Modern Angular

ANGULAR ARCHITECTS

2nd Extended Edition

Standalone, Improved APIs, Signals

MANFRED STEYER

Modern Angular

Manfred Steyer

Contents

Intro

At the beginning of 2023, Sarah Drasner, who as Director of Engineering at Google also heads the Angular team, coined the term Angular Renaissance. This term means a renewal of the framework that has supported us in the development of modern JavaScript solutions for seven years now.

This renewal is incremental and backwards compatible and takes current trends from the world of front-end frameworks into account. This is primarily about developer experience and performance. Standalone Components and Signals are two well-known features that have already emerged as part of this movement.

In this book, I discuss the innovations that come with the Angular Renaissance using several examples.

Structure

This book is subdivided into 14 chapters grouped to four parts discussing different aspects of modern Angular.

Part 1: Standalone Components

The first part discusses Standalone Components, how they play together with traditional NgModule-based code, and what they mean for your architecture.

Chapters in part 1:

- Standalone Components: Mental Model & Compatibility
- Architecture with Standalone Components

Part 2: Improved APIs

This part goes in depth with the new Standalone APIs – renewed Angular APIs for routing, lazy loading, http access, web components, and testing.

Chapters in part 2:

- Standalone APIs for Routing and Lazy Loading
- Angular Elements with Standalone Components
- The Refurbished HttpClient - Standalone APIs and Functional Interceptors
- Testing Angular Standalone Components
- Patterns for Custom Standalone APIs in Angular

Part 3: Preparing for and Migrating to Standalone

In this part you learn how to migrate your existing code to Standalone.

Chapters in part 3:

- How to prepare for Standalone Components?
- Automatic Migration to Standalone Components in 3 Steps

Part 4: Signals

Signals are the future of change detection in Angular. The fourth part shows how to use them in your applications.

Chapters in part 4:

- Signals in Angular: The Future of Change Detection
- Component Communication with Signals: Inputs, Two-Way Bindings, and Content/ View Queries
- Successful with Signals in Angular - 3 Effective Rules for Your Architecture

Part 5: Control Flow & Performance

The final part explains what's behind the new control flow syntax, how to improve the performance with deferable views, SSR, and hydration, and how to speed up the build with the new esbuild-based `ApplicationBuilder`.

Chapters in part 5:

- Built-in Control Flow and Deferrable Views
- esbuild and the new Application Builder

Help to Improve this Book!

Please let us know if you have any suggestions or find any mistakes. Just open an issue at or send a pull request to the book's GitHub repository[1].

[1] https://github.com/manfredsteyer/standalone-book.git

Spread the Word!

If you like this book, tell your contacts via Twitter/X[2], Facebook[3], and/or LinkedIn[4] about it.

Also, feel free to send the download link[5] to colleagues and friends via email or your company-internal chat platform like Slack or Teams.

Trainings and Consulting

If you and your team need support or trainings regarding Angular, we are happy to help with **workshops and consulting** (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
- Modern Angular Workshop
- Angular Micro Frontends Workshop
- Angular Testing Workshop (Cypress, Jest, etc.)
- Angular Performance Workshop
- Angular Design Systems Workshop (Figma, Storybook, etc.)
- Angular: Reactive Architectures (RxJS and NGRX)
- Angular Review Workshop
- Angular Upgrade Workshop

Please find the full list of our offers here[6].

[2] https://twitter.com/intent/post?text=Check%20out%20this%20free%20eBook%20about%20%23ModernAngular%20by%20%40manfredsteyer%20%0A%0Ahttps%3A%2F%2Fwww.angulararchitects.io%2Fen%2Febooks%2Fmodern-angular%2F%3Fbook

[3] https://www.facebook.com/sharer/sharer.php?u=https%3A%2F%2Fwww.angulararchitects.io%2Fen%2Febooks%2Fmodern-angular%2F%3Fbook

[4] https://www.linkedin.com/sharing/share-offsite/?url=https%3A%2F%2Fwww.angulararchitects.io%2Fen%2Febooks%2Fmodern-angular%2F%3Fbook

[5] https://www.angulararchitects.io/en/ebooks/modern-angular/?book

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

Modern Angular Workshop

Modern Angular (English)[7] | Modern Angular (German)[8]

We provide our workshops and consulting 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 at office@softwarearchitekt.at.

[7] https://www.angulararchitects.io/en/training/modern-angular-workshop/
[8] https://www.angulararchitects.io/training/modern-angular-workshop/

Standalone Components: 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.

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[9] and as an Nx workspace[10] 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

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

[9] https://github.com/manfredsteyer/standalone-example-cli
[10] https://github.com/manfredsteyer/standalone-example-nx

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:

```
@Component({
  standalone: true,
  selector: 'app-root',
  imports: [
    RouterOutlet,
    NavbarComponent,
    SidebarComponent,
  ],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
    [...]
}
```

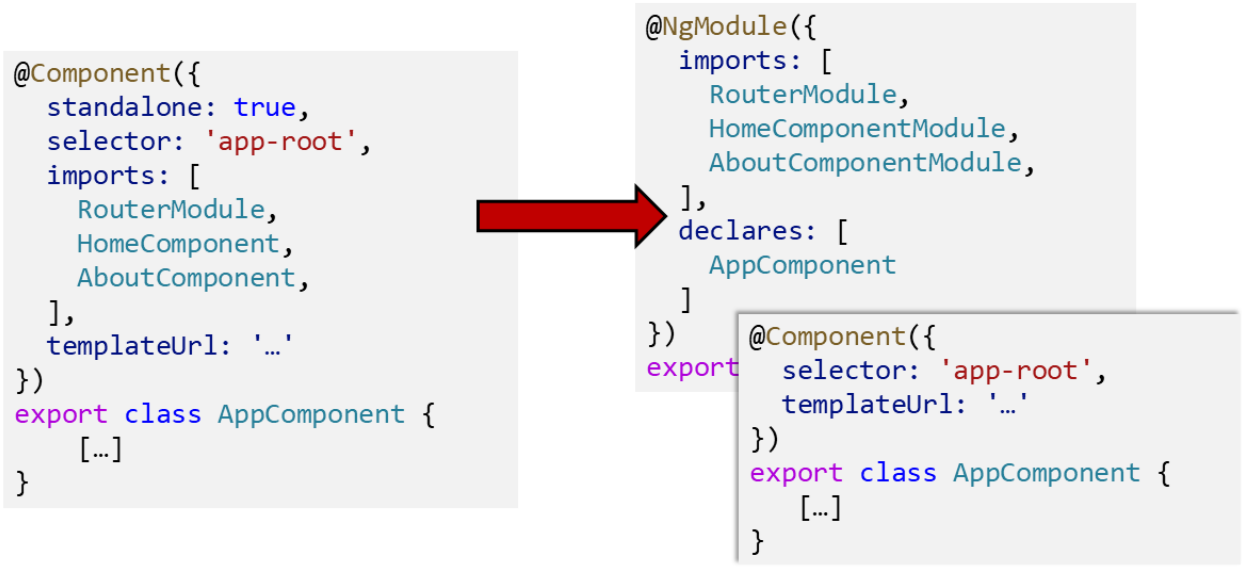
The `imports` define the compilation context: all the other building blocks the Standalone Components is allowed to use. For instance, you use it to import further Standalone Component, but also existing NgModules.

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:



Mental Model

This is similar to Lars Nielsen[11]'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:

[11] https://twitter.com/LayZeeDK

```
@Pipe ({
  standalone: true,
  name: 'city',
  pure: true
})
export class CityPipe implements PipeTransform {

  transform (value: string, format: string): string {[…]}

}
```

And here is an example for a standalone directive:

```
@Directive ({
    standalone: true,
    selector: 'input [appCity]',
    providers: […]
})
export class CityValidator implements Validator {

    [...]

}
```

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:

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

```
// main.ts

import { bootstrapApplication } from '@angular/platform-browser';
import { provideAnimations } from '@angular/platform-browser/animations';
import { AppComponent } from './app/app.component';
import { APP_ROUTES } from './app/app.routes';
import { provideRouter } from '@angular/router';
import { importProvidersFrom } from '@angular/core';

[...]

bootstrapApplication(AppComponent, {
  providers: [
    importProvidersFrom(HttpClientModule),
    provideRouter(APP_ROUTES),
    provideAnimations(),
    importProvidersFrom(TicketsModule),
    importProvidersFrom(LayoutModule),
  ]
});
```

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 `AppModule` 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:

```
import { Component, OnInit } from '@angular/core';
import { TicketsModule } from '../tickets/tickets.module';

@Component({
    selector: 'app-next-flight',
    standalone: true,
    imports: [
        // Existing NgModule imported
        // in this standalone component
        TicketsModule
    ],
    [...]
})
export class NextFlightComponent implements OnInit {
    [...]
}
```

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

```
@NgModule({
  imports: [
    CommonModule,

    // Imported Standalone Component:
    FlightCardComponent,
    [...]
  ],
  declarations: [
    MyTicketsComponent
  ],
  [...]
})
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:

```
import {
    AsyncPipe,
    JsonPipe,
    NgForOf,
    NgIf
} from "@angular/common";

[...]

@Component({
  standalone: true,
  imports: [
    // CommonModule,
    NgIf,
    NgForOf,
    AsyncPipe,
    JsonPipe,

    FormsModule,
    FlightCardComponent,
    CityValidator,
  ],
  selector: 'flight-search',
  templateUrl: './flight-search.component.html'
})
export class FlightSearchComponent implements OnInit {
    [...]
}
```

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`.

Conclusion

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.

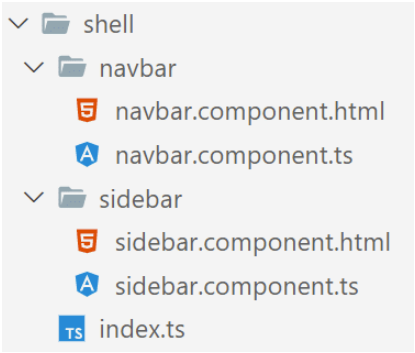
> The **source code** for this can be found in the form of a traditional Angular CLI workspace[12] and as an Nx workspace[13] that uses libraries as a replacement for NgModules.

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:

[12] https://github.com/manfredsteyer/standalone-example-cli
[13] https://github.com/manfredsteyer/standalone-example-nx

```
export { NavbarComponent } from './navbar/navbar.component';
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`:

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

@Component({
    standalone: true,
    selector: 'app-root',
    imports: [
        RouterOutlet,

        NavbarComponent,
        SidebarComponent,
    ],
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css']
})
export class AppComponent {
    [...]
}
```

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:

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

@Component({
    standalone: true,
    selector: 'app-root',
    imports: [
        RouterOutlet,

        NavbarComponent,
        SidebarComponent,
    ],
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css']
})
export class AppComponent {
    [...]
}
```

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 provided by your IDE will add the needed entries anyway. 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:

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

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

export const SHELL = [
    NavbarComponent,
    SidebarComponent
];
```

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:

```
import { SHELL } from './shell';

[...]

@Component({
    standalone: true,
    selector: 'app-root',
    imports: [
        RouterOutlet,

        // NavbarComponent,
        // SidebarComponent,
        SHELL
    ],
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css']
})
export class AppComponent {
    [...]
}
```

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:

```
import { SHELL } from '../../../../shell';

@Component ({
    standalone: true,
    selector: 'app-my-cmp',
    imports: [
        SHELL,
        [...]
    ]
})
export class MyComponent {
}
```

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):

```
"paths": {
 "@demo/shell": ["src/app/shell/index.ts"],
  [...]
}
```

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

```
// Import via mapped path:
import { SHELL } from '@demo/shell';

@Component ({
    standalone: true,
    selector: 'app-root',
    imports: [
        SHELL,
        [...]
    ]
})
export class MyComponent {
}
```

Workspace Libraries and Nx

These path mappings can of course be created manually. However, it is a little easier with the CLI extension Nx[14] which automatically generates such path mappings for each library created within a

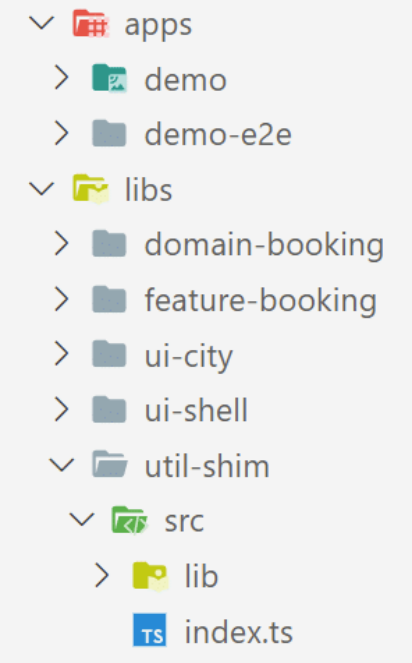
[14] https://nx.dev/

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

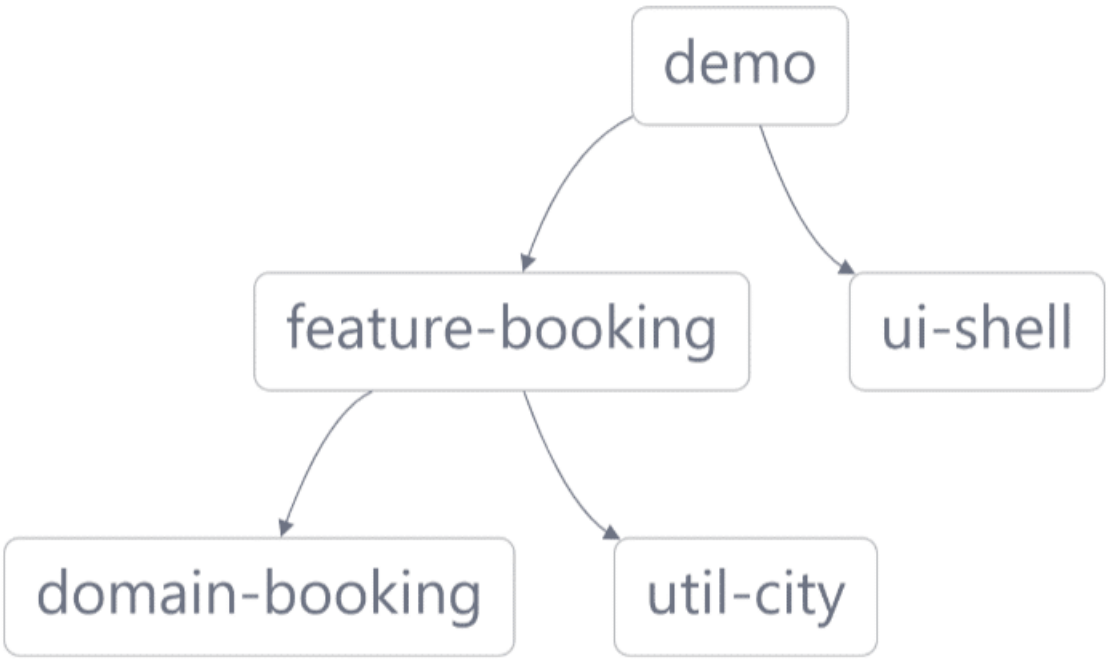
Each library receives a barrel that reflects the public API. The prefixes in the library names reflect a categorization recommended by the Nx team. For example, feature libraries contain smart components that know the use cases, while UI libraries contain reusable dumb components. The domain library 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. Your find the Source Code at GitHub[15].

[15] https://github.com/manfredsteyer/demo-nx-standalone

Besides enforcing module boundaries, Nx also comes with several additional important features: It allows for an incremental CI/CD that only rebuilds and retests parts of the monorepo that have been actually affected by a code change. Also, together with the Nx Cloud it allows to automatically parallelize the whole CI/CD process. Also, it comes with integrations into several useful tools like Storybook, Cypress, or Playwright.

Module Boundaries with Sheriff

Similar to Nx, the open source project Sheriff[16] also allows to enforce module boundaries. However, instead of using libraries for defining modules, it just goes with folders. This makes the application structure more lightweight.

Technically, Sheriff is used as via an eslint plugin. It works with traditional Angular CLI projects but also with Nx. We often combine it with Nx to get the best of both worlds: Incremental CI/CD provided by Nx and lightweight folder-based Module boundaries provided by Sheriff.

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.

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

Standalone APIs for 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 workspace[17] and as an Nx workspace[18] 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, {
```

[17] https://github.com/manfredsteyer/standalone-example-cli
[18] https://github.com/manfredsteyer/standalone-example-nx

```
  providers: [
    importProvidersFrom(HttpClientModule),
    provideRouter(APP_ROUTES,
      withPreloading(PreloadAllModules),
      withDebugTracing(),
    ),

    [...]

    importProvidersFrom(TicketsModule),
    provideAnimations(),
    importProvidersFrom(LayoutModule),
  ]
});
```

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:

```
@Component({
  standalone: true,
  selector: 'app-root',
  imports: [
    // Just import the RouterModule:
    // RouterModule,

    // Better: Just import what you need:
    RouterOutlet,
    RouterLinkWithHref, // Angular 14
    // RouterLink // Angular 15+

    NavbarComponent,
    SidebarComponent,
  ],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
    [...]
}
```

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:

```
// app.routes.ts

import { Routes } from '@angular/router';
import { HomeComponent } from './home/home.component';

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

    // Option 1: Lazy Loading another Routing Config
    {
        path: 'flight-booking',
        loadChildren: () =>
            import('./booking/flight-booking.routes')
                .then(m => m.FLIGHT_BOOKING_ROUTES)
    },

    // Option 2: Directly Lazy Loading a Standalone Component
    {
        path: 'next-flight',
        loadComponent: () =>
            import('./next-flight/next-flight.component')
                .then(m => m.NextFlightComponent)
    },
    [...]
];
```

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:

```
// booking/flight-booking.routes.ts

export const FLIGHT_BOOKING_ROUTES: Routes = [{
    path: '',
    component: FlightBookingComponent,
    providers: [
        provideBookingDomain(config)
    ],
    children: [
        {
            path: '',
            pathMatch: 'full',
            redirectTo: 'flight-search'
        },
        {
            path: 'flight-search',
            component: FlightSearchComponent
        },
        {
            path: 'passenger-search',
            component: PassengerSearchComponent
        },
        {
            path: 'flight-edit/:id',
            component: FlightEditComponent
        }
    ]
}];
```

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:

```
import { bootstrapApplication } from '@angular/platform-browser';

import { provideStore } from '@ngrx/store';
import { provideEffects } from '@ngrx/effects';
import { provideStoreDevtools } from '@ngrx/store-devtools';

import { reducer } from './app/+state';

[...]

bootstrapApplication(AppComponent, {
  providers: [
    importProvidersFrom(HttpClientModule),
    provideRouter(APP_ROUTES,
      withPreloading(PreloadAllModules),
      withDebugTracing(),
    ),

    // Setup NGRX:
    provideStore(reducer),
```

```
    provideEffects([]),
    provideStoreDevtools(),

    importProvidersFrom(TicketsModule),
    provideAnimations(),
    importProvidersFrom(LayoutModule),
  ]
});
```

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:

```
import { provideEffects } from "@ngrx/effects";
import { provideState } from "@ngrx/store";

export const FLIGHT_BOOKING_ROUTES: Routes = [{
    path: '',
    component: FlightBookingComponent,
    providers: [
        provideState(bookingFeature),
        provideEffects([BookingEffects])
    ],
    children: [
        {
            path: 'flight-search',
            component: FlightSearchComponent
        },
        {
            path: 'passenger-search',
            component: PassengerSearchComponent
        },
        {
            path: 'flight-edit/:id',
            component: FlightEditComponent
        }
    ]
}];
```

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`:

```
export const FLIGHT_BOOKING_ROUTES: Routes = [{
    path: '',
    component: FlightBookingComponent,
    providers: [
        importProvidersFrom(StoreModule.forFeature(bookingFeature)),
        importProvidersFrom(EffectsModule.forFeature([BookingEffects])),
        {
            provide: ENVIRONMENT_INITIALIZER,
            multi: true,
            useValue: () => inject(InitService).init()
        }
    ],
    children: [
        [...]
    ]
}
```

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.

Component Input Bindings

The router has also received a few nice roundings. For example, it can now be instructed to pass routing parameters directly to inputs of the respective component. For example, if a route is called with `;q=Graz`, the router assigns the value `Graz` to the input with the name `q`:

```
@Input ( ) q = '' ;
```

Retrieving the parameter values via the `ActivatedRoute` service is no longer necessary. This behavior applies to parameters in the `data` object, in the query string, as well as to the matrix parameters

that are usual in Angular. In the event of a conflict, this order also applies, e.g. if present, the value is taken from the `data` object, otherwise the query string is checked and then the matrix parameters. In order not to disrupt existing code, this option must be explicitly activated. For this, the `withComponentInputBinding` function is used when calling `provideRouter`:

```
provideRouter(
  APP_ROUTES,
  withComponentInputBinding()
),
```

In addition, the router now has a `lastSuccessfulNavigation` property that provides information about the current route:

```
router = inject(Router);
[…]
console.log(
  'lastSuccessfullNavigation',
  this.router.lastSuccessfulNavigation
);
```

Conclusion

The streamlined router API removes unnecessary indirections for lazy loading: Instead of pointing to a lazy NgModule, a routing configuration now directly points to another lazy routing configuration. Providers we used to register with lazy NgModules, e.g. providers for a feature slice, are directly added to the respective route and can also be used in every child route.

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 Code[19]

Providing a Standalone Component

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

```
import { Component, EventEmitter, Input, Output, ViewEncapsulation } from '@angular/\
core';
import { CommonModule } from '@angular/common';

@Component({
  selector: 'app-toggle',
  standalone: true,
  imports: [],
  template: `
    <div class="toggle" [class.active]="active" (click)="toggle()">
      <slot>Toggle!</slot>
    </div>
  `,
  styles: [`
    .toggle {
      padding:10px;
      border: solid black 1px;
      cursor: pointer;
      display: inline
    }

    .active {
      background-color: lightsteelblue;
    }
```

[19] 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:

```
// main.ts

import { createCustomElement } from '@angular/elements';
import { createApplication } from '@angular/platform-browser';
import { ToggleComponent } from './app/toggle/toggle.component';

(async () => {

  const app = await createApplication({
    providers: [
      /* your global providers here */
    ],
  });

  const toogleElement = createCustomElement(ToggleComponent, {
    injector: app.injector,
  });

  customElements.define('my-toggle', toogleElement);

})();
```

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 its 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:

```
const element1 = createCustomElement(ThisComponent, {
    injector: app.injector,
});

const element2 = createCustomElement(ThatComponent, {
    injector: app.injector,
});
```

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

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

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`:

```
<h1>Standalone Angular Element Demo</h1>
<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:

```
<script>
  const myToggle = document.getElementById('myToggle');

  myToggle.addEventListener('change', (event) => {
    console.log('active', event.detail);
  });

  setTimeout(() => {
    myToggle.active = true;
  }, 3000);
</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:

```
import { Component, CUSTOM_ELEMENTS_SCHEMA } from '@angular/core';

@Component({
  selector: 'app-root',
  standalone: true,
  schemas: [CUSTOM_ELEMENTS_SCHEMA],
  template: `
    <h2>Root Component</h2>
    <my-toggle
        [active]="active"
        (change)="change($event)">
        Hello!
    </my-toggle>
  `,
})
export class AppComponent {
    active = false;
    change(event: Event) {
        const customEvent = event as CustomEvent<boolean>;
        console.log('active', customEvent.detail);
    }
}
```

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:

```
// main.ts

[...]
// Register web components ...
[...]

app.injector.get(NgZone).run(() => {
  app.bootstrap(AppComponent);
});
```

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

```
<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 traditional webpack-based builder emits several bundles, e. g. a main bundle and a runtime bundle, the new esbuild-based `ApplicationBuilder` (see chapter *esbuild and the new Application Builder*) just gives us one bundle for our source code and another one for the polyfills.

The resulting bundles look like this:

```
     948 favicon.ico
     703 index.html
 100 177 main.43BPAPVS.js
  33 916 polyfills.M7XCYQVG.js
       0 styles.VFXLKGBH.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.

Conclusion

As a by-product of Standalone Components, Angular provides a streamlined way for using Angular Elements: We start with creating an `ApplicationRef` to get an `Injector`. Alongside a Standalone Component, we pass this injector to Angular Elements. The result is a Web Component we can register with the browser.

The Refurbished HttpClient - Standalone APIs and Functional Interceptors

Without any doubt, the `HttpClient` is one of the best-known services included in Angular. For version 15, the Angular team has now adapted it for the new standalone components. On this occasion, the interceptor concept was also revised.

In this chapter, I will describe these innovations.

 Source Code[20]

Standalone APIs for HttpClient

Beginning with version 15, the `HttpClient` can be set up without any reference to the `HttpClientModule`. Instead, we can use `provideHttpClient` when bootstrapping our application:

```
import { provideHttpClient, withInterceptors } from "@angular/common/http";

[...]

bootstrapApplication(AppComponent, {
        providers: [
                provideHttpClient(
                        withInterceptors([authInterceptor]),
                ),
        ]
});
```

This new function also enables optional features of the `HttpClient`. Each feature has its own function. For example, the `withInterceptors` function enables support for Http Interceptors.

The combination of a `provideXYZ` function and several optional `withXYZ` functions is not chosen arbitrarily here but corresponds to a pattern that the Angular team generally provides for standalone APIs. Application developers must therefore be on the lookout for functions that start with `provide` or `with` when setting up a new library.

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

Also, this pattern leads to a very pleasant side effect: libraries become more tree-shakable. This is because a static source code analysis makes it very easy to find out whether the application ever calls a function. In the case of methods, this is not so easy due to the possibility of polymorphic use of the underlying objects.

Functional Interceptors

When introducing standalone APIs, the Angular team also took the opportunity and revised the `HttpClient` a bit. One result of this are the new functional interceptors. They allow interceptors to be expressed as simple functions. A separate service that implements a predefined interface is no longer necessary:

```
import { HttpInterceptorFn } from "@angular/common/http";
import { tap } from "rxjs";

export const authInterceptor: HttpInterceptorFn = (req, next) => {

        console.log('request', req.method, req.url);
        console.log('authInterceptor')

        if (req.url.startsWith('https://demo.angulararchitects.io/api/')) {

                // Setting a dummy token for demonstration

                constheaders = req.headers.set('Authorization', 'Bearer Auth-1234567');

                req = req.clone({
                        headers
                });

        }

        return next(req).pipes(
                tap(resp => console.log('response', resp))
        );

}
```

The interceptor shown adds an example security token to HTTP calls that are directed to specific URLs. Except that the interceptor is now a function of type `HttpInterceptorFn`, the basic functionality of this concept has not changed. As shown above, functional interceptors can be set up using `withInterceptors` when calling `provideHttpClient` .

Interceptors and Lazy Loading

Interceptors in lazy modules have always led to confusion: As soon as a lazy module introduces its own interceptors, those of outer scopes – e.g. the root scope – are no longer triggered.

Even if modules with standalone components and APIs are a thing of the past, the basic problem remains, especially since (lazy) route configurations can now set up their own services:

```
export const FLIGHT_BOOKING_ROUTES: Routes = [{
        paths: '',
        component: FlightBookingComponent,
        providers: [
                MyService,
                provideState(bookingFeature),
                provideEffects([BookingEffects])
                provideHttpClient(
                        withInterceptors([bookingInterceptor]),
                        withRequestsMadeViaParent(),
                ),
        ],
}];
```

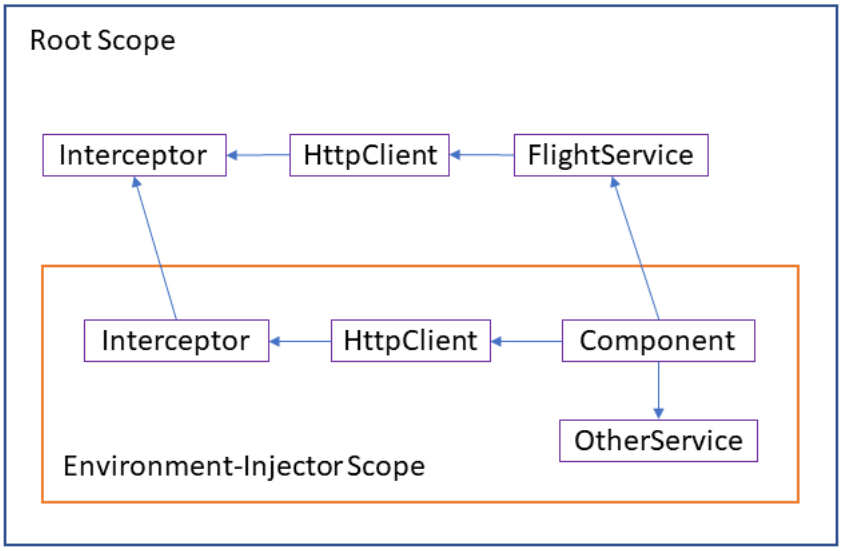
These services correspond to those the application previously registered in lazy modules. Technically, Angular introduces its own injector whenever such a `providers` array is available. This so-called environment injector defines a scope for the current route and its child routes.

The new `provideHttpClient` function can also be used in this `providers` array to register interceptors for the respective lazy part of the application. By default, the previously discussed rule applies: If there are interceptors in the current environment injector, Angular ignores the interceptors in outer scopes.

Exactly this behavior can be changed with `withRequestsMadeViaParent`: This method causes Angular to also trigger interceptors in outer scopes.

Pitfall with withRequestsMadeViaParent

The discussed `withRequestsMadeViaParent` function comes with a non-obvious pitfall: a root-scope service is unaware of inner scope and the interceptors registered there. It always accesses the `HttpClient` in the root scope and therefore only the interceptors set up there are executed:



Interceptors in multiple scopes

To solve this problem, the application could also register the outer service in the `providers` array of the route configuration and thus in the inner scope.

In general, however, it seems to be very difficult to keep track of such constellations. Therefore, it might make sense to do without interceptors in inner scopes altogether. As an alternative, a very generic interceptor in the root scope could be used. Such an interceptor may even load additional logic with a dynamic `import` from lazy applications parts.

Legacy Interceptors and Other Features

While the new functional interceptors are very charming, applications can still take advantage of the original class-based interceptors. This option can be enabled using the `withLegacyInterceptors feature`. Then, the class-based interceptors are to be registered as usual via a multi-provider:

```
bootstrapApplication(AppComponent, {
        providers: [
                provideHttpClient(
                        withInterceptors([authInterceptor]),
                        withLegacyInterceptors(),
                ),
                {
                        provide: HTTP_INTERCEPTORS,
                        useClass: LegacyInterceptor,
                        multiple: true,
```

```
                },
        ]
});
```

Further Features

The `HttpClient` has some additional features that can also be activated using `with-`functions: `withJsonpSupport`, for example, activates support for JSONP, and `withXsrfConfiguration` configures details on the use of XSRF tokens. If the application does not call `withXsrfConfiguration`, default settings are used. However, to completely disable the use of XSRF tokens, call `withNoXsrfProtection`.

Conclusion

The revised `HttpClient` now wonderfully harmonizes with standalone components and associated concepts such as environment injectors. The Angular team also took the opportunity to revise the interceptors: They can now be implemented in the form of simple functions. In addition, it is now also possible to consider interceptors in outer scopes.

Testing Angular Standalone Components

With Standalone Components, Angular becomes a lot more lightweight: NgModules are optional and hence we can work with lesser indirections. To make this possible, components now refer directly to their dependencies: other components, but also directives and pipes. There are so-called Standalone APIs for configuring services such as the `HttpClient`.

Additional Standalone APIs provide mocks for test automation. Here, I'm going to present these APIs. For this, I focus on on-board tools supplied with Angular. The examples[21] used can be found here[22]

If you don't want to use the on-board resources alone, you will find the same examples based on the new Cypress Component Test Runner and on Testing Library in the `third-party-testing` branch.

Test Setup

Even though Standalone Components make modules optional, the `TestBed` still comes with a testing module. It takes care of the test setup and provides all components, directives, pipes, and services for the test:

```
1  import { provideHttpClient } from '@angular/common/http';
2  import { HttpTestingController, provideHttpClientTesting }
3      from '@angular/common/http/testing';
4
5  […]
6
7  describe('FlightSearchComponent', () => {
8    let component: FlightSearchComponent;
9    let fixture: ComponentFixture<FlightSearchComponent>;
10   beforeEach(async () => {
11
12     await TestBed.configureTestingModule({
13       imports: [ FlightSearchComponent ],
14       providers: [
15         provideHttpClient(),
```

[21] https://github.com/manfredsteyer/standalone-example-cli.git
[22] https://github.com/manfredsteyer/standalone-example-cli.git

```
16          provideHttpClientTesting(),
17
18          provideRouter([]),
19
20          provideStore(),
21          provideState(bookingFeature),
22          provideEffects(BookingEffects),
23        ],
24      })
25      .compileComponents();
26
27      fixture = TestBed.createComponent(FlightSearchComponent);
28      component = fixture.componentInstance;
29      fixture.detectChanges();
30    });
31
32    it('should search for flights', () => { […] });
33  });
```

The example shown imports the Standalone Component to be tested and provides the required services via the `providers` array. This is exactly where the mentioned Standalone APIs come into play. They provide the services for the `HttpClient`, the router and NGRX.

The `provideStore` function sets up the NGRX store, `provideState` provides a feature slice required for the test, and `provideEffects` registers an associated effect. Below we will swap out these constructs for mocks.

The `provideHttpClientTesting` method is interesting: it overrides the `HttpBackend` used behind the scenes by the `HttpClient` with an `HttpTestingBackend` that simulates HTTP calls. It should be noted that it must be called after (!) `provideHttpClient`.

It is therefore first necessary to set up the `HttpClient` by default in order to then overwrite individual details for testing. This is a pattern we will see again below when testing the router.

The HttpClient Mock

Once the `HttpClient` and `HttpTestingBackend` have been set up, the individual tests are implemented as usual: The test uses the `HttpTestingController` to find out about pending HTTP requests and to specify the HTTP responses to be simulated:

```
it('should search for flights', () => {
  component.from = 'Paris';
  component.to = 'London';
  component.search();

  const ctrl = TestBed.inject(HttpTestingController);

  const req = ctrl.expectOne('https://[…]/flight?from=Paris&to=London');
  req.flush([{}, {}, {}]); // return 3 empty objects as dummy flights

  component.flights$.subscribe(flights => {
    expect(flights.length).toBe(3);
  });

  ctrl.verify();
});
```

The test then checks whether the component processed the simulated HTTP response as intended. In the case shown, the test assumes that the component offers the received flights via its `flights` property.

At the end, the test ensures that there are no further HTTP requests that have not yet been answered. To do this, it calls the `verify` method provided by the `HttpTestingController`. If there are still open requests at this point, `verify` throws an exception that causes the test to fail.

Shallow Testing

If you test a component, all sub-components, directives, and pipes used in the template are automatically tested as well. This is undesirable, especially for unit tests that focus on a single code unit. Also, this behavior slows down test execution when there are many dependencies.

Shallow tests are used to prevent this. This means that the test setup replaces all dependencies with mocks. These mocks must have the same interface as the replaced dependencies. In the case of components, this means – among other things – that the same properties and events (inputs and outputs) must be offered, but also that the same selectors must be used.

The `TestBed` offers the `overrideComponent` method for exchanging these dependencies:

```
await TestBed.configureTestingModule([…])
  .overrideComponent(FlightSearchComponent, {
    remove: { imports: [ FlightCardComponent ] },
    add: { imports: [ FlightCardMock ] }
  })
  .compileComponents();
```

In the case shown, the `FlightSearchComponent` uses another Standalone Component in its template: the `FlightCardComponent`. Technically, this means that the `FlightCardComponent` appears in the `imports` array of `FlightSearchComponent`. For implementing a shallow Test, this entry is removed. As a replacement, the `FlightCardMock` is added. The `remove` and `add` methods take care of this task.

The `FlightSearchComponent` is thus used in the test without real dependencies. Nevertheless, the test can check whether components behave as desired. For example, the following listing checks whether the `FlightSearchComponent` sets up an element named `flight-card` for each flight found.

```
it('should display a flight-card for each found flight', () => {
  component.from = 'Paris';
  component.to = 'London';
  component.search();

  const ctrl = TestBed.inject(HttpTestingController);

  const req = ctrl.expectOne('https://[…]/flight?from=Paris&to=London');
  req.flush([{}, {}, {}]);

  fixture.detectChanges();

  const cards = fixture.debugElement.queryAll(By.css('flight-card'));
  expect(cards.length).toBe(3);
});
```

Mock Router and Store

The test setup used so far only simulated the `HttpCient`. However, there are also Standalone APIs for mocking the router and NGRX:

```
import { provideRouter } from '@angular/router';
import { provideLocationMocks } from '@angular/common/testing';

import { provideMockStore } from '@ngrx/store/testing';
import { provideMockActions } from '@ngrx/effects/testing';

[…]

describe('FlightSearchComponent (at router level)', () => {
  let component: FlightSearchComponent;
  let fixture: ComponentFixture<FlightSearchComponent>;
  let actions$ = new Subject<Action>();

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      providers: [
        provideHttpClient(),
        provideHttpClientTesting(),

        provideRouter([
          { path: 'flight-edit/:id', component: FlightEditComponent }
        ]),
        provideLocationMocks(),

        provideMockStore({
          initialState: {
            [BOOKING_FEATURE_KEY]: {
              flights: [{ id:1 }, { id:2 }, { id:3 }],
            },
          },
        }),

        provideMockActions(() => actions$),
      ],
      imports: [FlightSearchComponent],
    }).compileComponents();

    fixture = TestBed.createComponent(FlightSearchComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  […]
```

```
});
```

As with testing the `HttpClient`, the test first sets up the router in the usual way. Then, it uses `provideLocationMocks` to override a couple of internally used services, namely `Location` and `LocationStrategy`. This procedure allows the route change to be simulated in the test cases. The `MockStore` which also ships with NGRX is used instead of the traditional one. It allows the entire content of the store to be freely defined. This is done either by calling `provideMockStore` or via its `setState` method. Also, `provideMockActions` gives us the ability to swap out the `actions$` observable that NGRX effects often rely on. A test case using this setup could look like as follows:

```
it('routes to flight-card', fakeAsync(() => {

  const link = fixture.debugElement.query(By.css('a[class*=btn-default ]'))
  link.nativeElement.click();

  flush();
  fixture.detectChanges();

  const location = TestBed.inject(Location);
  expect(location.path()).toBe('/flight-edit/1;showDetails=false')

}));
```

This test assumes that the `FlightSearchComponent` displays one link per flight in the (mock)store. It simulates a click on the first link and checks whether the application then switches to the expected route. In order for Angular to process the simulated click and trigger the route change, the change detection must be running. Unfortunately, this is not automatically the case with tests. Instead, it is to be triggered with the `detectChanges` method when required. The operations involved are asynchronous. Hence, `fakeAsync` is used so that the we don't need to burdened ourselves with this. It allows pending micro-tasks to be processed synchronously using `flush`.## Testing Effects

The `MockStore` does not trigger reducers or effects. The former are just functions and can be tested in a straight forward way. Replacing `action$` is a good way to test effects. The test setup in the last section has already taken care of that. A test based on this could now use the observable `action$` to send an action to which the tested effect reacts:

```
it('load flights', () => {
  const effects = TestBed.inject(BookingEffects);
  let flights: Flight[] = [];

  effects.loadFlights$.subscribe(action => {
    flights = action.flights; // Action returned from Effect
  });

  actions$.next(loadFlights({ from: 'Paris', to: 'London' }));
    // Action sent to store to invoke Effect

  const ctrl = TestBed.inject(HttpTestingController);
  const req = ctrl.expectOne('https://[…]/flight?from=Paris&to=London');
  req.flush([{}, {}, {}]);

  expect(flights.length).toBe(3);
});
```

In the case under consideration, the effect triggers an HTTP call answered by the `HttpTestingController`. The response contains three flights, represented by three empty objects for the sake of simplicity. Finally, the test checks whether the effect provided these flights via the outbound action.

Conclusion

More and more libraries offer Standalone APIs for mocking dependencies. These either provide a mock implementation or at least overwrite services in the actual implementation to increase testability. The `TestingModule` is still used to provide the test setup. Unlike before, however, it now imports the standalone components, directives, and pipes to be tested. Their classic counterparts, on the other hand, were declared. In addition, the `TestingModule` now includes providers setup by Standalone APIs.

Patterns for Custom Standalone APIs in Angular

Together with Standalone Components, the Angular team introduced Standalone APIs. They allow for setting up libraries in a more lightweight way. Examples of libraries currently providing Standalone APIs are the `HttpClient` and the `Router`. Also, NGRX is an early adopter of this idea.

In this chapter, I present several patterns for writing custom Standalone APIs inferred from the before mentioned libraries. For each pattern, the following aspects are discussed: intentions behind the pattern, description, example implementation, examples for occurrences in the mentioned libraries, and variations for implementation details.

Most of these patterns are especially interesting for library authors. They have the potential to improve the DX for the library's consumers. On the other side, most of them might be overkill for applications.

Big thanks to Angular's Alex Rickabaugh[23] for proofreading and providing feedback!

 Source code used in examples[24]

Case Study for Patterns

For presenting the inferred patterns, a simple logger library is used. This library is as simple as possible but as complex as needed to demonstrate the implementation of the patterns:

<img src="images/logger.png" style="max-width:300px">

Each log message has a `LogLevel`, defined by an enum:

```
export enum LogLevel {
  DEBUG = 0,
  INFO = 1,
  ERROR = 2,
}
```

For the sake of simplicity, we restrict our Logger library to just three log levels.

An abstract `LoggerConfig` defines the possible configuration options:

[23] https://twitter.com/synalx
[24] https://github.com/manfredsteyer/standalone-example-cli.git

```
export abstract class LoggerConfig {
  abstract level: LogLevel;
  abstract formatter: Type<LogFormatter>;
  abstract appenders: Type<LogAppender>[];
}
```

It's an abstract class on purpose, as interfaces cannot be used as tokens for DI. A constant of this class type defines the default values for the configuration options:

```
export const defaultConfig: LoggerConfig = {
  level: LogLevel.DEBUG,
  formatter: DefaultLogFormatter,
  appenders: [DefaultLogAppender],
};
```

The `LogFormatter` is used for formatting log messages before they are published via a `LogAppender`:

```
export abstract class LogFormatter {
  abstract format(level: LogLevel, category: string, msg: string): string;
}
```

Like the `LoggerConfiguration`, the `LogFormatter` is an abstract class used as a token. The consumer of the logger lib can adjust the formatting by providing its own implementation. As an alternative, they can go with a default implementation provided by the lib:

```
@Injectable()
export class DefaultLogFormatter implements LogFormatter {
  format(level: LogLevel, category: string, msg: string): string {
    const levelString = LogLevel[level].padEnd(5);
    return `[${levelString}] ${category.toUpperCase()} ${msg}`;
  }
}
```

The `LogAppender` is another exchangeable concept responsible for appending the log message to a log:

```
export abstract class LogAppender {
  abstract append(level: LogLevel, category: string, msg: string): void;
}
```

The default implementation writes the message to the console:

```
@Injectable()
export class DefaultLogAppender implements LogAppender {
  append(level: LogLevel, category: string, msg: string): void {
    console.log(category + ' ' + msg);
  }
}
```

While there can only be one `LogFormatter`, the library supports several `LogAppenders`. For instance, a first `LogAppender` could write the message to the console while a second one could also send it to the server.

To make this possible, the individual `LogAppenders` are registered via multi providers. Hence, the Injector returns all of them within an array. As an array cannot be used as a DI token, the example uses an `InjectionToken` instead:

```
export const LOG_APPENDERS = new InjectionToken<LogAppender[]>("LOG_APPENDERS");
```

The `LoggserService` itself receives the `LoggerConfig`, the `LogFormatter`, and an array with `LogAppenders` via DI and allows to log messages for several `LogLevels`:

```
@Injectable()
export class LoggerService {
  private config = inject(LoggerConfig);
  private formatter = inject(LogFormatter);
  private appenders = inject(LOG_APPENDERS);

  log(level: LogLevel, category: string, msg: string): void {
    if (level < this.config.level) {
      return;
    }
    const formatted = this.formatter.format(level, category, msg);
    for (const a of this.appenders) {
      a.append(level, category, formatted);
    }
  }

  error(category: string, msg: string): void {
    this.log(LogLevel.ERROR, category, msg);
  }

  info(category: string, msg: string): void {
    this.log(LogLevel.INFO, category, msg);
  }
```

```
24
25    debug(category: string, msg: string): void {
26      this.log(LogLevel.DEBUG, category, msg);
27    }
28  }
```

The Golden Rule

Before I start with presenting the inferred patterns, I want to stress out what I call the golden rule for providing services:

> Whenever possible, use `@Injectable({providedIn: 'root'})`!

Especially in application code but in several situations in libraries, this is what you want to have: It's easy, tree-shakable, and even works with lazy loading. The latter aspect is less a merit of Angular than the underlying bundler: Everything that is just needed in a lazy bundle is put there.

Pattern: Provider Factory

Intentions

- Providing services for a reusable lib
- Configuring a reusable lib
- Exchanging defined implementation details

Description

A Provider Factory is a function returning an array with providers for a given library. This Array is cross-casted into Angular's `EnvironmentProviders` type to make sure the providers can only be used in an environment scope – first and foremost, the root scope and scopes introduced with lazy routing configurations.

Angular and NGRX place such functions in files called `provider.ts`.

Example

The following Provider Function `provideLogger` takes a partial `LoggerConfiguration` and uses it to create some providers:

```
export function provideLogger(
  config: Partial<LoggerConfig>
): EnvironmentProviders {
  // using default values for missing properties
  const merged = { ...defaultConfig, ...config };

  return makeEnvironmentProviders([
    {
      provide: LoggerConfig,
      useValue: merged,
    },
    {
      provide: LogFormatter,
      useClass: merged.formatter,
    },
    merged.appenders.map((a) => ({
      provide: LOG_APPENDERS,
      useClass: a,
      multi: true,
    })),
  ]);
}
```

Missing configuration values are taken from the default configuration. Angular's `makeEnvironmentProviders` wraps the `Provider` array in an instance of `EnvironmentProviders`.

This function allows the consuming application to setup the logger during bootstrapping like other libraries, e. g. the `HttpClient` or the `Router`:

```
bootstrapApplication(AppComponent, {
  providers: [

    provideHttpClient(),

    provideRouter(APP_ROUTES),

    [...]

    // Setting up the Logger:
    provideLogger(loggerConfig),
  ]
}
```

Occurrences and Variations

- This is a usual pattern used in all examined libraries
- The Provider Factories for the `Router` and `HttpClient` have a second optional parameter that takes additional features (see Pattern *Feature*, below).
- Instead of passing in the concrete service implementation, e. g. LogFormatter, NGRX allows taking either a token or the concrete object for reducers.
- The `HttpClient` takes an array with functional interceptors via a `with` function (see Pattern *Feature*, below). These functions are also registered as services.

Pattern: Feature

Intentions

- Activating and configuring optional features
- Making these features tree-shakable
- Providing the underlying services via the current environment scope

Description

The Provider Factory takes an optional array with a feature object. Each feature object has an identifier called `kind` and a `providers` array. The `kind` property allows for validating the combination of passed features. For instance, there might be mutually exclusive features like configuring XSRF token handling and disabling XSRF token handling for the `HttpClient`.

Example

Our example uses a color feature that allows for displaying messages of different `LoggerLevels` in different colors:

<img src="images/feature.png" style="max-width:300px">

For categorizing features, an enum is used:

```
export enum LoggerFeatureKind {
    COLOR,
    OTHER_FEATURE,
    ADDITIONAL_FEATURE
}
```

Each feature is represented by an object of `LoggerFeature`:

```
export interface LoggerFeature {
  kind: LoggerFeatureKind;
  providers: Provider[];
}
```

For providing the color feature, a factory function following the naming pattern with*Feature* is introduced:

```
export function withColor(config?: Partial<ColorConfig>): LoggerFeature {
  const internal = { ...defaultColorConfig, ...config };

  return {
    kind: LoggerFeatureKind.COLOR,
    providers: [
      {
        provide: ColorConfig,
        useValue: internal,
      },
      {
        provide: ColorService,
        useClass: DefaultColorService,
      },
    ],
  };
}
```

The Provider Factory takes several features via an optional second parameter defined as a rest array:

```
export function provideLogger(
  config: Partial<LoggerConfig>,
  ...features: LoggerFeature[]
): EnvironmentProviders {
  const merged = { ...defaultConfig, ...config };

  // Inspecting passed features
  const colorFeatures =
    features?.filter((f) => f.kind === LoggerFeatureKind.COLOR)?.length ?? 0;

  // Validating passed features
  if (colorFeatures > 1) {
    throw new Error("Only one color feature allowed for logger!");
  }
```

```
15
16    return makeEnvironmentProviders([
17      {
18        provide: LoggerConfig,
19        useValue: merged,
20      },
21      {
22        provide: LogFormatter,
23        useClass: merged.formatter,
24      },
25      merged.appenders.map((a) => ({
26        provide: LOG_APPENDERS,
27        useClass: a,
28        multi: true,
29      })),
30
31      // Providing services for the features
32      features?.map((f) => f.providers),
33    ]);
34  }
```

The `kind` property of the feature is used to examine and validate the passed features. If everything is fine, the providers found in the feature are put into the returned `EnvironmentProviders` object.

The `DefaultLogAppender` gets hold of the `ColorService` provided by the color feature via dependency injection:

```
1  export class DefaultLogAppender implements LogAppender {
2    colorService = inject(ColorService, { optional: true });
3
4    append(level: LogLevel, category: string, msg: string): void {
5      if (this.colorService) {
6        msg = this.colorService.apply(level, msg);
7      }
8      console.log(msg);
9    }
10 }
```

As features are optional, the `DefaultLogAppender` passes `optional: true` to `inject`. Otherwise, we would get an exception if the feature is not applied. Also, the `DefaultLogAppender` needs to check for `null` values.

Occurrences and Variations

- The `Router` uses it, e. g. for configuring preloading or for activating debug tracing.
- The `HttpClient` uses it, e. g. for providing interceptors, configuring JSONP, and configuring/disabling the XSRF token handling
- Both, the `Router` and the `HttpClient`, combine the possible features to a union type (e.g. `export type AllowedFeatures = ThisFeature | ThatFeature`). This helps IDEs to propose built-in features.
- Some implementations inject the current `Injector` and use it to find out which features have been configured. This is an imperative alternative to using `optional: true`.
- Angular's feature implementations prefix the properties `kind` and `providers` with `ɵ` and hence declare them as internal properties.

Pattern: Configuration Provider Factory

Intentions

- Configuring existing services
- Providing additional services and registering them with existing services
- Extending the behavior of a service from within a nested environment scope

Description

Configuration Provider Factories extend the behavior of an existing service. They may provide additional services and use an `ENVIRONMENT_INITIALIZER` to get hold of instances of both the provided services as well as the existing services to extend.

Example

Let's assume an extended version of our `LoggerService` that allows for defining an additional `LogAppender` for each log category:

```
@Injectable()
export class LoggerService {

    private appenders = inject(LOG_APPENDERS);
    private formatter = inject(LogFormatter);
    private config = inject(LoggerConfig);
    [...]

    // Additional LogAppender per log category
    readonly categories: Record<string, LogAppender> = {};

    log(level: LogLevel, category: string, msg: string): void {

        if (level < this.config.level) {
            return;
        }

        const formatted = this.formatter.format(level, category, msg);

        // Lookup appender for this very category and use
        // it, if there is one:
        const catAppender = this.categories[category];

        if (catAppender) {
            catAppender.append(level, category, formatted);
        }

        // Also, use default appenders:
        for (const a of this.appenders) {
            a.append(level, category, formatted);
        }

    }

    [...]
}
```

To configurate a `LogAppender` for a category, we can introduce another Provider Factory:

```
export function provideCategory(
  category: string,
  appender: Type<LogAppender>
): EnvironmentProviders {
  // Internal/ Local token for registering the service
  // and retrieving the resolved service instance
  // immediately after.
  const appenderToken = new InjectionToken<LogAppender>("APPENDER_" + category);

  return makeEnvironmentProviders([
    {
      provide: appenderToken,
      useClass: appender,
    },
    {
      provide: ENVIRONMENT_INITIALIZER,
      multi: true,
      useValue: () => {
        const appender = inject(appenderToken);
        const logger = inject(LoggerService);

        logger.categories[category] = appender;
      },
    },
  ]);
}
```

This factory creates a provider for the `LogAppender` class. However, we don't need the class but rather an instance of it. Also, we need the `Injector` to resolve this instance's dependencies. Both happen when retrieving a `LogAppender` via inject.

Precisely this is done by the `ENVIRONMENT_INITIALIZER`, which is multi provider bound to the token `ENVIRONMENT_INITIALIZER` and pointing to a function. It gets the `LogAppender` injected but also the `LoggerService`. Then, the `LogAppender` is registered with the logger.

This allows for extending the existing `LoggerService` that can even come from a parent scope. For instance, the following example assumes the `LoggerService` in the root scope while the additional log category is setup in the scope of a lazy route:

```
export const FLIGHT_BOOKING_ROUTES: Routes = [
  {
    path: '',
    component: FlightBookingComponent,

    // Providers for this route and child routes
    // Using the providers array sets up a new
    // environment injector for this part of the
    // application.
    providers: [
      // Setting up an NGRX feature slice
      provideState(bookingFeature),
      provideEffects([BookingEffects]),

      // Provide LogAppender for logger category
      provideCategory('booking', DefaultLogAppender),
    ],
    children: [
      {
        path: 'flight-search',
        component: FlightSearchComponent,
      },
      [...]
    ],
  },
];
```

Occurrences and Variations

- `@ngrx/store` uses this pattern to register feature slices
- `@ngrx/effects` uses this pattern, to wire-up effects provided by a feature
- The feature `withDebugTracing` uses this pattern to subscribe to the `Router`'s `events` Observable.

Pattern: NgModule Bridge

Intentions

- Not breaking existing code using `NgModules` when switching to Standalone APIs.
- Allowing such application parts to set up `EnvironmentProviders` that come from a Provider Factory.

Remarks: For new code, this pattern seems to be overkill, because the Provider Factory can be directly called for the consuming (legacy) NgModules.

Description

The NgModule Bridge is a NgModule deriving (some of) its providers via a Provider Factory (see pattern *Provider Factory*). To give the caller more control over the provided services, static methods like `forRoot` can be used. These methods can take a configuration object.

Example

The following `NgModules` allows for setting up the Logger in a traditional way:

```
@NgModule({
  imports: [/* your imports here */],
  exports: [/* your exports here */],
  declarations: [/* your delarations here */],
  providers: [/* providers, you _always_ want to get, here */],
})
export class LoggerModule {
  static forRoot(config = defaultConfig): ModuleWithProviders<LoggerModule> {
    return {
      ngModule: LoggerModule,
      providers: [
        provideLogger(config)
      ],
    };
  }

  static forCategory(
    category: string,
    appender: Type<LogAppender>
  ): ModuleWithProviders<LoggerModule> {
    return {
      ngModule: LoggerModule,
      providers: [
        provideCategory(category, appender)
      ],
    };
  }
}
```

To avoid reimplementing the Provider Factories, the Module's methods delegate to them. As using such methods is usual when working with NgModules, consumers can leverage existing knowledge and conventions.

Occurrences and Variations

- All the examined libraries use this pattern to stay backwards compatible

Pattern: Service Chain

Intentions

- Making a service delegating to another instance of itself in a parent scope.

Description

When the same service is placed in several nested environment injectors, we normally only get the service instance of the current scope. Hence, a call to the service in a nested scope is not respected in the parent scope. To work around this, a service can look up an instance of itself in the parent scope and delegate to it.

Example

Let's assume we provide the logger library again for a lazy route:

```
export const FLIGHT_BOOKING_ROUTES: Routes = [
  {
    path: '',
    component: FlightBookingComponent,
    canActivate: [() => inject(AuthService).isAuthenticated()],
    providers: [
      // NGRX
      provideState(bookingFeature),
      provideEffects([BookingEffects]),

      // Providing **another** logger for this part of the app:
      provideLogger(
        {
          level: LogLevel.DEBUG,
          chaining: true,
```

```
          appenders: [DefaultLogAppender],
        },
        withColor({
          debug: 42,
          error: 43,
          info: 46,
        })
      ),

    ],
    children: [
      {
        path: 'flight-search',
        component: FlightSearchComponent,
      },
      [...]
    ],
  },
];
```

This sets up **another** set of the Logger's services in the environment injector of this lazy route and its children. These services are shadowing their counterparts in the root scope. Hence, when a component in the lazy scope calls the `LoggerService`, the services in the root scope are not triggered.

To prevent this, we can get the `LoggerService` from the parent scope. More precisely, it's not *the* parent scope but the "nearest ancestor scope" providing a `LoggerService`. After that, the service can delegate to its parent. This way, the services are chained:

```
@Injectable()
export class LoggerService {
  private appenders = inject(LOG_APPENDERS);
  private formatter = inject(LogFormatter);
  private config = inject(LoggerConfig);

  private parentLogger = inject(LoggerService, {
    optional: true,
    skipSelf: true,
  });
  [...]

  log(level: LogLevel, category: string, msg: string): void {

    // 1. Do own stuff here
```

```
16      [...]
17
18      // 2. Delegate to parent
19      if (this.config.chaining && this.parentLogger) {
20          this.parentLogger.log(level, category, msg);
21      }
22    }
23    [...]
24  }
```

When using inject to get hold of the parent's `LoggerService`, we need to pass the `optional: true` to avoid an exception if there is no ancestor scope with a `LoggerService`. Passing `skipSelf: true` makes sure, only ancestor scopes are searched. Otherwise, Angular would start with the current scope and retrieve the calling service itself.

Also, the example shown here allows activating/deactivating this behavior via a new `chaining` flag in the `LoggerConfiguration`.

Occurrences and Variations

- The `HttpClient` uses this pattern to also trigger `HttpInterceptors` in parent scopes. More details on chaining HttpInterceptors[25] can be found here[26]. Here, the chaining behavior can be activated via a separate feature. Technically, this feature registers another interceptor delegating to services in the parent scope.

Pattern: Functional Service

Intentions

- Making the usage of libraries more lightweight by using functions as services
- Reducing indirections by going with ad-hoc functions

Description

Instead of forcing the consumer to implement a class-based service following a given interface, a library also accepts functions. Internally, they can be registered as a service using `useValue`.

Example

In this example, the consumer can directly pass a function acting as a `LogFormatter` to `provideLogger`:

[25] https://www.angulararchitects.io/aktuelles/the-refurbished-httpclient-in-angular-15-standalone-apis-and-functional-interceptors/
[26] https://www.angulararchitects.io/aktuelles/the-refurbished-httpclient-in-angular-15-standalone-apis-and-functional-interceptors/

```
bootstrapApplication(AppComponent, {
  providers: [
    provideLogger(
      {
        level: LogLevel.DEBUG,
        appenders: [DefaultLogAppender],

        // Functional CSV-Formatter
        formatter: (level, cat, msg) => [level, cat, msg].join(";"),
      },
      withColor({
        debug: 3,
      })
    ),
  ],
});
```

To allow for this, the Logger uses a `LogFormatFn` type defining the function's signature:

```
export type LogFormatFn = (
  level: LogLevel,
  category: string,
  msg: string
) => string;
```

Also, as functions cannot be used as tokens, an `InjectionToken` is introduced:

```
export const LOG_FORMATTER = new InjectionToken<LogFormatter | LogFormatFn>(
  "LOG_FORMATTER"
);
```

This `InjectionToken` supports both class-based `LogFormatter` as well as functional ones. This prevents breaking existing code. As a consequence of supporting both, `provideLogger` needs to treat both cases in a slightly different way:

```
export function provideLogger(config: Partial<LoggerConfig>, ...features: LoggerFeat\
ure[]): EnvironmentProviders {

    const merged = { ...defaultConfig, ...config};

    [...]

    return makeEnvironmentProviders([
        LoggerService,
        {
            provide: LoggerConfig,
            useValue: merged
        },

        // Register LogFormatter
        //  - Functional LogFormatter:  useValue
        //  - Class-based LogFormatters: useClass
        (typeof merged.formatter === 'function' ) ? {
            provide: LOG_FORMATTER,
            useValue: merged.formatter
        } : {
            provide: LOG_FORMATTER,
            useClass: merged.formatter
        },

        merged.appenders.map(a => ({
            provide: LOG_APPENDERS,
            useClass: a,
            multi: true
        })),
        [...]
    ]);
}
```

While class-based services are registered with `useClass`, `useValue` is the right choice for their functional counterparts.

Also, the consumers of the `LogFormatter` need to be prepared for both the functional as well as class-based approach:

```
@Injectable()
export class LoggerService {
  private appenders = inject(LOG_APPENDERS);
  private formatter = inject(LOG_FORMATTER);
  private config = inject(LoggerConfig);

  [...]

  private format(level: LogLevel, category: string, msg: string): string {
    if (typeof this.formatter === 'function') {
        return this.formatter(level, category, msg);
    }
    else {
        return this.formatter.format(level, category, msg);
    }
  }

  log(level: LogLevel, category: string, msg: string): void {
    if (level < this.config.level) {
      return;
    }

    const formatted = this.format(level, category, msg);

    [...]
  }
  [...]
}
```

Occurrences and Variations

- The `HttpClient` allows using functional interceptors. They are registered via a feature (see pattern *Feature*).
- The `Router` allows using functions for implementing guards and resolvers.

Conclusion

Provider Factories are simple functions returning an array with providers. They are used to get all providers needed for setting up a subsystem or a library. By convention, such factories follow the naming pattern `privateXY`.

A provider factory can accept a configuration object and optional features. An optional feature is another function returning all providers needed for the feature in question. Their names follow the naming pattern `withXYZ`.

An `ENVIRONMENT_INITIALIZER` can be used to wire up services and using the inject together with parameters like `optional` and `skipSelf` allows for establishing a chain with another instance of the same service in a parent scope.

How to prepare for 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:

```
import { Component, OnInit } from '@angular/core';
import { TicketsModule } from '../tickets/tickets.module';

@Component({
    selector: 'app-next-flight',
    standalone: true,
    imports: [
        // Existing NgModule imported
        // in this standalone component
        TicketsModule
    ],
    [...]
})
export class NextFlightComponent implements OnInit {
    [...]
}
```

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

```
@NgModule({
  imports: [
    CommonModule,

    // Imported Standalone Component:
    FlightCardComponent,
    [...]
  ],
  declarations: [
    MyTicketsComponent
  ],
  [...]
})
export class TicketsModule { }
```

This mutual compatibility is made possible by the mental model[27] behind Standalone Components.

[27] https://www.angulararchitects.io/en/aktuelles/angulars-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 neighborhood: It contains all other Standalone Components, Standalone Directives, and Standalone Pipes but also NgModules the component in question needs.

Option 2a: Automatic Migration to Standalone

The Angular CLI helps with migrating an existing code base to Standalone Components. The next chapter will discuss this option in detail.

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:

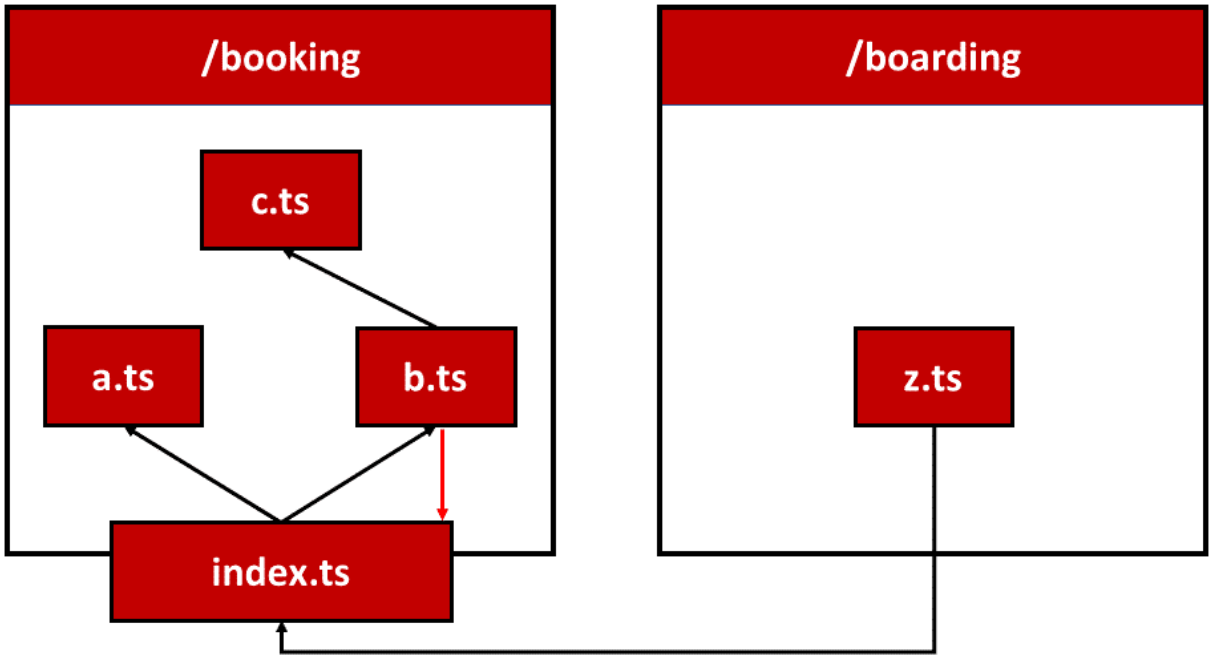
```
1 import { NavbarComponent, SidebarComponent } as shell from '../shell';
```

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';
```

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



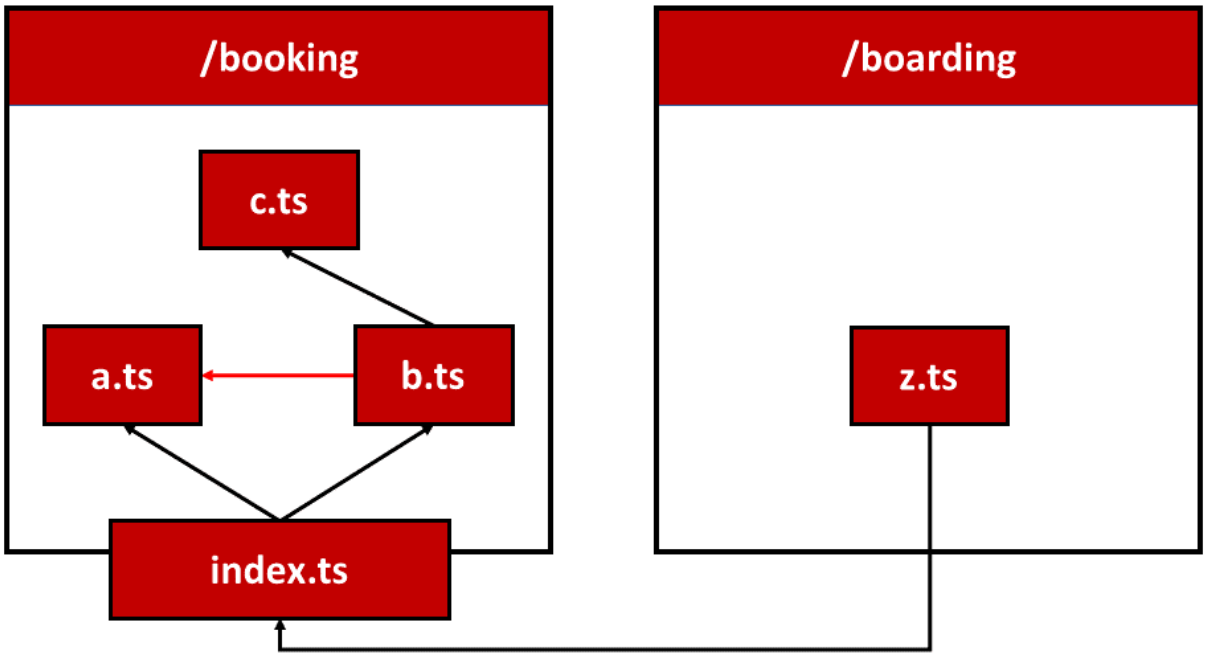
Cyclic dependencies via barrel

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:



Avoiding cyclic dependencies

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.

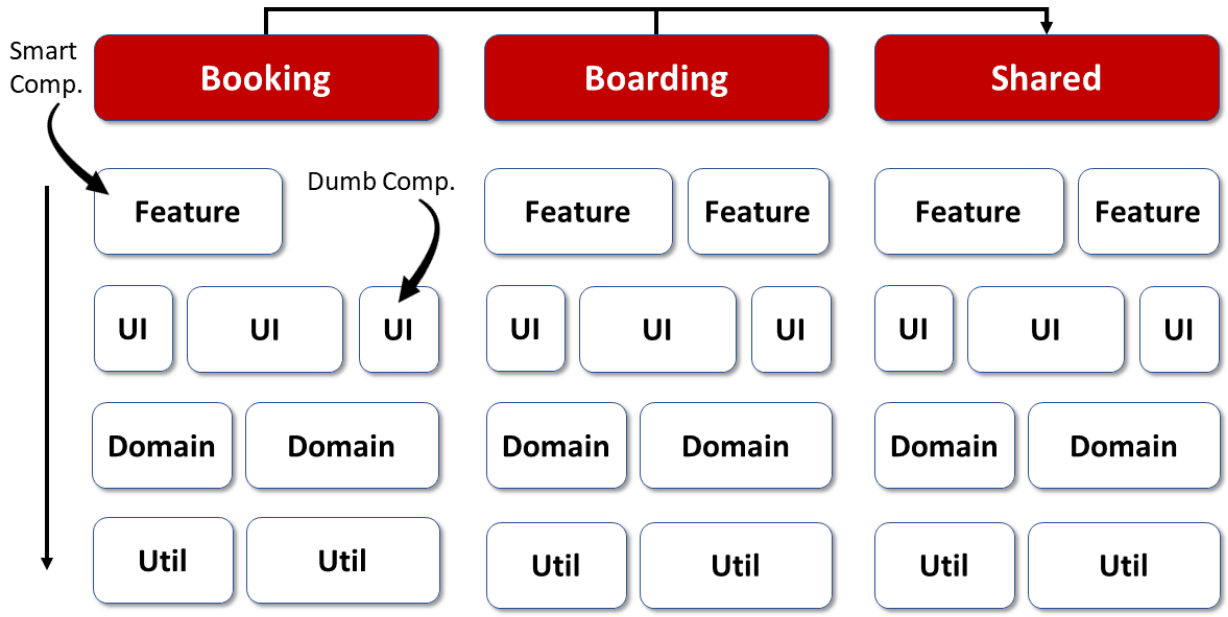
This problem can be solved with linting. A linting rule would have to detect and denounce unauthorized access. The popular tool Nx[28] comes with such a rule, which can also be used to prevent other unwanted accesses. The next section takes up this idea.

Option 4: Nx Workspace with Libraries and Linting Rules

The popular tool Nx[29] 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.

[28] https://www.angulararchitects.io/en/aktuelles/tutorial-first-steps-with-nx-and-angular-architecture/
[29] 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:



Architecture Matrix

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

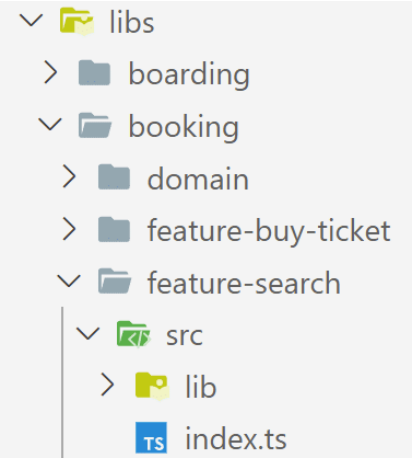
With the linting rules mentioned, it can now be ensured that each layer may only access the layers below it. Access to other domains can also be prevented. Libraries from the *Booking* area are therefore not allowed to access libraries in *Boarding*. If you want to use certain constructs across domains,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

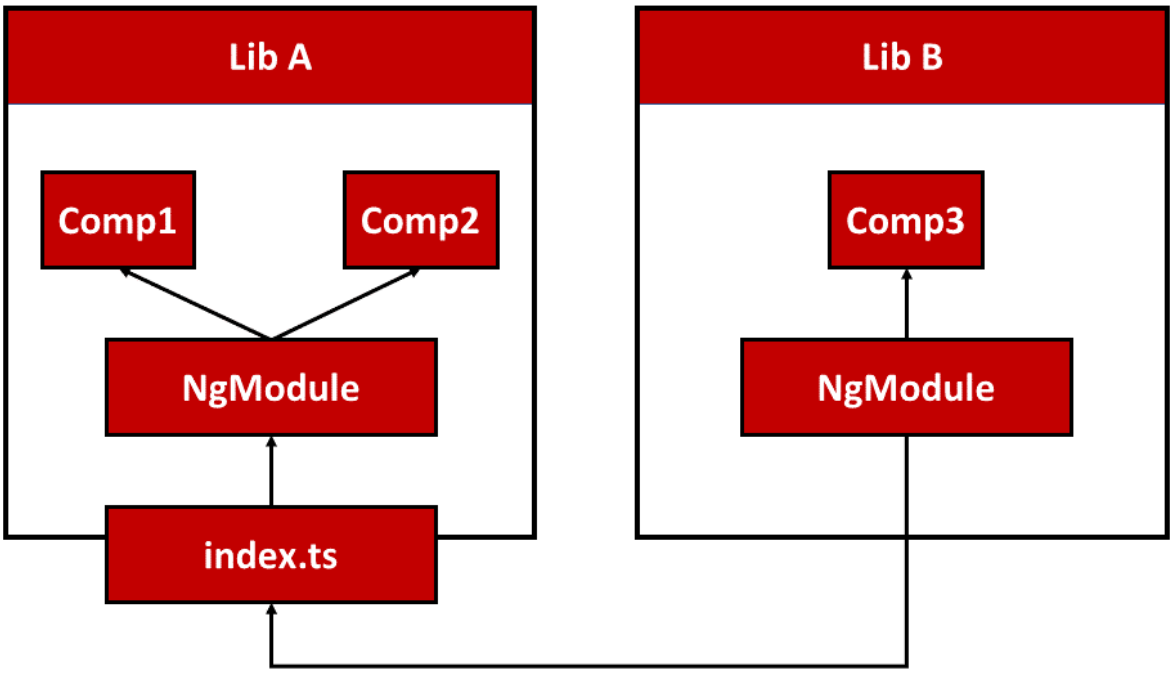
The folder structure used for this by Nx reflects the architecture matrix shown:



Structure of Nx workspace

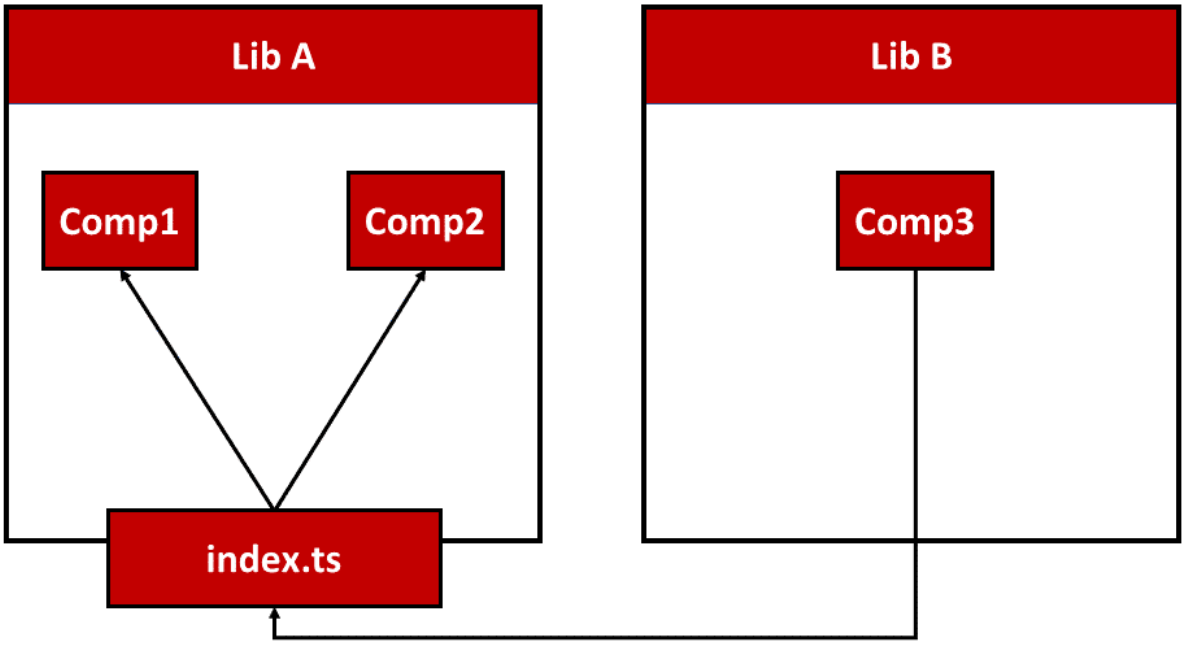
The subfolders in libs represent the domains. The libraries found in it get a prefix like `feature-` or `domain-`. These prefixes reflect the technical categories and thus the layers.

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.

Option 4a: Folder-based Module Boundaries with Sheriff

As discussed in one of the previous chapters, the open source tool Sheriff allows to enforce module boundaries on a per-folder basis. This makes the application structure more lightweight. It can be used with the Angular CLI but also with Nx to get the best of both worlds.

Conclusion

While Standalone Component are meanwhile the preferred way of using components, the traditional NgModule-based style is still a first-class citizen. Hence, we don't need to migrate existing code immediately.

As both options play well together, we can mix and match them. For instance, we could sick with NgModules for existing code and write new code using Standalone Components.

Barrels and Libraries can be used as a replacement for NgModule. They allow for hiding implementation details regardless of whether they are Angular-based or not. With tools like Nx or Sheriff we can prevent the usage of such implementation details and define which parts of the application can access which other parts.

Automatic Migration to Standalone Components in 3 Steps

While the new Standalone Components can be perfectly combined with traditional NgModule-based Angular code, people might want to fully migrate their projects to the new standalone world. Since Angular 15.2, there is a schematic that automates this task. In 3 steps it converts projects. After each step, we can check the current progress and manually take care of details the automatic process could not cope with.

In this short tutorial, I'm going to step through these 3 steps with you and migrate our demo app.

If you want to play through these steps, you can find the NgModule-based initial situation of our tutorial here:

 https://github.com/manfredsteyer/standalone-example-cli
 (**Branch** `ngmodules`)

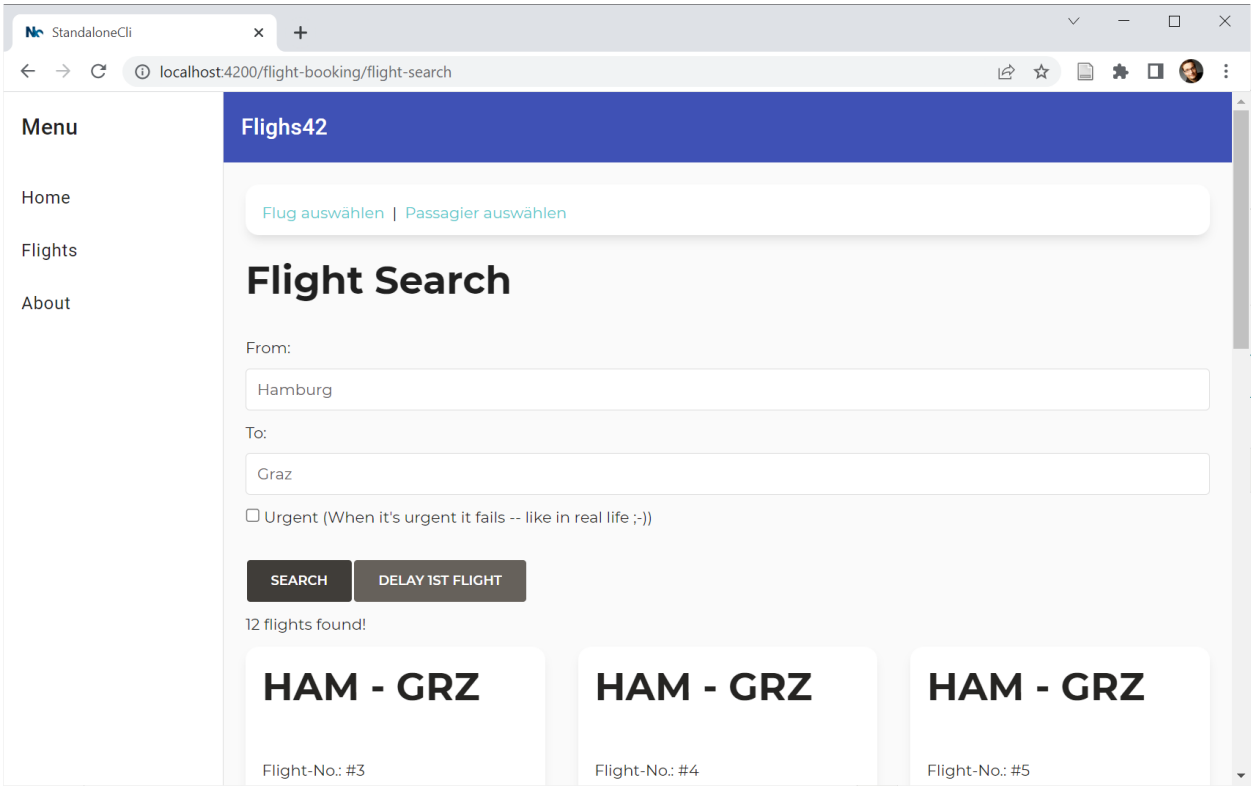
A First Look at the Application to Migrate

After checking out the `ngmodules` branch (!) of the above-mentioned project, it might be a good idea to go through the source code a bit. You should recognize the following NgModules:

```
   +----> SharedModule < --------+
   |                             |
AppModule --- (lazy) ---> FlightBookingModule
```

Also, start the application to get a first impression of it:

```
ng serve -o
```

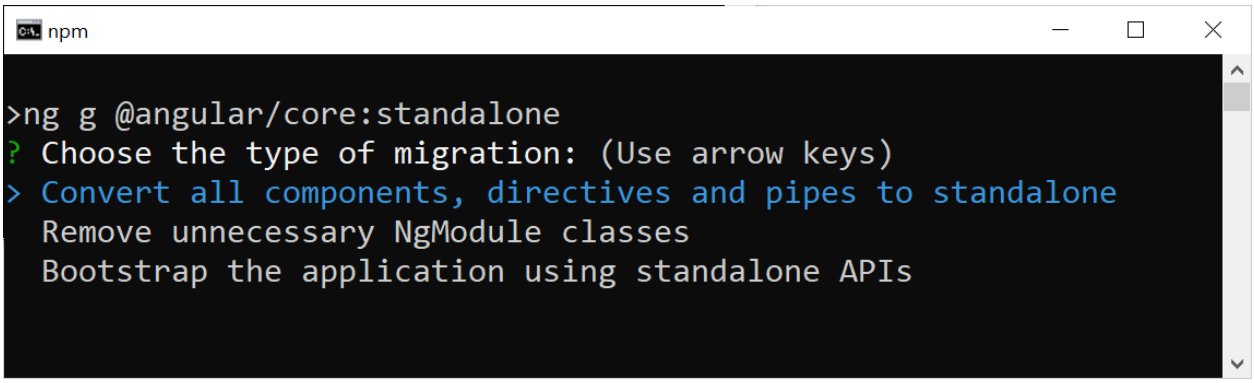


Step 1

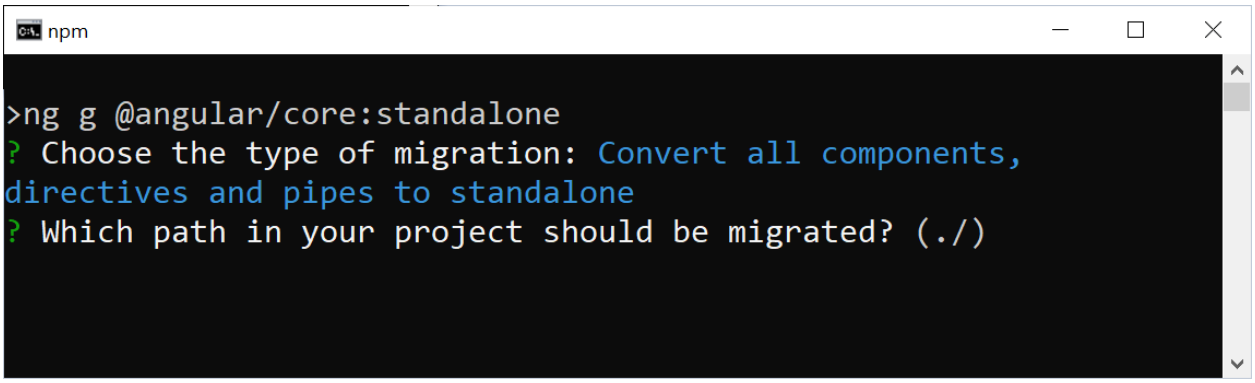
Now, let's run the migration schematic available from Angular 15.2:

```
ng g @angular/core:standalone
```

When asked about the type of migration, we select the first option (because it's such a nice tradition to "start at the beginning" ...).



When asked about the path to migrate, we go with the default value by pressing enter:

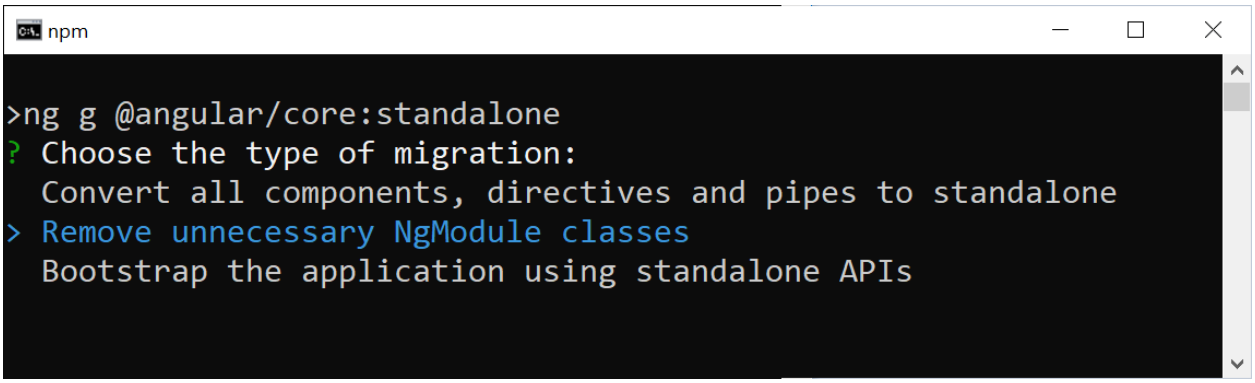


This default value `./` points to the project's root. Hence, the whole project will be migrated at once. For small and medium-sized applications this should be fine. For bigger applications, however, it might be interesting to migrate the project gradually.

After this first step, you should have a look at the source code and check if everything is fine. In the case of this example project, you don't need to bother. The schematics do a great job here!

Step 2

Now, let's run the schematic again for the second step:



The output shows that the `SharedModule` was removed and the other modules have been updated. It's fine that the `AppModule` is still in place – it will be removed in the 3rd step. However, all other NgModules should be gone by now. Unfortunately, the `FlightBookingModule` is still here:

```
// src/app/booking/flight-booking.module.ts

@NgModule({
    imports: [
        CommonModule,
        FormsModule,
        StoreModule.forFeature(bookingFeature),
        EffectsModule.forFeature([BookingEffects]),
        RouterModule.forChild(FLIGHT_BOOKING_ROUTES),
        FlightCardComponent,
        FlightSearchComponent,
        FlightEditComponent,
        PassengerSearchComponent
    ],
    exports: [],
    providers: []
})
export class FlightBookingModule { }
```

As this listing shows, the `FlightBookingModule` doesn't do much anymore. However, there are some calls to methods within the `imports` section. These methods are for setting up the router and the NGRX store. As they are quite library-specific, the schematic was not able to convert them into calls of equivalent Standalone APIs. So, we need to take care of this by hand.

`RouterModule.forChild` sets up some child routes that are loaded alongside the `FlightBookingModule`. However, in a standalone world, we don't need NgModules for setting up child routes anymore. Instead, the parent routing configuration can **directly** point to the child routes. Hence, let's switch to the file `app.routes.ts` and update the route triggering lazy loading:

```
// src/app/app.routes.ts

{
    path: 'flight-booking',
    canActivate: [() => inject(AuthService).isAuthenticated()],
    loadChildren: () =>
        import('./booking/flight-booking.routes')
                .then(m => m.FLIGHT_BOOKING_ROUTES)
},
```

Please note that the `import` now directly imports the flight booking routes. There is no indirection via the `FlightBookingModule` anymore. There is even the possibility of shortening this further: If the file `flight-booking.routes.ts` exports the routes as its **default export**, we can skip the subsequent `then` call:

```
{
    path: 'flight-booking',
    canActivate: [() => inject(AuthService).isAuthenticated()],
    loadChildren: () =>
        import('./booking/flight-booking.routes')
},
```

To make sure, the NGRX store is initialized for this lazy application part, we can register the respective providers directly for the lazy child routes:

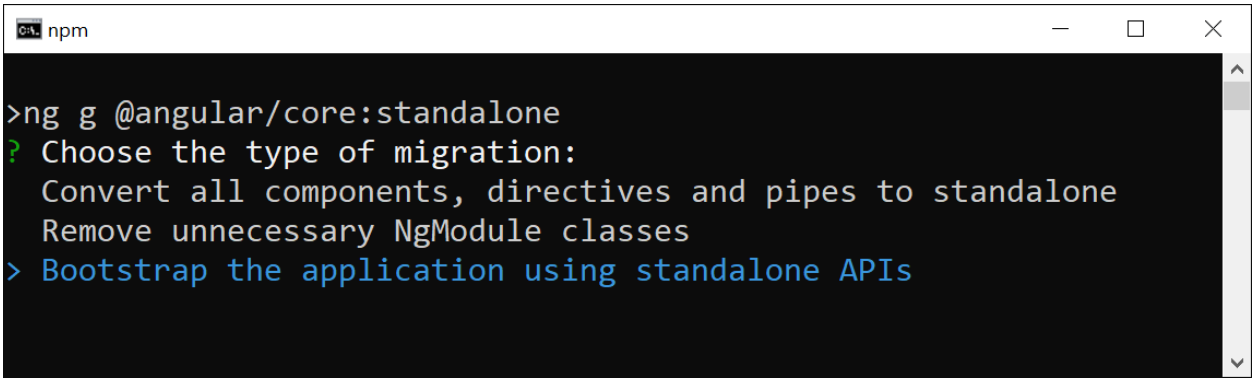
```
// src/app/booking/flight-booking.routes.ts
import { importProvidersFrom, inject } from '@angular/core';
[...]

export const FLIGHT_BOOKING_ROUTES: Routes = [
  {
    path: '',
    component: FlightBookingComponent,
    canActivate: [() => inject(AuthService).isAuthenticated()],
    providers: [
      importProvidersFrom(StoreModule.forFeature(bookingFeature)),
      importProvidersFrom(EffectsModule.forFeature([BookingEffects])),
    ],
    children: [
        [...]
    ],
  },
];
```

This new provider array sets up services that are only needed for the route at hand and its children. The function `importProvidersFrom` bridges over to the world of NgModules and allows retrieving their providers.

Now, we can delete the `FlightBookingModule` (`src/app/booking/flight-booking.module.ts`).

Step 3

Let's run our migration schematic for the 3rd time:



This removes the `AppModule` and updates the file `main.ts` to directly bootstrap the `AppComponent`. After this step, the application should work as before:

```
ng serve -o
```

Bonus: Moving to Standalone APIs

When we look into the `main.ts`, we see that it still references several modules with `importProvidersFrom`:

```
bootstrapApplication(AppComponent, {
  providers: [
    importProvidersFrom(
      BrowserModule,
      LayoutModule,
      LoggerModule.forRoot({
        level: LogLevel.DEBUG,
        appenders: [DefaultLogAppender],
        formatter: (level, cat, msg) => [level, cat, msg].join(';'),
      }),
      StoreModule.forRoot(reducer),
      EffectsModule.forRoot(),
      StoreDevtoolsModule.instrument(),
      MatToolbarModule,
      MatButtonModule,
      MatSidenavModule,
      MatIconModule,
      MatListModule
    ),
    {
```

```
      provide: HTTP_INTERCEPTORS,
      useClass: LegacyInterceptor,
      multi: true,
    },
    provideAnimations(),
    provideHttpClient(withInterceptorsFromDi()),
    provideRouter(APP_ROUTES, withPreloading(PreloadAllModules)),
  ],
});
```

Also, there is a traditional class-based `HttpInterceptor` registered and the `HttpClient` is made aware of this by calling `withInterceptorsFromDi`. By moving to Standalone APIs, this can be improved:

```
bootstrapApplication(AppComponent, {
  providers: [

    provideLogger({
      level: LogLevel.DEBUG,
      appenders: [DefaultLogAppender],
      formatter: (level, cat, msg) => [level, cat, msg].join(';'),
    }),

    provideStore(reducer),
    provideEffects(),
    provideStoreDevtools(),

    provideAnimations(),

    provideHttpClient(withInterceptors([authInterceptor])),
    provideRouter(APP_ROUTES, withPreloading(PreloadAllModules)),

    importProvidersFrom(
      LayoutModule,
      MatToolbarModule,
      MatButtonModule,
      MatSidenavModule,
      MatIconModule,
      MatListModule
    ),

  ],
});
```

This modification takes care of the following aspects:

- Removing the import of the `BrowserModule` that doesn't need to be explicitly imported when bootstrapping a Standalone Component.
- Setting up the custom Logger library with `provideLogger`.
- Setting up the NGRX store with `provideStore`, `provideEffects`, and `provideStoreDevtools`.
- Replacing the traditional `HttpInterceptor` with a functional interceptor[30] that is now passed to `withInterceptors`. To make this step easier, the functional counterpart has already been part of the code base since the beginning.

More information about custom Standalone APIs[31] like provideLogger can be found here[32].

NGRX expects that its Standalone APIs are used fully or not at all. Hence, we also need to go back to the `flight-booking.routes.ts` and replace the call to `importProvidersFrom` with calls to `provideState` and `provideEffects`:

```
export const FLIGHT_BOOKING_ROUTES: Routes = [
  {
    path: '',
    component: FlightBookingComponent,
    canActivate: [() => inject(AuthService).isAuthenticated()],
    providers: [
      provideState(bookingFeature),
      provideEffects(BookingEffects)
    ],
    children: [
        [...]
    ],
  },
];
```

Please note that while we call `provideStore` in the `main.ts` to set up the store, we need to call `provideState` (!) in further parts of the application to set up additional feature slices for them. However, `provideEffects` can be called in both places to set up effects for the root level but also feature slices.

After this modification, the application is migrated to Standalone Components and APIs. Run it via

```
ng serve -o
```

[30] https://www.angulararchitects.io/en/aktuelles/the-refurbished-httpclient-in-angular-15-standalone-apis-and-functional-interceptors/
[31] https://www.angulararchitects.io/en/aktuelles/patterns-for-custom-standalone-apis-in-angular/
[32] https://www.angulararchitects.io/en/aktuelles/patterns-for-custom-standalone-apis-in-angular/

Conclusion

The new schematics automate the migration to Standalone Components. In three steps, the whole application or just a part of it is moved over to the new lightweight way of working with Angular. After each step, we can check the performed modification and intervene.

Signals in Angular: The Future of Change Detection

Sarah Drasner, Director of Engineering at Google, spoke of an Angular renaissance on Twitter. That's pretty much it because several innovations have made Angular extremely attractive in the last few releases. Probably the most important are standalone components and standalone APIs.

Next, the Angular team takes care of renewing the change detection. It should be more lightweight and powerful. To do this, Angular will rely on a reactive mechanism called Signals, which several other frameworks have already adopted.

Signals will be available from Angular 16. Similar to standalone components, they initially come as a developer preview so that early adopters can gain initial experience.

In this chapter, I will go into this new mechanism and show how it can be used in an Angular application.

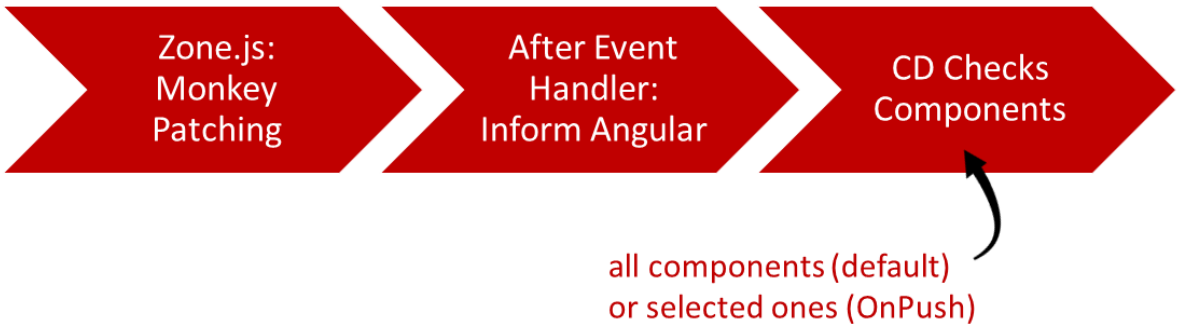
 Source Code[33](see **branches** `signals` and `signal-rxjs-interop`)

Change Detection Today: Zone.js

Angular currently assumes that any event handler can theoretically change any bound data. For this reason, after the execution of event handlers, the framework checks all bindings in all components for changes by default. In the more powerful OnPush mode, which relies on Immutables and Observables, Angular can drastically reduce the number of checked components.

Regardless if we go with the default behavior or OnPush, Angular needs to know when event handlers have run. This is challenging because the browser, not the framework, triggers the event handlers. Zone.js helps here: Using monkey patching, it extends JavaScript objects such as `window` or `document` and prototypes such as `HtmlButtonElement`, `HtmlInputElement` or `Promise`. By modifying such standard constructs, Zone.js can determines when an event handler has run. It then notifies Angular that it takes care of the change detection:

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



Change

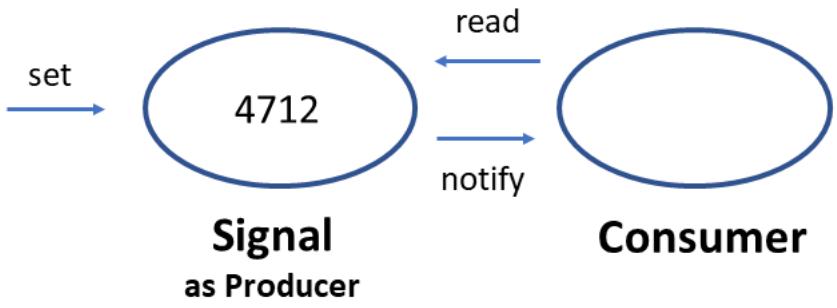
While this approach has worked well in the past, it still comes with a few downsides:

- Zone.js monkey patching is magic. Browser objects are modified, and errors are difficult to diagnose.
- Zone.js has an overhead of around 100 KB. While negligible for larger applications, this is a deal-breaker when deploying lightweight web components.
- Zone.js cannot monkey-patch `async` and `await` as they are keywords. Therefore, the CLI still converts these statements into promises, even though all supported browsers already support `async` and `await` natively.
- When changes are made, entire components including their predecessors, are always checked in the component tree. It is currently not possible to directly identify changed components or to just update the changed parts of a component.

Exactly these disadvantages are now compensated with Signals.

Change Detection Tomorrow: Signals

A signal is a simple reactive building block: it holds a value consumers can read. Depending on the nature of the signal, the value can also be changed, after which the signal notifies all consumers:



How

If the consumer is a template, it can notify Angular about changed bindings. In the terminology of the Angular team, the signal occurs as a so-called producer. As described below, there are also other building blocks that can fill this role.

Using Signals

For using Signals with data binding, properties to be bound are expressed as signals:

```
@Component([…])
export class FlightSearchComponent {

  private flightService = inject(FlightService);

  from = signal('Hamburg');
  to = signal('Graz');
  flights = signal<Flight[]>([]);

  […]

}
```

It should be noted here that a signal always has a value by definition. Therefore, a default value must be passed to the `signal` function. If the data type cannot be derived from this, the example specifies it explicitly via a type parameter.

The Signal's getter is used to read the value of a signal. Technically, this means that the signal is called like a function:

```
async search(): Promise<void> {
  if (!this.from() || !this.to()) {
    return;
  }
  const flights = await this.flightService.findAsPromise(this.from(), this.to());
  this.flights.set(flights);
}
```

To set the value, the signal offers an explicit setter in the form of a `set` method. The example shown uses the setter to stow the loaded flights. The getter is also used for data binding in the template:

```
<div *ngIf="flights().length > 0">
  {{flights().length}} flights found!
</div>

<div class="row">
  <div *ngFor="let f of flights()">
    <flight-card [item]="f" />
  </div>
</div>
```

Method calls were frowned upon in templates in the past, especially since they could lead to performance bottlenecks. However, this does not generally apply to uncomplex routines such as getters. In addition, the template will appear here as a consumer, and as such, it can be notified of changes.

As of writing this, `ngModel` has not directly supported two-way data binding. However, it can be done by hand by setting up a property binding reading the signal and an event binding updating the signal with the field's current value:

```
<form #form="ngForm">
  <div class="form-group">
    <label>From:</label>
    <input [ngModel]="from()" (ngModelChange)="from.set($event)" name="from" class="\
form-control">
  </div>

  <div class="form-group">
    <label>To:</label>
    <input [ngModel]="to()" (ngModelChange)="to.set($event)" name="to" class="form-c\
ontrol">
  </div>

```

```
14    <div class="form-group">
15      <button class="btn btn-default" (click)="search()">Search</button>
16      <button class="btn btn-default" (click)="delay()">Delay</button>
17    </div>
18  </form>
```

In a future version, the Angular team will adapt the forms handling to Signals.

Updating Signals

In addition to the setter shown earlier, Signals also provide an `update` method for projecting the current value to a new one:

```
1  this.flights.update(f => {
2    const flight = f[0];
3    const date = addMinutes(flight.date, 15);
4    const updated = {...flight, date};
5
6    return [
7      updated,
8      ...f.slice(1)
9    ];
10 });
```

Signals Need to be Immutable

By default, a Signal content has to be immutable. For this reason, just updating the flight date in the previous section would not be sufficient. Instead, we have to clone the changed parts, so that they get a new object reference.

By comparing these references, Angular's OnPush change detection strategy can efficiently determine the changed parts of an object managed by a Signal. In the previous section, the Array and the first flight get an new object reference. The other flights are not changed and hence just copied over using `slice`. As a result, they keep their object reference.

Calculated Values, Side Effects, and Assertions

Some values are derived from existing values. Angular provides calculated signals for this:

```
flightRoute = computed(() => this.from() + ' to ' + this.to());
```

Such a signal is read-only and appears as both a consumer and a producer. As a consumer, it retrieves the values of the signals used - here `from` and `to` - and is informed about changes. As a producer, it returns a calculated value.

If you want to consume signals programmatically, you can use the `effect` function:

```
constructor() {
    effect(() => {
        console.log('from:', this.from());
        console.log('route:', this.flightRoute());
    });
}
```

The `effect` function executes the transferred lambda expression and registers itself as a consumer for the signals used. When one of these signals changes, the effect is triggered again.

If a signal changes several times in a row, or if several signals change one after the other, undesired interim results could occur. Let's imagine we change the search filter `Hamburg - Graz` to `London - Paris`:

```
setTimeout(() => {
  this.from.set('London');
  this.to.set('Paris');
}, 2000);
```

Here, `London - Graz` could come immediately after the setting `from` to `London`. Like many other Signal implementations, Angular's implementation prevents such occurrences. The Angular team's readme[34], which also explains the push/pull algorithm used, calls this desirable assurance "glitch-free".

Effects Need an Injection Context

An essential limitation of Effects is that they can only be used in an injection context. This is everywhere `inject` is allowed: in the constructor, as default values of class fields, and in provider factories. Also, you can use the `runInInjectionContext` function to run code in an injection context.

Hence, the effect set up in the constructor above would fail when placed in `ngOnInit` or in another method:

[34] https://github.com/angular/angular/blob/71d5cdae195f916e345d977f1f23f9490e09482e/packages/core/src/signals/README.md

```
ngOnInit(): void {
    // Effects are not allowed here.
    // Hence, this will fail:
    effect(() => {
        console.log('route:', this.flightRoute());
    });
}
```

In this case, you'd get the following error:

> ERROR Error: NG0203: effect() can only be used within an injection context such as a constructor, a factory function,

The technical reason is that effects use `inject` to get hold of the current `DestroyRef`. This service provided since Angular 16 helps to find out about the life span of the current building block, e. g. the current component or service. The effect uses the `DestroyRef` to "unsubscribe" itself when this building block is about to be destroyed.

For this reason, you would typically setup your effects in the constructor as shown in the last section. If you really want to setup an effect somewhere else, you can go with the `runInInjectionContext` function. However, it needs a reference to an `Injector`:

```
injector = inject(Injector);

ngOnInit(): void {
  runInInjectionContext(this.injector, () => {
    effect(() => {
      console.log('route:', this.flightRoute());
    });
  });
}
```

Writing Signals in Effects

If an effect wrote a Signal, a cycle might be created. Just imagine an effect that reads the Signal `a` and updates Signal `b` and another effect doing the same the other way round. To prevent such situations, effects are not allowed to write Signals. For instance, let's assume an effect synchronizing our `from` and `to` Signals:

```
effect(() => {
  // Writing into signals is not allowed here:
    this.to.set(this.from());
});
```

This effect would lead to the following error:

> ERROR Error: NG0600: Writing to signals is not allowed in a `computed` or an `effect` by default. Use `allowSignalWrites` in the `CreateEffectOptions` to enable this inside effects.

As an alternative, you should consider the usage of `computed`. If this does not work or if you **really** want to write into a signal, setting the `allowSignalWrites` property to true deactivates this child safety facility:

```
effect(() => {
    this.to.set(this.from());
}, { allowSignalWrites: true })
```

Signals and Change Detection

Similar to an Observable bound to a template using the `async`-Pipe, a bound Signal triggers change detection. This also works for the more efficient `OnPush` mode:

```
@Component({
  [...]
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class FlightSearchComponent { [...] }

[...]

@Component({
  [...]
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class FlightCardComponent { [...] }
```

However, to help Angular in `OnPush` mode to also find out about child components to look at, you have to use Immutables, as discussed above.

RxJS Interop

Admittedly, at first glance, signals are very similar to a mechanism that Angular has been using for a long time, namely RxJS Observables. However, signals are deliberately kept simpler.

If you need the power of RxJS and its operators, you can however convert them to Observables. The namespace `@angular/core/rxjs-interop` contains a function `toObservable` converting a Signal into an Observable and a function `toSignal` for the other way around. They allow using the simplicity of signals with the power of RxJS.

The following listing illustrates the use of these two methods by expanding the example shown into a debounced type-ahead:

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

  from = signal('Hamburg');
  to = signal('Graz');
  basket = signal<Record<number, boolean>>({ 1: true });
  flightRoute = computed(() => this.from() + ' to ' + this.to());

  from$ = toObservable(this.from);
  to$ = toObservable(this.to);

  flights$ = combineLatest({ from: this.from$, to: this.to$ }).pipe(
    filter(c => c.from.length >= 3 && c.to.length >= 3),
    debounceTime(300),
    switchMap(c => this.flightService.find(c.from, c.to))
  );

  flights = toSignal(this.flights$, {
    initialValue: []
  });
}
```

The example converts the signals `from` and `to` into the observables `from$` and `to$` and combines them with `combineLatest`. As soon as one of the values changes, filtering and debouncing occur before `switchMap` triggers the backend request. One of the benefits flattening operators like `switchMap` come with are guarantees in terms of asynchronicity. These guarantees help to avoid race conditions.

The `initialValue` passed to `toSignal` is needed because Signals always have a value. On the contrary, Observables might not emit a value at all. If you are sure your Observable has an initial

value, e.g., because it's a `BehaviorSubject` or because of using the `startsWith` operator, you can also set `requireSync` to `true`:

```
flights = toSignal(this.flights, {
    requireSync: true
});
```

If you neither set `initialValue` nor `requireSync`, the type of the returned Signal also supports the `undefinied` type, allowing an initial value of `undefined`. In the example shown, this would result in a `Signal<Flight[] | undefinied>` instead of `Signal<Flight[]>`. Consequently, your code has to check for `undefined` too.

NGRX and Other Stores?

So far, we directly created and managed the Signals. However, stores like NGRX will provide some additional convenience. For instance, beginning with NGRX 16, the store will come with a `selectSignal` method:

```
store = inject(Store);
flights = this.store.selectSignal(selectFlights);
```

Other store implementations will have similar adoptions.

Conclusion

Signals make Angular lighter and point the way to a future without Zone.js. They enable Angular to find out about components that need to be updated directly.

The Angular team remains true to itself: Signals are not hidden in the substructure or behind proxies but made explicit. Developers therefore always know which data structure they are actually dealing with. Also, signals are just an option. No one needs to change legacy code and a combination of traditional change detection and signal-based change detection will be possible.

In general, it should be noted that Signals is still in an early phase and will ship with Angular 16 as a developer preview. This allows early adopters to try out the concept and provide feedback. With this, too, the Angular team proves that the stability of the ecosystem is important to them - an important reason why many large enterprise projects rely on the framework penned by Google.

Component Communication with Signals: Inputs, Two-Way Bindings, and Content/ View Queries

Signals will shape Angular's future. However, the Signals concept itself is just one part of the overall story. We also need a way to communicate with (sub)components via Signals. Angular 17.1 brought Input Signals, and with Angular 17.2 we've got Signal-based Two-way Bindings and support for content and view queries. To align with Input Signals, Version 17.3 provides a new output API.

In this chapter, I show how to use these new possibilities.

 Source Code[35] (see different branches)

Input Signals

Inputs Signals allow us to receive data via Property Bindings in the form of Signals. For describing the usage of Signal Inputs, I'm using a simple `OptionComponent` representing a – for the sake of simplicity – non-selectable option. Here, three of them are presented:

Three simple option components

Defining an Input Signal

Input Signals are the counterpart to the traditional @Input decorator:

[35] https://github.com/manfredsteyer/signals-component-communication

```
@Component({
  selector: 'app-option',
  standalone: true,
  imports: [],
  template: `
    <div class="option">
      {{ label() }}
    </div>
  `,
  styles: [...]
})
export class OptionComponent {
  label = input.required<string>();
}
```

This `input` function is picked up by the Angular Compiler, emitting source code for property bindings. Hence, we should only use it together with properties. The other communication concepts discussed here also use this technique.

Having a function instead of a decorator allows to inform TypeScript about the proper type and whether it includes `undefined`. In the example shown before, `label` becomes an `InputSignal<string>` – an Input Signal providing a `string`. An `undefined` value is not possible as `input.required` defines a mandatory property.

An `InputSignal` is always read-only and can be used like a `Signal`. The template above, for instance, requests its current value by calling the getter (`label()`).

Binding to an Input Signal

In the case of our `InputSignal<string>`, the caller has to pass a `string`:

```
<app-option label="Option #1">
<app-option [label]="myStringProperty">
```

If this string comes from a Signal, we have to read it in the template:

```
<app-option [label]="mySignalProperty()">
```

Computed Signals and Effects as a Replacement for Life Cycle Hooks

All changes to the passed Signal will be reflected by the `InputSignal` in the component. Internally, both Signals are connected via the graph Angular is maintaining. Life cycle hooks like `ngOnInit` and `ngOnChanges` can now be replaced with `computed` and `effect`:

```
markDownTitle = computed(() => '# ' + this.label())

constructor() {
  effect(() => {
    console.log('label updated', this.label());
    console.log('markdown', this.markDownTitle());
  });
}
```

Options for Input Signals

Here are some further options for setting up an `InputSignal`:

Source Code	Description
label = input<string>();	Optional property represented by an `InputSignal<string | undefined>`
label = input('Hello');	Optional property represented by an `InputSignal<string>` with an initial value of `Hello`
label = input<string | undefined>('Hello');	Optional property represented by an `InputSignal<string| undefined>` with an initial value of `Hello`

Required Inputs Cannot Have a Default Value!

By definition, `input.required` cannot have a default value. This makes sense at first glance, however, there is a pitfall: If you try to read the value of a required input before it's been bound, Angular throws an exception.

Hence, you cannot directly access it in the constructor. Instead, you can use `ngOnInit` or `ngOnChanges`. Also, using inputs within `computed` or `effect` is always safe, as they are only first triggered when the component has been initialized:

```
@Component([...])
export class OptionComponent implements OnInit, OnChanges {
  label = input.required<string>();

  // safe
  markDownTitle = computed(() => '# ' + this.label())

  constructor() {
          // this would cause an exception,
        // as data hasn't been bound so far
```

```
        console.log('label', this.label);

        effect(() => {
                // safe
                console.log('label', this.label);
        })
  }

  ngOnInit() {
          // safe
          console.log('label', this.label);
  }

  ngOnChanges() {
          // safe
          console.log('label', this.label);
  }
}
```

Aliases for Input Signals

Both `input` and `input.require` also take a parameter object that allows the definition of an `alias`:

```
label = input.required({ alias: "title" });
```

In this case, the caller needs to bind to the property name defined by the alias:

```
<app-option title="Option #1">
  <app-option [title]="myStringProperty">
    <app-option [title]="mySignalProperty()"></app-option></app-option
></app-option>
```

In most cases, you should prevent the usage of aliases, as they create an unnecessary indirection. An often-seen exception to this rule is renaming one of a Directive's properties to match the configured attribute selector.

Transformer for Input Signals

Transformer have already been available for traditional @Inputs. They allow the transformation of a value passed via a property binding. In the following case, the transformer `booleanAttribute` that can be found in `angular/core` is used:

```
@Component({
  selector: 'app-option',
  standalone: true,
  imports: [],
  template: `
    <div class="option">
      {{ label() }} @if (featured()) { ★ }
    </div>
  `,
  styles: [...]
})
export class OptionComponent {
  label = input.required<string>();
  featured = input.required({
    transform: booleanAttribute
  })
}
```

This transformer converts strings to booleans:

```
<app-option label="Option #1" featured="true"></app-option>
```

Also, if the attribute is present but no value was assigned, `true` is assumed:

```
<app-option label="Option #1" featured></app-option>
```

This Signal's type is `InputSignal<boolean, unknown>`. The first type parameter (`boolean`) represents the value received from the transformer; the second one (`unknown`) is the value bound in the caller's template and passed to the transformer. Besides `booleanAttribute`, @angular/core also provides a `numberAttribute` transformer that converts passed strings to numbers.

If you want to implement a custom transformer, just provide a function taking the bound value and returning the value that should be used by the called component:

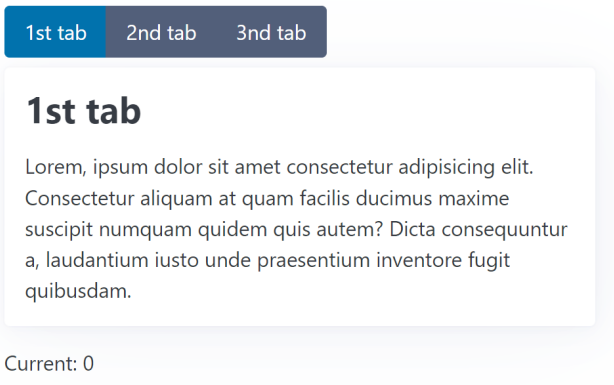
```
function boolTranformer(value: unknown): boolean {
  return value !== "no";
}
```

Then, register this function in your `input`:

```
@Component([...])
export class OptionComponent {
  label = input.required<string>();
  featured = input.required({
    transform: boolTranformer
  })
}
```

Two-Way Data Binding with Model Signals

Input Signals are read-only. If you want to pass a Signal that can be updated by the called component, you need to set up a so-called Model Signal. To demonstrate this, I'm using a simple `TabbedPaneComponent`:



TabbedPane

This is how the consumer can use this component:

```
<app-tabbed-pane [(current)]="current">
  <app-tab title="1st tab"> Lorem, ipsum dolor sit amet ... </app-tab>
  <app-tab title="2nd tab"> Sammas ergo gemma, ipsum dolor ... </app-tab>
  <app-tab title="3nd tab"> Gemma ham ipsum dolor sit ... </app-tab>
</app-tabbed-pane>

<p class="current-info">Current: {{ current() }}</p>
```

It gets several `TabComponents` passed. Also, a Signal `current` is bound via a Two-way Binding. For allowing this, the `TabbedPaneComponent` needs to provide a Model Signal using `model`:

```
@Component([...])
export class TabbedPaneComponent {
  current = model(0);
  [...]
}
```

Here, `0` is the initial value. The options are similar to the ones for input: `model.required` defines a mandatory property, and you can provide an alias via an options object. However, a transformer can not be defined.

If this component updates the Model Signal, the new value is propagated up to the the Signal bound in the template:

```
current.set(1);
```

Two-Way Data Binding as Combination of Input and Output

As usual in Angular, also Signal-based Two-way Bindings can be defined with a (read-only) Input and a respective Output. The Output's name must be the Input's name with the suffix `Change`. Hence, for `current` we need to define `currentChange`:

```
@Component([...])
export class TabbedPaneComponent {
  current = input(0);
  currentChange = output<number>();
}
```

For setting up an Output, we use the new output API. For triggering an event, the application has to call the output's `emit` method:

```
<button [...] (click)="currentChange.emit($index)">{{tab.title()}}</button>
```

Content Queries with Signals

The `TabbedPaneComponent` introduced in the previous section also allows us to showcase another option: Content Queries that get hold of projected Components or Directives.

As shown above, a `TabbedPaneComponent` gets several `TabComponents` passed. They are projected into the `TabbedPaneComponent`'s view. However, we only want to display one of them at a given time. Hence, the `TabbedPaneComponent` needs to get programmatic access to its `TabComponents`. This can be done with the new `contentChildren` function:

```
@Component({
  selector: 'app-tabbed-pane',
  standalone: true,
  imports: [],
  template: `
    <div class="pane">
      <div class="nav" role="group">
        @for(tab of tabs(); track tab) {
        <button
            [class.secondary]="tab !== currentTab()"
            (click)="activate($index)">
                {{tab.title()}}
        </button>
        }
      </div>
      <article>
        <ng-content></ng-content>
      </article>
    </div>
  `,
  styles: [...]
})
export class TabbedPaneComponent {
  current = model(0);
  tabs = contentChildren(TabComponent);
  currentTab = computed(() => this.tabs()[this.current()]);

  activate(active: number): void {
    this.current.set(active);
  }
}
```

The function `contentChildren` is the counterpart to the traditional @ContentChildren decorator. As `TabComponent` was passed as a so-called `locator`, it returns a Signal with an Array holding all projected `TabComponents`.

Having the projected nodes as a Signal allows us to project them using `computed` reactively. The shown example uses this option to derive a Signal `currentTab`.

The projected `TabComponent` uses this Signal to find out whether it should be visible:

```
@Component({
  selector: "app-tab",
  standalone: true,
  imports: [],
  template: `
    @if(visible()) {
    <div class="tab">
      <h2>{{ title() }}</h2>
      <ng-content></ng-content>
    </div>
    }
  `,
})
export class TabComponent {
  pane = inject(TabbedPaneComponent);

  title = input.required<string>();

  visible = computed(() => this.pane.currentTab() === this);
}
```

For this, we need to know that we can get all parents located in the DOM via dependency injection. The `visible` Signal is derived from the `currentTab` Signal.

This procedure is usual in the reactive world: Instead of imperatively setting values, they are declaratively derived from other values.

Content Queries for Descendants

By default, a Content Query only unveils direct content children. "Grandchildren", like the 3rd tab below, are ignored:

```
<app-tabbed-pane [(current)]="current">
  <app-tab title="1st tab"> Lorem, ipsum dolor sit amet ... </app-tab>
  <app-tab title="2nd tab"> Sammas ergo gemma, ipsum dolor ... </app-tab>

  <div class="danger-zone">
    <app-tab title="3nd tab">
      Here, you can delete the whole internet!
    </app-tab>
  </div>
</app-tabbed-pane>
```

To also get hold of such nodes, we can set the option `descendants` to `true`:

```
tabs = contentChildren(TabComponent, { descendants: true });
```

Output API

For the sake of API symmetricity, Angular 17.3 introduced a new output API. As already shown before, an `output` function is now used for defining an event provided by a component. Similar to the new input API, the Angular Compiler picks up the call to the `output` and emits respective code. The returned `OutputEmitterRef`'s `emit` method is used to trigger the event:

```
@Component([...])
export class TabbedPaneComponent {
  current = model(0);
  tabs = contentChildren(TabComponent);
  currentTab = computed(() => this.tabs()[this.current()]);

  tabActivated = output<TabActivatedEvent>();

  activate(active: number): void {
    const previous = this.current();
    this.current.set(active);
    this.tabActivated.emit({ previous, active });
  }
}
```

Providing Observables as Outputs

Besides this simple way of setting up outputs, you can use an Observable as the source for an output. For this, you find a function `outputFromObservable` in the RxJS interop layer:

```
import {
    outputFromObservable,
    toObservable
} from '@angular/core/rxjs-interop';
[...]

@Component([...])
export class TabbedPaneComponent {
  current = model(0);
  tabs = contentChildren(TabComponent);
  currentTab = computed(() => this.tabs()[this.current()]);
```

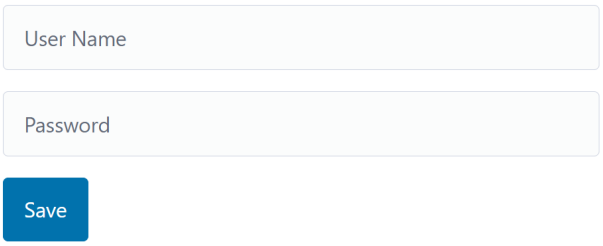
```
12
13   tabChanged$ = toObservable(this.current).pipe(
14     scan(
15       (acc, active) => ({ active, previous: acc.active }),
16       { active: -1, previous: -1 }
17     ),
18     skip(1),
19   );
20
21   tabChanged = outputFromObservable(this.tabChanged$);
22
23   activate(active: number): void {
24     this.current.set(active);
25   }
26
27 }
```

The function `outputFromObservable` converts an Observable to an `OutputEmitterRef`. In the shown example, the `scan` operator remembers the previous activated tab and `skip` ensures that no event is emitted when initially setting `current`. The latter one provides feature parity with the before-shown example.

View Queries with Signals

While a Content Query returns projected nodes, a View Query returns nodes from its own view. These are nodes found in the template of the respective component. In most cases, using data binding instead is the preferable solution. However, getting programmatic access to a view child is needed in some situations.

To demonstrate how to query view children, I use a simple form for setting a username and a password:



Form for setting username and password

Both `input` fields are marked as `required`. If the validation fails when pressing `Save`, the first field with a validation error should get the focus. For this, we need access to the `NgForm` directive the `FormModule` adds to our `form` tag as well as to the DOM nodes representing the `input` fields:

```
@Component({
  selector: "app-form",
  standalone: true,
  imports: [FormsModule, JsonPipe],
  template: `
    <h1>Form Demo</h1>
    <form autocomplete="off">
      <input
        [(ngModel)]="userName"
        placeholder="User Name"
        name="userName"
        #userNameCtrl
        required
      />
      <input
        [(ngModel)]="password"
        placeholder="Password"
        type="password"
        name="password"
        #passwordCtrl
        required
      />
      <button (click)="save()">Save</button>
    </form>
  `,
  styles: `
    form {
      max-width: 600px;
    }
  `,
})
export class FormDemoComponent {
  form = viewChild.required(NgForm);

  userNameCtrl =
    viewChild.required<ElementRef<HTMLInputElement>>("userNameCtrl");
  passwordCtrl =
    viewChild.required<ElementRef<HTMLInputElement>>("passwordCtrl");

  userName = signal("");
  password = signal("");

  save(): void {
```

```
44      const form = this.form();
45
46      if (form.controls["userName"].invalid) {
47        this.userNameCtrl().nativeElement.focus();
48        return;
49      }
50
51      if (form.controls["password"].invalid) {
52        this.passwordCtrl().nativeElement.focus();
53        return;
54      }
55
56      console.log("save", this.userName(), this.password());
57    }
58  }
```

Both are done using the `viewChild` function. In the first case, the example passes the type `NgForm` as the locator. However, just locating the fields with a type does not work, as there might be several children with this type. For this reason, the inputs are marked with handles (`#userName` and `#password`), and the respective handle's name is passed the locator.

View children can be represented by different types: The type of the respective Component or Directive, an `ElementRef` representing its DOM node, or a `ViewContainerRef`. The latter one is used in the next section.

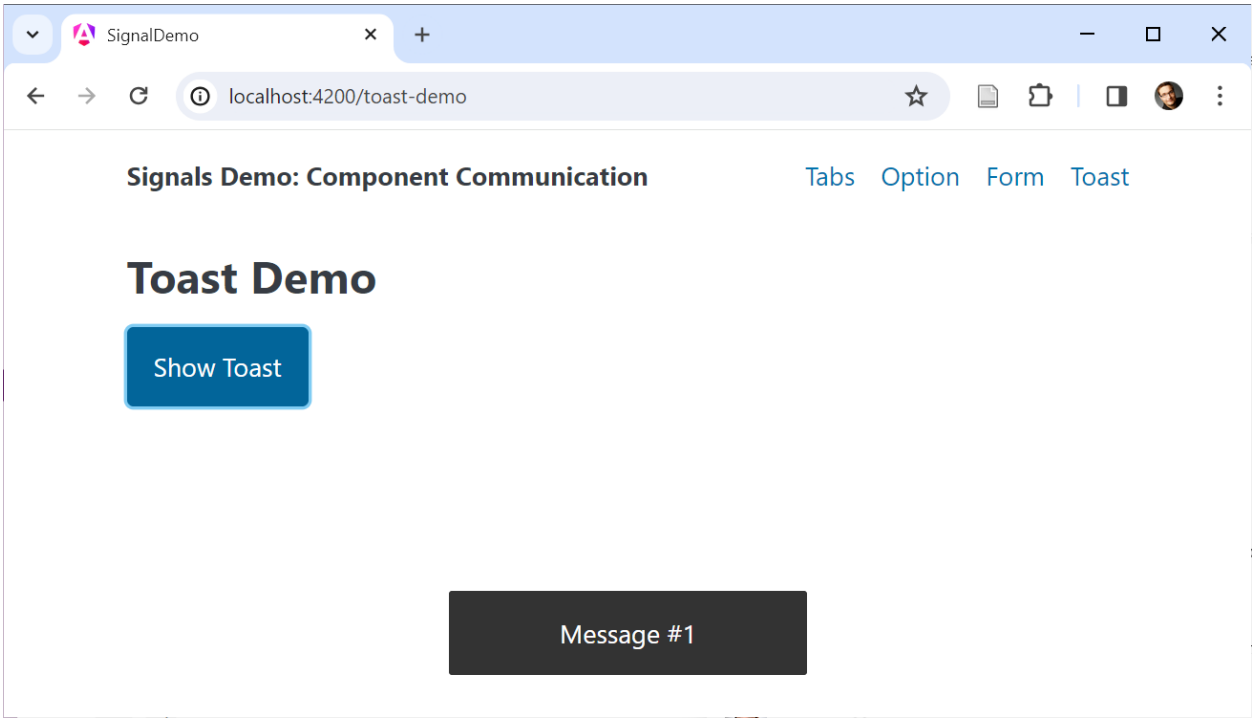
The desired type can be mentioned using the `read` option used in the previous example.

Queries and ViewContainerRef

There are situations where you need to dynamically add a component to a placeholder. Examples are modal dialogs or toasts. An easy way to achieve this is using the `*ngComponentOutlet` directive. A more flexible way is querying the `ViewContainerRef` of the placeholder.

You can think about a View Container as an invisible container around each Component and piece of static HTML. After getting hold of it, you can add further Components or Templates.

To demonstrate this, I'm using a simple example showing a toast:



Example displaying a toast

The example uses an `ng-container` as a placeholder:

```
@Component({
  selector: 'app-dynamic',
  standalone: true,
  imports: [],
  template: `
    <h2>Toast Demo</h2>
    <button (click)="show()">Show Toast</button>
    <ng-container #placeholder></ng-container>
  `,
  styles: [...]
})
export class ToastDemoComponent {
  counter = 0;
  placeholder = viewChild.required('placeholder', { read: ViewContainerRef });

  show() {
    const ref = this.placeholder()?.createComponent(ToastComponent);
    this.counter++;
    ref?.setInput('label', 'Message #' + this.counter);
    setTimeout(() => ref?.destroy(), 2000);
```

```
21     }
22
23 }
```

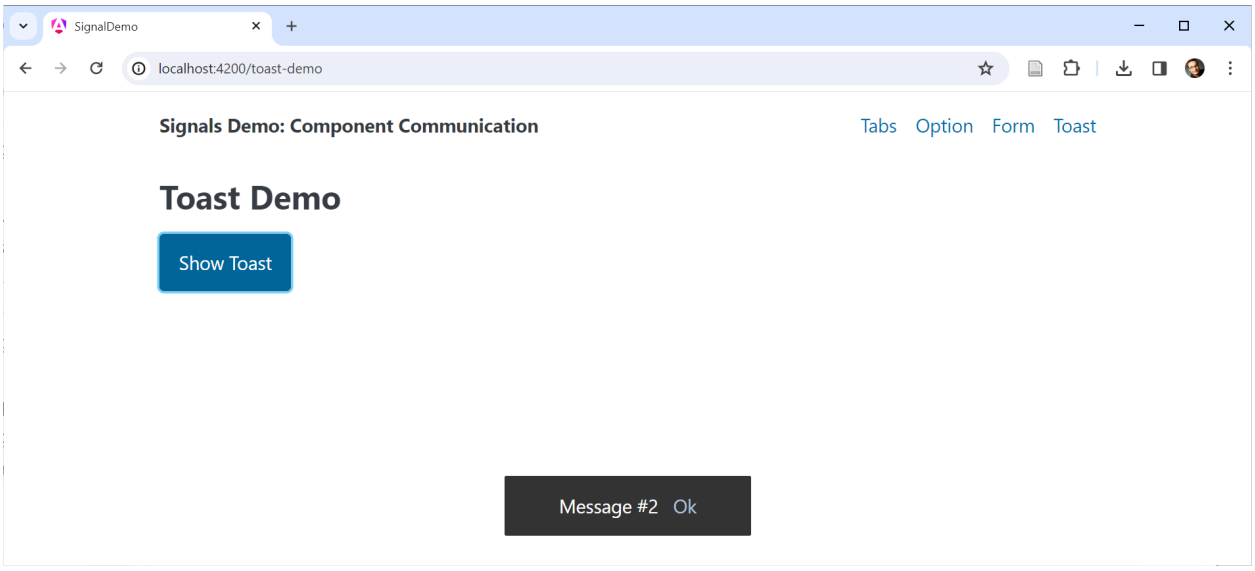
The `read` property makes clear that we don't want to read the placeholder component but its `ViewContainerRef`. The `createComponent` method instantiates and adds a `ToastComponent`. The returned `ComponentRef` is used to set the new component's `label` property. For this, its `setInput` method is used. After two seconds, the `destroy` method removes the toast again.

For the sake of simplicity, the component `ToastComponent` was hard-coded here. In more generic real-world scenarios, the component to use can be configured, e.g., by calling a service method, taking the Component type, and notifying another Component that adds a Component of this type to a placeholder.

Programmatically Setting up an Output

The previous example called `setInput` for assigning a value to the `ToastComponent`'s `title` input. Here, I want to discuss how to define event handlers for such dynamically added components.

Let's assume, the `ToastComponent` shows a confirmation link:



Toast with confirm link

When clicking this link, it emits an `confirmed` event:

```
@Component([...])
export class ToastComponent {
  label = input.required<string>();
  confirmed = output<string>();

  confirm(): void {
    this.confirmed.emit(this.label());
  }
}
```

To set up an handler for this event, we can directly use the returned `ComponentRef`'s `instance` property. It points to the added component instance and hence provides access to all its properties:

```
@Component([...])
export class ToastDemoComponent {
  counter = 0;
  placeholder = viewChild.required('placeholder', { read: ViewContainerRef });

  show() {
    const ref = this.placeholder()?.createComponent(ToastComponent);
    this.counter++;
    const title = 'Message #' + this.counter;
    ref.setInput('label', title);

    // Event handler for confirm output
    ref.instance.confirmed.subscribe(title => {
      ref?.destroy();
      console.log('confirmed', title);
    });

    setTimeout(() => ref?.destroy(), 5000);
  }

}
```

The `OutputEmitterRef`'s `subscribe` method allows to define an event handler. In our case, it just removes the toast using `destroy` and logs the received string to the console.

However, this example comes with a small beauty mistake. Regardless whether the user clicks the confirmation link or not, the example calls `destroy` after 5 seconds. Hence, the toast might be removed twice: Once after the confirmation and another time after displaying it for 5 seconds.

Fortunately, destroying a component twice does not result in an error. To solve this, we could introduce a `destroyed` flag . The next section shows a more powerful approach: Consuming outputs as Observables.

Consuming Outputs as Observables

Even though the `OutputEmitterRef` provides a `subscribe` method, it is not an Observable. However, the original `EventEmitter` used together with the @Output decorator was. To get back all the possibilities associated with Observable-based outputs, you can use the function `outputToObservable` that is part of the RxJS interop layer:

```
import { outputToObservable } from '@angular/core/rxjs-interop';
[...]

@Component([...])
export class ToastDemoComponent {
  counter = 0;
  placeholder = viewChild.required('placeholder', { read: ViewContainerRef });

  show() {
    const ref = this.placeholder()?.createComponent(ToastComponent);
    this.counter++;
    const title = 'Message #' + this.counter;
    ref.setInput('label', title);

    const confirmed$ = outputToObservable(ref.instance.confirmed)
      .pipe(map(title => ({ trigger: 'confirmed', title })));

    const timer$ = timer(5000);
      .pipe(map(() => ({ trigger: 'timeout', title })));

    race(confirmed$, timer$).subscribe(action => {
      ref?.destroy();
      console.log('action', action);
    });

  }

}
```

The function `outputToObservable` converts an `OutputEmitterRef` to an Observable. The shown example uses it to express both, the confirm event and the 5 sec timeout as observable. The `race` operator ensures that only the Observable that first issues a value is used.

The Observable returned by `outputToObservable` completes when Angular destroys the the output's component. For this reason, there is no need to unsubscribe by hand.

Feature Parity between Content and View Queries

So far, we have worked with `contentChildren` to query several projected children and `viewChild` to get hold of one node in the view. However, both concepts have feature parity: For instance, there is also a `contentChild` and a `viewChildren` function.

Also, all the options we've used above for View or Content Queries, like using handles as locators or using the `read` property, work for both kinds of queries.

Conclusion

Several new functions replace property decorators and help to set up data binding concepts. These functions are picked up by the Angular compiler emitting respective code.

The function `input` defines Inputs for property bindings, `model` defines Inputs for Two Way Data Binding, and `contentChild(ren)` and `viewChild(ren)` take care of Content and View Queries. Using these functions results in Signals that can be projected with `computed` and used within effects.

Successful with Signals in Angular - 3 Effective Rules for Your Architecture

It is undisputed that Signals will shape the future of Angular. At first glance, they seem very easy to use: The setters take new values, the getters deliver the current values and the templates as well as Effects are notified about changes.

Now you might be tempted to treat Signals like normal variables. This works in principle and can be a practical option when migrating existing code. However, in this case, the advantages of Signals and reactive systems only result to a limited extent. There are also some pitfalls and the code is not as straightforward and hence not as maintainable as it could be.

In order to prevent such situations, I would like to use this chapter to give you three simple rules of thumb that allow Signals to be used as idiomatically as possible.

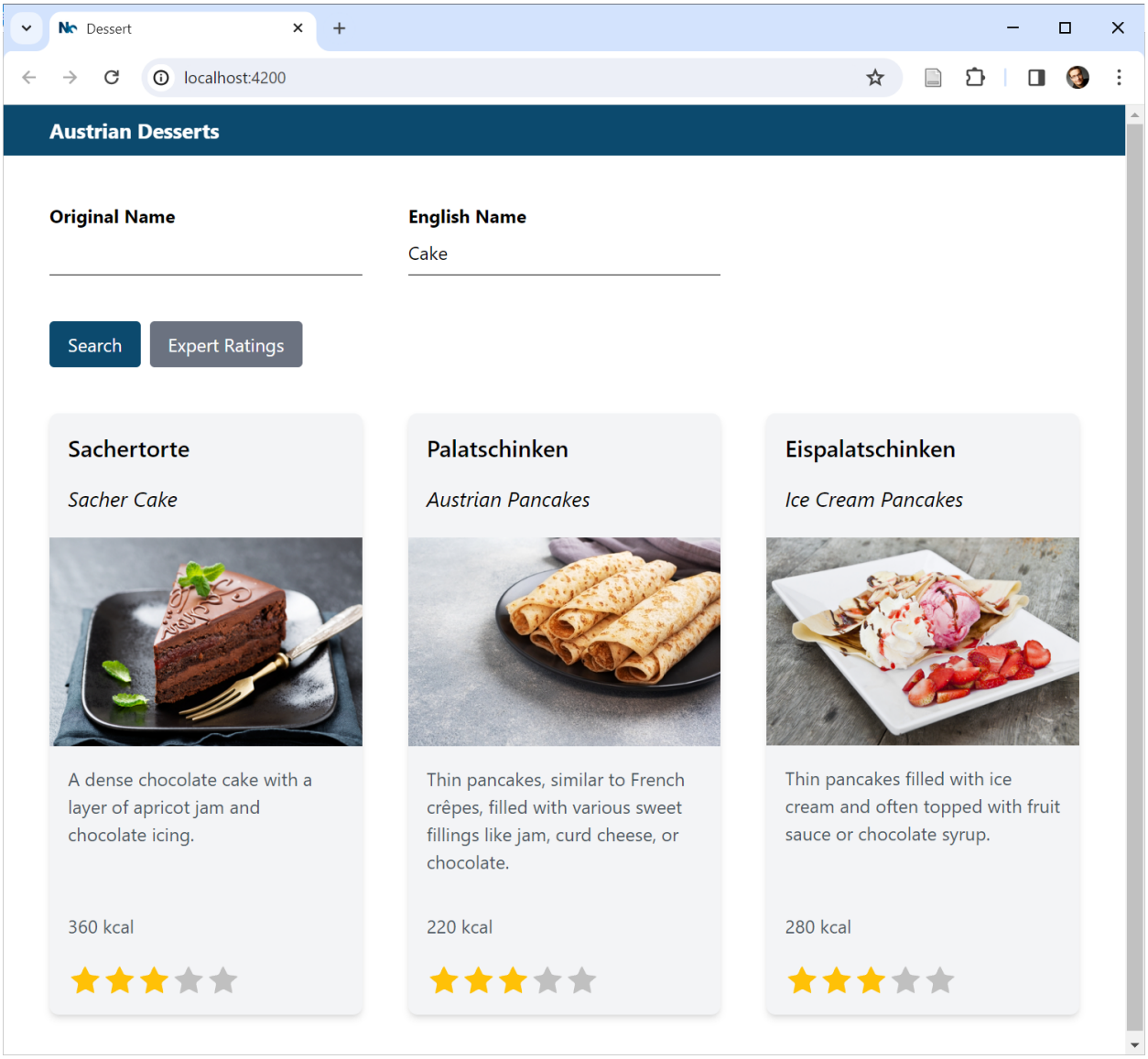
 Source Code[36] (see different branches!)

> Big thanks to Angular's Alex Rickabaugh[37] for a great discussion that led to the idea for this chapter and for providing feedback.

Initial Example With Some Room for Improvement

I would like to discuss the three promised rules using a simple Angular application:

[36] https://github.com/manfredsteyer/desserts.git
[37] https://twitter.com/synalx



Example application

The first implementation considered is not reactive and also offers some room for improvement:

```
@Component([…])
export class DessertsComponent implements OnInit {
  #dessertService = inject(DessertService);
  #ratingService = inject(RatingService);
  […]

  originalName = '';
  englishName = '';
  loading = false;
```

```
10
11    desserts: Dessert[] = [];
12
13    ngOnInit(): void {
14      this.search();
15    }
16
17    search(): void {
18      const filter: DessertFilter = {
19        originalName: this.originalName,
20        englishName: this.englishName,
21      };
22
23      this.loading = true;
24
25      this.#dessertService.find(filter).subscribe({
26        next: (desserts) => {
27          this.desserts = desserts;
28          this.loading = false;
29        },
30        error: (error) => { […] },
31      });
32    }
33
34    toRated(desserts: Dessert[], ratings: DessertIdToRatingMap): Dessert[] {
35      return desserts.map((d) =>
36        ratings[d.id] ? { ...d, rating: ratings[d.id] } : d,
37      );
38    }
39
40    loadRatings(): void {
41      this.loading = true;
42
43      this.#ratingService.loadExpertRatings().subscribe({
44        next: (ratings) => {
45          const rated = this.toRated(this.desserts, ratings);
46          this.desserts = rated;
47          this.loading = false;
48        },
49        error: (error) => { […] },
50      });
51    }
52    […]
```

```
53 }
```

Since the properties to be bound are neither Observables nor Signals, the strategy `OnPush` cannot be used for improving the data binding performance. Upon closer inspection, we also notice that the `loadRatings` method updates the `desserts` array, even though its actual task – loading ratings – has nothing to do with it.

Additionally, developers must remember that after any changes to the ratings, the `desserts` array must also be modified. This is exactly what can lead to hard-to-maintain code and hidden bugs – especially when both `desserts` and `ratings` change at different points. Things become even more complex when additional data structures have to be taken into account in these calculations. The first rule of thumb presented here solves this issue.

Rule 1: Derive State Synchronously Wherever Possible

The previously mentioned disadvantages can be compensated for with Signals. Since the introduction of Signals makes the component reactive, `OnPush` can now be activated. In addition, the component can derive its state from the individual Signals using `computed` synchronously:

```
1  @Component({
2    […],
3    changeDetection: ChangeDetectionStrategy.OnPush,
4  })
5  export class DessertsComponent implements OnInit {
6    #dessertService = inject(DessertService);
7    #ratingService = inject(RatingService);
8
9    originalName = signal('');
10   englishName = signal('');
11   loading = signal(false);
12
13   desserts = signal<Dessert[]>([]);
14   ratings = signal<DessertIdToRatingMap>({});
15   ratedDesserts = computed(() => this.toRated(this.desserts(), this.ratings()));
16
17   […]
18
19   loadRatings(): void {
20     this.loading.set(true);
21
22     this.#ratingService.loadExpertRatings().subscribe({
23       next: (ratings) => {
```

```
24          this.ratings.set(ratings);
25          this.loading.set(false);
26        },
27        error: (error) => { […] }
28    });
29
30    […]
31  }
```

This makes the code a lot more straightforward: The `loadRatings` method simply loads the ratings and places them in a signal. The `computed` Signal `ratedDesserts` takes care of mergining `desserts` and `ratings`. No matter when and where the application updates `desserts` or `ratings`, `ratedDesserts` is always up to date.

> Primary usage scenario of Signals: binding values reactively to the view.

When applying this pattern, it is important to note that `computed` can currently only derive state in a synchronous manner. This has to do with the primary usage scenario of Signals: binding values reactively to the view. It's possible that the Angular team will extend Signals to asynchronous scenarios over time. However, at the moment it is necessary to resort to other means. Rule 3 offers a straightforward approach to this.

Rule 2: Avoid Effects for Propagating State

Effects are the right choice when presenting values cannot be achieved via data binding. However, they bring some pitfalls when used for propagating change. In the next sections, I elaborate on this.

Proper Usage of Effects

In most cases, Signals are bound in the template. However, it happens that the desired form of output cannot be achieved via data binding. An example of this is outputting a Signal to the console for debugging purposes. Another example are toasts that can be activated via services and are intended to present the value of a Signal. For these cases, Angular provides Effects:

```
[…]
constructor() {
  effect(() => {
    console.log('originalName', this.originalName());
    console.log('englishName', this.englishName());
  });

  effect(() => {
    this.#toastService.show(this.desserts().length + ' desserts loaded!');
  });
}
[…]
```

Signals are Glitch-free

When writing code like in the previous section, we need to be aware that Signals are glitch-free. That means that if you change a signal several times in a row (within a stack frame), only the last change will be seen by the consumer, e.g. the effect:

```
@Component([...])
export class AboutComponent {

  constructor() {
    const signal1 = signal('A');
    const signal2 = signal('B');

    effect(() => {
      console.log('signal1', signal1());
      console.log('signal2', signal2());
    });

    signal1.set('C');
    signal1.set('D');

    signal1.set('E');

    signal2.set('F');
  }
}
```

In this case, we will only see the values `E` and `F` on the console. Indemediate values are skipped.

This shows that Signals are not intended for modelling events but for data we want to bind to the view. In the latter case, we just need the current value while binding intermediate values would be counterproductive. For this reasons, the effects shown in the previous section are only triggered once even if there are several changes in a row.

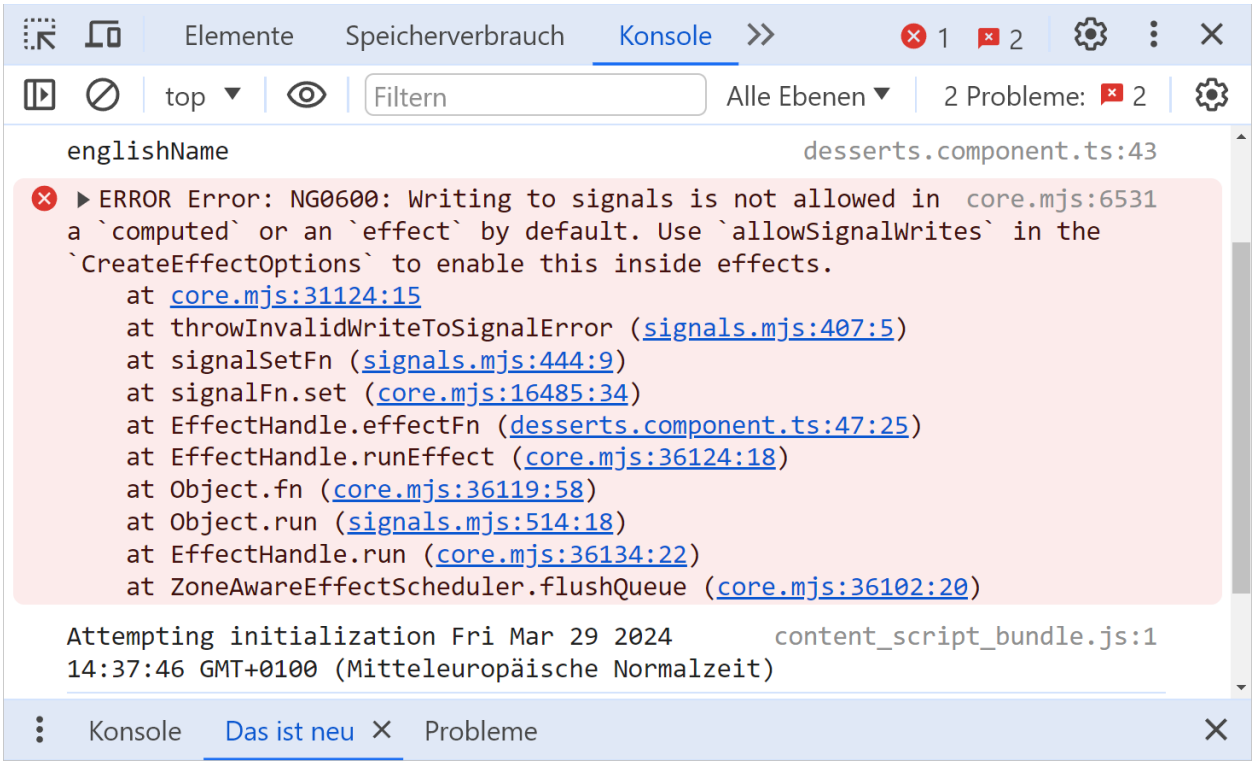
In cases where we want to express events, Observables are the way to go, as they don't have this glitch-free guarantee by design.

Problematic Use of Effects

Even when using Effects, Signals are primarily used to transport the desired data into the view. In theory, however, Effects could also be used to transmit state to other Signals:

```
1  effect(() => {
2      this.originalName.set(this.englishName());
3  });
```

However, such approaches have several disadvantages, which is why Angular prohibits writing Signals within Effects by default:



Error message when trying to set a Signal in an Effect

One of these disadvantages is that unmanageable change cascades and thus difficult-to-maintain code and cyclic dependencies can arise. Since Effects register implicitly with all Signals used, the

associated problems may not even be noticeable at first glance. If you still want to use Effects for writing, you can make Angular let things slide by setting `allowSignalWrites`:

```
// Try hard to avoid this
effect(() => {
    this.originalName.set(this.englishName());
  },
  { allowSignalWrites: true },
);
```

> Application code should only use `allowSignalWrites` as a last resort.

The general consensus in the community is that application code should only use `allowSignalWrites` as a last resort. On the other hand, libraries like NGRX use this option internally. In this case, however, the authors of the library are responsible for its correct use, so application developers don't have to worry about it.

It is also important to note that the Effect itself registers with Signals in called methods too. For instance, the following Effect is triggered when Signals change within `search`:

```
// Try hard to avoid this
effect(() => {
    this.search();
  },
  { allowSignalWrites: true },
);
```

This leads to a further increase in complexity. At least this problem could be alleviated with built-in features:

```
// Try hard to avoid this
effect(() => {
    const originalName = this.originalName();
    const englishName = this.englishName();
    untracked(() => {
        this.load(originalName, englishName);
    })
  }
);
```

The `untracked` function avoids the current reactive context spilling over to the called `search` method. Angular now also uses this pattern itself in selected cases[38]. An example of this is triggering events

[38] https://github.com/angular/angular/pull/54614

in sub-components so that the event handler does not run in the reactive context of the code that triggered the event. Further popular libraries that use this technique are NGRX, NGRX Signal Store or ngextensions.

Strategies for Preventing Effects With Signal Writes

Effects that spread data via Signal writes can in many cases be prevented using the following approaches:

- Consistent derivation of state using `computed` (see rule 1, above).
- Direct use of events that caused the Signal to change.

Instead of calling `search`, as indicated above, in an Effect, the application could instead use the change event of the input fields for the search filters. Observables can also be used as a source for such actions. The `search` method could, for example, also be triggered by the `valueChanges` observable of a `FormGroup`. In cases where you have just Signals, they can be converted into Observables using the RxJS Interop offered by Angular:

```
@Component([…])
export class DessertsComponent {
  #dessertService = inject(DessertService);
  #ratingService = inject(RatingService);
  #toastService = inject(ToastService);

  originalName = signal('');
  englishName = signal('Cake');
  loading = signal(false);

  originalName$ = toObservable(this.originalName);
  englishName$ = toObservable(this.englishName);

  desserts$ = combineLatest({
    originalName: this.originalName$,
    englishName: this.englishName$,
  }).pipe(
    filter((c) => c.originalName.length >= 3 || c.englishName.length >= 3),
    debounceTime(300),
    tap(() => this.loading.set(true)),
    switchMap((c) => this.findDesserts(c)),
    tap(() => this.loading.set(false)),
  );

```

```
25    desserts = toSignal(this.desserts$, {
26      initialValue: [],
27    });
28
29    ratings = signal<DessertIdToRatingMap>({});
30    ratedDesserts = computed(() => this.toRated(this.desserts(), this.ratings()));
31
32    findDesserts(c: DessertFilter): Observable<Dessert[]> {
33      return this.#dessertService.find(c).pipe(
34        catchError((error) => {
35          this.#toastService.show('Error loading desserts!');
36          console.error(error);
37          return of([]);
38        }),
39      );
40    }
41    […]
42  }
```

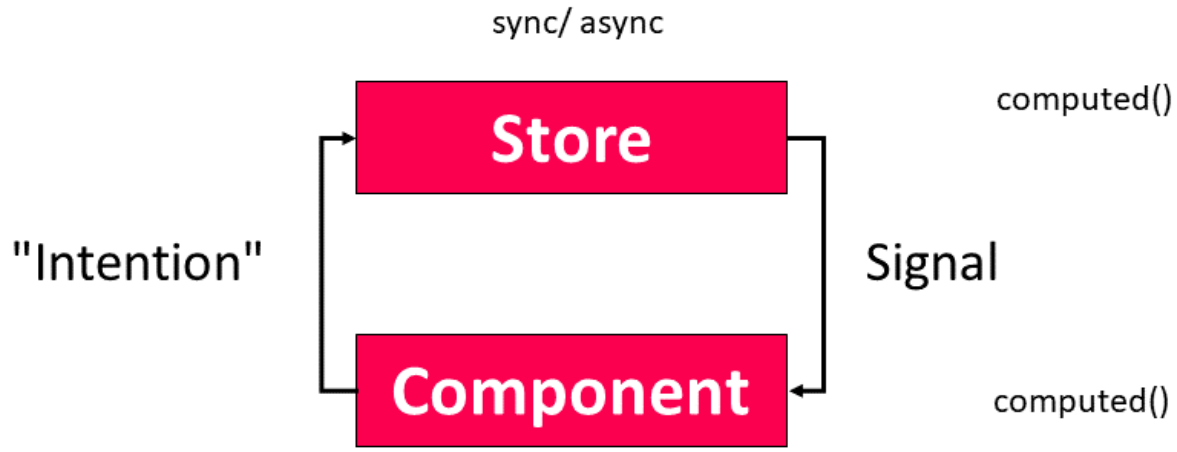
> The flattening operators offered by RxJS provide guarantees for overlapping asynchronous actions and thus prevent race conditions.

The use of Observables has several advantages at this point: - In contrast to Signals, Observables are also suitable for triggering asynchronous actions. - `toObservable` function strips the current reactive context using `untracked`. - RxJS comes with a lot of powerful operators, like `debounceTime`. - The flattening operators offered by RxJS provide guarantees for overlapping asynchronous actions and thus prevent race conditions. In the example shown, `switchMap` ensures that when search queries overlap, only the result of the last one is used and all others are canceled.

In many cases, one could argue that instead of converting a Signal into an Observable, it would be more appropriate to directly use the event that led to the Signal change, as proposed above. On the other hand, as Angular APIs increasingly adopt Signal-based approaches, using them directly will likely become more convenient and feel more intuitive. Therefore, this appears to be a gray area where we need to be mindful of the consequences, such as those associated with the glitch-free guarantee of Signals.

Rule 3: Stores Simplify Reactive Data Flow

Stores like the classic NGRX Store or the lightweight NGRX Signal Store not only take care of state management, but also help to keep the reactive data flow manageable:



Unidirectional data flow with a store

The application forwards its intention to the store as part of an event. At this point, I use the term intention in an abstract, technology-neutral way, especially since different stores realize this aspect differently. With Redux and therefore also when using the classic NGRX store the application sends an action to the store, which forwards it to Reducer and Effects. For lightweight stores like the NGRX Signal Store, the application delegates to a method offered by the store instead.

> Offloading asynchronous operations to the store also compensates for the fact that Signals are currently only designed for synchronous actions.

The store then takes action and initiates synchronous or asynchronous operations. If the application uses RxJS for this, race conditions can be avoided with the flattening operators, as mentioned above. Offloading asynchronous operations to the store also compensates for the fact that Signals are currently only designed for synchronous actions.

The result of these operations leads to a change in the state managed by the store. This state can be expressed by Signals, which can be mapped to other Signals using computing (see rule 1). Such mappings can occur both in the store and in the component (or in another consumer of the store). This depends on how local or global the store and the data to be derived are.

The bottom line is that the consistent use of this approach supports the so-called unidirectional data flow, which makes system behavior more understandable. The following listing demonstrates this from the perspective of a component that relies on the NGRX Signal Store.

```
@Component([…])
export class DessertsComponent {
  #store = inject(DessertStore);

  originalName = this.#store.filter.originalName;
  englishName = this.#store.filter.englishName;

  ratedDesserts = this.#store.ratedDesserts;
  loading = this.#store.loading;

  constructor() {
    this.#store.loadDesserts();
  }

  search(): void {
    this.#store.loadDesserts();
  }

  loadRatings(): void {
    this.#store.loadRatings();
  }

  updateRating(id: number, rating: number): void {
    this.#store.updateRating(id, rating);
  }

  updateFilter(filter: DessertFilter): void {
    this.#store.updateFilter(filter);
  }
}
```

Since the component only has to delegate to the store, it is very straightforward.

Conclusion

To truly leverage the benefits of Signals, the application must be designed as a reactive system. This means, among other things, that writing values is avoided in favor of deriving values from existing ones. This simplifies the program code, especially since derived values are automatically kept up to date.

Signals are currently primarily suitable for transporting data into the view. Effects are used when API calls are necessary for that, e.g. when displaying a toast. The current Signals implementation is

not intended to trigger asynchronous actions. Instead, classic events or observables are the way to go. Stores that can also handle asynchronous operations help establish unidirectional data flow and make reactive applications more manageable.

Built-in Control Flow and Deferrable Views

Angular 17 introduced a new template syntax for control flow bocks. Also, the same syntax is used to provide deferrable views. These are lazy loaded parts of a page that help to improve the initial load performance.

New Syntax for Control Flow in Templates

Since its early days, Angular has used structural directives like **ngIf* or **ngFor* for control flow. Since the control flow needs to be revised anyway to allow for the envisioned fine-grained change detection and for eventually going Zone-less, the Angular team has decided to put it on a new footing. The result is the new build-in control flow, which stands out clearly from the rendered markup:

```
@for (product of products(); track product.id) {
    <div class="card">
        <h2 class="card-title">{{product.productName}}</h2>
        […]
    </div>
}
@empty {
    <p class="text-lg">No Products found!</p>
}
```

One thing worth noting here is the new *@empty* block that Angular renders when the collection being iterated is empty.

Although signals were a driver for this new syntax, they are not a requirement for its use. The new control flow blocks can also be used with classic variables or with observables in conjunction with the *async* pipe.

The mandatory *track* expression allows Angular to identify individual elements that have been moved within the iterated collection. This enables Angular (to be more precise: Angular's new reconciliation algorithm) to drastically reduce the rendering effort and to reuse existing DOM nodes. When iterating over collections of primitive types, e.g. Arrays with numbers or strings, *track* could point to the pseudo variable *$index* according to information from the Angular team:

```
@for (group of groups(); track $index) {
    <a (click)="groupSelected(group)">{{group}}</a>
    @if (!$last) {
        <span class="mr-5 ml-5">|</span>
    }
}
```

In addition to *$index*, the other values known from **ngFor* are also available via pseudo variables: *$count*, *$first*, *$last*, *$even*, *$odd*. If necessary, their values can be stored in template variables too:

```
@for (group of groups(); track $index; let isLast = $last) {
    <a (click)="groupSelected(group)">{{group}}</a>
    @if (!isLast) {
        <span class="mr-5 ml-5">|</span>
    }
}
```

The new *@if* simplifies the formulation of *else*/ *else-if* branches:

```
@if (product().discountedPrice && product().discountMinCount) {
    […]
}
@else if (product().discountedPrice && !product().discountMinCount) {
    […]
}
@else {
    […]
}
```

In addition, different cases can also be distinguished using a *@switch*:

```
@switch (mode) {
    @case ('full') {
          […]
    }
    @case ('small') {
      […]
    }
    @default {
      […]
    }
}
```

Unlike *ngSwitch* and **ngSwitchCase*, the new syntax is type-safe. In the example shown above, the individual *@case* blocks must have string values, since the *mode* variable passed to *@switch* is also a string.

The new control flow syntax reduces the need to use structural directives, which are powerful but sometimes unnecessarily complex. Nevertheless, the framework will continue to support structural directives. On the one hand, there are some valid use cases for it and on the other hand, despite the many exciting innovations, the framework needs to be made backwards compatible.

Automatic Migration to Build-in Control Flow

If you would like to automatically migrate your program code to the new control flow syntax, you will now find a schematic for this in the `@angular/core package`:

```
ng g @angular/core:control-flow
```

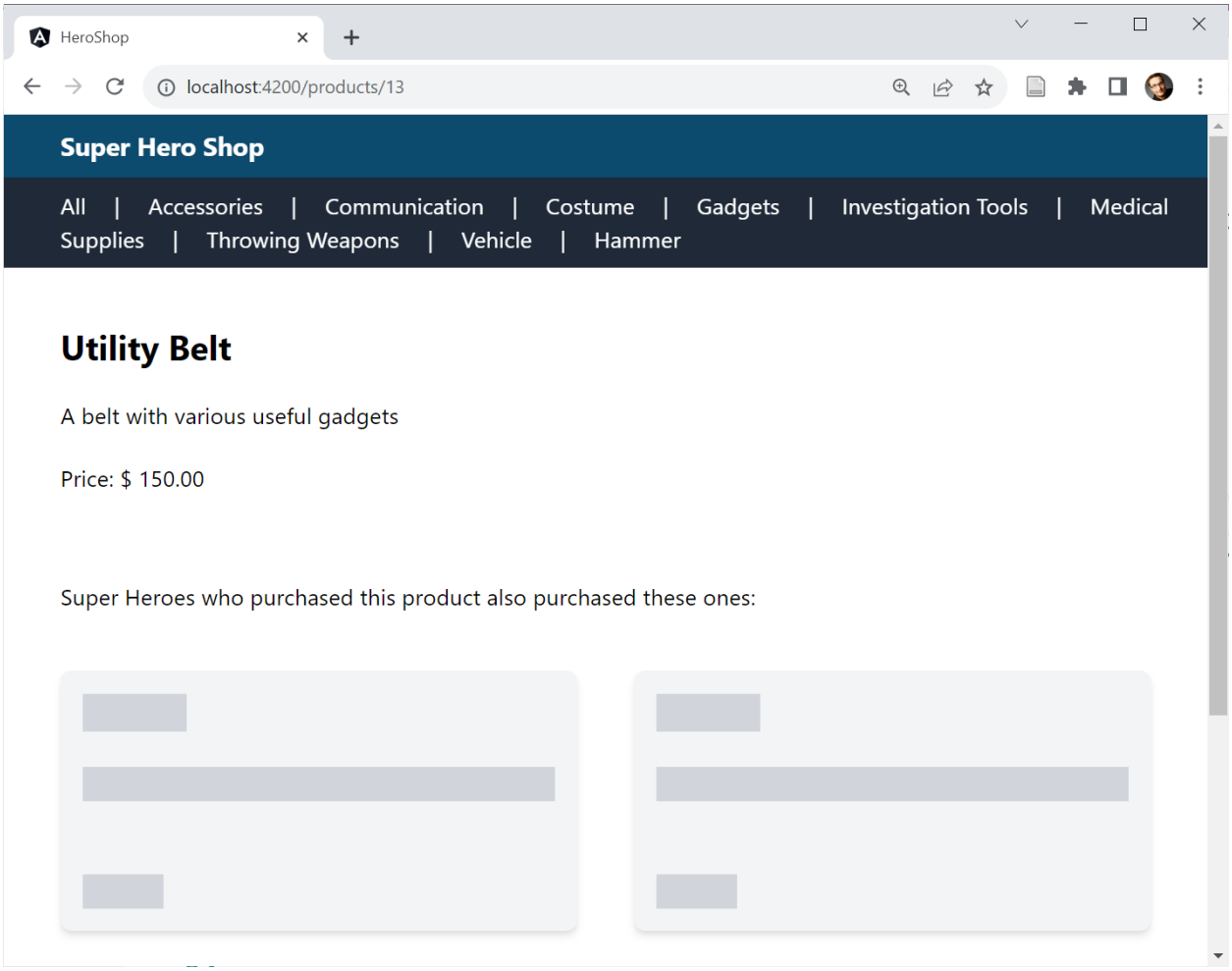
Delayed Loading

Typically, not all areas of a page are equally important. A product page is primarily about the product itself. Suggestions for similar products are secondary. However, this changes suddenly as soon as the user scrolls the product suggestions into the visible area of the browser window, the so-called view port.

For particularly performance-critical web applications such as web shops, it makes sense to defer loading less important page parts. This means that the really important elements are available more quickly. Until now, anyone who wanted to implement this idea in Angular had to do it manually. Angular 17 also dramatically simplifies this task with the new *@defer* block:

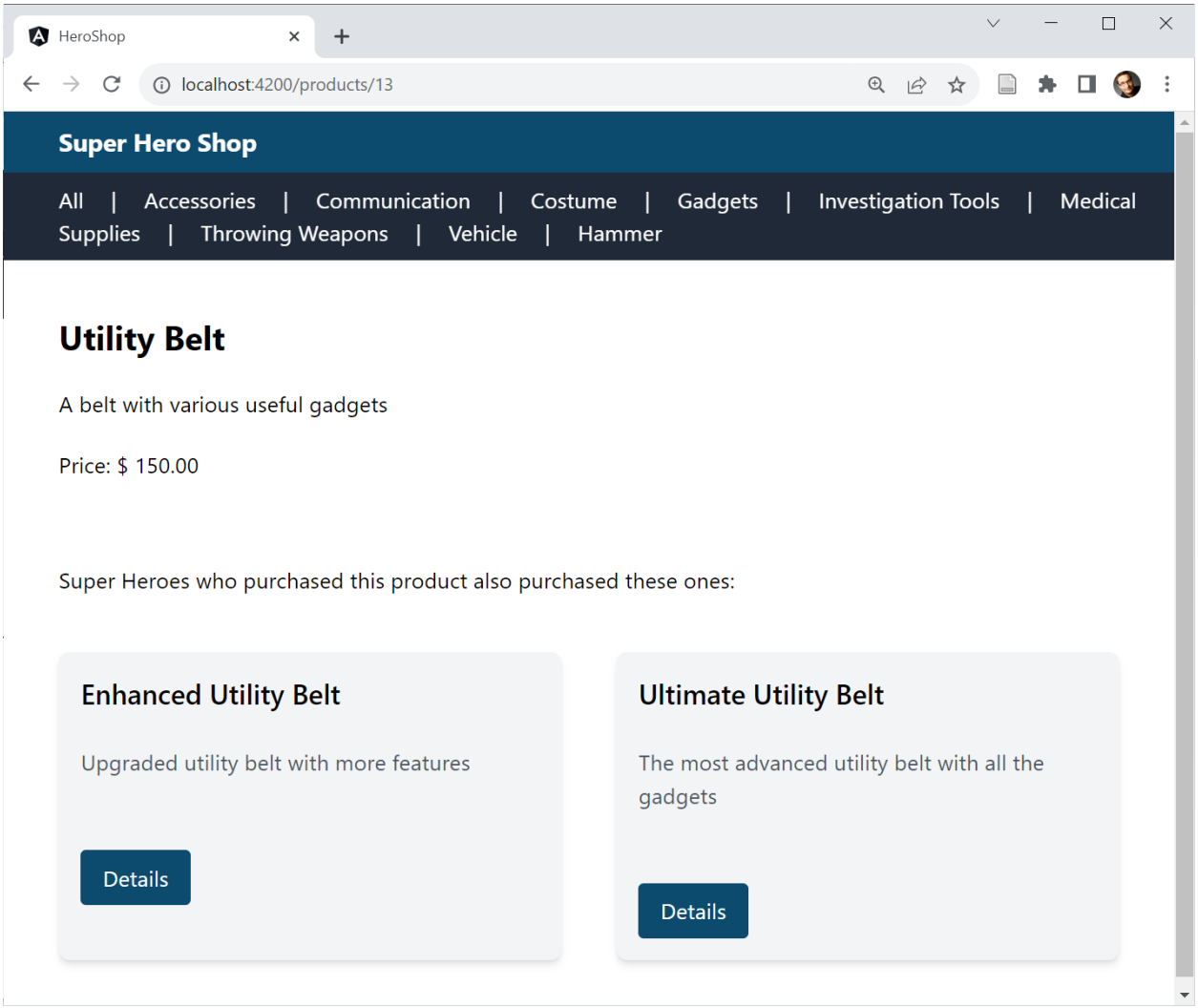
```
@defer (on viewport) {
    <app-recommentations [productGroup]="product().productGroup">
        </app-recommentations>
}
@placeholder {
    <app-ghost-products></app-ghost-products>
}
```

Using *@defer* delays the loading of the enclosed page part until a certain event occurs. As a replacement, it presents the placeholder specified under *@placeholder*. In the demo application used here, ghost elements are first presented for the product suggestions:



Ghost Elements as placeholders

Once loaded, *@defer* swaps the ghost elements for the actual suggestions:



@defer swaps the placeholder for the lazy-loaded component

In the discussed example, the *on viewport* event is used. It occurs once the placeholder has been scrolled into view. Besides this event, there are several other options too:

Triggers	Description
on idle	The browser reports that there are no critical tasks pending (default).
on viewport	The placeholder is loaded into the visible area of the page.
on interaction	The user begins to interact with the placeholder.
on hover	The mouse cursor is moved over the placeholder.
on immediate	As soon as possible after the page loads.
on timer(duration)	After a certain time, e.g. on timer(5s) to trigger loading after 5 seconds.
when condition	Once the specified condition is met, e.g. when (userName !=== null)

By default, *on viewport*, *on interaction*, and *on hover* force a *@placeholder* block to be specified. Alternatively, they can also refer to other page parts that can be referenced via a template variable:

```
<h1 #recommentations>Recommentations</h1>
@defer (on viewport(recommentations)) {
    <app-recommentations […] />
}
```

Additionally, *@defer* can be told to preload the bundle at an earlier time. As with preloading routes, this approach ensures that bundles are available as soon as you need them:

```
@defer(on viewport; prefetch on immediate) { […] }
```

In addition to *@placeholder*, *@defer* also offers two other blocks: *@loading* and *@error*. Angular displays the former one while it loads the bundle; the latter one is shown in the event of an error. To avoid flickering, *@placeholder* and *@loading* can be configured with a minimum display duration. The *minimum* property sets the desired value:

```
@defer ( […] ) {
    […]
}
@loading (after 150ms; minimum 150ms) {
    […]
}
@placeholder (minimum 150ms) {
    […]
}
```

The *after* property also specifies that the loading indicator should only be displayed if loading takes longer than 150 ms.

Conclusion

The new Built-in Control Flow Blocks stand out visually from the template's markup. The also pave the way for incrementally updating components. Deferrable Views use the same syntax and allow to postpone loading specific elements of a page to improve loading times.

esbuild and the new Application Builder

The new esbuild support provides faster builds. The subsequent `ApplicationBuilder` streamlines the use of SSR.

Build Performance with esbuild

Originally, the Angular CLI used webpack to generate bundles. However, webpack is currently being challenged by newer tools that are easier to use and a lot faster. esbuild[39] is one of these tools that, with over 20,000 downloads per week, has a remarkable distribution.

The CLI team has been working on an esbuild integration for several releases. In Angular 16, this integration was already included as a developer preview. As of Angular 17, this implementation is stable and used by default for new Angular projects via the *Application Builder* described below.

For existing projects, it is worth considering switching to esbuild. To do this, update the *builder* entry in *angular.json*:

```
1  "builder" : "@angular-devkit/build-angular:browser-esbuild"
```

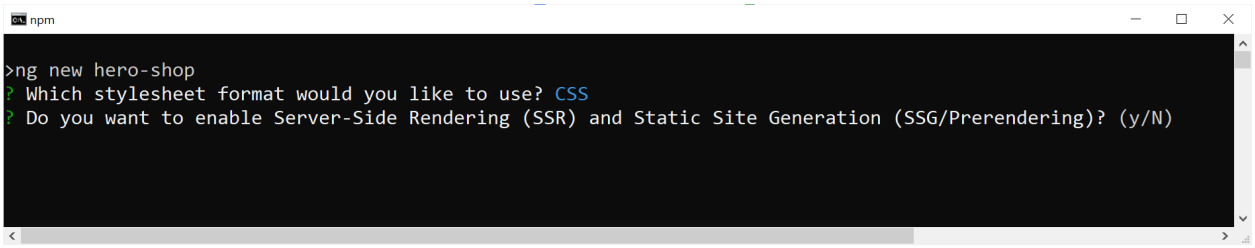
In other words: *-esbuild* must be added at the end. In most cases, *ng serve* and *ng build* should behave as usual, but be a lot faster. The former uses the vite[40] dev server to speed things up by only building npm packages when needed. In addition, the CLI team integrated several additional performance optimizations.

Calling *ng build* is also drastically accelerated by using esbuild. Factor 2 to 4 is often mentioned as the range.

SSR Without Effort with the new Application Builder

Support for server-side rendering (SSR) has also been drastically simplified with Angular 17. When generating a new project with *ng new, a –ssr* switch is now available. If this is not used, the CLI asks whether it should set up SSR:

[39] https://esbuild.github.io
[40] https://vitejs.dev/



ng new sets up SSR if desired

To enable SSR later, all you need to do is to add the *@angular/ssr package*:

```
ng add @angular/ssr
```

The *@angular* scope makes clear, this package comes directly from the Angular team. It is the successor to the community project Angular Universal. To directly take SSR into account during *ng build* and *ng serve*, the CLI team has provided a new builder. This so-called application builder uses the esbuild integration mentioned above and creates bundles that can be used both in the browser and on the server side.

A call to *ng serve* also starts a development server, which both renders on the server side and delivers the bundles for operation in the browser. A call to *ng build –ssr* also takes care of bundles for both worlds as well as building a simple Node.js-based server whose source code uses the schematics mentioned above.

If you can't or don't want to run a Node.js server, you can use *ng build –prerender* to prerender the individual routes of the application during build.

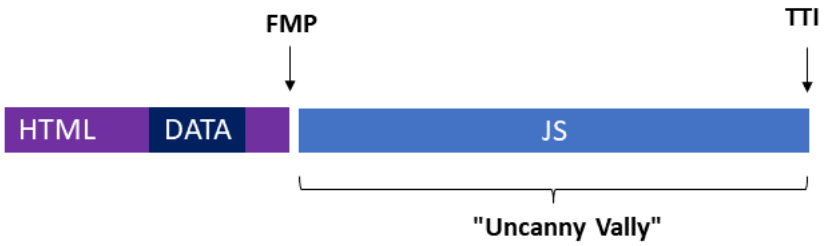
More than SSR: Non-destructive Hydration

Single page applications (SPAs) enable good runtime performance. However, the initial page load usually takes a few seconds longer than with classic web applications. This is because the browser has to load large amounts of JavaScript code in addition to the actual HTML page before it can render the page. The so-called First Meaningful Paint (FMP) only takes place after a few seconds:

Client-side rendering of a SPA

While these few seconds are hardly an issue in business applications, they actually pose a problem for public web solutions such as web shops. Here it is important to keep the bounce rate low and this can be achieved, among other things, by keeping waiting times as short as possible.

It is therefore common to render SPAs for such scenarios on the server side so that the server can already deliver a finished HTML page. The caller is thus quickly presented with a page. Once the JavaScript bundles have loaded, the page is also interactive. The next image illustrates this: The First Meaningful Paint (FMP) now takes place earlier. However, the site will only become interactive later (Time to Interactive, TTI).



Server-side rendering of an SPA

To support solutions where the initial page load matters, Angular has offered server-side rendering (SSR) since its early days. However, the behavior of this SSR implementation has been "destructive" in the past. This means that the loaded JavaScript code re-rendered the entire page. All server-side rendered markup was replaced with client-side rendered markup. Unfortunately, this is also accompanied by a slight delay and flickering. Metrics show that this degrades startup performance.

Angular 16 also addresses this issue by reusing the already server-side rendered markup from the JavaScript bundles loaded into the browser. We are talking about non-destructive hydration here. The word hydration describes the process that makes a loaded page interactive using JavaScript.

To use this new feature, first install the `@nguniversal/express-engine` package for SSR support:

```
ng add @nguniversal/express-engine
```

After that, non-destructive hydration is enabled with the standalone API `provideClientHydration`:

```
// app.config.ts
export const appConfig: ApplicationConfig = {
  providers: [
    provideClientHydration(),
  ]
};
```

The listing sown takes care of this in the `app.config.ts` file . The structure of the `ApplicationConfig` type published there is used in the `main.ts` file when bootstrapping the application. Incidentally, the `app.config.ts` file is set up by the CLI when a new application is set up with the `--standalone` switch.

To debug an application that relies on SSR or hydration, the using schematics set up the npm script `ssr:dev`:

```
npm run ssr:dev
```

Behind it is a development server that was developed by an extremely charming Austrian collaborator and runs the application in debug mode on both the server and client side.

More Details on Hydration in Angular

If the SPA calls Web APIs via HTTP during server-side rendering, the responses received are also automatically sent to the browser via a JSON fragment within the rendered page. When hydrating, the `HttpClient` in the browser uses this fragment instead of making the same request again. With this, Angular speeds up hydration. If this behavior is not desired, it can be deactivated with the `withNoHttpTransferCache` function:

```
provideClientHydration(
    withNoHttpTransferCache()
),
```

For non-destructive hydration to work, the markup rendered on the server side must match the markup on the client side. This cannot always be guaranteed, especially with third-party components or when using libraries that manipulate the DOM directly. In this case, however, non-destructive hydration can be deactivated for individual components with the `ngSkipHydration` attribute:

```
<app-flight-card
      ngSkipHydration
      [item]="f"
      [(selected)]="basket()[f.id]" />
```

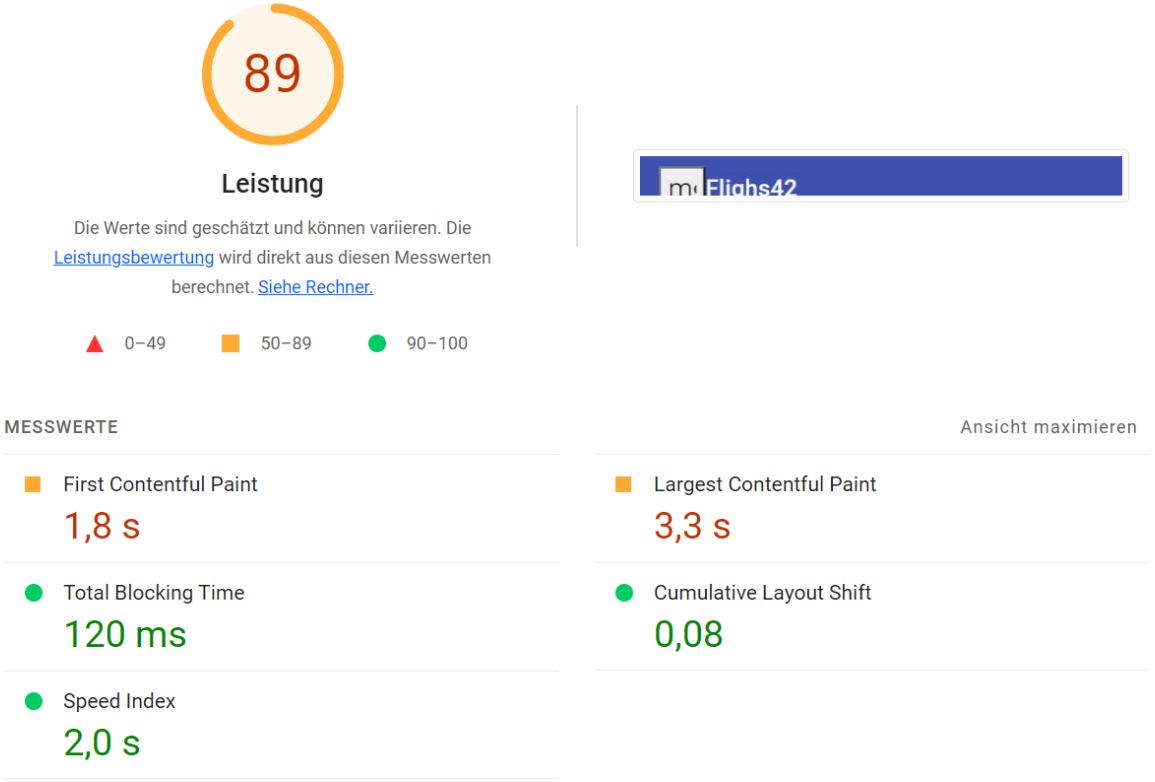
Angular does not allow data binding for this attribute. Also, Angular expects `ngSkipHydration` to be either zero or `true`. If you want to generally exclude hydration for a component, you can also set this attribute via a host binding:

```
@Component({
  […]
  host: { 'ngSkipHydration': 'true' }
})
```

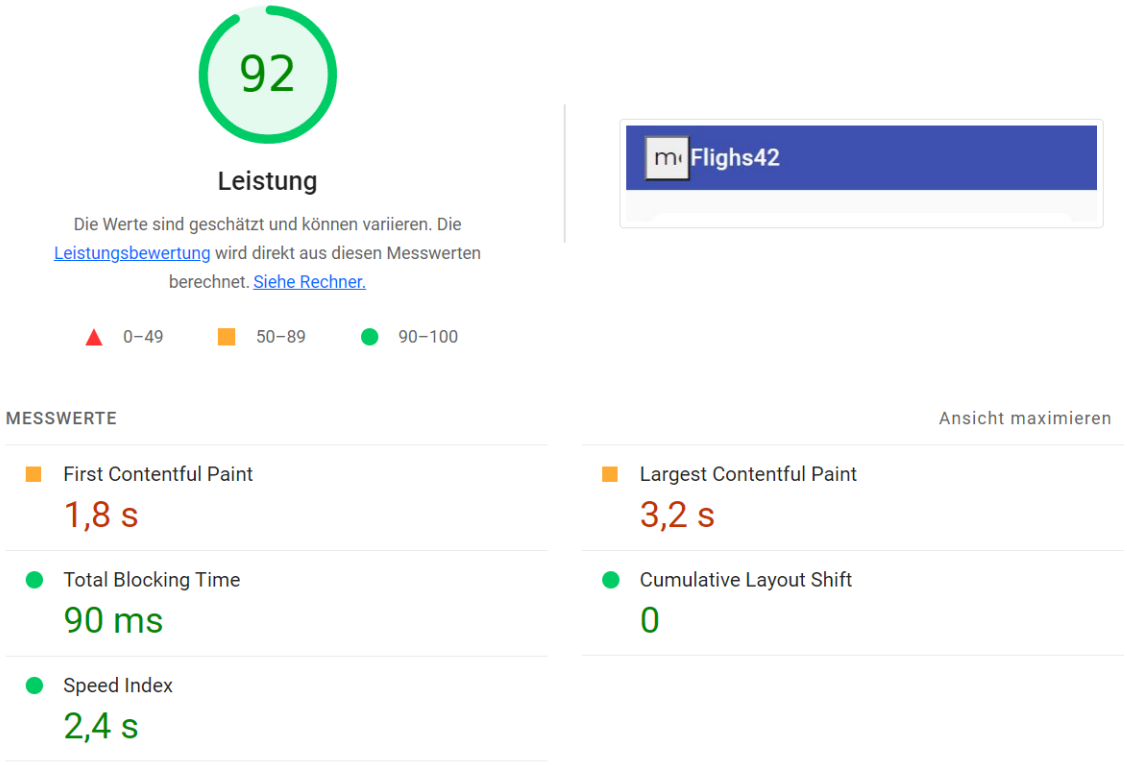
If several Angular applications run in parallel on one side, Angular must be able to distinguish between these applications using an ID. The token `APP_ID` is used for this:

```
{ provide: APP_ID, useValue: 'myApp' },
```

The result of the new non-destructive hydration is quite impressive. The following two figures show some Lighthouse metrics for the example application used here. The former refers to classic SSR and the latter to the new non-destructive hydration.



Classic SSR



SSR with Non-Destructive Hydration

Apart from creating a production build and enabling HTTP compression in the node-based web server responsible for server-side rendering, no optimizations have been implemented.

Conclusion

The new esbuild-based `ApplicationBuilder` accelerates the build process noticeably and directly supports Server-side Rendering (SSR). Together with Hydration, SSR improves the startup times of a page.

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[41] and Facebook[42] and find his blog here[43].

[41] https://twitter.com/ManfredSteyer
[42] https://www.facebook.com/manfred.steyer
[43] http://www.softwarearchitekt.at

Trainings and Consulting

If you and your team need support or trainings regarding Angular, we are happy to help with **workshops and consulting** (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
- Modern Angular Workshop
- Angular Micro Frontends Workshop
- Angular Testing Workshop (Cypress, Jest, etc.)
- Angular Performance Workshop
- Angular Design Systems Workshop (Figma, Storybook, etc.)
- Angular: Reactive Architectures (RxJS and NGRX)
- Angular Review Workshop
- Angular Upgrade Workshop

Please find the full list of our offers here[44].

Modern Angular Workshop

Modern Angular (English)[45] | Modern Angular (German)[46]

We provide our workshops and consulting 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 at office@softwarearchitekt.at.

[44] https://www.angulararchitects.io/en/angular-workshops/
[45] https://www.angulararchitects.io/en/training/modern-angular-workshop/
[46] https://www.angulararchitects.io/training/modern-angular-workshop/