**Architecture Styles**

**Module Federation**

Module Federation is a feature in Webpack 5 that allows multiple independently built and deployed front-end applications (also called micro frontends) to share code with each other at runtime. This helps in building applications that are modular, more scalable, and easier to maintain, especially when different teams work on separate parts of an application.

Here’s how Module Federation works:

1. Remote and Host Apps

* Host Application: This is the main application that will load code from other applications (remotes).
* Remote Applications: These are independent applications that expose parts of their code to be used by the host or other remotes.

2. Dynamic Imports

One of the key features of Module Federation is that it allows code to be loaded dynamically at runtime. When a host application needs a module from a remote app, it can fetch it on-demand without bundling the remote module during the host app's build process.

3.Shared Dependencies

Module Federation makes it possible for applications to share common dependencies (e.g., React, Angular, Lodash) without having to load multiple instances of the same dependency, which reduces bundle size and improves load time. Dependencies that should be shared can be explicitly defined in the configuration.

**Advantages: -**

* **Independently Deployable**: Different parts of an application (like components or even entire features) can be developed and deployed independently.
* **Incremental Upgrades**: Module Federation allows you to incrementally upgrade parts of an application without needing to refactor the entire app.
* **Better Code Reuse**: By sharing modules between applications, you avoid duplicating code, which can help reduce the overall size of your applications.
* **Team Independence**: Teams can work on different parts of an app with minimal coupling, promoting parallel development.

When using **Module Federation** with large-scale remote applications, **security** becomes a crucial aspect to manage since you are essentially allowing applications to share and load code dynamically across different domains or modules. Below are some key security strategies you can implement to ensure your remote apps are protected:

**Security in Host App:-**

* 1. **Authentication and Authorization: -**

Single Sign-On (SSO): Implement a centralized authentication system where the authentication process happens once, and the session is shared across all remote and host applications. For example, if you're using Microsoft Authenticator, you could centralize the authentication for all apps.

* 1. **Access Control for Shared Modules: -**
* Granular Access Control: Restrict which modules or components can be shared with other applications based on user roles or permissions. Ensure that the host app cannot load sensitive modules from the remote unless explicitly authorized.
* Role-Based Access Control (RBAC): Implement RBAC at the module or component level. For example, certain modules might be exposed but should only be accessible to users with specific roles.
  1. Monitoring and Auditing
* Logging: Implement proper logging in both host and remote apps to track access and potential security issues. Logs should include information about which components were loaded from which apps, by whom, and when.
* Security Audits: Regularly conduct security audits of both host and remote applications to ensure that there are no vulnerabilities introduced through the federation mechanism.

**Security in the Remote Apps**

Remote apps are equally responsible for securing their internal components and handling specific aspects of security. Their responsibilities include:

1. Module-Level Security:
   * Remote apps should ensure that only the intended and necessary components or modules are exposed to the host app. They should not expose sensitive internal code unless required.
   * Access Control: Remote apps should implement granular access control to ensure that only authorized users or apps can access specific components or features, even if they are exposed to the host app. In our case based on CEEP
2. Authorization Enforcement:
   * While the host app controls high-level access, remote apps should also enforce authorization on their side. For example, they should check if the user has the appropriate permissions before executing critical actions or accessing sensitive data.
   * This is often done by verifying user roles, permissions, or tokens provided by the host app.

Though we intend to show to the users all the apps available but, without access we won’t let them proceed. This ensures users are intrigued to the host app and want further access.

**Example Scenario: Securing Host and Remote Apps**

Let’s assume your host app is handling authentication using OAuth2 with Microsoft Authenticator. When a user tries to load a remote app (e.g., anurag), the host app:

* Authenticates the user via OAuth2 and generates a JWT token.
* Authorizes the user based on roles or permissions before requesting the remote app's modules.
* Lazy loads a remote module from anurag after verifying the user’s permissions.

**Monorepo Architecture Style**