ENTERPRISE ANGULAR

DDD, Nx Monorepos, and Micro Frontends

4th Extended Edition

Module Federation & More

MANFRED STEYER

Enterprise Angular

DDD, Nx Monorepos and Micro Frontends

Manfred Steyer

This book is for sale at http://leanpub.com/enterprise-angular

This version was published on 2021-12-28

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2019 - 2021 Manfred Steyer

Tweet This Book!

Please help Manfred Steyer by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I've just got my free copy of @ManfredSteyer's e-book about Enterprise Angular: DDD, Nx Monorepos, and Micro Frontends. https://leanpub.com/enterprise-angular

The suggested hashtag for this book is #EnterpriseAngularBook.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#EnterpriseAngularBook

Contents

Introduction

Over the last years, I've helped numerous companies with implementing large-scale enterprise applications with Angular.

One vital aspect here is decomposing the system into smaller libraries to reduce complexity. However, if this results in countless small libraries which are too intermingled, you haven't exactly made progress. If everything depends on everything else, you can't easily change or extend your system without breaking other parts.

Domain-driven design, especially strategic design, helps. Also, strategic design can be the foundation for building micro frontends.

This book, which builds on several of my blogposts about Angular, DDD, and micro frontends, explains how to use these ideas.

If you have any questions or feedback, please reach out at `manfred.steyer@angulararchitects.io`.
I'm also on Twitter (https://twitter.com/ManfredSteyer) and Facebook (https://www.facebook.com/manfred.steyer).
Stay in touch for updates about my Enterprise Angular work.

Help Improve this Book!

Please let me know if you have any suggestions. Send a pull request to the book's GitHub repository[1].

Trainings and Consultancy

If you and your team need support or trainings regarding Angular, we are happy to help with our on-site workshops and consultancy. In addition to several other kinds of workshop, we provide the following ones:

- Advanced Angular: Enterprise Solutions and Architecture
- Angular Essentials: Building Blocks and Concepts
- Angular Architecture Workshop
- Angular Testing Workshop (Cypress, Just, etc.)
- Angular: Reactive Architectures (RxJS and NGRX)
- Angular Review Workshop
- Angular Upgrade Workshop

[1] https://github.com/manfredsteyer/ddd-bk

Please find the full list of our offers here[2].

Advanced Angular Workshop

We provide our offer in various forms: **Online**, **public dates**, or as **dedicated company workshops** in **English** or **German**.

If you have any questions, reach out to us using office@softwarearchitekt.at.

Thanks

I want to thank several people who have helped me write this book:

- The great people at Nrwl.io[3] who provide the open-source tool Nx[4] used in the case studies here and described in the following chapters.
- Thomas Burleson[5] who did an excellent job describing the concept of facades. Thomas contributed to the chapter about tactical design which explores facades.
- The master minds Zack Jackson[6] and Jack Herrington[7] helped me to understand the API for Dynamic Module Federation.
- The awesome Tobias Koppers[8] gave me valuable insights into this topic and

[2] https://www.angulararchitects.io/en/angular-workshops/
[3] https://nrwl.io/
[4] https://nx.dev/angular
[5] https://twitter.com/thomasburleson?lang=de
[6] https://twitter.com/ScriptedAlchemy
[7] https://twitter.com/jherr
[8] https://twitter.com/wSokra

- The one and only Dmitriy Shekhovtsov[9] helped me using the Angular CLI/webpack 5 integration for this.

[9] https://twitter.com/valorkin

Strategic Domain-Driven Design

To make enterprise-scale applications maintainable, they need to be sub-divided into small, less complex, and decoupled parts. While this sounds logical, this also leads to two difficult questions: How to identify such parts and how can they communicate with each other?

In this chapter, I present a techniques I use to slice large software systems: Strategic Design – a discipline of the domain driven design[10] (DDD) approach.

What is Domain-Driven Design?

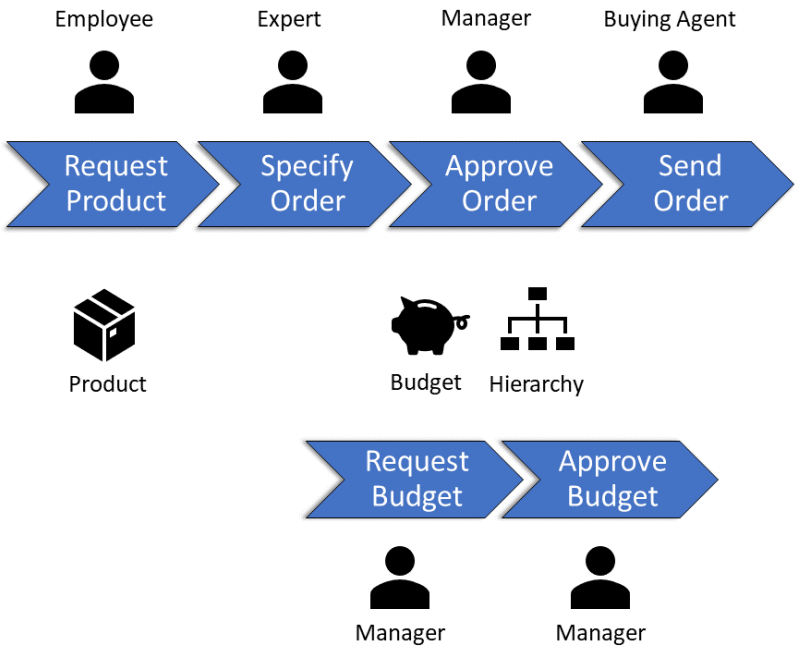
DDD describes an approach that bridges the gap between the requirements for complex software systems and an appropriate application design. Historically, DDD came with two disciplines: tactical design and strategic design. Tactical design proposes concrete concepts and design patterns. Meanwhile most of them are common knowledge. Examples are concepts like layering or patterns like factories, repositories, and entities.

By contrast, strategic design deals with subdividing a huge system into smaller, decoupled, and less complex parts. This is what we need to define an architecture for a huge system that can evolve over time.

Finding Domains with Strategic Design

The goal of strategic design is to identify so-called sub-domains that don't need to know much about each other. To recognize different sub-domains, it's worth taking a look at the processes automated by your system. For example, an e-procurement system that handles the procurement of office supplies could support the following two processes:

[10] https://www.amazon.de/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215/ref=sr_1_3?ie=UTF8&qid=1551688461&sr=8-3&keywords=ddd



To use these processes for identifying different domains, we can use several heuristics:

- **Organizational Structure**: Different roles or different divisions that are responsible for several steps of the process are in indicator for the existence of several sub-domains.
- **Vocabulary**: If the same term is used differently or has a significant different importance, we might have different sub-domains.
- **Pivotal Events**: Pivotal Events are locations in the process where a significant (sub)task is completed. After such an event, very often, the process goes on at another time and/or place and/or with other roles. If our process was a movie, we'd have a scene change after such an event. Such events are likely boundaries between sub-domains.

Each of these heuristics gives you candidates for cutting your process into sub-domains. However,it's your task to decide with which candidates to go. The general goal is to end up with slices that don't need to know much about each other.

The good message is: You don't need to do such decisions alone. You should do it together with other stakeholders like, first and foremost, business experts but also other architects, developers and product owners.

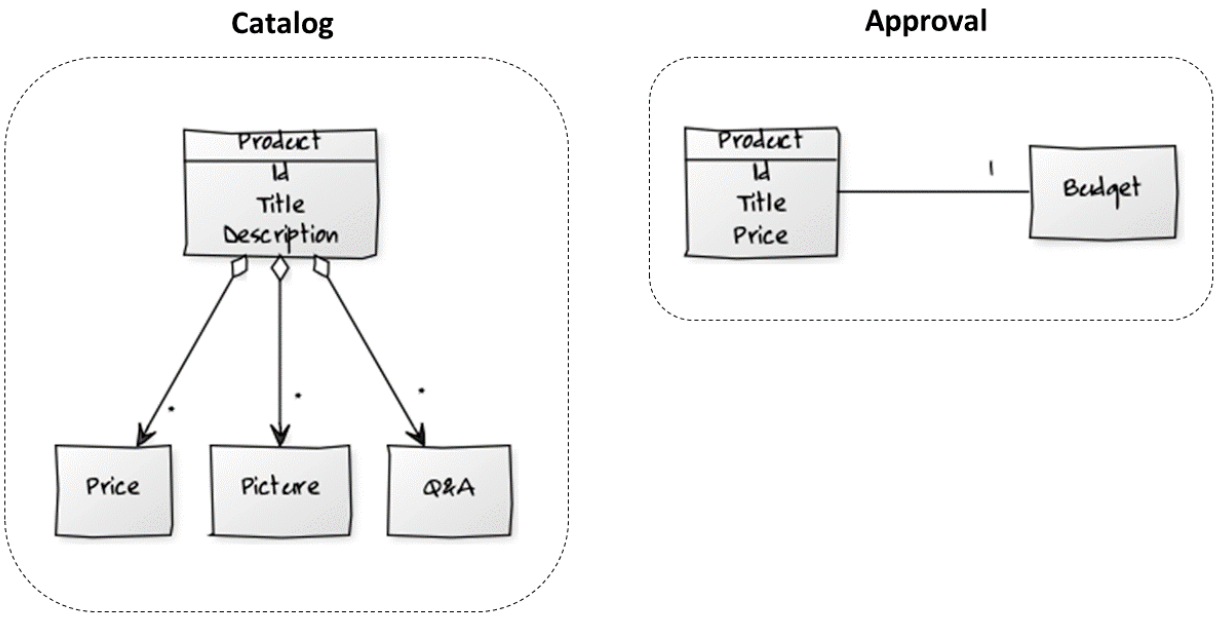
A modern approach for brining the knowledge of all these different people together is Event Storming[11]. It's a workshop format where different groups of stakeholders. For this, they model the processes together with post-its (sticky notes).

[11] https://www.eventstorming.com

Domains are Modelled Separately

Another important aspect of Strategic Design is that each domain is modelled separately. This is the key for decoupling domains from each other. While this might lead to redundancies, very often it doesn't because each domain has a very unique perspective to its entities.

For instance, a `product` is not exactly the same in all domains. For example, while a product description is very detailed in the catalogue, the approval process only needs a few key data:



In DDD, we distinguish between these two forms of a product. We create different models that are as concrete and meaningful as possible.

This approach prevents the creation of a single confusing model that attempts to describe the whole world. Such models have too many interdependencies that make decoupling and subdividing impossible.

We can still relate different views on the product entity at a logical level. If we use the same id on both sides, we know which "catalog product" and which "approval product" are different view to the same entity.

Hence, each model is only valid for a specific area. DDD calls this area the bounded context[12]. To put it in another way: The bounded context defines thought borders and only within these borders the model makes sense. Beyond these borders we have a different perspective to the same concepts. Ideally, each domain has its own bounded context.

Within such a bounded context, we use a ubiquitous language. This is mainly the language of the domain experts. That means we try to mirror the real world with our model and also

[12] https://martinfowler.com/bliki/BoundedContext.html

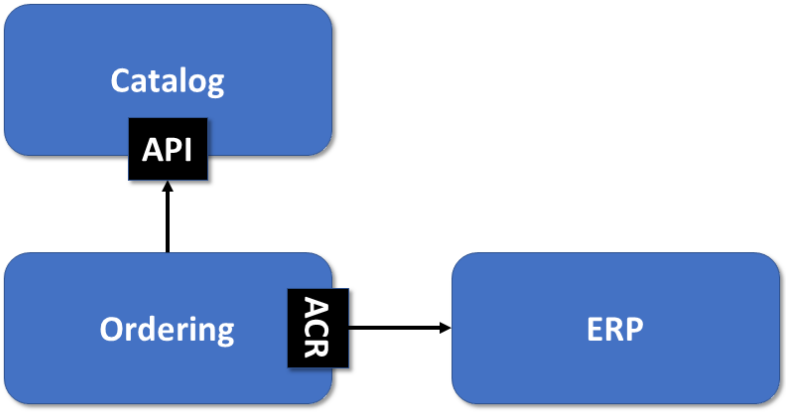
within our implementation. This makes the system more self-describing and reduces the risk for misunderstandings.

Context-Mapping

In our case study, we may find the following domains:

Although these domains should be as self-contained as possible, they still have to interact occasionally. Let's assume the `Ordering` domain for placing orders needs to interact with the `Catalogue` domain and a connected ERP system.

To define how these domains interact, we create a context map:



In principle, `Ordering` could have full access to `Catalog`. In this case, however, the domains aren't decoupled anymore and a change in `Catalog` could break `Ordering`.

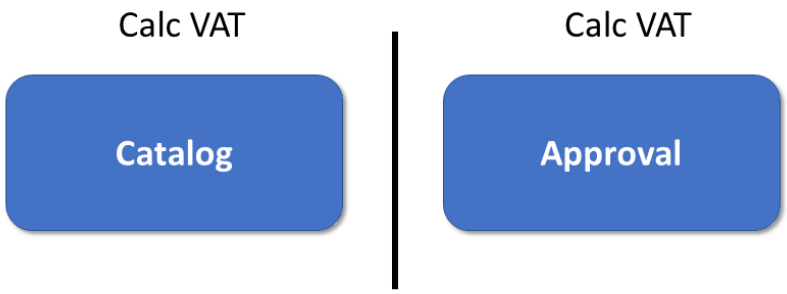
Strategic design defines several ways for dealing with such situations. For instance, in the context map shown above, `Catalog` offers an API (DDD calls it an `open/host service`) that exposes only

selected aspects for other domains. This API should be stable and backwards-compatible to prevent breaking other domains. Everything else is hidden behind this API and hence can be changed easily.

Since we cannot control the ERP system, `Ordering` uses a so-called anti-corruption layer (ACR) to access it. All calls to the ERP system are tunneled by this ACR. Hence, if something changes in the ERP system, we only need to update the ACR. Also, the ACR allows us to translate concepts from the ERP system into entities that make sense within our bounded context.

An existing system, like the shown ERP system, usually does not follow the idea of the bounded context. Instead, it contains several logical and intermingled ones.

Another strategy I want to stress here is `Separate Ways`. Specific tasks, like calculating VAT, could be separately implemented in several domains:



At first sight, this seems awkward because it leads to code redundancies and hence breaks the DRY principle (don't repeat yourself). Nevertheless, it can come in handy because it prevents a dependency on a shared library. Although preventing redundant code is important, limiting dependencies is vital because each dependency increases the overall complexity. Also, the more dependencies we have the more likely are braking changes when individual parts of our system evolve. Hence, it's good first to evaluate whether an additional dependency is truly needed.

Conclusion

Strategic design is about identifying loosely-coupled sub-domains. In each domain, we find ubiquitous language and concepts that only make sense within the domain's bounded context. A context map shows how those domains interact.

In the next chapter, we'll see we can implement those domains with Angular using an Nx[13]-based monorepo.

Learn more about this and further architecture topics regarding Angular and huge enterprise as well as industrial solution in our advanced Online Workshop[14]:

[13] https://nx.dev/
[14] https://www.angulararchitects.io/schulungen/advanced-angular-enterprise-anwendungen-und-architektur/

Advanced Angular Workshop

Save your ticket[15] for one of our **online or on-site** workshops now or request a company workshop[16] (online or In-House) for you and your team!

If you like our offer, keep in touch with us so that you don't miss anything.

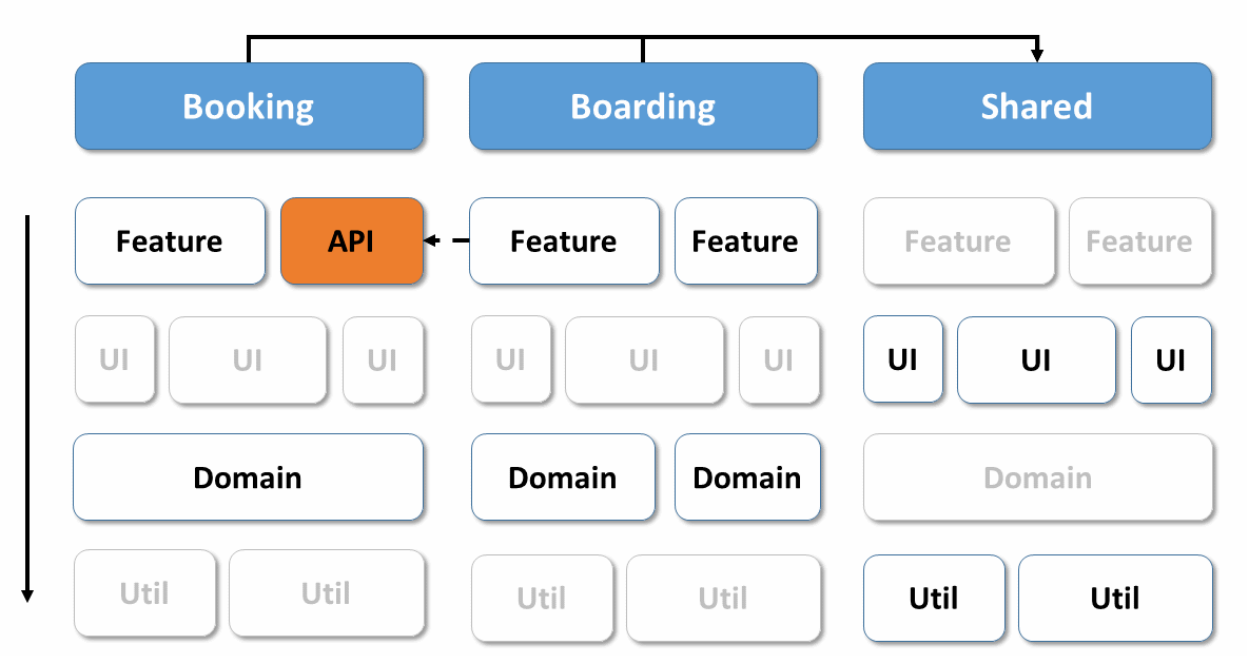
For this, you can subscribe to our newsletter[17] and/ or follow the book's author on Twitter[18].

[15] https://www.angulararchitects.io/en/angular-workshops/
[16] https://www.angulararchitects.io/en/contact/
[17] https://www.angulararchitects.io/en/subscribe/
[18] https://twitter.com/ManfredSteyer

Implementing Strategic Design with Nx Monorepos

In the previous chapter, I presented strategic design which allows to subdivide a software system into loosely coupled (sub-)domains. This chapter explores these domains' implementation with Angular and an Nx[19]-based monorepo.

The used architecture follows this architecture matrix:



As you see here, this matrix vertically cuts the application into sub domains. Also, it subdivides them horizontally into layers with different types of libraries.

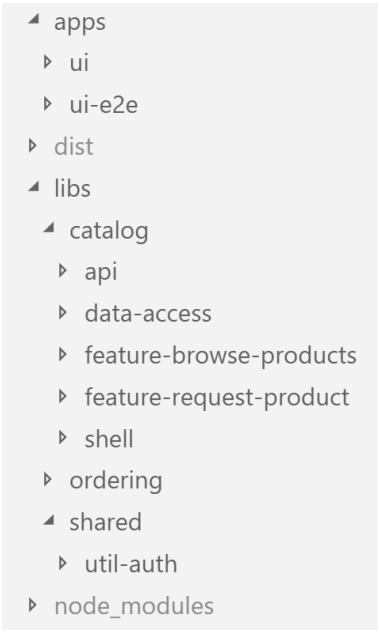
If you want to look at the underlying case study, you can find the source code here[20]

Implementation with Nx

We use an Nx-based workspace to implement the defined architecture. Nx is an extension for Angular CLI, which helps to break down a solution into different applications and libraries.

[19] https://nx.dev/
[20] https://github.com/manfredsteyer/strategic-design

By default, Nx puts all applications into an `apps` folder and the libraries into `libs`:



In our architecture, each domain is represented by a subfolder. For instance, within `libs`, you see subfolders like `catalog` or `ordering`. Everything that is shared across different folders goes into `shared`. Besides this, we use prefixes to denote the layer a specific library is part of. For instance, the prefix `feature-` defines that the library is part of the feature layer. Hence, the matrix shown above translates into this folder structure by using subfolders for the columns and prefixes for the rows.

Because such a workspace manages several applications and libraries in a common source code repository, there is also talk of a monorepo. This pattern is used extensively by Google and Facebook, among others, and has been the standard for the development of .NET solutions in the Microsoft ecosystem for about 20 years.

It allows source code sharing between project participants in a particularly simple way and prevents version conflicts by having only one central `node_modules` folder with dependencies. This arrangement ensures that, e.g., each library uses the same Angular version.

To create a new Nx-based Angular CLI project – a so-called workspace –, you can use the following command:

```
1 npm init nx-workspace e-proc
```

This command downloads a script which creates your workspace.

Within this workspace, you can use `ng generate` to add applications and libraries:

```
cd e-proc
ng generate app ui
ng generate lib feature-request-product --directory catalog
```

The `--directory` switch places the `feature-request-product` library in a folder `catalog` representing the sub domain with the same name.

Categories for Libraries

To reduce the cognitive load, the Nx team recommends to categorize libraries as follows:

- **feature**: Implements a use case with smart components
- **data-access**: Implements data accesses, e.g. via HTTP or WebSockets
- **ui**: Provides use case-agnostic and thus reusable components (dumb components)
- **util**: Provides helper functions

Please note the separation between smart and dumb components. Smart components within feature libraries are use case-specific. An example is a component which enables a product search.

On the contrary, dumb components do not know the current use case. They receive data via inputs, display it in a specific way, and issue events. Such presentational components "just" help to implement use cases and hence they are reusable. An example is a date-time picker, which is unaware of which use case it supports. Hence, it is available within all use cases dealing with dates.

In addition to this, I also use the following categories:

- **shell**: For an application that has multiple domains, a shell provides the entry point for a domain
- **api**: Provides functionalities exposed to other domains
- **domain**: Domain logic like calculating additional expenses (not used here), validations or facades for use cases and state management. I will come back to this in the next chapter.

Each category defines a layer in our architecture matrix. Also, each libary gets a prefix telling us to which category and hence layer it belongs to. This helps to maintain an overview.

Public APIs for Libraries

Each library has a public API exposed via a generated `index.ts` through which it publishes individual components. They hide all other components. These can be changed as desired:

```
export * from './lib/catalog-data-access.module';
export * from './lib/catalog-repository.service';
```

This structure is a fundamental aspect of good software design as it allows splitting into a public and a private part. Other libraries access the public part, so we have to avoid breaking changes as this would affect other parts of the system.

However, the private part can be changed at will, as long as the public part stays the same.

Check Accesses Between libraries

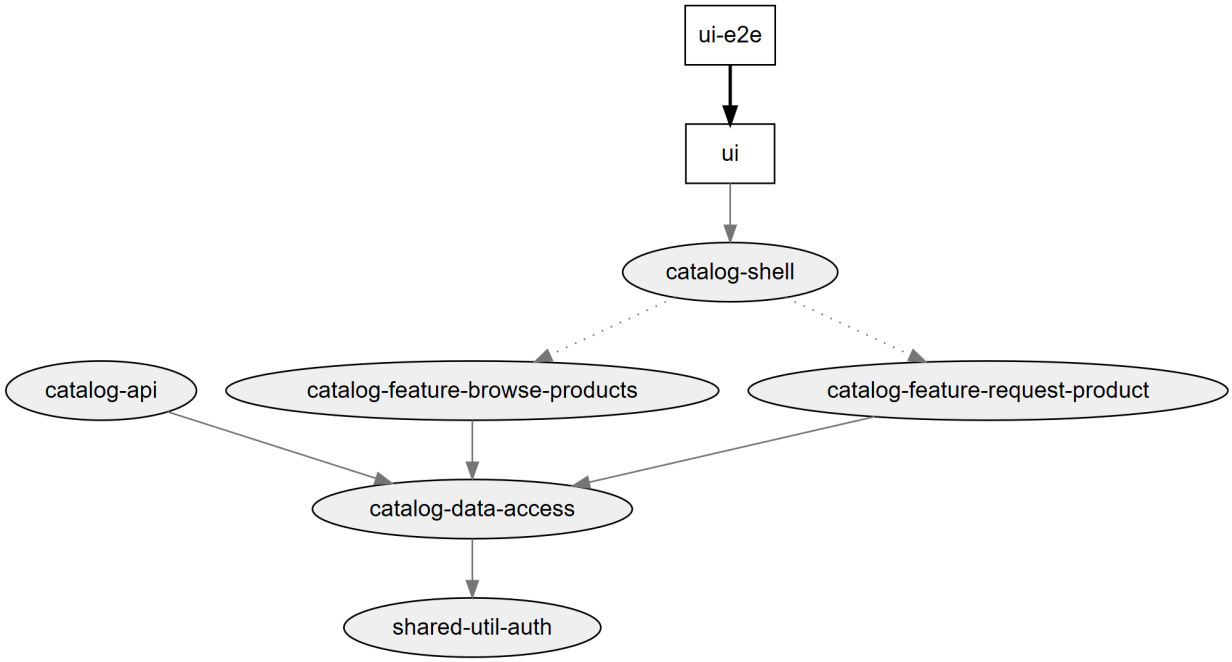
Minimizing the dependencies between individual libraries helps maintainability. This goal can be checked graphically using Nx' `dep-graph` script. To make our live easier, first, we install the Nx CLI:

```
npm i -g @nrwl/cli
```

Then, we can get the dependency graph via the following commend:

```
nx dep-graph
```

If we concentrate on the `Catalog` domain in our case study, the result is:



Access Restrictions for a Solid Architecture

Robust architecture requires limits to interactions between libraries. If there were no limits, we would have a heap of intermingled libraries where each change would affect all the other libraries, clearly negatively affecting maintainability.

Based on DDD, we have a few rules for communication between libraries to ensure consistent layering. For example, **each library may only access libraries from the same domain or shared libraries**.

Access to APIs such as `catalog-api` must be explicitly granted to individual domains.

We also define access restrictions on top of our layers shown in the matrix above. Each layer is only allowed to access layers below. For instance, a `feature` library can access `ui`, `domain` and `util` libraries.

To define such restrictions, Nx allows us to assign tags to each library. Based on these tags, we can define linting rules.

Tagging Libraries

In former Nx versions, the file `nx.json` defined the tags for our libraries:

```
"projects": {
  "ui": {
    "tags": ["scope:app"]
  },
  "ui-e2e": {
    "tags": ["scope:e2e"]
  },
  "catalog-shell": {
    "tags": ["scope:catalog", "type:shell"]
  },
  "catalog-feature-request-product": {
    "tags": ["scope:catalog", "type:feature"]
  },
  "catalog-feature-browse-products": {
    "tags": ["scope:catalog", "type:feature"]
  },
  "catalog-api": {
    "tags": ["scope:catalog", "type:api", "name:catalog-api"]
  },
  "catalog-data-access": {
    "tags": ["scope:catalog", "type:data-access"]
```

```
  },
  "shared-util-auth": {
    "tags": ["scope:shared", "type:util"]
  }
}
```

In current versions, Nx puts the same information into the `angular.json` that already contains a section for each library and application. Alternatively, these tags can be specified when setting up the applications and libraries.

According to a suggestion from the Nx team, the domains get the prefix `scope`, and the library types receive the prefix `type`. Prefixes of this type are intended to improve readability and can be freely assigned.

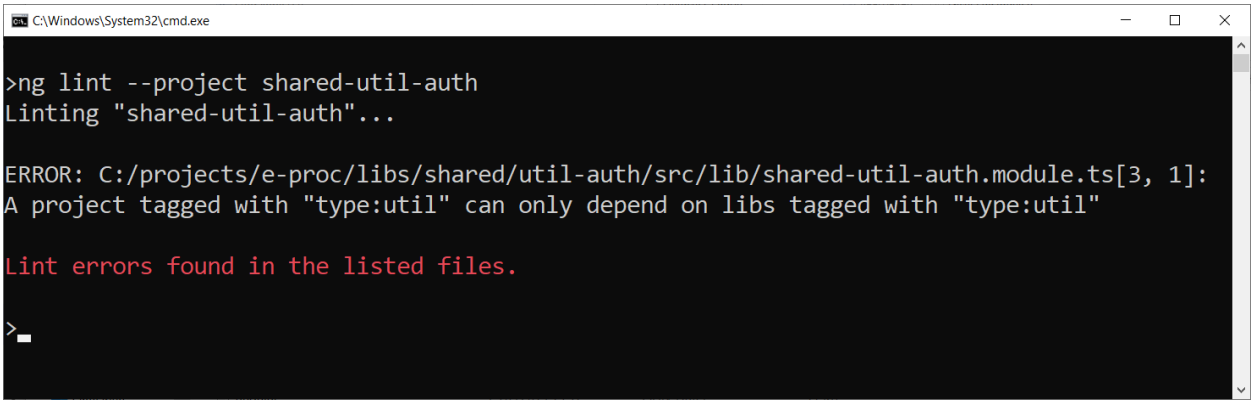
Defining Linting Rules Based Upon Tags

To enforce access restrictions, Nx comes with its own linting rules. As usual, we configure these rules within `.eslintrc.json`:

```
"@nrwl/nx/nx-enforce-module-boundaries": [
  "error",
  {
    "allow": [],
    "depConstraints": [
      { "sourceTag": "scope:app",
        "onlyDependOnLibsWithTags": ["type:shell"] },
      { "sourceTag": "scope:catalog",
        "onlyDependOnLibsWithTags": ["scope:catalog", "scope:shared"] },
      { "sourceTag": "scope:shared",
        "onlyDependOnLibsWithTags": ["scope:shared"] },
      { "sourceTag": "scope:booking",
        "onlyDependOnLibsWithTags":
          ["scope:booking", "scope:shared", "name:catalog-api"] },

      { "sourceTag": "type:shell",
        "onlyDependOnLibsWithTags": ["type:feature", "type:util"] },
      { "sourceTag": "type:feature",
        "onlyDependOnLibsWithTags": ["type:data-access", "type:util"] },
      { "sourceTag": "type:api",
        "onlyDependOnLibsWithTags": ["type:data-access", "type:util"] },
      { "sourceTag": "type:util",
        "onlyDependOnLibsWithTags": ["type:util"] }
    ]
```

```
25    }
26  ]
```

To test these rules, just call `ng lint` on the command line:



Development environments such as WebStorm/IntelliJ, or Visual Studio Code show such violations while typing. In the latter case, you need a respective eslint plugin.

> **Hint**: Run the linter before checking in your source code or before merging a pull request against your main branch. If you automate this check you can enforce your architecture matrix and prevent source code that violates your architecture.

Your Architecture by the Push of a Button: The DDD-Plugin

While the architecture described here already quite often proved to be very in handy, building it by hand includes several repeating tasks like creating libs of various kinds, defining access restrictions, or creating buildings blocks like facades.

To automate these tasks, you can use our Nx plugin `@angular-architects/ddd`. It provides the following features:

- Generating domains with domain libraries including facades, models, and data services
- Generating feature libraries including feature components using facades
- Adding linting rules for access restrictions between domains as proposed by the Nx team
- Adding linting rules for access restrictions between layers as proposed by the Nx team (supports tslint and eslint)
- Optional: Generating a skeleton for using NGRX (`--ngrx` switch)

You can use `ng add` for adding it to your Nx workspace:

```
ng add @angular-architects/ddd
```

Alternatively, you can `npm install` it and use its `init` schematic:

```
npm i @angular-architects/ddd
ng g @angular-architects/ddd:init
```

Then, you can easily create domains, features, and libraries of other kinds:

```
ng g @angular-architects/ddd:domain booking --addApp
ng g @angular-architects/ddd:domain boarding --addApp
ng g @angular-architects/ddd:feature search --domain booking --entity flight
ng g @angular-architects/ddd:feature cancel --domain booking
ng g @angular-architects/ddd:feature manage --domain boarding
```

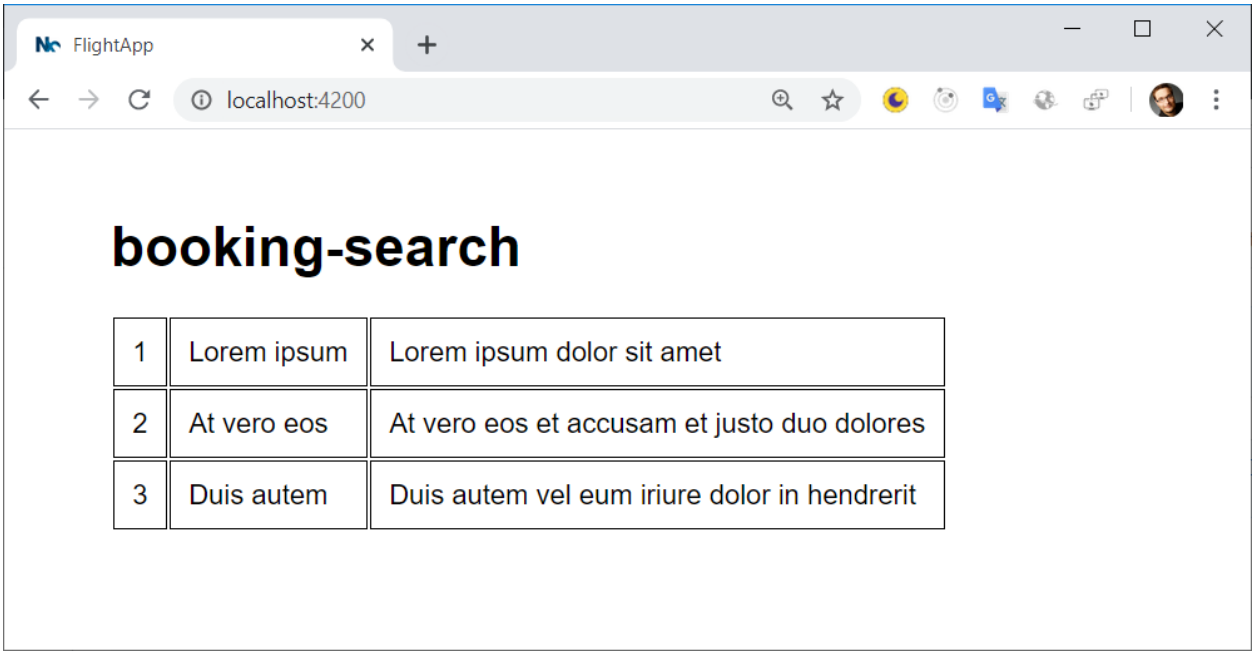
For NGRX support, just add the `--ngrx` switch:

```
ng g @angular-architects/ddd:domain booking --addApp --ngrx
ng g @angular-architects/ddd:feature search --domain booking --entity flight --ngrx
[...]
```

To see that the skeleton works end-to-end, call the generated feature component in your `app.component.html`:

```
<booking-search></booking-search>
```

You don't need any TypeScript or Angular imports. The plugin already took care about that. After running the example, you should see something like this:



Result proving that the generated skeleton works end-to-end

Conclusion

Strategic design is a proven way to break an application into self-contained domains. These domains have a ubiquitous language which all stakeholders must use consistently.

The CLI extension Nx provides a very elegant way to implement these domains with different domain-grouped libraries. To restrict access by other domains and to reduce dependencies, it allows setting access restrictions to individual libraries.

These access restrictions help ensure a loosely coupled system which is easier to maintain as a sole change only affects a minimum of other parts of the system.

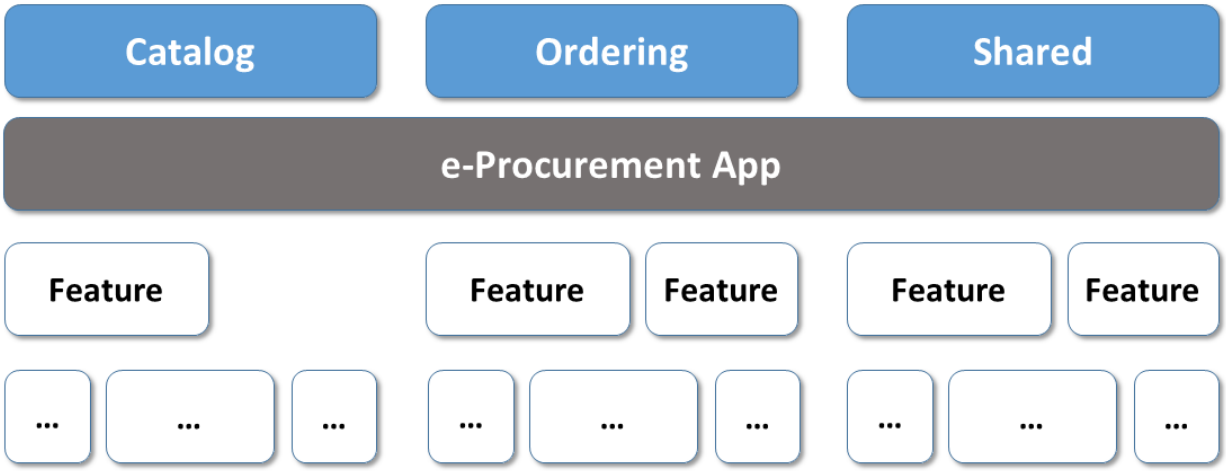
From Domains to Microfrontends

Let's assume you've identified the sub-domains for your system. The next question is how to implement them.

One option is to implement them within a large application – aka a deployment monolith. The second is to provide a separate application for each domain. Such applications are called micro frontends.

Deployment Monoliths

A deployment monolith is an integrated solution comprising different domains:



This approach supports a consistent UI and leads to optimized bundles by compiling everything together.

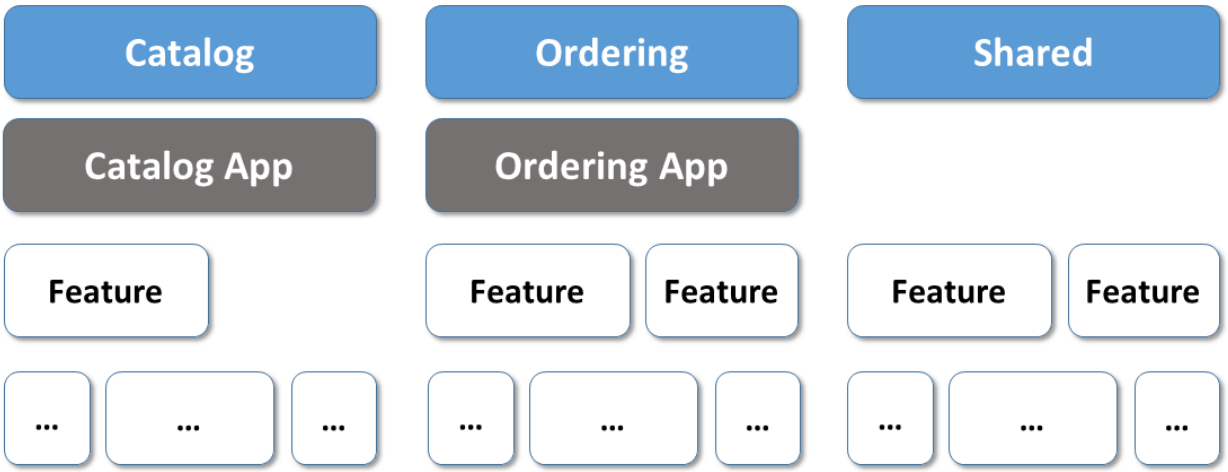
A team responsible for a specific sub-domain must coordinate with other sub-domain teams. They have to agree on an overall architecture and the leading framework. Also, they need to define a common policy for updating dependencies.

It is tempting to reuse parts of other domains. However, this may lead to higher coupling and – eventually – to breaking changes. To prevent this, we've used Nx and access restrictions between libraries in the last chapter.

Micro Frontends

To further decouple your system, you could split it into several smaller applications. If we assume that use cases do not overlap your sub-domains' boundaries, this can lead to more autarkic teams

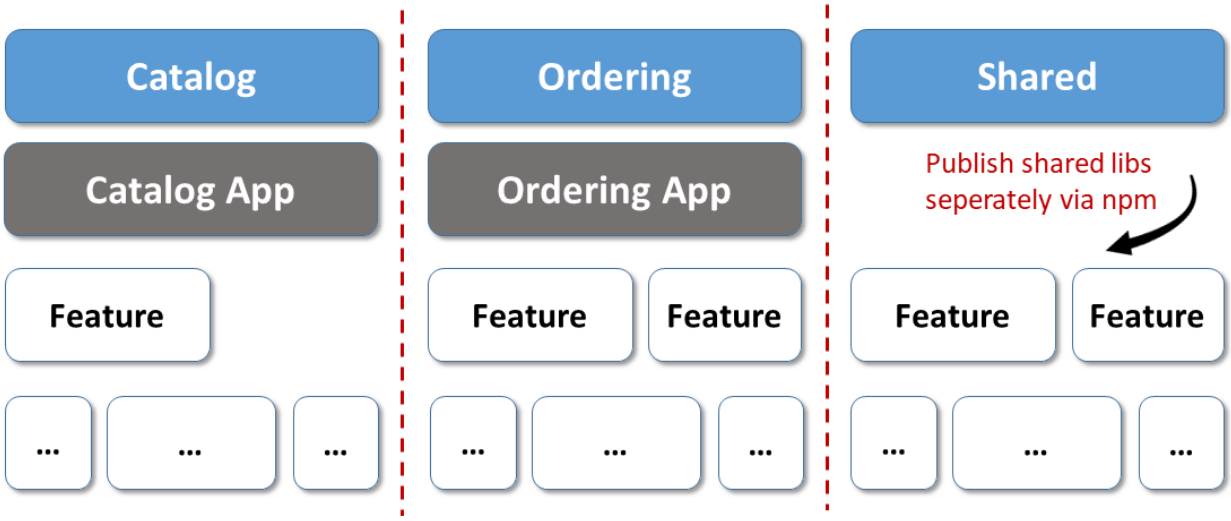
and applications which are separately deployable.



You now have something called micro frontends. Micro Frontends allow for autarkic teams: Each team can choose their architectural style, their technology stack, and they can even decide when to update to newer framework versions. They can use "the best technology" for the requirements given within the current sub-domain.

The option for deciding which frameworks to use per micro frontend is interesting when developing applications over the long term. If, for instance, a new framework appears in five years, we can use it to implement the next domain.

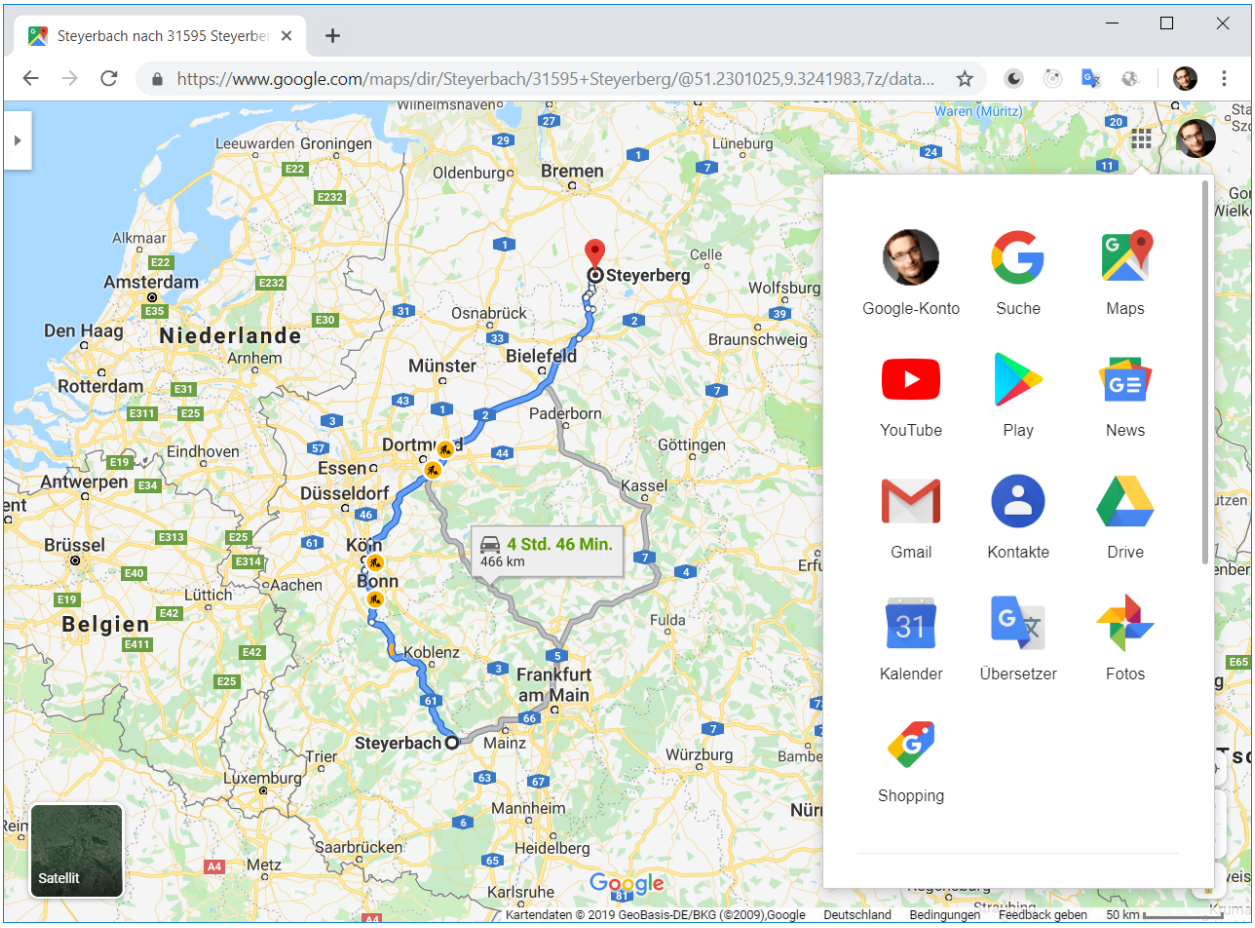
If you seek even more isolation between your sub-domains and the teams responsible for them, you could put each sub-domain into its individual repository:



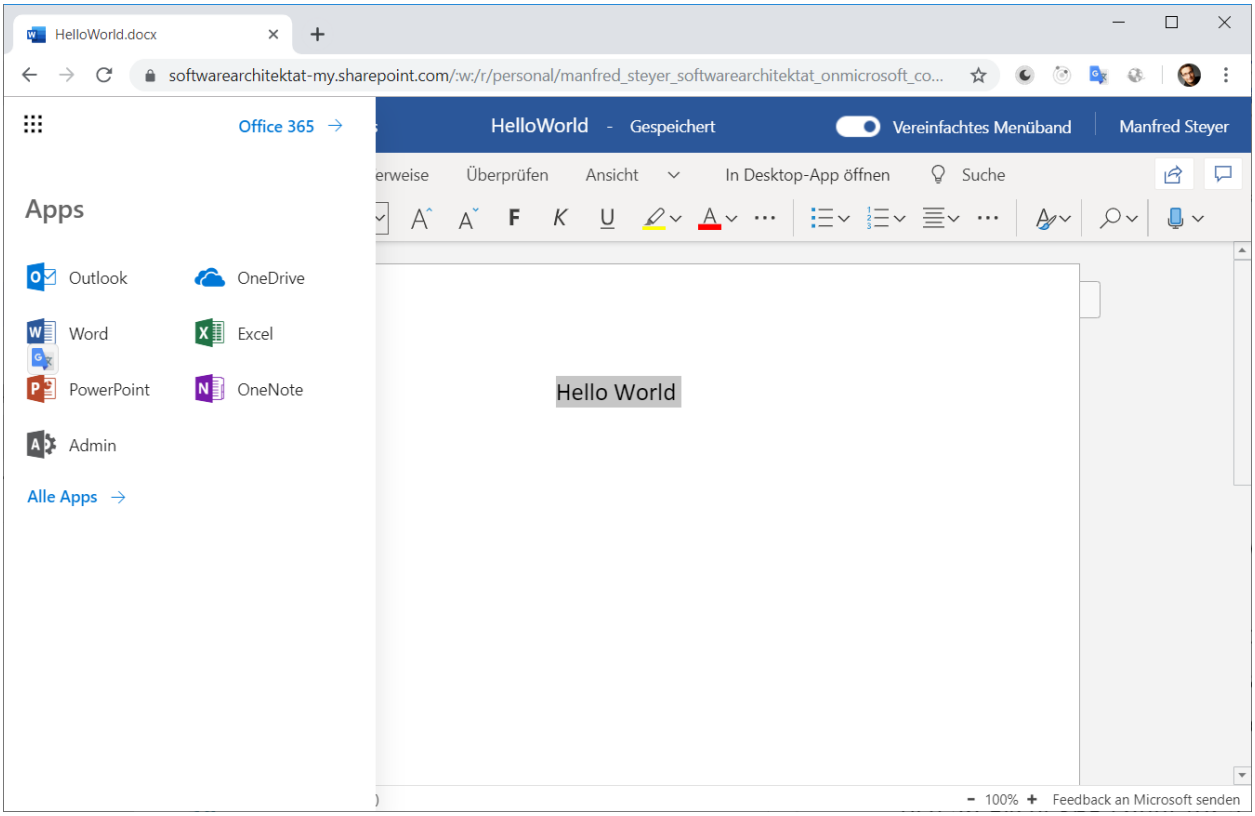
However, this has costs. Now you have to deal with shipping your shared libraries via npm. This comes with some efforts and forces you to version your libraries. You need to make sure that each micro frontend uses the right version. Otherwise, you end up with version conflicts.

UI Composition with Hyperlinks

Splitting a huge application into several micro frontends is only one side of the coin. Your users want to have one integrated solution. Hence, you have to find ways to integrate the different applications into one large system. Hyperlinks are one simple way to accomplish this:



This approach fits product suites like Google or Office 365 well:



Each domain is a self-contained application here. This structure works well because we don't need many interactions between the domains. If we needed to share data, we could use the backend. Using this strategy, Word 365 can use an Excel 365 sheet for a series letter.

This approach has several advantages:

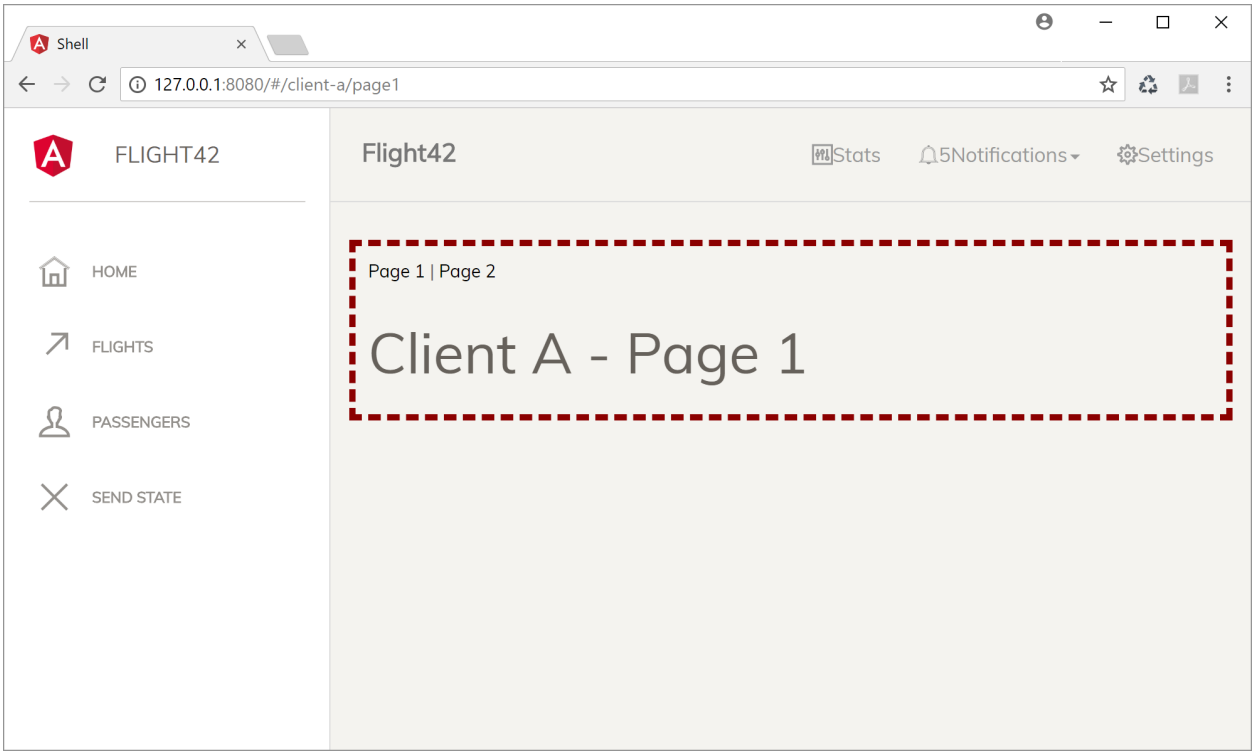
- It is simple
- It uses SPA frameworks as intended
- We get optimised bundles per domain

However, there are some disadvantages:

- We lose our state when switching to another application
- We have to load another application – which we wanted to prevent with SPAs
- We have to work to get a standard look and feel (we need a universal design system).

UI Composition with a Shell

Another much-discussed approach is to provide a shell that loads different single-page applications on-demand:



In the screenshot, the shell loads the microfrontend with the red border into its working area. Technically, it simply loads the microfrontend bundles on demand. The shell then creates an element for the microfrontend's root element:

```
const script = document.createElement('script');
script.src = 'assets/external-dashboard-tile.bundle.js';
document.body.appendChild(script);

const clientA = document.createElement('client-a');
clientA['visible'] = true;
document.body.appendChild(clientA);
```

Instead of bootstrapping several SPAs, we could also use iframes. While we all know the enormous disadvantages of iframes and have strategies to deal with most of them, they do provide two useful features:

1. Isolation: A microfrontend in one iframe cannot influence or hack another microfrontend in another iframe. Hence, they are handy for plugin systems or when integrating applications from other vendors.
2. They also allow the integration of legacy systems.

You can find a library that compensates most of the disadvantages of iframes for intranet applications

here[21]. Even SAP has an iframe-based framework they use for integrating their products. It's called Luigi[22] and you can find it here[23].

The shell approach has the following advantages:

- The user has an integrated solution consisting of different microfrontends.
- We don't lose the state when navigating between domains.

The disadvantages are:

- If we don't use specific tricks (outlined in the next chapter), each microfrontend comes with its own copy of Angular and the other frameworks, increasing the bundle sizes.
- We have to implement infrastructure code to load microfrontends and switch between them.
- We have to work to get a standard look and feel (we need a universal design system).

The Hero: Module Federation

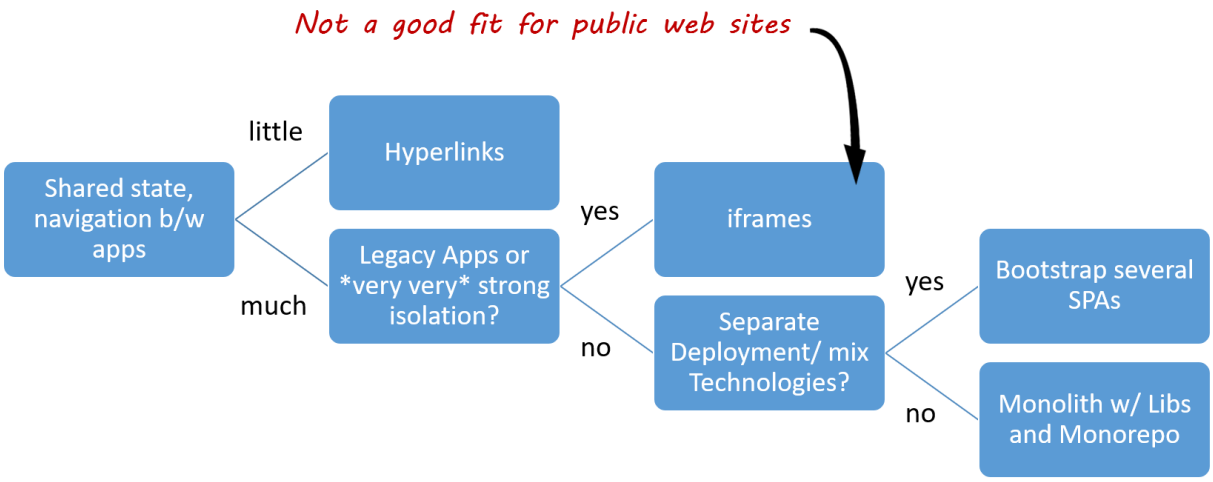
A quite new solution that compensates most of the issues outlined above is Webpack Module Federation. It allows to load code from an separately compiled and deployed application and is very straight forward. IMHO, currently, this is the best way for implementing a shell-based architecture. Hence, the **next chapters** concentrate on Module Federation.

Finding a Solution

Choosing between a deployment monolith and different approaches for microfrontends is tricky because each option has advantages and disadvantages.

I've created the following decision tree, which also sums up the ideas outlined in this chapter:

[21] https://www.npmjs.com/package/@microfrontend/common
[22] https://github.com/SAP/luigi
[23] https://github.com/SAP/luigi



Decision tree for Micro Frontends vs. Deployment Monoliths

As the implementation of a deployment monolith and the hyperlink approach is obvious, the next chapter discusses how to implement a shell.

Conclusion

There are several ways to implement microfrontends. All have advantages and disadvantages. Using a consistent and optimised deployment monolith can be the right choice.

It's about knowing your architectural goals and about evaluating the consequences of architectural candidates.

The Microfrontend Revolution: Using Module Federation with Angular

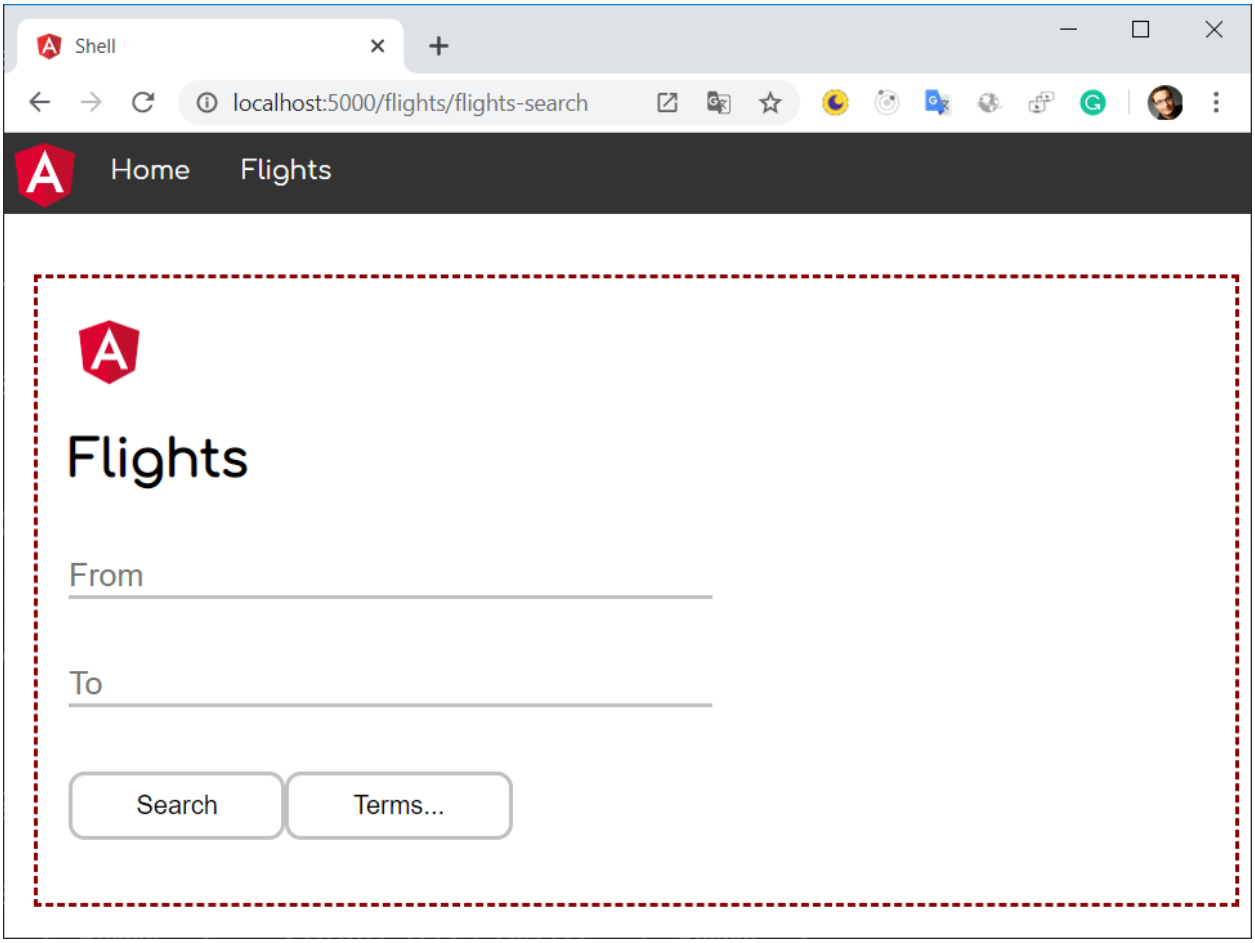
In the past, when implementing microfrontends, you had to dig a little into the bag of tricks. One reason is surely that build tools and frameworks did not know this concept. Fortunately, Webpack 5 initiated a change of course here.

Webpack 5 comes with an implementation provided by the webpack contributor Zack Jackson. It's called Module Federation and allows referencing parts of other applications not known at compile time. These can be microfrontends that have been compiled separately. In addition, the individual program parts can share libraries with each other, so that the individual bundles do not contain any duplicates.

In this chapter, I will show how to use Module Federation using a simple example.

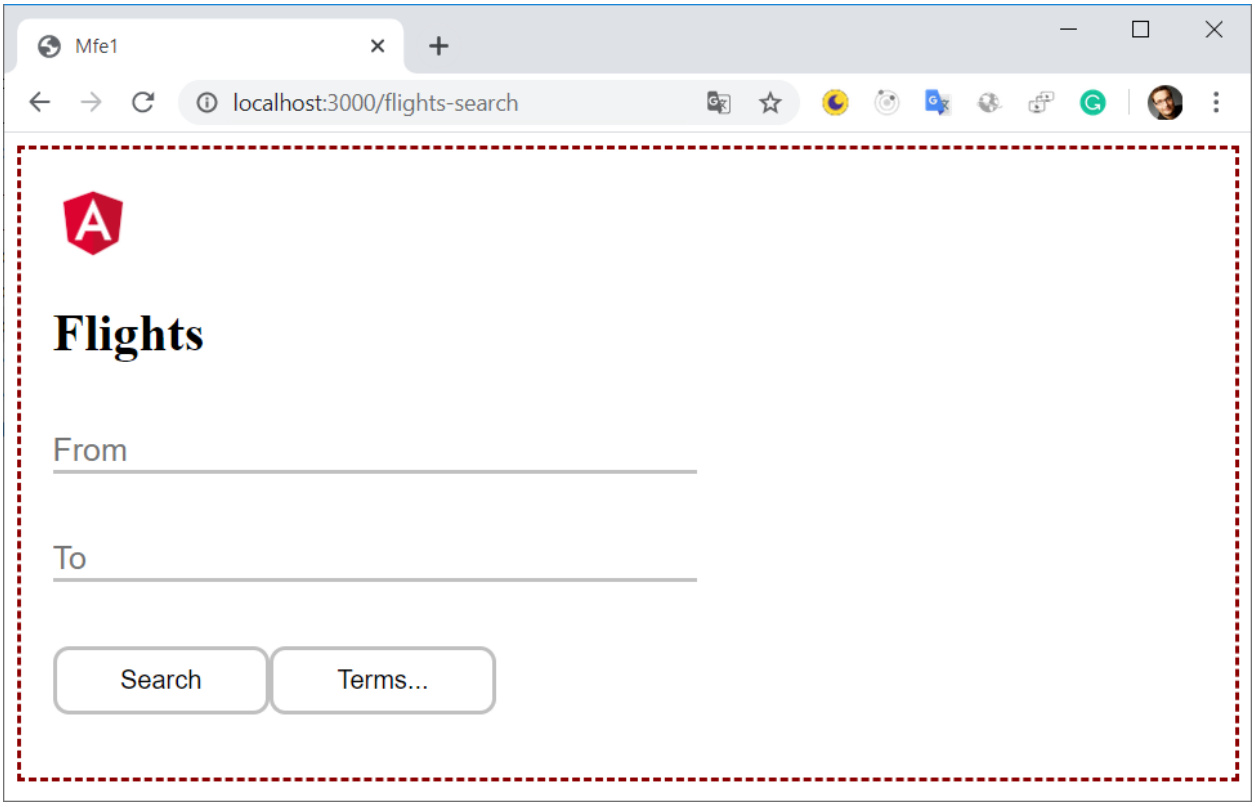
Example

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



Shell

The loaded microfrontend is shown within the red dashed border. Also, the microfrontend can be used without the shell:



Microfrontend without Shell

The source code[24] of the used example can be found in my GitHub account[25].

> **Important**: This book is written for Angular and **Angular CLI 13.1** and higher. Make sure you have a fitting version if you try out the examples outlined here! For more details on the differences/ migration to Angular 13.1 please see this migration guide[26].

Activating Module Federation for Angular Projects

The case study presented here assumes that both, the shell and the microfrontend are projects in the same Angular workspace. For getting started, we need to tell the CLI to use module federation when building them. However, as the CLI shields webpack from us, we need a custom builder.

The package @angular-architects/module-federation[27] provides such a custom builder. To get started, you can just "ng add" it to your projects:

[24] https://github.com/manfredsteyer/module-federation-with-angular
[25] https://github.com/manfredsteyer/module-federation-with-angular
[26] https://github.com/angular-architects/module-federation-plugin/blob/main/migration-guide.md
[27] https://www.npmjs.com/package/@angular-architects/module-federation

```
ng add @angular-architects/module-federation --project shell --port 5000
ng add @angular-architects/module-federation --project mfe1 --port 3000
```

While it's obvious that the project `shell` contains the code for the `shell`, `mfe1` stands for *Micro Frontend 1*.

The command shown does several things:

- Generating the skeleton of an `webpack.config.js` for using module federation
- Installing a custom builder making webpack within the CLI use the generated `webpack.config.js`.
- Assigning a new port for ng serve so that several projects can be served simultaneously.

Please note that the `webpack.config.js` is only a **partial** webpack configuration. It only contains stuff to control module federation. The rest is generated by the CLI as usual.

The Shell (aka Host)

Let's start with the shell which would also be called the host in module federation. It uses the router to lazy load a `FlightModule`:

```
export const APP_ROUTES: Routes = [
    {
      path: '',
      component: HomeComponent,
      pathMatch: 'full'
    },
    {
      path: 'flights',
      loadChildren: () => import('mfe1/Module').then(m => m.FlightsModule)
    },
];
```

However, the path `mfe1/Module` which is imported here, **does not exist** within the shell. It's just a virtual path pointing to another project.

To ease the TypeScript compiler, we need a typing for it:

```
// decl.d.ts
declare module 'mfe1/Module';
```

Also, we need to tell webpack that all paths starting with `mfe1` are pointing to an other project. This can be done by using the `ModuleFederationPlugin` in the generated `webpack.config.js`:

```
const ModuleFederationPlugin =
  require("webpack/lib/container/ModuleFederationPlugin");
const mf =
  require("@angular-architects/module-federation/webpack");
const path = require("path");
const share = mf.share;

[...]

module.exports = {
  output: {
    uniqueName: "shell",
    publicPath: "auto"
  },
  optimization: {
    runtimeChunk: false
  },
  resolve: {
    alias: {
      ...sharedMappings.getAliases(),
    }
  },
  experiments: {
     outputModule: true
  },
  plugins: [
    new ModuleFederationPlugin({
      library: { type: "module" },
      remotes: {
          "mfe1": "mfe1@http://localhost:3000/remoteEntry.js",
      },

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

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

        ...sharedMappings.getDescriptors()
      })

    }),
    sharedMappings.getPlugin(),
  ],
};
```

The `experiments` section is temporarily needed to activate a bug fix in webpack. The `remotes` section maps the URL `mfe1` to the remote. For this, it points to the path where the remote can be found – or to be more precise: to its remote entry. This is a tiny file generated by webpack when building the remote. Webpack loads it at runtime to get all the information needed for interacting with the microfrontend.

While specifying the remote entry's URL that way is convenient for development, we need a more dynamic approach for production. The next chapter deals with this.

The property `shared` contains the names of libraries our shell shares with the micro frontend(s). The combination of `singleton: true` and `strictVersion: true` makes webpack emit a runtime error when the shell and the micro frontend(s) need different incompatible versions (e. g. two different major versions). If we skipped `strictVersion` or set it to `false`, webpack would only emit a warning at runtime. More on this can be found in the chapter on version mismatches.

The setting `requiredVersion: 'auto'` is a little extra provided by the `@angular-architects/module-federation` plugin. It looks up the used version in your `package.json`. This prevents several issues.

> The helper function share used in this generated configuration replaces the value `'auto'` with the version found in your `package.json`.

In addition to the settings for the `ModuleFederationPlugin`, we also need to place some options in the `output` section.

The `uniqueName` is used to represent 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.

The Micro Frontend (aka Remote)

The micro frontend – also referred to as a *remote* with terms of module federation – looks like an ordinary Angular application. It has routes defined within in the `AppModule`:

```
export const APP_ROUTES: Routes = [
    { path: '', component: HomeComponent, pathMatch: 'full'}
];
```

Also, there is a `FlightsModule`:

```
@NgModule({
  imports: [
    CommonModule,
    RouterModule.forChild(FLIGHTS_ROUTES)
  ],
  declarations: [
    FlightsSearchComponent
  ]
})
export class FlightsModule { }
```

This module has some routes of its own:

```
export const FLIGHTS_ROUTES: Routes = [
    {
      path: 'flights-search',
      component: FlightsSearchComponent
    }
];
```

In order to make it possible to load the `FlightsModule` into the shell, we also need to reference the `ModuleFederationPlugin` in the remote's webpack configuration:

```
const ModuleFederationPlugin =
  require("webpack/lib/container/ModuleFederationPlugin");
const mf =
  require("@angular-architects/module-federation/webpack");
const path = require("path");

const share = mf.share;

[...]

module.exports = {
  output: {
    uniqueName: "mfe1",
    publicPath: "auto"
  },
  optimization: {
    runtimeChunk: false
  },
  resolve: {
    alias: {
      ...sharedMappings.getAliases(),
    }
  },
  experiments: {
    outputModule: true
  },
  plugins: [
    new ModuleFederationPlugin({
        library: { type: "module" },

        name: "mfe1",
        filename: "remoteEntry.js",
        exposes: {
            './Module': './projects/mfe1/src/app/flights/flights.module.ts',
        },
        shared: share({
          "@angular/core": {
            singleton: true,
            strictVersion: true,
            requiredVersion: 'auto'
          },
          "@angular/common": {
            singleton: true,
```

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

          ...sharedMappings.getDescriptors()
        })

    }),
    sharedMappings.getPlugin(),
  ],
};
```

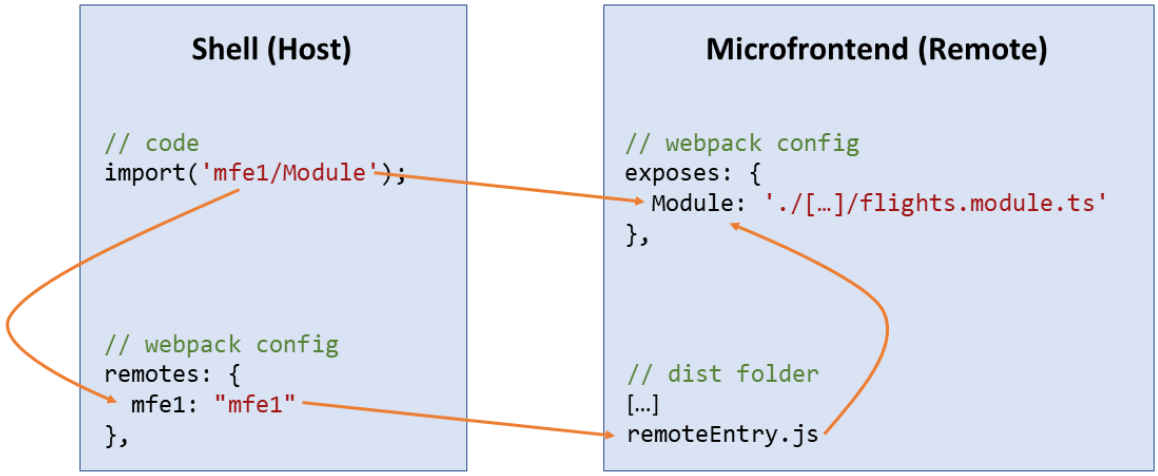
The configuration shown here exposes the `FlightsModule` under the public name `Module`. The section `shared` points to the libraries shared with the shell.

Trying it out

To try everything out, we just need to start the shell and the microfrontend:

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

Then, when clicking on `Flights` in the shell, the micro frontend is loaded:



Connecting the Shell and the Microfrontend

Hint: To start several projects with one command, you can use the npm package concurrently[28].

A little further detail

Ok, that worked quite well. But have you had a look into your `main.ts`?

It just looks like this:

```
1  import('./bootstrap')
2          .catch(err => console.error(err));
```

The code you normally find in the file `main.ts` was moved to the `bootstrap.ts` file loaded here. All of this was done by the `@angular-architects/module-federation` plugin.

While this doen't seem to make a lot of sense at first sight, it's a typical pattern you find in Module Federation-based applications. The reason is that Module Federation needs to decide which version of a shared library to load. If the shell, for instance, is using version 12.0 and one of the micro frontends is already built with version 12.1, it will decide to load the latter one.

To look up the needed meta data for this decision, Module Fedaration squeezes itself into dynamic imports like this one here. Other than the more tradtional static imports, dynamic imports are asynchronous. Hence, Module Federation can decide on the versions to use and actually load them.

More details on this can be found in the chapter on dynamic federation.

[28] https://www.npmjs.com/package/concurrently

Conclusion and Evaluation

The implementation of microfrontends has so far involved numerous tricks and workarounds. Webpack Module Federation finally provides a simple and solid solution for this. To improve performance, libraries can be shared and strategies for dealing with incompatible versions can be configured.

It is also interesting that the microfrontends are loaded by Webpack under the hood. There is no trace of this in the source code of the host or the remote. This simplifies the use of module federation and the resulting source code, which does not require additional microfrontend frameworks.

However, this approach also puts more responsibility on the developers. For example, you have to ensure that the components that are only loaded at runtime and that were not yet known when compiling also interact as desired.

One also has to deal with possible version conflicts. For example, it is likely that components that were compiled with completely different Angular versions will not work together at runtime. Such cases must be avoided with conventions or at least recognized as early as possible with integration tests.

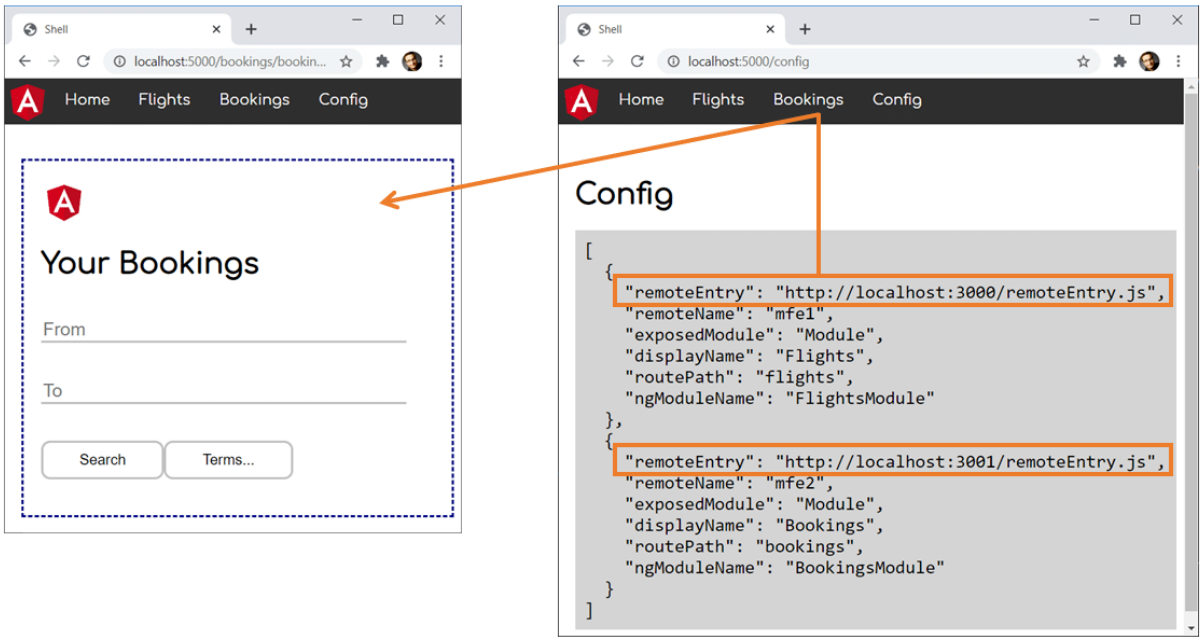
Dynamic Module Federation

In the chapter, I've shown how to use Webpack Module Federation for loading separately compiled microfrontends into a shell. As the shell's webpack configuration describes the microfrontends, we already needed to know them when compiling it.

In this chapter, I'm assuming a more dynamic situation where the shell does not know the microfrontends or even their number upfront. Instead, this information is provided at runtime via a lookup service.

Important: This book is written for Angular and **Angular CLI 13.1** and higher. Make sure you have a fitting version if you try out the examples outlined here! For more details on the differences/ migration to Angular 13.1 please see this migration guide[29].

The following image displays the idea used describe in this chapter:



The shell loads a microfrontend it is informed about on runtime

For all microfrontends the shell gets informed about at runtime it displays a menu item. When clicking it, the microfrontend is loaded and displayed by the shell's router.

As usual, the source code[30] used here can be found in my GitHub account[31].

[29] https://github.com/angular-architects/module-federation-plugin/blob/main/migration-guide.md
[30] https://github.com/manfredsteyer/module-federation-with-angular-dynamic.git
[31] https://github.com/manfredsteyer/module-federation-with-angular-dynamic.git

Module Federation Config

Let's start with the shell's Module Federation configuration. In this scenario, it uses the `ModuleFederationPlugin` as follows:

```
[...]
new ModuleFederationPlugin({
  library: { type: "module" },

  // No remote configured upfront anymore!
  remotes: { },

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

    ...sharedMappings.getDescriptors()
  })

}),
```

We don't define any remotes (microfrontends) upfront but configure the packages we want to share with the remotes we get informed about at runtime.

As mentioned in the previous chapter, the combination of `singleton: true` and `strictVersion: true` makes webpack emit a runtime error when the shell and the micro frontend(s) need different incompetible versions (e. g. two different major versions). If we skipped `strictVersion` or set it to `false`, webpack would only emit a warning at runtime.

The configuration of the microfrontends, however, looks like in the previous chapter:

```
const ModuleFederationPlugin =
  require("webpack/lib/container/ModuleFederationPlugin");
const mf =
  require("@angular-architects/module-federation/webpack");
const path = require("path");

const share = mf.share;

[...]

module.exports = {
  output: {
    uniqueName: "mfe1",
    publicPath: "auto"
  },
  optimization: {
    runtimeChunk: false
  },
  resolve: {
    alias: {
      ...sharedMappings.getAliases(),
    }
  },
  experiments: {
    outputModule: true
  },
  plugins: [
    new ModuleFederationPlugin({
        library: { type: "module" },

        name: "mfe1",
        filename: "remoteEntry.js",
        exposes: {
            './Module': './projects/mfe1/src/app/flights/flights.module.ts',
        },
        shared: share({
```

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

            ...sharedMappings.getDescriptors()
          })

    }),
    sharedMappings.getPlugin(),
  ],
};
```

Routing to Dynamic Microfrontends

To dynamically load a microfrontend at runtime, we can use the helper function loadRemoteModule provided by the `@angular-architects/module-federation` plugin:

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

[...]

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

Please note that beginning with Angular 13.1 we need to specificy type: 'module', as CLI 13 started with emitting EcmaScript modules instead of script files. For older versions you could use the following options:

```
// Before Angular 13:
// const routes: Routes = [
//     [...]
//     {
//         path: 'flights',
//         loadChildren: () =>
//             loadRemoteModule({
//                 type: 'script',
//                 remoteEntry: // 'http://localhost:3000/remoteEntry.js',
//                 remoteName: 'mfe1',
//                   // The remoteName is defined in the
//                   // remote's webpack config
//                 exposedModule: './Module'
//             })
//             .then(m => m.FlightsModule)
//     },
//     [...]
//];
```

As you might have noticed, we're just switching out the dynamic `import` normally used here by a

call to `loadRemoteModule` which also works with key data not known at compile time. The latter one uses the webpack runtime api to get hold of the remote on demand.

Improvement for Dynamic Module Federation

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. More details about this be found in the chapter on version mismatches.

To achieve this goal, let's use the helper function `loadRemoteEntry` in our `main.ts`

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

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

Also here, you need to set `type` to `module` for Angular CLI 13.1 or above. The call for version below 13 looks like this:

```
[...]
// For Angular 12
// loadRemoteEntry({
//   type: 'script',
//   remoteEntry: 'http://localhost:3000/remoteEntry.js',
//   remoteName: 'mfe1'
// })
[...]
```

Here, we need to remember, that the `@angular-architects/module-federation` plugin moves the contents of the original `main.ts` into the `bootstrap.ts` file. Also, it loads the `bootstrap.ts` with

a **dynamic import** in the `main.ts`. This is necessary because the dynamic import gives Module Federation the needed time to negotiate the verions of the shared libraries to use with all the remotes.

Also, loading the remote entry needs to happen before importing `bootstrap.ts` so that its metadata can be respected during the negotiation.

Bonus: Dynamic Routes for Dynamic Microfrontends

There might be situations where you don't even know the number of microfrontends upfront. Hence, we also need an approach for setting up the routes dynamically.

For this, I've defined a `Microfrontend` type holding all the key data for the routes:

```
export type Microfrontend = LoadRemoteModuleOptions & {
    displayName: string;
    routePath: string;
    ngModuleName: string;
}
```

`LoadRemoteModuleOptions` is the type that's passed to the above-discussed `loadRemoteModule` function. The type `Microfrontend` adds some properties we need for the dynamic routes and the hyperlinks pointing to them:

- **displayName**: Name that should be displayed within the hyperlink leading to route in question.
- **routePath**: Path used for the route.
- **ngModuleName**: Name of the Angular Module exposed by the remote.

For loading this key data, I'm using a `LookupService`:

```
@Injectable({ providedIn: 'root' })
export class LookupService {
    lookup(): Promise<Microfrontend[]> {
        [...]
    }
}
```

After receiving the `Microfrontend` array from the `LookupService`, we can build our dynamic routes:

```
export function buildRoutes(options: Microfrontend[]): Routes {

    const lazyRoutes: Routes = options.map(o => ({
        path: o.routePath,
        loadChildren: () => loadRemoteModule(o).then(m => m[o.ngModuleName])
    }));

    return [...APP_ROUTES, ...lazyRoutes];
}
```

This function creates one route per array entry and combines it with the static routes in `APP_ROUTES`.

Everything is put together in the shell's `AppComponent`. It's `ngOnInit` method fetches the key data, builds routes for it, and resets the Router's configuration with them:

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

  microfrontends: Microfrontend[] = [];

  constructor(
    private router: Router,
    private lookupService: LookupService) {
  }

  async ngOnInit(): Promise<void> {
    this.microfrontends = await this.lookupService.lookup();
    const routes = buildRoutes(this.microfrontends);
    this.router.resetConfig(routes);
  }
}
```

Besides this, the `AppComponent` is also rendering a link for each route:

```
<li *ngFor="let mfe of microfrontends">
    <a [routerLink]="mfe.routePath">{{mfe.displayName}}</a>
</li>
```

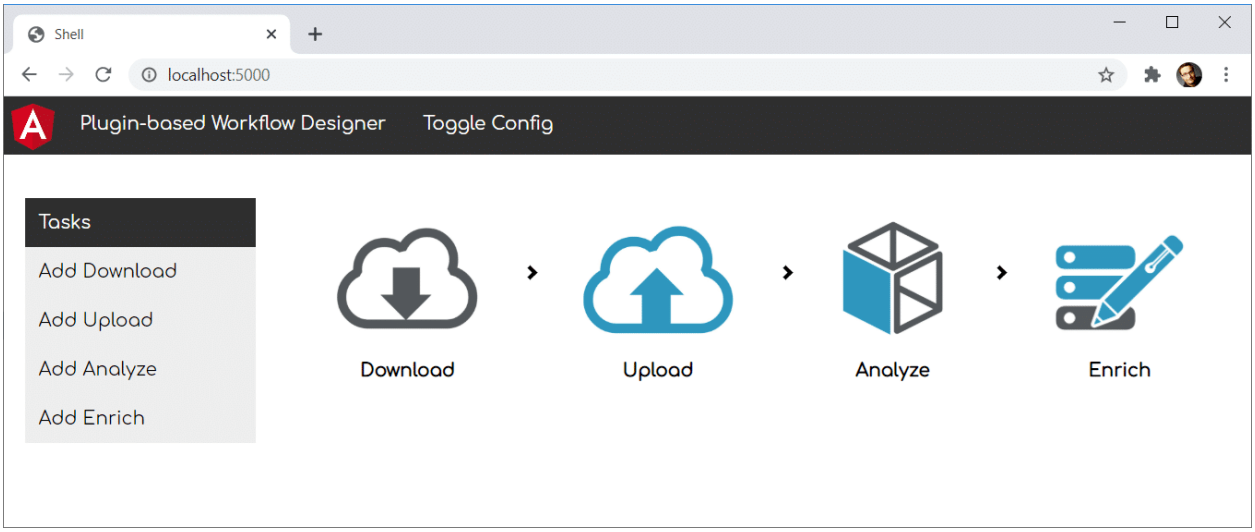
Conclusion

Dynamic Module Federation provides more flexibility as it allows loading microfrontends we don't have to know at compile time. We don't even have to know their number upfront. This is possible because of the runtime API provided by webpack. To make using it a bit easier, the `@angular-architects/module-federation` plugin wrap it nicely into some convenience functions.

Plugin Systems with Module Federation: Building An Extensible Workflow Designer

In the previous chapter, I showed how to use Dynamic Module Federation. This allows us to load micro frontends – or remotes, which is the more general term in Module Federation – not known at compile time. We don't even need to know the number of remotes upfront.

While the previous chapter leveraged the router for integrating remotes available, this one shows how to load individual components. The example used for this is a simple plugin-based workflow designer.



The Workflow Designer can load separately compiled and deployed tasks

The workflow designer acts as a so-called host loading tasks from plugins provided as remotes. Thus, they can be compiled and deployed individually. After starting the workflow designer, it gets a configuration describing the available plugins:

The configuration informs about where to find the tasks

Please note that these plugins are provided via different origins (http://localhost:3000 and http://localhost:3001), and the workflow designer is served from an origin of its own (http://localhost:5000).

As always, the source code[32] can be found in my GitHub account[33].

Important: This book is written for Angular and **Angular CLI 13.1** and higher. Make sure you have a fitting version if you try out the examples outlined here! For more details on the differences/ migration to Angular 13.1 please see this migration guide[34].

Building the Plugins

The plugins are provided via separate Angular applications. For the sake of simplicity, all applications are part of the same monorepo. Their webpack configuration uses Module Federation for exposing the individual plugins as shown in the previous chapters:

[32] https://github.com/manfredsteyer/module-federation-with-angular-dynamic-workflow-designer
[33] https://github.com/manfredsteyer/module-federation-with-angular-dynamic-workflow-designer
[34] https://github.com/angular-architects/module-federation-plugin/blob/main/migration-guide.md

```
new ModuleFederationPlugin({

  library: { type: "module" },

  name: "shell",
  filename: "remoteEntry.js",
  exposes: { },

  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'
    },

    ...sharedMappings.getDescriptors()
  })

}),
```

As also discussed in the previous chapters, this configuration assigns the (container) name `mfe1` to the remote. It shares the libraries `@angular/core`, `@angular/common`, and `@angular/router` with both, the host (=the workflow designer) and the remotes.

The combination of `singleton: true` and `strictVersion: true` makes webpack emit a runtime error when the shell and the micro frontend(s) need different incompatible versions (e. g. two different major versions). If we skipped `strictVersion` or set it to `false`, webpack would only emit a warning at runtime.

Besides, it exposes a remote entry point `remoteEntry.js` which provides the host with the necessary key data for loading the remote.

Loading the Plugins into the Workflow Designer

For loading the plugins into the workflow designer, I'm using the helper function `loadRemoteModule` provided by the `@angular-architects/module-federation` plugin. To load the above mentioned `Download` task, `loadRemoteModule` can be called this way:

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

[...]

const component = await loadRemoteModule({
    type: 'module',
    remoteEntry: 'http://localhost:3000/remoteEntry.js',
    exposedModule: './Download'
})
```

Providing Metadata on the Plugins

At runtime, we need to provide the workflow designer with key data about the plugins. The type used for this is called `PluginOptions` and extends the `LoadRemoteModuleOptions` shown in the previous section by a `displayName` and a `componentName`:

```
export type PluginOptions = LoadRemoteModuleOptions & {
    displayName: string;
    componentName: string;
};
```

While the `displayName` is the name presented to the user, the `componentName` refers to the TypeScript class representing the Angular component in question.

For loading this key data, the workflow designer leverages a `LookupService`:

```
@Injectable({ providedIn: 'root' })
export class LookupService {
    lookup(): Promise<PluginOptions[]> {
        return Promise.resolve([
            {
                type: 'module',
                remoteEntry: 'http://localhost:3000/remoteEntry.js',
                exposedModule: './Download',

                displayName: 'Download',
                componentName: 'DownloadComponent'
            },
            [...]
        ] as PluginOptions[]);
    }
}
```

For the sake of simplicity, the `LookupService` provides some hardcoded entries. In the real world, it would very likely request this data from a respective HTTP endpoint.

Dynamically Creating the Plugin Component

The workflow designer represents the plugins with a `PluginProxyComponent`. It takes a `PluginOptions` object via an input, loads the described plugin via Dynamic Module Federation and displays the plugin's component within a placeholder:

```
@Component({
    selector: 'plugin-proxy',
    template: `
        <ng-container #placeHolder></ng-container>
    `
})
export class PluginProxyComponent implements OnChanges {
    @ViewChild('placeHolder', { read: ViewContainerRef, static: true })
    viewContainer: ViewContainerRef;

    constructor() { }

    @Input() options: PluginOptions;

    async ngOnChanges() {
        this.viewContainer.clear();
```

```
17
18          const component = await loadRemoteModule(this.options)
19              .then(m => m[this.options.componentName]);
20
21          this.viewContainer.createComponent(component);
22      }
23  }
```

In versions before Angular 13, we needed to use a ComponentFactoryResolver to get the loaded component's factory:

```
1   // Before Angular 13, we needed to retrieve a ComponentFactory
2   //
3   // export class PluginProxyComponent implements OnChanges {
4   //     @ViewChild('placeHolder', { read: ViewContainerRef, static: true })
5   //     viewContainer: ViewContainerRef;
6
7   //     constructor(
8   //       private injector: Injector,
9   //       private cfr: ComponentFactoryResolver) { }
10
11  //     @Input() options: PluginOptions;
12
13  //     async ngOnChanges() {
14  //         this.viewContainer.clear();
15
16  //         const component = await loadRemoteModule(this.options)
17  //             .then(m => m[this.options.componentName]);
18
19  //         const factory = this.cfr.resolveComponentFactory(component);
20
21  //         this.viewContainer.createComponent(factory, null, this.injector);
22  //     }
23  // }
```

Wiring Up Everything

Now, it's time to wire up the parts mentioned above. For this, the workflow designer's `AppComponent` gets a `plugins` and a `workflow` array. The first one represents the `PluginOptions` of the available plugins and thus all available tasks while the second one describes the `PluginOptions` of the selected tasks in the configured sequence:

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

  plugins: PluginOptions[] = [];
  workflow: PluginOptions[] = [];
  showConfig = false;

  constructor(
    private lookupService: LookupService) {
  }

  async ngOnInit(): Promise<void> {
    this.plugins = await this.lookupService.lookup();
  }

  add(plugin: PluginOptions): void {
    this.workflow.push(plugin);
  }

  toggle(): void {
    this.showConfig = !this.showConfig;
  }
}
```

The `AppComponent` uses the injected `LookupService` for populating its `plugins` array. When a plugin is added to the workflow, the `add` method puts its `PluginOptions` object into the workflow array.

For displaying the workflow, the designer just iterates all items in the workflow array and creates a `plugin-proxy` for them:

```
<ng-container *ngFor="let p of workflow; let last = last">
    <plugin-proxy [options]="p"></plugin-proxy>
    <i *ngIf="!last" class="arrow right" style=""></i>
</ng-container>
```

As discussed above, the proxy loads the plugin (at least, if it isn't already loaded) and displays it.

Also, for rendering the toolbox displayed on the left, it goes through all entries in the `plugins` array. For each of them it displays a hyperlink calling bound to the `add` method:

```
<div class="vertical-menu">
    <a href="#" class="active">Tasks</a>
    <a *ngFor="let p of plugins" (click)="add(p)">Add {{p.displayName}}</a>
</div>
```

Conclusion

While Module Federation comes in handy for implementing micro frontends, it can also be used for setting up plugin architectures. This allows us to extend an existing solution by 3rd parties. It also seems to be a good fit for SaaS applications, which needs to be adapted to different customers' needs.

Using Module Federation with Nx

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.

This chapter gives some hints about using Module Federation for such a scenario. While the examples use a Nx workspace, the principal ideas can also be implemented with a classic Angular workspace. However, as you 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 need to be redeployed.

If you want to have a look at the source code[35] used here, you can check out this repository[36].

Important: This book 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[37].

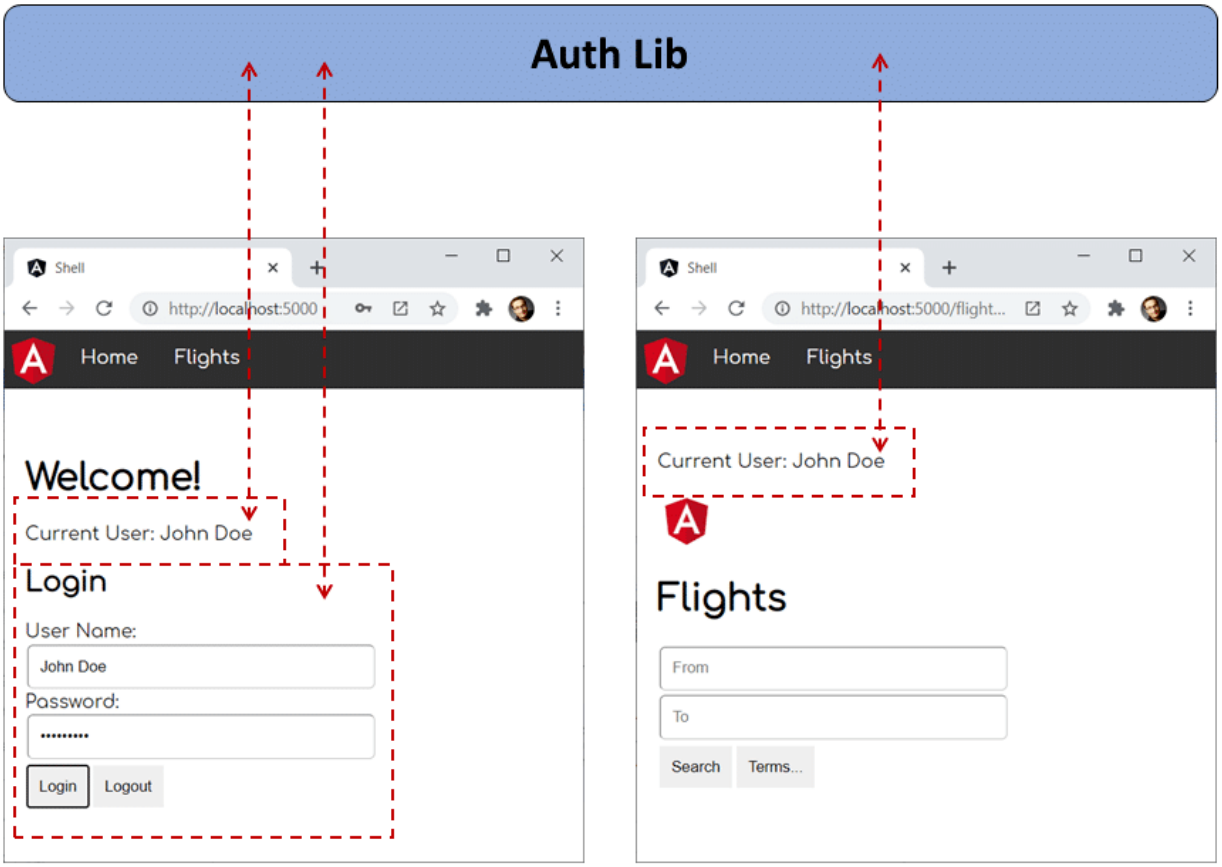
Example

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:

<img src="images/mf-nx-graph.png" width="300">

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`:

[35] https://github.com/manfredsteyer/module_federation_nx_mono_repo
[36] https://github.com/manfredsteyer/module_federation_nx_mono_repo
[37] https://github.com/angular-architects/module-federation-plugin/blob/main/migration-guide.md



Schema

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

As usual in Nx and Angular monorepos, libraries are referenced with path mappings defined in `tsconfig.json` (or `tsconfig.base.json` according to your project setup):

```
"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`:

```
@Injectable({
  providedIn: 'root'
})
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;
  }
}
```

The authentication method I'm using here, is what I'm calling "Authentication for honest users TM". Besides this service, there is also a `AuthComponent` with the UI for logging-in the user and a `UserComponent` 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:

```
export * from './lib/auth-lib.module';
export * from './lib/auth.service';

// Don't forget about your components!
export * from './lib/auth/auth.component';
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 chapter, 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:

```
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 ModuleFederationPlugin =
  require("webpack/lib/container/ModuleFederationPlugin");
const mf =
  require("@angular-architects/module-federation/webpack");
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
  },
  plugins: [
    new ModuleFederationPlugin({

      library: { type: "module" },

      name: "mfe1",
      filename: "remoteEntry.js", // <-- Metadata
      exposes: {
          './Module': './apps/mfe1/src/app/flights/flights.module.ts',
      },

      shared: share({
        "@angular/core": {
          singleton: true,
          strictVersion: true,
```

```
44            requiredVersion: 'auto'
45          },
46          "@angular/common": {
47            singleton: true,
48            strictVersion: true,
49            requiredVersion: 'auto'
50          },
51          "@angular/common/http": {
52            singleton: true,
53            strictVersion: true,
54            requiredVersion: 'auto'
55          },
56          "@angular/router": {
57            singleton: true,
58            strictVersion: true,
59            requiredVersion: 'auto'
60          },
61
62          ...sharedMappings.getDescriptors()
63        })
64
65      }),
66      sharedMappings.getPlugin()
67    ],
68  };
```

Everything you need to do is registering the library's mapped name with the `sharedMappings` instance at the top of this file. However, we need to do this for all projects that want to share this lib. Or to put in in another way: **Each application needs to opt-in for sharing a libaray**.

For the time being, this is all we need to know. In a section below, I'm going to explain what `SharedMappings` 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

As normally, libraries don't have versions in a monorepo, we should always redeploy all the changed micro frontends together. Fortunately, Nx helps with finding out which applications/ micro frontends have been 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?

So far, we've just treated the `SharedMappings` class as a black box. Now, let's find out what it does for us 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.

The second thing `SharedMappings` is doing, is rewriting the imports of the code produced by the Angular compiler. This is necessary because the code generated by the Angular compiler references shared libraries with a relative path:

```
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`:

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

For this, `SharedMappings` provides a method called `getPlugin` returning a configured `NormalModuleReplacementPlug` 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.

However, Module Federation is really flexible and hence it even allows to define versions for libraries in a monorepo. In the case of npm packages, both, the version provided by the library but also the versions needed 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 Module Federation. 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",
}
```

Now, the webpack config of a specific micro frontend/ of the shell only needs to contain the accepted version (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.

If you decide to go this road, the `SharedMappings` class has you covered. The `getDescriptor` method generates the shared entry needed for the passed library. Its second argument takes the expected version (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.

Sharing a library that is not even used

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

```
./projects/shared-lib/src/public-api.ts - Error: Module build failed (from ./node_mo\
dules/@ngtools/webpack/src/index.js):
Error: C:\Users\Manfred\Documents\projekte\mf-plugin\example\projects\shared-lib\src\
\public-api.ts is missing from the TypeScript compilation. Please make sure it is in\
 your tsconfig via the 'files' or 'include' property.
    at AngularCompilerPlugin.getCompiledFile (C:\Users\Manfred\Documents\projekte\mf\
-plugin\example\node_modules\@ngtools\webpack\src\angular_compiler_plugin.js:957:23)
    at C:\Users\Manfred\Documents\projekte\mf-plugin\example\node_modules\@ngtools\w\
ebpack\src\loader.js:43:31
```

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:

```
core.js:4610 ERROR Error: Uncaught (in promise): TypeError: Cannot read property '\c\
mp' of undefined
TypeError: Cannot read property 'cmp' of undefined
    at getComponentDef (core.js:1821)
```

Conflict: Multiple assets emit different content to the same filename

Add this to the output section of your webpack config:

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

Dealing with Version Mismatches in Module Federation

Webpack Module Federation makes it easy to load separately compiled code like micro frontends. It even allows us to share libraries among them. This prevents that the same library has to be loaded several times.

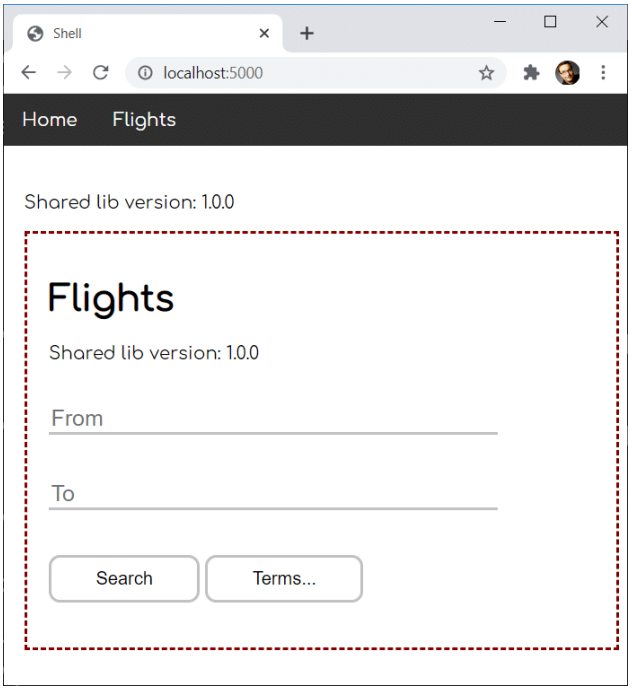
However, there might be situations where several micro frontends and the shell need different versions of a shared library. Also, these versions might not be compatible with each other.

For dealing with such cases, Module Federation provides several options. In this chapter, I present these options by looking at different scenarios. The source code[38] for these scenarios can be found in my GitHub account[39].

Example Used Here

To demonstrate how Module Federation deals with different versions of shared libraries, I use a simple shell application known from previous chapters. It is capable of loading micro frontends into its working area:

[38] https://github.com/manfredsteyer/module_federation_shared_versions
[39] https://github.com/manfredsteyer/module_federation_shared_versions



Shell loading microfrontends

The micro frontend is framed with the red dashed line.

For sharing libraries, both, the shell and the micro frontend use the following setting in their webpack configurations:

```
new ModuleFederationPlugin({
   [...],
   shared: ["rxjs", "useless-lib"]
})
```

The package useless-lib[40] is a dummy package, I've published for this example. It's available in the versions `1.0.0`, `1.0.1`, `1.1.0`, `2.0.0`, `2.0.1`, and `2.1.0`. In the future, I might add further ones. These versions allow us to simulate different kinds of version mismatches.

To indicate the installed version, `useless-lib` exports a `version` constant. As you can see in the screenshot above, the shell and the micro frontend display this constant. In the shown constellation, both use the same version (`1.0.0`), and hence they can share it. Therefore, `useless-lib` is only loaded once.

However, in the following sections, we will examine what happens if there are version mismatches between the `useless-lib` used in the shell and the one used in the `microfrontend`. This also allows me to explain different concepts Module Federation implements for dealing with such situations.

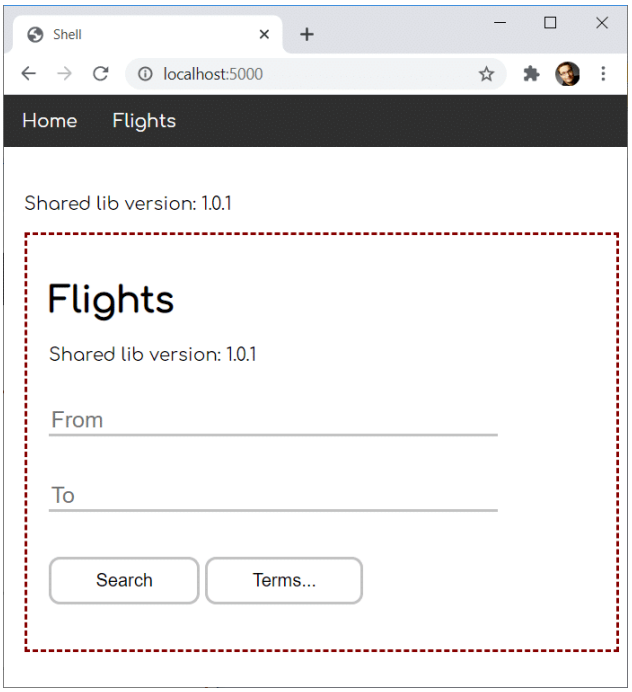
[40] https://www.npmjs.com/package/useless-lib

Semantic Versioning by Default

For our first variation, let's assume our `package.json` is pointing to the following versions:

- **Shell**: useless-lib@^1.0.0
- **MFE1**: useless-lib@^1.0.1

This leads to the following result:



Module Federation decides to go with version `1.0.1` as this is the highest version compatible with both applications according to semantic versioning (`^1.0.0` means, we can also go with a higher minor and patch versions).

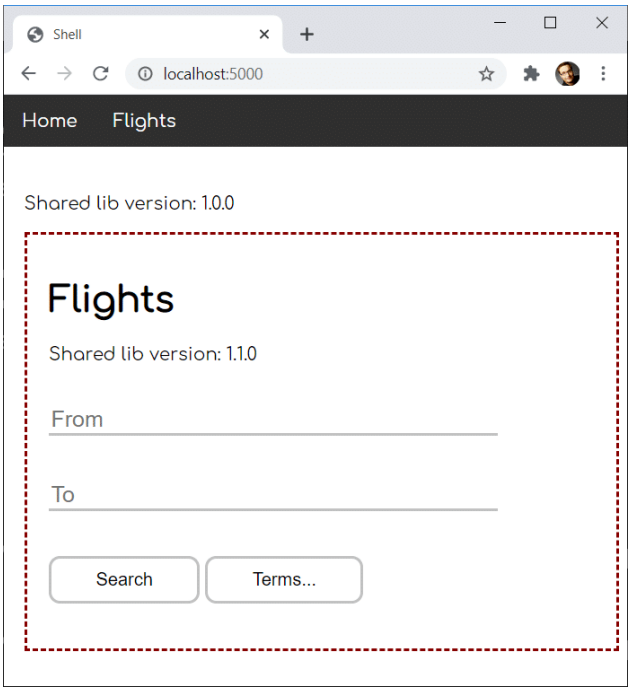
Fallback Modules for Incompatible Versions

Now, let's assume we've adjusted our dependencies in `package.json` this way:

- **Shell**: useless-lib@~1.0.0
- **MFE1**: useless-lib@1.1.0

Both versions are not compatible with each other (`~1.0.0` means, that only a higher patch version but not a higher minor version is acceptable).

This leads to the following result:



Using Fallback Module

This shows that Module Federation uses different versions for both applications. In our case, each application falls back to its own version, which is also called the fallback module.

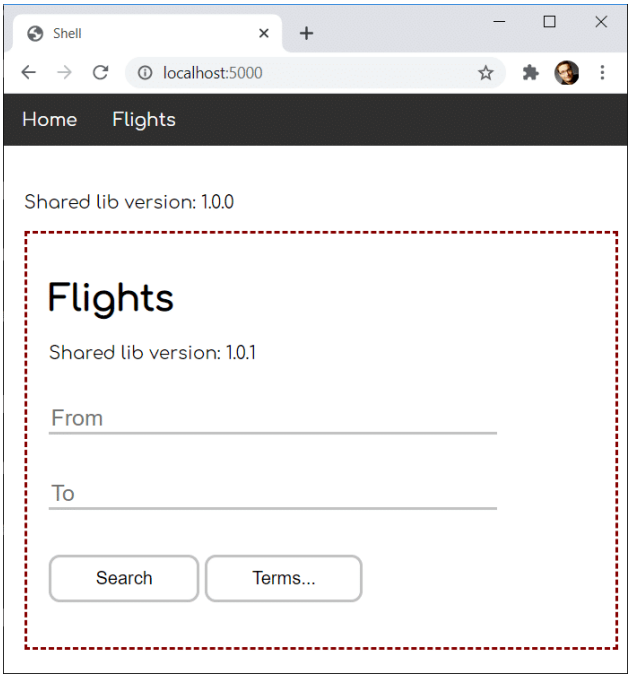
Differences With Dynamic Module Federation

Interestingly, the behavior is a bit different when we load the micro frontends including their remote entry points just on demand using Dynamic Module Federation. The reason is that dynamic remotes are not known at program start, and hence Module Federation cannot draw their versions into consideration during its initialization phase.

For explaining this difference, let's assume the shell is loading the micro frontend dynamically and that we have the following versions:

- **Shell**: useless-lib@^1.0.0
- **MFE1**: useless-lib@^1.0.1

While in the case of classic (static) Module Federation, both applications would agree upon using version `1.0.1` during the initialization phase, here in the case of dynamic module federation, the shell does not even know of the micro frontend in this phase. Hence, it can only choose for its own version:



Dynamic Microfrontend falls back to own version

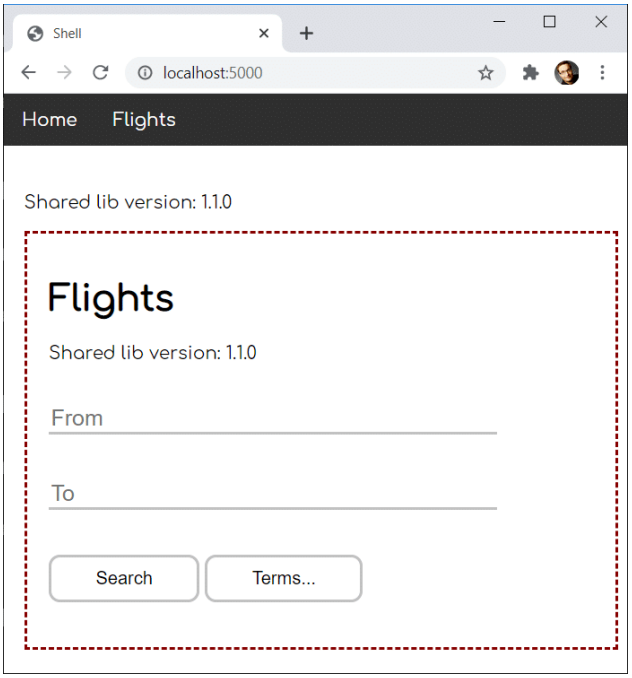
If there were other static remotes (e. g. micro frontends), the shell could also choose for one of their versions according to semantic versioning, as shown above.

Unfortunately, when the dynamic micro frontend is loaded, module federation does not find an already loaded version compatible with `1.0.1`. Hence, the micro frontend falls back to its own version `1.0.1`.

On the contrary, let's assume the shell has the highest compatible version:

- **Shell**: useless-lib@^1.1.0
- **MFE1**: useless-lib@^1.0.1

In this case, the micro frontend would decide to use the already loaded one:



Dynamic Microfrontend uses already loaded version

To put it in a nutshell, in general, it's a good idea to make sure your shell provides the highest compatible versions when loading dynamic remotes as late as possible.

However, as discussed in the chapter about Dynamic Module Federation, it's possible to dynamically load just the remote entry point on program start and to load the micro frontend later on demand. By splitting this into two loading processes, the behavior is exactly the same as with static ("classic") Module Federation. The reason is that in this case the remote entry's meta data is available early enoght to be considering during the negotation of the versions.

Singletons

Falling back to another version is not always the best solution: Using more than one version can lead to unforeseeable effects when we talk about libraries holding state. This seems to be always the case for your leading application framework/ library like Angular, React or Vue.

For such scenarios, Module Federation allows us to define libraries as **singletons**. Such a singleton is only loaded once.

If there are only compatible versions, Module Federation will decide for the highest one as shown in the examples above. However, if there is a version mismatch, singletons prevent Module Federation from falling back to a further library version.

For this, let's consider the following version mismatch:

- **Shell**: useless-lib@^2.0.0

- **MFE1**: useless-lib@^1.1.0

Let's also consider we've configured the `useless-lib` as a singleton:

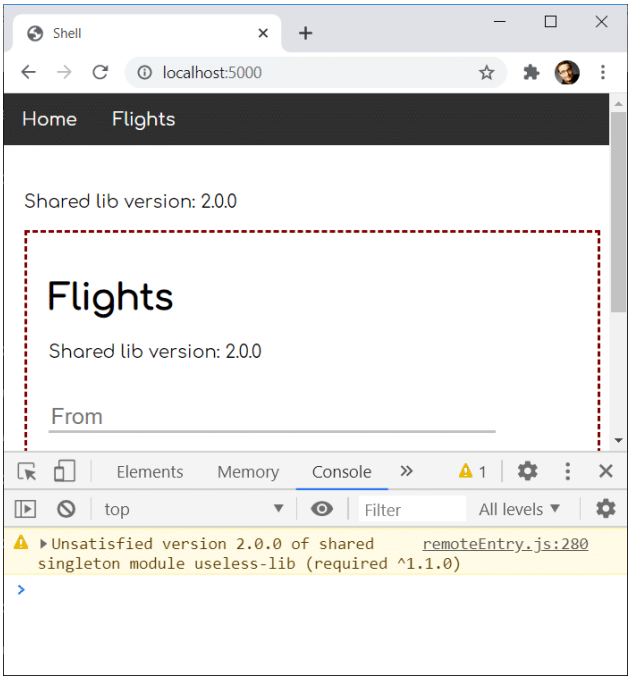
```
// Shell
shared: {
  "rxjs": {},
  "useless-lib": {
    singleton: true,
  }
},
```

Here, we use an advanced configuration for defining singletons. Instead of a simple array, we go with an object where each key represents a package.

If one library is used as a singleton, you will very likely set the singleton property in every configuration. Hence, I'm also adjusting the microfrontend's Module Federation configuration accordingly:

```
// MFE1
shared: {
    "rxjs": {},
    "useless-lib": {
        singleton: true
    }
}
```

To prevent loading several versions of the singleton package, Module Federation decides for only loading the highest available library which it is aware of during the initialization phase. In our case this is version `2.0.0`:



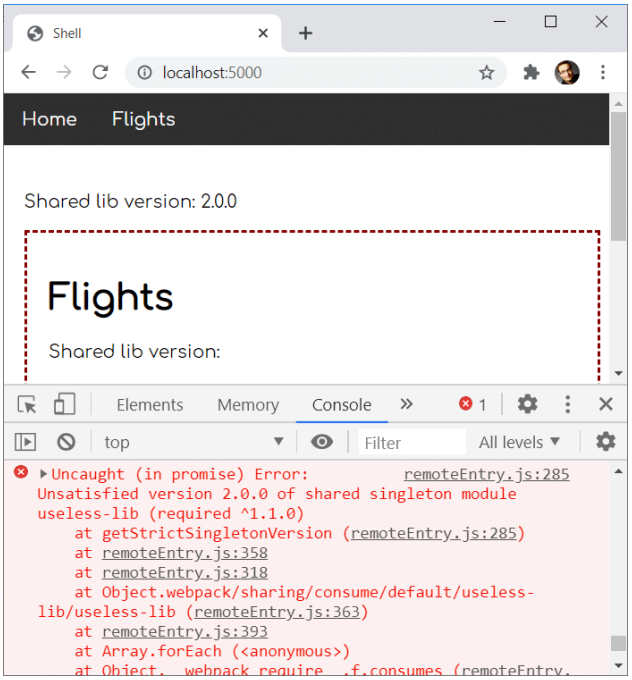
Module Federation only loads the highest version for singletons

However, as version `2.0.0` is not compatible with version `1.1.0` according to semantic versioning, we get a warning. If we are lucky, the federated application works even though we have this mismatch. However, if version `2.0.0` introduced braking changes we run into, our application might fail.

In the latter case, it might be beneficial to fail fast when detecting the mismatch by throwing an example. To make Module Federation behaving this way, we set `strictVersion` to `true`:

```
// MFE1
shared: {
  "rxjs": {},
  "useless-lib": {
    singleton: true,
    strictVersion: true
  }
}
```

The result of this looks as follows at runtime:



Version mismatches regarding singletons using strictVersion make the application fail

Accepting a Version Range

There might be cases where you know that a higher major version is backward compatible even though it doesn't need to be with respect to semantic versioning. In these scenarios, you can make Module Federation accepting a defined version range.

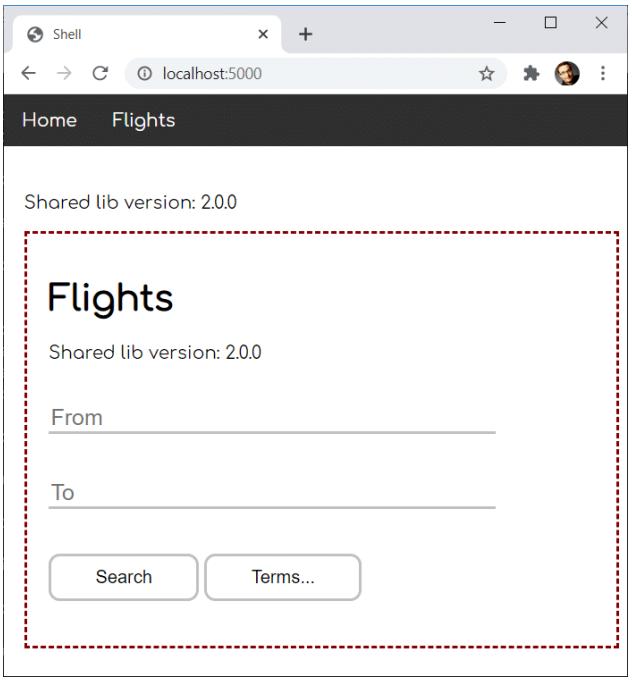
To explore this option, let's one more time assume the following version mismatch:

- **Shell**: useless-lib@^2.0.0
- **MFE1**: useless-lib@^1.1.0

Now, we can use the `requiredVersion` option for the `useless-lib` when configuring the microfrontend:

```
// MFE1
shared: {
  "rxjs": {},
  "useless-lib": {
    singleton: true,
    strictVersion: true,
    requiredVersion: ">=1.1.0 <3.0.0"
  }
}
```

According to this, we also accept everything having `2` as the major version. Hence, we can use the version `2.0.0` provided by the shell for the micro frontend:



Accepting incompatible versions by defining a version range

Conclusion

Module Federation brings several options for dealing with different versions and version mismatches. Most of the time, you don't need to do anything, as it uses semantic versioning to decide for the highest compatible version. If a remote needs an incompatible version, it falls back to such one by default.

In cases where you need to prevent loading several versions of the same package, you can define a shared package as a singleton. In this case, the highest version known during the initialization phase

is used, even though it's not compatible with all needed versions. If you want to prevent this, you can make Module Federation throw an exception using the `strictVersion` option.

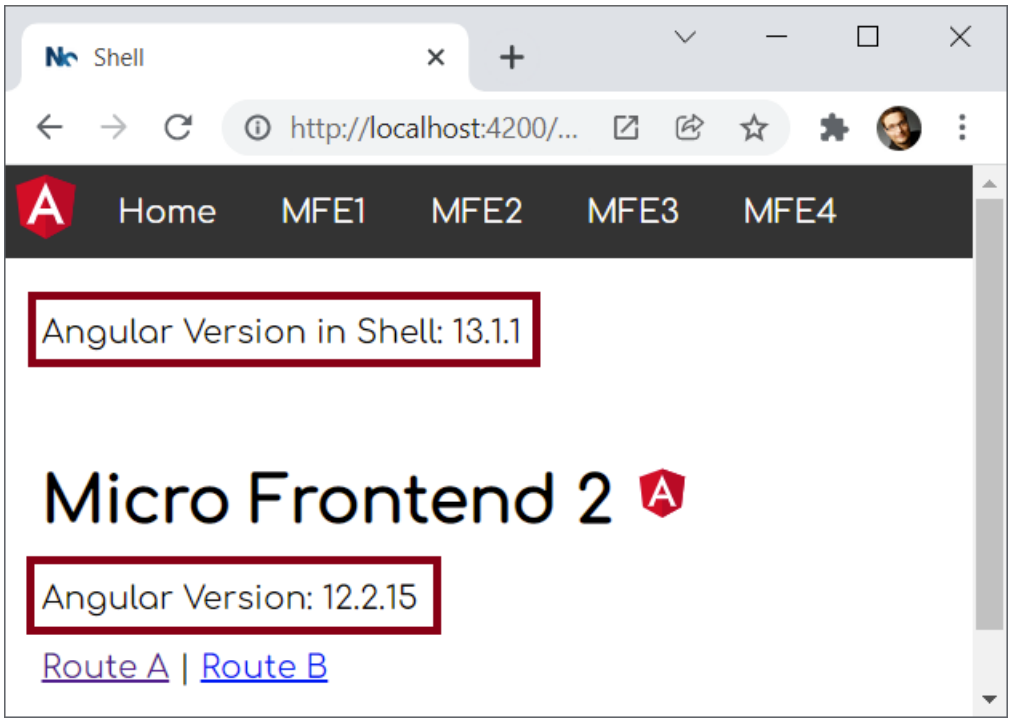
You can also ease the requirements for a specific version by defining a version range using `requestedVersion`. You can even define several scopes for advanced scenarios where each of them can get its own version.

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

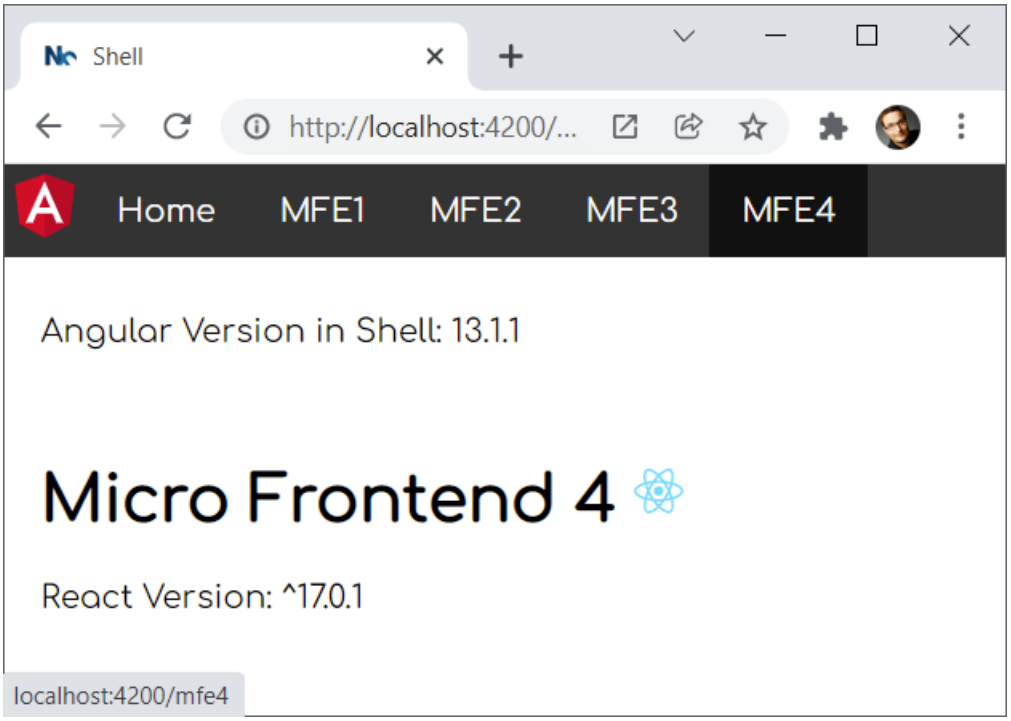
Some of my customers need to combine Micro Frontends built with different frameworks and/or framework versions. A quite modern and flexible approach for this is providing Web Components via Module Federation.

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

For this, here, I'm using an Angular-based shell that loads several Micro Frontends. They've been developed with two different Angular versions and React:

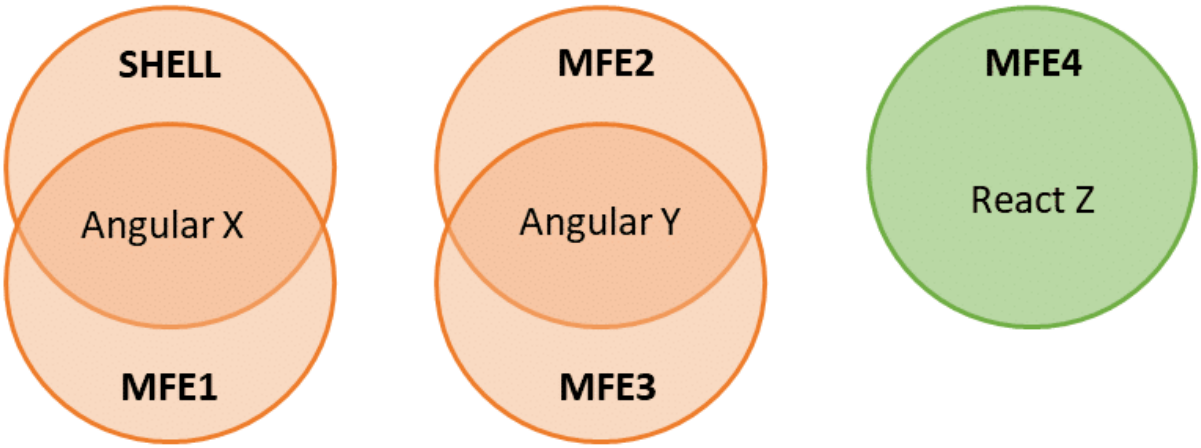


Micro Frontends using different Angular versions



Micro Frontends using different frameworks

The shell and *Micro Frontend 1* use the same Angular version. Hence, they shall share it. The same is true for *Micro Frontend 2* and *Micro Frontend 3*. *Micro Frontend 4*, however, uses React:



Different frameworks and versions can be shared

The source code for this case study can be found here[41].

[41] https://github.com/manfredsteyer/multi-framework-micro-frontend

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 ;-).

Seriously, while wrapping applications into web components and loading them with Module Federation is not that difficult, there are several pitfalls along the way. Hence, I want to start with two alternatives:

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.

Alternative 2: Relaxing Version Requirements + Heavy Testing

Another approach is to relax the requirements for needed versions. For instance, we could tell Module Federation that an application built with Angular 10 also works with Angular 11. For this, Module Federation provides the configuration property `requiredVersion` described here[42].

This might work because, normally, Angular's core didn't recently change much between major versions. 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[43] which is an add-on for @angular-architects/module-federation. Here you even find a live demo[44].

Nethertheless, this chapter 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.

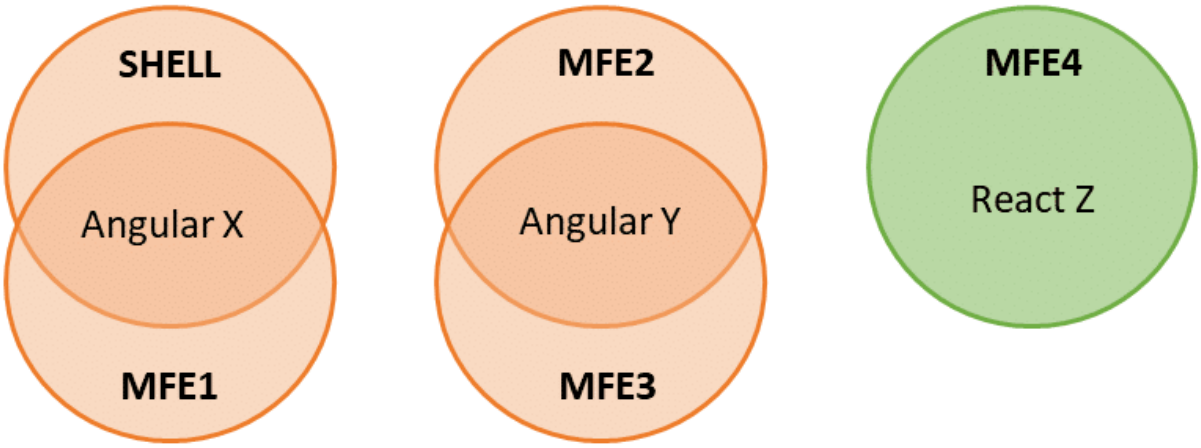
[42] https://www.angulararchitects.io/aktuelles/getting-out-of-version-mismatch-hell-with-module-federation/
[43] https://www.npmjs.com/package/@angular-architects/module-federation-tools
[44] https://red-ocean-0fe4c4610.azurestaticapps.net/

The Good

Okay, let's go on with finding out how the combination of Module Federation and Web Components allows building multi framework(version) Micro Frontends. Before we look into the pitfalls, I'm going to tell you about the good aspects of this.

Sharing Libraries

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



Different frameworks and versions can be shared

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

```
new ModuleFederationPlugin({
  [...],
  shared: ["@angular/core", "@angular/common", "@angular/router"]
})
```

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[45].

Exporting Web Components

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 Angular modules or a 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 file `bootstrap.ts` uses the same source code the CLI normally generates for your `main.ts`. This includes calling the platform's `bootstrapModule` method with our `AppModule`.

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

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

As mentioned in one of the previous chapters 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:

[45] https://www.angulararchitects.io/aktuelles/getting-out-of-version-mismatch-hell-with-module-federation/

```
[...]
import { createCustomElement } from '@angular/elements';
[...]

@NgModule({
  imports: [
    BrowserModule,
    RouterModule.forRoot([...])
  ],
  declarations: [
    [...]
    AppComponent
  ],
  providers: [],
  bootstrap: []
})
export class AppModule {
  constructor(private injector: Injector) {
  }

  ngDoBootstrap() {
    const ce = createCustomElement(AppComponent, {injector: this.injector});
    customElements.define('mfe1-element', ce);
  }

}
```

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.

Loading Micro Frontends

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` provided by the package `@angular-architects/module-federation`.

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

export const registry = {
    mfe1: () => loadRemoteModule({
        type: 'module',
        remoteEntry: 'http://localhost:4201/remoteEntry.js',
        exposedModule: './web-components'
    }),
    mfe2: () => loadRemoteModule({
        type: 'script',
        remoteEntry: 'http://localhost:4202/remoteEntry.js',
        remoteName: 'mfe2',
        exposedModule: './web-components'
    }),
    mfe3: () => loadRemoteModule({
        type: 'script',
        remoteEntry: 'http://localhost:4203/remoteEntry.js',
        remoteName: 'mfe3',
        exposedModule: './web-components'
    }),
    mfe4: () => loadRemoteModule({
        type: 'script',
        remoteEntry: 'http://localhost:4204/remoteEntry.js',
        remoteName: 'mfe4',
        exposedModule: './web-components'
    }),
};
```

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 the chapter on Dynamic Federation.

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

```
const element = document.createElement('mfe1-element');
document.body.appendChild(element);

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`:

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

```
      },
    ])
  ],
  declarations: [
    AppComponent,
    WrapperComponent
  ],
  providers: [],
  bootstrap: [AppComponent]
})
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`:

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

@Component({
  template: '<div #vc></div>',
})
export class WrapperComponent implements AfterContentInit {

  @ViewChild('vc', {read: ElementRef, static: true})
  vc: ElementRef;

  constructor(private route: ActivatedRoute) { }

  ngAfterContentInit(): void {

    const elementName = this.route.snapshot.data['elementName'];
```

```
25      const importName = this.route.snapshot.data['importName'];
26
27      const importFn = registry[importName];
28      importFn()
29        .then(_ => console.debug(`element ${elementName} loaded!`))
30        .catch(err => console.error(`error loading ${elementName}:`, err));
31
32      const element = document.createElement(elementName);
33      this.vc.nativeElement.appendChild(element);
34
35    }
36
37  }
```

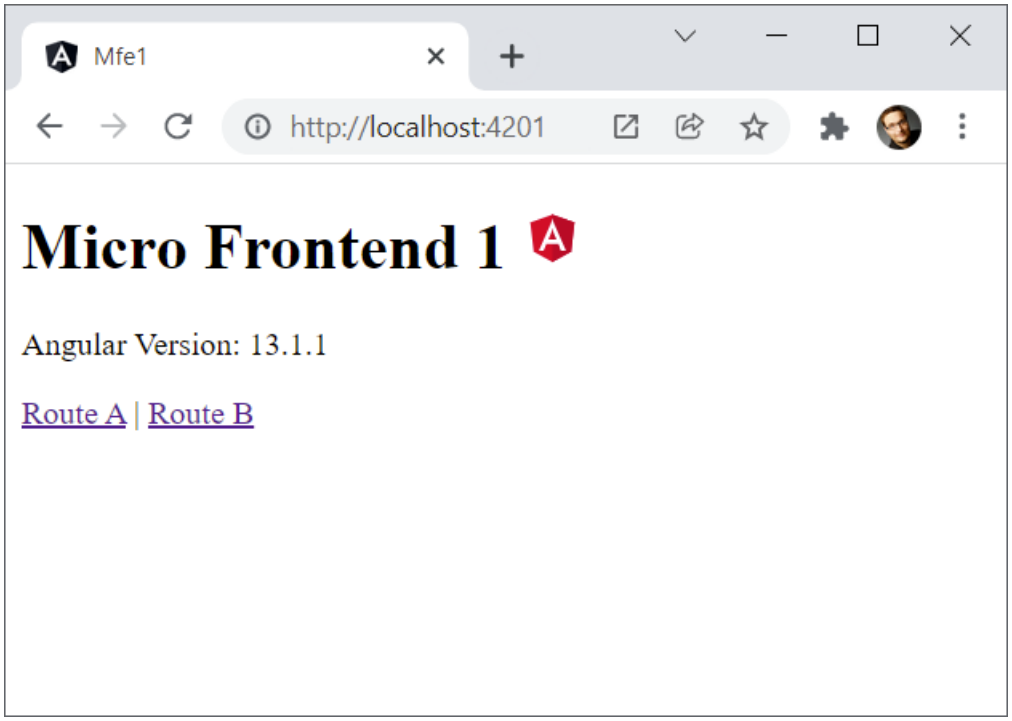
As an **alternative**, we could make this example more dynamic by skipping the registry and passing the key data for loading the remote and creating its root element **directly** to the wrapper component.

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:



Standalone Mode

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

Lazy Loading within Micro Frontends

Another advantage of this approach is that we can use traditional lazy loading within each Micro Frontend. In cases where we load Micro Frontends directly without Module Federation this can cause issues because the 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.

Several Routers must Work Together

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: 'mfe1-element' }
      },
      {
        matcher: startsWith('mfe2'),
        component: WrapperComponent,
        data: { importName: 'mfe2', elementName: 'mfe2-element' }
      },
      {
        matcher: startsWith('mfe3'),
        component: WrapperComponent,
        data: { importName: 'mfe3', elementName: 'mfe3-element' }
      },
      {
        matcher: startsWith('mfe4'),
        component: WrapperComponent,
        data: { importName: 'mfe4', elementName: 'mfe4-element' }
      },
    ])
  ],
  declarations: [
```

```
    AppComponent,
    WrapperComponent
  ],
  providers: [],
  bootstrap: [AppComponent]
})
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:

```
import { UrlMatcher, UrlSegment } from '@angular/router';

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

One issue that arises when using several Angular routers together, is that the inner routers don't recognize a route change. Hence, we need to "invite them for routing" manually every time the URL changes:

```
@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

Per shared Angular version, we are only allowed to create one platform. To remember that there is already a shared platform for our version, we could put it into a global dictionary mapping the version number to the platform instance:

```
declare const require: any;
const ngVersion = require('../package.json').dependencies['@angular/core']; // perha\
ps 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.

Of course, we could turn off Zone.js when bootstrapping the Micro Frontends:

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

However, this also means we need to do change detection by hand. My GDE colleague, Tomas Trajan[46] 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:

[46] https://twitter.com/tomastrajan

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

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

```
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.

Conclusion

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 route to web 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.

Pitfalls with Module Federation and Angular

In this chapter, I'm going to destroy my Module Federation example! However, you don't need to worry: It's for a very good reason. The goal is to show typical pitfalls that come up when using Module Federation together with Angular. Also, this chapter presents some strategies for avoiding these pitfalls.

While Module Federation is really a straight and thoroughly thought through solution, using Micro Frontends means in general to make runtime dependencies out of compile time dependencies. As a result, the compiler cannot protect you as well as you are used to.

If you want to try out the examples used here, you can fork this example[47].

"No required version specified" and Secondary Entry Points

For the first pitfall I want to talk about, let's have a look to our `shell`'s `webpack.config.js`. Also, let's simplify the `shared` node as follows:

```
shared: {
  "@angular/core": { singleton: true, strictVersion: true },
  "@angular/common": { singleton: true, strictVersion: true },
  "@angular/router": { singleton: true, strictVersion: true },
  "@angular/common/http": { singleton: true, strictVersion: true },

  // Uncomment for sharing lib of an Angular CLI or Nx workspace
  ...sharedMappings.getDescriptors()
},
```

As you see, we don't specify a `requiredVersion` anymore. Normally this is not required because webpack Module Federation is very smart with finding out which version you use.

However, now, when compiling the shell (`ng build shell`), we get the following error:

[47] https://github.com/manfredsteyer/module-federation-plugin-example.git

> shared module @angular/common - Warning: No required version specified and unable to automatically determine one. Unable to find required version for "@angular/common" in description file (C:UsersManfredDocuments\artikelModuleFederation-Pitfalls\example\node_modules@angular\common\package.json). It need to be in dependencies, devDependencies or peerDependencies.

The reason for this is the secondary entry point `@angular/common/http` which is a bit like an npm package within an npm package. Technically, it's just another file exposed by the npm package `@angular/common`.

Unsurprisingly, `@angular/common/http` uses `@angular/common` and webpack recognizes this. For this reason, webpack wants to find out which version of `@angular/common` is used. For this, it looks into the npm package's `package.json` (`@angular/common/package.json`) and browses the dependencies there. However, `@angular/common` itself is not a depdency of `@angular/common` and hence, the version cannot be found.

You will have the same challenge with other packages using secondary entry points, e. g. `@angular/material`.

To avoid this situation, you can assign versions to all shared libraries by hand:

```
shared: {
  "@angular/core": {
    singleton: true,
    strictVersion: true,
    requiredVersion: '12.0.0'
  },
  "@angular/common": {
    singleton: true,
    strictVersion: true,
    requiredVersion: '12.0.0'
  },
  "@angular/router": {
    singleton: true,
    strictVersion: true,
    requiredVersion: '12.0.0'
  },
  "@angular/common/http": {
    singleton: true,
    strictVersion: true,
    requiredVersion: '12.0.0'
  },

  // Uncomment for sharing lib of an Angular CLI or Nx workspace
  ...sharedMappings.getDescriptors()
},
```

Obviously, this is cumbersome and so we came up with another solution. Since version 12.3, @angular-architects/module-federation[48] comes with an unspectacular looking helper function called `shared`. If your `webpack.config.js` was generated with this or a newer version, it already uses this helper function.

```
[...]

const mf = require("@angular-architects/module-federation/webpack");
[...]
const share = mf.share;

[...]

shared: share({
  "@angular/core": {
    singleton: true,
    strictVersion: true,
    requiredVersion: 'auto'
  },
  "@angular/common": {
    singleton: true,
    strictVersion: true,
    requiredVersion: 'auto'
  },
  "@angular/router": {
    singleton: true,
    strictVersion: true,
    requiredVersion: 'auto'
  },
  "@angular/common/http": {
    singleton: true,
    strictVersion: true,
    requiredVersion: 'auto'
  },
  "@angular/material/snack-bar": {
    singleton: true,
    strictVersion: true,
    requiredVersion:'auto'
  },

  // Uncomment for sharing lib of an Angular CLI or Nx workspace
```

[48] https://www.angulararchitects.io/wp-content/uploads/2021/07/https://www.npmjs.com/package/@angular-architects/module-federation

```
  ...sharedMappings.getDescriptors()
})
```

As you see here, the `share` function wraps the object with the shared libraries. It allows to use `requiredVersion: 'auto'` and converts the value `auto` to the value found in your shell's (or micro frontend's) `package.json`.

Unobvious Version Mismatches: Issues with Peer Dependencies

Have you ever just ignored a peer dependency warning? Honestly, I think we all know such situations. And ignoring them is often okay-ish as we know, at runtime everything will be fine. Unfortunately, such a situation can confuses webpack Module Federation when trying to auto-detect the needed versions of peer dependencies.

To demonstrate this situation, let's install `@angular/material` and `@angular/cdk` in a version that is at least 2 versions behind our Angular version. In this case, we should get a peer dependency warnings.

In my case this is done as follows:

```
npm i @angular/material@10
npm i @angular/cdk@10
```

Now, let's switch to the Micro Frontend's (`mfe1`) `FlightModule` to import the `MatSnackBarModule`:

```
[...]
import { MatSnackBarModule  } from '@angular/material/snack-bar';
[...]

@NgModule({
  imports: [
    [...]
    // Add this line
    MatSnackBarModule,
  ],
  declarations: [
    [...]
  ]
})
export class FlightsModule { }
```

To make use of the snack bar in the `FlightsSearchComponent`, inject it into its constructor and call its `open` method:

```
[...]
import { MatSnackBar } from '@angular/material/snack-bar';

@Component({
  selector: 'app-flights-search',
  templateUrl: './flights-search.component.html'
})
export class FlightsSearchComponent {
  constructor(snackBar: MatSnackBar) {
    snackBar.open('Hallo Welt!');
  }
}
```

Also, for this experiment, make sure the `webpack.config.js` in the project `mfe1` does **not** define the versions of the dependencies shared:

```
shared: {
  "@angular/core": { singleton: true, strictVersion: true },
  "@angular/common": { singleton: true, strictVersion: true },
  "@angular/router": { singleton: true, strictVersion: true },
  "@angular/common/http": { singleton: true, strictVersion: true },

  // Uncomment for sharing lib of an Angular CLI or Nx workspace
  ...sharedMappings.getDescriptors()
},
```

Not defining these versions by hand forces Module Federation into trying to detect them automatically. However, the peer dependency conflict gives Module Federation a hard time and so it brings up the following error:

> Unsatisfied version 12.0.0 of shared singleton module @angular/core (required ^10.0.0 || ^11.0.0-0) ; Zone: <root> ; Task: Promise.then ; Value: Error: Unsatisfied version 12.0.0 of shared singleton module @angular/core (required ^10.0.0 || ^11.0.0-0)

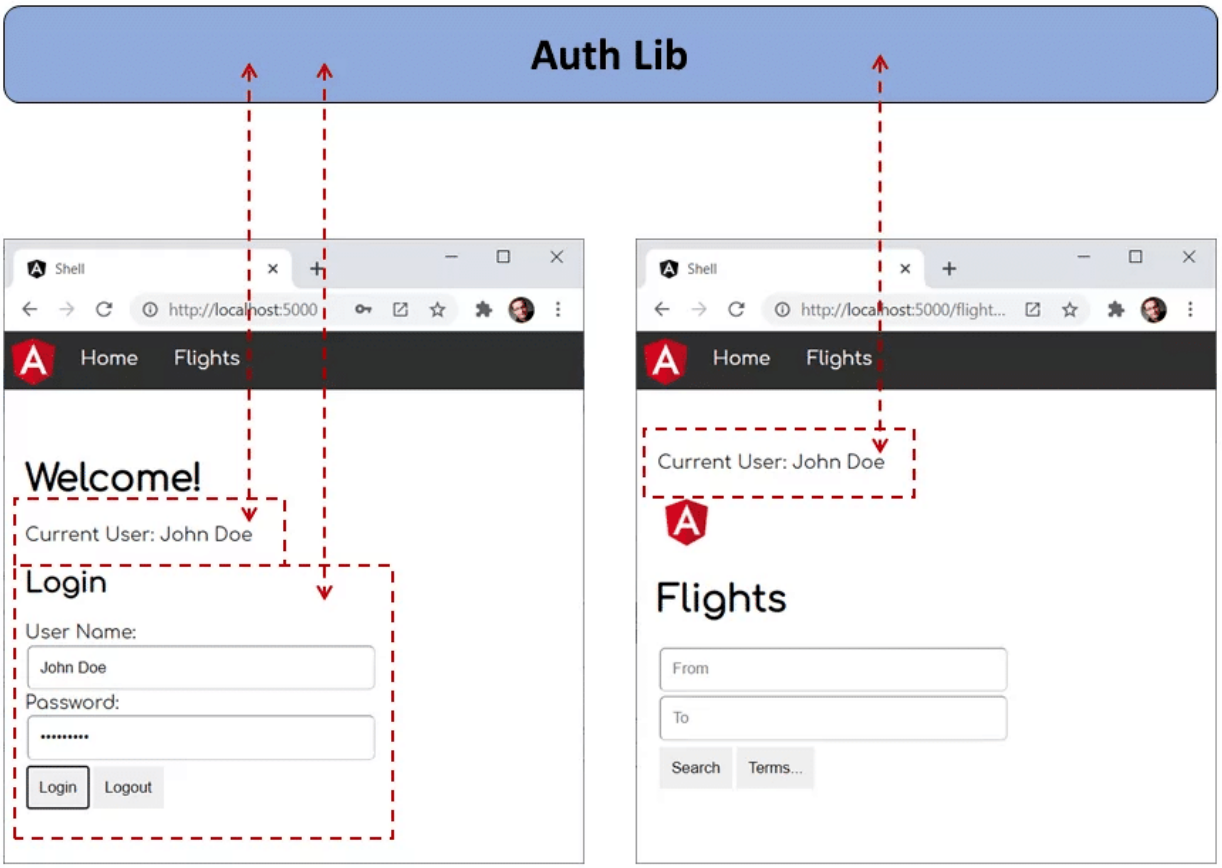
While `@angular/material` and `@angular/cdk` officially need `@angular/core` 10, the rest of the application already uses `@angular/core` 12. This shows that webpack looks into the `package.json` files of all the shared dependencies for determining the needed versions.

In order to resolve this, you can set the versions by hand or by using the helper function `share` that uses the version found in your project's `package.json`:

```
[...]

const mf = require("@angular-architects/module-federation/webpack");
[...]
const share = mf.share;

[...]

shared: share({
  "@angular/core": {
    singleton: true,
    strictVersion: true,
    requiredVersion: 'auto'
  },
  "@angular/common": {
    singleton: true,
    strictVersion: true,
    requiredVersion: 'auto'
  },
  "@angular/router": {
    singleton: true,
    strictVersion: true,
    requiredVersion: 'auto'
  },
  "@angular/common/http": {
    singleton: true,
    strictVersion: true,
    requiredVersion: 'auto'
  },
  "@angular/material/snack-bar": {
    singleton: true,
    strictVersion: true,
    requiredVersion:'auto'
  },

  // Uncomment for sharing lib of an Angular CLI or Nx workspace
  ...sharedMappings.getDescriptors()
})
```

Issues with Sharing Code and Data

In our example, the `shell` and the micro frontend `mfe1` share the `auth-lib`. Its `AuthService` stores the current user name. Hence, the `shell` can set the user name and the lazy loaded `mfe1` can access it:



Sharing User Name

If `auth-lib` was a traditional npm package, we could just register it as a shared library with module federation. However, in our case, the `auth-lib` is just a library in our monorepo. And libraries in that sense are just folders with source code.

To make this folder look like a npm package, there is a path mapping for it in the `tsconfig.json`:

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

Please note that we are directly pointing to the `src` folder of the `auth-lib`. Nx does this by default. If you go with a traditional CLI project, you need to adjust this by hand.

Fortunately, Module Federation got us covered with such scenarios. To make configuring such cases a bit easier as well as to prevent issues with the Angular compiler, `@angular-architects/module-federation` provides the helper class `SharedMappings`. To use it, just register the mapped path in the `webpack.config.js` of **all affected projects**:

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

Obviously, if you don't opt-in into sharing the library in one of the projects, these project will get their own copy of the `auth-lib` and hence sharing the user name isn't possible anymore.

However, there is a constellation with the same underlying issue that is everything but obvious. To construct this situation, let's add another library to our monorepo:

```
ng g lib other-lib
```

Also, make sure we have a path mapping for it pointing to its source code:

```
"paths": {
  "other-lib": [
    "projects/other-lib/src/public-api.ts"
  ],
}
```

Let's assume we also want to store the current user name in this library:

```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class OtherLibService {

  // Add this:
  userName: string;

  constructor() { }

}
```

And let's also assume, the `AuthLibService` delegates to this property:

```
import { Injectable } from '@angular/core';
import { OtherLibService } from 'other-lib';

@Injectable({
  providedIn: 'root'
})
export class AuthLibService {

  private userName: string;

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

  public get otherUser(): string {
    // DELEGATION!
    return this.otherService.userName;
  }

  constructor(private otherService: OtherLibService) { }

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

    // DELEGATION!
    this.otherService.userName = userName;
```

```
  }

}
```

The shell's `AppComponent` is just calling the `login` method:

```
import { Component } from '@angular/core';
import { AuthLibService } from 'auth-lib';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  title = 'shell';

  constructor(
    private service: AuthLibService
    ) {

    this.service.login('Max', null);
  }

}
```

However, now the Micro Frontend has three ways of getting the defined user name:

```
import { HttpClient } from '@angular/common/http';
import { Component } from '@angular/core';
import { AuthLibService } from 'auth-lib';
import { OtherLibService } from 'other-lib';

@Component({
  selector: 'app-flights-search',
  templateUrl: './flights-search.component.html'
})
export class FlightsSearchComponent {
  constructor(
    authService: AuthLibService,
    otherService: OtherLibService) {

    // Three options for getting the user name:
```

```
16      console.log('user from authService', authService.user);
17      console.log('otherUser from authService', authService.otherUser);
18      console.log('otherUser from otherService', otherService.userName);
19
20    }
21  }
```

At first sight, all these three options should bring up the same value. However, if we only share `auth-lib` **but not** `other-lib`, we get the following result:

```
user from authService Max
otherUser from authService Max
otherUser from otherService undefined
```

Issue with sharing libs

As `other-lib` is not shared, both, `auth-lib` but also the micro frontend get their very own version of it. Hence, we have two instances of it in place here. While the first one knows the user name, the second one doesn't.

What can we learn from this? Well, it would be a good idea to also share the dependencies of our shared libraries (regardless of sharing libraries in a monorepo or traditional npm packages!).

This also holds true for secondary entry points our shared libraries belong to.

Hint: `@angular-architects/module-federation` comes with a helper function `shareAll` for sharing all dependencies defined in your project's `package.json`:

```
shared: {
  ...shareAll({
      singleton: true,
      strictVersion: true,
      requiredVersion: 'auto'
  }),
  ...sharedMappings.getDescriptors()
}
```

This can at least lower the pain in such cases for prototyping. Also, you can make `share` and `shareAll` to include all secondary entry points by using the property `includeSecondaries`:

```
shared: share({
    "@angular/common": {
        singleton: true,
        strictVersion: true,
        requiredVersion: 'auto',
        includeSecondaries: {
            skip: ['@angular/http/testing']
        }
    },
    [...]
})
```

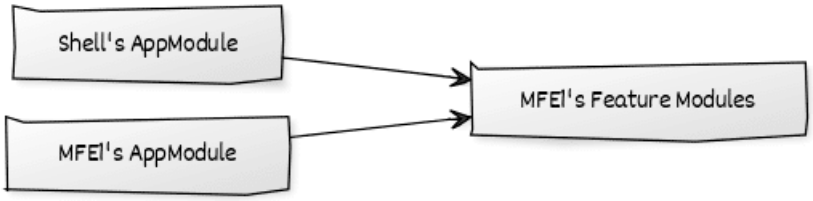
NullInjectorError: Service expected in Parent Scope (Root Scope)

Okay, the last section was a bit difficult. Hence, let's proceed with an easier one. Perhaps you've seen an error like this here:

```
ERROR Error: Uncaught (in promise): NullInjectorError: R3InjectorError(FlightsModule\
)[HttpClient -> HttpClient -> HttpClient -> HttpClient]:
  NullInjectorError: No provider for HttpClient!
NullInjectorError: R3InjectorError(FlightsModule)[HttpClient -> HttpClient -> HttpCl\
ient -> HttpClient]:
  NullInjectorError: No provider for HttpClient!
```

It seems like, the loaded Micro Frontend `mfe1` cannot get hold of the `HttpClient`. Perhaps it even works when running `mfe1` in standalone mode.

The reason for this is very likely that we are not exposing the whole Micro Frontend via Module Federation but only selected parts, e. g. some Features Modules with Child Routes:



Feature Modules exposed via Module Federation

Or to put it in another way: **DO NOT** expose the Micro Frontend's `AppModule`. However, if we expect the `AppModule` to provide some global services like the `HttpClient`, we also need to do this in the shell's `AppModule`:

```
// Shell's AppModule
@NgModule({
  imports: [
    [...]
    // Provide global services your micro frontends expect:
    HttpClientModule,
  ],
  [...]
})
export class AppModule { }
```

Several Root Scopes

In a very simple scenario you might try to just expose the Micro Frontend's `AppModule`.

AppModule loads exposed AppModule

As you see here, now, the shell's `AppModule` uses the Micro Frontend's `AppModule`. If you use the router, you will get some initial issues because you need to call `RouterModule.forRoot` for each `AppModule` (Root Module) on the one side while you are only allowed to call it once on the other side.

But if you just shared components or services, this might work at first sight. However, the actual issue here is that Angular creates a root scope for each root module. Hence, we have two root scopes now. This is something no one expects.

Also, this duplicates all shared services registered for the root scope, e. g. with `providedIn: 'root'`. Hence, both, the shell and the Micro Frontend have their very own copy of these services and this is something, no one is expecting.

A **simple but also non preferable solutions** is to put your shared services into the `platform` scope:

```
// Don't do this at home!
@Injectable({
  providedIn: 'platform'
})
export class AuthLibService {
}
```

However, normally, this scope is intended to be used by Angular-internal stuff. Hence, the only clean solution here is to not share your `AppModule` but only lazy feature modules. By using this practice,

you assure (more or less) that these feature modules work the same when loaded into the shell as when used in standalone-mode.

Different Versions of Angular

Another, less obvious pitfall you can run into is this one here:

```
node_modules_angular_core___ivy_ngcc___fesm2015_core_js.js:6850 ERROR Error: Uncaugh\
t (in promise): Error: inject() must be called from an injection context
Error: inject() must be called from an injection context
    at pr (node_modules_angular_core___ivy_ngcc___fesm2015_core_js.2fc3951af86e4bae0\
c59.js:1)
    at gr (node_modules_angular_core___ivy_ngcc___fesm2015_core_js.2fc3951af86e4bae0\
c59.js:1)
    at Object.e.□fac [as factory] (node_modules_angular_core___ivy_ngcc___fesm2015_c\
ore_js.2fc3951af86e4bae0c59.js:1)
    at R3Injector.hydrate (node_modules_angular_core___ivy_ngcc___fesm2015_core_js.j\
s:11780)
    at R3Injector.get (node_modules_angular_core___ivy_ngcc___fesm2015_core_js.js:11\
600)
    at node_modules_angular_core___ivy_ngcc___fesm2015_core_js.js:11637
    at Set.forEach (<anonymous>)
    at R3Injector._resolveInjectorDefTypes (node_modules_angular_core___ivy_ngcc___f\
esm2015_core_js.js:11637)
    at new NgModuleRef$1 (node_modules_angular_core___ivy_ngcc___fesm2015_core_js.js\
:25462)
    at NgModuleFactory$1.create (node_modules_angular_core___ivy_ngcc___fesm2015_cor\
e_js.js:25516)
    at resolvePromise (polyfills.js:10658)
    at resolvePromise (polyfills.js:10610)
    at polyfills.js:10720
    at ZoneDelegate.invokeTask (polyfills.js:10247)
    at Object.onInvokeTask (node_modules_angular_core___ivy_ngcc___fesm2015_core_js.\
js:28753)
    at ZoneDelegate.invokeTask (polyfills.js:10246)
    at Zone.runTask (polyfills.js:10014)
    at drainMicroTaskQueue (polyfills.js:10427)
```

With `inject() must be called from an injection context` Angular tells us that there are several Angular versions loaded at once.

To provoke this error, adjust your shell's `webpack.config.js` as follows:

```
shared: share({
  "@angular/core": { requiredVersion: 'auto' },
  "@angular/common": { requiredVersion: 'auto' },
  "@angular/router": { requiredVersion: 'auto' },
  "@angular/common/http": { requiredVersion: 'auto' },

  // Uncomment for sharing lib of an Angular CLI or Nx workspace
  ...sharedMappings.getDescriptors()
})
```

Please note, that these libraries are not configured to be singletons anymore. Hence, Module Federation allows loading several versions of them if there is no highest compatible version[49].

Also, you have to know that the shell's `package.json` points to Angular 12.0.0 *without* ^ or ~, hence we exactly need this very version.

If we load a Micro Frontend that uses a different Angular version, Module Federation falls back to loading Angular twice, once the version for the shell and once the version for the Micro Frontend. You can try this out by updating the shell's `app.routes.ts` as follows:

```
{
  path: 'flights',
  loadChildren: () => loadRemoteModule({
      remoteEntry:
        'https://brave-plant-03ca65b10.azurestaticapps.net/remoteEntry.js',
      remoteName: 'mfe1',
      exposedModule: './Module'
    })
    .then(m => m.AppModule)
},
```

To make exploring this a bit easier, I've provided this Micro Frontend via a Azure Static Web App found at the shown URL.

If you start your shell and load the Micro Frontend, you will see this error.

What can we learn here? Well, when it comes to your leading, stateful framework – e. g. Angular – it's a good idea to define it as a singleton. I've written down some details on this and on options for dealing with version mismatches[50] here.

If you really, really, really want to mix and match different versions of Angular, I've got you covered with an chapter of its own in this book. However, you know what they say: Beware of your wishes.

[49] https://www.angulararchitects.io/wp-content/uploads/2021/07/https://www.angulararchitects.io/aktuelles/getting-out-of-version-mismatch-hell-with-module-federation/

[50] https://www.angulararchitects.io/wp-content/uploads/2021/07/https://www.angulararchitects.io/aktuelles/getting-out-of-version-mismatch-hell-with-module-federation/

Bonus: Multiple Bundles

Let's finish this tour with something, that just looks like an issue but is totally fine. Perhaps you've already seen that sometimes Module Federation generated duplicate bundles with slights different names:

```
node_modules_angular_core___ivy_ngcc___fesm2015_core_js.js
node_modules_angular_material___ivy_ngcc___fesm2015_snack-bar_js-_36161.js
node_modules_angular_material___ivy_ngcc___fesm2015_snack-bar_js-_36160.js
node_modules_angular_router___ivy_ngcc___fesm2015_router_js-_48f40.js
node_modules_angular_router___ivy_ngcc___fesm2015_router_js-_48f41.js
node_modules_angular_common___ivy_ngcc___fesm2015_common_js-_57a20.js
node_modules_angular_common___ivy_ngcc___fesm2015_common_js-_57a21.js
node_modules_angular_common___ivy_ngcc___fesm2015_http_js-_f15b0.js
node_modules_angular_common___ivy_ngcc___fesm2015_http_js-_f15b1.js
projects_mfe1_src_bootstrap_ts.js
projects_mfe1_src_app_app_module_ts.js
projects_auth-lib_src_public-api_ts-_77080.js
projects_auth-lib_src_public-api_ts-_77081.js
projects_other-lib_src_public-api_ts-_f8920.js
projects_other-lib_src_public-api_ts-_f8921.js
```

Duplicate Bundles generated by Module Federation

The reason for this duplication is that Module Federation generates a bundle **per shared library per consumer**. The consumer in this sense is the federated project (shell or Micro Frontend) or a shared library. This is done to have a fall back bundle for resolving version conflicts. In general this makes sense while in such a very specific case, it doesn't bring any advantages.

However, if everything is configured in the right way, only one of these duplicates should be loaded at runtime. As long as this is the case, you don't need to worry about duplicates.

Conclusion

Module Federation is really clever when it comes to auto-detecting details and compensating for version mismatches. However, it can only be as good as the meta data it gets. To avoid getting off the rails, you should remember the following:

- **requiredVersion**: Assign the `requiredVersion` by hand, esp. when working with secondary entrypoints and when having peer dependency warnings. The plugin `@angular-architects/module-federation` get's you covered with its `share` helper function allowing the option `requiredVersion: 'auto'` that takes the version number from your project's `package.json`.
- **Share dependencies of shared libraries** too, esp. if they are also used somewhere else. Also think on secondary entry points.

- Make the **shell provide global services** the loaded Micro Frontends need, e. g. the `HttpClient` via the `HttpClientModule`.
- Never expose the `AppModule` via Module Federation. Prefer to expose lazy Feature modules.
- Use `singleton:true` for Angular and other stateful framework respective libraries.
- Don't worry about **duplicated bundles** as long as only one of them is loaded at runtime.

Bonus Chapter: Automate Your Architectures with Nx Workspace Generators

In the first part of the book, I've used the @angular-architects/ddd[51] plugin for automating the creation of domains and layers. I really want to encourage you to check this plugin out as it makes dealing with huge applications easier. However, sometimes you need your own tasks to be automated. Hence, this chapter shows how to do this with Nx Generators.

Schematics vs Generators

Both, Angular beginners and experts appreciate the CLI command `ng generate`. It generates recurring structures, such as the basic structure of components or modules, and thus automates monotonous tasks. Behind this instruction are code generators called schematics. Everyone can leverage the CLI's Schematics API to automate their own tasks. Unfortunately, this API is not always intuitive and often uses unneeded indirections.

Exactly this impassability is solved by Nx which provides a simplified version of the Schematics API called Generators. In this chapter, I'll cover the Generators API. The source code can be found here[52].

Workspace Generators

The easiest way to try out the Generator API are so-called workspace generators. These are generators that are not distributed via npm, but are placed directly in an Nx workspace (monorepo). Such a workspace generator can be created with the following command:

```
1 ng generate @nrwl/workspace: workspace-generator entity
```

The generator created here has the task of setting up an entity and data access service within a data access library. The example is deliberately chosen so that it contains as many generator concepts as possible.

The command places some boilerplate code for the generator in the file `tools/generators/entity/index.ts`. Even if we are going to replace this boilerplate with our own code, it is worth taking a look to get to know generators:

[51] https://www.npmjs.com/package/@angular-architects/ddd
[52] https://github.com/manfredsteyer/nx-workspace-generators

```
export default async function (host: Tree, schema: any) {

  await libraryGenerator(host, { name: schema.name });

  await formatFiles (host);

  return () => {
    installPackagesTask (host);
  };

}
```

The generator itself is just a function that takes two parameters. The first parameter of type `Tree` represents the file system that the generator changes. Strictly speaking, this is just a staging environment: the generator only writes the changes made to the disk at the end, when everything has worked. This prevents a generator that crashes in the middle of execution from leaving an inconsistent state.

If you want to initiate further commands after writing them back to the disk, you can place them within an anonymous function and return this function. An example of this is the call to `installPackagesTask`, which triggers commands like `npm install`, `yarn`, etc. For these instructions to make sense, the generator must first write the desired package names into the `package.json` file.

The second parameter with the name `schema` is an object with the command line parameters used when calling the generator. We'll take a closer look at it below.

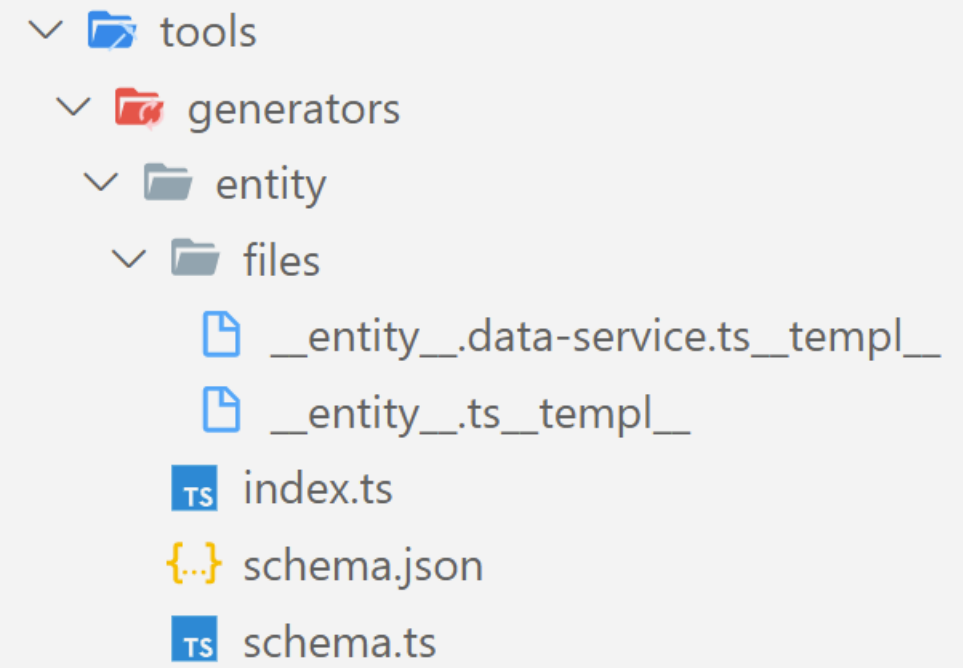
Calling `libraryGenerator` to generate a new library is also interesting here . It is actually a further generator - the one that is used with `ng generate library`. With the classic Schematics API, calling other Schematics was a lot more complicated.

To be fully honest, the call to `libraryGenerator` found in the generated boiler plate only does a subset of `ng generate library` when calling it in an Angular project. It's taken from the package `@nrwl/workspace/generators` that works in a framework-agnostic way. To also perform the tasks specific for Angular, take the `libraryGenerator` function from the `@nrwl/angular/generators` package. This package also contains other Angular-specific generators you very likely use on a regular basis. Examples are generators for generating an angular application (`ng generate application`) but also for generating components or services (`ng generate component`, `ng generate service`).

The instruction `formatFiles` which formats the entire source code, is useful at the end of generators to bring the generated code segments into shape. Nx uses the popular open source project Prettier for this.

Templates

In general, you could create new files using the passed `Tree` object's `write` method. However, using templates are in most cases more fitting. These are source code files with placeholders which the generator replaces with concrete values. While templates can be placed everywhere, it has become common practice to put them in a subfolder called `files`:



Generator with templates

The file names themselves can have placeholders. For example, the generator replaces the expression **entity** in the file name with the content of a variable entity. Also, our generator replaces the **templ** attached to the file with an empty string. With this trick, the template file does not end with `.ts` but in `.ts__templ__`. Thus the TypeScript compiler ignores them. That is a good thing, especially since it does not know the placeholders and would report them as errors.

The templates themselves are similar to classic ASP, PHP, or JSP files:

```
export class <%=classify(entity)%> {
    id: number;
    title: string;
    description: string;
}
```

Expressions within `<%` and `%>` evaluate generators and the combination of `<%=` and `%>` results in an output. This way, the example shown here outputs the name of the entity. The `classify` function formats this name so that it conforms to the rules and conventions of names for TypeScript classes.

The helper function `names` from `@nrwl/devkit` provides similar possibilities as the `strings` object. The expression `names('someName').className`, for instance, returns a class name like `string.classifiy` does.

Defining parameters

Generators can be controlled via command line parameters. You define the permitted parameters in the `schema.json` file that is generated in the same folder as the above shown `index.ts`:

```
{
  "cli": "nx" ,
  "id": "entity" ,
  "type": "object" ,
  "properties": {
    "entity": {
      "type": "string" ,
      "description": "Entity name" ,
      "x-prompt": "Entity name" ,
      "$default": {
        "$source": "argv" ,
        "index": 0
      }
    },
    "project": {
      "type": "string" ,
      "description": "Project name"
    }
  },
  "required": [ "entity" ]
}
```

It is a JSON scheme with the names, data types and descriptions of the parameters. Generators also use extensions introduced by Schematics. For example, the `$default` property defines that the value for the `entity` parameter can be found in the first command line argument (`"index": 0`) if the caller does not explicitly define it using `--entity`.

With `x-prompt`, the example specifies a question to be asked to the caller if there is no value for `entity`. The generator uses the response received for this parameter.

In order to be able to use these parameters in a type-safe manner in the generator, it is recommended to provide a corresponding interface:

```
export interface EntitySchema {
    entity : string;
    project?: string;
}
```

This interface is normally put into a `schema.ts` file. To avoid inconsistencies, it can be generated from the schema file using tools such as json-schema-to-typescript[53].

Implementing the Generator

Now we can turn to the implementation of the generator. For getting the command line parameters in a types way, the second parameter called `schema` is typed with the `EntitySchema` interface we've just been introduced:

```
import {
  Tree,
  formatFiles,
  readProjectConfiguration,
  generateFiles,
  joinPathFragments
} from '@nrwl/devkit';
import { EntitySchema } from './schema';
import { strings } from '@angular-devkit/core';
[...]

export default async function (host: Tree , schema: EntitySchema) {

  const workspaceConf = readWorkspaceConfiguration(host);

  if (!schema.project) {
    schema.project = workspaceConf.defaultProject;
  }

  [...]

  const projConf = readProjectConfiguration(host, schema.project);

  [...]

  await libraryGenerator(host, { name: schema.name });
```

[53] https://www.npmjs.com/package/json-schema-to-typescript

```
27
28    generateFiles(
29      host,
30      path.join(__dirname , 'files'),
31      path.join(projConf.SourceRoot, 'lib'),
32      {
33        entity: strings.dasherize(schema.entity),
34        templ: '' ,
35        ... strings
36      });
37
38    addLibExport(host, projConf, schema.entity);
39
40    await formatFiles (host);
41  }
```

The `readWorkspaceConfiguration` helper function reads the workspace configuration that can be found in files like `nx.json` or `angular.json`. It contains the name of the default project to use if a specific project has not been specified or the npm scope used by the monorepo like `@my-project`.

The `readProjectConfiguration` helper function, however, reads a project configuration. For Angular projects, this data can be found in the `angular.json` file in the main project directory. Both functions can be found in the `@nrwl/devkit` namespace.

The `generateFiles` function, which comes from `@nrwl/devkit` too, takes care of generating files based on the stored templates. In addition to the `Tree` object (`host`), which grants access to the staging environment, this function accepts the source directory with the templates as well as the target directory in which the generated files are to be placed.

The last parameter is an object with variables. In this way the desired entity name gets into the templates. The `dasherize` function transforms this name into a form that corresponds to the custom of filenames. The empty variable `templ` is necessary for the trick mentioned above and the object strings from the territory of the CLI (`@angular-devkit/core`) contains the `classify` function discussed above.

The function `addLibExport` is a self-written utility function. The next section goes into this in more detail.

Update Existing Source Code

So that the generated entity and the data access service are not only visible in their own libraries, they must be exported via their library's `index.ts`. This means that our generator has to expand these files. The `addLibExport` function takes care of this:

```
import { Tree, ProjectConfiguration } from '@nrwl/devkit';
import * as path from 'path';
import { strings } from '@angular-devkit/core';

export function addLibExport(
  host: Tree,
  projConfig: ProjectConfiguration,
  entity: string): void {
    const indexTsPath = path.join(projConfig.sourceRoot, 'index.ts');
    const indexTs = host.read(indexTsPath).toString();

    const entityFileName = `./lib/${strings.dasherize(entity)}`;
    const entityName = strings.classify(entity);
    const dataServiceFileName = `./lib/${strings.dasherize(entity)}.data-service`;
    const dataServiceName = strings.classify(entity) + 'DataService';

    const updatedIndexTs = indexTs + `
  export { ${entityName} } from '${entityFileName}';
  export { ${dataServiceName} } from '${dataServiceFileName}';
  `;

    host.write(indexTsPath, updatedIndexTs);
}
```

This implementation just reads the `index.ts` file, adds some export statements and write it back. In very simple cases this straightforward approach is fine. For more advanced scenarios, one can leverage the `applyChangesToString` helper function:

```
import {
  Tree,
  ProjectConfiguration,
  applyChangesToString,
  ChangeType
} from '@nrwl/devkit';
import * as path from 'path';
import { strings } from '@angular-devkit/core';

export function addLibExport(
  host: Tree,
  projConfig: ProjectConfiguration,
  entity: string): void {
    const indexTsPath = path.join(projConfig.sourceRoot, 'index.ts');
    const indexTs = host.read(indexTsPath).toString();
```

```
16
17      const entityFileName = `./lib/${strings.dasherize(entity)}`;
18      const entityName = strings.classify(entity);
19      const dataServiceFileName = `./lib/${strings.dasherize(entity)}.data-service`;
20      const dataServiceName = strings.classify(entity) + 'DataService';
21
22      const updatedIndexTs = applyChangesToString(
23        indexTs,
24        [
25          {
26            type: ChangeType.Insert,
27            index: indexTs.length,
28            text: `export { ${entityName} } from '${entityFileName}';\n`
29          },
30          {
31            type: ChangeType.Insert,
32            index: indexTs.length,
33            text: `export { ${dataServiceName} } from '${dataServiceFileName}';\n`
34          },
35          // Just for demonstration:
36          // {
37          //   type: ChangeType.Delete,
38          //   start: 0,
39          //   length: indexTs.indexOf('\n') + 1
40          // }
41        ]
42      )
43
44      host.write(indexTsPath, updatedIndexTs);
45  }
```

The function `applyChangesToString` takes care of modifying existing source code. In addition to the current content, it receives an array with objects that describe the desired changes. These objects contain the index positions at which new code is to be inserted or deleted.

Any modification naturally changes the index positions of subsequent lines. Fortunately, we don't have to worry about that – `applyChangesToString` takes this into account.

In this specific case, determining the index positions is very simple: We add the additional `export` instructions at the end of the file. In more complex cases, the `TypeScript API` can be used to parse existing source code files.

The project ts-query[54] helps to make the usage of the `TypeScript API` easier by

[54] https://www.npmjs.com/package/@phenomnomnominal/tsquery

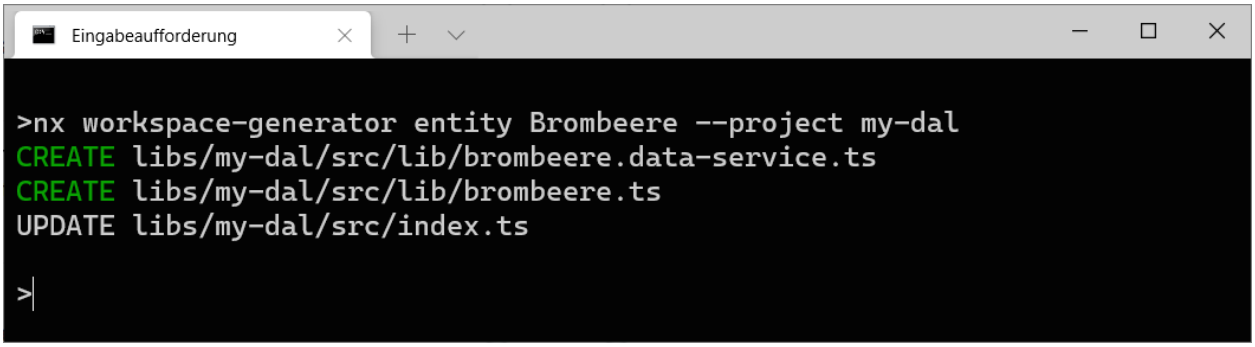
providing a query language for searching for specific constructs in existing code files.

Running the Generator

As our Generator is part of our monorepo, we can directly call it via the Nx CLI:

```
1 nx workspace-generator entity pumpkin-seed-oil --project my-dal
```

This call assumes that the new entity shall be created within a `my-dal` library in your monorepo.



Calling the Generator

Additional Hints

While the example shown here contains most concepts provided by Generators, here I want to give you some additional hints out of my project experience.

Finding Helper Functions

The `@nrwl/devkit` package used above turns out to be a pure gold mine. It contains lots of helper functions that come in handy for your generators. For instance, they allow to read configuration files, read and write json files, or to add dependencies to your `package.json`. Hence, taking some time to investigate functions exported by this package will be worth it.

Besides them, the generators provided by Nx have some additional ones. Hence, you can benefit from looking up their implementations at GitHub. Here, I want to present some of them that helped me in my projects.

For instance, the package `@nrwl/workspace/src/utilities/ast-utils` contains helper functions for dealing with TypeScript code like adding methods or checking for imports. The following example shows a function using this package for adding another `import` statement to a typescript file:

```
import { Tree } from '@nrwl/devkit';
import * as ts from 'typescript';
import { insertImport } from '@nrwl/workspace/src/utilities/ast-utils';

export function addImportToTsModule(tree: Tree, filePath: string, importClassName: s\
tring, importPath: string) {
  const moduleSource = tree.read(filePath, 'utf-8');

  let sourceFile = ts.createSourceFile(
    filePath,
    moduleSource,
    ts.ScriptTarget.Latest,
    true
  );

  // Adds sth like: import { importClassName } from 'importPath'
  insertImport(
    tree,
    sourceFile,
    filePath,
    importClassName,
    importPath);
}
```

The `sourceFile` variable represents the source code file with a type provided by the TypeScript API. It's expected by `insertImport`, hence we need to do this little detour.

Another convenient helper function can be found in the `@nrwl/angular/src/generators/utils` namespace. Its called `insertNgModuleProperty` and allows adding metadata like `imports` or `declarations` to an `NgModule`:

```
import { Tree } from '@nrwl/devkit';
import { insertNgModuleProperty } from '@nrwl/angular/src/generators/utils';

[...]
const filePath = '[...]/app.module.ts';
const importClassName = 'MyOtherModule';

insertNgModuleProperty(tree, filePath, importClassName, 'imports');
```

Using Existing Schematics

Sometimes, you need to reuse an Schematic provided by another package. Fortunately, the simplified Generators API is quite similar to the Schematic API and hence it's possible to create bridges for

converting between these two worlds. For instance, the `wrapAngularDevkitSchematic` helper wraps a Schematic into a Generator at runtime:

```
import { wrapAngularDevkitSchematic } from '@nrwl/devkit/ngcli-adapter';

const generateStore = wrapAngularDevkitSchematic('@ngrx/schematics', 'store');

await generateStore(tree, {
  project: appNameAndDirectoryDasherized,
  root: true,
  minimal: true,
  module: 'app.module.ts',
  name: 'state',
});
```

In this case, the `store` schematic from the `@ngrx/schematics` package is wrapped and invoked. Of course, to make this work, you also need to install the `@ngrx/schematics` package.

Using Existing Generators

Unlike schematics, existing generators are just functions that can be directly called within your generators. We already saw this when we looked at the generated boilerplate for our generator. It contained a call to `libraryGenerator` that was the implementation of `ng g lib`.

Let's assume you already have installed the @angular-architects/ddd[55] plugin that automates some of the stuff described in the first chapters of this book. Let's further assume, you need a bit more to be automated like referencing your design system or creating some default components. In this case, your generator could directly call this plugin's generators:

```
import { Tree } from '@nrwl/devkit';
import generateFeature from '@angular-architects/ddd/src/generators/feature';

export default async function (tree: Tree, schema: any) {

  generateFeature(tree, { name:'my-feature', domain: 'my-domain' });

  // Do additional stuff with generated feature

}
```

[55] https://www.npmjs.com/package/@angular-architects/ddd

Conclusion

Even if Nx Generators look like classic Schematics at first glance, they simplify development enormously. They get by with fewer indirections and are therefore more straightforward. An example of this is the possibility of using other generators directly as functions.

In addition, the Nrwl DevKit offers some auxiliary functions that make working with generators much easier. This includes functions for reading the project configuration or for updating existing source code files. Further helper functions can be found in the Generators Nx ships with.

Literature

- Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software[56]
- Wlaschin, Domain Modeling Made Functional[57]
- Ghosh, Functional and Reactive Domain Modeling[58]
- Nrwl, Monorepo-style Angular development[59]
- Jackson, Micro Frontends[60]
- Burleson, Push-based Architectures using RxJS + Facades[61]
- Burleson, NgRx + Facades: Better State Management[62]
- Steyer, Web Components with Angular Elements (article series, 5 parts)[63]

[56] https://www.amazon.com/dp/0321125215
[57] https://pragprog.com/book/swdddf/domain-modeling-made-functional
[58] https://www.amazon.com/dp/1617292249
[59] https://go.nrwl.io/angular-enterprise-monorepo-patterns-new-book
[60] https://martinfowler.com/articles/micro-frontends.html
[61] https://medium.com/@thomasburlesonIA/push-based-architectures-with-rxjs-81b327d7c32d
[62] https://medium.com/@thomasburlesonIA/ngrx-facades-better-state-management-82a04b9a1e39
[63] https://www.softwarearchitekt.at/aktuelles/angular-elements-part-i/

About the Author

Manfred Steyer

Manfred Steyer is a trainer, consultant, and programming architect with focus on Angular.

For his community work, Google recognizes him as a Google Developer Expert (GDE). Also, Manfred is a Trusted Collaborator in the Angular team. In this role he implemented differential loading for the Angular CLI.

Manfred wrote several books, e. g. for O'Reilly, as well as several articles, e. g. for the German Java Magazine, windows.developer, and Heise.

He regularly speaks at conferences and blogs about Angular.

Before, he was in charge of a project team in the area of web-based business applications for many years. Also, he taught several topics regarding software engineering at a university of applied sciences.

Manfred has earned a Diploma in IT- and IT-Marketing as well as a Master's degree in Computer Science by conducting part-time and distance studies parallel to full-time employments.

You can follow him on Twitter[64] and Facebook[65] and find his blog here[66].

[64] https://twitter.com/ManfredSteyer
[65] https://www.facebook.com/manfred.steyer
[66] http://www.softwarearchitekt.at

Trainings and Consulting

Learn more about this and further architecture topics regarding Angular and huge enterprise as well as industrial solution in our advanced Online Workshop[67]:

Advanced Angular Workshop

Save your ticket[68] for one of our **online or on-site** workshops now or request a company workshop[69] (online or In-House) for you and your team!

Besides this, we provide the following topics as part of our training or consultancy workshops:

- Angular Essentials: Building Blocks and Concepts
- Advanced Angular: Enterprise Solutions and Architecture
- Angular Testing Workshop (Cypress, Just, etc.)
- Reactive Architectures with Angular (RxJS and NGRX)
- Angular Review Workshop
- Angular Upgrade Workshop

[67] https://www.angulararchitects.io/en/angular-workshops/advanced-angular-enterprise-architecture-incl-ivy/
[68] https://www.angulararchitects.io/en/angular-workshops/advanced-angular-enterprise-architecture-incl-ivy/
[69] https://www.angulararchitects.io/en/angular-workshops/

Please find the full list with our offers here[70].

If you like our offer, keep in touch with us so that you don't miss anything.

For this, you can subscribe to our newsletter[71] and/ or follow the book's author on Twitter[72].

[70] https://www.angulararchitects.io/en/angular-workshops/
[71] https://www.angulararchitects.io/subscribe/
[72] https://twitter.com/ManfredSteyer