When using a rich JavaScript framework like React, we want React to generate the page. So an evolution of that request flow using React might look like this:

1. Browser makes a request to the server for this page.
2. **The server doesn't care about the pathname.** Instead, it just returns a standard `index.html` that includes the React app and any static assets.
3. The React app mounts.
4. The React app extracts the identifiers from the URL and uses these identifiers to make an API call to fetch the data for the artist and the album. It might make this call to the same server.
5. The React app renders the page using data it received from the API call.

Projects elsewhere in the book have mirrored this second request flow. One example is the timers app in "Components & Servers." The same `server.js` served both the static assets (the React app) and an API that fed that React app data.

This initial request flow for React is slightly more inefficient than the first. Instead of one round-trip from the browser to the server, there will be two or more: One to fetch the React app and then however many API calls the React app has to make to get all the data it needs to render the page.

However, the gains come after the initial page load. The user experience of our timers app with React is much better than it would be without. Without JavaScript, each time the user wanted to stop, start, or edit a timer, their browser would have to fetch a brand new page from the server. This adds noticeable delay and an unpleasant "blink" between page loads.

Single-page applications (SPAs) are web apps that load once and then dynamically update elements on the page using JavaScript. Every React app we've built so far has been a type of SPA.

So we've seen how to use React to make interface elements on a page fluid and dynamic. But other apps in the book have had only a single location. For instance, the product voting app had a single view: the list of products to vote on. What if we wanted to add a different page, like a product view page at the location `/products/:productId`? This page would use a completely different set of components.

Back to our music website example, imagine the user is looking at the React-powered album view page. They then click on an "Account" button at the top right of the app to view their account information. A request flow to support this might look like:

1. User clicks on the "Account" button which is a link to `/account`.
2. Browser makes a request to `/account`.
3. The server, again, doesn't care about the pathname. Again, it returns the same `index.html` that includes the full React app and static assets.
4. The React app mounts. It checks the URL and sees that the user is looking at the `/accounts` page.

5. The top-level React component, say `App`, might have a switch for what component to render based on the URL. Before, it was rendering `AlbumView`. But now it renders `AccountView`.
6. The React app renders and populates itself with an API request to the server (say `/api/account`).

This approach works and we can see examples of it across the web. But for many types of applications, there's a more efficient approach.

When the user clicks on the "Account" button, we could prevent the browser from fetching the next page from `/account`. Instead, we could instruct the React app to switch out the `AlbumView` component for the `AccountView` component. In full, that flow would look like this:

1. User visits `https://example.com.com/artists/87589/albums/1758221`.
2. The server delivers the standard `index.html` that includes the React app and assets.
3. The React app mounts and populates itself by making an API call to the server.
4. User clicks on the "Account" button.
5. The React app captures this click event. React updates the URL to `https://example.com/account` and re-renders.
6. When the React app re-renders, it checks the URL. It sees the user is viewing `/account` and it swaps in the `AccountView` component.
7. The React app makes an API call to populate the `AccountView` component.

When the user clicks on the "Account" button, the browser already contains the full React app. There's no need to have the browser make a new request to fetch the same app again from the server and re-mount it. The React app just needs to update the URL and then re-render itself with a new component-tree (`AccountView`).

This is the idea of a **JavaScript router**. As we'll see first hand, **routing** involves two primary pieces of functionality: (1) Modifying the location of the app (the URL) and (2) determining what React components to render at a given location.

There are many routing libraries for React, but the community's clear favorite is **React Router**. React Router gives us a wonderful foundation for building rich applications that have hundreds or thousands of React components across many different views and URLs.

React Router's core components

For *modifying* the location of an app, we use links and redirects. In React Router, links and redirects are managed by two React components, `Link` and `Redirect`.

For *determining what to render* at a given location, we also use two React Router components, `Route` and `Switch`.

To best understand React Router, we'll start out by building basic versions of React Router's core components. In doing so, we'll get a feel for what routing looks like in a component-driven paradigm.

We'll then swap out our components for those provided by the `react-router` library. We'll explore a few more components and features of the library.

In the second half of the chapter, we'll see React Router at work in a slightly larger application. The app we build will have multiple pages with dynamic URLs. The app will communicate with a server that is protected by an API token. We'll explore a strategy for handling logging and logging out inside a React Router app.

React Router v4

The latest version of React Router, v4, is a major shift from its predecessors. The authors of React Router state that the most compelling aspect of this version of the library is that it's "just React."

We agree. And while v4 was just released at the time of writing, we find its paradigm so compelling that we wanted to ensure we covered v4 as opposed to v3 here in the book. We believe v4 will be rapidly adopted by the community.

Because v4 is so new, it's possible the next few months will see some changes. But the essence of v4 is settled, and this chapter focuses on those core concepts.

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:


```
$ ls
README.md
nightwatch.json
package.json
public/
semantic/
semantic.json
src/
tests/
```

If you need a refresher on create-react-app, refer to the chapter "Using Webpack with create-react-app."

Our React app lives inside `src/`:

```
$ ls 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:

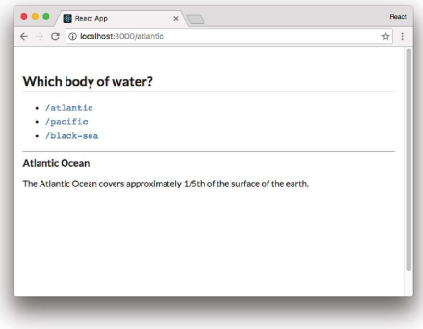
```
$ npm i
```

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";
```

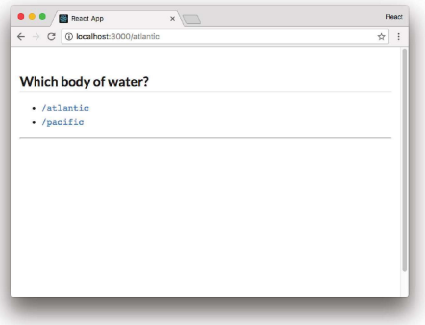
As in other create-react-app apps, the mounting of the React app to the DOM will take place here in `index.js`. Let's un-comment the line that mounts `App`:

```
// [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 />
    );
```

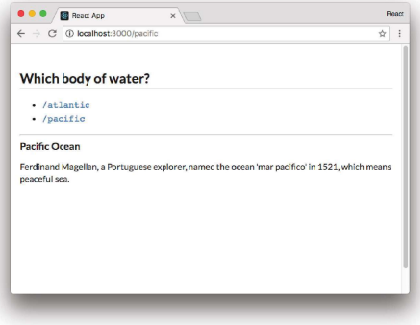
It's imperative when you do this that the component name is capitalized, which is why we extract the component as `Component` in the arguments. But when a component class is a dynamic variable as it is here, oftentimes React developers prefer to just use `React.createElement()` as opposed to JSX.

Try it out

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

```
$ npm start
```

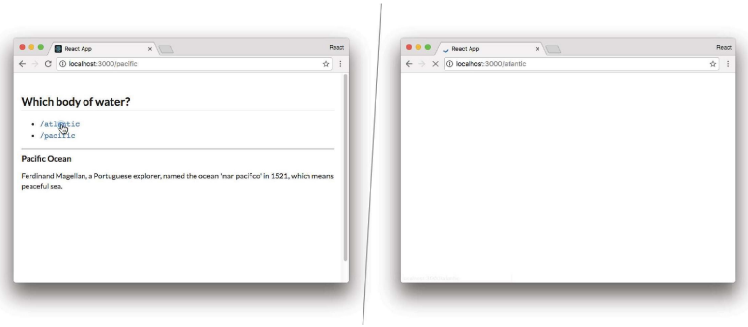
Head to the browser and visit the app. Notice that we're now rendering the appropriate component when we visit each location:



`/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.

For more info on the history API, check out the docs on MDN[77].

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:

[77] https://developer.mozilla.org/en-US/docs/Web/API/History_API

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`

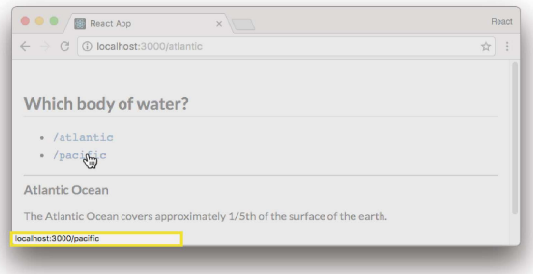
The `onClick` handler for the `<a>` tag first calls `preventDefault()` on the event object. Recall that the first argument passed to an `onClick` handler is always the event object. Calling `preventDefault()` prevents the browser from making a web request for the new location.

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:



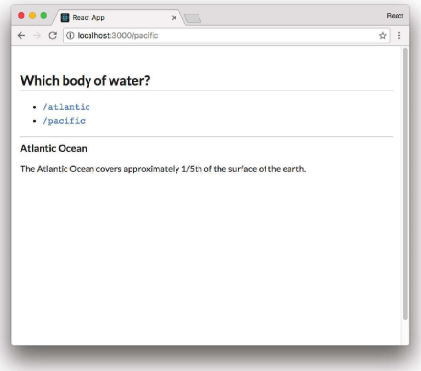
Hovering over a link

`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 `react-router` 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.

If you need a refresher on context, we cover this React feature in the chapter "Advanced Component Configuration."

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 first by defining `childContextTypes`:

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

```
  static childContextTypes = {
    history: PropTypes.object,
    location: PropTypes.object,
  };
```

JavaScript classes: `static`

The line defining `childContextTypes` inside the class is identical to doing this below the class definition:

```
Router.childContextTypes = {
  history: PropTypes.object,
  location: PropTypes.object,
};
```


This keyword allows us to define a property on the class `Router` itself as opposed to instances of `Router`.

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.

Because the state update depends on the current version of state, we're passing `setState()` a *function* as opposed to an *object*. We discuss this technique in the "Advanced Component Configuration" chapter.

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>
    );
  }
}
```

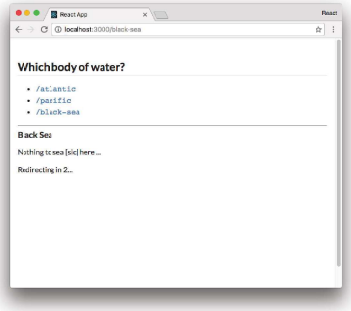
Three seconds after `BlackSea` mounts, our interval function decrements our `state.counter` to `0`. The `setState()` function triggers a re-render of the `BlackSea` component. Its output will include the `Redirect` component. When the `Redirect` component mounts, it will trigger the redirect.

In the `BlackSea` component, we use a ternary operator to control whether or not we render the `Redirect` component. Using ternary operators inside of JSX is common in React. This is because we can't embed multi-line statements like an if/else clause inside our JSX.

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.

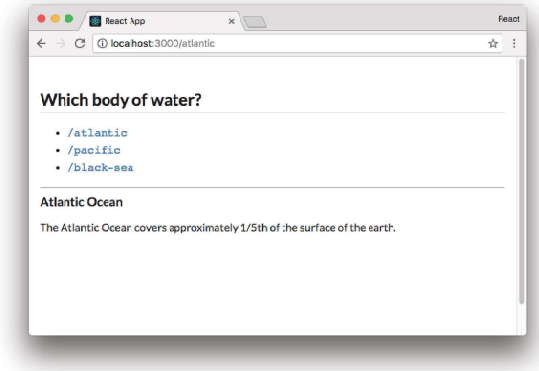
So far, we've used the prop `component` to instruct `Route` which component to render when the `path` matches the current location. `Route` also accepts the prop `render`. We can use this prop to define a render function in-line.

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

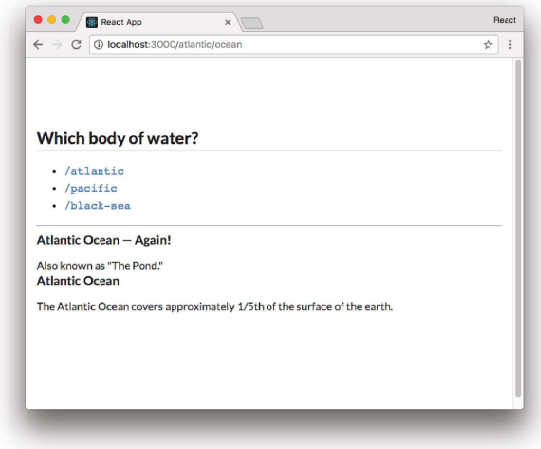
```
      <Route path='/atlantic/ocean' render={() => (
        <div>
          <h3>Atlantic Ocean — Again!</h3>
          <p>
            Also known as "The Pond."
          </p>
        </div>
      )} />
      <Route path='/atlantic' component={Atlantic} />
      <Route path='/pacific' component={Pacific} />
      <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.

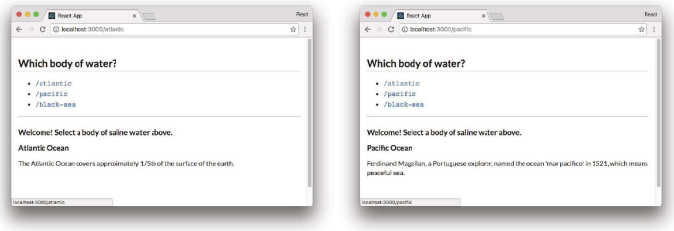
With this behavior in mind, what if we wanted to add a component that renders when the user visits the root (`/`)? It would be nice to have some text that instructs the user to click on one of the links.

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>
      )} />
```

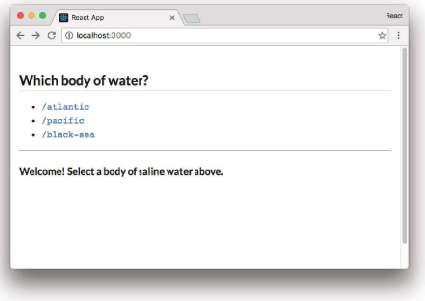
We're using some JSX syntactic sugar here. While we could set the prop explicitly like this:

```
<Route exact={true} path='/' render={() => (
  // ...
)}
```

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>
```

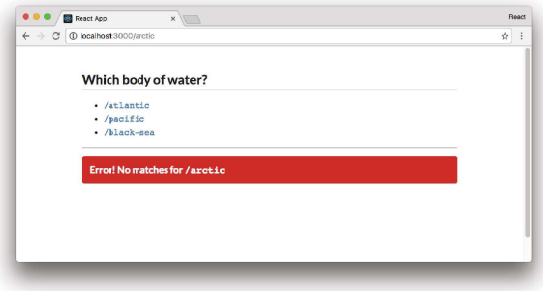


```
  </Router>
);
```

`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.

In the second half of this chapter, we'll apply these fundamentals to a more complex application.

Dynamic routing with React Router

In this section, we'll build off the foundations established in the last. We'll see how React Router's fundamental components work together inside of a slightly larger app, and explore a few different strategies for programming in its unique component-driven routing paradigm.

The app in this section has multiple pages. The app's main page has a vertical menu where the user can choose between five different music albums. Choosing an album immediately shows album information in the main panel. All album information is pulled from Spotify's API[78].

The server that our React app communicates with is protected by a token that requires a login. While not a genuine authentication flow, the setup will give us a feel for how to use React Router inside of an application that requires users to login.

The completed app

The code for this section is inside `routing/music`. From the root of the book's code folder, navigate to that directory:

```
$ cd routing/music
```

Let's take a look at this project's structure:

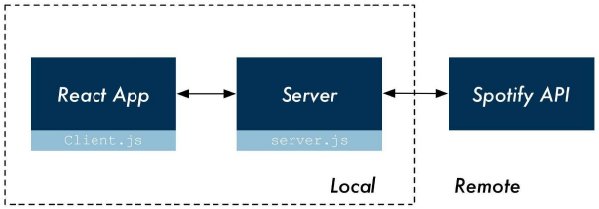
```
$ ls
SpotifyClient.js
client/
nightwatch.json
package.json
server.js
server.test.js
start-client.js
start-server.js
tests/
```

In the root of the project is a Node API server (`server.js`). Inside `client` is a React app powered by create-react-app:

```
$ ls client
package.json
public/
semantic/
semantic.json
src/
```

[78] https://developer.spotify.com/web-api/

This project's structure is identical to the food lookup app at the end of the chapter "Using Webpack with create-react-app." When in development, we boot two servers: `server.js` and the Webpack development server. The Webpack development server will serve our React app. Our React app interfaces with `server.js` to fetch data about a given album. `server.js`, in turn, communicates with the Spotify API to get the album data:



Communication diagram

Let's install the dependencies and see the running app. We have two `package.json` files, one for `server.js` and one for the React app. We'll run `npm i` for both:

```
$ npm i
$ cd client
$ npm i
$ cd ..
```

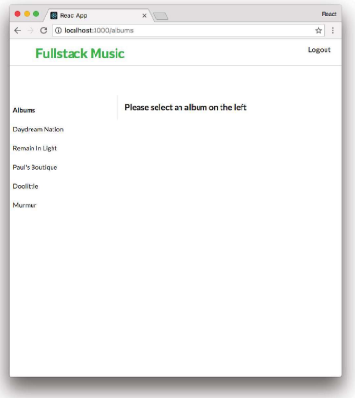
We can boot the app with `npm start` in the top-level directory. This uses Concurrently to boot both servers simultaneously:

```
$ npm start
```

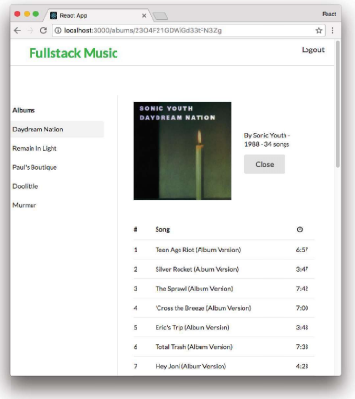
Find the app at `http://localhost:3000`.

The app will prompt you with a login button. Click login to "login." We are not prompted for a user name or password.

After logging in, we can see a list of albums in a vertical side menu:



Clicking on one of these albums displays it to the right of the vertical side menu. Furthermore, the URL of the app is updated:



The URL follows the scheme `/albums/:albumId` where `:albumId` is the **dynamic part** of the URL.

Clicking the "Logout" button in the top right, we're redirected to the login page at `/login`. If we try

to manually navigate back to `/albums` by typing that address into the URL bar, we are prevented from reaching that page. Instead, we are redirected back to `/login`.

Before digging into the React app, let's take a look at the server's API.

The server's API

POST `/api/login`

The server provides an endpoint for retrieving an API token, `/api/login`. This token is required for the endpoint `/api/albums`.

Unlike a real-world login endpoint, the `/api/login` endpoint does not require a user name or a password. `server.js` will always return a hard-coded API token when this endpoint is requested. That hard-coded token is a variable inside `server.js`:

routing/music/server.js

```
// A fake API token our server validates
export const API_TOKEN = 'D6W69PRgCoDKgHZGJmRUNA';
```

To test this endpoint yourself, with the server running you can use `curl` to make a POST request to that endpoint:

```
$ curl -X POST http://localhost:3001/api/login
{
  "success": true,
  "token": "D6W69PRgCoDKgHZGJmRUNA"
}
```

The React app stores this API token in `localStorage`. React will include this token in all subsequent requests to the API. Clicking the "Logout" button in the app removes the token from React and `localStorage`. The user will have to "login" again to access the app.

We interact with both the API and `localStorage` using the `Client` library declared in `client/src/Client.js`. We discuss this library more later.

The `localStorage` API allows you to read and write to a key-value store in the user's browser. You can store items to `localStorage` with `setItem()`:

```
localStorage.setItem('gas', 'pop');
```

And retrieve them later with `getItem()`:

```
localStorage.getItem('gas');
// => 'pop'
```

Note that items stored in `localStorage` have no expiry.

Security and client-side API tokens

Security on the web is a huge topic. And managing client-side API tokens is a delicate task. To build a truly secure web application, it's important to understand the intricacies of the topic. Unfortunately, it's far too easy to miss subtle practices which can end up leaving giant security holes in your implementation.

While using `localStorage` to store client-side API tokens works fine for hobby projects, there are significant risks. Your users' API tokens are exposed to cross-site scripting attacks. And tokens stored in `localStorage` impose no requirement on their safe transfer. If someone on your development team accidentally inserts code that makes requests over `http` as opposed to `https`, your tokens will be transferred over the wire exposed.

As a developer, you are obligated to be careful and deliberate when users entrust you with sensitive data. There are strategies you can use to protect your app and your users, like using JSON Web Tokens (JWTs) or cookies or both. Should you find yourself in this fortunate position, take the necessary time to carefully research and implement your token management solution.

GET `/api/albums`

The `/api/albums` endpoint returns data supplied by the Spotify API for a given list of albums. We supply the endpoint with a query parameter, `ids`, which we set to the list of desired album IDs:

```
/api/albums?ids=<id1>,<id2>
```

Note that IDs are comma-separated.

This endpoint also expects the API token to be included as the query param `token`. Including both the `ids` and `token` query params looks like this:


```
/api/albums?ids=<id1>,<id2>&token=<token>
```

Here's an example of querying the `/api/albums` endpoint with `curl`:

```
$ curl -X GET \
"http://localhost:3001/api/albums"\
"?ids=1DWWb4Q39mp1T3NgyscowF,2ANVost0y2y52ema1E9xAZ"\
"&token=D6W69PRgCoDKgHZGJmRUNA"
```

In bash, the `\` character allows us to split our command over multiple lines. Strings can be split over multiple lines in a similar manner:

```
$ echo "part1"\
"part2"
-> part1part2
```

We do this above for readability. Importantly, there is no whitespace between the `"` and `\` characters. Furthermore, `"part2"` does not have any whitespace ahead of it. If there was any whitespace, the strings would not concatenate properly.

If you're using Windows, you can just write this command as a single line.

The information for each album is extensive, so we refrain from including an example response here.

Starting point of the app

The completed components and the steps we take to get to them are located under `client/src/components-complete`. We'll be writing all the code for the rest of this chapter in `client/src/components`.

Touring around the existing code, our first stop is `index.js`. Check out its `import` statements:

routing/music/client/src/index.js

```
import React from "react";
import ReactDOM from "react-dom";

import { BrowserRouter as Router } from "react-router-dom";

import App from './components/App';

import "./styles/index.css";
import "../semantic/dist/semantic.css";

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

Note that we're importing `Router` here.

Like the last project, `index.js` includes `index-complete.js` which allows us to traverse each iteration of `App` inside `components/complete`. Let's comment out that import statement:

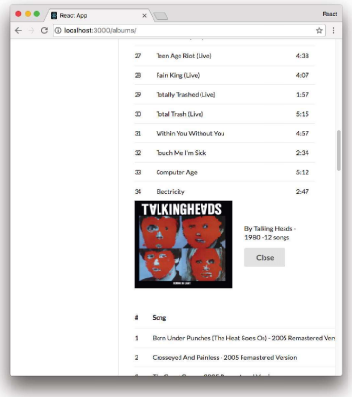
```
// [STEP 1] Comment out this line:
// import './index-complete';
```

Then, at the bottom of `index.js`, un-comment the call to `ReactDOM.render()`. Note that we wrap `<App>` in `<Router>`:

```
// [STEP 2] Un-comment these lines:
ReactDOM.render(
  <Router>
    <App />
  </Router>,
  document.getElementById("root")
);
```

Wrapping your `App` in `index.js` with `<Router>` is a common pattern for React Router apps.

Save `index.js`. With our development server still running, we'll see that our starting point is a stripped-down interface:



Initial App

Our starting point doesn't use React Router at all. The app lists out all the albums on one page. There is a "Logout" button and clicking it changes the URL but doesn't appear to do anything else. The "Close" buttons don't work either.

Next, let's take a look at `App.js`:

routing/music/client/src/components/App.js

```
import React from 'react';

import TopBar from './TopBar';
import AlbumsContainer from './AlbumsContainer';

import '../styles/App.css';

const App = () => (
  <div className='ui grid'>
    <TopBar />
    <div className='spacer row' />
    <div className='row'>
      <AlbumsContainer />
    </div>
  </div>
);
```

```
export default App;
```

`App` renders both our `TopBar` and `AlbumsContainer`.

As always, the `div` and `className` elements throughout the app are present just for structure and styling. As in other projects, this app uses SemanticUI[79].

We won't look at `TopBar` right now.

`AlbumsContainer` is the component that interfaces with our API to fetch the data for the albums. It then renders `Album` components for each album.

At the top of `AlbumsContainer`, we define our import statements. We also have a hard-coded list of `ALBUM_IDS` that `AlbumsContainer` uses to fetch the desired albums from the API:

routing/music/client/src/components/AlbumsContainer.js

```
import React, { Component } from 'react';

import Album from './Album';
import { client } from '../Client';

const ALBUM_IDS = [
  '2304F21GDWiGd33tFN3ZgI',
  '3AQgdwMNCiN7awXch5fAaG',
  '1kmyirVya5fRxdjsPFDM05',
  '6ymZBbRSmzAvoSGmwAFoxm',
  '4Mw9Gcu1LT7JaipXdwrq1Q',
];
```

`AlbumsContainer` is a stateful component with two state properties:

- `fetched`: Whether or not `AlbumsContainer` has successfully fetched the album data from the API yet
- `albums`: An array of all the album objects

Stepping through the component, we'll start with its initial state:

[79] http://semantic-ui.com

routing/music/client/src/components/AlbumsContainer.js

```
class AlbumsContainer extends Component {
  state = {
    fetched: false,
    albums: [],
  };
```

We use the `fetched` boolean to keep track of whether or not the albums have been retrieved from the server.

After our `AlbumsContainer` component mounts, we call the `this.getAlbums()` function. This populates the `albums` in state:

routing/music/client/src/components/AlbumsContainer.js

```
  componentDidMount() {
    this.getAlbums();
  }
```

In our `getAlbums()` function, we use the `Client` library (inside `src/Client.js`) to make a request to the API to grab the data for the albums specified in `ALBUM_IDS`. We use the method `getAlbums()` from the library, which expects as an argument an array of album IDs.

When we get the data back, we update the state to set `fetched` to `true` and `albums` to the result:

routing/music/client/src/components/AlbumsContainer.js

```
  getAlbums = () => {
    client.setToken('D6W69PRgCoDKgHZGJmRUNA');
    client.getAlbums(ALBUM_IDS)
      .then((albums) => (
        this.setState({
          fetched: true,
          albums: albums,
        })
      ));
  };
```

We introduce Fetch and promises in the chapter "Components & Servers."

Notice that before calling `client.getAlbums()` we call `client.setToken()`. As mentioned earlier, our API expects a token in the request to `/api/albums`. Since we don't yet have the login and logout functionality implemented in the app, we cheat by setting the token manually before making our request. This is the same token expected by `server.js`:

routing/music/server.js

```
export const API_TOKEN = 'D6W69PRgCoDKgHZGJmRUNA';
```

Finally, the `render` method for `AlbumsContainer` switches on `this.state.fetched`. If the data has yet to be fetched, we render the loading icon. Otherwise, we render all the albums in `this.state.albums`:

routing/music/client/src/components/AlbumsContainer.js

```
  render() {
    if (!this.state.fetched) {
      return (
        <div className='ui active centered inline loader' />
      );
    } else {
      return (
        <div className='ui two column divided grid'>
          <div
            className='ui six wide column'
            style={{ maxWidth: 250 }}
          >
            {/* VerticalMenu will go here */}
          </div>
          <div className='ui ten wide column'>
            {
              this.state.albums.map((a) => (
                <div
                  className='row'
                  key={a.id}
                >
                  <Album album={a} />
                </div>
              ))
            }
          </div>
        </div>
      );
    }
  }
```

Our first update will be to add the vertical menu that we saw in the completed version of the app. This vertical menu should allow us to choose which album we'd like to view. When an album in

the vertical menu is selected, the album should be displayed and the location of the app updated to `/albums/:albumId`.

Using URL params

At the moment, our `App` component is rendering both `TopBar` and `AlbumsContainer`:

routing/music/client/src/components/App.js

```
const App = () => (
  <div className='ui grid'>
    <TopBar />
    <div className='spacer row' />
    <div className='row'>
      <AlbumsContainer />
    </div>
  </div>
);
```

As we saw, we'll eventually have login and logout pages. We can keep `TopBar` in `App` as we want `TopBar` to appear on every page. Since we want the album listing page to appear on a route, we should nest `AlbumsContainer` inside a `Route`. We'll have it only render at the location `/albums`. This will prepare us for adding `/login` and `/logout` soon.

First, inside `App.js`, let's import the `Route` component:

routing/music/client/src/components-complete/App-1.js

```
import React from 'react';

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

Then, we'll use `Route` on the path `/albums`:

routing/music/client/src/components-complete/App-1.js

```
const App = () => (
  <div className='ui grid'>
    <TopBar />
    <div className='spacer row' />
    <div className='row'>
      <Route path='/albums' component={AlbumsContainer} />
    </div>
  </div>
);
```

Now, `AlbumsContainer` will only render when we visit the app at `/albums`.

We'll have `AlbumsContainer` render a child component, `VerticalMenu`. We'll have the parent (`AlbumsContainer`) pass the child (`VerticalMenu`) the list of `albums`. Let's first compose `VerticalMenu` and then we'll update `AlbumsContainer` to use it.

Open the file `src/components/VerticalMenu.js`. The current file contains the scaffold for the component:

routing/music/client/src/components/VerticalMenu.js

```
import React from 'react';

import '../styles/VerticalMenu.css';

const VerticalMenu = ({ albums }) => (
  <div className='ui secondary vertical menu'>
    <div className='header item'>
      Albums
    </div>
    {/* Render album menu here */}
  </div>
);

export default VerticalMenu;
```

As we can see, `VerticalMenu` expects the prop `albums`. We want to iterate over the `albums` prop, rendering a `Link` for each album. First, we'll import `Link` from `react-router`:

routing/music/client/src/components-complete/VerticalMenu-1.js

```
import React from 'react';

import { Link } from 'react-router-dom';

import '../styles/VerticalMenu.css';
```

We'll use `map` to compose our list of `Link` components. The `to` prop for each `Link` will be `/albums/:albumId`:

routing/music/client/src/components-complete/VerticalMenu-1.js

```
const VerticalMenu = ({ albums }) => (
  <div className='ui secondary vertical menu'>
    <div className='header item'>
      Albums
    </div>
    {
      albums.map((album) => (
        <Link
          to={`/albums/${album.id}`}
          className='item'
          key={album.id}
        >
          {album.name}
        </Link>
      ))
    }
  </div>
);
```

We set the `className` to `item` for styling. We're using SemanticUI's vertical menu.

`VerticalMenu` will now update the location of the app whenever the user clicks on one of its menu items. Let's update `AlbumsContainer` to both use `VerticalMenu` and render a single album based on the location of the app.

Open `src/components/AlbumsContainer.js`. Add `VerticalMenu` to the list of imports:

```
import Album from './Album';
import VerticalMenu from './VerticalMenu';
import { client } from '../Client';
```

Then, inside the `render` method, we'll add `VerticalMenu` nested inside a column:

routing/music/client/src/components-complete/AlbumsContainer-1.js

```
  render() {
    if (!this.state.fetched) {
      return (
        <div className='ui active centered inline loader' />
      );
    } else {
      return (
        <div className='ui two column divided grid'>
          <div
            className='ui six wide column'
            style={{ maxWidth: 250 }}
          >
            <VerticalMenu
              albums={this.state.albums}
            />
          </div>
          <div className='ui ten wide column'>
```

Because `VerticalMenu` does not need full album objects, it would indeed be cleaner to pass the component subset objects that look like this:

```
[{ name: 'Madonna', id: '1DWWb4Q39mp1T3NgyscowF' }]
```

This would also make `VerticalMenu` more flexible, as we could write it such that it can be a side menu for any list of items.

In the output for `AlbumsContainer`, in the column adjacent to `VerticalMenu`, we want to render a single album now as opposed to a list of all of them. We know that `VerticalMenu` is going to modify the location according to the scheme `/albums/:albumId`. We can use a `Route` component to match against this pattern and extract the parameter `:albumId` from the URL.

Still in `AlbumsContainer.js`, first add `Route` to the component's imports:

routing/music/client/src/components-complete/AlbumsContainer-1.js

```
import React, { Component } from 'react';

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

Then, inside `render` in the `div` tags below `VerticalMenu`, we'll replace the `map` call that renders all the albums. Instead, we'll define a `Route` component with a `render` prop. We'll take a look at it in full then break it down:

routing/music/client/src/components-complete/AlbumsContainer-1.js

```
          <div className='ui ten wide column'>
            <Route
              path='/albums/:albumId'
              render={({ match }) => {
                const album = this.state.albums.find(
                  (a) => a.id === match.params.albumId
                );
                return (
                  <Album
                    album={album}
                  />
                );
              }}
            />
          </div>
```

`path`

The string we're matching against is `/albums/:albumId`. The `:` is how we indicate to React Router that this part of the URL is a **dynamic parameter**. Importantly, **any value** will match this dynamic parameter.

`render`

We set a `render` prop on this `Route` to a function. `Route` invokes the render function with a few arguments, like `match`. We explore `match` more in a bit. Here, we're interested in the **`params`** property on `match`.

`Route` extracts all the dynamic parameters from the URL and passes them to the target component inside the object `match.params`. In our case, `params` will contain the property `albumId` which will correspond to the value of the `:albumId` part of the current URL (`/albums/:albumId`).

We use `find` to get the album that matches `params.albumId`, rendering a single `Album`.

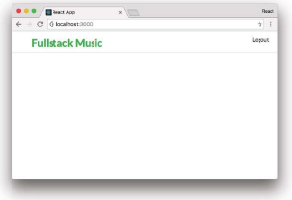
The user can now modify the location of the app with the links in `VerticalMenu`. And `AlbumsContainer` uses `Route` to read the location and extract the desired `albumId`, rendering the desired album.

Try it out

Save `AlbumsContainer.js`. If the app isn't still running, be sure to boot it from the top-level directory using the `npm start` command:

```
$ npm start
```

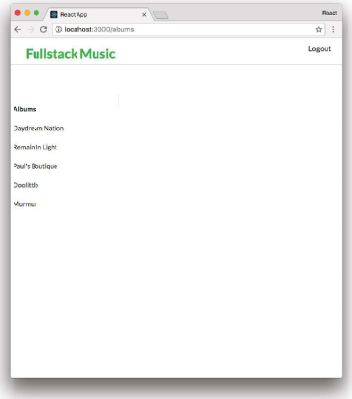
Currently only the `TopBar` will be visible when we visit `http://localhost:3000` in our browser:



Nothing matches at `/`

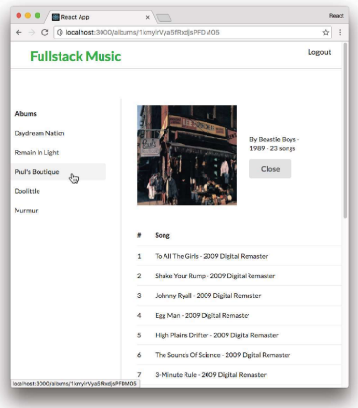
Recall that in `App` we wrapped `AlbumsContainer` in a `Route` for the pattern `/albums`. It doesn't match `/`, so the body of `App` remains empty.

Let's manually visit the `/albums` path by typing it in the browser URL bar and we'll see our `VerticalMenu` rendered:



`VerticalMenu` appears

Nothing is rendered in the column adjacent to `VerticalMenu` as that column is awaiting a URL matching `/albums/:albumId`. Clicking on one of the albums changes the location of the app:



Selecting an album

We're getting somewhere! While the "Close" and "Logout" buttons still don't work, we can switch between albums. The app's location updates without refreshing the page.

Let's wire up the "Close" button next.

Propagating pathnames as props

Inside `Album.js`, we render a "Close" button inside the album's header:

routing/music/client/src/components-complete/Album-1.js

```
        <div className='six wide column'>
          <p>
            {
              `By ${album.artist.name}
              - ${album.year}
              - ${album.tracks.length} songs`
            }
          </p>
          <div
            className='ui left floated large button'
          >
            Close
          </div>
```


To "close" the album, we need to change the app's location from `/albums/:albumId` to `/albums`. Knowing what we know about routing through this point, we could create a link to handle this behavior for us, like this:

```
// Valid "close" button
<Link
  to='/albums'
  className='ui left floated large button'
>
  Close
</Link>
```

This is perfectly valid. However, one consideration we have to make as we add routing to our apps is flexibility.

For example, what if we wanted to modify the app so that our albums page existed on `/` and not `/albums`? We'd have to change all references to `/albums` in the app.

More compelling, what if we wanted to display an album in different locations within the app? For example, we might add artist pages at `/artists/:artistId`. The user can then drill down to an individual album, opening the URL `/artists/:artistId/albums/:albumId`. We'd want a "Close" button in this situation to link to `/artists/:artistId`, not `/albums`.

A simple way to maintain some flexibility is to pass pathnames through the app as props. Let's see this in action.

Recall that inside `App.js`, we specify the path for `AlbumsContainer` is `/albums`:

routing/music/client/src/components-complete/App-1.js

```
      <Route path='/albums' component={AlbumsContainer} />
```

We just saw that `Route` invokes a function passed as a `render` prop with the argument `match`. `Route` also sets this prop on components rendered via the `component` prop. Regardless of how `Route` renders its component, it will always set three props:

- `match`
- `location`
- `history`

According to the React Router docs[80], the `match` object contains the following properties:

[80] https://reacttraining.com/react-router/web/api/Route

- `params` - (object) Key/value pairs parsed from the URL corresponding to the dynamic segments of the path
- `isExact` - true if the entire URL was matched (no trailing characters)
- `path` - (string) The path pattern used to match. Useful for building nested `<Route>`s
- `url` - (string) The matched portion of the URL. Useful for building nested `<Link>`s

Of interest to us is the property `path`. Inside `AlbumsContainer`, `this.props.match.path` will be `/albums`.

We can update the `Route` inside `AlbumsContainer` that wraps `Album`. Before, the `path` prop was `/albums/:albumId`. We can replace the root of the path (`/albums`) with the variable `this.props.match.path`.

First, let's declare a new variable, `matchPath`:

routing/music/client/src/components-complete/AlbumsContainer-2.js

```
  render() {
    if (!this.state.fetched) {
      return (
        <div className='ui active centered inline loader' />
      );
    } else {
      const matchPath = this.props.match.path;
```

Then, we can change the `path` prop on `Route` to use this variable:

routing/music/client/src/components-complete/AlbumsContainer-2.js

```
          <div className='ui ten wide column'>
            <Route
              path={`${matchPath}/:albumId`}
              render={({ match }) => {
                const album = this.state.albums.find(
                  (a) => a.id === match.params.albumId
                );
                return (
```

With this approach, `AlbumsContainer` doesn't make any assumptions about its location. We could, for example, update `App` so that `AlbumsContainer` matches at `/` instead of `/albums` and `AlbumsContainer` would need no changes.

We want the "Close" button in `Album` to link to this same path. "Closing" an album means changing the location back to `/albums`. Let's propagate the prop down to `Album`:

routing/music/client/src/components-complete/AlbumsContainer-2.js

```
                return (
                  <Album
                    album={album}
                    albumsPathname={matchPath}
                  />
                );
```

Then, inside `Album.js` we can extract the `albumsPathname` from the props object:

routing/music/client/src/components-complete/Album.js

```
const Album = ({ album, albumsPathname }) => (
```

Now, we can change the `div` element to a `Link` component. We'll set the `to` prop to `albumsPathname`:

routing/music/client/src/components-complete/Album.js

```
        <div className='six wide column'>
          <p>
            {
              `By ${album.artist.name}
              - ${album.year}
              - ${album.tracks.length} songs`
            }
          </p>
          <Link
            to={albumsPathname}
            className='ui left floated large button'
          >
            Close
          </Link>
```

Let's apply the same approach to `VerticalMenu`. Switch back to `AlbumsContainer.js`. We'll pass `VerticalMenu` the prop `albumsPathname`:

routing/music/client/src/components-complete/AlbumsContainer-2.js

```
          <div
            className='ui six wide column'
            style={{ maxWidth: 250 }}
          >
            <VerticalMenu
              albums={this.state.albums}
              albumsPathname={matchPath}
            />
          </div>
```

Now we can use this prop to compose the `to` attribute for `Link`:

routing/music/client/src/components-complete/VerticalMenu-2.js

```
const VerticalMenu = ({ albums, albumsPathname }) => (
  <div className='ui secondary vertical menu'>
    <div className='header item'>
      Albums
    </div>
    {
      albums.map((album) => (
        <Link
          to={`${albumsPathname}/${album.id}`}
          className='item'
          key={album.id}
        >
          {album.name}
        </Link>
      ))
    }
  </div>
);
```

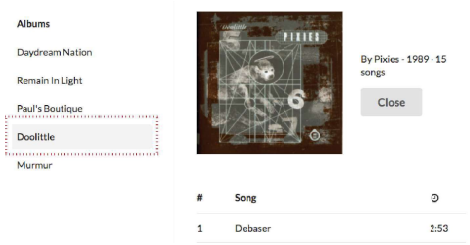
By isolating the number of places we specify pathnames we make our app more flexible for routing changes in the future. With this update, the only place in our app where we specify the path `/albums` is in `App`.

Try it out

Save `Album.js`. With the app running, visit `/albums`. Clicking on an album opens it. Clicking on "Close" closes it by changing the location back to `/albums`.

With the "Close" button working, there's an interface improvement we can make before moving on to implementing login and logout.

When an album is open, it would be nice if the sidebar indicated which album was active:



`VerticalMenu` indicating which album is active with a light highlight

We'll solve this next.

Dynamic menu items with `NavLink`

At the moment, all the menu items in our vertical menu have the class `item`. In SemanticUI's vertical menus, we can set the class of the active item to `active item` to have it show up as highlighted.

How will we know if an album is "active"? This state is maintained in the URL. If the album's `id` matches that of `:albumId` in the URL, we'd know that album was active.

Given this, we could drum up the following solution:

```
albums.map((album) => {
  const to = `${albumsPathname}/${album.id}`;
  const active = window.location.pathname === to;
  <Link
    to={to}
    className={active ? 'active item' : 'item'}
    key={album.id}
  >
    {album.name}
  </Link>
})
```

We grab the URL's pathname through the browser's `window.location` API. If the browser's location matches that of the link, we set its class to `active item`.

Styling active links is a common need. While the solution above would work, `react-router` provides a built-in strategy to handle this situation.

We can use another link component, **`NavLink`**. `NavLink` is a special version of `Link` that is intended for just this purpose. When a `NavLink` component's target location matches the current URL, it will add style attributes to the rendered element.

We can use `NavLink` like this:

routing/music/client/src/components-complete/VerticalMenu-3.js

```
        <NavLink
          to={`${albumsPathname}/${album.id}`}
          activeClassName='active'
          className='item'
          key={album.id}
        >
```

When the `to` prop on a `NavLink` matches the current location, the class applied to the element will be a combination of `className` and `activeClassName`. Here, that will render the class `active item` as desired.

In this instance, however, we don't need to set the prop `activeClassName`. The default value of `activeClassName` is the string `'active'`. So, we can omit it.

Import `NavLink` at the top of the file:

routing/music/client/src/components-complete/VerticalMenu.js

```
import { NavLink } from 'react-router-dom';
```

Then swap out `Link` for `NavLink`:

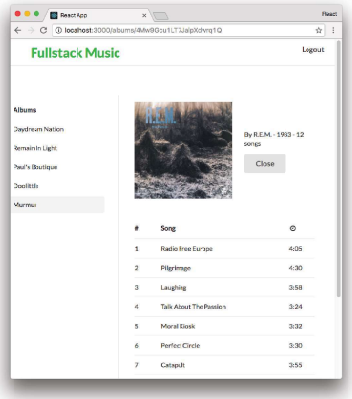
routing/music/client/src/components-complete/VerticalMenu.js

```
const VerticalMenu = ({ albums, albumsPathname }) => (
  <div className='ui secondary vertical menu'>
    <div className='header item'>
      Albums
    </div>
    {
      albums.map((album) => (
        <NavLink
          to={`${albumsPathname}/${album.id}`}
          className='item'
          key={album.id}
        >
          {album.name}
        </NavLink>
      ))
    }
  </div>
);
```


`NavLink` makes styling links easy. Use `NavLink` when you need this feature but stick to `Link` when you don't.

Try it out

Save `VerticalMenu.js`. In our browser, the vertical menu will now reflect the active album with a gray background:



Menu highlights the active album

Let's handle one last thing before moving on to login and logout. `AlbumsContainer` matches on `/albums`. If the user visits `/`, it would be nice if they were redirected to `/albums`. We'll add that now.

Adding a redirect for the root path

Inside `App.js`, we first import `Redirect`:

routing/music/client/src/components-complete/App-3.js

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

We'll then be able to add the `Redirect` component to the output of `App`. We want to `Route` the path `/` **exactly**:

routing/music/client/src/components-complete/App-3.js

```
    <div className='row'>
      <Route path='/albums' component={AlbumsContainer} />

      <Route exact path='/' render={() => (
        <Redirect
          to='/albums'
        />
      )} />
    </div>
```

Recall that the `exact` attribute is necessary here. Otherwise, the pattern `/` would match against every route, including `/albums` and `/login`.

Try it out

Save `App.js`. In your browser, visiting `/` redirects to `/albums` and the vertical menu of albums renders.

We've gotten to see some of React Router's components at work inside a slightly more complex interface:

- We matched a component against a dynamic URL
- We used some of the properties in the `match` argument that `Route` sets on its target component
- We propagated the pathname `/albums` down from `App`, a best practice
- We used `NavLink` to render styled link elements

Let's take this a bit further. In the next section, we'll implement a fake authentication system for our app. We'll explore a strategy for elegantly preventing a user from accessing certain locations without logging in first.

Supporting authenticated routes

As we saw when we explored the API endpoint `/api/albums` earlier, this endpoint requires a token to access. At the moment, we're cheating by manually setting the token before every request in `getAlbums()`:

routing/music/client/src/components-complete/AlbumsContainer-2.js

```
  getAlbums = () => {
    client.setToken('D6W69PRgCoDKgHZGJmRUNA');
    client.getAlbums(ALBUM_IDS)
```

To mimic a more real-world authentication flow, we want to remove the token string literal from our client app. Instead, we should have our app make a request to the API's `/api/login` endpoint. As we saw, to keep things simple our API doesn't require a user name or password. But, mimicking an actual login endpoint, it will return a token that we can store locally and use in subsequent requests.

The `Client` library

Packaged with our app is a client library inside `client/src/Client.js`. The `Client` library has all the methods we need to interact with the server's API.

The `Client` library has a method, `login()`, that will both perform a request to `/api/login` and store the token. This `login()` function executes the request, checks to make sure it's the expected `201` status, parses the `json` response value, and stores the token by using the `setToken()` function:

routing/music/client/src/Client.js

```
  login() {
    return fetch('/api/login', {
      method: 'post',
      headers: {
        accept: 'application/json',
      },
    }).then(this.checkStatus)
      .then(this.parseJson)
      .then((json) => this.setToken(json.token));
  }
```

`setToken()` stores the token in `localStorage`:

routing/music/client/src/Client.js

```
  setToken(token) {
    this.token = token;

    if (this.useLocalStorage) {
      localStorage.setItem(LOCAL_STORAGE_KEY, token);
    }
  }
```

When the app loads, `Client` first tries to load the token from `localStorage`. The token is kept in `localStorage` indefinitely and has no expiry. Therefore, the user will only ever be logged out of our app when they perform a logout.

To logout, we can use the `logout()` function implemented in the `Client` library:

routing/music/client/src/Client.js

```
  logout() {
    this.removeToken();
  }
```

`removeToken()` nullifies the token and removes it from `localStorage`:

routing/music/client/src/Client.js

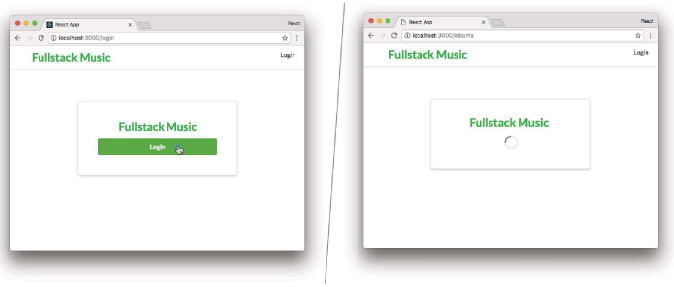
```
  removeToken() {
    this.token = null;

    if (this.useLocalStorage) {
      localStorage.removeItem(LOCAL_STORAGE_KEY);
    }
  }
```

With an idea of the functions we'll use in `Client`, let's begin by adding a new login component.

Implementing login

As we saw in the completed version of the app, the `Login` component displays a single "Login" button. Clicking this button fires off the login process. We want to display the loading indicator while the login is in process:



Clicking the "Login" button on `Login`

Open up `Login.js`. This file contains the scaffold for the component. We'll use a state property `loginInProgress` to indicate whether the login is underway. We'll use another property in state, `shouldRedirect`, to indicate if the login process has completed and the component should redirect to `/albums`. Let's declare both the initial state and the `performLogin()` function now:

routing/music/client/src/components-complete/Login-1.js

```
class Login extends Component {
  state = {
    loginInProgress: false,
    shouldRedirect: false,
  };

  performLogin = () => {
    this.setState({ loginInProgress: true });
    client.login().then(() => (
      this.setState({ shouldRedirect: true })
    ));
  };
```

We set the state properties `loginInProgress` and `shouldRedirect` initially to `false`.

In `performLogin()`, we first flip `loginInProgress` to `true`. When `client.login()` completes, we flip the state property `shouldRedirect` to `true`. We don't need to flip `loginInProgress` back to `false` as the component will be redirecting immediately anyway.

In `render()`, we first check to see if the component should redirect. If it should, we render a `Redirect` component. Otherwise, we can use `loginInProgress` to determine whether to show the "Login" button or the loading indicator:

routing/music/client/src/components-complete/Login-1.js

```
  render() {
    if (this.state.shouldRedirect) {
      return (
        <Redirect to='/albums' />
      );
    } else {
      return (
        <div className='ui one column centered grid'>
          <div className='ten wide column'>
            <div
              className='ui raised very padded text container segment'
              style={{ textAlign: 'center' }}
            >
              <h2 className='ui green header'>
                Fullstack Music
              </h2>
              {
                this.state.loginInProgress ? (
                  <div className='ui active centered inline loader' />
                ) : (
                  <div
                    className='ui large green submit button'
                    onClick={this.performLogin}
                  >
                    Login
                  </div>
                )
              }
            </div>
          </div>
        </div>
      );
    }
  }
```

Save `Login.js`. In order to test `Login`, we'll need to add our `Logout` component.

Logging out doesn't require an API request. We just need to call `client.logout()` which instantly removes the token stored locally. We can perform this call in `constructor()`. We'll then redirect to the login path, which will be `/login`.

First, inside `Logout.js`, import `Redirect` from `react-router`:

routing/music/client/src/components-complete/Logout.js

```
import React, { Component } from 'react';

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

Then we'll fill in the `Logout` component:

routing/music/client/src/components-complete/Logout.js

```
class Logout extends Component {

  constructor(props) {
    super(props);

    client.logout();
  }

  render() {
    return (
      <Redirect
        to='/login'
      />
    );
  }
}
```

Save `Logout.js`. With `Login` and `Logout` composed, we just need to add them to `App`.

First, we import the components:

```
// Top of `App.js`
import TopBar from './TopBar';
import AlbumsContainer from './AlbumsContainer';
import Login from './Login';
import Logout from './Logout';
```

And then we add `Route` components for each of them:

routing/music/client/src/components-complete/App-4.js

```
    <div className='row'>
      <Route path='/albums' component={AlbumsContainer} />

      <Route path='/login' component={Login} />
      <Route path='/logout' component={Logout} />

      <Route exact path='/' render={() => (
        <Redirect
          to='/albums'
        />
      )} />
    </div>
```

Finally, let's remove the manual `setToken()` call from inside `getAlbums()` in `AlbumsContainer`:

```
getAlbums() {
  client.setToken('D6W69PRgCoDKgHZGJmRUNA');
  client.getAlbums(ALBUM_IDS)
  // ...
};
```

Save `AlbumsContainer.js`. With `setToken()` removed, we can now test that our login functionality works. When we click the "Login" button at `/login`, it should trigger an API call to `/api/login` and set the token given in the response.

The component `TopBar` has been responsible for rendering the menu bar at the top of the page. The far right button has said "Logout" thus far as we've always been logged in. `TopBar` uses a function on `Client`, `isLoggedIn()`, to check if the user is logged in. This function checks if the token is present. We render either a "Login" link or a "Logout" link based on this boolean value:

routing/music/client/src/components/TopBar.js

```
    <div className='right menu'>
      {
        client.isLoggedIn() ? (
          <Link className='ui item' to='/logout'>
            Logout
          </Link>
        ) : (
          <Link className='ui item' to='/login'>
            Login
```



```
          </Link>
        )
      }
    </div>
```

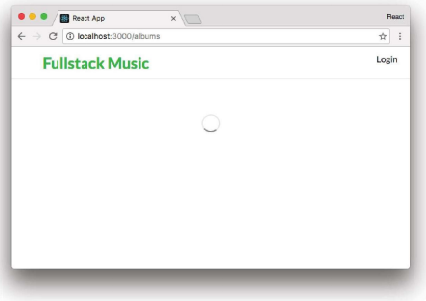
With this in mind, let's test the app out.

Try it out

We'll take some steps to make sure that login and logout are working properly.

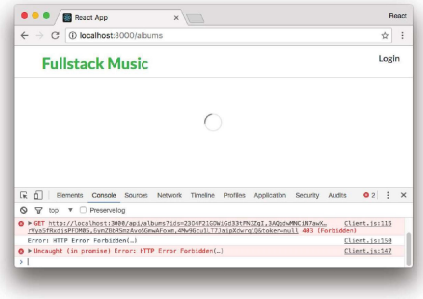
1. Load the page at `/albums`. We can see all the albums. In the top right, we see the "Logout" button.
2. Click the "Logout" button. We should be redirected to `/login`.
3. Without clicking the "Login" button, manually enter the path `/albums`.

When we visit `/albums` after logging out, we see the spinner on the page, indefinitely:



Page hangs indefinitely

Opening up the console, we see that our app is complaining that it received a `403` from the API:



Error in the console

Great. The API rejected our request because it didn't include a token. The `Logout` component did its job and removed the token.

Now, we should be able to log in and view this page again:

1. Click the "Login" button in the top right.
2. On the `/login` page, click the "Login" button.
3. We'll be redirected to `/albums`. The albums should now be visible again.

Our app is close! We're converging on a real-world authentication flow. But we just identified an improvement we need to make. When the user visits `/albums` without being logged in, the app silently fails. Instead, we should redirect the user to `/login`.

What will happen if the user visits another page, say `/albums/1DWWb`? We should redirect the user to `/login` in that instance as well. And, when the login completes, the best experience would be if we redirected the user back to where they came from (`/albums/1DWWb`).

We'll address each of these in turn.

`PrivateRoute`, a higher-order component

If the user visits a page under `/albums` and is not logged in, we want to redirect them to the login page.

Using the client library's `isLoggedIn()`, we could implement something like this in `AlbumsContainer`:


```
// In AlbumsContainer
render() {
  // Something like this would work fine
  if (!client.isLoggedIn()) {
    return (
      <Redirect to='/login' />
    );
  }
}
```

This would work fine for our purposes right now. But, as our app grows, we'd likely need many pages and their constituent components to be wrapped with this redirect.

This is where React Router's **composability** comes in handy. Because everything in React Router is a component, we can write **higher-order components** that wrap React Router's elements in custom functionality.

A higher-order component is a React component that wraps another one. This pattern is a powerful mechanism for extending or altering the functionality of a pre-existing component.

As we saw when we wrote `Route`, this component is an example of a higher-order component.

In this case, we can write a new component, `PrivateRoute`. `PrivateRoute` will be like our own custom flavor of the `Route` component. As we'll see, we'll use it to extend and focus the functionality of `Route`. We can interchange it with `Route` anywhere in our app where we want to assert that the user is logged in. Under the hood, it will use both `Route` and `Redirect` to operate.

Open up `App.js`. Let's import and use `PrivateRoute` to see what using it will look like. We'll then build the component.

Import it at the top of `App.js`:

routing/music/client/src/components-complete/App-5.js

```
import TopBar from './TopBar';
import PrivateRoute from './PrivateRoute';
import AlbumsContainer from './AlbumsContainer';
import Login from './Login';
import Logout from './Logout';
```

We want `PrivateRoute` to have the same interface as `Route`. `/albums` will be the only route that requires the user to be logged in. For `/albums`, we'll swap out `Route` for `PrivateRoute`:

routing/music/client/src/components-complete/App-5.js

```
      <Switch>
        <PrivateRoute path='/albums' component={AlbumsContainer} />
        <Route path='/login' component={Login} />
        <Route path='/logout' component={Logout} />
```

Open up `PrivateRoute.js`. A scaffold for the component already exists.

Again, higher-order components are functions that return components wrapped with new functionality. To wrap our head around this, let's consider what `PrivateRoute` would look like if all it did was return `Route`. This implementation is the same as using `Route` directly:

routing/music/client/src/components-complete/PrivateRoute-1.js

```
const PrivateRoute = (props) => (
  <Route {...props} />
);
```

In this example, `PrivateRoute` returns a `Route` component, passing along all its props to `Route`. While not useful, this version of `PrivateRoute` is the simplest implementation of a higher-order component.

Because all that `PrivateRoute` does is render a `Route` component, if you were to save `PrivateRoute.js` and reload the app, everything would be working just as before.

What we want to eventually do is have our component either render the component that `PrivateRoute` was given (the prop `component`) or redirect if the user is not logged in. Something like this:

routing/music/client/src/components-complete/PrivateRoute-2.js

```
const PrivateRoute = (props) => (
  <Route {...props} render={(props) => (
    client.isLoggedIn() ? (
      // render the component
      todo
    ) : (
      // render the redirect
      todo
    )
  )} />
);
```


We pass along all the props to `Route` as before. Except this time, we specify a `render` function. Remember, you can either set the prop `component` on `Route` or pass a function to `render`.

Inside of this render function, we would switch off of `client.isLoggedIn()`. This boolean value will tell us if we should render the component given to `PrivateRoute` or perform a redirect.

Here's one approach to doing this. First, we can use the destructuring syntax to grab our arguments:

routing/music/client/src/components-complete/PrivateRoute-3.js

```
const PrivateRoute = ({ component, ...rest }) => (
```

We grab the `component` prop then use the spread syntax to grab `...rest`, which is all the rest of the props set on `PrivateRoute`.

Next, we pass along all the props (`rest`) to `Route`:

routing/music/client/src/components-complete/PrivateRoute-3.js

```
const PrivateRoute = ({ component, ...rest }) => (
  <Route {...rest} render={(props) => (
```

If the user is logged in, we want to render the component:

routing/music/client/src/components-complete/PrivateRoute-3.js

```
  <Route {...rest} render={(props) => (
    client.isLoggedIn() ? (
      React.createElement(component, props)
    ) : (
```

Otherwise, we want to perform a redirect:

routing/music/client/src/components-complete/PrivateRoute-3.js

```
    ) : (
      <Redirect to={{
        pathname: '/login',
      }} />
    )
```

Our `PrivateRoute` component, in full:

routing/music/client/src/components-complete/PrivateRoute-3.js

```
const PrivateRoute = ({ component, ...rest }) => (
  <Route {...rest} render={(props) => (
    client.isLoggedIn() ? (
      React.createElement(component, props)
    ) : (
      <Redirect to={{
        pathname: '/login',
      }} />
    )
  )} />
);
```

Save `PrivateRoute.js`. Because we're already using the component in `App` to match the `/albums` endpoint, we're ready to test it out in the browser.

Higher-order components can be difficult to parse at first. If you're struggling with `PrivateRoute`, we recommend returning to the component later and playing around with it.

Try it out

With the app open, perform the following steps to verify things work:

1. Click the "Logout" button in the top right
2. We are redirected to `/login`
3. Now, when visiting `/albums` the app should redirect you to `/login`
4. When we login, we'll be redirected to `/albums` and this time it renders

We're almost finished tying things together. There's one last piece that we mentioned would be nice to have.

Redirect state

If the user visits a page on the site they can't access because they're not logged in, we redirect them to `/login`. When they log in, we should send them back to whichever page they came from. In order to do this, `Login` needs some way to know where the user came from.

React Router's `Redirect` allows us to set some state when we perform our redirect. That state will be available at the next location. We can have the `Redirect` in `PrivateRoute` set this state according to the user's location. We'll see how we can have `Login` read from this state to figure out where to send the user.

Open up `PrivateRoute.js`. Passing this `state` along in a `Redirect` looks like this:

routing/music/client/src/components-complete/PrivateRoute.js

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

We can set this state to whatever we want. We set the property `from` to the location of the app before the redirect. So, if the user tried to access `/albums/1DWWb4Q39mp1T3NgyscowF`, `state.from` would be set to this value.

Now, let's use this state in `Login.js` to determine where to redirect the user after they log in.

The `state` will be available to us here under the `location` prop that `Route` provides. Let's make a new class function on our `Login` component we'll call `redirectPath()` that reads this state:

routing/music/client/src/components-complete/Login.js

```
  redirectPath = () => {
    const locationState = this.props.location.state;
    const pathname = (
      locationState && locationState.from && locationState.from.pathname
    );
    return pathname || '/albums';
  };

  render() {
```

This function reads the location's `state` variable. If it sees a `from` property, it will return that string. Otherwise, it will default to `/albums`.

Now we can use it in our `Redirect`:

routing/music/client/src/components-complete/Login.js

```
    if (this.state.shouldRedirect) {
      return (
        <Redirect to={this.redirectPath()} />
```

Try it out

Now that we're including state in our redirect, logged-out users will be sent to whichever page they were trying to access after they log in.

To test this out, in the browser we:

1. Visit one of the albums.
2. Copy the full URL (e.g. `http://localhost:3000/albums/23O4F21GDWiGd33tFN3ZgI`).
3. Click "Logout."
4. Paste the full URL and hit enter (we can also hit the back button).
5. Our browser is trying to access a page protected by `PrivateRoute`. We're redirected to `/login`.
6. Click the "Login" button.
7. After the login completes, instead of being redirected to `/albums` we're redirected to the location we were trying to access.

Recap

In this chapter, we saw how we can use components from React Router to provide fast, JavaScript-powered navigation around our web apps. We can prevent the user's browser from doing full page loads when navigating around our site. We can build user-friendly URLs that are shareable. And the added complexity to our application is minimal.

React Router's declarative, component-driven paradigm is unique in the routing landscape. While it often means that we have to re-think how to approach routing solutions, the reward is that we're working with familiar React components. This limits the surface area of React Router's API and minimizes any "magic."

Further Reading

The React Router docs[81] contain several focused examples on a variety of React Router's concepts. At the time of writing, examples included common patterns like query params and ambiguous route matching. All those examples build on the foundations established in this chapter.

[81] https://reacttraining.com/react-router/