**Création de micro-front end avec Native fédération**

**(Avec Angular)**

Lien : <https://loiane.com/2025/01/angular-micro-frontends-with-native-federation/>

1. Creation du micro-frontend remote : produits

ng generate application products --prefix app-products (

**Hors d’un workspace : ng new products**

1. Creation de l’application host(shell)

**Hors d’un workspace : ng new shell**

1. Ajout de la librairie Native-federation à chacun des projets

npm i -D @angular-architects/native-federation

1. Configuration de Native-federation au niveau de chaque projet (shell comme remote)

Pour le **shell** :

cd shell && ng g @angular-architects/native-federation:init --port 4200 --type dynamic-host

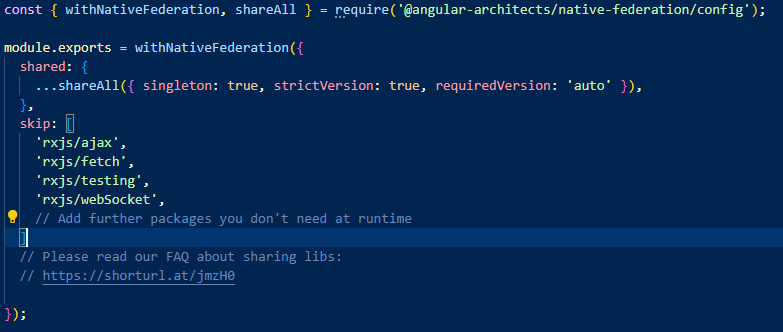
Cette cmd va generer un fichier dans /public (**federation.manifest.json**) à modifier relativement au remote que le shell est sensé charger dynamiquement, comme suit :

{

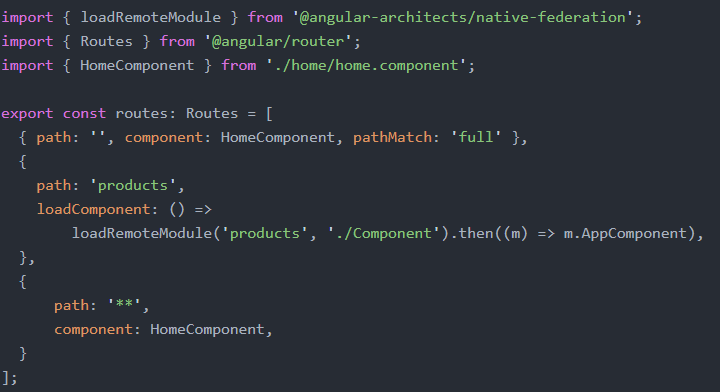
**"products": "**[**http://localhost:4201/remoteEntry.json**](http://localhost:4201/remoteEntry.json)**"**

}

Le fichier federation.config.js (on



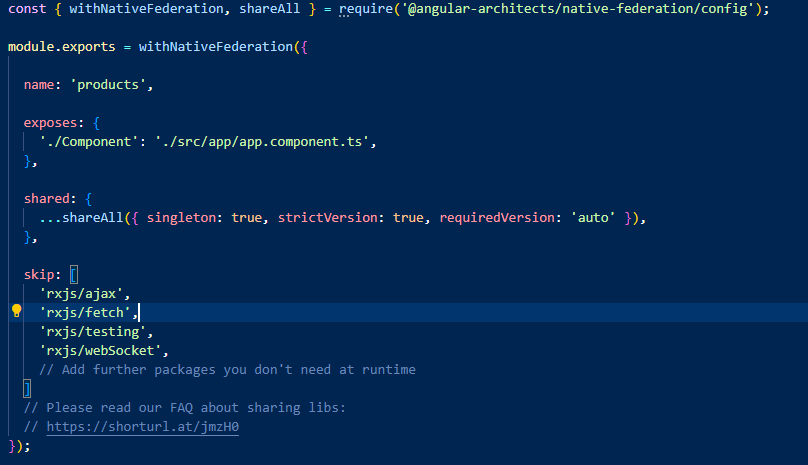
**app-routes.ts** avec chargement dynamique



Pour le **remote** :

cd products && ng g @angular-architects/native-federation:init --port 4201 --type remote

Configuration / définition de composants à exposer



**Combinaison Native-federation et Module-federation**

**(**Shell appli. En Native-fédération appelants des remote m-front en module-fed**)**

 ??????

**ChatGPT : how create native-federation based micro-frontend that exposes a shareable service**

Créer une architecture de **micro-frontend** basée sur **Native Federation** peut être une excellente manière de construire des applications front-end évolutives et maintenables, découplées et pouvant être développées, déployées et mises à l'échelle indépendamment. Dans ce cas, nous allons nous concentrer sur la manière d'exposer un service partageable depuis un micro-frontend vers d'autres en utilisant **Module Federation** (qui est une fonctionnalité de Webpack) et **Native Federation**, une fonctionnalité spécifique aux frameworks comme **React**, **Angular** ou **Vue**.

Voici comment vous pouvez procéder :

**Étapes pour créer un micro-frontend basé sur Native Federation qui expose un service partageable :**

**1. Configurer la Module Federation de Webpack**

Module Federation permet à différentes applications (appelées micro-frontends) de partager des modules et des services dynamiquement au moment de l'exécution. C'est ce qui permet la "fédération" entre différents micro-frontends.

Pour activer **Native Federation** dans une architecture de micro-frontend :

**A. Installer les dépendances**

Assurez-vous d'utiliser Webpack 5 (ou une version ultérieure), car il prend en charge la Module Federation. Vous devrez également installer les dépendances nécessaires comme webpack, webpack-cli, webpack-dev-server, et @module-federation/nextjs pour les frameworks comme Next.js.

npm install webpack webpack-cli webpack-dev-server --save-dev

**B. Configuration de Webpack (Exposer un service)**

Dans le projet où vous souhaitez exposer un service partageable (appelons-le app1), configurez le ModuleFederationPlugin de Webpack pour exposer un service.

// webpack.config.js pour app1

const ModuleFederationPlugin = require("webpack/lib/container/ModuleFederationPlugin");

module.exports = {

mode: "development",

devServer: {

port: 3001,

},

plugins: [

new ModuleFederationPlugin({

name: "app1",

filename: "remoteEntry.js", // Le point d'entrée pour votre service

exposes: {

'./MySharedService': './src/MySharedService', // Exposer un service/module

},

shared: ["react", "react-dom"], // Partager des bibliothèques communes

}),

],

};

Dans cet exemple, app1 expose MySharedService situé dans le dossier src.

**C. Code du Service (MySharedService.js)**

Par exemple, cela pourrait être un simple service que d'autres applications consommeront :

// src/MySharedService.js

export const MySharedService = {

getData: () => {

return fetch('https://api.example.com/data')

.then(response => response.json())

.then(data => data);

}

};

**D. Côté Consommateur (App2)**

Dans l'application consommatrice (app2), configurez le ModuleFederationPlugin pour consommer le service exposé.

// webpack.config.js pour app2

module.exports = {

mode: "development",

devServer: {

port: 3002,

},

plugins: [

new ModuleFederationPlugin({

name: "app2",

remotes: {

app1: "app1@http://localhost:3001/remoteEntry.js", // Lien vers le point d'entrée distant de app1

},

shared: ["react", "react-dom"],

}),

],

};

**E. Consommer le service exposé dans App2**

Dans app2, vous pouvez maintenant importer et utiliser dynamiquement MySharedService exposé par app1.

// Dans App2 (composant React)

import React, { useEffect, useState } from "react";

const App2 = () => {

const [data, setData] = useState(null);

useEffect(() => {

const loadService = async () => {

const { MySharedService } = await import("app1/MySharedService");

const data = await MySharedService.getData();

setData(data);

};

loadService();

}, []);

return (

<div>

<h1>App2</h1>

<pre>{JSON.stringify(data, null, 2)}</pre>

</div>

);

};

export default App2;

**2. Gérer la version des services partagés**

Vous souhaiterez peut-être gérer différentes versions des services partagés. Le ModuleFederationPlugin de Webpack peut gérer cela grâce à la propriété shared, où vous pouvez spécifier les versions. Vous pouvez définir la version des dépendances partagées, comme React ou toute autre bibliothèque, pour vous assurer que tous les micro-frontends sont synchronisés avec les mêmes versions.

shared: {

react: { singleton: true, requiredVersion: '^17.0.0' },

'react-dom': { singleton: true, requiredVersion: '^17.0.0' },

}

Cela garantit qu'une seule version de React et ReactDOM est utilisée à travers vos applications.

**3. Construire et exécuter**

Pour exécuter et voir les micro-frontends fonctionner ensemble, suivez ces étapes :

* **App1** (fournisseur de service) : Exécutez la première application exposant le service.
* npm run start -- --port 3001
* **App2** (consommateur) : Exécutez la deuxième application qui consomme le service exposé.
* npm run start -- --port 3002

Ouvrez les navigateurs à http://localhost:3001 et http://localhost:3002. App2 chargera et utilisera dynamiquement le service exposé par App1.

**4. Native Federation**

Bien que le ModuleFederationPlugin de Webpack soit le mécanisme central, **Native Federation** fait souvent référence à des frameworks qui offrent un support intégré pour de telles techniques de fédération. Des frameworks comme **Next.js** peuvent supporter la federation de modules avec une configuration minimale.

Par exemple, dans Next.js, vous pouvez configurer la fédération en utilisant @module-federation/nextjs pour une intégration simplifiée de la fédération de Webpack.

npm install @module-federation/nextjs

Et utilisez le plugin pour exposer des services et les consommer à travers vos micro-frontends.

**5. Considérations pour la production**

Pour les déploiements en production, vous voudrez généralement :

* **Optimiser la construction** : Utilisez les optimisations de production de Webpack.
* **CDN/URLs distantes** : Hébergez les fichiers d'entrée fédérés (remoteEntry.js) sur un CDN pour un accès rapide et distribué à l'échelle mondiale.
* **Mise en cache** : Mettez en cache les modules partagés pour vous assurer qu'ils ne sont chargés qu'une seule fois.

**Conclusion**

Dans cette configuration, nous avons utilisé **Module Federation de Webpack** pour exposer et consommer un service partagé (MySharedService) entre deux micro-frontends (app1 et app2). Les outils de **Native Federation**, comme @module-federation/nextjs pour Next.js, simplifient encore ce processus si vous utilisez des frameworks spécifiques, mais le concept de base reste le même.

Faites-moi savoir si vous souhaitez approfondir un aspect spécifique de ce processus !

**ççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççççç**

**chatGpt  prompt :**

**Dans un contexte module federation d'angular, générez-moi le code d'un micro-frontend exposant un service et celui qui consomme le service exposé.**

**Améliorer en considérant un deuxième micro-frontend remote consommant le même service**

Voici une version améliorée du projet Module Federation dans Angular avec deux micro-frontends (remote1 et remote2) consommant un service exposé par remote1.

## 🏗 ****Architecture du Projet****

Nous allons structurer notre projet avec trois applications :

1. **host** (application principale) → Consomme le service depuis remote1.
2. **remote1** (premier micro-frontend) → Expose un service (RemoteService).
3. **remote2** (deuxième micro-frontend) → Consomme le service exposé par remote1.

## 📌 ****1. Création des Applications****

Créez les trois applications Angular en exécutant :

ng new host --style=css --routing=true

cd host

ng add @angular-architects/module-federation

cd ..

ng new remote1 --style=css --routing=true

cd remote1

ng add @angular-architects/module-federation

cd ..

ng new remote2 --style=css --routing=true

cd remote2

ng add @angular-architects/module-federation

## 🎯 ****2. Configuration des Micro-Frontends****

### ✅ ****2.1. Configuration de**** remote1 ****(Exposant le Service)****

#### ****🛠 Modifiez**** webpack.config.js

Ajoutez l’exposition du service :

const { shareAll, withModuleFederationPlugin } = require('@angular-architects/module-federation/webpack');

module.exports = withModuleFederationPlugin({

name: 'remote1',

exposes: {

'./RemoteService': './src/app/remote.service.ts',

},

shared: {

...shareAll({ singleton: true, strictVersion: true, requiredVersion: 'auto' }),

},

});

#### ****🛠 Implémentez le Service (****remote.service.ts****)****

Dans src/app/remote.service.ts :

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

@Injectable({

providedIn: 'root'

})

export class RemoteService {

getMessage(): string {

return 'Hello from Remote1!';

}

}

#### ****🛠 Modifiez**** main.ts ****pour charger Module Federation****

import { loadRemoteEntry } from '@angular-architects/module-federation';

Promise.all([

loadRemoteEntry('http://localhost:4201/remoteEntry.js', 'remote1')

])

.then(() => import('./bootstrap'))

.catch(err => console.error(err));

### ✅ ****2.2. Configuration de**** remote2 ****(Consommant le Service de**** remote1****)****

#### ****🛠 Modifiez**** webpack.config.js

Ajoutez la dépendance vers remote1 :

const { shareAll, withModuleFederationPlugin } = require('@angular-architects/module-federation/webpack');

module.exports = withModuleFederationPlugin({

name: 'remote2',

remotes: {

remote1: 'remote1@http://localhost:4201/remoteEntry.js',

},

shared: {

...shareAll({ singleton: true, strictVersion: true, requiredVersion: 'auto' }),

},

});

#### ****🛠 Consommez le Service dans**** app.component.ts

import { Component, Inject, Injector } from '@angular/core';

@Component({

selector: 'app-root',

template: `<h2>Remote2 says: {{ message }}</h2>`,

})

export class AppComponent {

message: string = '';

constructor(private injector: Injector) {

this.loadRemoteService();

}

async loadRemoteService() {

const { RemoteService } = await import('remote1/RemoteService');

const remoteService = this.injector.get(RemoteService);

this.message = remoteService.getMessage();

}

}

## ✅ ****2.3. Configuration du**** host ****(Consommant**** remote1****)****

#### ****🛠 Modifiez**** webpack.config.js

Ajoutez la consommation de remote1 et remote2 :

const { shareAll, withModuleFederationPlugin } = require('@angular-architects/module-federation/webpack');

module.exports = withModuleFederationPlugin({

remotes: {

remote1: 'remote1@http://localhost:4201/remoteEntry.js',

remote2: 'remote2@http://localhost:4202/remoteEntry.js',

},

shared: {

...shareAll({ singleton: true, strictVersion: true, requiredVersion: 'auto' }),

},

});

#### ****🛠 Affichez les Messages dans**** app.component.ts

import { Component, Inject, Injector } from '@angular/core';

@Component({

selector: 'app-root',

template: `

<h1>Host Application</h1>

<h2>Message from Remote1: {{ messageRemote1 }}</h2>

<h2>Message from Remote2: {{ messageRemote2 }}</h2>

`,

})

export class AppComponent {

messageRemote1: string = '';

messageRemote2: string = '';

constructor(private injector: Injector) {

this.loadRemoteServices();

}

async loadRemoteServices() {

const { RemoteService } = await import('remote1/RemoteService');

const remoteService = this.injector.get(RemoteService);

this.messageRemote1 = remoteService.getMessage();

const { AppComponent: Remote2Component } = await import('remote2/AppComponent');

this.messageRemote2 = new Remote2Component().message;

}

}

## 🚀 ****3. Lancer les Applications****

Démarrez les applications sur des ports différents :

# Dans remote1

ng serve --port 4201

# Dans remote2

ng serve --port 4202

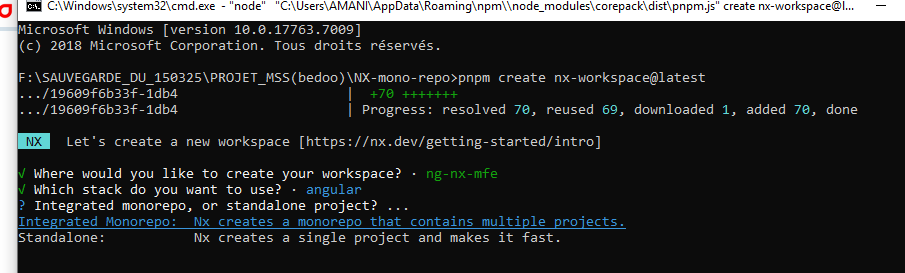
# Dans host

ng serve --port 4200

## 🎯 ****4. Résultat Final****

* host affiche :
  + ✅ **"Message from Remote1: Hello from Remote1!"**
  + ✅ **"Message from Remote2: Hello from Remote1!"**
* remote2 affiche :
  + ✅ **"Remote2 says: Hello from Remote1!"**

Ce setup démontre comment un micro-frontend (remote1) peut exposer un service utilisé à la fois par un autre micro-frontend (remote2) et par l'application hôte (host). 🚀

****