

Micro Frontend



Engineer your
NexTurn
on cloud



Contents

- What is micro frontend?
- Why use micro frontend?
- Micro frontend flow
- Module Federation
- Benefits of Module Federation
- Nx
- Benefits of Nx
- Summary

What is Micro Frontend?

Micro frontend is an architectural and organizational style in which the front-end of the app is decomposed into individual, loosely coupled “micro apps” that can be built, tested, and deployed independently. Simply put, each micro frontend is just code for a fragment — a separate feature — of the web page. These features are owned by independent teams, each having a distinct business area or mission it specialized in.

The micro frontend architecture resembles microservices where loosely coupled components are taken care of by different teams but extends the idea to the browser. With this pattern, these are web application UIs that are composed of semi-independent fragments. Also, teams aren't grouped around a skill or technology, but rather formed around a customer need or use case.

When micro frontend is a good idea

MFE is for teams that require applications to be deployed independently. It is important to consider the cost of MFEs and decide whether it makes sense for your own teams. The style of building micro frontends is ideal for scaling development and as such makes a good fit for larger projects with multiple teams. Having vertically sliced teams adds up to the overall productivity. But it comes with extra costs and maintenance challenges. If you put productivity first and are ready for overhead, micro frontends can be considered.

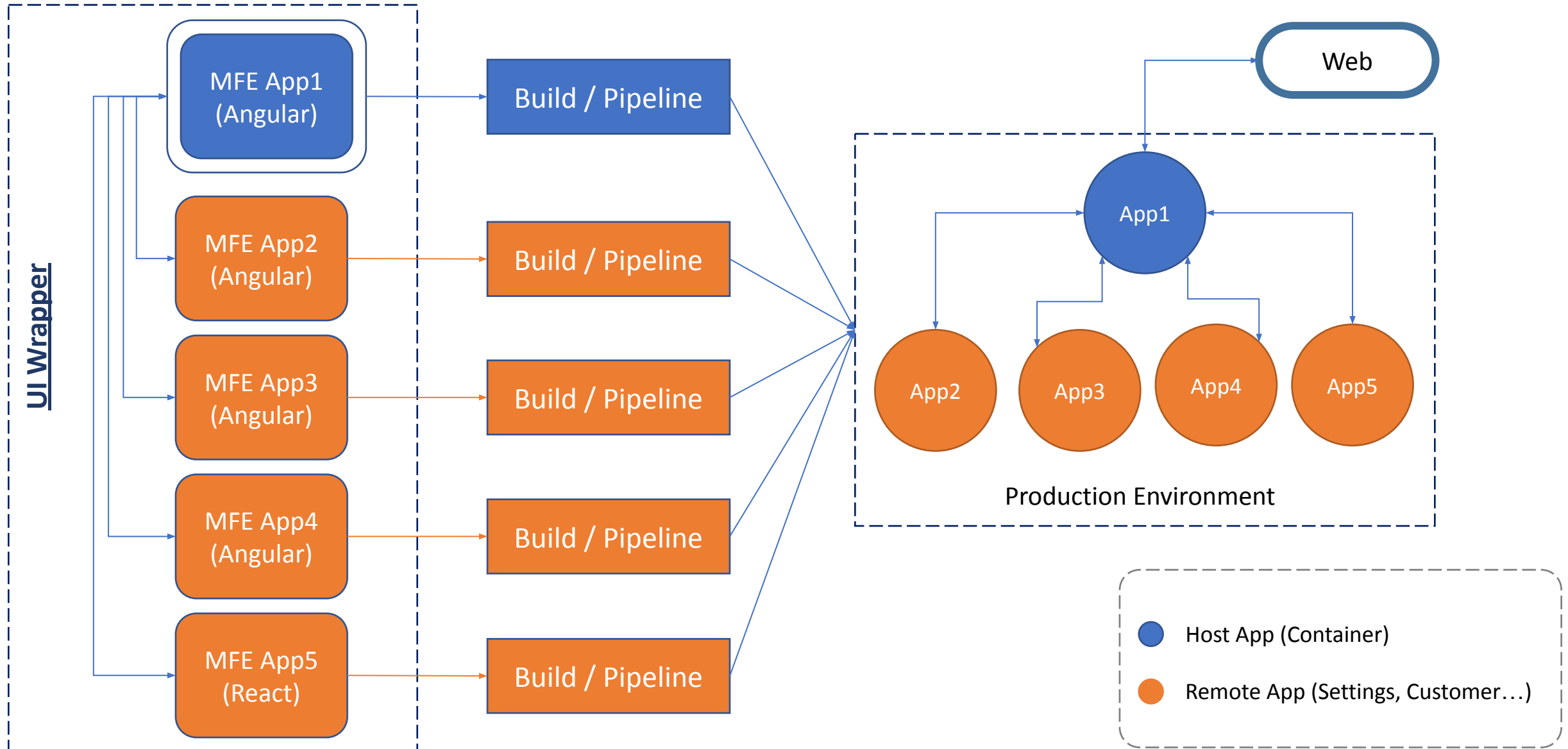
When micro frontend isn't a good idea

You need to know the business domain you're working in. If it's not clear what team is a perfect match for implementing a particular feature, you can end up with a messed-up work organization, more complexity, and fewer gains. You don't need to overcomplicate what can be done easily and quickly using a common monolithic approach

Why use Micro Frontend?

- **Better flexibility.** Each micro frontend is self-contained, meaning it can be built, tested, deployed, and updated independently. So, if one team is working on a feature and has pushed a bug fix and another team needs to add its own feature, they don't have to wait for the first team to finish their work.
- **Technology freedom.** Frontend developers working on different fragments of UI can choose different runtime environments, JavaScript frameworks, and overall technology stacks based on business requirements. A new framework can be used on top of the old architecture.
- **Multiple, smaller codebases.** Micro frontends will have their own codebases, each of which will be smaller in size and more manageable. There will be fewer developers working on a particular part of UI, leading to simpler code reviews and better organization of all things.
- **Easy app scaling.** Another advantage of micro frontends is that each feature can be scaled independently. This makes the entire process more cost- and time-effective than with monoliths when the whole application has to be scaled every time a new feature is added.
- **Autonomous teams and systems.** Each product team and therefore feature can operate with minimal dependency on others, which means it can function even when the neighboring components are down.
- **The lower entry barrier for new developers.** Micro frontends are easier to understand and manage than UI that isn't split up into smaller and simpler components. So, new players won't have to spend ages learning the whole code before they start bringing value to the project.

Micro Frontend Flow



Module Federation

Module Federation is an architecture that addresses these issues and revolutionized the micro-frontend strategy. With Module Federation, an application runs code dynamically from another bundle or build with its code-sharing and functionality-consuming abilities during runtime, paving the path to the successful utilization of micro-frontend technology. In addition, the use of sharable dependencies improves the compactness of the application. Module Federation gives a sense of familiarity to developers, too, as it is a part of the Webpack ecosystem the developers may have already used.

Components of Module Federation:

- **Host:** The Webpack build initialized first during a page build is the host. The host application contains typical features from a SPA or SSR application that boots and renders the components the user would see first.
- **Remote:** Remote is another Webpack build from which the host can consume a part. It can strictly be either a remote or a host. The major functionality of a remote is to expose modules to be consumed.

Key configuration options of Module Federation:

- **library:** The library contains a name (name of the library) and type options. It helps decide how the exposed code is retrieved and stored.
- **name:** The name configuration option uniquely identifies the exposed container.
- **filename:** This determines the filename of the output bundle in Module Federation.
- **exposes:** Through Module Federation, you can also share file types in addition to the modules. This configuration option depicts the path to the files or modules exposed by the container.
- **shared:** This is an essential configuration option, allowing you to share libraries on which the exposed module depends to run.

Benefits of Module Federation

- Using the shared option minimizes dependency duplication, as the remotes depend on the host's dependencies. If the host lacks a dependency, the remote downloads its dependency only when necessary.
- Server-side rendering is possible, as Module Federation can work in any environment.
- Enhances build performance, as Module Federation supports the micro-frontend approach, allowing different teams to work simultaneously on a larger application by building and deploying independent, split projects.
- Module Federation supports lazy loading bundles to load modules only when necessary, resulting in better web performance.
- Module Federation manages a dependency graph for shared dependencies. It helps download necessary dependencies even when there is an issue like a network failure.
- Improves both user experience and developer experience.

Nx

Nx is a smart, fast and extensible build system with first class monorepo support and powerful integrations. Nx makes scaling easy. Modern techniques such as distributed task execution and computation caching make sure your CI times remain fast, even as you keep adding projects to your workspace. There are two styles of monorepos that you can build with Nx: **integrated repos** and **package-based repos**.

- Package-based repos focus on flexibility and ease of adoption.
- Integrated repos focus on efficiency and ease of maintenance.

Package based Repo

A package-based repo is a collection of packages that depend on each other via package.json files and nested node_modules. With this set up, you typically have a different set of dependencies for each project. Build tools like Jest and Webpack work as usual, since everything is resolved as if each package was in a separate repo and all of its dependencies were published to npm. It's very easy to move an existing package into a package-based repo, since you generally leave that package's existing build tooling untouched. Creating a new package inside the repo is just as difficult as spinning up a new repo, since you have to create all the build tooling from scratch.









Integrated Repo

An integrated repo contains projects that depend on each other through standard import statements. There is typically a single version of every dependency defined at the root. Sometimes build tools like Jest and Webpack need to be wrapped in order to work correctly. It's harder to add an existing package to this style of repo, because the build tooling for that package may need to be modified. It's very simple to add a brand-new project to the repo, because all the tooling decisions have already been made.





Benefits of Nx

- Never rebuild the same code twice:** Nx is smart! It can figure out whether the same computation has run before and can restore the files and the terminal output from its cache.
- Distributed task execution (DTE):** Smart, automated, dynamic distribution of tasks across multiple machines to get maximum parallelization and CPU efficient CI runs.
- Remote caching:** Share your local computation cache with teammates and your CI system for maximum efficiency.
- Only run what changed:** Nothing is faster than not running a task. Nx analyzes your project graph and can diff it against a baseline to determine which projects changed and what tasks need to be re-run.
- Affected deployments:** When changes are merged, Nx allows to test and deploy the affected applications automatically. Using Nx we can have a single pipeline for all of our apps, which means we have less overhead in maintaining that pipeline.

Comparison

MFE Requirement	Module Federation	Nx	Remarks
Modular development			Module federation provides separate repositories for each application. Nx is primarily used for building monorepos, but supports micro frontend architecture using module federation.
Modular deployment			Module federation allows to have separate deployments. Nx provides deployment capabilities where the application with the change will only get deployed.
Polyglot – Mix of Angular, React etc			Nx doesn't promote MFE anarchy. In Nx integrating multi frameworks can be done using module federation.
Common access control logic			Needs to be explored.
Inter module communication, data sharing			Are designed to be decoupled, putting too many dependencies among them may result in debugging issues

Comparison

MFE Requirement	Module Federation	Nx	Remarks
Common User Experience			Need to explore how common libraries like error handlers, Toast messages etc can be implemented so that user experience doesn't change.
Common Stylesheets			Doesn't come out of the box. But can be done by creating a lib application with the assets.
Size (minimize footprint)			Micro-frontend may have duplicated code or functionality which increases the bundle size and memory consumption

Summary

- While Module Federation enables faster builds by vertically slicing your application into smaller ones, the MFE architecture layers independent deployments on top of federation. Teams should only choose MFEs if they want to deploy their host and remotes on different cadences.
- Teams should consider a process for changes to core libraries that require deploying all applications. These types of changes should occur infrequently as to not disrupt other releases for bug fixes or new features.
- Since deployments are not atomic, there can be cases of mismatched libraries between the host and remotes. We recommend that teams deploy their applications whenever changes to a shared library affects them. You can further mitigate mismatch issues by minimizing the amount of libraries you share.
- Teams should also avoid MFE anarchy, where competing technologies are mixed. Instead, teams should agree upon the adopted technologies, which allows easier collaboration across teams.

Proposed solution – Micro frontend using module federation

Q&A

- 1. How does device permissions, cookie storage, local storage etc. work in micro frontend in port-based hosting?**
Permission & local storage are set by the container application. Since the modules from remote applications are run inside container application the permissions and local storage in the container application only. Eg: Let's say the container application is running on port 80, the remote applications are running in 8080 and 8081 respectively. So, if the remote applications requires location or camera & microphone access it will be requested and set in the container application only. HttpCookie needs to be verified.
- 2. Evaluation on how push notifications will work in micro frontend?**
Needs to be verified
- 3. How to handle dependencies between applications?**
Dependencies which are created as a common library between applications (eg: bootstrap used in CSS) needs to be updated throughout all applications on major updates.
- 4. How access control related security works in remote applications?**
Auth Guards needs to be setup both in container as well as remote applications. Usage needs to be verified
- 5. How do we manage repository access for developers?**
Each developer will be having access to the container application and their assigned remote application. Container application will have all the common components like login, header, footer, menu etc.
- 6. How do we manage deployments?**
Each application will be having their own package files so each applications will have separate builds as well as pipelines. Hosting can be done in folder wise where each application is placed in their respective folders or port wise where each application will be hosted on separate ports. Needs to be verified by DevOps team.

Thank You