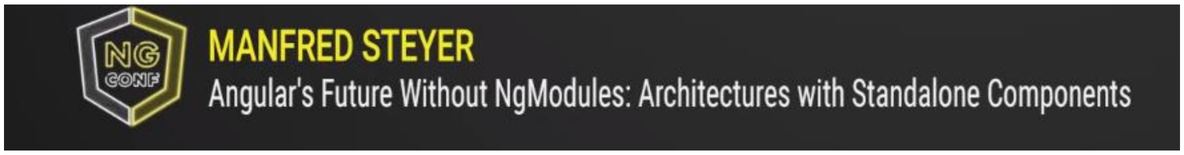
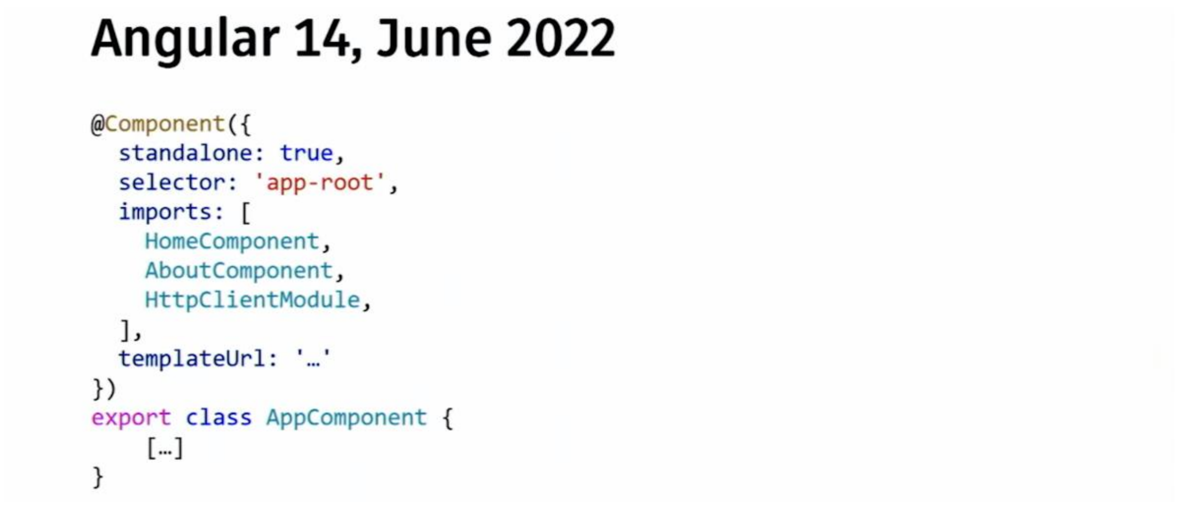


MANFRED STEYER
Angular's Future Without NgModules: Architectures with Standalone Components

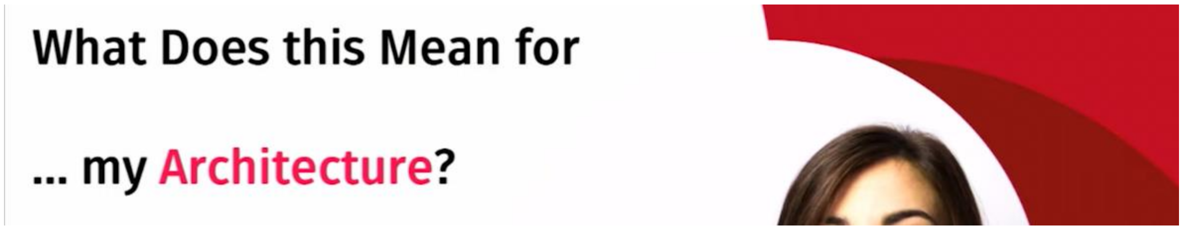
Thanks to standalone components, Angular applications will no longer need NgModules in future. That makes them more straightforward and lightweight. While the principles behind this new feature are quickly understood, the really interesting question is: How can applications be structured without NgModules? This question is answered here. After showing the basics and mental model of standalone components (pipes and directives), you will see several approaches for structuring your application and for grouping related building blocks. We also go into edge cases for lazy loading and the use of existing libraries based on NgModules. We discuss the interaction with tree-shakeable providers and how you can convert existing solutions step-by-step into standalone components. By the end you know, how to improve your architectures with standalone components.



Angular 14, June 2022

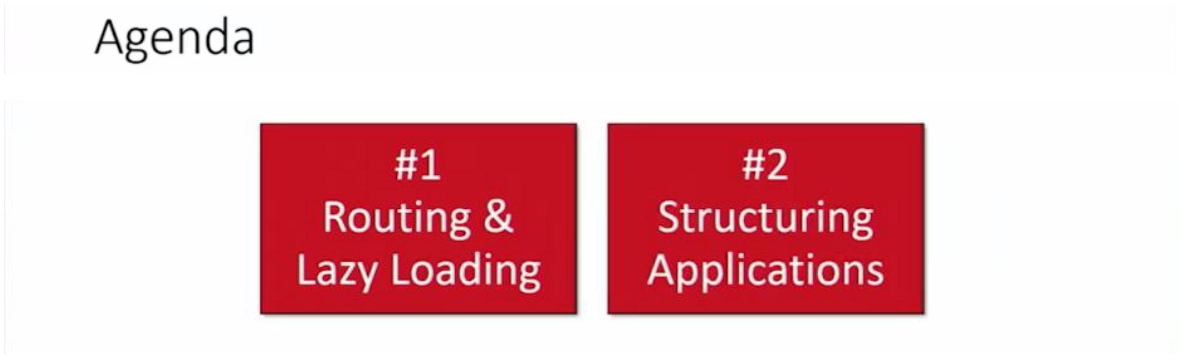
```
@Component({
  standalone: true,
  selector: 'app-root',
  imports: [
    HomeComponent,
    AboutComponent,
    HttpClientModule,
  ],
  templateUrl: '…'
})
export class AppComponent {
    […]
}
```

The Angular team switched out the compiler to Ivy and everything still works, they now made NgModule optional with no breaking changes to allow standalone components.



What Does this Mean for

… my Architecture?



Agenda

#1 Routing & Lazy Loading

#2 Structuring Applications

The Router is the lynchpin of your application that holds together all the parts and thus influence your architecture. You can now structure your applications with standalone components, this is modularization without using ng-modules.

About me...

Manfred Steyer

Manfred Steyer, ANGULARarchitects.io

ANGULAR ARCHITECTURES FOR ENTERPRISE-APPLICATIONS
Google Developers Experts 2019
Angular GDE

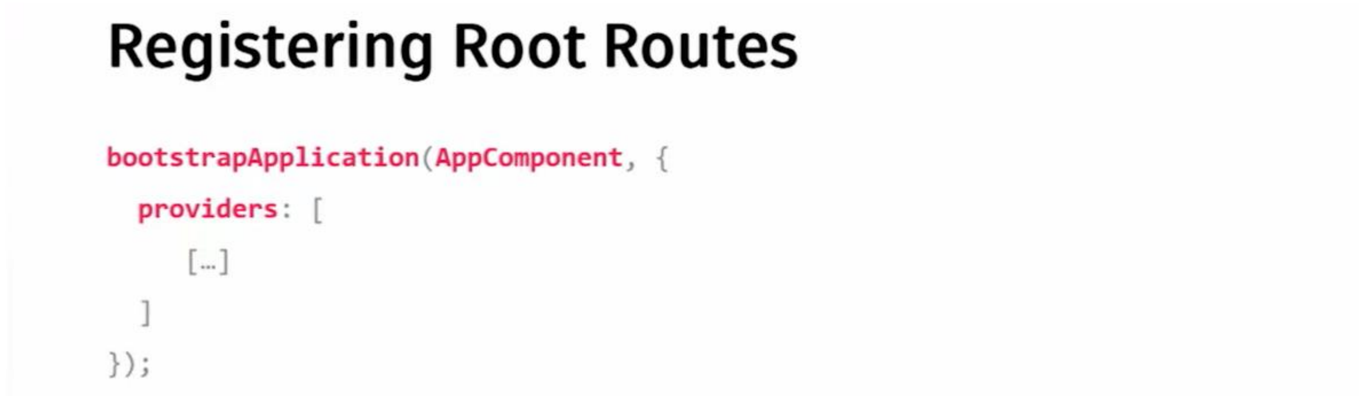
(Remote) Angular Workshops and Consulting

http://angulararchitects.io

Google Developer Expert for Angular

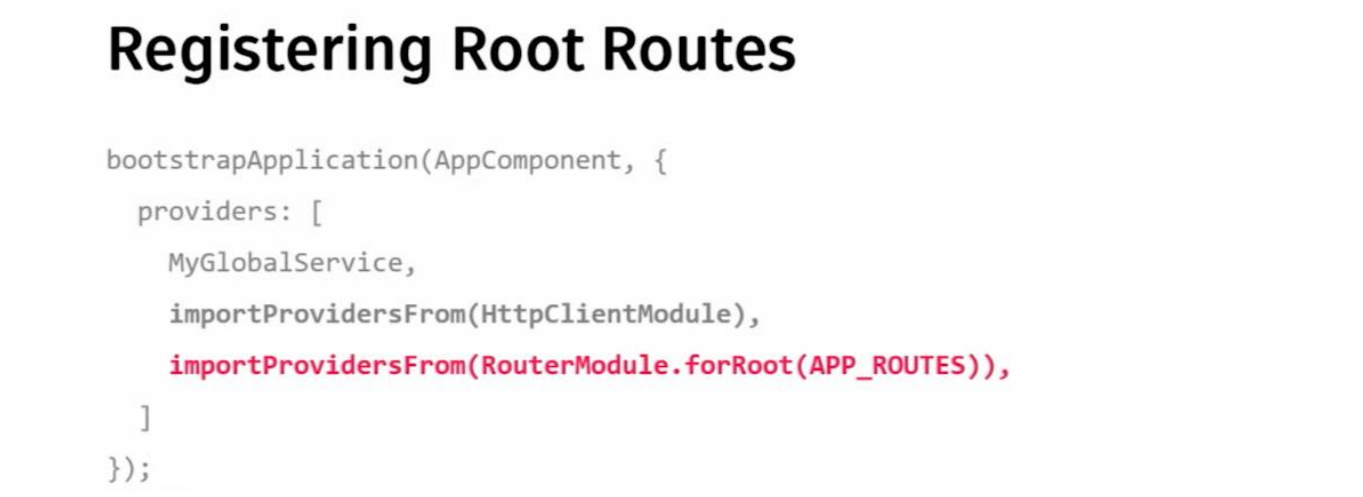
Trusted Collaborator in the Angular Team

#1: Routing & Lazy Loading without NgModules


Registering Root Routes

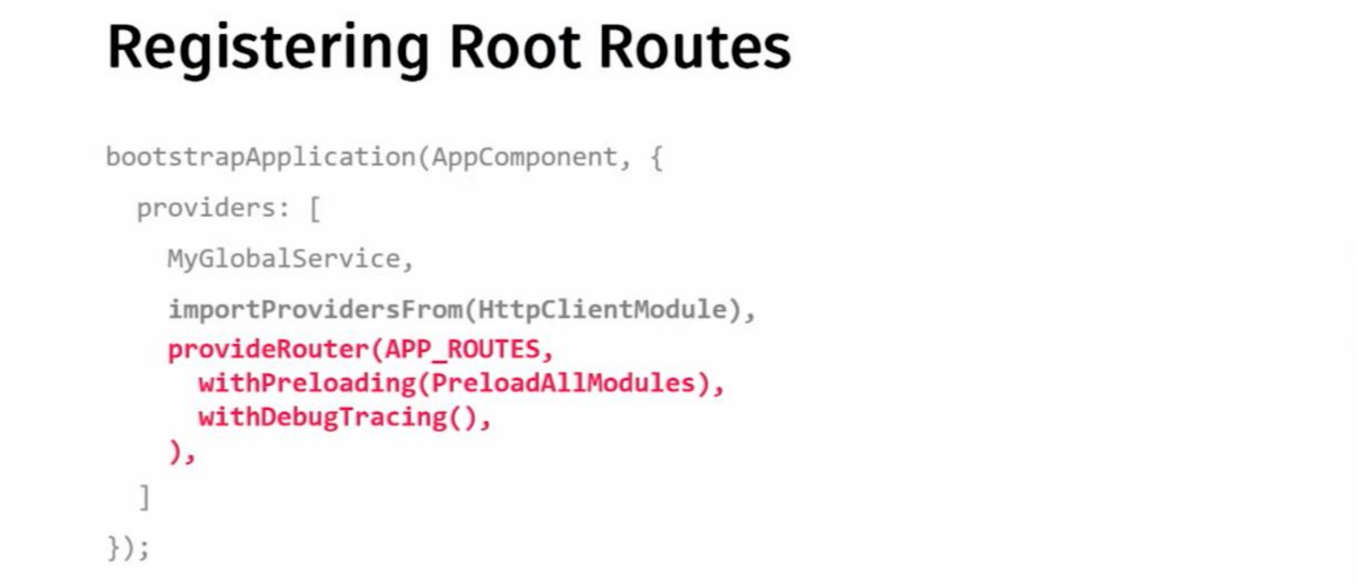
```
bootstrapApplication(AppComponent, {
  providers: [
     [...]
  ]
});
```

When building a standalone component, you don't need the ngModule anymore.


Registering Root Routes

```
bootstrapApplication(AppComponent, {
  providers: [
    MyGlobalService,
    importProvidersFrom(HttpClientModule),
    importProvidersFrom(RouterModule.forRoot(APP_ROUTES)),
  ]
});
```

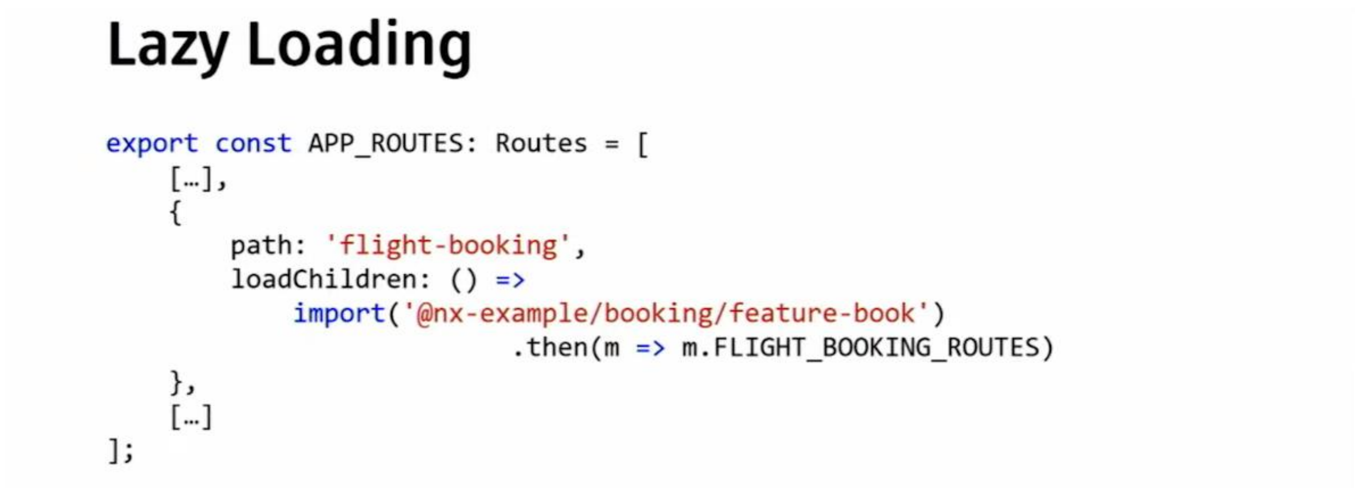
You can import providers from existing modules into your standalone function as above



Registering Root Routes

```
bootstrapApplication(AppComponent, {
  providers: [
    MyGlobalService,
    importProvidersFrom(HttpClientModule),
    provideRouter(APP_ROUTES,
      withPreloading(PreloadAllModules),
      withDebugTracing(),
    ),
  ]
});
```

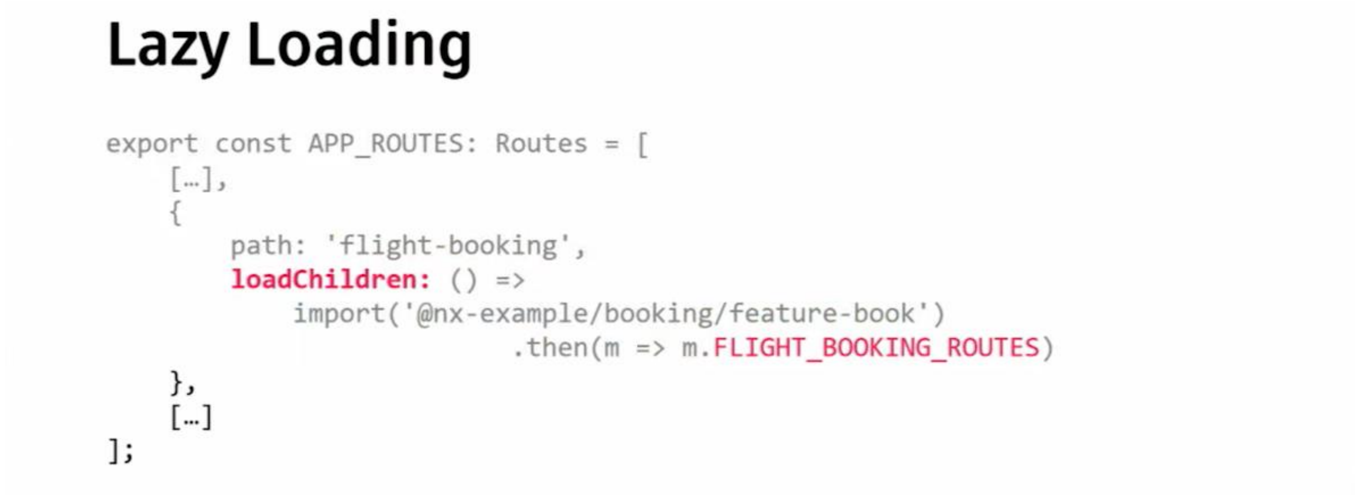
You can also use helper functions using the provideRouter approach in your standalone component as above, it helps treeshaking too



Lazy Loading

```
export const APP_ROUTES: Routes = [
    […],
    {
        path: 'flight-booking',
        loadChildren: () =>
            import('@nx-example/booking/feature-book')
                          .then(m => m.FLIGHT_BOOKING_ROUTES)
    },
    […]
];
```

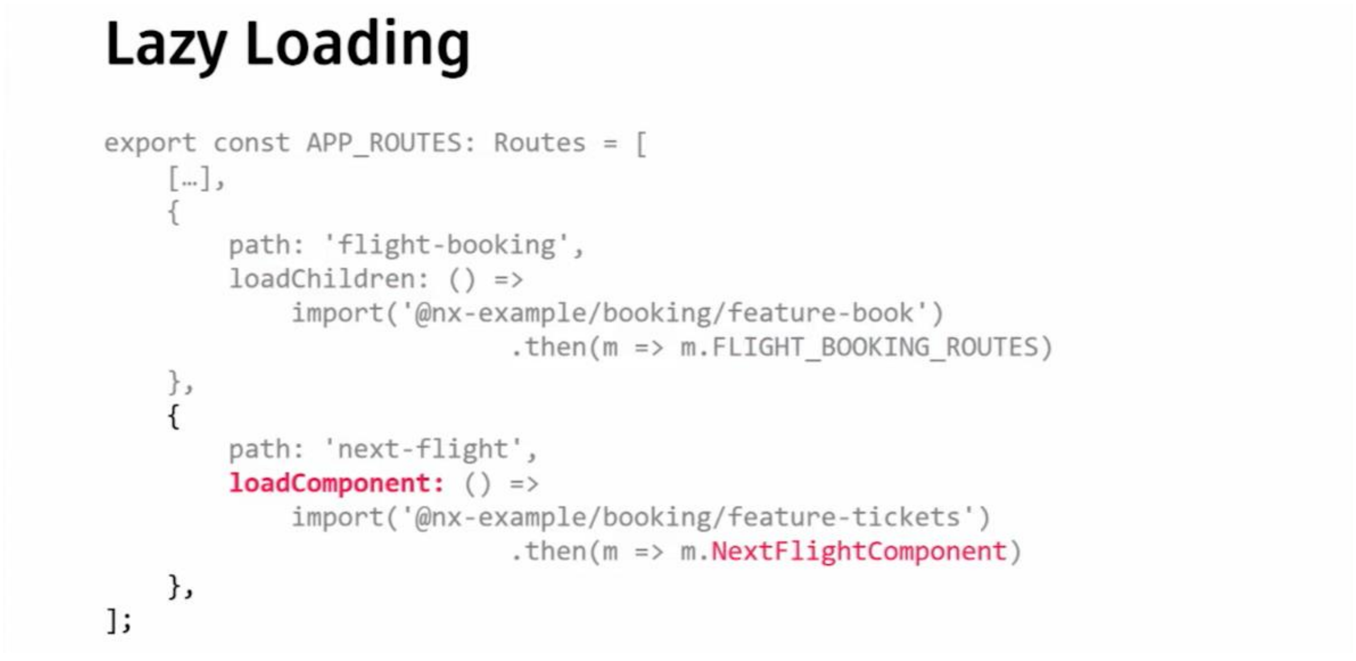
We can now directly point to a lazy router configuration as above



Lazy Loading

```
export const APP_ROUTES: Routes = [
    […],
    {
        path: 'flight-booking',
        loadChildren: () =>
            import('@nx-example/booking/feature-book')
                          .then(m => m.FLIGHT_BOOKING_ROUTES)
    },
    […]
];
```

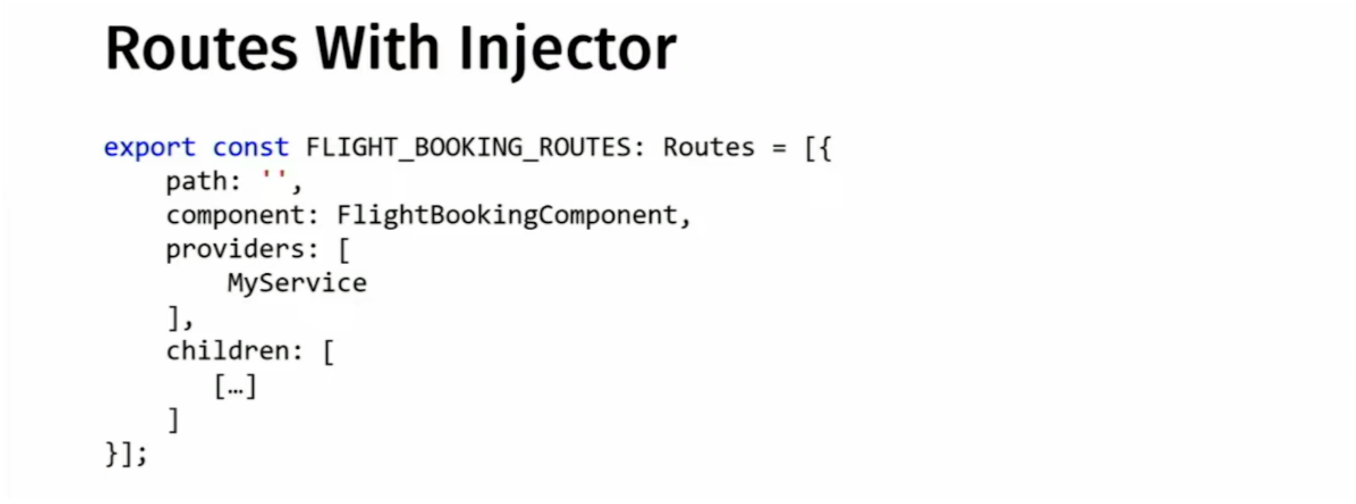
Now with **loadChildren**, we can directly point to our router configuration to the configuration with the child routes.



Lazy Loading

```
export const APP_ROUTES: Routes = [
    […],
    {
        path: 'flight-booking',
        loadChildren: () =>
            import('@nx-example/booking/feature-book')
                        .then(m => m.FLIGHT_BOOKING_ROUTES)
    },
    {
        path: 'next-flight',
        loadComponent: () =>
            import('@nx-example/booking/feature-tickets')
                        .then(m => m.NextFlightComponent)
    },
];
```

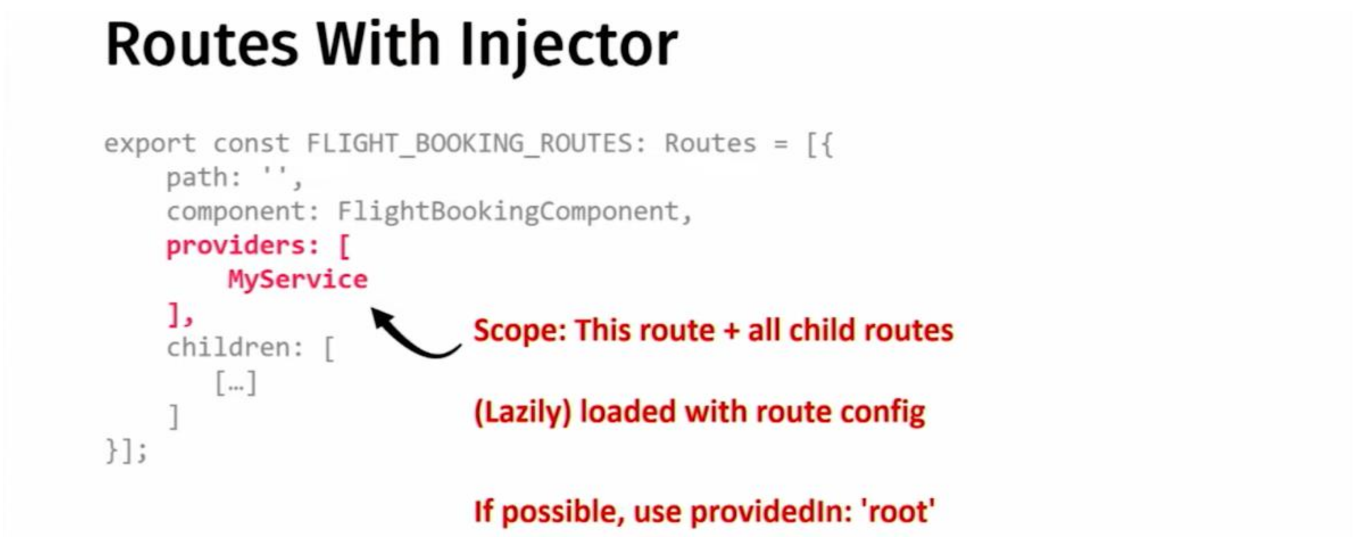
We can also use **loadComponent** to directly point to another standalone component that we want to lazy load.



Routes With Injector

```
export const FLIGHT_BOOKING_ROUTES: Routes = [{
    path: '',
    component: FlightBookingComponent,
    providers: [
        MyService
    ],
    children: [
       […]
    ]
}];
```

We can use a routing configuration as above; it has child routes or for the root route



Routes With Injector

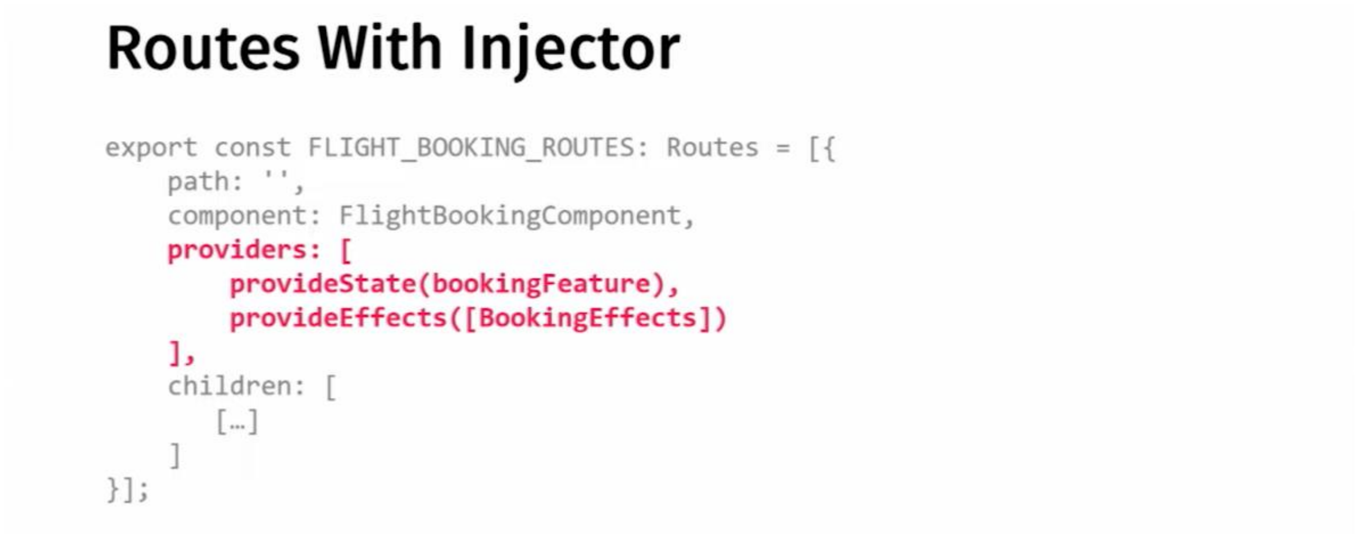
```
export const FLIGHT_BOOKING_ROUTES: Routes = [{
    path: '',
    component: FlightBookingComponent,
    providers: [
        MyService
    ],
    children: [
       […]
    ]
}];
```

Scope: This route + all child routes

(Lazily) loaded with route config

If possible, use providedIn: 'root'

We can now define our providers for a specific route and for all its child routes.

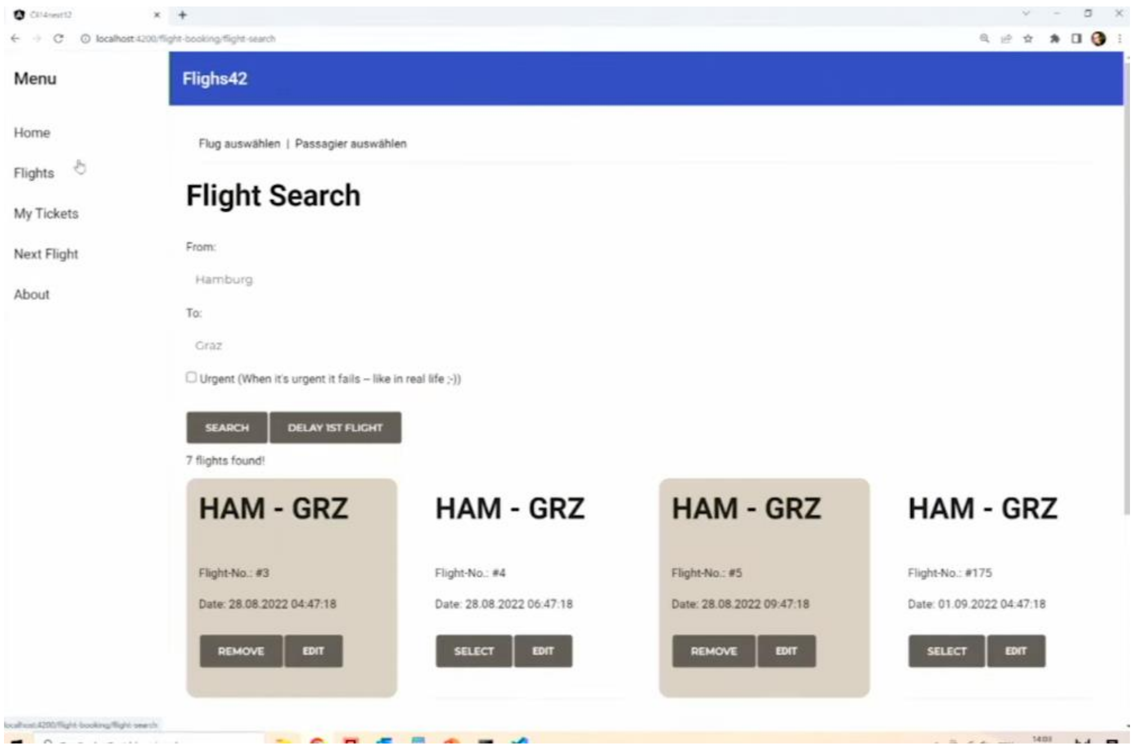


Routes With Injector

```
export const FLIGHT_BOOKING_ROUTES: Routes = [{
    path: '',
    component: FlightBookingComponent,
    providers: [
        provideState(bookingFeature),
        provideEffects([BookingEffects])
    ],
    children: [
       […]
    ]
}];
```

We can use this for cases where we want to configure something like a feature slice for an NgRx store like state and effects.

DEMO



localhost:4200/flight-booking/flight-search
Menu
Home
Flights
My Tickets
Next Flight
About
Flighs42
Flug auswählen | Passagier auswählen

Flight Search

From:

Hamburg

To:

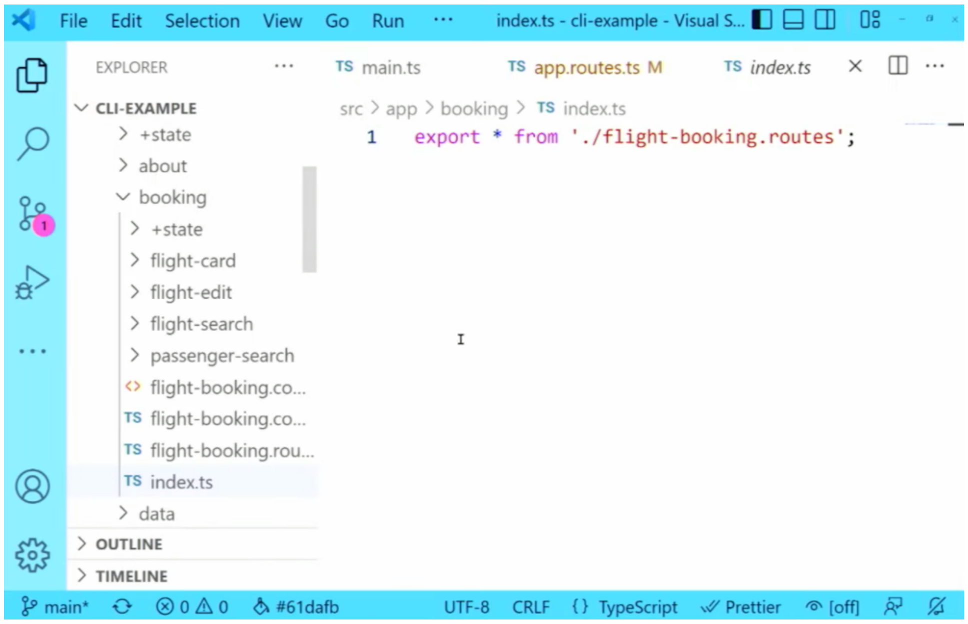
Graz

Urgent (When it's urgent it fails – like in real life ;-))

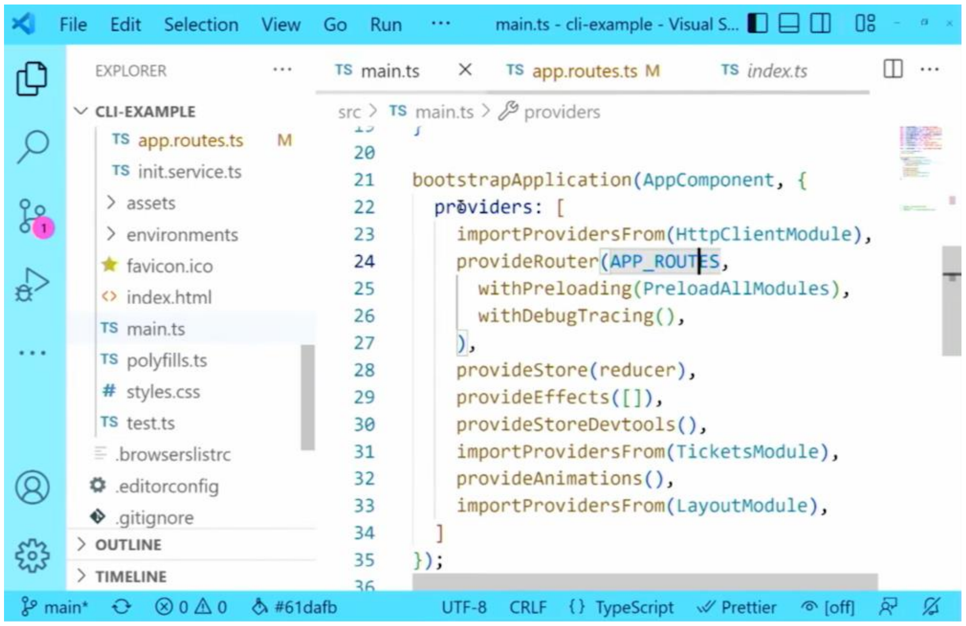
SEARCH | DELAY 1ST FLIGHT

7 flights found!

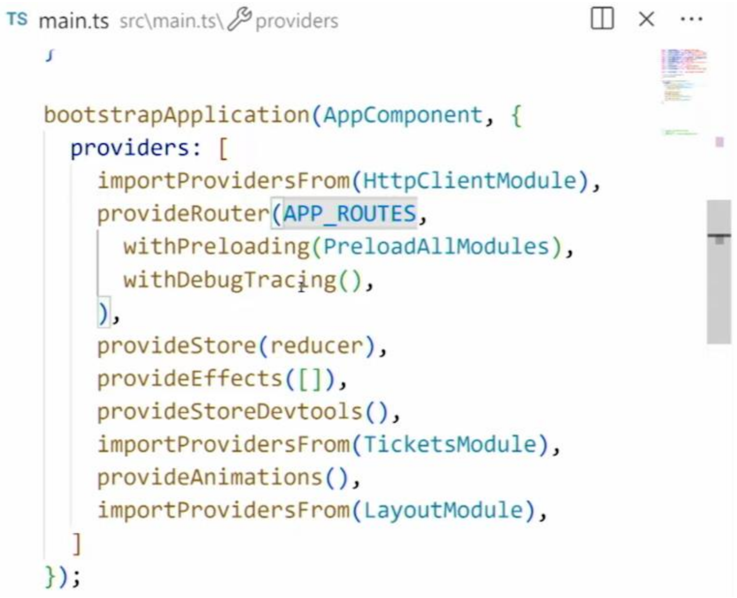
HAM - GRZ	HAM - GRZ	HAM - GRZ	HAM - GRZ
Flight-No.: #3	Flight-No.: #4	Flight-No.: #5	Flight-No.: #175
Date: 28.08.2022 04:47:18	Date: 28.08.2022 06:47:18	Date: 28.08.2022 09:47:18	Date: 01.09.2022 04:47:18
REMOVE EDIT	SELECT EDIT	REMOVE EDIT	SELECT EDIT



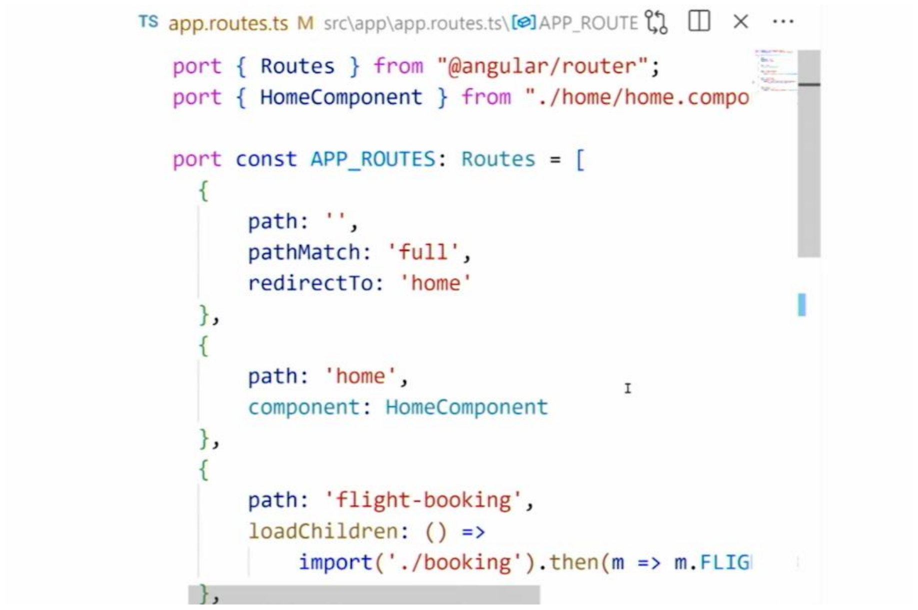
```
export * from './flight-booking.routes';
```



```
bootstrapApplication(AppComponent, {
  providers: [
    importProvidersFrom(HttpClientModule),
    provideRouter(APP_ROUTES,
      withPreloading(PreloadAllModules),
      withDebugTracing(),
    ),
    provideStore(reducer),
    provideEffects([]),
    provideStoreDevtools(),
    importProvidersFrom(TicketsModule),
    provideAnimations(),
    importProvidersFrom(LayoutModule),
  ]
});
```

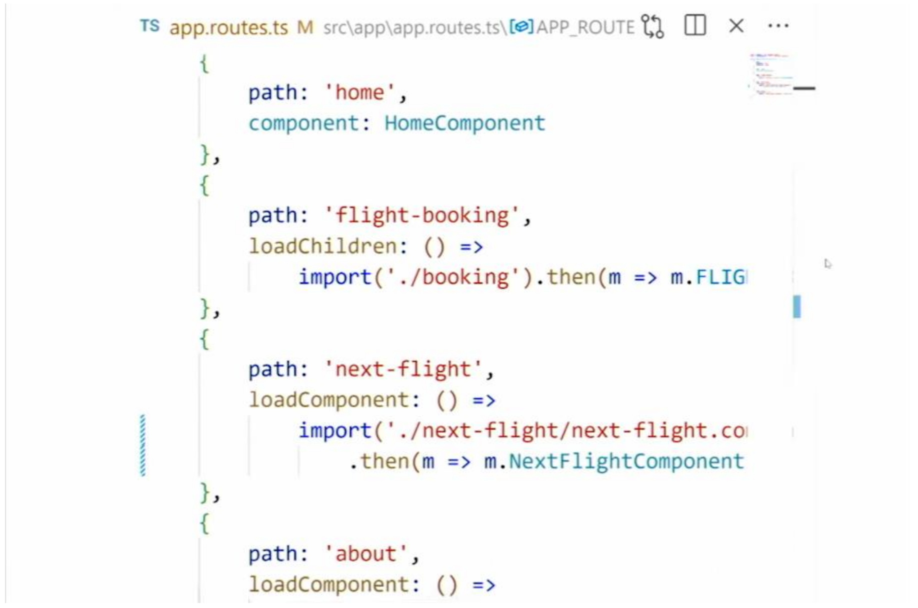


```
}

bootstrapApplication(AppComponent, {
  providers: [
    importProvidersFrom(HttpClientModule),
    provideRouter(APP_ROUTES,
      withPreloading(PreloadAllModules),
      withDebugTracing(),
    ),
    provideStore(reducer),
    provideEffects([]),
    provideStoreDevtools(),
    importProvidersFrom(TicketsModule),
    provideAnimations(),
    importProvidersFrom(LayoutModule),
  ]
});
```



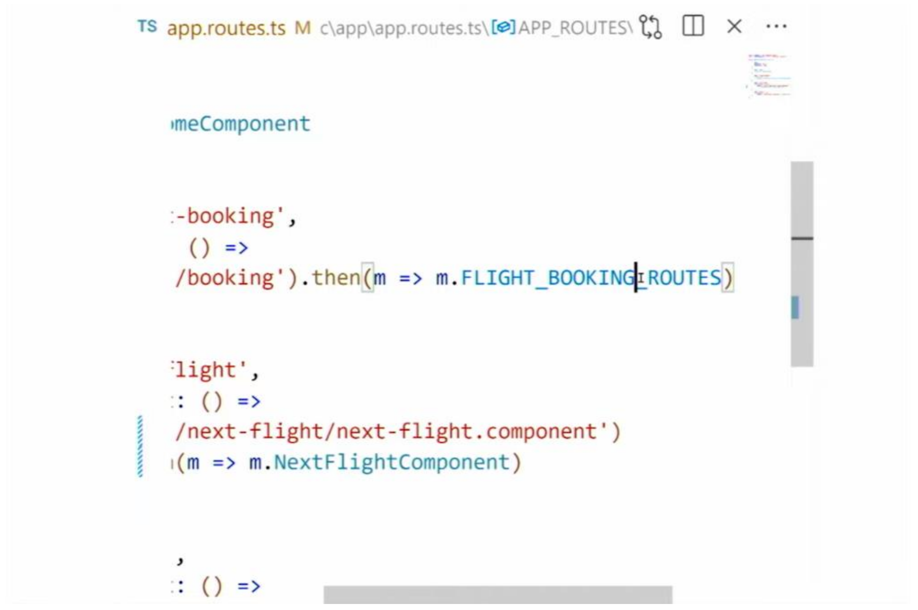
TS app.routes.ts M src\app\app.routes.ts\APP_ROUTE

```
port { Routes } from "@angular/router";
port { HomeComponent } from "./home/home.compo

port const APP_ROUTES: Routes = [
  {
      path: '',
      pathMatch: 'full',
      redirectTo: 'home'
  },
  {
      path: 'home',
      component: HomeComponent
  },
  {
      path: 'flight-booking',
      loadChildren: () =>
          import('./booking').then(m => m.FLIG
  },
```



TS app.routes.ts M src\app\app.routes.ts\APP_ROUTE

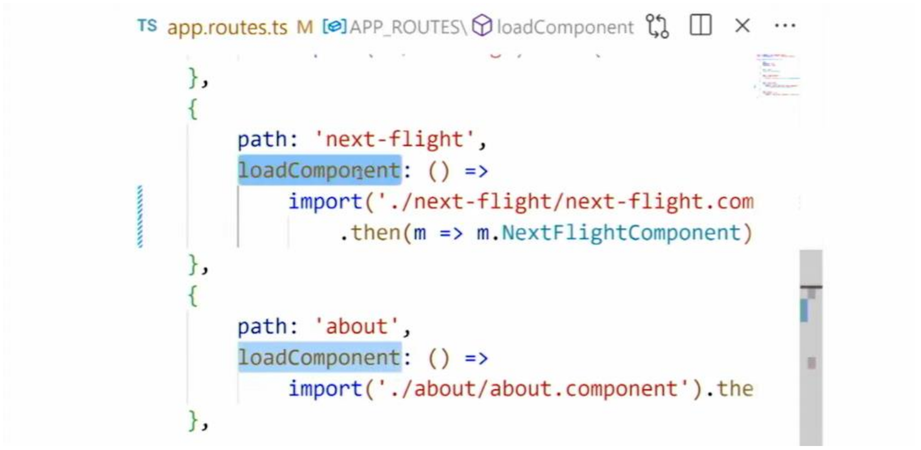
```
  {
      path: 'home',
      component: HomeComponent
  },
  {
      path: 'flight-booking',
      loadChildren: () =>
          import('./booking').then(m => m.FLIG
  },
  {
      path: 'next-flight',
      loadComponent: () =>
          import('./next-flight/next-flight.co
              .then(m => m.NextFlightComponent
  },
  {
      path: 'about',
      loadComponent: () =>
```



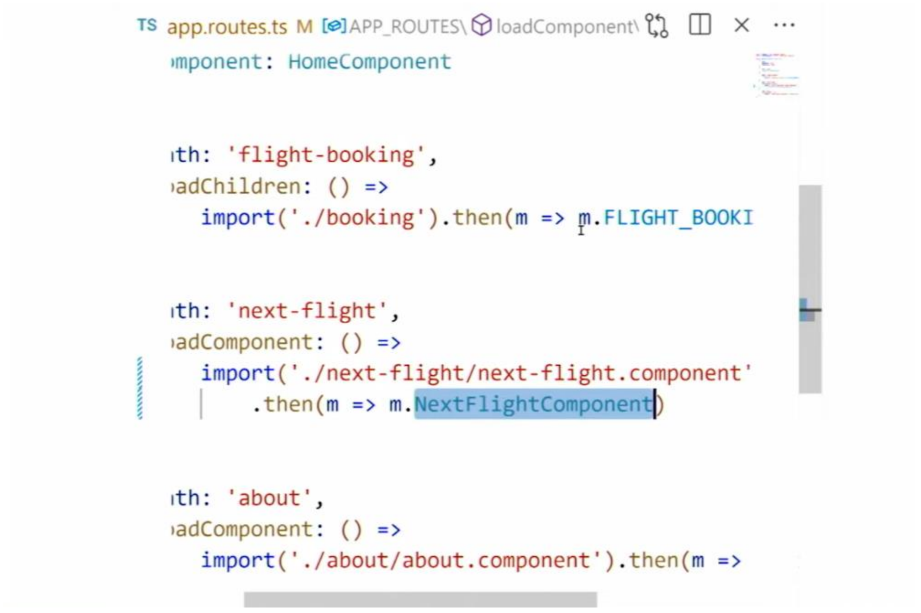
TS app.routes.ts M c\app\app.routes.ts\APP_ROUTES\

```
meComponent


:-booking',
 () =>
/booking').then(m => m.FLIGHT_BOOKING_ROUTES)


light',
: () =>
/next-flight/next-flight.component')
(m => m.NextFlightComponent)


,
: () =>
```

We are doing lazy loading using **loadChildren** and directly pointing to another routing configuration

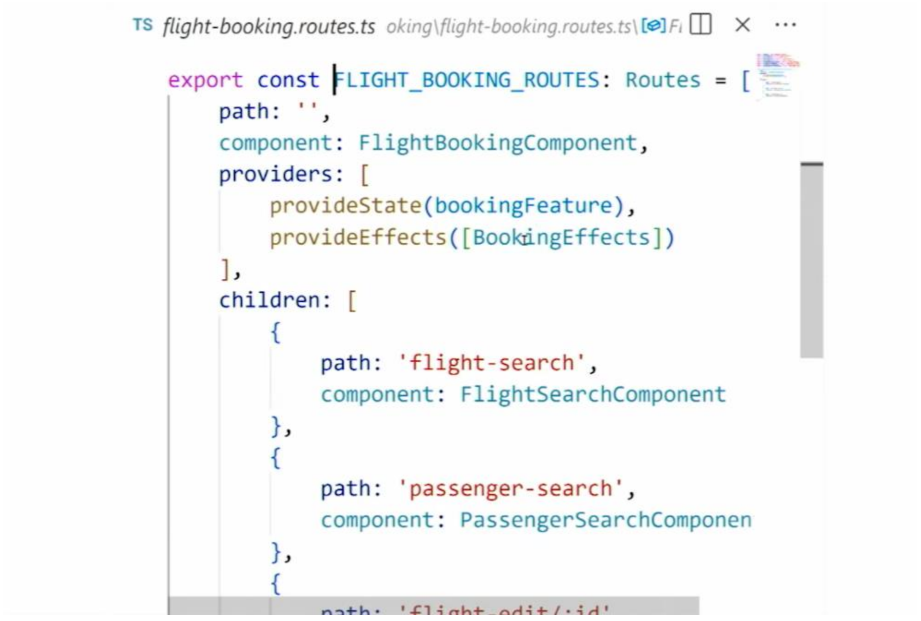

TS app.routes.ts M APP_ROUTES\ loadComponent

```
},
{
    path: 'next-flight',
    loadComponent: () =>
        import('./next-flight/next-flight.com
            .then(m => m.NextFlightComponent)
},
{
    path: 'about',
    loadComponent: () =>
        import('./about/about.component').the
},
```


TS app.routes.ts M APP_ROUTES\ loadComponent\

```
mponent: HomeComponent


ith: 'flight-booking',
adChildren: () =>
  import('./booking').then(m => m.FLIGHT_BOOKI


ith: 'next-flight',
adComponent: () =>
  import('./next-flight/next-flight.component'
      .then(m => m.NextFlightComponent)


ith: 'about',
adComponent: () =>
  import('./about/about.component').then(m =>
```

We are also using **loadComponent** to directly point to another standalone component

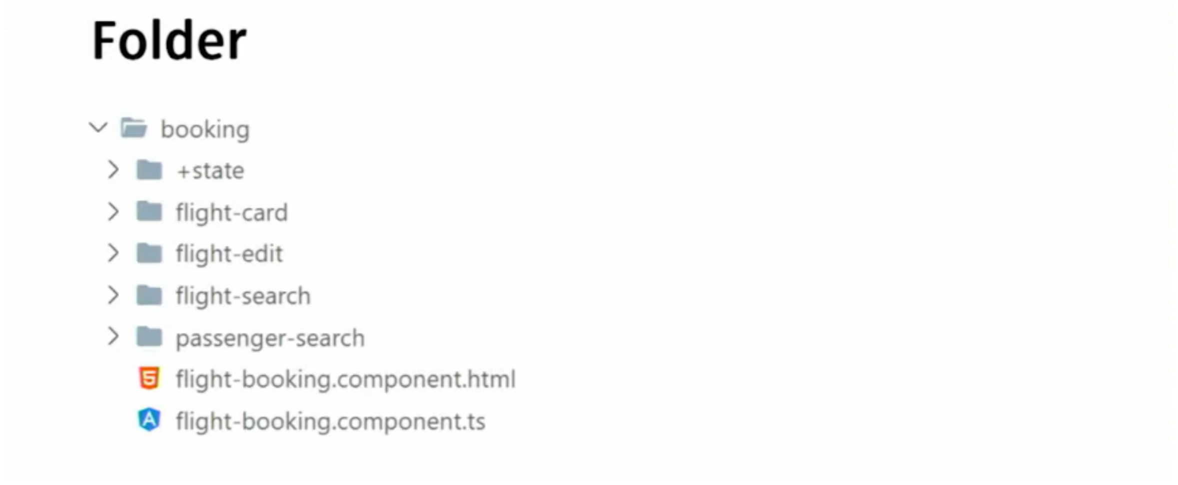

TS flight-booking.routes.ts oking\flight-booking.routes.ts\ Fi

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

Here is another routing configuration where we are providing some services like State and Effects for ngRx for this lazy section of our application.

#2: Structuring without NgModules

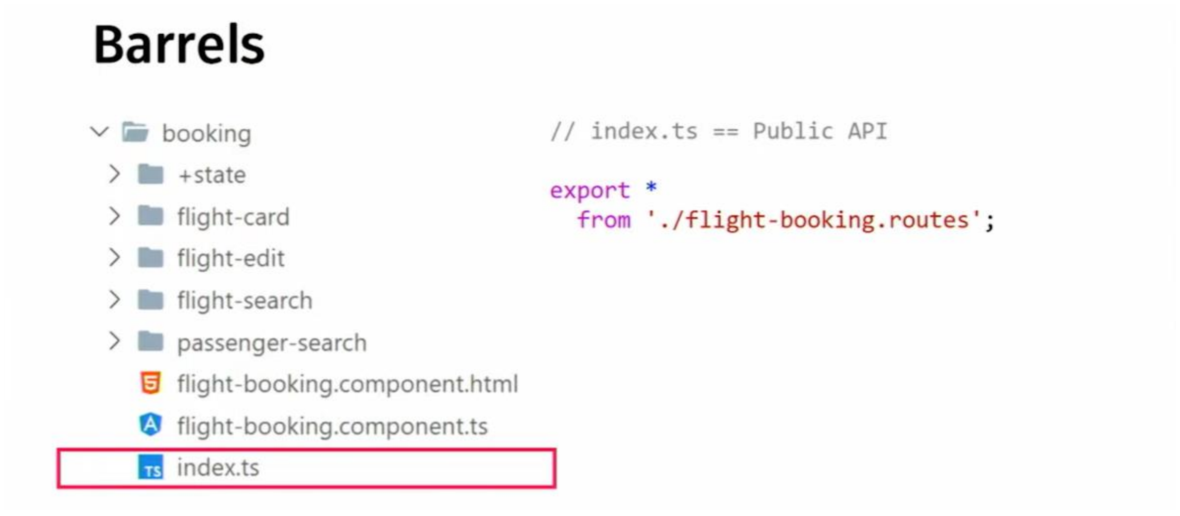
What does all these mean for structuring our application? How can we divide our app into pieces without ngModules?



Folder

```
booking
  +state
  flight-card
  flight-edit
  flight-search
  passenger-search
  flight-booking.component.html
  flight-booking.component.ts
```

Use folders to divide or use a barrel (index.ts file) as below



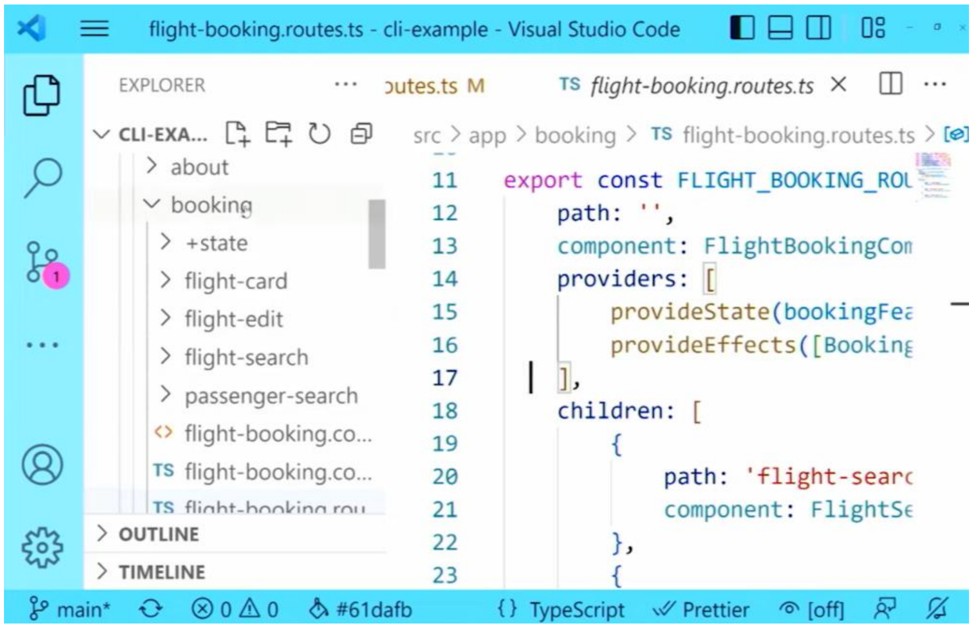
Barrels

```
booking
  +state
  flight-card
  flight-edit
  flight-search
  passenger-search
  flight-booking.component.html
  flight-booking.component.ts
  index.ts
```

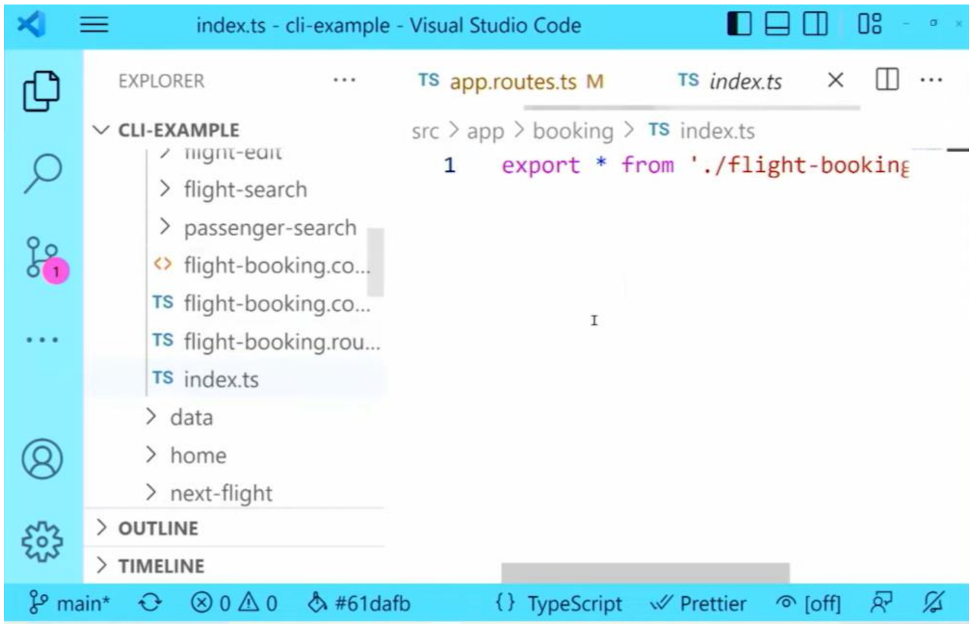
```
// index.ts == Public API

export *
  from './flight-booking.routes';
```

The barrel (index.ts) is your public API and everything you expose from it is intended for other parts of the application to be used, anything not exposed are internal and can be easily changed afterwards or rewritten and not backward compatible. Barrels are the better replacement for ngModules, they give you a real public API with simple vanilla JS.

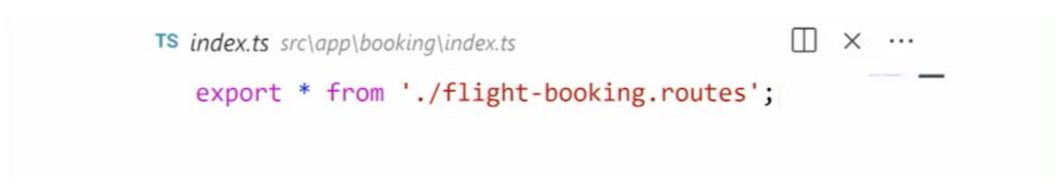
DEMO


flight-booking.routes.ts - cli-example - Visual Studio Code
EXPLORER
CLI-EXA...
about
booking
+state
flight-card
flight-edit
flight-search
passenger-search
flight-booking.co...
flight-booking.co...
OUTLINE
TIMELINE
src > app > booking > flight-booking.routes.ts
main* #61dafb TypeScript Prettier [off]

```
export const FLIGHT_BOOKING_ROU
    path: '',
    component: FlightBookingCom
    providers: [
        provideState(bookingFea
        provideEffects([Booking
    ],
    children: [
        {
            path: 'flight-searc
            component: FlightSe
        },
        {
```



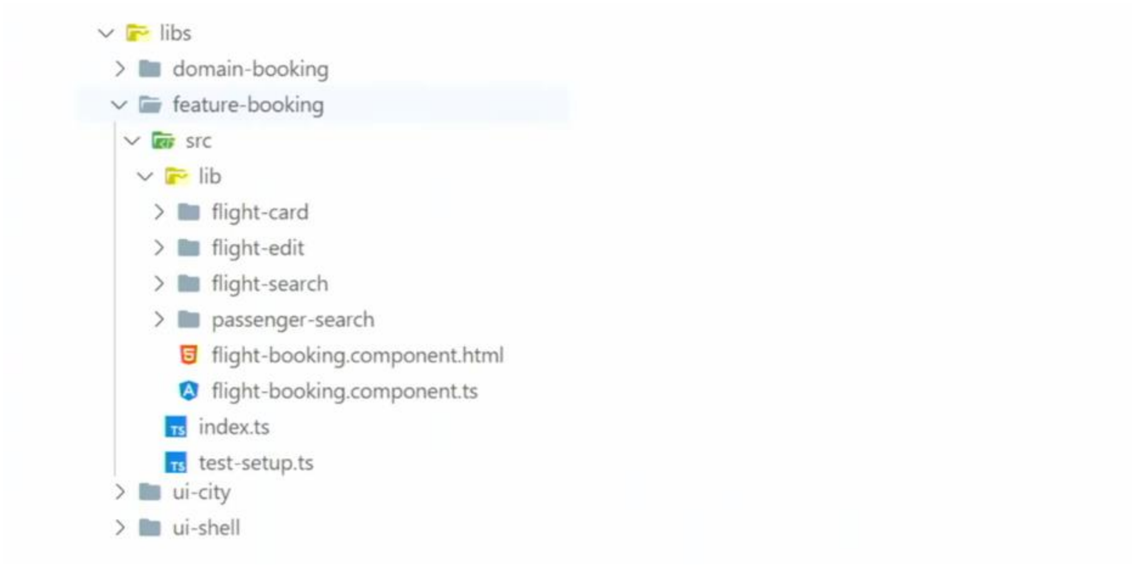
```
export * from './flight-booking
```



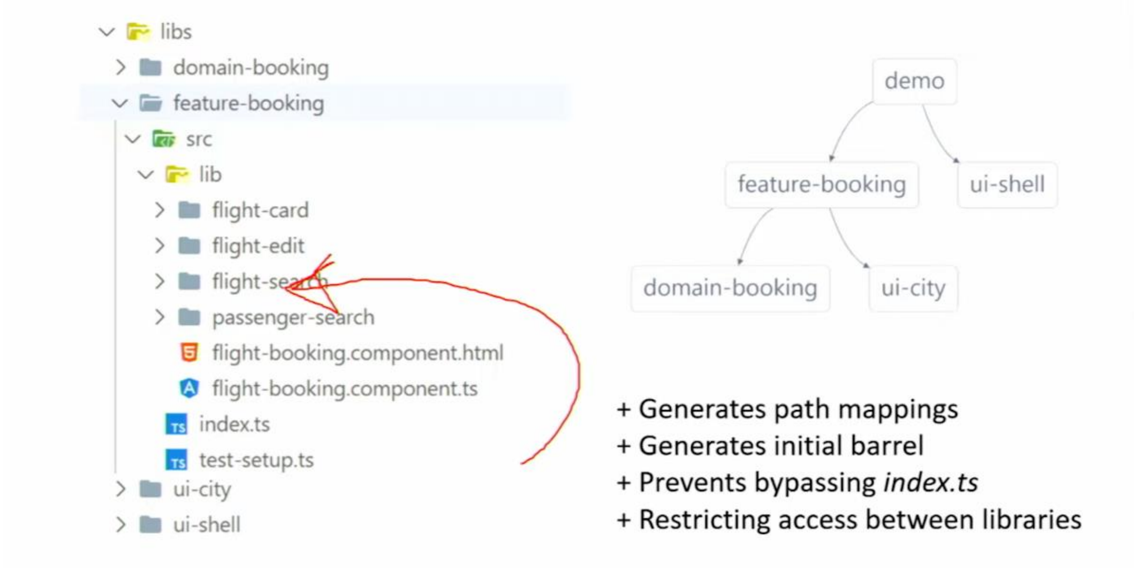
```
export * from './flight-booking.routes';
```

This barrel is exporting everything to other parts of the application to use.

The Next Logical Step: Nx Workspaces

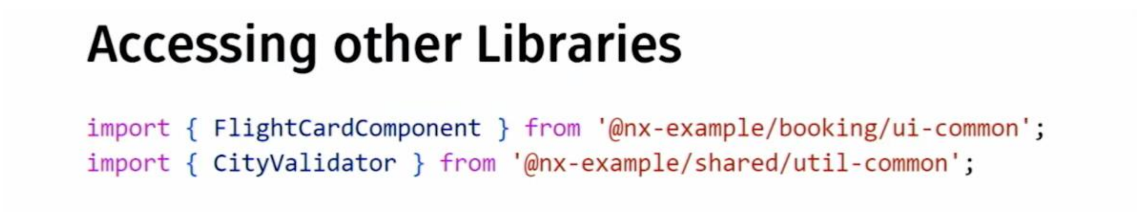


Nx easily allows you to subdivide a huge application into libraries, a **library** is just a folder with **source code** and a **barrel**.



+ Generates path mappings
+ Generates initial barrel
+ Prevents bypassing *index.ts*
+ Restricting access between libraries

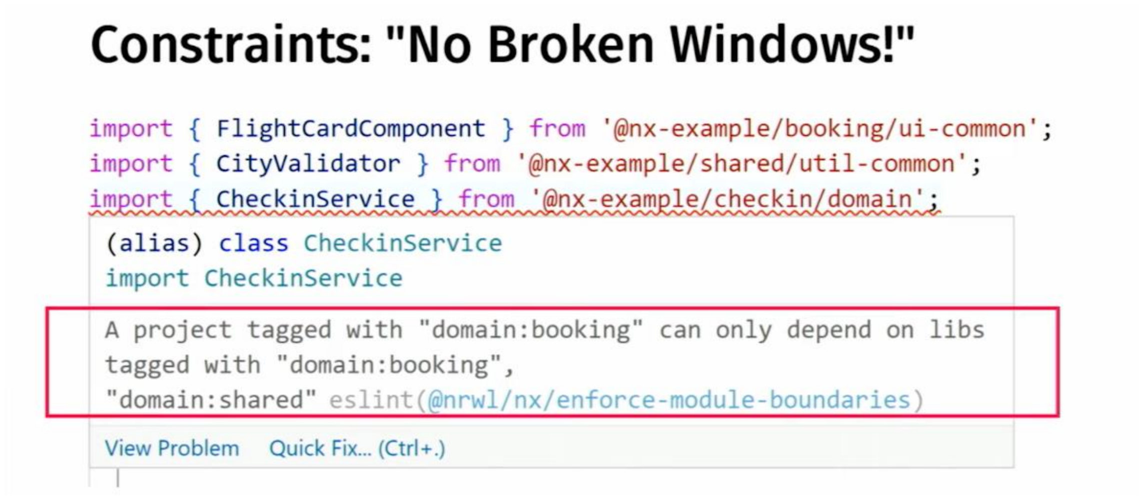
Nx gives you a dependency graph to see coupling and dependencies in your application. You can also generate path mappings so that all your libraries get a beautiful name instead of complex relative routes. You also get linting rules that prevent you from grabbing into the private parts of your defined library barrels.



Accessing other Libraries

```
import { FlightCardComponent } from '@nx-example/booking/ui-common';
import { CityValidator } from '@nx-example/shared/util-common';
```

You can import things from your libraries,



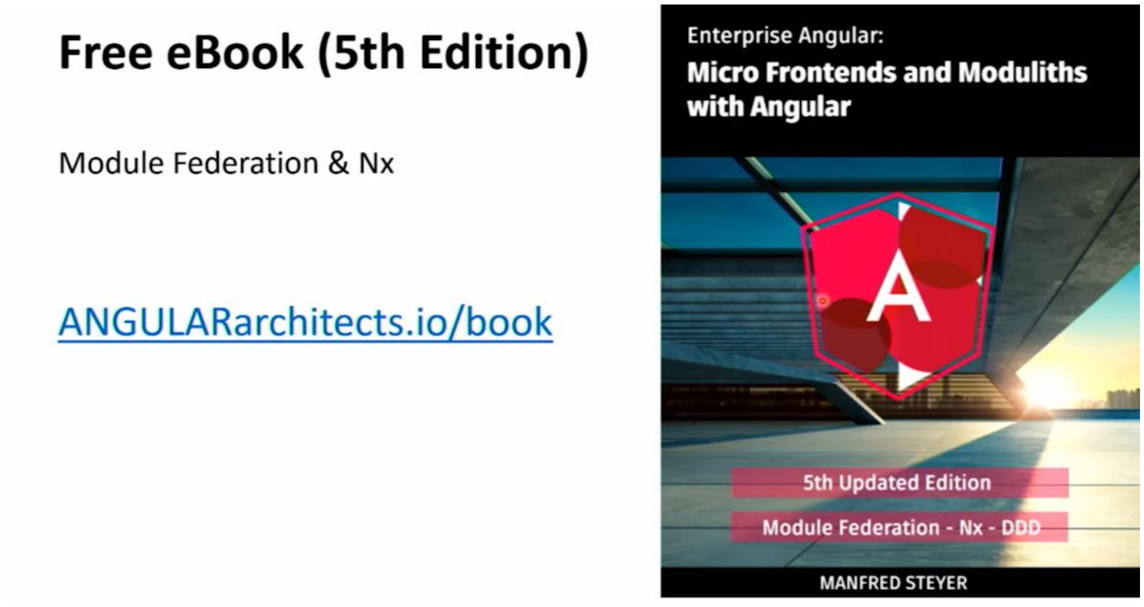
Constraints: "No Broken Windows!"

```
import { FlightCardComponent } from '@nx-example/booking/ui-common';
import { CityValidator } from '@nx-example/shared/util-common';
import { CheckinService } from '@nx-example/checkin/domain';
```

```
(alias) class CheckinService
import CheckinService
```

```
A project tagged with "domain:booking" can only depend on libs
tagged with "domain:booking",
"domain:shared" eslint(@nrwl/nx/enforce-module-boundaries)
```

You also get error messages when trying to use something you aren't allowed to use.

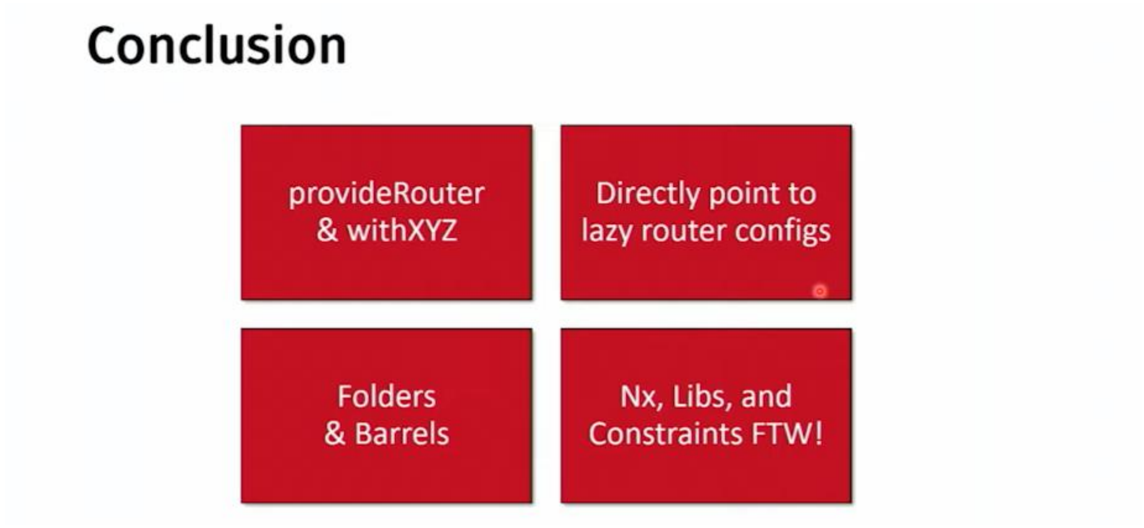


Free eBook (5th Edition)

Module Federation & Nx

ANGULARarchitects.io/book

Enterprise Angular:
Micro Frontends and Moduliths with Angular
5th Updated Edition
Module Federation - Nx - DDD
MANFRED STEYER



Conclusion

- provideRouter & withXYZ
- Directly point to lazy router configs
- Folders & Barrels
- Nx, Libs, and Constraints FTW!

Contact and Downloads

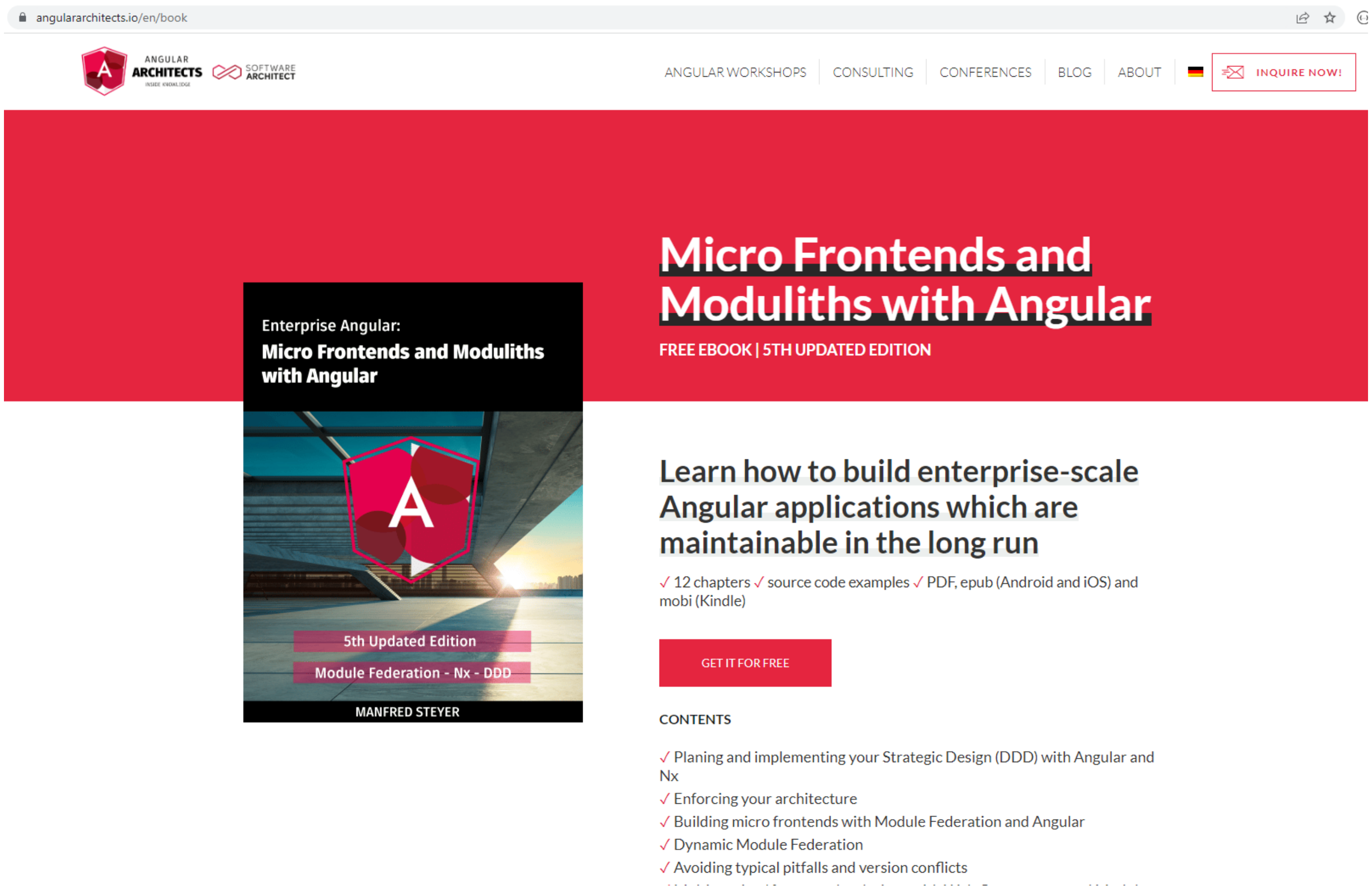
[web] ANGULARarchitects.io

[twitter] ManfredSteyer

Slides & Examples

Remote Company Workshops and Consulting

http://angulararchitects.io



angulararchitects.io/en/book

ANGULAR WORKSHOPS | CONSULTING | CONFERENCES | BLOG | ABOUT | INQUIRE NOW!

Micro Frontends and Moduliths with Angular

FREE EBOOK | 5TH UPDATED EDITION

Enterprise Angular:
Micro Frontends and Moduliths with Angular

5th Updated Edition
Module Federation - Nx - DDD
MANFRED STEYER

Learn how to build enterprise-scale Angular applications which are maintainable in the long run

✓ 12 chapters ✓ source code examples ✓ PDF, epub (Android and iOS) and mobi (Kindle)

GET IT FOR FREE

CONTENTS

✓ Planing and implementing your Strategic Design (DDD) with Angular and Nx
✓ Enforcing your architecture
✓ Building micro frontends with Module Federation and Angular
✓ Dynamic Module Federation
✓ Avoiding typical pitfalls and version conflicts