ChatGPT

Nuxt 3 + Rails 8 Integration Guide (Docker & Kubernetes)

Overview: This guide explains how to integrate a Nuxt 3 frontend with a Ruby on Rails 8 backend in a containerized (Docker + Kubernetes) environment. We cover server-side rendering (SSR) strategies, project structure and routing, authentication (Devise) setups, using Hotwire/Turbo alongside Nuxt, database integration (PostgreSQL + optional MongoDB), useful libraries, and deployment considerations. Code examples and configuration snippets are provided for clarity.

SSR Strategies: Rails vs Nuxt

Integrating Nuxt with Rails can be done in two primary ways for SSR:

- **Rails-Driven SSR (within the Rails app)** – Rails itself renders the Vue/Nuxt app's HTML (via Ruby-side JS execution or bridges).
- **Nuxt-Driven SSR (separate Node server)** – A Nuxt Node.js server handles SSR, with Rails acting as a JSON API.

We'll examine each approach:

SSR within Rails (Ruby-Driven SSR)

Rails 8 can perform SSR of the Vue/Nuxt app *without an external Node server*, preserving a more traditional "all-in-one" Rails deployment. This is achieved by executing the Nuxt (Vue) application's bundle on the Ruby side:

- **Using V8 (mini_racer) Execution:** Rails can embed a JavaScript engine (like V8 via the `mini_racer` gem or ExecJS) to run the Nuxt app's server-side bundle and produce HTML. For example, the **Humid** gem by thoughtbot provides a lightweight wrapper around mini_racer to evaluate a JS render function and return HTML [1]. In this setup, you bundle your Vue/Nuxt app for SSR (output a server-rendering JS file) and configure Rails to load that bundle. On each request, Rails calls a render function (passing any needed props/state) to get an HTML string, which it then serves. This technique allows SSR in Rails **without a persistent Node process**, keeping the app self-contained in Rails. However, note that the embedded JS environment lacks some Node APIs (timers, etc.) and libraries must be compatible with a non-DOM JS runtime [2] [3]. Performance is also a consideration: executing JS in-process can be slower than a dedicated Node server for large apps, and memory usage should be monitored.

- **Using a "Bridge" Node Service (e.g. Hypernova):** Another option is to run a minimal Node SSR service *alongside* Rails and have Rails call out to it for rendering. **Hypernova** (by Airbnb) is a tool that allows Rails to send component or page render requests to a Node server and get back HTML [4]. The Rails app uses a Hypernova client (Ruby gem) to render Vue components; if the Node service is unavailable or slow, it can fall back to client-side rendering [5]. This approach still keeps Rails in control of routing and responses, but offloads the heavy lifting of SSR to Node (often improving speed for complex UIs). The Hypernova gem can batch multiple component renders and handle errors gracefully. The downside is you now have a Node process to manage

(though not user-facing, it's an internal SSR service) and you must ensure the Node SSR stays in sync with your front-end code. It's a compromise between fully in-Rails SSR and a full Nuxt server: Rails remains the primary web server, but Node is used on-demand for rendering.

- **Using `render_vue_component` (Hydration-Only):** There are also Rails gems (e.g. **render_vue_component** [6] [7]) that allow you to embed Vue components in Rails views. These typically *do not SSR the HTML* on the server – instead, they inject a placeholder DIV and some JSON props, and then load Vue on the client to hydrate it. This is similar to Rails' classic approach with React (via `react-rails` or `react_on_rails`). While not true SSR (no initial HTML content for SEO), this approach keeps Rails page loads and just mounts Vue components for interactivity. It might be sufficient if you only need Vue for certain dynamic parts. The trade-off is poorer SEO and initial load performance compared to SSR, since the content isn't rendered until the client JS runs.

Summary – Rails-Driven SSR Pros/Cons: This approach means one unified app (no separate Node server for users to hit), and can feel more "Rails-like." Rails controllers can fetch data and render a view in one go. It simplifies deployment (just a Rails container) and avoids duplicating routing logic. However, implementing SSR in Rails has complexity: you must build your front-end bundle for SSR and manage the interface with Ruby. There are limitations in the JS execution environment and potential performance hits if misconfigured. In practice, Rails-driven SSR is less common than Node-driven for full Nuxt apps, but tools like Humid or Hypernova make it viable for those who want **Rails to serve the HTML directly without a separate Node service**.

SSR with a Nuxt 3 Server (Node-Driven SSR)

The more common pattern is to let Nuxt handle its own SSR using Node, treating Rails as a headless API. In this architecture, the Nuxt 3 app runs as a Node server (often on its own port or container). It receives HTTP requests from the user, renders the appropriate page (server-side) using Vue 3, and returns the HTML. During this render, it will fetch any needed data from the Rails backend via API calls.

How it works: Nuxt 3 supports "universal" mode (SSR) out of the box. When running `nuxt build` and `nuxt start`, Nuxt's server (built on Nitro) will generate HTML for each route on the fly [8]. The Rails 8 app, in this scenario, is an API-only service (e.g., generated with `rails new --api`) that exposes JSON endpoints (e.g. REST or GraphQL). The Nuxt app calls these endpoints either during SSR (for initial page load data) or on the client side after hydration.

There are two deployment variants here:

- **Separate Containers/Services:** The Rails API and the Nuxt SSR server run in different containers (or even different pods in Kubernetes). For example, you might have `rails-api` (Ruby Puma server on port 3000) and `nuxt-ssr` (Node server on port 3000 or 8080). A Kubernetes Ingress (or other proxy) routes user requests: e.g., requests to `/api/**` go to Rails, and all other requests go to the Nuxt server. This cleanly separates responsibilities. The Nuxt server can also serve the Nuxt static assets (`_nuxt/*` files) or those can be offloaded to a CDN. The main challenge is **routing and domain management** – ideally, both services share the same domain so that cookies and links work seamlessly. We discuss this in Deployment below. In development, you'd run `rails server` and `npm run dev` (Nuxt) separately as well, likely on different ports.

- **Shared Container:** Less commonly, you can run Rails and Nuxt SSR in one container (for instance, using a Procfile or a supervisor to start both Puma and the Nuxt Node process). This way, you have a single "application" container that listens on one interface and delegates internally. For example, Rails could be configured to proxy non-API requests to the local Nuxt server. This is somewhat analogous to running Nginx with a Node app – except here Rails might act as the proxy. While this ensures one container/pod encapsulates the whole app, it breaks the one-process-per-container principle and can complicate scaling (you must scale Rails and Nuxt together). We generally recommend separate containers unless your infrastructure demands a single unit.

Pros/Cons: Nuxt-driven SSR is the most straightforward way to leverage Nuxt 3's capabilities. Nuxt's SSR engine is optimized for Vue, supports Nitro (which can run as Node or serverless), and will output the correct hydration code (like the `window.__NUXT__` state). You get full SSR benefits (SEO-friendly, fast first paint) and can use Nuxt's ecosystem (middleware, modules) without workarounds. The decoupling also means front-end and back-end concerns are neatly separated – each can be scaled or deployed independently. The primary downside is **increased complexity in deployment and integration**: you are effectively managing two applications (Rails and Nuxt) [9]. This means two processes, potentially two codebases (though you can keep them in one repo), and coordinating things like authentication, routing, and data fetching across a network boundary. Additionally, SSR means the Nuxt server must handle potentially many requests (including spikes), so you'll want to ensure it's properly scaled and cached as needed.

Comparing SSR Approaches

For clarity, here's a comparison of the approaches:

SSR Approach	Description	Pros	Cons
Rails In-Process (mini_racer)	Use Ruby's embedded JS (V8 via mini_racer) to evaluate the Vue app's bundle and render HTML. Example: Humid gem.	- Single app/process (no external Node server). - Rails serves pages directly (familiar deployment).	- Setup is complex (must build SSR bundle and interface it to Rails). - JS runtime limitations (no Node APIs, polyfills needed [2]). - Performance may degrade for very large UIs or high throughput.
Rails with Bridge (Hypernova)	Run a Node SSR service (Hypernova) that Rails contacts for HTML. Rails sends data, Node returns rendered HTML for insertion.	- Still mostly Rails-controlled (fallback to client render if Node fails [5]). - Node SSR can be optimized separately (can be reused for multiple apps).	- Two processes (Rails + Node) in deployment, though Node not directly user-facing. - Must ensure Node service is running and kept in sync. - Added moving parts (network calls between Rails and Node SSR).

SSR Approach	Description	Pros	Cons
Nuxt SSR Server	Nuxt 3 running as a Node server, handling routing and SSR. Rails acts only as JSON API.	- Leverages Nuxt's full power (Nitro, etc.) out-of-box [8]. - Clear separation of front-end and back-end concerns (each can evolve independently). - Nuxt dev experience (HMR, etc.) remains available.	- Essentially two apps to manage (frontend & backend) [9]. - Needs integration for auth/ cookies and routing (see below). - More complex deployment (multiple services, ingress routing).
No SSR (SPA/ Static Nuxt)	(Not SSR) Nuxt generates static files or runs in client-side only mode. Rails serves a stub page (or the static built page) and then Vue takes over.	- Simplest integration: Rails just serves an `index.html` (or proxies to Nuxt dev in dev mode). - No Node SSR server needed in production if using static generation.	- Loses SSR benefits: initial page is just a placeholder [10], data is fetched client-side (slower first paint, SEO challenges). - Essentially becomes a pure SPA architecture.

In many cases, teams start with the **Nuxt SSR server** approach for robust SSR, or even the **No SSR (SPA)** approach for simplicity (accepting the trade-offs), since those align with a clear separation of front-end and API. The **Rails SSR** approaches are chosen when you explicitly want to minimize deviation from a typical Rails monolith deployment or have constraints on adding Node services.

Next, we'll assume you are interested in the **full integration** – i.e. using Nuxt as the frontend framework (with SSR in one of the above ways) and Rails 8 as the backend API (with possible SSR support). We'll cover how to structure the project, handle routing, and share assets and state.

Integrating Nuxt 3 into a Rails 8 Project

Even though Rails and Nuxt can be separate, you can organize them within a single project/repository for convenience. There are a few ways to set this up:

- **Single Repository (Monorepo):** Keep the Rails app and the Nuxt app in one repo (e.g., Rails in the root and Nuxt in a `frontend/` subdirectory). For example, one approach is to create a Rails API app, then create a Nuxt app inside a subfolder, and adjust file locations so they can be managed together [11]. You might move Nuxt's `package.json` to the repository root and specify `"srcDir": "frontend/"` in `nuxt.config.js` [11]. This way, you can version and deploy them together. The two apps can still be deployed as separate containers, but you can run them in one codebase (helpful for coordinated changes). The GitHub project *nuxt-rails-a-la-webpacker* demonstrates this: a Rails API and a Nuxt app in one repo, deployed to two Heroku apps [12] [13].

- **Separate Repositories:** Treat the front-end and back-end as entirely separate projects. This is cleaner in some respects (each can have its own lifecycle), but you'll need to coordinate API endpoint changes manually and perhaps deal with CORS in development (if using different origins).

For many, the monorepo approach is convenient: it feels like a single project. We will assume a monorepo in the following examples (though you can adapt if they are separate).

Routing and Project Structure

Rails Routes vs Nuxt Routes: In a Nuxt + Rails setup, you typically let Nuxt handle all **frontend routes** and Rails handle **API routes**. That means Rails routes file (`config/routes.rb`) would contain endpoints for your JSON API (e.g., `/api/...` paths or a namespaced route), and possibly a catch-all route to serve the Nuxt app for any non-API path. For example, if using the **SPA/static approach**, you might do something like:

```
# config/routes.rb (Rails)
namespace :api do
  # ... API routes here ...
end

# Catch-all for frontend routes:
# This sends any request that isn't caught by the above routes to the Rails controller that se
get '*unmatched_route', to: 'frontend#index', as: :frontend_app
```

And in `FrontendController#index` (or `ApplicationController#index`), render the Nuxt app's entry HTML:

```
# app/controllers/frontend_controller.rb
class FrontendController < ApplicationController
  def index
    # Serve the pre-built Nuxt app entry point (e.g., public/index.html if Nuxt is generated s
    render file: Rails.root.join('public/index.html'), layout: false
  end
end
```

This pattern is used when Nuxt is built as a static bundle – Rails simply serves `index.html`, and Nuxt's client-side code will take over routing from there. In deployment, you'd copy Nuxt's built files into `public/` (or serve them via Nginx). An example of this approach was given on StackOverflow: adding a wildcard route to Rails and serving the Nuxt static `index.html` so that "both the Nuxt app (static HTML) and the Rails API are served by the same Rails server" [14] [10]. The downside, as noted, is you can't server-render dynamic data – Nuxt will fetch via XHR after load [10].

If you are using **Nuxt SSR server** (Node), your routing is usually handled at a higher level (e.g., Kubernetes Ingress or Nginx): requests for `/api/**` go to Rails, and all others go to the Nuxt server. In that case, Rails might not need a catch-all; it only exposes API endpoints. Conversely, Nuxt's `pages/` directory defines the front-end routes (which become SSR routes on the Node server). You might still include a catch-all in Rails as a fallback (for example, to display a maintenance page or message if Nuxt is down), but typically it won't be used if the ingress is configured correctly.

For **Rails SSR (Humid/Hypernova)**, your Rails controllers will map to pages. For example, hitting `/dashboard` in the browser could trigger `DashboardController#index` in Rails, which calls `Humid.render` (or Hypernova render) with the appropriate Vue component/page and initial data. In

this case, Rails routing is in charge of all pages (just like a normal Rails app), and the Nuxt app might not even use its own router (you may instead structure your Vue app as components that Rails knows how to call). This is a more advanced integration – essentially treating Vue as a view layer for Rails. If pursuing this, consider organizing your Vue app components in a way that mirrors Rails routes, and use something like `Humid.setHumidRenderer` to register a render function that can output the app for a given route [15] [16]. Rails will also need to serve the JS/CSS assets for the Vue app (likely via the asset pipeline or webpack build outputs). This approach requires careful coordination: you might need to *pre-fetch data in Rails* and pass it into the Vue render function (since Rails can fetch from the DB easily). As one SSR guide notes, for Rails+SSR you typically load the data in Ruby first (to avoid rendering a loading state) [17].

Development Mode: During development, it's common to run Rails and Nuxt dev servers separately. You can start Rails on `localhost:3000` and Nuxt on `localhost:3001` (for example). To avoid CORS issues and to simulate the production routing, you can proxy API calls. Nuxt 3 (with Nitro) can use a runtime config or proxy to forward calls to Rails. For instance, you might set `NUXT_PUBLIC_API_BASE=/api` and configure Nitro's proxy to send `/api/**` to `http://localhost:3000/api/**` during dev. This way, when the Nuxt app (running on port 3001) calls `/api/whatever`, it forwards to Rails on port 3000. Another simpler approach is to disable strict CORS in development and have Nuxt call `http://localhost:3000/api`. (Be sure to enable correct same-site/cookie settings if doing so.)

If using a catch-all as in the static integration, in development you might *not* use the catch-all so that you can still use Nuxt's dev server for HMR. For example, you could conditionally mount the wildcard route only in production, and in development run Nuxt separately. The StackOverflow answer above followed that: in development they ran two servers, and only in deployment did Rails serve the static files [18].

Code Splitting, Hydration, and Assets

Nuxt 3, by default, builds your application with code-splitting: each page and common chunks are split into separate `.js` files for performance. It also produces a server bundle (for SSR) and a client bundle (for hydration). When using SSR (Rails or Nuxt), it's critical that the HTML output and the client JS are in sync so hydration can occur seamlessly.

Hydration & State Transfer: In SSR, the server-rendered HTML is sent to the browser, and then the Vue app hydrates (attaches) to that HTML. To do this, Nuxt will embed a state object (`window.__NUXT__` or similar JSON) in the HTML containing data fetched during SSR. If Rails is doing the SSR, you must ensure a similar mechanism – i.e., pass the initial state from Rails to the Vue app. For example, with Humid or Hypernova, you might embed a JSON blob in a script tag or directly pass props. Humid's usage example shows passing `initial_state` into the render call and getting HTML [19]. You would then need to output the `initial_state` again for the client script. Nuxt 3's server will handle this automatically if it is doing SSR; if Rails is doing SSR, you might be using lower-level Vue SSR utilities (like Vue's `@vue/server-renderer`). Ensure that the client-side bootstrapping script is included and pointed at the correct assets.

Asset Handling: You will need to serve Nuxt's generated assets (JS, CSS, images). There are a few scenarios:

- With a **Nuxt SSR server** (Node), the Nuxt process can serve its own assets (Nuxt's production server will by default serve the contents of `.output/public` on the `_nuxt/` path). In

production, you might instead deploy these static files to a CDN or serve them via Rails/nginx for efficiency, but it's not strictly required. If using an external CDN, you can set `app: { cdnURL: 'https://...'}` in nuxt.config or use an environment base URL for assets.

- With **Rails serving a static Nuxt build**, you would copy the compiled assets into `public/` (as in the earlier example). For instance, after running `nuxt build` or `nuxt generate`, copy the `dist/` or `.output/public` directory into `Rails.public_path`. Rails (through Puma or Nginx in front) will serve these files as static. Just ensure the base href or asset URLs in the Nuxt build are correct (you might set `router.base` if serving from a subpath).

- If using **Rails SSR with webpacker/jsbundling**: You might integrate Nuxt's build into Rails' asset pipeline. This is tricky because Nuxt is a full framework, not just a library. A simpler variant is to use **Vite** (with vite_ruby gem) and Vue plugins to manage front-end assets in Rails. In fact, some projects opt for *Inertia.js* or plain Vue with Vite instead of Nuxt, to simplify integration, but that sacrifices Nuxt-specific features. If you did want to incorporate Nuxt's output: you could run `nuxt build` as part of Rails' asset precompile, and have Rails serve the resulting files. Use the Rails helper `asset_path` if needed to refer to the files.

Links and Navigation: With a combined Nuxt+Rails app, you'll want to ensure that navigation links don't break the SPA experience. If you serve a static index with Nuxt as an SPA, make sure to use `<nuxt-link>` (in Vue components) for client-side nav. If a user manually refreshes or enters a URL, Rails' catch-all will serve the index again (ensuring the app loads). In SSR mode, users can navigate link-to-link without full reload (Nuxt can handle that as a single-page app once loaded, or even do soft reloads of segments if configured with client-side routing).

If you use Turbo (from Hotwire) in parts of the app, be careful: Turbo might intercept links and form submissions. It's usually best to **disable Turbo on Nuxt-managed links** (e.g., add `data-turbo="false"` on the `<a>` tags rendered by Vue, or simply not include the Turbo JS on pages where Nuxt is active). More on this in the Hotwire section.

Example: Static Nuxt served by Rails (Simplest Integration)

To illustrate one integration path: Suppose you decide *not* to use SSR initially, and just want Rails to serve the Nuxt app as a static SPA. You can:

1. **Build** Nuxt for production with `ssr: false` and `target: 'static'` (Nuxt config). This will generate a static `/index.html` and assets [18] [10].
2. **Copy** the generated files into `Rails.public_path` (or use a build script to do this).
3. **Add** a Rails route to catch-all and render `public/index.html` (as shown earlier).
4. In **development**, run `rails s` (which will 404 on non-API routes) *and* `nuxt dev`. You can navigate the frontend via Nuxt's dev server (with HMR), and configure your dev proxy or CORS for API calls to Rails. Alternatively, you can start Nuxt in `generate --watch` mode and let Rails serve the changing index.html, but that's more cumbersome than just running Nuxt dev.

This approach was reported to avoid CORS issues since the static Nuxt and Rails API are served from the same origin in production [10]. The drawback is no SSR (content is fetched after page load). You can later upgrade to a full SSR approach once the integration pieces (like auth) are figured out.

Now that we have the front-end/back-end interaction outlined, let's address **authentication**, since that is often the trickiest part of a Rails+SPA integration.

Authentication Strategies with Devise

The guide specifically calls out integrating **Devise** (popular Rails auth) with Nuxt. There are a few approaches to auth in this architecture:

1. **Traditional session-based auth (server-rendered, full page refresh):** Use Devise's default (cookie-based session) in a mostly traditional way, even though Nuxt is the frontend.
2. **Token-based API auth (e.g. JWT):** Treat the Rails backend as a pure API, and use token auth for the SPA (Devise with JWT or similar).
3. **Cookie-based API auth:** A hybrid where the SPA uses a cookie (typically a session cookie) for API calls, sharing it with Rails, without using explicit tokens.

We will cover each, with their implications.

1. Session-Based Auth (Devise + Server Sessions) with Nuxt

Rails' Devise by default uses cookies to maintain sessions (via Warden). In a classic Rails app, you'd render login forms server-side and set a session cookie upon login, then protected pages check `current_user` via that cookie session.

When introducing Nuxt as the frontend, **you can still use Devise's session cookies**, but you need to adapt how login/logout are performed:

- **Login via Rails form:** One option is to actually let Rails render the sign-in page and handle the form submission as usual, then redirect to the SPA. However, since Nuxt is handling your UI, you likely want the login page to be a Vue page for a seamless experience. You can achieve this by having Nuxt send the login request to Rails and Rails respond in a way that sets the cookie.

- **Using the Devise sessions controller (HTML):** If you send a normal form POST to `/users/sign_in`, by default Devise will set the session cookie and redirect (expecting an HTML workflow). You could allow the redirect to go to, say, `/` which your Nuxt app can handle (since the user is now logged in). However, dealing with CSRF tokens in this case is important (Rails will expect a CSRF token with the form submission). You can embed the token in the form via a meta tag and have Nuxt include it. Alternatively, you can relax CSRF protection on the session endpoints or use the `protect_from_forgery` with `with: :null_session` for API mode.

- **Using Devise with JSON:** Another approach is to override Devise sessions controller to accept JSON login requests. Devise can be configured to check `request.format`. For example, if `sign_in.json` is called with valid credentials, you manually sign in the user (`sign_in(user)` in controller) which sets the session, and respond with JSON (maybe `{ success: true }`). The Set-Cookie header for the session will be in the response. The Nuxt app can then proceed knowing the user is logged in (perhaps by another call to fetch the user's profile, or by decoding a user id that you send back).

Cookie considerations: If Nuxt and Rails share the **same parent domain**, the cookie set by Rails (say `yourapp_session`) can be made available to the Nuxt app's requests. For example, if both front and back are on `example.com` (with Nuxt at `example.com` and Rails API at `example.com/api`), the session cookie domain should default to `example.com` and it will be sent on each XHR automatically. If you are using *different subdomains* (e.g., `api.example.com` for Rails and `app.example.com` for Nuxt), you must configure Devise's cookie to be shared. In Rails, set

`config.session_store :cookie_store, key: '_yourapp_session', domain: :all, same_site: :none, secure: true` (for cross-site cookies). Modern browsers block third-party cookies unless `SameSite=None` and secure. So to allow an SPA on one subdomain to use a cookie from another, you need those flags. One StackOverflow discussion highlighted that an API subdomain's cookie won't be sent by the browser to the main app domain without proper SameSite settings [20] [21]. The simplest solution is actually to avoid cross-domain — e.g., serve both under the same domain and use path-based routing.

After login: Once the session cookie is set, you can call Rails API endpoints that are protected by Devise's `authenticate_user!` normally; Rails will see the session cookie and consider the user logged in. This works whether those calls come from the browser (XHR with cookie) or from the Nuxt SSR server (Node making a request on behalf of the user, where you'd need to forward the cookie header from the incoming request). Nuxt's Nitro allows access to the request headers on server side – you can forward cookies when using server-side fetch. For example, if using `useFetch` or `$fetch` in Nuxt 3, on SSR it can send along cookies if you use credentials. Ensure your fetch library is configured with `credentials: 'include'` and you might need to manually forward the cookie header to the backend in SSR.

Logout: This can be done by calling Devise's sign_out path (DELETE `/users/sign_out`) which will clear the cookie. You can trigger that via an Axios/fetch call or a form submission. Devise might require CSRF token for that request (since it's a state-changing action). You could also implement a simple API endpoint that calls `sign_out` and returns JSON.

Pros: This approach uses tried-and-true Devise sessions. You benefit from secure, HttpOnly cookies and Rails' session management. You don't have to implement token logic or store tokens in the client. Also, you can use the full Devise modules (confirmable, lockable, etc.) without re-implementing in the front-end.

Cons: It's somewhat tricky to set up initially because of CSRF and domain cookie issues. You must ensure the Nuxt app can get the CSRF token to include in login requests or disable CSRF for those routes. A technique to handle CSRF in SPA is to have Rails issue the CSRF token in a response (or set it in a cookie) so the SPA can use it [22] [23]. Also, if your front-end is at a different origin, dealing with third-party cookies is troublesome (as mentioned, better to unify the origin or use tokens instead).

Devise and Turbo: Note, Rails 8/Devise might use Turbo by default for authentication (Devise 4.8+ integrates with Turbo for redirect after sign in). In a SPA context, you should disable or ignore those, since Turbo isn't controlling the page loads for login here – Nuxt is.

2. Token-Based API Auth (JWT)

Using JWTs (JSON Web Tokens) or another token-based auth is often the go-to for SPAs. Devise has the **devise-jwt** gem which integrates JWT into Devise's strategies [24]. Essentially, instead of a session cookie, a JWT is issued on login and must be sent on each subsequent API call in an Authorization header.

How it works: After a user logs in (via a JSON API request to Rails), Devise-JWT will generate a token (often in the `Authorization: Bearer <token>` header or in the JSON response). The Nuxt app receives this token and stores it (commonly in memory or localStorage). For security, you might choose httpOnly cookies for tokens too, but that reintroduces some same-site issues; many just store in memory and use a refresh token if needed.

On each API request, the Nuxt app's client (Axios, $fetch) adds `Authorization: Bearer <token>`. Rails (Devise-JWT) will decode this and authenticate the user for that request. No server-side session is needed (Devise-JWT uses Warden but doesn't rely on cookies). Devise-JWT can also handle revocation strategies (so you can blacklist a token on logout to prevent reuse [25]).

Nuxt integration: You might implement a simple auth store in Nuxt (using Pinia or just useState) that holds the JWT and user info. Nuxt middleware can check auth status to redirect to login if needed. There are also community modules (in Nuxt 2 there was @nuxtjs/auth; for Nuxt 3, one might use simple composables or community-auth module). Essentially, after login, set the token in a global state and set axios defaults: `axios.defaults.headers.common['Authorization'] = 'Bearer ' + token`. Also handle token expiration (Devise-JWT tokens by default expire, you'd need to refresh them by re-login or implement a refresh endpoint).

Pros: This is fully decoupled and stateless. It works fine across domains (no cookies to worry about). The SPA can easily check if a token exists to know if user is logged in. It's also scalable – no session store needed (the token itself is the auth proof). Mobile apps or other clients can reuse the same API.

Cons: Managing JWTs has its own complexity – token expiration, storage (avoid localStorage if XSS is a concern, though SameSite=None cookies of JWT have their own Csrf issues). Also, you lose some of the convenience of sessions (e.g., no automatic logout on browser close unless you implement it, no built-in CSRF protection – but if you're not using cookies, CSRF is less of an issue). Logging out a JWT typically requires server-side revocation (Devise-JWT supports blacklist or whitelisting strategies). If your app doesn't need to be fully stateless, this might be overkill.

However, for an API-centric design, JWT is very common. The Medium article by Quinn Daley demonstrates using Devise-JWT with Rails API and a Nuxt frontend [26] [24] – you can refer to it for step-by-step setup. Key steps include adding `devise-jwt` gem, configuring it in Devise (in `config/initializers/devise.rb` you set the JWT secret and mappings), and adding the `jwt_revocation_strategy` to your User model [27]. Then update your Devise routes or controllers to handle JSON.

One tip: Make sure to set `config.jwt` in Devise to place the token in response body or header as you prefer. The standard is Authorization header.

3. Cookie-Based API Auth (Session Cookies without Full Reloads)

This strategy is somewhat a hybrid of the above two. The idea is to use the **Rails session cookie** for API authentication in a SPA context, but still operate the front-end as an SPA. In effect, it's similar to #1, but we treat the app as an API from the start (no need to render HTML from Rails). This can be achieved by enabling sessions in your Rails API and using cookie-based auth on XHR requests.

Setup: By default, a Rails API mode app might not include session middleware. You'd need to add `config.middleware.use ActionDispatch::Cookies` and session store configuration, etc., so that Rails can issue and read session cookies. Once that's in place, Devise (even in API mode) can use the session. You then proceed similarly to #1: the Nuxt app hits a login endpoint, Rails sets a cookie.

The difference from #1 is that we won't rely on any server-rendered pages at all – all pages are through Nuxt. The cookie is purely a means of auth state storage (like a token, but managed by the browser cookie jar). For instance:

- Nuxt hits POST `/api/sign_in` with credentials (could be a Devise custom controller that responds with JSON).
- Rails Devise signs the user in, and returns JSON `{ success: true, user: {...} }` and crucially sets `Set-Cookie: _yourapp_session=...; HttpOnly`.
- The browser stores this cookie. From now on, any subsequent `$fetch` or Axios call to the Rails API (same domain) will automatically include `_yourapp_session` cookie, so Rails knows who the user is.
- To get the current user at app startup, Nuxt could call `/api/current_user` – Rails will see the session cookie and respond with the user info if logged in.

This is very convenient: essentially it makes your SPA *stateful* via cookie, avoiding manual token handling. But you must address **CSRF** for state-changing requests. Since you are using cookies, you should protect against CSRF. Normally, APIs using tokens avoid CSRF because no cookie = no automatic credential. But here, your cookie *will* be sent on cross-site requests too (unless SameSite rules mitigate it). If your front-end and back-end share a top-level domain (likely), an attacker site could potentially trigger requests to your API from a user's browser. To prevent that, ensure one of: (a) all non-GET API endpoints require a valid CSRF token header, which your Nuxt app provides, or (b) set the session cookie SameSite=Lax/Strict so it's not sent on cross-site contexts (which might already be default). Rails by default sets session cookies to SameSite=Lax, which *does* allow cookie on top-level navigations (like clicking a link to the site) but not on background requests from another site. For APIs, that's probably sufficient, but adding CSRF protection on APIs is an extra safety net.

In practice, some developers choose to **disable Rails' CSRF check** for API routes (using `skip_before_action :verify_authenticity_token` in controllers), accepting the risk, especially if SameSite protections are in place. Alternatively, they issue a CSRF token as discussed earlier (Rails can send the token in JSON after login, and Nuxt can store it and include it in an `X-CSRF-Token` header on mutations).

Sharing cookies between Nuxt SSR and Rails: If you run Nuxt SSR in a separate Node server, you'll need to forward the cookie from the incoming request to Rails when doing server-side data fetching. For example, if a user hits `example.com/dashboard`, the request goes to Nuxt SSR (with cookie). Nuxt's server calls `GET /api/dashboard-data` to Rails. To maintain auth, that call must include the cookie. You can grab it in Nuxt via the request headers (`useRequestHeaders(['cookie'])` in Nitro) and include it in your server-side fetch. This way Rails sees the session and returns the appropriate data for SSR.

Logout: Rails can simply forget the session server-side (if using an in-memory/DB store) or you can instruct the browser to delete the cookie (set it expired). If using cookie store (default in Rails), clearing it client-side (through a Set-Cookie with empty value or client JS to remove cookie if not httpOnly) will effectively log out.

Pros: No token handling in JS, and no double-app maintenance. It's secure (cookies are httpOnly and signed) and leverages Devise fully. It's almost like the user is using a regular Rails app, except the UI is Nuxt.

Cons: It's somewhat non-standard for APIs – many API clients (like mobile apps) can't use this method easily since cookies are browser-specific. It ties your API to browser usage with cookies. But if your only client is the Nuxt app, it's fine. Also, as mentioned, careful with CSRF and SameSite settings.

Devise configurations: In Rails 8, Devise might require some tweaks in initialization if being used in API-only mode with sessions added back. Ensure `Devise.setup` has `config.navigational_formats` including `:json` if you use JSON requests for sign in, or it might treat it as navigational and redirect. You might also set `config.skip_session_storage = [:http_auth]` (keep :cookie). Devise-JWT would not be used in this approach (it's purely session-based).

Devise + Nuxt: Additional Tips

- **Devise Modules**: If using confirmable, etc., you'll need to handle those flows (e.g., confirmation emails have links – those could be handled by Rails views or you make Nuxt routes that hit the confirmation endpoint). For simplicity, you might disable modules that require server-rendered pages or implement custom controllers to return JSON.

- **Two-Factor or OAuth**: Those require additional steps. For instance, for OmniAuth (sign in with Google), you might do the OAuth flow on the Rails side and then redirect with a token or set a session, then Nuxt continues. That can be complex – possibly treat it as a separate case outside this guide's scope.

- **Development**: In dev, if Nuxt is on a different port (domain `localhost:3000` vs `localhost:3001`), cookies are domain-bound. `localhost` cookies are shared across ports (they are domain-scoped, not port). So if your Nuxt dev is at `localhost:3000` and Rails at `localhost:3001/api`, a cookie set by Rails for domain `localhost` will be sent to Nuxt dev domain as well (provided SameSite allows). You might still run into CSRF issues because different port might be seen as different origin for SameSite. Often, developers disable SameSite or CSRF in dev to simplify.

- **Integration Testing**: Consider writing end-to-end tests (with something like Cypress or Capybara + headless Chrome) to ensure the login flow works as expected in the browser with the combined system.

Now that auth is covered, let's discuss **Hotwire/Turbo vs Nuxt** and how (or whether) to use them together.

Hotwire (Turbo) in Parallel with Nuxt

Hotwire (specifically Turbo Drive/Frames/Streams) is Rails' answer to making server-rendered apps feel more dynamic by *replacing HTML over the wire* instead of full page loads [28] [29]. In a way, it gives a SPA-like experience without a separate front-end framework by letting Rails deliver HTML partials that update the page. Turbo can intercept link clicks and form submissions to do this seamlessly.

When you adopt Nuxt for your UI, you are firmly in the SPA camp for those parts of the application. Using Hotwire and Nuxt together in the **same portions** of the app is usually not advisable – they would

conflict (Turbo intercepting links that Nuxt expects to handle, etc.). However, there are scenarios where you might use both in one application **for different sections**:

- You may have a largely Rails-rendered admin interface where Turbo is used for interactivity, while the main end-user interface is a Nuxt app. This could happen if you're incrementally adding a SPA to a legacy Rails app: some pages become "Nuxt pages" while others remain ERB + Turbo.

- Or you might enhance a mostly Nuxt app with a Turbo-powered widget for something like real-time updates via Turbo Streams from Rails ActionCable.

Trade-offs and compatibility:

- **Routing**: If Turbo Drive (the navigation interceptor) is enabled on a page that is a Nuxt app, clicking a `<a>` might cause Rails to return an HTML page (which could either be a full page or possibly a JSON if not expected). This would break the Nuxt flow. To avoid this, you can disable Turbo on specific links or globally when Nuxt is active. One strategy: only include the Turbo JS on pages that are purely Rails-rendered. If your Nuxt app is mounted at a certain DOM element, you could configure Turbo to ignore that element (`data-turbo="false"` on a parent div). In summary, **don't let Turbo hijack links that should go through Nuxt**.

- **Coexisting**: It is technically possible for a Rails view to include a Turbo Frame that loads a Nuxt app within it, but that's likely not useful. More plausibly, a Nuxt page could include an `<iframe>` or an embed that sources a Rails URL with Turbo – but again, mixing them in one view will complicate things more than solve.

- **Instead of Nuxt**: It's worth noting why you'd choose one over the other. Hotwire excels if the bulk of your application logic can be done on the server (Ruby) and you want to avoid maintaining two stacks. It keeps state mostly on the server, which is great if your complexity is around database state [30]. Nuxt (or any FE framework) is better if your complexity is around rich client-side interactions, custom UI components, offline capability, etc. [31]. In many cases, you won't need both – you'll choose one approach globally. But the question explicitly asks for using Hotwire *in parallel*, so perhaps you have a scenario where part of the app is highly interactive (justifying Nuxt) and other parts are simple CRUD (where Hotwire could reduce overhead). A **hybrid approach** is mentioned by some as viable: "a mostly Hotwire app with small self-contained SPAs embedded in pages" [32].

Example hybrid: Imagine a Rails 8 app that serves mostly standard pages with Turbo (so navigation is fast and HTML is streamed). Now suppose one page needs a complex data visualization component – you could build that one page as a Nuxt app. You'd have a route for it, and when the user navigates there via Turbo, you might actually want to *disable Turbo for that navigation* so that Nuxt can boot properly (Turbo might try to replace the `<body>` which would interfere with Vue app initialization). Alternatively, link to that page with a regular link (full load) which triggers Nuxt app load. From the user's perspective, it's not too jarring if it's a distinct section.

Turbo Streams: You could potentially use Turbo Streams (server-initiated updates via websockets or SSE) even in a Nuxt app by including the `@hotwired/turbo` library and listening to stream messages to update parts of the DOM. However, mixing direct DOM manipulation (Turbo Streams) with Vue's virtual DOM can be problematic – Vue won't be aware of changes Turbo makes. If you needed real-time updates in Nuxt, you'd likely use a Vue-compatible solution (like VueSocket.io or Pinia store updates from an SSE endpoint). So Turbo Streams aren't very useful inside a Vue-managed DOM.

Recommendation: If your project is new, decide upfront if you want to go the Hotwire route or the Nuxt route for the majority. Mixing them should be done only if you have a clear delineation of concerns (and perhaps separate routes/subdomains for each to avoid conflicts). When mixed, carefully scope where each applies: e.g., Rails controllers A, B, C render views that use Turbo, and controllers X, Y, Z (or maybe a mounted Engine or a catch-all) serve the Nuxt app. Each can use what it's best at. Always test navigation flows to ensure, for instance, going from a Nuxt page to a Turbo page and back doesn't break session or state.

In summary, Hotwire and Nuxt can technically coexist, but treat them as oil and water – keep them mostly separated to avoid confusing behavior. Hotwire gives you an **HTML-first approach** (server renders HTML, minimal JS) [29], whereas Nuxt gives you a **JS-first approach**. They solve similar problems in different ways, so double-implementing features (e.g., form submissions) in both is unnecessary overhead unless you're transitioning from one to the other.

Database Considerations (PostgreSQL & MongoDB)

By default, Rails 8 (and earlier) works with PostgreSQL out of the box (ActiveRecord). We assume PostgreSQL is your primary data store. Incorporating **MongoDB** as a secondary store is possible and sometimes useful (e.g., using Mongo for certain documents or caching, while using Postgres for core relational data). Here's how to set up both:

PostgreSQL Setup

In Docker/Kubernetes, you will likely run Postgres as a separate service (or use a managed DB service). Rails will connect via environment variables. Ensure you have your `DATABASE_URL` or individual PG env vars (`PGHOST`, `PGUSER`, etc.) set in the Rails container.

When containerizing Rails, you might use a lightweight Postgres client library (the `pg` gem) which requires native extensions. In your Dockerfile for Rails (detailed later), ensure the Postgres client libraries (`libpq-dev` on Debian/Ubuntu) are installed during gem install so `pg` can compile.

For Kubernetes, you might deploy a Postgres StatefulSet or use a hosted DB (the latter is often easier in cloud). Rails 8's config (database.yml or ENV) should point to the DB service (e.g., `postgresql://user:pass@postgres-service.default.svc:5432/dbname` as an example DSN in k8s).

Use Rails migrations as usual to manage your schema in Postgres. If you plan to also use MongoDB, consider which data goes where to avoid duplication.

Adding MongoDB via Mongoid

Mongoid is the official ODM (Object-Document Mapper) for MongoDB in Ruby/Rails. You can include the `mongoid` gem in your Rails Gemfile to use Mongo in addition to ActiveRecord. Mongoid can be used alongside ActiveRecord in the same Rails app – they operate independently (you'll just have Models that inherit from `Mongoid::Document` instead of `ApplicationRecord`).

Setup steps:

1. Add `gem 'mongoid'` (ensure version compatible with MongoDB server version; as of Rails 8, Mongoid 7 or 8 likely).

2. Run `rails g mongoid:config` which generates a `mongoid.yml` in your config. In that file, configure your MongoDB connection (you can set it to use ENV variables for host, database, user, password).
3. By default, the generator might disable ActiveRecord in `application.rb`. **Do not disable ActiveRecord** if you want both; you can manually re-enable AR (Rails allows using both, but the generator assumes you might switch entirely). To use both, keep your `database.yml` for AR and `mongoid.yml` for Mongoid.

Now you'll have two sets of models: - ActiveRecord models (for Postgres) use `ApplicationRecord < ActiveRecord::Base`. - Mongoid models include `Mongoid::Document`. For example:

```
# app/models/user.rb (ActiveRecord example)
class User < ApplicationRecord
  has_many :user_logs  # relation to a Mongoid model? not directly possible, see below
end

# app/models/user_log.rb (Mongoid example)
class UserLog
  include Mongoid::Document
  field :message, type: String
  field :created_at, type: Time
  # potentially reference to User
  field :user_id, type: Integer
  index({ user_id: 1 })
end
```

You can't use ActiveRecord associations directly with Mongoid, but you can manually handle relationships (like store a user_id in a Mongo document and query accordingly). Mongoid does support referencing relations via its own mechanism if both sides are Mongoid, but here only one side might be.

Use cases for multiple DBs: maybe you use Postgres for transactional data and Mongo for logging or analytics (where flexible schema is nice). This is fine. Just ensure your transactions don't need to span both (they can't, easily).

From a Docker perspective, you'd also run a MongoDB container or use a managed Mongo service. If running in Kubernetes, you might deploy Mongo as a StatefulSet, or if using a cloud like AWS, perhaps use DocumentDB or Mongo Atlas (external).

Mongoid vs ActiveRecord concurrency: They are separate; keep in mind Mongoid has its own query API which differs from ActiveRecord queries. Developers need to be aware of which ORM to use for which model.

One catch: If you have a Rails app that uses both AR and Mongoid, your tests and setup need to handle both (e.g., cleaning DBs, seeding data in both). But at runtime it's straightforward.

Rails 6+ has support for multiple databases in ActiveRecord, but that's for multiple relational DBs. It does not cover MongoDB, which is a different adapter entirely – so using Mongoid is the way to go.

Alternative approach: Instead of using Mongoid in Rails, you could have a separate microservice for the part that needs Mongo, or have the Nuxt app talk to Mongo (via an API or directly via a Node package if security allows). But assuming you want Rails to interface with Mongo, Mongoid is the solution.

Configuration in Kubernetes

For Postgres, you'll mount a secret containing database credentials. For Mongo, similarly. In `mongoid.yml` and `database.yml`, reference environment variables. Example `mongoid.yml` snippet for production:

```
production:
  clients:
    default:
      uri: <%= ENV['MONGODB_URI'] %>
      options:
        server_selection_timeout: 5
```

Where `MONGODB_URI` might be like `mongodb://user:pass@mongo-svc.default.svc:27017/mydb`.

Be mindful of network: if Rails and Mongo run in same cluster, use the service DNS name. If using an external Mongo (like MongoDB Atlas), you'll provide the full URI and likely need to allow IP of your cluster.

One more note: If using both DBs, think about migrations for Mongo. Mongoid doesn't use migrations (since schema is usually dynamic). But you might still need to create indexes in Mongo – you can either do that via Mongoid's `index` definitions (which can be created with a rake task) or manually. For example, running `Mongoid::Tasks::Database.create_indexes` will create indexes defined in models. Ensure to incorporate that in your deployment (maybe in a startup task or manually).

Useful Libraries and Modules

To simplify the integration, consider these tools and packages:

- **Devise JWT:** As discussed, `devise-jwt` gem handles JWT auth nicely within Devise [24]. Use this if you go the token route, to avoid writing JWT encode/decode logic yourself.

- **Rack CORS:** If your Nuxt and Rails are on different domains (especially in development or even production (e.g., different subdomains)), you'll want to enable CORS on the Rails API so that the browser allowed the requests. The `rack-cors` gem can be configured to allow the Nuxt origin to access the Rails API. E.g., in `config/initializers/cors.rb`:

  ```
  Rails.application.config.middleware.insert_before 0, Rack::Cors do
    allow do
      origins 'http://localhost:3000', 'https://yourapp.com'  # front-end origins
      resource '*', headers: :any, methods: [:get, :post, :put, :delete, :options], credent
  ```

```
  end
end
```

The `credentials: true` is important if using cookies, so that cookies are allowed in cross-site requests.

- **Nuxt Auth (Third-party):** While Nuxt 3 doesn't yet have an official auth module like Nuxt 2 did, there are community efforts. Modules like `@sidebase/nuxt-auth` or using the underlying NextAuth.js with Nuxt as an experiment exist. But you can achieve what you need with basic composables. Possibly simpler is writing a plugin that on boot checks localStorage for a token or calls `/api/current_user` to set user state.

- **Pinia (state management):** If your app requires global state (like user info or cart info), use Pinia (the Vue 3 store, successor to Vuex). Nuxt 3 has built-in support for Pinia. For SSR, Pinia state can be serialized and injected into the Nuxt payload for hydration, so your user state persists across refreshes without an extra API call (if you fetch it during SSR).

- **Inertia.js:** This is an alternative to consider if you hadn't fully committed to Nuxt. Inertia allows you to build Vue pages while keeping Rails routing. It's sort of a middle ground – you write Vue single-file components for pages, and Rails controllers render them as responses. Inertia passes data as props to those components. The advantage is you don't need to build a separate API – controllers supply data just like Rails views, but the view is a Vue component. This can simplify auth (it's just Rails sessions) and use of Rails helpers, etc. However, using Nuxt with Inertia is not typical (Nuxt is a full framework itself, while Inertia gives you more manual control). If someone is deeply evaluating options: Nuxt for a full SPA experience vs Hotwire for minimal JS vs Inertia for a hybrid – each has merits. Since the question is Nuxt-focused, we stick to that path, but it's good to be aware of Inertia as a potential *gotcha* if someone expected an easier integration.

- **Foreman or Overmind:** For development, running two servers (Rails and Nuxt) can be streamlined using a process manager. A tool like **Foreman** (Procfile-based) or **Overmind** can run both and aggregate logs. You can create a `Procfile.dev`:

```
rails: bin/rails server -p 3000
nuxt: npm run dev -- --port 3001
```

Then `foreman start -f Procfile.dev` to start both. This way, developers just run one command to get both parts running.

- **Testing and Linting:** Use ESLint for Vue/Nuxt code and RuboCop for Ruby code to keep consistency. They will be in the same repo, so maybe segregate config or use package-specific commands.

- **gotchas/edge cases:**

- **Asset path differences:** If Rails serves assets, ensure Nuxt is configured to not also expect to serve them. For example, if using Nuxt static generation, by default it might expect to serve from root. But if you mount your app under a subdirectory or via rails, adjust `router.base` or asset URLs accordingly.

- **Time zone and locale:** Both Rails and Nuxt may have localization. If you plan to use Rails for i18n (e.g., error messages from Devise) and Nuxt for UI, you might want to share locales or at least ensure consistency. There's no automatic bridging, but you could generate a JSON of Rails i18n and use it in Nuxt.
- **Date formats & JSON serialization:** When Rails sends JSON (like a model to_json), it might include e.g. created_at in a certain format. Make sure your front-end can parse it (perhaps use libraries like Day.js or date-fns to format). Similarly, if using Mongo, the ObjectIDs will be strings – plan if you need to expose them.
- **Large payload SSR**: If a page requires a lot of data, SSR will pull it all at once. Consider pagination or incremental loading to avoid huge HTML. Also configure caching if possible (Rails can cache API responses, Nuxt can cache SSR pages via static generation or Nitropack).
- **SEO meta tags**: Nuxt can manage `<head>` tags easily (with useHead or head() in pages). If Rails is doing SSR, you might need to have Rails set meta tags too (perhaps via data passed to the Vue app). With a Node SSR (Nuxt), you get Nuxt's head management automatically.

With integration and development covered, let's move to **containerization and deployment** specifics in Docker and Kubernetes.

Docker & Kubernetes Deployment

Deploying a Rails + Nuxt app in Docker/K8s involves creating container images for the web services and managing them in pods with appropriate configuration. We will discuss:

- Dockerfiles for Rails and Nuxt (with multi-stage builds for efficiency).
- Docker Compose usage (for local development or initial testing).
- Kubernetes considerations: separating containers vs combined, health checks, config maps, etc.
- Production build processes (assets precompilation, etc.).

Dockerizing the Rails 8 API

Dockerfile (Rails): We want a reproducible image for the Rails 8 app. A common pattern is multi-stage: one stage for building gems (and assets), and a smaller stage for runtime.

Here's an example `Dockerfile` for Rails 8:

```
# Stage 1: Build environment
FROM ruby:3.2.2-slim AS builder  # Ruby 3.2 for Rails 8
# Install Linux packages needed for build
RUN apt-get update -qq && apt-get install -y build-essential libpq-dev git
# If using Node for asset building (optional, e.g., if precompiling JS or using webpacker):
# RUN apt-get install -y nodejs
# If using Yarn: RUN npm install -g yarn

WORKDIR /app
# Install bundler
RUN gem install bundler:2.4.10

# Cache bundle gems
COPY Gemfile Gemfile.lock ./
RUN bundle config set without 'development test' && bundle install -j4
```

```
# If Rails has assets (like using Sprockets or importmap, or if copying Nuxt static files):
# COPY package.json yarn.lock ./   # if needed for assets
# RUN yarn install --production   # if using Yarn for assets

# Copy the rest of the app code
COPY . ./

# Precompile assets (if any Rails assets, e.g., CSS, or copying Nuxt build if static approach)
# RUN bundle exec rails assets:precompile

# Stage 2: Runtime environment
FROM ruby:3.2.2-slim AS final
# Install needed runtime packages (like Postgres client for pg gem)
RUN apt-get update -qq && apt-get install -y libpq5
# libpq5 is runtime lib for Postgres (no dev files needed)

WORKDIR /app
# Copy app from builder stage
COPY --from=builder /app /app

# Expose port (if needed, say 3000)
EXPOSE 3000

# Set environment variables
ENV RAILS_ENV=production
# Possibly SECRET_KEY_BASE, DATABASE_URL will be injected at runtime via env or secrets

# Command to run the server (could also use an entrypoint script for migrations etc.)
CMD ["bundle", "exec", "puma", "-C", "config/puma.rb"]
```

Explanation: We used Ruby 3.2 (Rails 8 is compatible). We installed dependencies and did `bundle install`. We did not install Node here – if your Rails app doesn't need Node (e.g., if you're not using webpacker or asset pipeline with JS, because Nuxt handles front-end), then you can omit Node. If you do need to compile assets (like using Tailwind via jsbundling or something), you might need Node in the builder stage only.

If you plan to **copy Nuxt static build into Rails** image (monolithic image approach for static serving), you could do the Nuxt build in a separate stage and copy results. For example, have another stage:

```
# Stage X: Nuxt build (if using static deployment in Rails)
FROM node:18 AS nuxtbuilder
WORKDIR /app
COPY frontend/package.json frontend/package-lock.json ./frontend/
RUN npm ci --prefix frontend
COPY frontend ./frontend
RUN npm run build --prefix frontend  # build Nuxt (assuming it outputs to .output/public or di
```

Then in the Rails builder stage or final stage, copy from nuxtbuilder:

```
COPY --from=nuxtbuilder /app/frontend/.output/public /app/public
```

This would place the Nuxt generated static files into Rails' public directory to be served. Also copy the SSR bundle if Rails is going to use it (for mini_racer SSR, e.g., `.output/server/index.mjs` could be included somewhere accessible to Rails – you'd likely put it in `app/assets/javascripts` or similar and have a initializer load it). This gets complex, so only do this if pursuing the Rails SSR route.

Docker Compose: You can create a `docker-compose.yml` to orchestrate Rails, Postgres, (Mongo), and Nuxt for local or dev deployment:

```
version: '3.8'
services:
  rails:
    build:
      context: .
      dockerfile: Dockerfile
    env_file: .env   # contains RAILS_ENV, DB config, etc.
    ports:
      - "3000:3000"
    depends_on:
      - db
      - mongo
    # command: to override CMD if needed, e.g., run migrations then puma
  nuxt:
    build:
      context: .
      dockerfile: Dockerfile.nuxt   # separate Dockerfile for Nuxt (see below)
    env_file: .env
    ports:
      - "3001:3000"   # Nuxt typically uses 3000 internally by default; map to 3001 externally
    depends_on:
      - rails   # if nuxt SSR needs rails up for data (not strictly required to start, but you
  db:
    image: postgres:15-alpine
    environment:
      POSTGRES_USER: youruser
      POSTGRES_PASSWORD: yourpass
      POSTGRES_DB: yourdb
    ports:
      - "5432:5432"
  mongo:
    image: mongo:5
    ports:
      - "27017:27017"
```

This is a simple example. It allows you to run `docker-compose up` to start everything. In this setup, Rails is on port 3000 (API), Nuxt on 3001 (web). You'd typically visit `localhost:3001` to hit Nuxt, which will internally call Rails on `rails:3000` (the container host name). You'd need to configure Nuxt's API

base URL to `http://rails:3000` in this case (since inside the Docker network, it can reach the service by name). Or use docker-compose's networking to your advantage.

Dockerizing Nuxt 3 (SSR): If you choose the separate Nuxt SSR service deployment, here's a Dockerfile for Nuxt:

```
# Dockerfile.nuxt
FROM node:18-alpine AS builder
WORKDIR /app
COPY package.json package-lock.json ./
RUN npm ci  # install dependencies
COPY . ./
RUN npm run build  # build the nuxt app (outputs to .output)

# Use a smaller runtime image
FROM node:18-alpine AS runner
WORKDIR /app
# Copy only necessary files from builder
COPY --from=builder /app/.output ./.output
COPY --from=builder /app/package.json ./
# If Nuxt needs node_modules at runtime (for Nitro) copy those:
# It depends on Nitro preset; typically .output/server/node_modules may exist if externalized
COPY --from=builder /app/node_modules ./node_modules

# Nuxt uses PORT env or config, default to 3000
ENV NODE_ENV=production
EXPOSE 3000
# Use nitro's output start script:
CMD ["node", ".output/server/index.mjs"]
```

After `npm run build`, Nuxt 3 generates the production-ready app in `.output`. That contains a `server/` directory (with the Node server code, possibly as an index.mjs entry) and a `public/` directory (client assets). Because we're in standalone mode (node:18-alpine), we copy those over.

We include `node_modules` because some dependencies might not be fully bundled. Nuxt 3's Nitro can do a fully bundled output (especially if target is Node), but by default it might externalize some dependencies (to reduce bundle size). Check the `.output/server/node_modules` – if present, copy it.

Alternatively, run `npm ci --production` in the runner stage to install just prod dependencies. But Nitro's output often is independent enough that you don't need the full source code or all deps, just the output.

One container vs two in Kubernetes:

- If using **two containers** (Rails and Nuxt separate), you'd create two Deployments (or one Deployment with two containers, but better separate for independent scaling). For instance, `deployment/rails-deployment.yaml` and `deployment/nuxt-deployment.yaml`. Each

would have its own pods, maybe in the same namespace. Then use a Kubernetes **Ingress** or **Service** setup:

- A Service for Rails (clusterIP, name it "rails-api") and one for Nuxt (name "nuxt-web").
- The Ingress (assuming using an Nginx ingress controller or similar) with rules:

```
rules:
  - host: "yourapp.example.com"
    http:
      paths:
        - path: /api/
          pathType: Prefix
          backend:
            service: rails-api
        - path: /
          pathType: Prefix
          backend:
            service: nuxt-web
```

This means any request starting with /api goes to Rails, everything else goes to Nuxt. Nuxt's own internal routing will handle the rest of the paths. This way the user always goes to `yourapp.example.com` for both API and front-end, satisfying cookie domain issues (same origin).

You might also add an exception for `/_nuxt/` or static assets if you decided to serve them via a different route. But if Nuxt service handles them or you route `/api` specifically to Rails and everything else to Nuxt, that includes `/_nuxt/` assets going to Nuxt.

- If using **one container** approach (Rails doing SSR or serving static, etc.), you just have one Deployment (Rails) and it listens on one port for everything. In that case, no special ingress rules needed beyond sending all traffic to that service. The Rails app itself distinguishes API vs front-end via routes as described.

Environment Configs: In Kubernetes, prefer using ConfigMaps and Secrets for configuration: - Database URL, Redis URL (if any), etc., can be in Secret. - Rails SECRET_KEY_BASE should be in a Secret. - Nuxt public runtime config (like API base URL) can be provided via env vars prefixed with `NUXT_` or specifically using the Nuxt config. For example, in nuxt.config:

```
export default defineNuxtConfig({
  runtimeConfig: {
    public: { apiBase: process.env.API_BASE || '/api' }
  }
})
```

Then in the app use `$config.public.apiBase`. In production, set `API_BASE=/api` or perhaps the full URL if needed. Since our ingress uses same domain routing, `/api` is fine as base.

- For cookie-based auth, consider setting the cookie attributes via environment: In Rails, `config.action_dispatch.cookies_same_site_protection = :lax` or `:none` could be set based on env if needed. Also `Devise.rememberable_cookie_options = { domain: ".example.com" }` if subdomain usage, etc.

Health Checks: - Rails: You might create a simple endpoint `/health` or use the default `/up` (Rails 8 might have one, as Rails 7 did for health check) that returns 200. Use that for k8s livenessProbe. Liveness and readiness can both hit it. Or use a TCPSocket check on port 3000 (less specific, but ok). - Nuxt: You can health-check by calling its `/` or better, create a lightweight endpoint. Nuxt 3 with Nitro can define a server API route for health. For instance, create a file `server/api/health.get.ts` exporting a small handler that returns 200 "ok". Then point readinessProbe to `/api/health` on Nuxt service. Otherwise, hitting `/` might trigger a full page render (heavier). Alternatively, use a TCP check if you assume if port is listening, it's ready (not always true if app still warming up though). - Postgres and Mongo typically don't need app-level health checks in K8s beyond their own liveness (the DB itself usually we assume is managed or not part of same app health check).

Scaling: With separate deployments, you can scale Rails and Nuxt independently. For example, if SSR is heavy, you might scale Nuxt pods to handle many concurrent renders, while Rails maybe can handle with fewer pods if it's mainly serving JSON. Or vice versa if your API is heavy and SSR light. If using session cookies, ensure a **shared session store** if not using cookie store. If you stick to cookie store (encrypted cookies), scaling is no problem – each Rails instance can validate the signed cookie without shared state. If you used server-side sessions (e.g., redis or DB), you'd need to ensure all Rails instances point to the same store.

Logging: Nuxt and Rails will have separate logs. In K8s you can aggregate them. Be mindful that request IDs won't span the two automatically. For debugging, it's sometimes useful to log an identifier from Nuxt and include it in API requests (maybe as a header) to correlate logs.

Production asset build: A build pipeline (CI/CD) should build the Docker images. In CI: - Run tests (Ruby and maybe headless browser tests for integration). - Run `docker build` for Rails and `docker build` for Nuxt. - Then push images to registry and deploy.

Make sure to run Rails migrations (e.g., via a Kubernetes Job or init container) before or during deployment. Also, if using Mongoid and you added indexes, ensure to run that.

If you are using **Nuxt static generation** (no SSR) served by Rails, your deployment process would involve generating those static files as part of Rails image build (as shown with an extra stage). So a single image can contain everything. That simplifies deployment to one service, but you lose SSR.

Security: Do not forget to set up Rails to force SSL (if behind load balancer, use `X-Forwarded-Proto` with `config.force_ssl = true` in production). Also configure secure cookies (if using cookies, mark them secure and HttpOnly). Ensure your CORS settings are restrictive in production (maybe only allow your own domain). And consider rate limiting on the API if needed (via Rack Attack gem).

Monitoring: In production K8s, monitor both apps. For Rails, tools like Skylight or NewRelic APM can be used. For Nuxt (Node), ensure you have memory/CPU limits and maybe some basic uptime monitoring. Node processes can consume memory – you might set `-max-old-space-size` if needed to cap V8 memory.

Resource Requests/Limits: Perhaps start Rails with e.g. 200m CPU, 512Mi memory, and Nuxt with 200m CPU, 512Mi memory as baseline, adjust as needed based on usage.

Finally, test the whole system thoroughly in a staging environment to ensure routing (especially cookies and domain) works in a production-like scenario (e.g., with actual domain and https).

By following this guide, you should have a solid foundation for integrating Nuxt 3 with Rails 8 in a containerized environment. We covered SSR options (Rails doing SSR via mini_racer or Node bridges [1] [5], vs Nuxt's own SSR), embedding Nuxt in Rails routing, handling assets and hydration, multiple auth patterns (Devise with sessions, JWT [24], etc.), using Hotwire sparingly alongside Nuxt, using Postgres and Mongo together, and best practices for Docker/Kubernetes deployment. With careful planning and testing, Nuxt 3 and Rails 8 can work in harmony – Rails providing a robust back-end and API (remaining "as normal as possible"), and Nuxt delivering a modern, engaging front-end experience.

---

[1] [2] [3] [15] [16] [19] **GitHub - thoughtbot/humid: Javascript Server Side Rendering for Rails**
https://github.com/thoughtbot/humid

[4] **GitHub - vueonrails/vueonrails: Rails gem with the power of Vue.js components**
https://github.com/vueonrails/vueonrails

[5] [17] **Rails + React Server Side Rendering, with Webpacker + Hypernova**
https://bessey.dev/blog/2018/08/04/rails-webpacker-react-ssr/

[6] [7] **GitHub - kevinluo201/render_vue_component: Build html block and pass Props for mounting Vue component in Ruby on Rails**
https://github.com/kevinluo201/render_vue_component

[8] **Rendering Modes · Nuxt Concepts**
https://nuxt.com/docs/guide/concepts/rendering

[9] [30] [31] [32] **Should you use Hotwire or a Frontend framework on your next Rails project? | Radan Skorić's website**
https://radanskoric.com/articles/hotwire-or-frontend-framework

[10] [14] [18] **configuration - Integrate Nuxt3 js as frontend in Ruby on Rails Project - Stack Overflow**
https://stackoverflow.com/questions/77227000/integrate-nuxt3-js-as-frontend-in-ruby-on-rails-project

[11] [12] [13] **GitHub - KevinBerthier/nuxt-rails-a-la-webpacker: Nuxt & Rails (api) in the same repo & deployed in 2 heroku apps**
https://github.com/KevinBerthier/nuxt-rails-a-la-webpacker

[20] [21] **ruby on rails - How are cookie-http-only sessions supposed to work on a SPA with a separate API server? - Stack Overflow**
https://stackoverflow.com/questions/60579293/how-are-cookie-http-only-sessions-supposed-to-work-on-a-spa-with-a-separate-api

[22] [23] **Rails CSRF protection for SPA**
https://blog.eq8.eu/article/rails-api-authentication-with-spa-csrf-tokens.html

[24] [25] [26] [27] **How to separate frontend + backend with Rails API, Nuxt.js and Devise-JWT | by Quinn Daley | Medium**
https://medium.com/@fishpercolator/how-to-separate-frontend-backend-with-rails-api-nuxt-js-and-devise-jwt-cf7dd9da9d16

[28] [29] **Hotwire: A new (old) approach for modern web applications**
https://www.codecentric.de/knowledge-hub/blog/hotwire-new-approach-for-modern-web-applications