1. Add this to your *package.json* right above the dependencies sections:

"resolutions": {

"webpack": "^5.0.0"

},

1. Configure Ng CLI to use yarn using this command in your command line:

ng config cli.packageManager yarn

1. Add the *@angular-architects/module-federation package* to both your shell Angular application and your remote Angular applications using this command:

ng add @angular-architects/module-federation --project <<NAME\_OF\_SHELL> --port 5000

ng add @angular-architects/module-federation --project <<NAME\_OF\_REMOTE>> --port 3000

* 1. Doing this will complete the following tasks for you:
     1. Generates the skeleton of a partial webpack.config.js for using module federation (sits on top of the Angular webpack config that the CLI handles
     2. Installing a custom builder making webpack within the CLI use the generated webpack.config.js.
     3. Assigning a new port for ng serve so that several projects can be served simultaneously.
     4. Sets up your project’s bootstrap.ts and main.ts files for dynamic imports

1. We need to load the remote upfront when the Webpack CLI is compiling in order to make sure it can properly determine the version to use. We can do this with the loadRemoteEntry helper function. Add this to your main.ts file in your shell project:

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

Promise.all([

loadRemoteEntry('http://localhost:7000/remoteEntry.js', 'vii'),

loadRemoteEntry('http://localhost:8000/remoteEntry.js', 'viii')

]

.catch(err => console.error('Error loading remote entries', err))

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

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

1. Add a route to the shell router for the new remote module using the @angular-architects/module-federation package loadRemoteModule helper function like this:

export const APP\_ROUTES: Routes = [

[…]

{

path: 'vii',

loadChildren: () => loadRemoteModule({

// If we skipped step 4 we would need to include the url for remoteEntry.js

// remoteEntry: ‘http://localhost:7000/remoteEntry.js’,

remoteName: ‘vii’,

exposedModule: ‘./Module’

},[…]).then(m => m.HomeModule)

},

[…]

1. In the shell app’s *webpack.config.js* the remotes list should be empty (to be loaded dynamically with the Webpack CLI and add any npm packages you would like shared:

plugins: [

new ModuleFederationPlugin({

remotes: {},

shared: {

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

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

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

[…]

],

[…]

* 1. NOTES:
     1. Defining a shared package as a singleton will make it only exist in memory 1 time instead of being loaded separately by each app, resulting in smaller bundle size
     2. Setting strictVersion to true means that both apps are required to use the same version of that package in their *package.json*. If this is false or omitted Webpack will attempt to use the highest version available.

1. In the remote app’s *webpack.config.js* add the definition for that remote and what modules or components it exposes and add any npm packages you would like shared:

plugins: [

new ModuleFederationPlugin({

// For remotes (please adjust)

name: "vii",

filename: "remoteEntry.js",

exposes: {

'./vii: './angular/vii/src/app/home/home.module.ts',

},

shared: {

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

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

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

[…]

],

[…]