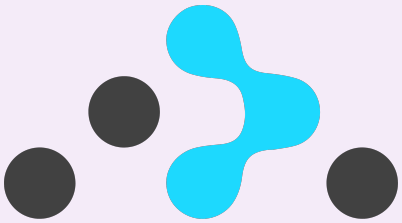


React Router

tl;dr

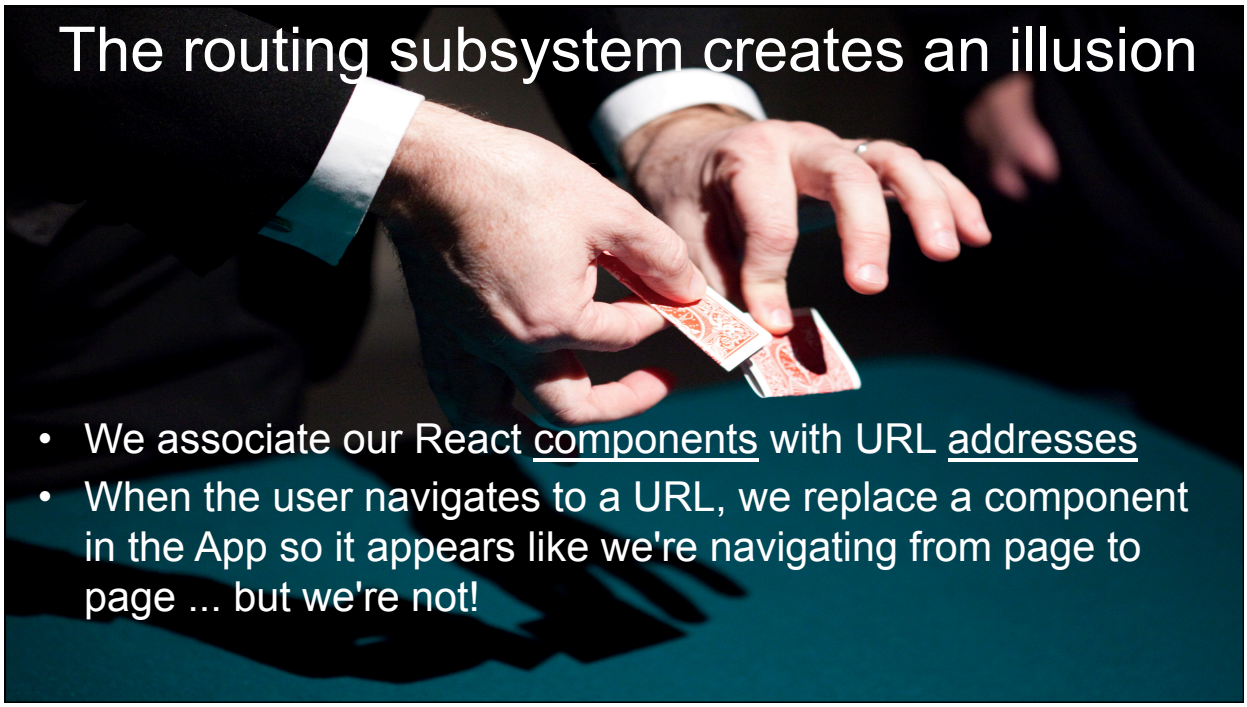
- Routing tricks the user into thinking they're navigating to pages when we're only swapping out components
- To route, you ...

1. Define the routing domain with a <Router>
2. Match URLs to components with <Route>s
3. Send the user to another route when ...
   - They type in a URL
   - They click on a <Link>
   - You push a URL onto the props.history stack
4. Read route parameters via props.match.params

React thinks in SPAs first

- But if there's only one page, how do we navigate from page to page?
- We don't!

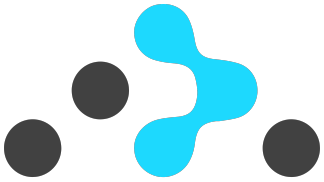
The routing subsystem creates an illusion



- We associate our React components with URL addresses
- When the user navigates to a URL, we replace a component in the App so it appears like we're navigating from page to page ... but we're not!

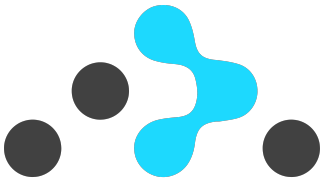
Routing is that process of keeping the browser URL in sync with what's being rendered on the page

React Router lets you handle routing declaratively.



React Router is NOT from Facebook!

- It's from the community
- But it is the de facto router



We route through JSX

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

Hey, wait! "<Route>" isn't a thing! What is it?

It's a React component!

So where did it come from?

```
npm install react-router-dom    //For web
// And then ...
import { Route } from 'react-router-dom';
```

We need to know these fundamental things to route:

1. How to define the domain of a router
2. How to create routes
3. How to enable users to get to a route
4. How to read route parameters out of the URL

1. How to define the domain of our router

Only <Route>s that are nested somewhere -- at any level -- inside a <Router> will be honored

So they're usually put around or directly inside the <App />

Let's see a few examples...

A Router **around** the App

index.js

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

A Router **inside** the App

App.js

```
function App {
return <div classname="App">
 <BrowserRouter>
  <header className="Header">
   {/* All ur code goes here */}
  </footer>
 </BrowserRouter>
</div>;
}
```

The router

<BrowserRouter>	<HashRouter>
• us.com/someUrl • Prettier and should be the default	• us.com/#/someUrl • Uglier but works in older browsers

Unless you need to support old browsers, just use <BrowserRouter>

2. How to create the routes

If the URL is a Regex match of "/foo", then cram the Bar component right here.

<Route path="/foo" component={Bar} />

Design the routes ...

URL	Component
/people	People
/people/123	Person
/teams	Team
/cart	ManageCart
/	Welcome
Anything else	FourOhFour

```
<Route path="/" component={Welcome} />
<Route path="people" component={People} />
<Route path="people/:personId" component={Person} />
<Route path="teams" component={Team} />
<Route path="cart" component={ManageCart} />
<Route path="*" component={FourOhFour} />
```

For example ...

App.js

```
<BrowserRouter>
...
 <Route path="/" component={Home}/>
 <Route path="/cat" component={Catalog}/>
 <Route path="/cart" component={Cart}/>
 <Route path="/cat/:id" component={Prod}/>
...
</BrowserRouter>
```

React Router uses *inclusive* routing

- All routes that match the path are included
- If the url were "/cat/123", we'd see Home, followed by Catalog, followed by Prod

App.js

```
<BrowserRouter>
...
 <Route path="/" component={Home}/>
 <Route path="/cat" component={Catalog}/>
 <Route path="/cart" component={Cart}/>
 <Route path="/cat/:id" component={Prod}/>
...
</BrowserRouter>
```

React router will look at the url, and scan for **all** Routes that match

http://us.com...	matched components
/	Home
/foo	Home
/cat	Home and Catalog
/cat/123	Home and Catalog and Prod

Put them in a *Switch* element which says to match only the first one.

Or add an *exact* attribute to each

App.js

```
<BrowserRouter>
 <Switch>
  ...
  <Route exact path="/" component={Home}/>
  <Route path="/cat" component={Catalog}/>
  <Route path="/cart" component={Cart}/>
  <Route path="/cat/:id" component={Prod}/>
  <Route component={FourOhFour} />
 </Switch>
</BrowserRouter>
```

What if I want just one?

Properties of a Route component

```
<Route path (component|render|children) [exact] />
```

path	A regex which will be matched against the URL
component	The component to place here
render	A function that returns the JSX to render
children	A function that renders if the route does not match
exact	Treat the path as a plain string instead of a regex

3. How to enable users to navigate to a route

How can a user request a page?

At run time, a user can ...

1. Type a url
2. Click a link
3. Get pushed there in a Component class

1) They can type in a URL

- When the user types in the URL, the browser has no choice but to request a resource from the server.
- The server will have been configured to serve the root page at any address.
- Routing will then will intercept that and route him to the proper component.

2) They can click on a link

- Write your links like this:

```
<Link to="/people">All people</Link>
<Link to="/people/{p.id}">This Person</Link>
<Link to="/teams">Teams</Link>
<Link to="/cart">My cart</Link>

<Link to="/someLink">Text to display</Link>
```

3) They can get pushed in the class

```
function Foo() {
 return <div>
  {someCondition && <Redirect to="/other" />}
 </div>;
}
```

If you can use JSX, the <Redirect> component is the simplest option

3) They can get pushed in the class

- To send a user programmatically
- props has a history which wraps the browser's history object
- so just go
- props.history.push('/WhereYouWantThemToGo');

If you do <u>not</u> have access to JSX, push() can send them to another route.

We normally send props in JSX like this:

```
<Foo prop1={val1} prop2={val2} />
```

But if we're using the history API, how do we get props into it?

Pass a second argument to to it like this:

```
props.history.push("/SomeRoute", {
 prop1:val1,
 prop2:val2
});
```

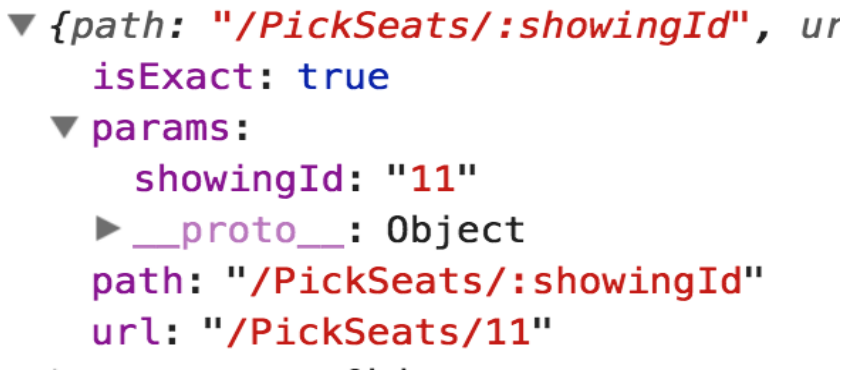
4. How to read route parameters out of the URL

When a Route is matched, the component is given a match object

- match.url - String. The url (eg. "/person/1234")
- match.path - String. The Route path (eg. "/person/:personId")
- match.isExact - boolean. true=exact. false=partial
- match.params - Aha!

Route parameters

- props.match is given to all components who were visited by the Router.
- The parameters will be found in props.match.params.foo



App.js

```
<BrowserRouter>
 <Route path="/cat/:id" component={Prod}/>
</BrowserRouter>
```

http://us.com/cat/1234

Prod.js

```
function Prod(props) {
 console.log(props.match.params.id);
 return <SomeJSX />
}
```

Match the thing on the left with its purpose on the right

A. <Switch>	1. Forces an exclusive match
B. <Link>	2. Takes the place of an <a>
C. <Route>	3. Conditionally inserts a component here
D. <Router>	4. Shows the boundaries of the router
E. <Redirect>	5. Forwards the user to another Route

tl;dr

- Routing tricks the user into thinking they're navigating to pages when we're only swapping out components
- To route, you ...

1. Define the routing domain with a <Router>
2. Match URLs to components with <Route>s
3. Send the user to another route when ...
   - They type in a URL
   - They click on a <Link>
   - You push a URL onto the props.history stack
4. Read route parameters via props.match.params