Tutorial: Getting Started with Webpack Module Federation and Angular

This tutorial shows how to use Webpack Module Federation together with the Angular CLI and the `@angular-architects/module-federation` plugin. The goal is to make a shell capable of **loading a separately compiled and deployed microfrontend**.

Step 1: Install and Inspect the MFE Starterkit

In this part you will clone the microfrontend starterkit and inspect its projects.

1. Install the dependencies:

```
yarn / npm i
```

2. Start the shell (`ng serve shell -o`) and inspect it a bit:

   i. Click on the `flights` link. It leads to a dummy route. This route will later be used for loading the separately compiled microfrontend.

   Please **ignore depreaction warnings**. They are a temporal issue in the current CLI beta when using webpack 5.

   ii. Have a look to the shell's source code.

   > Please note that the current CLI **beta** lacks some features when using it with webpack 5, e. g. **reloading an application in debug mode** (when using ng serve). Hence, you have to restart ng serve after changing a source file. This is just a temporal limitation and will be solved with one of the upcoming versions.

   iii. Stop the CLI (`CTRL+C`).

3. Do the same for the microfrontend. In this project, it's called `mfe1` (Microfrontend 1) You can start it with `ng serve mfe1 -o` .

Step 2: Activate and Configure Module Federation

Now, let's activate and configure module federation:

1. Install `@angular-architects/module-federation` into the shell and into the micro frontend:

```
ng add @angular-architects/module-federation --project shell --port 5000

ng add @angular-architects/module-federation --project mfe1 --port 3000
```

This activates module federation, assigns a port for ng serve, and generates the skeleton of a module federation configuration.

2. Switch into the project `mfe1` and open the generated configuration file `projects\mfe1\webpack.config.js`. It contains the module federation configuration for `mfe1`. Adjust it as follows:

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

[...]

module.exports = {
    output: {
        uniqueName: "mfe1"
    },
    optimization: {
        // Only needed to bypass a temporary bug
        runtimeChunk: false
    },
    plugins: [
        new ModuleFederationPlugin({<pre><code>        // For remotes (please adjust)
        name: "mfe1",
        filename: "remoteEntry.js",

        exposes: {
            './Module': './projects/mfe1/src/app/flights/flights.module.ts',
        },
        shared: {
            "@angular/core": { singleton: true, strictVersion: true },
            "@angular/common": { singleton: true, strictVersion: true },
            "@angular/router": { singleton: true, strictVersion: true },
            [...]
        }
    }),
    [...]
],</code></pre>};
```

This exposes the `FlightsModule` under the Name `./Module.`. Hence, the shell can use this path to load it.

3. Switch into the `shell` project and open the file `projects\shell\webpack.config.js`. Adjust it as follows:

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

[...]

module.exports = {
    output: {
        uniqueName: "shell"
    },
    optimization: {
        // Only needed to bypass a temporary bug
        runtimeChunk: false
    },
    plugins: [
        new ModuleFederationPlugin({
            // For hosts (please adjust)
            remotes: {
                'mfe1': "mfe1@http://localhost:3000/remoteEntry.js"
            },
            shared: {
                "@angular/core": { singleton: true, strictVersion: true },
                "@angular/common": { singleton: true, strictVersion: true },
                "@angular/router": { singleton: true, strictVersion: true },
                [...]
            }
        }),
        [...]
    ],
};
```

This references the separately compiled and deployed `mfe1` project. There are some alternatives to configure its URL (see links at the end).

4. Open the `shell`'s router config (`projects\shell\src\app\app.routes.ts`) and add a route loading the microfrontend:

```
{
    path: 'flights',
    loadChildren: () => import('mfe1/Module').then(m => m.FlightsModule)
},
```

Please note that the imported URL consists of the names defined in the configuration files above.

5. As the Url `mfe1/Module` does not exist at compile time, ease the TypeScript compiler by adding the following line to the file `projects\shell\src\decl.d.ts`:

```
declare module 'mfe1/Module';
```

Step 3: Try it out

Now, let's try it out!

1. Start the `shell` and `mfe1` side by side:

```
ng serve shell -o
ng serve mfe1 -o
```

Hint: You might use two terminals for this.

2. After a browser window with the shell opened (`http://localhost:5000`), click on `Flights`. This should load the microfrontend into the shell:

3. Also, ensure yourself that the microfrontend also runs in standalone mode at http://localhost:3000:

Congratulations! You've implemented your first Module Federation project with Angular!

Step 4: Switch to Dynamic Federation

Now, let's remove the need for registering the micro frontends upfront with with shell.

1. Switch to your `shell` application and open the file `webpack.config.js`. Here, remove the registered remotes:

```
remotes: {
    // Remove this line or comment it out:
    // "mfe1": "mfe1@http://localhost:3000/remoteEntry.js",
},
```

2. Open the file `app.routes.ts` and use the function `loadRemoteModule` instead of the dynamic `import` statement:

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

[...]
const routes: Routes = [
    [...]
    {
        path: 'flights',
        loadChildren: () =>
            loadRemoteModule({
                remoteEntry: 'http://localhost:3000/remoteEntry.js',
                remoteName: 'mfe1',
                exposedModule: './Module'
            })
            .then(m => m.FlightsModule)
    },
    [...]
]
```

3. You may need to restart both, the `shell` and the micro frontend (`mfe1`).

4. The shell should still be able to load the micro frontend. However, now it's loaded dynamically.

This was quite easy, wasn't it? However, we can improve this solution a bit. Ideally, we load the remote entry upfront before Angular bootstraps. In this early phase, Module Federation tries to determine the highest compatible versions of all dependencies. Let's assume, the shell provides version 1.0.0 of a dependency (specifying ^1.0.0 in its `package.json`) and the micro frontend uses version 1.1.0 (specifying ^1.1.0 in its `package.json`). In this case, they would go with version 1.1.0. However, this is only possible if the remote's entry is loaded upfront.

1. Switch to the `shell` project and open the file `main.ts` . Adjust it as follows:

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

Promise.all([
    loadRemoteEntry('http://localhost:3000/remoteEntry.js', 'mfe1')
])
.catch(err => console.error('Error loading remote entries', err))
.then(() => import('./bootstrap'))
.catch(err => console.error(err));
```

2. Open the file `app.routes.ts` and comment out (or remove) the property `remoteEntry` :

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

[...]
const routes: Routes = [
    [...]
    {
        path: 'flights',
        loadChildren: () =>
            loadRemoteModule({
                // remoteEntry: 'http://localhost:3000/remoteEntry.js',
                remoteName: 'mfe1',
                exposedModule: './Module'
            })
            .then(m => m.FlightsModule)
    },
    [...]
]
```

3. You may need to restart both, the `shell` and the micro frontend (`mfe1`).

4. The shell should still be able to load the micro frontend.

Bonus: Share a Library of Your Monorepo

1. Add a library to your monorepo:

```
ng g lib auth-lib
```

2. In your `tsconfig.json` in the project's root, adjust the path mapping for `auth-lib` so that it points to the libs entry point:

```
"auth-lib": [
 "projects/auth-lib/src/public-api.ts"
]
```

3. As most IDEs only read global configuration files like the `tsconfig.json` once, restart your IDE (Alternatively, your IDE might also provide an option for reloading these settings).

4. Open the `shell`'s `webpack.config.js` and register the created `auth-lib` with the `sharedMappings`:

```
const sharedMappings = new mf.SharedMappings();
sharedMappings.register(
    path.join(__dirname, '../../tsconfig.json'),
    ['auth-lib'] // <-- Add this entry!
);
```

5. Also open the micro frontends (`mfe1`) `webpack.config.js` and do the same.

6. Switch to your `auth-lib` project and open the file `auth-lib.service.ts`. Adjust it as follows:

```
@Injectable({
    providedIn: 'root'
})
export class AuthLibService {<pre><code>private userName: string;

public get user(): string {
    return this.userName;
}

constructor() { }

public login(userName: string, password: string): void {
    // Authentication for **honest** users TM. (c) Manfred Steyer
    this.userName = userName;
}</code></pre>}
```

7. Switch to your `shell` project and open its `app.component.ts`. Use the shared `AuthLibService` to login a user:

```
import { AuthLibService } from 'auth-lib';

@Component({
    selector: 'app-root',
    templateUrl: './app.component.html'
})
export class AppComponent {
    title = 'shell';<pre><code>constructor(private authLibService: AuthLibService) {
```

```
    this.authLibService.login('Laura', null);
}</code></pre>}
```

8. Switch to your `mfe1` project and open its `flights-search.component.ts`. Use the shared service to retrieve the current user's name:

```
export class FlightsSearchComponent {<pre><code>[...]

user = this.authLibService.user;

constructor(private authLibService: AuthLibService, [...]) { }

[...]</code></pre>}
```

9. Open this component's template(`flights-search.component.html`) and data bind the property `user` :

```
<div id="container">
    <div>{{user}}</div>
    [...]
</div>
```

10. You may need to restart both, the `shell` and the micro frontend (`mfe1`).

11. In the shell, navigate to the micro frontend. If it shows the same user name, the library is shared.

More Details on Module Federation

Have a look at this article series about Module Federation