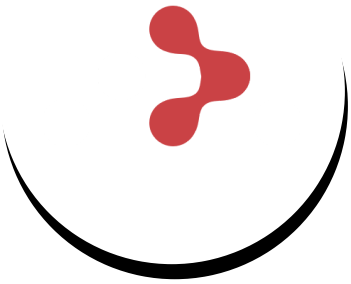


DECLARATIVE ROUTING

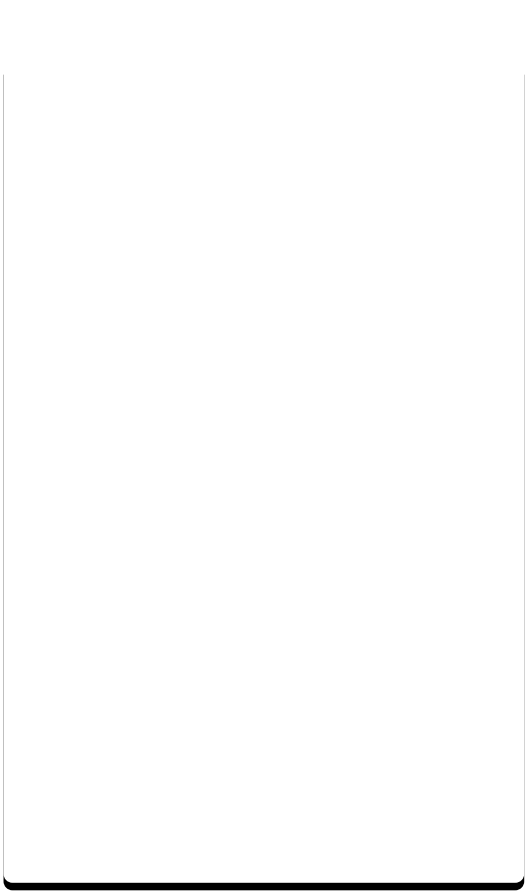
FOR REACT APPLICATIONS

Components are the heart of React's powerful, declarative programming model. React Router is a collection of **navigational components** that compose naturally with your application. Whether you want to have **bookmarkable URLs** for your web app or a composable way to navigate in **React Native**, React Router works wherever React is rendering.

LIVE EXAMPLES API DOCUMENTATION


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

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>
)

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

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

const Topics = ({ match }) => (
  <div>
    <h2>Topics</h2>
    <ul>
      <li><Link to={`${match.url}/rendering`}>Renderir
      <li><Link to={`${match.url}/components`}>Compone
      <li><Link to={`${match.url}/props-v-state`}>Prop
    </ul>

    <Route path={`${match.url}/:topicId`} component={T
```

All of these examples can be copy pasted into an app created with `create-react-app`. Just paste the code into `src/App.js` of your project.



```
    <Route exact path={match.url} render={() => (
      <h3>Please select a topic.</h3>
    )}/>
  </div>
)

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

export default BasicExample
```

Installation

React Router runs in multiple environments: browsers, servers, native, and even VR (works in the dev preview!) While many components are shared (like `Route`) others are specific to environment (like `NativeRouter`). Rather than requiring you install two packages, you only have to install the package for the target environment. Any shared components between the environments are re-exported from the environment specific package.

Web

```
npm install react-router-dom@next
# or
yarn add react-router-dom@next
```

All of the package modules can be imported from the top:

```
import {
  BrowserRouter as Router,
  StaticRouter, // for server rendering
```

```
  Route,
  Link
  // etc.
} from 'react-router-dom'
```

If you're going for a really minimal bundle sizes on the web you can import modules directly. Theoretically a tree-shaking bundler like Webpack makes this unnecessary but we haven't tested it yet. We welcome you to!

```
import Router from 'react-router-dom/BrowserRouter'
import Route from 'react-router-dom/Route'
// etc.
```

Native

We're still working on great documentation for the native capabilities of React Router. For now we recommend you read the source.

```
yarn add react-router-native@next
# or if not using the react-native cli
npm install react-router-native@next
```

All of the package modules can be imported from the top:

```
import {
  NativeRouter as Router,
  DeepLinking,
  AndroidBackButton,
  Link,
  Route
  // etc.
} from 'react-router-native'
```

Who-knows-where

```
yarn add react-router@next
# or if not using the react-native cli
npm install react-router@next
```

All of the package modules can be imported from the top:

```
import {
  MemoryRouter as Router,
  Route
  // etc.
} from 'react-router'
```

You can use React Router's navigation anywhere you run React, the navigation state is kept in a memory router. You can look at the implementation of NativeRouter to get an idea on how to integrate.

<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 <BrowserRouter>

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.

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

getUserConfirmation: func <BrowserRouter>

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 <BrowserRouter>

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 <BrowserRouter>

The length of location.key. Defaults to 6.

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

children: node

<BrowserRouter>

A single child element to render.

<NativeRouter>

A <Router> for native iOS and Android apps built using React Native.

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

<NativeRouter>
  <App/>
</NativeRouter>
```

getUserConfirmation: func

<NativeRouter>

A function to use to confirm navigation.

```
import { Alert } from 'react-native'

// This is the default behavior
const getConfirmation = (message, callback) => {
  Alert.alert('Confirm', message, [
    { text: 'Cancel', onPress: () => callback(false) },
    { text: 'OK', onPress: () => callback(true) }
  ])
}

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

keyLength: number

<NativeRouter>

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

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

children: node

<NativeRouter>

A single child element to render.

<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

<StaticRouter>

The base URL for all locations.

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

location: string

<StaticRouter>

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

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

location: object *<StaticRouter>*

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

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

context: object *<StaticRouter>*

A plain JavaScript object that records the results of the render. See the example above.

children: node *<StaticRouter>*

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 *<HashRouter>*

The base URL for all locations.

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

getUserConfirmation: func *<HashRouter>*

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 *<HashRouter>*

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 *<HashRouter>*

A single child element to render.

<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 *<MemoryRouter>*

An array of `locations` 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 <MemoryRouter>

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

getUserConfirmation: func <MemoryRouter>

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

keyLength: number <MemoryRouter>

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

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

children: node <MemoryRouter>

A single child element to render.

<Router>

The common low-level interface for all router components. Higher-level routers include:

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

Use a `<Router>` directly if you already have a `history` object.

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

const history = createBrowserHistory()

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

history: object <Router>

A history object to use for navigation.

children: node <Router>

A single child element to render.

<Route>

Renders some UI when a URL matches a location.

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

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

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

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

You should use only one of these props on a given <Route>. See their explanations below to understand why you have 3 options.

component: func <Route>

A React component to render when the location matches. The component receives all the properties on context.router.

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

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

When you use component (instead of render, below) the router uses React.createElement to create a new React element from the given component.

render: func <Route>

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. This function will be called with the same props that are passed to the component.

This allows for convenient inline match rendering and wrapping.

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

// wrapping/composing
```

```
const FadingRoute = ({ component: Component, ...rest }) => (
  <Route {...rest} render={matchProps => (
    <FadeIn>
      <Component {...matchProps}/>
    </FadeIn>
  )}/>
)

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

NOTE: <Route component> takes precendence over <Route render> so don't use both in the same <Route>.

children: func <Route>

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 prop will be called with an object that contains a `match` and a `history` property. `match` will be null if there was no match. This will allow you to dynamically adjust your UI based on if the route matches or not.

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>
)}/>
```

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

path: string <Route>

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

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

Routes without a `path` *always* match.

exact: bool <Route>

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 <Route>

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

NOTE: `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

<Switch>

Renders the first child <Route> 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:

<NativeRouter>
getUserConfirmati
func
keyLength:

number
children:
node

<StaticRouter>
basename:
string
location:
string
location:
object

context:
object
children:
node

<HashRouter>
basename:
string
getUserConfirmati
func
hashType:
string
children:
node

<MemoryRouter>
initialEntries:
array
initialIndex:
number
getUserConfirmati
func
keyLength:
number
children:
node

<Router>
history:
object
children:
node

<Route>
component:
func
render:
func
children:
func
path:
string
exact:
bool
strict:
bool

<Switch>
children:
node

<Link>
to:
object
to:
string
replace:
bool

<NavLink>
activeClassName:
string
activeStyle:
object
exact:
bool
strict:
bool
isActive:
func

<Redirect>
to:

```
<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>
```

children: node <Switch>

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

<Link>

Provides declarative, accessible navigation around your application.

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

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

to: object <Link>

The location to link to.

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

to: string <Link>

The pathname or location to link to.

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

replace: bool <Link>

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

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

<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" activeClassName="active">About</NavLink>
```

activeClassName: string <NavLink>

The class to give the element when it is active. There is no default active class. This will be joined with the `className` prop.

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

activeStyle: object *<NavLink>*

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

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

exact: bool *<NavLink>*

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

```
<NavLink exact to="/profile" activeClassName='active'>Profile</NavLink>
```

strict: bool *<NavLink>*

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/" activeClassName='active'>Events</NavLink>
```

isActive: func *<NavLink>*

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} activeClassName="active">Event 123</NavLink>
```

<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 *<Redirect>*

The URL to redirect to.

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

to: object *<Redirect>*

A location to redirect to.

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

push: bool *<Redirect>*

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

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

<Prompt>

Used to prompt the user before navigating away from a page. When your application enters a state that should prevent the user from navigating away (like a form is half-filled out), render a `<Prompt>`.

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

<Prompt
  when={formIsHalfFilledOut}
  message="Are you sure you want to leave?"
/>
```

message: string <Prompt>

The message to prompt the user with when they try to navigate away.

```
<Prompt message="Are you sure you want to leave?"/>
```

message: func <Prompt>

Will be called with the next `location` and `action` the user is attempting to navigate to. Return a string to show a prompt to the user or `true` to allow the transition.

```
<Prompt message={location => (
  `Are you sure you want to go to ${location.pathname}?`
)}/>
```

when: bool <Prompt>

Instead of conditionally rendering a `<Prompt>` behind a guard, you can always render it but pass `when={true}` or `when={false}` to prevent or allow navigation accordingly.

```
<Prompt when={formIsHalfFilledOut} message="Are you sure?"/>
```

withRouter

You can get access to the `router` object's properties via the `withRouter` higher-order component. This is the recommended way to access the `router` object. `withRouter` will re-render the component every time the route changes.

```
import React, { PropTypes } from 'react'
import { withRouter } from 'react-router'

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

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

// Create a new component that is "connected" (to borrow redux
// terminology) to the router. This component receives all of the
```

```
// router's properties as props.
const ShowTheLocationWithRouter = withRouter(ShowTheLocation)
```

context.router

Every <Router> puts a router object on context. This object opens a channel of communication between e.g. a <Router> and its descendant <Route>s, <Link>s, and <Prompt>s, etc.

While this interface is mainly for internal use, it can also occasionally be useful as public API. However, we encourage you to use it only as a last resort. Context itself is an experimental API and may break in a future release of React.

It may be helpful to think of context.router as the merging together of two interfaces:

1. the history object and
2. the match object.

Expressed in code, context.router would be

```
context.router = {
  ...history,
  match
}
```

Thus, the router object has all methods and properties of its underlying history instance (and updates in place the same as the history instance) with an additional match object that describes how the router matched the URL.

Just as components are nested, context.router objects are nested and create a hierarchy. Each time you render a new <Route>, it shadows context.router to be the point of reference for all its descendants. This lets us express nested routes by simply nesting components.

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` - (string) 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)

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

match

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

- `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

The majority of the time you can get a `match` object as a prop to your `<Route component>` or in your `<Route render>` callback, so you shouldn’t need to manually generate them.

However, you may find it useful to manually calculate the match if you have a pre-determined route config that you’d like to traverse in order to know which routes match. In that case, we also export our `matchPath` function so you can use it to match just like we do internally.

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

const match = matchPath('/the/pathname', '/the/:dynamicId', {
  exact: true,
  strict: false
})
```

Sign up to receive updates about React Router, our workshops, online courses, and more.

FIRST NAME EMAIL ADDRESS Subscribe

React Router is built and maintained by React Training and hundreds of contributors.