ENTERPRISE ANGULAR

DDD, Nx Monorepos, and Micro Frontends

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 2020-06-26

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

I've helped numerous companies over the years to implement Angular-based large-scale enterprise applications.

When implementing an Angular solution, it's vital to decompose 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 `book@softwarearchitekt.at`. 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.

Case Studies

This book explores the concepts by using case studies which you can find in my GitHub account:

- Case Study for Strategic Design[1]
- Case Study for Tactical Design[2]
- Case Study for Microfrontend Shell[3]

Help Improve this Book.

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

[1] https://github.com/manfredsteyer/strategic-design
[2] https://github.com/manfredsteyer/angular-ddd
[3] https://github.com/manfredsteyer/angular-microfrontend
[4] https://github.com/manfredsteyer/ddd-bk

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 Review Workshop
- Angular Upgrade Workshop

Please find the full list of our offers here[5].

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[6] who provide the open-source tool Nx[7] used in the case studies here and described in the following chapters.
- Thomas Burleson[8] who did an excellent job describing the concept of facades. Thomas contributed to the chapter about tactical design which explores facades.

[5] https://www.softwarearchitekt.at/angular-schulung/
[6] https://nrwl.io/
[7] https://nx.dev/angular
[8] https://twitter.com/thomasburleson?lang=de

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[9] (DDD) approach. I also explain how to implement its ideas with an Nx[10]-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[11] 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[12].

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

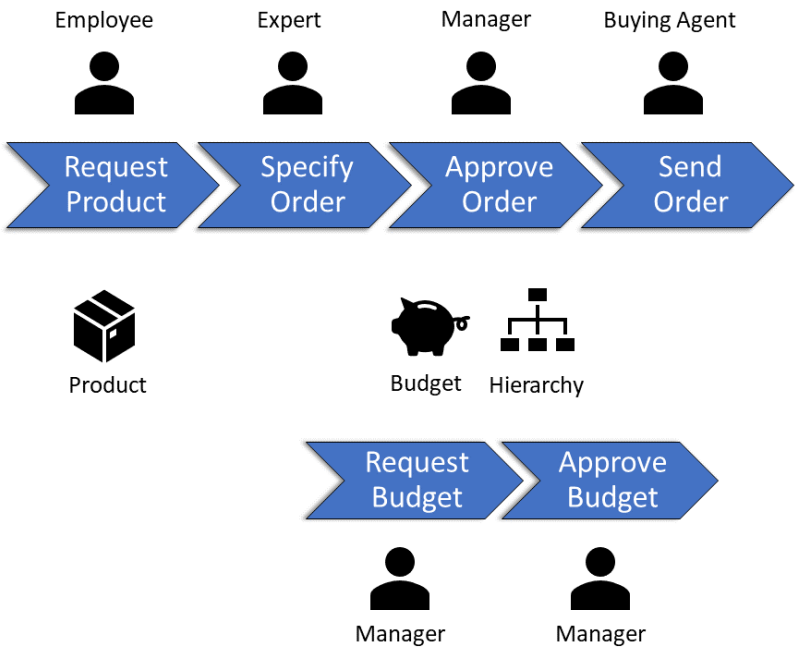
[9] https://www.amazon.de/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215/ref=sr_1_3?ie=UTF8&qid=1551688461&sr=8-3&keywords=ddd

[10] https://nx.dev/

[11] https://pragprog.com/book/swdddf/domain-modeling-made-functional

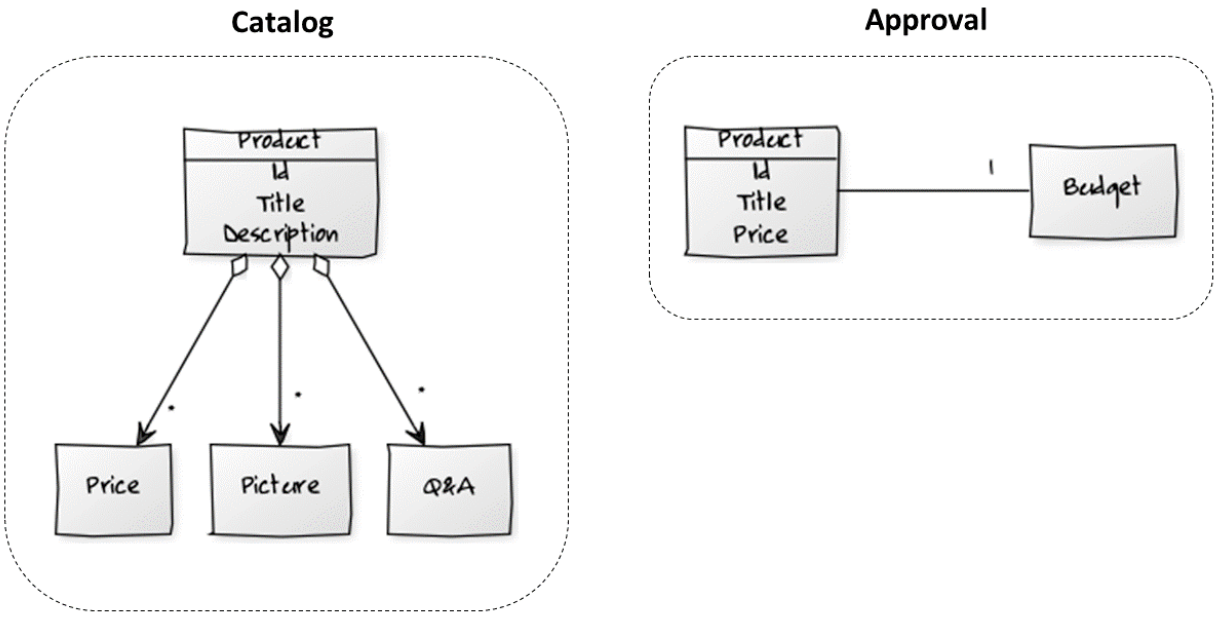
[12] https://martinfowler.com/bliki/UbiquitousLanguage.html

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[13]. 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.

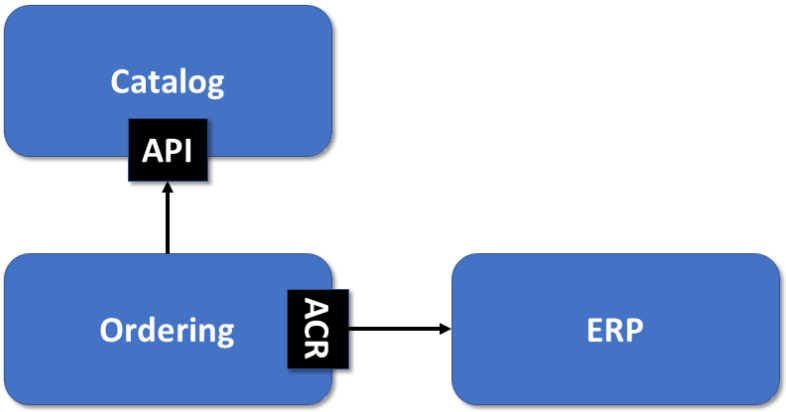
If we proceed with this analysis, we may find the following domains:

[13] 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[14]. 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.

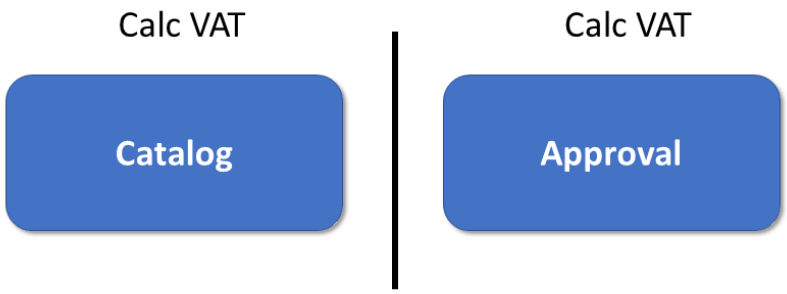
One domain can easily use the other. In this case, however, it is unclear how much power each is entitled to. Can the consumer impose specific changes on the provider and insist on backward compatibility? Or must the consumer be satisfied with what it gets from the provider?

[14] 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[15]-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[16]:

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

Advanced Angular Workshop

Save your ticket[17] now or request a company workshop[18] 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[19] and/ or follow the book's author on Twitter[20].

[17] https://www.angulararchitects.io/schulungen/advanced-angular-enterprise-anwendungen-und-architektur
[18] https://www.angulararchitects.io/schulungen/advanced-angular-enterprise-anwendungen-und-architektur
[19] https://www.angulararchitects.io/subscribe/
[20] 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[21]-based monorepo.

If you want to look at the underlying case study[22], you can find the source code here[23]

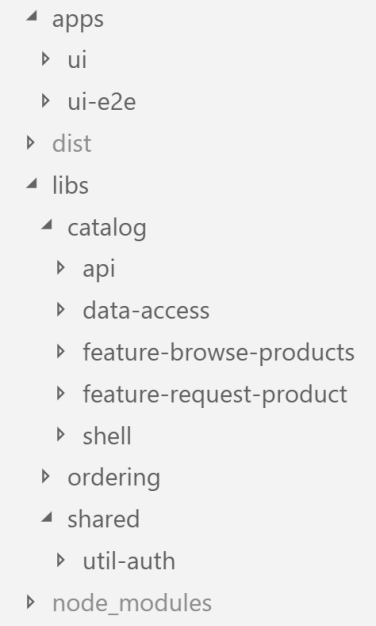
I'm following recommendations the Nx team recently described in their free e-book about Monorepo Patterns[24]. Before this was available, I used similar strategies. To help establish a common vocabulary and standard conventions in the community, I am now using the Nx team's methods.

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:

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



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:

```
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[25], Nrwl[26] – the company behind Nx – use the following categories for libraries:

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

- **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:

```
1 export * from './lib/catalog-data-access.module';
2 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.

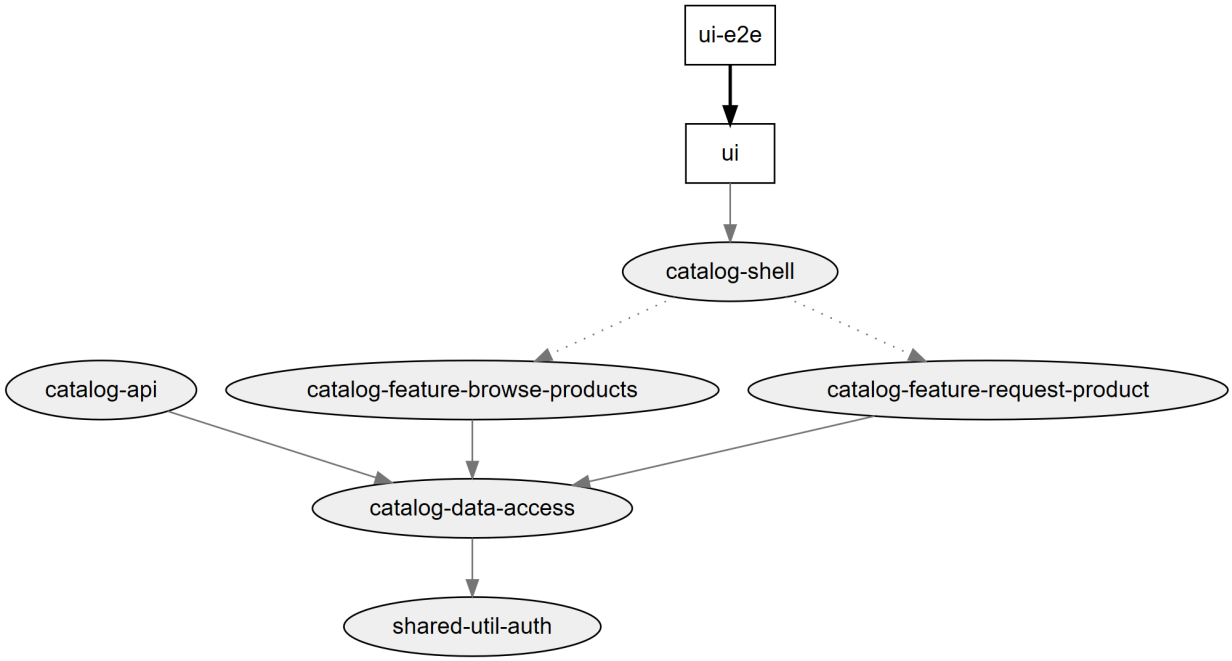
However, the private part can be changed at will, as long as the public part stays the same.

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[27], 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

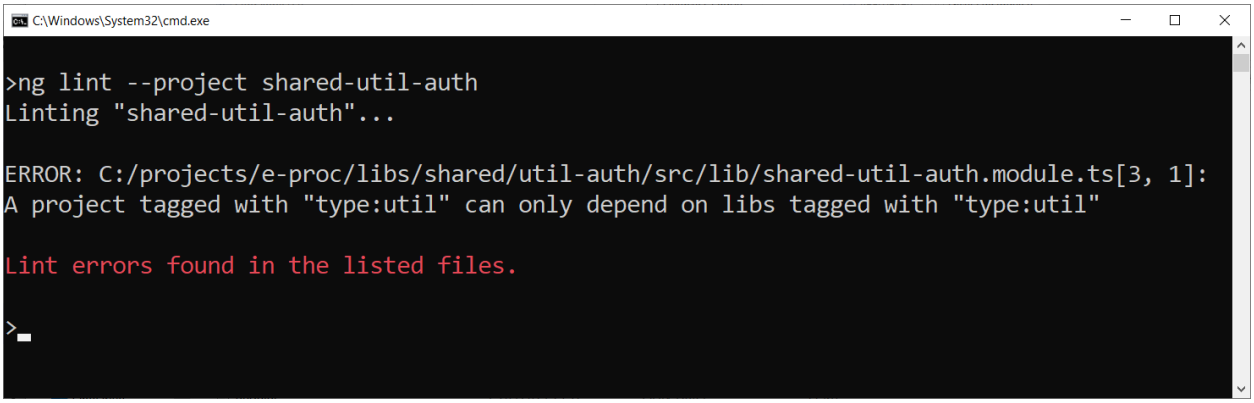
To enforce access restrictions, Nx comes with its own linting rules. As usual, we configure these rules within `tslint.json`:

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

```
"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:



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[28], 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.

[28] 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:

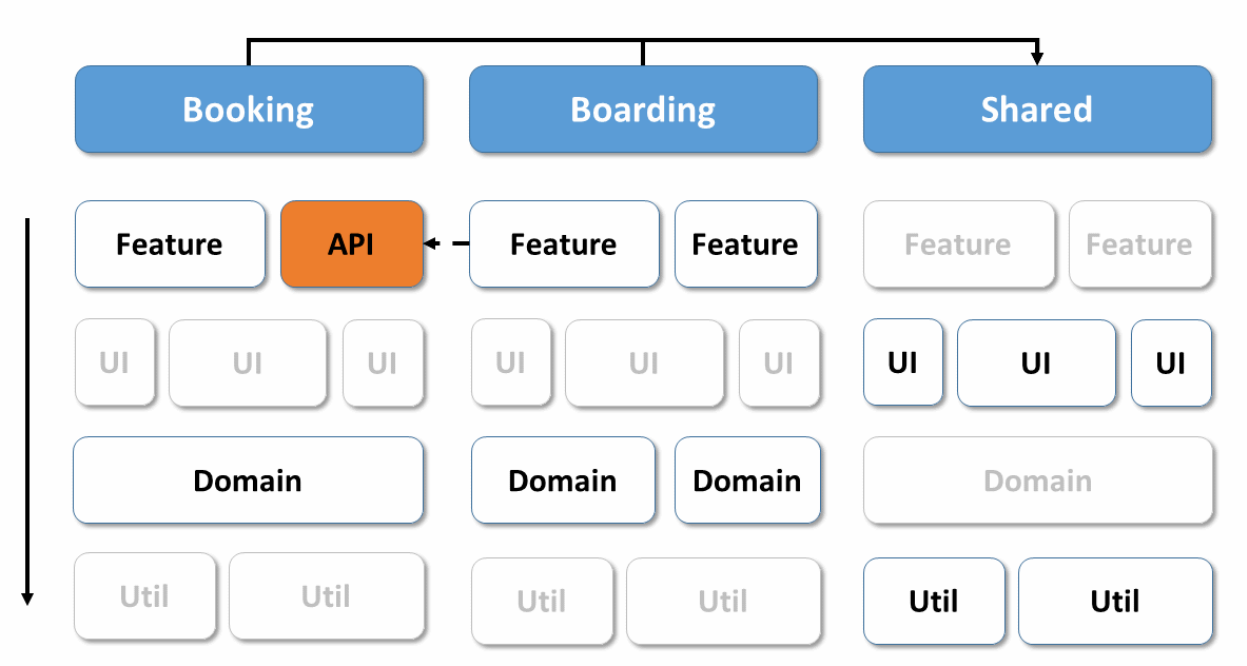
The source code[29] of this case study can be found here[30].

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:

[29] https://github.com/manfredsteyer/angular-ddd
[30] 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[31], I distinguish between five categories of layers or libraries:

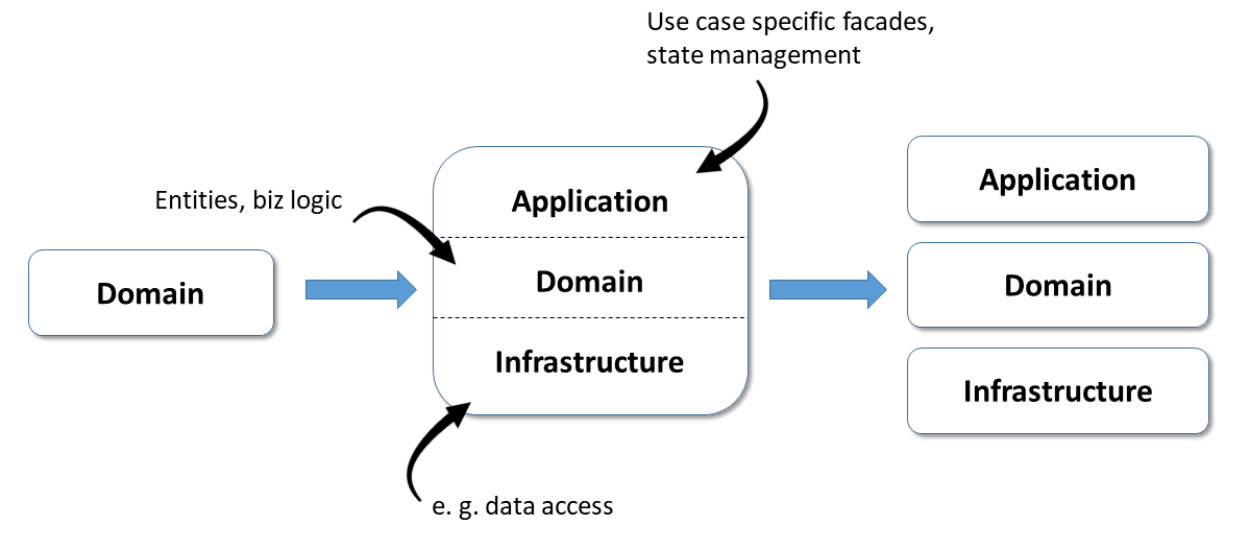
Category	Description	Exemplary content
feature	Contains components for a use case.	search-flight component
ui	Contains so-called "dumb components" that are use case-agnostic and thus reusable.	date-time component, address component, address pipe
api	Exports building blocks from the current subdomain 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:

[31] 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[32] offers additional features which are particularly valuable for large enterprises. These include

[32] https://nx.dev/

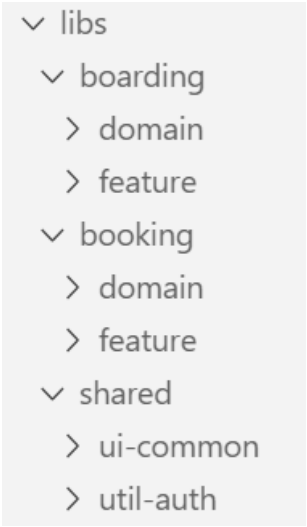
the previously discussed ways to introduce access restrictions between libraries[33]. These restrictions prevent each library from accessing one another, resulting in a highly coupled overall system.

One instruction suffices to create a library in a monorepo:

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

> 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.

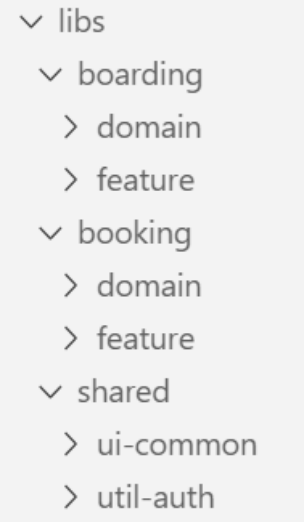
The switch `directory` provided by Nx specifies an optional subdirectory for the libraries, so they can be **grouped by domain**:



The 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:

[33] 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)[34].

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[35],
- Functional and Reactive Domain Modeling[36].

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

Functional programming splits the previously considered entity model into data and logic parts. Domain-Driven Design Distilled[37] 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.

[37] 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[38] 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[39] (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
```

[38] https://www.amzn.com/0134434420
[39] 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*.

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[40].

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

From Domains to Microfrontends

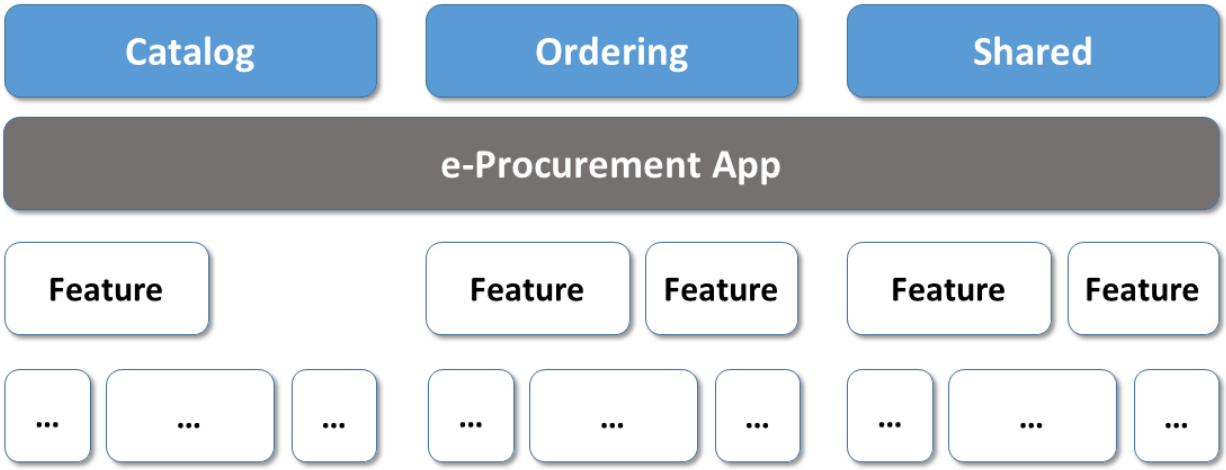
Let's assume you've identified the sub-domains for your system. The next question is how to implement them.

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.

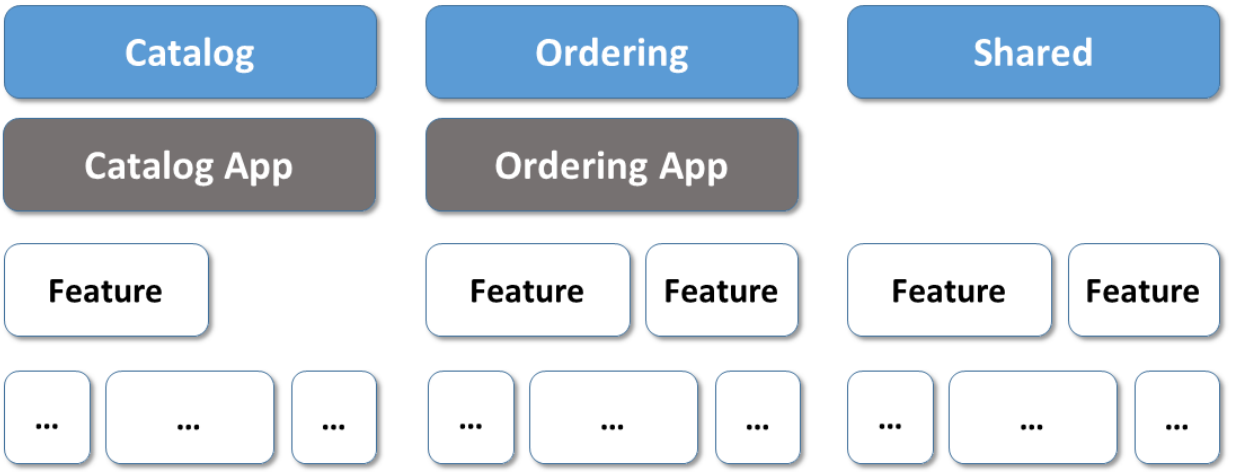
A team responsible for a specific sub-domain must coordinate with other sub-domain teams. They have to agree on overall architecture, the leading framework, and an updated policy for dependencies. Interestingly, you may consider this an advantage.

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[41]. For instance, Nx allows you to define access restrictions between the parts of your monorepo to enforce your envisioned architecture and loose coupling.

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

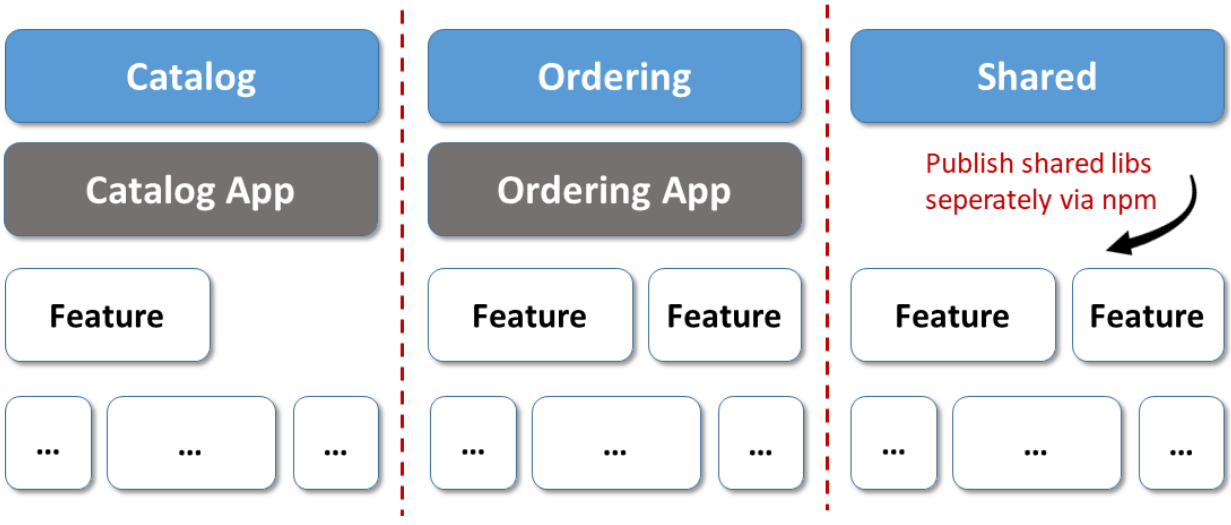
Deployment monoliths, microfrontends, or a mix?

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:



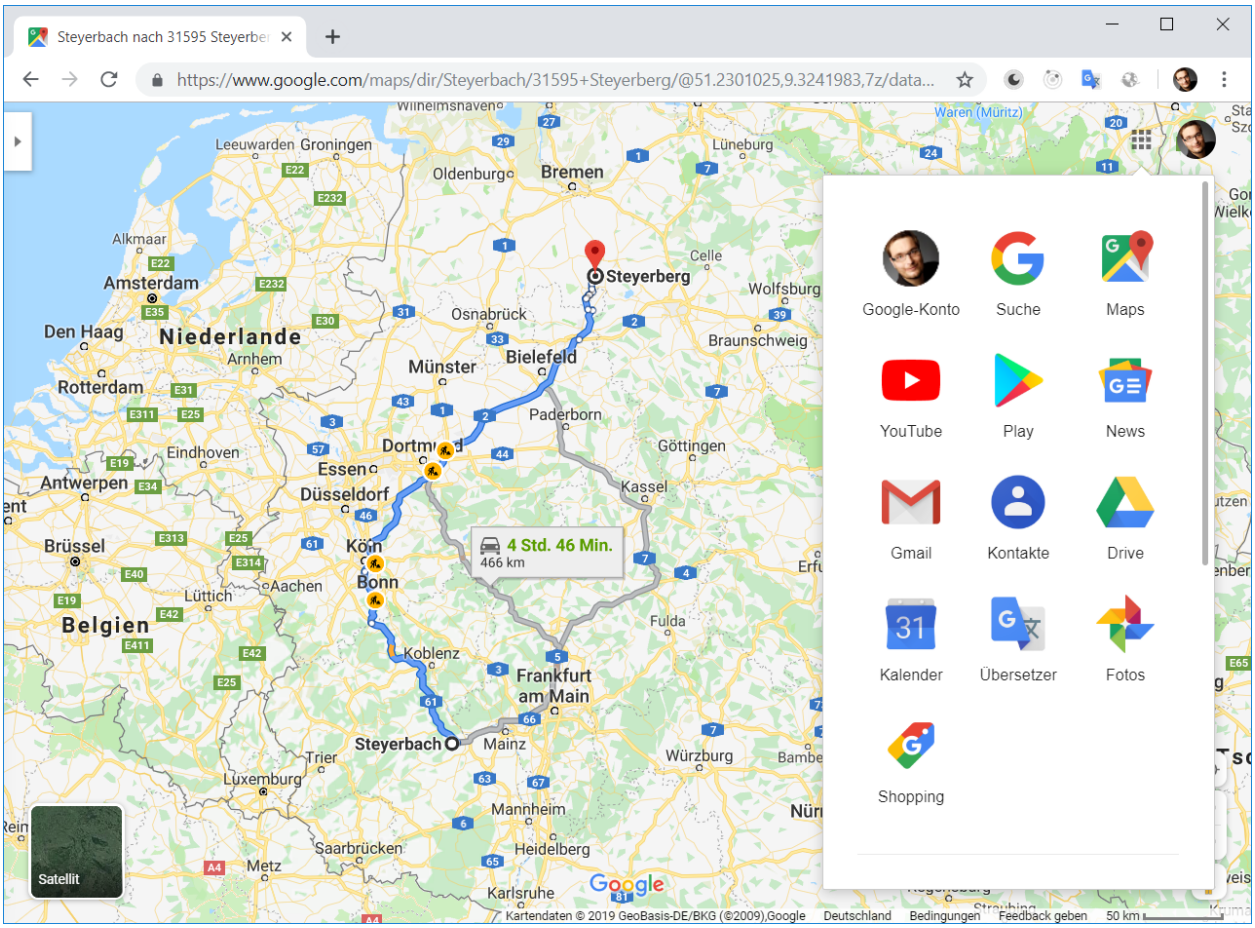
You now have something called microfrontends. Microfrontends allow individual teams to be as autarkic as possible. Each team can choose their architectural style, their technology stack, and they can even decide when to update to newer framework versions. They can use "the best technology" for the requirements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

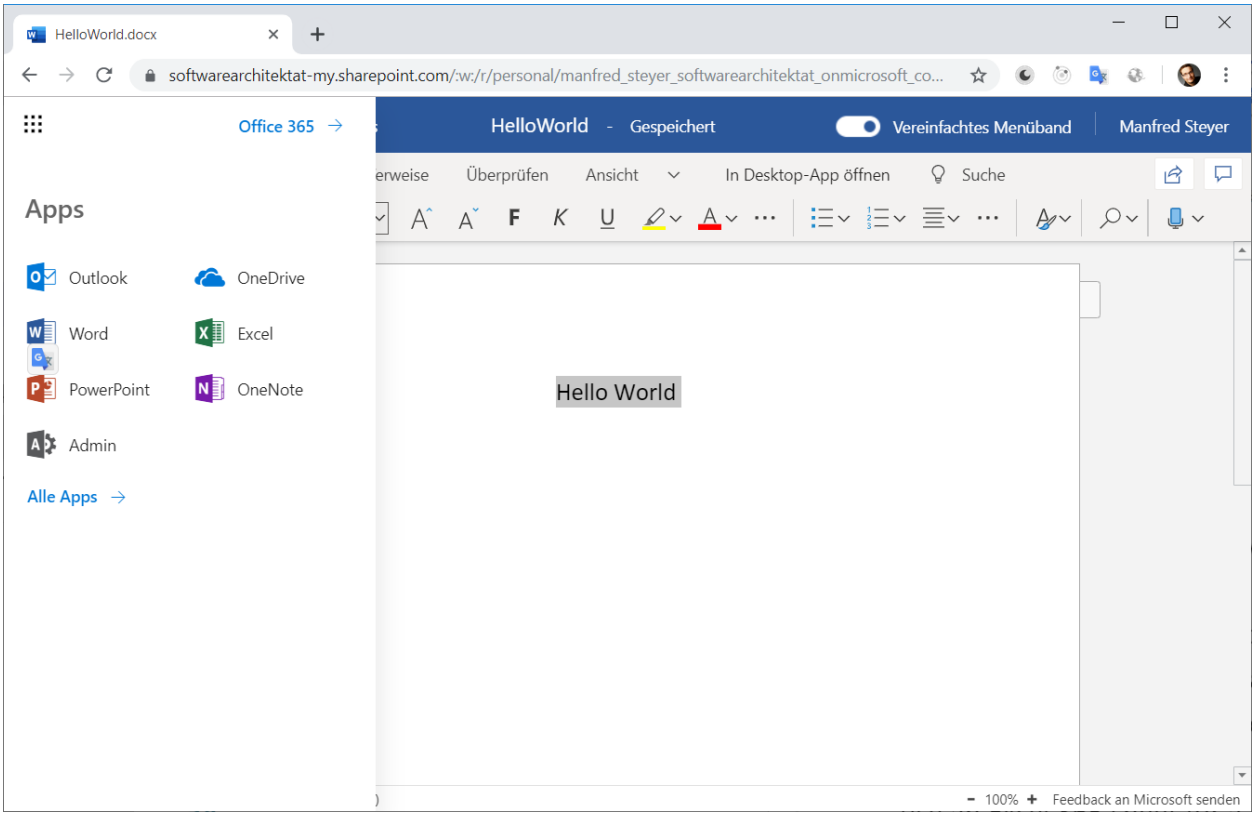
However, this has costs. Now you have to deal with shipping your shared libraries via npm, and this includes versioning which can lead to version conflicts.

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:

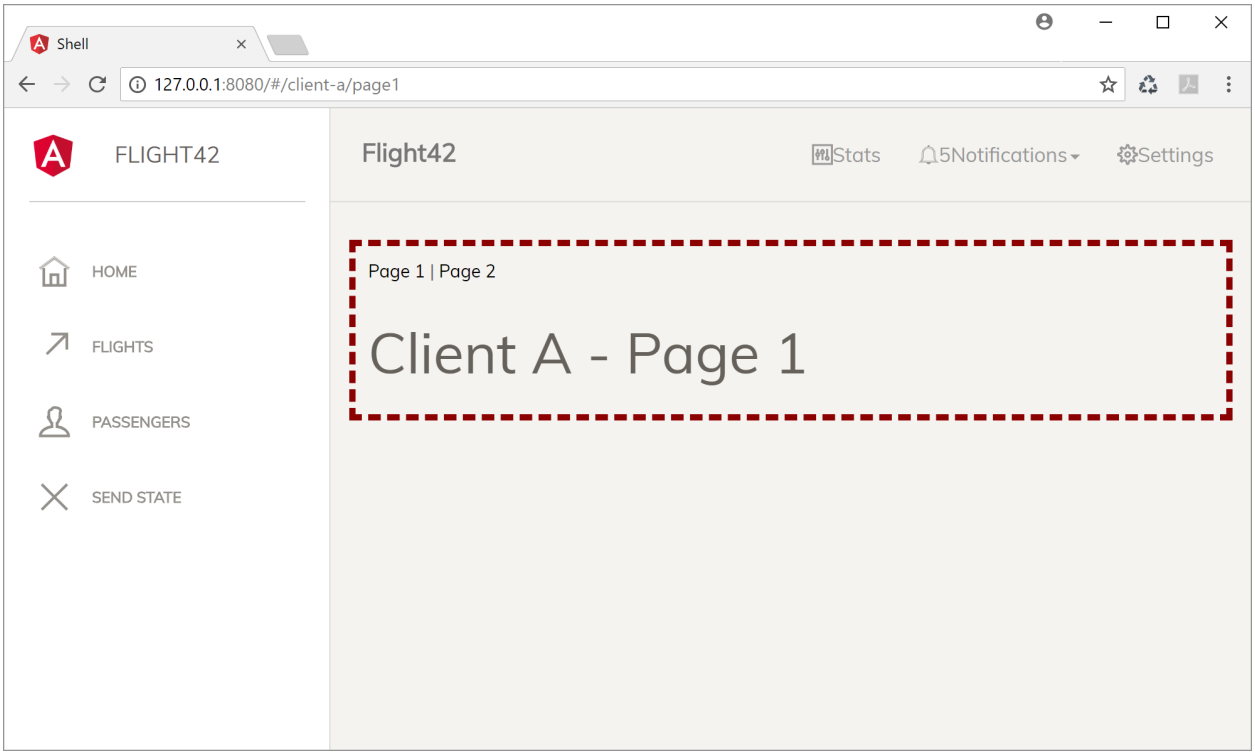
- It is simple
- It uses SPA frameworks as intended
- We get optimised bundles per domain

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[42].

[42] 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.

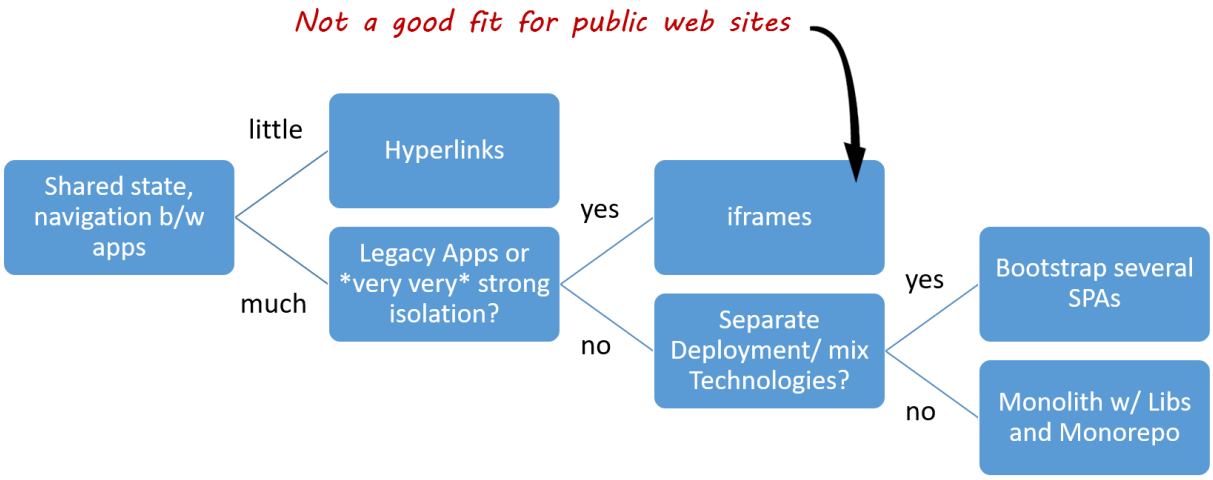
The disadvantages are:

- 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.

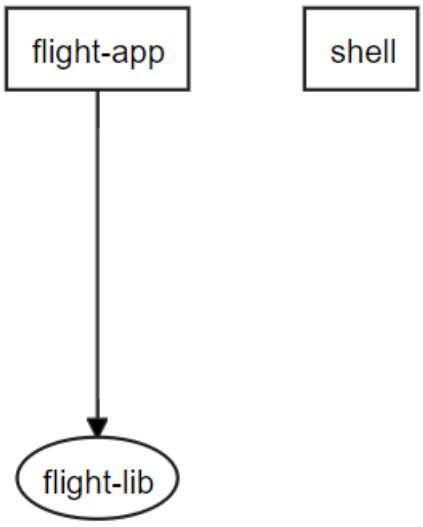
A lightweight Approach towards Microfrontends

For implementing Microfrontends with Angular, there are several approaches and here I present one of them: Loading separately compiled UMD bundles. The source code[43] can be found here[44].

> The approach presented here has been inspired by the work of Victor Savkin[45] and by a great conversation I had with Rob Wormald[46] several months ago.

The Case Study

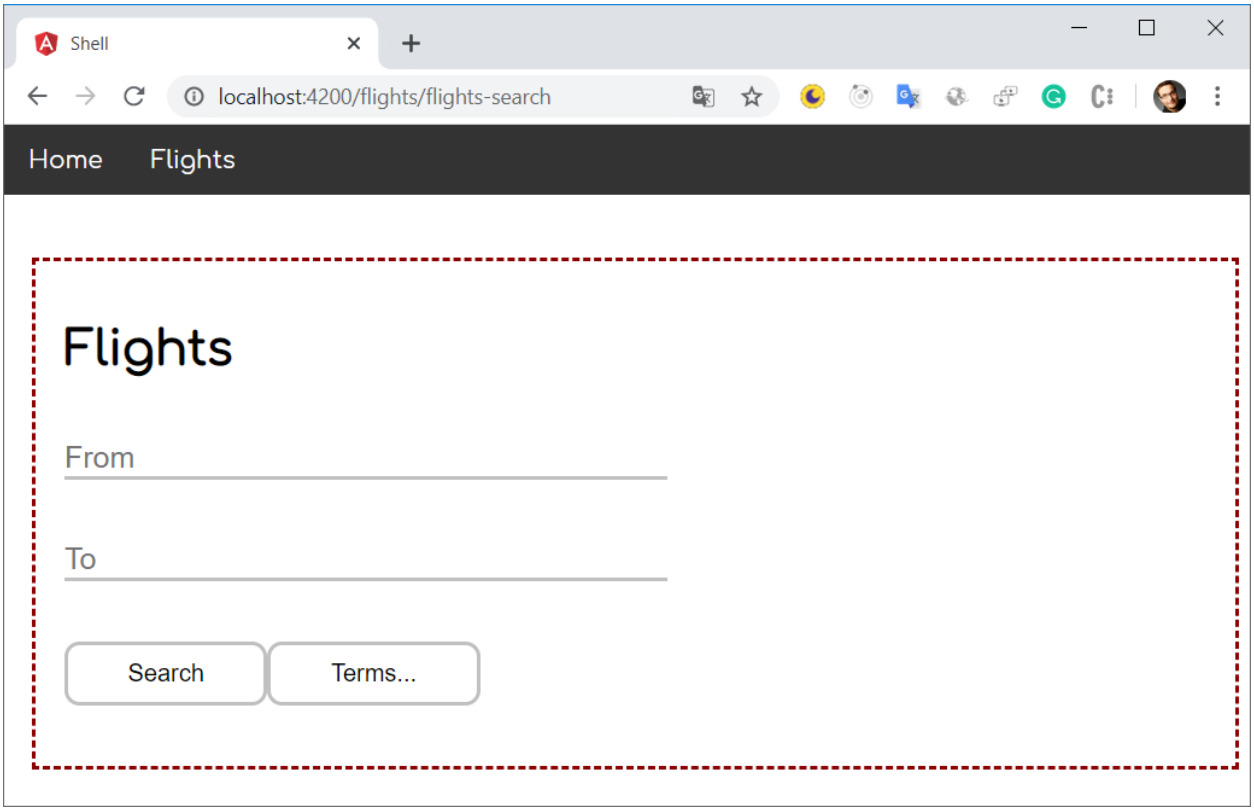
Even though everything I'm showing here works directly with Angular and the Angular CLI, I'm using an Nx workspace[47] because it provides lots of nice features for big systems subdivided into tiny parts. For instance, it allows visualizing them:



Our project's structure

The `shell` app in our example is capable of **loading separately compiled and deployed libraries** like the `flight-lib`:

[43] https://github.com/manfredsteyer/module-federation-light
[44] https://github.com/manfredsteyer/module-federation-light
[45] https://twitter.com/victorsavkin
[46] https://twitter.com/robwormald
[47] https://nx.dev/angular



flight-lib loaded into shell

If we wanted to start the `flight-lib` in **standalone mode**, you would use the `flight-app` which directly references it.

Prevent Duplicates with UMD Bundles

Of course, when loading the `flight-lib` into the shell, we don't want to load all the dependencies like Angular itself one more time. Instead, we want to **reuse the already loaded Angular** instance. For this, we are going to use **UMD bundles** which lookup dependencies in the global namespace if no module loader is used.

Also, UMD bundles can export elements to the global namespace. Our shell will leverage this to get hold of dynamically loaded libraries.

If you wonder how to get UMD bundles for our libraries, I have a good message for you: We get them automatically when building our library because the official Angular Package Format[48] prescribes it among several other formats like ES5 and ES2015 bundles using EcmaScript modules.

[48] https://docs.google.com/document/d/1CZC2rcpxffTDfRDs6p1cfbmKNLA6x5O-NtkJglDaBVs/edit#heading=h.k0mh3o8u5hx

Providing the Library

The `flight-lib` is a traditional feature library with child routes:

```
@NgModule({
  imports: [
    CommonModule,
    RouterModule.forChild([
      { path: 'flights-search', component: FlightComponent  },
      [...]
    ])
  ],
  declarations: [FlightComponent]
})
export class FlightLibModule {}
```

There is really nothing special about it. After building it (`ng build flight-lib`), we get the following folders:

```
dist/libs/flight-lib
├──bundles
├──esm2015
├──esm5
├──fesm2015
├──fesm5
└──lib
```

Each of these folders contains the library compiled into < specific module format. The UMD bundles can be found in the **bundles folder**:

```
18.767 flights-flight-lib.umd.js
30.145 flights-flight-lib.umd.js.map
 5.923 flights-flight-lib.umd.min.js
17.766 flights-flight-lib.umd.min.js.map
```

By looking at the file size of the minimized version, you can tell that it only contains our code and not the code of the dependencies referenced. Now, you only need to **deploy** these files – at least the minimized bundle – **to any webserver** of your choice. To simplify this demonstration, I've just copied it over to the shell's `assets` folder.

Implementing the Shell

Also the shell is a **traditional Angular application**. It just uses two **helper functions** for loading the separately compiled UMD bundle and for sharing libraries like Angular with it. In this section, I thread these functions as a black box. In the next section I will reveal their implementations.

To load the **UMD bundle**, I use the router's lazy loading capabilities:

```
@NgModule({
  declarations: [AppComponent, HomeComponent],
  imports: [
    BrowserModule,
    RouterModule.forRoot([
      {
        path: '',
        component: HomeComponent
      },
      {
        path: 'flights',
        loadChildren: () => loadModule('assets/flights-flight-lib.umd.min.js')
                            .then(g => g.flights['flight-lib'].FlightLibModule)
      }
    ])
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

At first sight, this really looks like traditional **lazy loading**. However, **instead** of a **dynamic import** I'm using the **helper function** `loadModule`. This is needed because when using webpack – and hence the Angular CLI – a dynamic import assumes that the referenced code is already available at compile time. Of course, here, this is not the case as we want to load separately compiled stuff.

After loading, the `then` handler is grabbing the `FlightLibModule` **out of the global namespace**. Fortunately, it is not directly put there but into the sub namespace `flight-lib` which is part of the sub namespace `flights`. Technically, we are talking about nested JavaScript objects here. As `flight-lib` contains a dash (-), we have to use the bracket syntax for referencing it.

> Here, `flights` is the name of the Angular project (and hence the Nx workspace) and `flight-lib` is the library's name. If you are not sure about these names you can look them up in the first 3 to 5 lines of your UMD bundle.

For such bundles, we also need to expose all libraries it needs via the global namespace. For this, I'm calling my helper function

```
initExternals(environment.production);
```

before bootstrapping in the file `main.ts`.

The Helper Functions

Now it's time to look into the two helper functions used by the shell. The function `loadModule` is basically dynamically creating a **script tag** to **load** the passed UMD **bundle**:

```
const moduleMap = {};

export function loadModule(umdFileName: string): Promise<any> {
    return new Promise<any>((resolve, reject) => {

        if (moduleMap[umdFileName]) {
            resolve(window);
            return;
        }

        const script = document.createElement('script');
        script.src = umdFileName;

        script.onerror = reject;

        script.onload = () => {
            moduleMap[umdFileName] = true;
            resolve(window); // window is the global namespace
        }

        document.body.append(script);
    });
}
```

This is a very common way of dynamic script loading which is normally hidden behind libraries. The `moduleMap` makes sure each bundle is only loaded once. Please also note that the returned promise is resolved with the `window` object which **represents the global namespace** in the browser.

The helper function `initExternals` is just **putting** the **shared libraries** into the **global namespace** where the separately compiled UMD bundles expects them:

```
declare const require: any;

export function initExternals(production: boolean) {
    (window as any).ng = {};
    (window as any).ng.core = require('@angular/core');
    (window as any).ng.forms = require('@angular/forms');
    (window as any).ng.common = require('@angular/common');
    (window as any).ng.router = require('@angular/router');
    (window as any).ng.platformBrowser = require('@angular/platform-browser');

    if (!production) {
        (window as any).ng.platformBrowserDynamic = require('@angular/platform-brows\
er-dynamic');
        (window as any).ng.compiler = require('@angular/compiler');
    }
}
```

To find out about the right location like `ng.core` or `ng.common`, just have a look into this libraries' UMD bundles. It should be used there in the first 3 to 5 lines of code.

Conclusion

The approach outlined here is **simple** and can be used with Angular libraries **already today**. The **bundles** are **small** because they only contain the lazily loaded code and no dependencies. They are always **shared** with the shell.

One downside is, that the **shell needs to provide all the dependencies** the library needs. They have to be either part of the shell upfront and exposed as shown here or loaded alongside the library in question. Also, traditional lazy loading does not work from within an UMD bundle as the CLI's build task is not splitting UMD such bundles. However, traditional lazy loading works well in the shell and an UMD bundle can load further UMD bundles as shown above.

The upcoming **Webpack 5 Module Federation provides** (see chapter about Module Federation) a lot **more convenience** out of the box: It doesn't demand you to create separate libraries and it allows the microfrontend to directly come with its own dependencies. To be more precise, both, the microfrontend and the shell can decide which dependencies they want to share. If the shell does not provide a given dependency, the microfrontend can fall back to loading it directly. Also, Module Federation defines all the details in a declarative way.

One benefit of the approach outlined here is that we **don't need to know the amount of microfrontends** during compile time. Instead we could load a configuration at runtime telling us how many microfronts are available and where they can be found. However, eventually there will be a webpack plugin which allows the same.

If you want to get started **today**, the "**light**" **approach** shown here might be the best you can have today. However, **keep an eye to** the further development of **Module Federation** and to approaches for integrating them into the Angular CLI.

The Microfrontend Revolution: Using Module Federation with Angular

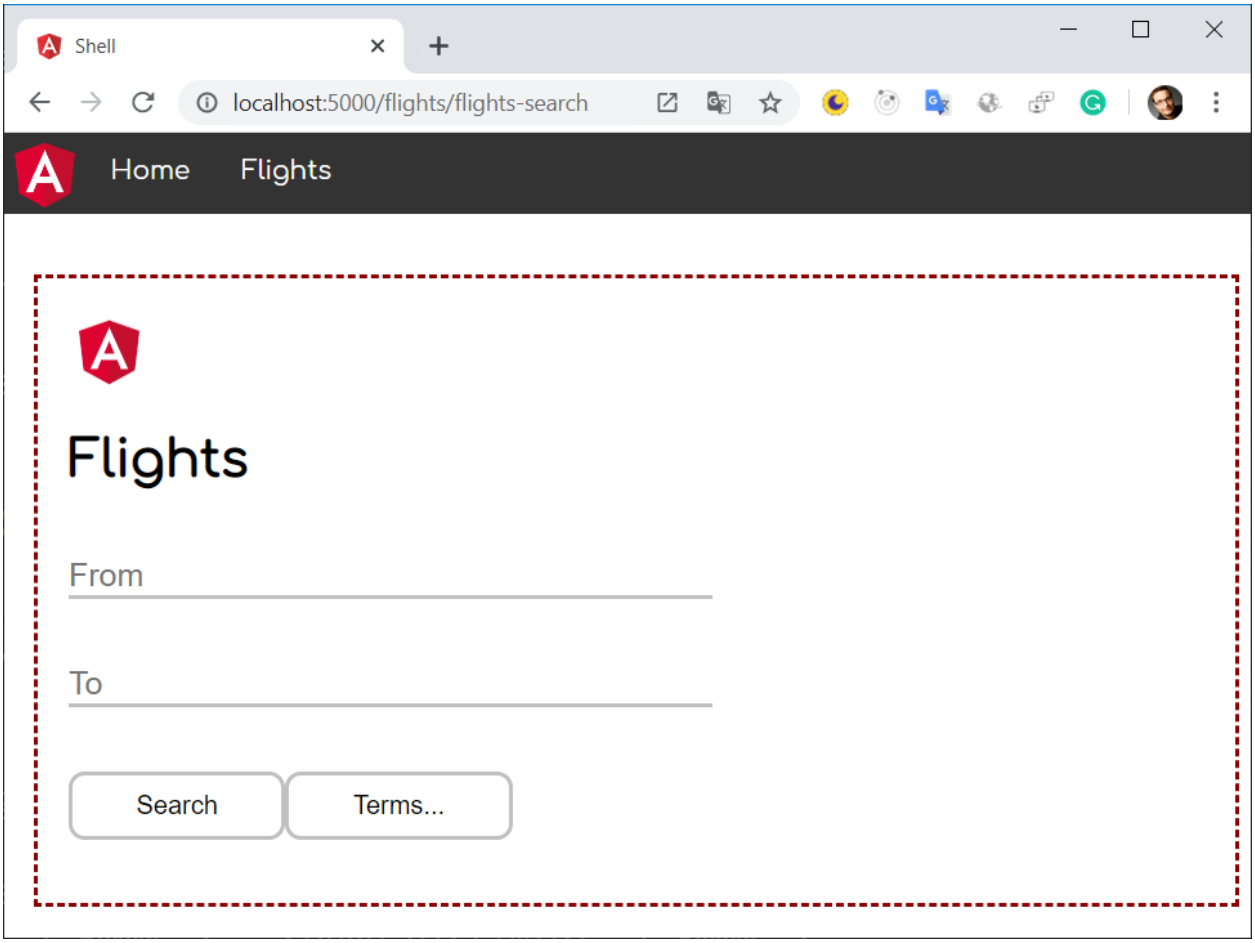
Until now, when implementing microfrontends, you had to dig a little into the bag of tricks. One reason is surely that current build tools and frameworks do not know this concept. Webpack 5, which is currently in BETA, will initiate a change of course here.

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.

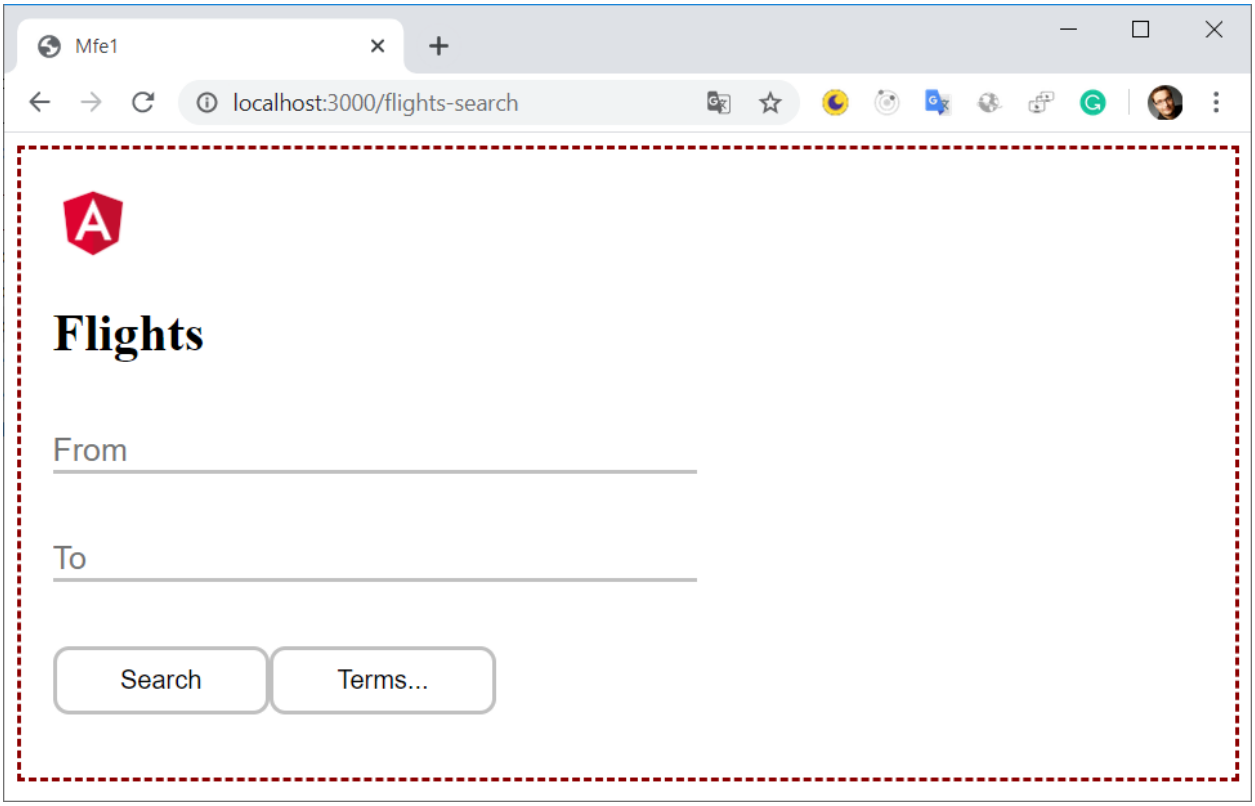
Example

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[49] of the used example can be found in my GitHub account[50].

The Shell (aka Host)

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

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

[49] https://github.com/manfredsteyer/module-federation-with-angular
[50] https://github.com/manfredsteyer/module-federation-with-angular

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 projects. This can be done with the `ContainerReferencePlugin`:

```
new ContainerReferencePlugin({
  remoteType: 'var',
  remotes: {
    mfe1: "mfe1"
  },
  overrides: [
      "@angular/core",
      "@angular/common",
      "@angular/router"
  ]
}),
```

The `remotes` section maps the internal name `mfe1` to the same one defined within the separately compile microfrontend. The **concrete path** of this project **isn't defined here**. That happens later when the shell loads a so called *remote entry point* provided by the microfrontend.

Also, `overrides` contains the names of libraries our shell shares with the microfrontend.

While the `ContainerReferencePlugin` shown here is intended for the shell, the `ContainerPlugin` used below is the one for configuring the microfrontends.

Perhaps you've noticed that the previous chapter only used the `ModuleFederationPlugin`. However, this plugin is just delegating to both[51], the `ContainerReferencePlugin` and the `ContainerPlugin`. Projects configured with it can be used as a shell and a microfrontend at the same time.

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`:

[51] https://github.com/webpack/webpack/blob/dev-1/lib/container/ModuleFederationPlugin.js

```
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 need to reference the `ContainerPlugin` in our webpack configuration:

```
new ContainerPlugin({
  name: "mfe1",
  filename: "remoteEntry.js",
  exposes: {
    Module: './projects/mfe1/src/app/flights/flights.module.ts'
  },
  library: { type: "var", name: "mfe1" },
  overridables: [
      "@angular/core",
      "@angular/common",
      "@angular/router"
  ]
}),
```

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

Also, the microfrontend's configuration must point to the location it will be deployed at:

```
output: {
    publicPath: "http://localhost:3000/",
    [...]
},
```

Obviously, it would be better if we could specify this path at runtime. Fortunately, when this chapter was written, there was already a Pull Request[52] for this.

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 `ContainerPlugin` or 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 check which libraries are already loaded by the shell and which ones need to be loaded in addition.

Unfortunately, the Angular Compiler assumes that some parts of Angular are imported in a static way. Also, the CLI produces scaffolds such code.

To bypass this show-stopper, we can use a tiny trick: Let's move the contents of the entry point `main.ts` into a file called `bootstrap.ts`. Then, dynamically import `bootstrap.ts` in `main.ts`.

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

This dynamic import gives the infrastructure the time necessary for loading everything the shell hasn't already loaded. Hence, in the remaining parts of the application – e. g. in `main.ts` – we can leverage the usual static imports:

[52] https://github.com/webpack/webpack/pull/10703

```
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));
```

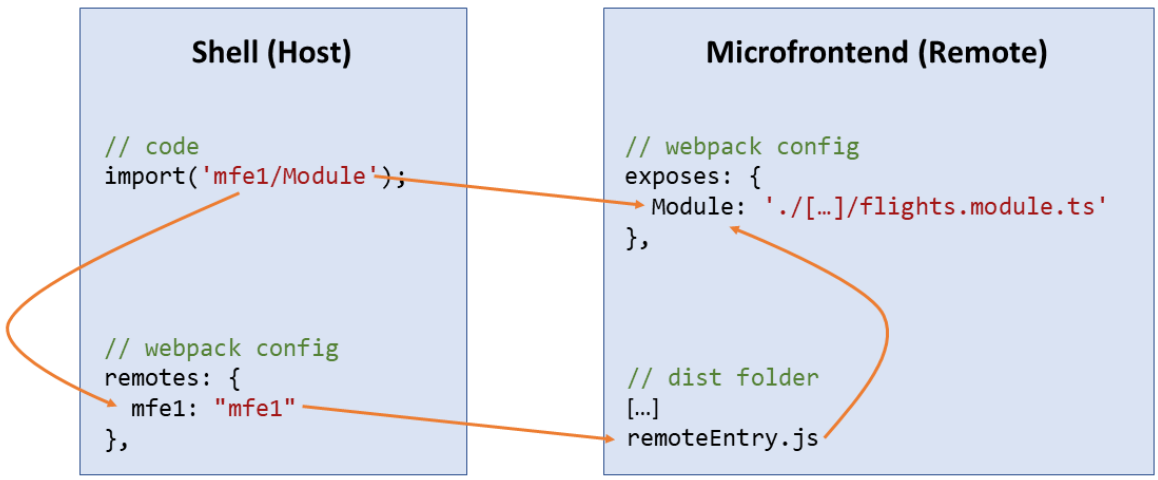
This trick is not needed for the shell shown above because it's configured with the `ContainerReferencePlugin`. If we configured it with the `ModuleFederationPlugin` instead, we would need to go with the workaround shown, however.

Connecting the Shell and the Microfrontend

Now, everything left to do, is letting the shell know where to find the microfrontend. For this, we just load the *remote entry point* created when webpack bundles the microfrontend into the shell:

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

This is just a tiny script bridging the gap between the shell and the microfrontend. As the result, the shell can now load the separately compiled `FlightsModule` using the path `mfe1/Module`:



Connecting the Shell and the Microfrontend

Evaluation and Outlook

With Module Federation we have a proper and official solution for building microfrontends with webpack and hence the Angular CLI for the first time.

It will be part of webpack 5 which was in BETA when writing this. As the current version of the CLI is still using webpack 4, we need a **custom webpack build**. This will hopefully change in the near future.

Also, we need to deal with the fact that **shared libraries** need to be referenced via **dynamic imports**. Their asynchronous behavior allows the infrastructure to load the libraries not already provided by the shell. As outlined, we can bypass this by using the **ContainerReferencePlugin** for the shell and by just dynamically importing the the **bootstrapping logic** in the microfrontends.

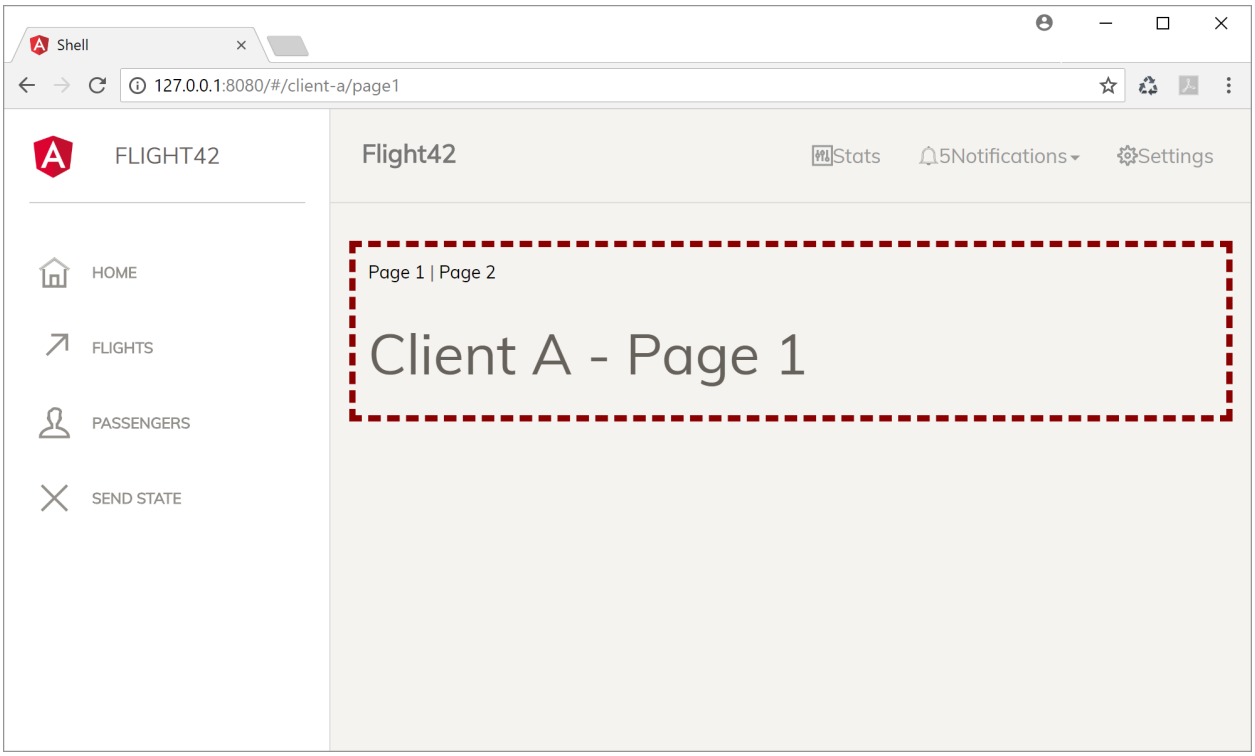
Six Steps to your Angular-based Microfrontend Shell (Heavy-weight approach)

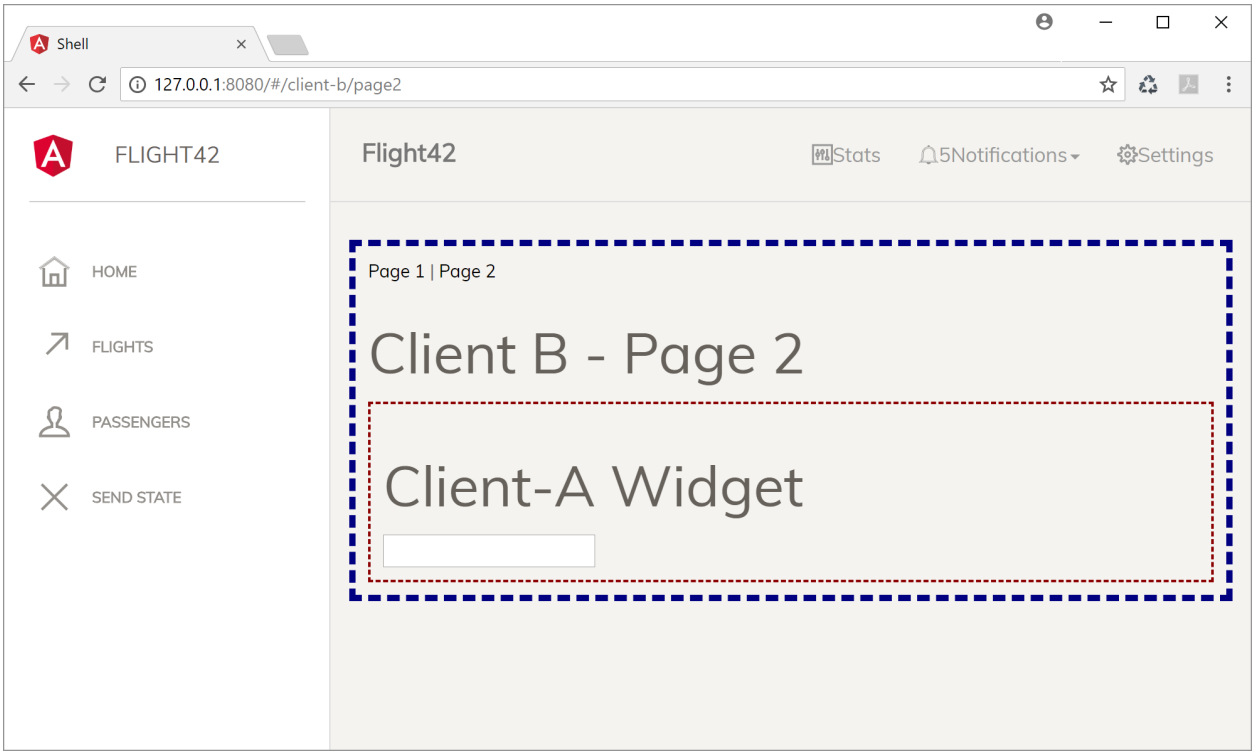
The former chapters showed rather lightweight approaches towards implementing microservices with Angular: One was about loading UMD bundles and the other one was using Webpack 5 Module Federation which was still in beta when writing this.

Both approaches assume that you are using first and foremost Angular. Hence, they can use the router for lazy loading the separately compiled microfrontends.

This chapter shows a more heavy-weight approach. It demands you to deal with more aspects by yourself. However, it also allows you to mix and match different SPA frameworks.

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[53] for this in my GitHub account here[54].

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[55]) 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[56] provides an excellent solution for

[53] https://github.com/manfredsteyer/angular-microfrontend
[54] https://github.com/manfredsteyer/angular-microfrontend
[55] https://martinfowler.com/chapters/microservices.html#ComponentizationViaServices
[56] 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[57] and lazy and external Angular Elements[58].

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

[57] https://www.softwarearchitekt.at/aktuelles/angular-elements-part-i/
[58] 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[59] 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;
```

[59] 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: `typescript (window as any).messageBus = new BehaviorSubject<MicroFrontendEvent>(null);`

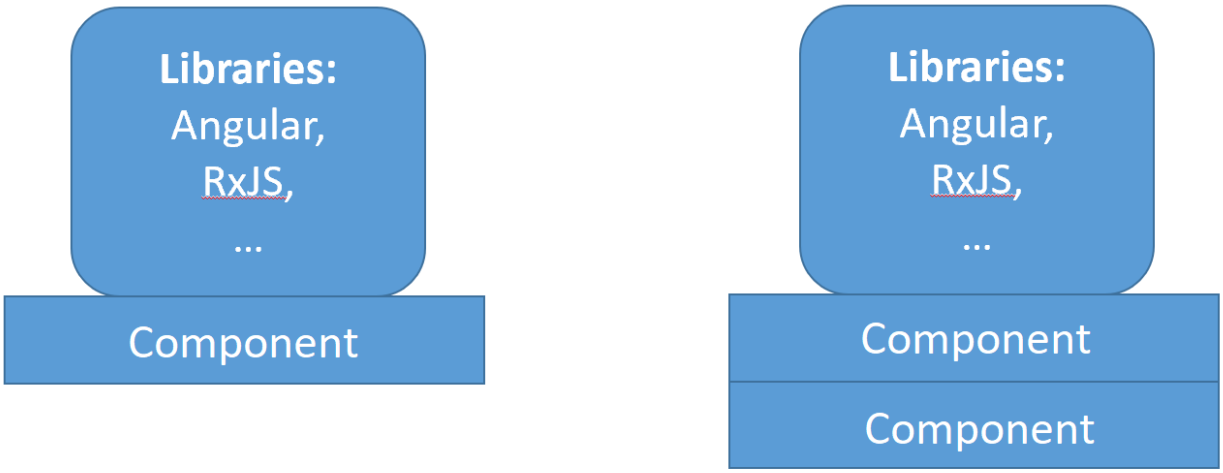
   The shell and the microfrontends can subscribe to this message bus and listen for specific events. Both can emit events.
3. Using custom events provided by the browser: "'typescript // Sender const customer = { id: 17, ... }; window.raiseEvent(new CustomEvent('CustomerSelected', {details: customer}))

   // Receiver window.addEventListener('CustomerSelected', (e) ⇒ { ... }) "'

Step 6: Sharing Libraries Between Microfrontends

As we have several self-contained microfrontends, each has its dependencies, e.g. Angular itself or RxJS. From a microservice perspective, this is perfect as it allows each microfrontend team to choose any library or framework and any version. They can even decide if and when to update to newer versions.

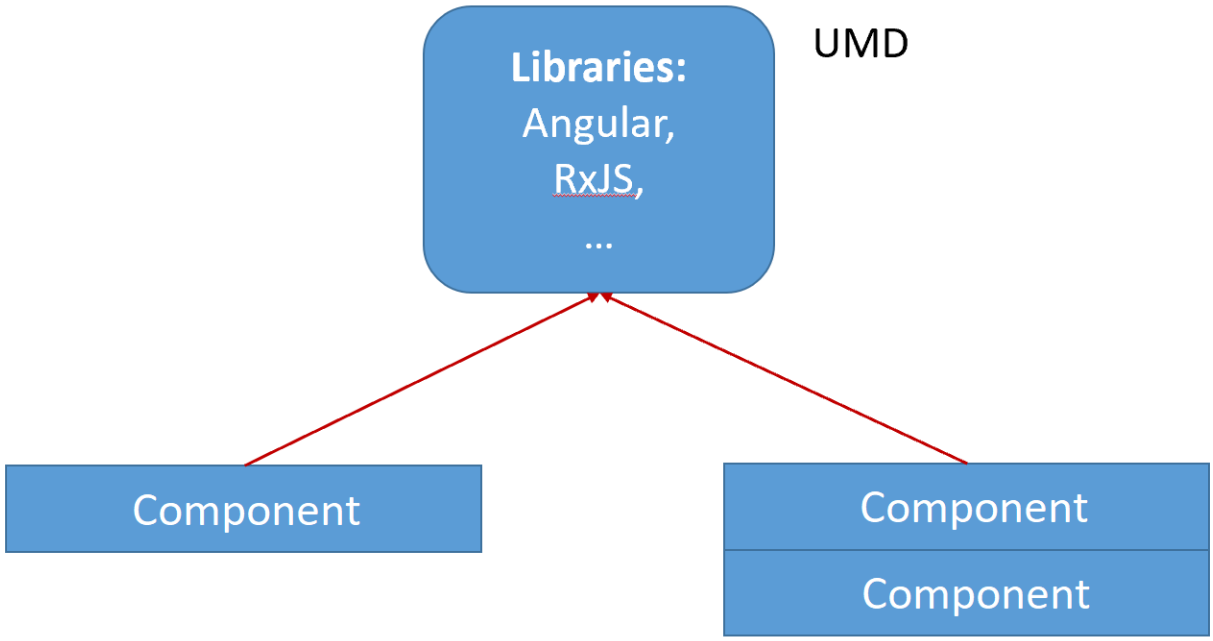
However, from the perspective of performance and loading time, this structure is less appealing as it leads to code duplication within the bundles. For instance, you could end up with a specific angular version in several bundles:



Fortunately, there is a solution: Webpack externals.

Externals allow us to share libraries by loading them into the global namespace. This approach was popular in the days of jQuery (which provided a global $ object) and is still sometimes done for simple react and vue applications.

We can use UMD bundles for most libraries. Simply tell webpack not to bundle them together with every microfrontend, but rather to reference it within the global namespace:



To use webpack externals together with the Angular CLI, you can leverage ngx-build-plus[60] which even comes with a schematic introducing the needed changes into your application.

As mentioned above, you can install it with `ng add`:

```
ng add ngx-build-plus
```

Then, call the following schematic:

```
ng g ngx-build-plus:externals
```

Please remember that within a monorepo you have to provide the name of the project in question:

```
ng add ngx-build-plus --project myProject
ng g ngx-build-plus:externals --project myProject
```

This approach also introduces an npm script `build:<project-name>:externals`. For the default project, there is a script `build:externals` too.

If you look into the `index.html` after running this script, you see that Angular is directly loaded:

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

```
<!-- Angular Packages -->
<script src="./assets/core/bundles/core.umd.js"></script>
<script src="./assets/common/bundles/common.umd.js"></script>
<script src="./assets/common/bundles/common-http.umd.js"></script>
<script src="./assets/elements/bundles/elements.umd.js"></script>

<script src="./assets/forms/bundles/forms.umd.js"></script>
<script src="./assets/router/bundles/router.umd.js"></script>
```

To optimise this, one could put those parts of Angular into one bundle.

If you look into the generated `webpack.externals.js`, you find a section mapping package names to global variables:

```
const webpack = require('webpack');

module.exports = {
    "externals": {
        "rxjs": "rxjs",
        "@angular/core": "ng.core",
        "@angular/common": "ng.common",
        "@angular/common/http": "ng.common.http",
        "@angular/platform-browser": "ng.platformBrowser",
        "@angular/platform-browser-dynamic": "ng.platformBrowserDynamic",
        "@angular/compiler": "ng.compiler",
        "@angular/elements": "ng.elements",
        "@angular/router": "ng.router",
        "@angular/forms": "ng.forms"
    }
}
```

This method, for instance, makes the produced bundle reference the global variable `ng.core` when it needs `@angular/core`. Hence, `@angular/core` no longer needs to be in the bundle.

Please note that this is not the default operation mode for Angular and has some risks.

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[61]
- Wlaschin, Domain Modeling Made Functional[62]
- Ghosh, Functional and Reactive Domain Modeling[63]
- Nrwl, Monorepo-style Angular development[64]
- Jackson, Micro Frontends[65]
- Burleson, Push-based Architectures using RxJS + Facades[66]
- Burleson, NgRx + Facades: Better State Management[67]
- Steyer, Web Components with Angular Elements (article series, 5 parts)[68]

[61] https://www.amazon.com/dp/0321125215
[62] https://pragprog.com/book/swdddf/domain-modeling-made-functional
[63] https://www.amazon.com/dp/1617292249
[64] https://go.nrwl.io/angular-enterprise-monorepo-patterns-new-book
[65] https://martinfowler.com/articles/micro-frontends.html
[66] https://medium.com/@thomasburlesonIA/push-based-architectures-with-rxjs-81b327d7c32d
[67] https://medium.com/@thomasburlesonIA/ngrx-facades-better-state-management-82a04b9a1e39
[68] 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 Consultancy

If you and your team need help regarding Angular, we are happy to help with our online and on-site workshops.

Company-Trainings (Online and On-Site, English or German)

We offer all our trainings in **English and German** as company trainings (online or on-site). Here you find an overview:

- Angular Workshop[69]
- Angular Architecture Workshop (Advanced)[70]

Consultancy (English or German)

We also offer Angular consultancy:

- Architecture Consultancy
- Reviews
- Project Support
- Q&A Workshops

Please find further details here[71].

Let's Keep In Touch

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[72] and/ or follow the book's author on Twitter[73].

[69] https://www.angulararchitects.io/schulungen/angular-strukturierte-einfuehrung/
[70] https://www.angulararchitects.io/schulungen/advanced-angular-enterprise-anwendungen-und-architektur/
[71] https://www.angulararchitects.io/beratung/
[72] https://www.angulararchitects.io/subscribe/
[73] https://twitter.com/ManfredSteyer