Mayur Kulkarni
Posted on Jul 30, 2025

Step-by-Step Guide to Angular Microfrontends with Nx and Dynamic Module Federati

Mayur Kulkarni

I am a senior frontend engineer. Expert in Angular, Vue.

JOINED
Jul 22, 2025

Trending on DEV Community 🔥

What was your win this week?
#weeklyretro
#discuss

The Introvert's Guide to Networking: How I Stopped Faking It and Started Growing
#discuss
#watercooler
#career
#networking

on

#nx #angular #webpack #microfrontend

In this article, we'll walk through how to implement a scalable Microfrontend architecture in Angular using Webpack Module Federation with Nx, with a focus on Dynamic Module Federation.

What is Microfrontend?

Microfrontend is an architectural approach where a large frontend application is broken down into **smaller, independent, self-contained apps** — similar to how microservices work on the backend.

Each smaller app (called a **remote**) represents a specific business domain or UI section — for example:

- A `login` page
- A `products` page
- A `cart` or `profile` section

These micro apps can:

- Be developed by **different teams**
- Be **deployed independently**
- Use different release cycles
- Be dynamically **loaded at runtime** by a central shell app

What is a Host App?

The **host** is the main application — the shell or container — that:

- Loads one or more remote apps at

runtime
- Manages global routing, layout, or shared services (like authentication or navigation)

Think of it like a dashboard or entry point where all microfrontends come together to form a complete UI.

What is a Remote App

A **remote** is a microfrontend — a standalone Angular app that:

- Has its own routing and components
- Can be developed/tested independently
- Is exposed to the host using **Webpack Module Federation**

The host consumes the

remote app's UI and logic as if it were a part of its own application — but without tight coupling.

In This Project Example

Host App: `dashboard`

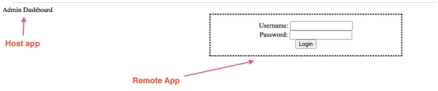
- Acts as the entry point for the user
- Loads other microfrontend apps dynamically
- Has shared layout and routing logic
- Reads the manifest file to dynamically load remote apps at runtime

Remote App: `login`

- Contains a simple login form and logic
- Can be developed and deployed independently
- Is exposed to the host using Module Federation

The host (`dashboard`) loads the remote (`login`) using Dynamic Module Federation.

When the user navigates to /login, the host dynamically loads the remote login app and renders its content as if it were a part of the host.



What is Nx

Nx is a powerful tool for building and managing monorepos — where you keep multiple frontend and backend apps, libraries, and shared code in a single workspace.

It helps you:

- Generate apps and libraries quickly
- Share code easily across projects

- Speed up builds with smart caching
- Scale large codebases efficiently

What are advantages of using Nx for creating microfrontends

- Simplified setup

Nx provides powerful generators (nx g @nx/angular:host, remote, etc.) that automate the boilerplate for setting up Webpack Module Federation, routing, and lazy loading — saving a lot of time.

- Monorepo Support

Nx is built for monorepos, allowing you to manage multiple microfrontend apps and shared libraries in one workspace. This

helps with:

- Better code sharing
- Easier dependency tracking
- Consistent tooling
- Automatic Dependency Sharing

When you use shared libraries (like an auth service), Nx automatically configures Module Federation sharing so they're not duplicated across remotes — no manual Webpack config needed.

- Faster Builds with Caching

Nx uses smart build and test caching — if nothing has changed in a remote, it skips rebuilding it. This makes development and CI/CD pipelines faster.

And many more advantages...

What is static and dynmaic module federation

When building Microfrontends with Angular and Nx, the host app (like dashboard) needs to know where the remote apps (like login) are located.

There are two ways to do this: Static and Dynamic.

Static Module Federation (Build-time URL)

In Static Federation:

- The remote app (login) is hardcoded in the host (dashboard) during the build.
- That means: if the

remote URL changes (for example, from http://localhost:4201 to https://staging.example.com/login), you must rebuild the host app to update the URL.

In This Project:

When we'll first create the dashboard and login apps using Nx generators, the host will be configured statically:

```
remotes: ['login'
```

This works fine locally — but when deploying to staging or production, it becomes a problem.

Dynamic Module Federation (Runtime

URL)

Dynamic Federation solves this.

Instead of hardcoding the remote URL, the host app (dashboard) loads the remote (login) dynamically at runtime using a small JSON file like this:

```
// module-federat
{
  "login": "https
}
```

Now, you don't need to rebuild the host.
You just change the manifest file depending on the environment.

In This Project:
We updated the main.ts of the dashboard app like this:

```
fetch('/module-fe
```

```
    .then(res => re
    .then(setRemote
    .then(() => imp
```

For real-world use (like CI/CD, multiple environments), always go with Dynamic Federation.
For learning or small apps, Static is fine to start with.

Let's begin to create.
We'll create following:

- A Host app (dashboard) – loads other apps
- A Remote app (login) – login form
- Shared logic (user-auth service)
- Then convert this setup to **Dynamic Module Federation**

Step-by-Step: Building Microfrontends with Nx

Create Nx Workspace

```
npx create-nx-wor
cd ng-mf
npx nx add @nx/ar
```

Create the Host App (Static Setup First)

```
nx g @nx/angular:
```

Create the Remote App (Login)

```
nx g @nx/angular:
```

After creating the apps, if you check module-federation.config.ts in both, you can see:

In the Host (dashboard)

```
const config: Moc
  name: 'dashboar
  remotes: ['logi
};
```

`remotes` : Lists the remote apps to load (e.g., login).

In the Remote (Login)

```
const config: Moc
  name: 'login',
  exposes: {
    './Routes': '
  },
};
```

`exposes` : Defines what the remote shares with the host — like routes or components.

Create Shared Library

```
nx g @nx/angular:
nx g @nx/angular:
```

Create a User service using RxJS BehaviorSubject to track login state

Add Login Form in

Login Remote

```
In apps/login/src
```

Update Dashboard App

- Show login form if user is not authenticated.
- Show dashboard content if user is logged in.

Add route (in apps/dashboard/src/app/app.routes.ts):

```
{
  path: 'login',
  loadChildren: (
}
```

This is where dashboard (host) loads login (remote).

Converting Static Federation to Dynamic

a. Create a manifest file

apps/dashboard/public/module-federation.manifest.json

```
{
  "login": "http:
}
```

b. Update main.ts in dashboard

```
fetch('/module-fe
  .then(res => re
  .then(definitic
  .then(() => imp
```

c. Remove remotes: ['login'] from module-federation.config.ts

d. Update route loading

```
loadChildren: ()
  loadRemoteModul
```

Serve the app

```
nx serve dashboar

nx serve login //
```

For the complete code implementation (including dashboard and login content), please refer my below github repository.

[GitHub repository for this project]

What's next?

- Try implementing your own host app
- Try adding a second remote app (e.g. `profile`, `cart`, or `settings`)
- Deploy remotes to different environments and update the manifest to test runtime switching

Thanks for reading!

Top comments (0)