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 2019-10-20

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 copy of @ManfredSteyer's free e-book about Enterprise Angular: DDD, Nx Monorepos, and Micro Frontends

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

In the last years, I've helped numerous companies implementing Angular based large scale enterprise applications. To handle the complexity of such applications, it's vital to decompose the whole system into smaller libraries.

However, if you end up with lots of libraries which are too much intermingled with each other, you don't win much. If everything depends on everything else, you cannot easily change or extend your system without introducing breaking changes.

Domain-driven Design, esp. its discipline strategic design, helps with this and it also can be the foundation for building micro frontends.

In this book, which bases upon several blog articles I've written about Angular, DDD, and Micro Frontends, I show how to use these ideas.

If you have any questions or feedback, feel free to reach out at `book@softwarearchitekt.at`. Perhaps you also want to connect with me at Twitter (https://twitter.com/ManfredSteyer) or Facebook (https://www.facebook.com/manfred.steyer) so that we can stay in touch and you get all the updates about my work regarding Enterprise Angular.

Case Studies

To show all the concepts in action, this book uses several case studies. You can find them in my GitHub account:

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

Help to Improve this Book!

If you find typos or if you have other ideas to improve this book, feel free to contribute. For this, just 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

Thanks

I want to thank several people who helped me with this work:

- The great people at Nrwl.io[5] who provide the open source tool Nx[6] which is also used in the case studies here and described in the following chapters.
- Thomas Burleson[7] who did a great job on describing the idea of facades. The chapter about Tactical Design uses it. Also, Thomas contributed to this chapter.

[5] https://nrwl.io/
[6] https://nx.dev/angular
[7] https://twitter.com/thomasburleson?lang=de

Strategic Domain Driven Design

Monorepos allow huge enterprise applications to be subdivided into small and maintainable libraries. This is, however, only one side of the coin: First, we need to define criteria for slicing our application into individual parts. Also, we must establish rules for the communication between them.

In this chapter, I present a methodology I'm using for subdividing big software systems into individual parts: It's called Strategic Design and it's part of the domain driven design[8] (DDD) approach. Also, I show how to implement its ideas with an Nx[9]-based monorepo.

What's Domain Driven Design about?

DDD describes an approach that bridges the gap between requirements for complex software systems on the one hand and a appropriate application design on the other. It can be subdivided into the disciplines Tactical Design and Strategic Design. The former proposes concrete concepts and patterns for an object-oriented design respective architecture. It has an opinionated view on using OOP. As an alternative, there are approaches like Functional Domain Modeling[10] that transfer the ideas behind it into the world of functional programming.

By contrast, strategic design deals with the breakdown of a large system into individual (sub-)domains and their design. No matter if you like DDD's opinionated view or not, some ideas from Strategic Design have proven to be useful for subdividing a system into smaller, self-contained parts. It is exactly these ideas that this chapter takes up and presents in the context of Angular. Whether the remaining aspects of DDD are also taken into consideration, however, is irrelevant for the chapter.

Finding Domains with Strategic Design

One goal of Strategic Design is to identify self-contained domains. Such are recognizable by a specific vocabulary. Both domain experts and developers must rigorously use this vocabulary in order to prevents misunderstandings. As this language is also used within the code the application mirrors its domain and hence is more self-describing. DDD refers to this as the ubiquitous language[11].

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

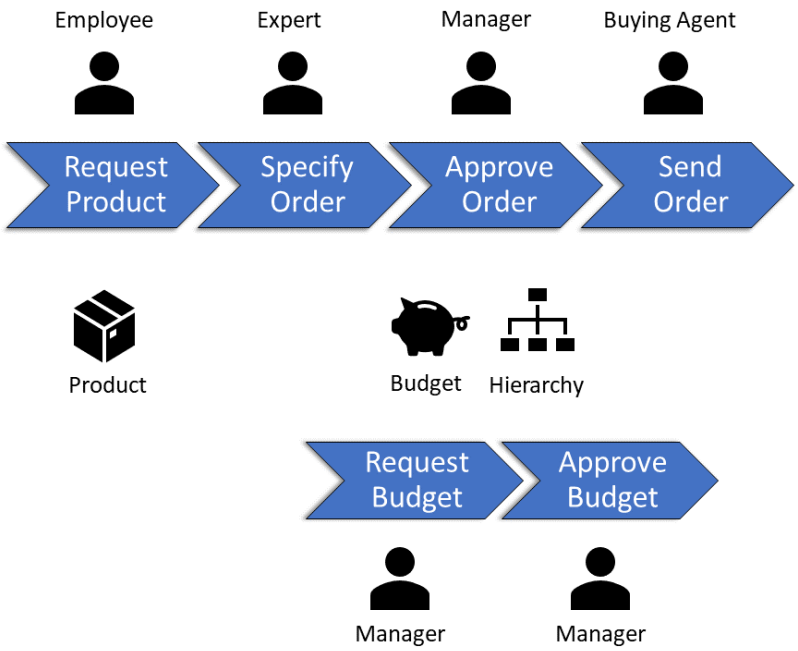
[8] https://www.amazon.de/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215/ref=sr_1_3?ie=UTF8&qid=1551688461&sr=8-3&keywords=ddd

[9] https://nx.dev/

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

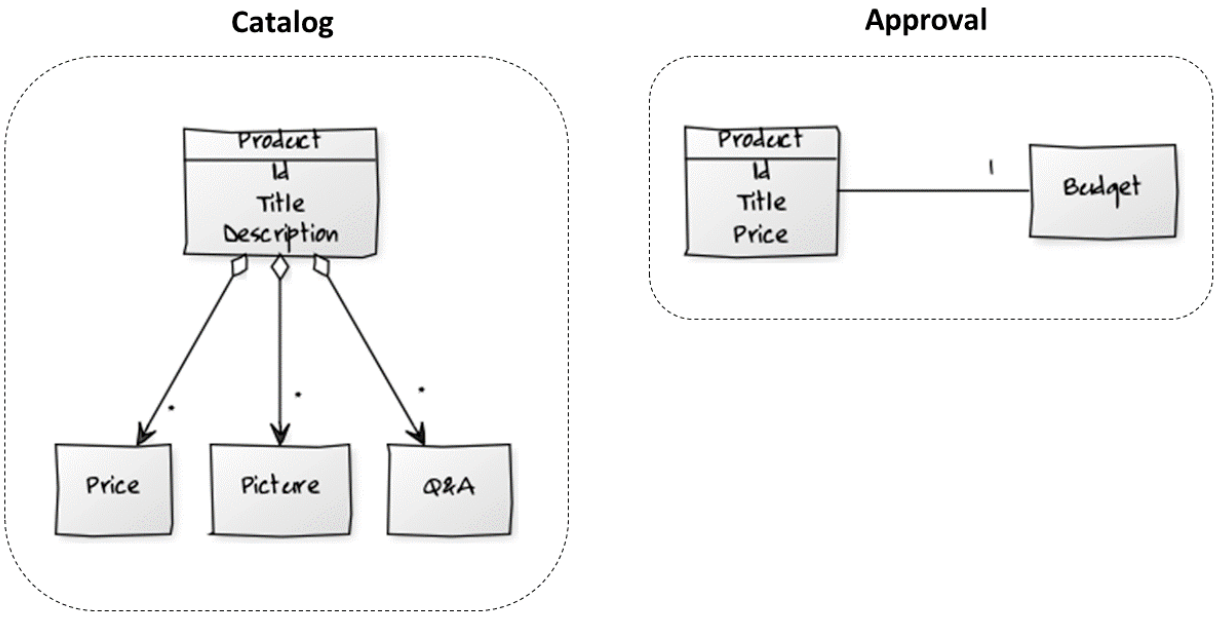
[11] https://martinfowler.com/bliki/UbiquitousLanguage.html

To recognize 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:



It is noticeable that the process steps `Approve Order`, `Request Budget` and `Approve Budget` primarily revolve around organizational hierarchies and the available budget. In addition, managers are primarily involved here. By contrast, the process step is primarily about employees and products.

Of course, it can be argued that products are omnipresent in an e-Procurement system. However, a closer look reveals that the word `product` denotes different things in some of the process steps shown here. For example, while a product is very detailed when it is selected in the catalog, the approval process only needs to remember a few key data:



In the sense of the ubiquitous language that prevails within each domain, a distinction must be made between these two forms of a product. This leads to the creation of different models that are as concrete as possible and therefore meaningful.

At the same time, this approach prevents the creation of a single model that attempts to describe the entire world. Such models are often confusing and ambiguous. In addition, they have too many interdependencies that make decoupling and subdividing impossible.

At a logical level, the individual views of the product may still be related. If this is expressed by the same id on both sides, this works without technical dependencies.

Thus, each model is valid only within a certain scope. DDD also calls this the bounded context[12]. Ideally, each domain has its own bound context. As the next section shows, however, this goal can not always be achieved when integrating third-party systems.

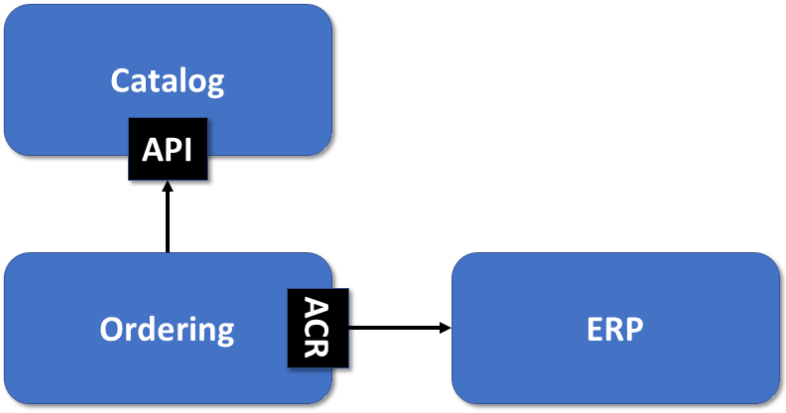
If we proceed with this kind of analysis we might come up with the following domains:

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

If you like the shown process oriented approach of identifying different domains alongside the vocabulary (entities) and groups of domain experts, you might love Event Storming[13]. This is a workshop format where several domain experts analyze business domains together.

Context-Mapping

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



The way these domains interact with each other is determined by a context map. In principle, `Ordering` and `Booking` could share the common model elements. In this case, however, care must be taken to ensure that one modification does not entail inconsistencies on the other.

One domain could easily use the other. In this case, however, the question arises as to who in this interaction is entitled to how much power. Can the consumer impose certain changes on the provider

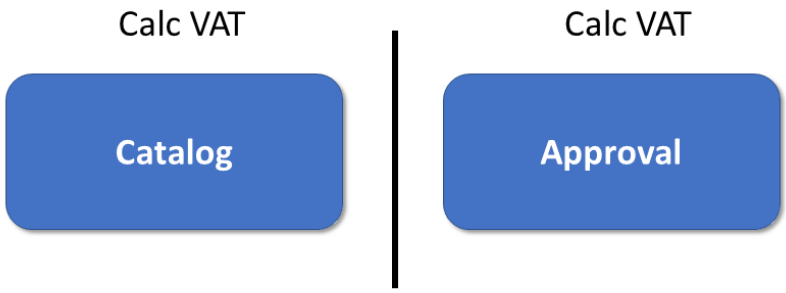
[13] https://www.eventstorming.com

and insist on backward compatibility? Or does the consumer have to be satisfied with what it gets from the provider?

In the case under consideration, Catalog offers an API to prevent changes in the domain from forcibly affecting consumers. Since ordering 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 has to update it. In addition, Strategic Design defines further strategies for the relationship between consumers and providers.

An existing system like the shown ERP system does normally not follow the idea of the bounded context. Rather, it contains several logical and intermingled sub-domains.

Another possible strategy I want to stress out here is `Separate Ways`, which means that a specific aspect like calculating the VAT is separately implemented in several domains:



At first sight this seems to be awkward, especially because it leads to code redundancies and hence breaks the DRY principle (don't repeat yourself). Nevertheless, in some situations it comes in handy because it prevents a dependency to a shared library. While preventing redundant code is an important goal preventing dependencies is as well vital. The reason is that each dependency also defines a contract and contracts are hard to change. Hence, it's a good idea to evaluate whether an additional dependency is worth it.

As above mentioned, each domain should have its own bounded context. The example shown here contains an exception: If we have to respect an existing system like the ERP system it might contain

Conclusion

Strategic Design is about identifying self-contained (sub-)domains. In each domain we find an ubiquitous language and concepts that only make sense within the domain's bounded context. A context map shows how those domains interact with each other.

In the next chapter we'll see we can implement those domains with an Angular using an Nx[14]-based monorepo.

[14] https://nx.dev/

Implementing Strategic Design with Nx Monorepos

In the previous chapter, I've presented the idea of Strategic Design which allows to subdivide a software system into several self-contained (sub-)domains. In this part, I show how to implement those domains with Angular and an Nx[15]-based monorepo.

If you want to have a look into the underlying case study[16], you find the source code here[17]

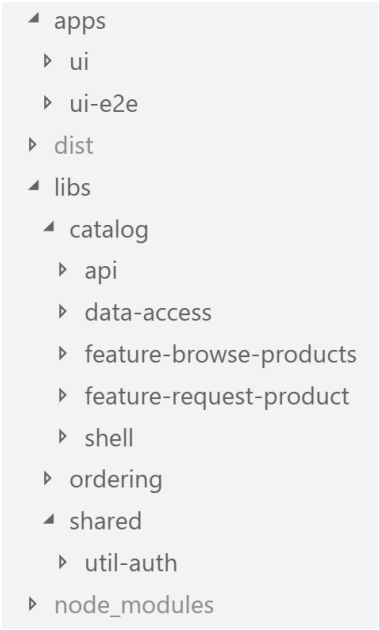
For this, I'm following some recommendations the Nx team recently wrote down in their free e-book about Monorepo Patterns[18]. Before this was available, I've used similar strategies but in order to help establishing a common vocabulary and common conventions in the community, I seek to use this official way.

Implementation with Nx

For the implementation of the defined architecture, a workspace based on Nx [Nx] is used. This is an extension for the Angular CLI, which among other things helps to break down a solution into different applications and libraries. Of course this is just one of several possible approaches. As an alternative, one could, for example, implement each domain as a completely separate solution. This would be called a micro-app approach.

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

[15] https://nx.dev/
[16] https://github.com/manfredsteyer/strategic-design
[17] https://github.com/manfredsteyer/strategic-design
[18] https://go.nrwl.io/angular-enterprise-monorepo-patterns-new-book



Because such a workspace consists of several applications and libraries that are managed 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 case for the development of .NET solutions in the Microsoft world for about 20 years.

It allows the sharing of source code between the project participants in a particularly simple way and also prevents version conflicts by having only one central `node_modules` folder with dependencies. This 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 just use the following command:

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

This downloads a script which created 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[19], Nrwl[20] – the company behind Nx – use the following categories for libraries:

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

- **feature**: Implements a use case using 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 here. Smart components within feature libraries are use case specific. An example is a component which allows to search for products.

On contrary, dumb components don't know the current use case at all. They just receive data via inputs, display it in a specific way and emit events. Such presentational components "just" help to implement use cases and hence they can be reused across them. A example is a date time picker, which does not know at all which use case it supports. Hence, it can be used within all use cases dealing with dates.

In addition to these categories, I'm also using the following categories:

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

To keep the overview, the categories are used as a prefix for the individual library folders. Thus, libraries of 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. On the other hand, 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 is a vital aspect of good software design as it allows to split them into a public and a private part. The public one is exposed for other libraries. Hence, here we have to take care about breaking changes as they would affect other parts of the system.

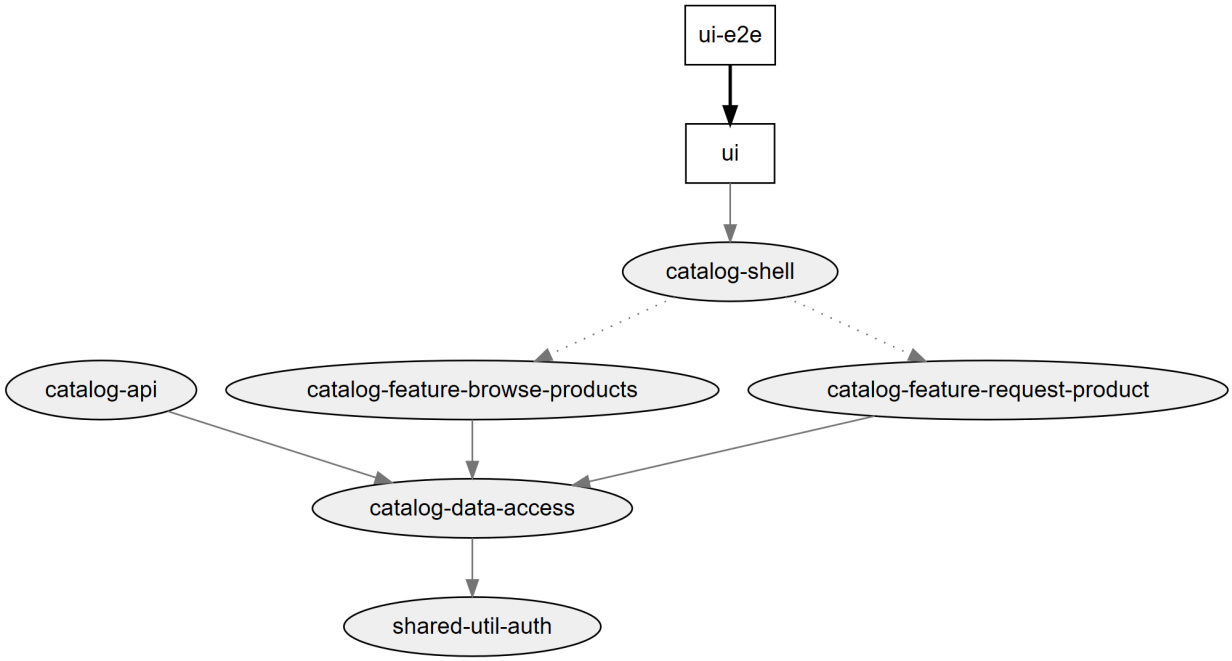
However, the private part can be changed quite flexible as long as you make sure the public part stays the same.

Check Accesses between libraries

To improve maintainability, it is important to minimize the dependencies between the individual libraries. The achievement of this goal can be checked graphically with Nx. For this, it provides the `dep-graph` npm script:

```
npm run dep-graph
```

If we just concentrate on the `Catalog` domain in our case study, the result looks as follows:



Access Restrictions for a Solid Architecture

To provide a solid architecture, it's vital to ensure that not every library can access every other library. If this was the case, we had a heap of intermingled libraries where each change would affect all the other ones. Obviously, this affects maintainability in a bad way!

In alignment with ideas of DDD, a few rules are used for the communication between libraries and these lead to a 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 categorization of libraries also has limitations: a `shell` only accesses `features` and a `feature` accesses `data-access` libraries. In addition, anyone can access `utils`.

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

Tagging Libraries

The tags for our libraries are defined in the file `nx.json`, which is generated by Nx:

```
"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 also be specified when setting up the applications and libraries.

According to a suggestion from the mentioned e-book about Monorepo Patterns[21], the domains are named with the prefix `scope` and The library types are prefixed with `kind`. Prefixes of this type are only intended to increase readability and can be freely assigned.

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

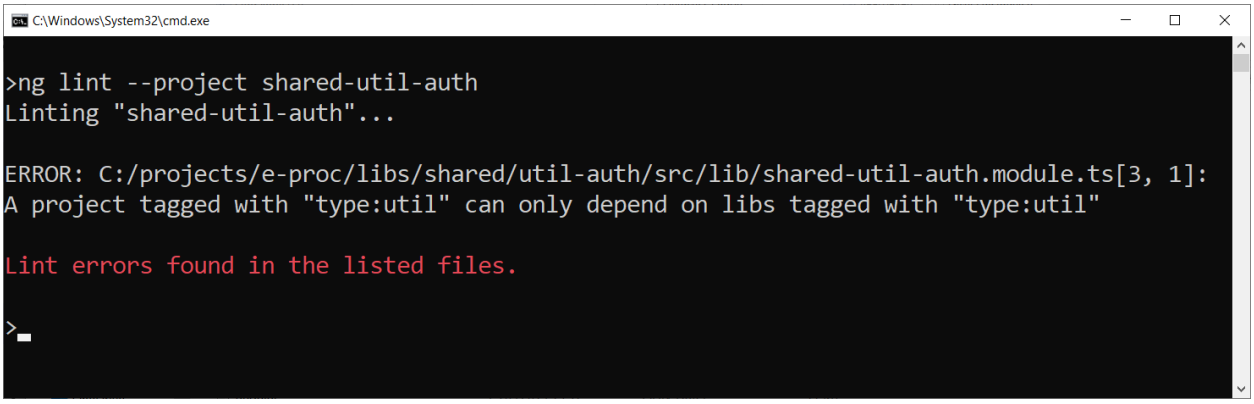
Defining Linting Rules based upon Tags

To enforce access restrictions, Nx comes with its own linting rules. As usual, they are configured 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 against 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, a corresponding plugin must be installed.

> Hint: Consider using Git Hooks, e. g. by leveraging Husky[22], which ensures that only code not violating your linting rules can be pushed to the repository.

Conclusion

Strategic Design provides a proven way to break an application into self-contained domains. These domains are characterized by their own specialized vocabulary, which must be used rigorously by all stakeholders.

The CLI extension Nx provides a very charming 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.

This helps to ensure a loosely coupled system which is easier to maintain as a sole change only affects a minimum of other parts of the system.

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

Tactical Domain-Driven Design with Angular und 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 about a travel web application which consists of the following sub-domains:

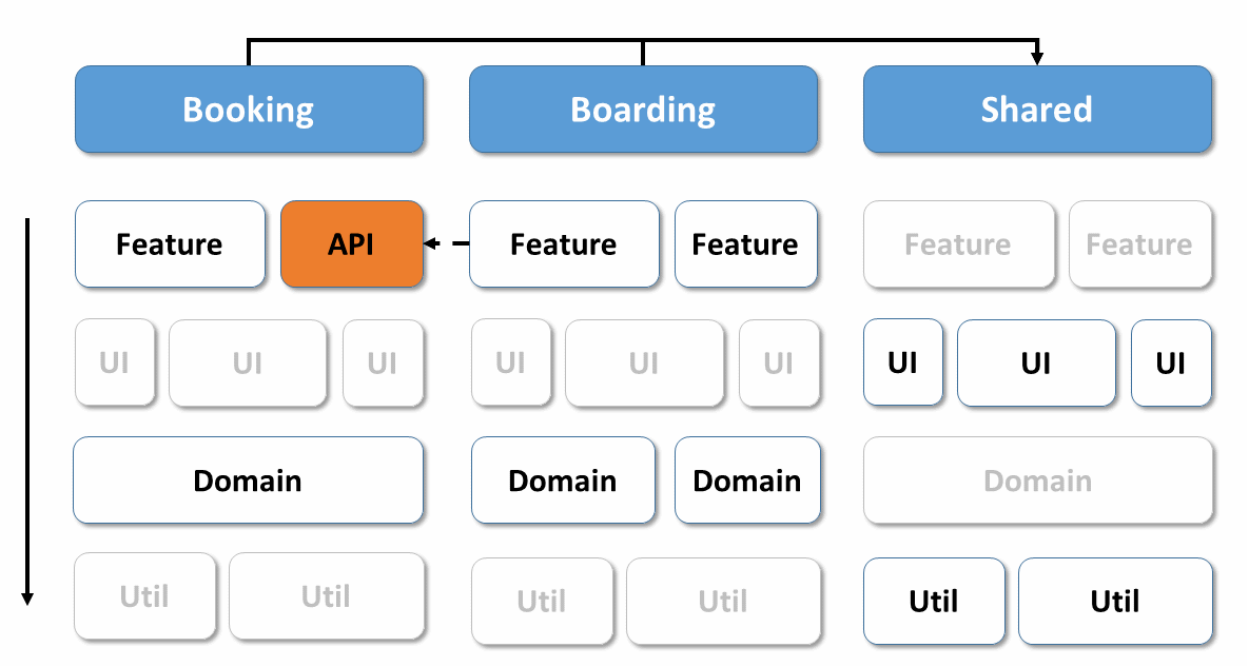
The source code[23] of this case study can be found here[24].

Now let's talk about how to structure our application with tactical design.

Domain Organization using Layers

For domains we use column subdivisions ("swimlanes"). Moreover, those domains can be organized with layers of libraries which leads to row subdivisions:

[23] https://github.com/manfredsteyer/angular-ddd
[24] https://github.com/manfredsteyer/angular-ddd



Note how each layer could consists one or more **libraries**

> While using layers is quite a traditional way of organizing an domain, there are also alternatives like hexagonal architectures or clean architecture.

What about shared functionality?

For those aspects that are to be *shared* and used across domains, an additional `shared` swimlane is used. Shared libraries can be quite 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 constrains 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 can result in loose coupling and thus increased maintainability.

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

Regarding the shared part, one can see the following two characteristics:

- As the grayed-out blocks indicate, most `util` libraries are in the `shared` area, especially as aspects such as authentication or logging are used 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 placed within its own domain.

While developers may elect to 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 Organization

Based on Nrwl.io's Enterprise MonoRepository Patterns[25], I distinguish between five categories of layers or libraries:

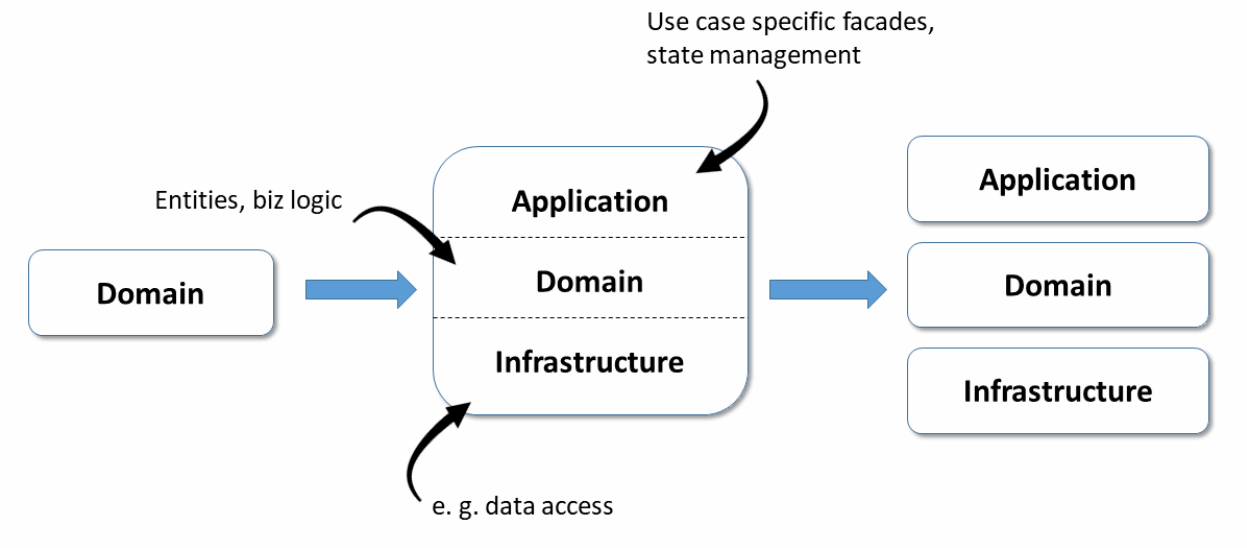
Category	Description	Exemplary content
feature	Contains components for an use case.	search-flight component
ui	Contains so-called "dumb components" that are use-case agnostic and thus reusable.	datetime-component, address-component, adress-pipe
api	Exports building blocks from the current subdomain for 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 developers I've worked with agreed that the code organization facilitates code reuse and future features.

Isolate the Domain

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

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



This facade can also deal with state mangement.

While Facades are currently quite popular in the Angular environment, this idea also correlates wonderfully with DDD (where they are called application services).

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

In an SPA, infrastructure concerns are – at least most of the time – 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, these layers can now also be packaged in their own libraries. For the sake of simplicity, it is also possible to store them in a single library, which is subdivided accordingly. This decision makes sense if these layers are usually used together and only need to be exchanged for unit tests.

Implementations in a Monorepos

Once the components of our architecture have been determined, the question arises of how they can be implemented in the world of Angular. A very common approach also used by Google itself is using a monorepo. It is a code repository that contains all the libraries of a software system.

While a project created with the Angular CLI can nowadays be used as a monorepo, the popular tool Nx[26] offers some additional possibilities which are especially valuable for large enterprise solutions.

[26] https://nx.dev/

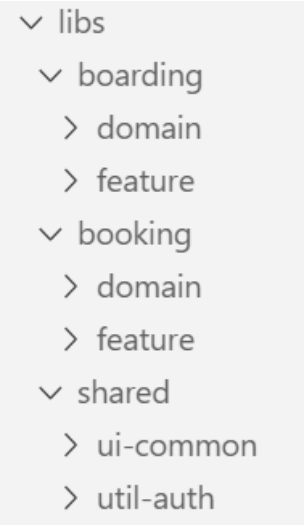
These include the previously discussed ways to introduce access restrictions between libraries[27]. This prevents each library from accessing each other, resulting in a highly coupled overall system.

To create a library in a monorepo, one instruction is enough:

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

> As you just use `ng generate library` instead of `ng generate module` there is not more effort for you. However, you get a cleaner structure, improved maintainability, and less coupling.

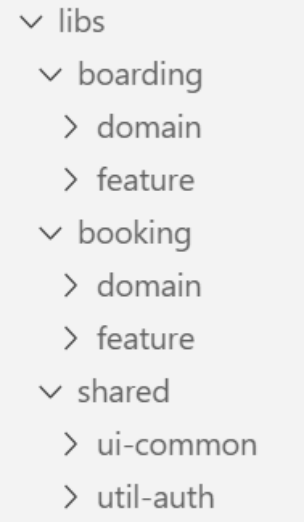
The switch `directory` provided by Nx specifies an optional subdirectory where the libraries are to be put. This way, they can be **grouped by domain**:



The names of the libraries also reflect the layers. If a layer has several libraries, it makes sense to use these names as a prefix. This results in names such as `feature-search` or `feature-edit`.

In order to isolate the actual domain model, the example shown here divides the domain library into the three further layers mentioned:

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



Builds within a Monorepo

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

This change information is used to recompile only the **affected** libraries or just run their **affected** tests. Obviously, this saves a lot of time on large systems that are stored in a repository as a whole.

Entities and your Tactical Design

Tactical Design provides many ideas for structuring the domain layer. At the center 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 that their state remains consistent. You implement this with private fields and public methods that operate on them.

These entities not only encapsulate data, but also business rules. At least the method `setStatus` indicates this circumstance. Only for cases where business rules can not be meaningfully accommodated in an entity, DDD defines so-called domain services.

> Entities that only represent data structures are frowned upon in DDD. The community calls them devaluing bloodless (anemic)[28].

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 important.

Typescript is a multi-paradigm language in which Functional Programming plays a big role. Books dealing with Functional DDD can be found here:

- Domain Modeling Made Funcitonal[29],
- Functional and Reactive Domain Modeling[30].

[28] https://martinfowler.com/bliki/AnemicDomainModel.html
[29] https://pragprog.com/book/swdddf/domain-modeling-made-functional
[30] https://www.amzn.com/1617292249

With functional programming, the previously considered entity model would therefore be separated into a data part and a logic part. Domain-Driven Design Distilled[31] which is one of the standard works for DDD and primarily relies on OOP, also admits 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

}
```

Here the entities also use public properties. This practice is quite common in FP; the excessive use of getters and setters, which only delegate to private properties, is often ridiculed!

Much more interesting, however, is the question of how the functional world avoids inconsistent states. The answer is amazingly simple: data structures are preferably **immutable**. The keyword `readonly` in the example shown emphasizes this.

A part of the program that wants to change such objects has to clone it, and if other parts of the program have first validated an object for their own purposes, they can assume that it remains valid.

> A wonderful side-affect of using immutable data structures is that change detection performance is optimized. No longer are *deep-comparisons* required. Instead, a *changed* object is actually a new instance and thus the object references will no longer be the same.

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

Tactial DDD with Aggregates

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

The state of all components of an aggregate must be consistent as a whole. For example, in the example outlined above, one could specify that `completed` in `BoardingList` may only be set to `true` if no `BoardingListEntry` has the status `WAIT_FOR_BOARDING` .

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

> Domain-Driven Design Distilled[32] 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[33] (aka Applications Services) are used to represent the domain logic in an use case specific way. The provide several advantages:

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

Independent of DDD, this idea has been very popular in the world of Angular for some time.

For our example, we could create the following Facade:

```
1  @Injectable ({providedIn: 'root'})
2  export class FlightFacade {
3
4      private notifier = new BehaviorSubject<Flight[]>([]);
5      public  flights$ = this.notifier.asObservable();
6
7      constructor(private flightService: FlightService) { }
8
9      search(from: string, to: string, urgent: boolean): void {
10         this.flightService
```

[32] https://www.amzn.com/0134434420
[33] 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. This means that the facade can auto-deliver updated flight information when conditions change. Another advantage of Facades is the ability to transparently introduce Redux and `@ngrx/store` later when needed. This can be done without affecting any of the external application components.

> For the consumer of the Facade it is not relevant whether it manages the state by it self or by delegating to a state management library.

Stateless Facades

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

A web SPA has state and that's what makes it user-friendly!

To avoid UX issues, Angular applications do not want to reload all the information from the server over and over again. Hence, the Facade outlined holds the loaded flights (within the observable discussed before).

Domain Events

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

This also perfectly fits to 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 to it.

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

> For Angular developers familiar with Redux or Ngrx: Domain events can be represented as *dispatched actions*!

Conclusion

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

Due to the object-functional nature of TypeScript and prevailing customs, a few rule changes are necessary. 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 building the basic structure.
- Access restrictions between libraries prevent coupling between domains.
- Facades prepare the domain model for individual use cases and take care of maintaining the state.
- If needed, Redux can be used behind the facade without the rest of the application noticing.

If you want to see all these topics in action, checkout our Angular Architecture workshop[34].

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

From Domains to Micro Frontends

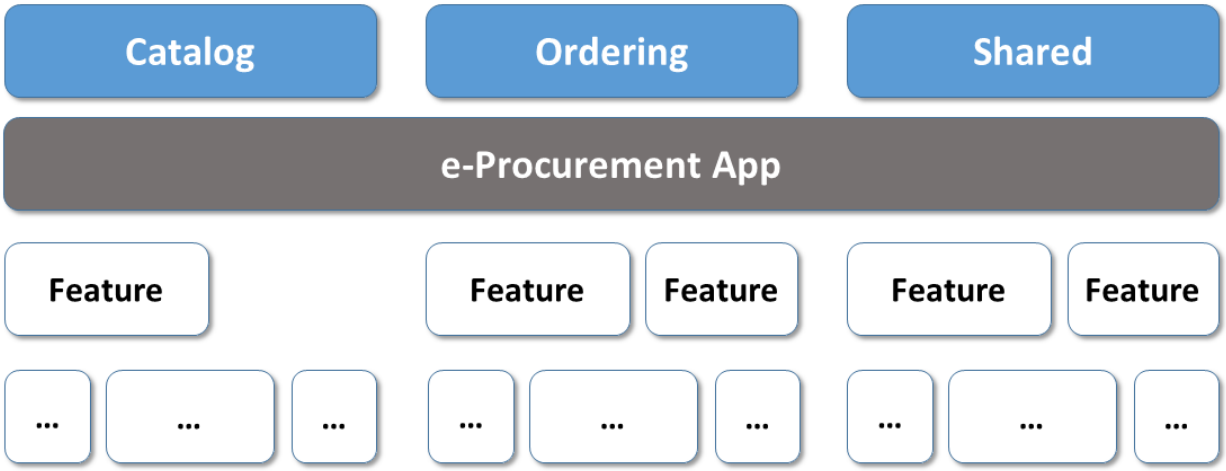
Let's assume, you've identified the sub-domains for your system. The next question is, how to implement them.

One option is to implement them within a big application – aka a deployment monolith – and the second one is about providing a separate application for each domain.

Such applications are nowadays called micro frontends.

Deployment Monoliths

A deployment monolith is a big integrated solution containing different domains:



This approach supports a consistent UI and leads to optimized bundles as everything is compiled together.

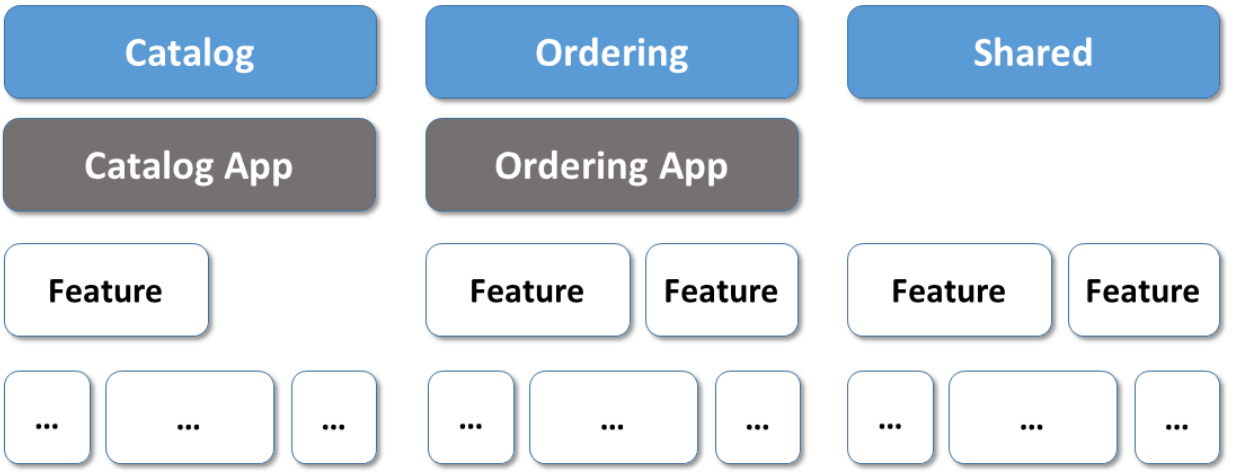
On the other side, a team responsible for one sub-domain needs to coordinate with other teams responsible for other sub-domains. They have to agree on an overall architecture, the leading framework, and an update policy for dependencies. Interestingly, you might also see this as an advantage.

Also, it is tempting to just reuse parts of other domains which may lead to higher coupling and – sooner or later – to breaking changes. To prevent this situation, you can go with free tools like Nrwl's Nx[35]. For instance, Nx allows to define access restrictions between the parts of your mono repo to enforce your envisioned architecture and loosely coupling.

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

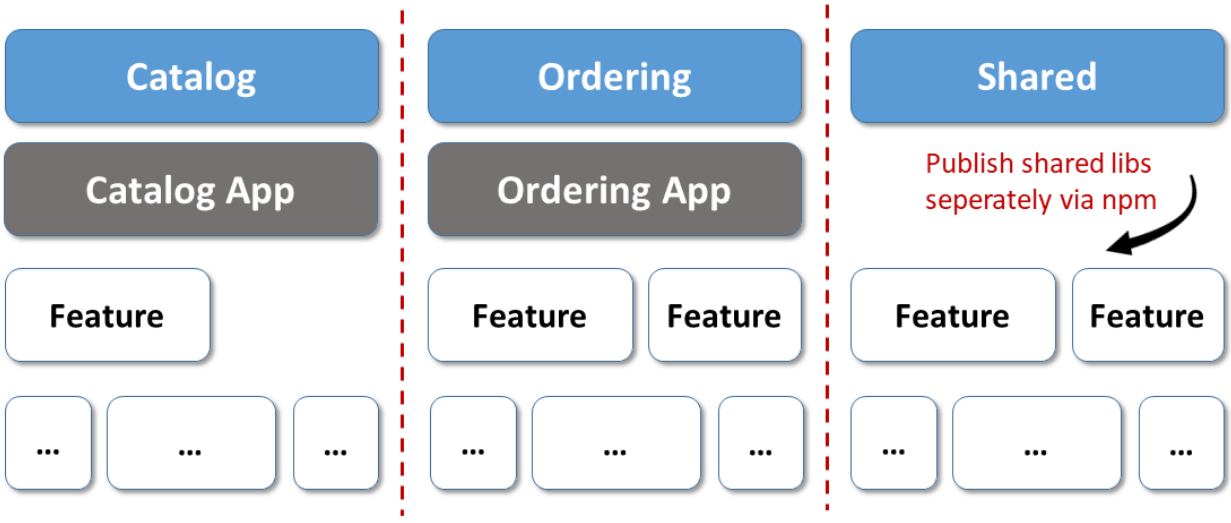
Deployment monoliths, micro frontends or something in between?

To further decouple the parts of your system, you could decide to split it into several smaller applications. If we assume that use cases are not overlapping the boundaries of your sub-domains, this can lead to more autarkic teams and applications which can be deployed separately.



Also, as we are having several tiny systems now, it decreases the complexity.

If you seek even more isolation between your sub-domains and teams responsible for them, you could decide to put each sub-domain into a (mono) repository of its own:



Now, you have something people are calling micro frontends nowadays. This allows the individual teams to be as autarkik as possible: They can choose for their own architectural style, their own technology stack and they can even decide by themselves when to update to newer versions of the

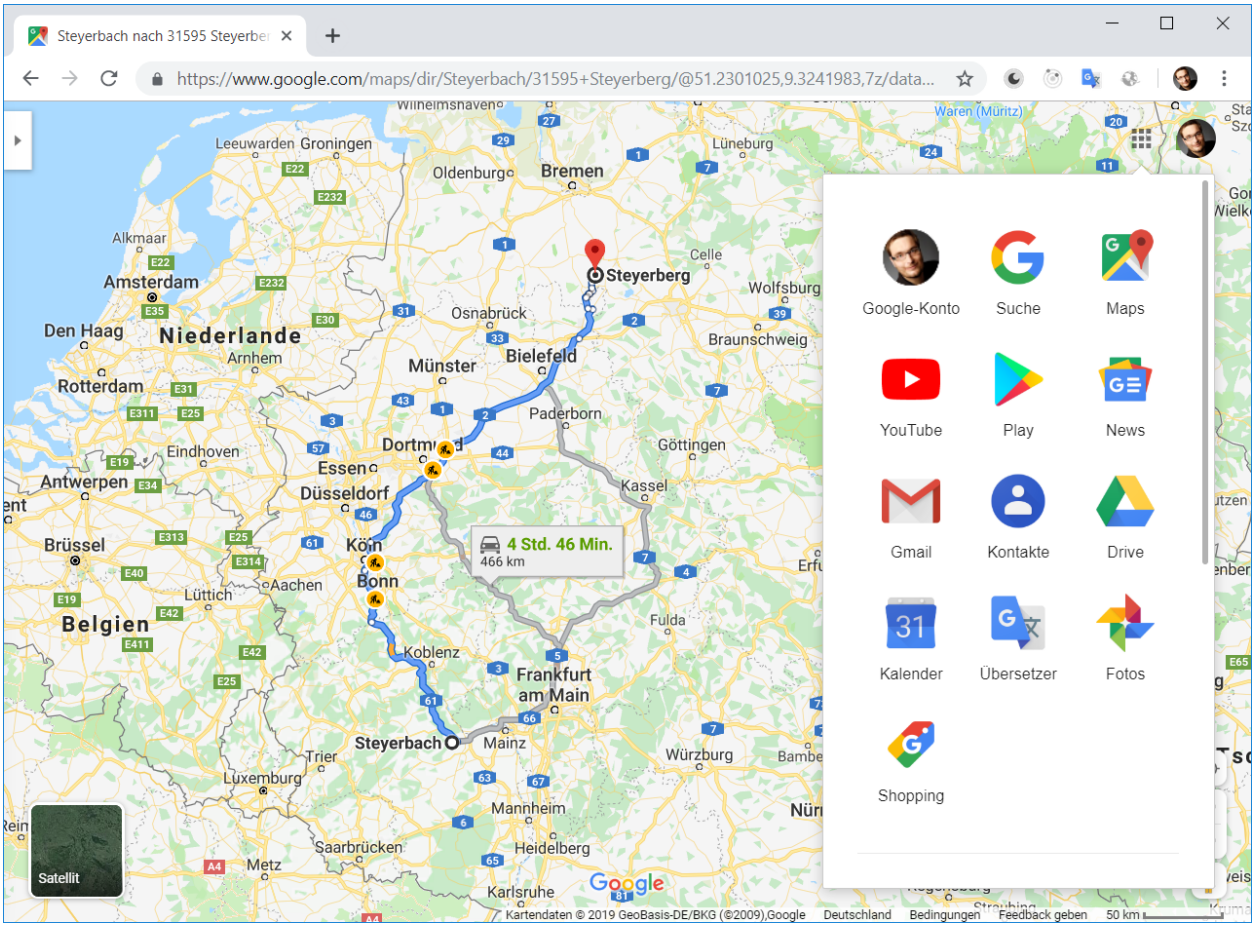
frameworks used. This also means they can use “the best technology” for the requirements given within the current sub-domain.

To possibility to use their own framework and architecture comes in handy for applications which are developed in long term. If, for instance, a new framework shows up in five years, we can just use it for implementing the next domain.

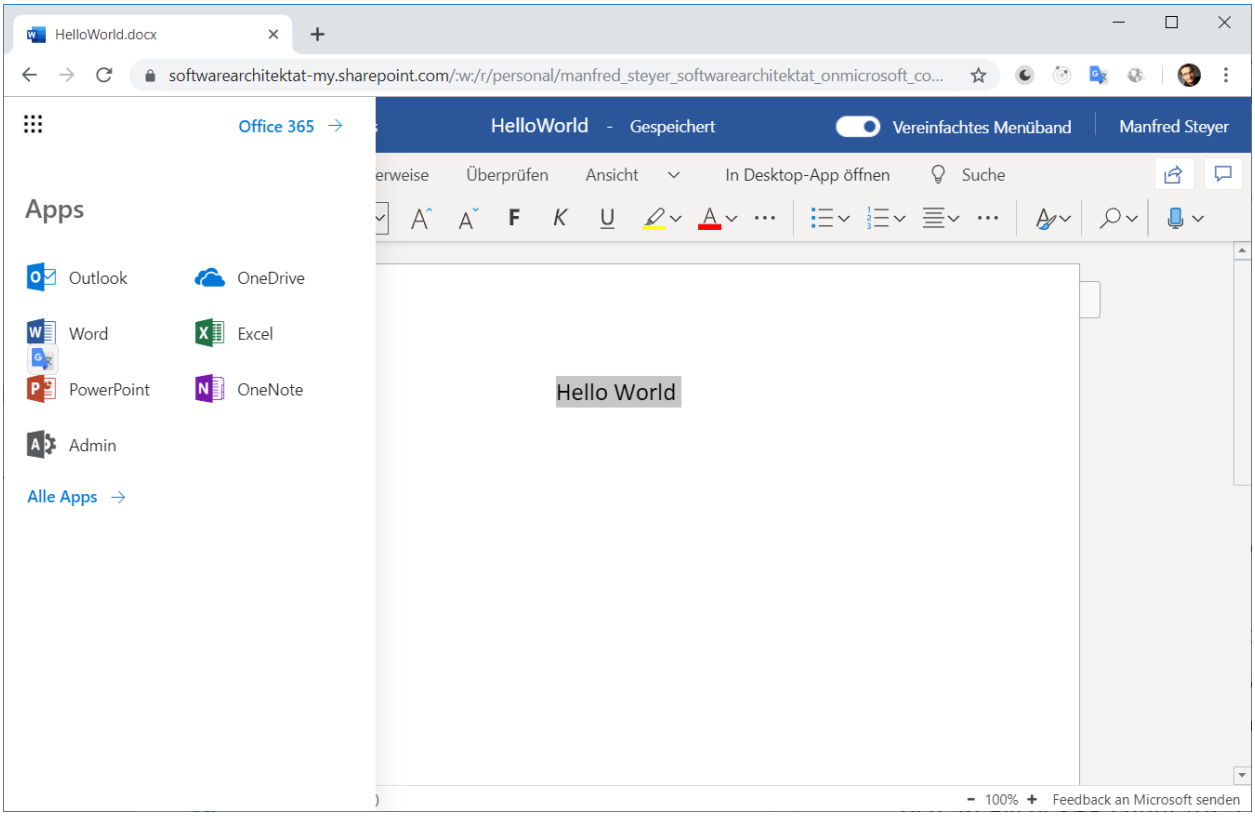
However, this does not come without costs: Now you have to deal with shipping your shared libraries via npm and this includes versioning which can lead to version conflicts.

UI Composition with Hyperlinks

Also, you have to find ways to integrate the different applications to one big system which is presented to your users. One easy way to accomplish this is using hyperlinks:



This approach seems to fit quite well for product suites like Google or Office 365:



Here, each domain is a self contained application. This worked quite well, because we don't need much interactions between the domains. If we needed to share data, we can use the backend. Using this strategy, Word 365 can use an Excel 365 sheet for a series letter.

This approach has several advantages:

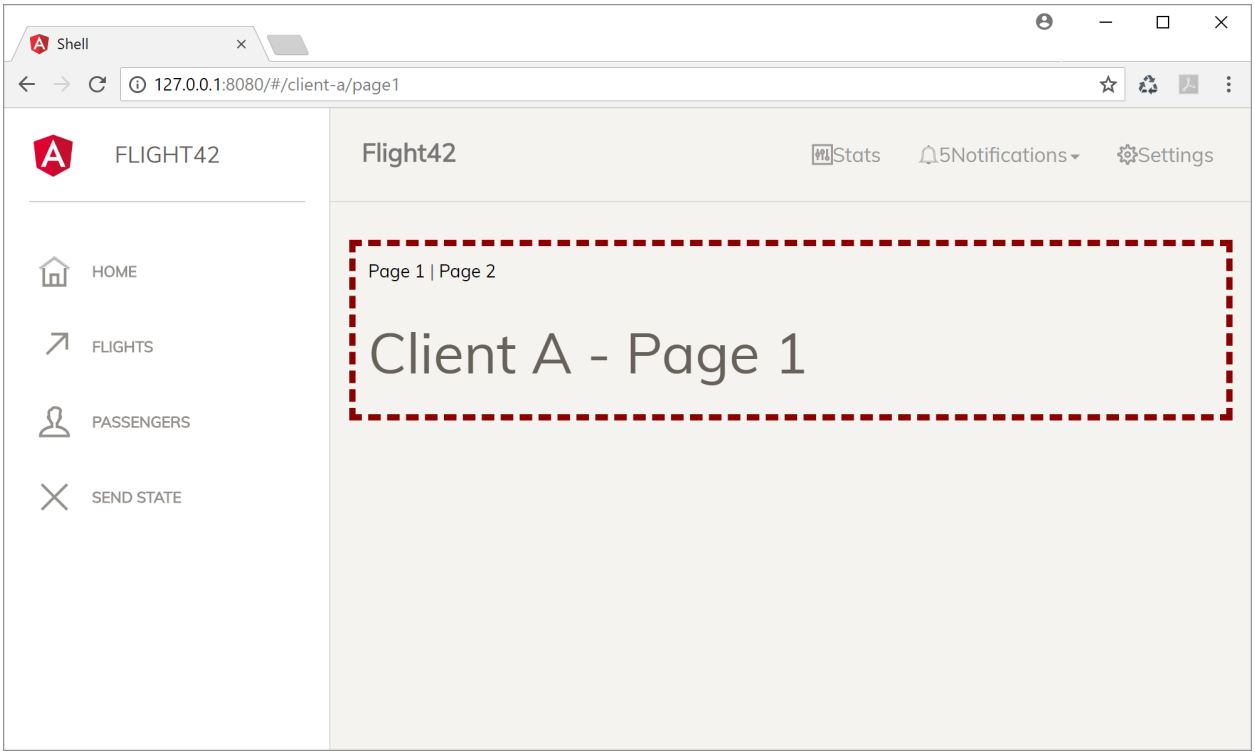
- It is easy
- It uses SPA frameworks as intended
- We get optimized bundles per domain

However, there are also some disadvantages:

- We loose our state when switching to another application
- We have to load another application – this is what we wanted to prevent with SPAs
- We have to make sure to get a common look and feel (we need a common design system).

UI Composition with a Shell

Another approach discussed a lot nowadays is providing a shell that loads different single page applications on demand:



In the shown screenshot, the shell loads the micro frontend with the red border into its working area. Technically, it just loads the micro frontends bundles on demand. After this, the shell creates an element for the micro frontend's root element:

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

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

Instead of bootstrapping several SPAs we could also use iframes. While we all know the huge disadvantages of iframes as well as strategies to deal with most of them, they provide two useful features here:

1) Isolation: An micro frontend in one iframe cannot influence or hack another microfrontend in another iframe. Hence, they come inhandy for plugin systems or when integrating applications from other vendors. 2) They also allow to integrate legacy systems.

A library that compensates most of the disadvantages of iframes for intranet applications can be found here[36].

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

The shell approach comes with the following advantages:

- The user gets an integrated solution which consists of different micro frontends.
- We don't loose state when navigating between domains.

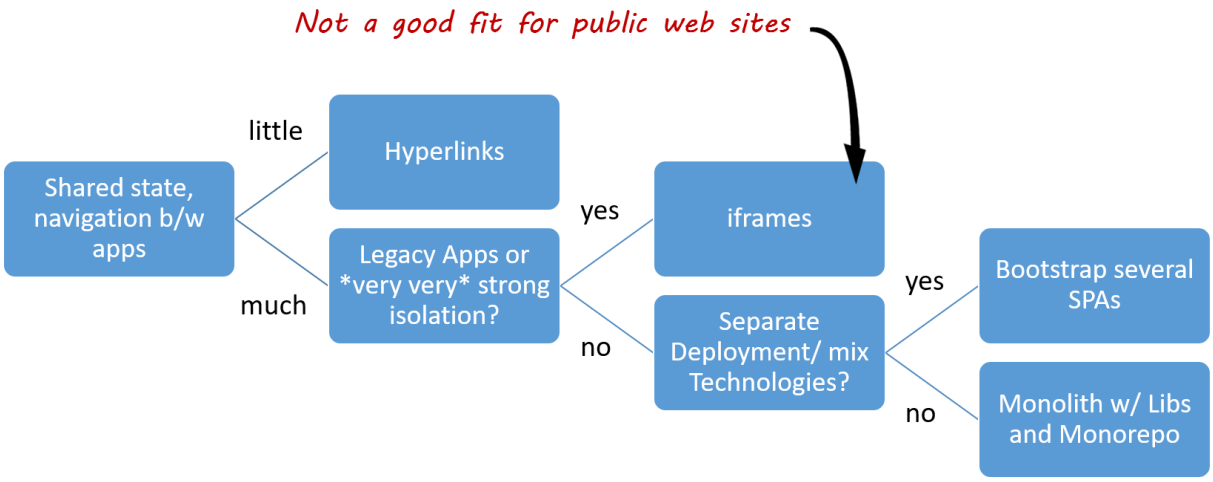
The disadvantages are:

- If we don't use specific tricks (outlined in the next chapter), each micro frontend comes with its own copy of Angular and the other frameworks. This increases the bundle sizes.
- We have to implement some infrastructure code for loading micro frontends and switching between them.
- Also here, we have to make sure to get a common look and feel (we need a common design system).

Finding a Solution

Deciding between a deployment monolith and different approaches for micro frontends is quite difficult because each option comes with its very own advantages and disadvantages.

To provide some guidance 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

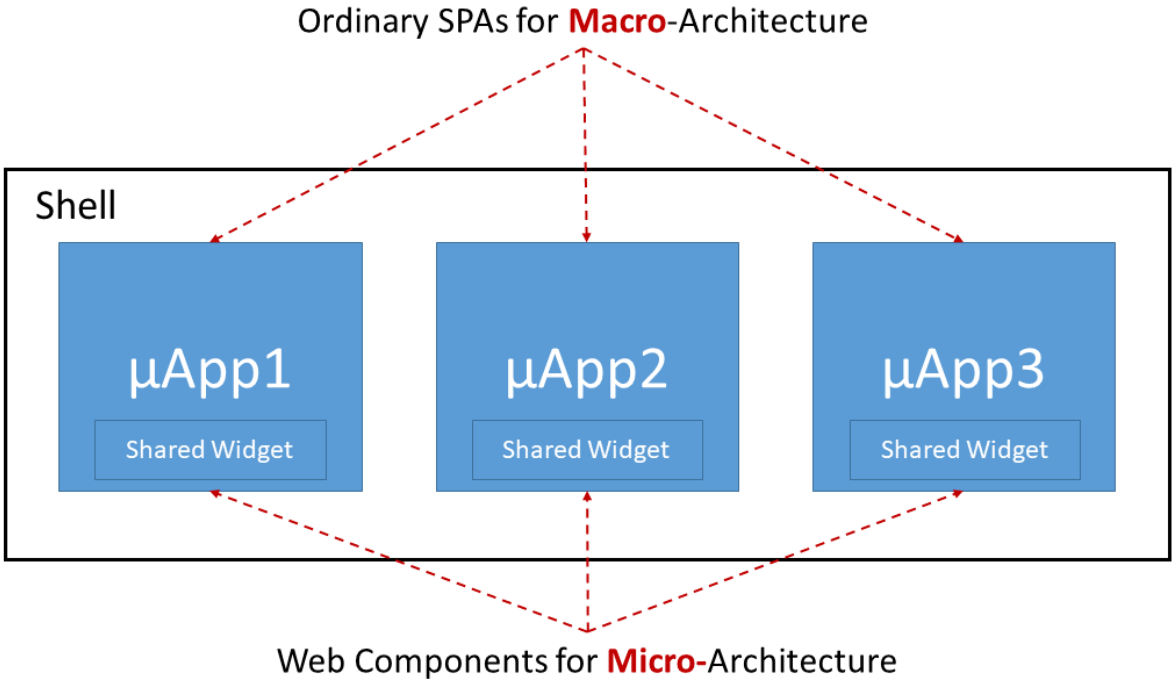
There are several ways for implementing micro frontends and all of them come with their very own advantages and disadvantages. Also, using a consistent and optimized deployment monolith can also be the right choice.

At the end of the day, it's about knowing your architectural goals and about evaluating them against the positive and negative consequences of architectural candidates available for your needs.

6 Steps to your Angular-based Micro Frontend Shell

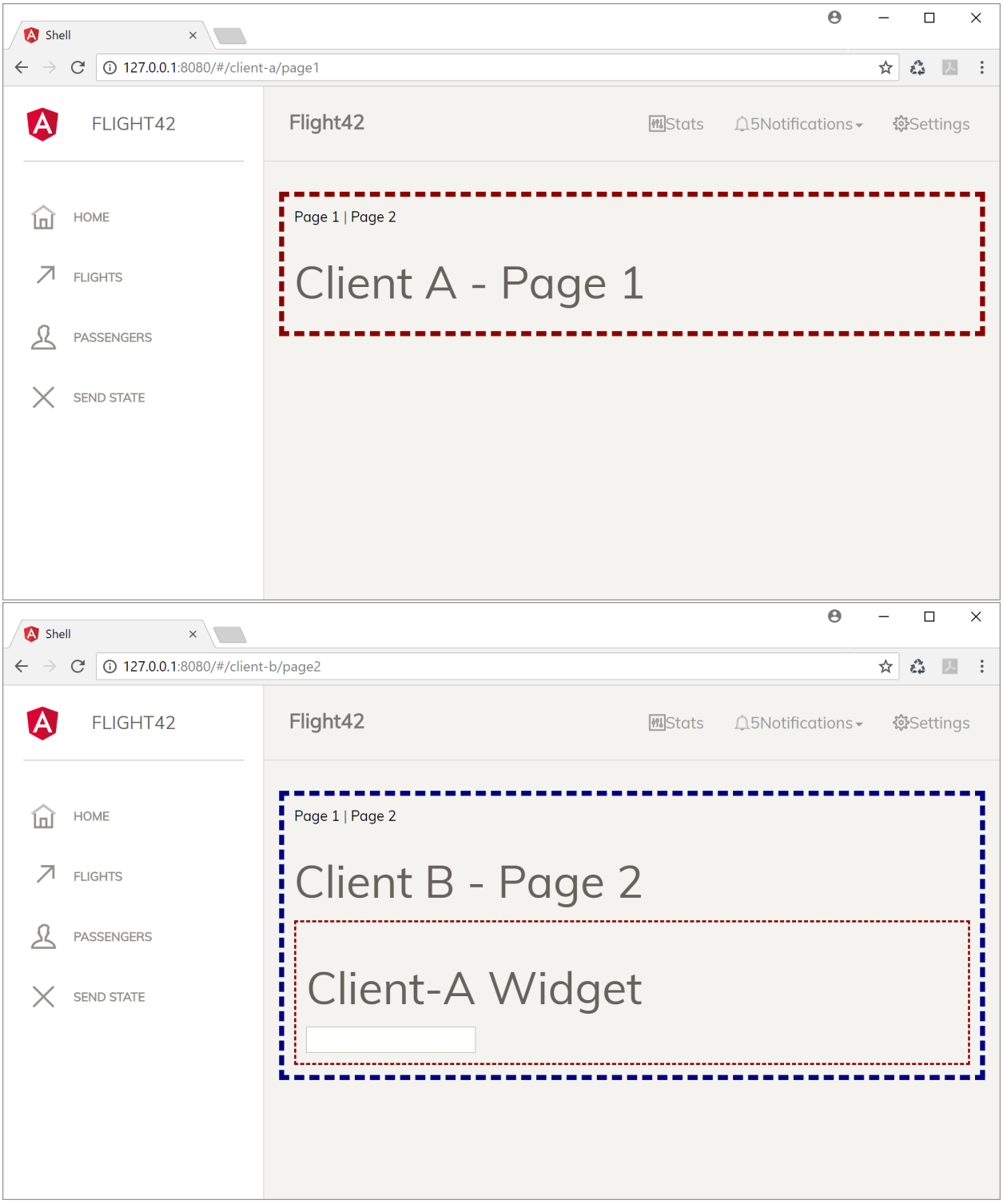
As discussed in the last chapter, there are several approaches for implementing SPA-based micro frontends.

In this chapter, I show how to implement one of them in 6 steps: A shell loading micro frontends on demand. Other than in my chapter about micro frontends and web components, I don't use web components for the macro architecture. Instead, I just go with ordinary SPAs which are loaded and bootstrapped on demand. For the micro architecture I still use web components.



While this decision simplifies the implementation, we can still isolate different applications using shadow DOM as Angular also supports this standard associated with web components for traditional Angular components since its first day.

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



The source code[37] for this can be found in my GitHub account here[38].

[37] https://github.com/manfredsteyer/angular-microfrontend
[38] https://github.com/manfredsteyer/angular-microfrontend

Step 0: Make sure you need it

Make sure this approach fits to your architectural goals. As discussed in the previous chapter, micro frontends come with a lot of consequences. Make sure you are aware of them.

Step 1: Implement Your SPAs

Implement your micro frontends as ordinary Angular applications. In an micro service architecture it's quite common that every part gets its own repository in order to decouple them as much as possible (see Componentization via Services[39]) On contrary, I've seen a lot of micro frontends based upon monorepos for practical reasons.

Of course, now, we could discuss when the term micro frontend is appropriate. I won't, because this discussion doesn't really help. What counts is to find a architecture that fits your goals and to be aware of its consequences.

If we go with a monorepo, we have to ensure, e. g. with linting rules, that the micro frontends are not coupled to each other. As discussed in a previous chapter, Nrwl's Nx[40] provides a great solution for that: It allows to set up access restrictions defining which library can access which other one. Also, Nx can detect which parts of your monorepo are affected by a change in order to only recompile and retest them.

To make routing across micro frontends 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 {
  [...]
}
```

[39] https://martinfowler.com/chapters/microservices.html#ComponentizationViaServices
[40] https://nx.dev/angular

Step 2: Expose Shared Widgets

Expose widgets you want to share as Web Components/ Custom Elements. Please note that from the perspective of micro services, you should avoid sharing code between micro frontends as much as possible. The reason is that this causes coupling which is exactly what you want to avoid with this architectural style.

For exposing an Angular Component as a Custom Element, you can use Angular Elements. My blog chapter about Angular Elements[41] and lazy and external Angular Elements[42] provides the necessary information.

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 are going to load several SPAs into one page, they will compete over this array, mess it up, and stop working.

For this dilemma, we have two solutions:

1. Just put everything into one bundle, so that this global array is not needed
2. Rename the global array

Here, I use solution 1) because a micro frontend is by definition small and just having one bundle makes loading it on demand easier. Also, 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'm using my free tool ngx-build-plus[43] 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 straight forward. All you need is some vanilla JavaScript code dynamically creating a `script` tag and the tag for application's root element:

[41] https://www.softwarearchitekt.at/aktuelles/angular-elements-part-i/
[42] https://www.softwarearchitekt.at/aktuelles/angular-elements-part-ii/
[43] https://www.npmjs.com/package/ngx-build-plus

```
// 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.

Also, you need some code to show and hide the loaded micro frontend on demand:

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

Step 5: Communication Between Microfrontends

In general, we should keep the communication between microfrontends at a minimum, as it couples them to each other.

To implement communication, we have several options. Here, I go with the least obtrusive one: using the query string. This has several advantages:

1. It does not matter in which order the micro frontends are loaded. When they are loaded, they can grab the current parameters from the url
2. It allows deep linking
3. It's like the web is supposed to work
4. It's easy to implement

Setting an url parameter with the Angular router is just a matter of calling one method:

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

The option `merge` makes sure that the existing url parameters are not lost. If there is already a parameter `id`, the router will overwrite it.

Also, listening for changes within url parameters is also something the Angular router can help with:

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

There are some alternatives for this:

1. If you wrap your micro frontends into web components, you could use their properties and events to communicate with the shell.
2. The shell could put a "message bus" into the global namespace: `typescript (window as any).messageBus = new BehaviorSubject<MicroFrontendEvent>(null);`

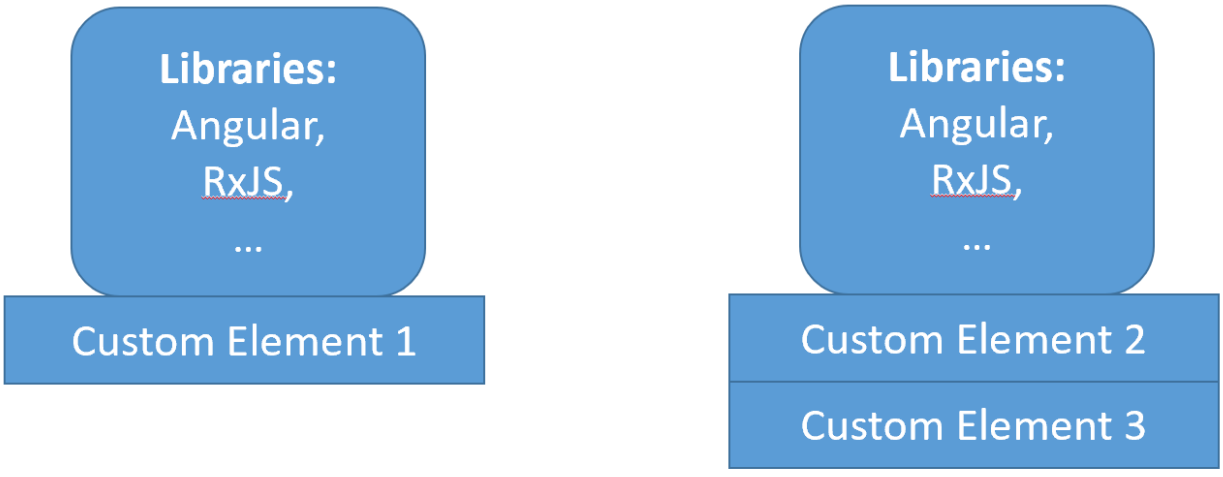
   Both, the shell and the Microfrontends could now subscribe to this message bus and listen for specific events they are interested into. Also, both can emit events.
3. Using custom Events provided by the browser: "'typescript // Sender const customer = { id: 17, ... }; window.raiseEvent(new CustomEvent('CustomerSelected', {details: customer}))

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

Step 6: Sharing Libraries Between Micro Frontends

Now, as we have several self-contained micro frontends, each of them has its own dependencies, e. g. Angular itself or RxJS. From an micro service perspective, this is perfect because it allows each team behind your micro frontends to choose any library or framework in any version. They can even decide by their own if and when to update to newer versions.

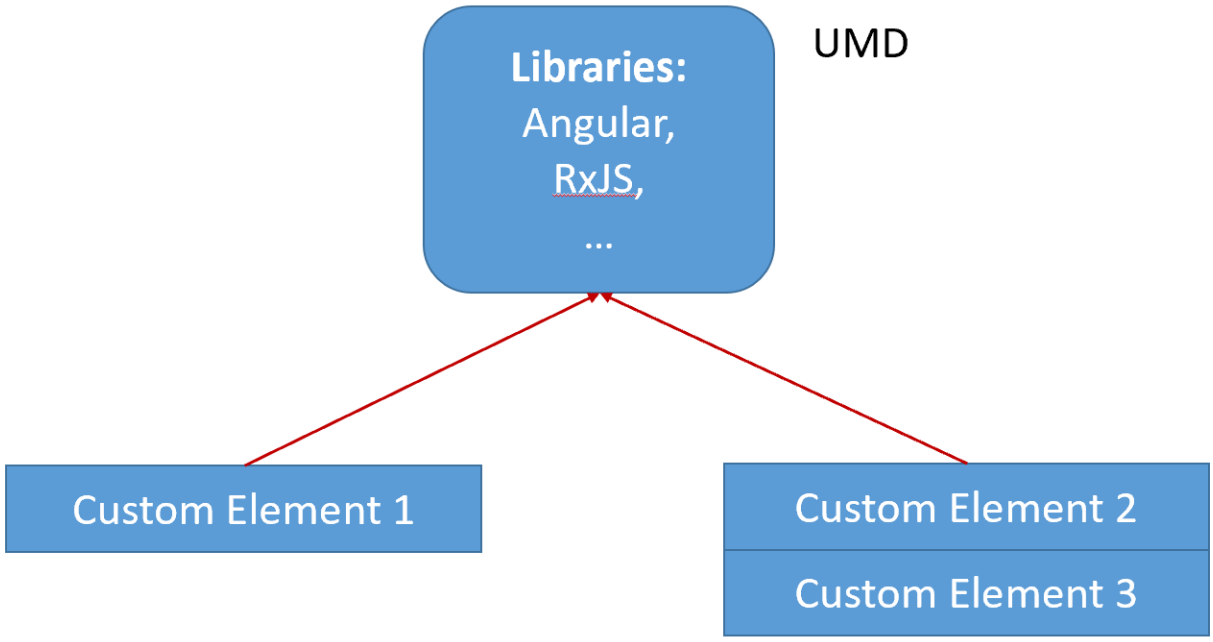
However, from the perspective of performance and loading time, this is a bad situation because it leads to code duplication within the bundles. For instance, we could end up with having a specific angular version in several of your bundles:



Fortunately, there is a solution for this: Webpack externals.

Externals allow us to share common libraries. For this, they are loaded into the global namespace. This was quite usual in the days of jQuery (which provided a global $ object) and is still sometimes done for simple react and vue applications nowadays.

In the case of most libraries we can use UMD bundles which do exactly this besides other things. Then, we have to tell webpack to not bundle them together with every micro frontend but to reference it within the global namespace instead:



To use webpack externals together with the Angular CLI you can leverage ngx-build-plus[44] which even comes with a schematic introducing the needed changes into your application.

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

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

Then, call the following schematic:

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

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

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

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

This 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:

```
<!-- 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 optimize this, one could also put those parts of Angular into one bundle.

Also, if you have a 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, for instance, makes the produced bundle to reference the global variable `ng.core` when it needs `@angular/core`. Hence, `@angular/core` does not need to be part of the bundle anymore.

Please note that this is not the default operation mode for Angular and hence it comes with some risks.

Conclusion

With the right wrinkles, implementing a shell for micro elements is not that difficult. However, this is only one way for implementing micro frontends and – as all – it comes with its very own advantages and disadvantages. Hence, before implementing it, make sure it fits your architectural goals.

Literature

- Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software[45]
- Wlaschin, Domain Modeling Made Functional[46]
- Ghosh, Functional and Reactive Domain Modeling[47]
- Nrwl, Monorepo-style Angular development[48]
- Jackson, Micro Frontends[49]
- Burleson, Push-based Architectures using RxJS + Facades[50]
- Burleson, NgRx + Facades: Better State Management[51]
- Steyer, Web Components with Angular Elements (article series, 5 parts)[52]

[45] https://www.amazon.com/dp/0321125215
[46] https://pragprog.com/book/swdddf/domain-modeling-made-functional
[47] https://www.amazon.com/dp/1617292249
[48] https://go.nrwl.io/angular-enterprise-monorepo-patterns-new-book
[49] https://martinfowler.com/articles/micro-frontends.html
[50] https://medium.com/@thomasburlesonIA/push-based-architectures-with-rxjs-81b327d7c32d
[51] https://medium.com/@thomasburlesonIA/ngrx-facades-better-state-management-82a04b9a1e39
[52] 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 on-site workshops and consultancy.

Please find our offers here[53].

[53] https://www.softwarearchitekt.at/angular-schulung/