A different one person framework: Ruby on Rails / Inertia.js / Whatever

Gianpiero Addis - RUGHH - May 2025

Gianpiero Addis

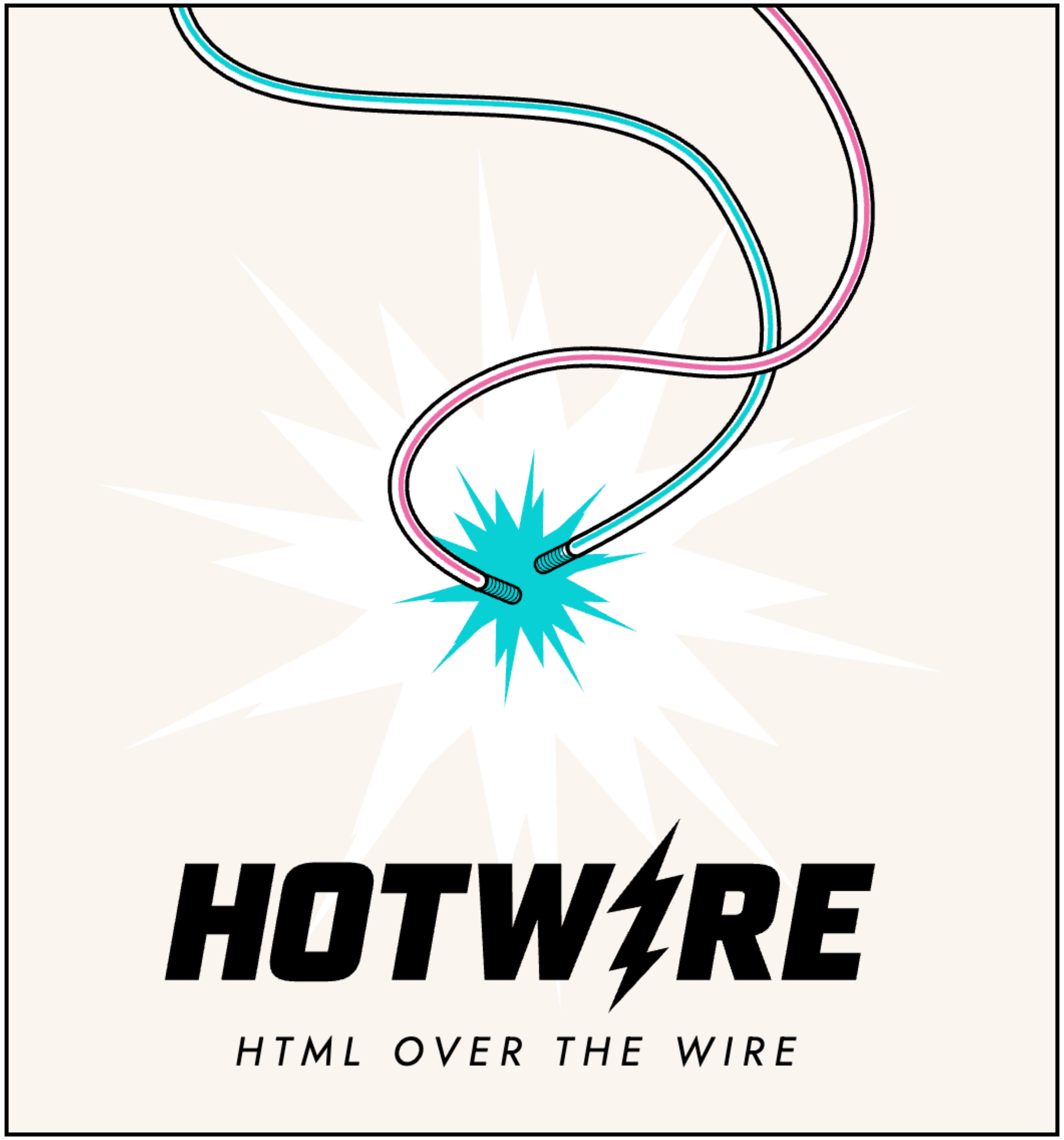
Freelance Software Developer

- Born and raised in Sardinia, moved to Hamburg in 2014
- Started programming as a kid, then forgot about it for years
- Rediscovered programming with PHP (Laravel) in 2017
- Working with Ruby / Ruby on Rails since 2020
- Freelancing full time since April 2025
- www.gpaddis.com / gpaddis

The One Person Framework

“A toolkit so powerful that it allows a single individual to create modern applications upon which they might build a competitive business. The way it used to be.

https://world.hey.com/dhh/the-one-person-framework-711e6318

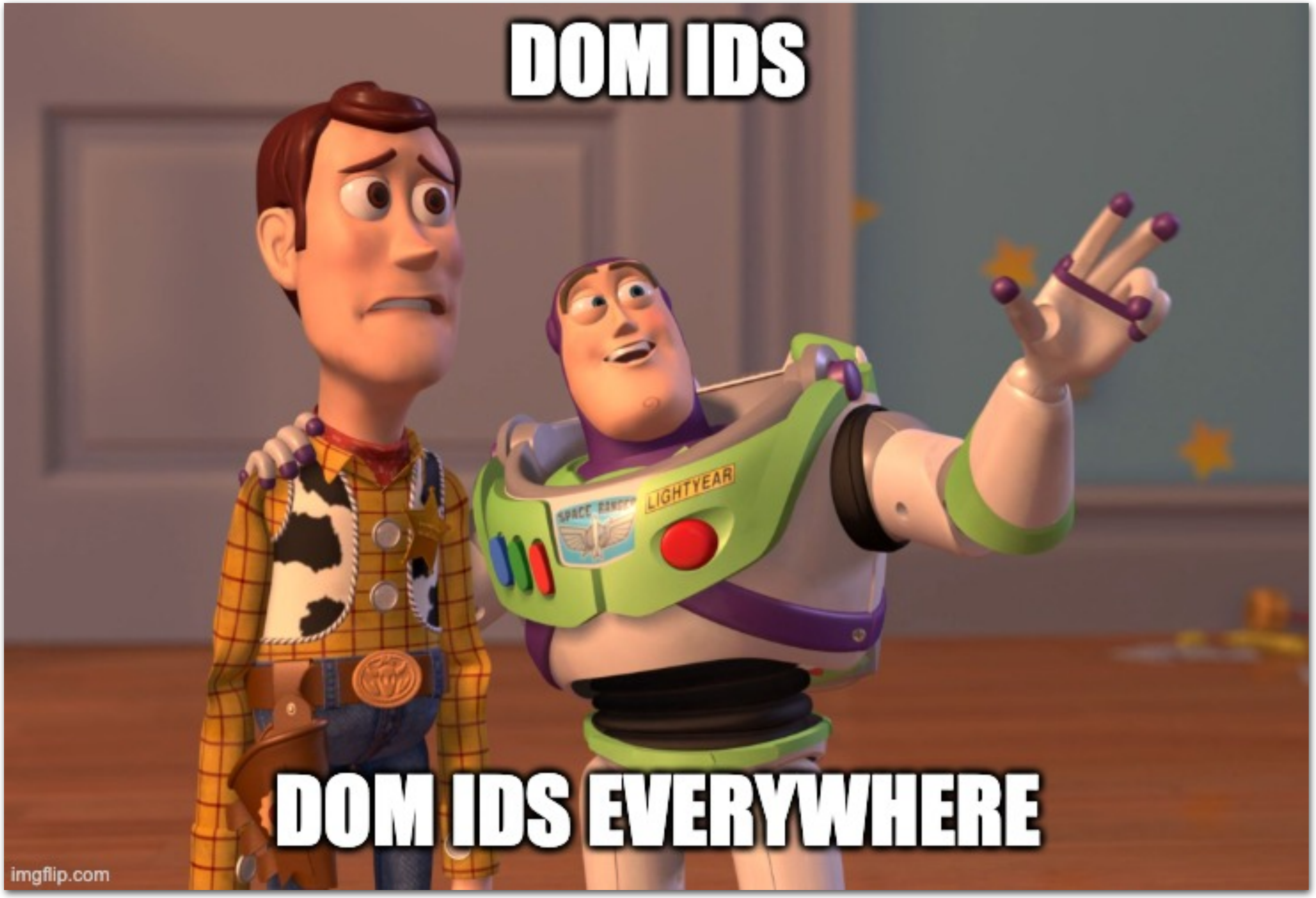

HOTWIRE
HTML OVER THE WIRE

Why not just do it “the Rails way” then?

PARENTAL
ADVISORY
EXPLICIT CONTENT

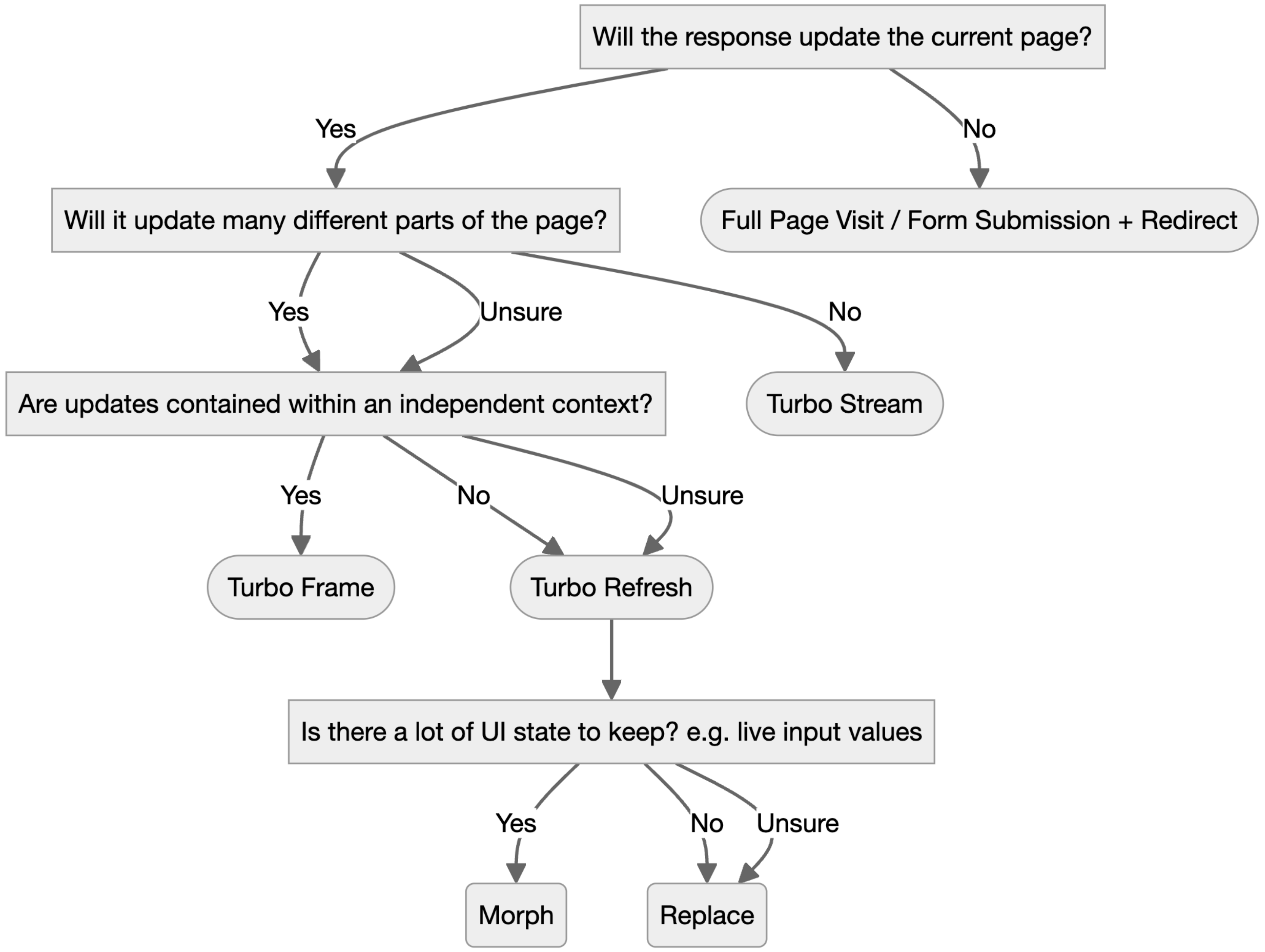
Why not just do it “the Rails way” then?

1. Kabelsalat



Why not just do it “the Rails way” then?

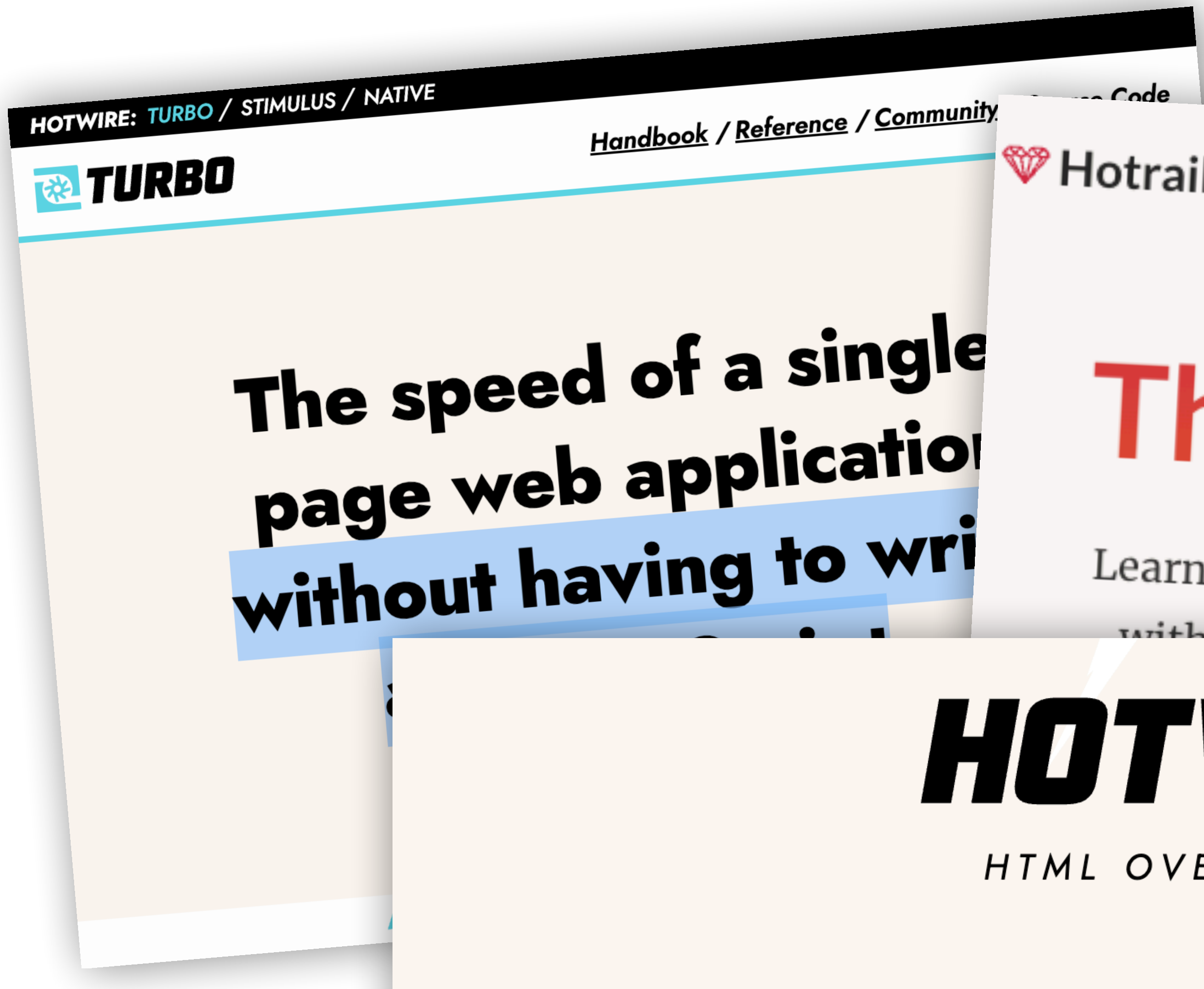
2. Complex State Management


Will the response update the current page?
Yes
No
Will it update many different parts of the page?
Full Page Visit / Form Submission + Redirect
Yes
Unsure
No
Are updates contained within an independent context?
Turbo Stream
Yes
No
Unsure
Turbo Frame
Turbo Refresh
Is there a lot of UI state to keep? e.g. live input values
Yes
No
Unsure
Morph
Replace

https://domchristie.co.uk/posts/turbo-refreshes-frames-streams/

Why not just do it “the Rails way” then?

3. JavaScript Phobia

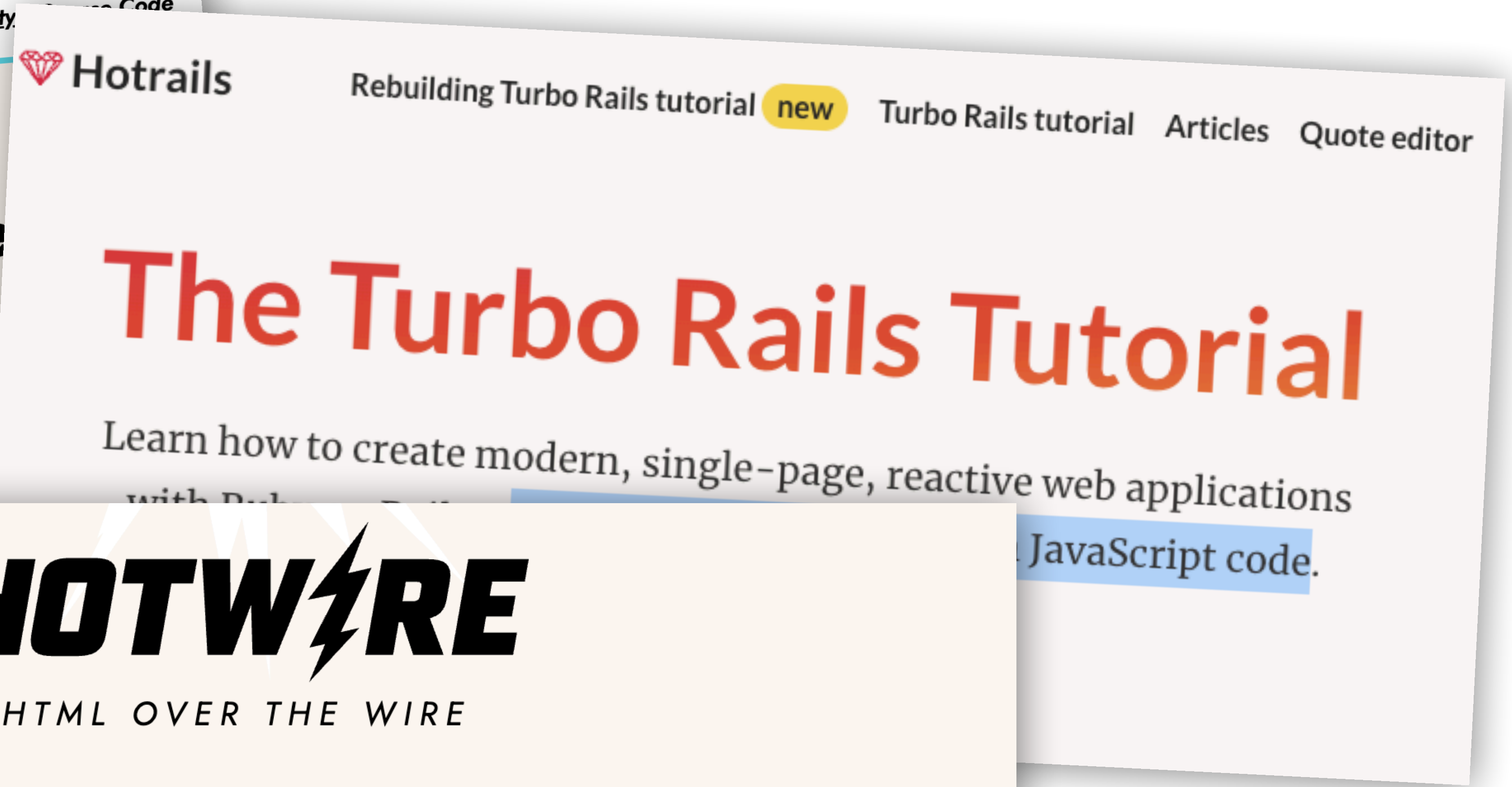


HOTWIRE: TURBO / STIMULUS / NATIVE

TURBO

Handbook / Reference / Community

The speed of a single
page web applicatio
without having to wri

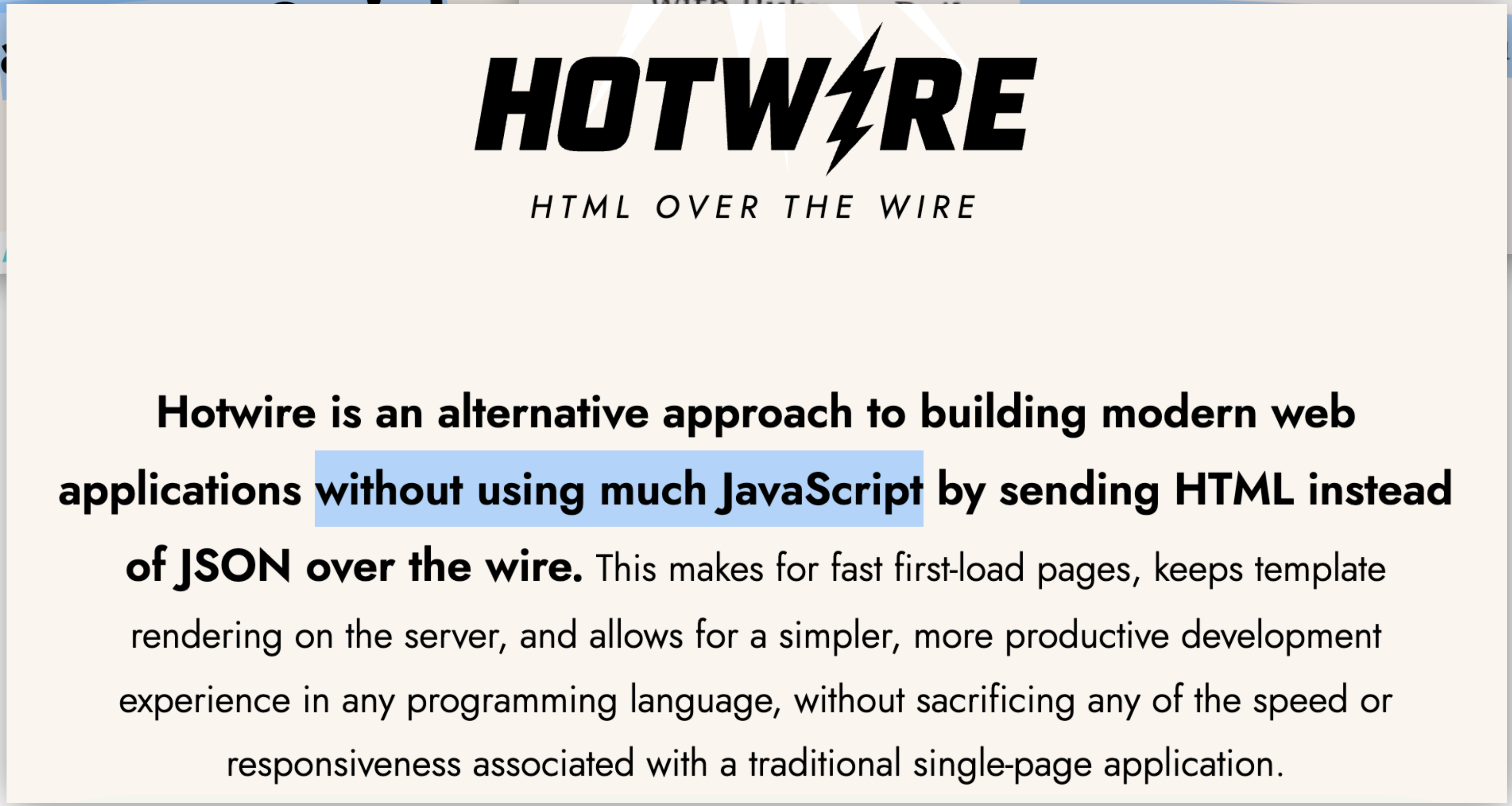


Hotrails

Rebuilding Turbo Rails tutorial new | Turbo Rails tutorial | Articles | Quote editor

The Turbo Rails Tutorial

Learn how to create modern, single-page, reactive web applications
JavaScript code.



HOTWIRE

HTML OVER THE WIRE

Hotwire is an alternative approach to building modern web applications without using much JavaScript by sending HTML instead of JSON over the wire. This makes for fast first-load pages, keeps template rendering on the server, and allows for a simpler, more productive development experience in any programming language, without sacrificing any of the speed or responsiveness associated with a traditional single-page application.

Why not just do it "the Rails way" then?

4: I never really enjoyed working with Rails' own FE

Inertia.js

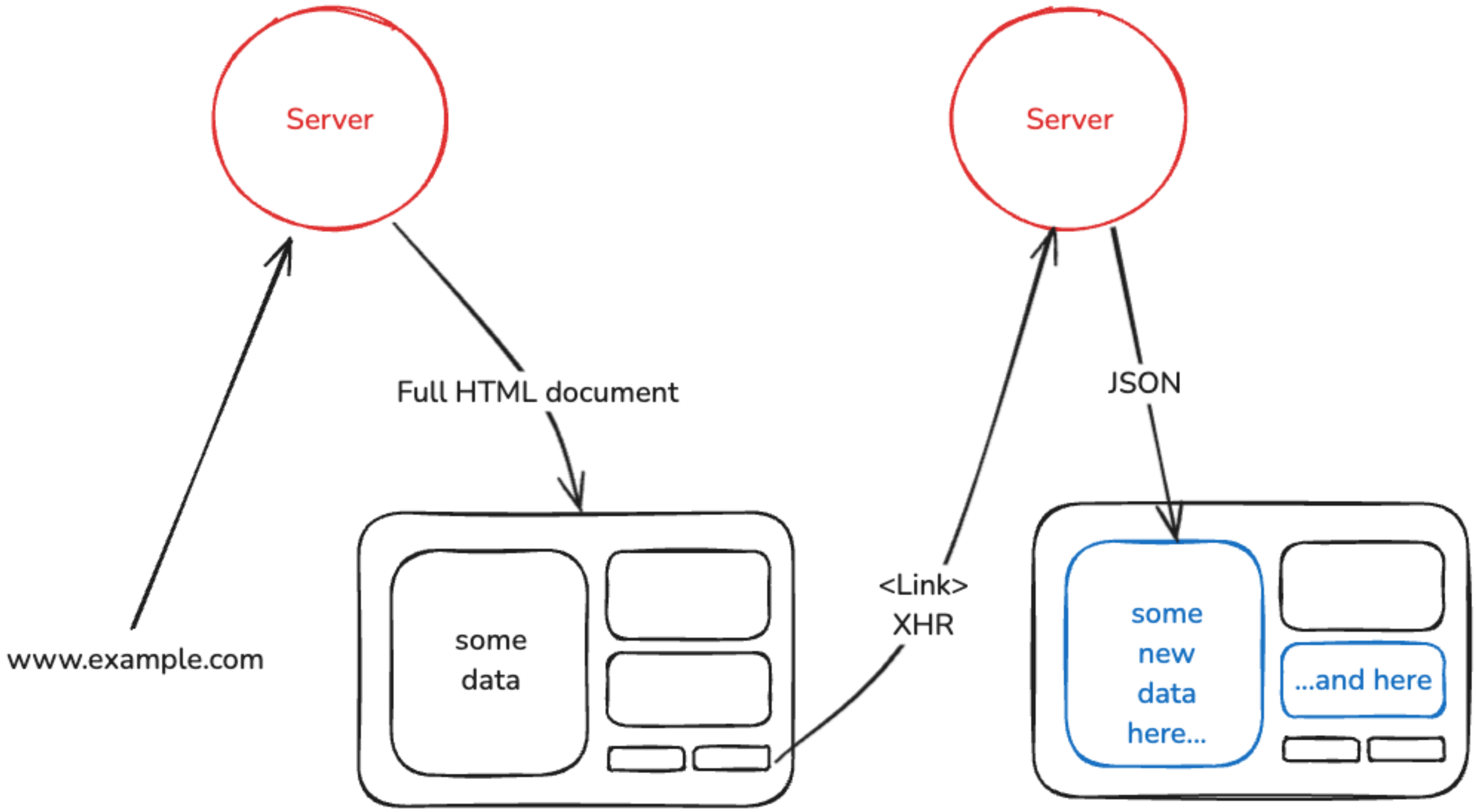
Some Facts

- Created in 2019 by **Jonathan Reinink**, one of the creators of Tailwind CSS
- Inspired by Turbolinks (intercepts click events to make XHR requests instead of full page visits)
- Immediately popular and widely adopted in the Laravel community
- With Inertia 2.0: great documentation of the Rails adapter and super easy installation and configuration with the gem inertia_rails
- inertia_rails is maintained by **Svyatoslav Kryukov** (Evil Martians), creator of turbo-mount

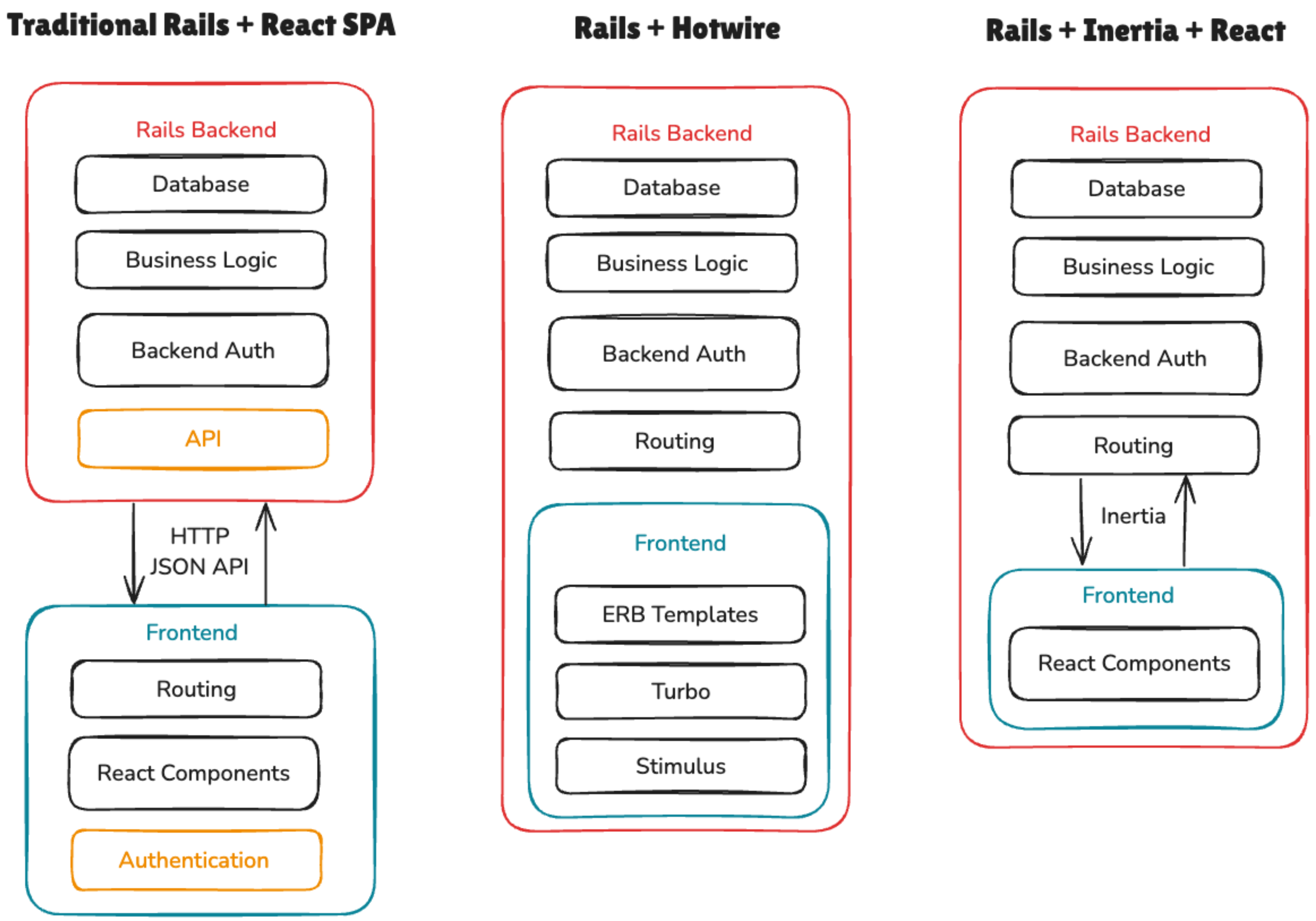
“ Inertia isn't a framework, nor is it a replacement for your existing server-side or client-side frameworks. Rather, it's designed to work with them. **Think of Inertia as glue that connects the two**. Inertia does this via adapters.

How does it work?

The protocol



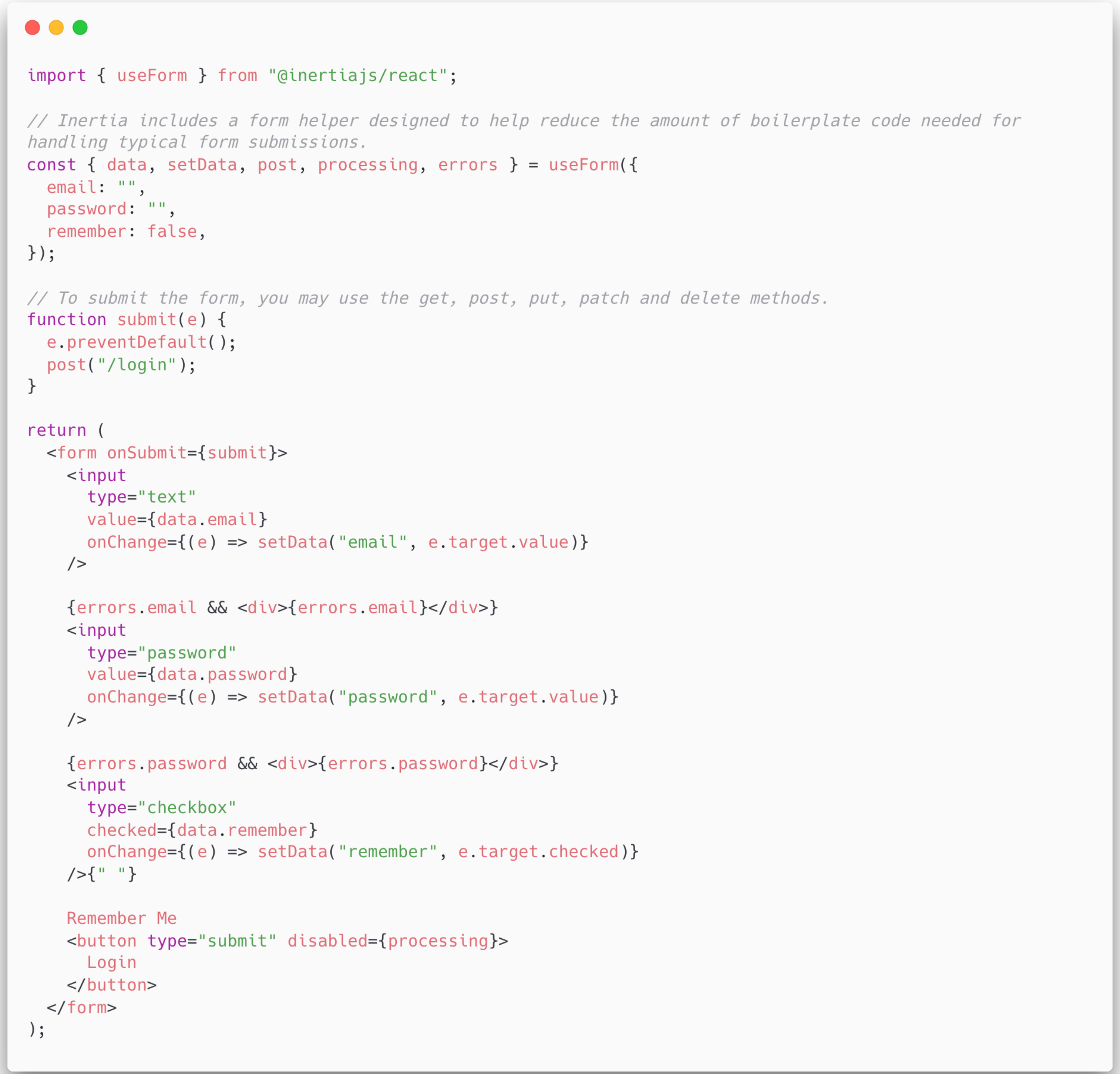
SPA / Hotwire / Inertia



Some Cool Features

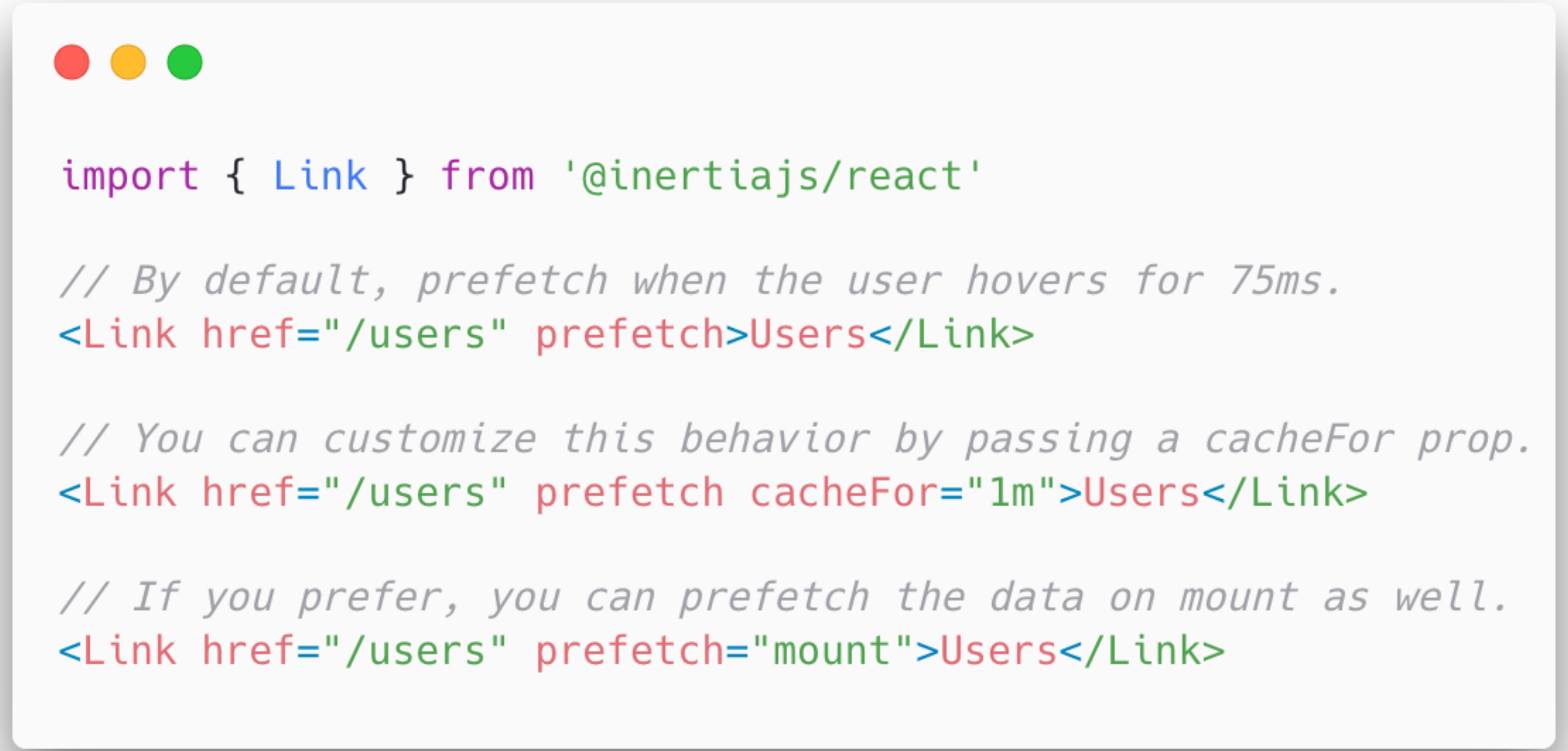
Demo

Form Helper



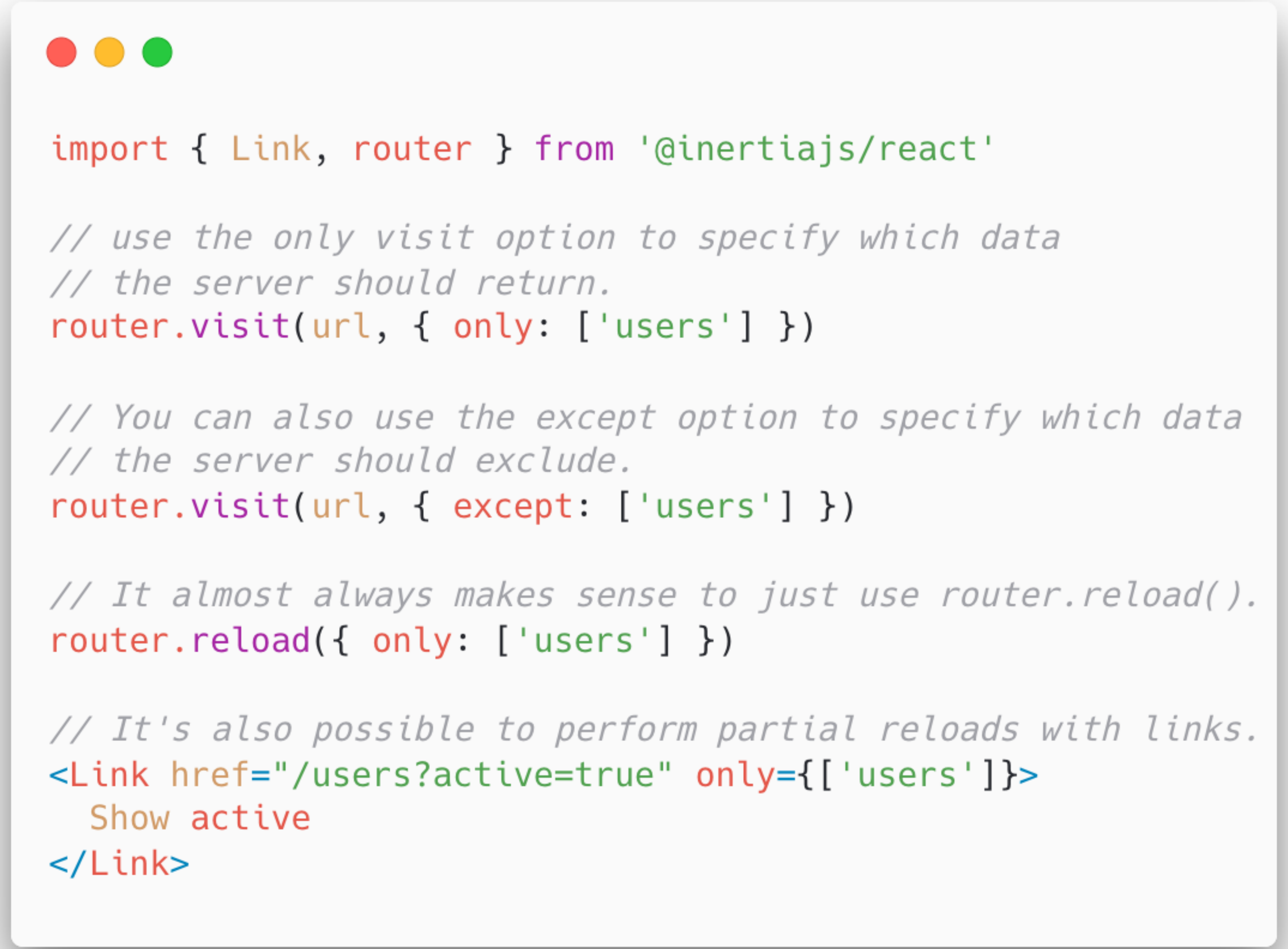
```
import { useForm } from "@inertiajs/react";

// Inertia includes a form helper designed to help reduce the amount of boilerplate code needed for
handling typical form submissions.
const { data, setData, post, processing, errors } = useForm({
  email: "",
  password: "",
  remember: false,
});

// To submit the form, you may use the get, post, put, patch and delete methods.
function submit(e) {
  e.preventDefault();
  post("/login");
}

return (
  <form onSubmit={submit}>
    <input
      type="text"
      value={data.email}
      onChange={(e) => setData("email", e.target.value)}
    />

    {errors.email && <div>{errors.email}</div>}
    <input
      type="password"
      value={data.password}
      onChange={(e) => setData("password", e.target.value)}
    />

    {errors.password && <div>{errors.password}</div>}
    <input
      type="checkbox"
      checked={data.remember}
      onChange={(e) => setData("remember", e.target.checked)}
    />{" "}

    Remember Me
    <button type="submit" disabled={processing}>
      Login
    </button>
  </form>
);
```

Prefetching



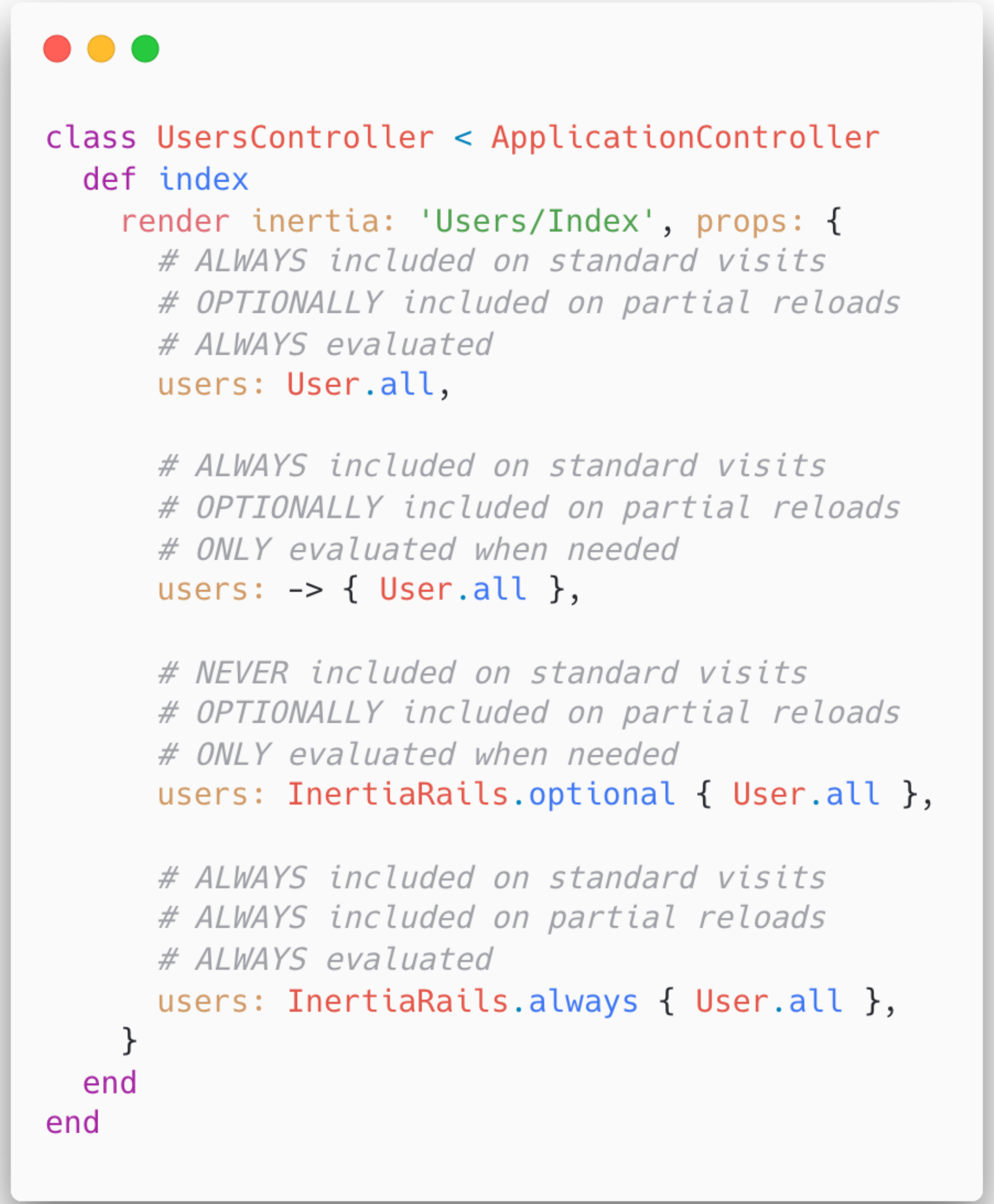
```jsx
import { Link } from '@inertiajs/react'

// By default, prefetch when the user hovers for 75ms.
<Link href="/users" prefetch>Users</Link>

// You can customize this behavior by passing a cacheFor prop.
<Link href="/users" prefetch cacheFor="1m">Users</Link>

// If you prefer, you can prefetch the data on mount as well.
<Link href="/users" prefetch="mount">Users</Link>
```

Partial Reloads - Client



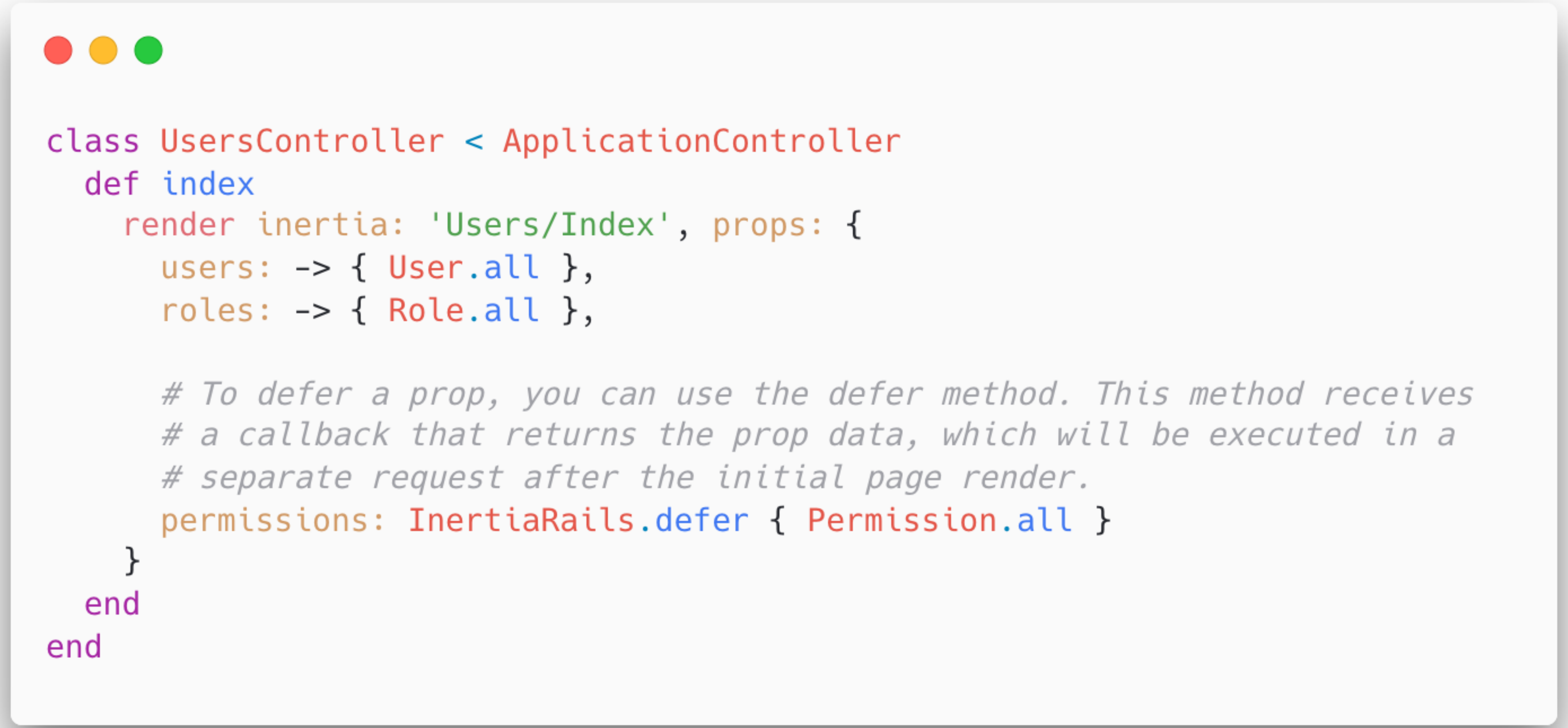
```jsx
import { Link, router } from '@inertiajs/react'

// use the only visit option to specify which data
// the server should return.
router.visit(url, { only: ['users'] })

// You can also use the except option to specify which data
// the server should exclude.
router.visit(url, { except: ['users'] })

// It almost always makes sense to just use router.reload().
router.reload({ only: ['users'] })

// It's also possible to perform partial reloads with links.
<Link href="/users?active=true" only={['users']}>
  Show active
</Link>
```

Partial Reloads - Server



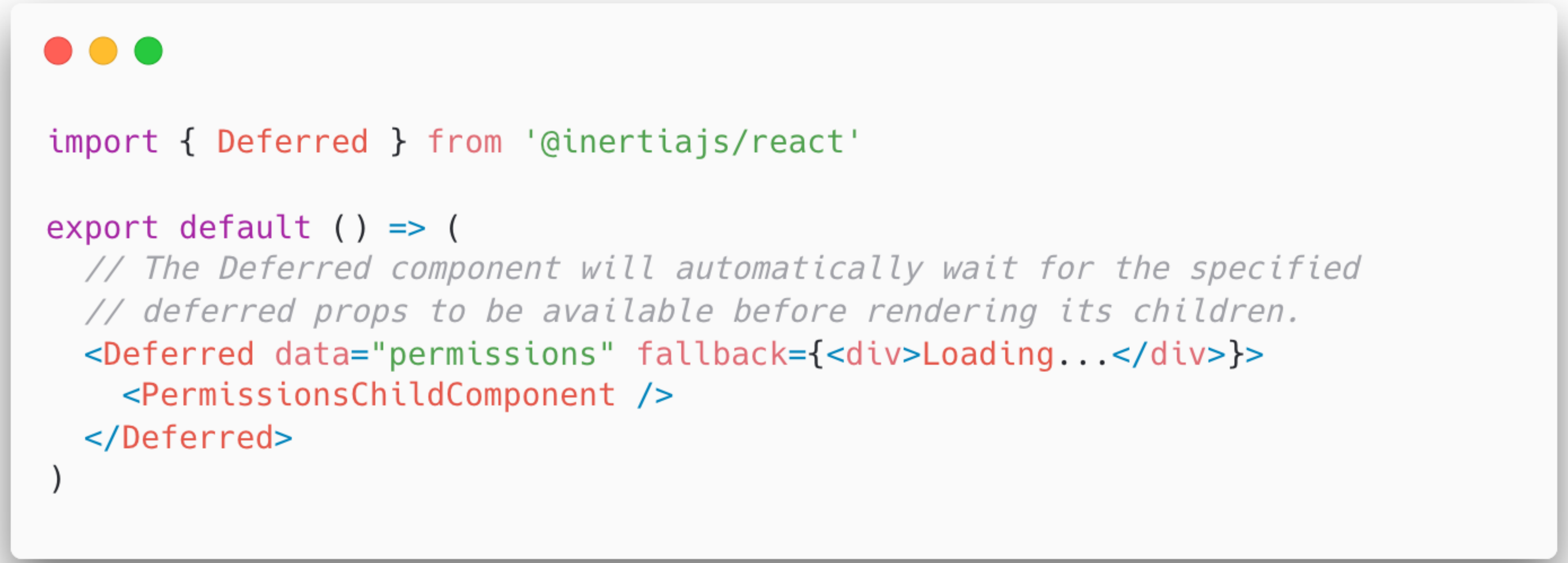
```ruby
class UsersController < ApplicationController
  def index
    render inertia: 'Users/Index', props: {
      # ALWAYS included on standard visits
      # OPTIONALLY included on partial reloads
      # ALWAYS evaluated
      users: User.all,

      # ALWAYS included on standard visits
      # OPTIONALLY included on partial reloads
      # ONLY evaluated when needed
      users: -> { User.all },

      # NEVER included on standard visits
      # OPTIONALLY included on partial reloads
      # ONLY evaluated when needed
      users: InertiaRails.optional { User.all },

      # ALWAYS included on standard visits
      # ALWAYS included on partial reloads
      # ALWAYS evaluated
      users: InertiaRails.always { User.all },
    }
  end
end
```

Deferred Props - Server



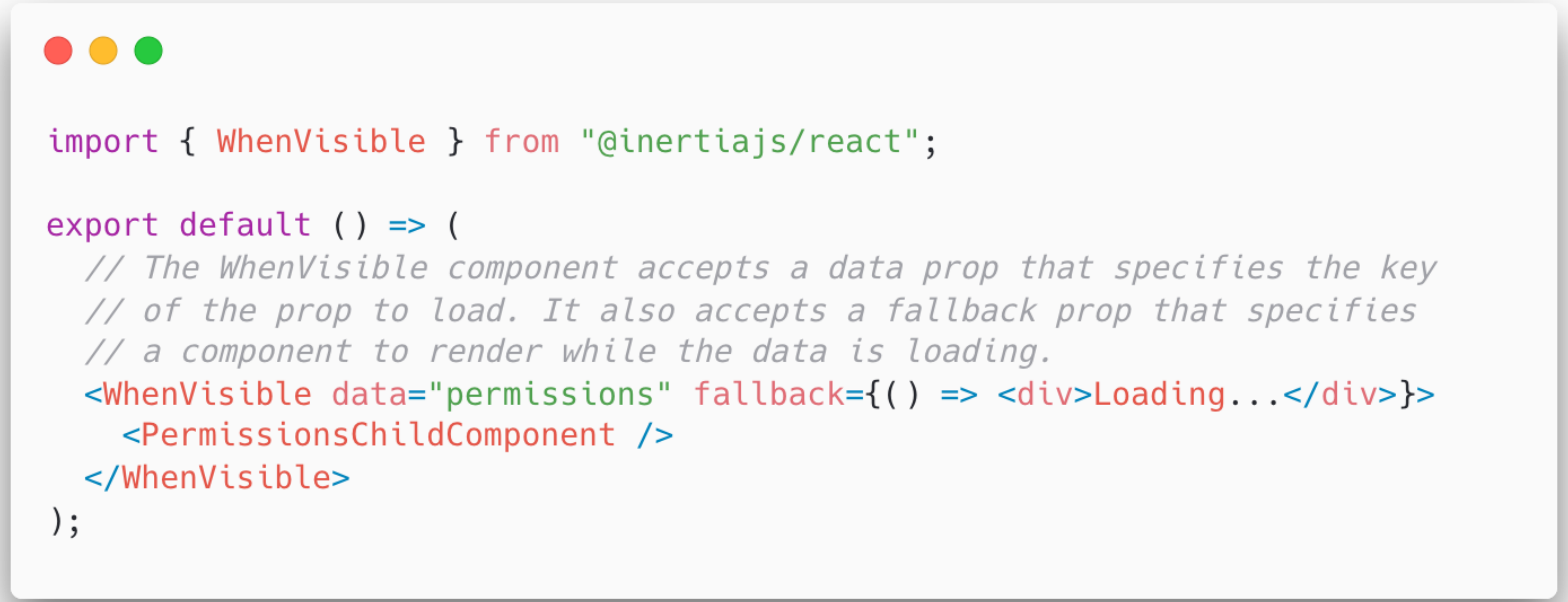
```ruby
class UsersController < ApplicationController
  def index
    render inertia: 'Users/Index', props: {
      users: -> { User.all },
      roles: -> { Role.all },

      # To defer a prop, you can use the defer method. This method receives
      # a callback that returns the prop data, which will be executed in a
      # separate request after the initial page render.
      permissions: InertiaRails.defer { Permission.all }
    }
  end
end
```

Deferred Props - Client



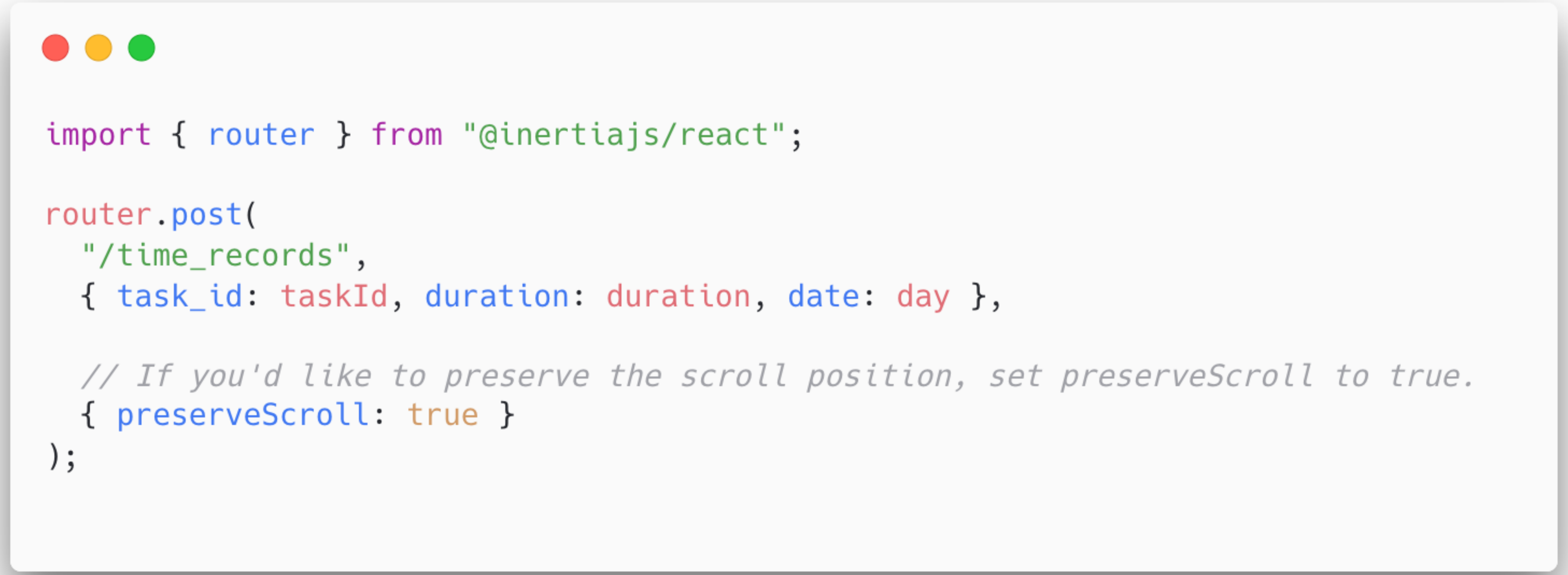
```jsx
import { Deferred } from '@inertiajs/react'

export default () => (
  // The Deferred component will automatically wait for the specified
  // deferred props to be available before rendering its children.
  <Deferred data="permissions" fallback={<div>Loading...</div>}>
    <PermissionsChildComponent />
  </Deferred>
)
```

Load When Visible

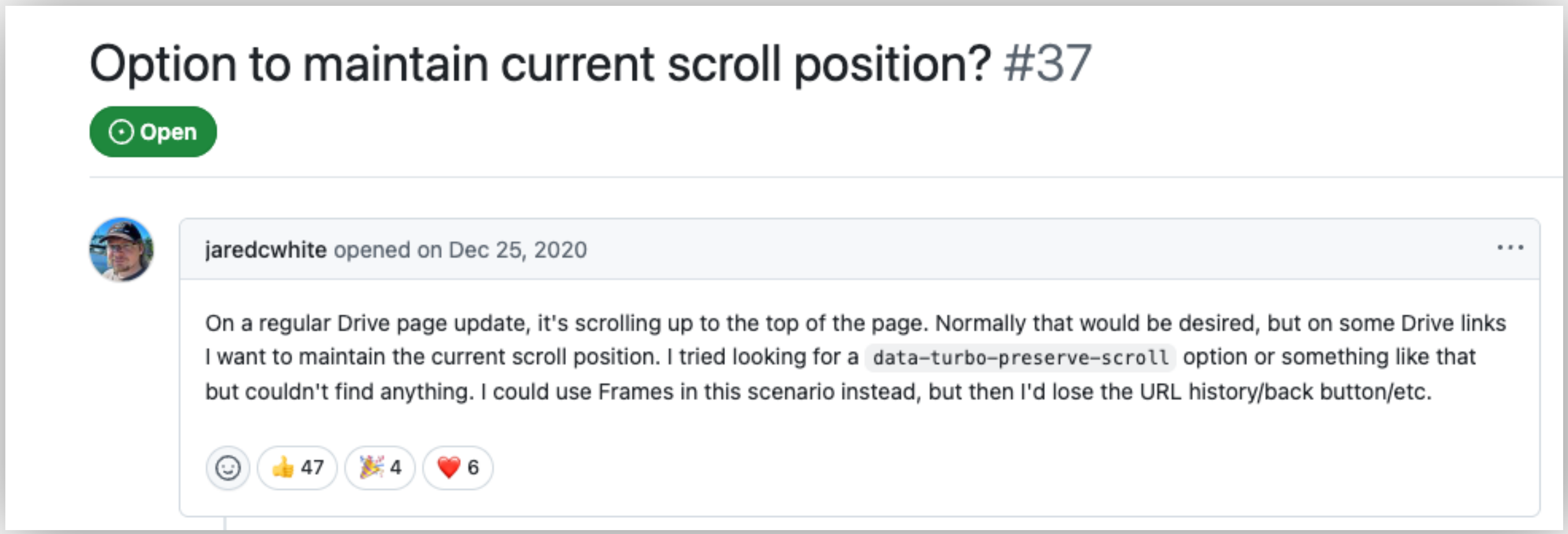


```jsx
import { WhenVisible } from "@inertiajs/react";

export default () => (
  // The WhenVisible component accepts a data prop that specifies the key
  // of the prop to load. It also accepts a fallback prop that specifies
  // a component to render while the data is loading.
  <WhenVisible data="permissions" fallback={() => <div>Loading...</div>}>
    <PermissionsChildComponent />
  </WhenVisible>
);
```

Preserve Scroll



```
import { router } from "@inertiajs/react";

router.post(
  "/time_records",
  { task_id: taskId, duration: duration, date: day },

  // If you'd like to preserve the scroll position, set preserveScroll to true.
  { preserveScroll: true }
);
```



Option to maintain current scroll position? #37

Open

jaredcwhite opened on Dec 25, 2020

On a regular Drive page update, it's scrolling up to the top of the page. Normally that would be desired, but on some Drive links I want to maintain the current scroll position. I tried looking for a `data-turbo-preserve-scroll` option or something like that but couldn't find anything. I could use Frames in this scenario instead, but then I'd lose the URL history/back button/etc.

👍 47 🎉 4 ❤️ 6

Conclusion

Some final thoughts

- Thinking in **components** feels more intuitive than thinking in frames and partials (explicit inputs, hierarchical structure).
- Having **immutable data** instead of hot AR model instances prevents unintentional N+1.
- Javascript / Typescript is great for IDE support, (AI) autocompletion, available libraries, types, documentation, debugging, etc.
- **ViewComponent** might be an alternative, but I don't really get the benefits for a small / middle sized application. Compatibility with Hotwire is also not optimal.
- **Phlex** comes with level of syntax abstraction over plain HTML and that's exactly the reason why I never liked Action View helpers, HAML & co. (please propose a talk for one of the next RUGHH and change my mind!).
- Just use **whatever you like** and makes you happy to do more coding, without overthinking the choice of the right technology!

Thank you!

www.gpaddis.com