The Microfrontend Revolution: Module Federation in Webpack 5

Module Federation allows loading separately compiled program parts. This makes it an official solution for implementing microfrontends.

22.04.2020 | share

Dies ist Beitrag 1 von 8 der Serie *"Module Federation"*

3. Dynamic Module Federation with Angular
4. Building A Plugin-based Workflow Designer With Angular and Module Federation
5. Getting Out of Version-Mismatch-Hell with Module Federation
6. Using Module Federation with (Nx) Monorepos and Angular
7. Multi-Framework and -Version Micro Frontends with Module Federation: The Good, the Bad, the Ugly
8. Pitfalls with Module Federation and Angular

> 2020-10-13: Updated to use webpack 5

> **Important:** This first part of the article series shows Module Federation with a simple "TypeScript-only example". If you look for an example also using **Angular**, please directly jump to the 2nd part of this series.

The Module Federation integrated in Webpack beginning with version 5 allows the loading of separately compiled program parts. Hence, it finally provides an official solution for the implementation of microfrontends.

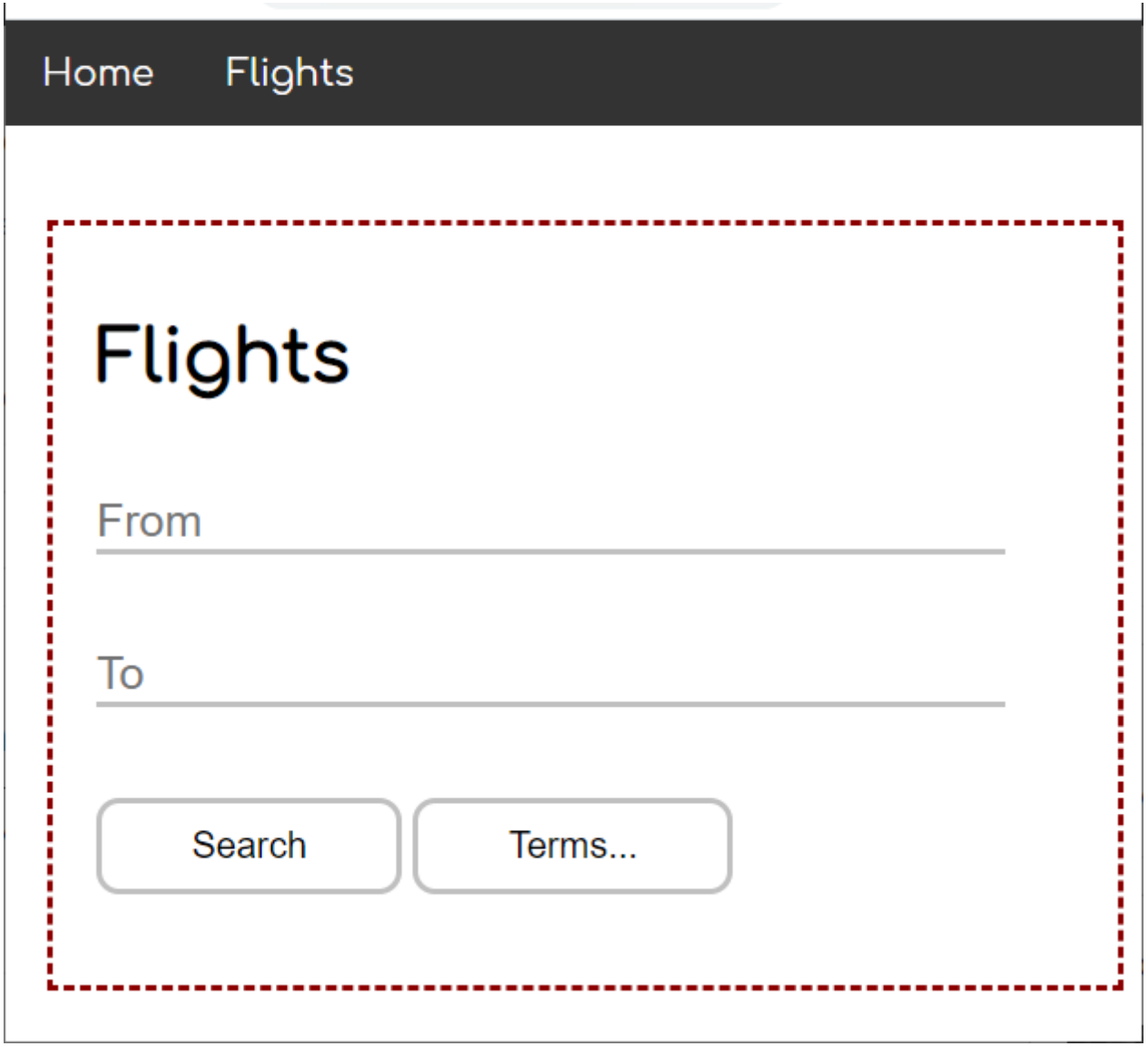
change of course here.

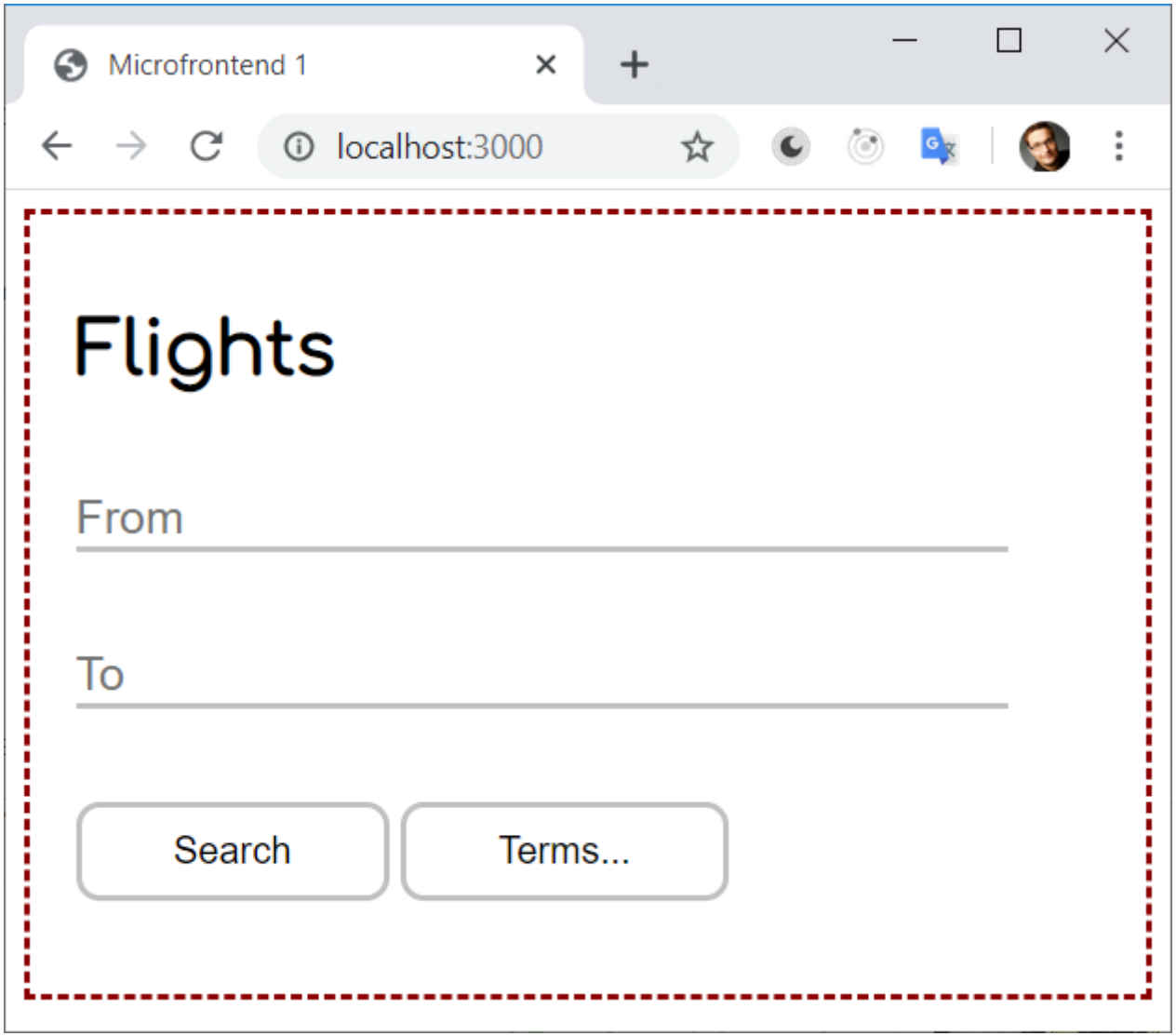
It allows an approach called Module Federation for referencing program parts that are not compile time. These can be self-compiled microfrontends. In addition, the individual program parts can share libraries with each other, so that the individual bundles do not contain any duplicates.

In this article, I will show how to use Module Federation using a simple example. The source code can be found here.

Example

The example used here consists of a shell, which is able to load individual, separately provided microfrontends if required:





Concepts of Module Federation

In the past, the implementation of scenarios like the one shown here was difficult, especially since tools like **Webpack assume** that the **entire program code is available when compiling**. Lazy loading is possible, but only from areas that were split off during compilation.

With microfrontend architectures, in particular, one would like to compile and provide the individual program parts separately. In addition, mutual referencing via the respective URL is necessary. Hence, constructs like this would be desirable:

```
1 | import('http://other-microfrontend');
```

Since this is not possible for the reasons mentioned, one had to resort to approaches such as externals and manual script loading. Fortunately, this will change with the Federation module in Webpack 5.

The idea behind it is simple: A so-called host references a **remote** using a configured name. What this name refers to is **not known at compile time**:

This reference is only **resolved at runtime by loading a so-called remote entry point**. It is a **minimal script** that provides the actual external url for such a configured name.

Implementation of the Host

The **host** is a JavaScript application that **loads a remote when needed**. A dynamic import is used for this.

The following host loads the component `mfe1/component` in this way — `mfe1` is the name of a configured remote and `component` the name of a file (an EcmaScript module) it provides.

```
1  const rxjs = await import('rxjs');
2
3  const container = document.getElementById('container');
4  const flightsLink = document.getElementById('flights');
5
6  rxjs.fromEvent(flightsLink, 'click').subscribe(async _ => {
7    const module = await import('mfe1/component');
```

```
11 | });
```


Webpack would normally take this reference into account when compiling and split off a separate bundle for it. To prevent this, the `ModuleFederationPlugin` is used:

```
1  const ModuleFederationPlugin = require("webpack/lib/container/ModuleFederationPlugin");
2
3  [...]
4   output: {
5      publicPath: "http://localhost:5000/",
6      uniqueName: 'shell',
7      [...]
8   },
9  plugins: [
10  new ModuleFederationPlugin({
11    name: "shell",
12    library: { type: "var", name: "shell" },
13    remoteType: "var",
14    remotes: {
15      mfe1: "mfe1"
16    },
```

With its help, the remote `mfe1` (Microfrontend 1) is defined. The configuration shown here internal application name `mfe1` to the same official name. Webpack **does not include** any `import` that now relates to `mfe1` in the bundles generated **at compile time**.

Libraries that the host should share with the remotes are mentioned in the `shared` array. In the case shown, this is `rxjs`. This means that the entire application only needs to load this library once. Without this specification, `rxjs` would end up in the bundles of the host as well as those of all remotes.

For this to work without problems, the host and remote must agree on a common version.

In addition to the settings for the `ModuleFederationPlugin`, we also need to place some options in the `output` section. The `publicPath` defines the URL under which the application can later be found. This reveals where the individual bundles of the application but also their assets, e.g. pictures or styles, can be found.

The `uniqueName` is used to represents the host or remote in the generated bundles. By default, webpack uses the name from `package.json` for this. In order to avoid name conflicts when using Monorepos with several applications, it is recommended to set the uniqueName manually.

For loading shared libraries, we must use **dynamic imports**:

```
1 | const rxjs = await import('rxjs');
```

They are asynchronous and this gives webpack the time necessary to decide upon which version to use and to load it. This is especially important in cases where different remotes and hosts use different versions of the same library. In general, webpack tries to load the highest compatible version. More about negotiating versions and dealing with version mismatches this can be found in a later article of this series.

To bypass this issue, it's a good idea to load the whole application with dynamic imports in the entry point used. For instance, the Micro Frontend could use a `main.ts` which looks like this:

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

This gives webpack the time needed for the negotiation and loading the shared libraries when the application starts. Hence, in the rest of the application one can always use static ("traditional") imports like

Implementation of the Remote

The remote is also a standalone application. In the case considered here, it is based on Web Components:

```
class Microfrontend1 extends HTMLElement {

  constructor() {
    super();
    this.attachShadow({ mode: 'open' });
  }

  async connectedCallback() {
    this.shadowRoot.innerHTML = `[...]`;
  }
}

const elementName = 'microfrontend-one';
customElements.define(elementName, Microfrontend1);

export { elementName };
```

The webpack configuration of the remote, which also uses the ` ModuleFederationPlugin ”, exports this component with the property exposes under the name component:

```
output: {
    publicPath: "http://localhost:3000/",
    uniqueName: 'mfe1',
    [...]
},
[...]
plugins: [
  new ModuleFederationPlugin({
    name: "mfe1",
    library: { type: "var", name: "mfe1" },
    filename: "remoteEntry.js",
    exposes: {
      './component': "./mfe1/component"
    },
    shared: ["rxjs"]
  })
]
```

and component. This results in the instruction shown above:


```
1 | import('mfe1/component')
```

However, the host must know the URL at which it finds `mfe1`. The next section shows how this can be accomplished.

Connect Host to Remote

To give the host the option to resolve the name `mfe1`, the host must load a remote entry point. This is a script that the `ModuleFederationPlugin` generates when the remote is compiled.

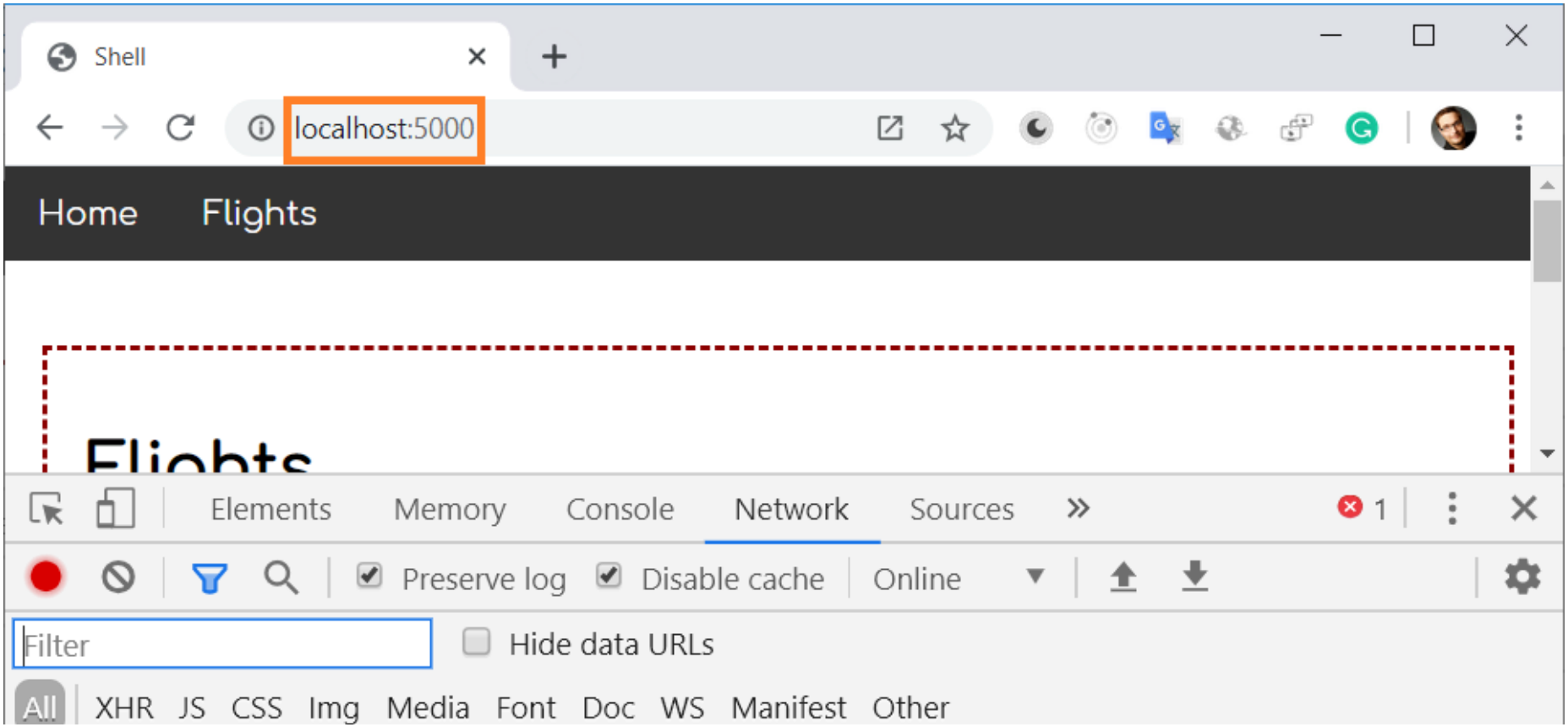
The name of this script can be found in the `filename` property shown in the previous section. The url of the microfrontend is taken from the `publicPath` property. This means that the url of the remote must already be known when it is compiled. Fortunately, there is already a PR which removed this need.

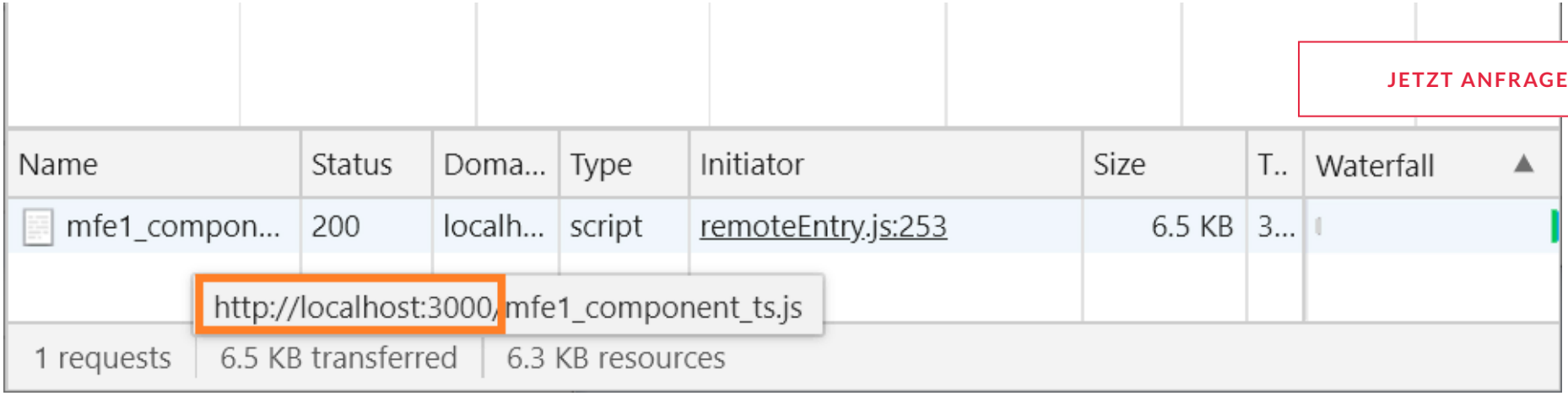
Now this script is only to be integrated into the host:

At runtime it can now be observed that the instruction


```
1 | import('mfe1/component');
```

causes the host to load the remote from its own url (which is `localhost:3000` in our case):





Conclusion and Outlook

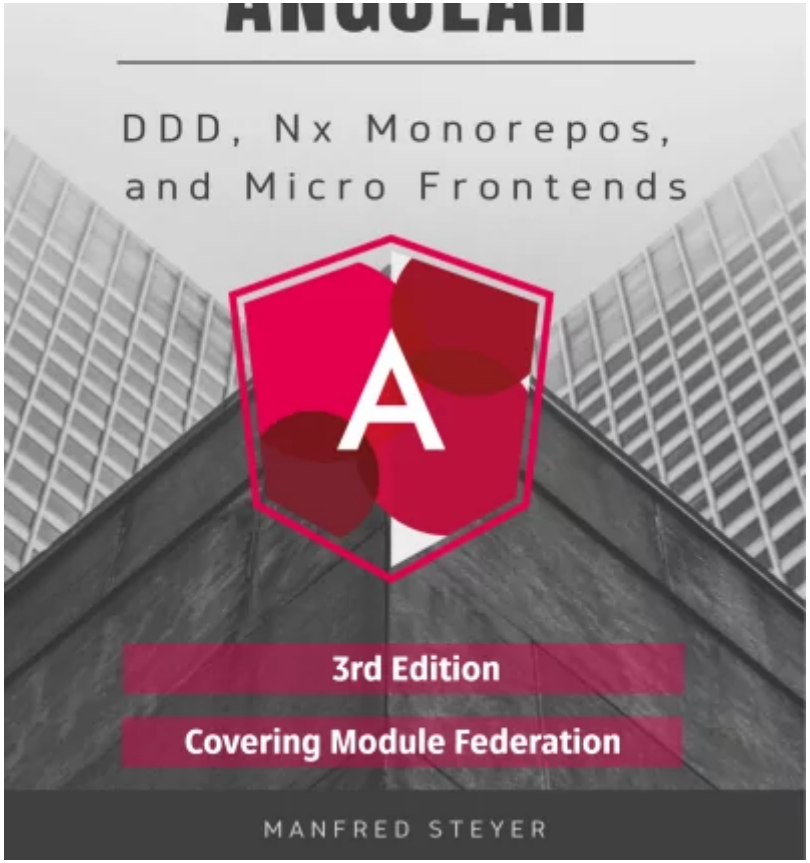
The Module Federation integrated in Webpack beginning with version 5 **fills a large gap** for microfrentends. Finally, separately compiled and provided program parts can be reloaded and already loaded libraries can be reused.

However, the teams involved in developing such applications must manually ensure that the individual parts interact. This also means that you have to define and comply with contracts between the individual microfrontends, but also that you have to agree on a version for each of the shared libraries.

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!

JETZT ANFRAGEN!

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: