



**ANGULAR**

## Micro Frontends with Angular, Module Federation, and Auth0

Module Federation allows loading Micro Frontends at runtime. Common dependencies like Angular or the Auth0 library can be shared and hence don't need to be loaded several times. This is also the key for sharing data like the current user or global filters.

**Manfred Steyer**

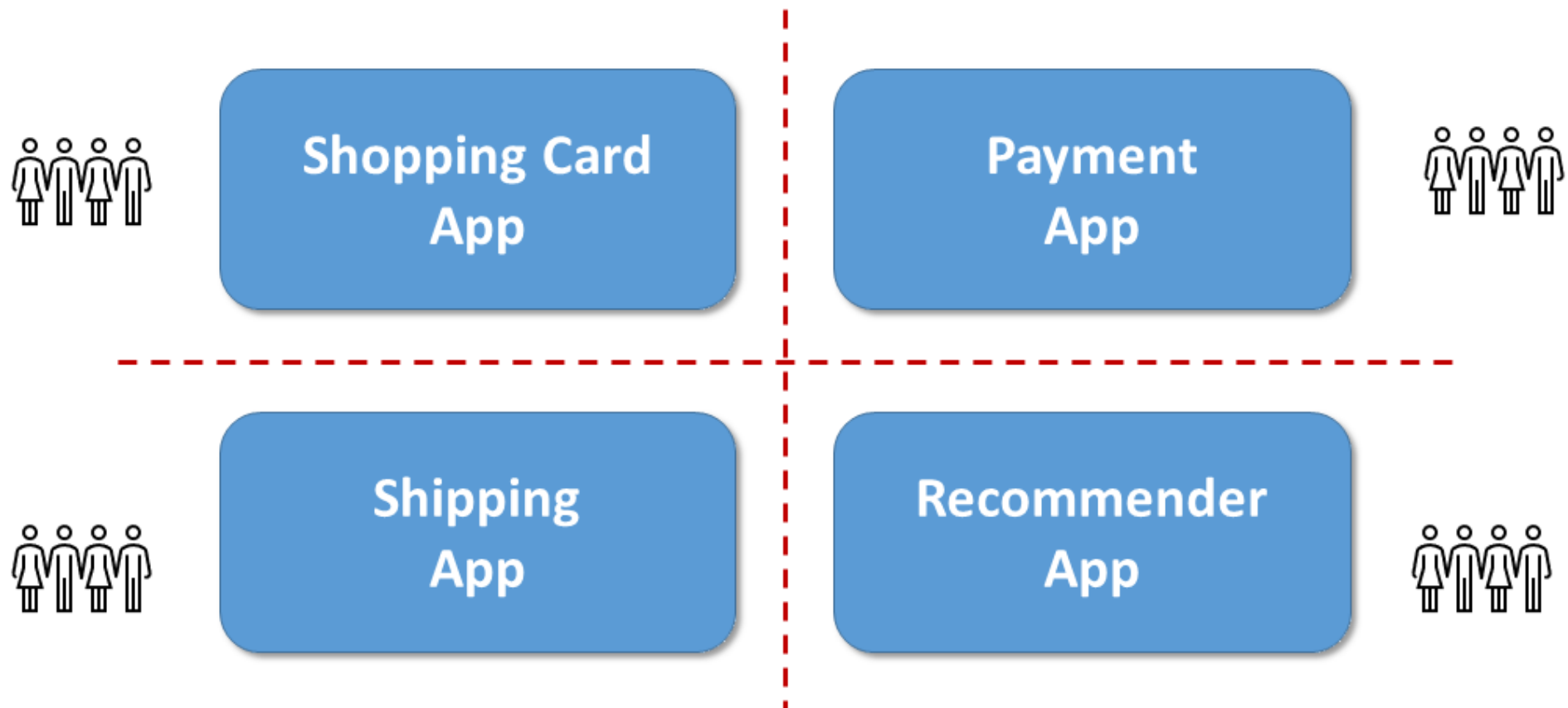
Trainer and Consultant with focus on Angular

Last Updated On: February 02, 2022

**TL;DR:** Module Federation allows loading Micro Frontends at runtime. Common dependencies like Angular or the Auth0 library can be shared and hence don't need to be loaded several times. This is also the key for sharing data like the current user or global filters.

## Micro Frontends

Recently, there have been a lot of discussions about Micro Frontends. The underlying idea is quite tempting: Splitting a huge software system into smaller parts and assigning each part to an individual team:



As these teams are -- more or less -- autonomous, they can work in a more flexible way and don't need to coordinate that much with others. This shows that Micro Frontends are mostly about scaling teams. Hence, if you don't have several frontend teams, Micro Frontends might be overkill.

## Module Federation

Splitting software systems into several parts is, however, only one side of the coin. Even though we developers might like this idea, the users don't! They aren't interested in starting several frontends but expect an integrated solution.

Accordingly, we need to find a way to make all our micro frontends appear as one (single page) application. Very often, this involves loading micro frontends into a shell application.

For instance, the example used here mimics a simple shopping system. One of its micro frontends deals with shipping:

**Mfe1**

http://localhost:3000/address

## Shipping Information

Company

First name Last name

Address

+ Add C/O, Apt, Suite, Unit

City State

Postal Code

0 / 5

☒ Free Shipping  
☐ Priority Shipping  
☐ Next Day Shipping

A blue rectangular button with rounded corners and a subtle drop shadow, containing the word "Submit" in white, bold, sans-serif font.

For the purpose of integration, this micro frontend needs to be loaded into a shell:

Shell

http://localhost:5000/mfe1/address

## Menu

- Home
- Shopping
- Payment
- Shipping**
- Recommender

## Awesome Shell

### Shipping Information

Company

First name Last name

Address

+ Add C/O, Apt, Suite, Unit

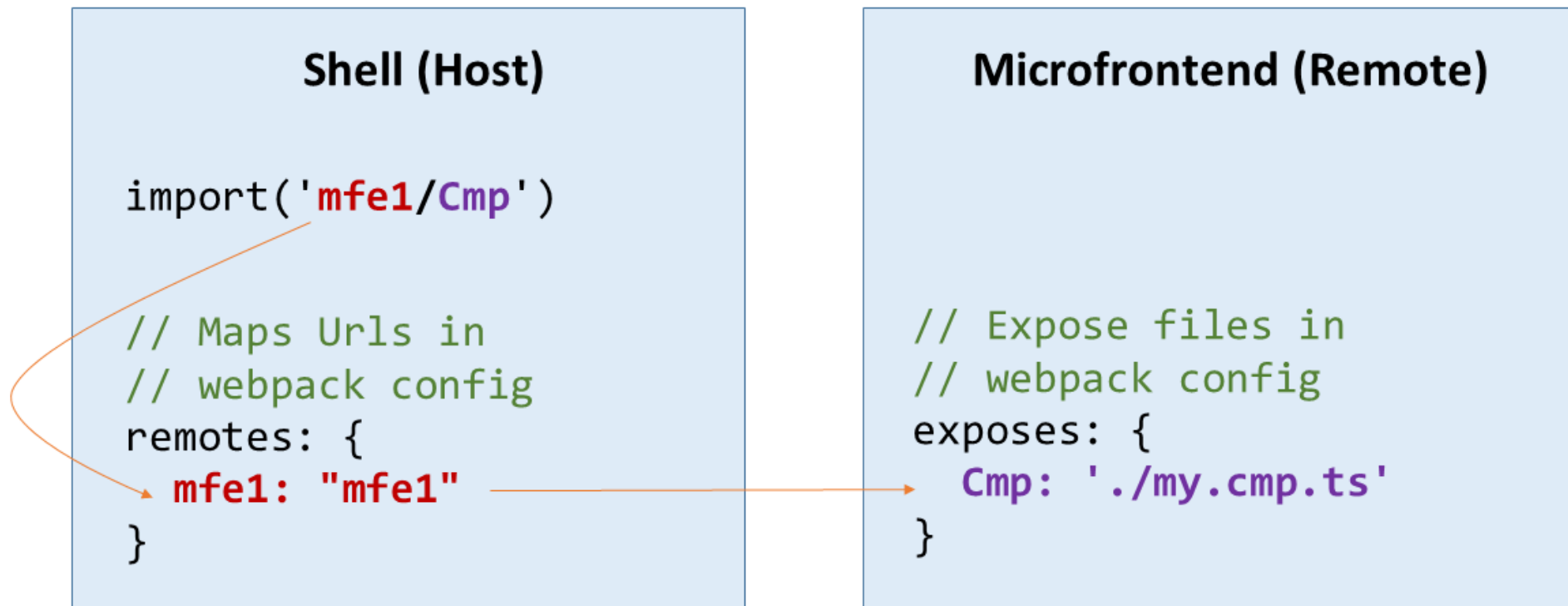
City State

The shell loads this but also other micro frontends on demand.

For such an implementation, you needed a lot of tricks in the past. Fortunately, Module Federation makes this task straightforward.

Module Federation is an integrated part of webpack 5, and hence, it works with all modern web frameworks. Also, as webpack is highly used in nearly all communities, Module Federation immediately becomes a widespread solution.

In order to allow loading separately compiled and deployed micro frontends, Module Federation defines two roles: the host and the remote:



In our case, the host is the shell, while the remote is the micro frontend. Both can be configured with their webpack configurations.

The host defines virtual URLs pointing to remotes. In the previous picture, for instance, the URL `mfe1` pointing to a micro frontend, also called `mfe1`, is defined.



The remote, in turn, exposes files -- or, to be more precise: EcmaScript modules -- for the host. To prevent that, the latter one needs to know the remote's file structure; exposed files can get a shorter alias. The example in the last picture goes with the alias `Cmp`.

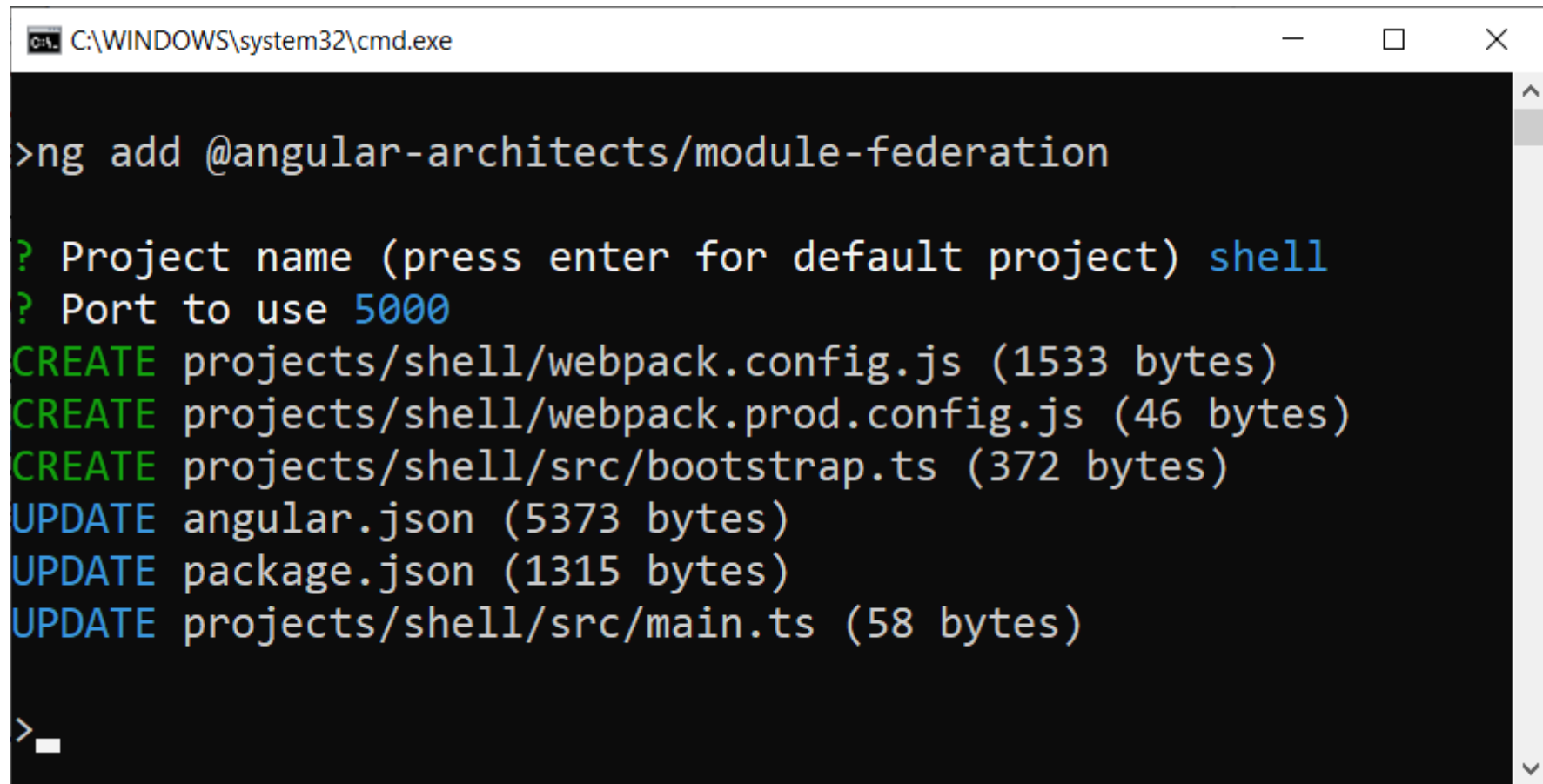
Exposed EcmaScript modules can easily be loaded into the host by using a dynamic `import` together with the mapped URLs and defined aliases. The best: From Angular's perspective, this is just like lazy loading.

As Angular doesn't even recognize that the application loads a micro frontend, we can use it exactly as it was intended to be used. We don't need any tricks or meta frameworks orchestrating different SPAs in our browser window. This eases our endeavor dramatically.

## Module Federation in Angular

Since version 12, the Angular CLI uses webpack 5. Hence, we also get Module Federation out of the box. To activate it, we need a custom builder that, e. g. ships with the community solution `@angular-architects/module-federation`. As it comes with respective schematics, you can easily `ng add` it to your CLI workspace:

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



```
C:\WINDOWS\system32\cmd.exe

>ng add @angular-architects/module-federation

? Project name (press enter for default project) shell
? Port to use 5000
CREATE projects/shell/webpack.config.js (1533 bytes)
CREATE projects/shell/webpack.prod.config.js (46 bytes)
CREATE projects/shell/src/bootstrap.ts (372 bytes)
UPDATE angular.json (5373 bytes)
UPDATE package.json (1315 bytes)
UPDATE projects/shell/src/main.ts (58 bytes)

>
```

The well-known CLI extension Nx is supported too. The schematic asks about the name of the project in your workspace you want to enable module federation for. Also, it asks which port to assign for `ng serve`.

After that, it generates several files, e. g. a partial webpack configuration we need to adjust for our needs. Also, it registers the custom builder in our `angular.json`.

For the example used here, I ran this command twice: Once for the shell and once for the above-shown shipping micro frontend. The latter one is just called `mfe1` here.

# Configuring the Micro Frontends

Now, we need to adjust the webpack configurations generated for the shell and the micro frontend. As the Angular CLI is generating most parts of the webpack configuration, we just need to define the remaining details for the module federation. First and foremost, this is about configuring the `ModuleFederationPlugin` that ships with webpack.

Let's start with the configuration for the micro frontend:

```
// projects/mfe1/webpack.config.js

[...]  
plugins: [  
  
  new ModuleFederationPlugin({  
  
    // For remotes (please adjust)  
    name: "mfe1",  
    filename: "remoteEntry.js",  
    exposes: {  
      './AddressModule': './projects/mfe1/src/app/address/address.module.ts',  
    },  
  
    shared: share({  
      "@angular/core": { singleton: true, strictVersion: true, requiredVersion: 'auto' },  
      "@angular/common": { singleton: true, strictVersion: true, requiredVersion: 'auto' },  
    })  
  })  
]
```

```
"@angular/common/http": { singleton: true, strictVersion: true, requiredVersion: 'auto' },
"@angular/router": { singleton: true, strictVersion: true, requiredVersion: 'auto' },
"@angular/material": { singleton: true, strictVersion: true, requiredVersion: 'auto', includeSecondaries: true },

...sharedMappings.getDescriptors()
})

}),
],
[...]
```

The property `name` is normally the project name and should be unique across all micro frontends. The `filename` points to a file webpack generates when building the project. It's often referred to as the remote entry point. This JavaScript file provides all the metadata the shell needs to know for loading the micro frontend.

While normally you can go with the generated values for `name` and `filename`, you very likely need to adjust the `exposes` section. As discussed above, it contains the aliases for the files to expose.

With `shared`, you define all the npm packages you want to share across the shell and the micro frontends. These packages are only loaded once at runtime. This is vital because having ten micro frontends doesn't mean you want to load Angular or other packages ten times!

Of course, when sharing packages at runtime, we might end up with version conflicts. Fortunately, Module Federation also got us covered here: It provides several strategies for dealing with version mismatches.

In our case, the combination of `singleton: true` and `strictVersion: true` is used. Hence, only one version of the shared packages is allowed, and if Module Federation detects several versions at runtime, it throws an Error. While this seems to be quite harsh at first sight, it allows our integration tests to immediately find out about version mismatches.

More details on [strategies for preventing version mismatches](#) can be found [here](#).

Besides `singleton` and `strictVersions`, there are two further properties used: `requiredVersion` and `includeSecondaries`. If we set the first one to `auto`, `@angular-architects/module-federation` assumes the version found in your project's `package.json`. This avoids several [pitfalls](#).

The option `includeSecondaries` is provided by `@angular-architects/module-federation` too. It automatically adds all secondary entry points to the list of shared libraries. In the case of libraries like Angular Material, this saves you a lot of typing as they provide one such entry point per component, e. g. `@angular/material/input` or `@angular/material/button`.

Now, let's have a look at the exposed `AddressModule`:

```
// projects/mfe1/src/app/address/address.module.ts

[...]
```

```
@NgModule({
  imports: [
    [...]
```

```
    RouterModule.forChild([
```



```
    path: 'address',
    component: AddressComponent
  ]])
],
declarations: [
  AddressComponent
],
})
export class AddressModule { }
```

As you see, it's just an ordinary Angular module. However, as it has child routes, we can immediately route to them after loading the whole module into the shell.

## Configuring the Shell

The shell's `webpack.config.js` is a bit simpler. Here, we only need to define the shared libraries:

```
// projects/shell/webpack.config.js

[...]
```

```
plugins: [
  new ModuleFederationPlugin({

    shared: share({
      "@angular/core": { singleton: true, strictVersion: true, requiredVersion: 'auto' },
```

```
"@angular/common": { singleton: true, strictVersion: true, requiredVersion: 'auto' },
"@angular/common/http": { singleton: true, strictVersion: true, requiredVersion: 'auto' },
"@angular/router": { singleton: true, strictVersion: true, requiredVersion: 'auto' },
"@angular/material": { singleton: true, strictVersion: true, requiredVersion: 'auto', includeSecondaries: true }

...sharedMappings.getDescriptors()
})

}),
],
[...]
```

In addition, we also could map URLs here, as outlined above. This would look like this:

```
// Alternative that maps URLs upfront
new ModuleFederationPlugin({
  remotes: {
    "mfe1": "mfe1@http://localhost:3000/remoteEntry.js"
  },
  [...]
})
```

The key is the mapped URL, and the value defines the name of the micro frontend as defined in its `webpack.config.js`. Here, the value also contains the location of the micro frontend's remote entry point. To load `mfe1`'s exposed Module, we could use a dynamic `import`:

```
// Using mapped URL:  
import('mfe1/AddressModule')
```

As the TypeScript compiler doesn't know the file `mfe1/Module`, we also needed a type definition. For this, we could create an arbitrary with the ending `.d.ts`:

```
declare module "mfe1/AddressModule";
```

However, while mapping URLs this way is simple, it also requires us to know about the micro frontends and their locations upfront. Hence, I'm using a more dynamic approach that does not demand us to define URLs for compile time.

Instead, we can tell Module Federation about possible Micro Frontends at runtime. To make use of it, I use the helper function `loadRemoteModule` provided by `@angular-architects/module-federation` together with lazy routes:

```
// projects/shell/src/app/app.module.ts  
  
[...]  
RouterModule.forRoot([  
  {  
    path: '',  
    component: HomeComponent  
  },  
  {  
    path: 'mfe1',
```

```
loadChildren: () => loadRemoteModule({
  remoteEntry: 'http://localhost:3000/remoteEntry.js',
  remoteName: 'mfe1',
  exposedModule: './AddressModule',
}).then(m => m.AddressModule)
}
])
[...]
```

The `loadRemoteModule` function takes the location of the remote entry point, the remote's name, and the alias of the exposed EcmaScript module. All these values are defined in the micro frontend's webpack configuration.

As these properties are just strings, we could even retrieve their values from a configuration file or from a service endpoint.

## Trying It Out

After starting both projects, e. g. with `ng serve shell`, and `ng serve mfe1`, we can use the lazy route shown above to load the micro frontend into the shell. It's also interesting to inspect the loaded files in your browser's dev tools:

The screenshot shows a web browser at `http://localhost:5000/mfe1/address` displaying a page titled "Awesome Shell" with a "Menu" and "Home" sidebar, and a "Shipping Information" section. Below the browser, the Chrome DevTools Network tab is open, showing a list of network requests. The "All" filter is selected, and the "3rd-party requests" checkbox is unchecked. The network requests are listed in a table with columns: Name, Status, Type, Initiator, Size, Time, and Waterfall.

Name	Status	Type	Initiator	Size	Ti...	Waterfall
remoteEntry.js	200	script	angular-architects-module-f...	34.1 kB	3...	
node_modules/angular_material__ivy_ngcc__fesm2015_form-field.js-5...	200	script	load script:41	608 kB	9 ...	
node_modules/angular_material__ivy_ngcc__fesm2015_input.js-1e991...	200	script	load script:41	493 kB	1...	
node_modules/angular_material__ivy_ngcc__fesm2015_select.js-044a...	200	script	load script:41	1.1 MB	2...	
node_modules/angular_material__ivy_ngcc__fesm2015_radio.js-3a021...	200	script	load script:41	1.0 MB	3...	
node_modules/angular_material__ivy_ngcc__fesm2015_card.js-14211.js	200	script	load script:41	388 kB	3...	
projects_mfe1_src_app_address_address_module_ts.js	200	script	load script:41	409 kB	4...	

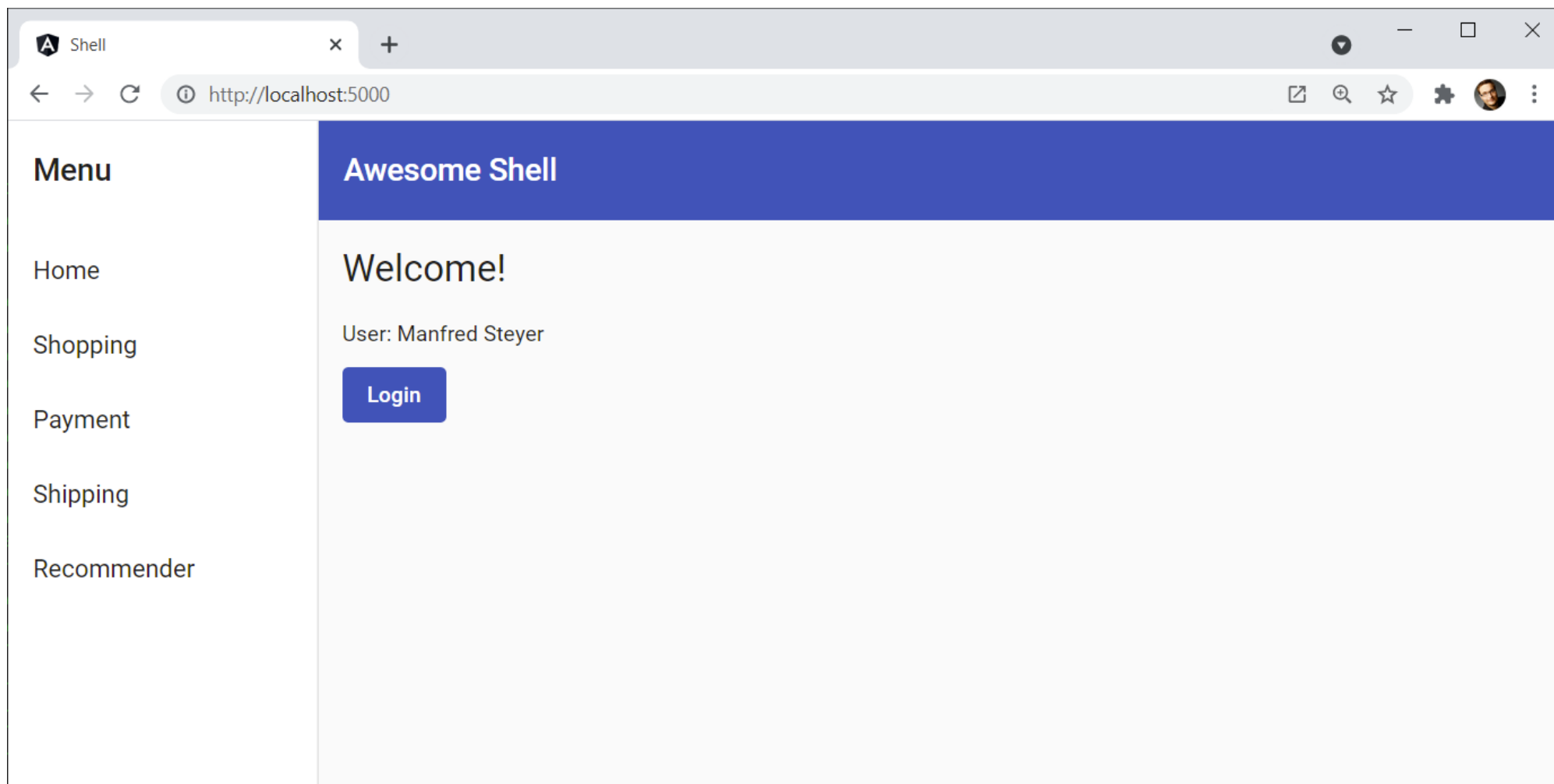
At the bottom of the network tab, it shows: 7 requests | 4.1 MB transferred | 4.0 MB resources.

At first sight, it really looks like lazy loading. However, when looking more closely, we see that the lazy chunks come from the micro frontend's origin. Also, for all the shared libraries, a bundle of its own is loaded. This is necessary because the module federation decides at runtime when to load which shared package.

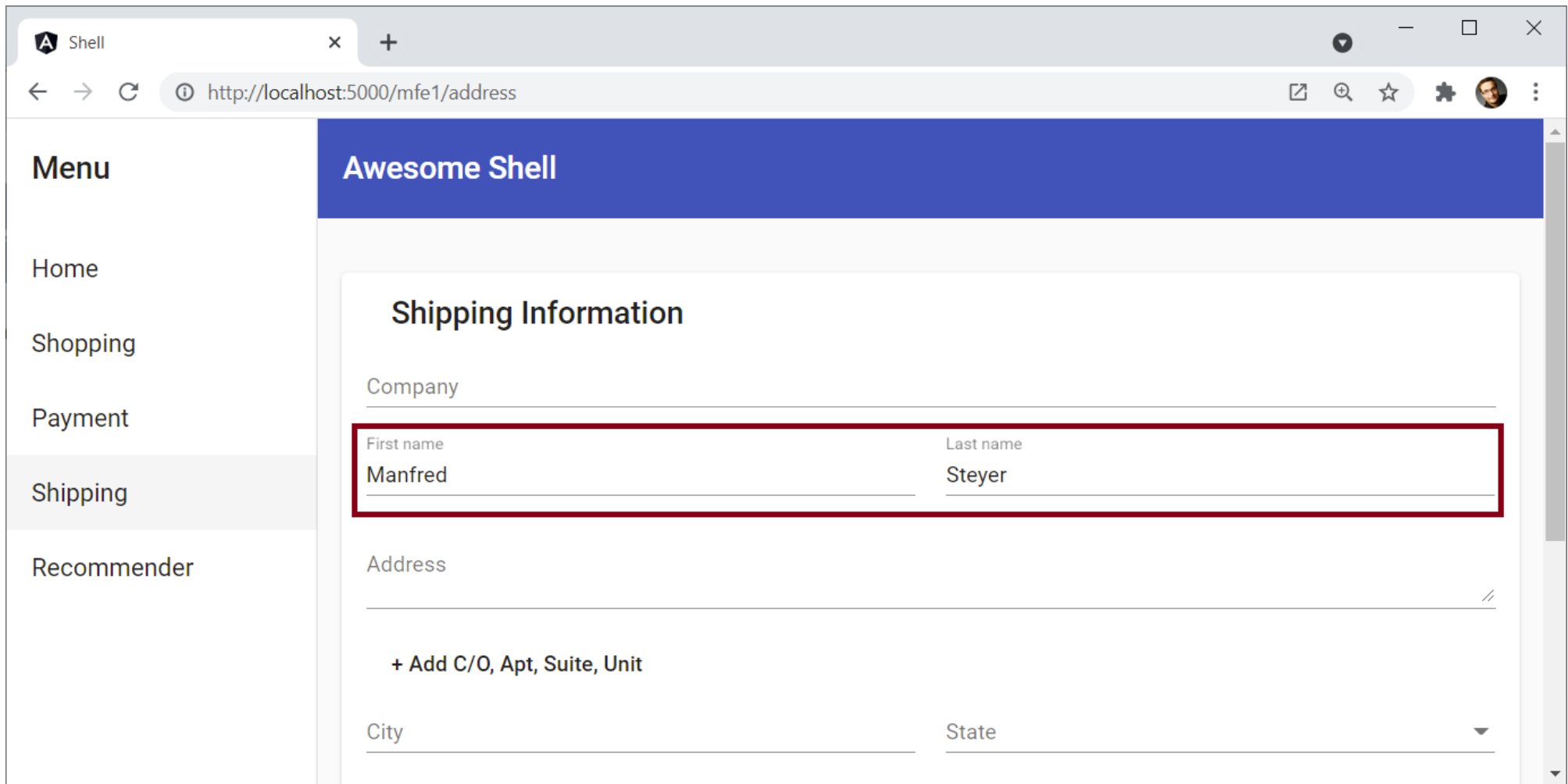


# Authentication for Micro Frontends

Sometimes we need to share some information between micro frontends and the shell, e. g. context information like the current user or global filters like the selected customer. The example used here allows logging in the user via the shell:



However, when loading the micro frontend on demand, it also knows about the current user:



The screenshot shows a web browser window with the address `http://localhost:5000/mfe1/address`. The application has a sidebar menu with options: Menu, Home, Shopping, Payment, Shipping (selected), and Recommender. The main content area is titled 'Awesome Shell' and displays a 'Shipping Information' form. The form fields are as follows:

- Company:
- First name:  (highlighted with a red border)
- Last name:  (highlighted with a red border)
- Address:
- + Add C/O, Apt, Suite, Unit:
- City:
- State:

To achieve this, we just need to share a package that holds the information to share. In our case, this is `@auth0/auth0-angular` as we are using Auth0 as our identity provider. It's available via npm:

```
npm i @auth0/auth0-angular
```

To make this work, I've configured a web application at <https://manage.auth0.com/>. If you want to try this out, don't forget to register your SPA's URL, e. g. <http://localhost:4200>, under *Allowed Callback URLs*. Besides this, I went with the default values proposed by the web portal.

To get started, we need to import the `AuthModule` provided by `@auth0/auth0-angular` into the `shell`'s but also into `mfe1`'s `AppModule`:

```
// projects/shell/src/app/app.module.ts
// and
// projects/mfe1/src/app/app.module.ts

[...]
```

```
import { AuthModule } from '@auth0/auth0-angular';

[...]
```

```
@NgModule({
  imports: [
    AuthModule.forRoot({
      domain: 'dev-ha-6vf7s.us.auth0.com',
      clientId: 'pQXuZGn3bf0fHpFd9ch7Wfa4xP4KhK1S'
    }),
    [...],
  ],
  [...]
```

```
  })  
  export class AppModule { }
```

The values for `domain` and `clientId` can be found at <https://manage.auth0.com> after registering your application.

After that, we can use the `AuthService` in the shell to login the current user:

```
// projects/shell/src/app/home/home.component.ts  
  
import { Component } from '@angular/core';  
import { AuthService } from '@auth0/auth0-angular';  
  
@Component({  
  selector: 'app-home',  
  templateUrl: './home.component.html',  
  styleUrls: ['./home.component.scss']  
})  
export class HomeComponent {  
  
  user$ = this.auth.user$;  
  
  constructor(private auth: AuthService) {}  
  
  login(): void {  
    this.auth.loginWithRedirect();  
  }  
}
```

```
}  
  
}
```

I also decided to display the user's name after they logged in:

```
<!--  
  projects/shell/src/app/home/home.component.html  
-->  
  
<h1>Welcome!</h1>  
  
<p *ngIf="user$ | async as user">  
  User: {{user.name}}  
</p>  
  
<div>  
  <button (click)="login()" mat-flat-button color="primary">Login</button>  
</div>
```

In the micro frontend's `AddressComponent`, I'm also using the `AuthService` to get some information about the current user:

```
// projects/mfe1/src/app/address/address.component.ts  
  
[...]
```



```
import { AuthService } from '@auth0/auth0-angular';

@Component({
  selector: 'app-address',
  templateUrl: './address.component.html',
  styleUrls: ['./address.component.scss']
})
export class AddressComponent {

  [...]

  constructor(
    private auth: AuthService,
    private fb: FormBuilder) {

    this.auth.user$.pipe(take(1)).subscribe(user => {
      if (!user) return;
      this.addressForm.controls['firstName'].setValue(user.given_name);
      this.addressForm.controls['lastName'].setValue(user.family_name);

    });

  }
```

```
[...]  
}
```

In order to make this work, we need to share the `@auth0/auth0-angular` package:

```
// projects/shell/webpack.config.js  
// AND  
// projects/mfe1/webpack.config.js  
  
[...]  
shared: share({  
  [...]  
  
  // Add this:  
  "@auth0/auth0-angular": { singleton: true, strictVersion: true, requiredVersion: 'auto' },  
  
  ...sharedMappings.getDescriptors()  
})  
[...]
```

This ensures there is only one `AuthService` at runtime we can use to share information about the current user.

After changing the webpack configuration, we need to restart our applications. If everything works well, we can log in the user via the shell and read the user's name in the lazy loaded micro frontend.

# Summary

Module Federation allows loading separately compiled applications at runtime. Also, we can share common dependencies. This also allows sharing common data like information on the current user or global filters.

As Module Federation allows runtime integration, it is also the key for plugin systems and micro frontend architectures. However, this approach turns to compile-time dependencies into runtime dependencies; we need a sound set of integration tests to detect issues.

In general, Micro Frontends only makes sense if you want to scale a project by splitting it into several smaller applications developed by different autarkic teams. If you only have one frontend team, Micro Frontend architectures are very likely an overhead.



**Manfred Steyer**

TRAINER AND CONSULTANT WITH FOCUS ON ANGULAR

Under the leadership of Manfred Steyer, ANGULARarchitects.io supports companies with the implementation of web-based business applications with Angular.

At the Angular seminars he designs, he combined his profound knowledge of Angular, for which he was recognized by Google as a Google Development Expert (GDE), with his experience, which he worked as a team leader in the field of software development, as a Professor at a university of applied sciences and as an external member of the Angular team.

Manfred has published books at O'Reilly, Microsoft Press, and Hanser and writes for Heise Online, windows.developer, and the Java magazine. He also regularly shares his knowledge at conferences.

[VIEW PROFILE ▶](#)

---

## More like this

### ACTIONS

**Country-Based Access with Auth0 Actions**

### ANGULAR

**Add OpenID Connect to Angular Apps Quickly**

ANGULAR

## State Management in Angular Using Akita - Pt. 1

Follow the conversation



Powered by the Auth0 Community. [Sign up](#) now to join the discussion. **Community links will open in a new window.**

---

Secure access for everyone. But not just anyone



Secure access for everyone. But not just anyone.

TRY AUTH0 FOR FREE

TALK TO SALES

## BLOG

Developers  
Identity & Security  
Business  
Leadership  
Culture  
Engineering  
Announcements

## COMPANY

About Us  
Customers  
Security  
Careers  
Partners  
Press  
Status  
Legal  
Privacy Policy  
Terms

## PRODUCT

Single Sign-On  
Password Detection  
Guardian  
M2M  
Universal Login  
Passwordless

## MORE

Auth0.com  
Ambassador Program  
Guest Author Program  
Auth0 Community  
Resources



© 2013-2022 Auth0 Inc. All Rights Reserved.