Using Module Federation with (Nx) Monorepos and Angular

The combination of Micro Frontends and monorepos can be quite tempting: Sharing libraries is easy while we can isolate domains using access restrictions and deploy them...

26.11.2020 | share

Dies ist Beitrag 6 von 8 der Serie *"Module Federation"*

> Updated on 2021-08-08 for Angular and Nx 12.x and upwards

> Updated on 2021-12-23 for Angular 13.1 or higher and Nx 13.4 or higher

The combination of Micro Frontends and monorepos can be quite tempting: Monorepos make it easy to share libraries. Thanks to access restrictions, individual business domains can be isolated. Also, having all micro frontends in the same monorepo doesn't prevent us from deploying them separately.

will see, Nx provides some really powerful features that make your life easier, e. g. the possibility of generating a visual dependency graph or finding out which applications have been changed and hence r

redeployed.

If you want to have a look at the source code used here, you can check out this repository.

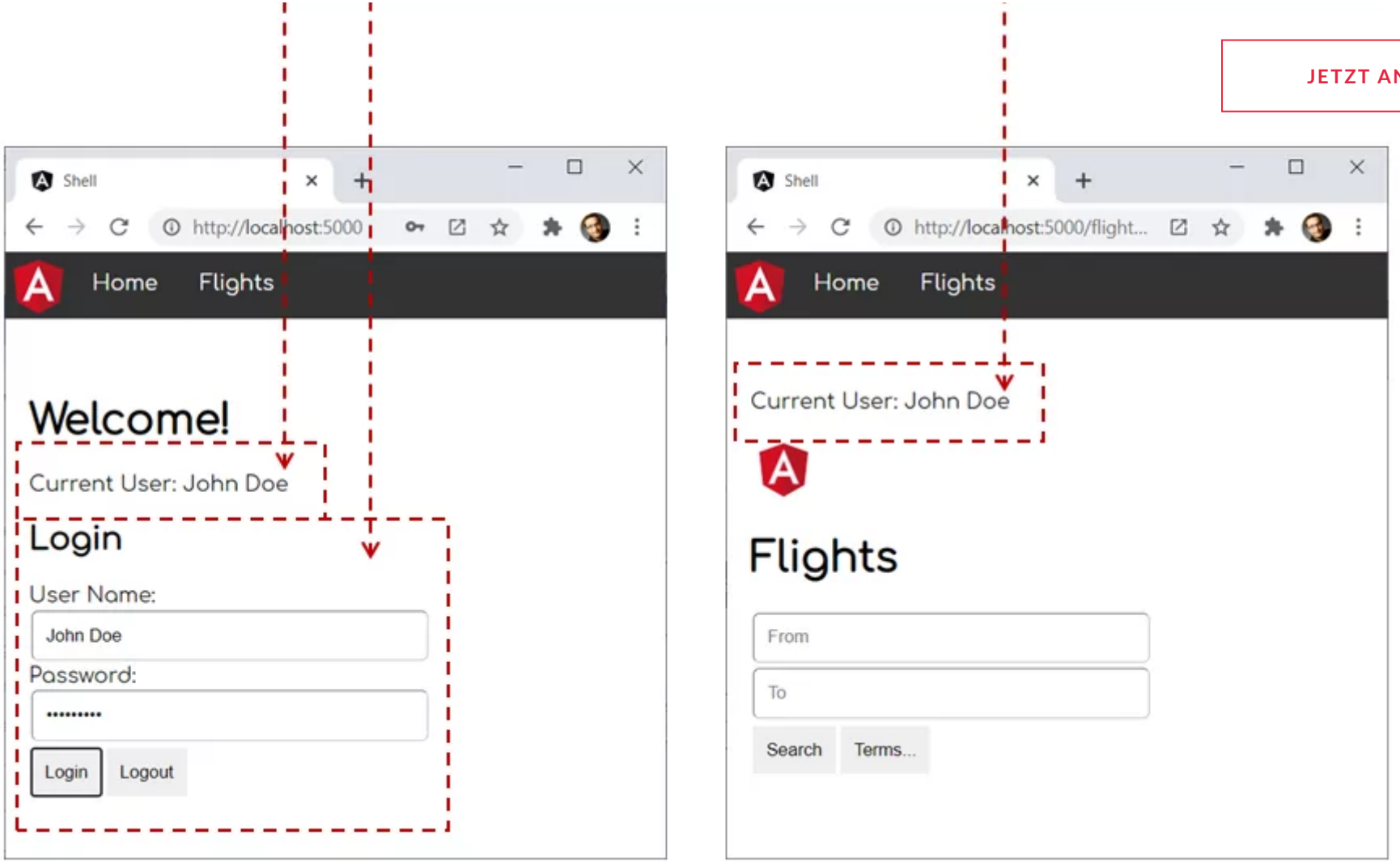
> Big thanks to the awesome Tobias Koppers who gave me valuable insights into this topic and to the one and only Dmitriy Shekhovtsov who helped me using the Angular CLI/webpack 5 integration for this.

Important: This article is written for **Angular 13.1** and higher. Hence you also need **Nx 13.3.12** or higher. To find out about the small differences for lower versions of Angular and for the migration from such a lower version, please have a look to our migration guide.

Example

The example used here is a Nx monorepo with a micro frontend shell (`shell`) and a micro frontend (`mfe1` , "micro frontend 1"). Both share a common library for authentication (`auth-lib`) that is also located in the monorepo:

The auth-lib provides two components. One is logging-in users and the other one displays the current user. They are used by both, the shell and mfe1 :



Also, the auth-lib stores the current user's name in a service.

```
"paths": {
  "@demo/auth-lib": [
    "libs/auth-lib/src/index.ts"
  ]
},
```


The `shell` and `mfe1` (as well as further micro frontends we might add in the future) need to be deployable in separation and loaded at runtime. However, we don't want to load the `auth-lib` twice or several times! Archiving this with an npm package is not that difficult. This is one of the most obvious and easy to use features of Module Federation. The next sections discuss how to do the same with libraries of a monorepo.

The Shared Lib

Before we delve into the solution, let's have a look at the `auth-lib`. It contains an `AuthService` that logs-in the user and remebers them using the property `_userName`:

```
});
export class AuthService {

 // tslint:disable-next-line: variable-name
 private _userName: string = null;

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

 constructor() { }

 login(userName: string, password: string): void {
  this._userName = userName;
 }

 logout(): void {
  this._userName = null;
 }
}
```

displaying the current user's name. Both components are registered with the library's NgModule:


```
@NgModule({
 imports: [
  CommonModule,
  FormsModule
 ],
 declarations: [
  AuthComponent,
  UserComponent
 ],
 exports: [
  AuthComponent,
  UserComponent
 ],
})
export class AuthLibModule {}
```

As every library, it also has a barrel `index.ts` (sometimes also called `public-api.ts`) serving as the entry point. It exports everything consumers can use:

```
4  // Don't forget about your components!
5  export * from './lib/auth/auth.component';
6  export * from './lib/user/user.component';
```


Please note that `index.ts` is also exporting the two components although they are already registered with the also exported `AuthLibModule`. In the scenario discussed here, this is vital in order to make sure it's detected and compiled by Ivy.

The Module Federation Configuration

As in the previous article, we are using the `@angular-architects/module-federation` plugin to enable Module Federation for the `shell` and `mfe1`. For this, just run this command twice and answer the plugin's questions:

```
1  ng add @angular-architects/module-federation
```

This generates a webpack config for Module Federation. The latest version of the plugin uses a helper class called `SharedMapping`:

```
const path = require('path');
const share = mf.share;

const sharedMappings = new mf.SharedMappings();
sharedMappings.register(
 path.join(__dirname, '../../tsconfig.base.json'),
 ['@demo/auth-lib']);

module.exports = {
 output: {
  uniqueName: "mfe1",
  publicPath: "auto"
 },
 optimization: {
  runtimeChunk: false
 },
 resolve: {
  alias: {
   ...sharedMappings.getAliases(),
  }
 },
 experiments: {
  outputModule: true
```

```
29
30      library: { type: "module" },
31
32      name: "mfe1",
33      filename: "remoteEntry.js", // <-- Metadata
34      exposes: {
35        './Module': './apps/mfe1/src/app/flights/flights.module.ts',
36      },
37
38      shared: share({
39        "@angular/core": { singleton: true, strictVersion: true, requiredVersion: 'auto' },
40        "@angular/common": { singleton: true, strictVersion: true, requiredVersion: 'auto' },
41        "@angular/common/http": { singleton: true, strictVersion: true, requiredVersion: 'auto' },
42        "@angular/router": { singleton: true, strictVersion: true, requiredVersion: 'auto' },
43
44        ...sharedMappings.getDescriptors()
45      }),
46
47    }),
48    sharedMappings.getPlugin()
49  ],
50 };
```

way. **Each application needs to opt-in for sharing a library**.

For the time being, this is all we need to know. In a section below, I'm going to explain what [illegible] is doing and why it's a good idea to hide these details in such a convenience class.

Trying it out

To try this out, just start the two applications:

```
1 ng serve shell -o
2 ng serve mfe1 -o
```

Then, log-in in the shell and make it to load `mfe1`. If you see the logged-in user name in `mfe1`, you have the proof that `auth-lib` is only loaded once and shared across the applications.

Deploying

changed or affected by a change. For this, just run the affected:apps script:


```
npm run affected:apps
```

You might also want to detect the changed applications as part of your CI pipeline. To make implementing such an automation script easier, leverage the Nx CLI (`npm i -g @nrwl/cli`) and call the `affected:apps` script with the `--plain` switch:

```
nx affected:apps --plain
```

This switch makes the CLI to print out all affected apps in one line of the console separated by a space. Hence, this result can be easily parsed by your scripts.

Also, as the micro frontends loaded into the shell don't know each other upfront but only meet at runtime, it's a good idea to rely on some e2e tests.

What Happens Behind the Covers?

Its method `getDescriptors` returns the needed entries for the `shared` section in the configuration:

```
"@demo/auth-lib": {
  import: path.resolve(__dirname, "../../libs/auth-lib/src/index.ts"),
  requiredVersion: false
},
```

These entries look a bit different than the ones we are used to. Instead of pointing to an npm package to share, it directly points to the library's entry point using its `import` property. The right path is taken from the mappings in the `tsconfig.json`.

Normally, Module Federation knows which version of a shared library is needed because it looks into the project's `package.json`. However, in the case of a monorepo, shared libraries (very often) don't have a version. Hence, `requiredVersion` is set to `false`. This makes Module Federation accepting an existing shared instance of this library without drawing its very version into consideration.

However, to make this work, we should always redeploy all changed parts of our overall system together, as proposed above.

with a relative path:


```
1 | import { UserComponent } from '../../libs/auth-lib/src/lib/user/user.component.ts';
```

However, to make Module Federation only sharing one version of our lib(s), we need to import them using a consistent name like `@demo/auth-lib`:

```
1 | import { UserComponent } from '@demo/auth-lib';
```

For this, `SharedMappings` provides a method called `getPlugin` returning a configured `NormalModuleReplacementPlugin` instance that takes care of rewriting such imports. The key data needed here is take from `tsconfig.json`.

Bonus: Versions in the Monorepo

As stated before, normally libraries don't have a version in a monorepo. The consequence is that we need to redeploy all the changed application together. This makes sure, all applications can work with the shared libs.

by its consumers are defined in the respective package.json files.

In our monorepo, we don't have either, hence we need to pass this information directly to M

If we didn't use the `SharedMappings` convenience class, we needed to place such an object into the config's shared section:

```
"auth-lib": {
  import: path.resolve(__dirname, "../../libs/auth-lib/src/index.ts"),
  version: '1.0.0',
  requiredVersion: '^1.0.0'
},
```

Here, `version` is the actual library version while `requiredVersion` is the version (range) the consumer (micro frontend or shell) accepts. To prevent repeating the library version in all the configurations for all the micro frontends, we could centralize it. Perhaps, you want to create a `package.json` in the library's folder:

```
{
 "name": "@demo/auth-lib",
 "version": "1.0.0",
}
```

(range) using `requiredVersion`.


```
"@demo/auth-lib": {
  import: path.resolve(__dirname, "../../libs/auth-lib/src/index.ts"),
  version: require('relative_path_to_lib/package.json').version,
  requiredVersion: '^1.0.0'
},
```

Because of this, we don't need to redeploy all the changed applications together anymore. Using the provided versions, Module Federation decides at runtime which micro frontends can safely share the same instance of a library and which micro frontends need to fall back to another version (e. g. the own one).

The drawback of this approach is that we need to rely on the authors of the libraries and the micro frontends to manage this meta data correctly. Also, if two micro frontends need two non-compatible versions of a shared library, the library is loaded twice. This is not only bad for performance but also leads to an issue in the case of stateful libraries as the state is duplicated. In our case, both `auth-lib` instances could store their own user name.

(range).


```
new ModuleFederationPlugin({
 [...]
 shared: {
  [...]
  ...sharedMappings.getDescriptor('@demo/auth-lib', '^1.0.0')
 }
}),
```

The rest of the configuration would be as discussed above.

Pitfalls

When using this approach, you might encounter some pitfalls. They are due to the fact that the CLI/webpack5 integration is still experimental in Angular 11 but also because we treat a folder with uncompiled typescript files like an npm package.

If you shared a local library that is not even used, you get the following error:

```
1 ./projects/shared-lib/src/public-api.ts - Error: Module build failed (from ./node_modules/@ngtools/webpack/src/
2 Error: C:\Users\Manfred\Documents\projekte\mf-plugin\example\projects\shared-lib\src\public-api.ts is missin
3   at AngularCompilerPlugin.getCompiledFile (C:\Users\Manfred\Documents\projekte\mf-plugin\example\node_
4   at C:\Users\Manfred\Documents\projekte\mf-plugin\example\node_modules\@ngtools\webpack\src\loader.js
```

Not exported Components

If you use a shared component without exporting it via your library's barrel (`index.ts` or `public-api.ts`), you get the following error at runtime:

```
1 core.js:4610 ERROR Error: Uncaught (in promise): TypeError: Cannot read property 'ɵcmp' of undefined
2 TypeError: Cannot read property 'ɵcmp' of undefined
3   at getComponentDef (core.js:1821)
```

Add this to the output section of your webpack config:

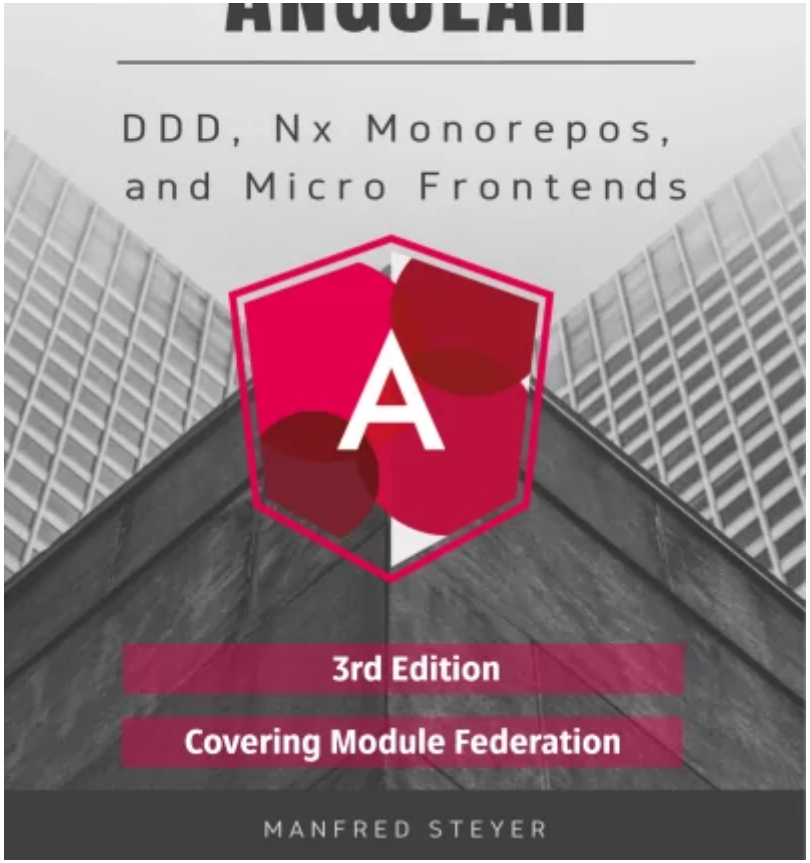
```
chunkFilename: '[name]-[contenthash].js',
```

What's next? More on Architecture!

So far, we've seen that Module Federation is a strightforward solution for creating Micro Frontends on top of Angular. However, when dealing with it, several additional questions come in mind:

- According to which criteria can we sub-divide a huge application into micro frontends?
- Which access restrictions make sense?
- Which proven patterns should we use?
- How can we avoid pitfalls when working with Module Federation?
- Which advanced scenarios are possible?

Our free eBook (about 100 pages) covers all these questions and more:



Feel free to download it here now!

Feel free to share it on social media
and subscribe to our newsletter

Don't Miss Anything!

Subscribe to our newsletter to get all the information about Angular.

Business EMail Address*:

Country*

Subscribe*

Unsere Angular-Schulungen

Hot!

Angular Schulung: Strukturierte Einführung

In dieser strukturierten Einführung lernen Einsteiger und Autodidakten alle Building-Blocks…

MEHR INFORMATIONEN

Hot!

Angular Architektur Workshop

In diesem weiterführenden Intensiv-Kurs lernen Sie, wie sich große und…

MEHR INFORMATIONEN

Micro Frontends mit

Lernen Sie große Angular-Lös
cro Frontends zu strukt

MEHR INFORMATIO

Aktuelle Blog-Artikel

ALLE ARTIKEL

VON: MANFRED STEYER, GDE

4 WAYS TO PREPARE FOR ANGULAR'S UPCOMING STANDALONE COMPONENTS

With the introduction of Standalone Components, NgModules will become optional. But how to prepare for…

MEHR ERFAHREN

VON: MANFRED STEYER, GD

STATE-MANAGEMENT MIT

In diesem Tutorial lernen Sie alles über NGRX in Ang
lektoren und Effects…

MEHR ERFAHREN

Nur einen Schritt entfernt!

Manfred Steyer

ist Trainer und Berater mit Fokus auf Angular. Er hat berufsbegleitend IT und IT-Marketing in Graz sowie ebenfalls berufsbegleitend Computer Science in Hagen studiert und eine vier-semestrige Ausbildung im Bereich der Erwachsenenbildung abgeschlossen.

JETZT ANFRAGEN

Manfred Steyer

Ludersdorf 219
8200 Gleisdorf
Österreich

manfred.steyer@softwarearchitekt.at

NEWSLETTER ABONNIEREN

Nichts mehr verpassen!

Hiermit erkläre ich mich damit einverstanden, dass der Betreiber dieser Seite meine E-Mail-Adresse zum Zwecke des Versands des Newsletters verarbeiten kann.
DATENSCHUTZ.

Email: