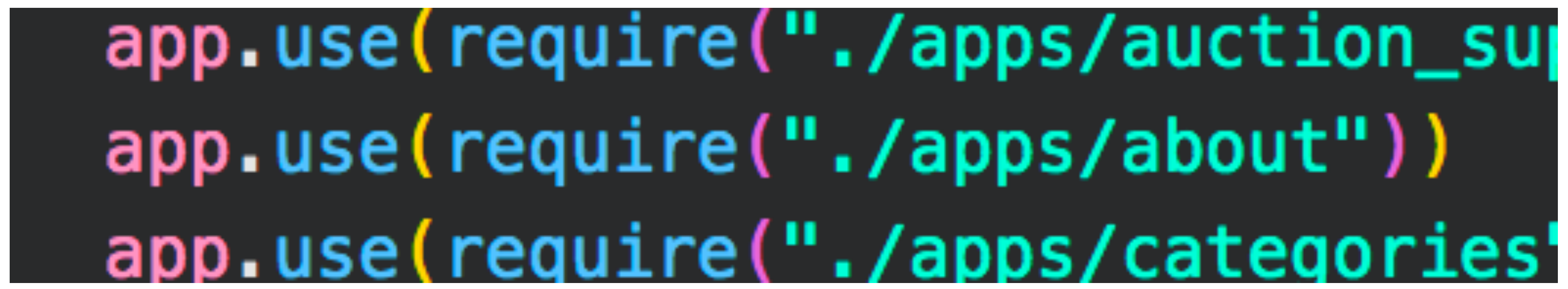
Artsy v2
(aka “New App Shell”)

A quick glance at where artsy.net's FE has been and is going

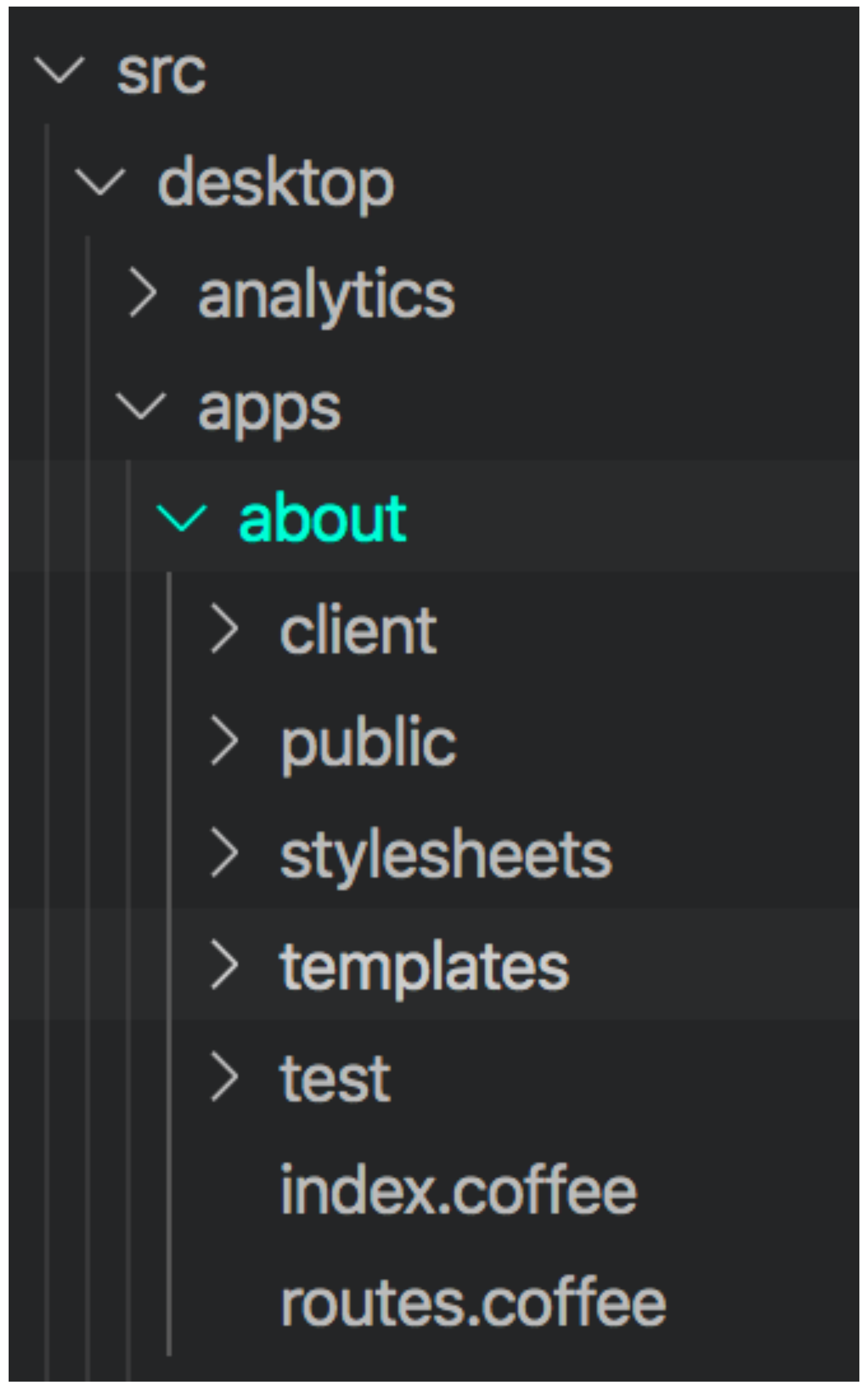
First, some recent history

artsy.net is a Node.js app built on top of Express

Historically, individual express sub-apps represented pages on the site (e.g. artsy.net/artist/pablo-picasso)



```
app.use(require("./apps/auction_su
app.use(require("./apps/about"))
app.use(require("./apps/categories'
```



```
src
  desktop
    analytics
    apps
      about
        client
        public
        stylesheets
        templates
        test
        index.coffee
        routes.coffee
```

Pages were built using Backbone.js, CoffeeScript and HTML was written in Jade

Dynamic data was rendered via Backbone models communicating RESTfully with Gravity

Then Metaphysics (our GraphQL layer) was introduced, which simplified how we wrote our UI by streamlining backend communication

React was introduced, and things started to really change

Reaction was born, introducing TypeScript and Relay

Stitch was written, allowing us to seamlessly mix new React-based code into our old artsy.net infrastructure, and allow for full-page React apps

More and more pages started being written in React, although patterns were inconsistent

We needed to start thinking about scale, from both an engineering and design perspective

In early 2018 a FE taskforce was established which laid out needs and goals and formalized a plan of action

Requirement 1:

Unify FE code behind a common stack and consolidate patterns

Requirement 2:

Pages built using this stack needed to be SSR (server-side-render) friendly, for SEO reasons

Requirement 3:

We needed a unified, consistent and **constrained** design system for building UI, which can be shared across web and mobile platforms

These requirements gave rise to two direction-defining pieces of FE infrastructure:

1. A new routing framework

2. Palette

With these two pieces our FE rapidly became more consistent and our team velocity greatly increased

Building UI became much more LEGO-like, and less about knowing the ins and outs of JS, CSS, etc

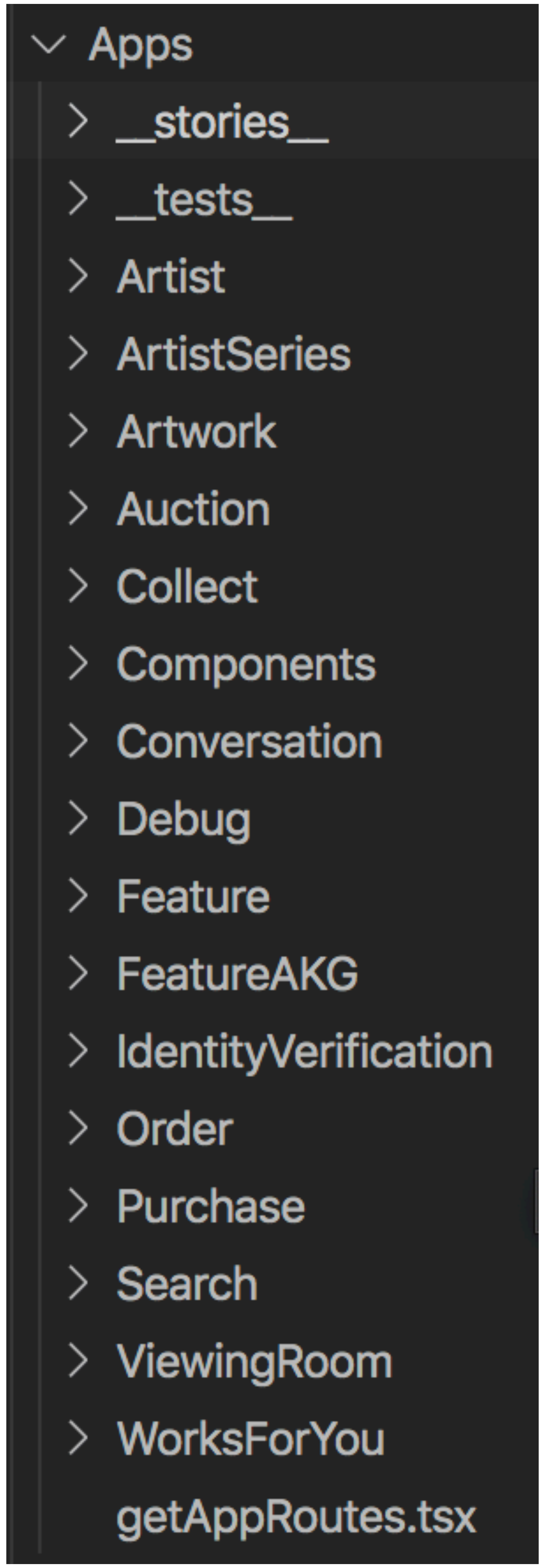
And code written for web could very quickly be adapted to mobile, and in many cases straight up copy / pasted

The FE became a friendlier and much more inclusive environment for our dev team!

Fast forward to 2020…

All of our main pages have been rewritten in React and Relay, on top of the new routing framework

New apps are being added monthly:



```
Apps
  __stories__
  __tests__
  Artist
  ArtistSeries
  Artwork
  Auction
  Collect
  Components
  Conversation
  Debug
  Feature
  FeatureAKG
  IdentityVerification
  Order
  Purchase
  Search
  ViewingRoom
  WorksForYou
  getAppRoutes.tsx
```

However, prior to the new App Shell there were inefficiencies

Each app still had to be individually mounted as an express sub app leading to a lot of code duplication

When navigating between pages every click led to a full reload in the browser

This in turn required additional requests to be made to artsy.net's node server, and everything remounted / reinitialized

On mobile phones in particular, this is a very slow and resource-intense user experience

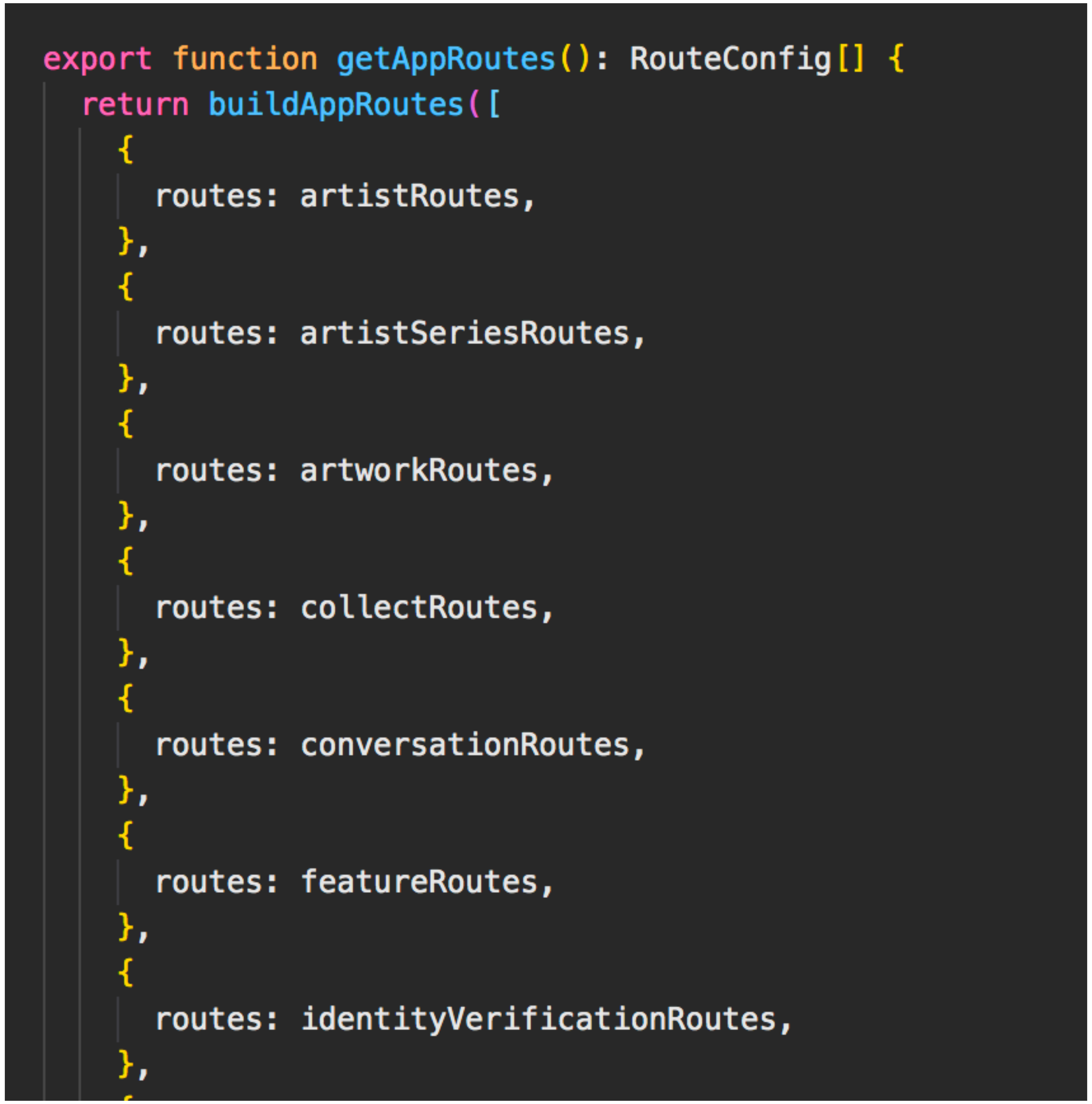
What if instead of multiple express sub-apps, there was just one?

And instead of the page doing a full hard-reload after every click, we simply re-rendered the page’s React code?

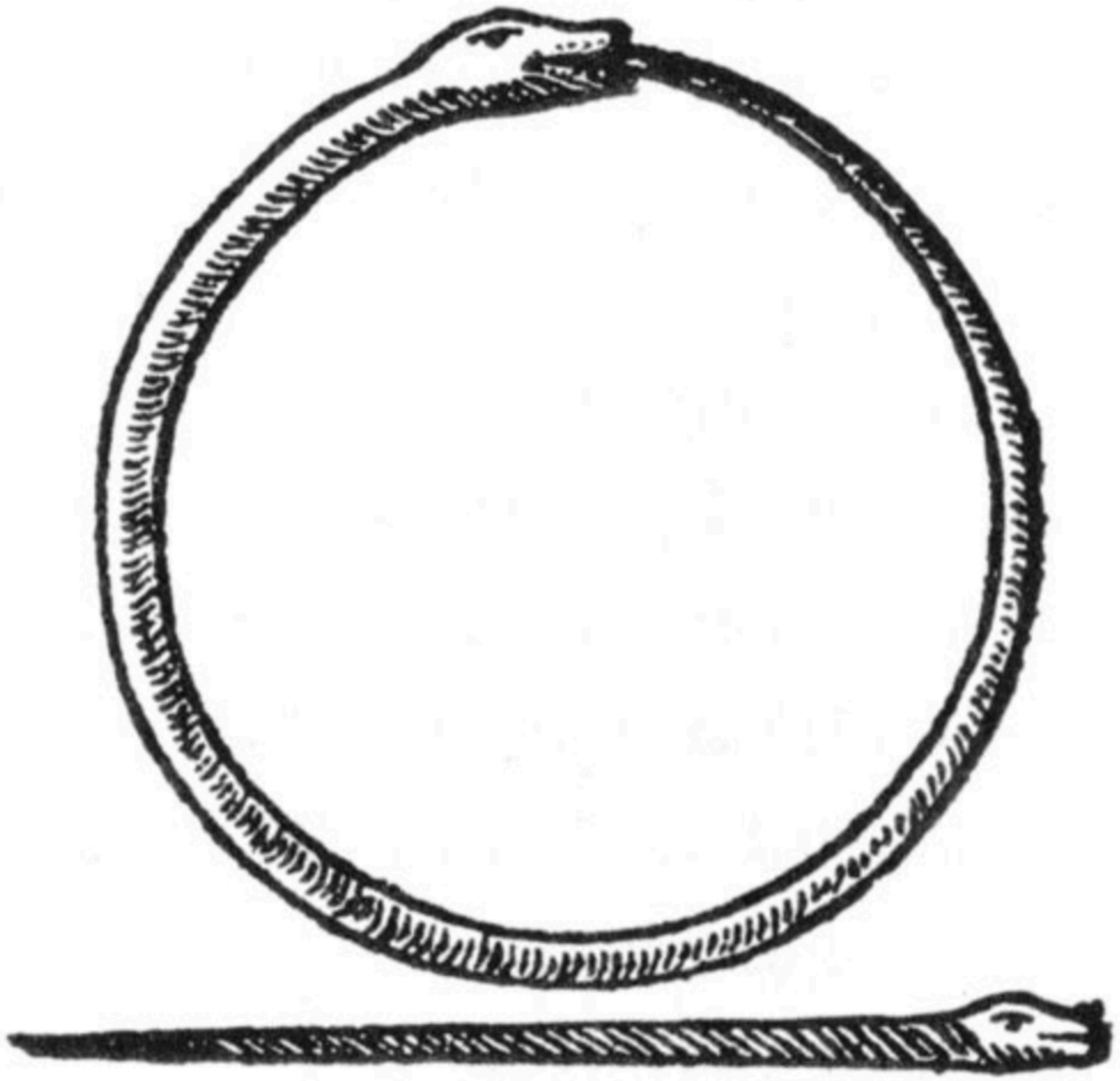
We were already doing this within individual apps — for example, **artist -> artist/works-for-sale**, or the order / checkout app

Because our framework standardized approaches…

…we were able to “lift up” our routing layer one level higher, and compose the routes of our individual apps into a global route manifest with very few changes



```
export function getAppRoutes(): RouteConfig[] {
  return buildAppRoutes([
    {
      routes: artistRoutes,
    },
    {
      routes: artistSeriesRoutes,
    },
    {
      routes: artworkRoutes,
    },
    {
      routes: collectRoutes,
    },
    {
      routes: conversationRoutes,
    },
    {
      routes: featureRoutes,
    },
    {
      routes: identityVerificationRoutes,
    },
```



Some benefits of this new architecture

1. Performance

For a given user session across all of our main pages, the artsy.net server only has to be hit once

All subsequent requests are made directly to Metaphysics, skipping artsy.net completely

After the initial server-side render, all page rendering happens entirely on the client, greatly increasing UX speed, especially on mobile

Navigating from page to page, Artsy.net then becomes as fast as the slowest offender: the time it takes to receive a response from Metaphysics

There’s also a network cache that hooks into Relay

When logged out, most network responses are cached in Redis and retrieved during the SSR pass (excluding /artwork/id)

And on the client, after a user has visited a particular page, the network response is stored in an in-memory cache

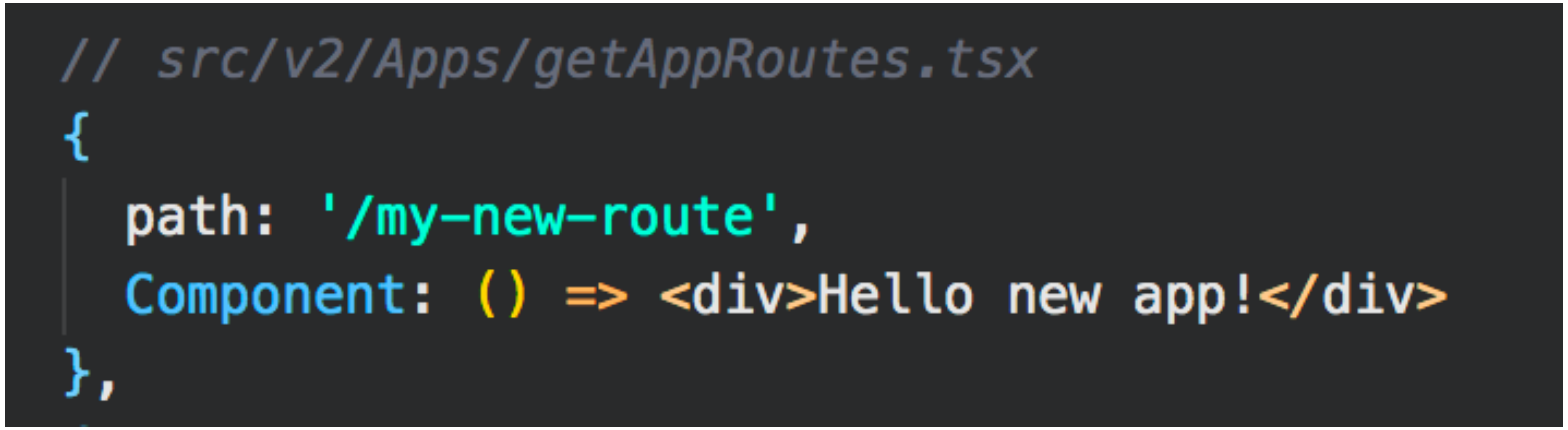
For the user, navigating between cached pages becomes instant

And as our power users go back and forth between artists and artworks (the data team has a name for this dance) the time saved, compared to old architecture, adds up fast

2. Simplicity

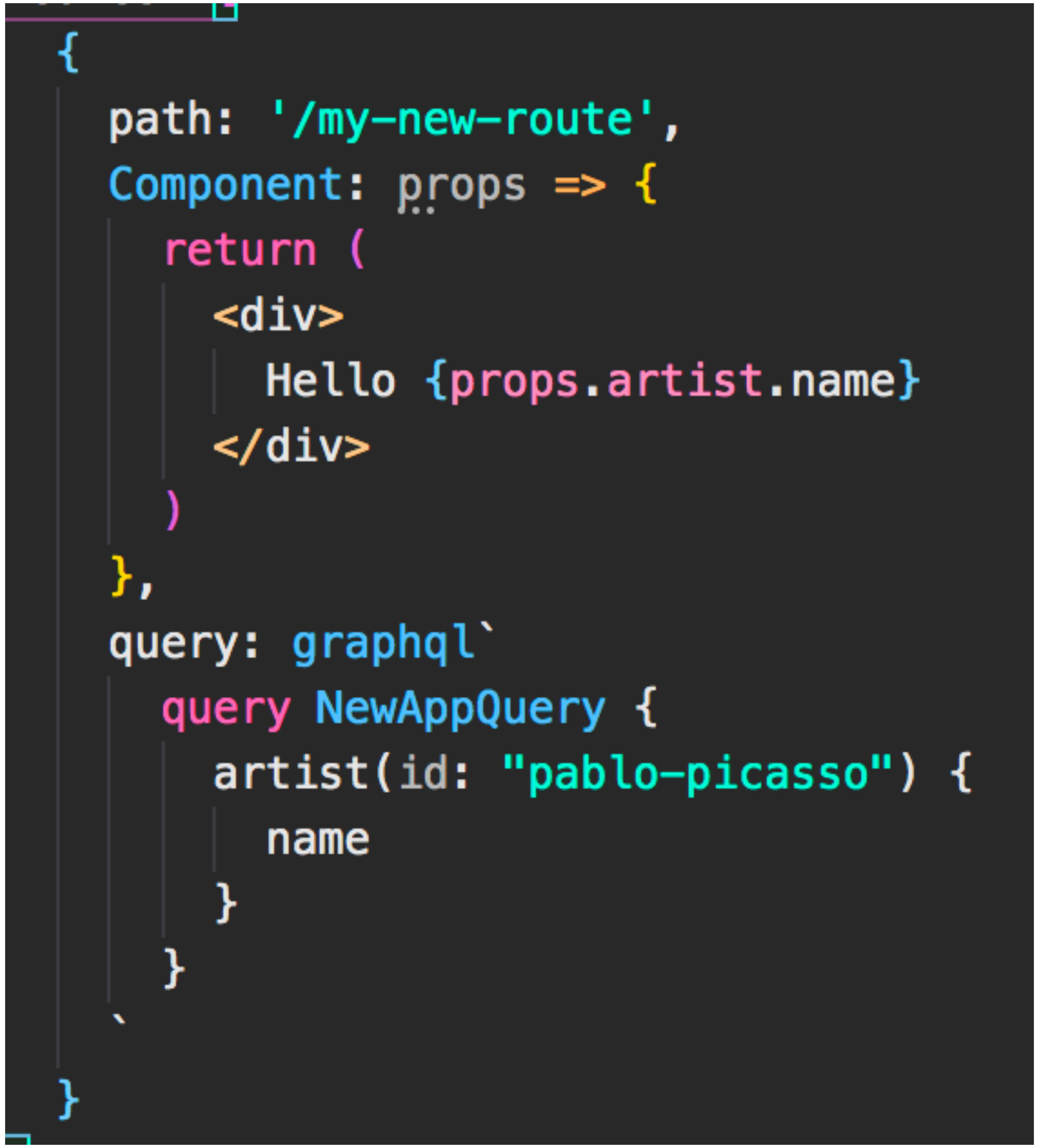
Want to add a new SSR-rendered react app route to force?

Only 2 lines of code:



```
// src/v2/Apps/getAppRoutes.tsx
{
  path: '/my-new-route',
  Component: () => <div>Hello new app!</div>
},
```

And connect that app to Metaphysics? Just a few lines more:



```
{
  path: '/my-new-route',
  Component: props => {
    return (
      <div>
        Hello {props.artist.name}
      </div>
    )
  },
  query: graphql`
    query NewAppQuery {
      artist(id: "pablo-picasso") {
        name
      }
    }
  `
}
```

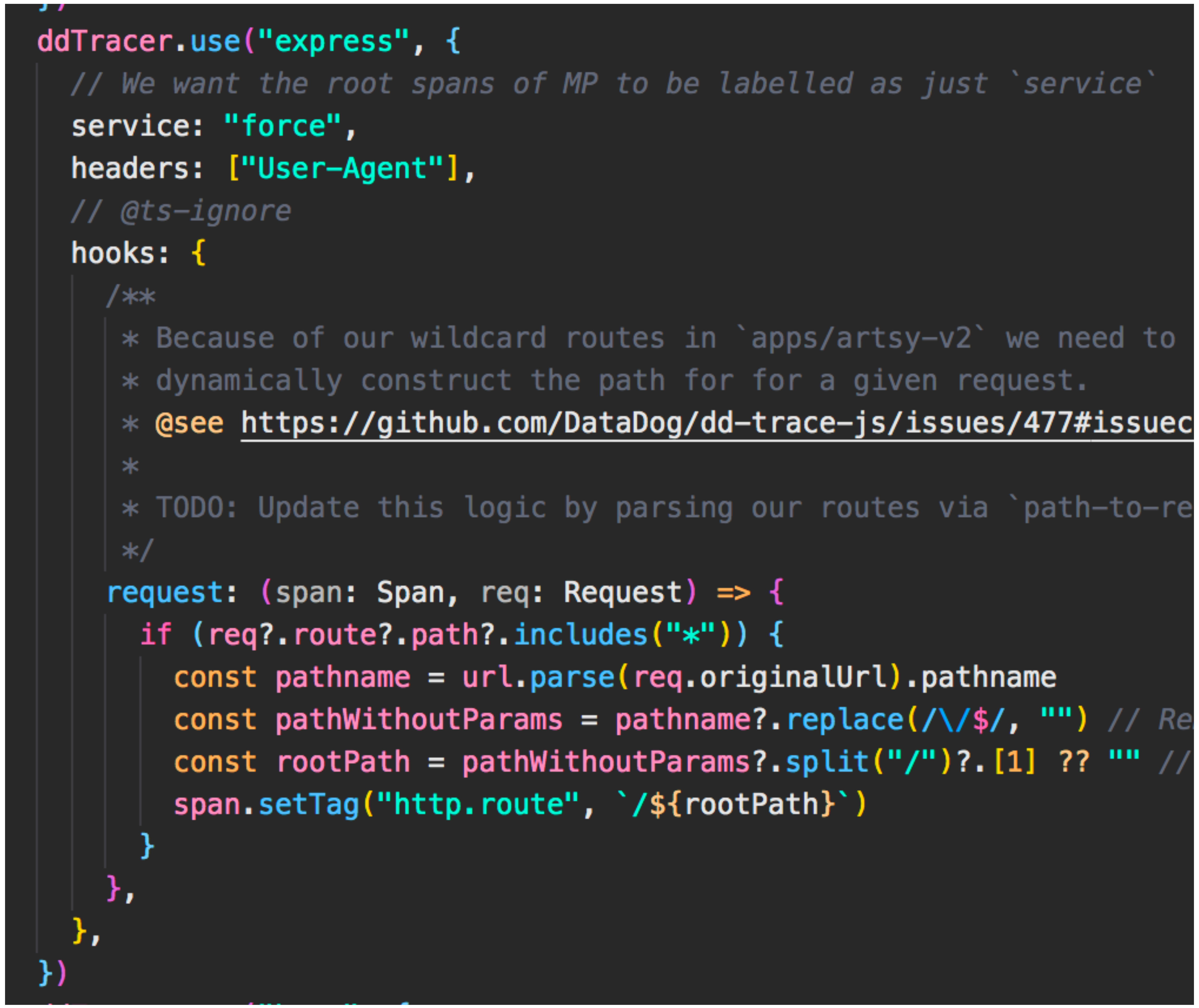
Some (opinionated) downsides to this new architecture

1. We’re now in SPA (Single Page App) territory

Things have to be thought about slightly differently, because historical thinking assumes hard jumps between pages…

…even though that kind of architecture is being seen less and less, especially on commerce-heavy websites.

Out of the box, some libs expect to be reinitialized when navigating from page to page



```
ddTracer.use("express", {
  // We want the root spans of MP to be labelled as just `service`
  service: "force",
  headers: ["User-Agent"],
  // @ts-ignore
  hooks: {
    /**
     * Because of our wildcard routes in `apps/artsy-v2` we need to
     * dynamically construct the path for for a given request.
     * @see https://github.com/DataDog/dd-trace-js/issues/477#issuec
     *
     * TODO: Update this logic by parsing our routes via `path-to-re
     */
    request: (span: Span, req: Request) => {
      if (req?.route?.path?.includes("*")) {
        const pathname = url.parse(req.originalUrl).pathname
        const pathWithoutParams = pathname?.replace(/\/$/, "") // Re
        const rootPath = pathWithoutParams?.split("/")?.[1] ?? "" //
        span.setTag("http.route", `/${rootPath}`)
      }
    },
  },
})
```

But most of the time, things can be fixed through configuration

Remember to read the docs!

And beware of FUD!

But, all this being said…

There can be some initial overhead, especially if refactoring an existing website

A “single page app” is literally a single server-rendered page, per user session. Everything else is JS, JSX, etc., rendered on the client.

If you have a bias against JS you’re not gonna like SPAs

2. Complexity

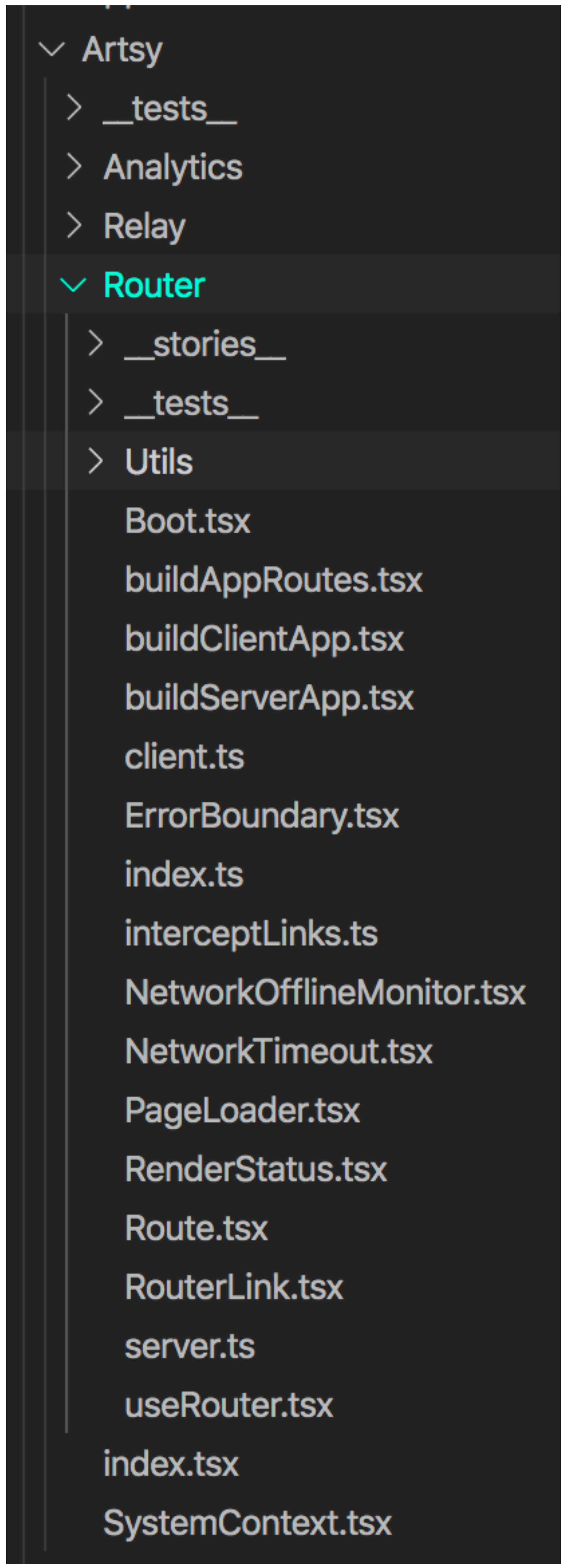
In the ‘Benefits’ slides one of the points was ‘Simplicity’. Under the hood, that took work.

In greenfield projects one would likely look to complete solutions like Next.js for handling routing, js asset bundle-splitting and SSR

However, bringing frameworks like this into older and / or larger apps often isn’t practical

We also had to deal with Relay’s lack of server-side rendering, and come up with some patterns of our own

And so we assembled a number of tools together and built a little micro framework



```
Artsy
  __tests__
  Analytics
  Relay
  Router
    __stories__
    __tests__
    Utils
    Boot.tsx
    buildAppRoutes.tsx
    buildClientApp.tsx
    buildServerApp.tsx
    client.ts
    ErrorBoundary.tsx
    index.ts
    interceptLinks.ts
    NetworkOfflineMonitor.tsx
    NetworkTimeout.tsx
    PageLoader.tsx
    RenderStatus.tsx
    Route.tsx
    RouterLink.tsx
    server.ts
    useRouter.tsx
  index.tsx
  SystemContext.tsx
```

Additionally, we needed a way to dynamically split up our JS assets between pages

JavaScript has a built in primitive asynchronously loading JS assets:

import()

This does not, however, take bundled JS and split it into files that can be loaded on demand. Webpack does this.

But webpack’s `import()` functionality doesn’t deal with SSR!

So as you can see, there’s lots of tooling here.

And this tooling is powerful and pretty low-level

Depending on who you ask,
this is a blessing or a curse

But once things are setup… it’s pretty sweet. And, at least in terms of 2020, the patterns are pretty vanilla / standard.

On the server:

```
res.send(React.renderToString(Router))
```

On the client:

React.hydrate(Router)

(Checking time)

* Code walk-through *

Thanks!