**🚀 Solution clé en main pour une architecture Micro-Frontend**

Pour **faciliter la mise en place** d’une architecture micro-frontend, je propose une **boîte à outils complète** avec une **CLI (Command Line Interface)** et une **structure modulaire** permettant de générer, configurer et orchestrer des micro-frontends Angular **rapidement**.

**🎯 Objectifs de la solution**

✔ **Automatiser la création des micro-frontends** (Shell + Remote Apps)  
✔ **Configurer Webpack Module Federation automatiquement**  
✔ **Simplifier la communication et le partage des dépendances**  
✔ **Gérer l’orchestration et la navigation entre micro-frontends**  
✔ **Intégrer des bonnes pratiques (Lazy Loading, State Management, Sécurité)**

**🏗 Solution : Une CLI pour automatiser l'architecture**

**📌 1️⃣ Outil CLI : ng-mfe-cli**

Créons un **outil CLI** qui permet de générer une architecture micro-frontend en **quelques commandes**.

**📍 Installation de la CLI**

npx ng-mfe-cli init my-microfrontend-app

Ou en l'installant globalement :

npm install -g ng-mfe-cli

ng-mfe-cli init my-microfrontend-app

**📍 Génération d’un Shell App**

ng-mfe-cli generate shell --port=4200

👉 **Résultat** : Un shell Angular configuré avec Webpack Module Federation.

**📍 Génération d’un Micro-Frontend (Remote App)**

ng-mfe-cli generate remote --name=dashboard --port=4201

👉 **Résultat** : Un micro-frontend **dashboard** indépendant avec son propre Webpack.

**📍 Ajout automatique au Shell App**

ng-mfe-cli add-remote dashboard

👉 **Résultat** : Ajoute automatiquement **dashboard** à la configuration du Shell et met à jour app-routing.module.ts.

**🏗 2️⃣ Architecture générée par la CLI**

Après exécution des commandes, on obtient cette **structure automatique** :

/my-microfrontend-app

├── /shell (Application hôte - port 4200)

├── /dashboard (Micro-frontend - port 4201)

├── /profile (Micro-frontend - port 4202)

├── package.json

├── README.md

**Configuration Webpack générée automatiquement**

**🔹 Shell (webpack.config.js)**

new ModuleFederationPlugin({

name: "shell",

remotes: {

dashboard: "dashboard@http://localhost:4201/remoteEntry.js",

profile: "profile@http://localhost:4202/remoteEntry.js"

},

shared: {

"@angular/core": { singleton: true, strictVersion: true },

"@angular/common": { singleton: true, strictVersion: true },

"@angular/router": { singleton: true, strictVersion: true },

},

})

**🔹 Micro-Frontend (webpack.config.js)**

new ModuleFederationPlugin({

name: "dashboard",

filename: "remoteEntry.js",

exposes: {

"./Module": "./src/app/dashboard/dashboard.module.ts"

},

shared: {

"@angular/core": { singleton: true, strictVersion: true },

"@angular/common": { singleton: true, strictVersion: true },

"@angular/router": { singleton: true, strictVersion: true },

},

})

**📡 3️⃣ Communication centralisée entre micro-frontends**

La CLI intègre automatiquement un **Event Bus** avec **RxJS**.

**Service event-bus.service.ts généré automatiquement**

import { Injectable } from '@angular/core';

import { Subject } from 'rxjs';

@Injectable({ providedIn: 'root' })

export class EventBusService {

private eventSubject = new Subject<any>();

emit(event: any) {

this.eventSubject.next(event);

}

get events$() {

return this.eventSubject.asObservable();

}

}

👉 **Résultat** : Chaque micro-frontend peut écouter ou envoyer des événements sans dépendance forte.

**🎨 4️⃣ Gestion avancée des styles et isolation**

Par défaut, la CLI ajoute **Shadow DOM** pour éviter les conflits CSS :

@Component({

selector: 'app-dashboard',

templateUrl: './dashboard.component.html',

styleUrls: ['./dashboard.component.css'],

encapsulation: ViewEncapsulation.ShadowDom

})

👉 **Résultat** : Chaque micro-frontend a son propre scope de styles.

**🔐 5️⃣ Sécurité et Authentification**

📌 La CLI génère un **service d’authentification centralisé** avec JWT.

**Service auth.service.ts**

import { Injectable } from '@angular/core';

@Injectable({ providedIn: 'root' })

export class AuthService {

private token = '';

login(user: string, password: string) {

this.token = 'fake-jwt-token'; // Simuler un token

}

getToken() {

return this.token;

}

}

👉 **Résultat** : Une authentification centralisée, **mutualisée entre micro-frontends**.

**🚀 6️⃣ Déploiement simplifié**

La CLI génère un **Dockerfile** pour chaque micro-frontend.

ng-mfe-cli deploy --target=docker

👉 **Résultat** : Génère un **conteneur Docker** pour chaque micro-frontend.

Exemple de **Dockerfile** :

FROM nginx:alpine

COPY dist/dashboard /usr/share/nginx/html

CMD ["nginx", "-g", "daemon off;"]

**🎯 Résumé des avantages de la solution ng-mfe-cli**

| **✅ Fonctionnalité** | **🎯 Bénéfice** |
| --- | --- |
| 📌 Génération automatique | Plus rapide à mettre en place |
| ⚡ Webpack pré-configuré | Evite les erreurs manuelles |
| 🔄 Event Bus intégré | Facilite la communication entre micro-frontends |
| 🎨 Isolation des styles | Empêche les conflits CSS |
| 🔐 Authentification JWT | Mutualisation de la sécurité |
| 🚀 Dockerisation simple | Déploiement rapide |

**🚀 Conclusion : Une solution efficace pour Angular Micro-Frontend**

Avec cette **CLI (ng-mfe-cli)**, vous pouvez **créer, configurer et déployer** une architecture **micro-frontend** en **quelques commandes** !

**📌 Étapes ultra-simples**

1️⃣ **npx ng-mfe-cli init my-microfrontend-app**  
2️⃣ **ng-mfe-cli generate shell --port=4200**  
3️⃣ **ng-mfe-cli generate remote --name=dashboard --port=4201**  
4️⃣ **ng-mfe-cli add-remote dashboard**  
5️⃣ **ng-mfe-cli deploy --target=docker**

💡 **Résultat** : Un **micro-frontend Angular opérationnel en moins de 5 minutes** ! 🚀

# 🚀 **Guide Complet : Architecture Micro-Frontend avec Angular & Webpack Module Federation**

L’architecture **micro-frontend** permet de diviser une application Angular en plusieurs sous-applications indépendantes, facilitant ainsi le développement, la scalabilité et le déploiement.  
Dans ce guide, nous allons utiliser **Angular, Webpack Module Federation et NX** pour mettre en place cette architecture.

# ✅ **1. Planification et Conception**

## 📌 ****Découpage des Micro-Frontends****

Avant de commencer, identifiez comment diviser l’application :

* **Par domaine fonctionnel** (ex : dashboard, produits, panier, profil utilisateur)
* **Par équipe** (chaque équipe gère un micro-frontend)
* **Par pages** (ex: /dashboard est un micro-frontend autonome)

## 📌 ****Choix de l’approche d’intégration****

* **Approche Route-based** (chargement par route, recommandé)
* **Approche Component-based** (chargement dynamique de composants)

# ✅ **2. Création du Workspace NX**

Si ce n'est pas encore fait, installez **NX** et créez un workspace Angular :

npx create-nx-workspace@latest my-mfe-app --preset=angular

cd my-mfe-app

👉 Cela crée un **monorepo** qui facilite la gestion des micro-frontends.

# ✅ **3. Création des Micro-Frontends avec Module Federation**

## 📌 ****Générer les Applications****

Créez une application shell (host) et des micro-frontends (remotes) avec **NX** :

nx generate @nrwl/angular:host shell --remotes=dashboard,profile,cart

Cela génère :

* shell (hôte principal)
* dashboard, profile, cart (micro-frontends)

## 📌 ****Configurer Webpack Module Federation****

Dans webpack.config.js de chaque application, configurez **Module Federation**.

### 🔹 ****Shell (Host)****

const { ModuleFederationPlugin } = require("webpack").container;

module.exports = {

plugins: [

new ModuleFederationPlugin({

name: "shell",

remotes: {

dashboard: "dashboard@http://localhost:4201/remoteEntry.js",

profile: "profile@http://localhost:4202/remoteEntry.js",

cart: "cart@http://localhost:4203/remoteEntry.js",

},

shared: ["@angular/core", "@angular/common", "@angular/router"],

}),

],

};

### 🔹 ****Dashboard (Remote)****

const { ModuleFederationPlugin } = require("webpack").container;

module.exports = {

plugins: [

new ModuleFederationPlugin({

name: "dashboard",

filename: "remoteEntry.js",

exposes: {

"./DashboardModule": "./src/app/dashboard/dashboard.module.ts",

},

shared: ["@angular/core", "@angular/common", "@angular/router"],

}),

],

};

👉 **Les autres remotes suivent la même logique.**

# ✅ **4. Intégration des Micro-Frontends**

## 📌 ****Configuration des Routes Dynamiques****

Dans app.routes.ts du **shell** :

const routes: Routes = [

{ path: 'dashboard', loadChildren: () => import('dashboard/DashboardModule').then(m => m.DashboardModule) },

{ path: 'profile', loadChildren: () => import('profile/ProfileModule').then(m => m.ProfileModule) },

];

👉 Cela permet de **charger les micro-frontends à la demande**.

## 📌 ****Configuration des Routes dans un Remote****

Dans app.routes.ts de dashboard :

const routes: Routes = [

{ path: '', component: DashboardComponent },

];

export const DashboardRoutingModule = RouterModule.forChild(routes);

# ✅ **5. Communication Entre Micro-Frontends**

## 🔹 ****Utilisation d’un Event Bus avec RxJS****

Dans event-bus.service.ts :

import { Subject } from 'rxjs';

export const eventBus = new Subject();

Un micro-frontend peut envoyer un événement :

eventBus.next({ type: 'ADD\_TO\_CART', payload: product });

Un autre peut écouter cet événement :

eventBus.subscribe(event => console.log(event));

## 🔹 ****Utilisation d’un Service Centralisé****

Créez un service partagé pour la gestion des états globaux avec **NGXS ou Akita**.

# ✅ **6. Gestion des Dépendances Partagées**

Dans webpack.config.js de chaque application, ajoutez :

shared: {

"@angular/core": { singleton: true, strictVersion: true },

"@angular/common": { singleton: true, strictVersion: true },

"@angular/router": { singleton: true, strictVersion: true },

}

👉 Cela évite **les conflits de versions et les chargements multiples**.

# ✅ **7. Déploiement des Micro-Frontends**

Chaque micro-frontend doit être **déployé indépendamment**.

### 📌 ****Hébergement des fichiers****

* Exposer remoteEntry.js sur un **CDN** ou serveur statique
* Le host charge dynamiquement les micro-frontends

### 📌 ****Exemple avec NGINX****

server {

listen 80;

location / {

root /usr/share/nginx/html;

index index.html;

}

location /dashboard/ {

proxy\_pass http://dashboard-service/;

}

}

👉 Le **shell** peut ensuite charger dashboard depuis /dashboard/remoteEntry.js.

# ✅ **8. Tests et Sécurité**

## 🔹 ****Tests Unitaires et End-to-End****

* **Tests unitaires** : Jasmine / Karma
* **Tests E2E** : Cypress

## 🔹 ****Sécurisation des Communications****

* **CORS Policy** : Vérifier les permissions d’accès
* **Auth et Permissions** : Utiliser un système JWT partagé
* **Monitoring et Logs** : Centraliser les logs avec **Sentry, ELK**

# 🎯 **Conclusion**

En appliquant ces **meilleures pratiques**, vous obtenez : ✔️ **Scalabilité et maintenabilité**  
✔️ **Déploiement modulaire et rapide**  
✔️ **Indépendance des équipes de développement**

Grâce à **NX + Webpack Module Federation**, votre architecture **micro-frontend Angular** devient **performante, flexible et évolutive**. 🚀🔥