Philosophy

This guide's purpose is to explain the mental model to have when using React Router. We call it "Dynamic Routing", which is quite different from the "Static Routing" you're probably more familiar with.

Static Routing

If you've used Rails, Express, Ember, Angular etc. you've used static routing. In these frameworks, you declare your routes as part of your app's initialization before any rendering takes place. React Router pre-v4 was also static (mostly). Let's take a look at how to configure routes in express:

```
app.get('/', handleIndex)
app.get('/invoices', handleInvoices)
app.get('/invoices/:id', handleInvoice)
app.get('/invoices/:id/edit', handleInvoiceEdit)

app.listen()
```

Note how the routes are declared before the app listens. The client side routers we've used are similar. In Angular you declare your routes up front and then import them to the top-level `AppModule` before rendering:

```
const appRoutes: Routes = [
  { path: 'crisis-center',
    component: CrisisListComponent
  },
  { path: 'hero/:id',
    component: HeroDetailComponent
  },
  { path: 'heroes',
    component: HeroListComponent,
    data: { title: 'Heroes List' }
  },
  { path: '',
    redirectTo: '/heroes',
    pathMatch: 'full'
  },
  { path: '**',
    component: PageNotFoundComponent
  }
];

@NgModule({
  imports: [
    RouterModule.forRoot(appRoutes)
  ]
})

export class AppModule { }
```

Ember has a conventional `routes.js` file that the build reads and imports into the application for you. Again, this happens before your app renders.

```
Router.map(function() {
  this.route('about');
  this.route('contact');
  this.route('rentals', function() {
    this.route('show', { path: '/:rental_id' });
  });
});
```

```
export default Router
```

Though the APIs are different, they all share the model of "static routes". React Router also followed that lead up until v4.

To be successful with React Router, you need to forget all that! :O

Backstory

To be candid, we were pretty frustrated with the direction we'd taken React Router by v2. We (Michael and Ryan) felt limited by the API, recognized we were reimplementing parts of React (lifecycles, and more), and it just didn't match the mental model React has given us for composing UI.

We were walking through the hallway of a hotel just before a workshop discussing what to do about it. We asked each other: "What would it look like if we built the router using the patterns we teach in our workshops?"

It was only a matter of hours into development that we had a proof-of-concept that we knew was the future we wanted for routing. We ended up with API that wasn't "outside" of React, an API that composed, or naturally fell into place, with the rest of React. We think you'll love it.

Dynamic Routing

When we say dynamic routing, we mean routing that takes place **as your app is rendering**, not in a configuration or convention outside of a running app. That means almost everything is a component in React Router. Here's a 60 second review of the API to see how it works:

First, grab yourself a `Router` component for the environment you're targeting and render it at the top of your app.

```
// react-native
import { NativeRouter } from 'react-router-native'

// react-dom (what we'll use here)
import { BrowserRouter } from 'react-router-dom'

ReactDOM.render((
  <BrowserRouter>
    <App/>
  </BrowserRouter>
), el)
```

Next, grab the link component to link to a new location:

```
const App = () => (
  <div>
    <nav>
      <Link to="/dashboard">Dashboard</Link>
    </nav>
  </div>
)
```

Finally, render a `Route` to show some UI when the user visits `/dashboard`.

```
const App = () => (
  <div>
```

```
    <nav>
      <Link to="/dashboard">Dashboard</Link>
    </nav>
    <div>
      <Route path="/dashboard" component={Dashboard}/>
    </div>
  </div>
)
```

The `Route` will render `<Dashboard {...props}/>` where `props` are some router specific things that look like `{ match, location, history }`. If the user is **not** at `/dashboard` then the `Route` will render `null`. That's pretty much all there is to it.

Nested Routes

Lots of routers have some concept of "nested routes". If you've used versions of React Router previous to v4, you'll know it did too! When you move from a static route configuration to dynamic, rendered routes, how do you "nest routes"? Well, how do you nest a `div`?

```
const App = () => (
  <BrowserRouter>
    {/* here's a div */}
    <div>
      {/* here's a Route */}
      <Route path="/tacos" component={Tacos}/>
    </div>
  </BrowserRouter>
)

// when the url matches `/tacos` this component renders
const Tacos  = ({ match }) => (
  // here's a nested div
  <div>
    {/* here's a nested Route,
        match.url helps us make a relative path */}
    <Route
      path={match.url + '/carnitas'}
      component={Carnitas}
    />
  </div>
)
```

See how the router has no "nesting" API? `Route` is just a component, just like `div`. So to nest a `Route` or a `div`, you just ... do it.

Let's get trickier.

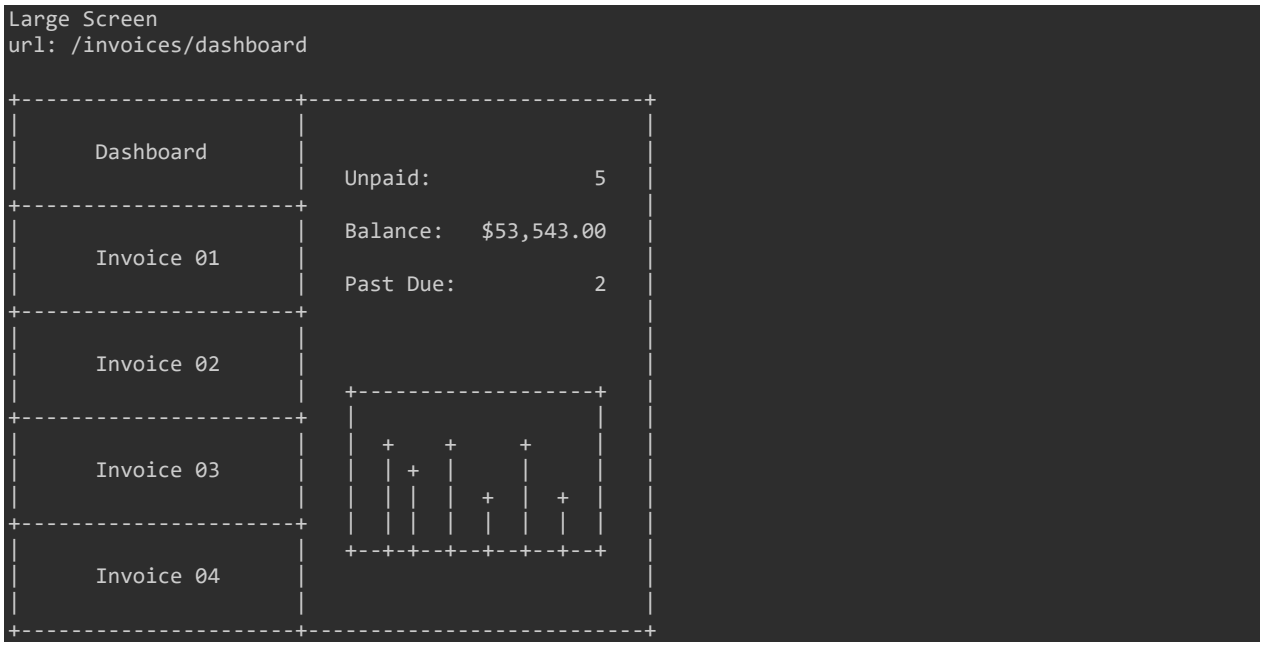
Responsive Routes

Consider a user navigates to `/invoices`. Your app is adaptive to different screen sizes, they have a narrow viewport, and so you only show them the list of invoices and a link to the invoice dashboard. They can navigate deeper from there.

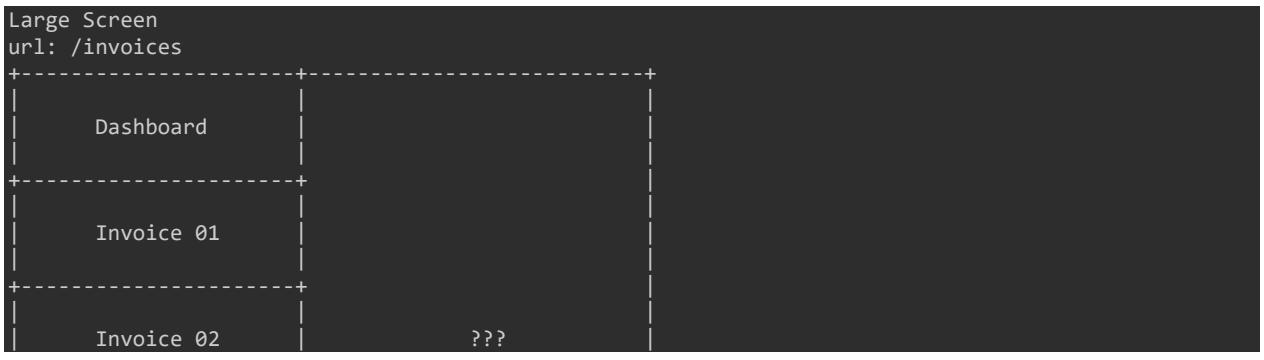
```
Small Screen
url: /invoices

+---------------------+
|                     |
|      Dashboard      |
```

On a larger screen we'd like to show a master-detail view where the navigation is on the left and the dashboard or specific invoices show up on the right.



Now pause for a minute and think about the `/invoices` url for both screen sizes. Is it even a valid route for a large screen? What should we put on the right side?



```
|                      |                           |
+----------------------+                           |
|                      |                           |
|      Invoice 03      |                           |
|                      |                           |
+----------------------+                           |
|                      |                           |
|      Invoice 04      |                           |
|                      |                           |
+----------------------+---------------------------+
```

On a large screen, `/invoices` isn't a valid route, but on a small screen it is! To make things more interesting, consider somebody with a giant phone. They could be looking at `/invoices` in portrait orientation and then rotate their phone to landscape. Suddenly, we have enough room to show the master-detail UI, so you ought to redirect right then!

React Router's previous versions' static routes didn't really have a composable answer for this. When routing is dynamic, however, you can declaratively compose this functionality. If you start thinking about routing as UI, not as static configuration, your intuition will lead you to the following code:

```
const App = () => (
  <AppLayout>
    <Route path="/invoices" component={Invoices}/>
  </AppLayout>
)

const Invoices = () => (
  <Layout>

    {/* always show the nav */}
    <InvoicesNav/>

    <Media query={PRETTY_SMALL}>
      {screenIsSmall => screenIsSmall
        // small screen has no redirect
        ? <Switch>
            <Route exact path="/invoices/dashboard" component={Dashboard}/>
            <Route path="/invoices/:id" component={Invoice}/>
          </Switch>
        // large screen does!
        : <Switch>
            <Route exact path="/invoices/dashboard" component={Dashboard}/>
            <Route path="/invoices/:id" component={Invoice}/>
            <Redirect from="/invoices" to="/invoices/dashboard"/>
          </Switch>
      }
    </Media>
  </Layout>
)
```

As the user rotates their phone from portrait to landscape, this code will automatically redirect them to the dashboard. *The set of valid routes change depending on the dynamic nature of a mobile device in a user's hands*.

This is just one example. There are many others we could discuss but we'll sum it up with this advice: To get your intuition in line with React Router's, think about components, not static routes. Think about how to solve the problem with React's declarative composability because nearly every "React Router question" is probably a "React question".

Basic Components

There are three types of components in React Router: router components, route matching components, and navigation components.

All of the components that you use in a web application should be imported from `react-router-dom`.

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

At the core of every React Router application should be a router component. For web projects, `react-router-dom` provides `<BrowserRouter>` and `<HashRouter>`routers. Both of these will create a specialized `history` object for you. Generally speaking, you should use a `<BrowserRouter>` if you have a server that responds to requests and a `<HashRouter>` if you are using a static file server.

```
import { BrowserRouter } from 'react-router-dom'
ReactDOM.render((
  <BrowserRouter>
    <App/>
  </BrowserRouter>
), holder)
```

There are two route matching components: `<Route>` and `<Switch>`.

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

Route matching is done by comparing a `<Route>`'s `path` prop to the current location's `pathname`. When a `<Route>` matches it will render its content and when it does not match, it will render `null`. A `<Route>` with no path will always match.

```
// when location = { pathname: '/about' }
<Route path='/about' component={About}/> // renders <About/>
<Route path='/contact' component={Contact}/> // renders null
<Route component={Always}/> // renders <Always/>
```

You can include a `<Route>` anywhere that you want to render content based on the location. It will often make sense to list a number of possible `<Route>`s next to each other. The `<Switch>` component is used to group `<Route>`s together.

```
<Switch>
  <Route exact path='/' component={Home}/>
  <Route path='/about' component={About}/>
  <Route path='/contact' component={Contact}/>
</Switch>
```

The `<Switch>` is not required for grouping `<Route>`s, but it can be quite useful. A `<Switch>` will iterate over all of its children `<Route>` elements and only render the first one that matches the current location. This helps when multiple route's paths match the same pathname, when animating transitions between routes, and in identifying when no routes match the current location (so that you can render a "404" component).

```
<Switch>
  <Route exact path='/' component={Home}/>
  <Route path='/about' component={About}/>
  <Route path='/contact' component={Contact}/>
```

```
  {/* when none of the above match, <NoMatch> will be rendered */}
  <Route component={NoMatch}/>
</Switch>
```

You have three prop choices for how you render a component for a given `<Route>`: `component`, `render`, and `children`. You can check the out the `<Route>`documentation for more information on each one, but here we'll focus on `component` and `render` because those are the two you will almost always use.

`component` should be used when you have an existing component (either a `React.Component` or a stateless functional component) that you want to render. `render`, which takes an inline function, should only be used when you have to pass in-scope variables to the component you want to render. You should **not**use the `component` prop with an inline function to pass in-scope variables because you will get undesired component unmounts/remounts.

```
const Home = () => <div>Home</div>

const App = () => {
  const someVariable = true;

  return (
    <Switch>
      {/* these are good */}
      <Route exact path='/' component={Home} />
      <Route
        path='/about'
        render={(props) => <About {...props} extra={someVariable} />}
      />
      {/* do not do this */}
      <Route
        path='/contact'
        component={(props) => <Contact {...props} extra={someVariable} />}
      />
    </Switch>
  )
}
```

React Router provides a `<Link>` component to create links in your application. Wherever you render a `<Link>`, an anchor (`<a>`) will be rendered in your application's HTML.

```
<Link to='/'>Home</Link>
// <a href='/'>Home</a>
```

The `<NavLink>` is a special type of `<Link>` that can style itself as "active" when its `to` prop matches the current location.

```
// location = { pathname: '/react' }
<NavLink to='/react' activeClassName='hurray'>React</NavLink>
// <a href='/react' className='hurray'>React</a>
```

Any time that you want to force navigation, you can render a `<Redirect>`. When a `<Redirect>` renders, it will navigate using its `to` prop.

Quick Start

The easiest way to get started with a React web project is with a tool called Create React App, a Facebook project with a ton of community help.

First install create-react-app if you don't already have it, and then make a new project with it.

```
npm install -g create-react-app
create-react-app demo-app
cd demo-app
```

React Router DOM is published to npm so you can install it with either `npm` or `yarn`. Create React App uses yarn, so that's what we'll use.

```
yarn add react-router-dom
# or, if you're not using yarn
npm install react-router-dom
```

Now you can copy/paste any of the examples into `src/App.js`. Here's the basic one:

```
import React from 'react'
import {
  BrowserRouter as Router,
  Route,
  Link
} from 'react-router-dom'

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

const About = () => (
  <div>
    <h2>About</h2>
  </div>
)

const Topic = ({ match }) => (
  <div>
    <h3>{match.params.topicId}</h3>
  </div>
)

const Topics = ({ match }) => (
  <div>
    <h2>Topics</h2>
    <ul>
      <li>
        <Link to={`${match.url}/rendering`}>
          Rendering with React
        </Link>
      </li>
      <li>
        <Link to={`${match.url}/components`}>
          Components
        </Link>
      </li>
```

```
      <li>
        <Link to={`${match.url}/props-v-state`}>
          Props v. State
        </Link>
      </li>
    </ul>

    <Route path={`${match.path}/:topicId`} component={Topic}/>
    <Route exact path={match.path} render={() => (
      <h3>Please select a topic.</h3>
    )}/>
  </div>
)

const BasicExample = () => (
  <Router>
    <div>
      <ul>
        <li><Link to="/">Home</Link></li>
        <li><Link to="/about">About</Link></li>
        <li><Link to="/topics">Topics</Link></li>
      </ul>

      <hr/>

      <Route exact path="/" component={Home}/>
      <Route path="/about" component={About}/>
      <Route path="/topics" component={Topics}/>
    </div>
  </Router>
)
export default BasicExample
```

Now you're ready to tinker. Happy routing!

Server Rendering

Rendering on the server is a bit different since it's all stateless. The basic idea is that we wrap the app in a stateless `<StaticRouter>` instead of a `<BrowserRouter>`. We pass in the requested url from the server so the routes can match and a `context` prop we'll discuss next.

```
// client
<BrowserRouter>
  <App/>
</BrowserRouter>

// server (not the complete story)
<StaticRouter
  location={req.url}
  context={context}
>
  <App/>
</StaticRouter>
```

When you render a `<Redirect>` on the client, the browser history changes state and we get the new screen. In a static server environment we can't change the app state. Instead, we use the `context` prop to find out what the result of rendering was. If we find a `context.url`, then we know the app redirected. This allows us to send a proper redirect from the server.

```
const context = {}
const markup = ReactDOMServer.renderToString(
  <StaticRouter
    location={req.url}
    context={context}
  >
    <App/>
  </StaticRouter>
)

if (context.url) {
  // Somewhere a `<Redirect>` was rendered
  redirect(301, context.url)
} else {
  // we're good, send the response
}
```

The router only ever adds `context.url`. But you may want some redirects to be 301 and others 302. Or maybe you'd like to send a 404 response if some specific branch of UI is rendered, or a 401 if they aren't authorized. The context prop is yours, so you can mutate it. Here's a way to distinguish between 301 and 302 redirects:

```
const RedirectWithStatus = ({ from, to, status }) => (
  <Route render={({ staticContext }) => {
    // there is no `staticContext` on the client, so
    // we need to guard against that here
    if (staticContext)
      staticContext.status = status
    return <Redirect from={from} to={to}/>
  }}/>
)

// somewhere in your app
const App = () => (
  <Switch>
```

```
    {/* some other routes */}
    <RedirectWithStatus
      status={301}
      from="/users"
      to="/profiles"
    />
    <RedirectWithStatus
      status={302}
      from="/courses"
      to="/dashboard"
    />
  </Switch>
)

// on the server
const context = {}

const markup = ReactDOMServer.renderToString(
  <StaticRouter context={context}>
    <App/>
  </StaticRouter>
)

if (context.url) {
  // can use the `context.status` that
  // we added in RedirectWithStatus
  redirect(context.status, context.url)
}
```

We can do the same thing as above. Create a component that adds some context and render it anywhere in the app to get a different status code.

```
const Status = ({ code, children }) => (
  <Route render={({ staticContext }) => {
    if (staticContext)
      staticContext.status = code
    return children
  }}/>
)
```

Now you can render a `Status` anywhere in the app that you want to add the code to `staticContext`.

```
const NotFound = () => (
  <Status code={404}>
    <div>
      <h1>Sorry, can’t find that.</h1>
    </div>
  </Status>
)

// somewhere else
<Switch>
  <Route path="/about" component={About}/>
  <Route path="/dashboard" component={Dashboard}/>
  <Route component={NotFound}/>
</Switch>
```

This isn’t a real app, but it shows all of the general pieces you’ll need to put it all together.

```
import { createServer } from 'http'
```

```
import React from 'react'
import ReactDOMServer from 'react-dom/server'
import { StaticRouter } from 'react-router'
import App from './App'

createServer((req, res) => {
  const context = {}

  const html = ReactDOMServer.renderToString(
    <StaticRouter
      location={req.url}
      context={context}
    >
      <App/>
    </StaticRouter>
  )

  if (context.url) {
    res.writeHead(301, {
      Location: context.url
    })
    res.end()
  } else {
    res.write(`
      <!doctype html>
      <div id="app">${html}</div>
    `)
    res.end()
  }
}).listen(3000)
```

And then the client:

```
import ReactDOM from 'react-dom'
import { BrowserRouter } from 'react-router-dom'
import App from './App'

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

There are so many different approaches to this, and there's no clear best practice yet, so we seek to be composable with any approach, and not prescribe or lean toward one or the other. We're confident the router can fit inside the constraints of your application.

The primary constraint is that you want to load data before you render. React Router exports the `matchPath` static function that it uses internally to match locations to routes. You can use this function on the server to help determine what your data dependencies will be before rendering.

The gist of this approach relies on a static route config used to both render your routes and match against before rendering to determine data dependencies.

```
const routes = [
  { path: '/',
    component: Root,
    loadData: () => getSomeData(),
  },
  // etc.
```

```
]
```

Then use this config to render your routes in the app:

```
import { routes } from './routes'

const App = () => (
  <Switch>
    {routes.map(route => (
      <Route {...route}/>
    ))}
  </Switch>
)
```

Then on the server you'd have something like:

```
import { matchPath } from 'react-router-dom'

// inside a request
const promises = []
// use `some` to imitate `<Switch>` behavior of selecting only
// the first to match
routes.some(route => {
  // use `matchPath` here
  const match = matchPath(req.path, route)
  if (match)
    promises.push(route.loadData(match))
  return match
})

Promise.all(promises).then(data => {
  // do something w/ the data so the client
  // can access it then render the app
})
```

And finally, the client will need to pick up the data. Again, we aren't in the business of prescribing a data loading pattern for your app, but these are the touch points you'll need to implement.

You might be interested in our React Router Config package to assist with data loading and server rendering with static route configs.

Code Splitting

One great feature of the web is that we don't have to make our visitors download the entire app before they can use it. You can think of code splitting as incrementally downloading the app. To accomplish this we'll use webpack, `babel-plugin-syntax-dynamic-import`, and `react-loadable`.

webpack has built-in support for dynamic imports; however, if you are using Babel (e.g., to compile JSX to JavaScript) then you will need to use the `babel-plugin-syntax-dynamic-import` plugin. This is a syntax-only plugin, meaning Babel won't do any additional transformations. The plugin simply allows Babel to parse dynamic imports so webpack can bundle them as a code split. Your `.babelrc` should look something like this:

```
{
  "presets": [
    "react"
```

```
  ],
  "plugins": [
    "syntax-dynamic-import"
  ]
}
```

`react-loadable` is a higher-order component for loading components with dynamic imports. It handles all sorts of edge cases automatically and makes code splitting simple! Here's an example of how to use `react-loadable`:

```
import Loadable from 'react-loadable';
import Loading from './Loading';

const LoadableComponent = Loadable({
  loader: () => import('./Dashboard'),
  loading: Loading,
})

export default class LoadableDashboard extends React.Component {
  render() {
    return <LoadableComponent />;
  }
}
```

That's all there is to it! Simply use `LoadableDashboard` (or whatever you named your component) and it will automatically be loaded and rendered when you use it in your application. The `loader` option is a function which actually loads the component, and `loading` is a placeholder component to show while the real component is loading.

Code Splitting and Server-Side Rendering

`react-loadable` includes a guide for server-side rendering. All you should need to do is include `babel-plugin-import-inspector` in your `.babelrc` and server-side rendering should just work™. Here is an example `.babelrc` file:

```
{
  "presets": [
    "react"
  ],
  "plugins": [
    "syntax-dynamic-import",
    ["import-inspector", {
      "serverSideRequirePath": true
    }]
  ]
}
```

Scroll Restoration

In earlier versions of React Router we provided out-of-the-box support for scroll restoration and people have been asking for it ever since. Hopefully this document helps you get what you need out of the scroll bar and routing!

Browsers are starting to handle scroll restoration with `history.pushState` on their own in the same manner they handle it with normal browser navigation. It already works in chrome and it's really great. Here's the Scroll Restoration Spec.

Because browsers are starting to handle the "default case" and apps have varying scrolling needs (like this website!), we don't ship with default scroll management. This guide should help you implement whatever scrolling needs you have.

Scroll to top

Most of the time all you need is to "scroll to the top" because you have a long content page, that when navigated to, stays scrolled down. This is straightforward to handle with a `<ScrollToTop>` component that will scroll the window up on every navigation, make sure to wrap it in `withRouter` to give it access to the router's props:

```
class ScrollToTop extends Component {
  componentDidUpdate(prevProps) {
    if (this.props.location !== prevProps.location) {
      window.scrollTo(0, 0)
    }
  }

  render() {
    return this.props.children
  }
}

export default withRouter(ScrollToTop)
```

Then render it at the top of your app, but below Router

```
const App = () => (
  <Router>
    <ScrollToTop>
      <App/>
    </ScrollToTop>
  </Router>
)

// or just render it bare anywhere you want, but just one :)
<ScrollToTop/>
```

If you have a tab interface connected to the router, then you probably don't want to be scrolling to the top when they switch tabs. Instead, how about a `<ScrollToTopOnMount>` in the specific places you need it?

```
class ScrollToTopOnMount extends Component {
  componentDidMount() {
    window.scrollTo(0, 0)
  }

  render() {
    return null
  }
}

class LongContent extends Component {
  render() {
    <div>
      <ScrollToTopOnMount/>
```

```
      <h1>Here is my long content page</h1>
    </div>
  }
}

// somewhere else
<Route path="/long-content" component={LongContent}/>
```

For a generic solution (and what browsers are starting to implement natively) we're talking about two things:

1. Scrolling up on navigation so you don't start a new screen scrolled to the bottom
2. Restoring scroll positions of the window and overflow elements on "back" and "forward" clicks (but not Link clicks!)

At one point we were wanting to ship a generic API. Here's what we were headed toward:

```
<Router>
  <ScrollRestoration>
    <div>
      <h1>App</h1>

      <RestoredScroll id="bunny">
        <div style={{ height: '200px', overflow: 'auto' }}>
          I will overflow
        </div>
      </RestoredScroll>
    </div>
  </ScrollRestoration>
</Router>
```

First, `ScrollRestoration` would scroll the window up on navigation. Second, it would use `location.key` to save the window scroll position *and* the scroll positions of `RestoredScroll` components to `sessionStorage`. Then, when `ScrollRestoration` or `RestoredScroll` components mount, they could look up their position from `sessionsStorage`.

What got tricky for me was defining an "opt-out" API for when I didn't want the window scroll to be managed. For example, if you have some tab navigation floating inside the content of your page you probably *don't* want to scroll to the top (the tabs might be scrolled out of view!).

When I learned that chrome manages scroll position for us now, and realized that different apps are going to have different scrolling needs, I kind of lost the belief that we needed to provide something–especially when people just want to scroll to the top (which you saw is straight-forward to add to your app on your own).

Based on this, we no longer feel strongly enough to do the work ourselves (like you we have limited time!). But, we'd love to help anybody who feels inclined to implement a generic solution. A solid solution could even live in the project. Hit us up if you get started on it :)

Redux Integration

Redux is an important part of the React ecosystem. We want to make the integration of React Router and Redux as seamless as possible for people wanting to use both.

Blocked Updates

Generally, React Router and Redux work just fine together. Occasionally though, an app can have a component that doesn't update when the location changes (child routes or active nav links don't update).

This happens if:

1. The component is connected to redux via `connect()(Comp)`.
2. The component is **not** a "route component", meaning it is not rendered like so: `<Route component={SomeConnectedThing}/>`

The problem is that Redux implements `shouldComponentUpdate` and there's no indication that anything has changed if it isn't receiving props from the router. This is straightforward to fix. Find where you `connect` your component and wrap it in `withRouter`.

```
// before
export default connect(mapStateToProps)(Something)

// after
import { withRouter } from 'react-router-dom'
export default withRouter(connect(mapStateToProps)(Something))
```

Some folks want to:

- synchronize the routing data with, and accessed from, the store
- be able to navigate by dispatching actions
- have support for time travel debugging for route changes in the Redux devtools

All of this requires deeper integration. Please note you don't need this deep integration:

- Route changes are unlikely to matter for time travel debugging.
- Rather than dispatching actions to navigate you can pass the `history`object provided to route components to your actions and navigate with it there.
- Routing data is already a prop of most of your components that care about it, whether it comes from the store or the router doesn't change your component's code.

However, we know some people feel strongly about this and so we want to provide the best deep integration possible. As of version 4 of React Router, the React Router Redux package is a part of the project. Please refer to it for deep integration.

Dealing with Update Blocking

React Router has a number of location-aware components that use the current `location` object to determine what they render. By default, the current `location`is passed implicitly to components using React's context model. When the location changes, those components should re-render using the new `location`object from the context.

React provides two approaches to optimize the rendering performance of applications:
the `shouldComponentUpdate` lifecycle method and the `PureComponent`. Both block the re-rendering of components unless the right conditions are met. Unfortunately, this means that React Router's location-aware components can become out of sync with the current location if their re-rendering was prevented.

Example of the Problem

We start out with a component that prevents updates.

```
class UpdateBlocker extends React.PureComponent {
  render() {
    return this.props.children
  }
}
```

When the `<UpdateBlocker>` is mounting, any location-aware child components will use the current `location` and `match` objects to render.

```
// location = { pathname: '/about' }
<UpdateBlocker>
  <NavLink to='/about'>About</NavLink>
  // <a href='/about' class='active'>About</a>
  <NavLink to='/faq'>F.A.Q.</NavLink>
  // <a href='/faq'>F.A.Q.</a>
</UpdateBlocker>
```

When the location changes, the `<UpdateBlocker>` does not detect any prop or state changes, so its child components will not be re-rendered.

```
// location = { pathname: '/faq' }
<UpdateBlocker>
  // the links will not re-render, so they retain their previous attributes
  <NavLink to='/about'>About</NavLink>
  // <a href='/about' class='active'>About</a>
  <NavLink to='/faq'>F.A.Q.</NavLink>
  // <a href='/faq'>F.A.Q.</a>
</UpdateBlocker>
shouldComponentUpdate
```

In order for a component that implements `shouldComponentUpdate` to know that it *should* update when the location changes, its `shouldComponentUpdate` method needs to be able to detect location changes.

If you are implementing `shouldComponentUpdate` yourself, you *could* compare the location from the current and next `context.router` objects. However, as a user, you should not have to use context directly. Instead, it would be ideal if you could compare the current and next `location` without touching the context.

Third-Party Code

You may run into issues with components not updating after a location change despite not calling `shouldComponentUpdate` yourself. This is most likely because `shouldComponentUpdate` is being called by third-party code, such as `react-redux`'s `connect` and `mobx-react`'s `observer`.

```
// react-redux
const MyConnectedComponent = connect(mapStateToProps)(MyComponent)

// mobx-react
const MyObservedComponent = observer(MyComponent)
```

With third-party code, you likely cannot even control the implementation of `shouldComponentUpdate`. Instead, you will have to structure your code to make location changes obvious to those methods.

Both `connect` and `observer` create components whose `shouldComponentUpdate`methods do a shallow comparison of their current `props` and their next `props`. Those components will only re-render when at least one prop has changed. This means that in order to ensure they update when the location changes, they will need to be given a prop that changes when the location changes.

`PureComponent`

React's `PureComponent` does not implement `shouldComponentUpdate`, but it takes a similar approach to preventing updates. When a "pure" component updates, it will do a shallow comparison of its current `props` and `state` to the next `props`and `state`. If the comparison does not detect any differences, the component will not update. Like with `shouldComponentUpdate`, that means that in order to force a "pure" component to update when the location changes, it needs to have a prop or state that has changed.

The Solution

Quick Solution

If you are running into this issue while using a higher-order component like `connect` (from react-redux) or `observer` (from Mobx), you can just wrap that component in a `withRouter` to remove the blocked updates.

```
// redux before
const MyConnectedComponent = connect(mapStateToProps)(MyComponent)
// redux after
const MyConnectedComponent = withRouter(connect(mapStateToProps)(MyComponent))

// mobx before
const MyConnectedComponent = observer(MyComponent)
// mobx after
const MyConnectedComponent = withRouter(observer(MyComponent))
```

This is not the most efficient solution, but will prevent the blocked updates issue. For more info regarding this solution, read the Redux guide. To understand why this is not the most optimal solution, read this thread.

Recommended Solution

The key to avoiding blocked re-renders after location changes is to pass the blocking component the `location` object as a prop. This will be different whenever the location changes, so comparisons will detect that the current and next location are different.

```
// location = { pathname: '/about' }
```

```
<UpdateBlocker location={location}>
  <NavLink to='/about'>About</NavLink>
  // <a href='/about' class='active'>About</a>
  <NavLink to='/faq'>F.A.Q.</NavLink>
  // <a href='/faq'>F.A.Q.</a>
</UpdateBlocker>

// location = { pathname: '/faq' }
<UpdateBlocker location={location}>
  <NavLink to='/about'>About</NavLink>
  // <a href='/about'>About</a>
  <NavLink to='/faq'>F.A.Q.</NavLink>
  // <a href='/faq' class='active'>F.A.Q.</a>
</UpdateBlocker>
```

Getting the location

In order to pass the current `location` object as a prop to a component, you must have access to it. The primary way that a component can get access to the `location` is via a `<Route>` component. When a `<Route>` matches (or always if you are using the `children` prop), it passes the current `location` to the child element it renders.

```
<Route path='/here' component={Here}/>
const Here = (props) => {
  // props.location = { pathname: '/here', ... }
  return <div>You are here</div>
}

<Route path='/there' render={(props) => {
  // props.location = { pathname: '/there', ... }
  return <div>You are there</div>
}}/>

<Route path='/everywhere' children={(props) => {
  // props.location = { pathname: '/everywhere', ... }
  return <div>You are everywhere</div>
}}/>
```

This means that given a component that blocks updates, you can easily pass it the `location` as a prop in the following ways:

```
// the Blocker is a "pure" component, so it will only
// update when it receives new props
class Blocker extends React.PureComponent {
  render() {
    <div>
      <NavLink to='/oz'>Oz</NavLink>
      <NavLink to='/kansas'>Kansas</NavLink>
    </div>
  }
}
```

1. A component rendered directly by a `<Route>` does not have to worry about blocked updates because it has the `location` injected as a prop.

```
// The <Blocker>'s location prop will change whenever
// the location changes
<Route path='/:place' component={Blocker}/>
```

2. A component rendered directly by a `<Route>` can pass that location prop to any child elements it creates.

```
<Route path='/parent' component={Parent} />

const Parent = (props) => {
  // <Parent> receives the location as a prop. Any child
  // element it creates can be passed the location.
  return (
    <SomeComponent>
      <Blocker location={props.location} />
    </SomeComponent>
  )
}
```

What happens when the component isn't being rendered by a `<Route>` and the component rendering it does not have the `location` in its variable scope? There are two approaches that you can take to automatically inject the `location` as a prop of your component.

1. Render a pathless `<Route>`. While `<Route>`s are typically used for matching a specific path, a pathless `<Route>` will always match, so it will always render its component.

```
// pathless <Route> = <Blocker> will always be rendered
const MyComponent= () => (
  <SomeComponent>
    <Route component={Blocker} />
  </SomeComponent>
)
```

2. You can wrap a component with the `withRouter` higher-order component and it will be given the current `location` as one of its props.

API References

<BrowserRouter>

A `<Router>` that uses the HTML5 history API (`pushState`, `replaceState` and the `popstate` event) to keep your UI in sync with the URL.

```
import { BrowserRouter } from 'react-router-dom'

<BrowserRouter
  basename={optionalString}
  forceRefresh={optionalBool}
  getUserConfirmation={optionalFunc}
  keyLength={optionalNumber}
>
  <App/>
</BrowserRouter>
```

basename: string

The base URL for all locations. If your app is served from a sub-directory on your server, you'll want to set this to the sub-directory. A properly formatted basename should have a leading slash, but no trailing slash.

```
<BrowserRouter basename="/calendar"/>
<Link to="/today"/> // renders <a href="/calendar/today">
```

getUserConfirmation: func

A function to use to confirm navigation. Defaults to using `window.confirm`.

```
// this is the default behavior
const getConfirmation = (message, callback) => {
  const allowTransition = window.confirm(message)
  callback(allowTransition)
}

<BrowserRouter getUserConfirmation={getConfirmation}/>
```

forceRefresh: bool

If `true` the router will use full page refreshes on page navigation. You probably only want this in browsers that don't support the HTML5 history API.

```
const supportsHistory = 'pushState' in window.history
<BrowserRouter forceRefresh={!supportsHistory}/>
```

keyLength: number

The length of `location.key`. Defaults to 6.

```
<BrowserRouter keyLength={12}/>
```

children: node

A single child element to render.

<HashRouter>

A `<Router>` that uses the hash portion of the URL (i.e. `window.location.hash`) to keep your UI in sync with the URL.

IMPORTANT NOTE: Hash history does not support `location.key` or `location.state`. In previous versions we attempted to shim the behavior but there were edge-cases we couldn't solve. Any code or plugin that needs this behavior won't work. As this technique is only intended to support legacy browsers, we encourage you to configure your server to work with `<BrowserHistory>` instead.

```
import { HashRouter } from 'react-router-dom'

<HashRouter>
  <App/>
</HashRouter>
```

basename: string

The base URL for all locations. A properly formatted basename should have a leading slash, but no trailing slash.

```
<HashRouter basename="/calendar"/>
<Link to="/today"/> // renders <a href="#/calendar/today">
```

getUserConfirmation: func

A function to use to confirm navigation. Defaults to using `window.confirm`.

```
// this is the default behavior
const getConfirmation = (message, callback) => {
  const allowTransition = window.confirm(message)
  callback(allowTransition)
}

<HashRouter getUserConfirmation={getConfirmation}/>
```

hashType: string

The type of encoding to use for `window.location.hash`. Available values are:

- `"slash"` - Creates hashes like `#/` and `#/sunshine/lollipops`
- `"noslash"` - Creates hashes like `#` and `#sunshine/lollipops`
- `"hashbang"` - Creates "ajax crawlable" (deprecated by Google) hashes like `#!/` and `#!/sunshine/lollipops`

Defaults to `"slash"`.

children: node

A single child element to render.

<Link>

Provides declarative, accessible navigation around your application.

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

```
<Link to="/about">About</Link>
```

to: string

A string representation of the location to link to, created by concatenating the location's pathname, search, and hash properties.

```
<Link to='/courses?sort=name'/>
```

to: object

An object that can have any of the following properties:

- `pathname`: A string representing the path to link to.
- `search`: A string represenation of query parameters.
- `hash`: A hash to put in the URL, e.g. `#a-hash`.
- `state`: State to persist to the `location`.

```
<Link to={{
  pathname: '/courses',
  search: '?sort=name',
  hash: '#the-hash',
  state: { fromDashboard: true }
}}/>
```

replace: bool

When `true`, clicking the link will replace the current entry in the history stack instead of adding a new one.

```
<Link to="/courses" replace />
```

innerRef: function

Allows access to the underlying `ref` of the component

```
const refCallback = node => {
  // `node` refers to the mounted DOM element or null when unmounted
}

<Link to="/" innerRef={refCallback} />
```

others

You can also pass props you'd like to be on the `<a>` such as a `title`, `id`, `className`, etc.

<NavLink>

A special version of the `<Link>` that will add styling attributes to the rendered element when it matches the current URL.

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

<NavLink to="/about">About</NavLink>
```

activeClassName: string

The class to give the element when it is active. The default given class is `active`. This will be joined with the `className` prop.

```
<NavLink
  to="/faq"
  activeClassName="selected"
>FAQs</NavLink>
```

activeStyle: object

The styles to apply to the element when it is active.

```
<NavLink
  to="/faq"
  activeStyle={{
    fontWeight: 'bold',
    color: 'red'
   }}
>FAQs</NavLink>
```

exact: bool

When `true`, the active class/style will only be applied if the location is matched exactly.

```
<NavLink
  exact
  to="/profile"
>Profile</NavLink>
```

strict: bool

When `true`, the trailing slash on a location's `pathname` will be taken into consideration when determining if the location matches the current URL. See the `<Route strict>` documentation for more information.

```
<NavLink
  strict
  to="/events/"
>Events</NavLink>
```

isActive: func

A function to add extra logic for determining whether the link is active. This should be used if you want to do more than verify that the link's pathname matches the current URL's `pathname`.

```
// only consider an event active if its event id is an odd number
const oddEvent = (match, location) => {
  if (!match) {
    return false
  }
  const eventID = parseInt(match.params.eventID)
  return !isNaN(eventID) && eventID % 2 === 1
}

<NavLink
  to="/events/123"
  isActive={oddEvent}
>Event 123</NavLink>
```

location: object

The `isActive` compares the current history location (usually the current browser URL). To compare to a different location, a `location` can be passed.

<Prompt>

Re-exported from core `Prompt`

<MemoryRouter>

A `<Router>` that keeps the history of your "URL" in memory (does not read or write to the address bar). Useful in tests and non-browser environments like React Native.

```
import { MemoryRouter } from 'react-router'

<MemoryRouter>
  <App/>
</MemoryRouter>
```

initialEntries: array

An array of `location`s in the history stack. These may be full-blown location objects with `{ pathname, search, hash, state }` or simple string URLs.

```
<MemoryRouter
  initialEntries={[ '/one', '/two', { pathname: '/three' } ]}
  initialIndex={1}
>
  <App/>
</MemoryRouter>
```

initialIndex: number

The initial location's index in the array of `initialEntries`.

getUserConfirmation: func

A function to use to confirm navigation. You must use this option when using `<MemoryRouter>` directly with a `<Prompt>`.

keyLength: number

The length of `location.key`. Defaults to 6.

```
<MemoryRouter keyLength={12}/>
```

children: node

A single child element to render.

<Redirect>

Rendering a `<Redirect>` will navigate to a new location. The new location will override the current location in the history stack, like server-side redirects (HTTP 3xx) do.

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

<Route exact path="/" render={() => (
  loggedIn ? (
    <Redirect to="/dashboard"/>
  ) : (
    <PublicHomePage/>
  )
)}/>
```

to: string

The URL to redirect to. Any valid URL path that `path-to-regexp` understands. All URL parameters that are used in `to` must be covered by `from`.

```
<Redirect to="/somewhere/else"/>
```

to: object

A location to redirect to. `pathname` can be any valid URL path that `path-to-regexp` understands.

```
<Redirect to={{
  pathname: '/login',
  search: '?utm=your+face',
  state: { referrer: currentLocation }
}}/>
```

push: bool

When `true`, redirecting will push a new entry onto the history instead of replacing the current one.

```
<Redirect push to="/somewhere/else"/>
```

from: string

A pathname to redirect from. Any valid URL path that `path-to-regexp`understands. All matched URL parameters are provided to the pattern in `to`. Must contain all parameters that are used in `to`. Additional parameters not used by `to`are ignored.

This can only be used to match a location when rendering a `<Redirect>` inside of a `<Switch>`. See `<Switch children>` for more details.

```
<Switch>
  <Redirect from='/old-path' to='/new-path'/>
  <Route path='/new-path' component={Place}/>
</Switch>
// Redirect with matched parameters
<Switch>
  <Redirect from='/users/:id' to='/users/profile/:id'/>
  <Route path='/users/profile/:id' component={Profile}/>
</Switch>
```

exact: bool

Match `from` exactly; equivalent to Route.exact.

strict: bool

Match `from` strictly; equivalent to Route.strict.

<Route>

The Route component is perhaps the most important component in React Router to understand and learn to use well. Its most basic responsibility is to render some UI when a location matches the route's `path`.

Consider the following code:

```
import { BrowserRouter as Router, Route } from 'react-router-dom'

<Router>
  <div>
    <Route exact path="/" component={Home}/>
    <Route path="/news" component={NewsFeed}/>
  </div>
</Router>
```

If the location of the app is `/` then the UI hierarchy will be something like:

```
<div>
  <Home/>
  <!-- react-empty: 2 -->
</div>
```

And if the location of the app is `/news` then the UI hierarchy will be:

```
<div>
  <!-- react-empty: 1 -->
  <NewsFeed/>
</div>
```

The "react-empty" comments are just implementation details of React's `null`rendering. But for our purposes, it is instructive. A Route is always technically "rendered" even though its rendering `null`. As soon as the app location matches the route's path, your component will be rendered.

Route render methods

There are 3 ways to render something with a `<Route>`:

- `<Route component>`
- `<Route render>`
- `<Route children>`

Each is useful in different circumstances. You should use only one of these props on a given `<Route>`. See their explanations below to understand why you have 3 options. Most of the time you'll use `component`.

Route props

All three render methods will be passed the same three route props

- match
- location
- history

component

A React component to render only when the location matches. It will be rendered with route props.

```
<Route path="/user/:username" component={User}/>

const User = ({ match }) => {
  return <h1>Hello {match.params.username}!</h1>
}
```

When you use `component` (instead of `render` or `children`, below) the router uses `React.createElement` to create a new React element from the given component. That means if you provide an inline function to the `component` prop, you would create a new component every render. This results in the existing component unmounting and the new component mounting instead of just updating the existing component. When using an inline function for inline rendering, use the `render` or the `children` prop (below).

render: func

This allows for convenient inline rendering and wrapping without the undesired remounting explained above.

Instead of having a new React element created for you using the `component` prop, you can pass in a function to be called when the location matches. The `render`prop receives all the same route props as the `component` render prop.

```
// convenient inline rendering
<Route path="/home" render={() => <div>Home</div>}/>

// wrapping/composing
const FadingRoute = ({ component: Component, ...rest }) => (
  <Route {...rest} render={props => (
    <FadeIn>
      <Component {...props}/>
    </FadeIn>
  )}/>
)

<FadingRoute path="/cool" component={Something}/>
```

Warning: `<Route component>` takes precedence over `<Route render>` so don't use both in the same `<Route>`.

children: func

Sometimes you need to render whether the path matches the location or not. In these cases, you can use the function `children` prop. It works exactly like `render`except that it gets called whether there is a match or not.

The `children` render prop receives all the same route props as the `component`and `render` methods, except when a route fails to match the URL, then `match` is `null`. This allows you to dynamically adjust your UI based on whether or not the route matches. Here we're adding an `active` class if the route matches

```
<ul>
  <ListItemLink to="/somewhere"/>
  <ListItemLink to="/somewhere-else"/>
</ul>

const ListItemLink = ({ to, ...rest }) => (
  <Route path={to} children={({ match }) => (
    <li className={match ? 'active' : ''}>
      <Link to={to} {...rest}/>
    </li>
  )}/>
)
```

This could also be useful for animations:

```
<Route children={({ match, ...rest }) => (
  {/* Animate will always render, so you can use lifecycles
      to animate its child in and out */}
  <Animate>
    {match && <Something {...rest}/>}
  </Animate>
)}/>
```

Warning: Both `<Route component>` and `<Route render>` take precedence over `<Route children>` so don't use more than one in the same `<Route>`.

path: string

Any valid URL path that `path-to-regexp` understands.

```
<Route path="/users/:id" component={User}/>
```

Routes without a `path` *always* match.

exact: bool

When `true`, will only match if the path matches the `location.pathname` *exactly*.

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

path	*location.pathname*	*exact*	*matches?*
`/one`	`/one/two`	`true`	no
`/one`	`/one/two`	`false`	yes

strict: bool

When `true`, a `path` that has a trailing slash will only match a `location.pathname`with a trailing slash. This has no effect when there are additional URL segments in the `location.pathname`.

```
<Route strict path="/one/" component={About}/>
```

path	location.pathname	matches?
/one/	/one	no
/one/	/one/	yes
/one/	/one/two	yes

Warning: `strict` can be used to enforce that a `location.pathname` has no trailing slash, but in order to do this both `strict` and `exact` must be `true`.

```
<Route exact strict path="/one" component={About}/>
```

path	location.pathname	matches?
/one	/one	yes
/one	/one/	no
/one	/one/two	no

location: object

A `<Route>` element tries to match its `path` to the current history location (usually the current browser URL). However, a `location` with a different `pathname` can also be passed for matching.

This is useful in cases when you need to match a `<Route>` to a location other than the current history location, as shown in the Animated Transitions example.

If a `<Route>` element is wrapped in a `<Switch>` and matches the location passed to the `<Switch>` (or the current history location), then the `location` prop passed to `<Route>` will be overridden by the one used by the `<Switch>` (given here).

sensitive: bool

When `true`, will match if the path is **case sensitive**.

```
<Route sensitive path="/one" component={About}/>
```

path	location.pathname	sensitive	matches?
/one	/one	true	yes
/One	/one	false	no

<Router>

The common low-level interface for all router components. Typically apps will use one of the high-level routers instead:

- `<BrowserRouter>`
- `<HashRouter>`
- `<MemoryRouter>`
- `<NativeRouter>`
- `<StaticRouter>`

The most common use-case for using the low-level `<Router>` is to synchronize a custom history with a state management lib like Redux or Mobx. Note that this is not required to use state management libs alongside React Router, it's only for deep integration.

```
import { Router } from 'react-router'
import createBrowserHistory from 'history/createBrowserHistory'

const history = createBrowserHistory()

<Router history={history}>
  <App/>
</Router>
```

history: object

A `history` object to use for navigation.

```
import createBrowserHistory from 'history/createBrowserHistory'

const customHistory = createBrowserHistory()
<Router history={customHistory}/>
```

children: node

A single child element to render.

```
<Router>
  <App/>
</Router>
```

<StaticRouter>

A `<Router>` that never changes location.

This can be useful in server-side rendering scenarios when the user isn't actually clicking around, so the location never actually changes. Hence, the name: static. It's also useful in simple tests when you just need to plug in a location and make assertions on the render output.

Here's an example node server that sends a 302 status code for `<Redirect>`s and regular HTML for other requests:

```
import { createServer } from 'http'
import React from 'react'
import ReactDOMServer from 'react-dom/server'
import { StaticRouter } from 'react-router'

createServer((req, res) => {

  // This context object contains the results of the render
  const context = {}

  const html = ReactDOMServer.renderToString(
```

```
    <StaticRouter location={req.url} context={context}>
      <App/>
    </StaticRouter>
  )

  // context.url will contain the URL to redirect to if a <Redirect> was used
  if (context.url) {
    res.writeHead(302, {
      Location: context.url
    })
    res.end()
  } else {
    res.write(html)
    res.end()
  }
}).listen(3000)
```

basename: string

The base URL for all locations. A properly formatted basename should have a leading slash, but no trailing slash.

```
<StaticRouter basename="/calendar">
  <Link to="/today"/> // renders <a href="/calendar/today">
</StaticRouter>
```

location: string

The URL the server received, probably `req.url` on a node server.

```
<StaticRouter location={req.url}>
  <App/>
</StaticRouter>
```

location: object

A location object shaped like `{ pathname, search, hash, state }`

```
<StaticRouter location={{ pathname: '/bubblegum' }}>
  <App/>
</StaticRouter>
```

context: object

A plain JavaScript object. During the render, components can add properties to the object to store information about the render.

```
const context = {}
<StaticRouter context={context}>
  <App />
</StaticRouter>
```

When a `<Route>` matches, it will pass the context object to the component it renders as the `staticContext` prop. Check out the Server Rendering guide for more information on how to do this yourself.

After the render, these properties can be used to to configure the server's response.

```
if(context.status === '404') {
  // ...
}
```

`children: node`

A single child element to render.

`<Switch>`

Renders the first child `<Route>` or `<Redirect>` that matches the location.

How is this different than just using a bunch of `<Route>`s?

`<Switch>` is unique in that it renders a route *exclusively*. In contrast, every `<Route>` that matches the location renders *inclusively*. Consider this code:

```
<Route path="/about" component={About}/>
<Route path="/:user" component={User}/>
<Route component={NoMatch}/>
```

If the URL is `/about`, then `<About>`, `<User>`, and `<NoMatch>` will all render because they all match the path. This is by design, allowing us to compose `<Route>`s into our apps in many ways, like sidebars and breadcrumbs, bootstrap tabs, etc.

Occasionally, however, we want to pick only one `<Route>` to render. If we're at `/about` we don't want to also match `/:user` (or show our "404" page). Here's how to do it with `Switch`:

```
import { Switch, Route } from 'react-router'

<Switch>
  <Route exact path="/" component={Home}/>
  <Route path="/about" component={About}/>
  <Route path="/:user" component={User}/>
  <Route component={NoMatch}/>
</Switch>
```

Now, if we're at `/about`, `<Switch>` will start looking for a matching `<Route>`. `<Route path="/about"/>` will match and `<Switch>` will stop looking for matches and render `<About>`. Similarly, if we're at `/michael` then `<User>` will render.

This is also useful for animated transitions since the matched `<Route>` is rendered in the same position as the previous one.

```
<Fade>
  <Switch>
    {/* there will only ever be one child here */}
    <Route/>
    <Route/>
  </Switch>
</Fade>

<Fade>
  <Route/>
  <Route/>
  {/* there will always be two children here,
      one might render null though, making transitions
      a bit more cumbersome to work out */}
</Fade>
```

`location: object`

A `location` object to be used for matching children elements instead of the current history location (usually the current browser URL).

`children: node`

All children of a `<Switch>` should be `<Route>` or `<Redirect>` elements. Only the first child to match the current location will be rendered.

`<Route>` elements are matched using their `path` prop and `<Redirect>` elements are matched using their `from` prop. A `<Route>` with no `path` prop or a `<Redirect>`with no `from` prop will always match the current location.

When you include a `<Redirect>` in a `<Switch>`, it can use any of the `<Route>`'s location matching props: `path`, `exact`, and `strict`. `from` is just an alias for the `path` prop.

If a `location` prop is given to the `<Switch>`, it will override the `location` prop on the matching child element.

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

  <Route path="/users" component={Users}/>
  <Redirect from="/accounts" to="/users"/>

  <Route component={NoMatch}/>
</Switch>
```

history

The term “history” and "`history` object" in this documentation refers to the `history` package, which is one of only 2 major dependencies of React Router (besides React itself), and which provides several different implementations for managing session history in JavaScript in various environments.

The following terms are also used:

- “browser history” - A DOM-specific implementation, useful in web browsers that support the HTML5 history API
- “hash history” - A DOM-specific implementation for legacy web browsers
- “memory history” - An in-memory history implementation, useful in testing and non-DOM environments like React Native

`history` objects typically have the following properties and methods:

- `length` - (number) The number of entries in the history stack
- `action` - (string) The current action (`PUSH`, `REPLACE`, or `POP`)
- `location` - (object) The current location. May have the following properties:
    - `pathname` - (string) The path of the URL
    - `search` - (string) The URL query string

    - `hash` - (string) The URL hash fragment
    - `state` - (object) location-specific state that was provided to e.g. `push(path, state)` when this location was pushed onto the stack. Only available in browser and memory history.
- `push(path, [state])` - (function) Pushes a new entry onto the history stack
- `replace(path, [state])` - (function) Replaces the current entry on the history stack
- `go(n)` - (function) Moves the pointer in the history stack by `n` entries
- `goBack()` - (function) Equivalent to `go(-1)`
- `goForward()` - (function) Equivalent to `go(1)`
- `block(prompt)` - (function) Prevents navigation (see the history docs)

history is mutable

The history object is mutable. Therefore it is recommended to access the `location` from the render props of `<Route>`, not from `history.location`. This ensures your assumptions about React are correct in lifecycle hooks. For example:

```
class Comp extends React.Component {
  componentWillReceiveProps(nextProps) {
    // will be true
    const locationChanged = nextProps.location !== this.props.location

    // INCORRECT, will *always* be false because history is mutable.
    const locationChanged = nextProps.history.location !== this.props.history.location
  }
}

<Route component={Comp}/>
```

Additional properties may also be present depending on the implementation you're using. Please refer to the history documentation for more details.

location

Locations represent where the app is now, where you want it to go, or even where it was. It looks like this:

```
{
  key: 'ac3df4', // not with HashHistory!
  pathname: '/somewhere'
  search: '?some=search-string',
  hash: '#howdy',
  state: {
    [userDefined]: true
  }
}
```

The router will provide you with a location object in a few places:

- Route component as `this.props.location`
- Route render as `({ location }) => ()`
- Route children as `({ location }) => ()`
- withRouter as `this.props.location`

It is also found on `history.location` but you shouldn't use that because its mutable. You can read more about that in the history doc.

A location object is never mutated so you can use it in the lifecycle hooks to determine when navigation happens, this is really useful for data fetching and animation.

```
componentWillReceiveProps(nextProps) {
  if (nextProps.location !== this.props.location) {
    // navigated!
  }
}
```

You can provide locations instead of strings to the various places that navigate:

- Web Link to
- Native Link to
- Redirect to
- history.push
- history.replace

Normally you just use a string, but if you need to add some "location state" that will be available whenever the app returns to that specific location, you can use a location object instead. This is useful if you want to branch UI based on navigation history instead of just paths (like modals).

```
// usually all you need
<Link to="/somewhere"/>

// but you can use a location instead
const location = {
  pathname: '/somewhere',
  state: { fromDashboard: true }
}

<Link to={location}/>
<Redirect to={location}/>
history.push(location)
history.replace(location)
```

Finally, you can pass a location to the following components:

- Route
- Switch

This will prevent them from using the actual location in the router's state. This is useful for animation and pending navigation, or any time you want to trick a component into rendering at a different location than the real one.

match

A `match` object contains information about how a `<Route path>` matched the URL. `match` objects contain the following properties:

- `params` - (object) Key/value pairs parsed from the URL corresponding to the dynamic segments of the path
- `isExact` - (boolean) `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

You'll have access `match` objects in various places:

- Route component as `this.props.match`
- Route render as `({ match }) => ()`
- Route children as `({ match }) => ()`
- withRouter as `this.props.match`
- matchPath as the return value

If a Route does not have a `path`, and therefore always matches, you'll get the closest parent match. Same goes for `withRouter`.

matchPath

This lets you use the same matching code that `<Route>` uses except outside of the normal render cycle, like gathering up data dependencies before rendering on the server.

```
import { matchPath } from 'react-router'

const match = matchPath('/users/123', {
  path: '/users/:id',
  exact: true,
  strict: false
})
```

pathname

The first argument is the pathname you want to match. If you're using this on the server with Node.js, it would be `req.path`.

props

The second argument are the props to match against, they are identical to the matching props `Route` accepts:

```
{
  path, // like /users/:id
  strict, // optional, defaults to false
  exact // optional, defaults to false
}
```

withRouter

You can get access to the `history` object's properties and the closest `<Route>`'s `match` via the `withRouter` higher-order component. `withRouter` will pass updated `match`, `location`, and `history` props to the wrapped component whenever it renders.

```
import React from 'react'
```

```
import PropTypes from 'prop-types'
import { withRouter } from 'react-router'

// A simple component that shows the pathname of the current location
class ShowTheLocation extends React.Component {
  static propTypes = {
    match: PropTypes.object.isRequired,
    location: PropTypes.object.isRequired,
    history: PropTypes.object.isRequired
  }

  render() {
    const { match, location, history } = this.props

    return (
      <div>You are now at {location.pathname}</div>
    )
  }
}

// Create a new component that is "connected" (to borrow redux
// terminology) to the router.
const ShowTheLocationWithRouter = withRouter(ShowTheLocation)
```

Important Note

`withRouter` does not subscribe to location changes like React Redux's `connect`does for state changes. Instead, re-renders after location changes propagate out from the `<Router>` component. This means that `withRouter` does *not* re-render on route transitions unless its parent component re-renders. If you are using `withRouter` to prevent updates from being blocked by `shouldComponentUpdate`, it is important that `withRouter` wraps the component that implements `shouldComponentUpdate`. For example, when using Redux:

```
// This gets around shouldComponentUpdate
withRouter(connect(...)(MyComponent))
// or
compose(
  withRouter,
  connect(...)
)(MyComponent)

// This does not
connect(...)(withRouter(MyComponent))
// nor
compose(
  connect(...),
  withRouter
)(MyComponent)
```

See this guide for more information.

Static Methods and Properties

All non-react specific static methods and properties of the wrapped component are automatically copied to the "connected" component.

`Component.WrappedComponent`

The wrapped component is exposed as the static property `WrappedComponent` on the returned component, which can be used for testing the component in isolation, among other things.

```
// MyComponent.js
export default withRouter(MyComponent)

// MyComponent.test.js
import MyComponent from './MyComponent'
render(<MyComponent.WrappedComponent location={{...}} ... />)
```

wrappedComponentRef: func

A function that will be passed as the `ref` prop to the wrapped component.

```
class Container extends React.Component {
  componentDidMount() {
    this.component.doSomething()
  }

  render() {
    return (
      <MyComponent wrappedComponentRef={c => this.component = c}/>
    )
  }
}
```

Examples

Modal Gallery

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

// This example shows how to render two different screens
// (or the same screen in a different context) at the same url,
// depending on how you got there.
//
// Click the colors and see them full screen, then "visit the
// gallery" and click on the colors. Note the URL and the component
// are the same as before but now we see them inside a modal
// on top of the old screen.

class ModalSwitch extends React.Component {
  // We can pass a location to <Switch/> that will tell it to
  // ignore the router's current location and use the location
  // prop instead.
  //
  // We can also use "location state" to tell the app the user
  // wants to go to `/img/2` in a modal, rather than as the
  // main page, keeping the gallery visible behind it.
  //
  // Normally, `/img/2` wouldn't match the gallery at `/`.
  // So, to get both screens to render, we can save the old
  // location and pass it to Switch, so it will think the location
  // is still `/` even though its `/img/2`.
  previousLocation = this.props.location;

  componentWillUpdate(nextProps) {
    const { location } = this.props;
    // set previousLocation if props.location is not modal
    if (
      nextProps.history.action !== "POP" &&
      (!location.state || !location.state.modal)
    ) {
      this.previousLocation = this.props.location;
    }
  }

  render() {
    const { location } = this.props;
    const isModal = !!(
      location.state &&
      location.state.modal &&
      this.previousLocation !== location
    ); // not initial render
    return (
      <div>
        <Switch location={isModal ? this.previousLocation : location}>
          <Route exact path="/" component={Home} />
```

```
          <Route path="/gallery" component={Gallery} />
          <Route path="/img/:id" component={ImageView} />
        </Switch>
        {isModal ? <Route path="/img/:id" component={Modal} /> : null}
      </div>
    );
  }
}

const IMAGES = [
  { id: 0, title: "Dark Orchid", color: "DarkOrchid" },
  { id: 1, title: "Lime Green", color: "LimeGreen" },
  { id: 2, title: "Tomato", color: "Tomato" },
  { id: 3, title: "Seven Ate Nine", color: "#789" },
  { id: 4, title: "Crimson", color: "Crimson" }
];

const Thumbnail = ({ color }) => (
  <div
    style={{
      width: 50,
      height: 50,
      background: color
    }}
  />
);

const Image = ({ color }) => (
  <div
    style={{
      width: "100%",
      height: 400,
      background: color
    }}
  />
);

const Home = () => (
  <div>
    <Link to="/gallery">Visit the Gallery</Link>
    <h2>Featured Images</h2>
    <ul>
      <li>
        <Link to="/img/2">Tomato</Link>
      </li>
      <li>
        <Link to="/img/4">Crimson</Link>
      </li>
    </ul>
  </div>
);

const Gallery = () => (
  <div>
    {IMAGES.map(i => (
      <Link
        key={i.id}
```

```
        to={{
          pathname: `/img/${i.id}`,
          // this is the trick!
          state: { modal: true }
        }}
      >
        <Thumbnail color={i.color} />
        <p>{i.title}</p>
      </Link>
    ))}
  </div>
);

const ImageView = ({ match }) => {
  const image = IMAGES[parseInt(match.params.id, 10)];
  if (!image) {
    return <div>Image not found</div>;
  }

  return (
    <div>
      <h1>{image.title}</h1>
      <Image color={image.color} />
    </div>
  );
};

const Modal = ({ match, history }) => {
  const image = IMAGES[parseInt(match.params.id, 10)];
  if (!image) {
    return null;
  }
  const back = e => {
    e.stopPropagation();
    history.goBack();
  };
  return (
    <div
      onClick={back}
      style={{
        position: "absolute",
        top: 0,
        left: 0,
        bottom: 0,
        right: 0,
        background: "rgba(0, 0, 0, 0.15)"
      }}
    >
      <div
        className="modal"
        style={{
          position: "absolute",
          background: "#fff",
          top: 25,
          left: "10%",
          right: "10%",
          padding: 15,
```

```
          border: "2px solid #444"
        }}
      >
        <h1>{image.title}</h1>
        <Image color={image.color} />
        <button type="button" onClick={back}>
          Close
        </button>
      </div>
    </div>
  );
};

const ModalGallery = () => (
  <Router>
    <Route component={ModalSwitch} />
  </Router>
);

export default ModalGallery;
```

Animated Transitions

```
import React from "react";
import { TransitionGroup, CSSTransition } from "react-transition-group";
import {
  BrowserRouter as Router,
  Switch,
  Route,
  Link,
  Redirect
} from "react-router-dom";

/* you'll need this CSS somewhere
.fade-enter {
  opacity: 0;
  z-index: 1;
}

.fade-enter.fade-enter-active {
  opacity: 1;
  transition: opacity 250ms ease-in;
}
*/

const AnimationExample = () => (
  <Router>
    <Route
      render={({ location }) => (
        <div style={styles.fill}>
          <Route
            exact
            path="/"
            render={() => <Redirect to="/hsl/10/90/50" />}
          />

          <ul style={styles.nav}>
            <NavLink to="/hsl/10/90/50">Red</NavLink>
            <NavLink to="/hsl/120/100/40">Green</NavLink>
            <NavLink to="/rgb/33/150/243">Blue</NavLink>
            <NavLink to="/rgb/240/98/146">Pink</NavLink>
          </ul>

          <div style={styles.content}>
            <TransitionGroup>
              {/* no different than other usage of
                CSSTransition, just make sure to pass
                `location` to `Switch` so it can match
                the old location as it animates out
            */}
              <CSSTransition key={location.key} classNames="fade"
timeout={300}>
                <Switch location={location}>
                  <Route exact path="/hsl/:h/:s/:l" component={HSL} />
                  <Route exact path="/rgb/:r/:g/:b" component={RGB} />
```

```
                {/* Without this `Route`, we would get errors during
                    the initial transition from `/` to `/hsl/10/90/50`
                */}
                  <Route render={() => <div>Not Found</div>} />
                </Switch>
              </CSSTransition>
            </TransitionGroup>
          </div>
        </div>
      )}
    />
  </Router>
);

const NavLink = props => (
  <li style={styles.navItem}>
    <Link {...props} style={{ color: "inherit" }} />
  </li>
);

const HSL = ({ match: { params } }) => (
  <div
    style={{
      ...styles.fill,
      ...styles.hsl,
      background: `hsl(${params.h}, ${params.s}%, ${params.l}%)`
    }}
  >
    hsl({params.h}, {params.s}%, {params.l}%)
  </div>
);

const RGB = ({ match: { params } }) => (
  <div
    style={{
      ...styles.fill,
      ...styles.rgb,
      background: `rgb(${params.r}, ${params.g}, ${params.b})`
    }}
  >
    rgb({params.r}, {params.g}, {params.b})
  </div>
);

const styles = {};

styles.fill = {
  position: "absolute",
  left: 0,
  right: 0,
  top: 0,
  bottom: 0
};

styles.content = {
  ...styles.fill,
  top: "40px",
```

```
  textAlign: "center"
};

styles.nav = {
  padding: 0,
  margin: 0,
  position: "absolute",
  top: 0,
  height: "40px",
  width: "100%",
  display: "flex"
};

styles.navItem = {
  textAlign: "center",
  flex: 1,
  listStyleType: "none",
  padding: "10px"
};

styles.hsl = {
  ...styles.fill,
  color: "white",
  paddingTop: "20px",
  fontSize: "30px"
};

styles.rgb = {
  ...styles.fill,
  color: "white",
  paddingTop: "20px",
  fontSize: "30px"
};

export default AnimationExample;
```