A <Router /> with Hooks and Suspense

By James K Nelson

Navi is a new kind of router for React. It lets you declaratively map URLs to content, even when that content is asynchronous.

18th February, 2019

So the React team just released a new API called *Hooks*. It's amazing. It lets you declaratively model state and side effects. You've probably read about it elsewhere on the internet, so I'm not going to tell you about hooks themselves, but...

With a new API comes new possibilities. And to cut to the chase, Navi's new `<Router>` component uses Hooks and Suspense to make routing simpler than ever before. It makes all sorts of things possible—you can even add animated loading transitions in *just 3 lines of code*.

So how do you use these new superpowered hooks? We'll get to that in a moment. But before we do, *what the hell is a `<Router>`?*

TL;DR?

Asynchronous routing has never been this simple.

- View the docs »
- View the GitHub repo »

How many routes could a `<Router routes />` route...

It can route as many as you'd like, because Navi lets you dynamically `import()` entire routing trees on demand. But *how*?

The trick is in Navi's method for declaring routes. For simple routes, you can just use Navi's `mount()` and `route()` functions. But for heavier content, you can declare dependencies on asynchronous data and views using `async/await`—or you can even split out entire routing trees using `lazy()`.

```
<Router routes={
  mount({
    '/': route({
      title: 'My Shop',
      getData: () => api.fetchProducts(),
      view: <Landing />,
    }),
    '/products': lazy(() => import('./productsRoutes')),
  })
} />
```

It could route over 9000 routes...

If you take a look at this example, you'll see that you've got yourself a `<Router>` with a couple routes, including a shop's landing page and a lazily loadable `/products` URL.

Let's build the rest of the shop.

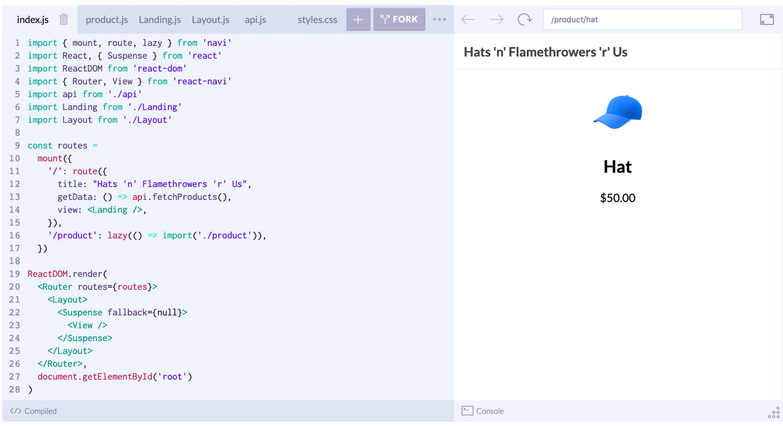
For your next step, you'll need to decide *where* to render the current route's `view` element. And to do that, you just plonk down a `<View />` element somewhere inside your `<Router>`.

```
ReactDOM.render(
  <Router routes={routes}>
    <Layout>
      <View />
    </Layout>
  </Router>,
  document.getElementById('root')
)
```

Simple, huh? But waaait a minute... what if you view the lazily loadable `/products` URL? Then the route will be loaded via an `import()`, which returns a Promise, and so at first there'll be nothing to render. Luckily, React's new `<Suspense>` feature lets you declaratively wait for promises to resolve. So just wrap your `<View>` in a `<Suspense>` tag, and you're off and racing!

Isn't **`<View>`** part of react-native?

It is! But react-native apps would be far better served by react-native-navigation or react-navigation than by Navi.



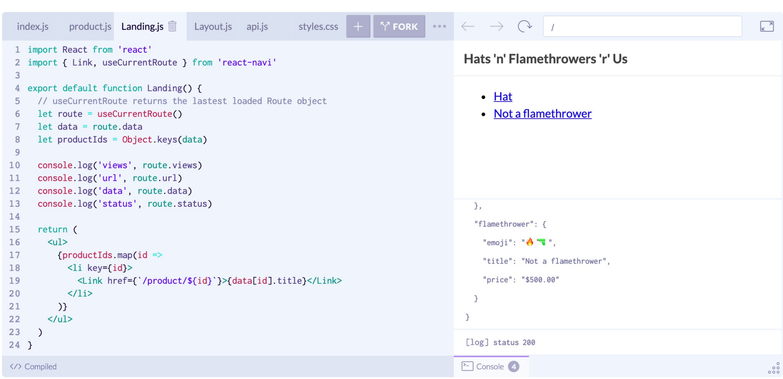
Bro, just give me the hooks?

Ok, so you've seen how to render a route's view. But did you notice that your route also defines a `getData()` function?

```
route({
  title: 'My Shop',
  getData: () => api.fetch('/products'),
  view: <Landing />,
})
```

How do you access the data? With React hooks!

Navi's `useCurrentRoute()` hook can be called from any function component that is rendered within the `<Router>` tag. It returns a `Route` object that contains everything that Navi knows about the current URL.



Ok. So far, so good. But imagine that you've just clicked a link to `/products`—which is dynamically imported. It's going to take some time to fetch the route, so what are you going to display in the meantime?