ChatGPT

Micro-Frontends vs. Iframes for App Integration: A Quick Evaluation

What Are Micro-Frontends?

Micro-frontends are an architectural approach where a web app is split into smaller, independently deployable frontend applications that together form a single UI [1]. In practice, this means different teams or modules can build and deploy their part of the UI (e.g. one in React, another in Angular) without a tightly coupled monolithic codebase. The key idea is that these micro-apps are composed in the browser to appear as one seamless application to users. This approach can improve modularity and enable separate development streams, much like microservices do on the backend.

Communication in Micro-Frontend Architectures

Because each micro-frontend is its own application, a common question is how they communicate or share data. In general, micro-frontends **minimize direct communication** to avoid tight coupling [2]. When communication is needed, typical strategies include:

- **Custom Events:** Micro-apps can publish and listen for DOM events (or a pub-sub system) as an indirect communication channel [3]. For example, a "userLoggedIn" event can be dispatched by a login micro-frontend and caught by others. This keeps modules loosely coupled, though it requires clear conventions on event names and payloads.
- **Props/Callbacks via a Container:** If using a micro-frontend framework or library, the shell (container app) can initialize micro-apps with certain data or callbacks. This is akin to React props – e.g. passing a function from the container to a micro-app to call when it needs something. It makes the interface between parts more explicit.
- **Global State or Shared Store:** In some cases, a global state management (like a Redux store) is shared. However, this is used carefully since it introduces coupling; often each micro-frontend keeps its own state store to remain self-contained [4]. Shared state is only used for truly global concerns (e.g. user session info), and even then events or APIs are preferred over directly mutating a common store.

In summary, micro-frontends tend to communicate via browser APIs or through the framework's mechanisms rather than direct function calls across boundaries, keeping each piece as isolated as possible.

Why Iframes Cause Issues (CORS, OAuth2, X-Frame-Options)

Integrating a second app via an `<iframe>` provides strong isolation but also brings specific challenges. By design, an iframe keeps the embedded app separate from the parent page, which leads to several integration hurdles:

- **Cross-Origin Resource Sharing (CORS):** If the main app and the iframe content are on different domains (or ports), the browser's same-origin policy restricts interactions. Any API calls from the iframe to the main app's domain (or vice versa) are treated as cross-origin requests and must be permitted by CORS headers. Sharing data or calling APIs across the boundary requires careful setup (or hacks like `postMessage`). One developer notes that with iframes "things like CORS and communication [between frames]" become hard to manage [5]. In contrast, micro-frontend architectures typically load all micro-apps into the same page context, often under the same origin, which **reduces or avoids CORS issues**. For example, using a build tool like Module Federation can fetch a micro-app's code from another origin but execute it in the host page, avoiding cross-domain scripting at runtime [6]. Additionally, many teams use a reverse proxy so that each micro-frontend is served under a unified domain path (e.g. `mainapp.com/app1` and `mainapp.com/app2`), eliminating cross-origin calls from the browser's perspective [7]. (You may still need to configure CORS on static asset requests, but this is easier than handling persistent cross-window communication in iframes.)

- **OAuth2 Authentication Flows:** OAuth2 (and OpenID Connect) often involves redirects to an identity provider (IdP) or third-party login page. Many IdPs set the header **X-Frame-Options** to `DENY` or `SAMEORIGIN` on their login pages, which prevents them from being displayed inside an iframe on a different site [8] [9]. This means an OAuth login popup or redirect will simply not show or will be blocked if your second app is running inside an iframe. A Stack Overflow discussion concluded that there is "no good solution" to fully handle OAuth flows inside a locked-down iframe, aside from breaking out of the frame (opening a new window for login) [10]. With micro-frontends, this problem is largely avoided. Since micro-frontends are part of the host page, they can trigger OAuth flows in the top window as a normal redirect or popup, which IdPs allow. All micro-frontends can share the authenticated session of the user (e.g. via a shared cookie or token storage on the same domain) once the user logs in. In short, a micro-frontend architecture enables a **single sign-on** experience: the user logs in once, and all parts of the application (each micro-app) recognize the session. There's no need to embed a login flow in a hidden iframe because the micro-apps run as first-party components of the site.

- **X-Frame-Options Restrictions:** Aside from OAuth providers, your own integrated app might send X-Frame-Options headers (for security against clickjacking) that prevent it from being framed in another application. If the second app was not originally intended to be embedded, it might default to `SAMEORIGIN` policy, causing the exact "refused to display in a frame" errors. Micro-frontends, by doing away with iframes, inherently bypass any X-Frame-Options issues. The content is not being embedded as an external page; it's being assembled as part of the single-page application. This means you don't have to relax security headers to integrate the apps. The micro-frontend approach **sidesteps frame restrictions** by design – there is no iframe container, so X-Frame-Options never comes into play for your internal micro-apps.

How Micro-Frontends Mitigate These Issues

Adopting micro-frontends can solve or reduce the above problems through a combination of architectural choices and tools:

- **Single DOM/Window Context:** All micro-frontend parts run in the same page context, so they can interact like normal JavaScript modules. This avoids the sandbox of an iframe and its attendant cross-window communication headaches. For instance, sharing a user token from one micro-app to another can be as simple as calling a function or using a mutual storage (provided you've designed a way to do so), rather than messaging between frame boundaries. As a result, issues with CORS and third-party cookies are minimized because everything is happening in a first-party context (especially if served under one domain).

- **Module Federation and Similar Tools:** Modern micro-frontend implementations often use tools like **Webpack Module Federation** (or analogues in other build systems) to load remote micro-apps at runtime. This allows the micro-app's code to execute inside the main application's JavaScript context. The benefit is a **deeper integration**: it "avoids iframe-specific challenges like … cross-origin communication" by letting the micro-frontend code call the same APIs and use the same global objects as the host app [6]. In practice, the host app might pull in a remote module (say, the entire second app's bundle) with CORS-enabled script fetch, then bootstrap it as part of the page. From that point on, that micro-app behaves as if its code were part of the host application, eliminating frame barriers.

- **Custom Events and Global Pub-Sub:** As mentioned, micro-frontends commonly use custom browser events to communicate [3]. This is a straightforward way to have, for example, the main app notify all micro-apps of a theme change or user action. Unlike with iframes, you don't need `postMessage` to cross a window boundary – events and JavaScript objects are all in one page. This drastically simplifies communication logic (and removes the need for CORS entirely in most intra-app messaging, since no HTTP requests are involved in local messaging).

- **Unified Routing and Proxying:** A common strategy is to use a **reverse proxy or path-based routing** on the web server or CDN level to serve all micro-frontends under a single origin [7]. For example, the main app could live at `example.com`, and the second app could be deployed at `example.com/second-app` (even if internally it's hosted elsewhere). To the browser, everything is coming from `example.com`, so embedding and resource sharing are seamless. This setup means you won't hit CORS or X-Frame-Options issues because from the client perspective there's no cross-origin at all. Each micro-frontend can still be developed and deployed independently behind the scenes, but the user gets one unified domain.

- **Shared Authentication (SSO):** In micro-frontend architectures, authentication is often centralized. You might have a dedicated auth micro-frontend or a shared auth service, but crucially, once a user logs in, their session/token is stored in a way all micro-apps can access (for example, an HTTP-only cookie on the common domain or a token in `localStorage`). This means subsequent micro-frontends won't need a separate OAuth flow; they'll detect the existing login. If using an OAuth2/OIDC provider, you can perform the login redirect in the main app (or a specific micro-frontend responsible for login) and then distribute the credentials. Because there is no iframe involved, the

login page won't be blocked. In scenarios where different microfrontends are on different subdomains, a common technique is to use a **parent domain cookie** for the OAuth session (e.g. set cookie on `.example.com` so that `app1.example.com` and `app2.example.com` both see it). This avoids the third-party cookie problems that iframes would encounter and ensures a one-time sign-in for the user across the micro-apps.

- **Web Components or JavaScript APIs:** Some micro-frontend frameworks leverage Web Components or similar standards to encapsulate micro-apps. For instance, a micro-frontend could be delivered as a custom HTML element (`<my-micro-app></my-micro-app>`). These still run in the main DOM and thus bypass iframe restrictions, while providing style and script isolation. Communication can be done via custom element attributes, DOM events, or a shared global. The key point is that these standards-based approaches still keep everything on the client side in one origin, thereby reducing CORS and X-Frame issues.

Conclusion: Are Micro-Frontends a Suitable Solution?

Micro-frontends are likely to solve the specific integration issues you're encountering with iframes. They provide a way to integrate a second app without the hard separation that an iframe imposes, thereby avoiding the **X-Frame-Options login blocker** and making CORS management more straightforward (often unnecessary) at the UI level. By assembling the apps into a single frontend application, you can handle OAuth2 flows in a unified way and share authentication state across modules more easily. In short, the problems with "app-in-an-iframe" integration – cross-origin script limits, clunky communication channels, and framing restrictions – are largely eliminated when you switch to a micro-frontend architecture.

However, it's worth noting that micro-frontends come with their own complexity and overhead. Adopting this approach is a significant architectural decision; it introduces considerations around build processes, deployment coordination, and performance (loading multiple micro-apps). In scenarios with only a couple of frontend modules and a single team, some engineers caution that a full micro-frontend architecture might be overkill [11] [12]. That said, given your experience with React and Angular, you would be leveraging known tools (like module federation or single-spa) to implement it. With proper planning, those tools can **overcome the integration pains** you described and allow the two apps to coexist as one product without the iframe drawbacks.

Bottom line: Micro-frontends can be a suitable solution to your CORS, OAuth2, and X-Frame-Options problems. They offer a more elegant integration path than iframes by avoiding cross-origin iframes entirely. If you need tight integration and a smoother user experience between your main app and the new module, micro-frontends are worth considering. Just weigh the added complexity – if those issues are critical blockers right now, the architectural shift can be justified. Overall, you can expect significantly fewer cross-domain headaches with a micro-frontend approach than with an iframe-based embedding [5] [6], so it is likely the right direction to resolve the challenges you're facing.

Sources:

- Fowler, M. *Micro Frontends* – Definition of micro-frontend architecture [1]
- Stack Overflow – Discussion of iframe drawbacks (CORS, communication, cookie sharing) [13] [5]

- Tyagi, U. – *Module Federation vs. Iframes* (Medium) – Notes that module federation avoids iframe cross-origin issues [6]
- Stack Overflow – *OAuth in iframe issue* – OAuth2 login blocked by X-Frame-Options, no good workaround inside iframe [9] [10]
- Klee, E. – *Sharing Authentication in Micro-Frontends* (Dev.to) – On using single domain (reverse proxy) to avoid cross-origin in micro-frontends [7]
- Fowler, M. – *Micro Frontends* – On micro-frontend communication via events and avoiding tight coupling [3]

---

[1] [2] [3] [4] Micro Frontends
https://martinfowler.com/articles/micro-frontends.html

[5] [11] [13] html - Micro frontend architecture advice - Stack Overflow
https://stackoverflow.com/questions/47922293/micro-frontend-architecture-advice

[6] Micro Frontend Architecture (Iframes x Module Federation) | by Vinicius Marson | Medium
https://medium.com/@viniciusmarson/micro-frontend-architecture-iframes-x-module-federation-1782026d95c6

[7] How do you share authentication in micro-frontends - DEV Community
https://dev.to/kleeut/how-do-you-share-authentication-in-micro-frontends-5glc

[8] X-Frame-Options - HTTP | MDN
https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/X-Frame-Options

[9] [10] javascript - Refused to display...in a frame because it set 'X-Frame-Options' to 'same origin' in React & oAuth - Stack Overflow
https://stackoverflow.com/questions/63637153/refused-to-display-in-a-frame-because-it-set-x-frame-options-to-same-origin

[12] Are microfrontend a viable architecture for real world apps? : r/react
https://www.reddit.com/r/react/comments/1iok5n1/are_microfrontend_a_viable_architecture_for_real/