Enterprise Angular:

Micro Frontends and Moduliths with Angular

6th Extended Edition

Federation - Nx - DDD - Signal Store

MANFRED STEYER

Enterprise Angular: Micro Frontends and Moduliths with Angular

Module Federation - Nx - DDD

Manfred Steyer

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

This version was published on 2024-01-31

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 - 2024 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 **1**
Structure of This Book 1
Trainings and Consultancy 3
Help to Improve this Book! 4
Thanks 4

Strategic Domain-Driven Design **5**
What is Domain-Driven Design? 5
Finding Domains with Strategic Design 5
Domains are Modelled Separately 7
Context-Mapping 8
Sub Domains vs. Bounded Contexts 9
Team Topologies and Strategic Design 10
Conclusion 10

Architectures with Sheriff and Standalone Components **11**
The Architecture Matrix 11
Project Structure for the Architecture Matrix 13
Enforcing your Architecture with Sheriff 14
Lightweight Path Mappings 16
Conclusion 17

Build Performance with Nx **18**
Incremental Builds – Getting Started 18
More comfortable and more powerful: Nx 20
Incremental Builds with Nx 22
Side Note: Micro Frontends 23
Distributed Cache with Nx Cloud 23
Even Faster: Parallelization with Nx Cloud 25
Conclusion 26

Nx & Sheriff - Friends for Life **27**
Module Boundaries in Nx 27
Recap: Different Types of Boundaries 31
Options with Sheriff 32

Conclusion . . . 34

From Domains to Micro Frontends . . . 35
Deployment Monoliths . . . 35
Micro Frontends . . . 35
UI Composition with Hyperlinks . . . 37
UI Composition with a Shell . . . 38
The Hero: Module Federation . . . 40
Finding a Solution . . . 40
Consequences of Micro Frontends . . . 41
Conclusion . . . 42

The Micro Frontend Revolution: Using Module Federation with Angular . . . 43
Example . . . 43
Activating Module Federation for Angular Projects . . . 44
The Shell (aka Host) . . . 45
The Micro Frontend (aka Remote) . . . 47
Trying it out . . . 48
A Further Detail . . . 50
More Details: Sharing Dependencies . . . 50
More on This . . . 51
Conclusion and Evaluation . . . 52

Dynamic Module Federation . . . 54
A Simple Dynamic Solution . . . 55
Going "Dynamic Dynamic" . . . 59
Some More Details . . . 62
Conclusion . . . 64

Plugin Systems with Module Federation: Building An Extensible Workflow Designer . . . 65
Building the Plugins . . . 66
Loading the Plugins into the Workflow Designer . . . 67
Providing Metadata on the Plugins . . . 68
Dynamically Creating the Plugin Component . . . 69
Wiring Up Everything . . . 70
Conclusion . . . 71

Using Module Federation with Nx Monorepos and Angular . . . 72
Multiple Repos vs. Monorepos . . . 72
Multiple Repositories: Micro Frontends by the Book . . . 72
Micro Frontends with Monorepos . . . 74
Monorepo Example . . . 75
The Shared Lib . . . 77
The Module Federation Configuration . . . 78

Trying it out . . . 80
Isolating Micro Frontends . . . 81
Incremental Builds . . . 83
Deploying . . . 84
Conclusion . . . 85

Dealing with Version Mismatches in Module Federation . . . 86
Example Used Here . . . 86
Semantic Versioning by Default . . . 88
Fallback Modules for Incompatible Versions . . . 89
Differences With Dynamic Module Federation . . . 90
Singletons . . . 92
Accepting a Version Range . . . 94
Conclusion . . . 96

Multi-Framework and -Version Micro Frontends with Module Federation . . . 97
Pattern or Anti-Pattern? . . . 98
Micro Frontends as Web Components? . . . 98
Do we also need Module Federation? . . . 99
Implementation in 4 steps . . . 100

Pitfalls with Module Federation and Angular . . . 109
"No required version specified" and Secondary Entry Points . . . 109
Unobvious Version Mismatches: Issues with Peer Dependencies . . . 112
Issues with Sharing Code and Data . . . 114
NullInjectorError: Service expected in Parent Scope (Root Scope) . . . 120
Several Root Scopes . . . 121
Different Versions of Angular . . . 121
Bonus: Multiple Bundles . . . 123
Conclusion . . . 124

Module Federation with Angular's Standalone Components . . . 125
Router Configs vs. Standalone Components . . . 125
Initial Situation: Our Micro Frontend . . . 126
Activating Module Federation . . . 127
Static Shell . . . 128
Alternative: Dynamic Shell . . . 130
Bonus: Programmatic Loading . . . 132

From Module Federation to esbuild and Native Federation . . . 135
Native Federation with esbuild . . . 135
Native Federation: Setting up a Micro Frontend . . . 137
Native Federation: Setting up a Shell . . . 139
Exposing a Router Config . . . 140

Communication between Micro Frontends . . . 142
Conclusion . . . 144

The new NGRX Signal Store for Angular: 3 + n Flavors . . . 145
Getting the Package . . . 145
Flavor 1: Lightweight with signalState . . . 145
Side Effects . . . 147
Flavor 2: Powerful with signalStore . . . 149
Custom Features - n Further Flavors . . . 153
Flavor 3: Built-in Features like Entity Management . . . 156
Conclusion . . . 158

Smarter, Not Harder: Simplifying your Application With NGRX Signal Store and Custom Features . . . 160
Goal . . . 160
DataService Custom Feature . . . 162
Implementing A Generic Custom Feature . . . 163
Providing a Fitting Data Service . . . 165
Undo/Redo-Feature . . . 166
Using the Store in a Component . . . 168
Conclusion and Outlook . . . 169

NGRX Signal Store Deep Dive: Flexible and Type-Safe Custom Extensions . . . 170
A Simple First Extension . . . 170
Now it Really Starts: Typing . . . 171
Typing and Dynamic Properties – How do They Work Together? . . . 174
More Examples: CRUD and Undo/Redo . . . 178
Out of the Box Extensions . . . 178
Conclusion . . . 179

The NGRX Signal Store and Your Architecture . . . 180
Where to Put it? . . . 180
Combining the Signal Store With the Traditional NGRX Store . . . 182
The Best of Both Worlds via Custom Features . . . 183
How Large Should a Signal Store be? . . . 183
May a Signal Store Access Other Signal Stores? . . . 183
Preventing Cycles, Redundancies, and Inconsistencies . . . 184
Conclusion . . . 184

Bonus: Automate your Architecture with Nx Workspace Plugins . . . 186
Creating a Workspace Plugin With a Generator . . . 186
Templates for Generators . . . 187
Implementing a Generator . . . 187
True Treasures: Helper Methods for Generators in Nx . . . 189

Trying out Generators . . . 189
Testing Generators . . . 190
Exporting Plugins via NPM . . . 191
Conclusion . . . 192

Bonus: The Core of Domain-Driven Design . . . 193
DDD in a Nutshell . . . 193
How to Define DDD? . . . 195
When Can we Call it DDD? . . . 196
What's the Core of DDD and Why did People get a Wrong Impression About that? . . . 197
Is Tacticial Design Object-Oriented? Is There a Place for FP? . . . 197
Further Adaptions of Tactical Design . . . 198
Conclusion . . . 199

Literature . . . 200

About the Author . . . 201

Trainings and Consulting . . . 202

Introduction

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

One vital aspect is decomposing the system into smaller modules to reduce complexity. However, if this results in countless small modules 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.

Another topics I'm adding to this edition of the book is the new NGRX Signal Store. It's lightweight, fully Signal-based and highly extensible. However, it also changes some rules known from the Redux-based world.

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`. You also find me on Twitter[1] and on Facebook[2]. Let's stay in touch for updates about my work on Angular for enterprise-scale applications!

Structure of This Book

This book is subdivided into 20 chapters grouped to four parts discussing different aspects of your architecture.

Part 1: Strategic Design with Nx and Sheriff

This part introduces an reference architecture acting as the leading theory of our work. This architecture can be adapted to different needs. In this first part, it's implemented using an Nx monorepo and Sheriff.

Chapters in part 1:

- Strategic Domain-Driven Design
- Architectures with Sheriff and Standalone Components .
- Build Performance with Nx

[1] https://twitter.com/ManfredSteyer
[2] https://www.facebook.com/manfred.steyer

- Nx & Sheriff - Friends for Life

Part 2: Micro Frontends with Federation

Here we discuss how different domains can be implemented using Micro Frontends. For this, we look into several aspects of Module Federation and discuss the new tooling-agnostic Native Federation.

Chapters in part 2:

- From Domains to Micro Frontends
- The Micro Frontend Revolution: Using Module Federation with Angular
- Dynamic Module Federation
- Plugin Systems with Module Federation: Building An Extensible Workflow Designer
- Using Module Federation with Nx Monorepos and Angular
- Dealing with Version Mismatches in Module Federation
- Multi-Framework and -Version Micro Frontends with Module Federation
- Pitfalls with Module Federation and Angular .
- Module Federation with Angular's Standalone Components
- From Module Federation to esbuild and Native Federation .

Part 3: State Management with the new NGRX Signal Store

Most Angular applications need to preserve some state. The new NGRX Signal Store is a lightweight and highly extensible state management solution we see a lot of potential in. This part shows how it works, how it can be extended to cover repeating requirements, and how it fits your architecture.

Chapters in part 3:

- The new NGRX Signal Store for Angular: 3 + n Flavors
- Smarter, Not Harder: Simplifying your Application With NGRX Signal Store and Custom Features
- NGRX Signal Store Deep Dive: Flexible and Type-Safe Custom Extensions
- The NGRX Signal Store and Your Architecture .

Part 4: Bonus Chapters

The bonus chapters provide some further ideas and insights for your architectures.

Chapters in part 4:

- Automate your Architecture with Nx Workspace Plugins
- The Core of Domain-Driven Design

Trainings and Consultancy

If you and your team need support or trainings regarding Angular, we are happy to help with workshops and consultancy (on-site or remote). 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, Jest, etc.)
- Angular Performance Workshop
- Angular Design Systems Workshop (Figma, Storybook, etc.)
- Angular: Reactive Architectures (RxJS and NGRX)
- Angular Review Workshop
- Angular Upgrade Workshop

Please find the full list of our offers here[3].

Advanced Angular Workshop

We provide our offer in various forms: **remote** or **on-site**; **public** or as **dedicated company workshops**; in **English** or in **German**.

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

[3] https://www.angulararchitects.io/en/angular-workshops/

Help to Improve this Book!

Please let me know if you have any suggestions. Send a pull request to the book's GitHub repository[4].

Thanks

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

- The great people at Nrwl.io[5] who provide the open-source tool Nx[6] used in the case studies here and described in the following chapters.
- Thomas Burleson[7] 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[8] and Jack Herrington[9] helped me to understand the API for Dynamic Module Federation.
- The awesome Tobias Koppers[10] gave me valuable insights into this topic and
- The one and only Dmitriy Shekhovtsov[11] helped me using the Angular CLI/webpack 5 integration for this.

[4] https://github.com/manfredsteyer/ddd-bk
[5] https://nrwl.io/
[6] https://nx.dev/angular
[7] https://twitter.com/thomasburleson?lang=de
[8] https://twitter.com/ScriptedAlchemy
[9] https://twitter.com/jherr
[10] https://twitter.com/wSokra
[11] 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[12] (DDD) approach.

What is Domain-Driven Design?

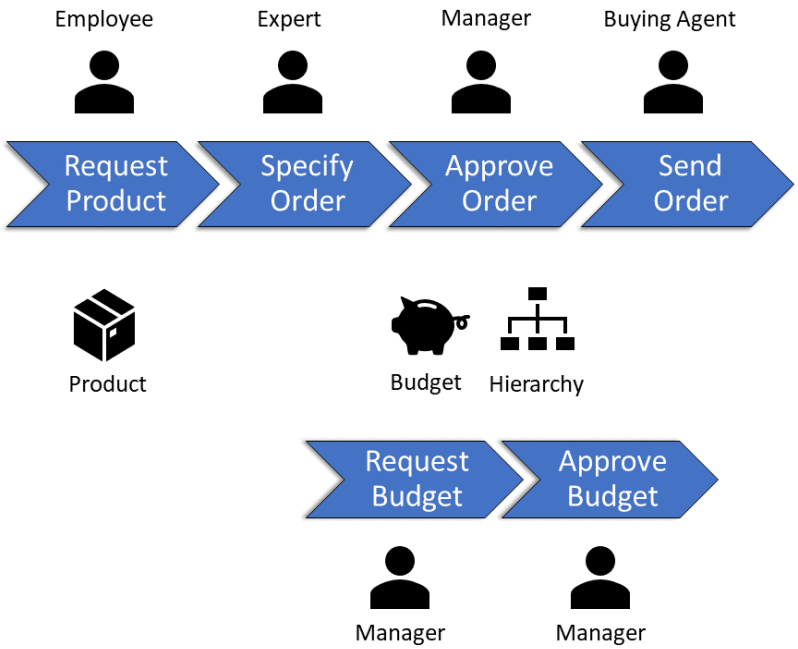
DDD describes an approach that bridges the gap between the requirements for complex software systems and an appropriate application design. Historically, DDD came with two disciplines: tactical design and strategic design. Tactical design proposes concrete concepts and design patterns. Meanwhile most 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:

[12] 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 slicing 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.

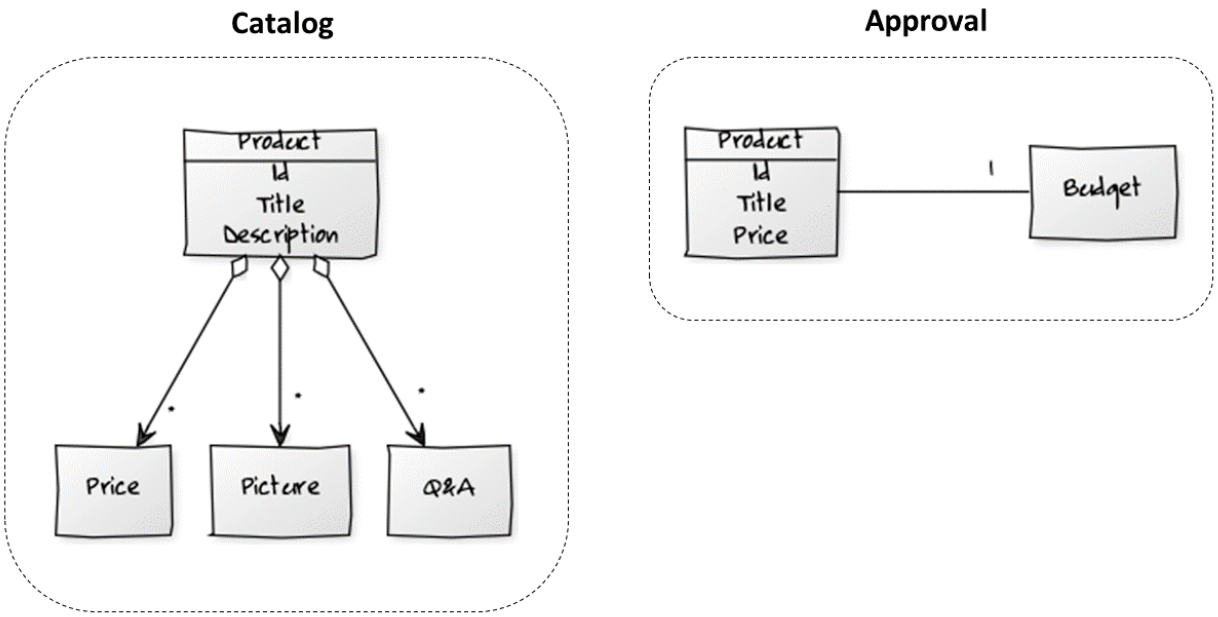
A modern approach for brining the knowledge of all these different people together is Event Storming[13]. It's a workshop format where different groups of stakeholders. For this, they model the processes together with post-its (sticky notes).

[13] 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[14]. 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

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

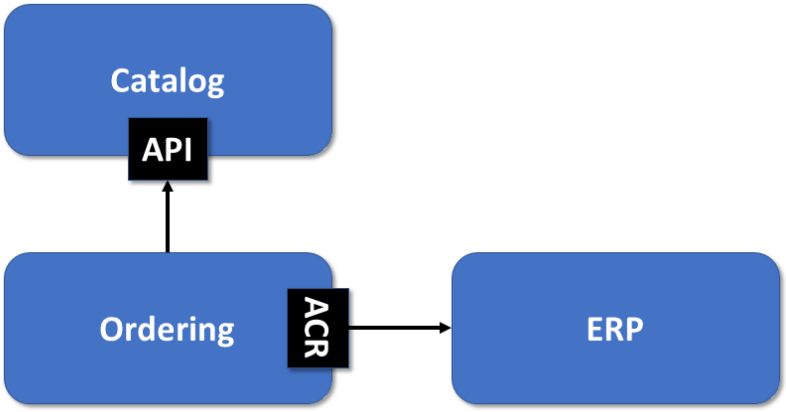
within our implementation. This makes the system more self-describing and reduces the risk for misunderstandings.

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`.

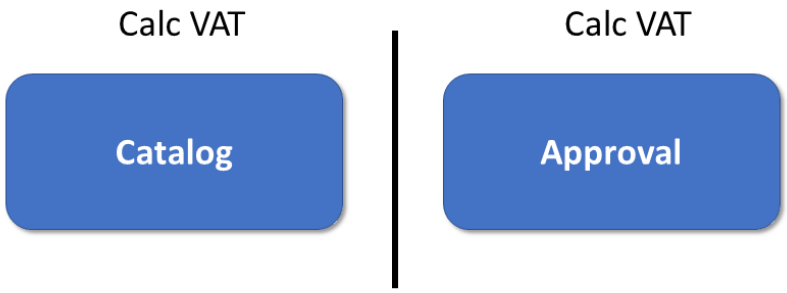
Strategic design defines several ways for dealing with such situations. For instance, in the context map shown above, `Catalog` offers an API (DDD calls it an `open/host service`) that exposes only

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.

Sub Domains vs. Bounded Contexts

Sub Domains and Bounded Contexts are two sides of the same coin: While the term Sub Domain refers to an area of the real-world (problem space), the bounded context represents a part of the solution (solution space).

Ideally, both concepts align with each other, as our goal is to mirror the real world. However, there might be situations where you decide for divergent contexts. An example often mentioned is the presence of a legacy system overlapping several sub domains. On the other side, you might decide to decompose a sub domain into several bounded contexts for technical reasons or to be capable of assigning them to different teams.

Team Topologies and Strategic Design

Team Topologies is a relatively young concept for organizing software development teams. It describes several kinds of teams and patterns of how such teams interact with each other. It also emphasizes the importance of ensuring that the team design allows the individual teams to deal with the cognitive load correlating with its responsibilities.

For subdividing a software system into several parts that can be assigned to different teams, Team Topologies defines the notion of fracture planes. The favored fracture plane is a bounded context mirroring a subdomain in the sense of DDD.

Besides this, there are several other possible fracture planes:

- Regulatory Compliance
- Change Cadence
- Team Location
- Risk
- Performance Isolation
- Technology
- User Personas

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[15]-based monorepo.

[15] https://nx.dev/

Architectures with Sheriff and Standalone Components

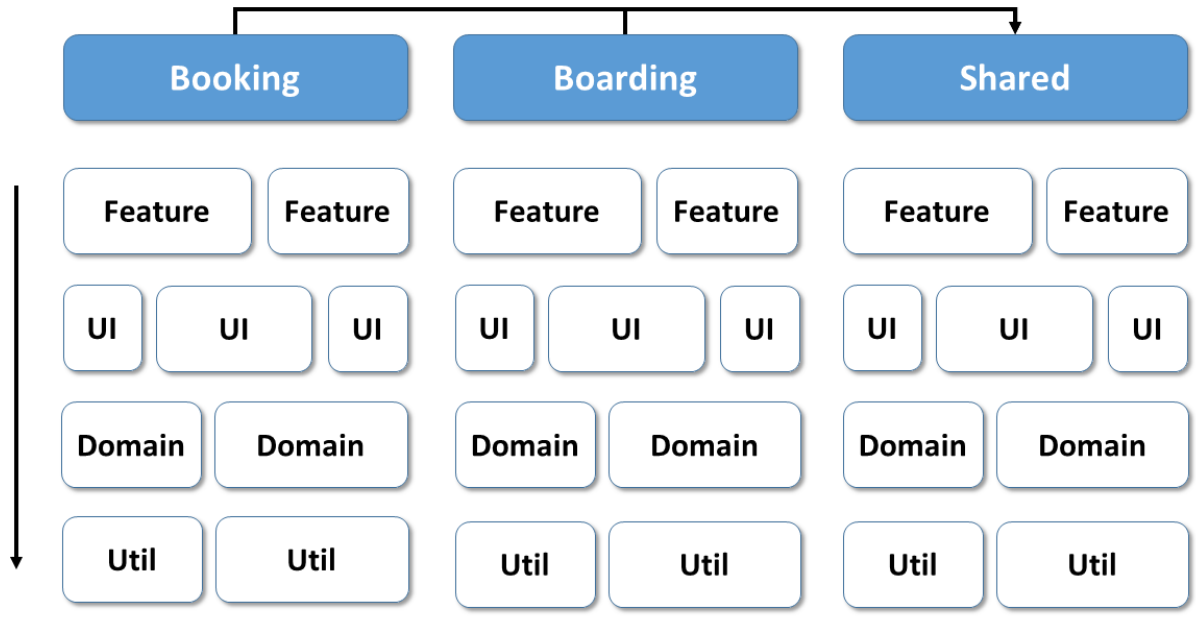
In the previous chapter, I've shown how to define your Strategic Design. This chapter highlights the implementation of your Strategic Design based on Standalone Components and Standalone APIs. The specified architecture is enforced with the open-source project Sheriff.

The examples used here work with a traditional Angular CLI-Project but also with Nx the next chapter focuses on.

Source Code[16]

The Architecture Matrix

For implementing our strategic design, it makes sense to further subdivide the individual domains into different modules:



Architecture Matrix

[16] https://github.com/manfredsteyer/modern-arc.git

This matrix is often the starting point of our projects and can be tailored to individual needs. Each cell results in a module in the source code. Nrwl[17] suggests the following categories (originally for libraries), among others, which have proven helpful in our daily work:

- **feature**: A feature module implements a use case with so-called smart components. Due to their focus on a feature, such components are not very reusable. Smart Components communicate with the backend. Typically, in Angular, this communication occurs through a store or services.
- **ui**: UI modules contain so-called dumb or presentational components. These are reusable components that support the implementation of individual features but do not know them directly. The implementation of a design system consists of such components. However, UI modules can also contain general technical components that are used across all use cases. An example of this would be a ticket component, which ensures that tickets are presented in the same way in different features. Such components usually only communicate with their environment via properties and events. They do not get access to the backend or a store outside of the module.
- **data**: Data modules contain the respective domain model (actually the client-side view of it) and services that operate on it. Such services validate e.g. Entities and communicate with the backend. State management, including the provision of view models, can also be accommodated in data modules. This is particularly useful when multiple features in the same domain are based on the same data.
- **util**: General helper functions etc. can be found in utility modules. Examples of this are logging, authentication, or working with date values.

Another special aspect of the implementation in the code is the shared area, which offers code for all domains. This should primarily have technical code – use case-specific code is usually located in the individual domains.

The structure shown here brings order to the system: There is less discussion about where to find or place certain sections of code. In addition, two simple but effective rules can be introduced on the basis of this matrix:

- In terms of strategic design, each domain may only communicate with its own modules. An exception is the shared area to which each domain has access.
- Each module may only access modules in lower layers of the matrix. Each module category becomes a layer in this sense.

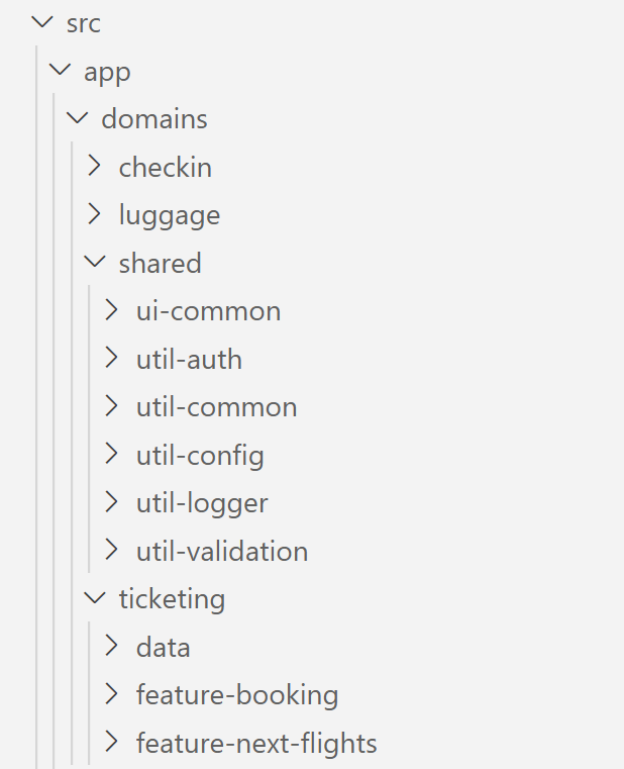
Both rules support the decoupling of the individual modules or domains and help to avoid cycles.

Being a reference architecture, this matrix is often adopted to project-specific needs. Some teams simplify it by reducing the amount of layers and access rules; some teams add additional ones. In some projects, the data layer is called domain or state layer and there are projects where the aspects of these different names end up in layers of their own.

[17] https://go.nrwl.io/angular-enterprise-monorepo-patterns-new-book

Project Structure for the Architecture Matrix

The architecture matrix can be mapped in the source code in the form of folders: Each domain has its own folder, which in turn has a subfolder for each of its modules:



Folder Structure for Architecture Matrix

The module names are prefixed with the name of the respective module category. This means that you can see at first glance where the respective module is located in the architecture matrix. Within the modules are typical Angular building blocks such as components, directives, pipes, or services.

The use of Angular modules is no longer necessary since the introduction of standalone components (directives and pipes). Instead, the *standalone* flag is set to *true:*

```
@Component({
  selector: 'app-flight-booking',
  standalone: true,
  imports: [CommonModule, RouterLink, RouterOutlet],
  templateUrl: './flight-booking.component.html',
  styleUrls: ['./flight-booking.component.css'],
})
export class FlightBookingComponent {
}
```

In the case of components, the so-called compilation context must also be imported. These are all other standalone components, directives and pipes that are used in the template.

An *index.ts* is used to define the module's public interface. This is a so-called barrel that determines which module components may also be used outside of the module:

```
1  export * from './flight-booking.routes';
```

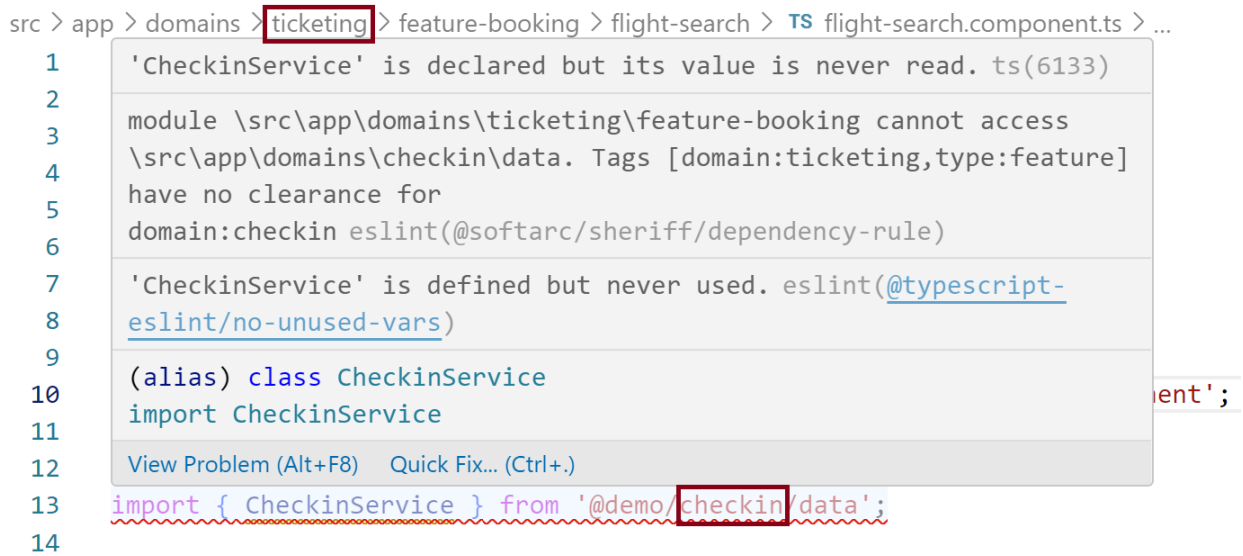
Care should be taken in maintaining the published constructs, as breaking changes tend to affect other modules. Everything that is not published here, however, is an implementation detail of the module. Changes to these parts are, therefore, less critical.

Enforcing your Architecture with Sheriff

The architecture discussed so far is based on several conventions:

- Modules may only communicate with modules of the same domain and *shared*
- Modules may only communicate with modules on below layers
- Modules may only access the public interface of other modules

The Sheriff[18] open-source project allows these conventions to be enforced via linting. Violation is warned with an error message in the IDE or on the console:



Sheriff informs about architecture violations

[18] https://github.com/softarc-consulting/sheriff

The error message in the IDE provides instant feedback during development and the linter output on the console can be used to automate these checks in the build process. Hence, source code that violates the defined architecture can be prevented being committed.

To set up Sheriff, the following two packages must be obtained via npm:

```
npm i @softarc/sheriff-core @softarc/eslint-plugin-sheriff -D
```

The former includes Sheriff, the latter is the bridge to *eslint*. To use this bridge, it must be registered in the *.eslintrc.json* found in the project root:

```
{
  [...],
  "overrides": [
    [...]
    {
      "files": ["*.ts"],
      "extends": ["plugin:@softarc/sheriff/default"]
    }
  ]
}
```

Sheriff considers any folder with an *index.ts* as a module. By default, Sheriff prevents this *index.js* from being bypassed and thus access to implementation details by other modules. The *sheriff.config.ts* to be set up in the root of the project defines categories (*tags*) for the individual modules and defines dependency rules (*depRules*) based on them. The following shows a Sheriff configuration for the architecture matrix discussed above:

```
import { noDependencies, sameTag, SheriffConfig } from '@softarc/sheriff-core';

export const sheriffConfig: SheriffConfig = {
  version: 1,

  tagging: {
    'src/app': {
      'domains/<domain>': {
        'feature-<feature>': ['domain:<domain>', 'type:feature'],
        'ui-<ui>': ['domain:<domain>', 'type:ui'],
        'data': ['domain:<domain>', 'type:data'],
        'util-<ui>': ['domain:<domain>', 'type:util'],
      },
    },
  },
```

```
  depRules: {
    root: ['*'],

    'domain:*': [sameTag, 'domain:shared'],

    'type:feature': ['type:ui', 'type:data', 'type:util'],
    'type:ui': ['type:data', 'type:util'],
    'type:data': ['type:util'],
    'type:util': noDependencies,
  },
};
```

The tags refer to folder names. Expressions such as `<domain>` or `<feature>` are placeholders. Each module below `src/app/domains/<domain>` whose folder name begins with `feature-*` is therefore assigned the categories `domain:<domain>` and `type:feature`. In the case of `src/app/domains/booking`, these would be the categories `domain:booking` and `type:feature`.

The dependency rules under `depRules` pick up the individual categories and stipulate, for example, that a module only has access to modules in the same domain and to `domain:shared`. Further rules define that each layer only has access to the layers below it. Thanks to the `root: ['*']` rule, all non-explicitly categorized folders in the root folder and below are allowed access to all modules. This primarily affects the shell of the application.

Lightweight Path Mappings

Path mappings can be used to avoid unreadable relative paths within the imports. These allow, for example, instead of

```
import { FlightBookingFacade } from '../../data';
```

to use the following:

```
import { FlightBookingFacade } from'@demo/ticketing/data' ;
```

Such three-part imports consist of the project name or name of the workspace (e.g. `@demo`), the domain name (e.g. `ticketing`), and a module name (e.g. `data`) and thus reflect the desired position within the architecture matrix.

This notation can be enabled independently of the number of domains and modules with a single path mapping within *tsconfig.json* in the project root:

```
{
  "compileOnSave": false,
  "compilerOptions": {
    "baseUrl": "./",
    [...]
    "paths": {
      "@demo/*": ["src/app/domains/*"],
    }
  },
  [...]
}
```

IDEs like Visual Studio Code should be restarted after this change. This ensures that they take this change into account.

The build system Nx, introduced in the next chapter, adds such path mappings automatically to your project when adding a library.

Conclusion

Strategic design subdivides a system into different ones that are implemented as independently as possible. This decoupling prevents changes in one area of application from affecting others. The architecture approach shown subdivides the individual domains into different modules, and the open-source project Sheriff ensures that the individual modules only communicate with one another in respecting the established rules.

This approach allows the implementation of large and long-term maintainable frontend monoliths. Due to their modular structure, the language is sometimes also of moduliths. A disadvantage of such architectures is increased build and test times. This problem can be solved with incremental builds and tests. The next chapter addresses this.

Build Performance with Nx

So far, we laid the foundation for a maintainable Angular architecture. We've been thinking about domain slicing, categorizing modules, and enforcing rules based on them with Sheriff.

This chapter supplements our solution with measures to improve build performance. For this, we will switch to the well-known build system Nx.

 Source Code[19] (see different branches)

Incremental Builds – Getting Started

Incremental builds are about just rebuilding changed parts of the repository and hence allow for tremendously speeding up the build process. For this purpose, the solution is divided into several applications. This means only the application that has just been changed must be rebuilt. The same applies to running tests.

The following statement creates another application in a workspace:

```
ng g app miles
```

Libraries can be used to share code between applications:

```
ng g lib auth
```

All applications and libraries set up this way are part of the same workspace and repo. It is, therefore, not necessary to distribute the libraries via npm:

[19] https://github.com/manfredsteyer/modern-arc.git

Folder structure of a library

The file `public-api.ts`, sometimes also called `index.ts`, has a special task. It defines the library's public API:

```
1  // public-api.ts
2
3  export * from "./lib/auth.service";
```

All constructs published here are visible to other libraries and applications. The rest is considered a private implementation detail. In order to grant other libraries and applications in the same workspace access to a library, a corresponding path mapping must be set up in the central `tsconfig.json`:

```
[…]
"paths": {
  "@demo/auth": [
    "auth/src/public-api.ts"
  ],
  […]
}
[…]
```

Calling `ng g lib` takes care of this path mapping. However, the implementation of the Angular CLI makes point to the `dist` folder and, therefore, to the compiled version. This means the author would have to rebuild the library after every change. To avoid this annoying process, the previous listing has the mapping point to the library's source code version. Unlike the CLI, the below-mentioned tool Nx takes care of this automatically.

Once path mapping is set up, individual applications and libraries can import public API exports:

```
import { AuthService } from "@demo/auth";
```

More comfortable and more powerful: Nx

The solution in the last section is simple, but it has a catch: developers must know which applications have changed and manually trigger the corresponding build command. And the build server probably still has to rebuild and test all applications to be on the safe side.

It would be better to let the tooling figure out which applications have changed. To do this, you could calculate a hash value for all source files that flow into your applications. Whenever a hash value changes, it can be assumed that the corresponding application needs to be rebuilt or tested.

Nx is a popular solution that supports this idea and comes with many additional features. In addition to Angular, it also supports other technologies such as React or Node.js-based backends, and integrates numerous tools commonly used in the development of web-based solutions. Examples are the testing tools Jest, Cypress, and Playwright, the npm server *verdaccio,* and Storybook used for interactive component documentation. Developers do not have to invest any effort in setting up such tools but can get started straight away.

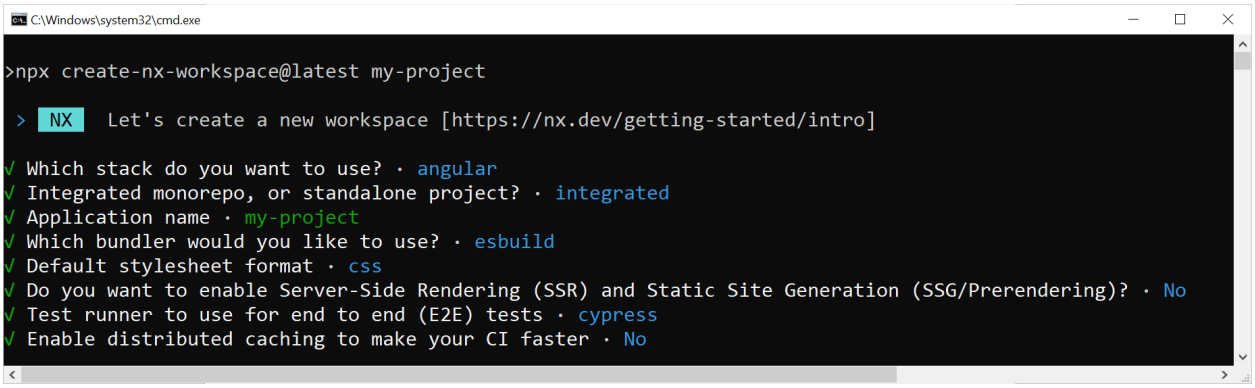
For incremental builds, Nx uses a build cache. Because Nx analyzes dependencies between the individual program parts, these mechanisms often require no manual configuration. Nx feels very natural, especially for Angular developers: The Nx CLI can be used similarly to the Angular CLI. You simply swap out the `ng` instruction for `nx` - the usual arguments remain largely the same (`nx build`, `nx serve`, `nx g app`, `nx g lib`, etc.). The Nx CLI is installed via npm:

```
npm i -g nx
```

To create a new Nx workspace, run the following command:

```
npx create-nx-workspace@latest my-project
```

For your first project, select the following options:



Options for your first Nx project

This command causes npm to load a script that sets up an Nx workspace with the current Nx version. There are also scripts for migrating CLI workspaces to Nx, although they do not always activate the full range of Nx features. For this reason, we had better experiences creating a new Nx workspace and – if necessary – copying over the existing source code. As usual with the Angular CLI, the workspace can then be divided into several applications and libraries:

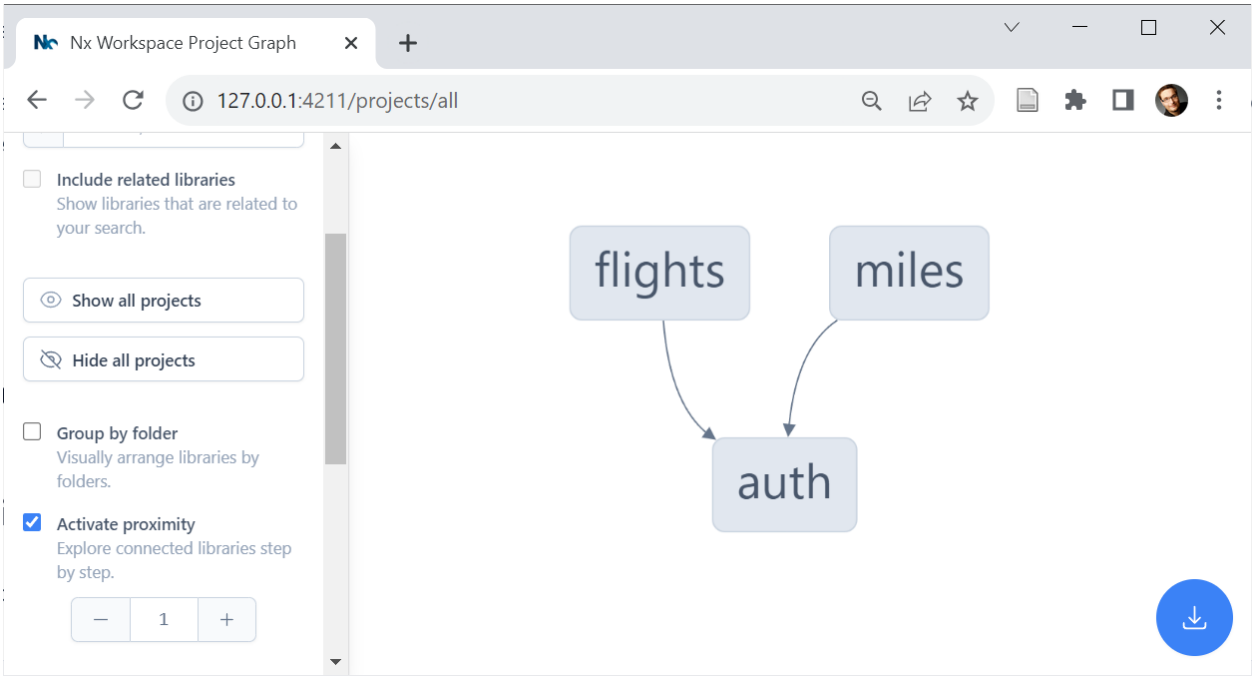
```
nx g app apps/appName

nx g lib libs/libName
```

It's a usual Nx convention to place Angular apps in the `apps` folder and Angular libs in the `libs` folder. Also here, use the default settings for your first Nx projects. However, I would suggest one exception to this rule: Start with the new esbuild builder as it provides a better build performance compared to the traditional webpack-based one.

A call to

```
nx graph
```

illustrates the dependencies between applications and libraries:



A simple dependency graph

Incremental Builds with Nx

The data used for the dependency graph is also the basis for incremental builds that Nx offers out of the box. To build a specific project, you can go with `nx build`:

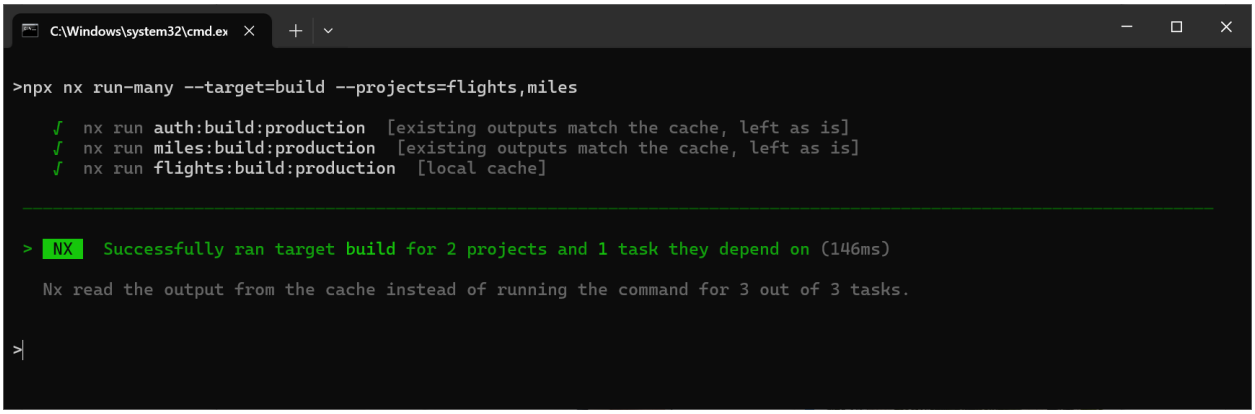
```
nx build miles
```

If the source files that flow into the affected application have not changed, you will immediately receive the result from the local cache. By default, this is located in a `.nx` folder excluded in your project's `.gitignore`.

Nx can also be instructed to rebuild certain or all projects:

```
npx nx run-many --target=build --projects=flights,miles

npx nx run-many --target=build --all
```

In this case, too, Nx falls back to the cache if the source files have not changed:



Nx allows incremental builds without configuration

Unit tests, E2E tests, and linting can also be carried out incrementally in the same way. Nx even goes one step further and caches these actions at the library level. This improves performance by dividing the application across several libraries.

In principle, this would also be possible for `nx build`, provided individual libraries are created as `buildable` (`nx g lib myLib --buildable`). However, it has been shown that this approach rarely leads to performance advantages and that incremental application rebuilds are preferable.

Side Note: Micro Frontends

The separately built applications can be integrated at runtime, giving users the feeling of working with a single application. For this purpose, techniques known from the world of micro frontends are used. This topic is discussed in several other chapters.

Distributed Cache with Nx Cloud

By default, Nx sets up a local cache. If you want to go one step further, use a distributed cache to which the entire project team and the build server can access. This means you also benefit from the builds that others have already performed. The Nx Cloud[20] – a commercial add-on to the free Nx – offers such a cache. If you don't want to or aren't allowed to use cloud providers, you can also host the Nx Cloud yourself.

To connect an Nx workspace to the Nx Cloud, all it needs is one command:

```
npx nx connect-to-nx-cloud
```

Technically, this command activates the `nx-cloud` task runner in the `nx.json` located in the project root:

[20] https://nx.app/

```
"tasksRunnerOptions": {
  "default": {
    "runner": "nx-cloud",
    "options": {
      "cacheableOperations": [
        "build",
        "test",
        "lint"
      ],
      "accessToken": "[…]"
    }
  }
},
```

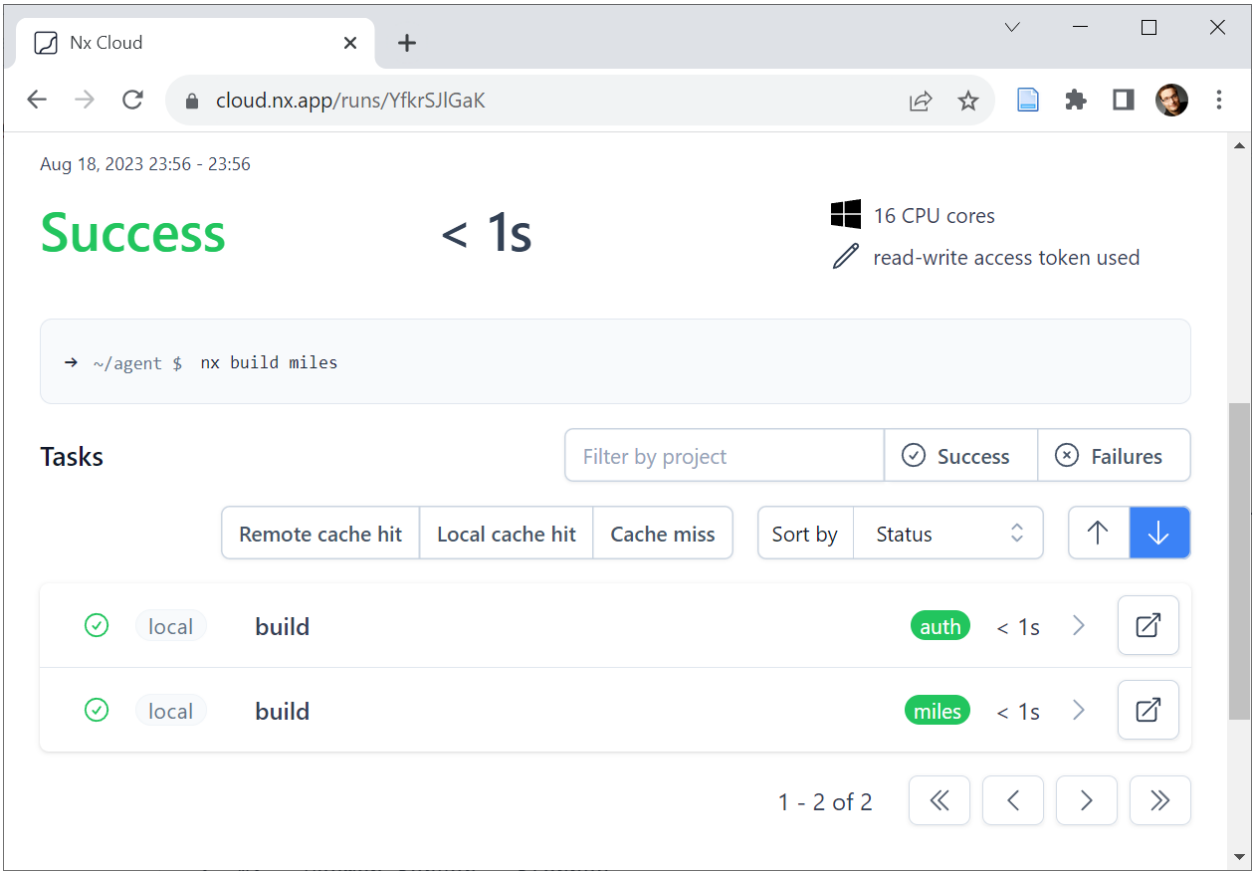
A task runner takes care of the execution of individual tasks, such as those behind `nx build`, `nx lint` or `nx test`. The default implementation caches the results of these tasks in the file system, as discussed above. The `nx-cloud` Task Runner, on the other hand, delegates to an account in the Nx Cloud.

This also shows that the task runner and, thus, the caching strategy can be exchanged relatively easily. Some open-source projects take advantage of this and offer task runners that leverage their own data sources like AWS (see here[21] and here[22]), GCP[23], Azure[24], or Minio[25]. Thanks to Lars Gyrup Brink Nielsen[26] for pointing me to these solutions.

However, it should be noted that the task runner's API is not public and can, therefore, change from version to version.

The task runner for the Nx Cloud also needs to be configured with an access token (see above). Commands like *nx build* output a link to a dynamically created cloud account. When accessing for the first time, it is advisable to create users to restrict access to them. You can also find a dashboard under this link that provides information about the builds carried out:

[21] https://www.npmjs.com/package/@magile/nx-distributed-cache
[22] https://github.com/bojanbass/nx-aws
[23] https://github.com/MansaGroup/nx-gcs-remote-cache
[24] https://npmjs.com/package/nx-remotecache-azure
[25] https://npmjs.com/package/nx-remotecache-minio
[26] https://twitter.com/LayZeeDK



The Nx dashboard provides information about completed builds

Even Faster: Parallelization with Nx Cloud

To further accelerate the build process, the Nx Cloud offers the option of parallelizing individual build tasks. Here, too, the dependency graph proves to be an advantage: Nx can use it to find out the order in which individual tasks must take place or which tasks can be parallelized.

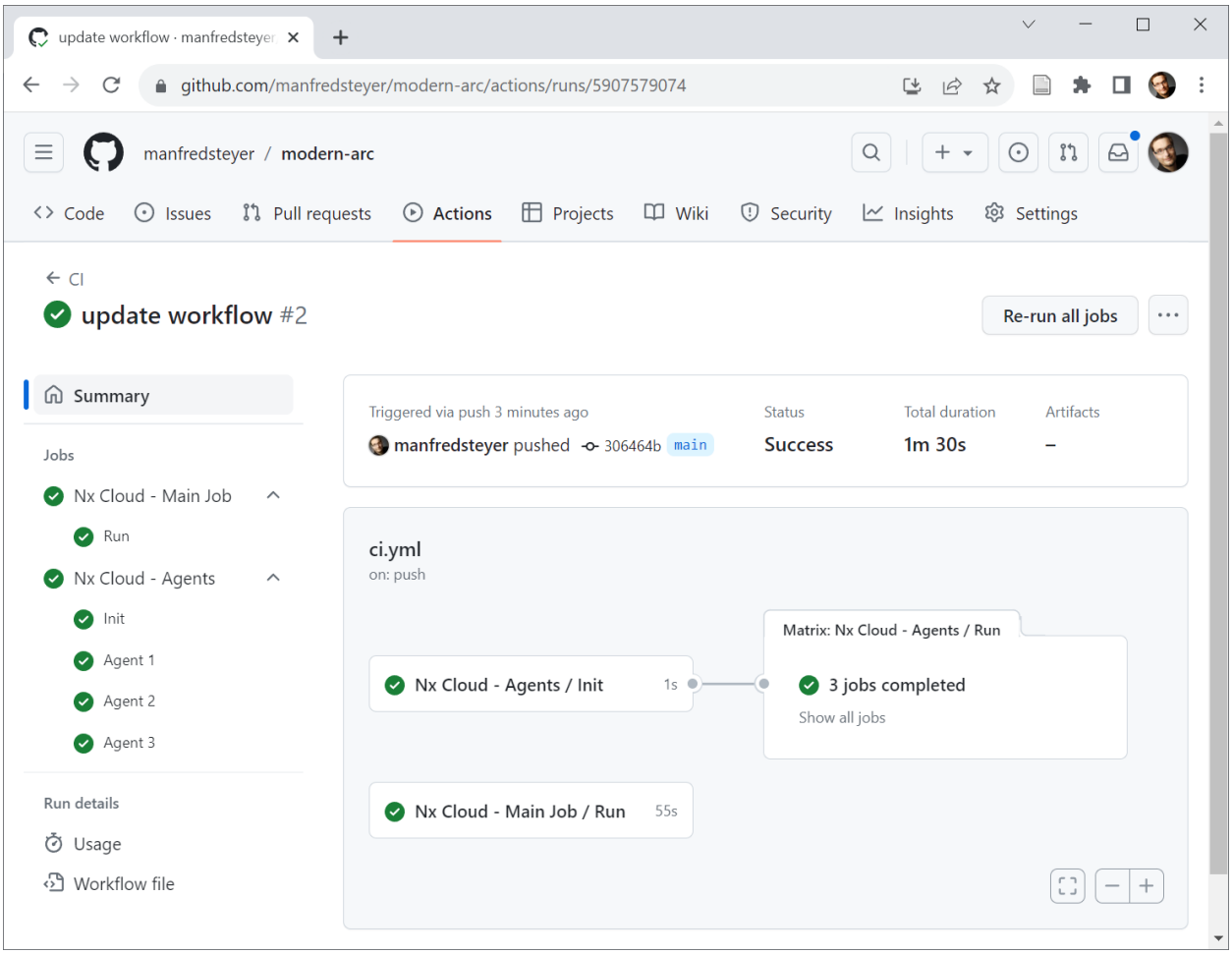
Different nodes in the cloud are used for parallelization: a main node takes over the coordination, and several worker nodes take care of the individual tasks in parallel. Nx can even generate build scripts that start and provide tasks to these nodes. For example, the following statement generates a workflow for GitHub:

```
1 nx generate @nx/workspace:ci-workflow --ci=github
```

This command supports CircleCI (`--ci=circleci`) and Azure (`--ci==azure`) too. If you go with another environment, you can at least use the generated workflows as a starting point. Essentially, these scripts specify the desired number of worker nodes and the number of parallel processes per worker node. The triggered commands are divided into three groups: commands that are executed

sequentially for initialization (`init-commands`), commands that are executed in parallel on the main node (`parallel-commands`) and commands that the workers execute in parallel (`parallel-commands`) on agents.

The scripts are triggered whenever the main branch of the repo is changed - either by a direct `push` or by merging a pull request:



Parallelization with Nx Cloud and GitHub Actions

Conclusion

Nx enables build tasks to be dramatically accelerated. This is made possible, among other things, by incremental builds, in which only the application parts that have actually changed are rebuilt or tested. The Nx Cloud offers further acceleration options with its distributed cache. It also allows the individual builds to be parallelized. Because Nx analyzes the program code and recognizes dependencies between individual applications and libraries, these options often do not require manual configuration.

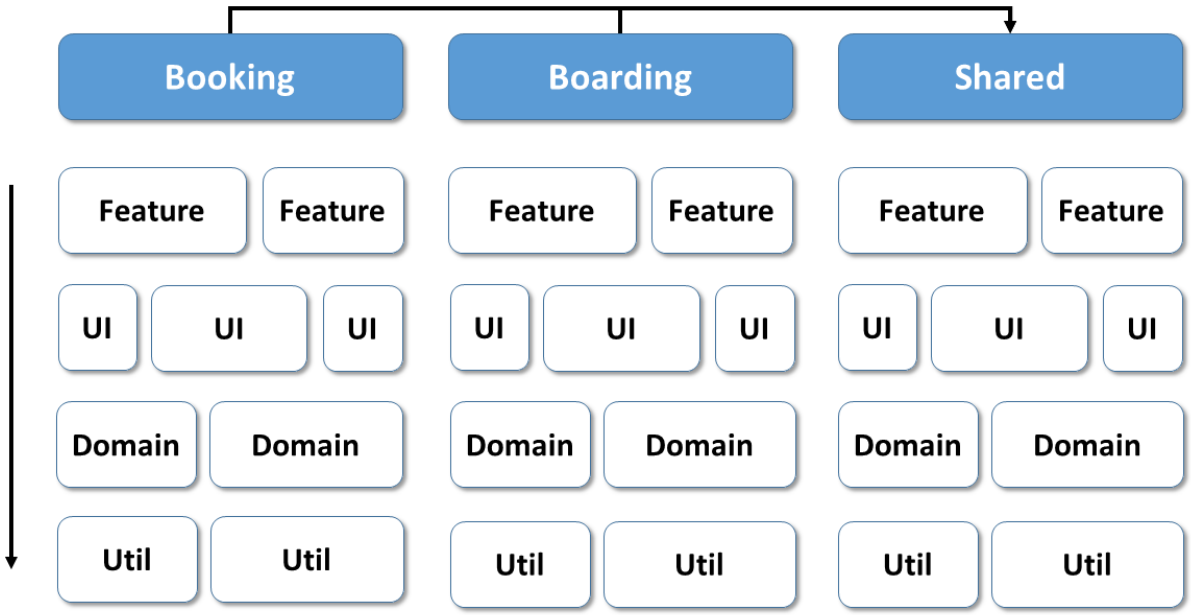
Nx & Sheriff - Friends for Life

Nx provides a lot of features (not only) for Angular teams: A fast CI thanks to the build cache and parallelization, integration into popular tools like Jest, Cypress, Playwright, or Storybook by the push of a button, and linting rules for enforcing module boundaries are just a few examples. Sheriff, on the other hand, focuses on enforcing module boundaries.

At first glance, Sheriff seems to be a small subset of Nx. However, we quite often use both tools together in our customer projects. In this chapter, I explain why and how your architectures can benefit from this combination.

Module Boundaries in Nx

By default, Nx allows to enforce module boundaries like those in our architecture matrix:



Here, a technical layer can only access the below layers, and domains like booking and boarding are not allowed to access each other. However, they can access the shared area (see arrows in the previous image).

Tagging Apps and Libs

To enforce these boundaries with Nx, each cell in our matrix is implemented as a library of its own. Instead of using an overarching `angular.json`, Nx creates a local `project.json` for each app and lib. This file allows you to tag the app or lib in question:

```
{
  [...]
  "tags": ["domain:tickets", "type:domain-logic"]
}
```

Tags are just strings. In the shown case, they reflect the lib's or app's position in the architecture matrix. The prefixes `domain` and `type` help to distinguish the two dimensions (columns with domains and rows with types). This is just to improve readability - for Nx they don't add any meaning.

Defining Module Boundaries

On top of the tags, we can set up module boundaries telling Nx which apps and libs can access which other libs. These boundaries are defined using a linting rule in the `.eslintrc.json` found in the workspace root:

```
"rules": {
  "@nx/enforce-module-boundaries": [
    "error",
    {
      "enforceBuildableLibDependency": true,
      "allow": [],
      "depConstraints": [
        {
          "sourceTag": "type:app",
          "onlyDependOnLibsWithTags": [
            "type:api",
            "type:feature",
            "type:ui",
            "type:domain-logic",
            "type:util"
          ]
        },
        {
          "sourceTag": "type:feature",
          "onlyDependOnLibsWithTags": [
            "type:ui",
```

```
            "type:domain-logic",
            "type:util"
          ]
        },
        {
          "sourceTag": "type:ui",
          "onlyDependOnLibsWithTags": ["type:domain-logic", "type:util"]
        },
        {
          "sourceTag": "type:domain-logic",
          "onlyDependOnLibsWithTags": ["type:util"]
        },


        {
          "sourceTag": "domain:booking",
          "onlyDependOnLibsWithTags": ["domain:booking", "shared"]
        },
        {
          "sourceTag": "domain:boarding",
          "onlyDependOnLibsWithTags": ["domain:boarding", "shared"]
        },
        {
          "sourceTag": "shared",
          "onlyDependOnLibsWithTags": ["shared"]
        },

      ]
    }
  ]
}
```

There is a set of restrictions for each dimension found in the matrix. As we don't add new types of layers and new domains regularly, these linting rules don't come with a lot of maintenance effort. After changing these rules, restart your IDE to ensure it rereads the modified files.

Enforcing Module Boundaries

When your source code breaks the defined rules, your IDE should give you a linting error:

boarding-feature-manage.module.ts

```
A project tagged with "domain:boarding" can only depend on
libs tagged with "domain:boarding", "shared" (nx-enforce-
module-boundaries) tslint(1)
Peek Problem  Quick Fix...
import { BookingDomainModule } from '@ddd-demo/booking/domain';

@NgModule({
  imports: [
    CommonModule,
    BoardingDomainModule,
    BookingDomainModule
  ],
  declarations: [ManageComponent],
  exports: [ManageComponent]
})
export class BoardingFeatureManageModule {}
```

Also, a call to `nx lint` will unveil the same linting errors. This allows your build tasks to check for alignment with the architecture defined. Using git hooks and tools like husky[27], you can also prevent people from checking in source code that breaks the rules.

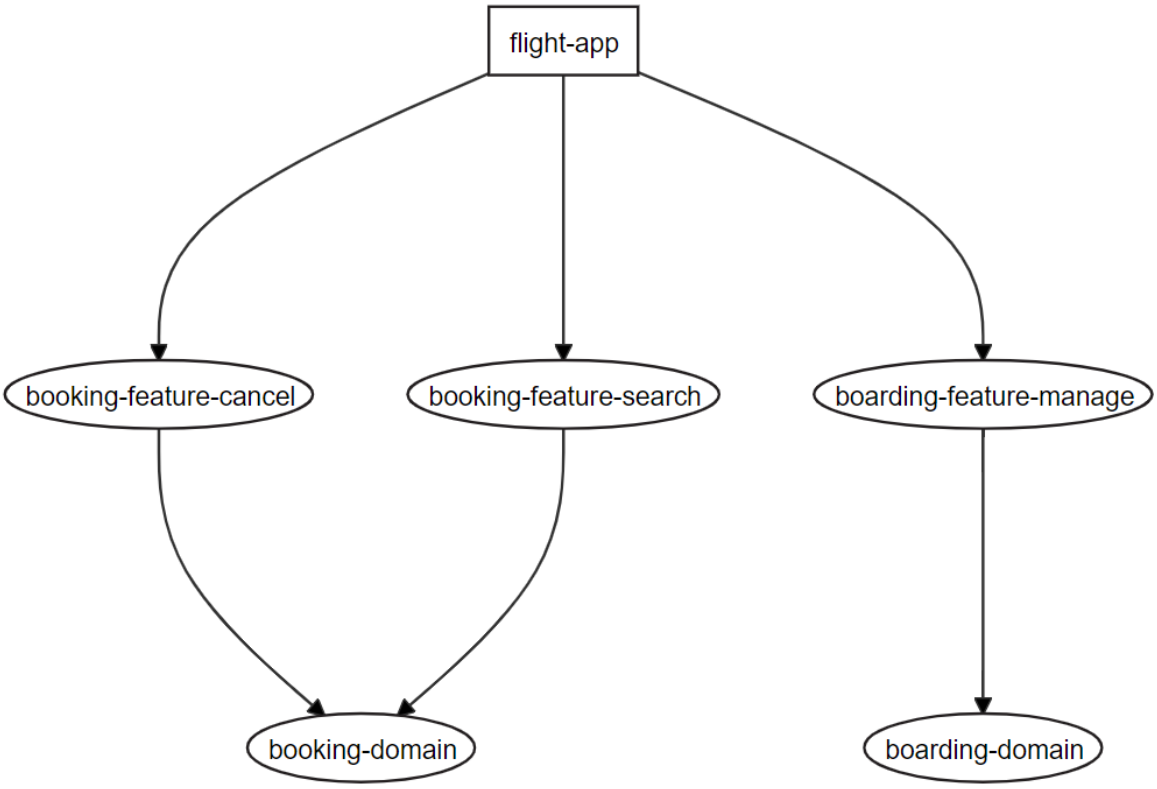
Your Architecture by the Push of a Button

If you want to automate the creation of all these libraries for each matrix cell, defining tags, and linting rules, you might like our DDD Nx plugin[28]. Using this plugin, the following commands are all you need to set up two domains with some features:

[27] https://typicode.github.io/husky/
[28] https://www.npmjs.com/package/@angular-architects/ddd

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

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

If you visualize this architecture with the command `nx graph`, you get the following graph:



Recap: Different Types of Boundaries

So far, we've discussed how to introduce boundaries in Nx. However, if we look closer to Nx, we see that there are two types of boundaries:

- Boundaries for modularization

- Boundaries for incremental CI/CD

Both boundary types align with each other and are implemented as apps and libs.

However, there are situations where having that many apps and libs feels a bit overwhelming, and such a fine-grained incremental CI/CD is not needed. In some cases, the build might already be fast enough or might not benefit much from further apps and libs as the amount of build agents is limited too.

On the other hand, having module boundaries on this granularization level is essential for our architecture. Hence, we need to find a way to decouple these two types of boundaries from each other. For this, we combine Nx with Sheriff[29] introduced in the chapter *Architectures with Sheriff and Standalone Components*:

- Fewer, more coarse-grained libraries define the boundaries for incremental CI/CD
- The usual fine-grained boundaries for modularization are implemented on a per-folder level with Sheriff
- As so often, this is a trade-off situation: We trade in the possibility of a more fine-grained incremental CI/CD for a simplified project structure.

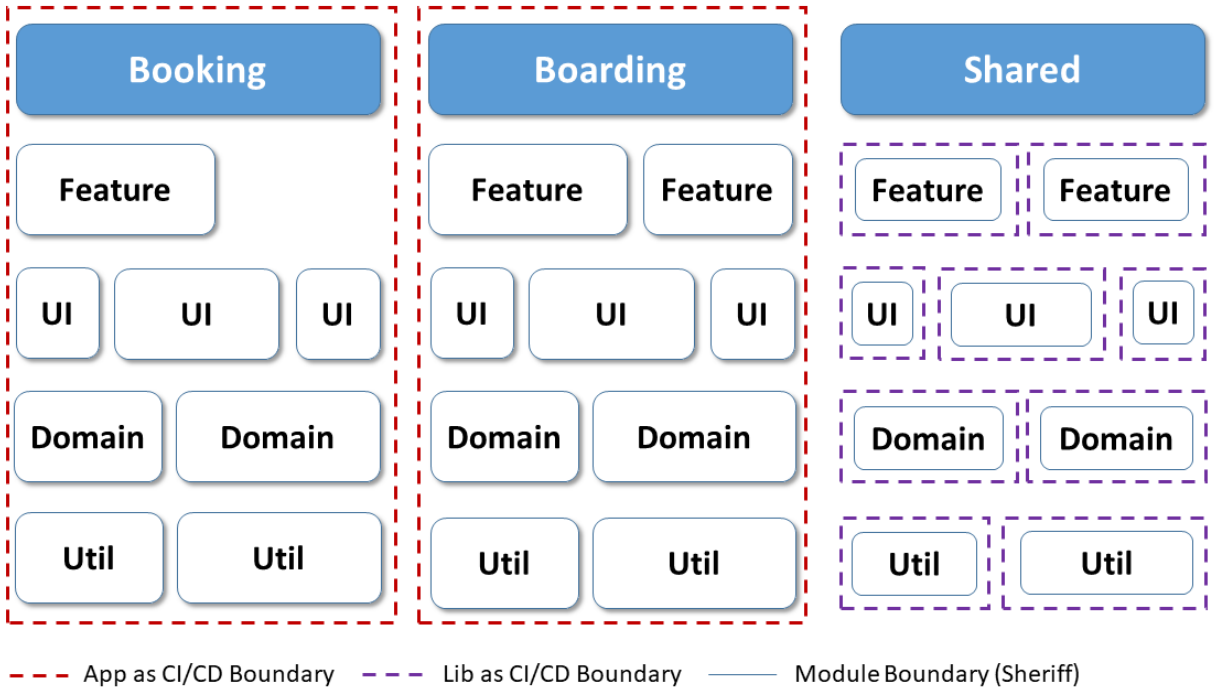
Options with Sheriff

There are several ways to combine Sheriff with Nx. Here, I want to show two options we often use: Having an app per domain and having a lib per domain.

App per Domain

The first approach I want to mention here is creating an application per sub domain:

[29] https://github.com/softarc-consulting/sheriff



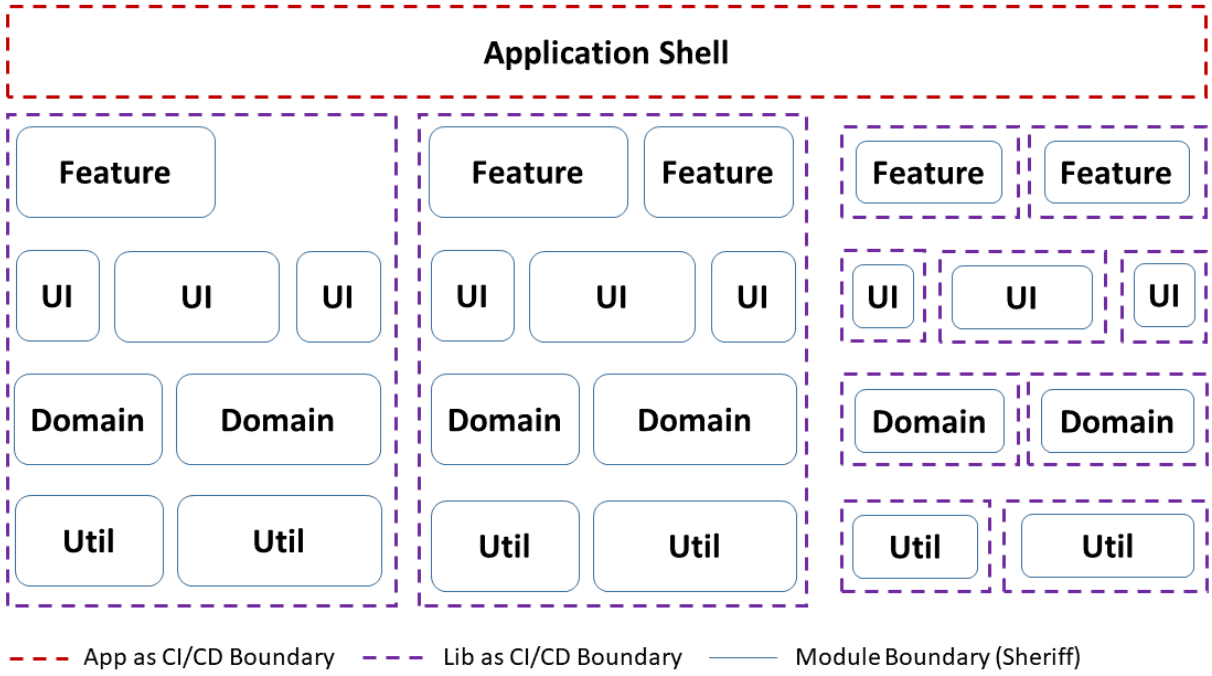
This strategy was already used in the chapter *Architectures with* Sheriff and Standalone Components_.

Shared modules are still implemented as separate libraries. This approach is fitting when we go with several applications that might be integrated using Hyperlinks or technologies also used for Micro Frontends, e.g., Federation. More information about Micro Frontends and Federation can be found in the preceding chapters.

This style gives us a great performance in terms of both incremental builds and incremental testing and linting. Even though Micro Frontend Technologies might be involved, this does not necessarily lead to a Micro Frontend architecture, especially if all applications are deployed together.

Lib per Domain

If your application is intended to be a modular monolith, sometimes referred to as a “Modulith”, you only might have one application consisting of several Domains:



In this case, putting the domains in different libraries helps to speed up incremental testing and linting. However, in this case, the potential for speeding up the build performance is limited as each change leads to a rebuild of the whole application.

Conclusion

Nx is a great build system that uses a build cache and parallelization to speed up your CI tremendously. It comes with integrations into popular tools like Jest, Cypress, Playwright, and Storybook. To enforce our architecture, module boundaries can be configured.

Apps and libs define the boundaries for incremental CI and the module boundaries. Hence, we need to split our software system into several apps and libs.

While having fine-grained module boundaries is preferable, having too many small apps and libraries might be overwhelming and not needed to improve CI performance. This is where Sheriff comes in: It allows defining module boundaries on a per-folder basis, while Nx establishes boundaries on a per-app and per-lib basis.

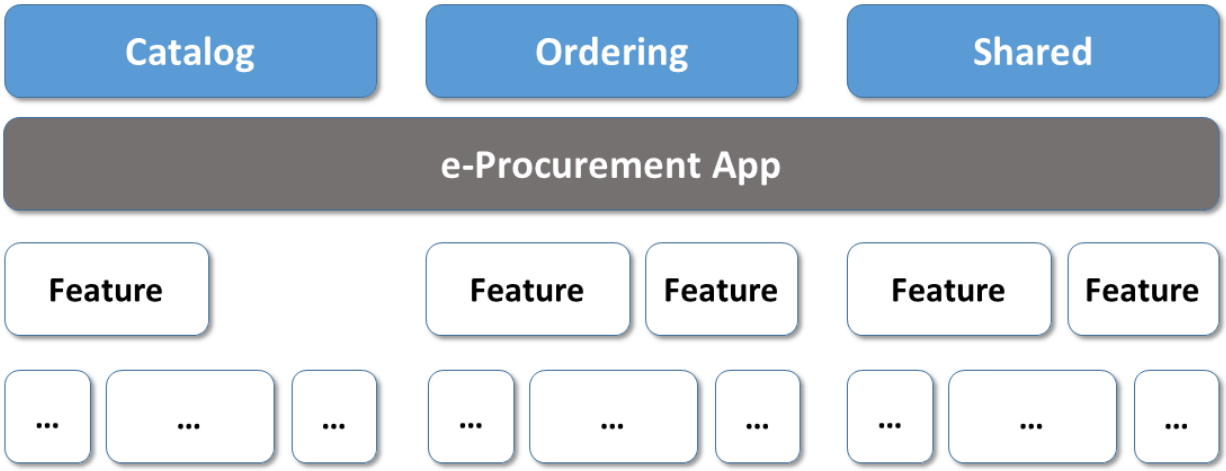
From Domains to Micro Frontends

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:

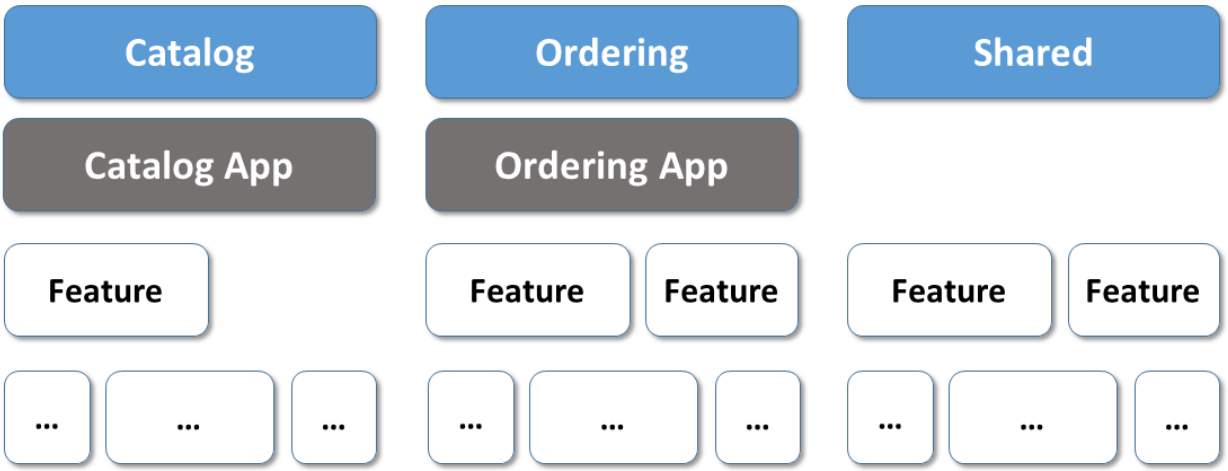


This approach supports a consistent UI and leads to optimized bundles by compiling everything together. A team responsible for a specific sub-domain must coordinate with other sub-domain teams. They have to agree on an overall architecture and the leading framework. Also, they need to define a common 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

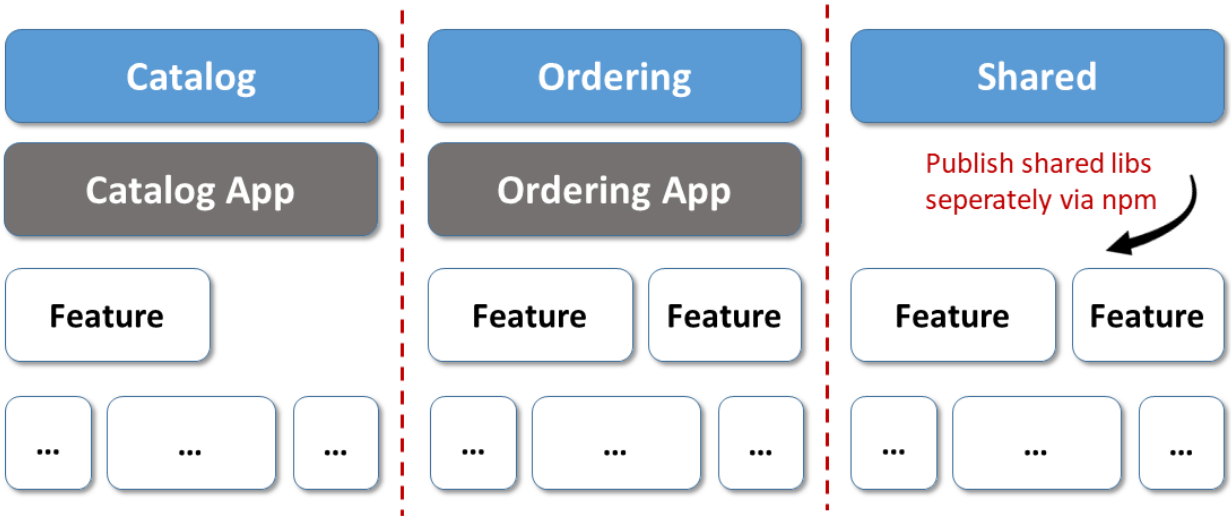
To further decouple your system, you could split it into several smaller applications. If we assume that use cases do not overlap your sub-domains' boundaries, this can lead to more autarkic teams and applications which are separately deployable.



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.

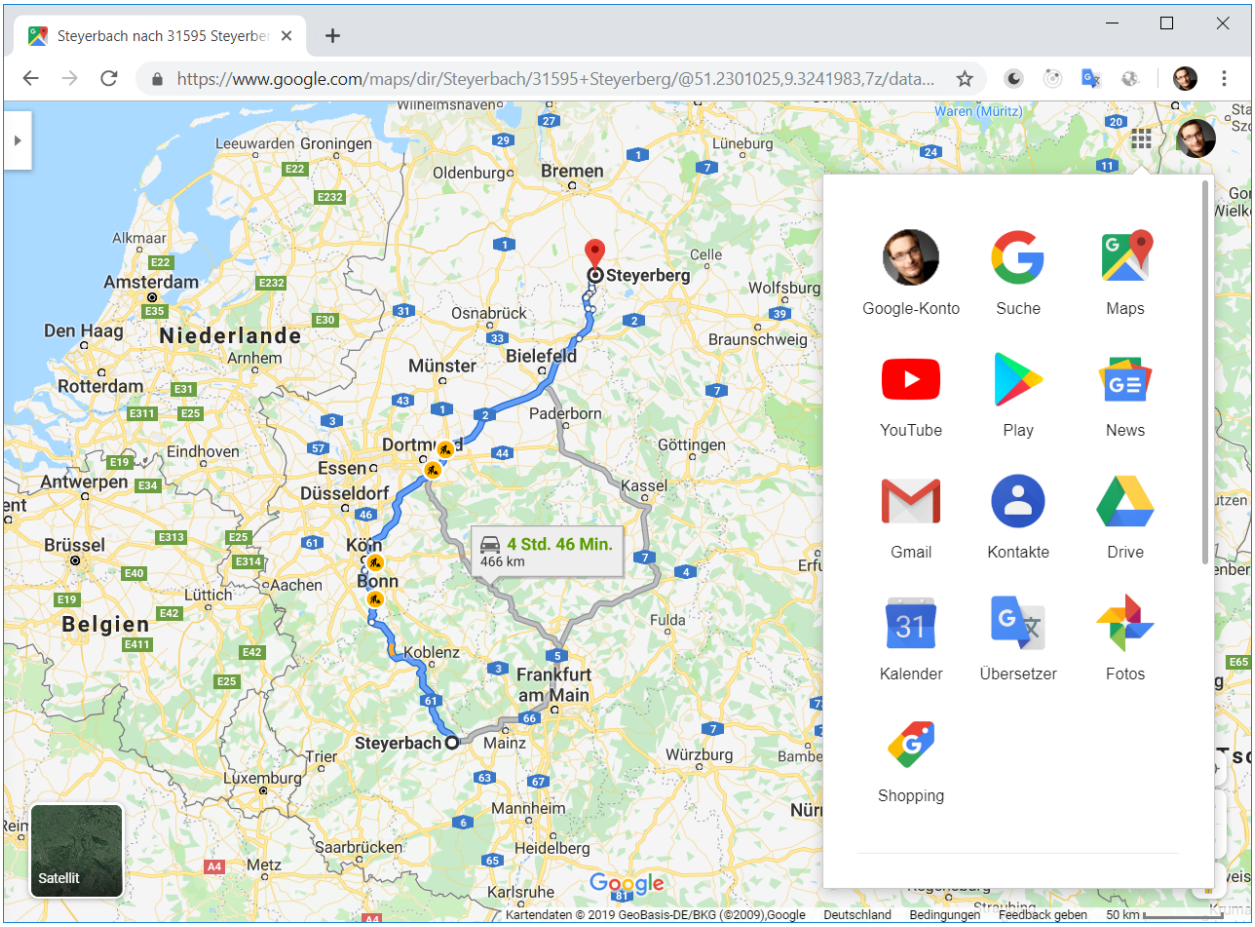
If you seek even more isolation between your sub-domains and the teams responsible for them, you could put each sub-domain into its individual repository:



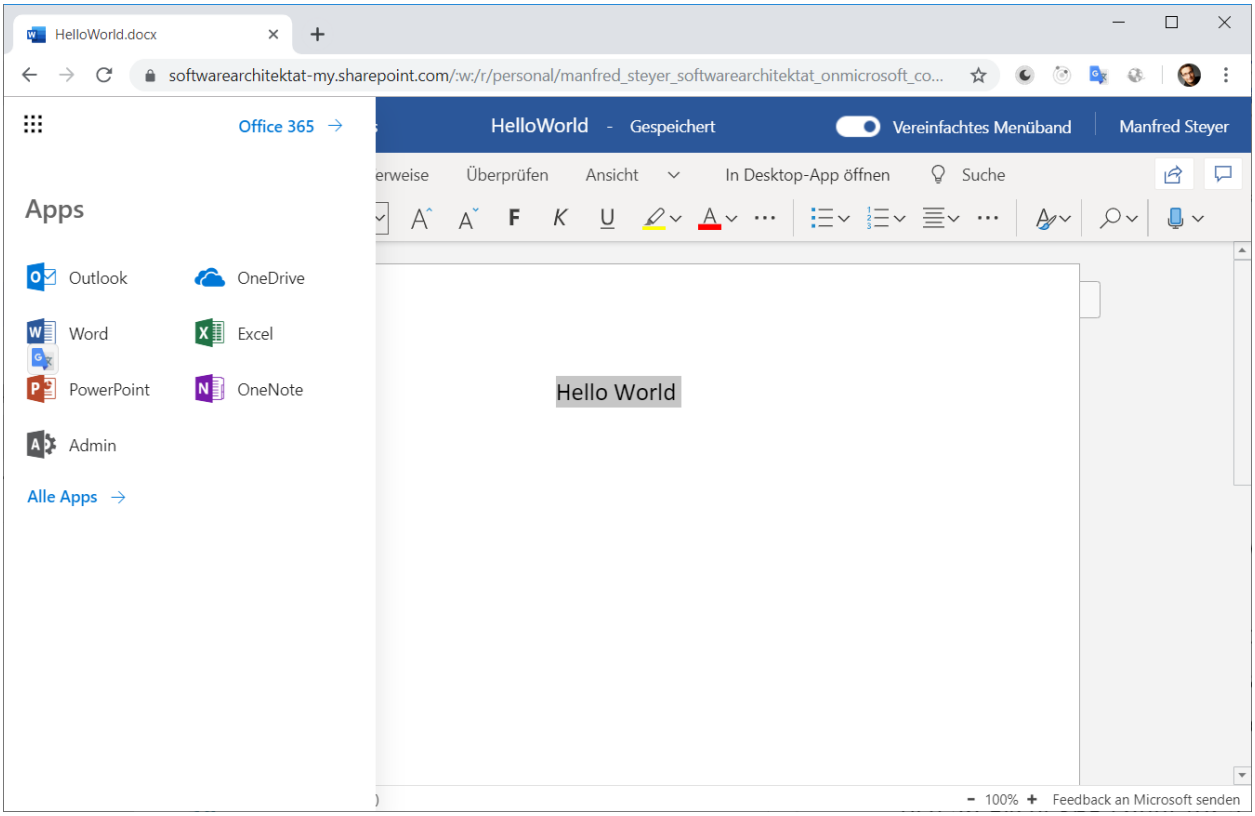
However, this has costs. Now you have to deal with shipping your shared libraries via npm. This comes with some efforts and forces you to version your libraries. You need to make sure that each Micro Frontend uses the right version. Otherwise, you end up with version conflicts.

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:

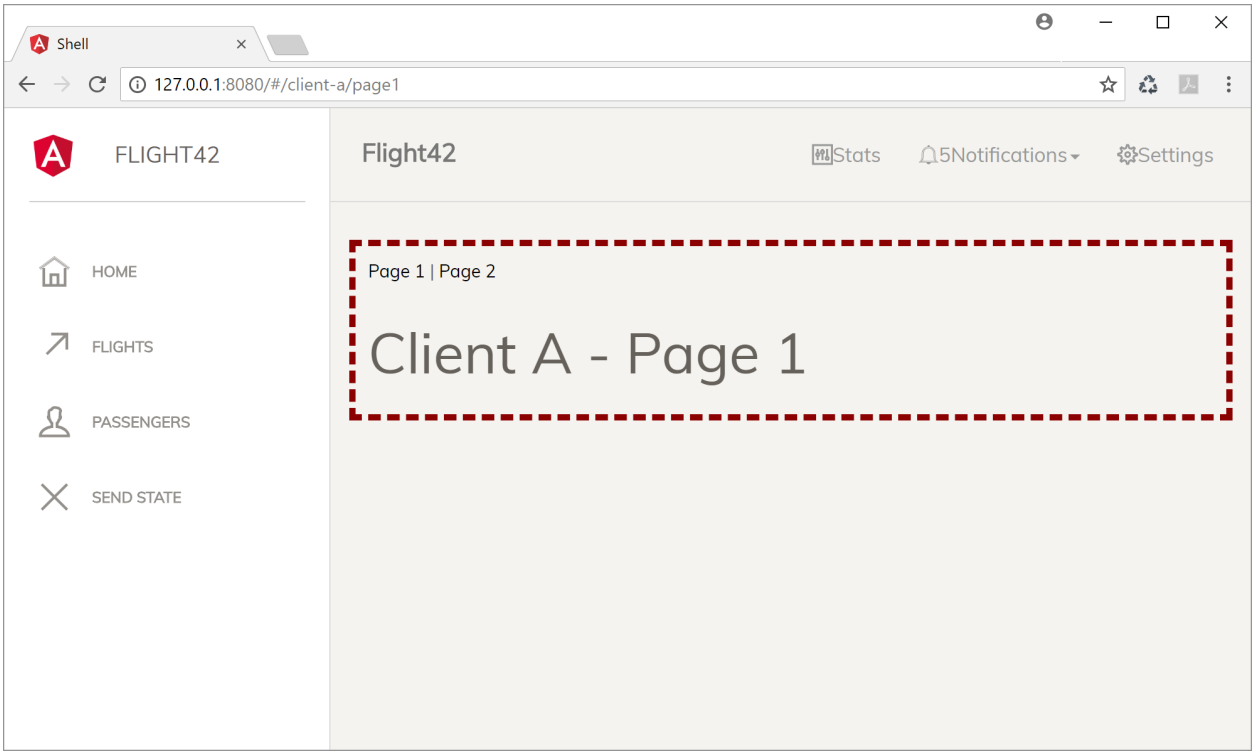
- It is simple
- It uses SPA frameworks as intended
- We get optimised bundles per domain

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 Micro Frontend with the red border into its working area. Technically, it simply loads the Micro Frontend bundles on demand. The shell then creates an element for the Micro Frontend'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 Micro Frontend in one iframe cannot influence or hack another Micro Frontend 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[30]. Even SAP has an iframe-based framework they use for integrating their products. It's called Luigi[31] and you can find it here[32].

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 do some work to get a standard look and feel (we need a universal design system).

The Hero: Module Federation

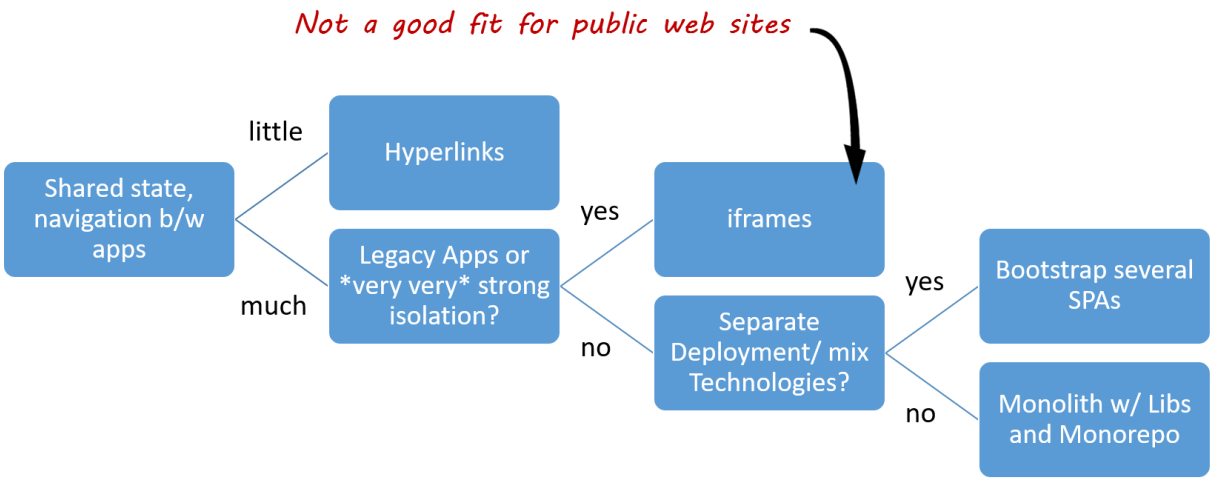
A quite new solution that compensates most of the issues outlined above is Webpack Module Federation. It allows to load code from an separately compiled and deployed application and is very straight forward. IMHO, currently, this is the best way for implementing a shell-based architecture. Hence, the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

[30] https://www.npmjs.com/package/@microfrontend/common
[31] https://github.com/SAP/luigi
[32] 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.

Consequences of Micro Frontends

Each architecture candidate comes with consequences: positive and negative ones. In our survey, conducted in fall 2023, we asked 153 practitioners about the consequences they observed when implementing this architectural style.

The goal was to get answers to the following questions:

- What benefits did practitioners observe, and how do they rate their positive impact?
- What drawbacks did practitioner observe, and how do they rate their negative impact?
- How did practitioners compensate for drawbacks, and how effective have the used counter-measures been?

These questions were broken down to several technical and organisatorical topics. The inquired questions have been subdivided into the following groups:

1. About the Interviewee
2. Project Context
3. Architectural Decisions
4. Perceived Technical Benefits
5. Perceived Organisational Benefits
6. Perceived Technical Drawbacks
7. Perceived Organisational Drawbacks

If you are interested, you can download the survey results here[33].

[33] https://www.angulararchitects.io/wp-content/uploads/2023/12/report.pdf

Conclusion

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

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

The Micro Frontend Revolution: Using Module Federation with Angular

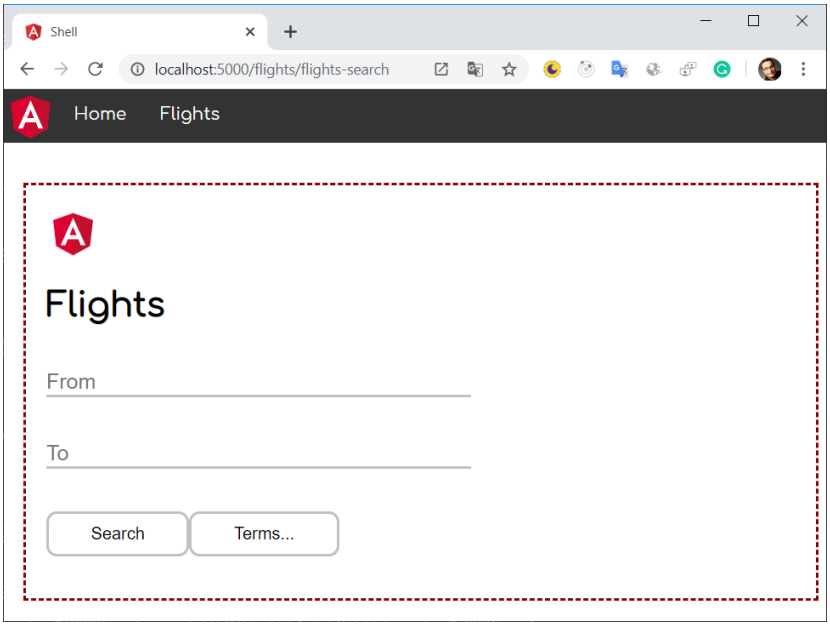
In the past, when implementing Micro Frontends, you had to dig a little into the bag of tricks. One reason is surely that build tools and frameworks did not know this concept. Fortunately, Webpack 5 initiated a change of course here.

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 Micro Frontends 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.

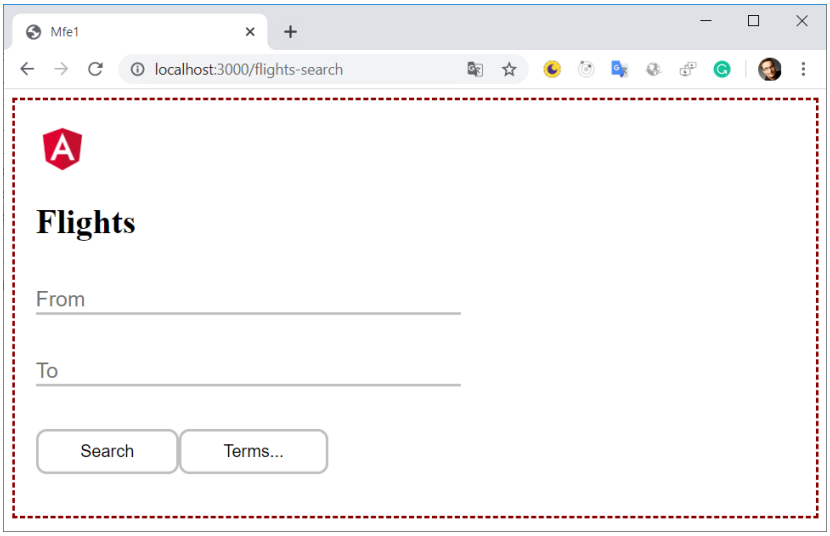
Example

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



Shell

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



Micro Frontend without Shell

Source Code[34] (see branch *static*)

Activating Module Federation for Angular Projects

The case study presented here assumes that both, the shell and the Micro Frontend 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[35] provides such a custom builder. To get started, you can just "ng add" it to your projects:

```
ng add @angular-architects/module-federation
  --project shell --port 4200 --type host

ng add @angular-architects/module-federation
  --project mfe1 --port 4201 --type remote
```

If you use Nx, you should `npm install` the library separately. After that, you can use the `init` schematic:

[34] https://github.com/manfredsteyer/module-federation-plugin-example/tree/static
[35] https://www.npmjs.com/package/@angular-architects/module-federation

```
npm i @angular-architects/module-federation -D

ng g @angular-architects/module-federation:init
  --project shell --port 4200 --type host

ng g @angular-architects/module-federation:init
  --project mfe1 --port 4201 --type remote
```

> The command line argument `--type` was added in version 14.3 and makes sure, only the needed configuration is generated.

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 in the generated `webpack.config.js`:

```
const { shareAll, withModuleFederationPlugin } =
  require('@angular-architects/module-federation/webpack');

module.exports = withModuleFederationPlugin({

  remotes: {
    "mfe1": "http://localhost:4201/remoteEntry.js",
  },

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

});
```

The `remotes` section maps the path `mfe1` to the separately compiled Micro Frontend – 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 Micro Frontend.

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

The property `shared` defines the npm packages to be shared between the shell and the Micro Frontend(s). For this property, The generated configuration uses the helper method `shareAll` that is basically sharing all the dependencies found in your `package.json`. While this helps to quickly get a working setup, it might lead to too much shared dependencies. A later section here addresses this.

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 information[36] about dealing with version mismatches can be found in one of the subsequent chapters.

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

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`.

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 expose it via the remote's webpack configuration:

```
const { shareAll, withModuleFederationPlugin } =
  require('@angular-architects/module-federation/webpack');

module.exports = withModuleFederationPlugin({

  name: 'mfe1',

  exposes: {
    './Module': './projects/mfe1/src/app/flights/flights.module.ts',
  },

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

});
```

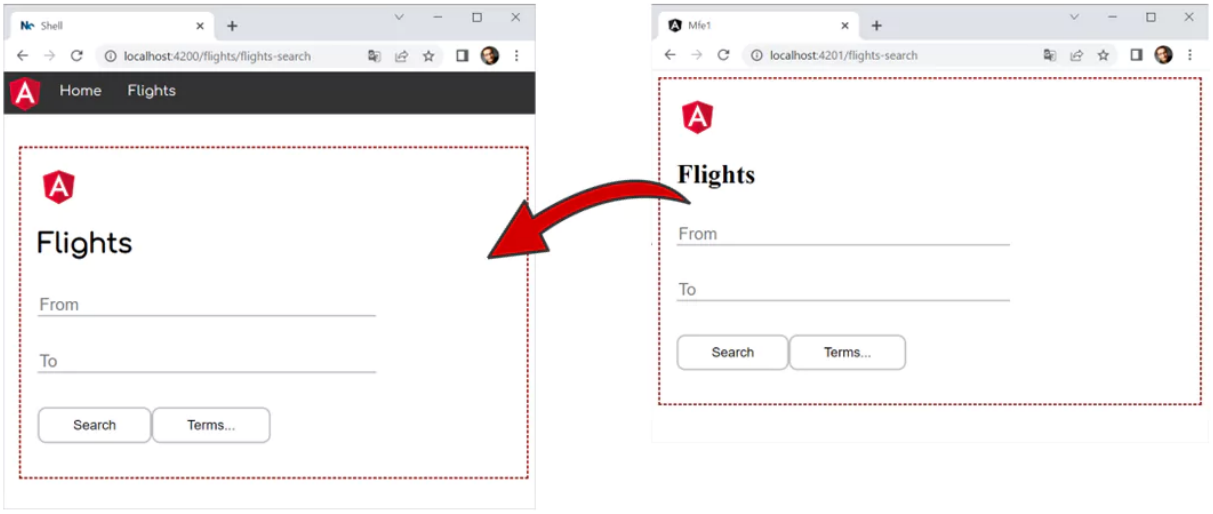
The configuration shown here exposes the `FlightsModule` under the public name `Module`. The section `shared` points to the libraries shared with the shell.

Trying it out

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

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

Then, when clicking on `Flights` in the shell, the Micro Frontend is loaded:



Shell

Hint: You can also use the npm script run:all the plugin installs with its ng-add and init schematics:

```
npm run run:all
```

```
Windows PowerShell

> mf-demo@0.0.0 run:all
> node node_modules/@angular-architects/module-federation/src/server/mf-dev-server.js

DEVSVR | shell 5000
DEVSVR | mfe1 3000
mfe1   | - Generating browser application bundles (phase: setup)...
shell  | - Generating browser application bundles (phase: setup)...
shell  | √ Browser application bundle generation complete.
mfe1   | √ Browser application bundle generation complete.
shell  |
shell  | Initial Chunk Files                                                                                                                  | Names         |   Raw Size
shell  | polyfills.js                                                                                                                         | polyfills     |  383.09 kB |
shell  | styles.css, styles.js                                                                                                                | styles        |  243.44 kB |
shell  | vendor.js                                                                                                                            | vendor        |  226.36 kB |
shell  | main.js                                                                                                                              | main          |   40.17 kB |
shell  |
shell  | | Initial Total |  893.06 kB
shell  |
shell  | Lazy Chunk Files                                                                                                                     | Names         |   Raw Size
shell  | node_modules_angular_core_fesm2020_core_mjs.js                                                                                       | -             |    1.12 MB |
shell  | node_modules_angular_core_fesm2020_testing_mjs-_d65d0.js                                                                             | -             | 1023.40 kB |
shell  | node_modules_angular_core_fesm2020_testing_mjs-_d65d1.js                                                                             | -             | 1023.40 kB |
shell  | node_modules_angular_compiler_fesm2020_compiler_mjs.js                                                                               | -             |  829.10 kB |
shell  | node_modules_rxjs__esm2015_operators_index_js.js                                                                                     | -             |  353.26 kB |
shell  | node_modules_angular_forms_fesm2020_forms_mjs-_2e340.js                                                                              | -             |  286.13 kB |
shell  | node_modules_angular_forms_fesm2020_forms_mjs-_2e341.js                                                                              | -             |  286.13 kB |
shell  | node_modules_angular_router_fesm2020_router_mjs-_6f000.js                                                                            | -             |  256.33 kB |
shell  | node_modules_angular_router_fesm2020_router_mjs-_6f001.js
shell  | | -             |  256.33 kB |
shell  | node_modules_angular_common_fesm2020_common_mjs-_ec490.js                                                                            | -             |  231.89 kB |
shell  | node_modules_angular_common_fesm2020_common_mjs-_ec491.js                                                                            | -             |  231.89 kB |
shell  | node_modules_rxjs__esm2015_index_js.js                                                                                               | -             |  195.46 kB |
shell  | node_modules_angular_animations_fesm2020_browser_mjs-_54670.js                                                                       | -             |  172.77 kB |
shell  | node_modules_angular_animations_fesm2020_browser_mjs-_54671.js                                                                       | -             |  172.77 kB |
shell  | node_modules_angular_platform-browser_fesm2020_platform-browser_mjs-_18080.js                                                        | -             |   89.53 kB |
shell  | node_modules_angular_platform-browser_fesm2020_platform-browser_mjs-_18081.js                                                        | -             |   89.53 kB |
shell  | node_modules_angular_common_fesm2020_http_mjs-_68760.js                                                                              | -             |   86.80 kB |
shell  | node_modules_angular_common_fesm2020_http_mjs-_68761.js                                                                              | -             |   86.80 kB |
shell  | node_modules_angular_animations_fesm2020_animations_mjs.js                                                                           | -             |   40.49 kB |
shell  | node_modules_angular_animations_fesm2020_browser_testing_mjs.js                                                                      | -             |   35.51 kB |
shell  | node_modules_angular-architects_module-federation-tools_fesm2015_angular-architects-module-fe-3c905c1.js                             | -             |   32.73 kB |
shell  | node_modules_angular_platform-browser_fesm2020_animations_mjs.js                                                                     | -             |   22.64 kB |
shell  | node_modules_angular_common_fesm2020_testing_mjs-_e7c50.js                                                                           | -             |   18.92 kB |
```

run:all script

To just start a few applications, add their names as command line arguments:

```
npm run run:all shell mfe1
```

A Further Detail

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

It just looks like this:

```
import('./bootstrap')
        .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 glance, 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: Sharing Dependencies

As mentioned above, the usage of `shareAll` allows for a quick first setup that "just works". However, it might lead to too much shared bundles. As shared dependencies cannot be tree shaken and by default end up in separate bundles that need to be loaded, you might want to optimize this behavior by switching over from `shareAll` to the `share` helper:

```
// Import share instead of shareAll:
const { share, withModuleFederationPlugin } =
  require('@angular-architects/module-federation/webpack');

module.exports = withModuleFederationPlugin({

    // Explicitly share packages:
    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'
        },
    }),

});
```

More on This

Learn more about this and further architecture topics regarding Angular and huge enterprise as well as industrial solution in our advanced Online Workshop[37]:

[37] https://www.angulararchitects.io/schulungen/advanced-angular-enterprise-anwendungen-und-architektur/

Advanced Angular Workshop

Save your ticket[38] for one of our **online or on-site** workshops now or request a company workshop[39] (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[40] and/ or follow the book's author on Twitter[41].

Conclusion and Evaluation

The implementation of Micro Frontends 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 Micro Frontends 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 Micro Frontend 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.

[38] https://www.angulararchitects.io/en/angular-workshops/
[39] https://www.angulararchitects.io/en/contact/
[40] https://www.angulararchitects.io/en/subscribe/
[41] https://twitter.com/ManfredSteyer

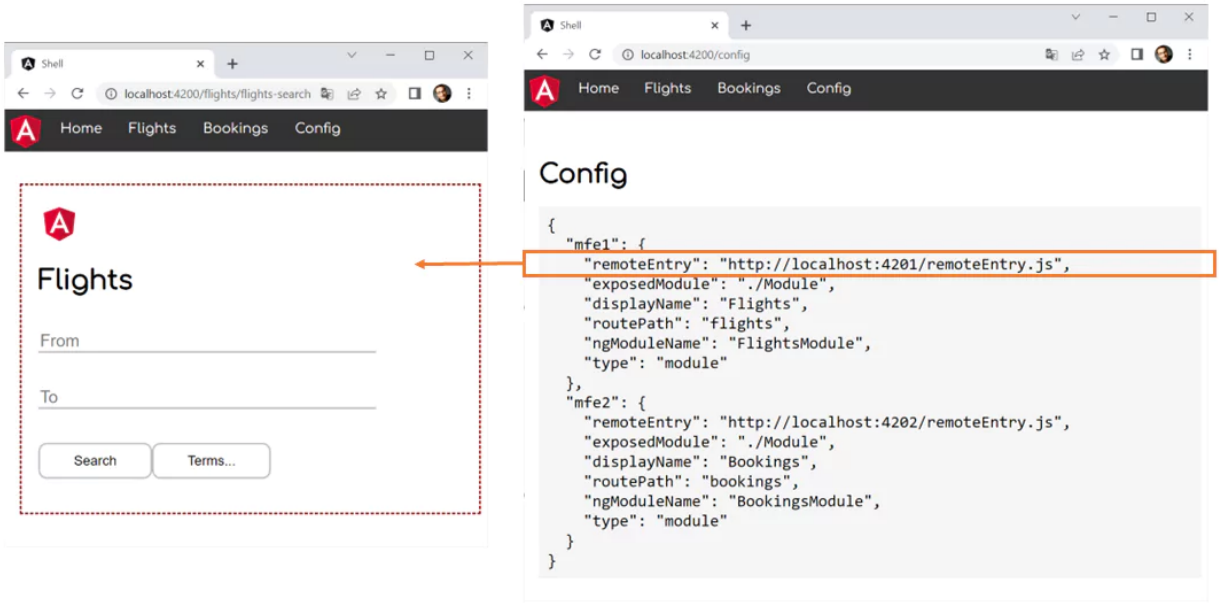
One also has to deal with possible version conflicts. For example, it is likely that components that were compiled with completely different Angular versions will not work together at runtime. Such cases must be avoided with conventions or at least recognized as early as possible with integration tests.

Dynamic Module Federation

In the previous chapter, I've shown how to use webpack Module Federation for loading separately compiled Micro Frontends into a shell. As the shell's webpack configuration describes the Micro Frontends, 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 Micro Frontend upfront. Instead, this information is provided at runtime via a configuration file. While this file is a static JSON file in the examples shown here, it's content could also come from a Web API.

The following image displays the idea described here:



The shell loads a Micro Frontend it is informed about on runtime

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

Source Code (simple version, see branch: simple)[42]

Source Code (full version)[43]

[42] https://github.com/manfredsteyer/module-federation-with-angular-dynamic/tree/simple
[43] https://github.com/manfredsteyer/module-federation-with-angular-dynamic.git

A Simple Dynamic Solution

Let's start with a simple approach. For this, we assume that we know the Micro Frontends upfront and just want to switch out their URLs at runtime, e. g. with regards to the current environment. A more advanced approach, where we don't even need to know the number of Micro Frontends upfront is presented afterwards.

Adding Module Federation

The demo project used contains a shell and two Micro Frontends called `mfe1` and `mfe2`. As in the previous chapter, we add and initialize the Module Federation plugin for the Micro Frontends:

```
npm i -g @angular-architects/module-federation -D

ng g @angular-architects/module-federation
  --project mfe1 --port 4201 --type remote

ng g @angular-architects/module-federation
  --project mfe2 --port 4202 --type remote
```

Generating a Manifest

Beginning with the plugin's version 14.3, we can generate a **dynamic host** that takes the key data about the Micro Frontend from a JSON file – called the Micro Frontend Manifest – at runtime:

```
ng g @angular-architects/module-federation
  --project shell --port 4200 --type dynamic-host
```

This generates:

- a webpack configuration
- the manifest and
- some code in the `main.ts` loading the manifest.

The manifest can be found here: `projects/shell/src/assets/mf.manifest.json`. This is what it looks like:

```
{
        "mfe1": "http://localhost:4201/remoteEntry.js",
        "mfe2": "http://localhost:4202/remoteEntry.js"
}
```

After generating the manifest, make sure the ports match.

Loading the Manifest

The generated `main.ts` file loads the manifest:

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

loadManifest("/assets/mf.manifest.json")
  .catch(err => console.error(err))
  .then(_ => import('./bootstrap'))
  .catch(err => console.error(err));
```

By default, `loadManifest` not just loads the manifest but also the remote entries the manifest points to. Hence, Module Federation gets all the required metadata for fetching the Micro Frontends on demand.

Loading the Micro Frontends

To load the Micro Frontends described by the manifest, we go with the following routes:

```
import { Routes } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { loadRemoteModule } from '@angular-architects/module-federation';

export const APP_ROUTES: Routes = [
    {
      path: '',
      component: HomeComponent,
      pathMatch: 'full'
    },
    {
      path: 'flights',
      loadChildren: () => loadRemoteModule({
          type: 'manifest',
          remoteName: 'mfe1',
          exposedModule: './Module'
```

```
        })
        .then(m => m.FlightsModule)
    },
    {
      path: 'bookings',
      loadChildren: () => loadRemoteModule({
          type: 'manifest',
          remoteName: 'mfe2',
          exposedModule: './Module'
        })
        .then(m => m.BookingsModule)
    },
];
```

The option `type: 'manifest'` makes `loadRemoteModule` to look up the key data needed in the loaded manifest. The property `remoteName` points to the key that was used in the manifest.

Configuring the Micro Frontends

We expect both Micro Frontends to provide an NgModule with sub routes via `'./Module'`. The NgModules are exposed via the `webpack.config.js` in the Micro Frontends:

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

const { shareAll, withModuleFederationPlugin } =
  require('@angular-architects/module-federation/webpack');

module.exports = withModuleFederationPlugin({

  name: 'mfe1',

  exposes: {
    // Adjusted line:
    './Module': './projects/mfe1/src/app/flights/flights.module.ts'
  },

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

```
  },

});
```

```
// projects/mfe2/webpack.config.js

const { shareAll, withModuleFederationPlugin } =
  require('@angular-architects/module-federation/webpack');

module.exports = withModuleFederationPlugin({

  name: 'mfe2',

  exposes: {
    // Adjusted line:
    './Module': './projects/mfe2/src/app/bookings/bookings.module.ts'
  },

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

});
```

Trying it Out

For each route loading a Micro Frontend, the shell's `AppComponent` contains a `routerLink`:

```
<!-- projects/shell/src/app/app.component.html -->
<ul>
    <li><img src="../assets/angular.png" width="50"></li>
    <li><a routerLink="/">Home</a></li>
    <li><a routerLink="/flights">Flights</a></li>
    <li><a routerLink="/bookings">Bookings</a></li>
</ul>

<router-outlet></router-outlet>
```

That's it. Just start all the three projects (e. g. by using `npm run run:all`). The main difference to the result in the previous chapter is that now the shell informs itself about the Micro Frontends at runtime. If you want to point the shell to different Micro Frontends, just adjust the manifest.

Going "Dynamic Dynamic"

The solution we have so far is suitable in many situations: The usage of the manifest allows to adjust it to different environments without rebuilding the application. Also, if we switch out the manifest for a dynamic REST service, we could implement strategies like A/B testing.

However, in some situation you might not even know about the number of Micro Frontends upfront. This is what we discuss here.

Adding Custom Metadata to The Manifest

For dynamically setting up the routes, we need some additional metadata. For this, you might want to extend the manifest:

```
{
        "mfe1": {
                "remoteEntry": "http://localhost:4201/remoteEntry.js",

                "exposedModule": "./Module",
                "displayName": "Flights",
                "routePath": "flights",
                "ngModuleName": "FlightsModule"
        },
        "mfe2": {
                "remoteEntry": "http://localhost:4202/remoteEntry.js",

                "exposedModule": "./Module",
                "displayName": "Bookings",
```

```
15                  "routePath": "bookings",
16                  "ngModuleName": "BookingsModule"
17          }
18  }
```

Besides `remoteEntry`, all other properties are **custom**.

Types for Custom Configuration

To represent our extended configuration, we need some types in the shell's code:

```
1  // projects/shell/src/app/utils/config.ts
2
3  import {
4    Manifest,
5    RemoteConfig
6  } from "@angular-architects/module-federation";
7
8  export type CustomRemoteConfig = RemoteConfig & {
9    exposedModule: string;
10   displayName: string;
11   routePath: string;
12   ngModuleName: string;
13 };
14
15 export type CustomManifest = Manifest<CustomRemoteConfig>;
```

The `CustomRemoteConfig` type represents the entries in the manifest and the `CustomManifest` type the whole manifest.

Dynamically Creating Routes

Now, we need an utility function iterating through the whole manifest and creating a route for each Micro Frontend described there:

```
// projects/shell/src/app/utils/routes.ts

import { loadRemoteModule } from '@angular-architects/module-federation';
import { Routes } from '@angular/router';
import { APP_ROUTES } from '../app.routes';
import { CustomManifest } from './config';

export function buildRoutes(options: CustomManifest): Routes {

    const lazyRoutes: Routes = Object.keys(options).map(key => {
        const entry = options[key];
        return {
            path: entry.routePath,
            loadChildren: () =>
                loadRemoteModule({
                    type: 'manifest',
                    remoteName: key,
                    exposedModule: entry.exposedModule
                })
                .then(m => m[entry.ngModuleName])
        }
    });

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

This gives us the same structure, we directly configured above.

The shell's `AppComponent` glues everything together:

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent implements OnInit  {

  remotes: CustomRemoteConfig[] = [];

  constructor(
    private router: Router) {
  }

  async ngOnInit(): Promise<void> {
```

```
    const manifest = getManifest<CustomManifest>();

    // Hint: Move this to an APP_INITIALIZER
    //  to avoid issues with deep linking
    const routes = buildRoutes(manifest);
    this.router.resetConfig(routes);

    this.remotes = Object.values(manifest);
  }
}
```

The `ngOnInit` method retrieves the loaded manifest (it's still loaded in the `main.ts` as shown above) and passes it to `buildRoutes`. The retrieved dynamic routes are passed to the router. Also, the values of the key/value pairs in the manifest, are put into the `remotes` field. It's used in the template to dynamically create the menu items:

```
<!-- projects/shell/src/app/app.component.html -->

<ul>
    <li><img src="../assets/angular.png" width="50"></li>
    <li><a routerLink="/">Home</a></li>

    <!-- Dynamically create menu items for all Micro Frontends -->
    <li *ngFor="let remote of remotes">
      <a [routerLink]="remote.routePath">{{remote.displayName}}</a>
    </li>

    <li><a routerLink="/config">Config</a></li>
</ul>

<router-outlet></router-outlet>
```

Trying it Out

Now, let's try out this "dynamic dynamic" solution by starting the shell and the Micro Frontends (e. g. with `npm run run:all`).

Some More Details

So far, we used the high-level functions provided by the plugin. However, for cases you need more control, there are also some low-level alternatives:

- `loadManifest(...)`: The above used `loadManifest` function provides a second parameter called `skipRemoteEntries`. Set it to `true` to prevent loading the entry points. In this case, only the manifest is loaded:

```
loadManifest("/assets/mf.manifest.json", true)
    .catch(...)
    .then(...)
    .catch(...)
```

- `setManifest(...)`: This function allows to directly set the manifest. It comes in handy if you load the data from somewhere else.
- `loadRemoteEntry(...)`: This function allows to directly load the remote entry point. It's useful if you don't use the manifest:

```
Promise.all([
    loadRemoteEntry({
      type: 'module',
      remoteEntry: 'http://localhost:4201/remoteEntry.js'
    }),
    loadRemoteEntry({
      type: 'module',
      remoteEntry: 'http://localhost:4202/remoteEntry.js'
    })
])
.catch(err => console.error(err))
.then(_ => import('./bootstrap'))
.catch(err => console.error(err));
```

- `loadRemoteModule(...)`: Also, if you don't want to use the manifest, you can directly load a Micro Frontend with `loadRemoteModule`:

```
{
    path: 'flights',
    loadChildren: () =>
        loadRemoteModule({
            type: 'module',
            remoteEntry: 'http://localhost:4201/remoteEntry.js',
            exposedModule: './Module',
        }).then((m) => m.FlightsModule),
},
```

In general I think most people will use the manifest in the future. Even if one doesn't want to load it from a JSON file with `loadManifest`, one can define it via `setManifest`.

The property `type: 'module'` defines that you want to load a "real" EcmaScript module instead of "just" a JavaScript file. This is needed since Angular CLI 13. If you load stuff not built by CLI 13 or higher, you very likely have to set this property to `script`. This can also happen via the manifest:

```
{
        "non-cli-13-stuff": {
        "type": "script",
        "remoteEntry": "http://localhost:4201/remoteEntry.js"
    }
}
```

> If an entry in the manifest does not contain a `type` property, the plugin assumes the value `module`.

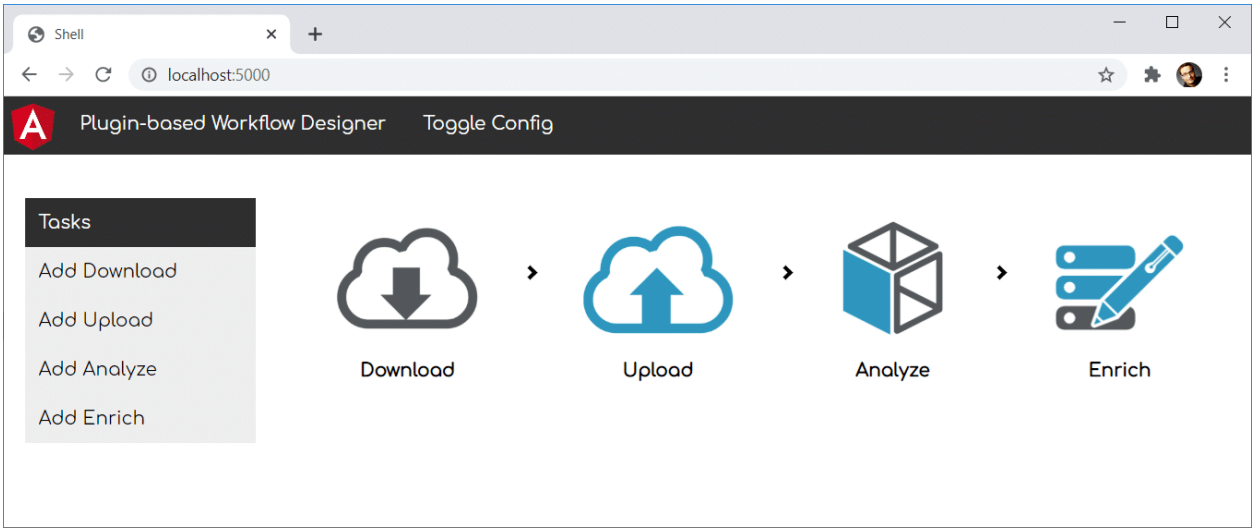
Conclusion

Dynamic Module Federation provides more flexibility as it allows loading Micro Frontends 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 chapter 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:4201 and http://localhost:4202), and the workflow designer is served from an origin of its own (http://localhost:4200).

Source Code[44]

Thanks to Zack Jackson[45] and Jack Herrington[46], who helped me to understand the rater new API for Dynamic Module Federation.

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 of this book:

[44] https://github.com/manfredsteyer/module-federation-with-angular-dynamic-workflow-designer
[45] https://twitter.com/ScriptedAlchemy
[46] https://twitter.com/jherr

```
const { shareAll, withModuleFederationPlugin } =
  require('@angular-architects/module-federation/webpack');

module.exports = withModuleFederationPlugin({

  name: 'mfe1',

  exposes: {
    './Download': './projects/mfe1/src/app/download.component.ts',
    './Upload': './projects/mfe1/src/app/upload.component.ts'
  },

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

});
```

One difference to the configurations shown in the previous chapter is that here we are directly exposing standalone components. Each component represents a task that can be put into the workflow.

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.

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:4201/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;
};
```

> An alternative to this is extending the Module Federation Manifest as shown in the previous chapter.

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:4201/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({
    standalone: true,
    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();

        const Component = await loadRemoteModule(this.options)
            .then(m => m[this.options.componentName]);

        this.viewContainer.createComponent(Component);
    }
}
```

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

```
// Before Angular 13, we needed to retrieve a ComponentFactory
//
// export class PluginProxyComponent implements OnChanges {
//     @ViewChild('placeHolder', { read: ViewContainerRef, static: true })
//     viewContainer: ViewContainerRef;

//     constructor(
//       private injector: Injector,
//       private cfr: ComponentFactoryResolver) { }

//     @Input() options: PluginOptions;

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

//         const component = await loadRemoteModule(this.options)
//             .then(m => m[this.options.componentName]);

//         const factory = this.cfr.resolveComponentFactory(component);

//         this.viewContainer.createComponent(factory, null, this.injector);
//     }
// }
```

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) {
  }
```

```
11
12    async ngOnInit(): Promise<void> {
13      this.plugins = await this.lookupService.lookup();
14    }
15
16    add(plugin: PluginOptions): void {
17      this.workflow.push(plugin);
18    }
19
20    toggle(): void {
21      this.showConfig = !this.showConfig;
22    }
23  }
```

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:

```
1  <ng-container *ngFor="let p of workflow; let last = last">
2      <plugin-proxy [options]="p"></plugin-proxy>
3      <i *ngIf="!last" class="arrow right" style=""></i>
4  </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:

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

While it sounds like a contradiction, the combination of Micro Frontends and Monorepos can actually be quite tempting: No **version conflicts** by design, easy code sharing and **optimized bundles** are some of the benefits you get. Also, you can still **deploy** Micro Frontends **separately** and **isolate** them from each other.

This chapter **compares the consequences** of using **several repositories** ("Micro Frontends by the book") and one sole **monorepo**. After that, it shows with an example, how to use Module Federation in an Nx monorepo.

If you want to have a look at the source code[47] used here, you can check out this repository[48].

> Big thanks to the awesome Tobias Koppers[49] who gave me valuable insights into this topic and to the one and only Dmitriy Shekhovtsov[50] who helped me using the Angular CLI/webpack 5 integration for this.

Multiple Repos vs. Monorepos

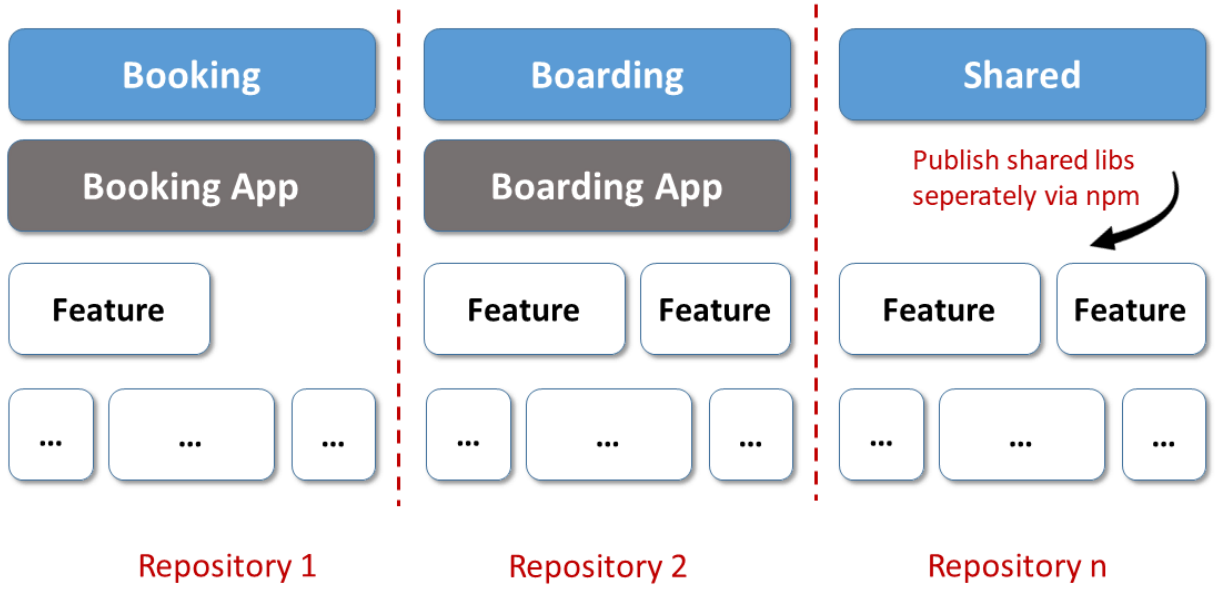
I know, the discussion about using multiple repos vs. monorepos can be quite emotional. Different people made different experiences with both approaches. However, I can tell you: I've seen both working in huge real-world projects. Still, both comes **with different consequences**, I'm going to discuss in the following two section.

At the end of the day, you need to **evaluate** these consequences against your very project situation and **goals**. This is what software architecture is about.

Multiple Repositories: Micro Frontends by the Book

A traditional approach uses a separate repository per Micro Frontend:

[47] https://github.com/manfredsteyer/nx-module-federation-demo
[48] https://github.com/manfredsteyer/nx-module-federation-demo
[49] https://twitter.com/wSokra
[50] https://twitter.com/valorkin



One Repository per Micro Frontend

This is also a quite usual for Micro Services and it provides the following **advantages**:

- Micro Frontends – and hence the individual business domains – are isolated from each other. As there are no dependencies between them, different teams can evolve them separately.
- Each team can concentrate on their Micro Frontend. They only need to focus on their very own repository.
- Each team has the maximum amount of freedom in their repository. They can go with their very own architectural decisions, tech stacks, and build processes. Also, they decide by themselves when to update to newer versions.
- Each Micro Frontend can be separately deployed.

As this best fits the original ideas of Micro Frontends, I call this approach "Micro Frontends by the book". However, there are also some **disadvantages**:

- We need to version and distribute shared dependencies via npm. This can become quite an overhead, as after every change we need to assign a new version, publish it, and install it into the respective Micro Frontends.
- As each team can use its own tech stack, we can end up with different frameworks and different versions of them. This might lead to version conflicts in the browser and to increased bundle sizes.

Of course, there are approaches to **compensate for these drawbacks**: For instance, we can automate the distribution of shared libraries to minimize the overhead. Also, we can avoid version conflicts by not sharing libraries between different Micro Frontends. Wrapping these Micro Frontends into web components further abstracts away the differences between frameworks.

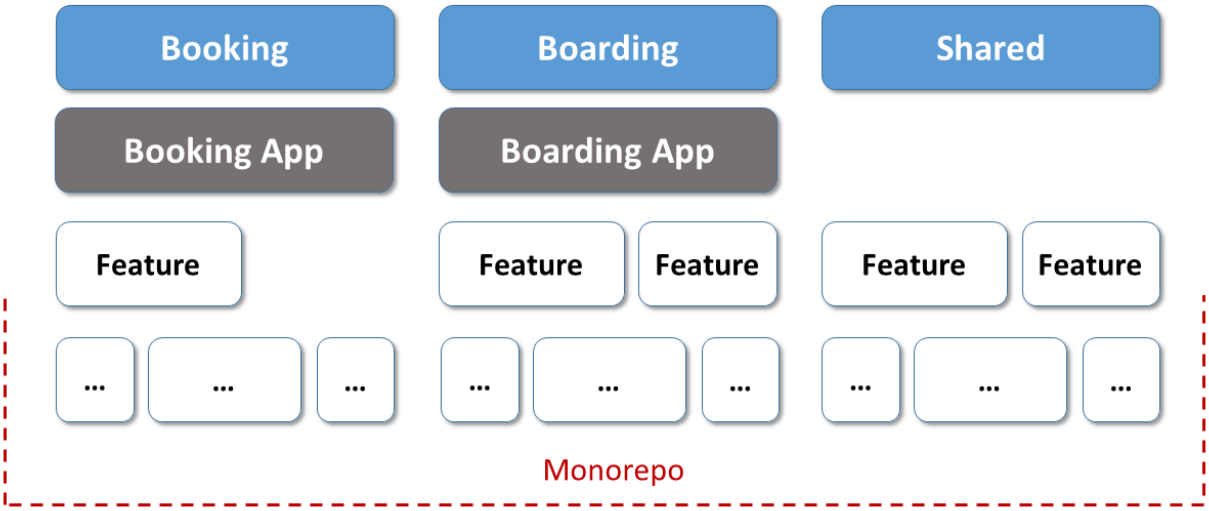
While this prevents version conflicts, we still have increased bundle sizes. Also, we might need some workarounds here or there as Angular is not designed to work with another version of itself in the same browser window. Needless to say that there is no support by the Angular team for this idea.

If you find out that the advantages of this approach outweigh the disadvantages, you find a solution for mixing and matching different frameworks and versions in one of the next chapters.

However, if you feel that the disadvantages weigh heavier, the next sections show an alternative.

Micro Frontends with Monorepos

Nearly all of the disadvantages outlined above can be prevented by putting all Micro Frontends into one sole monorepo:



Micro Frontends in a monorepo

Now, sharing libraries is easy and there is only one version of everything, hence we don't end up with version conflicts in the browser. We can also **keep some advantages outlined above**:

- Micro Frontends can be **isolated** from each other by using **linting** rules. They prevent one Micro Frontend from depending upon others. Hence, teams can separately evolve their Micro Frontend.
- Micro Frontends can still be **separately deployed**.

Now, the question is, where's the catch? Well, the thing is, now we are **giving up** some of the **freedom**: Teams need to agree on **one version** of dependencies like Angular and on a common update cycle for them. To put it in another way: We trade in some freedom to prevent version conflicts and increased bundle sizes.

One more time, you need to evaluate all these consequences for your very project. Hence, you need to know your architecture goals and prioritize them. As mentioned, I've seen both working in the wild in several projects. It's all about the different consequences.

Monorepo Example

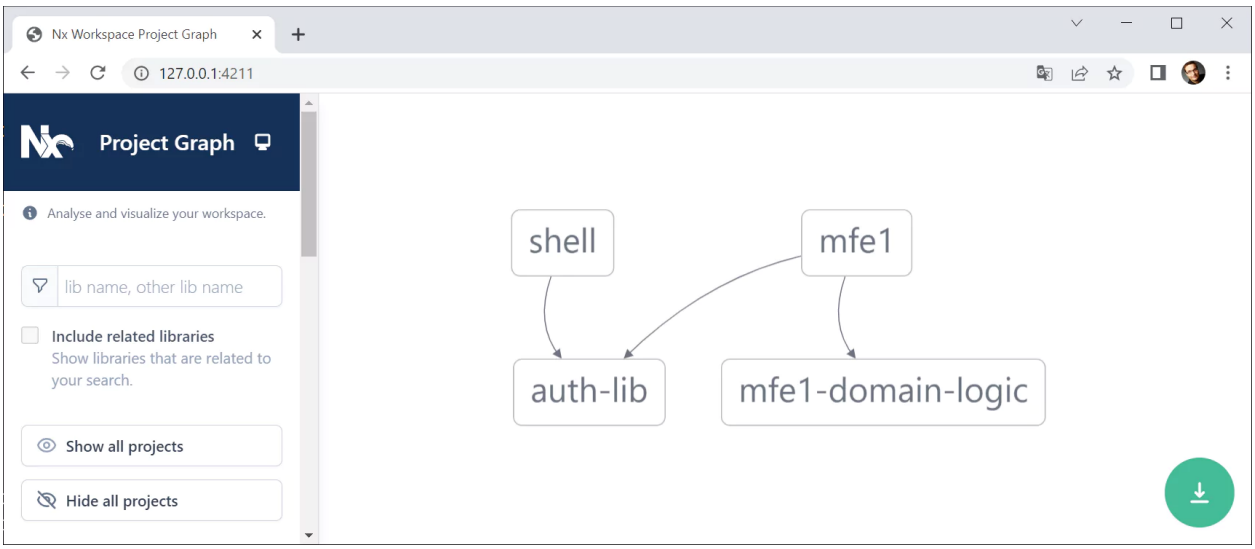
After discussing the consequences of the approach outlined here, let's have a look at an implementation. The example used here is a Nx monorepo with a Micro Frontend shell (`shell`) and a Micro Frontend (`mfe1`, "micro frontend 1"). Both share a common library for authentication (`auth-lib`) that is also located in the monorepo. Also, `mfe1` uses a library `mfe1-domain-logic`.

If you haven't used Nx before, just assume a CLI workspace with tons additional features. You can find more infos on Nx in our tutorial[51].

To visualize the monorepo's structure, one can use the Nx CLI to request a dependency graph:

```
1 nx graph
```

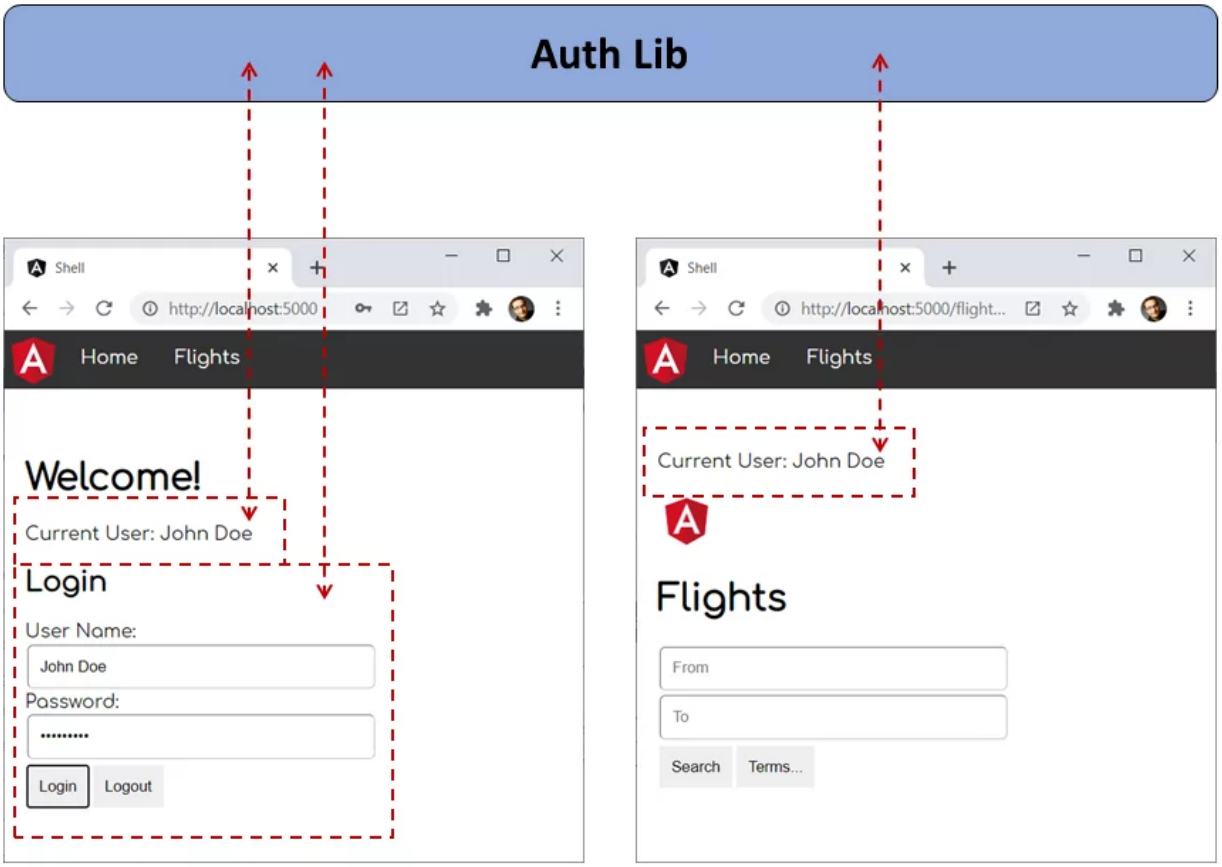
If you don't have installed this CLI, you can easily get it via npm (`npm i -g nx`). The displayed graph looks like this:



Dependency Graph generated by Nx

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`:

[51] https://www.angulararchitects.io/aktuelles/tutorial-first-steps-with-nx-and-angular-architecture/



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.base.json` (Nx) or `tsconfig.json` (Angular CLI):

```
"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 remembers 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 {
    // Authentication for honest users
    // (c) Manfred Steyer
    this._userName = userName;
  }

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

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.

Let's assume the shell is using the `AuthComponent` and `mfe1` uses the `UserComponent`. As our goal is to only load the `auth-lib` once, this also allows for sharing information on the logged-in user.

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 the following commands:

```
npm i @angular-architects/module-federation -D

npm g @angular-architects/module-federation:init
  --project shell --port 4200 --type host

npm g @angular-architects/module-federation:init
  --project mfe1 --port 4201 --type remote
```

> Meanwhile, Nx also ships with its own support for Module Federation[52]. Beyond the covers, it handles Module Federation in a very similar way as the plugin used here.

This generates a webpack config for Module Federation. Since version 14.3, the `withModuleFederationPlugin` provides a property `sharedMappings`. Here, we can register the monorepo internal libs we want to share at runtime:

```
// apps/shell/webpack.config.js

const { shareAll, withModuleFederationPlugin } =
  require('@angular-architects/module-federation/webpack');

module.exports = withModuleFederationPlugin({

    remotes: {
        'mfe1': "http://localhost:4201/remoteEntry.js"
    },

    shared: shareAll({
      singleton: true,
      strictVersion: true,
      requiredVersion: 'auto'
    }),

    sharedMappings: ['@demo/auth-lib'],

});
```

As sharing is always an opt-in in Module Federation, we also need the same setting in the Micro Frontend's configuration:

[52] https://nx.dev/module-federation/micro-frontend-architecture

```
// apps/mfe1/webpack.config.js

const { shareAll, withModuleFederationPlugin } =
  require('@angular-architects/module-federation/webpack');

module.exports = withModuleFederationPlugin({

    name: "mfe1",

    exposes: {
      './Module': './apps/mfe1/src/app/flights/flights.module.ts',
    },

    shared: shareAll({
      singleton: true,
      strictVersion: true,
      requiredVersion: 'auto'
    }),

    sharedMappings: ['@demo/auth-lib'],

});
```

> Since version 14.3, the Module Federation plugin shares all libraries in the monorepo by default. To get this default behavior, just skip the `sharedMappings` property. If you use it, only the mentioned libs are shared.

Trying it out

To try this out, just start the two applications. As we use Nx, this can be done with the following command:

```
nx run-many --target serve --all
```

The switch `--all` starts all applications in the monorepo. Instead, you can also go with the switch `--projects` to just start a subset of them:

```
nx run-many --target serve --projects shell,mfe1
```

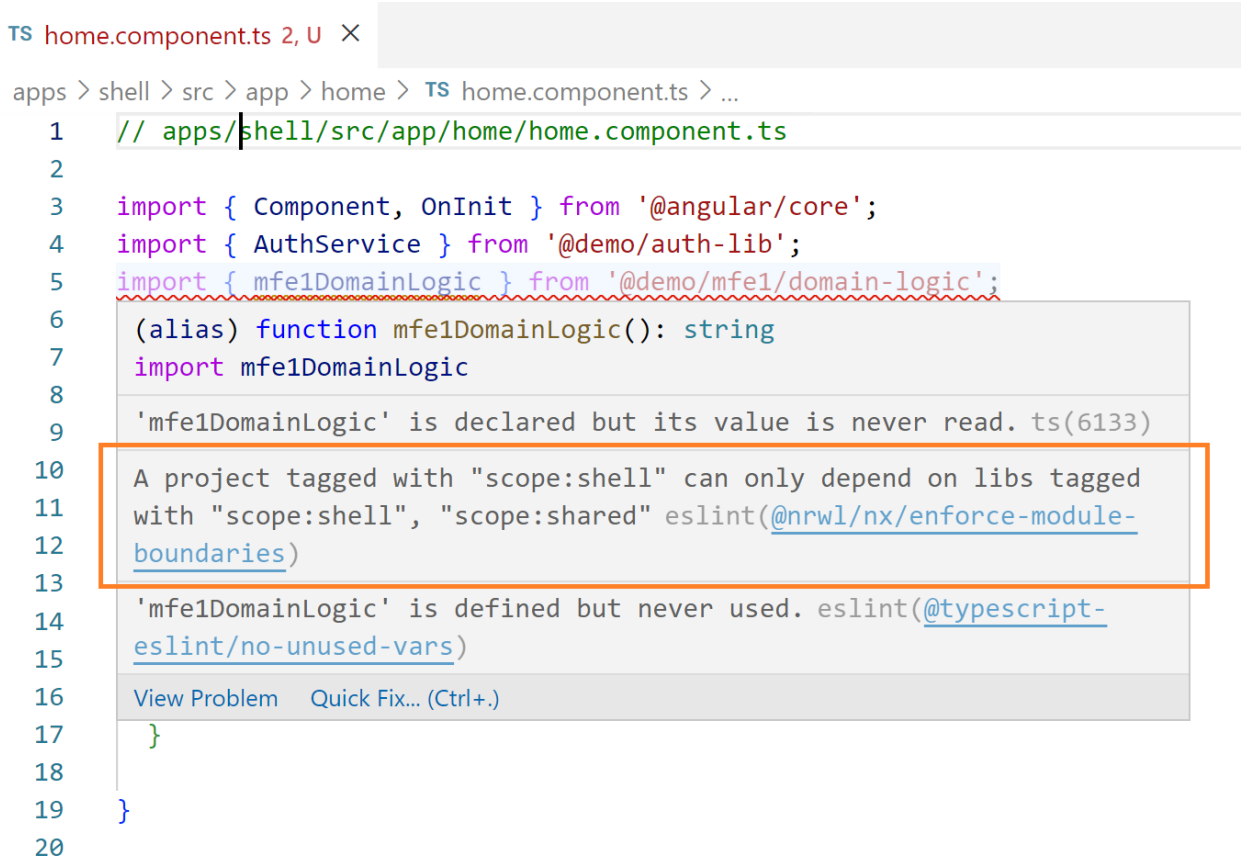
> `--project` takes a comma-separated list of project names. Spaces are **not** allowed.

After starting the applications, 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.

Isolating Micro Frontends

One important goal of a Micro Frontend architecture is to isolate Micro Frontends from each other. Only if they don't depend on each other, they can be evolved by autarkic teams. For this, Nx provides **linting** rules. Once in place, they give us errors when we directly reference code belonging to another Micro Frontend and hence another business domain.

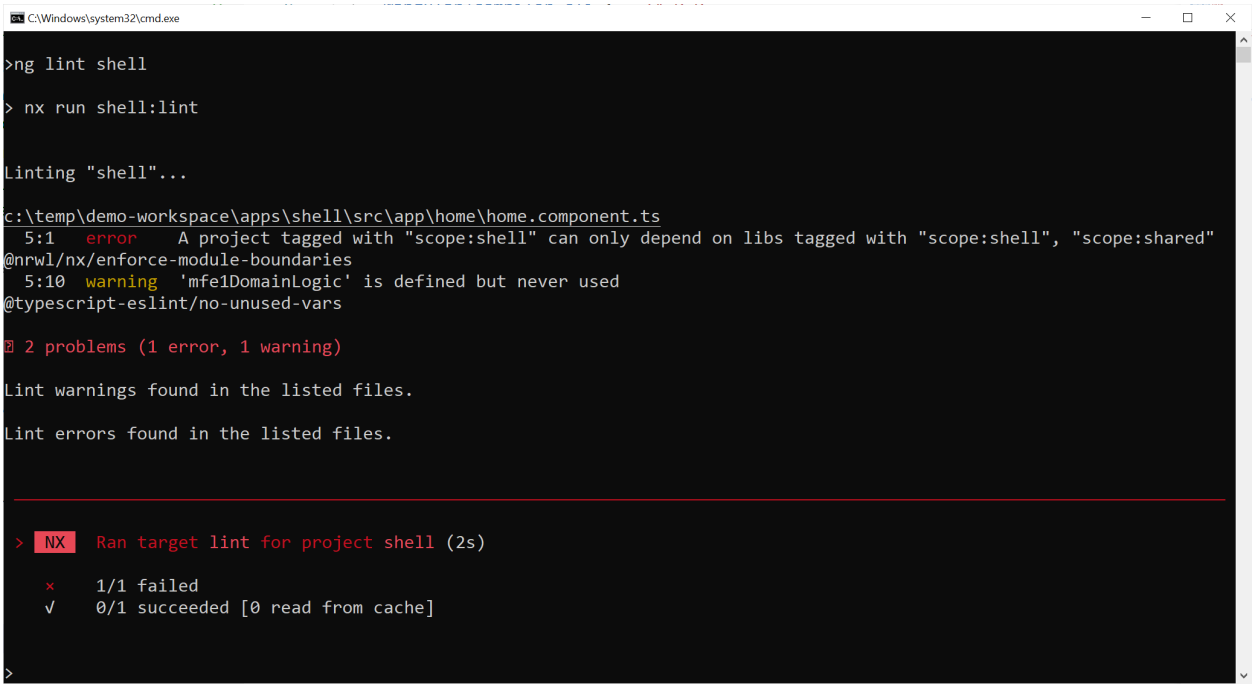
In the following example, the shell tries to access a library belonging to `mfe1`:



Linting prevents coupling between Micro Frontends

To make these error messages appear in your IDE, you need **eslint** support. For Visual Studio Code, this can be installed via an extension.

Besides checking against linting rules in your IDE, one can also call the linter on the command line:



Linting on the command line

The good message: If it works on the command line, it can be automated. For instance, your **build process** could execute this command and **prevent merging** a feature into the main branch if these linting rules fail: No broken windows anymore.

For configuring these linting rules, we need to **add tags** to each app and lib in our monorepo. For this, you can adjust the `project.json` in the app's or lib's folder. For instance, the `project.json` for the shell can be found here: `apps/shell/project.json`. At the end, you find a property tag, I've set to `scope:shell`:

```
{
    [...]
    "tags": ["scope:shell"]
}
```

The value for the tags are just strings. Hence, you can set any possible value. I've repeated this for `mfe1` (`scope:mfe1`) and the `auth-lib` (`scope:auth-lib`).

Once the tags are in place, you can use them to define **constraints** in your **global eslint** configuration (`.eslintrc.json`):

```
"@nrwl/nx/enforce-module-boundaries": [
  "error",
  {
    "enforceBuildableLibDependency": true,
    "allow": [],
    "depConstraints": [
      {
        "sourceTag": "scope:shell",
        "onlyDependOnLibsWithTags": ["scope:shell", "scope:shared"]
      },
      {
        "sourceTag": "scope:mfe1",
        "onlyDependOnLibsWithTags": ["scope:mfe1", "scope:shared"]
      },
      {
        "sourceTag": "scope:shared",
        "onlyDependOnLibsWithTags": ["scope:shared"]
      }
    ]
  }
]
```

After changing global configuration files like the `.eslintrc.json`, it's a good idea to restart your IDE (or at least affected services of your IDE). This makes sure the changes are respected.

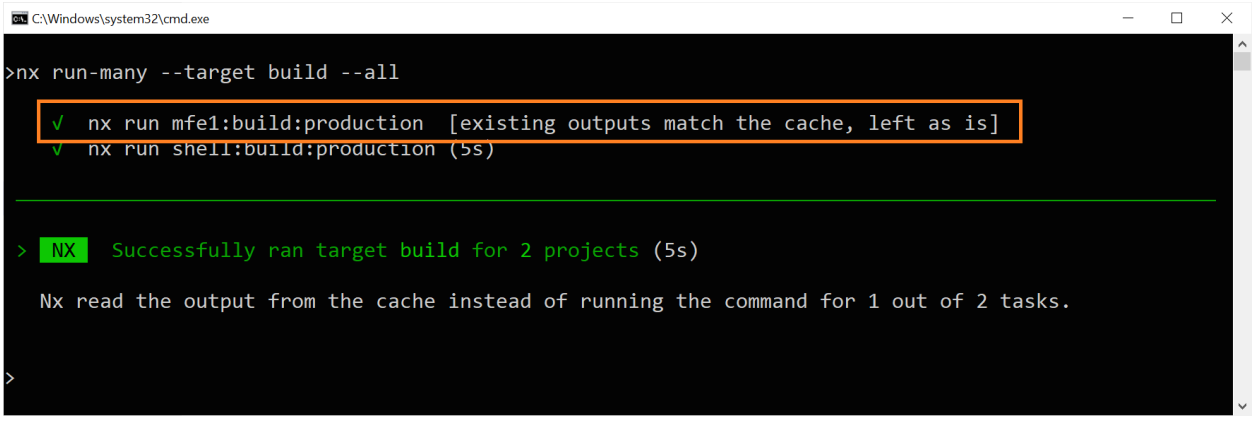
More on these ideas and their implementation with Nx can be found in the chapters on Strategic Design.

Incremental Builds

To build all apps, you can use Nx' `run-many` command:

```
nx run-many --target build --all
```

However, this does not mean that Nx always rebuilds all the Micro Frontends as well as the shell. Instead, it **only rebuilds** the **changed** apps. For instance, in the following case mfe1 was not changed. Hence, only the shell is rebuilt:



Nx only rebuilds changed apps

> Using the build cache to only recompile changed apps can **dramatically speed up your build times**.

This also works for **testing**, **e2e-tests**, **and linting** out of the box. If an application (or library) hasn't been changed, it's neither retested nor relinted. Instead, the result is taken from the Nx **build cache**.

By default, the build cache is located in `node_modules/.cache/nx`. However, there are several options for configuring how and where to cache.

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**:

```
1 nx print-affected --type app --select projects
```

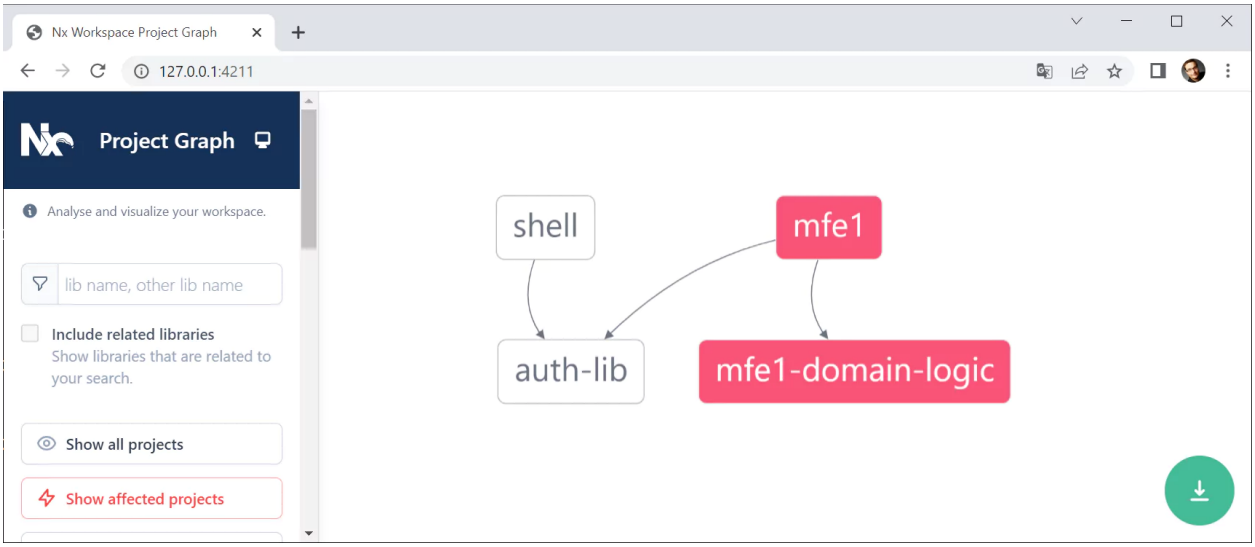
You might also want to detect the changed applications as part of your **build process**.

> Redeploying all applications that have been changed or affected by a (lib) change is vital, if you share libraries at runtime. If you have a **release branch**, it's enough to just redeploy all apps that have been changed in this branch.

If you want to have a **graphical** representation of the changed parts of your monorepo, you can request a dependency graph with the following command:

```
nx affected:graph
```

Assuming we changed the `domain-logic` lib used by `mfe1`, the result would look as follows:



Dependency graph shows affected projects

By default, the shown commands **compare** your current working directory with the **main branch**. However, you can use these commands with the switches `--base` and `--head`.

```
nx print-affected --type app --select projects --base branch-or-commit-a --head bran\
ch-or-commit-b
```

These switches take a **commit hash** or the name of a **branch**. In the latter case, the last commit of the mentioned branch is used for the comparison.

Conclusion

By using monorepos for Micro Frontends you trade in some freedom for preventing issues. You can still deploy Micro Frontends separately and thanks to linting rules provided by Nx Micro Frontends can be isolated from each other.

However, you need to agree on common versions for the frameworks and libraries used. Therefore, you don't end up with version conflicts at runtime. This also prevents increased bundles.

Both works, however, both has different consequences. It's on you to evaluate these consequences for your very project.

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.

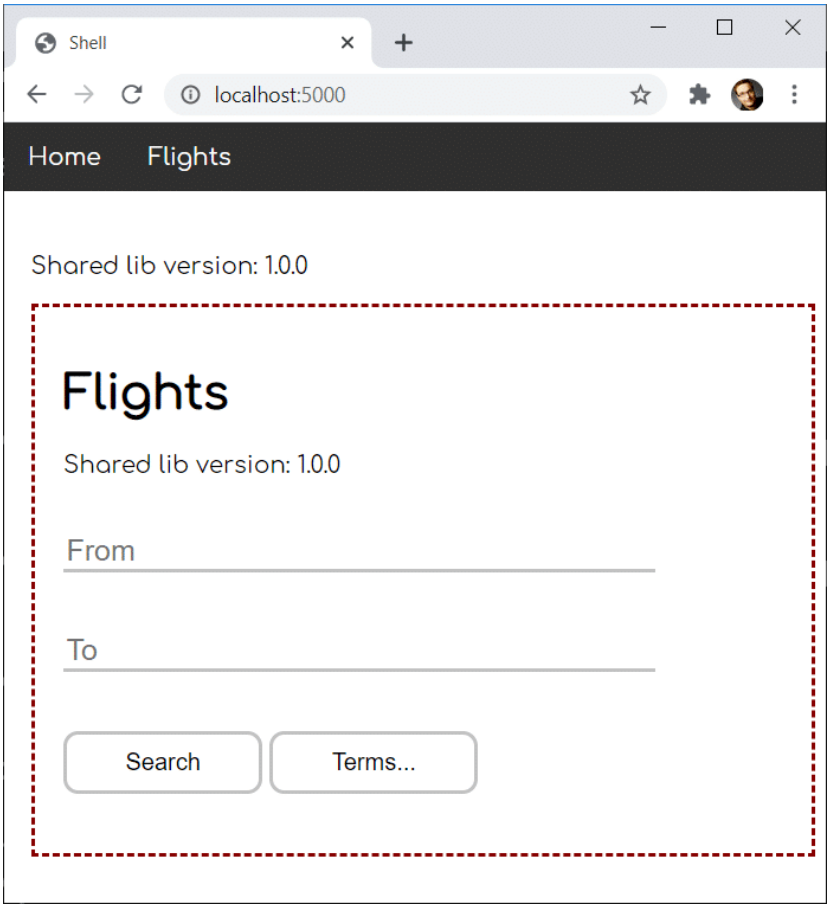
For dealing with such cases, Module Federation provides several options. In this chapter, I present these options by looking at different scenarios. The source code[53] for these scenarios can be found in my GitHub account[54].

> Big thanks to Tobias Koppers[55], founder of webpack, for answering several questions about this topic and for proofreading this chapter.

Example Used Here

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

[53] https://github.com/manfredsteyer/module_federation_shared_versions
[54] https://github.com/manfredsteyer/module_federation_shared_versions
[55] https://twitter.com/wSokra



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"]
})
```

If you are new to Module Federation, you can find an explanation about it here[56].

The package useless-lib[57] 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,

[56] https://www.angulararchitects.io/aktuelles/the-microfrontend-revolution-module-federation-in-webpack-5/
[57] https://www.npmjs.com/package/useless-lib

both use the same version (`1.0.0`), and hence they can share it. Therefore, `useless-lib` is only loaded once.

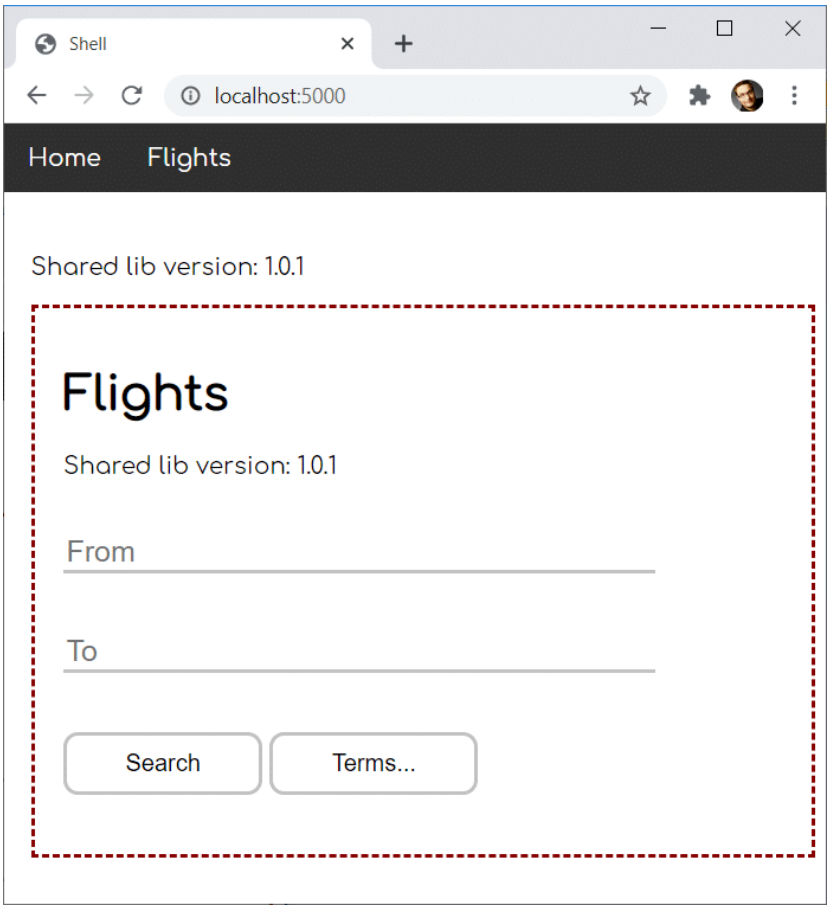
However, in the following sections, we will examine what happens if there are version mismatches between the `useless-lib` used in the shell and the one used in the `microfrontend`. This also allows me to explain different concepts Module Federation implements for dealing with such situations.

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).

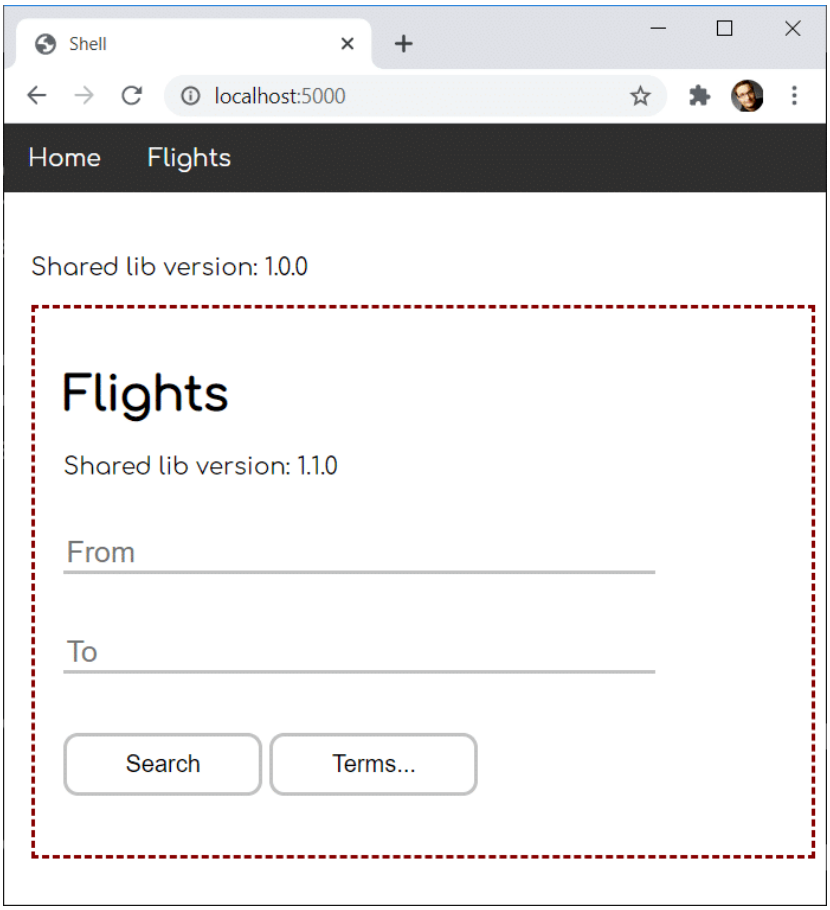
Fallback Modules for Incompatible Versions

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.

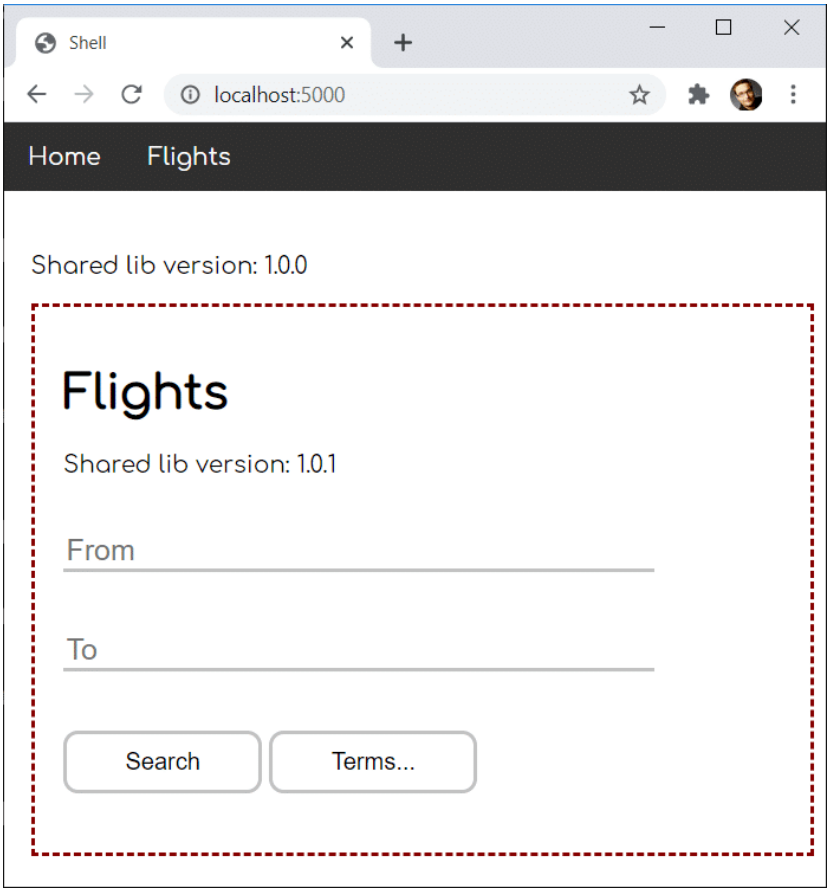
Differences With Dynamic Module Federation

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

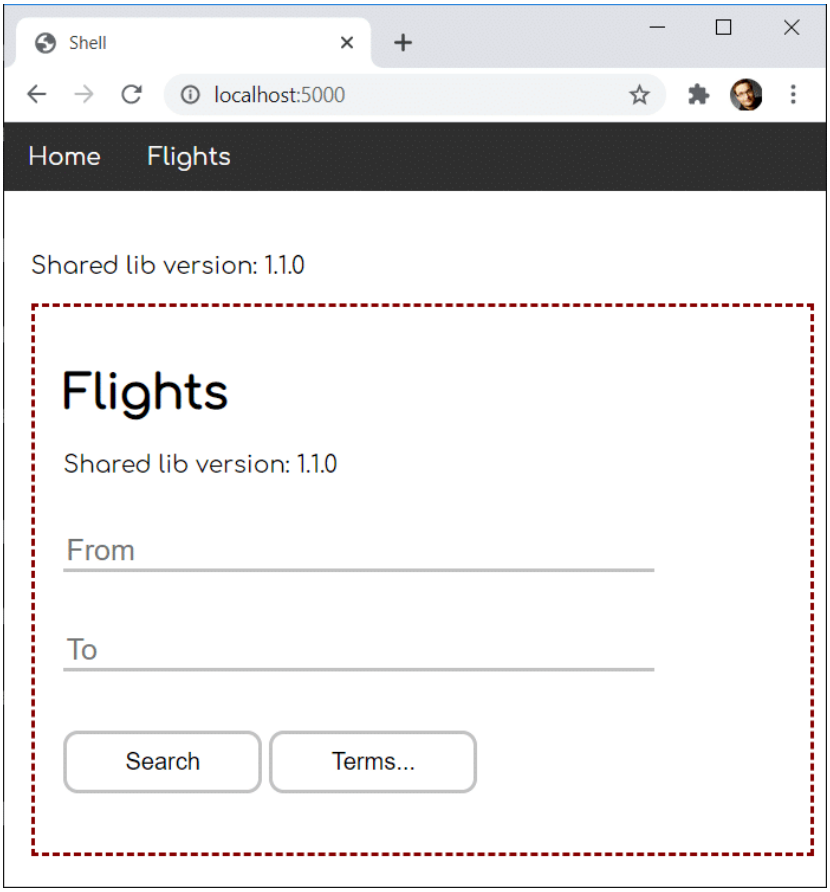
If there were other static remotes (e. g. micro frontends), the shell could also choose for one of their versions according to semantic versioning, as shown above.

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:

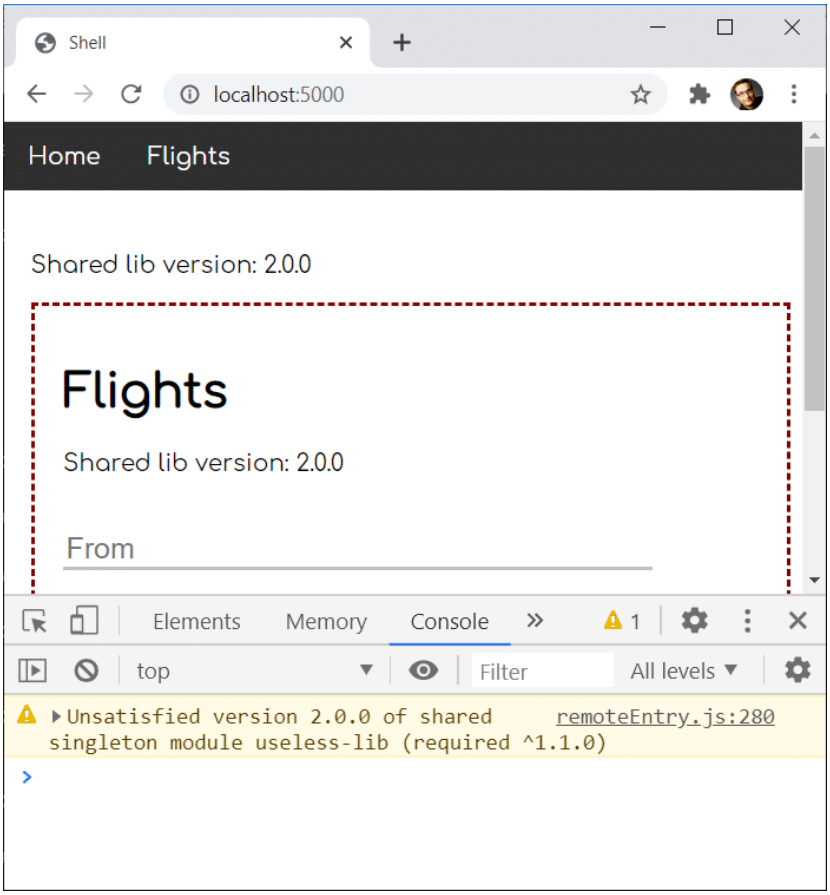
```
// Shell
shared: {
  "rxjs": {},
  "useless-lib": {
    singleton: true,
  }
},
```

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`:



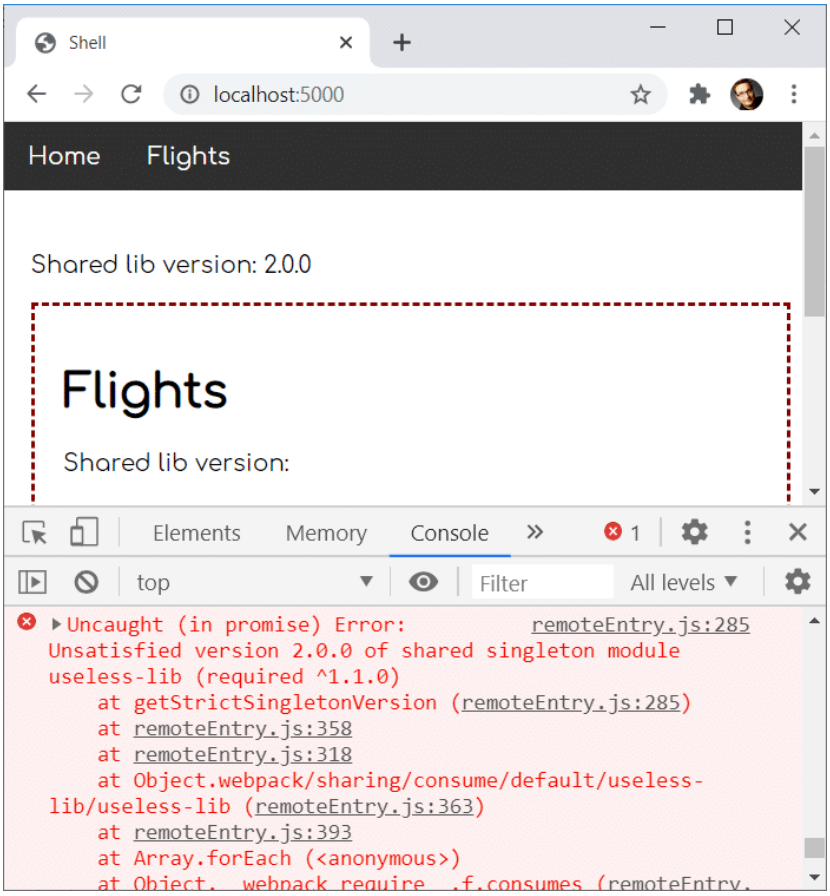
Module Federation only loads the highest version for singletons

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.

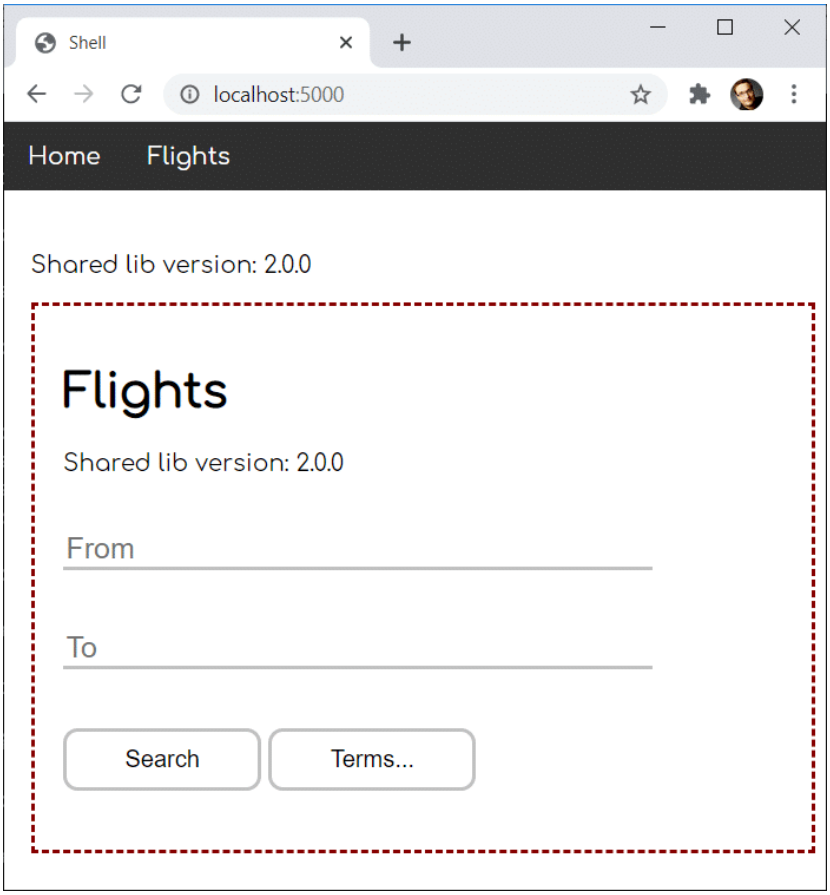
To explore this option, let's one more time assume the following version mismatch:

- **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

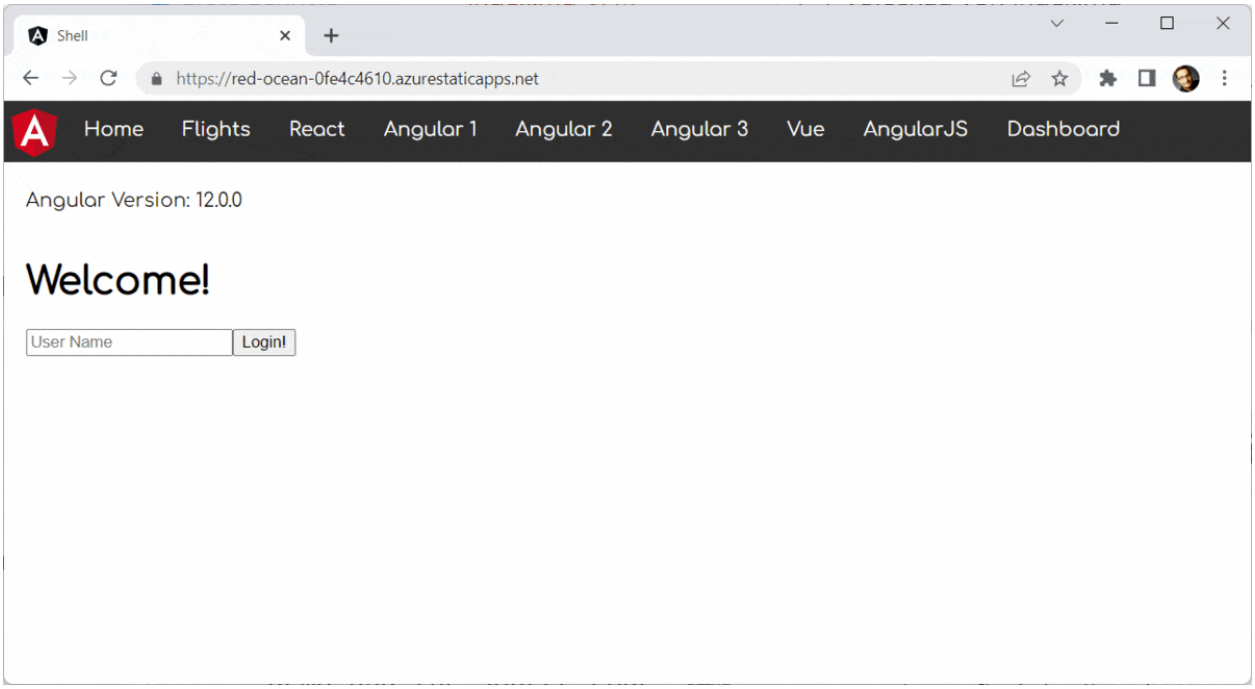
Module Federation brings several options for dealing with different versions and version mismatches. Most of the time, you don't need to do anything, as it uses semantic versioning to decide for the highest compatible version. If a remote needs an incompatible version, it falls back to such one by default.

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

Most articles on Module Federation assume, you have just one version of your major Framework, e. g. Angular. However, what to do if you have to mix and match different versions or different frameworks? No worries, we got you covered. This chapter uses an example to explain how to develop such a scenario in 4 steps.



Example

Please find the live demo and the source code here:

- Live Example[58]
- Source Code Shell[59]
- Source Code for Micro Frontend[60]
- Source Code for Micro Frontend with Routing[61]
- Source Code for Micro Frontend with Vue[62]

[58] https://red-ocean-0fe4c4610.azurestaticapps.net
[59] https://github.com/manfredsteyer/multi-framework-version
[60] https://github.com/manfredsteyer/angular-app1
[61] https://github.com/manfredsteyer/angular3-app
[62] https://github.com/manfredsteyer/vue-js

- Source Code for Micro Frontend with React[63]
- Source Code for Micro Frontend with AngularJS[64]

Pattern or Anti-Pattern?

In his recent talk on Micro Frontend Anti Patterns[65], my friend Luca Mezzalira[66] mentions using several frontend frameworks in one application. He calls this anti pattern the Hydra of Lerna[67]. This name comes from a water monster in Greek and Roman mythology having several heads.

There's a good reason for considering this an anti pattern: Current frameworks are not prepared to be bootstrapped in the same browser tab together with other frameworks or other versions of themselves. Besides leading to bigger bundles, this also increases the complexity and calls for some workarounds.

However, Luca also explains that there are some situations where such an approach **might be needed**. He brings up the following examples:

1. Dealing with legacy systems
2. Migration to a new UI framework/ library
3. After merging companies with different tech stacks

This all speaks right from my heart and perfectly correlates with my "story" I'm telling a lot at conferences and at our company workshops: Try to avoid mixing frameworks and versions in the browser. However, if you have a good reason for doing it after ruling out the alternatives, there are ways for making Multi-Framework/ Multi-Version Micro Frontends work.

As always in the area of software architecture – and probably in life as general – it's all about **trade-offs**. So if you find out that this approach comes with less drawbacks than alternatives with respect to your very **architecture goals**, lets go for it.

Micro Frontends as Web Components?

While not 100% necessary, it can be a good idea to wrap your Micro Frontends in Web Components.

[63] https://github.com/manfredsteyer/react-app
[64] https://github.com/manfredsteyer/angularjs-app
[65] https://www.youtube.com/watch?v=asXPRrg6M2Y
[66] https://twitter.com/lucamezzalira
[67] https://en.wikipedia.org/wiki/Lernaean_Hydra

Micro Frontends wrapped in Web Components

This brings several advantages:

- Abstracting differences between frameworks
- Mounting/ Unmounting Web Components is easy
- Shadow DOM helps with isolating CSS styles
- Custom Events and Properties allow to communicate

The first two options correlate with each other. We need to display and hide our Micro Frontends on demand, e. g. when activating a specific menu item. As each Micro Frontend is a self-contained frontend, this also means we have to bootstrap it on demand in the middle of our page. For this, different frameworks provide different methods or functions. When wrapped into Web Components, all we need to do is to add or remove the respective HTML element registered with the Web Component.

Isolating CSS styles via Shadow DOM helps to make teams more self-sufficient. However, I've seen that quite often teams trade in a bit of independence for some global CSS rules provided by the shell. In this case, the Shadow DOM emulation provided by Angular (with and without Web Components) is a good choice: While it prevents styles from other components bleeding into yours, it allows to share global styles too.

Also, Custom Events and Properties seem to be a good choice for communicating at first glance. However, for the sake of simplicity, meanwhile, I prefer a simple object acting as a mediator or "mini message bus" in the global namespace.

In general, we have to see that such Web Components wrapping whole Micro Frontends are no typical Web Components. I'm stressing this out because sometimes people confuse the idea of a (Web) Component with the idea of a Micro Frontend or use these terms synonymously. This leads to far too fine-grained Micro Frontends causing lots of issues with integration.

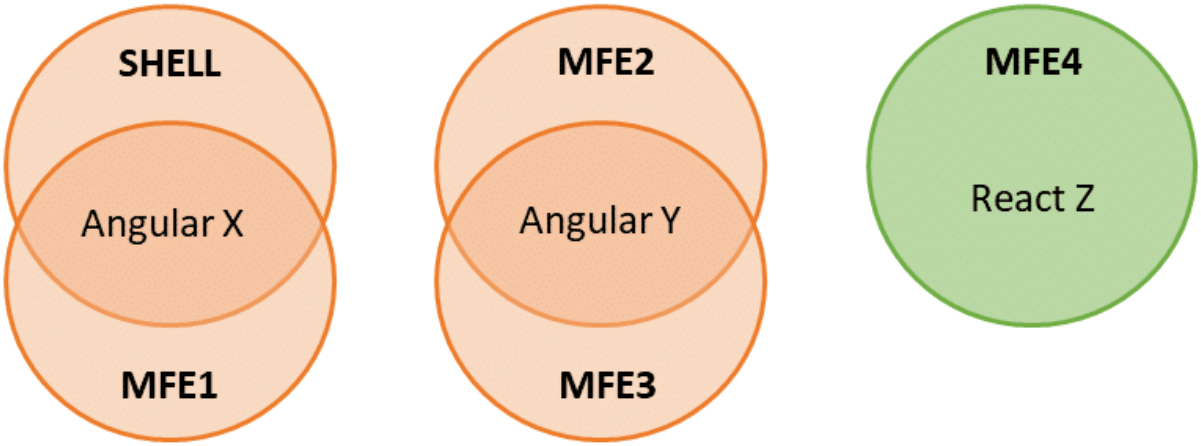
Do we also need Module Federation?

Module Federation makes it easy to load parts of other applications into a host. In our case, the host is the Micro Frontend shell. Also, Module Federation allows for sharing libraries between the shell

and the micro frontends.

It even comes with several strategies for dealing with versions mismatches. For instance, we could configure it to reuse an existing library if the versions match exactly. Otherwise, we could instruct it to load the version is has been built with.

Loading the discussed Micro Frontends with Module Federation hence gives us the best of both worlds. We can share libraries when possible and load our own when not:



The Best of Both Worlds

Implementation in 4 steps

Now, after discussing the implementation strategy, let's look at the promised 4 steps for building such a solution.

Step 1: Wrap your Micro Frontend in a Web Component

For wrapping Angular-based Micro Frontends in a Web Component, you can go with Angular Elements provided by the Angular team. Install it via npm:

```
1  npm i @angular/elements
```

After installing it, adjust your `AppModule` as follows:

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

@NgModule({
  [...]
  declarations: [
    AppComponent
  ],
  bootstrap: [] // No bootstrap components!
})
export class AppModule implements DoBoostrap {
  constructor(private injector: Injector) {
  }

  ngDoBootstrap() {
    const ce = createCustomElement(AppComponent, {injector: this.injector});
    customElements.define('angular1-element', ce);
  }

}
```

This does several things:

- By going with an empty bootstrap array, Angular won't directly bootstrap any component on startup. However, in such cases, Angular demands us of placing a custom bootstrap logic in the method `ngDoBootstrap` described by the `DoBoostrap` interface.
- `ngDoBootstrap` uses Angular Elements' `createCustomElement` to wrap your `AppComponent` in a Web Component. To make it work with DI, you also need to pass the current `Injector`.
- The method `customElements.define` registers the Web Component under the name `angular1-element` with the browser.

The result of this is that the browser will mount the Application in every `angular1-element` tag that occurs in your application.

If your framework doesn't directly support web components, you can also hand-wrap your application. For instance, a React component could be wrapped as follows:

```
// app.js
import React from 'react'
import ReactDOM from 'react-dom'

class App extends React.Component {

  render() {
    const reactVersion = require('./package.json').dependencies['react'];

    return ([
        <h1>
          React
        </h1>,
        <p>
          React Version: {reactVersion}
        </p>
    ])
  }
}

class Mfe4Element extends HTMLElement {
  connectedCallback() {
    ReactDOM.render(<App/>, this);
  }
}

customElements.define('react-element', Mfe4Element);
```

Step 2: Expose your Web Component via Module Federation

To be able to load the Micro Frontends into the shell, we need to expose the Web Components wrapping them via Module Federation. For this, add the package `@angular-architects/module-federation` to your Angular-based Micro Frontend:

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

This installs and initializes the package. If you go with Nx and Angular, its more usual to do both steps separately:

```
npm i @angular-architects/module-federation -D

ng g @angular-architects/module-federation:init
```

In the case of other frameworks like React or Vue, this all is just about adding the `ModuleFederationPlugin` to the webpack configuration. Please remember that you need to bootstrap your application asynchronously in most cases. Hence, your entry file will more or less just contain a dynamic `import` loading the rest of the application.

For this reason, the above discussed React-based Micro Frontend uses the following `index.js` as the entry point:

```
// index.js
import('./app');
```

Similarly, `@angular-architects/module-federation` is moving the bootstrap code from `main.ts` into a newly created `bootstrap.ts` and imports it:

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

This common pattern gives Module Federation the necessary time for loading the shared dependencies.

After setting up Module Federation, expose the Web Component-based wrapper via the webpack configuration:

```
// webpack.config.js
[...]
module.exports = {
  [...]
  plugins: [
    new ModuleFederationPlugin({

      name: "angular1",
      filename: "remoteEntry.js",

      exposes: {
        './web-components': './src/bootstrap.ts',
      },

      shared: share({
        "@angular/core": { requiredVersion: "auto" },
```

```
        "@angular/common": { requiredVersion: "auto" },
        "@angular/router": { requiredVersion: "auto" },
        "rxjs": { requiredVersion: "auto" },

        ...sharedMappings.getDescriptors()
      }),
      [...]
    })
  ],
};
```

> As the goal is to show how to mix different versions of Angular, this Micro Frontend uses Angular 12 while the shell shown below uses a more recent Angular version. Hence, also an older version of @angular-architects/module-federation and the original more verbose configuration is used. Please find details on different versions[68] here.

The settings in the section `shared` make sure we can mix several versions of a framework but also reuse an already loaded one if the version numbers fit exactly. For this, `requiredVersion` should point to the installed version – the one, you also find in your `package.json`. The helper method `share` that comes with `@angular-architects/module-federation` takes care of this when setting `requiredVersion` to `auto`.

> While according to semantic versioning an Angular library with a higher minor or patch version is backwards compatible, there are no such guarantees for already compiled code. The reason is that the code emitted by the Angular compiler uses Angular's internal APIs semantic does not apply for. Hence, you should use an exact version number (without any ^ or ~).

Step 3: Perform Workarounds for Angular

To make several Angular application work together in one browser window, we need some workarounds[69]. The good message is, we've implemented them in a very slim add-on to `@angular-architects/modul` called @angular-architects/module-federation-tools[70].

Just install it (`npm i @angular-architects/module-federation-tools -D`) into **both, your Micro Frontends and your shell**. Then, bootstrap your shell and your Micro Frontends with its `bootstrap` method instead of with Angular's one:

[68] https://github.com/angular-architects/module-federation-plugin/blob/main/migration-guide.md
[69] https://www.angulararchitects.io/aktuelles/multi-framework-and-version-micro-frontends-with-module-federation-the-good-the-bad-the-ugly/
[70] https://www.npmjs.com/package/@angular-architects/module-federation-tools

```
// main.ts
import { AppModule } from './app/app.module';
import { environment } from './environments/environment';
import { bootstrap } from '@angular-architects/module-federation-tools';

bootstrap(AppModule, {
  production: environment.production,
  appType: 'microfrontend'  // for micro frontend
  // appType: 'shell',      // for shell
});
```

Step 4: Load Micro Frontends into the Shell

Also, enable Module Federation in your shell. If it is an Angular-based shell, add the `@angular-architects/module-f` plugin:

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

As mentioned above, in the case of Nx and Angular, perform the installation and initialization separately:

```
npm i @angular-architects/module-federation -D
ng g @angular-architects/module-federation:init --type host
```

> The switch –type host generates a typical host configuration. It is available since plugin version 14.3 and hence since Angular 14.

For this example, we don't need to adjust the generated `webpack.config.js`:

```
// webpack.config.js
const { shareAll, withModuleFederationPlugin } =
  require('@angular-architects/module-federation/webpack');

module.exports = withModuleFederationPlugin({

    shared: {
        ...shareAll({
          singleton: true,
          strictVersion: true,
          requiredVersion: 'auto'
```

```
        }),
    },

});
```

Other settings provided by the `ModuleFederationPlugin` aren't needed here.

After this, all you need is a lazy route, loading the Micro Frontends in question:

```
import { WebComponentWrapper, WebComponentWrapperOptions }
  from '@angular-architects/module-federation-tools';

export const APP_ROUTES: Routes = [
  [...]
  {
      path: 'react',
      component: WebComponentWrapper,
      data: {
        remoteEntry:
          'https://witty-wave-0a695f710.azurestaticapps.net/remoteEntry.js',
        remoteName: 'react',
        exposedModule: './web-components',

        elementName: 'react-element'
      } as WebComponentWrapperOptions
  },
  [...]
]
```

The `WebComponentWrapper` used here is provided by `@angular-architects/module-federation-tools`. It just loads the Web Component via Module Federation using the given key data. In the shown case, this react application is deployed as an Azure Static Web App. The values for `remoteName` and `exposedModule` can be found in the Micro Frontend's webpack configuration.

The wrapper component also creates an HTML element with the name `react-element` the Web Component is mounted in.

If you load a Micro Frontend compiled with Angular 13 or higher, you need to set the property `type` to `module`:

```
export const APP_ROUTES: Routes = [
    [...]
    {
        path: 'angular1',
        component: WebComponentWrapper,
        data: {
          type: 'module',
          remoteEntry: 'https://your-path/remoteEntry.js',
          exposedModule: './web-components',

          elementName: 'angular1-element'
        } as WebComponentWrapperOptions
    },
    [...]
}
```

Also, in the case of Angular 13+ you don't need the `remoteName` property. The reason for these two differences is that Angular CLI 13+ don't emit "old-style JavaScript" files anymore but JavaScript modules. Their handling in Module Federation is a bit different.

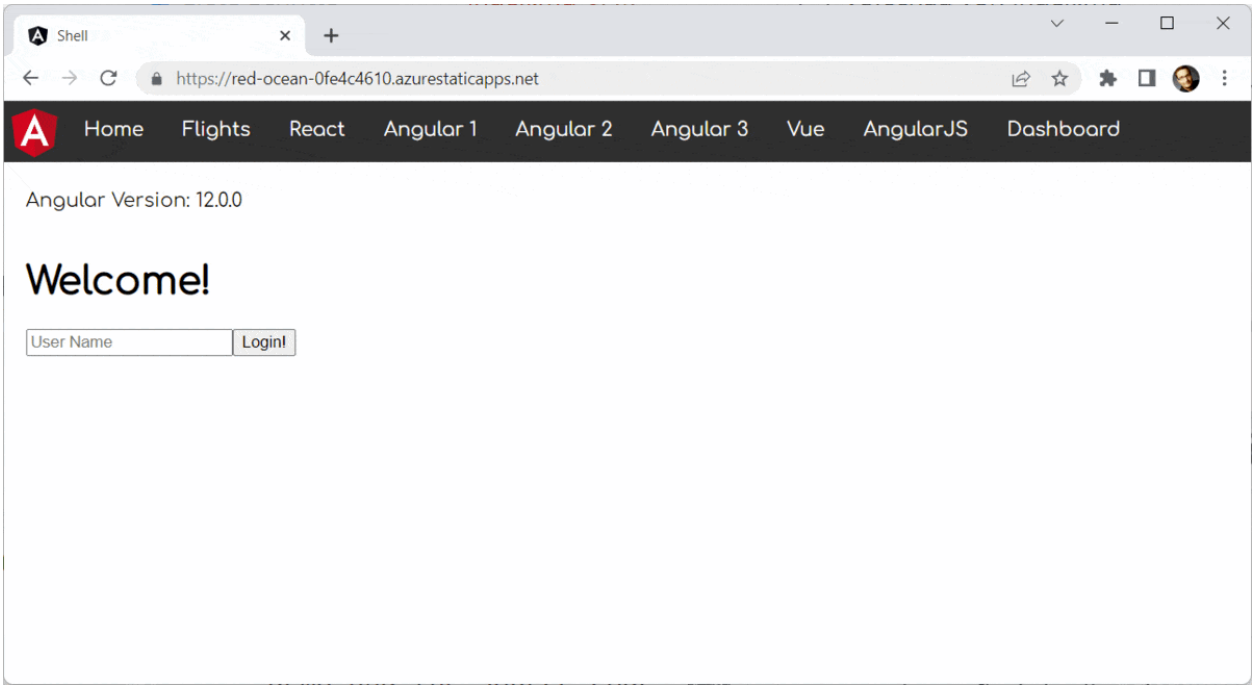
If your Micro Frontend brings its own router, you need to tell your shell that the Micro Frontend will append further segments to the URL. For this, you can go with the `startsWith` matcher also provided by `@angular-architects/module-federation-tools`:

```
import {
    startsWith,
    WebComponentWrapper,
    WebComponentWrapperOptions
}
from '@angular-architects/module-federation-tools';

[...]

export const APP_ROUTES: Routes = [
    [...]
    {
        matcher: startsWith('angular3'),
        component: WebComponentWrapper,
        data: {
            [...]
        } as WebComponentWrapperOptions
    },
    [...]
}
```

To make this work, the path prefix `angular3` used here needs to be used by the Micro Frontend too. As the routing config is just a data structure, you will find ways to add it dynamically.

Result

The result of this endeavor is an application that consists of different frameworks respective framework-versions:



Example

Whenever possible, the framework is shared. Otherwise, a new framework (version) is loaded by Module Federation. Another advantage of this approach is that it works without any additional meta framework. We just need some thin helper functions.

The drawbacks are increased complexity and bundle sizes. Also, we are leaving the path of the supported use cases: None of the frameworks has been officially tested together with other frameworks or other versions of itself in the same browser tab.

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, I present 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[71].

"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 },
},
```

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:

> 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.

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

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 dependency 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'
  },
},
```

Obviously, this is cumbersome and so we came up with another solution. Since version 12.3, @angular-architects/module-federation[72] 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.

[72] https://www.npmjs.com/package/@angular-architects/module-federation

```
[...]

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'
  },
})
```

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 your 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 },
},
```

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)

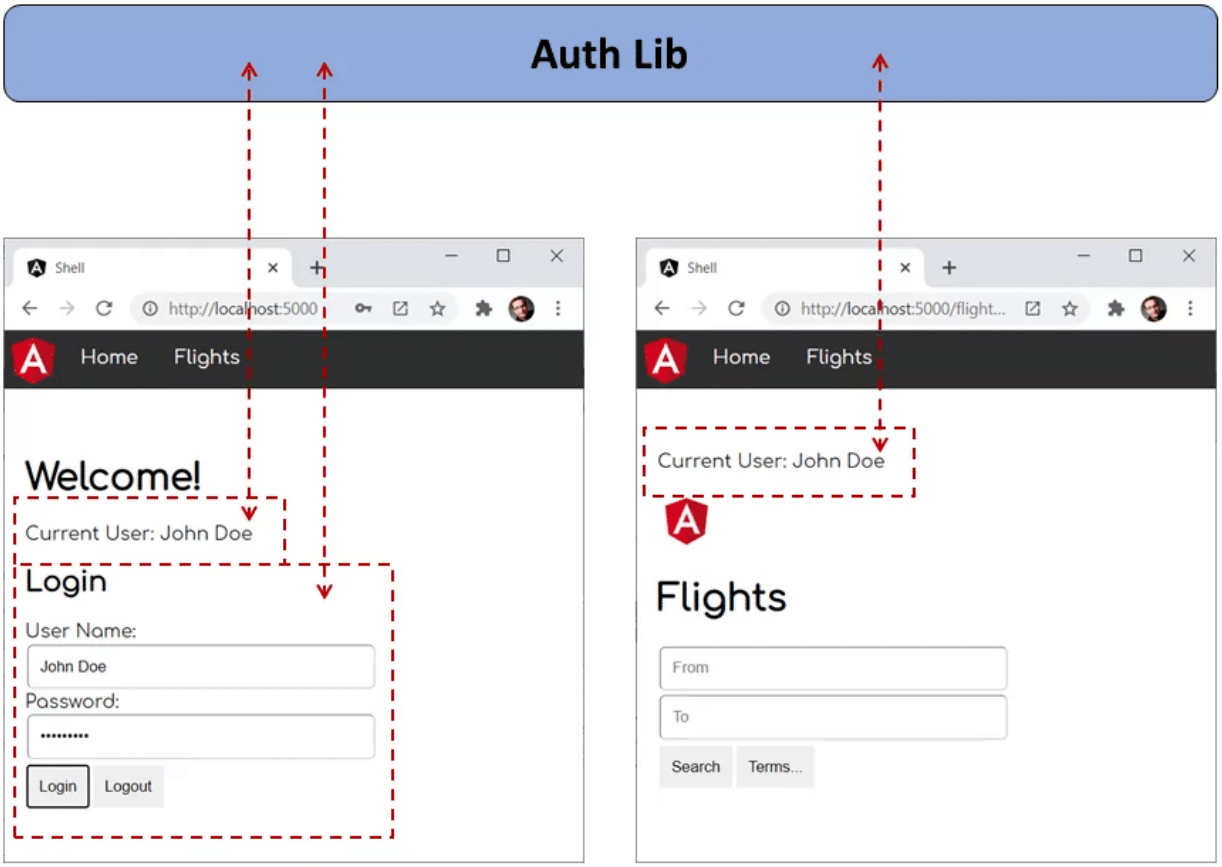
While `@angular/material` and `@angular/cdk` officially need `@angular/core` 10, the rest of the application already uses `@angular/core` 12. This shows that webpack looks into the `package.json` files of all the shared dependencies for determining the needed versions.

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'
  },
})
```

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 a configuration property called:

```
module.exports = withModuleFederationPlugin({

    // Shared packages:
    shared: [...],

    // Explicitly share mono-repo libs:
    sharedMappings: ['auth-lib'],

});
```

> **Important**: Since Version 14.3, the withModuleFederationPlugin helper automatically shares **all** mapped paths if you don't use the property `sharedMappings` at all. Hence, the issue described here, will not happen.

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;
  }

}
```

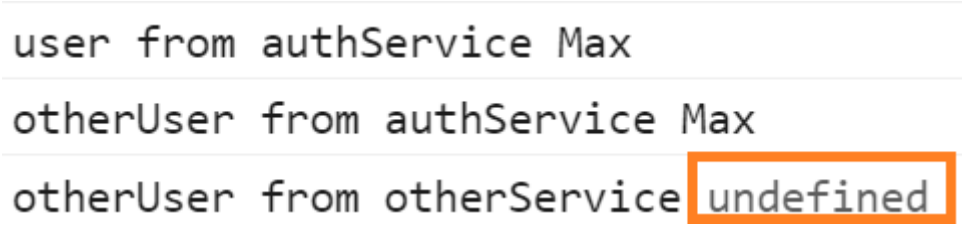
The shell's `AppComponent` is just calling the `login` method:

```
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:
    console.log('user from authService', authService.user);
    console.log('otherUser from authService', authService.otherUser);
    console.log('otherUser from otherService', otherService.userName);

  }
}
```

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:



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'
  }),
}
```

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

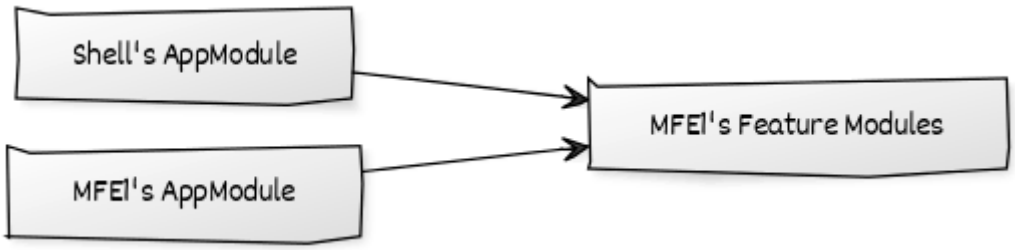
NullInjectorError: Service expected in Parent Scope (Root Scope)

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' },
})
```

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.

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 in the chapter on version mismatches.

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

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

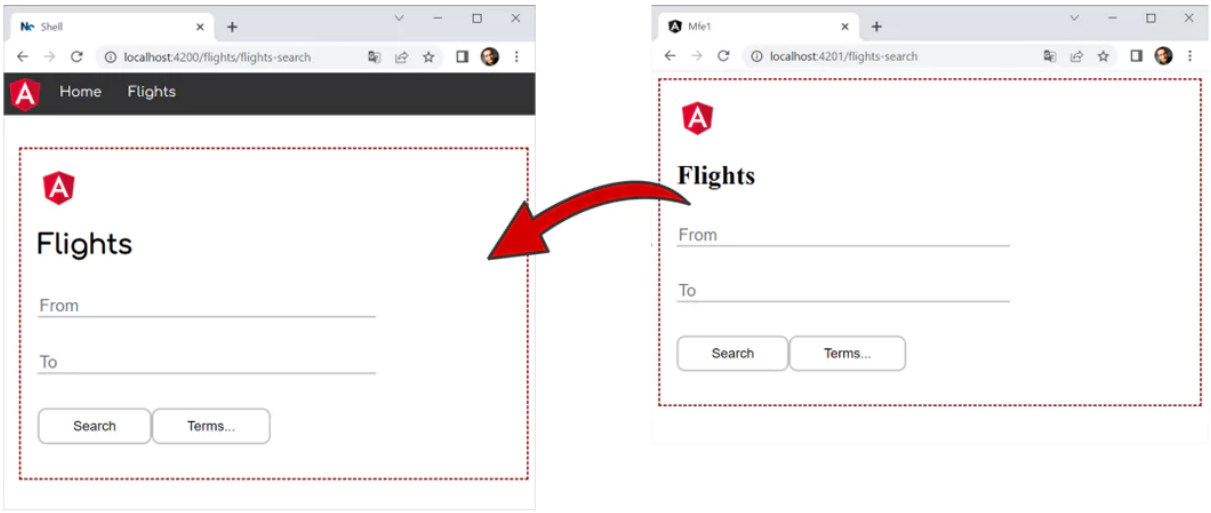
Module Federation is really clever when it comes to auto-detecting details and compensating for version mismatches. However, it can only be as good as the meta data it gets. To avoid getting off the rails, you should remember the following:

- **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.

Module Federation with Angular's Standalone Components

Most tutorials on Module Federation and Angular expose Micro Frontends in the form of NgModules. However, with the introduction of Standalone Components we will have lightweight Angular solutions not leveraging NgModules anymore. This leads to the question: How to use Module Federation in a world without NgModules?

In this chapter, I give answers. We see both, how to expose a bunch of routes pointing to Standalone Components and how to load an individual Standalone Component. For this, I've updated my example to fully work without NgModules:



The example was updated to fully use Standalone Components

Source code[73] (branch: standalone-solution).

Router Configs vs. Standalone Components

In general, we could directly load Standalone Components via Module Federation. While such a fine-grained integration seems to be fine for plugin-systems, Micro Frontends are normally more coarse-grained. It's usual that they represent a whole business domain which in general contains several use cases belonging together.

[73] https://github.com/manfredsteyer/module-federation-plugin-example/tree/standalone-solution

Interestingly, Standalone Components belonging together can be grouped using a router config. Hence, we can expose and lazy load such router configurations.

Initial Situation: Our Micro Frontend

The Micro Frontend used here is a simple Angular application bootstrapping a Standalone Component:

```
// projects/mfe1/src/main.ts

import { environment } from './environments/environment';
import { enableProdMode, importProvidersFrom } from '@angular/core';
import { bootstrapApplication } from '@angular/platform-browser';
import { AppComponent } from './app/app.component';
import { RouterModule } from '@angular/router';
import { MFE1_ROUTES } from './app/mfe1.routes';


if (environment.production) {
  enableProdMode();
}

bootstrapApplication(AppComponent, {
  providers: [
    importProvidersFrom(RouterModule.forRoot(MFE1_ROUTES))
  ]
});
```

When bootstrapping, the application registers its router config `MFE1_ROUTES` via services providers. This router config points to several Standalone Components:

```
// projects/mfe1/src/app/mfe1.routes.ts

import { Routes } from '@angular/router';
import { FlightSearchComponent }
    from './booking/flight-search/flight-search.component';
import { PassengerSearchComponent }
    from './booking/passenger-search/passenger-search.component';
import { HomeComponent } from './home/home.component';

export const MFE1_ROUTES: Routes = [
```

```
    {
        path: '',
        component: HomeComponent,
        pathMatch: 'full'
    },
    {
        path: 'flight-search',
        component: FlightSearchComponent
    },
    {
        path: 'passenger-search',
        component: PassengerSearchComponent
    }
];
```

Here, `importProvidersFrom` bridges the gap between the existing `RouterModule` and the world of Standalone Components. As a replacement for this, future versions of the router will expose a function for setting up the router's providers. According to the underlying CFP, this function will be called `configureRouter`.

The shell used here is just an ordinary Angular application. Using lazy loading, we are going to make it reference the Micro Frontend at runtime.

Activating Module Federation

To get started, let's install the Module Federation plugin and activate Module Federation for the Micro Frontend:

```
npm i @angular-architects/module-federation

ng g @angular-architects/module-federation:init
    --project mfe1 --port 4201 --type remote
```

This command generates a `webpack.config.js`. For our purpose, we have to modify the `exposes` section as follows:

```
const { shareAll, withModuleFederationPlugin } =
    require("@angular-architects/module-federation/webpack");

module.exports = withModuleFederationPlugin({
  name: "mfe1",

  exposes: {
    // Preferred way: expose corse-grained routes
    "./routes": "./projects/mfe1/src/app/mfe1.routes.ts",

    // Technically possible, but not preferred for Micro Frontends:
    // Exposing fine-grained components
    "./Component":
        "./projects/mfe1/src/app/my-tickets/my-tickets.component.ts",
  },

  shared: {
    ...shareAll({
        singleton: true,
        strictVersion: true,
        requiredVersion: "auto"
    }),
  }

});
```

This configuration exposes both, the Micro Frontend's router configuration (pointing to Standalone Components) and a Standalone Component.

Static Shell

Now, let's also activate Module Federation for the shell. In this section, I focus on Static Federation. This means, we are going to map the paths pointing to our Micro Frontends in the webpack.config.js.

> The next section shows how to switch to Dynamic Federation, where we can define the key data for loading a Micro Frontend at runtime.

To enable Module Federation for the shell, let's execute this command:

```
ng g @angular-architects/module-federation:init
    --project shell --port 4200 --type host
```

The `webpack.config.js` generated for the shell needs to point to the Micro Frontend:

```
const { shareAll, withModuleFederationPlugin } =
    require("@angular-architects/module-federation/webpack");

module.exports = withModuleFederationPlugin({

  remotes: {
    "mfe1": "http://localhost:4201/remoteEntry.js",
  },

  shared: {
    ...shareAll({
        singleton: true,
        strictVersion: true,
        requiredVersion: "auto"
    }),
  }

});
```

As we are going with static federation, we also need typings for all configured paths (EcmaScript modules) referencing Micro Frontends:

```
// projects/shell/src/decl.d.ts

declare module 'mfe1/*';
```

Now, all it takes is a lazy route in the shell, pointing to the routes and the Standalone Component exposed by the Micro Frontend:

```
// projects/shell/src/app/app.routes.ts

import { Routes } from '@angular/router';
import { HomeComponent }
    from './home/home.component';
import { NotFoundComponent }
    from './not-found/not-found.component';
import { ProgrammaticLoadingComponent }
    from './programmatic-loading/programmatic-loading.component';

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

    {
      path: 'booking',
      loadChildren: () => import('mfe1/routes').then(m => m.BOOKING_ROUTES)
    },

    {
      path: 'my-tickets',
      loadComponent: () =>
          import('mfe1/Component').then(m => m.MyTicketsComponent)
    },

    [...]

    {
      path: '**',
      component: NotFoundComponent
    }
];
```

Alternative: Dynamic Shell

Now, let's move to dynamic federation. Dynamic Federation means, we don't want to define our remote upfront in the shell's `webpack.config.js`. Hence, let's comment out the remote section there:

```
const { shareAll, withModuleFederationPlugin } =
    require("@angular-architects/module-federation/webpack");

module.exports = withModuleFederationPlugin({

  // remotes: {
  //   "mfe1": "http://localhost:4201/remoteEntry.js",
  // },

  shared: {
    ...shareAll({
        singleton: true,
        strictVersion: true,
        requiredVersion: "auto"
    }),
  }

});
```

Also, in the shell's router config, we need to switch out the dynamic `imports` used before by calls to `loadRemoteModule`:

```
// projects/shell/src/app/app.routes.ts

import { Routes } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { NotFoundComponent } from './not-found/not-found.component';
import { ProgrammaticLoadingComponent }
    from './programmatic-loading/programmatic-loading.component';
import { loadRemoteModule } from '@angular-architects/module-federation';

export const APP_ROUTES: Routes = [
    {
      path: '',
      component: HomeComponent,
      pathMatch: 'full'
    },
    {
      path: 'booking',
      loadChildren: () =>
        loadRemoteModule({
          type: 'module',
          remoteEntry: 'http://localhost:4201/remoteEntry.js',
```

```
          exposedModule: './routes'
        })
        .then(m => m.MFE1_ROUTES)
    },
    {
      path: 'my-tickets',
      loadComponent: () =>
        loadRemoteModule({
          type: 'module',
          remoteEntry: 'http://localhost:4201/remoteEntry.js',
          exposedModule: './Component'
        })
        .then(m => m.MyTicketsComponent)
    },
    [...]
    {
      path: '**',
      component: NotFoundComponent
    }
];
```

The `loadRemoteModule` function takes all the key data, Module Federation needs for loading the remote. This key data is just several strings, hence you can load it from literally everywhere.

Bonus: Programmatic Loading

While most of the times, we will load Micro Frontends (remotes) via the router, we can also load exposed components programmatically. For this, we need a placeholder marked with a template variable for the component in question:

```
<h1>Programmatic Loading</h1>

<div>
    <button (click)="load()">Load!</button>
</div>

<div #placeHolder></div>
```

We get hold of this placeholder's `ViewContainer` via the `ViewChild` decorator:

```
// projects/shell/src/app/programmatic-loading/programmatic-loading.component.ts

import {
    Component,
    OnInit,
    ViewChild,
    ViewContainerRef
} from '@angular/core';

@Component({
  selector: 'app-programmatic-loading',
  standalone: true,
  templateUrl: './programmatic-loading.component.html',
  styleUrls: ['./programmatic-loading.component.css']
})
export class ProgrammaticLoadingComponent implements OnInit {

  @ViewChild('placeHolder', { read: ViewContainerRef })
  viewContainer!: ViewContainerRef;

  constructor() { }

  ngOnInit(): void {
  }

  async load(): Promise<void> {

      const m = await import('mfe1/Component');
      const ref = this.viewContainer.createComponent(m.MyTicketsComponent);
      // const compInstance = ref.instance;
      // compInstance.ngOnInit()
  }

}
```

This example shows a solution for Static Federation. Hence a dynamic `import` is used for getting hold of the Micro Frontend.

After importing the remote component, we can instantiate it using the `ViewContainer`'s `createComponent` method. The returned reference (ref) points to the component instance with it's `instance` property. The instance allows to interact with the component, e. g. to call methods, set property, or setup event handlers.

If we wanted to switch to Dynamic Federation, we would again use `loadRemoteModule` instead of

the dynamic `import`:

```
async load(): Promise<void> {

    const m = await loadRemoteModule({
      type: 'module',
      remoteEntry: 'http://localhost:4201/remoteEntry.js',
      exposedModule: './Component'
    });

    const ref = this.viewContainer.createComponent(m.MyTicketsComponent);
    // const compInstance = ref.instance;
}
```

From Module Federation to esbuild and Native Federation

Beginning with version 17, the Angular CLI uses `esbuild` instead of `webpack` for new projects. As a result, both `ng serve` and `ng build` run noticeably faster.

However, switching to `esbuild` brings a challenge for Micro Frontends: The popular Module Federation comes with webpack and is not available for `esbuild`. To preserve the proven mindset of webpack, we started the Native Federation project. It consequently uses web standards and hence is a solution for the long-term. While it can be used with any bundler, the reference implementation currently delegates to the CLI's esbuild bundler.

It's API surface and configuration files looks like the ones in Module Federation. Hence, everything you read in the previous chapters works the same with Native Federation.

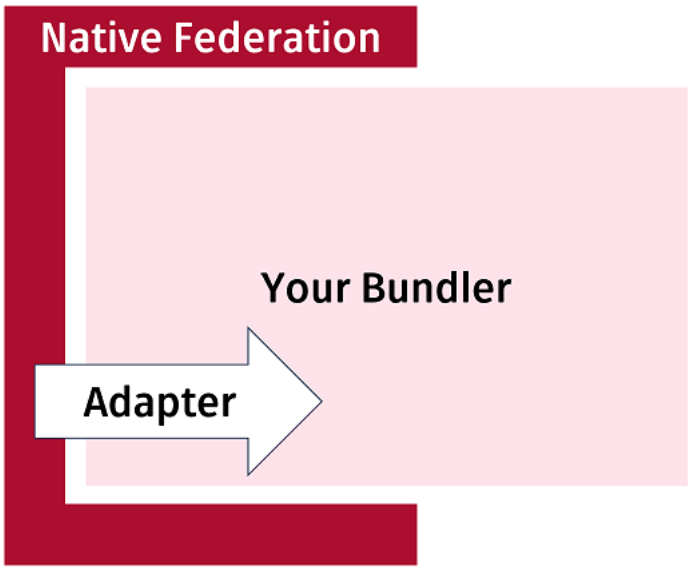
Source Code[74] (see branches `nf-standalone-solution` and `nf-standalone-router-config`)

Native Federation with esbuild

In order to be able to use the proven mental model of Module Federation independently of webpack, the Native Federation[75] project was created. It offers the same options and configuration as Module Federation, but works with all possible build tools. It also uses browser-native technologies such as EcmaScript modules and Import Maps[76]. This measure is intended to ensure long-term support from browsers and also allow alternative implementations.

Native Federation is called before and after the actual bundler in the build process. That's why it doesn't matter which bundler is actually used:

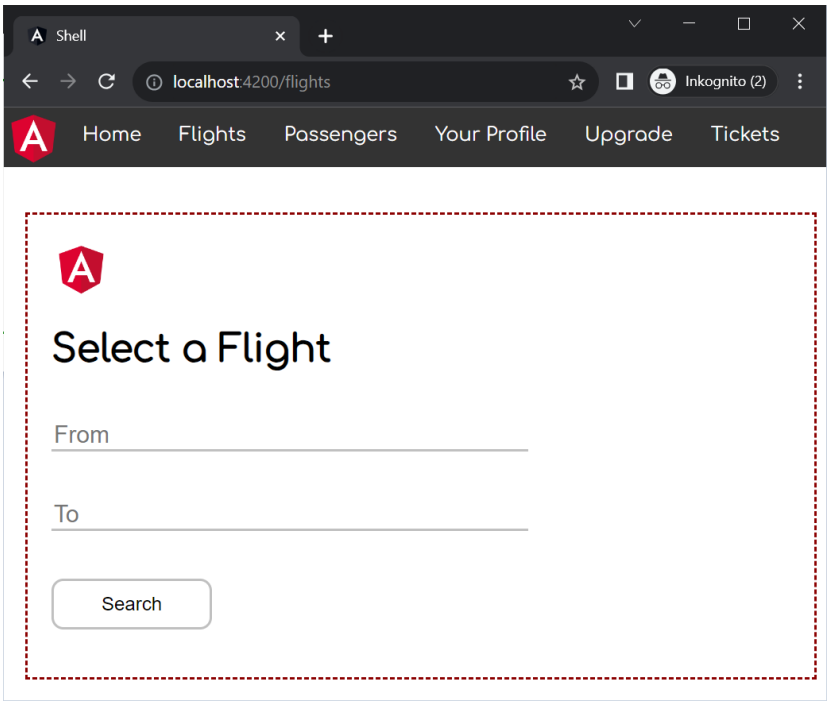
[74] https://github.com/manfredsteyer/module-federation-plugin-example.git
[75] https://www.npmjs.com/package/@angular-architects/native-federation
[76] https://developer.mozilla.org/en-US/docs/Web/HTML/Element/script/type/importmap



Native Federation extends existing build scripts

Since Native Federation also needs to create a few bundles, it delegates to the bundler of choice. The individual bundlers are connected via interchangeable adapters.

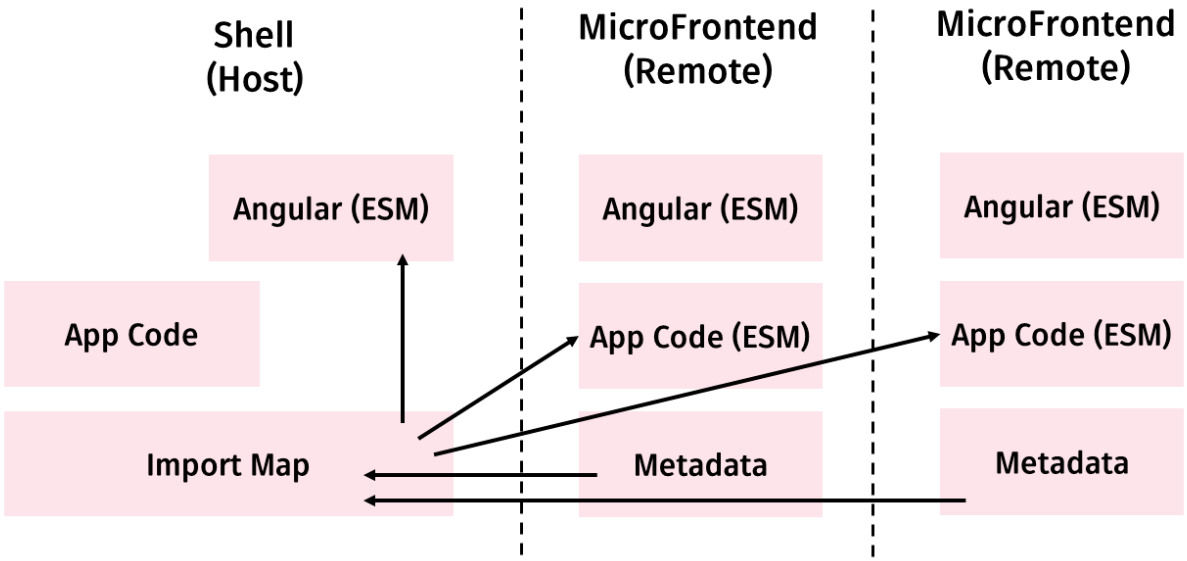
The following image shows an example built with Angular, esbuild, and Native Federation:



Shell with separately deployed micro frontend

The shell shown here has loaded a separately developed and deployed Micro Frontend into its workspace using Native Federation.

Although both the shell and the micro frontend are based on Angular, Native Federation **only loaded Angular once**. To make this possible, Native Federation, following the ideas of Module Federation, places the remotes and the libraries to be shared in their own bundles. For this, it uses standards-compliant EcmaScript bundles that could also be created by other tools. Information about these bundles is placed in metadata files:



Native Federation at runtime

These metadata files are the basis for a standard-compliant Import Map that informs the browser from where which bundles are to be loaded.

Native Federation: Setting up a Micro Frontend

For use with Angular and the CLI, Native Federation offers an `ng-add` schematic. The following statement adds Native Federation to the Angular project `mfe1` and configures it as a `remote` acting as a Micro Frontend:

```
ng add @angular-architects/native-federation --project mfe1 --port 4201 --type remote
```

The `ng-add`-Schematic also creates a `federation.config.js` controlling Native Federation's behavior:

```
const { withNativeFederation, shareAll } =
  require('@angular-architects/native-federation/config');

module.exports = withNativeFederation({

  name: 'mfe1',

  exposes: {
    './Component': './projects/mfe1/src/app/app.component.ts',
  },

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

  skip: [
    'rxjs/ajax',
    'rxjs/fetch',
    'rxjs/testing',
    'rxjs/webSocket',
    // Add further packages you don't need at runtime
  ]

});
```

The property `name` defines a unique name for the remote. The `exposes` section specifies which files the remote should expose to the host. Although these files are built and deployed together with the remote, they can be loaded into the host at runtime. Since the host doesn't care about the full file paths, `exposes` maps them to shorter names.

In the case shown, the remote just publishes its `AppComponent` for simplicity. However, any system component could be published instead, e.g. lazy routing configurations that reference multiple components of a feature.

Under `shared`, the configuration lists all dependencies that the remote wants to share with other remotes and the host. In order to avoid an exhaustive list of all required npm packages, the `shareAll` helper function is used. It includes all packages that are in the `package.json` under `dependencies`. Details about the parameters passed to *shareAll* can be found in one of the previous chapters about Module Federation.

Packages `shareAll` should not share are entered under `skip`. This can improve the build and startup performance of the application slightly. In addition, packages that are intended for use with **NodeJS must be added to `skip`**, since they cannot be compiled for use in the browser.

Native Federation: Setting up a Shell

The host acting as a Micro Frontend Shell can also be set up with `ng add`:

```
ng add @angular-architects/native-federation --project shell --port 4200 --type dyna\
mic-host
```

The type `dynamic-host` indicates that the remotes to be loaded are defined in a configuration file:

```
{
    "mfe1" : "http://localhost:4201/remoteEntry.json"
}
```

This `federation.manifest.json` is generated in the host's `assets` folder by default. By treating it as an asset, the manifest can be exchanged during deployment. The application can therefore be adapted to the respective environment.

The manifest maps the names of the remotes to their metadata, which Native Federation places in the `remoteEntry.json` file during build. Even if `ng add` generates the manifest, it should be checked in order to adjust ports if necessary or to remove applications that are not remotes.

The `ng add`-command also generates a `federation.config.js` for hosts:

```
const { withNativeFederation, shareAll } =
  require('@angular-architects/native-federation/config');

module.exports = withNativeFederation({

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

  skip: [
    'rxjs/ajax',
    'rxjs/fetch',
    'rxjs/testing',
    'rxjs/webSocket',
    // Add further packages you don't need at runtime
  ]

});
```

The `exposes` entry known from the remote's config is not generated for hosts because hosts typically do not publish files for other hosts. However, if you want to set up a host that also acts as a remote for other hosts, there is nothing wrong with adding this entry.

The `main.ts` file, also modified by `ng add`, initializes Native Federation using the manifest:

```
import { initFederation } from '@angular-architects/native-federation';

initFederation('/assets/federation.manifest.json')
  .catch(err => console.error(err))
  .then(_ => import('./bootstrap'))
  .catch(err => console.error(err));
```

The `initFederation` function reads the metadata of each remote and generates an Import Map used by the browser to load shared packages and exposed modules. The program flow then delegates to the `bootstrap.ts`, which starts the Angular solution with the usual instructions (`bootstrapApplication` or `bootstrapModule`).

All files considered so far were set up using `ng add`. In order to load a program part published by a remote, the host must be expanded to include lazy loading:

```
[…]
import { loadRemoteModule } from '@angular-architects/native-federation';

export const APP_ROUTES: Routes = [
  […],
  {
    path: 'flights',
    loadComponent: () =>
      loadRemoteModule('mfe1', './Component').then((m) => m.AppComponent),
  },
  […]
];
```

The lazy route uses the `loadRemoteModule` helper function to load the `AppComponent` from the remote. It takes the name of the remote from the manifest (`mfe1`) and the name under which the remote publishes the desired file (`./Component`).

Exposing a Router Config

Just exposing one component via Native Federation is a bit fine-grained. Quite often, you want to expose a whole feature that consists of several components. Fortunately, we can expose all kinds of TypeScript/EcmaScript constructs. In the case of coarse-grained features, we could expose an NgModule with subroutes or – if we go with Standalone Components – just a routing config. Here, the latter one is the case:

```
import { Routes } from "@angular/router";
import { FlightComponent } from "./flight/flight.component";
import { HolidayPackagesComponent }
  from "./holiday-packages/holiday-packages.component";

export const APP_ROUTES: Routes = [
    {
        path: '',
        redirectTo: 'flights',
        pathMatch: 'full'
    },
    {
        path: 'flight-search',
        component: FlightComponent
    },
    {
        path: 'holiday-packages',
        component: HolidayPackagesComponent
    }
];
```

This routing config needs to be added to the `exposes` section in the Micro Frontend's `federation.config.js`:

```
const { withNativeFederation, shareAll } =
  require('@angular-architects/native-federation/config');

module.exports = withNativeFederation({

  name: 'mfe1',

  exposes: {
    './Component': './projects/mfe1/src/app/app.component.ts',

     // Add this line:
    './routes': '././projects/mfe1/src/app/app.routes.ts',
  },

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

  skip: [
    'rxjs/ajax',
```

```
    'rxjs/fetch',
    'rxjs/testing',
    'rxjs/webSocket',
    // Add further packages you don't need at runtime
  ]

});
```

In the shell, you can directly route to this routing configuration:

```
[...]
import { loadRemoteModule } from '@angular-architects/native-federation';

export const APP_ROUTES: Routes = [
  [...]

  {
    path: 'flights',
    // loadChildreas instead of loadComponent !!!
    loadChildren: () =>
      loadRemoteModule('mfe1', './routes').then((m) => m.APP_ROUTES),
  },

  [...]
];
```

Also, we need to adjust the routes in the shell's navigation:

```
<ul>
    <li><img src="../assets/angular.png" width="50"></li>
    <li><a routerLink="/">Home</a></li>
    <li><a routerLink="/flights/flight-search">Flights</a></li>
    <li><a routerLink="/flights/holiday-packages">Holidays</a></li>
</ul>

<router-outlet></router-outlet>
```

Communication between Micro Frontends

Communication between Micro Frontends can also be enabled via shared libraries. I'd like to say in advance that this option should only be used with caution: Micro Frontend architectures are intended

to decouple individual frontends from each other. However, if a frontend expects information from other frontends, exactly the opposite is achieved. Most solutions I've seen only share a handful of contextual information. Examples include the current username, the current client and a few global filters.

To share information, we first need a shared library. This library can be a separately developed npm package or a library within the current Angular project. The latter can be generated with

```
ng g lib auth
```

The name of the library in this case is set as `auth`. To share data, this library receives a stateful service. For the sake of brevity, I'm using the simplest stateful service I can think about:

```
@Injectable({
  providedIn: 'root'
})
export class AuthService {
  userName = '';
}
```

In this very simple scenario, the service is used as a black board: A Micro Frontend writes information into the service and another one reads the information. However, a somewhat more convenient way to share information would be to use a publish/subscribe mechanism through which interested parties can be informed about value changes. This idea can be realized, for example, by using RxJS subjects.

If Monorepo-internal libraries are used, they should be made accessible via path mapping in the `tsconfig.json`:

```
"compilerOptions": {
    "paths": {
      "@demo/auth": [
        "projects/auth/src/public-api.ts"
      ]
     },
     […]
}
```

Please note that I'm pointing to `public-api.ts` in the **lib's source code**. This strategy is also used by Nx, but the CLI points to the `dist` folder by default. Hence, in the latter case, you need to update this entry by hand.

It must also be ensured that all communication partners use the same path mapping.

Conclusion

The new esbuild builder provides tremendous improvements in build performance. However, the popular Module Federation is currently bound to webpack. Native Federation provides the same mental model and is implemented in a tooling-agnostic way. Hence, it works with all possible bundlers. Also, it uses web standards like EcmaScript modules and Import Maps. This also allows for different implementation and makes it a reliable solution in the long run.

The new NGRX Signal Store for Angular: 3 + n Flavors

Most Angular applications need to preserve some state so that the same data doesn't need to fetched time and again from the backend. Examples are information that are needed when switching back from a details view to a list view or information collected during clicking through a wizard.

So far, the default state management solution in the Angular world has been the Redux-based NGRX Store. Since the advent of Signals in Angular, the NGRX team has been working on a new store that leverages this new reactive building block. Compared to the traditional NGRX Store, the Signal Store is lightweight, easy to use, and highly extensible.

This chapter shows how to incorporate it in your application. For this, it shows up `3+1` different flavors of using it.

Source Code[77]

Getting the Package

To install the Signal Store, you just need to add the package `@ngrx/signals` to your application:

```
1 npm i @ngrx/signals
```

Flavor 1: Lightweight with signalState

Branch: `arc-signal-store`

A very lightweight way of managing Signals with the Signal Store is its `signalState` function (not to be confused with the `signalStore` function). It creates a simple container for managing the passed state using Signals. This container is represented by the type `SignalState`:

[77] https://github.com/manfredsteyer/standalone-example-cli

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

import { signalState } from '@ngrx/signals';

[...]

export class FlightBookingFacade {

    [...]

    private state = signalState({
        from: 'Paris',
        to: 'London',
        preferences: {
          directConnection: false,
          maxPrice: 350,
        },
        flights: [] as Flight[],
        basket: {} as Record<number, boolean>,
    });

    // fetch read-only signals
    flights = this.state.flights;
    from = this.state.from;
    to = this.state.to;
    basket = this.state.basket;

  [...]
}
```

Each top-level state property gets its own Signal. These properties are retrieved as read-only Signals, ensuring a separation between reading and writing: Consumers using the Signals can just read their values. For updating the state, the service encapsulating the state provides methods (see below). This ensures that the state can only be updated in a well-defined manner.

Also, nested objects like the one provided by the `preferences` property above result in nested signals. Hence, one can retrieve the whole `preferences` object as a Signal but also its properties:

```
const ps = this.state.preferences();
const direct = this.state.preferences.directConnection();
```

Currently, this isn't implemented for arrays, as Angular's envisioned Signal Components will solve this use case by creating a Signal for each iterated item.

Selecting and Computing Signals

As the Signal Store provides the state as Signals, we can directly use Angular's `computed` function:

```
selected = computed(() =>
  this.flights().filter((f) => this.basket()[f.id])
);
```

Here, `computed` serves the same purpose as Selectors in the Redux-based NGRX Store: It enables us to calculate different state representations for different use cases. These so-called View Models are only recomputed when at least one of the underlying signals changes.

Updating State

For updating the `SignalState`, Signal Store provides us with a `patchState` function:

```
import { patchState } from '@ngrx/signals';

[...]

updateCriteria(from: string, to: string): void {
  patchState(this.state, { from, to })
}
```

Here, we pass in the state container and a partial state. As an alternative, one can pass a function taking the current state and transforming it to the new state:

```
updateBasket(id: number, selected: boolean): void {
  patchState(this.state, state => ({
    basket: {
      ...state.basket,
      [id]: selected,
    },
  }));
}
```

Side Effects

Besides updating the state, methods can also trigger side effects like loading and saving objects:

```
async load() {
  if (!this.from() || !this.to()) return;

  const flights = await this.flightService.findPromise(
    this.from(),
    this.to()
  );

  patchState(this.state, { flights });
}
```

Decoupling Intention from Execution

Sometimes, the caller of `patchState` only knows that some state needs to be updated without knowing where it's located. For such cases, you can provide Updaters. Updaters are just functions taking a current state and returning an updated version of it:

```
type BasketSlice = { basket: Record<number, boolean> };
type BasketUpdateter = (state: BasketSlice) => BasketSlice;

export function updateBasket(flightId: number, selected: boolean): BasketUpdateter {
  return (state) => ({
    ...state,
    basket: {
      ...state.basket,
      [flightId]: selected,
    },
  });
}
```

It's also fine to just return a partial state. It will be patched over the current state:

```
type BasketSlice = { basket: Record<number, boolean> };
type BasketUpdateter = (state: BasketSlice) => BasketSlice;

export function updateBasket(flightId: number, selected: boolean): BasketUpdateter {
  return (state) => ({
    basket: {
      ...state.basket,
      [flightId]: selected,
    },
  });
}
```

If you don't need to project the current state, just returning a partial state is fine too. In this case, you can skip the inner function:

```
export function updateFlights(flights: Flight[]) {
  return { flights };
}
```

Updater can be defined in the `Store`'s (`signalState`'s) "sovereign territory". For the consumer, it is just a black box:

```
patchState(updateBasket(id, selected))
```

Passing an Updater to `patchState` expresses an intention. This is similar to dispatching an Action in the classic NGRX store. However, other than with Redux, no eventing is involved, and we cannot prevent the caller from directly passing their own Updaters. For the latter reason, I'm hiding the `SignalStore` behind a facade.

Flavor 2: Powerful with signalStore

Branch: `arc-signal-store-2`

Similar to `signalState`, the `signalStore` function creates a container managing state with Signals. However, now, this container is a fully-fledged Store that not only comes with state Signals but also with computed Signals as well as methods for updating the state and for triggering side effects. Hence, there is less need for crafting a facade by hand, as shown above.

Technically, the Store is an Angular service that is composed of several pre-existing features:

```
export const FlightBookingStore = signalStore(
  { providedIn: 'root' },
  withState({
    from: 'Paris',
    to: 'London',
    initialized: false,
    flights: [] as Flight[],
    basket: {} as Record<number, boolean>,
  }),

  // Activating further features
  withComputed([...]),
  withMethods([...]),
  withHooks([...]),
)
```

In this case, the service is also registered in the root scope. When skipping `{ providedIn: 'root' }`, one needs to register the service by hand, e. g., by providing it when bootstrapping the application, within a router configuration, or on component level.

Selecting and Computing Signals

The `withComputed` feature takes the store with its state Signals and defines an object with calculated signals:

```
withComputed((store) => ({
  selected: computed(() => store.flights().filter((f) => store.basket()[f.id])),
  criteria: computed(() => ({ from: store.from(), to: store.to() })),
})),
```

The returned computed signals become part of the store. A more compact version might involve directly destructuring the passed store:

```
withComputed(({ flights, basket, from, to }) => ({
  selected: selectSignal(() => flights().filter((f) => basket()[f.id])),
  criteria: selectSignal(() => ({ from: from(), to: to() })),
})),
```

Methods for Updating State and Side Effects

Similar to `withComputed`, `withMethods` also takes the store and returns an object with methods:

```
withMethods((state) => {
  const { basket, flights, from, to, initialized } = state;
  const flightService = inject(FlightService);

  return {
    updateCriteria: (from: string, to: string) => {
      patchState(state, { from, to });
    },
    updateBasket: (flightId: number, selected: boolean) => {
      patchState(state, {
        basket: {
          ...basket(),
          [flightId]: selected,
        },
      });
```

```
    },
    delay: () => {
      const currentFlights = flights();
      const flight = currentFlights[0];

      const date = addMinutes(flight.date, 15);
      const updFlight = { ...flight, date };
      const updFlights = [updFlight, ...currentFlights.slice(1)];

      patchState(state, { flights: updFlights });
    },
    load: async () => {
      if (!from() || !to()) return;
      const flights = await flightService.findPromise(from(), to());
      patchState(state, { flights });
    }
  };
}),
```

`withMethods` runs in an injection context and hence can use `inject` to get hold of services. After `withMethods` was executed, the retrieved methods are added to the store.

Consuming the Store

From the caller's perspective, the store looks a lot like the facade shown above. We can inject it into a consuming component:

```
@Component([...])
export class FlightSearchComponent {
  private store = inject(FlightBookingStore);

  from = this.store.from;
  to = this.store.to;
  basket = this.store.basket;
  flights = this.store.flights;
  selected = this.store.selected;

  async search() {
    this.store.load();
  }

  delay(): void {
```

```
16      this.store.delay();
17    }
18
19    updateCriteria(from: string, to: string): void {
20      this.store.updateCriteria(from, to);
21    }
22
23    updateBasket(id: number, selected: boolean): void {
24      this.store.updateBasket(id, selected);
25    }
26  }
```

Hooks

The function `withHooks` provides another feature allowing to setup lifecycle hooks to run when the store is initialized or destroyed:

```
withHooks({
  onInit({ load }) {
    load()
  },
  onDestroy({ flights }) {
    console.log('flights are destroyed now', flights());
  },
}),
```

Both hooks get the store passed. One more time, by using destructuring, you can focus on a subset of the stores members.

rxMethod

Branch: `arc-signal-store-rx`

While Signals are easy to use, they are not a full replacement for RxJS. For leveraging RxJS and its powerful operators, the Signal Store provides a secondary entry point `@ngrx/signals/rxjs-interop`, containing a function `rxMethod<T>`. It allows working with an Observable representing side-effects that automatically run when specific values change:

```
import { rxMethod } from '@ngrx/signals/rxjs-interop';

[...]


withMethods(({ $update, basket, flights, from, to, initialized }) => {
  const flightService = inject(FlightService);

  return {
    [...]
    connectCriteria: rxMethod<Criteria>((c$) => c$.pipe(
      filter(c => c.from.length >= 3 && c.to.length >= 3),
      debounceTime(300),
      switchMap((c) => flightService.find(c.from, c.to)),
      tap(flights => patchState(state, { flights }))
    ))
  }
});
```

The type parameter `T` defines the type the `rxMethod` works on. While the `rxMethod` receives an `Observable<T>`, the caller can also pass an `Observable<T>`, a `Signal<T>`, or `T` directly. In the latter two cases, the passed values are converted into an Observable.

After defining the `rxMethod`, somewhere else in the application, e. g. in a hook or a regular method, you can call this effect:

```
withHooks({
  onInit({ loadBy, criteria }) {
    connectCriteria(criteria);
  },
})
```

Here, the criteria Signal – a computed signal – is passed. Every time this Signal changes, the effect within `connectCriteria` is re-executed.

Custom Features - n Further Flavors

Branch: `arc-signal-store-custom`

Besides configuring the Store with baked-in features, everyone can write their own features to automate repeating tasks. The playground provided by Marko Stanimirović[78], the NGRX contributor behind the Signal Store, contains several examples of such features.

[78] https://twitter.com/MarkoStDev

One of the examples found in this repository is a CallState feature[79] defining a state property informing about the state of the current HTTP call:

```
export type CallState = 'init' | 'loading' | 'loaded' | { error: string };
```

In this section, I'm using this example to explain how to provide custom features.

Defining Custom Features

A feature is usually created by calling `signalStoreFeature`. This function constructs a new feature on top of existing ones.

```
// Taken from: https://github.com/markostanimirovic/ngrx-signal-store-playground/blo\
b/main/src/app/shared/call-state.feature.ts

import { computed } from '@angular/core';
import {
  signalStoreFeature,
  withComputed,
  withState,
} from '@ngrx/signals';

export type CallState = 'init' | 'loading' | 'loaded' | { error: string };

export function withCallState() {
  return signalStoreFeature(
    withState<{ callState: CallState }>({ callState: 'init' }),
    withComputed(({ callState }) => ({
      loading: computed(() => callState() === 'loading'),
      loaded: computed(() => callState() === 'loaded'),
      error: computed(() => {
        const state = callState();
        return typeof state === 'object' ? state.error : null
      }),
    }))
  );
}
```

For the state properties added by the feature, one can provide `Updaters`:

[79] https://github.com/markostanimirovic/ngrx-signal-store-playground/blob/main/src/app/shared/call-state.feature.ts

```
export function setLoading(): { callState: CallState } {
  return { callState: 'loading' };
}

export function setLoaded(): { callState: CallState } {
  return { callState: 'loaded' };
}

export function setError(error: string): { callState: CallState } {
  return { callState: { error } };
}
```

Updaters allows the consumer to modify the feature state without actually knowing how it's structured.

Using Custom Features

For using Custom Features, just call the provided factory when setting up the store:

```
export const FlightBookingStore = signalStore(
  { providedIn: 'root' },
  withState({ [...] }),

  // Add feature:
  withCallState(),
  [...]

  withMethods([...])
  [...]
);
```

The provided properties, methods, and Updaters can be used in the Store's methods:

```
load: async () => {
  if (!from() || !to()) return;

  // Setting the callState via an Updater
  patchState(state, setLoading());

  const flights = await flightService.findPromise(from(), to());
  patchState(state, { flights });

  // Setting the callState via an Updater
  patchState(state, setLoaded());
},
```

The consumer of the store sees the properties provided by the feature too:

```
private store = inject(FlightBookingStore);

flights = this.store.flightEntities;
loading = this.store.loading;
```

As each feature is transforming the Store's properties and methods, make sure to call them in the right order. If we assume that methods registered with `withMethods` use the `CallState`, `withCallState` has to be called before `withMethods`.

Flavor 3: Built-in Features like Entity Management

Branch: `arc-signal-store-entities`

The NGRX Signal Store already comes with a convenient extension for managing entities. It can be found in the secondary entry point `@ngrx/signals/entities` and provides data structures for entities but also several Updaters, e. g., for inserting entities or for updating a single entity by id.

To setup entity management, just call the `withEntities` function:

```
import { withEntities } from '@ngrx/signals/entities';

const BooksStore = signalStore(
  [...]

  // Defining an Entity
  withEntities({ entity: type<Flight>(), collection: 'flight' }),

  // withEntities created a flightEntities signal for us:
  withComputed(({ flightEntities, basket, from, to }) => ({
    selected: computed(() => flightEntities().filter((f) => basket()[f.id])),
    criteria: computed(() => ({ from: from(), to: to() })),
  })),

  withMethods((state) => {
    const { basket, flightEntities, from, to, initialized } = state;
    const flightService = inject(FlightService);

    return {
      [...],

      load: async () => {
        if (!from() || !to()) return;
        patchState(state, setLoading());

        const flights = await flightService.findPromise(from(), to());

        // Updating entities with out-of-the-box setAllEntities Updater
        patchState(state, setAllEntities(flights, { collection: 'flight' }));
        patchState(state, setLoaded());
      },

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

The passed collection name prevents naming conflicts. In our case, the collection is called `flight`, and hence the feature creates several properties beginning with `flight`, e.g., `flightEntities`.

There is quite an amount of ready-to-use Updaters:

- `addEntity`

- addEntities
- removeEntity
- removeEntities
- removeAllEntities
- setEntity
- setEntities
- setAllEntities
- updateEntity
- updateEntities
- updateAllEntities

Similar to `@ngrx/entities`, internally, the entities are stored in a normalized way. That means they are stored in a dictionary, mapping their primary keys to the entity objects. This makes it easier to join them together to View Models needed for specific use cases.

As we call our collection `flight`, `withEntities` creates a Signal `flightEntityMap` mapping flight ids to our flight objects. Also, it creates a Signal `flightIds` containing all the ids in the order. Both are used by the also added computed signal `flightEntities` used above. It returns all the flights as an array respecting the order of the ids within `flightIds`. Hence, if you want to rearrange the positions of our flights, just update the `flightIds` property accordingly.

For building the structures like the `flightEntityMap`, the Updaters need to know how the entity's id is called. By default, it assumes a property `id`. If the id is called differently, you can tell the Updater by using the `idKey` property:

```
patchState(
  state,
  setAllEntities(flights, {
    collection: 'flight', idKey: 'flightId' }));
```

The passed property needs to be a `string` or `number`. If it's of a different data type or if it doesn't exist at all, you get a compilation error.

Conclusion

The upcoming NGRX Signal Store allows managing state using Signals. The most lightweight option for using this library is just to go with a `SignalState` container. This data structure provides a Signal for each state property. These signals are read-only. For updating the state, you can use the `patchState` function. To make sure updates only happen in a well-defined way, the `signalState` can be hidden behind a facade.

The `SignalStore` is more powerful and allows to register optional features. They define the state to manage but also methods operating on it. A `SignalStore` can be provided as a service and injected into its consumers.

The `SignalStore` also provides an extension mechanism for implementing custom features to ease repeating tasks. Out of the box, the Signal Store comes with a pretty handy feature for managing entities.

Smarter, Not Harder: Simplifying your Application With NGRX Signal Store and Custom Features

What would you say if you could implement a Signal Store for a (repeating) **CRUD use case** including **Undo/Redo** in **just 7 (!) lines of code?**

To make this possible, we need some custom features for the Signal Store. In this chapter, I show how this all works.

As always, my work is highly inspired by the implementation of the NGRX Signal and the examples provided by Marko Stanimirović[80], the NGRX core team member who envisioned and implemented the Signal Store.

Source Code[81] (Branch: arc-signal-store-custom-examples)

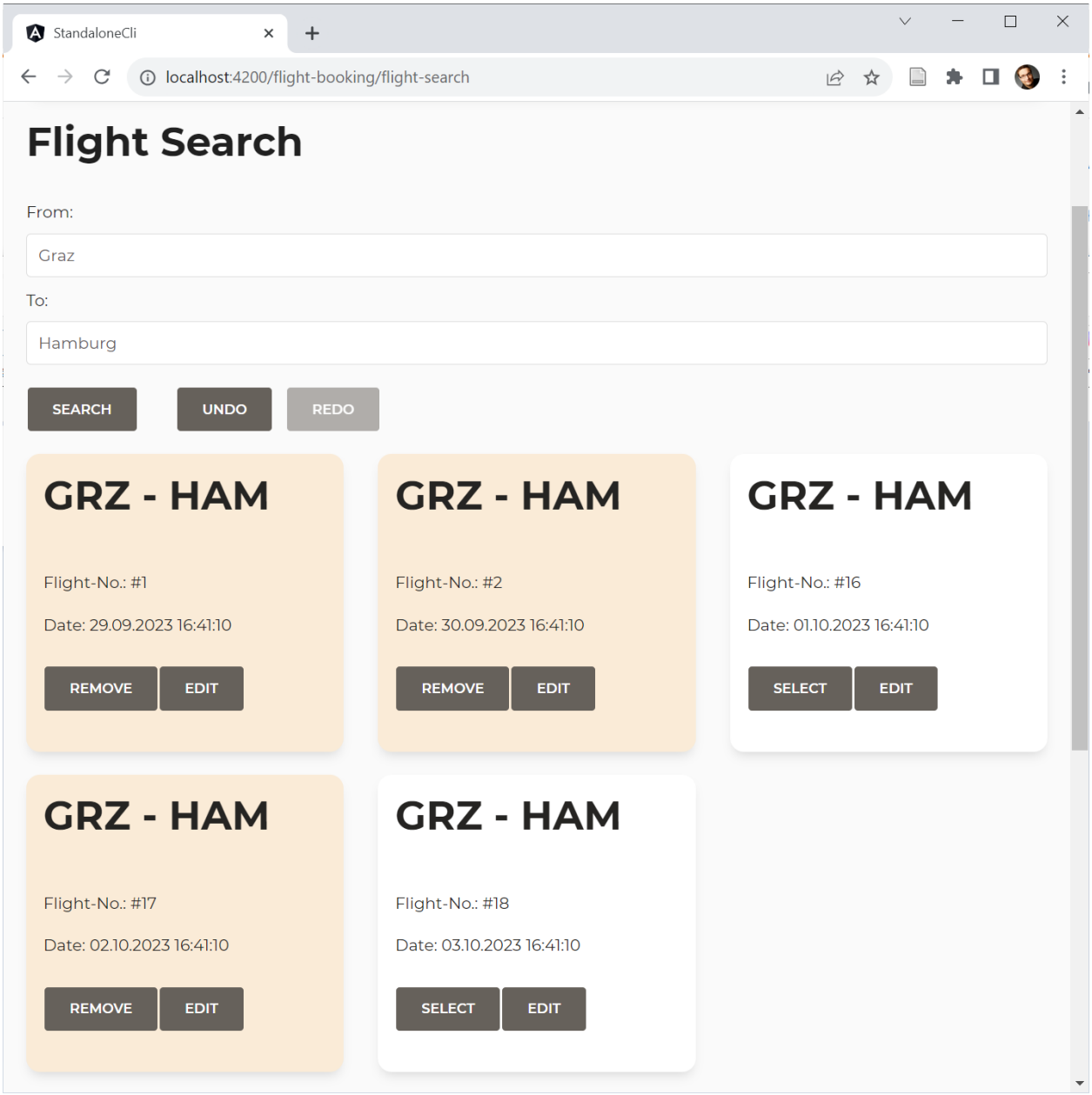
Goal

The goal of this chapter is to show how to implement custom features for the Signal Store that allow for the following:

- Searching for entities
- Selecting several entities
- Displaying the selected entities
- Undo/Redo

This is how the demo application I've built on top of these custom features looks like:

[80] https://twitter.com/MarkoStDev
[81] https://github.com/manfredsteyer/standalone-example-cli/tree/arc-signal-store-custom-examples



Demo Application

And this is the whole code we need to set up the store, including Undo/Redo and connecting it to a Data Service fetching the entities from the backend:

```
export const FlightBookingStore = signalStore(
  { providedIn: 'root' },
  withEntities<Flight>(),
  withCallState(),
  withDataService(FlightService, { from: 'Graz', to: 'Hamburg'} ),
  withUndoRedo(),
);
```

As you can see, I'm using the `@ngrx/signals/entities` package for managing entities. Besides this, I moved the remaining logic into three reusable custom features: `withCallState` was already discussed in a previous chapter. This chapter discusses `withDataService` and provides the code for `withUndoRedo`.

DataService Custom Feature

The idea behind the `DataService` feature is to set up state for a search filter and to connect an Angular Service that uses this filter to search for entities. In a further development stage, the feature could also call the `DataService` for saving and deleting entities. However, as these implementations would not add additional insights here, I decided to skip them for the sake of brevity.

To make the `DataService` feature generic, we need some general types describing everything the feature interacts with:

```
import { EntityId } from "@ngrx/signals/entities";
[...]

export type Filter = Record<string, unknown>;
export type Entity = { id: EntityId };

export interface DataService<E extends Entity, F extends Filter> {
    load(filter: F): Promise<E[]>;
}
```

These types describe how our search filter is structured, what we mean when referring to an entity, and how a `DataService` should look like. The type `EntityId` comes from `@ngrx/signals/entities` and accepts a `string` or a `number`.

Expecting that an entity is an arbitrary object with an `id` property is one of the conventions `@ngrx/signals/entities` provides for shorten your code. If your primary key is called otherwise, you can tell `@ngrx/signals/entities` accordingly. However, to keep the presented example small, I've decided to stick with this convention.

Implementing A Generic Custom Feature

The function `withDataService` returns the `DataService` feature:

```
export function withDataService<E extends Entity, F extends Filter, S extends DataSe\
rvice<E, F>>(dataServiceType: Type<S>, filter: F) {
    [...]
}
```

Its type parameter describes the Entity to manage, the corresponding search filter, and the `DataService`. When calling this generic method we just need to pass in the `DataService` and an initial filter. TypeScript infers the rest:

```
withDataService(FlightService, { from: 'Graz', to: 'Hamburg'} ),
```

The `withDataService` function calls `signalStoreFeature` to setup our custom feature:

```
export function withDataService<E extends Entity, F extends Filter, S extends DataSe\
rvice<E, F>>(dataServiceType: Type<S>, filter: F) {
    return signalStoreFeature(
        // Our expectations to the store:
        {
            state: type<{
                callState: CallState,
                entityMap: Record<EntityId, E>,
                ids: EntityId[]
            }>(),
            signals: type<{
                entities: Signal<Entity[]>
            }>(),
            methods: type<{}>()
        },

        // Composing several features:
        withState( [...] ),
        withComputed( [...] ),
        withMethods( [...] )
    );
}
```

As shown in the previous chapter, the `signalStoreFeature` function basically composes existing features into a new one. For instance, we can introduce new state properties with `withState`, computed Signals with `withComputed`, or methods with `withMethods`.

However, one little thing is a bit different this time: Our feature has some **expectations** for the Signal Store it is used with. It expects the `callState` feature and the `entity` feature to be in place. The former one sets up a `callState` property we need; the latter one sets up an `entityMap` and an `ids` property as well as a calculated Signal `entities`.

These expectations are defined by the first parameter passed to `signalStoreFeature`. It describes the expected state properties (`state`), computed signals (`signals`), and methods. As we don't expect any methods, we can also omit the key `methods` instead of pointing to `type<{}>()`.

To avoid naming conflicts, the `entity` feature allows using different property names. To keep things simple, I'm sticking with the default names here. However, in a following chapter, you learn how to deal with dynamic property names in a type-safe way.

The remaining parts of this custom feature are just about adding state properties, computed Signals, and methods on top of the expected features:

```
export function withDataService<E extends Entity, F extends Filter, S extends DataSe\
rvice<E, F>>(dataServiceType: Type<S>, filter: F) {
    return signalStoreFeature(
        // First parameter contains
        // Our expectations to the store:
        // If they are not fulfilled, TypeScript
        // will prevent adding this feature!
        {
            state: type<{
                callState: CallState,
                entityMap: Record<EntityId, E>,
                ids: EntityId[]
            }>(),
            signals: type<{
                entities: Signal<Entity[]>
            }>(),
            methods: type<{}>()
        },
        withState({
            filter,
            selectedIds: {} as Record<EntityId, boolean>,
        }),
        withComputed(({ selectedIds, entities }) => ({
            selectedEntities: computed(() => entities().filter(e => selectedIds()[e.\
id]))
        })),
        withMethods((store) => {
            const dataService = inject(dataServiceType)
            return {
```

```
                updateFilter(filter: F): void {
                    patchState(store, { filter });
                },
                updateSelected(id: EntityId, selected: boolean): void {
                    patchState(store, ({ selectedIds }) => ({
                        selectedIds: {
                            ...selectedIds,
                            [id]: selected,
                        }
                    }));
                },
                async load(): Promise<void> {
                    patchState(store, setLoading());
                    const result = await dataService.load(store.filter());
                    patchState(store, setAllEntities(result));
                    patchState(store, setLoaded());
                }
            };
        })
    );
}
```

Providing a Fitting Data Service

To make our data services work with our custom feature, they need to implement the above-mentioned `DataService` interface that is to be typed with the Entity in question and a search filter expected by the `load` method:

```
export type FlightFilter = {
    from: string;
    to: string;
}

@Injectable({
  providedIn: 'root'
})
export class FlightService implements DataService<Flight, FlightFilter> {
  baseUrl = `https://demo.angulararchitects.io/api`;

  constructor(private http: HttpClient) {}

```

```
  load(filter: FlightFilter): Promise<Flight[]> {
    [...]
  }

  [...]
}
```

Undo/Redo-Feature

The Undo/Redo feature is implemented in a very similar way. Internally, it managed two stacks: an undo stack and a redo stack. The stacks are basically arrays with `StackItems`:

```
export type StackItem = {
    filter: Filter;
    entityMap: Record<EntityId, Entity>,
    ids: EntityId[]
};
```

Each `StackItem` represents a snapshot of the current search filter and the information the entity feature uses (`entityMap`, `ids`).

For configuring the feature, a `UndoRedoOptions` type is used:

```
export type UndoRedoOptions = {
    maxStackSize: number;
}

export const defaultUndoRedoOptions: UndoRedoOptions = {
    maxStackSize: 100
}
```

The options object allows us to limit the stack size. Older items are removed according to the First In, First Out rule if the stack grows to large.

The `withUndoRedo` function adds the feature. It is structured as follows:

```
export function withUndoRedo(options = defaultUndoRedoOptions) {

    let previous: StackItem | null = null;
    let skipOnce = false;

    const undoStack: StackItem[] = [];
    const redoStack: StackItem[] = [];

    [...]

    return signalStoreFeature(
        // Expectations to the store:
        {
            state: type<{
                filter: Filter,
                entityMap: Record<EntityId, Entity>,
                ids: EntityId[]
            }>(),
        },
        [...]
        withMethods((store) => ({
            undo(): void { [...] },
            redo(): void { [...] }
        })),
        withHooks({
            onInit(store) {
                effect(() => {
                    const filter = store.filter();
                    const entityMap = store.entityMap();
                    const ids = store.ids();

                    [...]
                });
            }
        })

    )
}
```

Similar to the `withDataService` function discussed above, it calls `signalStoreFeature` and defines its expectations for the store using the first argument. It introduces an `undo` and a `redo` method, restoring the state from the respective stacks. To observe the state, the `onInit` hook at the end creates an effect. After each change, this effect stores the original state on the undo stack.

One thing is a bit special about this implementation of the Undo/Redo feature: The feature itself holds some internal state – like the `undoStack` and the `redoStack` – that is not part of the Signal Store.

Please find the full implementation of this feature in my GitHub repository[82] (Branch: arc-signal-store-custom-examples). If you want to see a different implementation that also stores the feature-internal state in the Signal Store, please look at the `arc-signal-custom-examples-undoredo-alternative` branch.

Using the Store in a Component

To use our 7-lines-of-code-signal-store in a component, just inject it and delegate to its signals and methods:

```
@Component( [...] )
export class FlightSearchComponent {
  private store = inject(FlightBookingStore);

  // Delegate to signals
  from = this.store.filter.from;
  to = this.store.filter.to;
  flights = this.store.entities;
  selected = this.store.selectedEntities;
  selectedIds = this.store.selectedIds;

  // Delegate to methods
  async search() {
    this.store.load();
  }

  undo(): void {
    this.store.undo();
  }

  redo(): void {
    this.store.redo();
  }

  updateCriteria(from: string, to: string): void {
    this.store.updateFilter({ from, to });
  }
```

[82] https://github.com/manfredsteyer/standalone-example-cli/tree/arc-signal-store-custom-examples

```
28
29    updateBasket(id: number, selected: boolean): void {
30      this.store.updateSelected(id, selected);
31    }
32
33  }
```

Conclusion and Outlook

Implementing repeating tasks with generic custom features allows you to shrink down your source code dramatically. In this chapter, we implemented a Signal Store for a simple use case with just 7 lines of code. While implementing such features in a generic way adds some overhead in the first place, this effort pays off for sure once you have several use cases structured that way.

To reuse existing custom features, our custom feature delegates to them. The API provided by the NGRX Signal Store allows the custom feature to ensure the other features have been configured. It defines which state properties, computed signals, and methods it expects. If they are not present, TypeScript brings up a compilation error.

For the sake of simplicity, we just went with the default property names introduced by the orchestrated features. However, to avoid naming conflicts, it is also possible to configure these names. For instance, the `entity` feature that ships with the Signal Store supports such dynamic properties without compromising type safety. In the next chapter, I show how to use this idea for our custom features, too.

NGRX Signal Store Deep Dive: Flexible and Type-Safe Custom Extensions

The NGRX Signal Store, released shortly after Angular 17, offers a very lightweight solution for state management. With no direct dependencies on RxJS, it relies entirely on Signals. However, its greatest strength is undoubtedly its high degree of expandability. With so-called custom features, recurring tasks can be implemented very easily in a central way.

The first example of custom features presented here is very straightforward. After that, things get a little more challenging: The consumer of a feature must be able to determine the names of the signals and methods set up by the feature. Otherwise, naming conflicts will arise quickly. As the following examples show, this does not contradict strict typing in TypeScript.

The examples shown here are inspired by an example from Marko Stanimirović[83], the NGRX core team member behind the Signal Store, and the Entity management solution `@ngrx/signals/entity` shipped with the Signal Store.

Source Code[84] (Branch: arc-signal-store-custom-typed)

A Simple First Extension

Let's use the `CallState` feature from the last chapter as our starting point:

```
import {
  SignalStoreFeature,
  signalStoreFeature,
  withComputed,
  withState,
} from '@ngrx/signals';

[…]

export type CallState = 'init' | 'loading' | 'loaded' | { error: string };

export function withCallState() {
  return signalStoreFeature(
    withState<{ callState: CallState }>({ callState: 'init' }),
```

[83] https://twitter.com/MarkoStDev
[84] https://github.com/manfredsteyer/standalone-example-cli/tree/arc-signal-store-custom-typed

```
    withComputed(({ callState }) => ({
      loading: computed(() => callState() === 'loading'),
      loaded: computed(() => callState() === 'loaded'),
      error: computed(() => {
        const state = callState();
        return typeof state === 'object' ? state.error : null
      }),
    }))
  );
}
```

This is a function that returns the result of `signalStoreFeature`. The `signalStoreFeature` function, in turn, simply groups existing features: `withState` introduces the `callState` property, and `withComputed` defines the previously discussed calculated signals based on it.

The Updaters provided by the feature only return a partial state object with the property to be updated:

```
export function setLoading(): { callState: CallState } {
  return { callState: 'loading' };
}

export function setLoaded(): { callState: CallState } {
  return { callState: 'loaded' };
}

export function setError(error: string): { callState: CallState } {
  return { callState: { error } };
}
```

Now it Really Starts: Typing

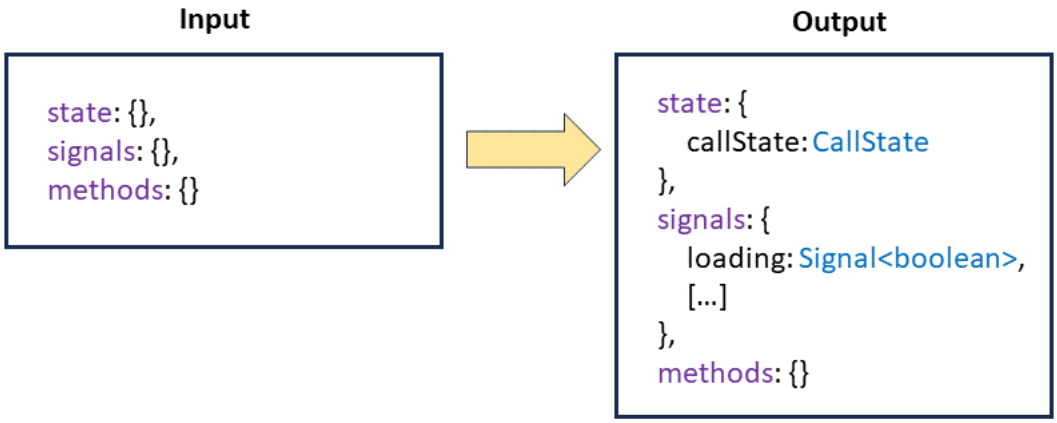
The `CallState` implementation in the last section briefly summarizes the solution to a recurring requirement. Once implemented, individual applications can integrate the feature into their stores.

A drawback of this implementation, however, is that the signals introduced have fixed names: `callState`, `loading`, `loaded`, and `error`. This quickly leads to naming conflicts, especially if the same store requires the feature more than once. An example of this is a store that wants to manage several `callStates` for different entities, e.g. for flights and passengers.

In this case, the consumer should be able to specify the names of the signals introduced. That's precisely what we'll take care of below. To make this extension type-safe, we first have to think a little about the typing of the `withCallState` function.

Our `withCallState` function does not currently have an explicit return type. Therefore, TypeScript infers this type by looking at the return value in the function. The compiler realizes that a `callState` property is available.

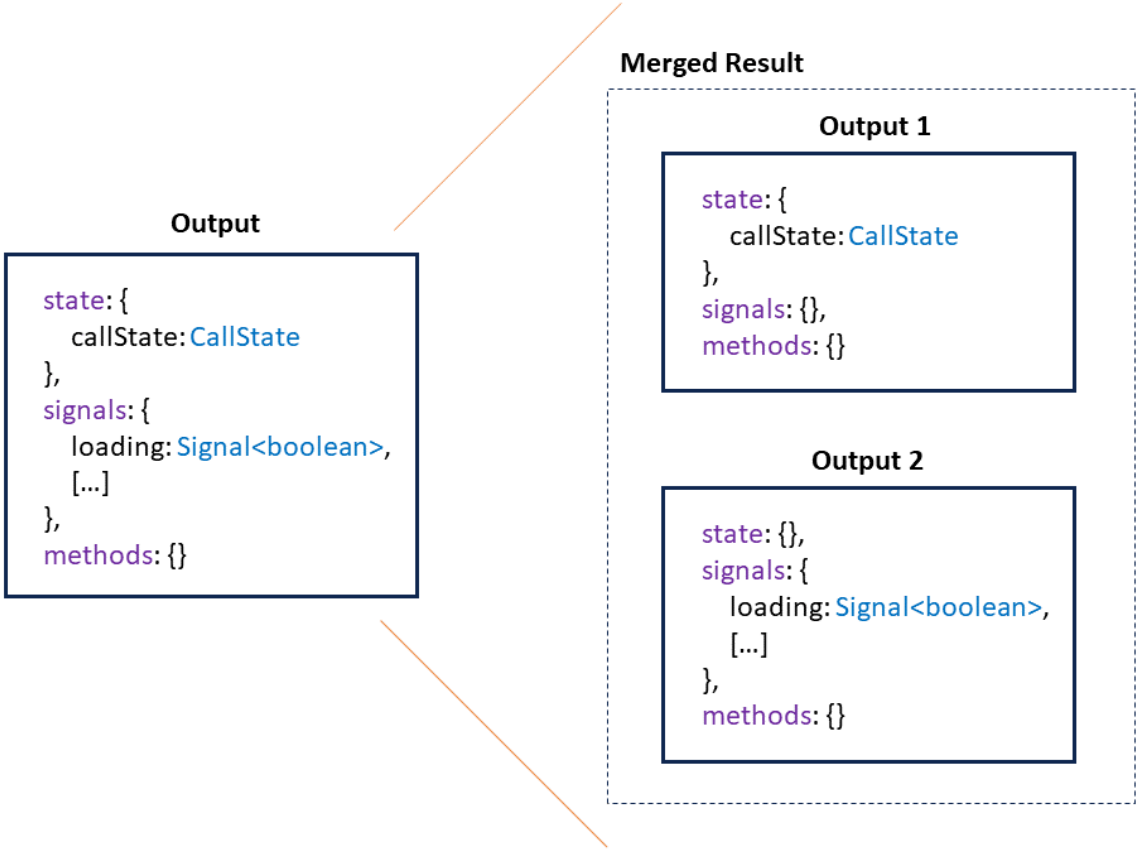
The type determined here by inference is a `SignalStoreFeature<Input, Output>`. The type parameter `Input` defines which signals and methods the feature expects from the store, and `Output` specifies which additional signals and methods the feature provides. Our feature does not place any expectations on the store, but provides a `callState` signal as well as several calculated signals such as `loading`. Respectively, our `Input` and `Output` types looks as follows:



Custom Extensions from the type system perspective

It should be noted that `state` describes the signal to be introduced, and the `signals` property represents the signals calculated from it. This representation at least corresponds to the simplified external view.

The internal view is a little more complex, especially since `withState` first introduces the `callState` signal and only then `withComputed` adds the calculated signals. That's why the inside view has two outputs, which are combined using a helper type.



Custom Extensions from the type system perspective

For the sake of simplicity, the previous image calls the helper type `Merged Result`. However, the truth is that the Signal Store has several internal types for this.

On a logical level, the the internal view und the external one are equivalent. TypeScript may need a little nudge in the form of a type assertion to recognize this. However, explicitly defining the internal view is a bit annoying and currently not really possible because the required helper types are not part of the Signal Store's public API. That's why I 'm using a pattern here that can also be found several times in the Signal Store code: A combination of a function overload with the external view and a function implementation that uses `SignalStoreFeature` instead of `SignalStoreFeature<Input, Output>` for the internal view:

```
// Overloading with External View
export function withCallState()
  : SignalStoreFeature<
    {
      state: {},
      signals: {},
      methods: {}
    },
    {
      state: {
        callState: CallState
      },
      signals: {
        loading: Signal<boolean>,
        loaded: Signal<boolean>,
        error: Signal<{ error: string } | null>
      },
      methods: {}
    }>;
// Implementation with Internal View
export function withCallState(): SignalStoreFeature {
  return signalStoreFeature(
    withState<{ callState: CallState }>({ callState: 'init' }),
    withComputed(({ callState }) => ({
      loading: computed(() => callState() === 'loading'),
      loaded: computed(() => callState() === 'loaded'),
      error: computed(() => {
        const state = callState();
        return typeof state === 'object' ? state.error : null
      }),
    }))
  );
}
```

The `SignalStoreFeature` type without type parameters uses more general types for `Input` and `Output` that do not assume specific names or data types.

Typing and Dynamic Properties – How do They Work Together?

Now that the basic structure of the typing is in place, we can extend it with configurable property names. Following the example of `@ngrx/signals/entity`, consumers should have the option to

define a prefix when activating the feature:

```
export const FlightBookingStore = signalStore(
  { providedIn: 'root' },

  withState({ … }),
  withComputed(( … ) => ({ … })),

  withCallState({ prop: 'flights' }),
  withCallState({ prop: 'passengers'}),

  […]
);
```

This prefix should now be included in the property names defined by the feature. For example, the first call to `withCallState` should produce the following properties:

- `flightsCallState` (state)
- `flightsLoading` (computed)
- `flightsLoaded` (computed)
- `flightsError` (computed)

The second call analogously leads to these properties:

- `passengersCallState` (state)
- `passengersLoading` (computed)
- `passengersLoaded` (computed)
- `passengersError` (computed)

Setting up these properties at runtime isn't a big problem in the world of TypeScript, especially since the underlying JavaScript is a dynamic language anyway. The challenge, however, is to also inform the type system about these properties.

For this task, you first need to find a way to express the prefix in a type declaration. At this point, we benefit from the fact that literals can also be used as data types:

```
export type BoxStatus = 'open' | 'closed';
const candyBox: BoxStatus = 'open';
```

Such String Literal Union Types are often used in TypeScript applications to express enums. This is even closer to EcmaScript than using TypeScript's `enum` keyword. Funnily, nobody is forcing us to offer multiple options. That's why this variant is completely ok:

```
export type BoxStatusAfterHolidays = 'closed';
```

So here we have a type that can hold exactly a single string value. We use this exact pattern to inform the type system about our prefix. First, we create a type that defines the name of the signal to be introduced based on the prefix:

```
export type NamedCallState<Prop extends string> = {
  [K in Prop as `${K}CallState`]: CallState;
};
```

This is a so-called mapped type, which maps one type to a new one. The type parameter `Prop extends string` describes the original type. It can be any string used as a type. String must also be written in lowercase because, at this point, we are referring to a specific string and not the `String object type`. The notation `K in Prop` also reduces to this string. In more complex cases, one could use the keyword `in`, for instance, to loop through the properties of the original type.

We can proceed analogously for the calculated signals to be introduced:

```
export type NamedCallStateComputed<Prop extends string> = {
  [K in Prop as `${K}Loading`]: Signal<boolean>;
} & {
    [K in Prop as `${K}Loaded`]: Signal<boolean>;
} & {
    [K in Prop as `${K}Error`]: Signal<string | null>;
};
```

Since a mapped type can only have a single mapping, several mapped types are used here. They are combined with the `&`-operator (intersection operator). With these two types we can now specify the typing of our `withCallState` function:

```
export function withCallState<Prop extends string>(config: {
  prop: Prop;
}): SignalStoreFeature<
  { state: {}, signals: {}, methods: {} },
  {
    state: NamedCallState<Prop>,
    signals: NamedCallStateComputed<Prop>,
    methods: {}
  }
>;
export function withCallState<Prop extends string>(config: {
  prop: Prop;
}): SignalStoreFeature {
```

```
[…]
}
```

Now, the type system knows about our configured properties. In addition, it is now important to set up these properties at runtime. An auxiliary function `getCallStateKeys` is used for this purpose:

```
function getCallStateKeys(config: { prop: string }) {
  return {
    callStateKey: `${config.prop}CallState`,
    loadingKey: `${config.prop}Loading`,
    loadedKey: `${config.prop}Loaded`,
    errorKey: `${config.prop}Error`,
  };
}
```

This helper function returns the same mappings at runtime as the previously introduced types during compile time. The updated implementation of `withCallState` picks up these names and sets up corresponding properties:

```
[…]
export function withCallState<Prop extends string>(config: {
  prop: Prop;
}): SignalStoreFeature {
  const { callStateKey, errorKey, loadedKey, loadingKey } =
    getCallStateKeys(config);

  return signalStoreFeature(
    withState({ [callStateKey]: 'init' }),
    withComputed((state: Record<string, Signal<unknown>>) => {

      const callState = state[callStateKey] as Signal<CallState>;

      return {
        [loadingKey]: computed(() => callState() === 'loading'),
        [loadedKey]: computed(() => callState() === 'loaded'),
        [errorKey]: computed(() => {
          const v = callState();
          return typeof v === 'object' ? v.error : null;
        })
      }
    })
  );
}
```

So that the updaters can cope with the dynamic properties, they also receive a corresponding parameter:

```
export function setLoading<Prop extends string>(
  prop: Prop
): NamedCallState<Prop> {
  return { [`${prop}CallState`]: 'loading' } as NamedCallState<Prop>;
}
```

This idea can also be found in `@ngrx/signals/entity`. The updater is then used as follows:

```
load: async () => {
  patchState(state, setLoading('flights'));
  […]
}
```

More Examples: CRUD and Undo/Redo

In the previous chapter, I demonstrated features for implementing CRUD and Undo/Redo. The following repo contains a version of these custom features using dynamic properties as shown here.

Source Code[85] (see branch `arc-signal-store-custom-examples-typed`)

Out of the Box Extensions

Knowing how to write such custom features gives you a lot of possibilities. However, quite often you just want to focus on what you are really paid for and this might be writing application code and not writing infrastructure code. That's why we have put several extensions for the Signal Store into the npm package @angular-architects/ngrx-toolkit[86]. Out of the box, it provides several custom features:

- Redux Dev Tools support
- Using the Redux pattern with the Signal Store
- Connecting Data Services to the Signal Store as shown in the previous chapter but with dynamic properties and full CRUD support
- Undo/Redo support

[85] https://github.com/manfredsteyer/standalone-example-cli/tree/arc-signal-store-custom-examples-typed
[86] https://www.npmjs.com/package/@angular-architects/ngrx-toolkit

Conclusion

The NGRX team is known for being exceptionally skilled at leveraging the possibilities of the TypeScript type system. The result is an extremely easy-to-use and type-safe API.

In this chapter, we switched perspectives and discussed how you can leverage the patterns used by the NGRX team for your custom Signal Store features. This enables to configure property names and thus avoid naming conflicts without compromising type safety.

To do this, we have to deal with aspects of TypeScript that application developers usually don't get in contact that often. That's why the patterns used may sometimes seem a bit complicated. The good news is that we only need these patterns if we are developing highly reusable solutions. As soon as we switch back to the role of application developer, we have a type-safe solution that is comfortable to use.

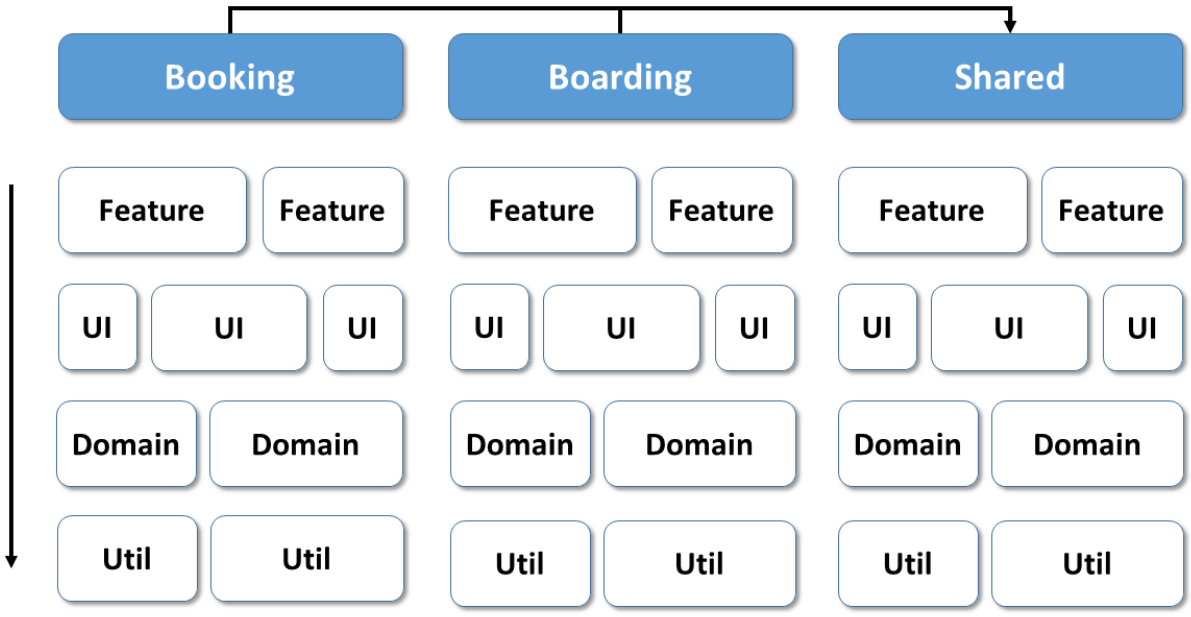
The NGRX Signal Store and Your Architecture

The NGRX Signal Store is a modern and lightweight state management solution. However, when adding it to your application, several architectural questions come up: Where to put it? How large should it be? Is a store allowed to access other stores? Can it be used for global state? Can it be used together with or instead of the traditional Redux-based NGRX Store?

This chapter provides answers and shows that **lightweight stores change some of the rules** known from the world of Redux-oriented stores.

Where to Put it?

Inspired by Strategic Design (DDD) and Nrwl's work on Nx, our reference architecture slices a larger frontend into several decoupled domains (bounded contexts) consisting of several technical layers:



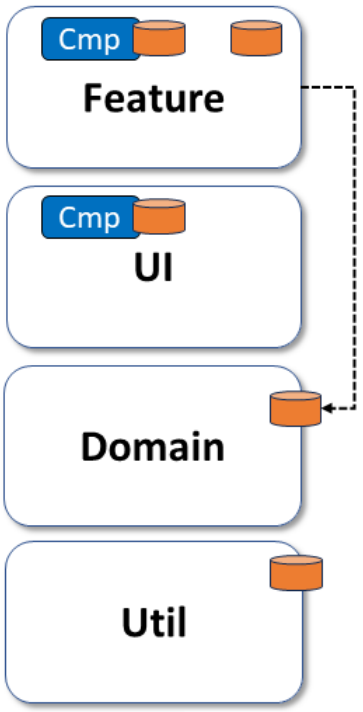
Reference Architecture with decoupled domains and layers

This architecture, which often acts as the starting point and can be tailored to individual requirements, is described in this book's first chapters.

When going with the **traditional Redux-based NGRX Store**, we subdivide the state into feature slices. While they can be associated with the feature layer, we often push them down to the domain level, as the same state is often needed in several features of the same domain.

> When talking about this reference architecture, we should also keep in mind there are several flavors. For instance, some teams have a data layer or state layer where they put feature slices needed by several features. These layers can be an alternative but also an addition to the domain layer.

When we incorporate a **lightweight store like the NGRX Signals Store**, we encounter **different rules**: In general, lightweight stores can be found in all technical layers:



- **Feature Layer**: We can use a store on the component level for managing component state or on the feature level so that several components of the same feature can access it. In the latter case, an example is a wizard delegating to different components.
- **UI**: UI components for sure have state. Some of them have quite extensive ones that need to be shared with child components. An example is a sophisticated scheduler with different views demanding several child components. Such a state can be managed by a lightweight store directly connected to the component.
- **Domain**: State that is needed by several features in the same domain is defined here. A lightweight store used for this is exposed by this layer so that the feature layer can access it.

- **Util**: Quite often, utilities are stateless: Think about functions validating inputs or calculating dates. However, there are also some stateful utility libs where a store can be helpful. An example is a generic authentication library managing some data about the current user or a translation library holding translation texts.

A Store used on the component level is directly provided by the component in question:

```
@Component({
    [...],
    providers: [MySignalStore]
})
export class MyComp {
    [...]
}
```

This also makes the Store available to child components. However, this also means that the store is destroyed when the component is destroyed.

For the other use cases, we can provide the Store via the root injector:

```
export const MySignalStore = signalStore(
    { providedIn: 'root'},
    withState([...]),
    [...]
)
```

The Angular team told the community several times this is the way to go in most cases. In general, we could also provide such stores on the level of (lazy) routes. However, this does not add much value, as `forRoot` services also work with lazy loading: If only used in a lazy application part, the bundler puts them into the respective chunk. More information about when to use so-called Environment providers on the route level can be found here[87].

Combining the Signal Store With the Traditional NGRX Store

You might wonder, why not stick with the traditional NGRX Store for the feature and domain level? You can totally do this: This Store was mainly developed for global state we find in these layers, and it also perfectly supports Signals. Also, if you have already added the traditional NGRX Store to your app and you are happy with it, I would not change this.

[87] https://www.angulararchitects.io/en/blog/routing-and-lazy-loading-with-standalone-components/

However, I also think more and more people will reconsider using "Redux by default". If you feel that you don't benefit from the strength of this approach in your very case, you might want to go with a more lightweight alternative like the NGRX Signal Store instead. This can also be observed in other communities where lightweight stores have been popular for years.

To be clear, the Redux pattern should be a part of your toolbox. However, if you find a more lightweight solution that fits better, go with it.

The Best of Both Worlds via Custom Features

As the NGRX Signal Store is highly extensible, you can even use the best concepts of both worlds. Let's say you miss the indirections or the eventing provided by Redux: No one prevents you from writing a custom feature adding this to the Signal Store.

If you look for an out of the box solution for Redux on top of the Signal Store, you find one in our npm package @angular-architects/ngrx-toolkit[88].

How Large Should a Signal Store be?

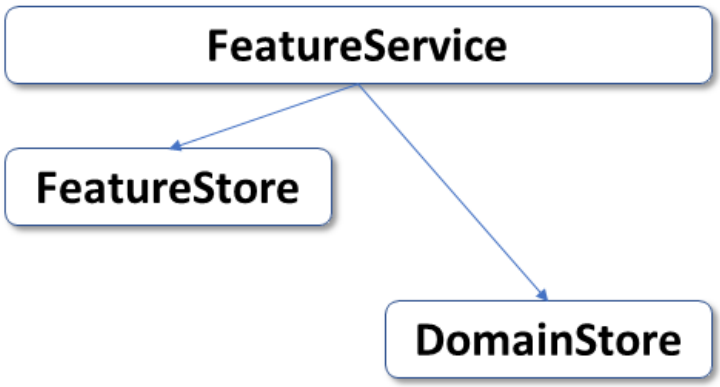
When coming from the traditional NGRX Store, as a rule of thumb, a Signal Store can have the granularity of a feature slice. However, since a Signal Store is just a service, we should also align with the single responsibility principle. Hence, splitting a feature slice into more fine-grained ones can be a good idea.

May a Signal Store Access Other Signal Stores?

When we spread our state to several lightweight stores in different layers, it's not unusual that a use case might need a state from more than one Store. In general, I would avoid stores accessing other stores. Each Store should just focus on its task, which is managing its state properties. Also, we want to prevent cycles.

Fortunately, there is an alternative. Go with a (feature) service orchestrating the stores:

[88] https://www.npmjs.com/package/@angular-architects/ngrx-toolkit



Such a service is similar to facades often used for state management. But as it's part of the feature and doesn't abstract a sub-system, I prefer the name feature service.

Preventing Cycles, Redundancies, and Inconsistencies

The layering introduced with our reference architecture and the rule that stores are not allowed to access each other prevents cycles. In general, our different stores can become redundant and hence inconsistent if we don't take care. However, the same risk exists with different independent feature slices when going with the traditional NGRX Store.

Having a way to visualize the state seems to be vital in both cases, as it helps with detecting such issues early. For the traditional NGRX Store, we use the Redux Dev Tools. However, the Signal Store does not come with out-of-the-box support for them. Instead, the Angular Dev Tools might eventually support a general visualization of the Signals used in an application. Nevertheless, having Redux Dev Tools support seems to be extremely helpful because they provide a history and allow for time travel debugging. Fortunately, implementing such support with custom features should be straightforward. So, like with the NGRX Component Store, it's likely the community will come up with such an implementation.

Another way to prevent inconsistencies is making use of eventing. This allows us to inform other system parts about changes so that they can update their state accordingly. In Redux, eventing is part of the game. For the Signal Store, we could add eventing using custom features.

Conclusion

Lightweight stores like the NGRX Signal Store change some of the rules known from Redux-based: Such stores can be defined on different technical layers, and they can be provided within the root provider, a (lazy) route, or on the component level.

Redux is not going away, and it belongs to our toolbox. However, if you feel a more lightweight approach is more fitting for your needs, the NGRX Signal Store is quite tempting. Also, you can have the best of both worlds by combining both stores or by extending the Signal Store with custom features that provide missing Redux features.

In view of the single responsibility principle, I would not allow lightweight stores to access each other; instead, you can introduce a feature service orchestrating the needed stores.

Bonus: Automate your Architecture with Nx Workspace Plugins

Nx is quite popular when it comes to large Angular-based business applications. Thanks to its plugin concept, Nx can also be extended very flexibly. The Nx plugin registry[89] lists numerous such plugins that take care of recurring tasks and integrate proven tools.

In addition to community plugins for the general public, project-internal plugins can also make sense to automate highly project-related tasks. This includes generating code sections and implementing patterns, the target architecture specifies: repositories, facades, entities or CRUD forms are just a few examples.

Unfortunately, implementing plugins was not trivial in the past: Each plugin had to be published as a package via npm and installed in your own Nx workspace. This procedure had to be repeated for each new plugin version.

This back and forth is a thing of the past thanks to workspace plugins. These are plugins that Nx sets up in the form of a library in the current workspace. This means that changes can be made quickly and tested immediately. If necessary, proven workspace plugins can also be exported via npm for other projects.

In this chapter I show how workspace plugins with generators automating repeating tasks can be implemented and used.

Source Code[90]

Creating a Workspace Plugin With a Generator

The `@nrwl/nx-plugin` package can be used to generate new plugins. It also comes with numerous helper methods that support the development of plugins. You can use the following instructions for creating a new Nx workspace with a plugin:

[89] https://nx.dev/plugin-registry
[90] https://github.com/manfredsteyer/nx-plugin-demo

```
npx create-nx-workspace@latest plugin-demo

cd my plugin-demo

npm i @nrwl/nx-plugin

nx generate @nrwl/nx-plugin:plugin libs/my-plugin
```

When asked, select the options `Angular` and `Integrated Monorepo`; for the remaining options you can go with the defaults.

After that, add a generator to your plugin:

```
nx generate @nx/plugin:generator my-generator --directory libs/my-plugin/src/generat\
ors/my-generator
```

Templates for Generators

Generators often use templates usually put into the *files* subfolder. Template are files with placeholders that the generator copies into the target project. For example, the following template uses a placeholder *projectName* and generates ten constants:

```
<% /* Filename: libs\my-plugin\src\generators\my-generator\files\src\index.ts.templa\
te */ %>

<% for (let i=0; i<10; i++) { %>
const constant<%=i%> = '<%= projectName %>';
<% } %>
```

If you follow the instructions here step by step, please copy the contents of this listing into the generated file `libs\my-plugin\src\generators\my-generator\files\src\index.ts.template`.

Wildcards can be found not only in the files, but also in the file names. For example, Nx would replace *__projectName__* in a file name with the value of *projectName*.

Implementing a Generator

Technically speaking, a generator is just an asynchronous function receiving two parameters: A tree object representing the file system and an `options` object with the parameters passed when calling the generator at the command line:

```
// libs/my-plugin/src/generators/my-generator/generator.ts

import {
  formatFiles,
  generateFiles,
  getWorkspaceLayout,
  Tree,
} from '@nrwl/devkit';

import {
  libraryGenerator
} from '@nrwl/angular/generators';

import * as path from 'path';
import { MyGeneratorGeneratorSchema } from './schema';

export default async function (tree: Tree, options: MyGeneratorGeneratorSchema) {

  tree.write('readme.txt', 'Manfres was here!');

  await libraryGenerator(tree, options);

  const libsDir = getWorkspaceLayout(tree).libsDir;
  const projectRoot = `${libsDir}/${options.name}`;

  const templateOptions = {
    projectName: options.name,
    template: ''
  };

  generateFiles(
    tree,
    path.join(__dirname, 'files'),
    projectRoot,
    templateOptions
  );

  await formatFiles(tree);
}
```

This example illustrates some typical tasks that generators perform:

- The `tree.write` method creates a new file

- The `libraryGenerator` method from the `@nrwl/angular/generators` package represents the generator that the `ng g lib` statement triggers. The call shown thus generates a new library in the current workspace.
- With `generateFiles`, the generator copies all templates from the `files` folder into the root directory of the new project. The values for the placeholders are in the `templateOptions` object.
- The call to `formatFiles` formats the generated files with Prettier. This simplifies the structure of the templates.

What's particularly useful is the fact that generators are simply functions that can be called in other generators. This means that existing generators can be combined to create new ones.

To add additional parameters passed via the options object, the interface in the file `schema.d.ts` as well as the JSON schema in `schema.json` need to be extended accordingly. The former one is used in the TypeScript code and the latter one is used by Nx to validate the parameters passed at command line.

True Treasures: Helper Methods for Generators in Nx

In addition to the methods used here, the `@nrwl/devkit` package offers some other useful auxiliary constructs for developing generators. Here is a selection of methods that are often used in practice:

- `readJson` and `updateJson`: Reading and updating a JSON file
- `readNxJson`: Reads the file `nx.json`, the control file of Nx
- `readWorkspaceConfiguration`: Reads the workspace configuration (originally part of `angular.json`, now part of `nx.json`).
- `readProjectConfiguration` and `updateProjectConfiguration`: Reads or updates the configuration of a specific project from the respective `project.json`.
- `applyChangesToString`: Performs multiple inserts and deletes on a file.
- `names`: Formats strings to conform to conventions for file names (kebab case) or class names (pascal case).

If it is necessary to change existing TypeScript files, the TypeScript Compiler API[91] can help. This API is included in TypeScript and represents code files as syntax trees.

The tsquery[92] package, which is very popular in the community, supports searching these data structures. It allows you to formulate queries that are based on CSS selectors. Such queries, for example, can determine functions, classes, or methods that are located in a file.

Trying out Generators

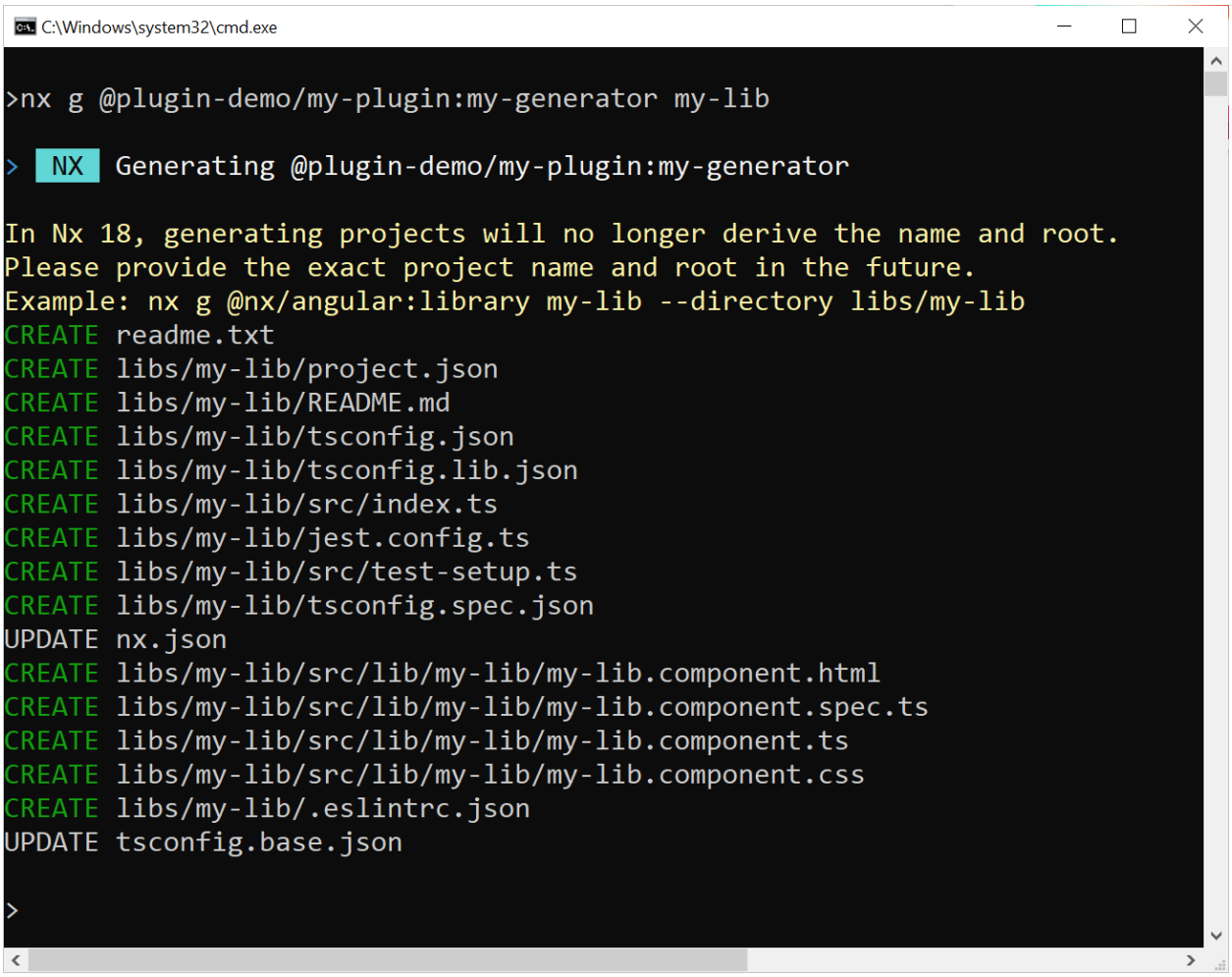
The shown generator can now be run on the console with `nx generate`:

[91] https://github.com/microsoft/TypeScript/wiki/Using-the-Compiler-API
[92] https://www.npmjs.com/package/@phenomnomnominal/tsquery

```
nx g @plugin-demo/my-plugin:my-generator my-lib
```

Here, `@my-workspace` is the name of the current workspace and `my-plugin` is the name of the library with our workspace plugin. The name `my-generator` refers to the generator we've added to the plugin. `my-lib` is the value for the `name` parameter. Actually this should be specified with `--name mylib`. However, the generator's `schema.json` by default specifies that this value can alternatively be taken from the first command line argument.

If everything goes as planned, the generator creates a new library and a file based on the template shown. It also generates a `readme.txt`:



Testing Generators

Nx also simplifies the automated testing of generators. It also offers auxiliary constructs, such as a *Tree* object, which only simulates a file system in main memory and does not write it to disk.

In addition, Nx also generates the basic structure for a unit test per generator. To make it fit our implementation shown above, let's update it as follows:

```
// libs/my-plugin/src/generators/my-generator/generator.spec.ts

import { createTreeWithEmptyWorkspace } from '@nrwl/devkit/testing';
import { Tree, readProjectConfiguration } from '@nrwl/devkit';

import generator from './generator';
import { MyGeneratorGeneratorSchema } from './schema';

describe('my-plugin generator', () => {
  let appTree: Tree;
  const options: MyGeneratorGeneratorSchema = { name: 'test-lib' };

  beforeEach(() => {
    appTree = createTreeWithEmptyWorkspace();
  });

  it('should export constant0', async () => {
    await generator(appTree, options);
    const config = readProjectConfiguration(appTree, 'test-lib');
    expect(config).toBeDefined();

    const generated = `${config.sourceRoot}/index.ts`;
    const content = appTree.read(generated, 'utf-8');
    expect(content).toContain(`const constant0 = 'test-lib';`);
  });
});
```

The unit test shown here creates a memory-based `Tree` object using `createTreeWithEmptyWorkspace` and calls our generator. It then checks whether there is a configuration for the generated library and whether it has the generated file.

To run this unit test, call

```
nx test my-plugin
```

Exporting Plugins via NPM

If you want to use your plugin not only in the current Nx workspace, but also in other projects, all you have to do is build it and deploy it via npm:

```
nx build my plugin

npm publish dist\libs\my-plugin --registry http://localhost:4873
```

Here, we assume that verdaccio is used as the npm registry and that it's started locally on port 4873. Without specifying the `--registry` switch, npm uses the public registry at `registry.npmjs.org`.

The npm package simply needs to be installed in the consuming workspace. After that, you can then use your generator as usual:

```
npm i my-plugin --registry http://localhost:4873

nx g @my-workspace/my-plugin:my-generator my-lib
```

Conclusion

Workspace plugins significantly simplify the development of plugins to automate repeating project-internal tasks. This is not only due to the numerous helper methods, but above all to the tooling: Nx generates the basic structure of plugins and generators including unit tests Changes can be tried out immediately in the current workspace. If necessary, these libraries can also be exported via npm and used in other projects.

Another plus point is the straightforward API that Nx provides us: Generators are just functions that can easily call each other. This means that existing functionalities can be orchestrated into new ones.

Bonus: The Core of Domain-Driven Design

It's been a bit more than 20 years since the publication of Eric Evans' groundbreaking book Domain-Driven Design: Tackling Complexity in the Heart of Software[93] that established the DDD movement. This book is still a best-seller, and a vivid community has extended DDD since then: There are dedicated international conferences, books, courses, and new concepts that practitioners have added.

While there are several perspectives on DDD, I want to reflect on the core of this approach here.

My goal is to - shed some light on the focus of DDD, - why there are wrong impressions about it, - its relationship to object-orientation, - and whether it can be adapted to fit other paradigms.

For this, I'm primarily citing interviews with and presentations of Eric Evans. To provide additional examples, I also cite further sources. Before, I start with a quick overview of DDD to get everyone into the boat.

DDD in a Nutshell

Domain-driven Design focuses on a deep understanding of the real-world (problem) domain a software system is written for. Domain experts (e.g., experts for invoicing) work closely together with software experts to create a models of that domain. A model represents aspects of the real world (concepts, relationships, processes) that are interesting for the software in question and is directly expressed in the source code.

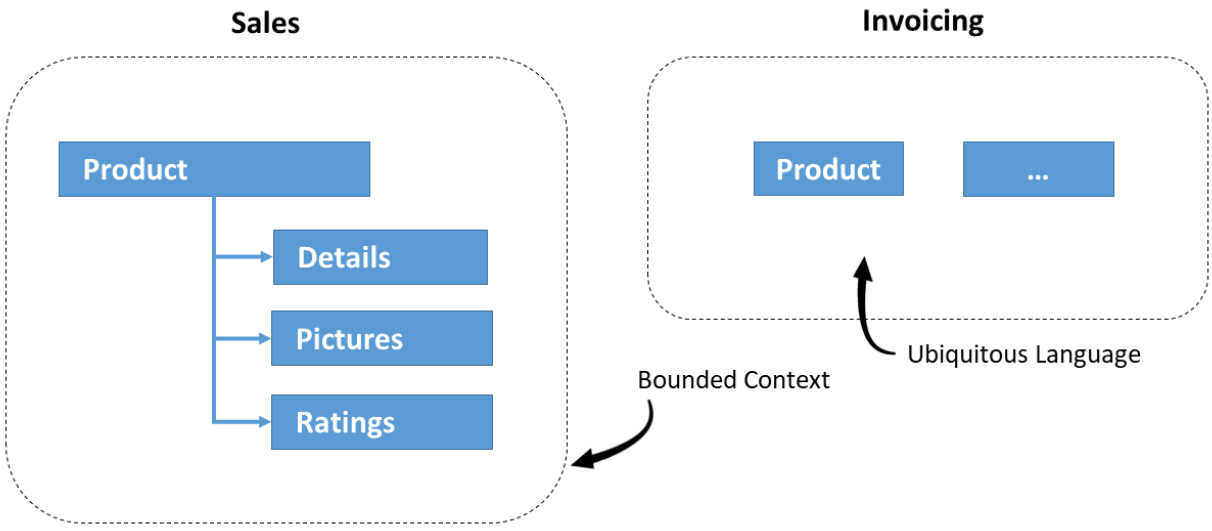
Strategic Design

DDD consists of two original disciplines: Strategic Design[94] is about discovering subdomains that represent individual parts of the problem domain. For subdomains, bounded contexts[95] are defined. Each bounded context gets an own model that follows an Ubiquitous Language[96]. This Ubiquitous Language reflects the vocabulary used in real-world and is used by domain experts as well as by software experts – verbally, in written form, in diagrams, and in code.

Having several individual models instead of one sole overly system-wide model allows for a more meaningful representation of the different sub-domains. This also prevents tight coupling and reduces complexity.

[93] https://www.youtube.com/watch?v=7yUONWp-CxM
[94] https://www.thoughtworks.com/en-cl/insights/blog/evolutionary-architecture/domain-driven-design-in-10-minutes-part-one
[95] https://martinfowler.com/bliki/BoundedContext.html
[96] https://martinfowler.com/bliki/UbiquitousLanguage.html

The following example shows two bounded contexts. Each of them has its own view on the concept of a product and, hence its own representation:



Sales and Invoicing are two different bounded contexts with their own representation of an product

Tactical Design

While Strategic Design leads to an overarching architecture, Tactical Design[97] provides several building blocks that help to implement the model within the individual contexts. Examples are Value Objects and Entities[98], Aggregates[99] defining whole-part relationships (e.g. an Order with Order Lines) with consistency rules (invariants) that create some implications for transaction management, and repositories for persisting and loading aggregates and entities.

Recent Developments in DDD

Since its first days, several ideas and concepts have been added to DDD.

One example for a pattern that is meanwhile part of DDD but was not explicitly mentioned in the original book are Domain Events.

An example of a new discipline added during the years is Collaborative Modelling: It provides approaches and workshop formats such as Event Storming[100] and Domain Story Telling[101] that bring domain experts and software experts together and help them to explore a domain.

[97] https://www.thoughtworks.com/en-ca/insights/blog/evolutionary-architecture/domain-driven-design-part-two
[98] https://martinfowler.com/bliki/EvansClassification.html
[99] https://martinfowler.com/bliki/DDD_Aggregate.html
[100] https://www.eventstorming.com
[101] https://domainstorytelling.org/

Strategic Design has also been adopted by the Microservice community[102] to identify boundaries between services. Similarly[103], the Micro Frontend community is leveraging Strategic Design too. Besides this, it is also used for monolithic applications[104].

Team Topologies[105] is another relatively young discipline that favors the Bounded Context for splitting a system into individual parts different teams can work on.

More on DDD

You find more details on DDD in the blog articles linked above. If you prefer recordings, you find an excellent one about Strategic Design here[106] and a discussion about prioritizing bounded contexts which leads to the idea of a Core Domain there[107].

How to Define DDD?

In his keynote "DDD Isn't Done" at Explore DDD 2018[108] in Denver, Eric Evans talked about how to define DDD. He stressed that a good balance needs to be found so that DDD can help write "valuable software." A too rigorous definition would make it a "trivial cookbook," while, on the other hand, just being a "handwavy good advice" is also counterproductive.

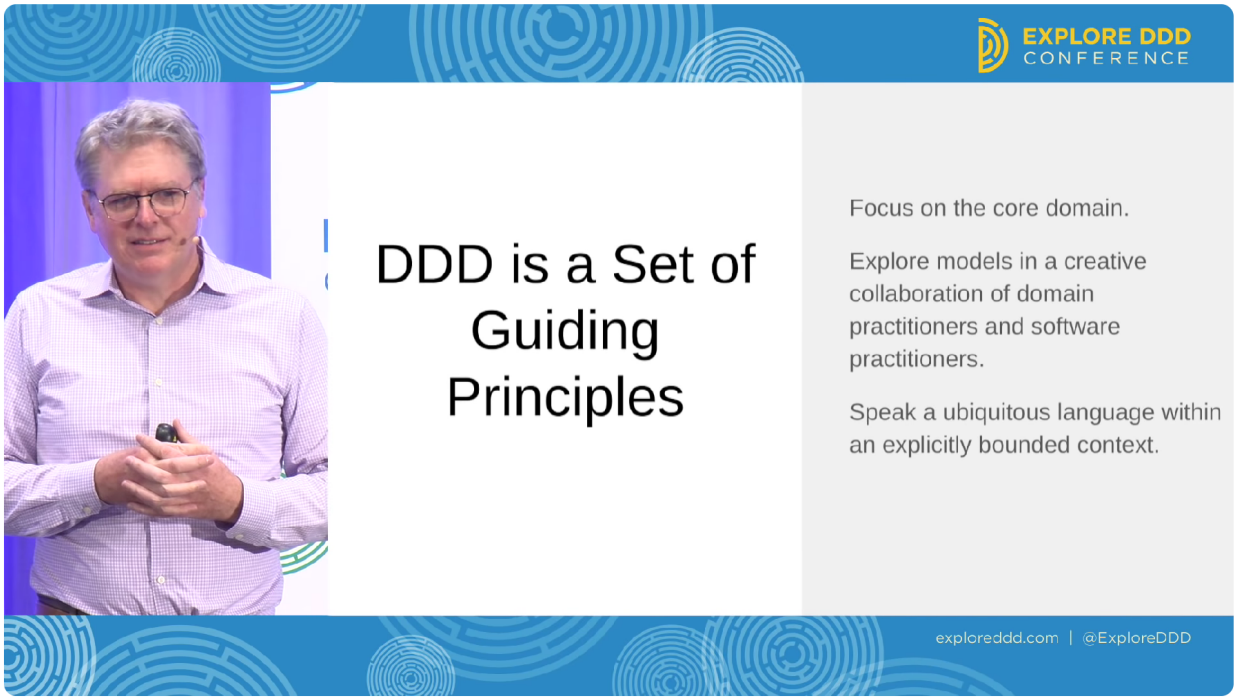
He also adds:

> We need some room to move. Different people need to be able to operate in a space and have different views and innovate.

His definition of DDD mainly focuses on a set of guiding principles:

- Focus on the core domain.
- Explore models in a collaboration of domain practitioners and software practitioners.
- Speak a ubiquitous language within an explicitly bounded context.

[102] https://www.amazon.de/Building-Microservices-Designing-Fine-Grained-Systems/dp/1492034029
[103] https://www.amazon.de/Building-Micro-Frontends-Projects-Empowering-Developers/dp/1492082996
[104] https://www.amazon.de/Strategic-Monoliths-Microservices-Architecture-Addison-Wesley/dp/0137355467/
[105] https://www.amazon.de/Team-Topologies-Organizing-Business-Technology/dp/1942788819
[106] https://www.youtube.com/watch?v=Z3tM0UHhhwI
[107] https://www.youtube.com/watch?v=84ofg2q14CY
[108] https://www.youtube.com/watch?v=R2IAgnpkBck



Eric Evans providing a definition for DDD at Explore DDD 2018

When Can we Call it DDD?

The definition outlined in the previous section also fits an answer Eric Evans gave at an interview with InfoQ[109]. Asked about the minimal set of practices a team has to follow to practice DDD effectively, his answer focused on Ubiquitous Language and the Bounded Context:

> [...] the most fundamental pattern of Domain-driven Design is probably the ubiquitous language. [...]
>
> [A model] applies within a certain context, and that context has a definitely defined limit, [it's] a bounded context.
>
> With those two ingredients, I would say, someone is doing Domain-driven Design, and there are a lot of other practices that help solve more specific problems.

[109] https://www.infoq.com/interviews/domain-driven-design-eric-evans/

What's the Core of DDD and Why did People get a Wrong Impression About that?

In an interview with the IEEE Computer Society[110] on the occasion of DDD's 10th anniversary, Eric Evans was asked what he would do differently if he rewrote his book on DDD. Interestingly, he thinks his book gave a lot of readers the wrong impression that DDD is mainly about the building blocks associated with Tactical Design, while the core of DDD is Strategic Design:

> [...] all the strategic design stuff is way back at the back. [...] it's so far back that most people never get to it really.
>
> Another thing I would do is try to change the presentation of the building blocks [...] things like the entities and value objects [...] [People] come away thinking that that's really the core of DDD, whereas, in fact, it's really not.
>
> I really think that the way I arranged the book gives people the wrong emphasis, so that's the biggest part of what I do is rearrange those things.

However, he adds that Tactical Design is important because it helps to translate the conceptual model into code.

A similar point of view is expressed in Eric Evans' keynote at DDD Europe 2016[111], where he criticizes the "over-emphasis on building blocks".

Is Tacticial Design Object-Oriented? Is There a Place for FP?

In the previously mentioned keynote[112], Eric Evans stresses that DDD is not bound to a specific paradigm. He also mentions that FP can be a "powerful mechanism for abstraction" and that functional languages can help to express a model in code.

In general, he points out that nowadays, there are several new languages that are more expressive and, hence, help to implement models in a concise way. Another way to make code more expressive is using libraries like Reactive Extensions.

When asked about using Functional Programming (FP), while his book clearly focuses on Object-orientation, in the interview with the IEEE Computer Society[113] he said:

[110] https://www.youtube.com/watch?v=GogQor9WG-c
[111] https://www.youtube.com/watch?v=dnUFEg68ESM
[112] https://www.youtube.com/watch?v=dnUFEg68ESM
[113] https://www.youtube.com/watch?v=GogQor9WG-c

> The reason that everything is expressed in terms of objects is because objects were king in 2003-2004, and what else would I have described it as people [...] used objects.

He explains that there need to be some changes to apply Tactical Design to FP:

> If you are going at it from a functional point of view, then [...] your implementations are going to look quite different.

Also here[114], he mentions the need for "rethinking [...] building blocks" when switching to FP.

This needed adaption is also slightly addressed in Vaughn Vernon's book Domain-Driven Design Distilled[115] that is considered a standard reference in the DDD community and known for its easy readability. He mentions that in functional DDD, the data structures are Immutables (records), and pure functions implement the business logic:

> Rather than modifying the data that functions receive as arguments, the functions return new values. These new values may be the new state of an Aggregate or a Domain Event that represents a transition in an Aggregate's state.

More insights on functional DDD can be found in Functional and Reactive Domain Modeling[116] and Domain Modeling Made Functional[117].

Further Adaptions of Tactical Design

On several occasions (e. g. here[118] and here[119]) when discussing the adaptation of Tactical Design to fit other ideas and paradigms, Eric Evans mentions event sourcing and CQRS. Initially, both were not part of DDD but have been incorporated by the community. Another example of an adaptation of Tactical Design patterns mentioned here[120] is using the actor model for implementing Aggregates:

> [An actor] can maintain that state in a consistent [...] way [...] that respects the invariance of that particular aggregate. [...]

This discussion also fits the recently prominently observed talk "The Aggregate is dead. Long live the Aggregate!"[121] by Milan Savić and Sara Pellegrini. This talk, presented at several conferences,

[114] https://www.youtube.com/watch?v=dnUFEg68ESM
[115] https://www.amazon.de/Domain-Driven-Design-Distilled-Vaughn-Vernon/dp/0134434420/
[116] https://www.amazon.de/Functional-Reactive-Domain-Modeling-Debasish/dp/1617292249
[117] https://www.amazon.de/Domain-Modeling-Made-Functional-Domain-Driven/dp/1680502549/ref=sr_1_1?dib=eyJ2IjoiMSJ9.lh7eRp45b3q6zBsCJIM6A9pbWVDAtn1NHumx_LEXffyVsOwKnxYYneAoMlkH2uNgkp_HFULSJ-rwTdOFOEarPA.bLt4ucE-6GEo4H-Q2mzBuGdY5ezuSBBo62I6AOzkLLg&dib_tag=se&keywords=ddd+functional&qid=1704980818&sr=8-1
[118] https://www.youtube.com/watch?v=GogQor9WG-c
[119] https://www.youtube.com/watch?v=R2IAgnpkBck
[120] https://www.youtube.com/watch?v=GogQor9WG-c
[121] https://www.youtube.com/watch?v=Q89patz4lgU

discusses some criticism of the traditional implementation of Aggregates and proposes an alternative implementation using messaging and event sourcing.

More generally, such approaches correlate with Eric Evans's above-cited keynote from 2018[122], where he emphasizes the need to give people room to innovate DDD.

At DDD Europe 2016[123], Eric Evans mentioned two further paradigms that can be used for creating models in DDD:

- Relational
- Graphs

Relational modeling might come as a surprise. However, he does not refer to a comprehensive (generalized) normalized schema that is the opposite of thinking in bounded contexts. Instead, having several specialized schemas fits the mindset of DDD. Also, he finds that SQL can be a good way to express how to compare and manipulate big sets.

With Graphs, Eric Evans means more than just using a Graph Database. He sees graph theory as a "classic modeling paradigm that goes back long before computer [science]." For him, graphs are a way to model "a certain kind of problems" using nodes and edges as abstractions.

Conclusion

In its core, DDD emphasizes that Domain Experts and Software Experts should jointly explore a domain and model individual, prioritized bounded contexts respecting an ubiquitous language.

Tactical Design as described by the original book on DDD helps to implement these models in an Object-oriented way. In addition, there are alternatives and adaptions (e.g. for Functional Programming).

Some communities just refer to Strategic Design (e.g., Micro Services, Micro Frontends, Team Topologies) and use it to subdivide a system along domain boundaries.

[122] https://www.youtube.com/watch?v=R2IAgnpkBck
[123] https://www.youtube.com/watch?v=dnUFEg68ESM

Literature

- Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software[124]
- Wlaschin, Domain Modeling Made Functional[125]
- Ghosh, Functional and Reactive Domain Modeling[126]
- Nrwl, Monorepo-style Angular development[127]
- Jackson, Micro Frontends[128]
- Burleson, Push-based Architectures using RxJS + Facades[129]
- Burleson, NgRx + Facades: Better State Management[130]
- Steyer, Web Components with Angular Elements (article series, 5 parts)[131]

[124] https://www.amazon.com/dp/0321125215
[125] https://pragprog.com/book/swdddf/domain-modeling-made-functional
[126] https://www.amazon.com/dp/1617292249
[127] https://go.nrwl.io/angular-enterprise-monorepo-patterns-new-book
[128] https://martinfowler.com/articles/micro-frontends.html
[129] https://medium.com/@thomasburlesonIA/push-based-architectures-with-rxjs-81b327d7c32d
[130] https://medium.com/@thomasburlesonIA/ngrx-facades-better-state-management-82a04b9a1e39
[131] 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[132] and Facebook[133] and find his blog here[134].

[132] https://twitter.com/ManfredSteyer
[133] https://www.facebook.com/manfred.steyer
[134] 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[135]:

Advanced Angular Workshop

Save your ticket[136] for one of our **remote or on-site** workshops now or request a company workshop[137] (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

[135] https://www.angulararchitects.io/en/angular-workshops/advanced-angular-enterprise-architecture-incl-ivy/
[136] https://www.angulararchitects.io/en/angular-workshops/advanced-angular-enterprise-architecture-incl-ivy/
[137] https://www.angulararchitects.io/en/angular-workshops/

Please find the full list with our offers here[138].

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[139] and/ or follow the book's author on Twitter[140].

[138] https://www.angulararchitects.io/en/angular-workshops/
[139] https://www.angulararchitects.io/subscribe/
[140] https://twitter.com/ManfredSteyer