ChatGPT

JWT Authentication in Decoupled Frontends (Next.js, Nuxt, Vue) with a Rails API

Architectural Overview of JWT-Based Authentication

Decoupled (or "headless") architecture means the frontend (built with frameworks like Next.js, Nuxt, or Vue.js) is completely separate from the backend (a Ruby on Rails API). Authentication in this setup is **stateless** – instead of using traditional server-side sessions (cookies tied to server memory or a database), the backend issues a **JSON Web Token (JWT)** upon successful login. The JWT is a signed token that encodes user identity and claims (like an expiration time). The client stores this token and sends it with each request to authenticate itself. The Rails API, knowing the secret key, can verify the token's signature and trust the claims to identify the user on each request, without any server-side session storage [1]. This approach scales well (no session store needed) and works across domains and clients (web, mobile, etc.).

How the JWT flow works: At a high level, the authentication process is as follows:

1. **User Login:** The user provides credentials (e.g. username/email and password) via the frontend app's login form. The frontend sends these credentials to the Rails API's authentication endpoint (e.g. an `/api/login` or Devise sign_in URL) over HTTPS.
2. **Server Verification:** The Rails API verifies the credentials against the database (for example, using Devise to check the email/password). If valid, the server creates a JWT containing the user's ID (and possibly other claims like roles or an expiry timestamp). It signs this token with a secret key known only to the server [1].
3. **Token Issuance:** The Rails API returns the generated JWT to the frontend. This could be in a JSON response body (e.g. `{ "token": "<JWT>" }`), or in an HTTP header (Devise-JWT by default adds it to the `Authorization` response header on sign-in) [2]. The JWT itself is just a Base64 string (header.payload.signature) that can be decoded by the server using the secret.
4. **Client Stores Token:** The frontend application receives the JWT and stores it for subsequent use. (We'll discuss storage options – cookies vs. localStorage vs. memory – in the next section.) [1]
5. **Authenticated Requests:** For each subsequent API request that needs authentication, the frontend attaches the JWT. Typically, this is done by setting an HTTP header: `Authorization: Bearer <JWT>` [3]. The Rails API sees this header, extracts the token, and verifies it on the server. If the token is valid (correct signature and not expired), the request is allowed and the server knows which user is making the request. If the token is missing, invalid, or expired, the API returns an unauthorized error (e.g. HTTP 401) and the frontend may prompt the user to log in again.
6. **Server Verification & Response:** On each request, the Rails backend uses its JWT library (e.g. the Devise-JWT gem or the `jwt` gem) to decode the token and validate the signature against the secret. It also checks token claims like expiration (`exp`). If valid, the server treats the user as authenticated (often setting `current_user` in Rails based on the token's user ID) and processes the request, returning the data. If the token is not valid, the server returns a 401 Unauthorized or 403 Forbidden response.

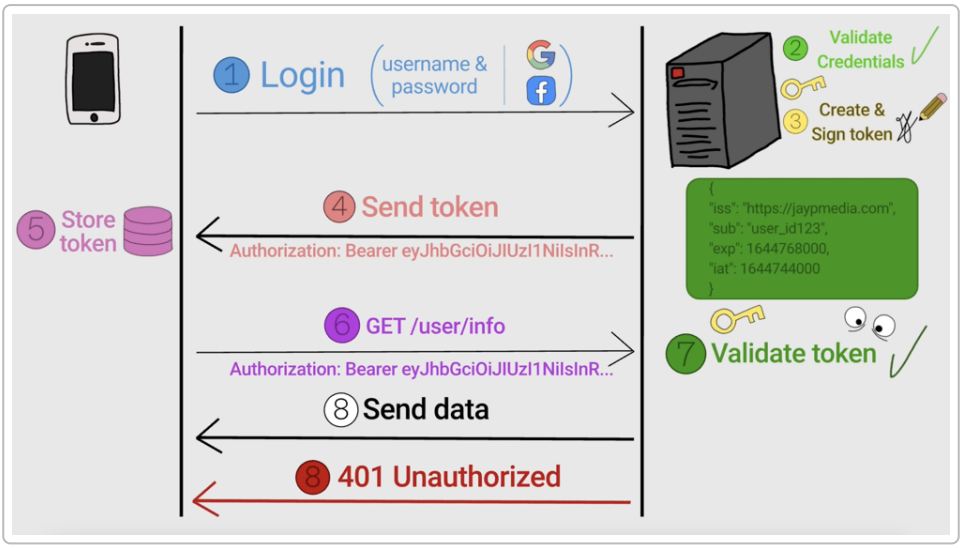

Illustration of a JWT authentication flow in a decoupled client–server architecture. The client logs in (1–3), receives a signed JWT from the server (steps 2–4), stores it (5), and includes the token in an "Authorization: Bearer …" header for subsequent API calls (6). The server verifies the token on each request (7) and returns the protected data if the token is valid (8), or an error if not (e.g. step 8 showing a 401 Unauthorized).

This stateless JWT approach means the server doesn't need to keep track of sessions – **the token itself carries the user's identity and session info**. Only the server's secret key is needed to verify tokens, and any server instance can do so (which is great for scaling horizontally) [1]. However, it also means that securing the token is critically important, since whoever possesses it can act as the user. In the next sections, we'll drill down into how tokens are stored and protected on the client, how the login flow is implemented in Next.js, Nuxt, and Vue, how to handle token refresh, and how to mitigate security risks like XSS and CSRF in this architecture.

Token Issuance and Storage Best Practices

Once the Rails API issues a JWT to the client, the frontend must store it somewhere to include in future requests. Choosing **where and how to store the JWT** on the frontend is a crucial security decision, as it affects exposure to attacks:

- **Browser Local Storage / Session Storage:** Storing the JWT in `localStorage` or `sessionStorage` is convenient because it's easy to implement and persists (localStorage persists across sessions, sessionStorage for a tab). Many simple SPA examples use `localStorage` to save the token and then read it for each request. **However, this makes the token vulnerable to XSS (Cross-Site Scripting) attacks.** If an attacker finds a way to inject malicious JavaScript into your app, that script can read localStorage and steal the JWT [4]. For example, any third-party script included on the page runs with the same privileges and could exfiltrate the token. This risk is significant: "An XSS attack happens when an attacker can run JavaScript on your website… the attacker can just take the access token that you stored in localStorage" [4]. Because of this, **many security experts advise against localStorage for JWT** (or at least stress that you must rigorously prevent XSS in your app). The main benefit of localStorage is that it's straightforward and works well with APIs that expect the token in an Authorization header (which you can set via JavaScript) [5], but you must weigh the XSS risk.

- **HTTP-Only Cookies:** A more secure alternative is to store the JWT in a cookie that is marked `HttpOnly`. An HttpOnly cookie is not accessible via JavaScript (document.cookie cannot read

it), which means even if an XSS attack occurs, the malicious script cannot directly steal the token from the cookie [6]. The browser will automatically include the cookie in requests to the backend domain (if CORS and SameSite policies allow), or you can manually forward it. By using cookies, you also leverage built-in cookie security flags: you can mark the cookie as `Secure` (so it only travels over HTTPS) and set `SameSite` restrictions to mitigate CSRF (more on CSRF shortly). In practice, many developers choose HttpOnly cookies for JWT storage precisely to **reduce the risk of client-side JS access** [7]. *For example, one developer notes that items in localStorage "can be read from JavaScript, making them vulnerable to XSS attacks," and thus decided to use an HttpOnly cookie to store the JWT instead* [7]. This aligns with OWASP recommendations and is generally considered a best practice for sensitive tokens.

- **In-Memory Storage:** Another approach is to avoid persistent storage entirely and keep the token in a JavaScript variable (memory) that is not written to disk. This means the token exists only as long as the single-page app is running and is lost on refresh or navigation (so the user would have to log in again or you'd need to rehydrate it some other way). Storing in memory (or a Vuex/Redux store without persistence) minimizes exposure – if an attacker loads your page in another context they can't get the token because it's not stored anywhere accessible. However, an XSS exploit running in *the context of your app* **can still read the in-memory token** (since the script runs in the same context). In-memory storage mainly protects against tokens lingering in storage if an attacker gains later access, and against some attacks like refresh token theft if you only keep short-lived tokens in memory. In fact, Auth0 recommends storing tokens in memory (ideally isolated in web worker context) as the most secure option for SPAs, albeit with the trade-off that tokens don't persist across page reloads [8] [9]. This approach often goes hand-in-hand with using refresh tokens (the memory-held access token can be short-lived, and you use a refresh token in a secure cookie to get new ones – see "Token Refresh" below).

In summary, **cookies vs. localStorage** is a trade-off between XSS and CSRF: localStorage is not automatically sent to the server (so it's immune to CSRF by default), but is readable by JS (so vulnerable to XSS); HttpOnly cookies cannot be read by JS (mitigating XSS token theft) [6], but are automatically sent to the server, which opens the door to CSRF attacks if not protected [10]. A community article sums it up well: both localStorage and cookies can be vulnerable to XSS, "but it's harder for the attacker to do the attack when you're using HttpOnly cookies," and cookies are vulnerable to CSRF but that "can be mitigated using the `SameSite` flag and anti-CSRF tokens." [10].

Best Practice: Many implementations use **HttpOnly cookies for JWTs** to minimize XSS risk, combined with **proper SameSite settings or CSRF tokens** to handle cross-site request forgery [10]. Another recommended pattern (often called "split token storage") is to store the short-lived access JWT in memory and store a longer-lived refresh token in an HttpOnly cookie [11]. This way, even if an attacker manages to run script on your page, they might steal the current access token (which expires soon) but *cannot steal the refresh token* to continue minting new tokens, limiting the damage [12]. We'll discuss refresh tokens shortly.

Secure transmission of the token: Regardless of where the token is stored, it should always be transmitted securely: - Always use **HTTPS** so that tokens (whether in headers or cookies) are encrypted in transit. An attacker sniffing network traffic should not be able to see the JWT. - When sending the JWT in an HTTP header (Authorization), the frontend code (e.g. via Axios or Fetch) should include it only to the correct API origin. For example, using Axios:

```
axios.get("/api/protected_resource", {
  headers: { Authorization: "Bearer " + token }
})
```

This is the typical pattern [3]. If using cookies, the browser will include them automatically in requests to the cookie's domain, or you may need to configure `fetch/axios` to send credentials. Make sure CORS is configured on the Rails API to allow the frontend origin and credentials if using cookies.

- Set cookie flags: `HttpOnly` (prevents JS access), `Secure` (HTTPS only), and `SameSite` (`Strict` or `Lax` if possible). In practice, **SameSite=Lax** is often a good balance: it allows the cookie to be sent on same-site navigations and top-level GETs (so if your frontend is on the same domain or a subdomain, API calls will include it), but it disallows sending the cookie from totally unrelated sites (mitigating CSRF in many cases). `Strict` is even stricter (cookie not sent on any cross-site request, even when following a link from another site), which can be very secure but might interfere if your auth flow involves redirects. Some implementations use `SameSite=None` (which means the cookie *will* be sent on cross-site requests) but **this must be accompanied by the** `Secure` **flag**, and in such cases you **absolutely need another CSRF mitigation** because other sites could trigger requests to your API. We'll touch on CSRF mitigation in a moment.

- **Token content:** Keep the JWT payload minimal – do not store sensitive data like passwords or personal info in it. Remember that JWTs are usually just **signed, not encrypted**. Anyone who intercepts or inspects the token can decode its payload (the data is just base64-encoded JSON) and read it; the secret key only protects against tampering, not viewing. So, treat the JWT as an *auth credential*, not as a data container. A good JWT will include an identifier for the user (subject claim or user ID) and an expiry (`exp`) and maybe a few authorization claims (roles, scopes), but nothing you wouldn't want to expose. As one source notes, "for signed non-encrypted tokens, we should not store sensitive info because JWT is protected against tampering but can be decoded and read by anyone" [13].

- **Token lifetime:** Always set an expiration (`exp` claim) on JWTs. Unlike server sessions which you could invalidate at any time server-side, a JWT by nature remains valid until it expires (unless you implement a revocation list). Setting a reasonable expiration (e.g. 15 minutes or 1 hour for access tokens) limits the window of risk if a token is stolen. Longer expirations increase the risk window but reduce how often a user has to re-authenticate or use a refresh token. Find a balance based on your app's sensitivity. (Often, a short-lived access token combined with a refresh token is ideal – see below.)

Now that the token is stored securely on the client, let's look at how the login and token usage flows work in specific frontend frameworks and how to implement best practices in each.

Login Flow and JWT Usage in Next.js (API Routes & SSR)

Next.js (React) applications often need to handle auth both on the client and potentially on the server (for server-side rendering). In a decoupled setup with a Rails API, there are two primary ways to handle JWTs in Next:

1. Using Next.js API routes as a proxy for authentication: Next.js API routes (or the newer App Router **Route Handlers** in Next 13+) can act as a middle layer between the frontend and the Rails API.

In this approach, the Next app doesn't talk to the Rails auth endpoint directly from the browser; instead, the browser calls a Next API route (e.g. `/api/login`) which then forwards the credentials to the Rails API, receives the JWT, and then sets a cookie. This approach is great for SSR and security because the JWT can be set as an HttpOnly cookie from the server side.

- **Login Process with API Route:** The user submits the login form on the Next.js app (e.g. clicking a login button). The client-side code calls the Next.js API route, e.g. `POST /api/login`, with the user's credentials. On the server, the Next API route handler receives the credentials and then performs a server-to-server call to the Rails API (for example, using `axios` or `fetch` on the server) to the Rails `/sign_in` endpoint. If Rails responds with a JWT, the Next API route sets a cookie on the response to the browser. For example, using Next 13's `cookies()` utility:

```
// Next.js route handler (app/api/login/route.ts for Next 13 App Router)
const res = await axios.post(`${RAILS_API_URL}/api/login`, { email, password });
const token = res.data.token; // assuming the Rails API returns { token: 'JWT' }
cookies().set("accessToken", token, {
  httpOnly: true,
  secure: true,
  maxAge: 60*60*24,
  sameSite: "strict"
});
return NextResponse.json({ success: true });
```

In this snippet, the Next API route sets an `accessToken` cookie that is HttpOnly, Secure, and Strict SameSite, containing the JWT [14]. The client will not be able to read this cookie via JS (HttpOnly), but it will be sent automatically on future requests to the Next app's domain. Essentially, the Next app is managing the JWT on the client's behalf. *One developer using this approach explained that using an HttpOnly cookie prevents JavaScript XSS from stealing the token, which was a key reason for this design* [7].

- **Using the Cookie for SSR and API calls:** Now that the JWT is stored in a cookie (say your Next app is running on `frontend.example.com` and cookie is for that domain, or even `.example.com` to share with API subdomain if needed), how do we use it? If your Next app needs to server-side render protected pages (for example, using `getServerSideProps` to fetch user data), it can read the cookie sent with the request. In Next API routes or `getServerSideProps`, you have access to `req.cookies` (in Next 12/pages router) or you can use the `cookies()` function (Next 13 app router) to retrieve cookies on the server. The Next server code can take the JWT from the cookie and attach it as a Bearer token when requesting data from the Rails API. For example, in `getServerSideProps`:

```
export async function getServerSideProps({ req }) {
  const token = req.cookies.accessToken;
  if (!token) {
    // Not logged in, redirect to login
    return { redirect: { destination: '/login', permanent: false } };
  }
  // Attach token to API call
  const res = await fetch(`${RAILS_API_URL}/api/some_protected_data`, {
    headers: { 'Authorization': `Bearer ${token}` }
  });
```

```
  // ... fetch data and return as props
}
```

In the above, the JWT is pulled from the cookie and used to authorize a request to Rails. Since this code runs on the server side, it's safe from exposure to the client. On the client side, for subsequent navigation or API calls, you have two choices: (a) continue to use Next API routes as a proxy (e.g. a Next API route for `/api/watchlist` that reads the cookie and calls Rails) or (b) call the Rails API directly from the browser and rely on the cookie being sent. If the Rails API is on a different domain, approach (b) would require CORS with credentials and likely setting the cookie domain to allow the browser to send it – often, it's simpler to use the Next server as a proxy so that all calls originate from the same domain. Approach (a) is essentially what the Next API route did for login: it can do the same for other requests, always reading the JWT from the HttpOnly cookie and forwarding the request to Rails. This keeps the JWT handling out of the client code entirely.

- **Protecting Pages and Routes:** Next.js allows implementing route guards through middleware or in page logic. For example, you can create a Next.js **Middleware** (a special `_middleware.js` or in Next 13, a middleware in `middleware.ts`) that runs on every request. This middleware can check `req.cookies` for the presence of a valid token (potentially even validate it using the same secret – though to validate a Rails JWT, the Next app would need access to the Rails JWT secret, which you might share via env variable). If the token is missing or invalid, you can redirect the user to the login page before rendering the protected page. Another simpler approach is to use client-side checks: e.g., if you have a React context or useState for auth, redirect in a `useEffect` if not logged in. But a **server-side redirect via middleware** provides a better UX by not even rendering the protected page HTML if not authenticated.

- **NextAuth or other libraries:** It's worth mentioning that Next.js has popular authentication libraries like **next-auth**. However, next-auth is more geared toward integrating with third-party OAuth providers or its own email/password flow; using it with a decoupled Rails API is not straightforward. (It can be done with NextAuth's Credentials provider, which essentially calls an API to verify credentials and returns a JWT or session – but next-auth would then manage its own session cookie). Many have found next-auth not directly applicable to external JWT auth [15], so a custom approach like described above is common. Another useful library is **iron-session** (for Next.js), which can encrypt data into a cookie. But since Rails is issuing JWTs, you might just stick to using that JWT rather than introducing a separate session mechanism.

2. Using Next.js purely client-side for auth: In this approach, you treat Next.js just like any SPA (React) – the client code directly calls the Rails API for login and gets back a token. You would then store it in localStorage or a cookie via client JavaScript. This is simpler but has downsides for SSR as mentioned. For example, after a successful login, you could do:

```
// In a Next.js React component (executing on client)
const res = await fetch(`${RAILS_API_URL}/api/login`, {
  method: 'POST',
  body: JSON.stringify({ email, password })
});
const data = await res.json();
localStorage.setItem('token', data.token);
// and set auth state...
```

Subsequent calls from the client can read `localStorage.token` and attach the Authorization header. The major caveat: if you server-render a page and need user data, the Next server won't automatically have access to that token (it's sitting in the user's browser localStorage). One could workaround by having the client send the token in a cookie anyway, or by doing a first pass render without user-specific data. Generally, if you need SSR, the cookie method is preferable. If your Next app is mostly client-rendered (or uses static generation), then a pure client-side JWT handling can work. Just be mindful of XSS/CSRF as discussed (localStorage use means XSS can steal the token; if you use a cookie via JS, that cookie cannot be HttpOnly if set by JS, so it's similar risk as localStorage).

Summary (Next.js): Next.js provides flexibility – for robust applications, using API routes + HttpOnly cookies is a best practice, as it "hides" the JWT in a secure cookie and enables server-side rendering with authentication. The Next server mediates all auth interactions (login, logout, token usage) and the browser never directly touches the JWT. This approach was demonstrated in a 2024 example where a Next API route set the JWT in a SameSite=Strict, HttpOnly cookie to avoid XSS, and then used Axios interceptors to include that cookie when making requests to the backend [16] [14]. If implementing this, remember to also implement a logout route (that clears the cookie) and perhaps a token refresh mechanism (you could have a `/api/refresh` on Next that calls a refresh endpoint on Rails, then sets a new cookie). Next's middleware can protect pages by checking the cookie. And always ensure your Rails API accepts the token via header or cookie as configured (Devise-JWT expects the `Authorization: Bearer` header by default).

JWT in Nuxt.js (Vue SSR) with Vuex and Middleware

Nuxt.js (especially Nuxt 2) is a framework for Vue.js that supports server-side rendering and has a powerful module ecosystem. Implementing JWT auth in Nuxt shares some similarities with Next.js, but the typical pattern uses the Vuex store and middleware for route guarding. There's also an official **@nuxtjs/auth** module that can greatly simplify the process by providing built-in schemes for JWT/local auth.

Using Vuex for JWT storage: In a Nuxt app, one common approach is to store the JWT in the Vuex store state (e.g. in an `auth` module of the store). When the user logs in, you commit a mutation to save the token (and maybe user info) in the store. To persist the token across refreshes, you have two options: use the browser's storage or cookies. Nuxt code runs both on server (for SSR) and client, so you have to be mindful:

- If you use localStorage to persist the token, you can retrieve it after page reload on the client side. However, on the initial server render, `localStorage` isn't accessible (since that's Node, not a browser). Nuxt's recommended approach in their auth module is to use a **combination**: they store the token in localStorage for persistence and also set a copy in a cookie (not HttpOnly, a regular cookie) so that the Nuxt server can read it on load to hydrate the store [17]. But storing the token in a non-HttpOnly cookie has similar risks to localStorage (the token is still exposed to JS). As mentioned earlier, some developers prefer to avoid localStorage and use cookies only. For example, one guide recommends **disabling localStorage in Nuxt Auth module config and using only cookies with an expiration** (7 days in that case) for the JWT [17] [18]. With that setup, the token lives in a client cookie that the server can parse, and not in localStorage. The cookie could be HttpOnly if you set it from the server side (Nuxt can set a HttpOnly cookie via server middleware or an API, though by default Nuxt Auth's cookie storage is not HttpOnly because it's set on client-side to sync state).

- If not using the Nuxt Auth module, you can manually implement a similar strategy: when the user logs in (perhaps via a form that calls the Rails API), you get the JWT in the client. You then call `this.$cookies.set('token', token)` using a cookie helper (using a library like `cookie-universal-nuxt` which can set cookies on both client and server). And also commit the token to the Vuex store state. The cookie here serves for persistence/SSR, and the Vuex store is the primary source of truth during client runtime.

Login flow in Nuxt (manual): Here's how a typical manual implementation might work: 1. The user submits login credentials (e.g. via a Vue component that dispatches a Vuex action like `store.dispatch('auth/login', { email, password })`). 2. The `login` action in Vuex uses `$axios` (Nuxt's axios module) or `fetch` to POST to the Rails API login endpoint with the credentials. For example: `this.$axios.$post('/api/users/sign_in', { email, password })`. Nuxt's Axios module can be configured with a baseURL pointing to the Rails API. 3. The Rails API (using Devise-JWT or custom auth) responds with a JWT (often in JSON). The Vuex action receives the token. It commits a mutation to save the token (and possibly user info from response) to state: e.g. `commit('SET_TOKEN', token)`. 4. Additionally, the action can use a cookie utility to save the token in a cookie. For instance, using `this.$cookies.set('token', token, { maxAge: 60*60*24*7 })` to set a 7-day cookie [18]. If this code runs on the server during SSR (e.g. if someone triggered login from an SSR context), the `cookie-universal-nuxt` library will correctly set it in the response. If on client, it sets in browser. 5. After login, you might redirect the user to a protected page (Nuxt can do this with `this.$router.push('/dashboard')` or similar).

Middleware and route guarding: Nuxt allows you to define route **middleware** – small functions that run before entering a route. You can create an `auth.js` middleware that checks `store.state.auth.token` (or a getter like `isAuthenticated`) and if not present and the route requires auth, redirect to `/login`. In your protected pages' Nuxt page component, you can specify `middleware: 'auth'` in the page's script. This way, whenever that page is navigated to, Nuxt will run the `auth` middleware. On the server-side render of that page, the middleware runs on server too. If the token is stored in a cookie, the Nuxt server can populate the Vuex store before middleware via either the Nuxt Auth module or using a `nuxtServerInit` action in the store.

- **nuxtServerInit:** In Nuxt 2, if you define an action `nuxtServerInit` in your Vuex store, Nuxt will call it on server-side renders, passing in the request context. You can use this to check for an auth cookie and set the store state. For example:

```
// store/index.js
actions: {
  nuxtServerInit({ commit }, { req }) {
    const token = req.headers.cookie && parseTokenFromCookie(req.headers.cookie);
    if (token) {
      commit('auth/SET_TOKEN', token);
    }
  }
}
```

This way, when a user who is already logged in (token in cookie) hits your app, the server will initialize the store with their token (and you could also fetch their user profile from Rails and commit that). The middleware will then see `store.state.auth.token` is set, and allow the user to proceed to the page.

- **Nuxt Auth Module:** It's worth noting that Nuxt's official auth module (`@nuxtjs/auth` for Nuxt 2) can do a lot of this heavy lifting. For example, it has a **local scheme** for JWT where you configure the login endpoint, user info endpoint, token name, etc. It will automatically handle storing the token in localStorage/cookie, fetch the user on login, and add an Axios interceptor to send the token. However, as one article points out, the Nuxt auth module's local scheme did **not support refresh tokens out of the box** [19], so if you need refresh tokens, you have to extend it. In one guide, the author manually implemented refresh logic on top of Nuxt Auth: after login, they stored the refresh token in a cookie and set up a periodic refresh call, since the library itself didn't handle rotating the tokens [20] [21]. Despite that, the module is very handy for basic JWT auth – it expects the JWT to be returned in the response of the login call (either in the response JSON or a specific header). In fact, when using Devise-JWT with Nuxt Auth, you might need to adjust how Rails responds. One developer noted *"Nuxt's own auth library expects the JWT to be returned in the body of the sign-in request, and also wants to make a separate API request to get user details"*, which led them to override the Devise SessionsController to include the token in the response body and provide a `/users/current` endpoint for user info [22].

Attaching JWT to requests (Nuxt): Whether using Nuxt Auth or a manual approach, you need to ensure that every API request from Nuxt to the Rails API includes the JWT for authentication. If using Axios, the simplest way is to set a default authorization header once the token is available. For example, you can do:

```
this.$axios.setToken(store.state.auth.token, 'Bearer');
```

This will add the header `Authorization: Bearer <token>` to all subsequent Axios requests. If the token is stored in a cookie and you're not manually attaching headers, you must configure Axios to send credentials and have the correct CORS setup on Rails so that the cookie is included. In a Nuxt server-rendered context, if you fetch data during SSR (like in the `asyncData()` of a page), that code runs on the server – it should use the token from the store (which, if `nuxtServerInit` set it from the cookie, is available). The Nuxt Axios module actually automatically sends cookies (if `withCredentials` is true and CORS allows) but it's often easier to just manually inject the header.

Example: Suppose you have a Vuex state `token`. You could watch that state and set Axios header whenever it changes. Or in the login action after committing token, call `this.$axios.setHeader('Authorization', 'Bearer ' + token)` (or use the `setToken` helper). One detailed guide suggests manually setting the Axios auth header after login and also handling refresh token storage manually [23]. Once set, your Rails API (which likely checks `Authorization` header in a before_action via Devise-JWT) will receive the JWT on all calls.

Route Middleware Example: Create a file `middleware/auth.js`:

```
export default function ({ store, redirect }) {
  if (!store.state.auth.token) {
    return redirect('/login');
  }
}
```

Then in any page component that needs protection, add:

```
export default {
  middleware: 'auth'
  // ...other page component code
}
```

Nuxt will run this on every request to that page, either client-side navigation or initial server render. If the token is missing, it redirects to login.

Logout: Implement a logout action that simply clears the Vuex state, and removes the token from cookie/localStorage. If using Devise-JWT with blacklist, you might also call the Rails API's sign_out endpoint to invalidate the token server-side (Devise-JWT can blacklist the JTI). Otherwise, just clearing it on client will effectively log the user out (the token will eventually expire on server, but you've removed it from client so it won't be sent anymore).

Nuxt 3 differences: If using Nuxt 3 (with Vue 3 and optionally Pinia instead of Vuex), the concepts remain similar. You'd use useState or Pinia to store the token, use Nitro server routes or the auth middleware to set cookies, etc. The Nuxt 3 ecosystem is moving away from the old auth module (a Nuxt 3 version is in progress). But the fundamental JWT handling is the same: store token securely, attach to requests, guard routes.

JWT in a Vue.js SPA (Vuex + Vue Router Guards)

For a standalone Vue.js SPA (without Next/Nuxt), the JWT auth pattern is straightforward and quite common. Here, everything happens client-side (no SSR to worry about). The typical setup uses **Vuex (or Pinia)** for state management, and **Vue Router** for navigation guards.

Login process: - The user submits credentials via a login component. That component (or a Vuex action it calls) will send an AJAX request (using `fetch` or `axios`) to the Rails API's login endpoint (e.g. `POST /api/login`). - On success, the Rails API returns the JWT (and possibly user info). The Vue app then saves the token, usually in the Vuex store state (e.g. `store.state.auth.token`). It may also store some user info (like `store.state.auth.user = {...}`) if returned or fetchable via a `/me` endpoint. - The app can also persist the token in localStorage so that the user doesn't get logged out on refresh. For example, after login, do:

```
localStorage.setItem('token', store.state.auth.token);
```

and on app startup (main.js), check if `localStorage.getItem('token')` exists and if so, load it into the store and set the auth header. Alternatively, use a Vuex persistence plugin.

- After storing the token, you typically update axios defaults:

  ```
  axios.defaults.headers.common['Authorization'] = `Bearer ${store.state.auth.token}`;
  ```

  This ensures all subsequent axios requests include the token in the Authorization header. (If using `fetch`, you'd manually include the header each time or wrap fetch calls to do so.)

Route guarding with vue-router: - In your `router.js`, you can define global **navigation guards**. For example:

```
router.beforeEach((to, from, next) => {
  const isLoggedIn = !!store.state.auth.token;
  if (to.meta.requiresAuth && !isLoggedIn) {
    next('/login');
  } else {
    next();
  }
});
```

Here, routes that have `meta: { requiresAuth: true }` will be intercepted: if no token is present, the user is redirected to the login page. This is a UX measure to prevent access to UI that requires login. Remember, this is purely client-side – it doesn't actually stop someone from forging requests to the API; the real enforcement is the Rails API checking the token. But it improves the user flow by not even showing protected components when not logged in.

- You also might have a guard to prevent logged-in users from hitting the login page (redirect them to dashboard if they already have a token).

Using Vuex for auth state: A simple Vuex module might have state `{ token: null, user: null }`, mutations like `SET_TOKEN` and `SET_USER`, and actions like `login({ commit }, creds)` and `logout({ commit })`. The `login` action does the API call and commits the data. The `logout` action could clear localStorage and reset the state (and perhaps inform the server if needed).

Example implementation details: - *Axios interceptor:* Instead of setting the header once, you can set up an interceptor to attach the token from store for every request. E.g.:

```
axios.interceptors.request.use(config => {
  const token = store.state.auth.token;
  if (token) {
    config.headers.Authorization = `Bearer ${token}`;
  }
  return config;
});
```

This way, if the token is updated or if you reinitialize the app with a stored token, all requests will have it.

- *Auto login on refresh:* On page reload, your SPA reboots. If you stored the token in localStorage, your main script can pull it and set the Vuex state before the app mounts. For instance:

```
const token = localStorage.getItem('token');
if(token) {
  store.commit('SET_TOKEN', token);
  axios.defaults.headers.common['Authorization'] = `Bearer ${token}`;
```

```
  // optionally fetch user info with this token to populate store.state.auth.user
}
```

This ensures the user stays logged in across refreshes (until the token expires).

- *Token expiration handling:* With a JWT, if it has an expiry, you might want the client to react if the token is expired (though the server will also reject it). You could decode the JWT on the client to inspect the `exp` claim (JWT payloads are Base64; you can use a library like jwt-decode, or simply attempt to parse at your own risk). If you find it expired, log the user out locally. More often, you'll find out when an API call fails with 401. At that point, you can intercept that response (axios response interceptor) – if you get 401 Unauthorized, you can, for example, dispatch `logout` action and redirect to login. This way if the token expires or is invalid, the app clears it and requires login again.

- *Security:* As discussed, a pure SPA often uses localStorage out of simplicity. Be aware of the XSS risk – ensure you are sanitizing any dynamic HTML and not injecting unsafe content, use CSP headers if possible. Some SPA developers choose to store the token in a cookie even without a backend proxy: you could set a cookie via JS (not HttpOnly). But that gives no real security benefit over localStorage (since JS can still read that cookie). The real secure method would be to have your Rails API set an HttpOnly cookie on login. For example, your Rails login response could include `Set-Cookie: token=...; HttpOnly; Secure; SameSite=Lax`. If the SPA is served on the same top-level domain (or if CORS allows, etc.), the browser would store it. Then you could rely on that cookie instead of localStorage. But then you need to deal with CSRF. Many SPA+API setups instead accept the trade-off of localStorage for simplicity and manage XSS through other means.

Logout and token invalidation: In a JWT setup, logout on the client just means removing the token (and perhaps telling the server to blacklist it). The Rails backend using Devise-JWT can implement **blacklisting**: e.g., when the user hits a logout endpoint, you can blacklist the JWT's JTI (unique identifier) so it can't be used again [24] [25]. If you don't do this, a JWT remains valid until it expires – so a user hitting logout and deleting their token doesn't inform the server. If an attacker had stolen that token, they could still use it until expiry. Blacklisting (or "revocation") is one solution. Another is using very short-lived tokens so that even without logout, a stolen token dies quickly. The downside is then you need refresh tokens or frequent logins. According to a JWT introduction, *"On server side, [if you want to] mark non-expired JWT as invalid on logout… a method used is token blacklisting, which involves maintaining a list of invalidated tokens on the server side"* [26]. Devise-JWT directly supports this (via blacklist or whitelist strategies), or you can implement your own tracking.

To summarize the Vue SPA case: it's simpler since everything is on the client – **store the JWT (often in localStorage), keep it in a central store (Vuex/Pinia), attach it on every API call, and use router guards for UX.** Many tutorials exist for this pattern and it's a proven approach for SPAs. Just remember the security caveats. If higher security is needed, consider moving to a cookie-based approach or a hybrid with an Node.js backend that handles the cookies (like the Next.js approach).

Token Refresh and Renewal Strategies

One big consideration in JWT authentication is how to handle **token expiration** and keep the user "logged in" without forcing them to log in again too frequently. The common solution is to use **refresh tokens**.

Why refresh tokens? If you make your JWT long-lived (say valid for 2 weeks or 2 months), then if it's ever stolen, the attacker has a long window to abuse it. Also, you cannot easily revoke it (unless you keep a blacklist of all issued tokens, which starts to negate the stateless benefit). On the other hand, if you make your JWT short-lived (like 15 minutes), it reduces the risk window and you can enforce that the user gets a new token frequently – but asking the user to log in every 15 minutes would be awful. Refresh tokens solve this: the idea is to issue **two tokens** at login – an **access token** (the JWT) that is short-lived, and a **refresh token** that is long-lived. The refresh token is usually an opaque random string (or it could be a JWT as well) that **is only sent to the server when requesting a new access JWT**, not with every request. The refresh token typically is stored in a very secure way (often HttpOnly cookie) and the server keeps a record of it (in a database or in-memory store) so it can verify its validity. The access JWT is used for regular API calls and expires quickly (e.g. 5-15 minutes). When it expires, the client can use the refresh token to get a new one without bothering the user.

In our Rails + decoupled frontend scenario, implementing refresh tokens would look like: - Rails upon login generates a JWT access token (exp 15 min) and also generates a refresh token (could be a secure random string stored in a `refresh_tokens` DB table associated with the user, or could be another JWT with a longer exp and a special claim). - The Rails API sends both to the client. For example, response could be `{ "token": "<access_jwt>", "refresh_token": "<refresh_token_string>" }`. The access token might also be delivered via header as usual, and refresh token perhaps in body or even as an HttpOnly cookie. - The frontend stores the refresh token in a safe place. **Best practice:** store the refresh token in an HttpOnly cookie (so it's not accessible to JS at all), and store the access token in memory or a non-persistent store. Some implementations store both in cookies (with different names) or both in localStorage (less secure for refresh). But keeping refresh tokens HttpOnly is important because if an XSS steals a refresh token, they can then continually get new access tokens – basically full account takeover until refresh expires. In contrast, if an XSS can only steal the access token (not refresh), they have a limited window until it expires [11].

- When the access JWT is nearing expiration or has expired, the client needs to get a new one using the refresh token. There are two common strategies:
- **Silent refresh (timed):** The client proactively refreshes the token in the background. For example, if the access token is valid for 15 minutes, the client might start a timer to refresh at the 14-minute mark. One Nuxt example did exactly this: they parsed the JWT to get its `exp` time, set a timeout for 75% of its lifespan, and when that timer hit, they called the refresh API to get a new token [21]. This ensures the user's session is continuously refreshed without interrupting them. The refresh API (Rails endpoint) would verify the refresh token (from the cookie or request) and return a new JWT (and possibly a new refresh token with an updated expiry, depending on implementation).

- **Lazy refresh (on-demand):** The client only tries to refresh when it gets a 401 response indicating the access token is expired. The flow is: an API call fails with "token expired", the frontend catches that error, then calls the refresh token endpoint. If refresh succeeds (HTTP 200 + new token), it updates the stored token and retries the original request. If the refresh token is also expired or invalid, then it means the user must log in again (perhaps redirect to login). This strategy is a bit simpler in that you don't have to manage timers, but it can introduce a slight delay on the first request after expiration (the user experiences one failing call that triggers a refresh behind the scenes).

- **Rails implementation for refresh:** Rails (Devise-JWT) out of the box does **not** have a refresh token mechanism (Devise-JWT focuses on access tokens only – the gem author notes that if you need never-expiring sessions, you should implement refresh tokens or use OAuth2 [27]). So you would have to create a custom controller action for refresh. For instance, you might have `POST`

`/api/token/refresh` that expects a refresh token (maybe in the request body or as a cookie). You'd look up that refresh token in the DB, ensure it's valid (and maybe not expired or not used), and if valid, issue a new JWT (and possibly rotate the refresh token). There are gems that can help with this, like `jwt_sessions` gem, which provides a refresh token workflow in Rails. But if using Devise, you might integrate refresh manually or using another gem.

- **Refresh token security:** Make sure refresh tokens are **one-time use or short-lived** if possible. A common practice is refresh token rotation – every time you use a refresh token, the server issues a new refresh token along with the new access token, and invalidates the old refresh token. This prevents replay (if someone intercepted a refresh token, and the legit user already used it, it's no longer valid). Rotation can be implemented by storing a `refresh_token` record with an `used` flag or expiry.

- **Client-side handling:** In frameworks:

- Next.js – you could implement refresh by adding a Next API route `/api/refresh` that reads the refresh token cookie and calls Rails for a new token, then sets a new cookie. Or use an http-only cookie so that the browser sends the refresh token to Rails directly on a `fetch('/api/refresh_token', { credentials: 'include' })` call.
- Nuxt/Vue – you might store refresh token in a cookie or localStorage. One guide using Nuxt stored the refresh token in a cookie (because Nuxt auth module didn't do it automatically) and then on app mount set an `setInterval` to periodically refresh using that token [21].

Avoiding refresh tokens (alternate method): Another approach some use is **sliding sessions** – basically each time the user makes a request, issue a new JWT with a fresh expiration (or extend the expiration). This can be done by the server: e.g., Rails could embed a new JWT in each response's header if it's nearing expiry. Devise-JWT doesn't do this by default, but it's conceptually possible. However, this approach is less common in REST APIs and can lead to a lot of token churn. It's simpler to separate the concerns with refresh tokens.

Summary of refresh: Short-lived access tokens and refresh tokens are considered best practice for a robust JWT auth system [28]. *Access tokens are usually short-lived JWTs, included in every request, and refresh tokens are long-lived (often stored in a database) used to get new access tokens when the old one expires* [28]. This gives the best of both worlds: the access token is only valid for a short time (reducing impact of leakage), and the refresh token can be securely protected and is only exchanged occasionally. Just don't forget to implement proper refresh logic on the client (timers or 401-intercepts) and refresh token revocation on logout or suspicious activity.

Security Considerations: Mitigating XSS, CSRF, and Other Attacks

When using JWTs in a decoupled app, security is paramount. We've touched on many of these points, but let's recap how to mitigate common attacks:

- **Cross-Site Scripting (XSS):** As discussed, XSS can compromise tokens if you store them in browser-accessible storage. Mitigations include using HttpOnly cookies (so even if an attacker injects JS, `document.cookie` won't reveal the JWT) [6], and generally writing secure frontend code. Use frameworks (React, Vue) which automatically escape content, avoid `v-html` or `dangerouslySetInnerHTML` with user input, and consider a Content Security Policy (CSP) to restrict script sources. If you use third-party libraries, keep them updated to avoid known XSS vulnerabilities. Essentially, assume that if an XSS vulnerability exists, *any secret in JavaScript can be*

stolen. So minimize those secrets (maybe store only a short-lived token in JS, not a refresh token). Storing the token in a cookie means an XSS attack cannot directly read it [6], but note: an XSS attack could still make AJAX requests on behalf of the user – for instance, the malicious script could call your API endpoints using the user's session (cookies will be sent, or it could read token from store if not HttpOnly). However, that kind of attack is more limited in that the script cannot exfiltrate the token for later use (if cookie is HttpOnly). It can only act while it's on the page. **Bottom line:** Protect against XSS through coding practices and framework features, and lean on HttpOnly cookies to guard tokens [29].

- **Cross-Site Request Forgery (CSRF):** If you use cookies for JWT, by default browsers will include those cookies on requests to your API domain, even if the request is triggered from a malicious site (unless SameSite prevents it). For example, if your frontend is at `app.example.com` and backend at `api.example.com`, and you store JWT in a cookie for `.example.com`, then a malicious page could script an AJAX call to `api.example.com` and the browser would attach the JWT cookie. This would be blocked by your CORS (because the evil site isn't allowed by CORS in most cases), but if you had misconfigured CORS it could be an issue. If your frontend and backend share the *same* domain (which is often the case if you serve the frontend and API under one domain), then CSRF is a real concern – an attacker could make the user's browser send a request to your API with their cookie without using XHR (like a simple `<img src="https://yourapi.com/delete_account">` or a form submission). To mitigate CSRF in a JWT scenario:

- Use the `SameSite` cookie attribute. `SameSite=Lax` or `Strict` will significantly reduce CSRF risk by not sending the cookie on cross-site contexts (Lax will not send on cross-site *submissions* except top-level navigations like following a link, Strict will not send at all cross-site). In the earlier Next.js example, they set SameSite `strict` on the JWT cookie [30] specifically to avoid the cookie being sent on any request not coming from your site.
- Implement CSRF tokens for state-changing requests. Since our API is stateless, traditional CSRF tokens (like Rails' protect_from_forgery) might not be automatic. But you can roll your own: e.g., when the user logs in, generate a random CSRF token and store it in an HttpOnly cookie *and* send it in the response body. The client then stores the CSRF token value in a JS variable (or localStorage). For subsequent requests (especially POST/PUT/DELETE), require a header like `X-CSRF-Token` with that value. The browser on its own can't read the HttpOnly cookie to get the token, and an attacker on another domain can't read your response body due to SOP. Only your JS knows the token, so only it can set the header. Rails could verify this header against the cookie. This double-submit technique ensures that even if a browser automatically includes the JWT cookie, the request will be missing the CSRF header if it didn't originate from your JS. Many modern libraries forego this if SameSite is enough, but it's an extra layer.
- Another approach: for APIs, simply require all non-GET requests to have a custom header (like `X-Requested-With: XMLHttpRequest` or any custom header). Normal <form> or <img> requests from other sites cannot set custom headers, so such requests would be rejected. This is a lightweight CSRF protection (not foolproof if attacker uses XHR with CORS and you allow that, but generally you won't).

- If using Rails with Devise-JWT and cookies, consider enabling Rails' built-in CSRF protection if the frontend is served by Rails. But in an API-only app, you have to handle it as above.

- **Token Theft & Replay:** If an attacker somehow steals a JWT (say via XSS or network if not HTTPS), they can use it until it expires. To reduce impact: keep token lifespan short, and ideally bind the token to a single user session. One advanced technique is to include a *nonce* or *fingerprint* in the JWT – e.g., a claim that is tied to the device or session (like a hash of the user's User-Agent or an identifier stored in an HttpOnly cookie). That way, even if stolen, the token

might not be usable from a different device or without the accompanying cookie. This is more advanced and not always used. Many rely on refresh token rotation and blacklists to invalidate tokens as needed.

- **Logout and invalidation:** As mentioned, JWTs by design don't have a built-in logout mechanism. If a user logs out, ideally you want the token to be immediately invalid. Options:

- **Blacklist on logout:** Maintain a server-side list (in DB or cache) of token IDs (JWT `jti` claim or the full token string hashed) that are no longer valid, and check against it on each request (Devise-JWT can do this via a `JwtBlacklist` model [24] [25]). This brings back some statefulness but only for tokens that are revoked early.
- **Short expiry:** Accept that logout only clears the client, and a stolen token could be used until expiry. Mitigate by short life.

- **Reissue/rotate on frequent intervals:** e.g., every 5 minutes if user is still active, issue a new token (this is effectively a silent refresh strategy).

- **Do not trust the client:** Always enforce authorization on the server. The frontend might hide or show certain UI based on the token or user role, but never rely on that for security. For example, a Vue router guard might prevent a user from seeing the "Admin" page if they're not admin – but an attacker could manually call the admin API endpoint. The Rails API *must* validate the JWT and also check any authorization claims (e.g., is user an admin?) for protected resources. This seems obvious, but it's a good reminder that client-side checks are convenience/UX only.

- **Libraries and framework security:** Use battle-tested libraries for JWT handling. On Rails, **Devise-JWT** is a solid choice that takes care of a lot of the token security (it adds expiration, audience, and other claims, and integrates with Warden). It follows secure defaults (for instance, it will not accept tokens after password change by default, etc.). The Devise-JWT README emphasizes following the "secure by default" principle and notes that if you need never-expiring tokens you should consider OAuth2 with refresh tokens [27] (highlighting that their solution forces an exp claim). On the client side, if using an auth module (like Nuxt Auth or NextAuth), read their security guides – for instance, NextAuth's JWT are encrypted by default when stored, etc.

By covering these bases – proper token storage, short lifetimes with refresh strategy, XSS/CSRF mitigations, and robust server-side checks – you can build a secure JWT-based authentication for your decoupled app.

Ruby on Rails API Implementation (Devise-JWT and Alternatives)

On the Rails side, implementing JWT auth usually involves either an authentication library (Devise with an extension) or writing custom token logic. The most common approach in the Rails community is to use **Devise** (a popular auth solution) with the **devise-jwt** gem, which integrates JWT into Devise's sign_in/sign_out workflow.

Devise-JWT setup: To use Devise-JWT, you add it to your Gemfile along with Devise, run Devise's installation (creating a User model with Devise modules), and include the JWT module. For example: - Add to Gemfile: `gem 'devise'`, `gem 'devise-jwt'`, then `bundle install`. - Run `rails g devise:install` and `rails g devise User` to set up Devise normally (if not already). - In the Devise initializer (`config/initializers/devise.rb`), configure the JWT settings. You need to set

the `jwt_secret_key` (often from Rails credentials or an environment variable) and specify which requests dispatch JWTs and which revoke them. For instance:

```
config.jwt do |jwt|
  jwt.secret = Rails.application.credentials.devise_jwt_secret_key
  jwt.dispatch_requests = [
    ['POST', %r{^/api/users/sign_in$}],
  ]
  jwt.revocation_requests = [
    ['DELETE', %r{^/api/users/sign_out$}],
  ]
  jwt.expiration_time = 30.minutes.to_i
end
```

This tells Devise to issue a JWT on sign_in and revoke on sign_out (more on revocation in a second). - In your User model (or whichever model uses Devise), add the JWT module: e.g.

```
class User < ApplicationRecord
  devise :database_authenticatable, :registerable, # ... other modules
         :jwt_authenticatable, jwt_revocation_strategy: JwtBlacklist
end
```

Here, `JwtBlacklist` is a model or strategy for revocation. Devise-JWT supports two strategies: **Blacklist** (you maintain a table of JWT IDs that are revoked) or **Whitelist** (you maintain a table of only allowed tokens, which is less common). The Medium article example created a `JwtBlacklist` model with `jti:string` and `exp:datetime` to store revoked tokens [24].

- After this setup, Devise's endpoints (/users/sign_in, /users/sign_out, etc., under your API namespace) will automatically handle JWT. On successful sign_in (HTTP 200 OK), the response will include an `Authorization` header: `Authorization: Bearer <jwt>` [2]. On sign_out (assuming you send the correct Authorization header of the token to revoke), Devise-JWT will add that token's `jti` to the blacklist (making it invalid for future use).

- The JWT payload by default includes the user's `sub` (subject, usually user id), `exp` (expiration), `jti` (JWT ID, a unique random identifier per token), and `aud` (optional audience if configured). The secret key is used to sign it (usually HS256). The Rails server, via Warden/Devise, will on each authenticated route, look at the `Authorization: Bearer ...` header, decode the JWT, and if valid, set `current_user`. From the developer's perspective, you can then use `before_action :authenticate_user!` in your controllers to protect them – Devise's JWT strategy will ensure that sets `current_user` or returns 401 if not.

Customizing Devise responses for frontend: By default, Devise's JSON response for sign_in might be empty or minimal (since it traditionally sets a cookie, but in API mode it doesn't). The token is by default only in the header. Many developers choose to override the Devise SessionsController in the API to render a JSON body with the token as well (especially because the client code might find it easier to get it from response body than header). For example, in the earlier example, they overrode the sessions#create to include `render json: { token: token, user: user }` [31] [32]. Another thing often done is providing an endpoint to get current user info (since after login the client may want

to know user's email, etc. without decoding the JWT on client). They added an endpoint `/api/users/current` which returns user info if the JWT is valid [32]. This is handy for the frontend to verify "is my token still valid and who am I".

Devise-JWT and refresh tokens: As mentioned, devise-jwt doesn't support refresh tokens out-of-the-box. It relies on the token's expiration and (optionally) blacklist for security. The documentation even suggests using OAuth2 for long-lived sessions [27]. If you want refresh functionality, you have to implement it separately. One way is to create a new model (say RefreshToken with fields user_id, token, expires_at) and a controller#action (e.g. `refresh#create`) that checks if the provided refresh token is valid, and if so, issue a new JWT (using Devise JWT's token generation maybe) and a new refresh token. Some community solutions like the **Knock gem** or **JWT::Sessions (JWTSessions)** gem support refresh tokens inherently: - **Knock gem:** An older gem that provides JWT auth without Devise. You'd include Knock, configure it with your JWT secret and token lifetime, and it gives you a controller mixin to authenticate JWTs. It doesn't integrate with Devise, so you either use it standalone or in addition. Some people used Knock for APIs while still using Devise for traditional logins. Knock's usage has faded in favor of Devise-JWT, but it's an option (especially if you want a lightweight solution and don't need Devise's full features). It basically requires you to write an auth controller that calls `Knock::AuthToken.new(payload: { sub: user.id }).token` to create tokens, etc. - **devise_token_auth gem:** This gem is different – it's not JWT, but it is a token-based auth for Rails specifically designed for SPAs (used often with React Native, etc.). It issues a client token and access token pair and expects the client to manage them. However, since JWT has become more popular, devise_token_auth is less common nowadays, but legacy projects might use it.

- **JWTSession gem:** This gem by @tuwukee provides a full JWT auth solution with access and refresh tokens, sliding expiration, CSRF protection (via a CSRF token in headers), etc. It can work in Rails API without Devise. If one didn't want to use Devise, this is a good alternative to implement robust JWT with refresh.

Given the question specifically mentions Devise-JWT, we'll focus on that as the typical solution for a Rails API. **Using Devise-JWT, the Rails API essentially treats JWT like a replacement for cookies** – it will automatically add a JWT on login and look for it on protected routes. In fact, the Devise-JWT docs emphasize *"This gem is just a replacement for cookies when these can't be used. As with cookies, a devise-jwt token will have an expiration time. If you need your users to never sign out, you will be better off with a solution using refresh tokens."* [27]. This means that if, for example, you were building a traditional Rails app you'd use cookie sessions, but for an API you use JWT as the session – the concept of expiration and needing to re-authenticate (or use refresh) remains.

Rails controllers and JWT: Once set up, using JWT is transparent in Rails: you can use `before_action :authenticate_user!` (Devise will check for the token), and then use `current_user` in the controller as usual. If `authenticate_user!` fails (no valid JWT), Devise will render an unauthorized response (401) with maybe an empty body. You might want to customize the response for unauthorized (to return JSON error message). This can be done with a failure app or rescue in ApplicationController.

Sending the JWT from client: Ensure your frontend actually sends the token in the correct way the Rails expects. By default, Devise-JWT expects it in the `Authorization` header, Bearer format. So your Axios/fetch calls should include that. If you chose to store the token in an HttpOnly cookie on the frontend domain, Rails wouldn't automatically know about it (since it's not reading cookies, unless the cookie is on the Rails domain and you custom parse it). The simplest path is always sending `Authorization: Bearer <token>` header. In some cases, developers tweak Rails to also accept the

token via a cookie (for example, using Rack middleware to check cookies for a token). But that's custom – standard is Authorization header.

Devise-JWT and CSRF: If you are purely using Authorization headers, CSRF protection in Rails can typically be disabled for API (Rails API mode doesn't have CSRF enabled by default because it expects you're not using cookies for session). So that's fine. If you mix cookies into it, be careful. But generally, with JWT auth, Rails API should have `protect_from_forgery` disabled or not applicable, and you handle auth via token.

Testing the setup: Using a tool like cURL or a REST client: 1. Create a user (sign up via API or rails console). 2. `POST /api/users/sign_in` with `{"email":"...","password":"..."}` and you should get a `Authorization: Bearer <token>` in response [2]. 3. Use that token in `Authorization` header of another request (like `GET /api/some_protected`) – it should succeed. 4. If you configured revocation, try `DELETE /api/users/sign_out` with the header – it should blacklist the token (Devise-JWT will intercept that request, add the jti to blacklist). Subsequent use of that token should then be rejected (even if not expired). Blacklisting does require a lookup on each auth request (to check if jti is blacklisted), which is a slight performance hit, but likely negligible unless huge traffic.

Alternative: Custom JWT without Devise: Some developers opt to not use Devise at all, and just write a simple token controller. For instance: - They might have an `AuthController` with a `login` action that checks User.find_by(email).authenticate(password). If success, they use the `jwt` gem to encode a payload `{ user_id: user.id, exp: 24.hours.from_now.to_i }` with a secret. Then render json: `{ token: token }`. - For each request, they use a middleware or a `before_action` in ApplicationController to look at `Authorization` header, decode the token (using same secret), and if valid, set `Current.user`. - This is perfectly doable and gives full control, but you have to handle things like expiration, maybe token invalidation on logout or password change, etc. Devise-JWT and similar libraries basically encapsulate this logic and integrate with the rest of Devise's features (like password recovery, etc., which you might still want).

Relevant libraries summary (for this stack): - **Devise + Devise-JWT (Rails)** – as discussed, the go-to solution for JWT auth in Rails API [33]. - **Devise Token Auth (Rails)** – alternative token-based auth (not JWT) often used for mobile clients. - **Warden JWT Auth (Rails)** – the underlying mechanism used by Devise-JWT (Devise-JWT is essentially a wrapper around Warden JWT). - **Knock (Rails)** – simpler JWT auth gem if not using Devise. - **jwt_sessions (Rails)** – gem providing JWT with access/refresh and CSRF protection in Rails. - **NextAuth (Next.js)** – authentication library for Next.js (not specifically tailored to Rails APIs, but can be configured with Credentials). - **@nuxtjs/auth (Nuxt 2)** – authentication module for Nuxt which supports JWT via local or cookie schemes. - **Pinia Authentication templates (Vue 3)** or **vuex-auth** – there are community examples of setting up Vuex/Pinia stores for JWT. - **Axios interceptors** – while not a library, using Axios's interceptor feature is a common technique to attach tokens and handle 401 globally. - **jwt-decode (JS)** – a small library to decode JWTs on the client (useful for extracting expiration or user info without verifying signature, for display/logging purposes). - **JS Cookie** – if managing cookies in the browser (non-HttpOnly), this library helps set/get cookies easily in JavaScript.

Finally, always keep an eye on updates: the ecosystem evolves, e.g. Nuxt 3 might have a different recommended approach (there's a Nuxt 3 auth module in the works), and security recommendations can change. But as of now, the above represents a robust approach to using JWTs in a decoupled Next/Nuxt/Vue frontend with a Rails API backend: a stateless, JSON-based authentication mechanism that,

when implemented with best practices (secure storage, refresh tokens, server verification), provides a smooth and secure login experience for users.

Sources:

- High-level JWT stateless auth concept [1]
- JWT auth flow steps and example Authorization header usage [3] [34]
- Rationale for HttpOnly cookie storage vs localStorage (XSS concerns) [7] [6]
- Comparison of storage options and security implications (XSS/CSRF) [10] [11]
- Nuxt.js JWT implementation details (disabling localStorage in favor of cookies) [17] [18]
- Nuxt Auth module JWT expectations [22]
- Next.js cookie-based auth example (setting HttpOnly cookie in API route) [14]
- Devise-JWT usage in Rails (setup and token dispatch) [2] [24]
- Devise-JWT philosophy (stateless replacement for cookie session) [27]
- JWT refresh token concept (access vs refresh token definitions) [28]
- Token theft and the importance of secure storage (do not store sensitive info in JWT, use HttpOnly) [13] [35]
- Token revocation via blacklisting on logout [26] .

---

[1] **Rails 7: API-only app with Devise and JWT for authentication | by Michael Epelboim | Medium**
https://sdrmike.medium.com/rails-7-api-only-app-with-devise-and-jwt-for-authentication-1397211fb97c

[2] [22] [24] [25] [31] [32] [33] **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

[3] [13] [26] [34] [35] **JWT explained in 4 minutes (With Visuals) - DEV Community**
https://dev.to/jaypmedia/jwt-explained-in-4-minutes-with-visuals-g3n

[4] [5] [6] [10] [11] [12] [28] [29] **LocalStorage vs Cookies: All You Need To Know About Storing JWT Tokens Securely in The Front-End - DEV Community**
https://dev.to/cotter/localstorage-vs-cookies-all-you-need-to-know-about-storing-jwt-tokens-securely-in-the-front-end-15id

[7] [14] [16] [30] **NextJS Authentication Flow — Store JWT In Cookie | by Andy Chow | JavaScript in Plain English**
https://javascript.plainenglish.io/nextjs-authentication-flow-store-jwt-in-cookie-fa6e6c8c0dca

[8] [9] **Token Storage**
https://auth0.com/docs/secure/security-guidance/data-security/token-storage

[15] **Authentication with decoupled Rails API? - nextjs - Reddit**
https://www.reddit.com/r/nextjs/comments/nvv98q/authentication_with_decoupled_rails_api/

[17] [18] [19] [20] [21] [23] **How to make nuxt auth working with JWT - a definitive guide - DEV Community**
https://dev.to/mrnaif2018/how-to-make-nuxt-auth-working-with-jwt-a-definitive-guide-9he

[27] **Devise-JWT Alternatives - Ruby Authentication and OAuth | LibHunt**
https://ruby.libhunt.com/devise-jwt-alternatives