**

**</div>**

**);**

**}**

**}**

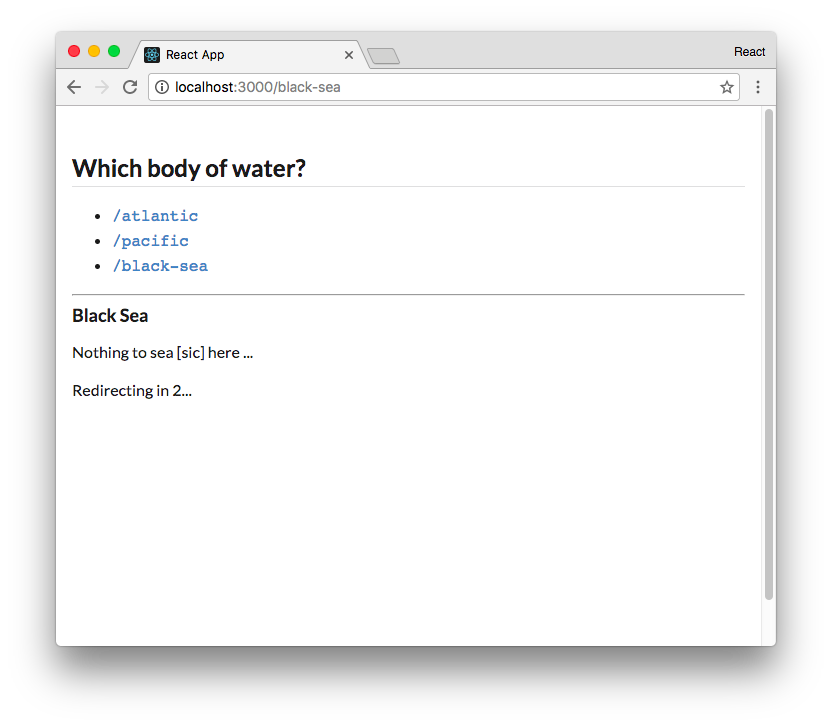
Three seconds after BlackSea mounts, our interval function decrements our state.counter to 0. The setState() function triggers a re-render of the BlackSea component. Its output will include the Redirect component. When the Redirect component mounts, it will trigger the redirect.

This mechanism for triggering a redirect might seem peculiar at first. But this paradigm is powerful. We have complete control of routing by rendering components and passing props. Again, the React Router team prides itself on the fact that the interface to the library is just React. As we’ll explore more in the second half of the chapter, this property gives us lots of flexibility.

Try it out

With Redirect built and in use, let’s try it out. Save App.js.

Visiting /black-sea in the browser, we witness the component rendering before it performs the redirect:



The Black Sea countdown

At this point, we have an understanding of how three of React Router’s fundamental components work to read and update the browser’s location state. We also see how they work with the context of the top-most Router component.

Let’s scrap our hand-rolled React Router components and instead use the library’s routing components. After doing so, we can explore a couple more features of the Route component supplied by react-router. Further, we’ll see how Switch provides one last key piece of functionality.

### Using react-router

We’ll import the components that we want to use from the react-router package and remove the ones we’ve written so far.

The react-router library encompasses a few different npm packages such as react-router-dom and react-router-native. Each corresponds to a supported environment for React. Because we’re building a web app, we’ll use the react-router-dom npm package. react-router-dom is already included in this project’s package.json.

At the top of our App.js file, remove the import statement for createBrowserHistory. React Router will take care of history management:

import React from 'react';

~~import createHistory from 'history/createBrowserHistory';~~

We’ll add an import statement that includes each of the components we want to use. Then, we’ll delete all our custom react-router components. All our other components, like App, can remain unchanged:

**routing/basics/src/complete/App-5.js**

import React from 'react';

**import {**

**BrowserRouter as Router,**

**Route,**

**Link,**

**Redirect,**

**} from 'react-router-dom'** const App = () => (

react-router-dom exports its router under the name BrowserRouter to distinguish it from the routers included in other environments, like NativeRouter. It is common practice to use the alias Router by using the as keyword as we do here.

Save App.js. After this change, we’ll see that everything is still working as it was before we switched to using React Router.

### More Route

Now that we’re using the react-router library, our imported Route component has several additional features.

So far, we’ve used the prop component to instruct Route which component to render when the path matches the current location. Route also accepts the prop render. We can use this prop to define a render function in-line.

To see an example of this, let’s add another Route declaration to App. We’ll insert it above the rest of our existing Route components. This time, we’ll use render:

**routing/basics/src/complete/App-5.js**

**<Route path='/atlantic/ocean' render={() => (**

**<div>**

**<h3>Atlantic Ocean — Again!</h3>**

**<p>**

**Also known as "The Pond."**

**</p>**

**</div>**

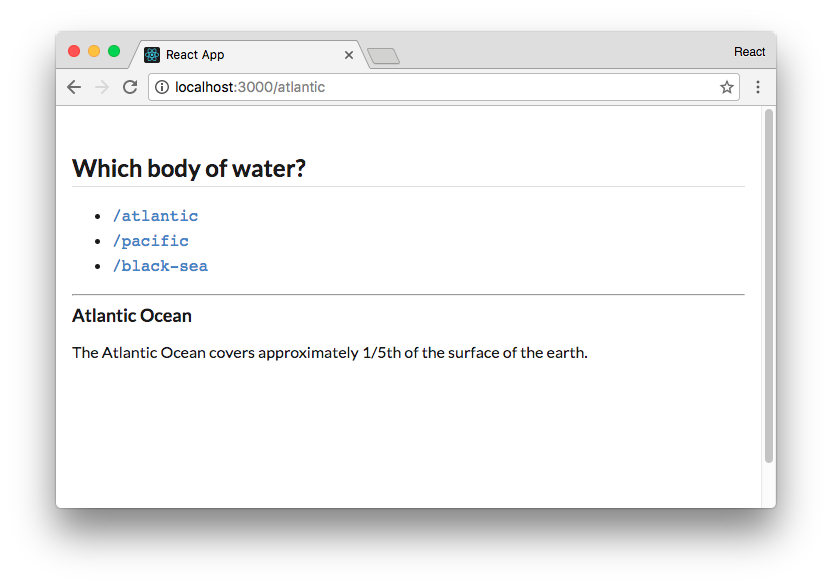
**)} />**

<Route path='/atlantic' component={Atlantic} />

<Route path='/pacific' component={Pacific} />

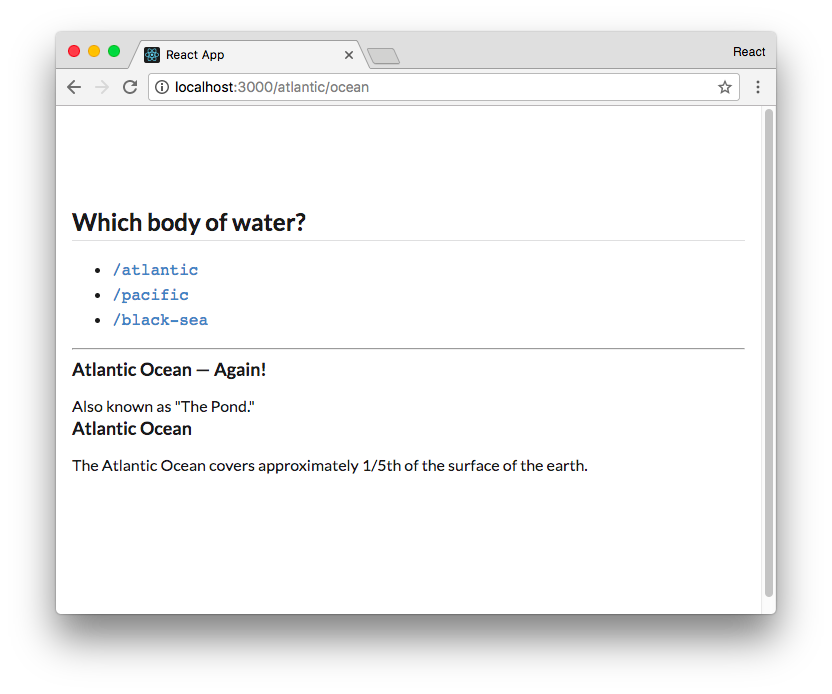
<Route path='/black-sea' component={BlackSea} />

Save App.js. If we visit the app at /atlantic we see just the Atlantic component, as expected:



Just **Atlantic** shows on **/atlantic**

What happens if we visit /atlantic/ocean? We don’t have a Link to this path so type it into the address bar. We notice both the Atlantic component and our new anonymous render function, one stacked atop the other:



Both Atlantic components appear on **/atlantic/ocean**

Why do we see both components? Because of how Route matches path against the location.

Recall that our Route component performed the match like this:

**routing/basics/src/complete/App-1.js**

**if** (pathname.match(path)) {

Consider how that behaves:

**const** routePath = '/atlantic';

**let** browserPath = '/atl'; browserPath.match(routePath); *// -> no match*

browserPath = '/atlantic'; browserPath.match(routePath); *// -> matches*

browserPath = '/atlantic/ocean' browserPath.match(routePath); *// -> matches*

Therefore, both Route declarations for our two Atlantic components match the location /atlantic/ocean. So they both render.

We hadn’t observed this behavior of Route up until now. But given how Route works, this behavior makes sense. Any number of components might match a given location and they will all render. Route does not impose any kind of exclusivity.

Sometimes, this behavior is undesirable. We’ll see a strategy for managing this a bit later.

The example above and our implementation of pathname matching are not strictly accurate. As you might expect, Route matches against the start of the pathname. Therefore,

/atlantic/ocean/pacific will not match for the component Pacific even though that path contains the /pacific substring.

With this behavior in mind, what if we wanted to add a component that renders when the user visits the root (/)?

It would be nice to have some text that instructs the user to click on one of the links. We now know that a solution like this would be problematic:

**routing/basics/src/complete/App-5.js**

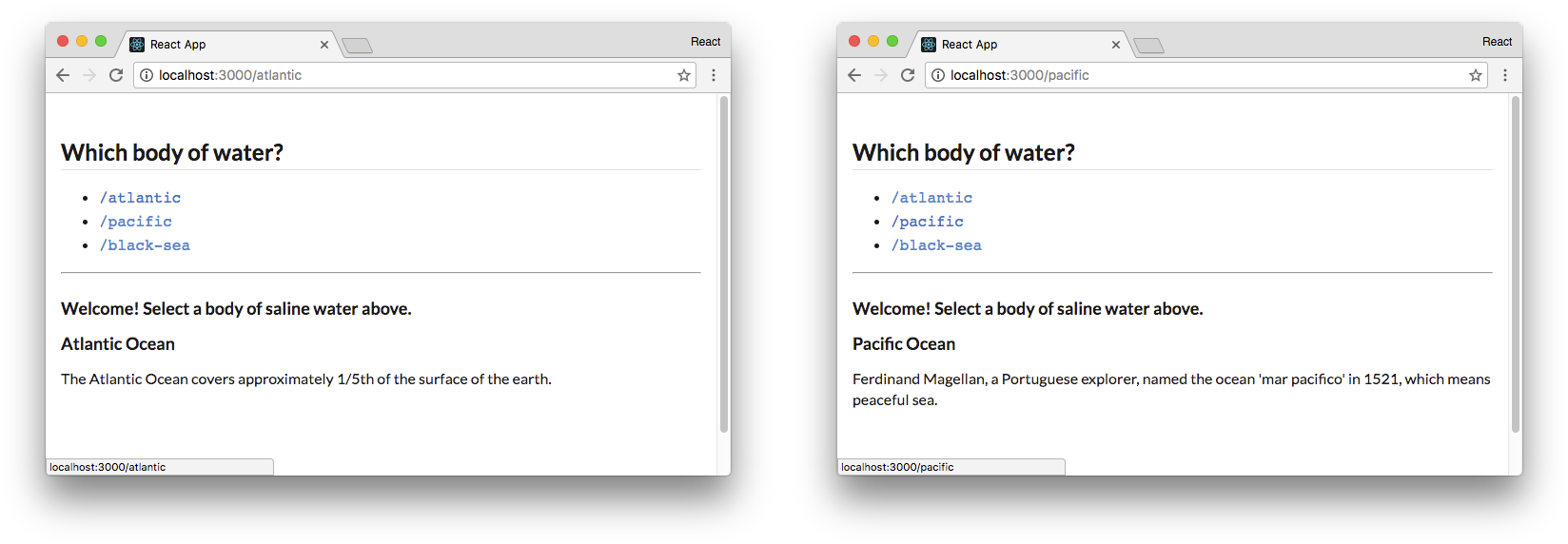
{*/\* This solution is problematic \*/*}

<Route path='/' render={() => ( <h3>

Welcome! Select a body **of** saline water above. </h3>

)} />

Given that / matches paths like /atlantic and /pacific, this component would render on every page of the app:



The **Route** for **/** matches every location

This behavior is not quite what we want. By adding the prop exact to the Route component, we can specify that the path must exactly match the location.

Add the Route for / now:

**routing/basics/src/complete/App-6.js**

<Route path='/atlantic/ocean' render={() => (

<div>

<h3>Atlantic Ocean — Again!</h3>

<p>

Also known as "The Pond."

</p>

</div>

)} />

<Route path='/atlantic' component={Atlantic} />

<Route path='/pacific' component={Pacific} />

<Route path='/black-sea' component={BlackSea} />

**<Route exact path='/' render={() => ( <h3>**

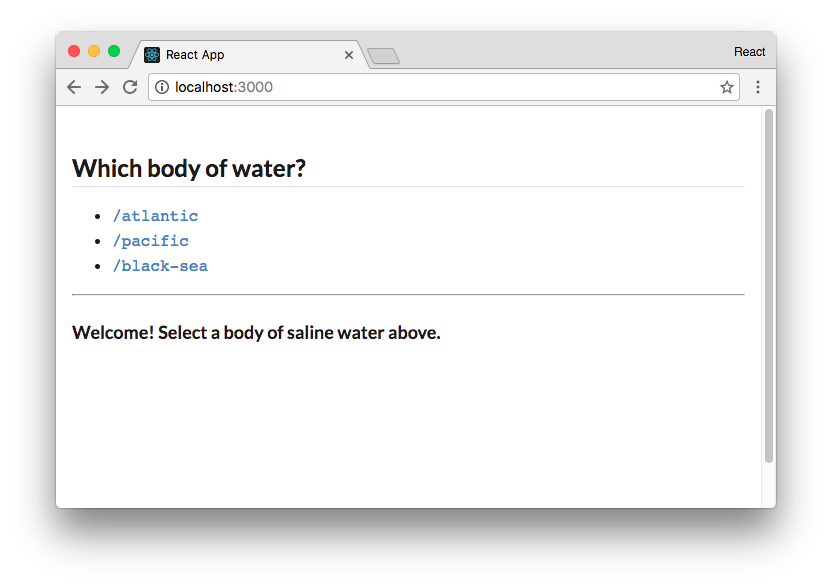
**Welcome! Select a body of saline water above. </h3>**

**)} />**

In JSX, if the prop is listed but not assigned to a value it defaults the value to true.

Try it out

Save App.js. Visiting / in our browser, we see our welcome component. Importantly, the welcome component does not appear on any other path:



We welcome the user at **/**

We now have a proper handler for when the user visits /.

The Route component is a powerful yet simple way to declare which components we’d like to appear on which routes. However, Route alone has some limitations:

1. As we saw earlier with the route /atlantic/ocean, we’ll often want only one Route to match a given path.
2. Furthermore, we don’t yet have a strategy for handling the situation where a user visits a location that our app doesn’t specify a match for.

To work around these, we can wrap our Route components in a **Switch** component.

### **Using Switch**

When Route components are wrapped in a Switch component, only the first matching Route will be displayed.

This means we can use Switch to overcome the two limitations we’ve witnessed so far with Route:

1. When the user visits /atlantic/ocean, the first Route will match and the subsequent Route matching /atlantic will be ignored.
2. We can include a catch-all Route at the bottom of our Switch container. If none of the other Route components match, this component will be rendered.

Let’s see this in practice.

In order to use the Switch component, let’s import the Switch component from react-router:

**routing/basics/src/complete/App-7.js**

import React from 'react';

import {

BrowserRouter as Router,

Route,

Link,

Redirect,

**Switch,**

} from 'react-router-dom' const App = () => (

We’ll wrap all of our Route components in a Switch component.

Add the opening Switch component tag above the first Route:

**routing/basics/src/complete/App-7.js**

<hr />

**<Switch>**

<Route path='/atlantic/ocean' render={() => (

Next, we’ll add our “catch-all” Route beneath our existing Route components. Because we don’t specify a path prop, this Route will match every path:

**routing/basics/src/complete/App-7.js**

<Route exact path='/' render={() => ( <h3>

Welcome! Select a body of saline water above. </h3>

)} />

**<Route render={({ location }) => (**

**<div className='ui inverted red segment'>**

**<h3>**

**Error! No matches for <code>{location.pathname}</code> </h3>**

**</div>**

**)} />**

**</Switch>**

</div>

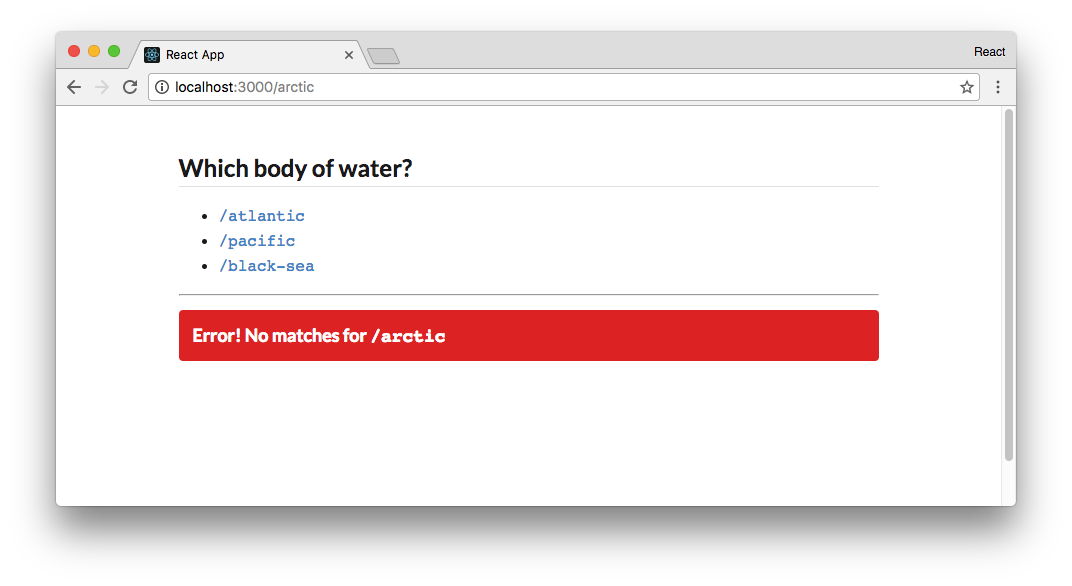
</Router>

);

Route passes the prop location to the render function. Route always passes this prop to its target. We’ll explore this more in the second half of this chapter.

Try it out

Save App.js. Visit /atlantic/ocean and note that the component matching /atlantic is gone. Next, manually enter a path for the app that doesn’t exist. Our catch-all Route component will render:



No matches for **/arctic**

At this point, we’re familiar with React Router’s fundamental components. We wrap our app in Router, which supplies any React Router components in the tree with location and history APIs and ensures our React app is re-rendered whenever the location is changed. Route and Switch both help us control what React components are displayed at a given location. And Link and Redirect give us the ability to modify the location of the app without a full page load.