ENTERPRISE ANGULAR

DDD, Nx Monorepos, and Micro Frontends

3rd Edition

Covering Module Federation

MANFRED STEYER

Enterprise Angular

DDD, Nx Monorepos and Micro Frontends

Manfred Steyer

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

This version was published on 2020-12-05

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 - 2020 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**
Help Improve this Book! 1
Trainings and Consultancy 1
Thanks 2

Strategic Domain-Driven Design **4**
What is Domain-Driven Design? 4
Finding Domains with Strategic Design 4
Context-Mapping 7
Conclusion 8

Implementing Strategic Design with Nx Monorepos **10**
Implementation with Nx 11
Categories for Libraries 12
Public APIs for Libraries 12
Check Accesses between libraries 13
Access Restrictions for a Solid Architecture 13
Conclusion 16

Tactical Domain-Driven Design with Angular and Nx **17**
Code Organisation 19
Isolate the Domain 19
Implementations in a Monorepos 20
Builds within a Monorepo 22
Entities and your Tactical Design 22
Tactical DDD with Functional Programming 23
Tactical DDD with Aggregates 25
Facades 25
Stateless Facades 26
Domain Events 26
Your Architecture by the Push of a Button: The DDD-Plugin 27
Conclusion 28

Short Note: Incremental Builds to Speed up Your CI Process **29**

From Domains to Microfrontends **33**
Deployment Monoliths 33
Deployment monoliths, microfrontends, or a mix? 34
UI Composition with Hyperlinks 35
UI Composition with a Shell 36
Finding a Solution 38
Conclusion 38

The Microfrontend Revolution: Using Module Federation with Angular **39**
Example 39
Getting started 41
Activating Module Federation for Angular Projects 42
The Shell (aka Host) 42
The Microfrontend (aka Remote) 44
Standalone-Mode for Microfrontend 46
Trying it out 46
Bonus: Loading the Remote Entry 47
Conclusion and Evaluation 48

Dynamic Module Federation with Angular **49**
Module Federation Config 50
Routing to Dynamic Microfrontends 50
Improvement for Dynamic Module Federation 51
Bonus: Dynamic Routes for Dynamic Microfrontends 53
Conclusion 54

Plugin Systems with Module Federation: Building An Extensible Workflow Designer . . **55**
Building the Plugins 56
Loading the Plugins into the Workflow Designer 57
Providing Metadata on the Plugins 58
Dynamically Creating the Plugin Component 58
Wiring Up Everything 59
Conclusion 61

Using Module Federation with Nx **62**
Example 62
The Shared Lib 64
The Module Federation Configuration 65
Trying it out 67
Deploying 67
What Happens Behind the Covers? 67
Bonus: Versions in the Monorepo 68
Pitfalls (Important!) 70

Dealing with Version Mismatches in Module Federation **72**
Example Used Here 72
Semantic Versioning by Default 74
Fallback Modules for Incompatible Versions 74
Differences With Dynamic Module Federation 75
Singletons 77
Accepting a Version Range 80
Conclusion 81

Micro Frontends with Web Components/ Angular Elements – An Alternative Approach . **83**
Step 0: Make sure you need it 84
Step 1: Implement Your SPAs 84
Step 2: Expose Shared Widgets 85
Step 3: Compile your SPAs 85
Step 4: Create a shell and load the bundles on demand 86
Step 5: Communication Between Microfrontends 87
Conclusion 88

Literature **89**

About the Author **90**

Trainings and Consulting **91**

Introduction

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

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

Domain-driven design, and especially strategic design, helps. DDD can be the foundation for building microfrontends.

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

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

Help Improve this Book!

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

Trainings and Consultancy

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

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

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

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

Advanced Angular Workshop

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

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

Thanks

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

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

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

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

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

Strategic Domain-Driven Design

Monorepos allow large enterprise applications to subdivide into small maintainable libraries. First, however, we need to define criteria to slice our application into individual parts. We must also establish rules for communication between them.

In this chapter, I present the techniques I use to subdivide large software systems: strategic design. It's part of the domain driven design[10] (DDD) approach. I also explain how to implement its ideas with an Nx[11]-based monorepo.

What is Domain-Driven Design?

DDD describes an approach that bridges the gap between requirements for complex software systems and appropriate application design. Within DDD, we can look at the tactical design and the strategic design. The tactical design proposes concrete concepts and patterns for object-oriented design or architecture and has clear views on using OOP. As an alternative, there are approaches like Functional Domain Modeling[12] that transfer the ideas behind it into the functional programming world.

By contrast, strategic design deals with the breakdown of an extensive system into individual (sub-)domains and their design. No matter if you like DDD's views or not, some ideas from strategic design have proven useful for subdividing a system into smaller, self-contained parts. It is these ideas that this chapter explores in the context of Angular. The remaining aspects of DDD, however, are irrelevant for this chapter.

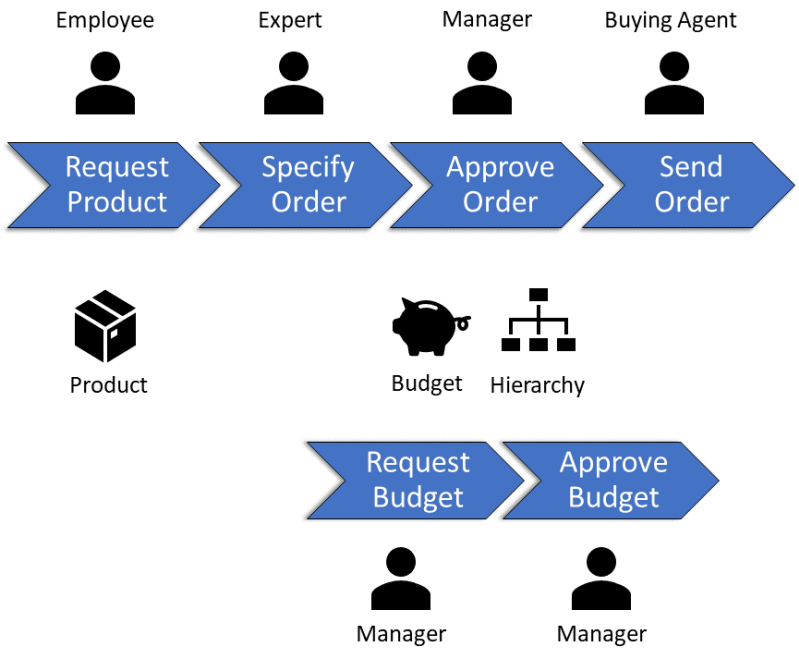
Finding Domains with Strategic Design

One goal of strategic design is to identify self-contained domains. Their vocabulary identifies these domains. Domain experts and developers must use this vocabulary to prevent misunderstandings. As the code uses this language, the application mirrors its domain and hence is more self-describing. DDD refers to this as the ubiquitous language[13].

Another characteristic of domains is that often only one or a few groups of experts primarily interact with them.

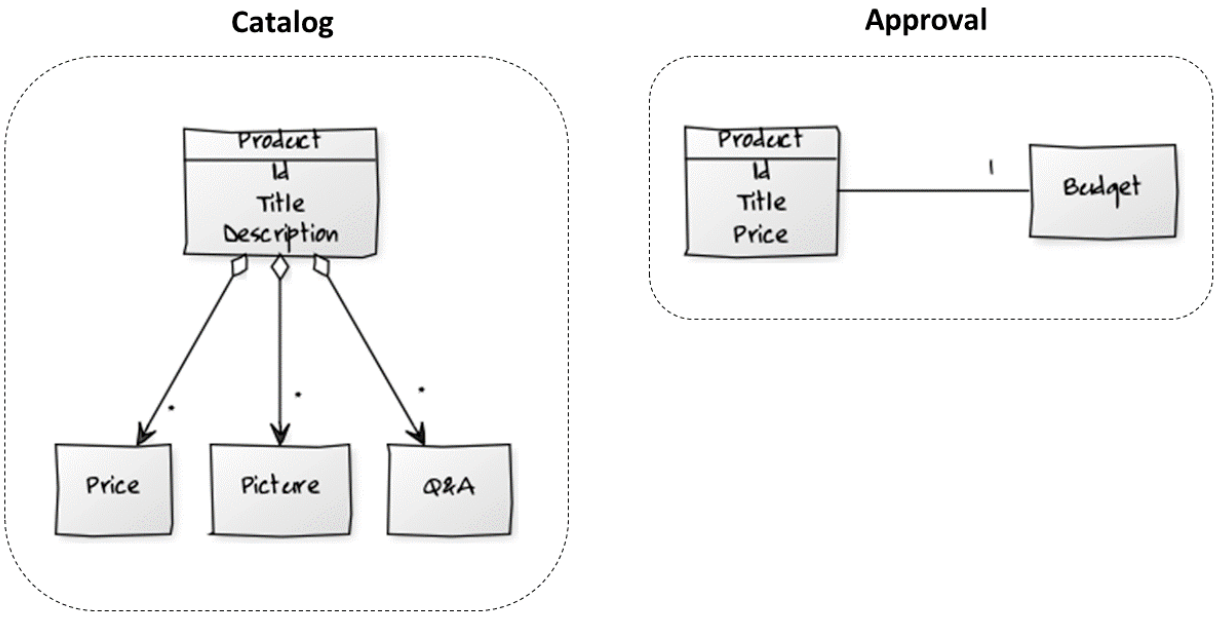
[10] https://www.amazon.de/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215/ref=sr_1_3?ie=UTF8&qid=1551688461&sr=8-3&keywords=ddd
[11] https://nx.dev/
[12] https://pragprog.com/book/swdddf/domain-modeling-made-functional
[13] https://martinfowler.com/bliki/UbiquitousLanguage.html

To recognise domains, it is worth taking a look at the processes in the system. For example, an e-Procurement system that handles the procurement of office supplies could support the following two processes:



We can see that the process steps `Approve Order`, `Request Budget` and `Approve Budget` primarily revolve around organisational hierarchies and the available budget. Managers are principally involved here. By contrast, the process step is fundamentally about employees and products.

Of course, we could argue that products are omnipresent in an e-Procurement system. However, a closer look reveals that the word `product` denotes different items in some of the process steps shown. For example, while a product description is very detailed in the catalogue, the approval process only needs a few key data:



We must distinguish between these two forms of a product in the ubiquitous language that prevails within each domain. 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 everything. These models also have too many interdependencies that make decoupling and subdividing impossible.

We can still relate personal views of the product at a logical level. If the same id on both sides expresses this, it works without technical dependencies.

Thus, each model is valid only within a specific scope. DDD calls this the bounded context[14]. Ideally, each domain has its own bound context. As the next section shows, however, this goal cannot always be achieved when integrating third-party systems.

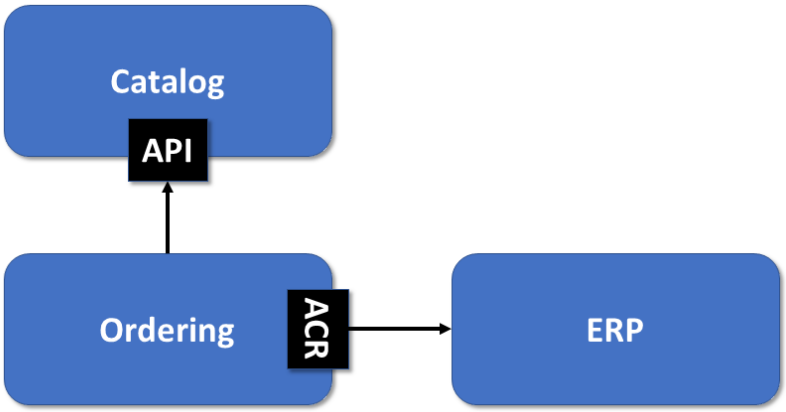
If we proceed with this analysis, we may find the following domains:

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

If you like the process-oriented approach of identifying different domains alongside the vocabulary (entities) and groups of domain experts, you might love Event Storming[15]. At this workshop, domain experts analyse business domains.

Context-Mapping

Although the individual domains are as self-contained as possible, they still have to interact occasionally. In our example, the `ordering` domain for sending orders could access both the `catalogue` domain and a connected ERP system:



A context map determines how these domains interact. In principle, `Ordering` and `Booking` could share the common model elements. In this case, however, we must ensure that modifying one does not cause inconsistencies.

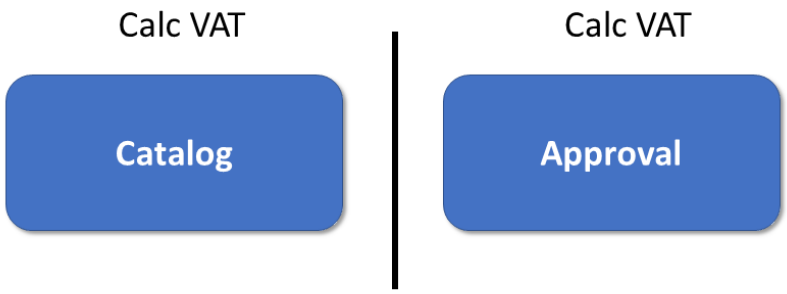
One domain can easily use the other. In this case, however, it is unclear how much power each is entitled to. Can the consumer impose specific changes on the provider and insist on backward compatibility? Or must the consumer be satisfied with what it gets from the provider?

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

Strategic design defines further strategies for the relationship between consumers and providers. In our example, Catalog offers an API to prevent changes in the domain from forcibly affecting consumers. Since order has little impact on the ERP system, it uses an anti-corruption layer (ACR) for access. If something changes in the ERP system, it only needs an update.

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 sub-domains.

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



At first sight, this seems awkward because it leads to code redundancies, breaking the DRY principle (don't repeat yourself). Nevertheless, it can come in handy because it prevents dependency on a shared library. Although preventing redundant code is important, limiting dependencies is vital because each dependency defines a contract, and contracts are hard to change. Hence, it's good first to evaluate whether an additional dependency is truly needed.

As mentioned, each domain should have a bounded context. Our example has an exception: If we have to respect an existing system like the ERP system, it might contain several bounded contexts not isolated from each other.

Conclusion

Strategic design is about identifying self-contained (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[16]-based monorepo.

Learn more about this and further architecture topics regarding Angular and huge enterprise as well as industrial solution in our advanced Online Workshop[17]:

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

Advanced Angular Workshop

Save your ticket[18] for one of our **online or on-site** workshops now or request a company workshop[19] (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.

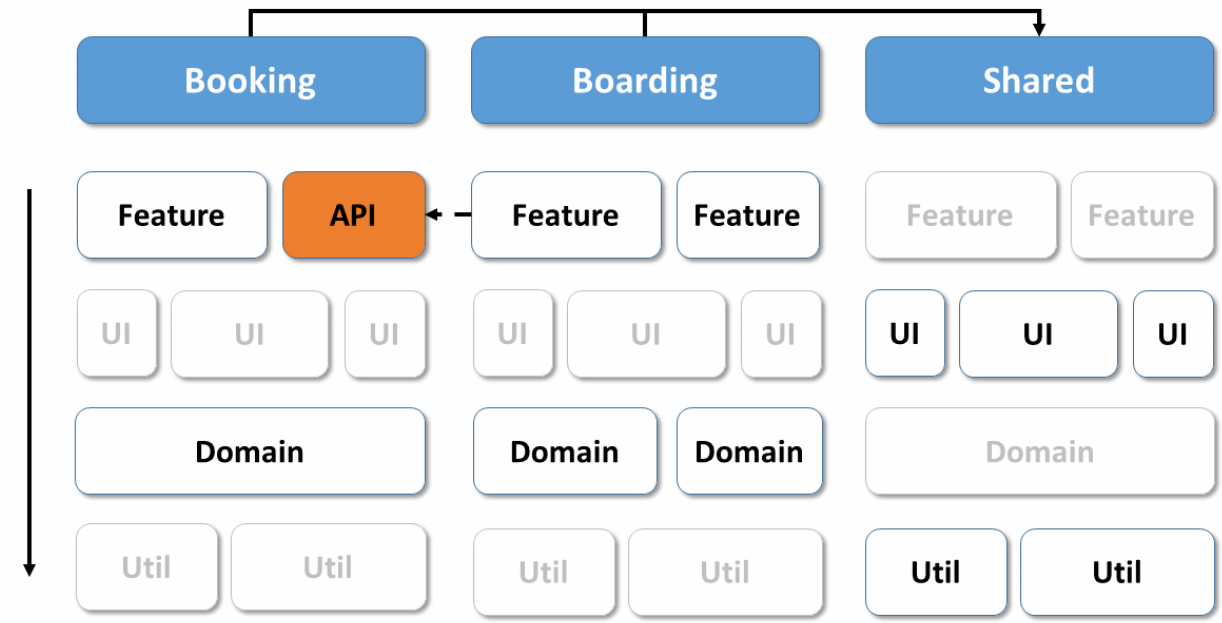
For this, you can subscribe to our newsletter[20] and/ or follow the book's author on Twitter[21].

[18] https://www.angulararchitects.io/en/angular-workshops/
[19] https://www.angulararchitects.io/en/contact/
[20] https://www.angulararchitects.io/en/subscribe/
[21] https://twitter.com/ManfredSteyer

Implementing Strategic Design with Nx Monorepos

In the previous chapter, I presented strategic design which allows a software system's subdivision into self-contained (sub-)domains. This chapter explores these domains' implementation with Angular and an Nx[22]-based monorepo.

The used architecture follows this architecture matrix:



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

If you want to look at the underlying case study[23], you can find the source code here[24]

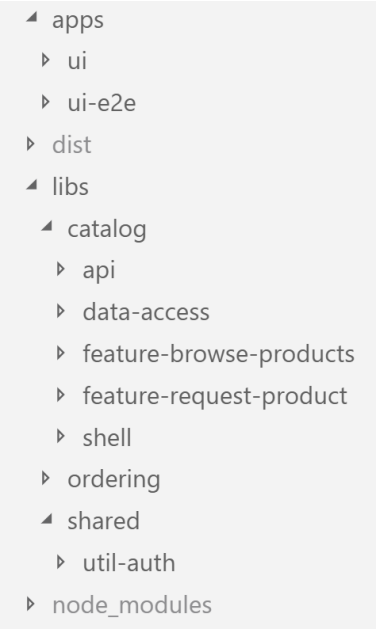
I'm following recommendations the Nx team recently described in their free e-book about Monorepo Patterns[25]. Before this was available, I used similar strategies. To help establish a common vocabulary and standard conventions in the community, I am now aligning with the Nx team's ideas and terms.

[22] https://nx.dev/
[23] https://github.com/manfredsteyer/strategic-design
[24] https://github.com/manfredsteyer/strategic-design
[25] https://go.nrwl.io/angular-enterprise-monorepo-patterns-new-book

Implementation with Nx

We use an Nx [Nx]-based workspace to implement the defined architecture. This workspace is an extension for Angular CLI, which helps to break down a solution into different applications and libraries. Of course, this is one of several possible approaches. Alternatively, one could implement each domain as a completely separate solution, a so-called micro-app approach.

The solution shown here puts all applications into one `apps` folder, while grouping all the reusable libraries by the respective domain name in the `libs` folder:



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

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

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

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

This command downloads a script which creates your workspace.

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

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

Categories for Libraries

In their free e-book about Monorepo Patterns[26], Nrwl[27] – the company behind Nx – use the following categories for libraries:

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

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

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

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

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

The categories are used as a prefix for the individual library folders, thus helping maintain an overview. Libraries within the same category are presented next to each other in a sorted overview.

Public APIs for Libraries

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

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

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

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

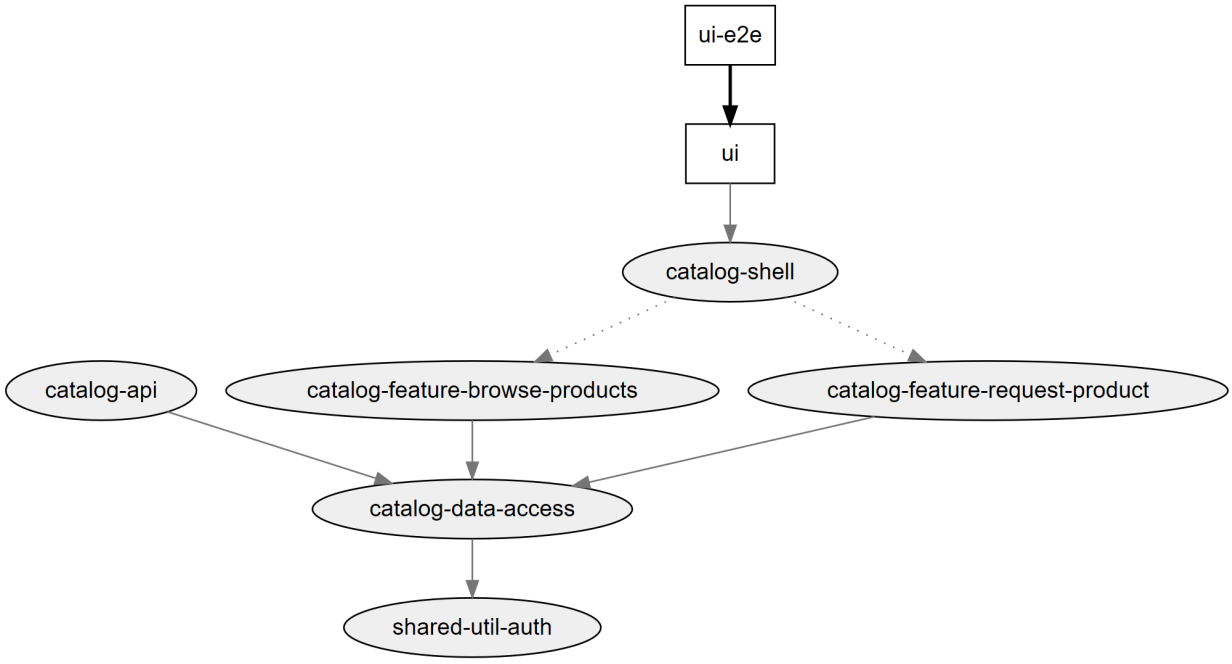
However, the private part can be changed at will, as long as the public part stays the same.

Check Accesses between libraries

Minimising the dependencies between individual libraries helps maintainability. This goal can be checked graphically by Nx with the `dep-graph` npm script:

```
npm run dep-graph
```

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



Access Restrictions for a Solid Architecture

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

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

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

The categorisation of libraries has limitations. A `shell` only accesses `features` and a `feature` accesses `data-access` libraries. Anyone can access `utils`.

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

Tagging Libraries

The file `nx.json` defines the tags for our libraries. Nx generates the file:

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

Alternatively, these tags can be specified when setting up the applications and libraries.

According to a suggestion from the mentioned e-book about Monorepo Patterns[28], the domains get the prefix `scope`, and the library types receive the prefix `kind`. Prefixes of this type are intended to improve readability and can be freely assigned.

Defining Linting Rules based upon Tags

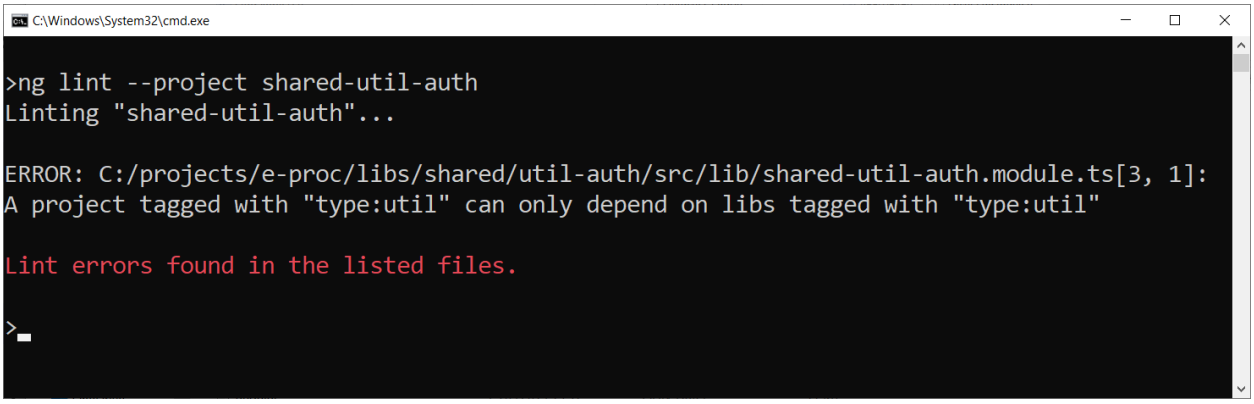
To enforce access restrictions, Nx comes with its own linting rules. As usual, we configure these rules within `tslint.json`:

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

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

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

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



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

> Hint: Consider using Git Hooks, e. g. by leveraging Husky[29], which ensures that only code not violating your linting rules can be pushed to the repository.

Conclusion

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

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

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

[29] https://github.com/typicode/husky

Tactical Domain-Driven Design with Angular and Nx

The previous chapters showed how to use the ideas of strategic design with Angular and Nx. This chapter builds upon the outlined ideas and describes further steps to respect tactical design too.

The case study used in this chapter is a travel web application which has the following sub-domains:

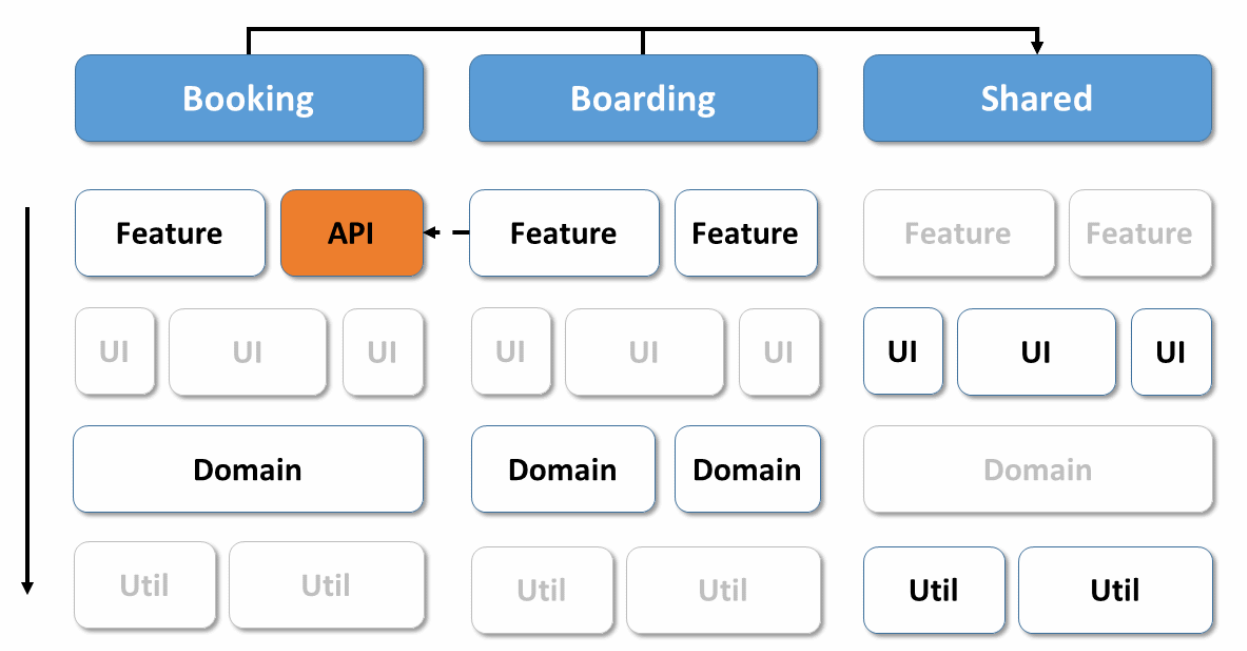
The source code[30] of this case study can be found here[31].

Let's turn to structuring our application with tactical design.

Domain Organisation using Layers

We use column subdivisions ("swimlanes") for domains. We can organise these domains with layers of libraries which leads to row subdivisions:

[30] https://github.com/manfredsteyer/angular-ddd
[31] https://github.com/manfredsteyer/angular-ddd



Note how each layer can consist of one or more **libraries**

> Using layers is a traditional way of organising a domain. There are alternatives like hexagonal architectures or clean architecture.

What about shared functionality?

For those aspects that are *shared* and used across domains, we use an additional `shared` swimlane. Shared libraries can be useful. Consider, for example, shared libraries for authentication or logging.

> Note: the `shared` swimlane corresponds to the Shared Kernel proposed by DDD and also includes technical libraries to share.

How to prevent high coupling?

As discussed in the previous chapter, access constraints define which libraries can use/depend upon other libraries. Typically, each layer is only allowed to communicate with underlying layers. Cross-domain access is allowed only with the `shared` area. The benefit of using these restrictions is loose coupling and thus increased maintainability.

> To prevent too much logic from being put into the `shared` area, the approach presented uses APIs that publish building blocks for other domains. This approach corresponds to Open Services in DDD.

We can see the following two characteristics in the shared part:

- As the greyed-out blocks indicate, most `util` libraries are in the `shared` area, primarily because we use aspects such as authentication or logging across systems.
- The same applies to general UI libraries that ensure a system-wide look and feel.

What about feature-specific functionality?

Notice that domain-specific `feature` libraries, however, are not in the `shared` area. Feature-related code should be within its own domain.

While developers may share feature code (between domains), this practice can lead to shared responsibilities, more coordination effort, and breaking changes. Hence, it should only be shared sparingly.

Code Organisation

Based on Nrwl.io's Enterprise MonoRepository Patterns[32], I distinguish between five categories of layers or libraries:

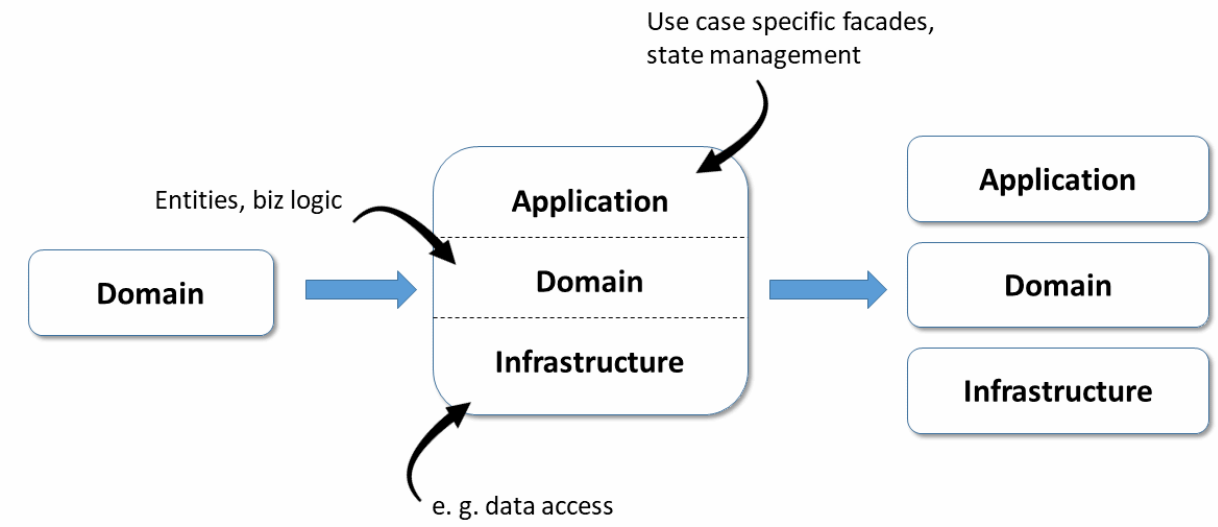
Category	Description	Exemplary content
feature	Contains components for a use case.	search-flight component
ui	Contains so-called "dumb components" that are use case-agnostic and thus reusable.	date-time component, address component, address pipe
api	Exports building blocks from the current subdomain for others.	Flight API
domain	Contains the **domain models** (classes, interfaces, types) that are used by the domain (swimlane)	
util	Include general utility functions	formatDate

This complete architectural matrix is initially overwhelming. But after a brief review, almost all the developers I've worked with agreed that the code organisation facilitates code reuse and future features.

Isolate the Domain

To isolate the domain logic, we hide it behind facades which represent it in a use case-specific manner:

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



This facade can also deal with state management.

While facades are currently popular in the Angular environment, this idea correlates beautifully with DDD (where they are called application services).

It is crucial to architecturally separate *infrastructure requirements* from the actual domain logic.

In an SPA, infrastructure concerns are – generally – asynchronous communication with the server and data exchanges. Maintaining this separation results in three additional layers:

- the application services/facades,
- the actual domain layer, and
- the infrastructure layer.

Of course, we can package these layers in their own libraries. For the sake of simplicity, it is also possible to store them in a single subdivided library. This subdivision makes sense if these layers are ordinarily used together and only exchanged for unit tests.

Implementations in a Monorepos

Once we have determined our architecture's components, we consider how to implement them in Angular. A common approach by Google is monorepos. Monorepos are a code repository of all the libraries in a software system.

While a project created with the Angular CLI can nowadays serve as a monorepo, the popular tool Nx[33] offers additional features which are particularly valuable for large enterprises. These include

[33] https://nx.dev/

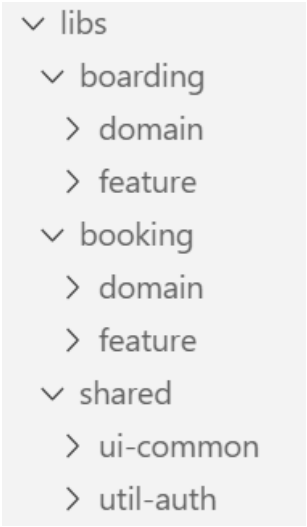
the previously discussed ways to introduce access restrictions between libraries[34]. These restrictions prevent each library from accessing one another, resulting in a highly coupled overall system.

One instruction suffices to create a library in a monorepo:

```
ng generate library domain --directory boarding --buildable
```

> You use `ng generate library` instead of `ng generate module`, requiring no extra effort. However, you get a cleaner structure, improved maintainability, and less coupling.

The switch `directory` provided by Nx specifies an optional subdirectory for the libraries, so they can be **grouped by domain**:

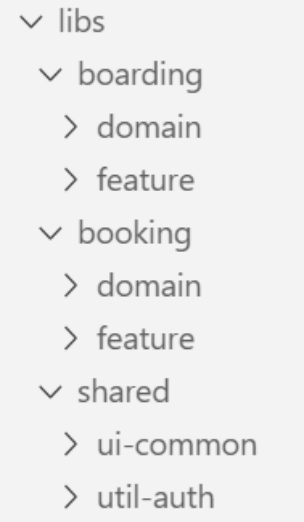


The second switch, `buildable`, enables incremental builds as described below.

The libraries' names reflect the layers. If a layer has multiple libraries, it makes sense to use these names as a prefix, producing names such as `feature-search` or `feature-edit`.

This example divides the domain library into the three further layers mentioned to isolate the exact domain model:

[34] https://www.softwarearchitekt.at/aktuelles/sustainable-angular-architectures-2/



Builds within a Monorepo

By looking at the git commit log, Nx can identify which libraries are *affected by the latest code changes*.

This change information recompiles only the **affected** libraries or just run their **affected** tests, saving time on large systems entirely stored in a repository.

Entities and your Tactical Design

Tactical design provides many ideas for structuring the domain layer. At the centre of this layer, there are **entities**, the reflecting real-world domain, and constructs.

The following listing shows an enum and two entities that conform to the usual practices of object-oriented languages such as Java or C#.

```
public enum BoardingStatus {
  WAIT_FOR_BOARDING,
  BOARDED,
  NO_SHOW
}

public class BoardingList {

  private int id;
  private int flightId;
  private List<BoardingListEntry> entries;
```

```
12    private boolean completed;
13
14    // getters and setters
15
16    public void setStatus (int passengerId, BoardingStatus status) {
17      // Complex logic to update status
18    }
19
20  }
21
22  public class BoardingListEntry {
23
24    private int id;
25    private boarding status status;
26
27    // getters and setters
28  }
```

As usual in OO-land, these entities use information hiding to ensure their state remains consistent. You implement this with private fields and public methods that operate on them.

These entities encapsulate data and business rules. The `setStatus` method indicates this circumstance. DDD defines so-called domain services only in cases where you cannot meaningfully accommodate business rules in an entity.

> DDD frowns upon e ntities that only represent data structures. The community calls them devaluing bloodless (anaemic)[35].

Tactical DDD with Functional Programming

From an object-oriented point of view, the previous approach makes sense. However, with languages such as JavaScript and TypeScript, object-orientation is less critical.

Typescript is a multi-paradigm language in which functional programming plays a major role. Here are some books which explore functional DDD:

- Domain Modeling Made Funcitonal[36],
- Functional and Reactive Domain Modeling[37].

[35] https://martinfowler.com/bliki/AnemicDomainModel.html
[36] https://pragprog.com/book/swdddf/domain-modeling-made-functional
[37] https://www.amzn.com/1617292249

Functional programming splits the previously considered entity model into data and logic parts. Domain-Driven Design Distilled[38] which is one of the standard works for DDD and primarily relies on OOP, also concedes that this rule change is necessary in the world of FP:

```
export type BoardingStatus = 'WAIT_FOR_BOARDING' | 'BOARDED' | 'NO_SHOW' ;

export interface BoardingList {
    readonly id: number;
    readonly flightId: number;
    readonly entries: BoardingListEntry [];
    readonly completed: boolean;
}

export interface BoardingListEntry {
    readonly passengerId: number;
    readonly status: BoardingStatus;
}
```

```
export function updateBoardingStatus (
                    boardingList: BoardingList,
                    passengerId: number,
                    status: BoardingStatus): Promise <BoardingList> {

        // Complex logic to update status

}
```

The entities also use public properties here. This practice is common in FP. Excessive use of getters and setters, which only delegate to private properties, is often ridiculed.

More interesting, however, is how the functional world avoids inconsistent states. The answer is amazingly simple: Data structures are preferentially **immutable**. The keyword `read-only` in the example shown emphasises this.

Any part of the programme that seeks to change such objects has first to clone it. If other parts of the programme have first validated an object for their purposes, they can assume that it remains valid.

> A pleasant side-effect of using immutable data structures is that it optimises change detection performance. *Deep-comparisons* are no longer required. Instead, a *changed* object is a new instance, and thus the object references are no longer the same.

[38] https://www.amzn.com/0134434420

Tactical DDD with Aggregates

To keep track of the components of a domain model, tactical DDD combines entities into aggregates. In the previous example, `BoardingList` and `BoardingListEntry` form such an aggregate.

The state of an aggregate's components must be consistent as a whole. For instance, in the above example, we could specify that `completed` in `BoardingList` may only be `true` if no `BoardingListEntry` has the status `WAIT_FOR_BOARDING`.

Furthermore, different aggregates may not reference each other through object references. Instead, they can use IDs. Using IDs should prevent unnecessary coupling between aggregates. Large domains can thus be broken down into smaller groups of aggregates.

> Domain-Driven Design Distilled[39] suggests making aggregates as small as possible. First of all, consider each entity as an aggregate and then merge aggregates that need to be changed together without delay.

Facades

Facades[40] (aka applications services) are used to represent the domain logic in a use case-specific way. They have several advantages:

- Encapsulating complexity
- Taking care of state management
- Simplified APIs

Independent of DDD, this idea has been prevalent in Angular for some time.

For our example, we could create the following facade:

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

    private notifier = new BehaviorSubject<Flight[]>([]);
    public  flights$ = this.notifier.asObservable();

    constructor(private flightService: FlightService) { }

    search(from: string, to: string, urgent: boolean): void {
        this.flightService
```

[39] https://www.amzn.com/0134434420
[40] https://go.nrwl.io/angular-enterprise-monorepo-patterns-new-book

```
11              .find(from, to, urgent)
12              .subscribe (
13                (flights) => this.notifier.next(flights),
14                (err) => console.error ('err', err);
15              );
16      }
17  }
```

Note the use of RxJS and observables in the facade. The facade can auto-deliver updated flight information when conditions change. Facades can introduce Redux transparently and `@ngrx/store` later when needed, without affecting any external application components.

> For the consumer of the facade it is irrelevant whether it manages the state by itself or by delegating to a state-management library.

Stateless Facades

While it is good practice to make server-side services stateless, this goal is frequently not performant for services in web/client-tier.

A web SPA has a state, and that's what makes it user-friendly .

To prevent UX issues, Angular applications avoid repeatedly reloading all the information from the server. Hence, the facade outlined holds the loaded flights (within the observable discussed before).

Domain Events

Besides performance improvements, using observables provides a further advantage. Observables allow further decoupling since the sender and receiver do not have to know each other directly.

This structure also perfectly fits DDD, where the use of **domain events** are now part of the architecture. If something interesting happens in one part of the application, it sends a domain event, and other application parts can react.

In the shown example, a domain event could indicate that a passenger is now `BOARDED`. If this is relevant for other parts of the system, they can execute specific logic.

> For Angular developers familiar with Redux or Ngrx: We can represent domain events as *dispatched actions*.

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

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

To ease this task, you can use our Nx plugin `@angular-architects/ddd`. It provides the following features:

- Generating domains with domain libraries including a facades, models, and data services
- Generating feature libraries including a feature components using the facades
- Adding linting rules for access restrictions between domains as proposed by Nrwl
- Adding linting rules for access restrictions between layers as proposed by Nrwl (supports tslint and eslint)
- Optionally generates skeleton for NGRX and integrates it into the DDD design (`--ngrx` switch)

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

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

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

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

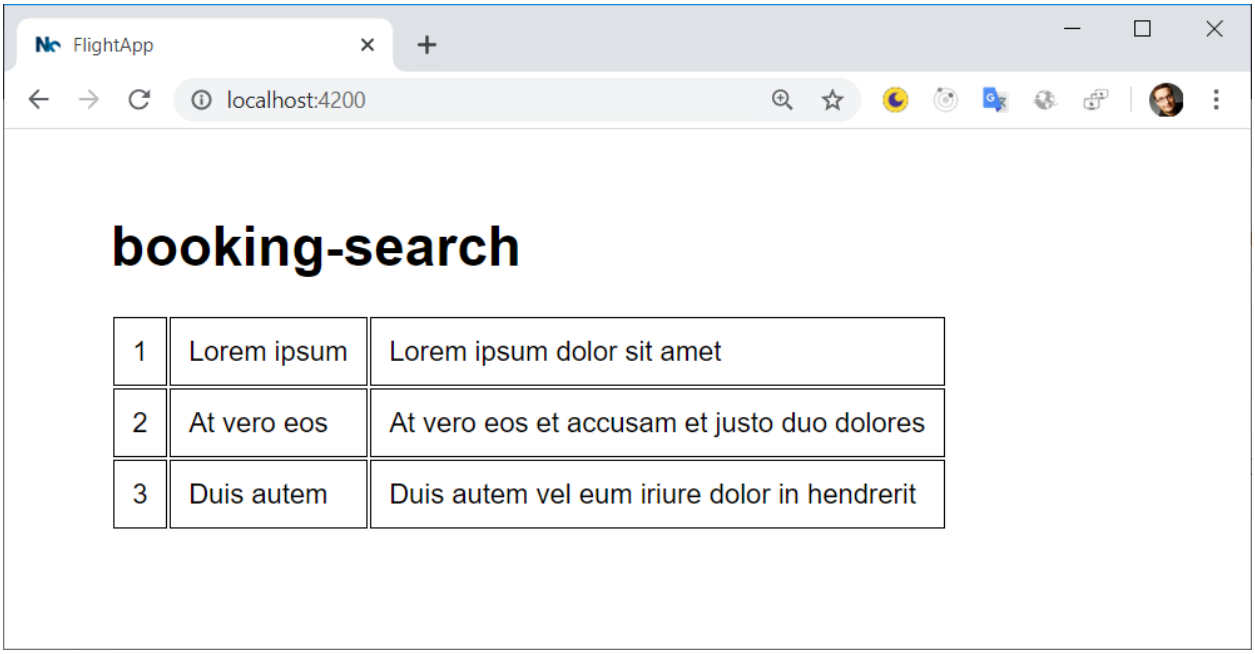
For NGRX support, just add the `--ngrx` switch:

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

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

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

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



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

Conclusion

Modern single-page applications (SPAs) are often more than just recipients of data transfer objects (DTOs). They often have significant domain logic which adds complexity. Ideas from DDD help developers to manage and scale with the resulting complexity.

A few rule changes are necessary due to the object-functional nature of TypeScript and prevailing customs. For instance, we usually use immutables and separate data structures from the logics operating on them.

The implementation outlined here bases upon the following ideas:

- The use of monorepos with multiple libraries grouped by domains helps to build the basic structure.
- Access restrictions between libraries prevent coupling between domains.
- Facades prepare the domain model for individual use cases and maintain the state.
- If needed, Redux can be used behind the facade without affecting the rest of the application.

If you want to see all these topics in action, check out our Angular architecture workshop[41].

[41] https://www.softwarearchitekt.at/schulungen/advanced-angular-enterprise-anwendungen-und-architektur/

Short Note: Incremental Builds to Speed up Your CI Process

Having a huge solution leads to increased build times. This is a burden for many teams. To deal with this situation, Nx provides a great feature: Incremental Builds and Tests. That means, that only the changed libraries are rebuilt and retested in terms of unit and e2e tests.

To make use of incremental builds, make sure you create your libs with the `--buildable` or `--publishable` flag. The latter one is only needed if you plan to publish your libraries to an NPM registry. In all other cases go with `--buildable` because its faster:

```
ng g lib my-lib --buildable
```

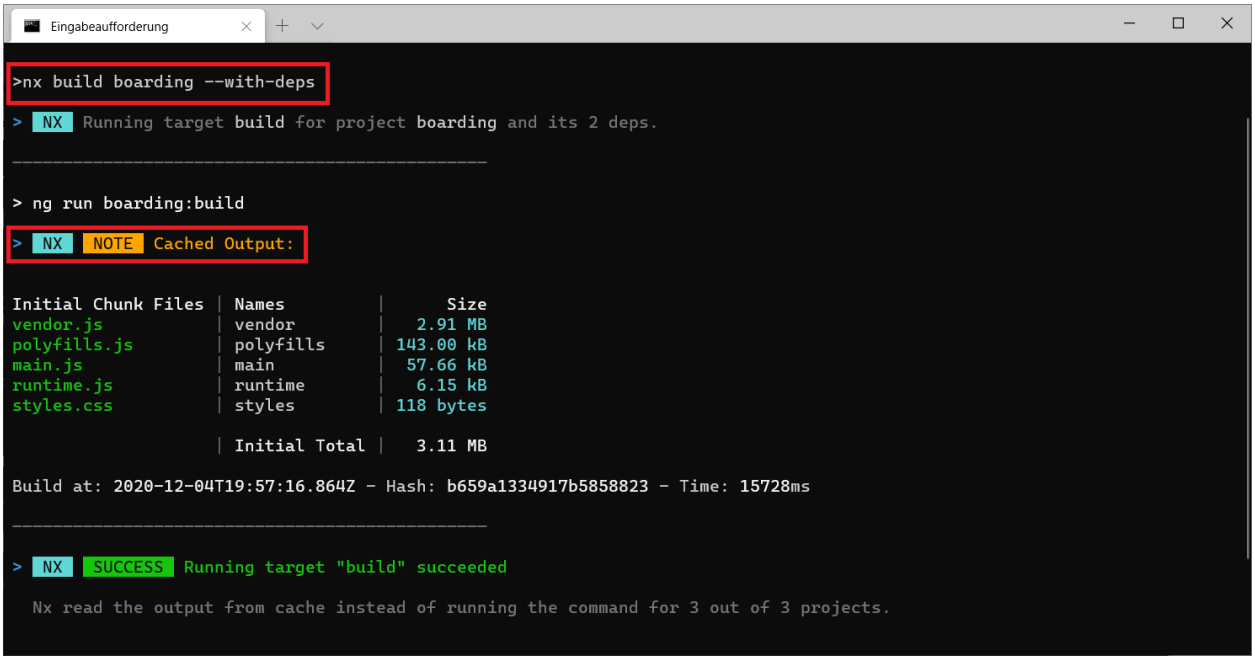
For incremental builds, you need the Nx CLI. Install it via your package manager:

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

Then call `nx build` with the `with-deps` switch:

```
nx build my-app --with-deps
```

This new switch compiles your libraries separately so that the individual libraries and applications can be cached. If you repeat this command, just the changed libraries are rebuilt – the rest is taken out of your cache. Obviously, this speeds up your whole build:



The same is done for executing unit tests (`nx test`), e2e tests (`nx e2e`), and linting (`nx lint`). The `nx.json` in your project's root defines, which of those tasks use the cache:

```
"tasksRunnerOptions": {
    "default": {
        "runner": "@nrwl/workspace/tasks-runners/default",
        "options": {
        "cacheableOperations": ["build", "lint", "test", "e2e"]
        }
    }
},
```

By default, the cache is on for all of them. If you wonder where to find the cache, have a look to your `node_modules/.cache/nx` directory. <!– ## Using a Distributed Build Cache

If you want further speed up your whole build and CI process, you can use a distributed build cache. In this case you can benefit from tasks your colleagues already executed. Nothing needs to be done more than once within the whole team.

The smart people behind Nx provide an official solution for this called **Nx Cloud**.

Besides this, you can also implement your own cache. This is exactly what the open source project @apployees-nx/level-task-runner[42] does. It allows using several databases for caching. I've successfully used it with mongoDB, Redis, PostgreSQL, and MySQL. However, please keep in mind that this is an **unofficial** solution that might be affected by breaking changes in the future. Also, Nx Cloud provides some **further features** like a cool dashboard.

[42] https://www.npmjs.com/package/@apployees-nx/level-task-runner

Here, I'm showing how to use `@apployees-nx/level-task-runner` together with a **mongoDB**. For this, you need to install `@apployees-nx/level-task-runner` and the mongoDB driver:

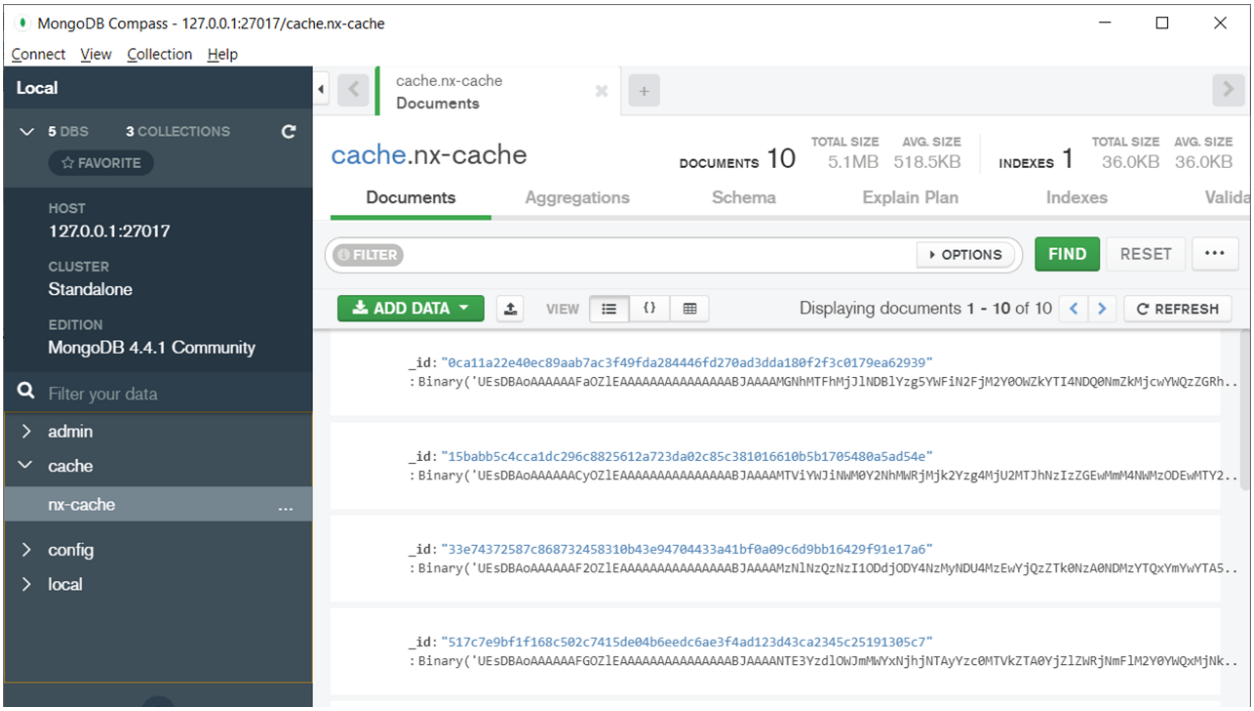
```
npm install @apployees-nx/level-task-runner -D
npm install mongodown -D
```

Then, you can adjust your `taskRunnerOptions` within `nx.json` to use the installed task running together with your mongoDB:

```
"tasksRunnerOptions": {
    "default": {
        "runner": "@apployees-nx/level-task-runner",
        "options": {
            "cacheableOperations": ["build", "test", "lint", "e2e"],
            "levelTaskRunnerOptions": {
                "driver": "mongodown",
                "host": "127.0.0.1",
                "port": 27017,
                "name": "cache",
                "collection": "nx-cache"
            }
        }
    }
},
```

After calling `nx build my-app --with-deps` you find several key/values pairs in the defined mongoDB collection:

The keys are the hash values of all the files in a given library and the values are the cached build, test, or linting results:


MongoDB Compass - 127.0.0.1:27017/cache.nx-cache
Connect View Collection Help
Local
5 DBS 3 COLLECTIONS
FAVORITE
HOST
127.0.0.1:27017
CLUSTER
Standalone
EDITION
MongoDB 4.4.1 Community
Filter your data
admin
cache
nx-cache
config
local
cache.nx-cache
Documents
cache.nx-cache
DOCUMENTS 10
TOTAL SIZE 5.1MB
AVG. SIZE 518.5KB
INDEXES 1
TOTAL SIZE 36.0KB
AVG. SIZE 36.0KB
Documents
Aggregations
Schema
Explain Plan
Indexes
FILTER
OPTIONS
FIND
RESET
ADD DATA
VIEW
Displaying documents 1 - 10 of 10
REFRESH
_id: "0ca11a22e40ec89aab7ac3f49fda284446fd270ad3dda180f2f3c0179ea62939"
: Binary('UEsDBAoAAAAAAFaOZlEAAAAAAAAAAAAAAAABJAAAAMGNhMTFhMjJlNDBlYzg5YWFiN2FjM2Y0OWZkYTI4NDQ0NmZkMjcwYWQzZGRh..
_id: "15babb5c4cca1dc296c8825612a723da02c85c381016610b5b1705480a5ad54e"
: Binary('UEsDBAoAAAAAACyOZlEAAAAAAAAAAAAAAAABJAAAAMTViYWJiNWM0Y2NhMWRjMjk2Yzg4MjU2MTJhNzIzZGEwMmM4NWMzODEwMTY2..
_id: "33e74372587c868732458310b43e94704433a41bf0a09c6d9bb16429f91e17a6"
: Binary('UEsDBAoAAAAAAF2OZlEAAAAAAAAAAAAAAAABJAAAAMzNlNzQzNzI1ODdjODY4NzMyNDU4MzEwYjQzZTk0NzA0NDMzYTQxYmYwYTA5..
_id: "517c7e9bf1f168c502c7415de04b6eedc6ae3f4ad123d43ca2345c25191305c7"
: Binary('UEsDBAoAAAAAAFGOZlEAAAAAAAAAAAAAAAABJAAAANTE3YzdlOWJmMWYxNjhjNTAyYzc0MTVkZTA0YjZlZWRjNmFlM2Y0YWQxMjNk..

>

From Domains to Microfrontends

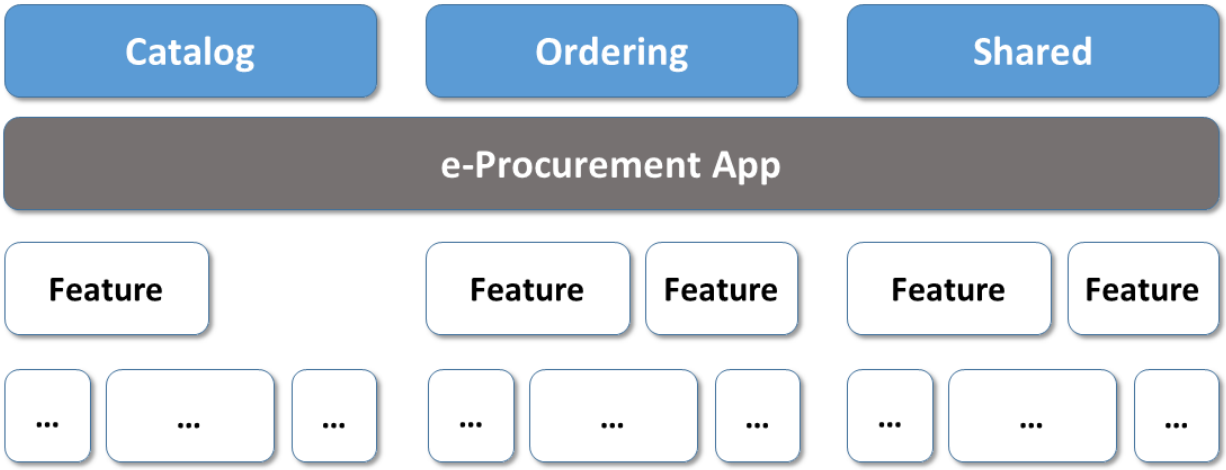
Let's assume you've identified the sub-domains for your system. The next question is how to implement them.

One option is to implement them within a large application – aka a deployment monolith. The second is to provide a separate application for each domain.

Such applications are called microfrontends.

Deployment Monoliths

A deployment monolith is an integrated solution comprising different domains:



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

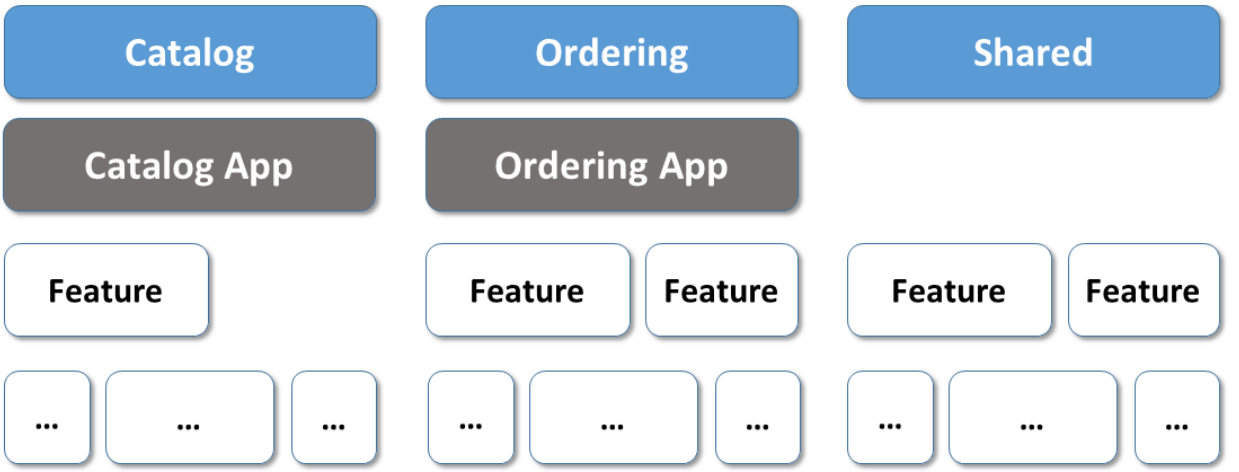
A team responsible for a specific sub-domain must coordinate with other sub-domain teams. They have to agree on overall architecture, the leading framework, and an updated policy for dependencies. Interestingly, you may consider this an advantage.

It is tempting to reuse parts of other domains which may lead to higher coupling and – eventually – to breaking changes. To prevent this, you can use free tools like Nrwl's Nx[43]. For instance, Nx allows you to define access restrictions between the parts of your monorepo to enforce your envisioned architecture and loose coupling.

[43] https://nx.dev/angular

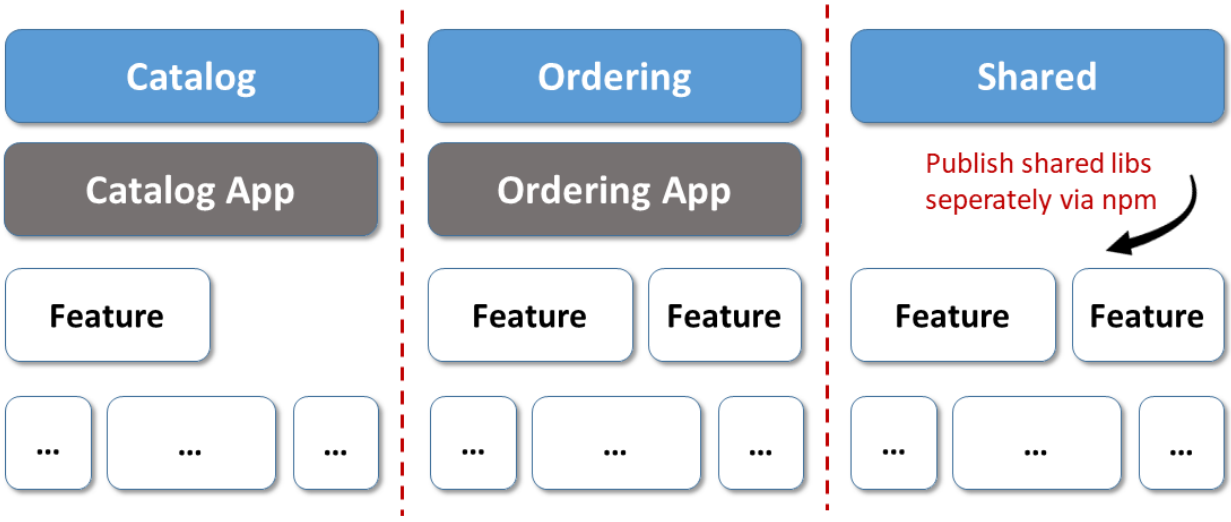
Deployment monoliths, microfrontends, or a mix?

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.



Having several tiny systems decreases complexity.

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 (mono) repository:



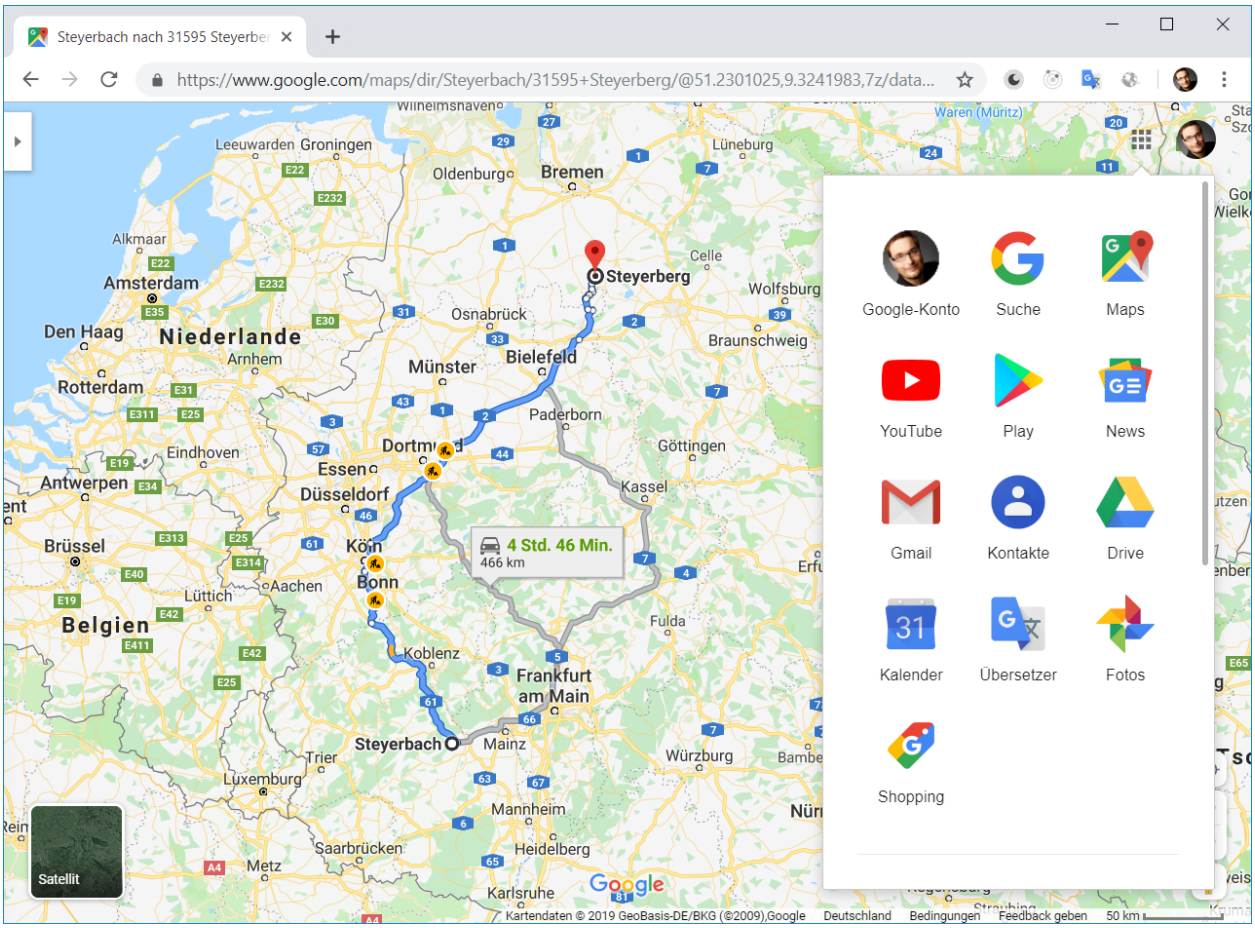
You now have something called microfrontends. Microfrontends allow individual teams to be as autarkic as possible. Each team can choose their architectural style, their technology stack, and they can even decide when to update to newer framework versions. They can use "the best technology" for the requirements given within the current sub-domain.

The option to use their framework and architecture is useful 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.

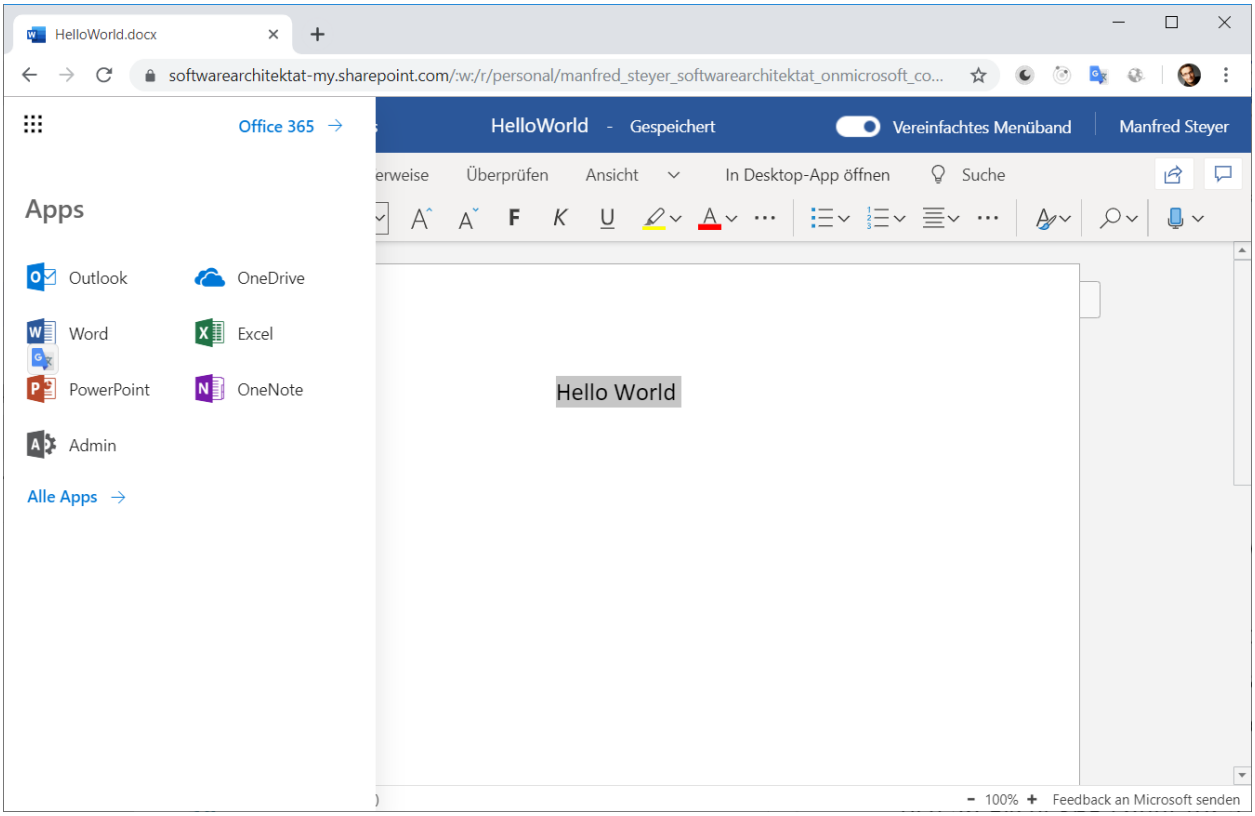
However, this has costs. Now you have to deal with shipping your shared libraries via npm, and this includes versioning which can lead to version conflicts.

UI Composition with Hyperlinks

You have to find ways to integrate the different applications into one large system for your users. Hyperlinks are one simple way to accomplish this:



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



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

This approach has several advantages:

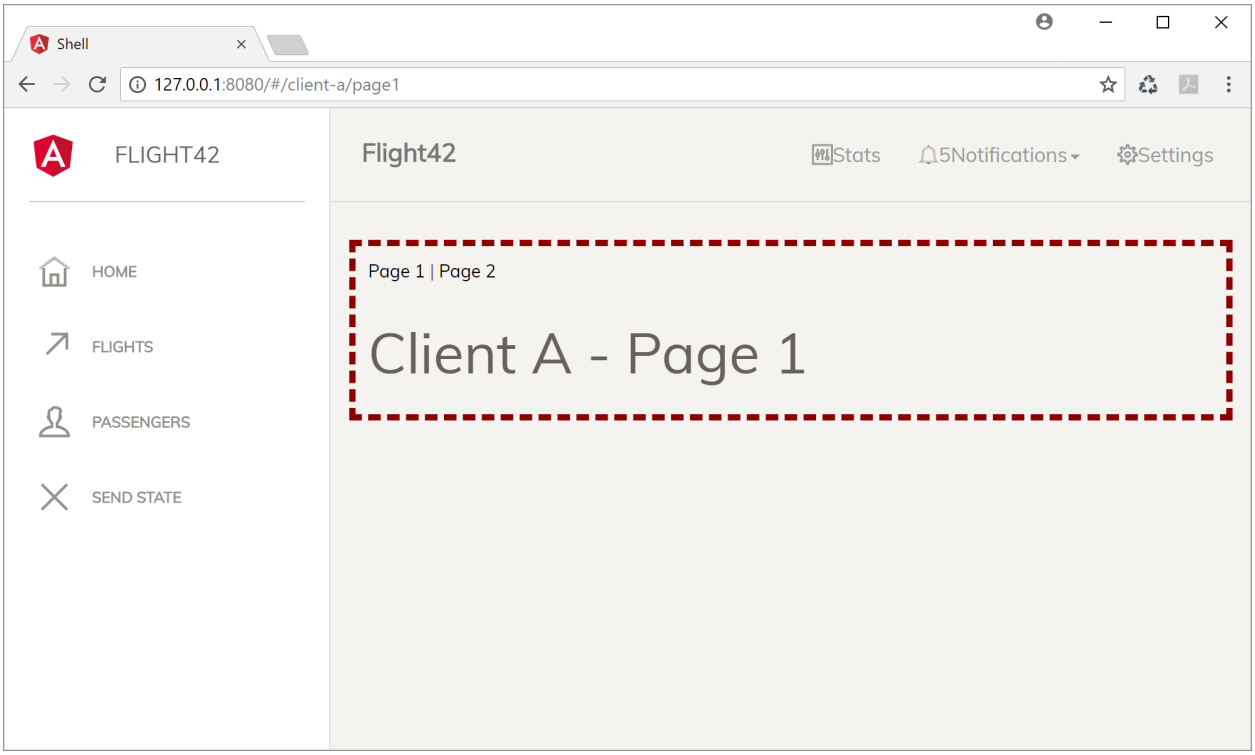
- It is simple
- It uses SPA frameworks as intended
- We get optimised bundles per domain

However, there are some disadvantages:

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

UI Composition with a Shell

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



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

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

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

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

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

You can find a library that compensates most of the disadvantages of iframes for intranet applications here[44].

[44] https://www.npmjs.com/package/@microfrontend/common

The shell approach has the following advantages:

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

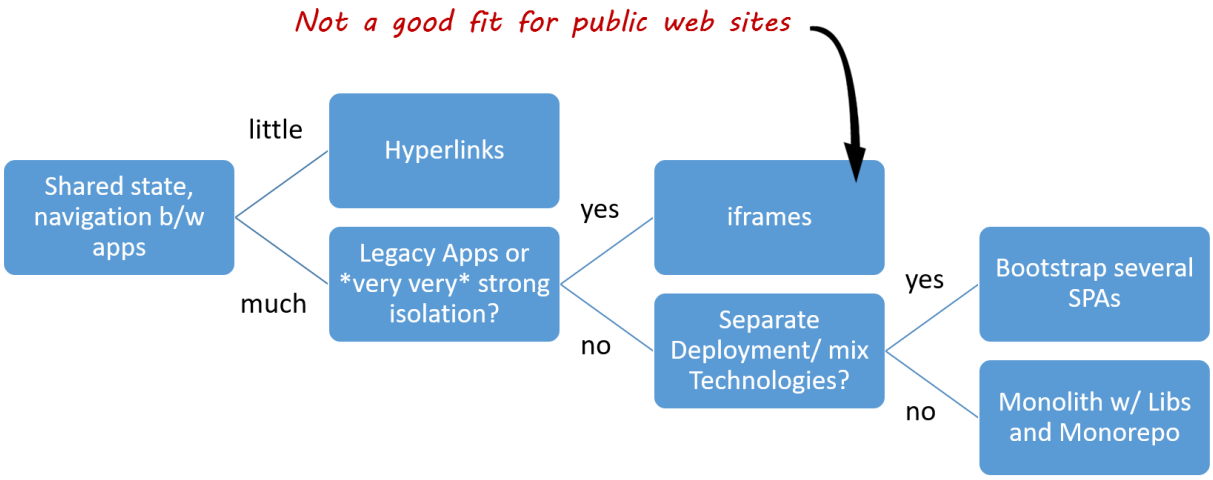
The disadvantages are:

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

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:



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

Conclusion

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

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

The Microfrontend Revolution: Using Module Federation with Angular

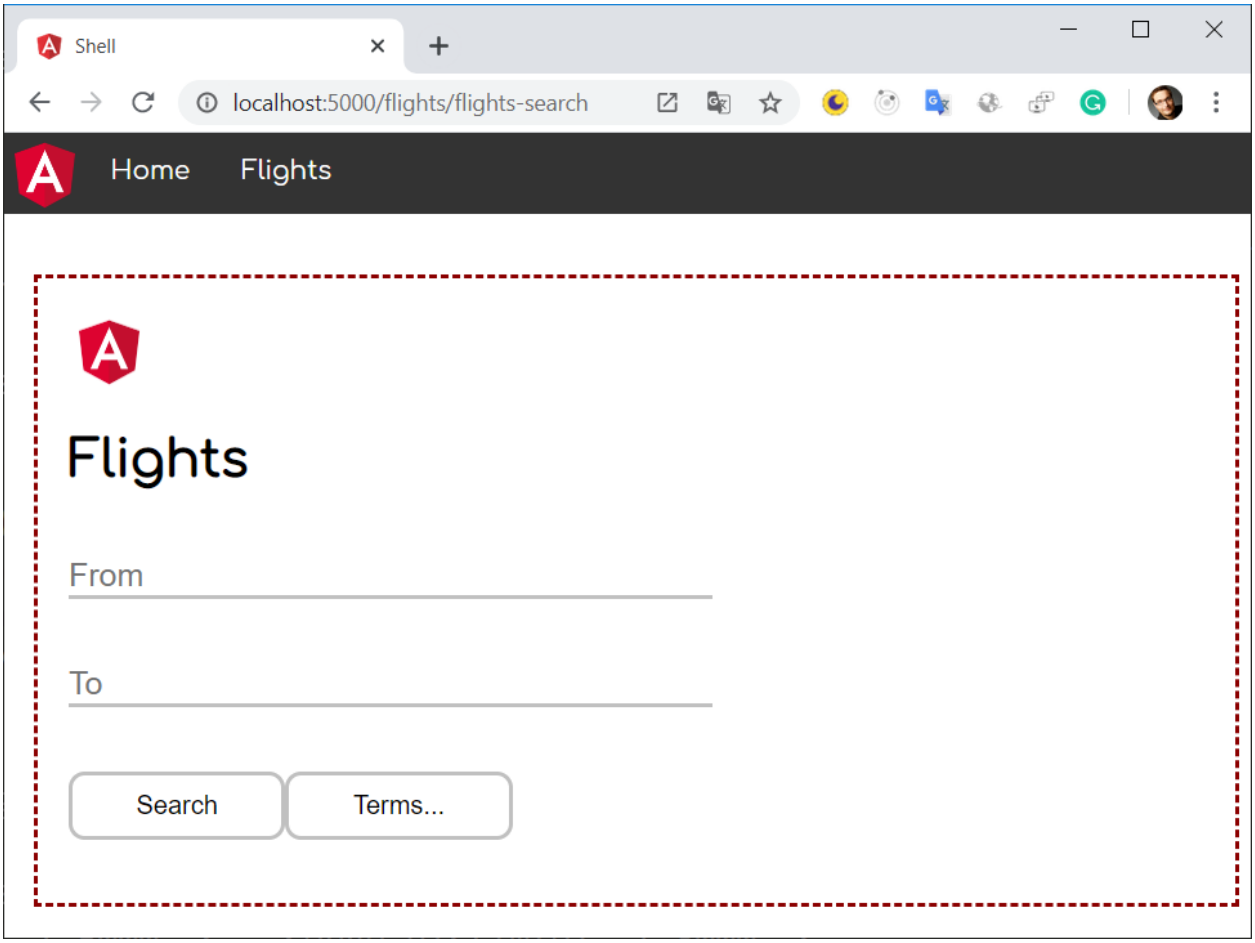
Until now, when implementing microfrontends, you had to dig a little into the bag of tricks. One reason is surely that current build tools and frameworks do not know this concept. Webpack 5, which is currently in BETA, will initiate a change of course here.

It allows an approach implemented by the webpack contributor Zack Jackson. It's called Module Federation and allows referencing program parts not yet known at compile time. These can be self-compiled microfrontends. In addition, the individual program parts can share libraries with each other, so that the individual bundles do not contain any duplicates.

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

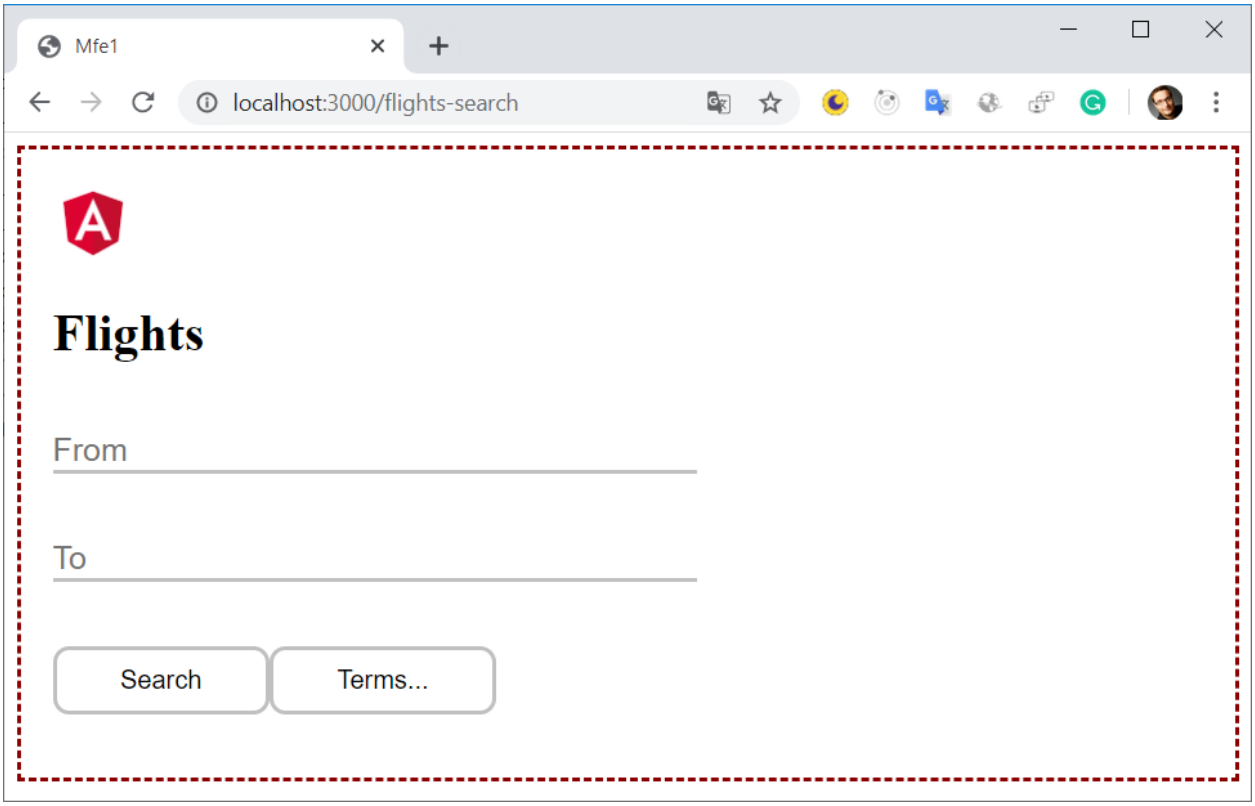
Example

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



Shell

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



Microfrontend without Shell

The source code[45] of the used example can be found in my GitHub account[46].

Getting started

To get started, we need an Angular CLI version supporting webpack 5. As it looks like, Angular CLI 11 which is due in fall 2020 will at least support webpack 5 as an opt-in. When writing this, there was already a beta version (v11.0.0-next.6) allowing to try everything out.

For opting-in, add this segment to your package.json, e. g. in front of the dependency section:

```
"resolutions": {
  "webpack": "5.0.0"
},
```

Then, install your dependencies again using **yarn** (!). Using yarn instead of npm is vital because it uses the shown `resolutions` section to force all installed dependencies like the CLI into using webpack 5.

[45] https://github.com/manfredsteyer/module-federation-with-angular
[46] https://github.com/manfredsteyer/module-federation-with-angular

To make the CLI use yarn by default when calling commands like `ng add` or `ng update`, you can use the following command:

```
ng config cli.packageManager yarn
```

> Please note that the CLI version `v11.0.0-next.6` does currently **not support recompilation in dev mode** when using webpack 5. Hence, you need to restart the dev server after changes. This issue will be solved with one of the upcoming beta versions of CLI 11.

Activating Module Federation for Angular Projects

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

The package @angular-architects/module-federation[47] 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 5000
ng add @angular-architects/module-federation --project mfe1 --port 3000
```

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

The command shown does several things:

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

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

The Shell (aka Host)

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

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

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

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

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

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

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

```
const ModuleFederationPlugin = require("webpack/lib/container/ModuleFederationPlugin\
");

module.exports = {
  output: {
    publicPath: "http://localhost:5000/",
    uniqueName: "shell"
  },
  optimization: {
    // Only needed to bypass a temporary bug
    runtimeChunk: false
  },
  plugins: [
    new ModuleFederationPlugin({
        remotes: {
            'mfe1': "mfe1@http://localhost:3000/remoteEntry.js"
        },
        shared: ["@angular/core", "@angular/common", "@angular/router"]
    })
  ],
};
```

The `remotes` section maps the internal name `mfe1` to the same one defined within the separately compiled microfrontend. It also points to the path where the remote can be found – or to be more precise: to its remote entry. This is a tiny file generated by webpack when building the remote. Webpack loads it at runtime to get all the information needed for interacting with the microfrontend.

While specifying the remote entry's URL that way is convenient for development, we need a more dynamic approach for production. Fortunately, there are several options for doing this. One option is presented in a below sections.

The property `shared` contains the names of libraries our shell shares with the microfrontend.

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

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

The Microfrontend (aka Remote)

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

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

Also, there is a `FlightsModule`:

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

This module has some routes of its own:

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

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

```
const ModuleFederationPlugin = require("webpack/lib/container/ModuleFederationPlugin\
");

module.exports = {
  output: {
    publicPath: "http://localhost:3000/",
    uniqueName: "mfe1"
  },
  optimization: {
    // Only needed to bypass a temporary bug
    runtimeChunk: false
  },
  plugins: [
    new ModuleFederationPlugin({

        // For remotes (please adjust)
        name: "mfe1",
        library: { type: "var", name: "mfe1" },
        filename: "remoteEntry.js",
        exposes: {
            './Module': './projects/mfe1/src/app/flights/flights.module.ts',
        },

        shared: ["@angular/core", "@angular/common", "@angular/router"]
    })
  ],
};
```

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

Standalone-Mode for Microfrontend

For microfrontends that also can be executed without the shell, we need to take care about one tiny thing: Projects configured with the `ModuleFederationPlugin` need to load **shared libraries** using **dynamic imports!**.

The reason is that these imports are asynchronous and so the infrastructure has some time to decide upon which version of shared libraries to use. This is especially important when the shell and the micro frontend provide different versions of the libraries shared. As I describe in this article[48], by default, webpack tries to load the highest compatible version. If there is not such a thing as "the highest compatible version", Module Federation provides several fallbacks. They are also described in the article mentioned.

So that developers are not constantly confronted with this limitation, it is advisable to load the entire application via a dynamic import instead. The entry point of the application – in an Angular CLI project this is usually the `main.ts` file – thus only consists of a single dynamic import:

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

This loads another TypeScript module called `bootstrap.ts`, which takes care of bootstrapping the application:

```
import { AppModule } from './app/app.module';
import { environment } from './environments/environment';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { enableProdMode } from '@angular/core';

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

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));
```

As you see here, the `bootstrap.ts` file contains the very code normally found in `main.ts`.

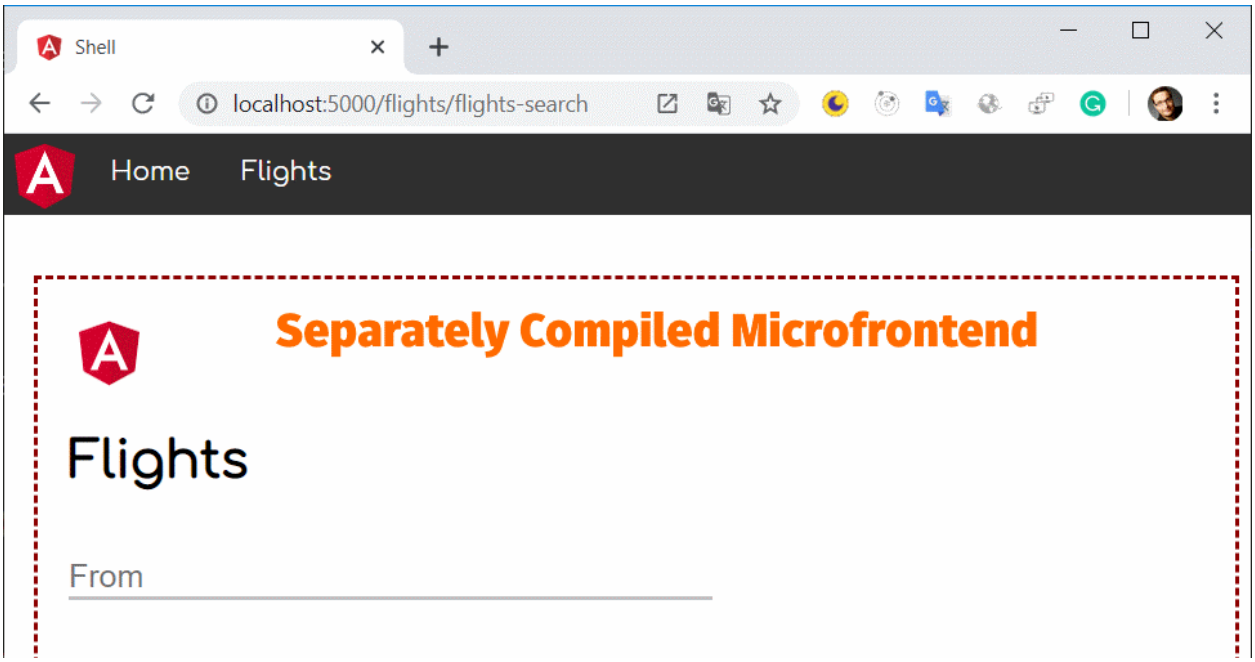
Trying it out

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

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

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

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



Connecting the Shell and the Microfrontend

Hint: To start several projects with one command, you can use the npm package concurrently[49].

Bonus: Loading the Remote Entry

As discussed above, the microfrontend's remote entry can be defined in the shell's webpack configuration. However, this demands us to foresee the microfrontend's URL when compiling the shell.

As an alternative, we can also load the remote entry by referencing it with a script tag:

```
<script src="http://localhost:3000/remoteEntry.js"></script>
```

This script tag can be dynamically created, e. g. by using server side templates or by manipulating the DOM on the client side.

To make this work, we need to switch the `remoteType` in the shell's config to `var`:

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

```
new ModuleFederationPlugin({
    remoteType: 'var',
    [...]
})
```

There are even more dynamic ways allowing you to inform the shell just at runtime how many microfrontends to respect, what's their names and where to find them. The next chapter describes such an approach called Dynamic Module Federation.

Conclusion and Evaluation

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

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

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

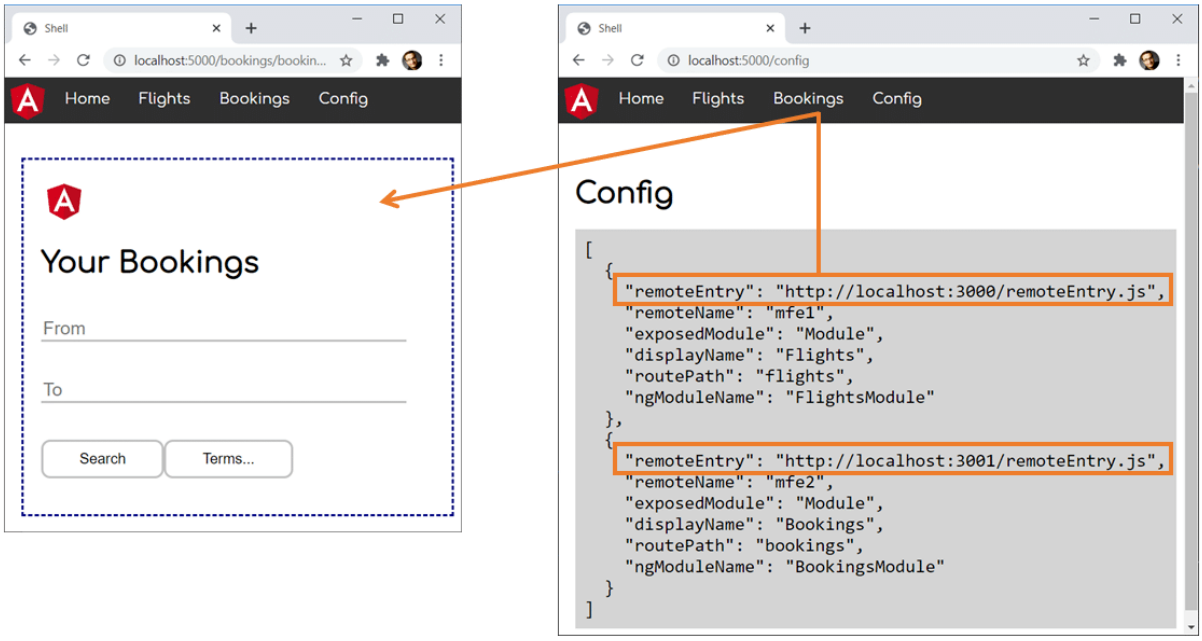
One also has to deal with possible version conflicts. For example, it is likely that components that were compiled with completely different Angular versions will not work together at runtime. Such cases must be avoided with conventions or at least recognized as early as possible with integration tests.

Dynamic Module Federation with Angular

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

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

The following image displays this idea:



The shell loads a microfrontend it is informed about on runtime

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

As usual, the source code[50] used here can be found in my GitHub account[51].

[50] https://github.com/manfredsteyer/module-federation-with-angular-dynamic.git
[51] https://github.com/manfredsteyer/module-federation-with-angular-dynamic.git

Module Federation Config

Let's start with the shell's Module Federation configuration. In this scenario, it's as simple as this:

```
new ModuleFederationPlugin({
  remotes: {},
  shared: {
    "@angular/core": { singleton: true, strictVersion: true },
    "@angular/common": { singleton: true, strictVersion: true },
    "@angular/router": { singleton: true, strictVersion: true }
   }
}),
```

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

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

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

```
new ModuleFederationPlugin({
  name: "mfe1",
  filename: "remoteEntry.js",
  exposes: {
    './Module': './projects/mfe1/src/app/flights/flights.module.ts'
  },
  shared: {
    "@angular/core": { singleton: true, strictVersion: true },
    "@angular/common": { singleton: true, strictVersion: true },
    "@angular/router": { singleton: true, strictVersion: true },
        [...]
  }
}),
```

Routing to Dynamic Microfrontends

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

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

[...]

const routes: Routes = [
    [...]
    {
        path: 'flights',
        loadChildren: () =>
            loadRemoteModule({
                remoteEntry: 'http://localhost:3000/remoteEntry.js',
                remoteName: 'mfe1',
                exposedModule: './Module'
            })
            .then(m => m.FlightsModule)
    },
    [...]
];
```

As you might have noticed, we're just switching out the dynamic `import` normally used here by a call to `loadRemoteModule` which also works with key data not known at compile time. The latter one uses the webpack runtime api to get hold of the remote on demand.

Improvement for Dynamic Module Federation

This was quite easy, wasn't it? However, we can improve this solution a bit. Ideally, we load the remote entry upfront before Angular bootstraps. In this early phase, Module Federation tries to determine the highest compatible versions of all dependencies.

Let's assume, the shell provides version 1.0.0 of a dependency (specifying `1.0.0` in its package.json) and the micro frontend uses version `1.1.0` (specifying 1.1.0 in its package.json). In this case, they would go with version 1.1.0. However, this is only possible if the remote's entry is loaded upfront. More details about how Module Federation deals with different versions can be found in one of the following chapters.

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

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

Promise.all([
    loadRemoteEntry('http://localhost:3000/remoteEntry.js', 'mfe1')
])
.catch(err => console.error('Error loading remote entries', err))
.then(() => import('./bootstrap'))
.catch(err => console.error(err));
```

Here, we need to remember, that the `@angular-architects/module-federation` plugin moves the contents of the original `main.ts` into the `bootstrap.ts` file. Also, it loads the `bootstrap.ts` with a **dynamic import** in the `main.ts`. This is necessary because the dynamic import gives Module Federation the needed time to negotiate the verions of the shared libraries to use with all the remotes.

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

After this, we don't need to pass the remote entry's url to loadRemoteModule when we lazy load the micro frontend with the router:

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

[...]
const routes: Routes = [
    [...]
    {
        path: 'flights',
        loadChildren: () =>
            loadRemoteModule({
                                // We don't need this anymore b/c its loaded upfront now
                // remoteEntry: 'http://localhost:3000/remoteEntry.js',
                remoteName: 'mfe1',
                exposedModule: './Module'
            })
            .then(m => m.FlightsModule)
    },
    [...]
]
```

However, we could also stick with it, because `loadRemoteModule` remembers what was loaded and never loads a thing twice.

Bonus: Dynamic Routes for Dynamic Microfrontends

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

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

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

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

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

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

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

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

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

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

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

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

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

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

  microfrontends: Microfrontend[] = [];

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

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

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

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

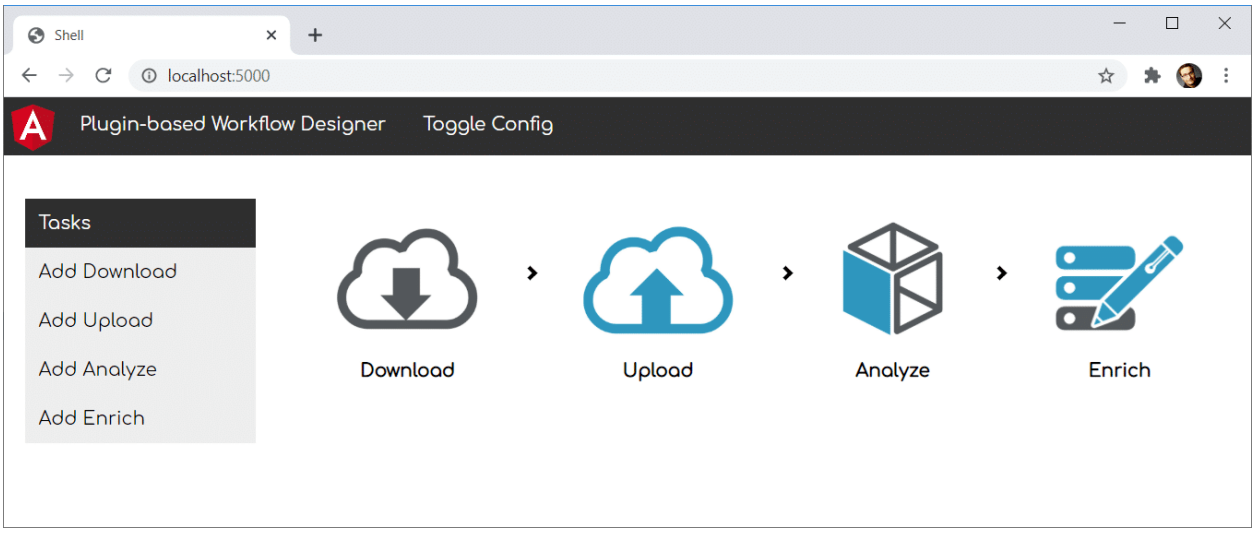
Conclusion

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

Plugin Systems with Module Federation: Building An Extensible Workflow Designer

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

While the previous chapter leveraged the router for integrating remotes available, this chapter shows how to load individual components. The example used for this is a simple plugin-based workflow designer. This shows that Module Federation is not limited to micro frontends but one can also use it for plugin systems in general.



The Workflow Designer can load separately compiled and deployed tasks

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

The configuration informs about where to find the tasks

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

As always, the source code[52] can be found in my GitHub account[53].

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 series:

[52] https://github.com/manfredsteyer/module-federation-with-angular-dynamic-workflow
[53] https://github.com/manfredsteyer/module-federation-with-angular-dynamic-workflow

```
new ModuleFederationPlugin({
  name: "mfe1",
  filename: "remoteEntry.js",
  exposes: {
    './Download': './projects/mfe1/src/app/download.component.ts',
    './Upload': './projects/mfe1/src/app/upload.component.ts'
  },
  shared: {
    "@angular/core": { singleton: true, strictVersion: true },
    "@angular/common": { singleton: true, strictVersion: true },
    "@angular/router": { singleton: true, strictVersion: true }
  }
}),
```

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

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

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

Loading the Plugins into the Workflow Designer

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

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

[...]

const component = await loadRemoteModule({
    remoteEntry: 'http://localhost:3000/remoteEntry.js',
    remoteName: 'mfe1',
    exposedModule: './Download'
})
```

Providing Metadata on the Plugins

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

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

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

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

```
@Injectable({ providedIn: 'root' })
export class LookupService {
    lookup(): Promise<PluginOptions[]> {
        return Promise.resolve([
            {
                remoteEntry: 'http://localhost:3000/remoteEntry.js',
                remoteName: 'mfe1',
                exposedModule: './Download',

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

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

Dynamically Creating the Plugin Component

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

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

    constructor(
      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) {
  }

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

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

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

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

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

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

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

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

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

Conclusion

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

Using Module Federation with Nx

The combination of Micro Frontends and monorepos can be quite tempting: Monorepos make it easy to share libraries. Thanks to access restrictions discussed in a previous chapter, individual business domains can be isolated. Also, having all micro frontends in the same monorepo doesn't prevent us from deploying them separately.

This chapter gives some hints about using Module Federation for such a scenario. While the examples use a Nx workspace, the principal ideas can also be implemented with a classic Angular workspace. However, as you will see, Nx provides some really powerful features that make your life easier, e. g. the possibility of generating a visual dependency graph or finding out which applications have been changed and hence need to be redeployed.

If you want to have a look at the source code[54] used here, you can check out this repository[55].

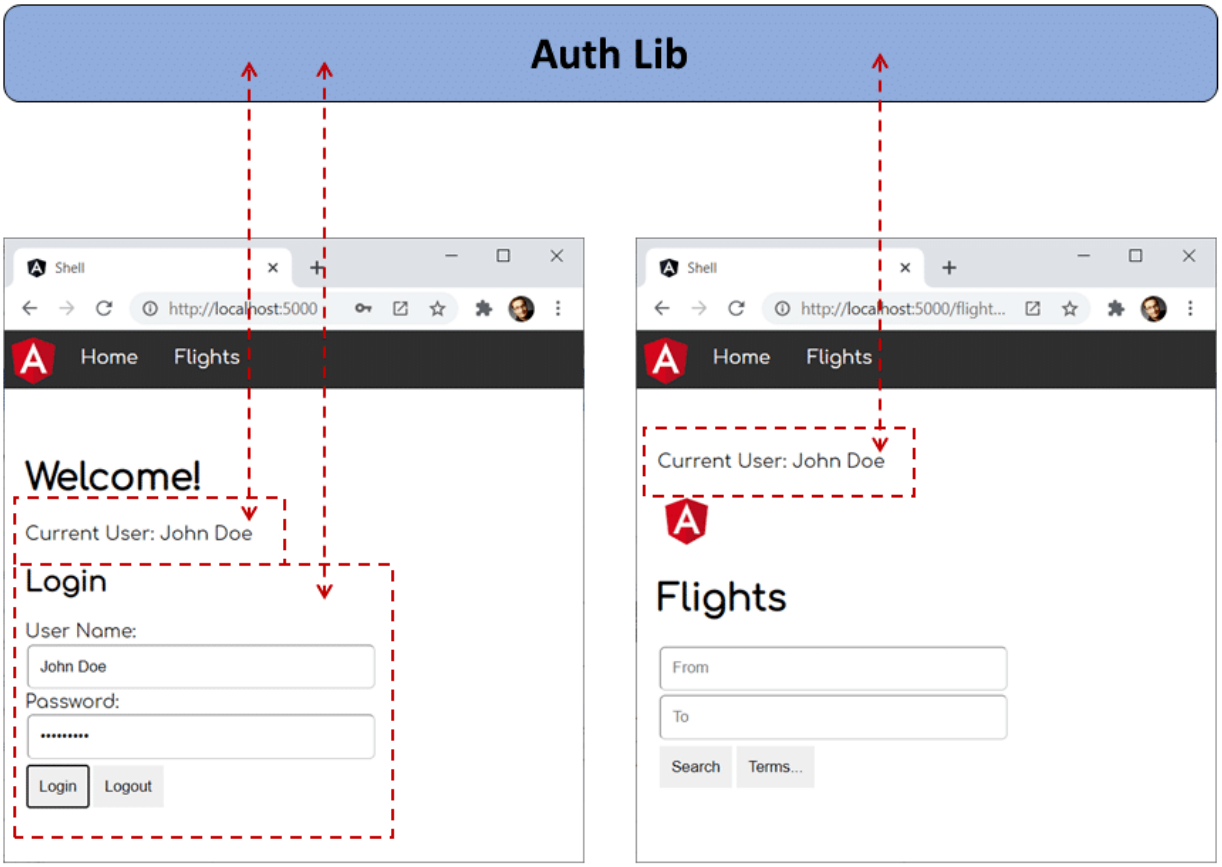
Example

The example used here is a Nx monorepo with a micro frontend shell (`shell`) and a micro frontend (`mfe1`, "micro frontend 1"). Both share a common library for authentication (`auth-lib`) that is also located in the monorepo:

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

The `auth-lib` provides two components. One is logging-in users and the other one displays the current user. They are used by both, the `shell` and `mfe1`:

[54] https://github.com/manfredsteyer/module_federation_nx_mono_repo
[55] https://github.com/manfredsteyer/module_federation_nx_mono_repo



Schema

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

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

```
"paths": {
    "@demo/auth-lib": [
        "libs/auth-lib/src/index.ts"
    ]
},
```

The `shell` and `mfe1` (as well as further micro frontends we might add in the future) need to be deployable in separation and loaded at runtime. However, we don't want to load the `auth-lib` twice or several times! Archiving this with an npm package is not that difficult. This is one of the most obvious and easy to use features of Module Federation. The next sections discuss how to do the same with libraries of a monorepo.

The Shared Lib

Before we delve into the solution, let's have a look at the `auth-lib`. It contains an `AuthService` that logs-in the user and remebers them using the property `_userName`:

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

  // tslint:disable-next-line: variable-name
  private _userName: string = null;

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

  constructor() { }

  login(userName: string, password: string): void {
    this._userName = userName;
  }

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

The authentication method I'm using here, is what I'm calling "Authentication for honest users TM". Besides this service, there is also a `AuthComponent` with the UI for logging-in the user and a `UserComponent` displaying the current user's name. Both components are registered with the library's NgModule:

```
@NgModule({
  imports: [
    CommonModule,
    FormsModule
  ],
  declarations: [
    AuthComponent,
    UserComponent
  ],
  exports: [
    AuthComponent,
    UserComponent
  ],
})
export class AuthLibModule {}
```

As every library, it also has a barrel `index.ts` (sometimes also called `public-api.ts`) serving as the entry point. It exports everything consumers can use:

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

// Don't forget about your components!
export * from './lib/auth/auth.component';
export * from './lib/user/user.component';
```

Please note that `index.ts` is also exporting the two components although they are already registered with the also exported `AuthLibModule`. In the scenario discussed here, this is vital in order to make sure it's detected and compiled by Ivy.

The Module Federation Configuration

As in the previous chapter, we are using the `@angular-architects/module-federation` plugin to enable Module Federation for the `shell` and `mfe1`. For this, just run this command twice and answer the plugin's questions:

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

This generates a webpack config for Module Federation. The latest version of the plugin uses a helper class called `SharedMapping`:

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

const sharedMappings = new mf.SharedMappings();
sharedMappings.register(
    path.join(__dirname, '../../tsconfig.base.json'),
    ['@demo/auth-lib']  // <-- register your libs here!
);

module.exports = {
  output: {
    uniqueName: "mfe1"
  },
  optimization: {
    // Only needed to bypass a temporary bug
    runtimeChunk: false
  },
  plugins: [
    new ModuleFederationPlugin({
      name: "mfe1",
      filename: "remoteEntry.js",
      exposes: {
          './Module': './apps/mfe1/src/app/flights/flights.module.ts',
      },
      shared: {
        "@angular/core": { singleton: true, strictVersion: true },
        "@angular/common": { singleton: true, strictVersion: true },
        "@angular/router": { singleton: true, strictVersion: true },
        ...sharedMappings.getDescriptors()
      }
    }),
    sharedMappings.getPlugin(),
  ],
};
```

Everything you need to do is registering the library's mapped name with the `sharedMappings` instance at the top of this file.

For the time being, this is all we need to know. In a section below, I'm going to explain what `SharedMappings` is doing and why it's a good idea to hide these details in such a convenience class.

Trying it out

To try this out, just start the two applications:

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

Then, log-in in the shell and make it to load `mfe1`. If you see the logged-in user name in `mfe1`, you have the proof that `auth-lib` is only loaded once and shared across the applications.

Deploying

As normally, libraries don't have versions in a monorepo, we should always redeploy all the changed micro frontends together. Fortunately, Nx helps with finding out which applications/ micro frontends have been changed or affected by a change. For this, just run the `affected:apps` script:

```
npm run affected:apps
```

You might also want to detect the changed applications as part of your CI pipeline. To make implementing such an automation script easier, leverage the Nx CLI (`npm i -g @nrwl/cli`) and call the `affected:apps` script with the `--plain` switch:

```
nx affected:apps --plain
```

This switch makes the CLI to print out all affected apps in one line of the console separated by a space. Hence, this result can be easily parsed by your scripts.

Also, as the micro frontends loaded into the shell don't know each other upfront but only meet at runtime, it's a good idea to rely on some e2e tests.

What Happens Behind the Covers?

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

Its method `getDescriptors` returns the needed entries for the `shared` section in the configuration:

```
"@demo/auth-lib": {
    import: path.resolve(__dirname, "../../libs/auth-lib/src/index.ts"),
    requiredVersion: false
},
```

These entries look a bit different than the ones we are used to. Instead of pointing to an npm package to share, it directly points to the library's entry point using its `import` property. The right path is taken from the mappings in the `tsconfig.json`.

Normally, Module Federation knows which version of a shared library is needed because it looks into the project's `package.json`. However, in the case of a monorepo, shared libraries (very often) don't have a version. Hence, `requiredVersion` is set to `false`. This makes Module Federation accepting an existing shared instance of this library without drawing its very version into consideration.

However, to make this work, we should always redeploy all changed parts of our overall system together, as proposed above.

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

```
import { UserComponent } from '../../libs/auth-lib/src/lib/user/user.component.ts';
```

However, to make Module Federation only sharing one version of our lib(s), we need to import them using a consistent name like `@demo/auth-lib`:

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

For this, `SharedMappings` provides a method called `getPlugin` returning a configured `NormalModuleReplacementPlug` instance that takes care of rewriting such imports. The key data needed here is take from `tsconfig.json`.

Bonus: Versions in the Monorepo

As stated before, normally libraries don't have a version in a monorepo. The consequence is that we need to redeploy all the changed application together. This makes sure, all applications can work with the shared libs.

However, Module Federation is really flexible and hence it even allows to define versions for libraries in a monorepo. In the case of npm packages, both, the version provided by the library but also the versions needed by its consumers are defined in the respective `package.json` files.

In our monorepo, we don't have either, hence we need to pass this information directly to Module Federation. If we didn't use the `SharedMappings` convenience class, we needed to place such an object into the config's shared section:

```
"auth-lib": {
    import: path.resolve(__dirname, "../../libs/auth-lib/src/index.ts"),
    version: '1.0.0',
    requiredVersion: '^1.0.0'
},
```

Here, `version` is the actual library version while `requiredVersion` is the version (range) the consumer (micro frontend or shell) accepts. To prevent repeating the library version in all the configurations for all the micro frontends, we could centralize it. Perhaps, you want to create a `package.json` in the library's folder:

```
{
  "name": "@demo/auth-lib",
  "version": "1.0.0",
}
```

Now, the webpack config of a specific micro frontend/ of the shell only needs to contain the accepted version (range) using `requiredVersion`:

```
"@demo/auth-lib": {
    import: path.resolve(__dirname, "../../libs/auth-lib/src/index.ts"),
    version: require('relative_path_to_lib/package.json').version,
    requiredVersion: '^1.0.0'
},
```

Because of this, we don't need to redeploy all the changed applications together anymore. Using the provided versions, Module Federation decides at runtime which micro frontends can safely share the same instance of a library and which micro frontends need to fall back to another version (e. g. the own one).

The drawback of this approach is that we need to rely on the authors of the libraries and the micro frontends to manage this meta data correctly. Also, if two micro frontends need two non-compatible versions of a shared library, the library is loaded twice. This is not only bad for performance but also leads to an issue in the case of stateful libraries as the state is duplicated. In our case, both `auth-lib` instances could store their own user name.

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

```
new ModuleFederationPlugin({
  [...]
  shared: {
    [...]
    ...sharedMappings.getDescriptor('@demo/auth-lib', '^1.0.0')
  }
}),
```

The rest of the configuration would be as discussed above.

Pitfalls (Important!)

When using this approach to shared libs within a monorepo, you might encounter some pitfalls. They are due to the fact that the CLI/webpack5 integration is still experimental in Angular 11 but also because we treat a folder with uncompiled typescript files like an npm package.

Bug with styleUrls

Currently, there is, unfortunately, a bug in the experimental CLI/webpack5 integration causing issues when using shared libraries together with components pointing to `styleUrls`. For the time being, you can work around this issue by removing all `styleUrls` in your applications and libraries.

Sharing a library that is not even used

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

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

Not exported Components

If you use a shared component without exporting it via your library's barrel (`index.ts` or `public-api.ts`), you get the following error at runtime:

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

Conflict: Multiple assets emit different content to the same filename

Add this to the output section of your webpack config:

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

Dealing with Version Mismatches in Module Federation

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

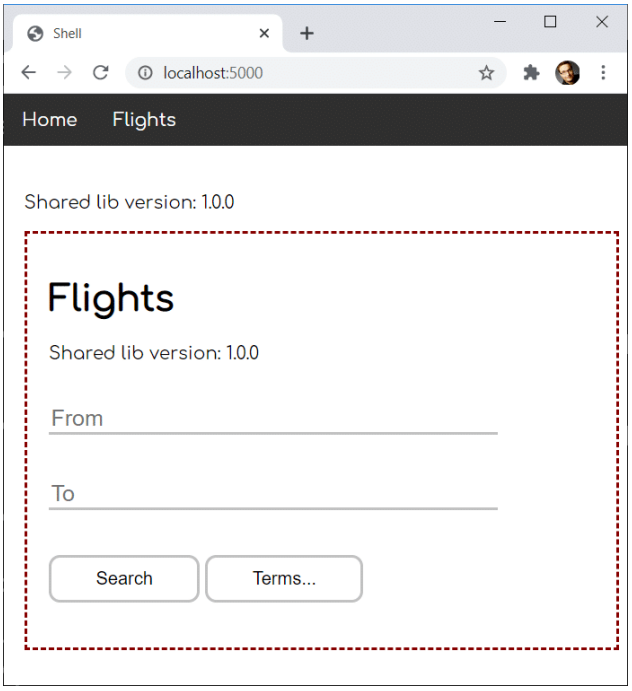
However, there might be situations where several micro frontends and the shell need different versions of a shared library. Also, these versions might not be compatible with each other.

For dealing with such cases, Module Federation provides several options. In this chapter, I present these options by looking at different scenarios. The source code[56] for these scenarios can be found in my GitHub account[57].

Example Used Here

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

[56] https://github.com/manfredsteyer/module_federation_shared_versions
[57] https://github.com/manfredsteyer/module_federation_shared_versions



Shell loading microfrontends

The micro frontend is framed with the red dashed line.

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

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

The package useless-lib[58] is a dummy package, I've published for this example. It's available in the versions `1.0.0`, `1.0.1`, `1.1.0`, `2.0.0`, `2.0.1`, and `2.1.0`. In the future, I might add further ones. These versions allow us to simulate different kinds of version mismatches.

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

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

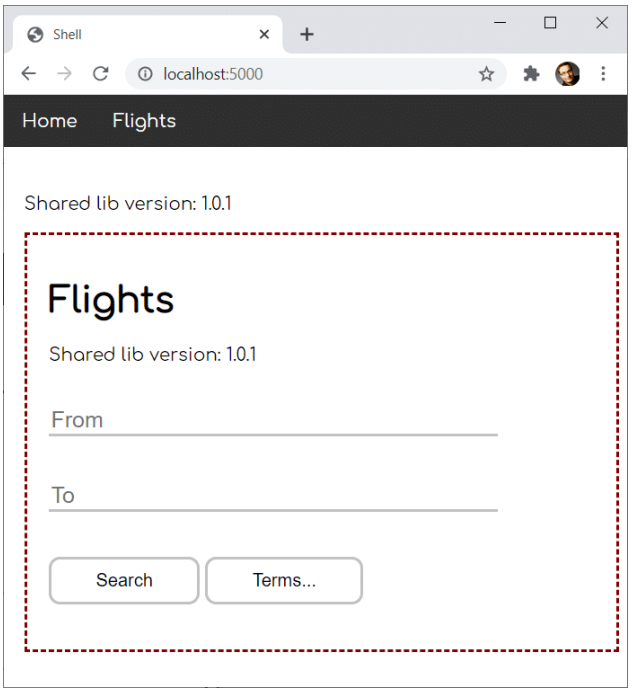
[58] https://www.npmjs.com/package/useless-lib

Semantic Versioning by Default

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

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

This leads to the following result:



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

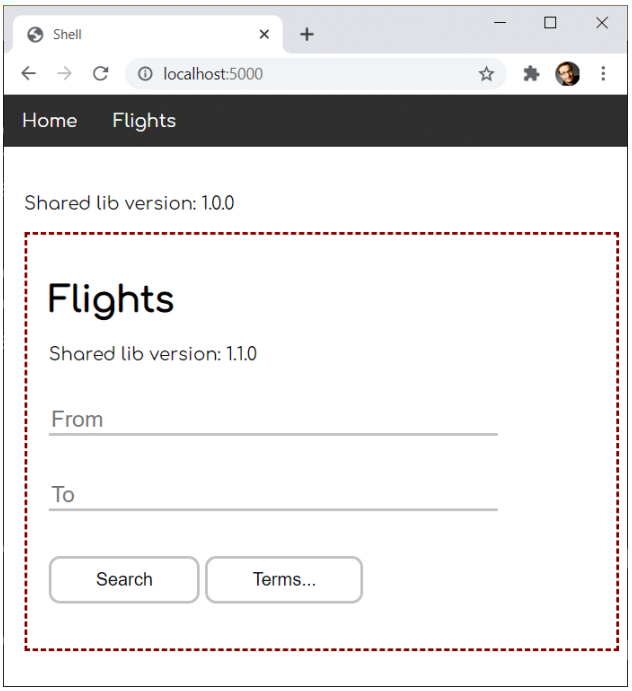
Fallback Modules for Incompatible Versions

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

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

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

This leads to the following result:



Using Fallback Module

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

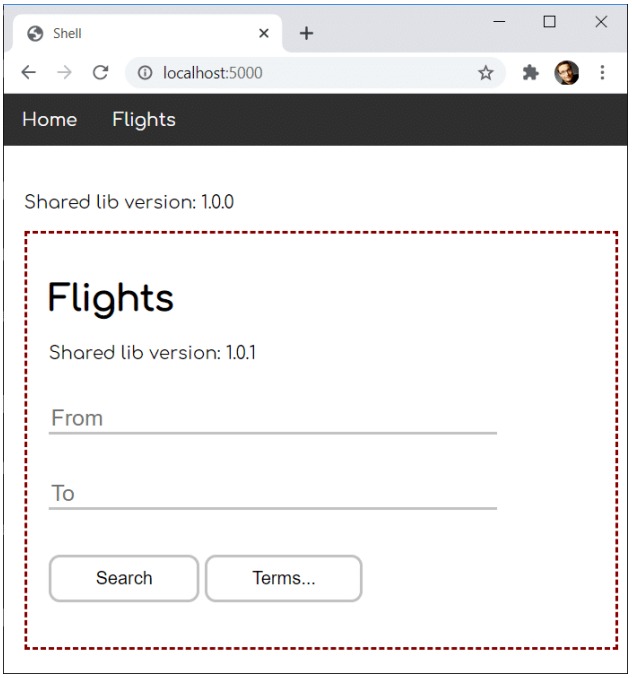
Differences With Dynamic Module Federation

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

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

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

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



Dynamic Microfrontend falls back to own version

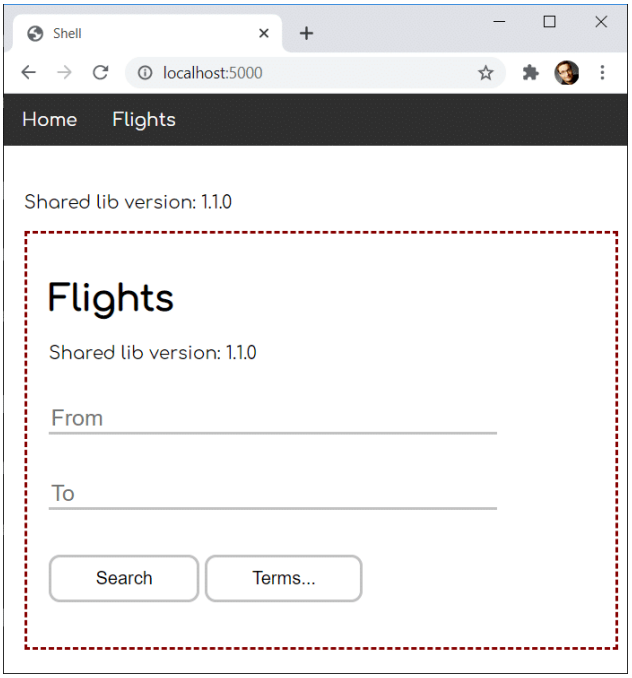
If there were other static remotes (e. g. micro frontends), the shell could also choose for one of their versions according to semantic versioning, as shown above.

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

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

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

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



Dynamic Microfrontend uses already loaded version

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

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

Singletons

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

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

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

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

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

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

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

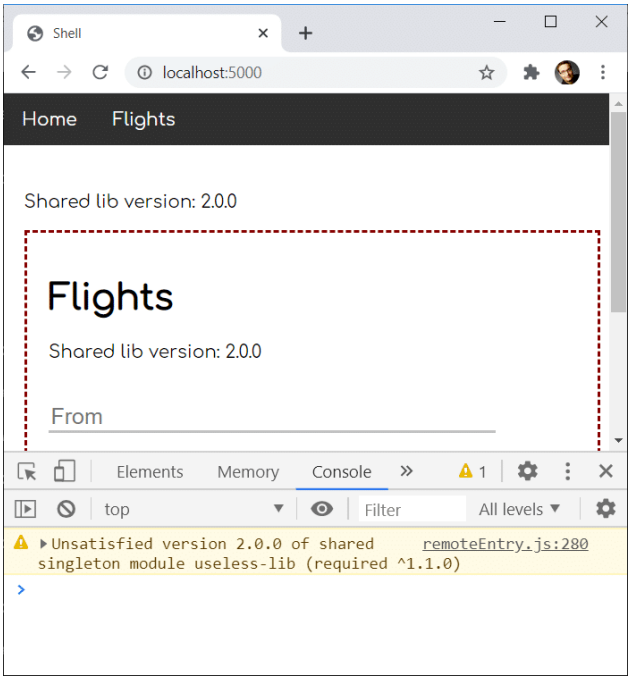
```
// Shell
shared: {
  "rxjs": {},
  "useless-lib": {
    singleton: true,
  }
},
```

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

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

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

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



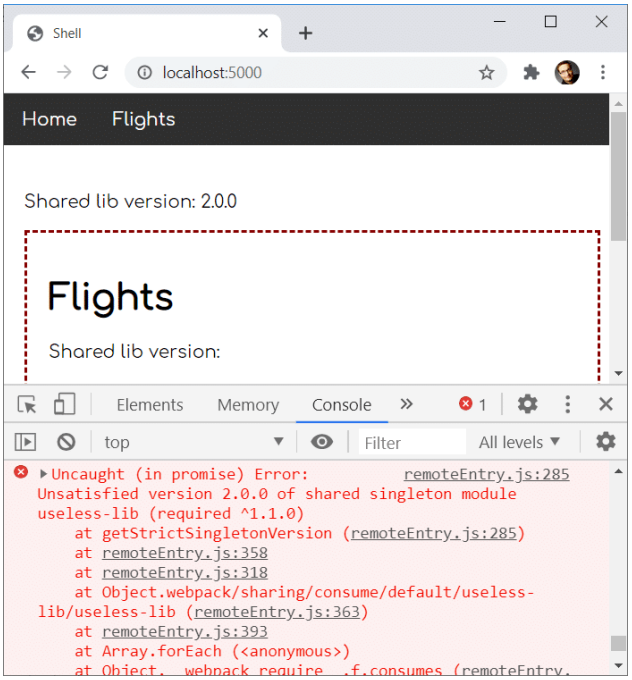
Module Federation only loads the highest version for singletons

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

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

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

The result of this looks as follows at runtime:



Version mismatches regarding singletons using strictVersion make the application fail

Accepting a Version Range

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

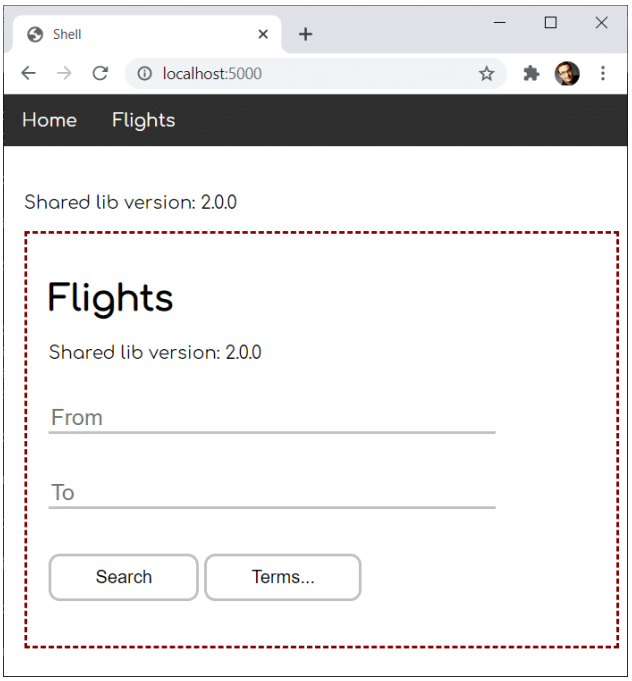
To explore this option, let's one more time assume the following version mismatch:

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

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

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

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



Accepting incompatible versions by defining a version range

Conclusion

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

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

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

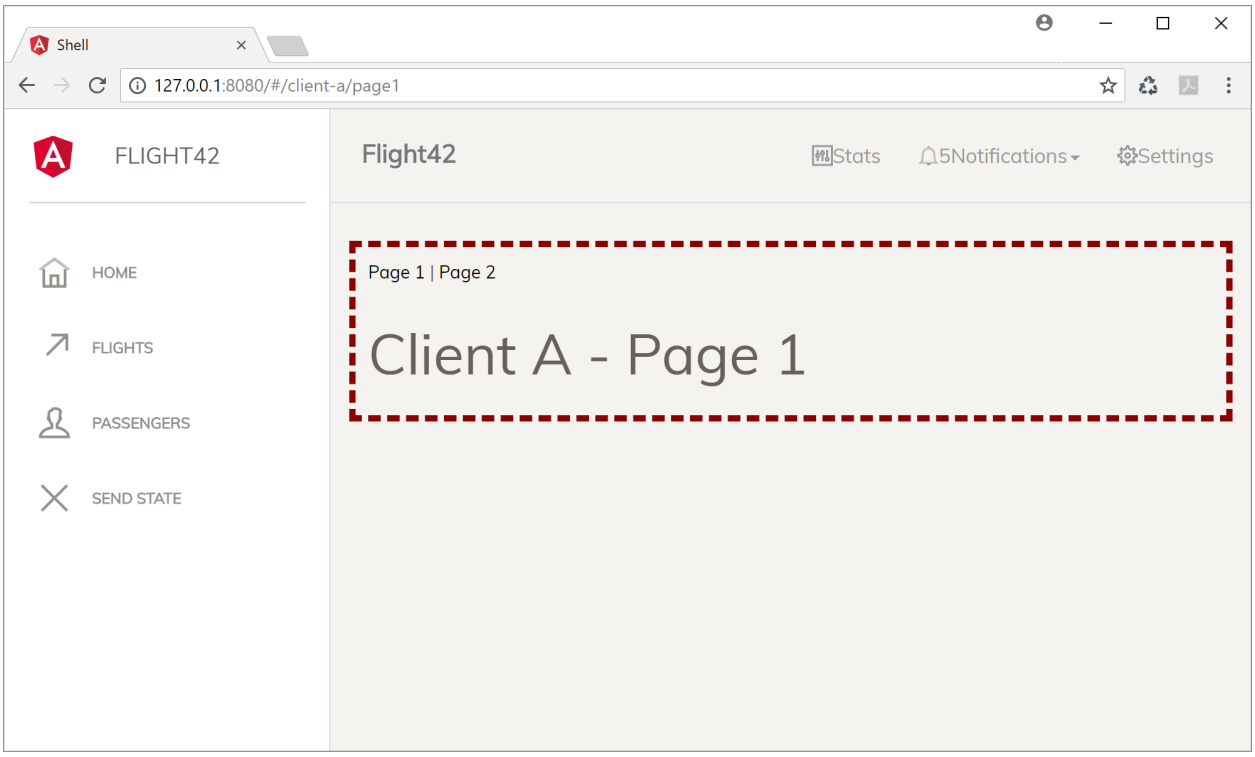
You can also ease the requirements for a specific version by defining a version range using `requestedVersion`. You can even define several scopes for advanced scenarios where each of them can get its own version.

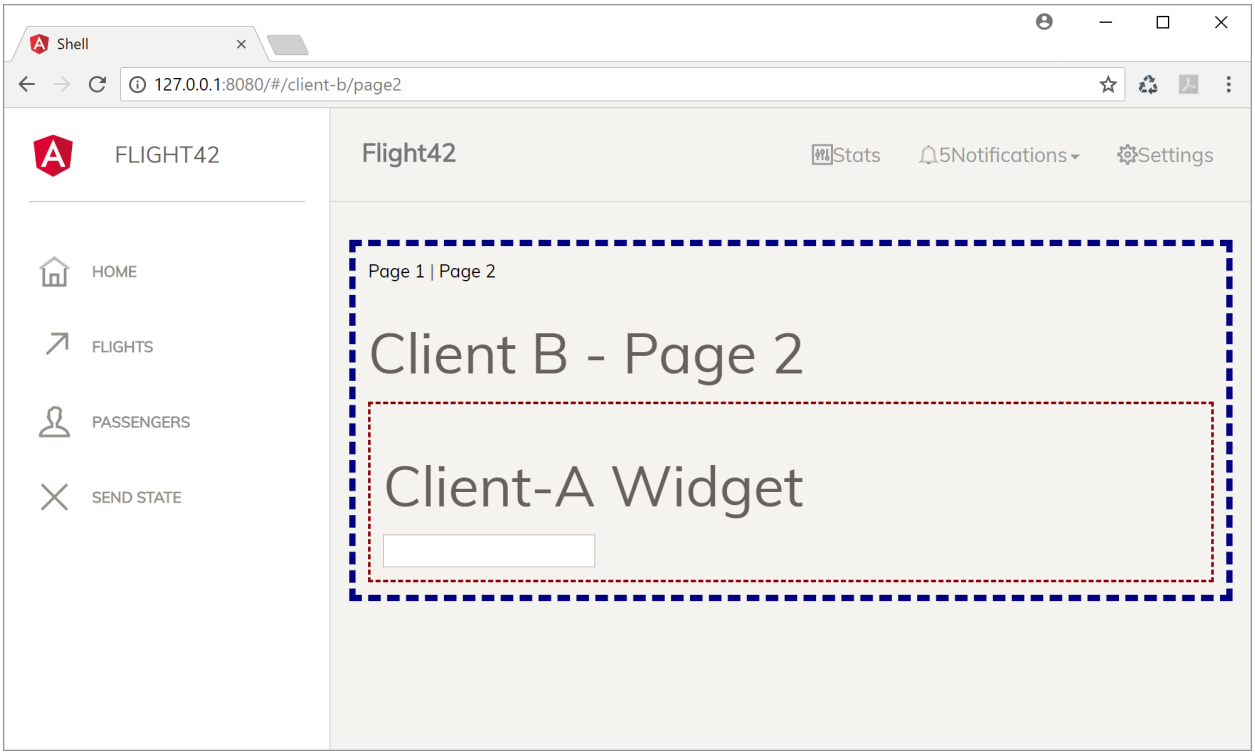
Micro Frontends with Web Components/ Angular Elements – An Alternative Approach

Using Module Federation is quite easy: You can share libraries and you can use Angular as your meta-framework loading microfrontends on demand.

This chapter shows a more difficult alternative approach using Web Components/ Angular Elements. While it demands you to deal with more aspects by yourself and to make use of some workarounds, it also allows you to mix and match different SPA frameworks. Also, you can use this approach to use different versions of the same framework.

The case study used here loads a simple `client-a` and `client-b` into the shell. The former shares a widget with the latter:





You can find the source code[59] for this in my GitHub account here[60].

Step 0: Make sure you need it

Make sure this approach fits your architectural goals. As discussed in the previous chapter, microfrontends have many consequences. Make sure you are aware of them.

Step 1: Implement Your SPAs

Implement your microfrontends as ordinary Angular applications. In a microservice architecture, it's quite common that every part gets an individual repository to decouple them as much as possible (see Componentization via Services[61]) I've seen a lot of microfrontends based upon monorepos for practical reasons.

Of course, we could discuss when the term microfrontend is appropriate. More important is to find an architecture that fits your goals and to be aware of its consequences.

If we use a monorepo, we have to ensure, e.g. with linting rules, not to couple the microfrontends with each other. As discussed in a previous chapter, Nrwl's Nx[62] provides an excellent solution for

[59] https://github.com/manfredsteyer/angular-microfrontend
[60] https://github.com/manfredsteyer/angular-microfrontend
[61] https://martinfowler.com/chapters/microservices.html#ComponentizationViaServices
[62] https://nx.dev/angular

that: It enables restrictions defining which libraries can access each other. Nx can detect which parts of your monorepo are affected by a change and only recompile and retest those.

To make routing across microfrontends easier, it's a good idea to prefix all the routes with the application's name. In the following case, the application name is `client-a`

```
@NgModule({
  imports: [
    ReactiveFormsModule,
    BrowserModule,
    RouterModule.forRoot([
      { path: 'client-a/page1', component: Page1Component },
      { path: 'client-a/page2', component: Page2Component },
      { path: '**', component: EmptyComponent}
    ], { useHash: true })
  ],
  [...]
})
export class AppModule {
  [...]
}
```

Step 2: Expose Shared Widgets

Expose widgets you want to share as web components/custom elements. Please note that from the perspective of microservices, you should minimise code sharing between microfrontends as it causes coupling in this architecture.

To expose an angular component as a custom element, you can use Angular elements. Take a look at my blogposts about Angular Elements[63] and lazy and external Angular Elements[64].

Step 3: Compile your SPAs

Webpack, and hence the Angular CLI, use a global array for registering bundles. It enables different (lazy) chunks of your application to find each other. However, if we load several SPAs into one page, they compete over this array, mess it up, and stop working.

We have two solutions:

1. Put everything into one bundle, so that this global array is not needed

[63] https://www.softwarearchitekt.at/aktuelles/angular-elements-part-i/
[64] https://www.softwarearchitekt.at/aktuelles/angular-elements-part-ii/

2. Rename the global array

I use solution 1) because a microfrontend is, by definition, small. Just having one bundle makes loading it on demand easier. As we will see later, we can share libraries like RxJS or Angular itself between them.

To tweak the CLI to produce one bundle, I use my free tool ngx-build-plus[65] which provides a `--single-bundle` switch:

```
ng add ngx-build-plus
ng build --prod --single-bundle
```

Within a monorepo, you have to provide the project in question:

```
ng add ngx-build-plus --project myProject
ng build --prod --single-bundle --project myProject
```

Step 4: Create a shell and load the bundles on demand

Loading the bundles on demand is straightforward. All you need is some vanilla JavaScript code to dynamically create a `script` tag and the tag for application's root element:

```
// add script tag
const script = document.createElement('script');
script.src = '[...]/client-a/main.js';
document.body.appendChild(script);

// add app
const frontend = document.createElement('client-a');
const content = document.getElementById('content');
content.appendChild(frontend);
```

Of course, you can also wrap this into a directive.

You need some code to show and hide the loaded microfrontend on-demand:

```
frontend['visible'] = false;
```

[65] https://www.npmjs.com/package/ngx-build-plus

Step 5: Communication Between Microfrontends

In general, we should minimise communication between microfrontends, as it couples them .

We have several options to implement communication. I used the least obtrusive one here: the query string. This approach has several advantages:

1. It is irrelevant which order the microfrontends are loaded. When loaded they can grab the current parameters from the URL
2. It allows deep linking
3. It's how the web is supposed to work
4. It's easy to implement

Setting a URL parameter with the Angular router is a simple matter of calling one method:

```
this.router.navigate(['.'], {
  queryParamsHandling: 'merge', queryParams: { id: 17 }});
```

The `merge` option saves the existing URL parameters. If there is already an `id` parameter, the router overwrites it.

The Angular router can help listen for changes within URL parameters:

```
route.queryParams.subscribe(params => {
    console.debug('params', params);
});
```

There are some alternatives:

1. If you wrap your microfrontends into web components, you can use their properties and events to communicate with the shell.
2. The shell can put a "message bus" into the global namespace:

   ```
   (window as any).messageBus = new BehaviorSubject<MicroFrontendEvent>(null);
   ```

   The shell and the microfrontends can subscribe to this message bus and listen for specific events. Both can emit events.
3. Using custom events provided by the browser:

```
// Sender
const customer = { id: 17, ... };
window.raiseEvent(new CustomEvent('CustomerSelected', {details: customer}))

// Receiver
window.addEventListener('CustomerSelected', (e) => { ... })
```

Conclusion

With the right wrinkles, implementing a shell for microelements is not difficult. However, this is only one approach to implementing microfrontends and has advantages and disadvantages. Before implementing it, make sure it fits your architectural goals.

Literature

- Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software[66]
- Wlaschin, Domain Modeling Made Functional[67]
- Ghosh, Functional and Reactive Domain Modeling[68]
- Nrwl, Monorepo-style Angular development[69]
- Jackson, Micro Frontends[70]
- Burleson, Push-based Architectures using RxJS + Facades[71]
- Burleson, NgRx + Facades: Better State Management[72]
- Steyer, Web Components with Angular Elements (article series, 5 parts)[73]

[66] https://www.amazon.com/dp/0321125215
[67] https://pragprog.com/book/swdddf/domain-modeling-made-functional
[68] https://www.amazon.com/dp/1617292249
[69] https://go.nrwl.io/angular-enterprise-monorepo-patterns-new-book
[70] https://martinfowler.com/articles/micro-frontends.html
[71] https://medium.com/@thomasburlesonIA/push-based-architectures-with-rxjs-81b327d7c32d
[72] https://medium.com/@thomasburlesonIA/ngrx-facades-better-state-management-82a04b9a1e39
[73] 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 (https://twitter.com/ManfredSteyer) and Facebook (https://www.facebook.com/manf and find his blog here (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[74]:

Advanced Angular Workshop

Save your ticket[75] for one of our **online or on-site** workshops now or request a company workshop[76] (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

[74] https://www.angulararchitects.io/en/angular-workshops/advanced-angular-enterprise-architecture-incl-ivy/
[75] https://www.angulararchitects.io/en/angular-workshops/advanced-angular-enterprise-architecture-incl-ivy/
[76] https://www.angulararchitects.io/en/angular-workshops/

Please find the full list with our offers here[77].

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[78] and/ or follow the book's author on Twitter[79].

[77] https://www.angulararchitects.io/en/angular-workshops/
[78] https://www.angulararchitects.io/subscribe/
[79] https://twitter.com/ManfredSteyer