

Enterprise Angular: Standalone Components and APIs



Mental Model & Compatibility

Routing, Lazy Loading, and NGRX

MANFRED STEYER

Lightweight Angular with Standalone Components

Manfred Steyer

This book is for sale at <http://leanpub.com/standalone-components>

This version was published on 2022-10-17



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

© 2022 Manfred Steyer

Contents

Intro	1
Trainings and Consultancy	1
Mental Model & Compatibility	2
Why Did we Even Get NgModules in the First Place?	2
Getting Started With Standalone Components	3
The Mental Model	4
Pipes, Directives, and Services	4
Bootstrapping Standalone Components	5
Compatibility With Existing Code	6
Side Note: The CommonModule	8
Interim Conclusion: Standalone Components – and now?	9
Architecture with Standalone Components	10
Grouping Building Blocks	10
Importing Whole Barrels	12
Barrels with Pretty Names: Path Mappings	13
The Next Logical Step: Workspace Libraries and Nx	14
More on Architectures with Nx	17
Conclusion	17
Routing and Lazy Loading	18
Providing the Routing Configuration	18
Using Router Directives	19
Lazy Loading with Standalone Components	20
Environment Injectors: Services for Specific Routes	22
Setting up NGRX and Feature Slices	23
Setting up Your Environment: ENVIRONMENT_INITIALIZER	25
Angular Elements with Standalone Components	26
Providing a Standalone Component	26
Installing Angular Elements	27
Bootstrapping with Angular Elements	27
Side Note: Bootstrapping Multiple Components	28
Calling an Angular Element	29

CONTENTS

Calling a Web Component in an Angular Component	30
Bonus: Compiling Self-contained Bundle	31
Migrating for Angular Standalone Components	33
Option 1: Ostrich Strategy	33
Option 2: Just Throw Away Angular Modules	33
Option 3: Replace Angular Modules with Barrels	35
Option 4: Nx Workspace with Libraries and Linting Rules	37
More on Architectures with Nx	41
Trainings and Consulting	42

Intro

Doubtless, NgModules have always been one of the most controversial building blocks in Angular. While they allow to group elements that go together and hence provide a structure for your application, they also cause some redundancy as EcmaScript already comes with an own module system.

Beginning with Angular 14, NgModules are optional. The new Standalone Components are self-contained and hence make Angular development more lightweight. While the theory behind Standalone Components seems to be easy at first glance, it brings several questions: How to deal with routing and lazy loading? How to structure an application without NgModules? How to use existing NgModules-based code?

In this small book, I try to answer all these questions. For this, I'm using an example that is as small as possible but also as big as needed to show the underlying ideas and concepts. At the end, I also provide some thoughts about migrating existing applications to Standalone Components.

Trainings and Consultancy

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

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

Please find the full list of our offers here¹.

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

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

¹<https://www.angulararchitects.io/en/angular-workshops/>

Mental Model & Compatibility

Standalone Components is one of the most exciting new Angular features since quite a time. They allow for working without NgModules and hence are the key for more lightweight and straightforward Angular solutions. A first implementation already landed in Angular 14 BETA and the Angular Team tries hard to make them available until version 14 is released in the first half of 2022.

In this book, I'm going to demonstrate how to leverage this innovation. For this, I'm using an example application completely written with Standalone Components.

The source code for this can be found in the form of a traditional [Angular CLI workspace](#)² and as an [Nx workspace](#)³ that uses libraries as a replacement for NgModules.

Why Did we Even Get NgModules in the First Place?

The main reason for initially introducing NgModules was pragmatic: We needed a way to group building blocks that are used together. Not only to increase the convenience for developers, but also for the Angular Compiler whose development lagged a little behind. In the latter case, we are talking about the compilation context. From this context, the compiler learns where the program code is allowed to call which components:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';
[...]
@NgModule({
  imports: [BrowserModule, OtherModule],
  declarations: [AppComponent, OtherComponent, OtherDirective],
  providers: [],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

Compilation Context

NgModules provide the Compilation Context

²<https://github.com/manfredsteyer/standalone-example-cli>

³<https://github.com/manfredsteyer/standalone-example-nx>

However, the community was never really happy with this decision. Having another modular system besides that of EcmaScript didn't feel right. In addition, it raised the entry barrier for new Angular developers. That is why the Angular team designed the new Ivy compiler so that the compiled application works without modules at runtime. Each component compiled with Ivy has its own compilation context. Even if that sounds grandiose, this context is just represented by two arrays that refer to adjacent components, directives, and pipes.

Since the old compiler and the associated execution environment have now been permanently removed from Angular as of Angular 13, it was time to anchor this option in Angular's public API. For some time there has been a design document and an associated RFC [RFC]. Both describe a world where Angular modules are optional. The word optional is important here: Existing code that relies on modules is still supported.

Getting Started With Standalone Components

In general, implementing a Standalone Component is easy. Just set the `standalone` flag in the `Component` decorator to `true` and import everything you want to use:

```
1 @Component({
2   standalone: true,
3   selector: 'app-root',
4   imports: [
5     RouterOutlet,
6     NavbarComponent,
7     SidebarComponent,
8   ],
9   templateUrl: './app.component.html',
10  styleUrls: ['./app.component.css']
11 })
12 export class AppComponent {
13   [...]
14 }
```

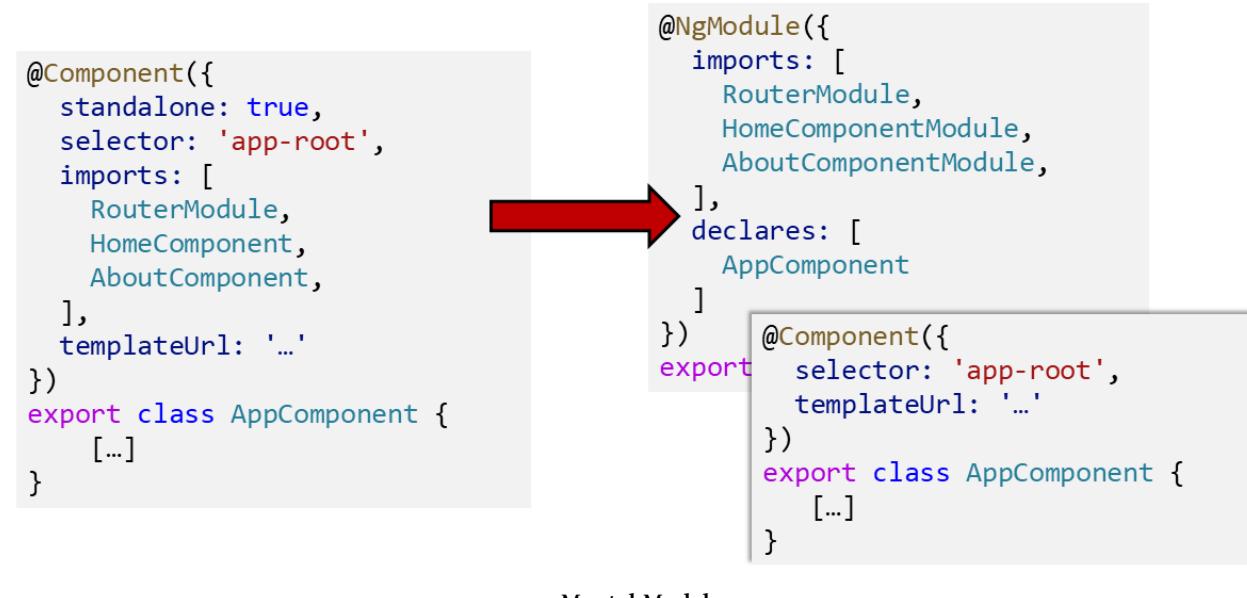
The `imports` define the compilation context: all the other building blocks the Standalone Components is allowed to use. For instance, you use it to import further Standalone Component, but also existing NgModules.

The exhaustive listing of all these building blocks makes the component self-sufficient and thus increases its reusability in principle. It also forces us to think about the component's dependencies. Unfortunately, this task turns out to be extremely monotonous and time consuming.

Therefore, there are considerations to implement a kind of auto-import in the Angular Language Service used by the IDEs. Analogous to the auto-import for TypeScript modules, the IDE of choice could also suggest placing the corresponding entry in the `imports` array the first time a component, pipe or directive is used in the template.

The Mental Model

The underlying mental model helps to better understand Standalone Components. In general, you can imagine a Standalone Component as a component with its very own NgModule:



This is similar to [Lars Nielsen](#)⁴'s SCAM pattern. However, while SCAM uses an explicit module, here we only talk about a thought one.

While this mental model is useful for understanding Angular's behavior, it's also important to see that Angular doesn't implement Standalone Components that way underneath the covers.

Pipes, Directives, and Services

Analogous to standalone components, there are also standalone pipes and standalone directives. For this purpose, the pipe and directive decorators also get a `standalone` property. This is what a standalone pipe will look alike:

⁴<https://twitter.com/LayZeeDK>

```

1  @Pipe ({
2    standalone: true,
3    name: 'city',
4    pure: true
5  })
6  export class CityPipe implements PipeTransform {
7
8    transform (value: string, format: string): string {[...]}
9
10 }

```

And here is an example for a standalone directive:

```

1  @Directive ({
2    standalone: true,
3    selector: 'input [appCity]',
4    providers: [...]
5  })
6  export class CityValidator implements Validator {
7
8    [...]
9
10 }

```

Thanks to tree-shakable providers, on the other hand, services have worked without modules for quite a time. For this purpose the property `providedIn` has to be used:

```

1  @Injectable ({
2    providedIn: 'root'
3  })
4  export class FlightService {[...]}

```

The Angular team recommends, to use `providedIn: 'root'` whenever possible. It might come as a surprise, but `providedIn: 'root'` also works with lazy loading: If you only use a service in lazy parts of your application, it is loaded alongside them.

Bootstrapping Standalone Components

Until now, modules were also required for bootstrapping, especially since Angular expected a module with a bootstrap component. Thus, this so called `AppModule` or “root module” defined the main component alongside its compilation context.

With Standalone Components, it will be possible to bootstrap a single component. For this, Angular provides a method `bootstrapApplication` which can be used in `main.ts`:

```
1 // main.ts
2
3 import { bootstrapApplication } from '@angular/platform-browser';
4 import { provideAnimations } from '@angular/platform-browser/animations';
5 import { AppComponent } from './app/app.component';
6 import { APP_ROUTES } from './app/app.routes';
7 import { provideRouter } from '@angular/router';
8 import { importProvidersFrom } from '@angular/core';
9
10 [...]
11
12 bootstrapApplication(AppComponent, {
13   providers: [
14     importProvidersFrom(HttpClientModule),
15     provideRouter(APP_ROUTES),
16     provideAnimations(),
17     importProvidersFrom(TicketsModule),
18     importProvidersFrom(LayoutModule),
19   ],
20 });
21
```

The first argument passed to `bootstrapApplication` is the main component. Here, it's our `AppComponent`. Via the second argument, we pass application-wide service providers. These are the providers, you would register with the `NgModule` when going with `NgModules`.

The provided helper function `importProvidersFrom` allows bridging the gap to existing `NgModules`. Please also note, that `importProvidersFrom` works with both `NgModules` but also `ModuleWithProviders` as returned by methods like `forRoot` and `forChild`.

While this allows to immediately leverage existing `NgModule`-based APIs, we will see more and more functions that replace the usage of `importProvidersFrom` in the future. For instance, to register the router with a given configuration, the function `provideRouter` is used. Similarly, `provideAnimations` setup up the Animation module.

Compatibility With Existing Code

As discussed above, according to the mental model, a Standalone Component is just a component with its very own `NgModule`. This is also the key for the compatibility with existing code still using `NgModules`.

On the one side, we can import whole `NgModules` into a Standalone Component:

```

1 import { Component, OnInit } from '@angular/core';
2 import { TicketsModule } from './tickets/tickets.module';
3
4 @Component({
5   selector: 'app-next-flight',
6   standalone: true,
7   imports: [
8     // Existing NgModule imported
9     // in this standalone component
10    TicketsModule
11  ],
12  [...]
13})
14 export class NextFlightComponent implements OnInit {
15  [...]
16}

```

But on the other side, we can also import a Standalone Component (Directive, Pipe) into an existing NgModule:

```

1 @NgModule({
2   imports: [
3     CommonModule,
4
5     // Imported Standalone Component:
6     FlightCardComponent,
7     [...]
8   ],
9   declarations: [
10     MyTicketsComponent
11   ],
12   [...]
13})
14 export class TicketsModule { }

```

Interestingly, standalone components are **imported** like modules and not declared like classic components. This may be confusing at first glance, but it totally fits the mental model that views a standalone component a component with its very own NgModule.

Also, declaring a traditional component defines a strong whole-part relationship: A traditional component can only be declared by one module and then, it belongs to this module. However, a standalone component doesn't belong to any NgModule but it can be reused in several places. Hence, using `imports` here really makes sense.

Side Note: The CommonModule

Doubtless, one of the most known NgModules in Angular was the `CommonModule`. It contains built-in directives like `*ngIf` or `*ngFor` and built-in pipes like `async` or `json`. While you can still import the whole `CommonModule`, meanwhile it's also possible to just import the needed directives and pipes:

```
1 import {
2     AsyncPipe,
3     JsonPipe,
4     NgForOf,
5     NgIf
6 } from "@angular/common";
7
8 [...]
9
10 @Component({
11     standalone: true,
12     imports: [
13         // CommonModule,
14         NgIf,
15         NgForOf,
16         AsyncPipe,
17         JsonPipe,
18
19         FormsModule,
20         FlightCardComponent,
21         CityValidator,
22     ],
23     selector: 'flight-search',
24     templateUrl: './flight-search.component.html'
25 })
26 export class FlightSearchComponent implements OnInit {
27     [...]
28 }
```

This is possible, because the Angular team made Standalone Directives and Standalone Pipes out of the building blocks provided by the `CommonModule`. Importing these building blocks in a fine grained way will be especially interesting once IDEs provide auto-imports for standalone components. In this case, the first usage of an building block like `*ngIf` will make the IDE to add it to the `imports` array.

As we will see in a further part of this book, meanwhile also the `RouterModule` comes with Standalone building-blocks. Hence, we can directly import `RouterOutlet` instead of

going with the whole `RouterModule`. When writing this, this was not yet possible for other modules like the `FormsModule` or the `HttpClientModule`.

Interim Conclusion: Standalone Components – and now?

So far we've seen how to use Standalone Components to make our Angular applications more lightweight. We've also seen that the underlying mental model guarantees compatibility with existing code.

However, now the question arises how this all will influence our application structure and architecture. The next chapter will shed some light on this.

Architecture with Standalone Components

In last chapter, I've shown how Standalone Components will make our Angular applications more lightweight in the future. In this part, I'm discussing options for improving your architecture with them.

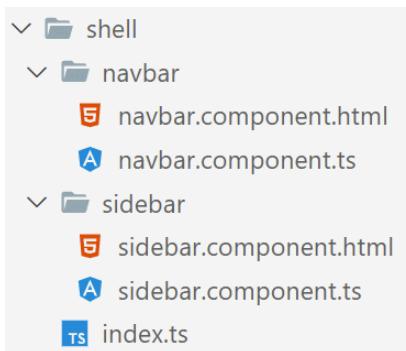
The source code for this can be found in the form of a traditional [Angular CLI workspace⁵](#) and as an [Nx workspace⁶](#) that uses libraries as a replacement for NgModules.

Grouping Building Blocks

Unfortunately, the examples shown so far cannot keep up with one aspect of NgModules. Namely the possibility of grouping building blocks that are usually used together.

Obviously, the easiest approach for grouping stuff that goes together is using folders. However, you might go one step further by leveraging barrels: A barrel is an EcmaScript file that exports related elements.

These files are often called `public-api.ts` or `index.ts`. The example project used contains such an `index.ts` to group two navigation components from the shell folder:



Grouping two Standalone Components with a barrel

The barrel itself re-exports the two components:

⁵<https://github.com/manfredsteyer/standalone-example-cli>

⁶<https://github.com/manfredsteyer/standalone-example-nx>

```
1 export { NavbarComponent } from './navbar/navbar.component';
2 export { SidebarComponent } from './sidebar/sidebar.component';
```

The best of this is, you get real modularization: Everything the barrel experts can be used by other parts of your application. Everything else is your secret. You can modify these secrets as you want, as long as the public API defined by your barrel stays backwards compatible.

In order to use the barrel, just point to it with an `import`:

```
1 import {
2     NavbarComponent,
3     SidebarComponent
4 } from './shell/index';
5
6 @Component({
7     standalone: true,
8     selector: 'app-root',
9     imports: [
10         RouterOutlet,
11
12         NavbarComponent,
13         SidebarComponent,
14     ],
15     templateUrl: './app.component.html',
16     styleUrls: ['./app.component.css']
17 })
18 export class AppComponent {
19     [...]
20 }
```

If you call your barrel `index.ts`, you can even omit the file name, as `index` is the default name when configuring the TypeScript compiler to use Node.js-based conventions. Something that is the case in the world of Angular and the CLI:

```
1 import {
2     NavbarComponent,
3     SidebarComponent
4 } from './shell';
5
6 @Component({
7     standalone: true,
8     selector: 'app-root',
9     imports: [
10         RouterOutlet,
11
12         NavbarComponent,
13         SidebarComponent,
14     ],
15     templateUrl: './app.component.html',
16     styleUrls: ['./app.component.css']
17 })
18 export class AppComponent {
19     [...]
20 }
```

Importing Whole Barrels

In the last section, the `NavbarComponent` and the `SidebarComponent` were part of the shell's public API. Nevertheless, Angular doesn't provide a way to import everything a barrel provides at once.

In most of the cases, this is the totally fine. Auto-imports will add the needed entries anyway, hence this style of programming is easy. Also, being explicit about what you need helps enables tree-shaking.

However, in some edge-cases where you know that some building blocks **always** go together, e. g. because there is a strong mutual dependency, putting them into an array can help to make our lives easier. For instance, think about all the directives provided by the `FormsModule`. Normally, we don't even know their exact names nor which of them play together.

The following example demonstrates this idea:

```

1 import { NavbarComponent } from './navbar/navbar.component';
2 import { SidebarComponent } from './sidebar/sidebar.component';
3
4 export { NavbarComponent } from './navbar/navbar.component';
5 export { SidebarComponent } from './sidebar/sidebar.component';
6
7 export const SHELL = [
8   NavbarComponent,
9   SidebarComponent
10];

```

Interestingly, such arrays remind us to the exports section of NgModules. Please note that your array needs to be a constant. This is needed because the Angular Compiler uses it already at compile time.

Such arrays can be directly put into the imports array. No need for spreading them:

```

1 import { SHELL } from './shell';
2
3 [...]
4
5 @Component({
6   standalone: true,
7   selector: 'app-root',
8   imports: [
9     RouterOutlet,
10
11     // NavbarComponent,
12     // SidebarComponent,
13     SHELL
14   ],
15   templateUrl: './app.component.html',
16   styleUrls: ['./app.component.css']
17 })
18 export class AppComponent {
19 [...]
20 }

```

One more time I want to stress out that this array-based style should only be used with caution. While it allows to group things that always go together it also makes your code less tree-shakable.

Barrels with Pretty Names: Path Mappings

Just using import statements that directly point to other parts of your application often leads to long relative and confusing paths:

```

1 import { SHELL } from './../../../../shell';
2
3 @Component ({
4     standalone: true,
5     selector: 'app-my-cmp',
6     imports: [
7         SHELL,
8         [...]
9     ]
10 })
11 export class MyComponent {
12 }

```

To bypass this, you can define path mappings for your barrels you import from in your TypeScript configuration (`tsconfig.json` in the project's root):

```

1 "paths": {
2     "@demo/shell": ["src/app/shell/index.ts"],
3     [...]
4 }

```

This allows direct access to the barrel using a well-defined name without having to worry about - sometimes excessive - relative paths:

```

1 // Import via mapped path:
2 import { SHELL } from '@demo/shell';
3
4 @Component ({
5     standalone: true,
6     selector: 'app-root',
7     imports: [
8         SHELL,
9         [...]
10    ]
11 })
12 export class MyComponent {
13 }

```

The Next Logical Step: Workspace Libraries and Nx

These path mappings can of course be created manually. However, it is a little easier with the CLI extension [Nx⁷](#) which automatically generates such path mappings for each library created within a

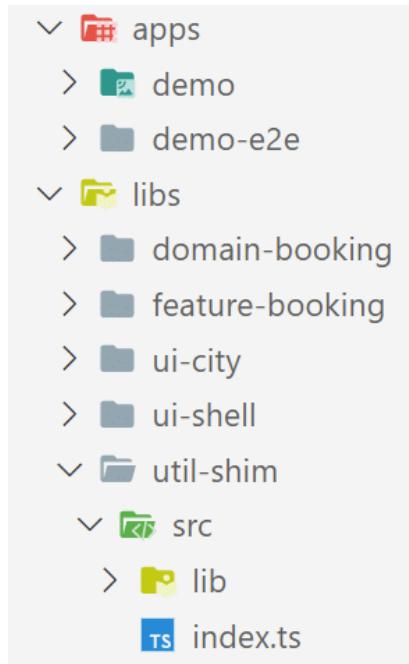
⁷<https://nx.dev/>

workspace. Libraries seem to be the better solution anyway, especially since they subdivide it more and Nx prevents bypassing the barrel of a library.

This means that every library consists of a public – actually published – and a private part. The library's public and private APIs are also mentioned here. Everything the library exports through its barrel is public. The rest is private and therefore a “secret” of the library that other parts of the application cannot access.

It is precisely these types of “secrets” that are a simple but effective key to stable architectures, especially since everything that is not published can easily be changed afterwards. The public API, on the other hand, should only be changed with care, especially since a breaking change can cause problems in other areas of the project.

An Nx project (workspace) that represents the individual sub-areas of the Angular solution as libraries could use the following structure:



Structure of an Nx Solution

Each library receives a barrel that reflects the public API. The prefixes in the library names reflect a categorization recommended by the Nx team. For example, feature libraries contain smart components that know the use cases, while UI libraries contain reusable dumb components. The domain library comes with the client-side view of our domain model and the services operating on it, and utility libraries contain general auxiliary functions.

On the basis of such categories, Nx allows the definition of linting rules that prevent undesired access between libraries. For example, you could specify that a domain library should only have access to utility libraries and not to UI libraries:

```
import { NavbarComponent } from '@demo/ui-shell';
(alias) class NavbarComponent
import NavbarComponent

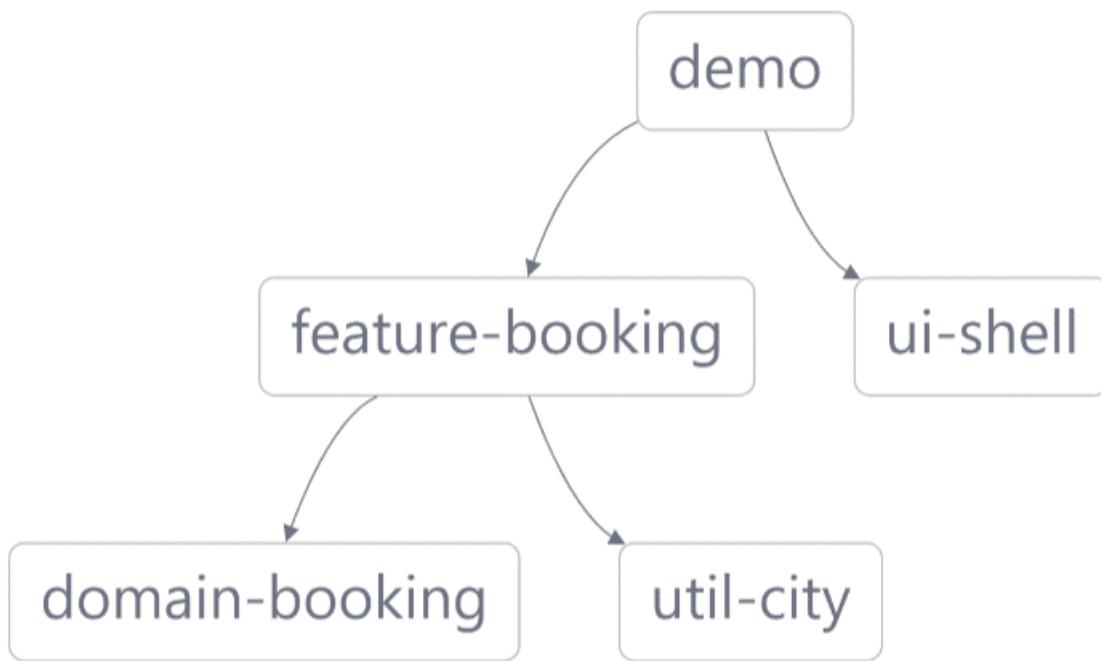
"NavbarComponent" ist deklariert, aber der zugehörige Wert
wird nie gelesen. ts(6133)
```

A project tagged with "domain" can only depend on libs tagged with "util" eslint([@nrwl/nx/enforce-module-boundaries](#))

'NavbarComponent' is defined but never
used. eslint([@typescript-eslint/no-unused-vars](#))

Nx prevents unwanted access between libraries via linting

In addition, Nx allows the dependencies between libraries to be visualized:



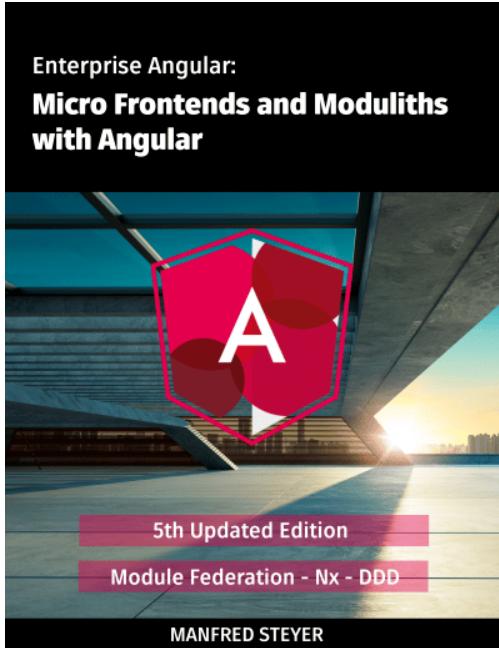
Nx visualizes the dependencies between libraries

If you want to see all of this in action, feel free to have a look at the Nx version of the example used here. You find the [Source Code at GitHub](#)⁸.

⁸<https://github.com/manfredsteyer/demo-nx-standalone>

More on Architectures with Nx

More information about architectures with Nx can be found in our free eBook (12 chapters) on Angular Architectures:



Free ebook

Feel free to download it here⁹ now!

Conclusion

Standalone Components make the future of Angular applications more lightweight. We don't need NgModules anymore. Instead, we just use EcmaScript modules. This makes Angular solutions more straightforward and lowers the entry barrier into the world of the framework. Thanks to the mental model, which regards standalone components as a combination of a component and a NgModule, this new form of development remains compatible with existing code.

For the grouping of related building blocks, simple barrels are ideal for small solutions. For larger projects, the transition to monorepos as offered by the CLI extension Nx seems to be the next logical step. Libraries subdivide the overall solution here and offer public APIs based on barrels. In addition, dependencies between libraries can be visualized and avoided using linting.

⁹<https://www.angulararchitects.io/book>

Routing and Lazy Loading

Since its first days, the Angular Router has always been quite coupled to NgModules. Hence, one question that comes up when moving to Standalone Components is: How will routing and lazy loading work without NgModules? This chapter provides answers and also shows, why the router will become more important for Dependency Injection.

The **source code** for the examples used here can be found in the form of a traditional [Angular CLI workspace¹⁰](#) and as an [Nx workspace¹¹](#) that uses libraries as a replacement for NgModules.

Providing the Routing Configuration

When bootstrapping a standalone component, we can provide services for the root scope. These are services you used to provide in your AppModule. Meanwhile, the Router provides a function `provideRouter` that returns all providers we need to register here:

```
1 // main.ts
2
3 import { importProvidersFrom } from '@angular/core';
4 import { bootstrapApplication } from '@angular/platform-browser';
5 import {
6   PreloadAllModules,
7   provideRouter,
8   withDebugTracing,
9   withPreloading,
10  withRouterConfig
11 }
12 from '@angular/router';
13
14 import { APP_ROUTES } from './app/app.routes';
15 [...]
16
17 bootstrapApplication(AppComponent, {
18   providers: [
19     importProvidersFrom(HttpClientModule),
```

¹⁰<https://github.com/manfredsteyer/standalone-example-cli>

¹¹<https://github.com/manfredsteyer/standalone-example-nx>

```
20     provideRouter(APP_ROUTES,
21         withPreloading(PreloadAllModules),
22         withDebugTracing(),
23     ),
24
25     [...]
26
27     importProvidersFrom(TicketsModule),
28     provideAnimations(),
29     importProvidersFrom(LayoutModule),
30   ]
31 );
```

The function `provideRouter` not only takes the root routes but also the implementation of additional router features. These features are passed with functions having the naming pattern `withXYZ`, e. g. `withPreloading` or `withDebugTracing`. As functions can easily be tree-shaken, this design decisions makes the whole router more tree-shakable.

With the discussed functions, the Angular team also introduces a naming pattern, library authors should follow. Hence, when adding a new library, we just need to look out for an `provideXYZ` and for some optional `withXYZ` functions.

As currently not every library comes with a `provideXYZ` function yet, Angular comes with the bridging function `importProvidersFrom`. It allows to get hold of all the providers defined in existing NgModules and hence is the key for using them with Standalone Components.

I'm quite sure, the usage of `importProvidersFrom` will peak off over time, as more and more libraries will provide functions for directly configuring their providers. For instance, NGRX recently introduced a `provideStore` and a `provideEffects` function.

Using Router Directives

After setting up the routes, we also need to define a placeholder where the Router displays the activated component and links for switching between them. To get the directives needed for this, you might directly import the `RouterModule` into your Standalone Component. However, a better alternative is to just import the directives you need:

```
1  @Component({
2      standalone: true,
3      selector: 'app-root',
4      imports: [
5          // Just import the RouterModule:
6          // RouterModule,
7
8          // Better: Just import what you need:
9          RouterOutlet,
10         RouterLinkWithHref, // Angular 14
11         // RouterLink // Angular 15+
12
13         NavbarComponent,
14         SidebarComponent,
15     ],
16     templateUrl: './app.component.html',
17     styleUrls: ['./app.component.css']
18 })
19 export class AppComponent {
20     [...]
21 }
```

Just importing the actually needed directives is possible, because the router exposes them as Standalone Directives. Please note that in Angular 14, `RouterLinkWithHref` is needed if you use `routerLink` with an `a`-tag; in all other cases you should import `RouterLink` instead. As this is a bit confusing, the Angular Team refactored this for Angular 15: Beginning with this version, `RouterLink` is used in all cases.

In most cases, this is nothing we need to worry about when IDEs start providing auto-imports for Standalone Components.

Lazy Loading with Standalone Components

In the past, a lazy route pointed to an `NgModule` with child routes. As there are no `NgModules` anymore, `loadChildren` can now directly point to a lazy routing configuration:

```

1 // app.routes.ts
2
3 import { Routes } from '@angular/router';
4 import { HomeComponent } from './home/home.component';
5
6 export const APP_ROUTES: Routes = [
7   {
8     path: '',
9     pathMatch: 'full',
10    redirectTo: 'home'
11  },
12  {
13    path: 'home',
14    component: HomeComponent
15  },
16
17 // Option 1: Lazy Loading another Routing Config
18 {
19   path: 'flight-booking',
20   loadChildren: () =>
21     import('./booking/flight-booking.routes')
22       .then(m => m.FLIGHT_BOOKING_ROUTES)
23 },
24
25 // Option 2: Directly Lazy Loading a Standalone Component
26 {
27   path: 'next-flight',
28   loadComponent: () =>
29     import('./next-flight/next-flight.component')
30       .then(m => m.NextFlightComponent)
31 },
32 [...]
33 ];

```

This removes the indirection via an NgModule and makes our code more explicit. As an alternative, a lazy route can also directly point to a Standalone Component. For this, the above shown `loadComponent` property is used.

I expect that most teams will favor the first option, because normally, an application needs to lazy loading several routes that go together.

Environment Injectors: Services for Specific Routes

With NgModules, each lazy module introduced a new injector and hence a new injection scope. This scope was used for providing services only needed by the respective lazy chunk.

To cover such use cases, the Router now allows for introducing providers for each route. These services can be used by the route in question and their child routes:

```

1 // booking/flight-booking.routes.ts
2
3 export const FLIGHT_BOOKING_ROUTES: Routes = [
4     {
5         path: '',
6         component: FlightBookingComponent,
7         providers: [
8             provideBookingDomain(config)
9         ],
10        children: [
11            {
12                path: '',
13                pathMatch: 'full',
14                redirectTo: 'flight-search'
15            },
16            {
17                path: 'flight-search',
18                component: FlightSearchComponent
19            },
20            {
21                path: 'passenger-search',
22                component: PassengerSearchComponent
23            },
24            {
25                path: 'flight-edit/:id',
26                component: FlightEditComponent
27            }
28        ]
29    ];
30];

```

As shown here, we can provide services for several routes by grouping them as child routes. In these cases, a component-less parent route with an empty path (`path: ''`) is used. This pattern is already used for years to assign Guards to a group of routes.

Technically, using adding a `providers` array to a router configuration introduces a new injector at the level of the route. Such an injector is called Environment Injector and replaces the concept of the

former (Ng)Module Injectors. The root injector and the platform injector are further Environment Injectors.

Interestingly, this also decouples lazy loading from introducing further injection scopes. Previously, each **lazy** NgModule introduced a new injection scope, while **non-lazy** NgModules never did. Now, lazy loading itself doesn't influence the scopes. Instead, now, you define new scopes by adding a providers array to your routes, **regardless** if the route is lazy or not.

The Angular team recommends to use this providers array with caution and to **favor** providedIn: 'root' instead. As already mentioned in a previous chapter, also providedIn: 'root' allows for lazy loading. If you just use a services provided with providedIn: 'root' in lazy parts of your application, they will only be loaded together with them.

However, there is one situation where providedIn: 'root' does not work and hence the providers array shown is needed, namely if you need to pass a configuration to a library. I've already indicated this in the above example by passing a config object to my custom provideBookingDomain. The next section provides a more elaborated example for this using NGRX.

Setting up NGRX and Feature Slices

To illustrate how to use libraries adopted for Standalone Components with lazy loading, let's see how to setup NGRX. Let's start with providing the needed global services:

```
1 import { bootstrapApplication } from '@angular/platform-browser';
2
3 import { provideStore } from '@ngrx/store';
4 import { provideEffects } from '@ngrx/effects';
5 import { provideStoreDevtools } from '@ngrx/store-devtools';
6
7 import { reducer } from './app/+state';
8
9 [...]
10
11 bootstrapApplication(AppComponent, {
12   providers: [
13     importProvidersFrom(HttpClientModule),
14     provideRouter(APP_ROUTES,
15       withPreloading(PreloadAllModules),
16       withDebugTracing(),
17     ),
18
19     // Setup NGRX:
20     provideStore(reducer),
```

```

21     provideEffects([]),
22     provideStoreDevtools(),
23
24     importProvidersFrom(TicketsModule),
25     provideAnimations(),
26     importProvidersFrom(LayoutModule),
27   ]
28 });

```

For this, we go with the functions `provideStore`, `provideEffects`, and `provideStoreDevtools` NGRX comes with since version 14.3.

To allow lazy parts of the application to have their own feature slices, we call `provideState` and `provideEffects` in the respective routing configuration:

```

1 import { provideEffects } from "@ngrx/effects";
2 import { provideState } from "@ngrx/store";
3
4 export const FLIGHT_BOOKING_ROUTES: Routes = [{{
5   path: '',
6   component: FlightBookingComponent,
7   providers: [
8     provideState(bookingFeature),
9     provideEffects([BookingEffects])
10  ],
11   children: [
12     {
13       path: 'flight-search',
14       component: FlightSearchComponent
15     },
16     {
17       path: 'passenger-search',
18       component: PassengerSearchComponent
19     },
20     {
21       path: 'flight-edit/:id',
22       component: FlightEditComponent
23     }
24   ]
25 }]];

```

While `provideStore` sets up the store at root level, `provideState` sets up additional feature slices. For this, you can provide a feature or just a branch name with a reducer. Interestingly, the function

`provideEffects` is used at the root level but also at the level of lazy parts. Hence, it provides the initial effects but also effects needed for a given feature slice.

Setting up Your Environment: ENVIRONMENT_INITIALIZER

Some libraries used the constructor of lazy `NgModule` for their initialization. To further support this approach without `NgModules`, there is now the concept of an `ENVIRONMENT_INITIALIZER`:

```
1 export const FLIGHT_BOOKING_ROUTES: Routes = [{  
2     path: '',  
3     component: FlightBookingComponent,  
4     providers: [  
5         importProvidersFrom(StoreModule.forFeature(bookingFeature)),  
6         importProvidersFrom(EffectsModule.forFeature([BookingEffects])),  
7         {  
8             provide: ENVIRONMENT_INITIALIZER,  
9             multi: true,  
10            useValue: () => inject(InitService).init()  
11        }  
12    ],  
13    children: [  
14        [...]  
15    ]  
16 }
```

Basically, the `ENVIRONMENT_INITIALIZER` provides a function executed when the Environment Injector is initialized. The flag `multi: true` already indicates that you can have several such initializers per scope.

Angular Elements with Standalone Components

Since Angular 14.2, it's possible to use Standalone Components as Angular Elements. In this chapter, I'm going to show you, how this new feature works.

[Source Code¹²](#)

Providing a Standalone Component

The Standalone Component I'm going to use here is a simple Toggle Button called ToggleComponent:

```
1 import { Component, EventEmitter, Input, Output, ViewEncapsulation } from '@angular/\n2 core';\n3 import { CommonModule } from '@angular/common';\n4\n5 @Component({\n6   selector: 'app-toggle',\n7   standalone: true,\n8   imports: [],\n9   template: `'\n10    <div class="toggle" [class.active]="active" (click)="toggle()">\n11      <slot>Toggle!</slot>\n12    </div>\n13  `,\n14   styles: [`'\n15     .toggle {\n16       padding:10px;\n17       border: solid black 1px;\n18       cursor: pointer;\n19       display: inline\n20     }\n21\n22     .active {\n23       background-color: lightsteelblue;\n24     }\n25`]\n26 })\n27\n28 export class ToggleComponent {\n29   active = false;\n30\n31   toggle() {\n32     this.active = !this.active;\n33   }\n34 }
```

¹²<https://github.com/manfredsteyer/standalone-components-elements>

```
25      `],
26      encapsulation: ViewEncapsulation.ShadowDom
27  })
28  export class ToggleComponent {
29
30    @Input() active = false;
31    @Output() change = new EventEmitter<boolean>();
32
33    toggle(): void {
34      this.active = !this.active;
35      this.change.emit(this.active);
36    }
37
38 }
```

By setting encapsulation to `ViewEncapsulation.ShadowDom`, I'm making the browser to use "real" Shadow DOM instead of Angular's emulated counterpart. However, this also means that we have to use the Browser's `slot` API for content projection instead of Angular's `ng-content`.

Installing Angular Elements

While `Angular Elements` is directly provided by the Angular team, the CLI doesn't install it. Hence, we need to do this by hand:

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

In former days, `@angular/elements` also supported `ng add`. This support came with a schematic for adding a needed polyfill. However, meanwhile, all browsers supported by Angular can deal with Web Components natively. Hence, there is no need for such a polyfill anymore and so the support for `ng add` was already removed some versions ago.

Bootstrapping with Angular Elements

Now, let's bootstrap our application and expose the `ToggleComponent` as a Web Component (Custom Element) with Angular Elements. For this, we can use the function `createApplication` added with Angular 14.2:

```
1 // main.ts
2
3 import { createCustomElement } from '@angular/elements';
4 import { createApplication } from '@angular/platform-browser';
5 import { ToggleComponent } from './app/toggle/toggle.component';
6
7 (async () => {
8
9   const app = await createApplication({
10     providers: [
11       /* your global providers here */
12     ],
13   });
14
15   const toogleElement = createCustomElement(ToggleComponent, {
16     injector: app.injector,
17   });
18
19   customElements.define('my-toggle', toogleElement);
20
21 })();
```

We could pass an array with providers to `createApplication`. This allows to provide services like the `HttpClient` via the application's root scope. In general, this option is needed when we want to configure these providers, e. g. with a `forRoot` method or a `provideXYZ` function. In all other cases, it's preferable to just go with tree-shakable providers (`providedIn: 'root'`).

The result of `createApplication` is a new `ApplicationRef`. We can pass it's Injector alongside the `ToggleComponent` to `createCustomElement`. The result is a custom element that can be registered with the browser using `customElements.define`.

Please note that the current API does not allow for setting an own zone instance like the `noop` zone. Instead, the Angular team wants to concentrate on new features for zone-less change detection in the future.

Side Note: Bootstrapping Multiple Components

The API shown also allows to create several custom elements:

```
1 const element1 = createCustomElement(ThisComponent, {
2   injector: app.injector,
3 });
4
5 const element2 = createCustomElement(ThatComponent, {
6   injector: app.injector,
7 });
```

Besides working with custom elements, the `ApplicationRef` at hand also allows for bootstrapping several components as Angular applications:

```
1 app.injector.get(NgZone).run(() => {
2   app.bootstrap(ToggleComponent, 'my-a');
3   app.bootstrap(ToggleComponent, 'my-b');
4});
```

When bootstrapping a component this way, one can overwrite the selector to use. Please note, that one has to call `bootstrap` within a zone in order to get change detection.

Bootstrapping several components was originally done by placing several components in your `AppModule`'s `bootstrap` array. The `bootstrapApplication` function used for bootstrapping Standalone Components does, however, not allow for this as the goal was to provide a simple API for the most common use case.

Calling an Angular Element

To call our Angular Element, we just need to place a respective tag in our `index.html`:

```
1 <h1>Standalone Angular Element Demo</h1>
2 <my-toggle id="myToggle">Click me!</my-toggle>
```

As a custom element is threaded by the browser as a normal DOM node, we can use traditional DOM calls to set up events and to assign values to properties:

```

1 <script>
2   const myToggle = document.getElementById('myToggle');
3
4   myToggle.addEventListener('change', (event) => {
5     console.log('active', event.detail);
6   });
7
8   setTimeout(() => {
9     myToggle.active = true;
10    }, 3000);
11 </script>

```

Calling a Web Component in an Angular Component

If we call a web component within an Angular component, we can directly data bind to it using brackets for properties and parenthesis for events. This works regardless whether the web component was created with Angular or not.

To demonstrate this, let's assume we have the following AppComponent:

```

1 import { Component, CUSTOM_ELEMENTS_SCHEMA } from '@angular/core';
2
3 @Component({
4   selector: 'app-root',
5   standalone: true,
6   schemas: [CUSTOM_ELEMENTS_SCHEMA],
7   template: `
8     <h2>Root Component</h2>
9     <my-toggle
10       [active]="active"
11       (change)="change($event)">
12       Hello!
13     </my-toggle>
14   `,
15 })
16 export class AppComponent {
17   active = false;
18   change(event: Event) {
19     const customEvent = event as CustomEvent<boolean>;
20     console.log('active', customEvent.detail);
21   }
22 }

```

This Standalone Component calls our `my-toggle` web component. While the Angular compiler is aware of all possible Angular components, it doesn't know about web components. Hence, it would throw an error when seeing the `my-toggle` tag. To avoid this, we need to register the `CUSTOM_ELEMENTS_SCHEMA` schema.

Before, we did this with all the NgModules we wanted to use together with Web Components. Now, we can directly register this schema with Standalone Components. Technically, this just disables the compiler checks regarding possible tag names. This is binary - the checks are either on or off – and there is no way to directly tell the compiler about the available web components.

To make this component appear on our page, we need to bootstrap it:

```

1 // main.ts
2
3 [...]
4 // Register web components ...
5 [...]
6
7 app.injector.get(NgZone).run(() => {
8   app.bootstrap(AppComponent);
9 });

```

Also, we need to add an element for `AppComponent` to the `index.html`:

```
1 <app-root></app-root>
```

Bonus: Compiling Self-contained Bundle

Now, let's assume, we only provide a custom element and don't bootstrap our `AppComponent`. In order to use this custom element in other applications, we need to compile it into a self contained bundle. While the default webpack-based builder emits several bundles, e. g. a main bundle and a runtime bundle, the new – still experimental – esbuild-based one just gives us one bundle for our source code and another one for the polyfills.

To activate it, adjust your project settings in your `angular.json` as follows:

```

1 "build": {
2   "builder": "@angular-devkit/build-angular:browser-esbuild",
3   [...]
4 }

```

Normally, you just have to add `-esbuild` at the end of the default builder.

The resulting bundles look like this:

```
1      948 favicon.ico
2      703 index.html
3  100 177 main.43BPAPVS.js
4    33 916 polyfills.M7XCYQVG.js
5        0 styles.VFVLKGBH.css
```

If you use your web component in an other web site, e. g. a CMS-driven one, just reference the main bundle there and add a respective tag. Also, reference the polyfills. However, when using several such bundles, you have to make sure, you only load the polyfills once.

Migrating for Angular Standalone Components

After getting started with Standalone Components the question arises how to migrate an existing Angular solution for a future without Angular modules. In this chapter I show four options to do so.

Option 1: Ostrich Strategy

Let's start with the simplest option - the ostrich strategy. Stick your head in the sand and ignore everything around you:



Ostrich sticking its head into the sand

Even if that sounds smug, there is actually nothing wrong with it. Nobody is forcing us to convert applications to Standalone Components. Angular will continue to support Angular modules. After all, the entire ecosystem is based on it. You can therefore safely ignore Standalone Components or only use this new option in new applications or application parts.

Option 2: Just Throw Away Angular Modules

This strategy also seems smug at first glance: You simply remove all Angular modules from your source code. This doesn't have to be done in one go either, because Standalone Components play

wonderfully together with Angular modules. Angular modules can be imported into Standalone Components and vice versa.

For instance, the following listing shows a Standalone Component importing further NgModules:

```

1 import { Component, OnInit } from '@angular/core';
2 import { TicketsModule } from './tickets/tickets.module';
3
4 @Component({
5   selector: 'app-next-flight',
6   standalone: true,
7   imports: [
8     // Existing NgModule imported
9     // in this standalone component
10    TicketsModule
11  ],
12  [...]
13})
14 export class NextFlightComponent implements OnInit {
15  [...]
16}

```

To illustrate the other way round, this listing shows an NgModule importing a Standalone Component:

```

1 @NgModule({
2   imports: [
3     CommonModule,
4
5     // Imported Standalone Component:
6     FlightCardComponent,
7     [...]
8   ],
9   declarations: [
10     MyTicketsComponent
11   ],
12   [...]
13})
14 export class TicketsModule { }

```

This mutual compatibility is made possible by the [mental model¹³](#) behind Standalone Components.

¹³<https://www.angulararchitects.io/en/aktuelles/angulards-future-without-ngmodules-lightweight-solutions-on-top-of-standalone-components/>

Accordingly, a standalone component is a combination of a component and a module. Even if the actual technical implementation does not set up any dedicated Angular modules, this idea helps to bridge the gap between the two worlds. It also explains why Angular modules and Standalone Components can import each other.

If you go with this strategy, you need to import the compilation context directly into the Standalone Component using its `imports` array. I like to think about this compilation context as about the component's neighbor hood: It contains all other Standalone Components, Standalone Directives, and Standalone Pipes but also NgModules the component in question needs.

Hopefully, the Angular language service and hence editors/ IDEs like Visual Studio Code or WebStorm/ IntelliJ will provide `auto imports` for this task. To help with this too, my colleague Rainer Hahnekamp wrote a nice [schematic¹⁴](#) that automates some of the steps needed. Currently, we don't think that full automation is possible here, because for this we need some knowledge about the application. For instance, we need to know which providers are used where.

Option 3: Replace Angular Modules with Barrels

Barrels are EcmaScript files that (re)export related building blocks:

```
1 import { NavbarComponent } from './navbar/navbar.component';
2 import { SidebarComponent } from './sidebar/sidebar.component';
```

The consumer can now import everything the barrel provides:

```
1 import { NavbarComponent, SidebarComponent } as shell from './shell';
```

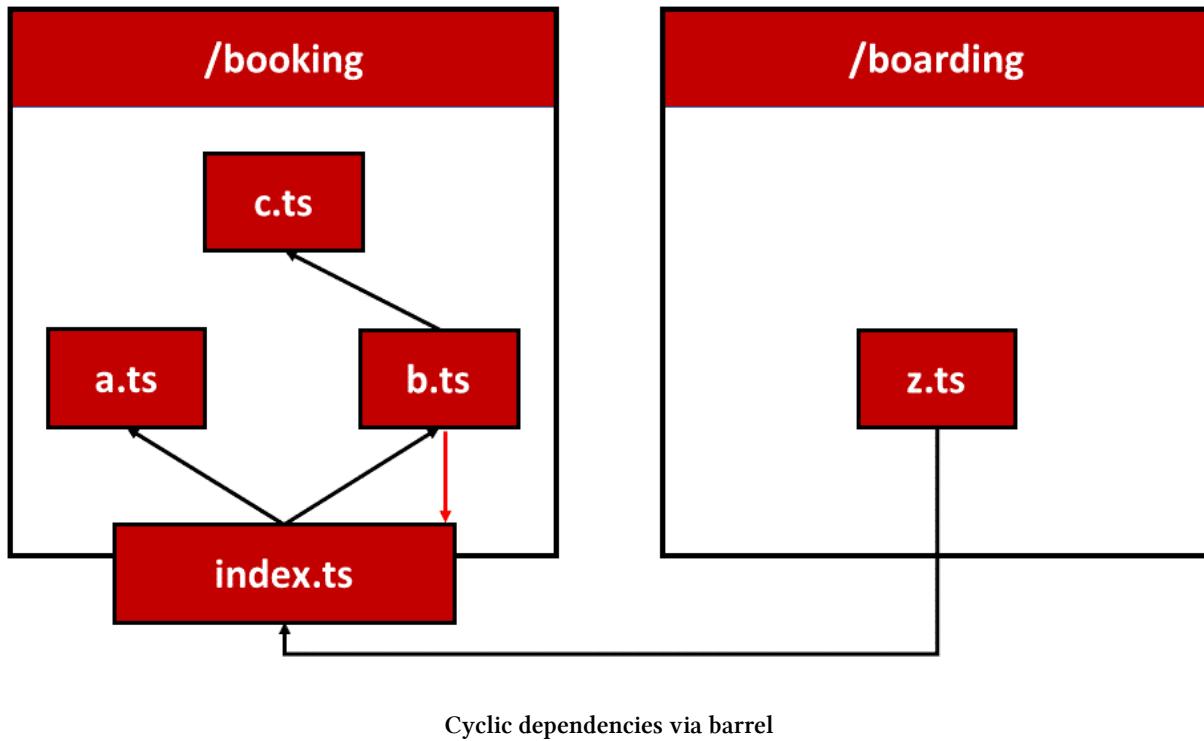
If the barrel is called `index.ts`, it is sufficient to import only the barrel folder. In addition to grouping, this approach also has the advantage that barrels can be used to define **public APIs**: All building blocks exported by the barrel can be used by other parts of the application. They just need to import from the barrel. Everything else is considered an implementation detail that should not be accessed by other application parts. Hence, such implementation details are quite easy to change without producing breaking changes somewhere else. This is a simple but effective measure for stable software architectures.

In a further step, each barrel could also receive a path mapping in the `tsconfig.json`. In this case, the application can access the barrel using nice names similar to npm package names:

```
1 import { NavbarComponent, SidebarComponent } from '@demo/shell';
```

¹⁴<https://www.npmjs.com/package/@angular-architects/sam4sc>

However, barrels also come with challenges: For example, they are often the cause of cyclical dependencies:

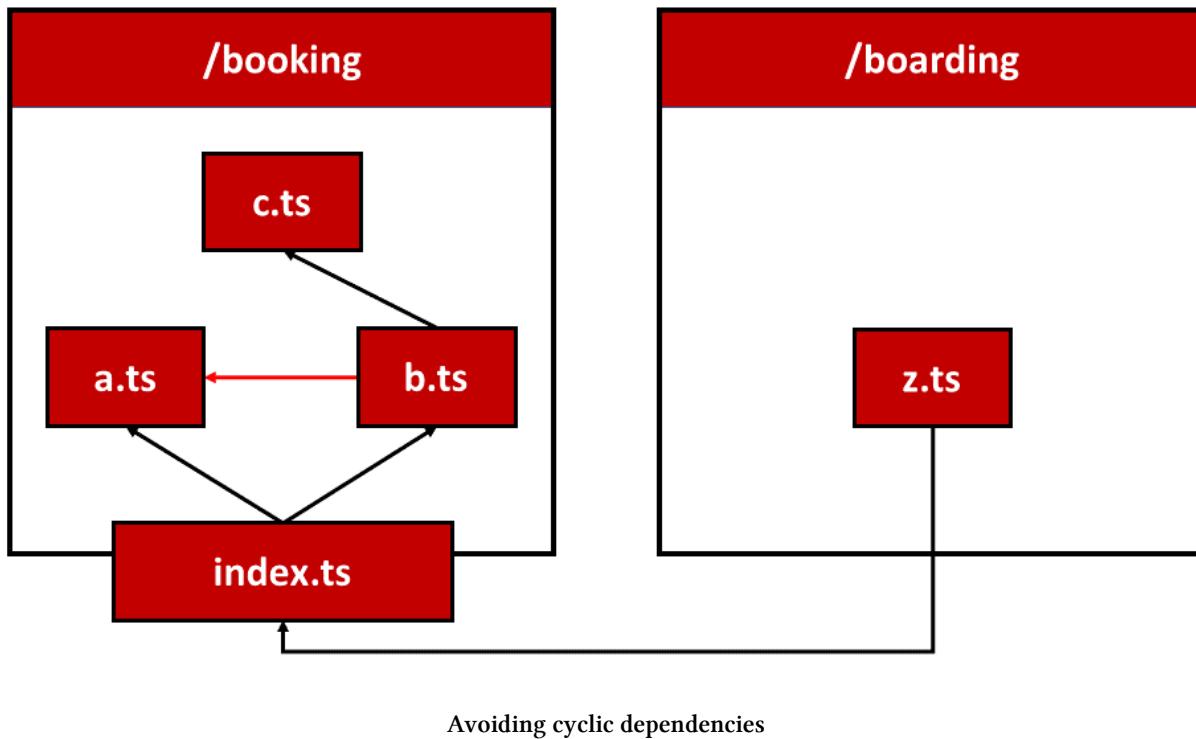


Here, `b.ts` on the one hand is referenced by the barrel `index.ts` and on the other hand accesses the barrel.

This problem can be avoided from the start with two simple rules that must be followed consistently:

- A barrel may only publish elements from its “area”. The “area” extends over the barrel’s folder as well as its subfolders.
- Within each “area”, files reference each other using relative paths without using the barrel.

Although these rules sound a bit abstract at first glance, the implementation of this rule is easier than you would think:



Here, `b.ts` directly accesses `a.ts` located in the same “area” to avoid the cycle shown earlier. The detour the barrel is avoided.

Another disadvantage is that each part of the program can bypass the specified barrels - and thus the public API created with them. Relative paths to private parts of the respective “areas” are sufficient for this.

This problem can be solved with linting. A linting rule would have to detect and denounce unauthorized access. The popular tool [Nx¹⁵](#) comes with such a rule, which can also be used to prevent other unwanted accesses. The next section takes up this idea.

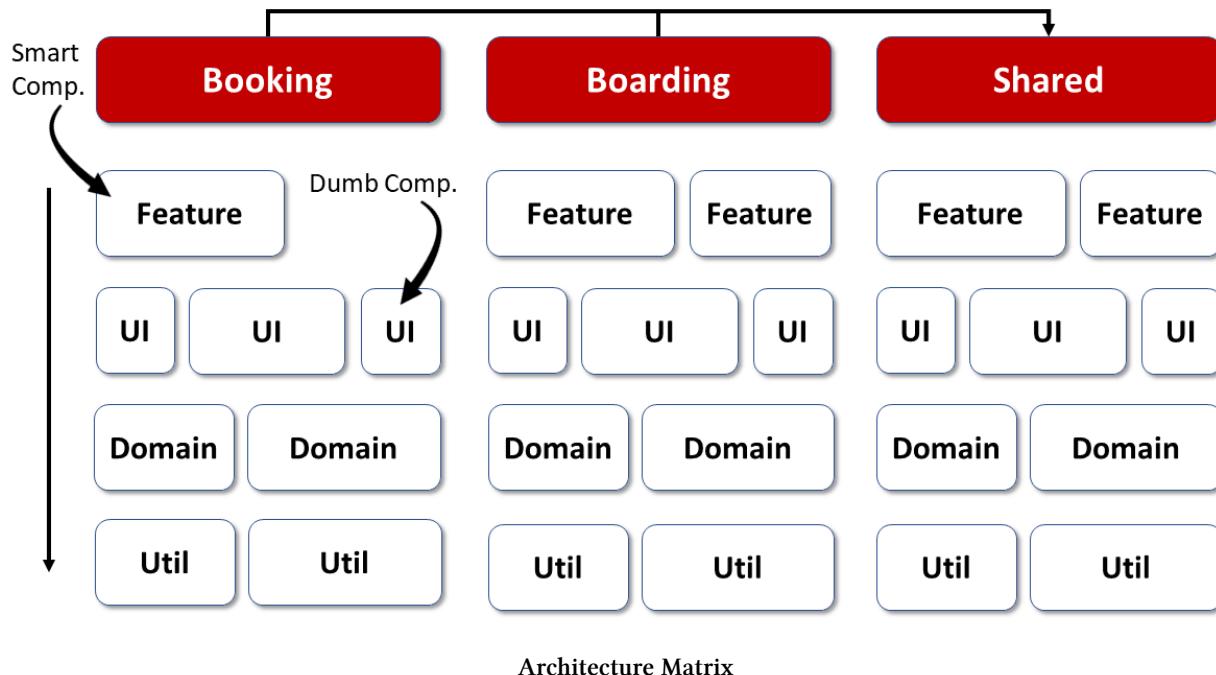
Option 4: Nx Workspace with Libraries and Linting Rules

The popular tool [Nx¹⁶](#) is based on the Angular CLI and brings a lot of convenience for developing enterprise-scale solutions. Nx allows a large project to be broken down into different applications and libraries. Each library has a public API that specifies a barrel named `index.ts`. Nx also provides path mapping for all libraries. In addition, Nx brings a linting rule that prevents bypassing the barrel and also allows other restrictions.

¹⁵<https://www.angulararchitects.io/en/aktuelles/tutorial-first-steps-with-nx-and-angular-architecture/>

¹⁶<https://www.angulararchitects.io/en/aktuelles/tutorial-first-steps-with-nx-and-angular-architecture/>

This linting rule allows enforcing a fixed frontend architecture. For example, the Nx team recommends dividing a large application vertically by subject domains and horizontally by technical library categories:



Feature libraries contain smart components that implement use cases, while UI libraries house reusable dumb components. Domain libraries encapsulate the client-side domain model and services that operate on it, and utility libraries group general utility functions.

With the linting rules mentioned, it can now be ensured that each layer may only access the layers below it. Access to other domains can also be prevented. Libraries from the *Booking* area are therefore not allowed to access libraries in *Boarding*. If you want to use certain constructs across domains, they should be placed in the shared area, for example.

If someone violates one of these rules, the linter gives instant feedback:

```
import { NavbarComponent } from '@demo/ui-shell';
(alias) class NavbarComponent
import NavbarComponent

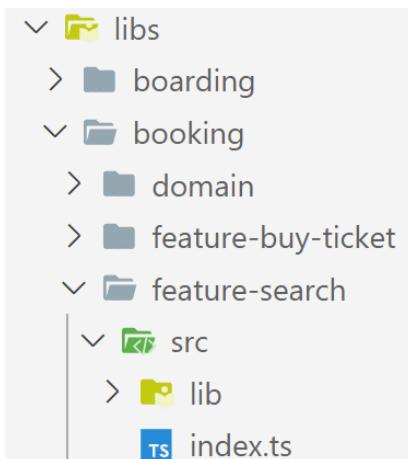
"NavbarComponent" ist deklariert, aber der zugehörige Wert
wird nie gelesen. ts(6133)

A project tagged with "domain" can only depend on libs tagged
with "util" eslint(@nrwl/nx/enforce-module-boundaries)

'NavbarComponent' is defined but never
used. eslint(@typescript-eslint/no-unused-vars)
```

Linting Rule Feedback

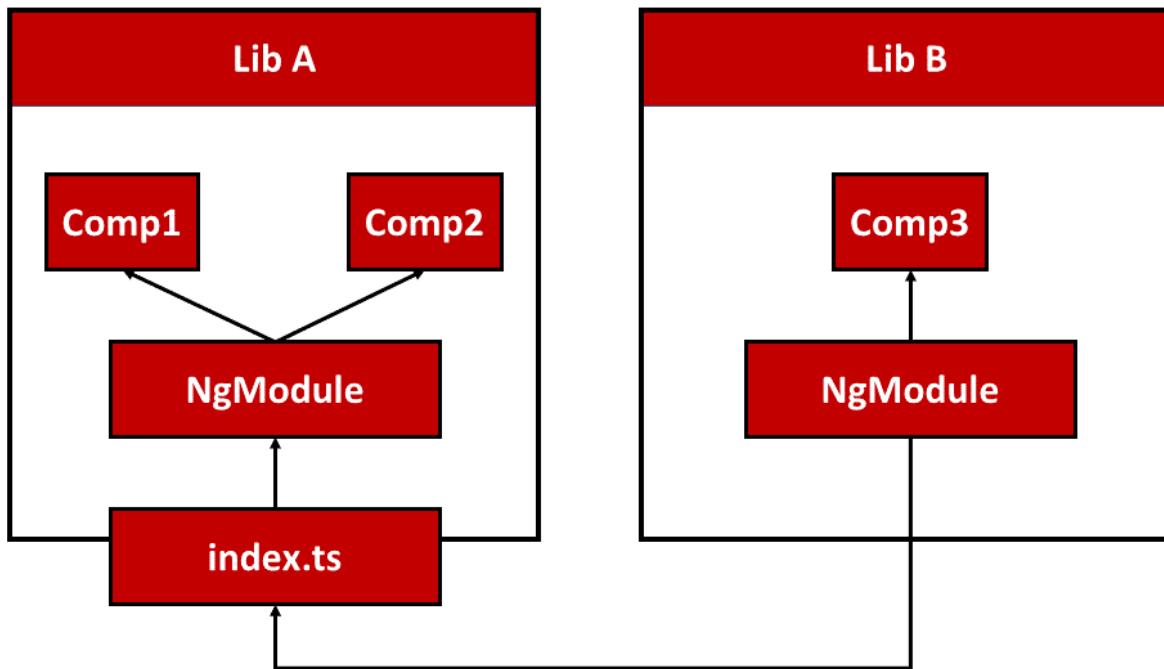
The folder structure used for this by Nx reflects the architecture matrix shown:



Structure of Nx workspace

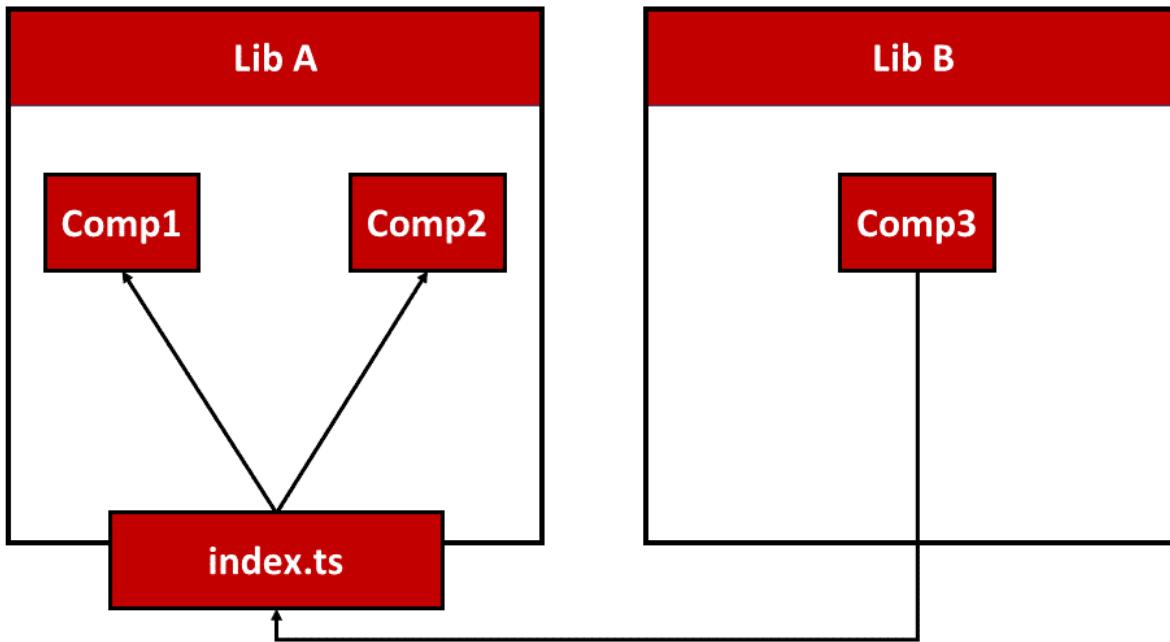
The subfolders in libs represent the domains. The libraries found in it get a prefix like `feature-` or `domain-`. These prefixes reflect the technical categories and thus the layers.

The nice thing about this fourth option is that it has long proven itself in interaction with Angular modules for structuring large solutions:



Nx libs with NgModules

Thanks to Standalone Components, the Angular modules can be omitted now:

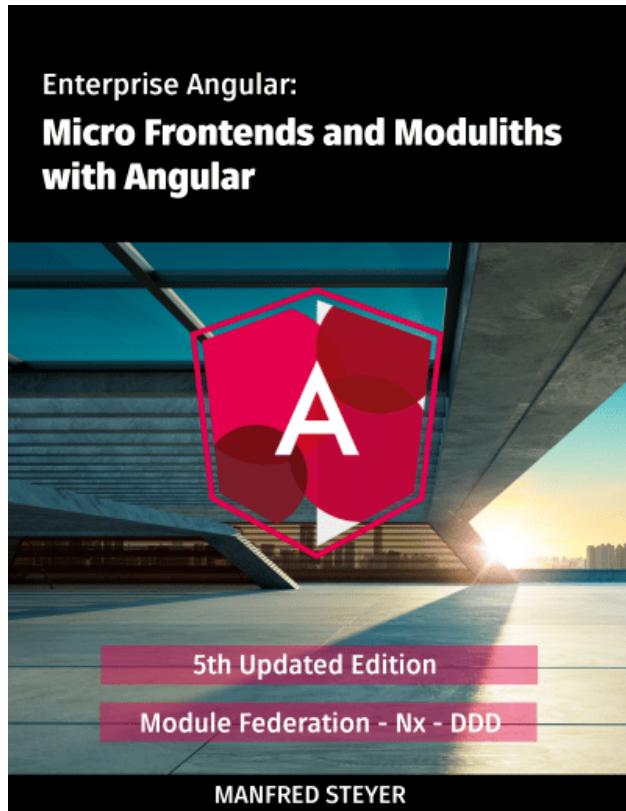


Nx libs without NgModules

In this case, only the libraries are used for structuring: their barrels group related building blocks, such as Standalone Components, and thanks to the linting rules mentioned, we can enforce our architectures.

More on Architectures with Nx

More information about architectures with Nx can be found in our free eBook (12 chapters) on Angular Architectures:



Free ebook

Feel free to download it here¹⁷ now!

¹⁷<https://www.angulararchitects.io/book>

Trainings and Consulting

Learn more about this and further architecture topics regarding Angular and huge enterprise as well as industrial solution in our [advanced Online Workshop¹⁸](#):



Advanced Angular Workshop

Save your [ticket¹⁹](#) for one of our **remote or on-site** workshops now or [request a company workshop²⁰](#) (online or In-House) for you and your team!

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

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

¹⁸<https://www.angulararchitects.io/en/angular-workshops/advanced-angular-enterprise-architecture-incl-ivy/>

¹⁹<https://www.angulararchitects.io/en/angular-workshops/advanced-angular-enterprise-architecture-incl-ivy/>

²⁰<https://www.angulararchitects.io/en/angular-workshops/>

Please find the full list with our offers here²¹.

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

For this, you can subscribe to our newsletter²² and/ or follow the book's author on Twitter²³.

²¹<https://www.angulararchitects.io/en/angular-workshops/>

²²<https://www.angulararchitects.io/subscribe/>

²³<https://twitter.com/ManfredSteyer>