**Micro-Frontend Architecture & Caching Strategies**

Micro-frontends allow each app to deploy independently. But caching becomes tricky because:

✔ Different micro-apps may share **React, libraries, styles**  
✔ You want faster load time without breaking compatibility  
✔ Updates in one app should not break others

So we apply **smart caching strategies** for:

1️⃣ App shell (root-config)  
2️⃣ Remote MFEs (federated apps)  
3️⃣ Shared dependencies  
4️⃣ API/data layer caching

Root UI (Single-SPA / Custom Router)

├── MFE-1: Dashboard

├── MFE-2: Admin Panel

├── MFE-3: Notifications

└── Shared Libraries: React, utils, styles

**🚀 Caching Strategies for Micro-Frontends**

**✅ 1️⃣ Module Federation + Runtime Versioning**

Webpack Module Federation helps share libraries like React.

✔ Remote Entry files can be aggressively cached  
✔ But update detection needs **runtime versioning**

**Best practice**

remoteEntry.[contenthash].js ✅

remoteEntry.js ❌ (bad cache)

Upload new name → browser fetches fresh file automatically.

**✅ Shared Library Caching**

React, React-DOM **should be cached** globally using:

* singleton: true
* eager: false

shared: {

react: { singleton: true, requiredVersion: "18.2.0" },

"react-dom": { singleton: true }

}

📌 Reduces duplicate downloads  
📌 Ensures same version in all MFEs

**✅ 4️⃣ LocalStorage Fallback for Remote Apps**

If a remote MFE fails to load (network down → app breaks)

✅ Load the last working version from LocalStorage:

Steps:

1. Save remoteEntry file in browser storage
2. If network fails → load cached remote

This ensures **offline resilience + no blank screen** 💪

**API Response Caching**

Use:

* HTTP cache headers (ETag, max-age)
* IndexedDB for large data
* Stale-while-revalidate logic

fetch('/api/products', {

cache: 'force-cache'

});

**🧠 Version Mismatch Problem & Fix**

⚠️ If one MFE upgrades React → others may fail  
✔ Solution: **Central shared dependency ownership**

* Root-config owns library versions
* Remotes consume from root scope

**📌 Recommended Caching Setup Table**

|  |  |  |
| --- | --- | --- |
| Layer | Strategy | Benefit |
| RemoteEntry.js | hashed filename + SW cache | Faster boot, auto refresh |
| Shared libs | Module Federation scope cache | No duplication |
| Page shell | Hard cache + long max-age | Super fast load |
| Data/API | IndexedDB + revalidate | Offline + reduced API load |
| UI Fragments | Lazy load + chunk split | Faster initial page |

**Final Summary**

Micro-Frontend architecture needs a **smart caching model**:

✅ Aggressively cache remote MFEs  
✅ Share libraries to avoid duplication  
✅ Service Worker for offline + background updates  
✅ Runtime versioning to avoid stale bundles  
✅ LocalStorage fallback for high resilience

**Module Federation + Runtime Versioning**

In a microfrontend setup using Module Federation, to avoid caching issues and ensure that the host app always loads the latest remoteEntry file, you should use the remoteEntry file with a content hash in its filename (e.g., remoteEntry.[contenthash].js). This hashed file name changes whenever the remote bundle changes, effectively busting the cache.

**How to Access the Hashed remoteEntry in Another App**

1. Dynamic Remote URL with Content Hash:  
   To load remoteEntry.[contenthash].js dynamically from another app, you cannot hardcode the filename because the hash changes on every build. Instead, use:
   * A manifest or JSON file on the remote server that maps to the latest remoteEntry file name.
   * Or generate and push the filename via CI/CD to the host app config.
2. Example Approach:
   * Host app: Make a request to the remote server’s manifest (like /remoteEntry-manifest.json) at runtime to get the exact filename with the hash.
   * Then: Use that filename in your Module Federation configuration dynamically, for example:

// pseudo-code

const remoteManifest = await fetch('https://remote-app.com/remoteEntry-manifest.json').then(res => res.json());

const remoteEntryUrl = `https://remote-app.com/${remoteManifest.remoteEntry}`;

\_\_webpack\_init\_sharing\_\_('default');

const container = await window.loadRemoteEntry(remoteEntryUrl);

// then consume remote modules as usual

1. Webpack Module Federation Config:  
   In the host app, instead of statically declaring the remote like this:

js

remotes: {

remoteApp: 'remoteApp@https://remote-app.com/remoteEntry.js',

}

you need a dynamic loader that gets the actual hashed path.

1. Benefits:
   * The host app always requests the latest hashed remoteEntry file.
   * The browser cache isn't stale since the hashed filename changes every deployment.

**Summary**

* Avoid using a fixed remoteEntry.js file as it results in bad caching.
* Use remoteEntry.[contenthash].js with a mechanism (like a manifest or API call) to tell the host the current hashed filename dynamically.
* Dynamically load the remoteEntry in the host app based on the information obtained.