ChatGPT

Integrating React 19 with Rails 8: A Comprehensive Guide

This guide covers how to integrate **React 19** into a **Rails 8** application using **TypeScript and TSX**. We will explore two approaches to integration (native Rails vs. separate front-end), discuss server-side rendering (SSR) considerations, outline Docker/Kubernetes deployment best practices, and examine how **StimulusJS** can coexist with React. The guide is structured into sections with clear headings, code snippets, and comparison tables for easy scanning.

1. Native Rails 8 Integration (esbuild vs. importmap)

Rails 8 (like Rails 7) embraces modern JavaScript tooling by moving away from Webpacker in favor of simpler alternatives. The two primary ways to use React within a Rails app without Webpacker are:

- **Using** `jsbundling-rails` **with esbuild** – a Node-based bundler pipeline.
- **Using** `importmap-rails` – a bundler-free approach where the browser imports modules directly.

Both can be used to set up React 19 with TypeScript/TSX, but the configuration and capabilities differ significantly.

1.1 Rails + esbuild (jsbundling-rails) with TypeScript/TSX

Using **esbuild** via the `jsbundling-rails` gem is a straightforward way to bundle React and transpile TypeScript/TSX:

- **Install jsbundling-rails and initialize esbuild:** Add the gem and run the installer. For example:

```
bundle add jsbundling-rails
rails javascript:install:esbuild
```

This sets up an esbuild config, an empty `app/javascript/application.js` (or `.tsx`) entry point, and updates your Rails config to serve bundled output from `app/assets/builds` [1]. The installer also adds a build script to your **package.json** (e.g. `esbuild app/javascript/*.* --bundle --sourcemap --outdir=app/assets/builds --public-path=assets` by default [2]).

- **Install React 19, TypeScript, and related packages:** Using Yarn or npm, add React, ReactDOM, TypeScript, and type definitions. For example, in the app directory:

```
yarn add react@19 react-dom@19 @types/react @types/react-dom typescript
```

This brings in React 19 and the TypeScript compiler and types. (Adjust versions as needed once React 19 is officially released; React 18 uses similar steps [3].)

- **Create a TypeScript config:** Add a **tsconfig.json** at the project root to configure TypeScript compilation. For a Rails+esbuild setup, a basic config is:

```
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "es2016",
    "jsx": "react",
    "esModuleInterop": true,
    "strict": true,
    "forceConsistentCasingInFileNames": true,
    "skipLibCheck": true
  }
}
```

Key setting: `"jsx": "react"` enables TSX syntax for React [4]. (You might use `"jsx": "react-jsx"` with React 17+, but either works with proper runtime settings.)

- **Define an entry-point TSX file:** For example, create `app/javascript/application.tsx` (or place it in an entrypoints folder as suggested by some for organization [5]). In this file, you can write React code with TSX. For instance:

```
import React from 'react';
import { createRoot } from 'react-dom/client';

const App: React.FC = () => <h1>Hello from React 19!</h1>;

const container = document.getElementById('root');
if (container) {
  const root = createRoot(container);
  root.render(<App />);
}
```

This simple component will render a heading. Note how we use the modern `createRoot` API from ReactDOM (introduced in React 18) to enable React's concurrent mode features.

- **Include the pack in Rails views:** Ensure your Rails layout loads the bundled JS. In Rails 8 with esbuild, the installer will typically have inserted a helper like:

```
<%= javascript_include_tag "application", "data-turbo-track": "reload", defer: true %>
```

This pulls in `app/assets/builds/application.js` (generated by esbuild). Also, add a mounting point in your HTML, e.g. `<div id="root"></div>` in the layout or specific view, as the container for the React app.

- **Build and verify:** Run `yarn build` (which runs the esbuild bundler). Rails will output the compiled JavaScript to `app/assets/builds`. On loading the page, you should see "Hello from React 19!" rendered, confirming TSX compilation works. Esbuild handles bundling and transpiling of TSX out of the box (it uses the esbuild TypeScript loader by default for `.ts` / `.tsx` files). The Rails asset pipeline will serve the final bundle.

Why esbuild? Using esbuild brings the benefit of fast builds and the ability to use TSX, ES6+ etc. with minimal config. Rails just treats the output as static assets to serve [1]. This approach does require Node/Yarn in your build process, but is officially supported and "just works" for React+TS. It's well-suited for larger or more complex front-ends.

Example: The snippet below shows a minimal React+TSX setup with esbuild – an App component and its mounting logic:

```
// app/javascript/application.tsx
import React from "react";
import ReactDOM from "react-dom/client";

// Simple React component (using TSX and a functional component)
const App: React.FC = () => <h2>Hello from React 19 in Rails 8!</h2>;

// Find the container and mount the React component
const rootElement = document.getElementById("root");
if (rootElement) {
  const root = ReactDOM.createRoot(rootElement);
  root.render(<App />);
}
```

When Rails serves a view containing `<div id="root"></div>` and includes the compiled `application.js`, the React App will load. The `jsbundling-rails` default config ensures everything in `app/javascript` (or your entrypoints) gets bundled into a file Rails can serve [1].

1.2 Rails + importmap (no bundler) with TypeScript/TSX

Rails introduced **import maps** to allow using JavaScript without a build process. Importmap lets you "pin" JavaScript packages (like React) and directly import them in your code, which the browser resolves (either from local assets or a CDN). This eliminates Node and bundlers at runtime. However, **importmap has no built-in transpilation**, so using TypeScript/TSX requires additional steps [6].

Key point: *"Import map serves the JavaScript directly without compilation. TypeScript is compiled. So if you want TS, you'll need to use jsbundling-rails."* [6]. – In other words, out-of-the-box importmaps expect plain JS. To use TSX, you must introduce a transpilation layer *somewhere*.

That said, it is possible to use importmap with TS/TSX by leveraging the Rails asset pipeline (Sprockets) or third-party tools:

- **Pin React and dependencies in importmap:** First, ensure React is available to the browser. In `config/importmap.rb`, you would add entries for React 19, ReactDOM, etc. For example:

```
pin "react", to: "https://ga.jspm.io/npm:react@19.0.0/index.js"
pin "react-dom", to: "https://ga.jspm.io/npm:react-dom@19.0.0/index.js"
pin "react/jsx-runtime", to: "react--jsx-runtime.js"  # if using the JSX runtime
```

(The `pin` command can be done via `bin/importmap pin react@19 react-dom@19` which fetches the exact URLs.) These lines ensure that when you import `"react"` in your code, the browser will fetch the React 19 UMD module from JSPM (or another source). You can also host the files locally by downloading them, but CDN is common for importmaps.

- **Write your React code modules** (with TSX) in `app/javascript` as usual. For example, `app/javascript/controllers/my_component.tsx` or similar.

- **Transpile TSX to JS for the browser:** Since importmap won't transpile, you have two main options:

- **Use Babel/Sprockets to compile on-the-fly or at assets:precompile time.** A community approach (as shown by George Gavrilchik [7]) is to register a custom Sprockets transformer for `.tsx` files. This involves using the `nodo` gem (which runs Node scripts via Sprockets) with Babel. For example, one can configure a `TSXCompiler` that uses `@babel/plugin-transform-react-jsx` and `@babel/plugin-transform-typescript` to convert TSX into plain JS when assets are served or precompiled [8] [9]. Below is a simplified initializer illustrating this:

```
# config/initializers/sprockets_tsx.rb
require "nodo/core"
class TSXCompiler < Nodo::Core
  # Require Babel and necessary plugins
  require babelCore: '@babel/core',
          pluginTransformTypescript: '@babel/plugin-transform-typescript',
          pluginTransformReactJSX: '@babel/plugin-transform-react-jsx'
  class << self
    def call(input)
      source = input[:data]
      result = babelCore.transformSync(source, {
        "plugins": [
          ["@babel/plugin-transform-react-jsx", { "runtime": "automatic" }],
          ["@babel/plugin-transform-typescript", { "isTSX": true, "allExtensions": true }]
        ]
      })
      { data: result["code"] }
    end
  end
end
```

```
# Register .tsx and .ts to be processed by our compiler
Sprockets.register_mime_type 'text/tsx', extensions: %w(.tsx .ts), charset: :unicode
Sprockets.register_transformer 'text/tsx', 'application/javascript', TSXCompiler
```

This tells Sprockets to treat `.ts` / `.tsx` files as a custom MIME type and to transform them into JavaScript using Babel (with TSX support) [8] [9]. With this in place, you can include your `.tsx` modules via importmap and they will be compiled to JS as needed. Keep in mind that in development you might need to tweak importmap's file watcher to detect changes in `.jsx/.tsx` (by default it watched only `.js` files, requiring a cache-busting workaround [10]).

- **Alternatively, use the TypeScript compiler (tsc) as a build step** to transpile your files to plain `.js` (which could be output to `app/javascript` or `app/assets`). Essentially, you'd run `tsc` separately (perhaps via a rake task or npm script) whenever you update the code or as part of deployment, generating equivalent `.js` files that importmap can serve. This is a more manual approach and begins to resemble using a bundler (but without actual bundling).

- **Include and use the React components in Rails views:** With importmap, you might not have a single "bundle" file. Instead, you directly import modules in a script tag. For example, in your layout you might have:

```
<%= javascript_importmap_tags %>
<script type="module">
  import { createRoot } from "react-dom";
  import React from "react";
  import App from "./components/app.tsx";  // your TSX module (will be transpiled via Sprocket
  const container = document.getElementById('root');
  if (container) {
    createRoot(container).render(React.createElement(App));
  }
</script>
```

The `javascript_importmap_tags` helper will output the necessary `<script type="importmap">` with all your pinned packages listed. The inline module script then uses ES module imports to load React, ReactDOM, and your component. The browser will fetch those as needed (React from CDN, your app code from your server, transpiled to JS).

Pros and Cons: Using importmap keeps things *Node-free* in production (aside from any build/precompile step). It's conceptually simpler (no bundling, you just write modules). However, **TypeScript integration is not native** – as we saw, it needs either a custom pipeline or switching to plain JS. DHH (Rails creator) suggests using plain JS if you want to stick to importmap [6]. Importmap is great for small "sprinkles" of JS and quick setups, but for large-scale React apps or heavy TS usage, the esbuild approach is usually less hassle.

Summary – esbuild vs. importmap in Rails 8:

Aspect	Rails + esbuild (jsbundling)	Rails + importmap
Transpilation	Built-in via esbuild (supports JSX/TSX out of the box).	No built-in transpiler. Requires Babel/Sprockets or manual tsc to use TS/TSX [6].
Bundling	Yes – bundles modules into few files for efficiency.	No bundling – browser loads each module as import.
Setup Complexity	Needs Node/Yarn and initial config (`jsbundling-rails` setup). Rails new can generate this.	Minimal initial setup (just pin packages), but TSX configuration is **more** involved due to lack of native support.
Performance	Fast builds (esbuild is very fast) and efficient delivery (one HTTP request for the bundle).	No build step overhead. In dev, changes may require page reload; multiple HTTP requests for modules (mitigated by HTTP/2).
Use case suitability	Best for rich front-ends or when using extensive React/TS features.	Good for simple enhancements or smaller apps, especially if sticking to JS. TSX usage makes it less straightforward.

Note: You can mix approaches – for example, some Rails apps might use importmap for most JS but add jsbundling for React components. However, this is advanced and usually one chooses one path or the other [11]. In general, if you need TypeScript and React, using the esbuild pipeline is the path of least resistance.

2. Separate Frontend Architecture (React vs. Vue Front-Ends with a Rails 8 Backend)

Instead of embedding React in the Rails project, an alternative is a **decoupled architecture**: Rails 8 serves as a JSON API (and perhaps basic HTML shell), while a separate front-end application (built with React or Vue) handles the UI. This section breaks down two popular React approaches (Next.js and a Vite-powered SPA) and compares them with their Vue counterparts (Nuxt.js and a plain Vue SPA).

In a separate front-end scenario, Rails is typically responsible for data and business logic (and maybe initial HTML template), whereas the front-end is a standalone project responsible for rendering the UI and interacting with Rails via HTTP. Rails might still render the initial HTML (especially if doing server-side rendering or to include base layout), or we might serve the front-end's static HTML directly. We'll explore both arrangements.

2.1 React 19 + Next.js (SSR Front-End) with a Rails 8 API Backend

Next.js is a popular React framework that supports server-side rendering, static site generation, routing, and more out of the box. Using Next.js for the front-end while Rails 8 acts as a backend API is a powerful combination:

- **How it works:** In this setup, the user's browser initially requests a page from the Next.js application (running on a Node.js server). Next.js will **server-render** the React 19 components on the fly (or serve a pre-built static page), often by fetching data from the Rails backend during the rendering phase. The browser gets a fully formed HTML page (with content), which React then

hydrates to a client-side application. The Rails app exposes endpoints (REST or GraphQL) that Next.js calls to get data.

- **Rails as origin vs. Next as origin:** Typically, in a Next + Rails architecture, **Next.js is the web server that the user directly hits** (Rails is not serving the HTML; it's providing data). The question mentions *"Rails as the origin for initial HTML"* – you could achieve this by having Rails render a basic HTML that includes a Next.js server response, but usually it's simpler to let Next.js handle the web request directly. Another approach is to use a reverse proxy: user hits Rails, Rails internally proxies to Next to get the HTML. While possible, this adds complexity. In practice, it's more common to consider Next.js as the origin for web pages (with Rails just providing JSON). We'll focus on that architecture.

- **Data fetching with SSR:** Next.js can fetch data on the server side using functions like `getServerSideProps` or `getInitialProps` (for older versions) in pages, or using React 18+ **Server Components** in the new app directory. For example, a Next.js page might do:

```
// pages/products.tsx (React component for a Products page)
import React from 'react';
import { GetServerSideProps } from 'next';

const ProductsPage = ({ products }) => {
  return (
    <div>
      <h1>Products</h1>
      {products.map(p => <div key={p.id}>{p.name}</div>)}
    </div>
  );
};

export const getServerSideProps: GetServerSideProps = async () => {
  const res = await fetch(`${process.env.RAILS_API_URL}/products.json`);
  const products = await res.json();
  return { props: { products } };
};

export default ProductsPage;
```

In this example, when a request comes in for `/products`, Next will call the Rails API (`/products.json`) on the server, get the JSON, then render the React component to HTML string with that data, and send it to the browser. The user sees a list of products in the HTML without waiting for client-side JS. Once loaded, React 19 hydrates the page and subsequent navigation or interactions can be handled client-side (using Next's client-side routing or further API calls).

- **SSR benefits:** The advantage of this SSR approach is a **faster initial load and better SEO**, since content is present in the initial HTML [12]. React 19 enhances this with features like **streaming rendering** (the ability to progressively send HTML chunks as the server renders, which can improve Time To First Byte) and **improved concurrent rendering**, making SSR even more efficient [13]. Next.js abstracts most of this – you just write React components and data fetching logic, and it handles the streaming/hydration under the hood.

- **Client-side interaction:** After the initial render, the Next.js app can use the Rails API for any client-side data updates (e.g., using `fetch` or libraries like SWR or React Query). Rails might provide endpoints for creating or updating data which the React app calls as needed. Essentially, Rails becomes a pure **REST API** (or GraphQL endpoint) with no knowledge of the front-end's internal rendering.

- **TypeScript:** Next.js has first-class TypeScript support. By initializing your Next app with TypeScript (`npx create-next-app@latest --typescript`), you get a `tsconfig.json` and can write all components in TS/TSX. React 19's types (e.g., new types for concurrent features) will be picked up via `@types/react` etc., similar to any React TS project.

- **Integration details:** You will need to configure CORS on the Rails side (so that the Next.js server or browser can request resources from it). If Next is served on a different domain (say `frontend.myapp.com` vs Rails at `api.myapp.com`), ensure CORS headers allow that. Another approach is to deploy Next and Rails under the same domain (different paths) via a reverse proxy or Kubernetes ingress (we'll discuss in deployment).

- **Nuxt.js comparison:** The Vue equivalent is **Nuxt.js** (Nuxt 3 for Vue 3). It works very similarly: Nuxt can server-render Vue components, fetching data from Rails in the process. From Rails' perspective, it doesn't matter if the front-end is React or Vue – it just serves JSON. Nuxt uses Vue's SSR mechanism to produce HTML. Both Next and Nuxt support features like routing, code-splitting, and prefetching out of the box. In terms of performance and SSR, React 19 and Vue 3 are comparable; React 19's streaming SSR has an analogue in Vue's SSR as well (with Nitro engine in Nuxt3). If your team prefers Vue, Nuxt would be the SSR choice.

When to choose Next (React SSR) or Nuxt (Vue SSR)? If SEO, initial load performance, and a unified server-rendered experience are top priority – for example, content-heavy sites, e-commerce, marketing pages – an SSR front-end framework is ideal. Rails can focus on providing a clean JSON API and maybe handle other responsibilities (like authentication, which can still be via session cookies if Next proxies requests, or via tokens). The trade-off is increased complexity: you now have a Node.js server to manage alongside Rails.

2.2 React 19 + Vite (Single-Page Application) with Rails 8 Backend

Another approach is building a **single-page application (SPA)** in React (or Vue) and using Rails strictly as an API. Here, **React 19 + Vite** is a great combination. **Vite** is a fast build tool that can scaffold a React app (including TypeScript support) and replace older tools like Create React App or Webpack. In this setup:

- **How it works:** The React app is built as a standalone bundle (HTML, JS, CSS) which is served to the user. The initial HTML might just be a minimal shell (e.g., a `<div id="root"></div>` and script tags). The actual content is rendered client-side by React, which on startup will call the Rails API to load any initial data. Essentially, the rendering is **all client-side (CSR)** – the browser does the work after getting an empty or loading page.

- **Serving the app:** There are a few options to serve the React SPA:

- You can host it on a separate static server (like using Vite's dev server in development, and in production serve the `dist` files via Nginx or Vercel or S3/CloudFront, etc.).

- Or serve it via Rails itself by leveraging the Rails public folder or a controller action. For instance, after you run `vite build` and get an `index.html` and asset files, you could put them in `public/` and Rails can be configured to serve that file for any unmatched route. A common pattern is to set a catch-all route in Rails that renders the front-end's index file:

```
# config/routes.rb (Rails)
get '*path', to: 'spa#index', constraints: ->(req) { !req.path.starts_with?('/api') }
```

  Then the `SpaController#index` simply reads `public/index.html` (the React app entry) and renders it. This way, navigating to `https://myapp.com/some/page` goes through Rails, but Rails just returns the React app's HTML, which then boots up and uses the browser URL to display the correct page (React router handles it).

- Alternatively, use Rack middleware or Nginx to serve the static files for the front-end, while Rails listens on `/api` paths.

- **Initial HTML and SSR:** In this SPA approach, the initial HTML does **not** contain the rendered content (no SSR). It may contain a loading spinner or just the framework scripts. Thus, the user sees either a blank page or a loading indicator until the JavaScript loads and executes. This means slower perceived load and poor SEO (search crawlers will mostly see an empty page, unless you use dynamic rendering or pre-rendering techniques). For many applications (internal dashboards, apps behind login, etc.), this is acceptable. For public-facing content sites, this is a downside.

- **Data fetching:** Once the React app is running, it will call the Rails API (e.g., using `fetch` or Axios) to retrieve data. For example, a React component might use an `useEffect` hook to load initial state from Rails:

```
useEffect(() => {
  fetch("/api/products.json")
    .then(res => res.json())
    .then(data => setProducts(data));
}, []);
```

  If you packaged your API as JSON endpoints, this is straightforward. For more complex state management, you might use Redux or React Context, but conceptually the API calls are the same.

- **TypeScript and Vite:** Vite supports TypeScript without extra config. If you initialize a project (e.g., `npm create vite@latest` and choose React + TypeScript template), you get a `tsconfig.json` and can write TSX. Vite's dev server also supports Hot Module Replacement for a great dev experience. It will bundle your app (using esbuild/Rollup under the hood) into static files for production.

- **Vue alternative:** Instead of React+Vite, one could use **Vue 3** with Vite (or the older Vue CLI which now also uses Vite in Vue 3). The story is the same: a pure client-side app. Vue's ecosystem (Pinia for state, Vue Router for routing, etc.) parallels React's (Redux or context, React Router, etc.).

- **When to choose SPA (React or Vue):** This approach shines for web applications where SEO is not critical (e.g., an admin interface, a project management tool, or anything behind authentication). It's also simpler to deploy in some ways: you can host static files easily without managing a Node SSR process. The trade-off is initial load performance; however, you can mitigate this with techniques like code-splitting (both React and Vue support lazy-loading routes/components out of the box, especially with Vite) so that the user downloads only the necessary code for the first screen.

- **Combining with Rails HTML:** The question phrased *"Rails as the origin for initial HTML"* – in the SPA scenario, Rails could still send an HTML file, but it's basically just the static file that includes the React app. Rails isn't doing dynamic rendering of content; it's merely serving the file (or you might bypass Rails and let a CDN serve it). So Rails is the origin in the sense of hosting the file, but not in the sense of SSR. If SSR-like behavior is needed, the SPA approach isn't sufficient; you'd need to switch to Next/Nuxt or embed React in Rails (as in section 1).

Comparison of Separate Front-End Options:

Let's summarize and compare the separate front-end approaches in a table, including React vs Vue:

Front-End	Tech Stack	SSR (Initial HTML content?)	Data Fetching Strategy	Notes and Use Case
Next.js app	React 19 + Next.js	**Yes (SSR)** – Next.js server renders React on Node [12]	Next fetches from Rails API during SSR (e.g. via `getServerSideProps`), and client-side for subsequent requests.	Great for SEO, fast first paint. More moving parts (Node server).
React SPA	React 19 + Vite	**No (CSR)** – initial HTML is just a stub, React renders on client	React app calls Rails API from the browser (e.g., `fetch` in `useEffect` or via state libraries).	Simple deployment (static files + Rails API). Good for apps where SEO not needed.
Nuxt.js app	Vue 3 + Nuxt 3	**Yes (SSR)** – Nuxt (Node) renders Vue components to HTML	Nuxt calls Rails API in `asyncData` or server hooks, and uses browser API calls after hydration.	Vue equivalent of Next – similar trade-offs and benefits.
Vue SPA	Vue 3 + Vite (or CLI)	**No (CSR)** – same as React SPA approach	Vue app calls Rails API from browser (e.g., in Vue `created()` or Composition API effects).	Similar to React SPA. Simpler, but no instant content for crawlers.

In both Next.js and Nuxt.js cases, you have the **SSR option** which provides a server-rendered HTML for each page, improving initial load and SEO. Both also support **static generation** for pages that don't change often (e.g., Next's `getStaticProps` or Nuxt's `generate` mode), which could be used to pre-build pages at deploy time and serve them (with Rails data fetched at build time or via revalidation). However, that approach is typically used if the data is mostly static or can be cached for long periods.

2.3 Rails as a JSON API and SSR "origin" considerations

When using a separate front-end, Rails is usually running in **API mode** (possibly you even create the Rails app with `--api` flag, which skips view/template setup). But you might still have Rails deliver the base HTML in some setups. For example, some architectures use Rails to serve a base layout and perhaps handle authentication (with Rails sessions), then include a front-end bundle. A technique is to have Rails render an ERB template that outputs some initial state or CSRF tokens, and includes the front-end script. However, with frameworks like Next, this is rarely needed because Next can integrate with authentication tokens or cookies directly.

One scenario where Rails would send initial HTML in a decoupled setup is if you implement **Rails-side server rendering** of React via a gem like `react-rails` or `react_on_rails`. But those blur the line between "separate" and "integrated" – they actually allow Rails to render a React component (using ExecJS or a Node runtime) and send it in HTML, then have the front-end hydrate it. This is a hybrid approach: e.g., you could build a React app that is compiled separately but use `react_on_rails` to do SSR in the Rails view by invoking the compiled assets. This is advanced and specific, so unless you have a strong reason to keep Rails as the entry point, it's usually simpler to let Next/Nuxt handle SSR.

Nuxt vs Next vs Rails views: It's worth noting that Rails 7/8 with **Hotwire (Turbo+Stimulus)** tries to cover a lot of use-cases that traditionally pushed people to React or Vue. Turbo Streams and partial page updates can make many pages interactive without a full SPA. But for complexity beyond what Turbo can handle, that's when embedding a React component or going full SPA might be justified. We'll talk more about Stimulus/Turbo interplay with React in a later section.

In summary, a separate front-end (React or Vue) with Rails gives you a clear separation of concerns: Rails is purely a backend service. You gain flexibility in scaling front-end and back-end independently and can use the latest front-end tools. The downsides are the complexity of two systems and potential duplication (validation logic on both sides, etc.). SSR frameworks (Next/Nuxt) mitigate the user experience issues of SPAs at the cost of running Node servers.

3. Server-Side Rendering (SSR) in a Rails 8 + React 19 Stack

Server-side rendering means generating the HTML for React components on the server (either Rails or Node) rather than in the browser. We have touched on this in previous sections, but here we'll detail SSR in both contexts: **integrated (Rails rendering React)** and **separate (Next.js/Nuxt rendering React/Vue)**.

3.1 SSR in a Native Rails context (React embedded in Rails)

When using React inside Rails (esbuild or importmap approach), the default is client-side rendering – the Rails view contains a blank `<div>` and React renders into it on the client. But Rails can do SSR with React using libraries like **react-rails** or **react_on_rails**:

- `react-rails` **gem:** This gem provides a Rails view helper to render React components. For example, in a Rails view (ERB/Haml) you can do:

```
<%= react_component("HelloWorld", { greeting: "Hi" }, prerender: true) %>
```

This will, at request time, server-render the `HelloWorld` React component with the given props into an HTML string (if `prerender: true`) [14]. It uses an ExecJS runtime (like mini_racer or therubyracer) or can integrate with a Node process to run the React code. The gem will also ensure the same component is initialized on the client side (so it includes a JavaScript pack that on page load will hydrate or render the component, enabling interactivity).

With React 17 and below, hydration was via `ReactDOM.hydrate`. In React 18/19, the recommended approach is `ReactDOM.createRoot(container).hydrate()` or `hydrateRoot` from `react-dom/client`. The `react-rails` gem has been updated to support modern React; it still works by injecting the HTML into the Rails response [15]. The client JavaScript it provides (React UJS) will detect the pre-rendered components and hydrate them.

- **Custom SSR without gem:** It's possible to roll your own SSR by using a JavaScript runtime in Ruby. For example, you can use **MiniRacer** (which provides a V8 runtime in Ruby) to evaluate your React component with `ReactDOMServer.renderToString`. But doing so requires bundling your component code in a way the runtime can execute (often via a separate bundle file for server use). The react-rails gem essentially does this heavy lifting (it even creates a `server_rendering.js` pack for you to require components for server rendering [16]). Reinventing it is usually not necessary unless you have very custom needs.

- **Implications of SSR in Rails:** When Rails server-renders React, each request will run JS. This can be CPU-intensive, especially with larger components. You'll want to ensure the ExecJS runtime is optimized (using mini_racer which embeds V8 is much faster than older Ruby Racer or even Node via exec). Caching can help – e.g., caching the rendered output for identical props so you don't recompute on every request. Also, memory leaks in a long-running Node process or V8 context could be a concern (the gem usually reuses a single V8 context for performance).

- **State and hydration:** If you SSR on Rails, you often need to pass the initial state to the front-end. This can be done by embedding a JSON in the page (e.g., in a `<script type="application/json" id="props">` tag) or by ensuring the hydrated component picks up the same props. React 19's SSR is fully supported – you can even stream the HTML from Rails as it's being rendered, but that would require more low-level handling (not sure if react-rails supports streaming rendering yet; it might render to a string fully before sending). For most cases, a full string render is fine.

- **Support levels:** The good news is libraries like **react-rails** have been around for years and support modern React features (including concurrent mode). They explicitly allow using JSX/TSX (via Sprockets or Webpacker/Shakapacker) [15]. So if you want true SSR and you prefer Rails to be in charge of rendering, you can use these. Rails 8 doesn't natively introduce anything radically new for SSR of JavaScript, so it's about using these gems.

- **When to SSR in Rails:** One scenario is SEO for specific pages. For example, you have a largely server-rendered app but a few pages are built in React – you could prerender those for bots or initial load. Another is initial page speed – maybe you want to render the first screen on the server for performance, even if the rest of the app is client-driven. Keep in mind SSR can't access browser-specific APIs (no `window` or `document` during server render), so components must be written to handle that (e.g., only use those APIs in effects that don't run during SSR).

- **Example:** Using react-rails, if we generated a component `HelloWorld`, the gem would create a file (with JSX or TSX) and we include it in asset pipeline. We could then call

`react_component("HelloWorld", {name: "Foo"}, prerender: true)` in the view. On response, Rails includes something like:

```
<div data-react-class="HelloWorld" data-react-props="{\"name\":\"Foo\"}">
  <h1>Hello, Foo</h1>
</div>
```

That `div` has the pre-rendered HTML inside. Then the client script will find `data-react-class="HelloWorld"` and hydrate it (attach event listeners, etc.).

Summary: SSR in an integrated setup is doable but adds load to the Rails server. It's fully supported via gems (automatically rendering React server-side and client-side [17]). The level of effort is moderate: configure the gem, ensure your packs are set up, use the helper in views.

3.2 SSR in Separate Front-end (Next.js/Nuxt) context

In the separate front-end scenario (Section 2.1), SSR is handled by the Node.js front-end server (Next or Nuxt). The **implications and support** here are somewhat different:

- **Performance and scalability:** Next.js is built for SSR; it employs techniques like incremental static regeneration, caching, and since React 18, streaming responses. This means it can handle a large amount of SSR traffic efficiently, often more so than a Ruby ExecJS approach, because V8 in a long-lived Node process is quite fast at JS execution (and you can cluster Node for multi-core). Rails is not burdened by rendering the UI in this case, it only handles data. So you split the workload: Node does CPU-intensive rendering, Rails does database and data processing. This separation can actually improve scalability by allowing the two to scale independently (you can run more Next.js instances or more Rails instances as needed).

- **Development complexity:** You do need to run two servers in dev (Rails on e.g. port 3000, Next on 3001, or vice versa). However, frameworks like Next handle auto-refresh and such very well, and Rails can focus on API. The developer experience is usually fine, just some setup to concurrently run both (e.g., via Foreman or separate terminals).

- **SSR support in frameworks:** Next.js and Nuxt have *first-class* support for SSR – it's a core feature, not an afterthought. This means things like routing, code-splitting, and even caching strategies are built around the SSR model. For example, Next 13 introduced the app directory and React Server Components, which let you write some components that only render on the server (never sent to client), simplifying data loading and reducing bundle size. Nuxt 3 similarly has server-only components and uses Nitro (a server engine) that can even run on serverless environments.

- **State transfer:** When Next or Nuxt renders a page, they will include a hydration script with JSON of the page's props/state. So the client can pick up from exactly that state without refetching immediately. This is seamless – you usually don't have to manually do anything (the framework injects a `<script id="__NEXT_DATA__">` or similar with the JSON). In contrast, with Rails SSR you might manually embed state.

- **SEO and meta tags:** Next and Nuxt allow setting `<head>` tags for SEO on a per-page basis (e.g., via Next's Head component or Nuxt's useMeta). With an SPA, you'd need something like

React Helmet to manage meta tags and you'd still have the issue of them not being present to crawlers until JS runs. So SSR front-ends clearly win for SEO use cases.

- **Edge SSR / CDN**: An emerging trend is using edge functions (like Cloudflare Workers or Vercel Edge) to run SSR close to users. Next.js and Nuxt are adapting to this world (since they can output serverless handlers). Rails, being a heavier backend, isn't typically run on edge in the same way. If SEO and global performance are paramount, a Next.js front-end that can deploy to a global CDN (e.g., Vercel) and talk to a Rails API might be a very attractive architecture.

Support levels: React 19 is fully supported by Next.js (assuming by 2025 Next has been updated accordingly). In fact, Next often is ahead in adopting React features – e.g., React 18 streaming and Suspense were leveraged by Next 12/13 shortly after React 18's release. So any new SSR improvements in React 19 (like improved streaming, or async transitions during SSR) will likely be usable in Next. Vue/Nuxt likewise keep up with Vue core capabilities. The support and community for SSR in these frameworks is strong.

Summary: If you have a separate front-end, leveraging SSR at that layer is recommended for public-facing sites. Rails itself doesn't need to do SSR since the Node layer does it. It might feel odd that "Rails is not sending HTML at all, just JSON", but that is by design in headless architectures. Rails could still serve some HTML (like an email template or an admin not covered by the front-end), but for the main app it doesn't.

One thing to consider: **time-to-first-byte (TTFB)** might be higher when Next has to fetch from Rails on each request. This is a two-hop SSR (Node must wait on Rails). Solutions include caching API responses, or using GraphQL batch queries, or co-locating the servers (so network latency is low). In some cases, people avoid this by letting the front-end directly query the database or use a replication – but that defeats the purpose of having Rails. So a well-designed API and maybe some caching (e.g., Next request to Rails is fast because data is cached in Redis or Rails's memory) can help. Also, Next can parallelize data requests (multiple fetches in parallel in `getServerSideProps`) if the page needs data from multiple endpoints.

To sum up SSR trade-offs in Rails vs Node:

- Rails SSR (with react-rails): **Pro:** Only one server (Rails) to deploy, can use Rails view logic alongside. **Con:** Puts load on Rails, requires ExecJS, maybe not as scalable for heavy JS rendering.
- Node SSR (Next/Nuxt): **Pro:** Offloads rendering to a dedicated JS environment built for it, very high support, good scaling. **Con:** Two systems to maintain, integration via API calls.

Many choose Node SSR for large apps, and Rails SSR for smaller interactive pieces in an otherwise standard Rails app.

SSR Recap – Why bother?

Server-side rendering in any stack provides several benefits: **faster initial load, SEO friendliness, and better perceived performance** by delivering HTML that browsers/users can see immediately [12]. React 19's improvements (streaming, server components) further blur the line between what runs on server vs client, enabling a sort of hybrid rendering paradigm. If SEO isn't a concern (e.g., internal apps), SSR can be optional and you might skip it to simplify your stack.

4. Deployment with Docker and Kubernetes (for Both Architectures)

Whether you choose to integrate React in Rails or build a separate front-end, Docker and Kubernetes can help standardize deployment. This section outlines best practices for containerizing the application(s) and setting up a build pipeline, highlighting differences between the **monolithic deployment** (Rails + React together) and a **two-service deployment** (Rails API + separate front-end).

4.1 Containerizing a Rails 8 app with integrated React (esbuild/importmap)

For a Rails app that includes React (especially via esbuild or Webpacker-like pipelines), the deployment artifact is a single container image containing the Rails app, which also needs to include the compiled JavaScript assets.

Best practices:

- **Use multi-stage builds:** This produces a lean final image. For example, use a builder stage with Node and a runtime stage with just Ruby. Rails 7.1+ even provides a default multi-stage Dockerfile example [18] [19]. In our case, we need Node in the build stage to compile assets:

```
# Stage 1: Build assets and install gems
FROM ruby:3.2-slim AS build
# Install Node (for JS build) and yarn if not already present
RUN apt-get update -qq && apt-get install -y curl && \
    curl -sL https://deb.nodesource.com/setup_18.x | bash - && \
    apt-get install -y nodejs && \
    npm install -g yarn

WORKDIR /app
# Install gems (use bundler cache)
COPY Gemfile Gemfile.lock ./
RUN bundle install --without development test

# Install JS deps and build assets
COPY package.json yarn.lock ./
RUN yarn install --frozen-lockfile
# Copy the rest of the code
COPY . .
# Precompile assets (which runs yarn build via jsbundling, and compiles CSS if any)
RUN bundle exec rails assets:precompile

# Stage 2: Final image
FROM ruby:3.2-slim AS app
WORKDIR /app
# Copy compiled gems and assets from build stage
COPY --from=build /app /app

# Expose port and define startup (assuming using Puma or similar)
```

```
EXPOSE 3000
CMD ["bundle", "exec", "puma", "-C", "config/puma.rb"]
```

In this example, the build stage installs both Ruby gems and Node packages, then runs `rails assets:precompile` (which will invoke the esbuild build or importmap asset compilation). The result is that `app/assets/builds` (or `public/assets` for precompiled Sprockets assets) now contain the JS/CSS. The final stage copies everything from the build stage – including the `vendor/bundle` (gems) and the compiled assets – but we could slim it down further by only copying specific directories (to avoid dev/test stuff).

This approach ensures the final image doesn't need Node or even yarn installed – it has just the static assets. **Rails 8 note:** If using importmap and no asset precompile, you might skip the Node steps; but since we have TS, we likely do need to compile via Babel or tsc, so Node is involved anyway.

- **Environment configuration:** In Kubernetes, you'd supply environment variables for things like `RAILS_ENV`, database URL, Redis URL, etc. Do not bake secrets into the image; use Kubernetes Secrets for things like `RAILS_MASTER_KEY` (Rails 7+ reads `RAILS_MASTER_KEY` env to decrypt credentials). In the Dockerfile above, we didn't copy `config/master.key` – in production you'd provide the key via secret.

- **Continuous Integration (CI):** Your CI pipeline should build the image (running tests, etc., before possibly). Given the multi-stage file, building the image will also build the JS assets. Ensure that any test pipeline also runs `yarn build` or `rails assets:precompile` so tests run against compiled packs if needed (for system tests).

- **Image size:** The slim Ruby image plus node adds some weight, but since Node is only in build stage, final image might be on the order of a few hundred MBs (mostly the Ruby runtime and gems). You can further slim by using `bundle install --deployment --without development test` and cleaning caches.

- **Kubernetes deployment:** Deploy this single container as a Deployment (or many replicas behind a Service). Standard best practices:

- Use a readiness probe that checks an endpoint (like `/health` or `/up`) to ensure Rails is responsive before receiving traffic.
- Use a liveness probe to restart if the app hangs.
- Mount volume for any persistent data (usually not needed if stateless).
- Attach a PersistentVolume for logs if needed, or use a sidecar for log shipping. But typically logs can go to STDOUT/STDERR and use cluster logging.

- Scale replicas based on resource usage or traffic. If SSR via Rails (React components), watch CPU closely – SSR can spike CPU.

- **Asset serving in production:** With the compiled assets in `app/assets/builds` and the Rails asset pipeline configured, Rails will serve these static files. In high-traffic scenarios, it's common to offload static asset serving to a CDN or at least the ingress/Nginx (serving directly from a volume or an object store). If that's desired, you could copy the `app/assets/builds` contents out to a storage (like S3) as part of CI/CD. But many deployments just let Rails serve them (especially if they're fingerprinted and cached).

- **Auto scaling & load balancing:** The container can be duplicated behind a load balancer. If using SSR (prerender) inside Rails, scaling horizontally is important to handle load (each request does more work). Kubernetes HPA (Horizontal Pod Autoscaler) can scale based on CPU if you find SSR causes high CPU usage.

4.2 Deployment for Rails + Separate Front-End (Next.js or static SPA)

When front-end and back-end are separate, you will be dealing with **two images and two deployments** (most likely). One for Rails API, one for the front-end.

Rails API deployment: This is similar to above minus the asset compilation: - If Rails is truly API-only (no compiled JS), you don't need Node at all in the build. The Dockerfile simplifies – just install gems and run. Still use multi-stage to keep image slim (for example, use an Alpine or Slim base and only include necessary libs). - You'll still have `RAILS_MASTER_KEY`, database configs, etc. to manage via env/Secrets. - All the standard Rails in Docker considerations apply (precompile assets step can be skipped if there are none; or if you still use some CSS or importmap Stimulus, you might still run `assets:precompile` for CSS or importmap manifest). - Migrate database in an init container or as part of CI deployment step (`rails db:migrate`).

Next.js front-end deployment: There are a couple of modes: - **Next.js server mode:** Run the Next app as a Node server (to do SSR for each request). The Dockerfile would: - Build the Next app (`next build`). - Then start it with `next start` (which starts an Express-like server on a port, default 3000). - Multi-stage: use a Node builder image to install deps and build, then use a lighter Node image for running. - Ensure to set `NODE_ENV=production` for the build and runtime. - Example snippet:

```
FROM node:18 AS build
WORKDIR /app
COPY package.json package-lock.json ./
RUN npm install
COPY . .
RUN npm run build    # runs next build

FROM node:18-slim AS serve
WORKDIR /app
COPY --from=build /app . .
EXPOSE 3000
CMD ["npm", "run", "start"]   # or "next", "start"
```

This image will listen on port 3000 serving the Next SSR app. In Kubernetes, you'd have a Service for it, etc.

- **Next.js static export (optional):** In some cases, if you used `next export` (no SSR, purely static), then you can serve the files via CDN or a simple nginx container. But that forfeits SSR benefits. That scenario is more like a CRA or Vite app; see SPA below.

- **Vite/React or Vue SPA deployment:** Here you don't need a Node server at runtime, just a static file host. Options:

- Use Nginx: Build the static files (`vite build` outputs to `dist/` directory). Use an Nginx image that copies those files to `/usr/share/nginx/html`. Configure Nginx to serve them and perhaps route API calls to the Rails service (or simply let the SPA call the Rails service via its URL).
- Use a CDN or object storage: e.g., deploy the static assets to S3 and serve via CloudFront, which is a common pattern. In that case you might not even need a container for the front-end – it becomes part of the CI/CD to push to CDN.
- Or use a minimal Node/serve: Node has `serve` package or you could use a simple Express to serve static. But usually Nginx or CDN is simpler.

If using Kubernetes and want to keep it self-contained, an Nginx Deployment or an Nginx sidecar can serve the SPA. One could even serve the static files from the Rails container's `/public` directory (if you copy them there), but separating concerns is cleaner.

Networking and Coordination:

- **Ingress and service routing:** In Kubernetes, you likely want the front-end and back-end to appear under one domain. For instance, `myapp.com` should serve the Next.js (or SPA) app, and API calls to `myapp.com/api/...` should route to Rails. This can be achieved with an Ingress Controller (like NGINX or Traefik ingress). You'd configure path-based routing: e.g., any request that starts with `/api` goes to the Rails service, and everything else goes to the front-end service. If using separate subdomains (e.g., `api.myapp.com` and `app.myapp.com`), then you'd use host-based routing or just set up two Ingresses with different host rules.

- **CORS and CSRF:** If using the same top-level domain and just different paths, you can avoid CORS issues by having the browser always talk to `myapp.com/api` (same domain). If using different subdomains or domains, you need to enable CORS on Rails (allow the front-end origin). Also, authentication cookies need special care in cross-site scenarios (you'd mark them with the proper SameSite attributes or use token auth). A common setup is to use JWTs or OAuth for the API auth if completely separate. If you prefer Rails session cookies, then having the front-end served on the same parent domain (so cookies are shared) is one way (e.g., serve front-end on `myapp.com` and Rails API on `myapp.com` as well via the path routing).

- **Scaling:** In Kubernetes you'd scale the Rails API deployment based on typical metrics (request latency, CPU if it's CPU-bound). For the Next.js deployment, scale based on Node CPU or response time. Next.js can also cache rendered pages (ISR – Incremental Static Regeneration) which means not every request hits Rails if data hasn't changed, etc. These are app-level caching strategies you can leverage.

- **Monitoring:** Monitor both services. For example, gather metrics: Rails (through Puma stats or Skylight etc.), Next (through V8 metrics or just general Node process metrics). Logging: ensure both log to stdout so Kubernetes can aggregate logs.

- **Pipeline:** You will build and publish two images. They can be versioned together (if in a monorepo, you might tag them with the same version or commit SHA). Or you might even use a single repository for both with a docker-compose to run them together in dev. Regardless, in CI/CD ensure to test integration (e.g., end-to-end tests hitting the deployed dev environment to catch any contract mismatch between front-end and API).

Kubernetes example (conceptual):

Imagine we have a cluster and we want `https://myapp.com` to serve our Next.js front-end (React 19 SSR) and `https://myapp.com/api/...` to reach Rails. We might configure an ingress like:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: myapp-ingress
spec:
  rules:
  - host: myapp.com
    http:
      paths:
      - path: /api(/|$)(.*)
        pathType: Prefix
        backend:
          service:
            name: rails-api-service
            port:
              number: 3000   # assuming Rails listens on 3000
      - path: /(.*)
        pathType: Prefix
        backend:
          service:
            name: frontend-next-service
            port:
              number: 3000   # Next listening on 3000
```

This way, a request to `/api/users` goes to Rails, and `/home` or `/about` goes to Next. The Next app in turn might call `/api/users` internally to get data. You could also have Next call the Rails service via the internal cluster network (service DNS), which avoids going out through ingress again – i.e., `fetch('http://rails-api-service:3000/users')` from the Next server. This is efficient, but you must ensure to allow that or use the service DNS name and perhaps an internal API auth.

For a static SPA, the ingress is simpler: everything not `/api` just serves static files (which could be an Nginx serving from a ConfigMap or something). Or you could even configure the ingress to serve the static index.html as a default backend.

Docker and K8s Recap: Use multi-stage builds to optimize image size, separate config from code, and leverage Kubernetes networking to integrate the services. Keep environment-specific settings out of the image (use env vars). For the monolith approach, the process is simpler (one service to deploy, but potentially larger single point of failure). For the two-service approach, you gain modularity (can deploy front-end and back-end independently, e.g., update front-end without touching back-end) at the cost of more complex orchestration.

4.3 Other Considerations for Deployment

- **Caching and CDNs:** Regardless of approach, consider using a CDN in front of your app for static assets. For a separate front-end, the entire front-end might be static and served via CDN. For a Rails integrated app, you can configure Rails asset host to a CDN for the compiled JS/CSS. CDNs will reduce load on your pods for static content.

- **Docker base images:** Ensure you keep them updated for security. Ruby slim images and Node LTS images are good bases. Some people use Alpine for smaller size, but it can complicate building native extensions. Slim (Debian-based) is a good compromise.

- **Kubernetes resources:** Set requests/limits appropriately. For example, a Next.js container might need a few hundred MB of RAM and moderate CPU. A Rails container may need more memory (depending on garbage usage, etc.). Monitor and tune as needed.

- **Helm or YAML:** Use a consistent deployment strategy (like Helm charts or Kustomize) to manage the two deployments. This makes it easier to spin up entire environments (dev, staging, prod) with the front-end and back-end together.

By following these practices, you can achieve smooth CI/CD for both architectures. For instance, a CI pipeline can run tests, build images for rails-backend and frontend, push them, and then a deployment step (via kubectl or ArgoCD) updates the Kubernetes cluster with the new images. Kubernetes ensures zero-downtime (if configured properly with readiness probes) by rolling updates.

5. StimulusJS and React: Integration and When to Use Each

StimulusJS (part of Hotwire) is the default modest JavaScript framework in modern Rails. It's important to understand how Stimulus can coexist with React, and in what scenarios you might use one or the other (or both together).

What is Stimulus? Stimulus is a framework for enhancing static HTML with minimal JavaScript, by attaching controllers to HTML via `data-*` attributes. It doesn't render or manage a virtual DOM like React; instead, it augments server-rendered HTML with behaviors [20]. It's excellent for small interactive elements (show/hide sections, form validation hints, etc.) that can be rendered on the server and manipulated on the client. Stimulus is **sufficient and recommended** for many cases of moderate interactivity in Rails apps [21].

React vs Stimulus responsibilities:

- **When Stimulus is useful:** If your application largely renders HTML on the server (ERB/Haml templates) and you just need small "sprinkles" of interactivity, Stimulus is lightweight and easy. For example, toggling a dropdown, adding/removing a CSS class on click, or handling a form submission with AJAX can all be done with a few dozen lines of Stimulus (often without needing to manage heavy state). Stimulus shines in scenarios where using React would be overkill – it attaches to the "HTML you own" and enhances it [20].

- **When React is useful:** When your UI needs complex state management, dynamic updates without full page reloads, or rich components (grids, charts, etc.), React (or another SPA framework) is more suitable. React takes over rendering entirely for a portion of the page or the whole page. Many developers familiar with React find it challenging to implement very dynamic UIs with only Stimulus/Turbo [21] – for example, a live preview with lots of client-side logic, or an interactive data dashboard.

- **Using Stimulus and React together:** They can **coexist** peacefully if you partition the DOM they control. A common pattern is to use Stimulus as the orchestrator to **mount React components** when needed. For instance, your Rails view is mostly static HTML, but you have a `<div data-controller="react-loader" data-react-component="Chart" data-`

`props="{...}"></div>`. A Stimulus controller (`react-loader_controller.js`) can read the `data-react-component` value and dynamically import and mount that React component into the div. This way, Stimulus drives the high-level behavior (maybe it triggers the React component when it becomes visible or when a user clicks "Edit"), and React handles the complex UI within that div. An example integration snippet:

```
// controllers/react_loader_controller.js (Stimulus)
import { Controller } from "@hotwired/stimulus"
export default class extends Controller {
  static values = { component: String, props: Object }
  async connect() {
    // Dynamically import the React module (assuming it's exported from a known path)
    const componentName = this.componentValue;  // e.g. "Chart"
    const module = await import(`../components/${componentName}.tsx`);
    const Component = module.default;
    // Mount the React component into this element
    const root = ReactDOM.createRoot(this.element);
    root.render(React.createElement(Component, this.propsValue));
    this.root = root;
  }
  disconnect() {
    // Unmount on controller disconnect to clean up
    if(this.root) this.root.unmount();
  }
}
```

In the HTML:

```
<div data-controller="react-loader"
     data-react-loader-component-value="Chart"
     data-react-loader-props-value='<%= { data: @chart_data }.to_json %>'>
</div>
```

When the Stimulus controller connects (on page load), it loads the `Chart` React component and renders it into the div. On disconnect (navigating away if using Turbo Drive, etc.), it unmounts the React tree to avoid memory leaks. This pattern is demonstrated in various blogs [22] [23] and allows **mixing technologies**: use React where you need heavy lifting, keep using Rails+Stimulus for the rest [24].

- **Redundancy and conflicts:** If an entire page or feature is built in React, Stimulus may become redundant for that part. In fact, having Stimulus and React try to manipulate the same DOM can cause conflicts. **Avoid overlapping control** – for example, don't have Stimulus targeting elements inside a React-rendered component, because React's virtual DOM might re-render and remove those elements or overwrite attributes, breaking Stimulus. Conversely, if Stimulus changes the structure that React expects to manage, React might not see those changes or might override them on next render. Essentially, React expects to own the DOM subtree it renders; Stimulus should either own separate elements or be used at the boundary to trigger React.

- **Stimulus with separate front-end:** If you go for a fully separate React front-end (Next or SPA), **Stimulus isn't used in the front-end at all** – it's a Rails-specific tool. In that architecture, any small behaviors would likely be implemented with React itself or perhaps other lightweight libraries, since the Rails views are not serving UI (except maybe some minimal layout). Stimulus could still be used in Rails-rendered emails or an admin interface that isn't part of the React app, but it wouldn't interact with the React app. Essentially, in a headless Rails API, Stimulus/Turbo are usually not present (unless you keep some hybrid pages).

- **Stimulus with integrated front-end:** In a Rails app that has some React components (via importmap or esbuild as earlier), Stimulus can handle things like Turbo Drive events, or controlling modals outside React components. You might have both Stimulus controllers and React components on the same page, as long as they deal with different parts of the DOM or one is nested in the other in a controlled way. For example, you could have a Stimulus controller that handles a file upload via `<input>` and then passes the file data to a React component for preview – though in many cases you'd just do the whole widget in React or Stimulus solely.

- **When Stimulus becomes redundant:** If over time your app becomes a rich JavaScript app, you might find that almost all interactive parts are handled by React. At that point, Stimulus (and possibly even Turbo for navigation) might not provide much value. For instance, if clicking links triggers React route changes rather than full page loads, Turbo Drive is not in use. In such cases, some teams decide to remove Stimulus/Turbo entirely and treat the Rails app as an API only, because maintaining both can be unnecessary. On the other hand, you might still keep Turbo for forms or non-SPA pages (there's nuance, but generally a fully SPA approach doesn't need Hotwire).

Example scenario: Imagine a Rails 8 app where most pages are traditional (server-rendered) using Turbo and Stimulus, but one page requires a **complex interactive map** component built with React. You can use Stimulus to render that map: the Rails view outputs the HTML container and perhaps some data in JSON, Stimulus controller sees it and mounts the React Map component. The map itself is a self-contained React world (with maybe Redux or context, etc.). When the user navigates away, Turbo might unload that frame, triggering Stimulus disconnect which unmounts React. This way, the two technologies are separated but cooperative. The developer uses the right tool for each job: Stimulus for small stuff, React for the map.

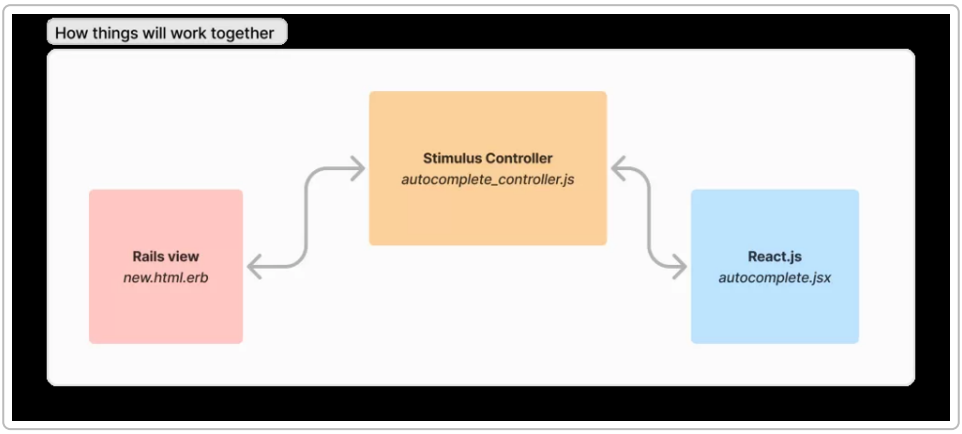
As Gabriel Quaresma notes, Stimulus is not a "React killer" but a different paradigm [21]. For developers already comfortable with React, integrating it into a Rails+Stimulus setup can yield the best of both: you enhance Rails views incrementally with React as needed [25].

Potential pitfalls: one must ensure that global state or events are managed. For example, Stimulus controllers can communicate via dispatching events. React components have their own event system (or use context for global state). If a Stimulus controller needs to tell a React component something, a pattern is to dispatch a DOM event which the React component (if rendered in the DOM) could listen to, or vice versa [26]. This is an edge case but solvable (AppSignal's blog shows using custom events to keep React and Stimulus in sync [26]).

Summary: Use Stimulus for simplicity when you can, and bring in React for the heavy lifting when needed [24]. In a native Rails integration, Stimulus and React can complement each other – Stimulus controlling page lifecycle and minor tweaks, React powering complex widgets. In a separate front-end, Stimulus is mostly out of the picture, since the front-end is not Rails-rendered. Knowing when Stimulus becomes redundant is key: if you find yourself writing a Stimulus controller that is basically reinventing

a dynamic UI that a React component could handle more cleanly (especially with lots of state), it might be time to switch that part to React.

Diagram – Stimulus + React integration:



Stimulus controller mediates between a Rails view and a React component. In this architecture, the Rails view (server-rendered HTML) is enhanced by a Stimulus controller (`autocomplete_controller.js` in this example), which in turn initializes a React component (`autocomplete.jsx`). Stimulus handles the connection between server HTML and the React UI [20] [21] *.*

In the diagram above, the Rails view might include an input field and results container. Stimulus listens for input events and passes them to the React component (which could manage a list of suggestions). The React component renders suggestions dynamically, and could even communicate back to Stimulus or Rails via events or direct calls. This separation of concerns keeps Rails in control of structure and initial data, Stimulus in control of wiring, and React in control of complex rendering.

Conclusion: Integrating React 19 with Rails 8 can be achieved through multiple paths. The **native approach** (esbuild or importmap) keeps everything within Rails, which is convenient but may require some configuration for TSX. The **separate front-end approach** (Next.js, Vite SPA, or analogous Vue setups) provides a clear split and can leverage full SSR capabilities of modern front-end frameworks, at the cost of a more complex deployment. SSR can greatly enhance performance and SEO, whether done via Rails or Node, but it should be applied where needed given its complexity. Finally, **StimulusJS can coexist with React** in a Rails app, allowing a pragmatic mix-and-match – using the right tool for each part of the user experience. By following the practices outlined and understanding the trade-offs, you can build a Rails 8 + React 19 stack that is both **highly interactive** and **maintainable**, deploying it confidently on modern infrastructure.

---

[1] [2] [3] [4] [5] **Rails 7, React, TypeScript, ESBuild and View Components - Ryan Bigg**
https://ryanbigg.com/2023/06/rails-7-react-typescript-setup

[6] **Does importmap-rails not work with a Rails app that uses TypeScript? · Issue #124 · rails/importmap-rails · GitHub**
https://github.com/rails/importmap-rails/issues/124

[7] [8] [9] [10] **Ruby on Rails with React on Typescript using importmaps - DEV Community**
https://dev.to/gavrilarails/ruby-on-rails-with-react-on-typescript-using-importmaps-5082

11 **In Rails 7 can one use import maps along with a js bundling solution ...**
https://stackoverflow.com/questions/73720174/in-rails-7-can-one-use-import-maps-along-with-a-js-bundling-solution-or-are-they

12 13 **Mastering Server-Side Rendering (SSR) in React 19 with Vite: The Ultimate Guide for Developers - DEV Community**
https://dev.to/yugjadvani/mastering-server-side-rendering-ssr-in-react-19-with-vite-the-ultimate-guide-for-developers-4mgm

14 15 16 17 **RubyDoc.info: File: README – Documentation for react-rails (3.0.0) – RubyDoc.info**
https://rubydoc.info/gems/react-rails/3.0.0

18 19 **Containerize your app | Docker Docs**
https://docs.docker.com/guides/ruby/containerize/

20 21 25 **Rails + Stimulus + React The definitive (and easy) way to integrate - The Miners**
https://blog.codeminer42.com/rails-stimulus-react/

22 23 24 **Using React with Stimulus JS - DEV Community**
https://dev.to/dropconfig/using-react-with-stimulus-js-di

26 **Connecting React.js and StimulusJS with JavaScript Events**
https://blog.appsignal.com/2023/12/18/connecting-react-and-stimulusjs-with-javascript-events.html