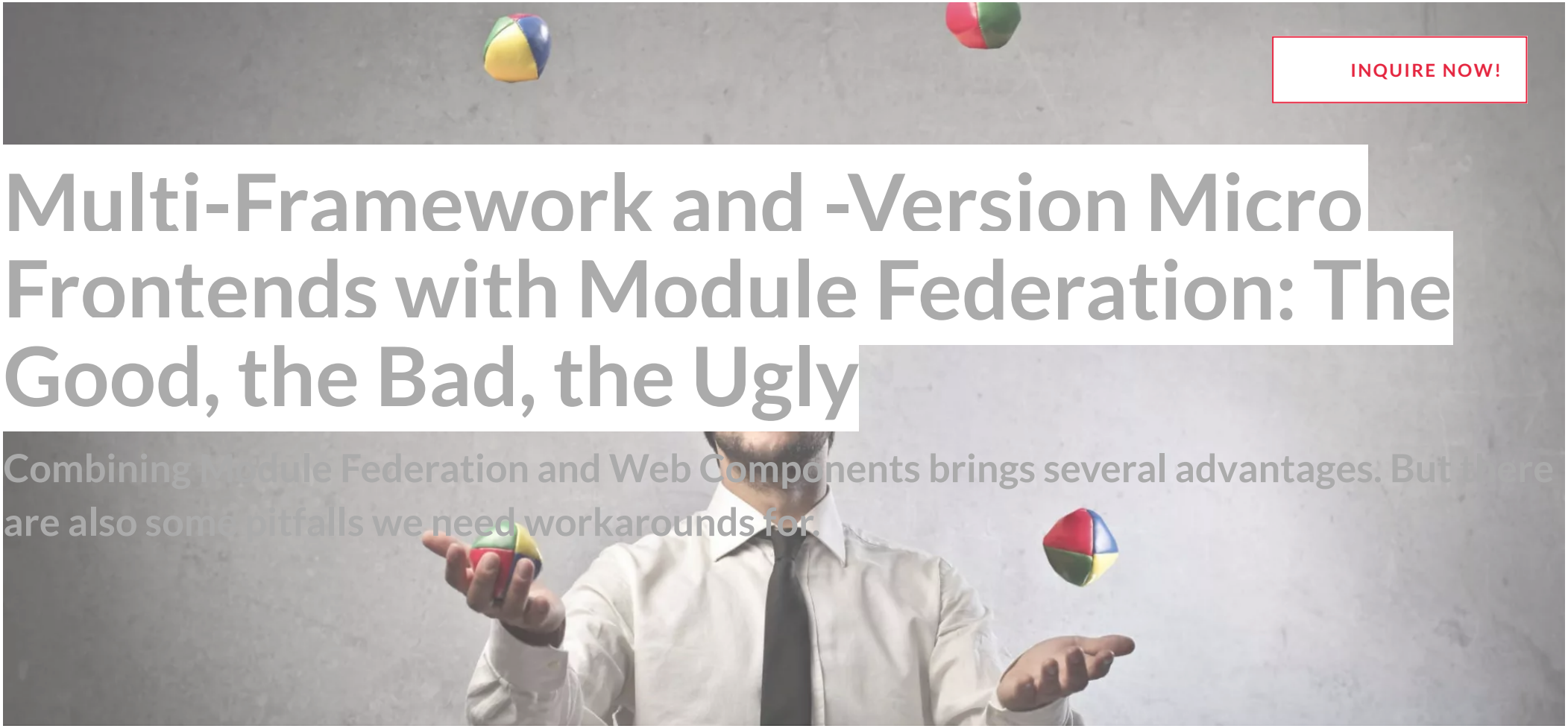
Multi-Framework and -Version Micro Frontends with Module Federation: The Good, the Bad, the Ugly



Combining Module Federation and Web Components brings several advantages. But there are also some pitfalls we need workarounds for.

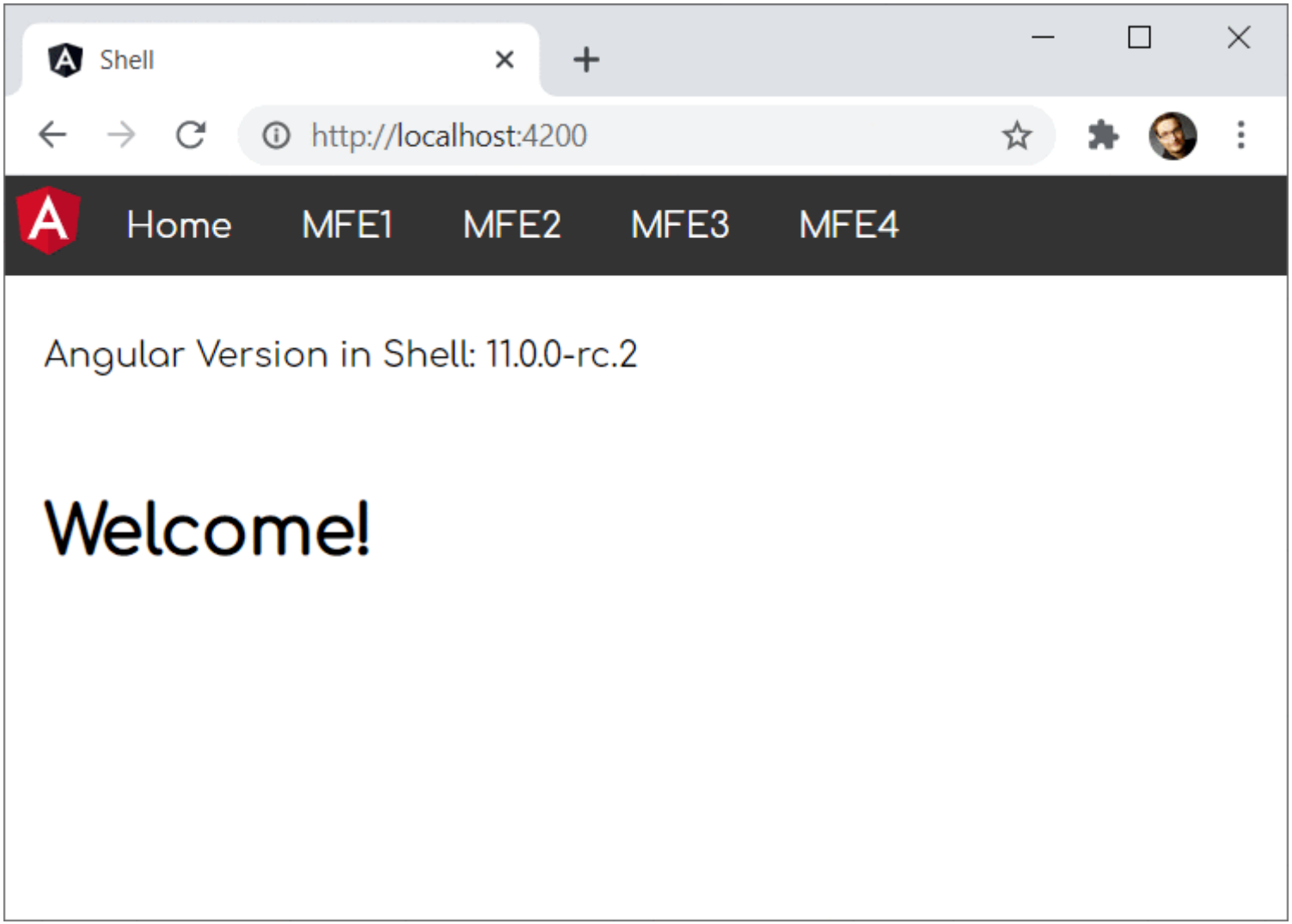
Article by Manfred Steyer, GDE | 01.06.2021 | share

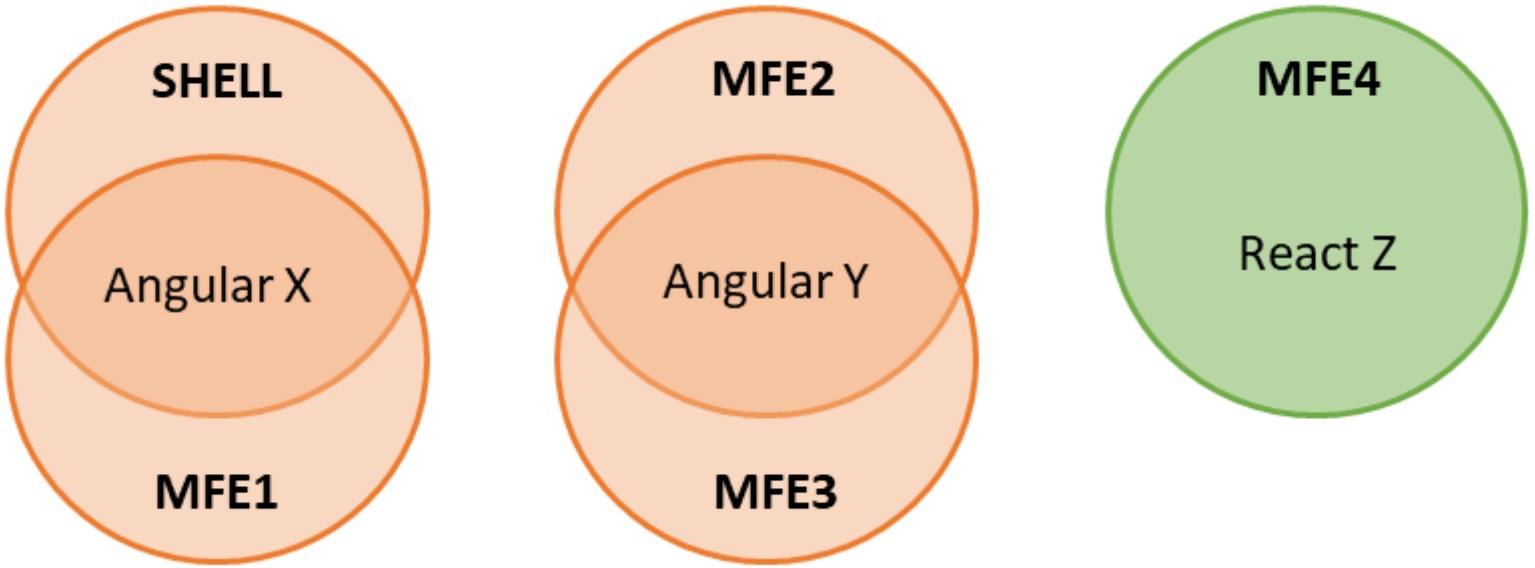
This is post 7 of 8 in the series *"Module Federation"*

> Updated on 2021-12-23 for CLI 13.1.x and above

Recently, I've been asked several times how to combine Micro Frontends built with different frameworks and/or framework versions. A quite modern and flexible approach for this is providing Web Components via Module Federation.

Nevertheless, in general, combining different frameworks and versions is nothing the individual frameworks have been built for. Hence, there are some pitfalls, this articles shows some workarounds for.


Shell
http://localhost:4200
Home
MFE1
MFE2
MFE3
MFE4
Angular Version in Shell: 11.0.0-rc.2
Welcome!



The source code for this case study can be found here.

The 1st Rule for Multi Framework(version) MFEs

Let's start with the 1st rule for multi framework and multi version micro frontend architectures: Don't do it ;-).

Alternative 1: Evergreen Version with Module Federation

The Angular team is heavily investing in seamless upgrades. The Angular CLI command `ng update` is providing the results of this by the push of a button. It executes several migration scripts lifting your source code to the newest version. This and respective processes at Google allow them to always use the newest Angular version for all of their more than 2600 Angular applications.

Also, if all parts of your system use the same major version, using Module Federation for integrating them is straightforward. We **don't need** Web Components to bridge the gap between different versions and from our framework's perspective, everything we do is using lazy loading. Underneath the covers, Module Federation takes care of loading separately compiled code at runtime.

An example demonstrating this can be found in this article that is part of the article series at hand.

Alternative 2: Relaxing Version Requirements + Heavy Testing

provides the configuration property requiredVersion described here.

This might work because, normally, Angular's core didn't recently change much between major However, this is not officially supported and hence you need a huge amount of E2E tests to make sure everything works seamlessly together. On the other side, you need E2E tests anyway as micro frontends are runtime dependencies not known upfront at compilation time.

A Good Message Before We Start

Most of the tricks and tweks I'm presenting here are meanwhile automated by my library @angular-architects/module-federation-tools which is an add-on for @angular-architects/module-federation. Here you even find a live demo.

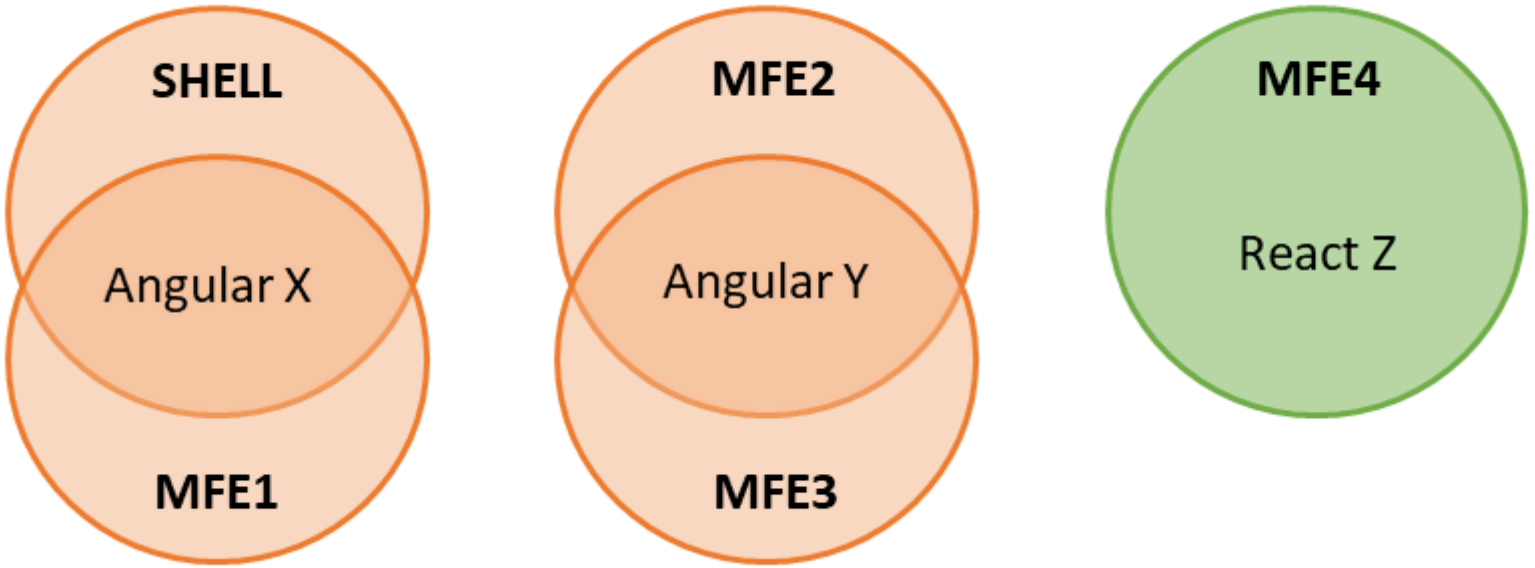
Nethertheless, this article can be vital for you because it helps you to understand both, the underlying ideas and also how to use them — automated via a library or hand-written — in an Angular application.

The Good

about the good aspects of this.

Sharing Libraries

As mentioned before, our case study is sharing two versions of Angular:



For this, the shell and each Micro Frontend just needs to mention the libraries to share in their Module Federation config:

```
3   shared: [ '@angular/core', '@angular/common', '@angular/router' ]
4 })
```


By default, Module Federation is using semantic versioning to find out the highest compatible version available. Let's say, we have the following constellation:

- Shell: @angular/core@^**12.0**.0
- MFE1: @angular/core@^**12.1**.0
- MFE2: @angular/core@^**13.1**.0
- MFE3: @angular/core@^**13.0**.0

In this case, Module Federation decides to go with the following versions:

- Shell and MFE1: @angular/core@^**12.1**.0
- MFE2 and MFE3: @angular/core@^**13.1**.0

In both cases, the respective highest compatible version is used. More details about this clever mechanism and configuration options to influence this behavior can be found here.

Using Module Federation, a Micro Frontend (officially called *remote*) can expose all possible code fragments. In cases where all parts of the system use the same framework version, this could be an Angula component.

If we have different framework versions, we can expose web components:

```
new ModuleFederationPlugin({
 [...],
 exposes: {
  './web-components': './src/bootstrap.ts',
 },
 [...]
})
```

The `bootstrap.ts` file is bootstrapping an Angular application providing an Angular component as a Web Component with Angular Elements. To add Angular Elements to your project, use the following CLI command:

```
ng add @angular/elements
```

The `main.ts` file only contains the following dynamic `import`:


```
1 | import('./bootstrap');
```

As mentioned in one of the previous articles of this series, this is a typical pattern for Module Federation. It gives the application the time necessary for negotiating the library versions to use and for loading them.

In order to provide a web component, the Micro Frontend's `AppModule` uses Angular Elements:

```
1  | [...]
2  | import { createCustomElement } from '@angular/elements';
3  | [...]
4  |
5  | @NgModule({
6  |  imports: [
7  |   BrowserModule,
8  |   RouterModule.forRoot([...])
9  |  ],
10 |  declarations: [
11 |   [...]
```

```
15    bootstrap: []
16  })
17  export class AppModule {
18    constructor(private injector: Injector) {
19    }
20
21    ngDoBootstrap() {
22      const ce = createCustomElement(AppComponent, {injector: this.injector});
23      customElements.define('mfe1-element', ce);
24    }
25
26  }
```


Please note that this `AppModule` doesn't have a bootstrap component. The `bootstrap` array is empty. Hence, Angular calls the module's `ngDoBootstrap` method. Here, `createCustomElement` wraps an Angular component as a web component (a custom element to be more precise).

Also, it registers this Web Component with the browser using `customElements.define`. This method assigns the tag name `mfe1-element` to it.

Also, loading a separately compiled micro frontend on demand is straightforward with Module Federation. For this, the shell (officially called *host*) can leverage the helper method `loadRemoteModule` p

package `@angular-architects/module-federation`.

```
1  import { loadRemoteModule } from '@angular-architects/module-federation';
2
3  export const registry = {
4    mfe1: () => loadRemoteModule({
5      type: 'module',
6      remoteEntry: 'http://localhost:4201/remoteEntry.js',
7      exposedModule: './web-components'
8    }),
9    mfe2: () => loadRemoteModule({
10     type: 'script',
11     remoteEntry: 'http://localhost:4202/remoteEntry.js',
12     remoteName: 'mfe2',
13     exposedModule: './web-components'
14   }),
15   mfe3: () => loadRemoteModule({
16     type: 'script',
17     remoteEntry: 'http://localhost:4203/remoteEntry.js',
```

```
21    mfe4: () => loadRemoteModule({
22      type: 'script',
23      remoteEntry: 'http://localhost:4204/remoteEntry.js',
24      remoteName: 'mfe4',
25      exposedModule: './web-components'
26    }),
27  };
```


The calls needed for this example have been placed in the registry object shown. Please note that the first call is using `type: 'module'` while the others are going with `type: 'script'`. The reason is that the first one uses Angular 13.1 or higher. Beginning with Angular 13, the CLI emits EcmaScript modules instead of just "plain old" JavaScript files.

As mfe2 and mfe3 are based on Angular 2 and mfe4 on a classical webpack build for React, we need to go with `type: 'script'` here. In this case, we also have to specify the `remoteName` property which is defined in the remotes' webpack configurations.

More about working with these methods can be found in my article on Dynamic Federation.

To load the micro frontends, we just need to call the methods in the registry object:

```
4 await registry.mfe1();
```


After loading the web component, we can immediately use it. For this, we just need to add an element with the registered name.

Routing to Web Components

Of course, just loading our Micro Frontends and using them as dynamic web components is not enough. Our shell also needs to be able of routing to it.

For routing to such a Web Component, the case study at hand uses a `WrapperComponent`:

```
1 @NgModule({
2  imports: [
3   BrowserModule,
4   RouterModule.forRoot([
5    { path: '', component: HomeComponent, pathMatch: 'full' },
6    { [...], component: WrapperComponent, data: { importName: 'mfe1', elementName: 'mfe1-element' }},
7    { [...], component: WrapperComponent, data: { importName: 'mfe2', elementName: 'mfe2-element' }},
```

```
11      ])
12    ],
13    declarations: [
14      AppComponent,
15      WrapperComponent
16    ],
17    providers: [],
18    bootstrap: [AppComponent]
19  })
20  export class AppModule { }
```


To configure this wrapper, the router config's `data` property is used. It points to the remote's name mapped in the Module Federation config (`importName`) and to the respective Web Component's element name (`elementName`).

The wrapper's implementation just loads the web component and adds it to a placeholder referenced with a `ViewChild` :

```
1  import { AfterContentInit, Component, ElementRef, OnInit, ViewChild, ViewContainerRef } from '@angular/core
2  import { ActivatedRoute } from '@angular/router';
   import { registry } from '../registry';
```

```
6  })
7  export class WrapperComponent implements AfterContentInit {
8
9   @ViewChild('vc', {read: ElementRef, static: true})
10  vc: ElementRef;
11
12  constructor(private route: ActivatedRoute) { }
13
14  ngAfterContentInit(): void {
15
16   const elementName = this.route.snapshot.data['elementName'];
17   const importName = this.route.snapshot.data['importName'];
18
19   const importFn = registry[importName];
20   importFn()
21    .then(_ => console.debug(`element ${elementName} loaded!`))
22    .catch(err => console.error(`error loading ${elementName}:`, err));
23
24   const element = document.createElement(elementName);
25   this.vc.nativeElement.appendChild(element);
26
27  }
28
```

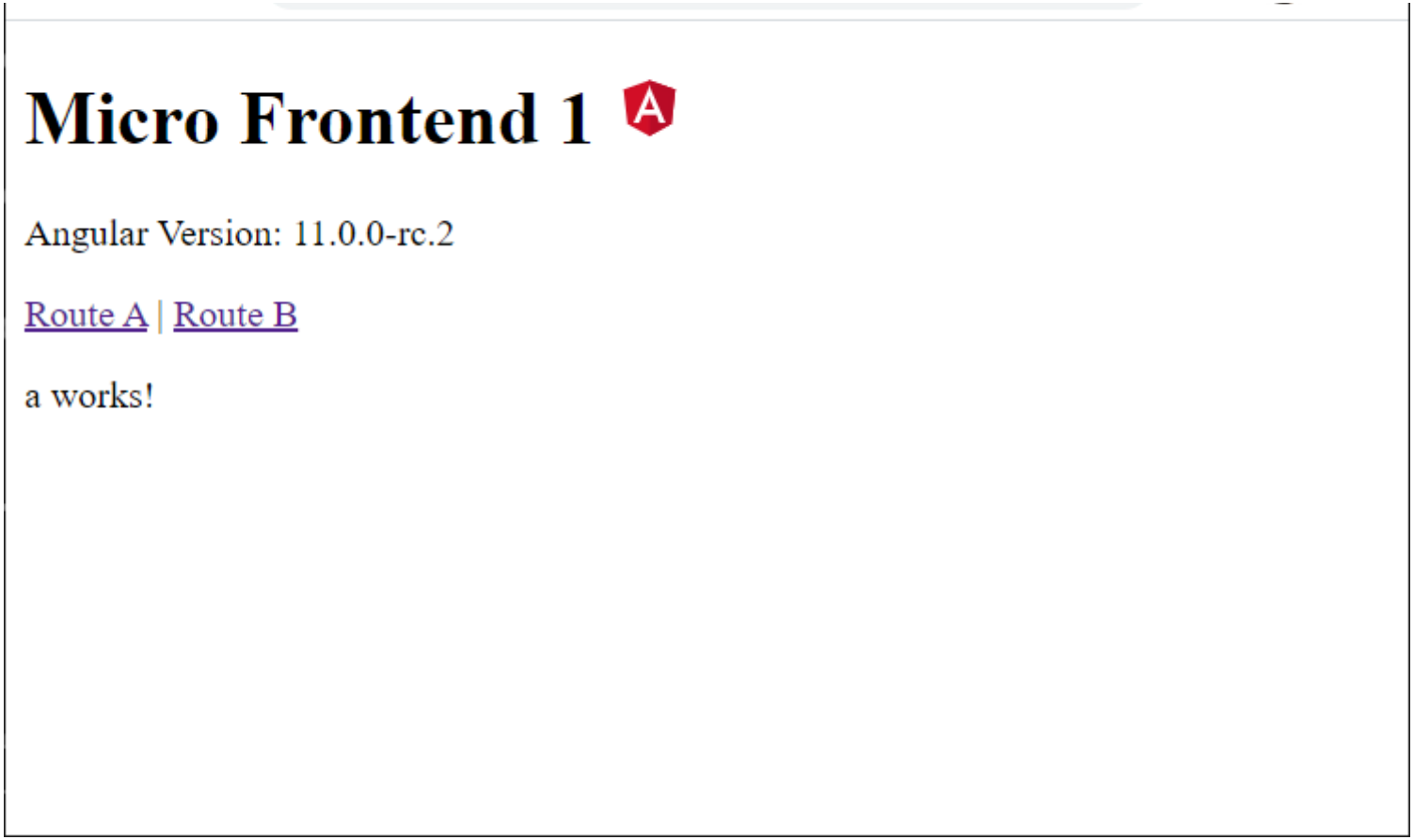
As an **alternative**, we could make this example more dynamic by skipping the registry and pass for loading the remote and creating its root element **directly** to the wrapper component.

No Need for a Separate Meta Framework

One of the best things of using Module Federation in general is that your framework — in this case Angular — does not even recognize that we are loading a separately compiled Micro Frontend. From Angular's perspective, this is just traditional lazy loading. Webpack Module Federation takes care of the heavy lifting underneath the covers. Hence, we don't need a separate meta framework and hence our scenario becomes easier.

Standalone Mode

All the Micro Frontends can also be started in standalone mode:



This is important because we want to develop, test, and deploy our Micro Frontends separately.

Lazy Loading within Micro Frontends

Micro Frontend does not remember from which URL it was loaded and hence doesn't know where to find its lazy chunks.

The Bad

Now, let's proceed with some of the challenges this architecture comes with.

Bundle Size

Obviously, using several versions of the same framework but also using several frameworks together increases the bundle size. More stuff needs to be downloaded into the browser. However, if you have a lot of returning users, they will benefit from cache hits.

Nevertheless, you need to respect the impact of increased bundles sizes when deciding for or against this architecture. While in some cases, this can for sure be neglected, e. g. in intranet scenarios, there are cases where this trade-off is not acceptable, e. g. in mobile scenarios or when the conversion rate is critical. Being able to run individual Micro Frontends in standalone mode, can help here for sure.

In cases where the loaded Micro Frontends also use routing, we need to coordinate several routers: The shell's router and the router found in the individual Micro Frontends.

Let's say, we have the URL `mfe1/a`. In this case, the shell should only care about the first part, `mfe1` and route to the Web Component provided by Micro Frontend 1. The Micro Frontend in turn, should only respect the ending, `/a` and activate the respective route.

To achieve this, we could use `UrlMatcher` instead of concrete paths in the router configs. For instance, our shell is using the custom `UrlMather` `startsWith`:

```
@NgModule({
 imports: [
  BrowserModule,
  RouterModule.forRoot([
   { path: '', component: HomeComponent, pathMatch: 'full' },
   { matcher: startsWith('mfe1'), component: WrapperComponent, data: { importName: 'mfe1', elementName: 'm
   { matcher: startsWith('mfe2'), component: WrapperComponent, data: { importName: 'mfe2', elementName: 'm
   { matcher: startsWith('mfe3'), component: WrapperComponent, data: { importName: 'mfe3', elementName: 'm
   { matcher: startsWith('mfe4'), component: WrapperComponent, data: { importName: 'mfe4', elementName: 'm
  ])
```

```
14      WrapperComponent
15    ],
16    providers: [],
17    bootstrap: [AppComponent]
18  })
19  export class AppModule { }
```


All routes starting with `mfe1` will make the `WrapperComponent` loading *Micro Frontend 1*, for instance. The rest of the path is ignored by the shell and can be utilized by the Micro Frontend itself.

This is what the implementation of `startsWith` looks like:

```
1  import { UrlMatcher, UrlSegment } from '@angular/router';
2
3  export function startsWith(prefix: string): UrlMatcher {
4    return (url: UrlSegment[]) => {
5      const fullUrl = url.map(u => u.path).join('/');
6      if (fullUrl.startsWith(prefix)) {
7        return ({ consumed: url});
8      }
       return null;
```

In the Micro Frontends, however, our case study only analyzes the end of the route with a respective `endsWith` function:


```
@NgModule({
 imports: [
  BrowserModule,
  RouterModule.forRoot([
   { matcher: endsWith('a'), component: AComponent},
   { matcher: endsWith('b'), component: BComponent},
  ])
 ],
 declarations: [
  AComponent,
  BComponent,
  AppComponent
 ],
 providers: [],
 bootstrap: []
})
export class AppModule {

```

Here is its implementation:


```
export function endsWith(prefix: string): UrlMatcher {
  return (url: UrlSegment[]) => {
    const fullUrl = url.map(u => u.path).join('/');
    if (fullUrl.endsWith(prefix)) {
      return ({ consumed: url});
    }
    return null;
  };
}
```

The Ugly

As all these frameworks are not designed to work side-by-side with different versions of itself or other frameworks, we also need some "special" workarounds. This section discusses them.

Bypassing Routing Issues

```
@Component([...])
export class AppComponent implements OnInit {

 [...]

 constructor(private router: Router) { }

 ngOnInit(): void {
  this.router.navigateByUrl(location.pathname.substr(1));
  window.addEventListener('popstate', () => {
   this.router.navigateByUrl(location.pathname.substr(1));
  });
 }
}
```


For hash-based routing, we'd use `location.hash` and the `hashchanged` event.

Reuse Angular Platform

platform instance.


```
declare const require: any;
const ngVersion = require('../package.json').dependencies['@angular/core']; // perhaps just take the major version

(window as any).plattform = (window as any).plattform || {};
let platform = (window as any).plattform[ngVersion];
if (!platform) {
 platform = platformBrowser();
 (window as any).plattform[ngVersion] = platform;
}
platform.bootstrapModule(AppModule)
 .catch(err => console.error(err));
```

Angular Elements and Zone.js

The last one is a general one regarding Angular Elements: Even if we just load Zone.js once, we get several Zone.js instances: One for the shell and one per each Micro Frontend. This can lead to issues with change detection when data crosses the border of micro frontends.

```
1 platformBrowser()
2  .bootstrapModule(AppModule, { ngZone: 'noop' })
```


However, this also means we need to do change detection by hand. My GDE colleague, Tomas Trajan came up with another idea: Sharing one Zone.js instance. For this, the shell is grabbing the current `NgZone` instance and puts it into the global namespace:

```
1 export class AppModule {
2  constructor(private ngZone: NgZone) {
3   (window as any).ngZone = this.ngZone;
4  }
5 }
```

All the Micro Frontends take it from there and reuse it when bootstrapping:

```
1 platformBrowser().bootstrapModule(AppModule, { ngZone: (window as any).ngZone })
```

If this global `ngZone` property is undefined, the micro frontend's Angular instance uses a `ngZone` instance of its own. This is also the default behavior.

Using Module Federation together with Web Components/ Angular Elements leads to a huge amount of advantages: We can easily share libraries, provide and dynamically load Web Components and components using a wrapper. Also, our main framework — e. g. Angular — also becomes our meta framework so that we don't need to deal with additional technologies. The loaded Web Components can even make use of lazy loading.

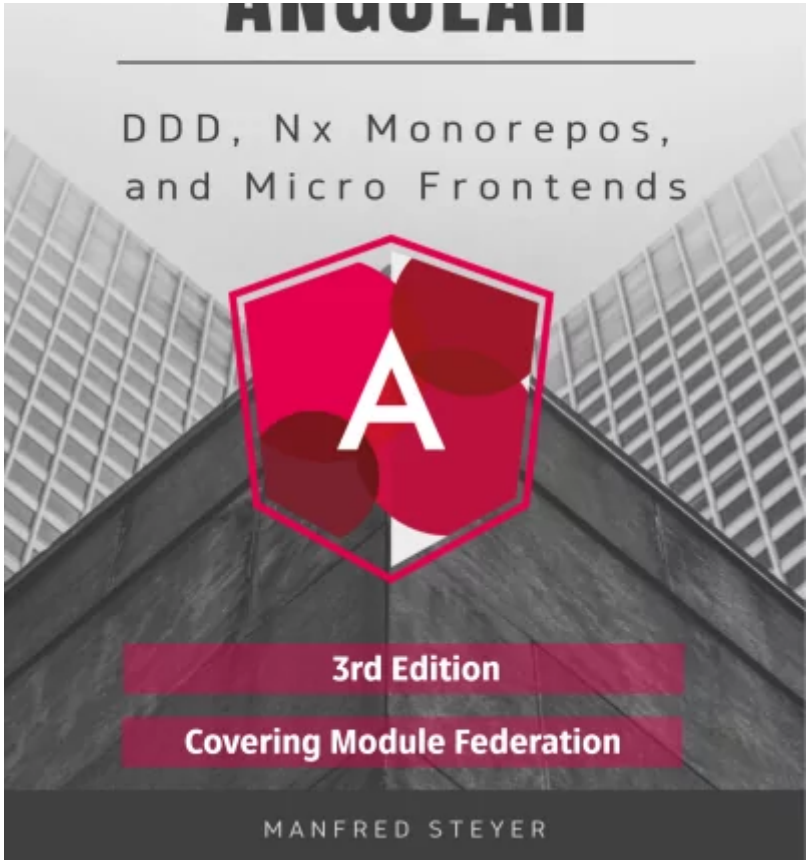
However, this comes with costs: Bundle Sizes increase and we need several tricks and workarounds to make everything work seamlessly.

In the past years I've helped numerous companies building Micro Frontend architectures using Web Components. Adding Module Federation to the game makes this by far simpler. Nevertheless, if you can somehow manage to just use one framework and version in your whole software system, using Module Federation is even more straightforward.

What's next? More on Architecture!

- 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

INQUIRE NOW!

Don't Miss Anything!

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

Business EMail Address*:

Country*

Subscribe*

Unsere Angular-Schulungen

Angular Workshop – Structured Introduction

Workshop with all backgrounds on the building blocks in Angular.

MEHR INFORMATIONEN

Angular Architecture Workshop

Workshop with strategies for your large and long-lasting business applications.

MEHR INFORMATIONEN

Professional An Testing

Quality assurance with mode Cypress and Storyb

MEHR INFORMATIO

Current Blog Articles

ALL ARTICLES

STATE-MANAGEMENT MIT

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

READ MORE

READ MORE

INQUIRE NOW!

and IT marketing in Graz and part-time computer science in Hagen and completed a four-semester training in the field of adult education.

INQUIRE NOW

Austria

manfred.steyer@softwarearchitekt.at

SUBSCRIBE TO NEWSLETTER

purpose of sending the newsletter.

DATA PROTECTIN

INQUIRE NOW!

Email:

SUBSCRIBE TO NEWSLETTER