netmind
a bts company

Angular 14 - 20

Angular Modules and optimization

Scope on modules

- **NgModule** is the first basic structure you encounter when coding an application with Angular, but it's also the most subtle and complex, due to the different scopes.

- The purpose of an NgModule is to declare each element that is created in Angular and group them together (like Java packages or C# namespaces).

- There are two main types of structures:
  - **Declarations** is for everything you'll use in templates: mainly components (~ views: the classes that display data), but also directives and pipes,
  - **Providers** is for services (~ models: the classes that get and manage data).

```
...
@NgModule({
  declarations: [SomeComponent, SomeDirective, SomePipe],
  providers: [SomeService]
})
export class SomeModule {}
```

Scope on modules

- The confusion starts with components and services that do not have the same scope:
  - declarations/components have local scope (private visibility),
  - providers/services generally have global reach (public visibility).

- This means that components that are declared can only be used in the current module.

- If you need to use them outside, in other modules, we must **export** them.

- On the other hand, the **services** will be generally available or **injectable anywhere** in the application, in all modules.

```
...
@NgModule({
    declarations: [SomeComponent, SomeDirective, SomePipe],
    exports: [SomeComponent, SomeDirective, SomePipe]
    ...
})
export class SomeModule {}
```

When to import an ngModule

- Things get complicated because, as in any framework, we will not only have to import one module, but many.

- Angular is subdivided into different modules (core, common, http, etc.).

- In a module we must import other NgModules or standalone components that are going to be used in this:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { HttpClientModule } from '@angular/common/http';

import { FeatureModule } from '../feature/feature.module';

@NgModule({
  imports: [CommonModule, HttpClientModule, FeatureModule]
})
export class SomeModule {}
```

When to import an ngModule

- We need to know how these other modules are imported, or what features are going to be used.
  - To use **components** (or other template-related sas elements, like directives and pipes)?
  - or is it to use **services**?

- Why? Because given the difference in scope between components and services:
  - if the module is imported to use components, you will need to import it in every module that needs it,
  - if the module is imported to use services, you will need to import it only once, in the first module of the application.

- If they are not imported correctly, we will have errors in the components that are not available, because we forgot to import your module again.

- Or if we import a module to use services more than once, we can generate errors in advanced scenarios like lazy-loading.

When to import an ngModule

- **Modules to import whenever you need them**
  - CommonModule (all basic Angular template elements: bindings, *ngIf, *ngFor...), except in the first module of the app, because it's already part of the BrowserModule
  - FormsModule and ReactiveFormsModule
  - MatXModule and other UI modules
  - Any other module that provides you with components, directives or pipes

- **Modules to import only once**
  - HttpClientModule
  - BrowserAnimationsModule or NoopAnimationsModule
  - any other module that provides services only.

- This is why with Angular CLI CommonModule is automatically imported when you create a new module.

Mixed ngModules

- An example we are already familiar with is **RouterModule**, which provides a component (`<router-outlet>`) and a directive (`routerLink`), but also services (`ActivatedRoute`, for getting URL parameters, router to navigate…).

- Fortunately, the mess is handled by the module itself. The routing files are automatically generated by the Angular CLI.

- But there is a subtle difference between the routing of the first application module and the routing of the submodules.

  - For the **main module**:

    ```
    RouterModule.forRoot(routes)
    ```

  - For **other modules**:

    ```
    RouterModule.forChild(routes)
    ```

- This is so because in the application module, **forRoot()** will provide the router components and provide the router services.

- In submodules, **forChild()** will only provide the router components (and will not provide the services again).

Lazy-loading modules

- Load a module is easy, just set the reference in the route with `loadChildren`:

```
const routes: Routes = [
  { path: 'admin', loadChildren: './admin/admin.module#AdminModule' }
];
```

- This will generate a different bundle and module, which will be loaded only when in demand, therefore it will not be included in the global scope of the application.

- For components, nothing changes: we'll need to import the CommonModule and other component modules again, just like in any submodule.

- For services, we will still have access to the services already provided in the app (such as HttpClient and its own services), but the services provided in the lazy-loading module will only be available in this module, not in the whole app.

Create lazy-loaded modules

🔗 **As an example we will create an application with routing**

```
ng new customer-app –routing
```

🔗 **Next we will create a submodule or "feature" module with its own routing:**

```
ng generate module customers --routing
```

🔗 **We can add a component**

```
ng generate component customers/customer-list -m customers
```

🔗 **We will also add another submodule "orders"**

```
ng generate module orders --routing
ng generate component orders/order-list -m orders
```

Create lazy-loaded modules

- Finally we will generate the UI

```html
<h1>
  {{title}}
</h1>

<button routerLink="/customers">Customers</button>
<button routerLink="/orders">Orders</button>
<button routerLink="">Home</button>

<router-outlet></router-outlet>
```
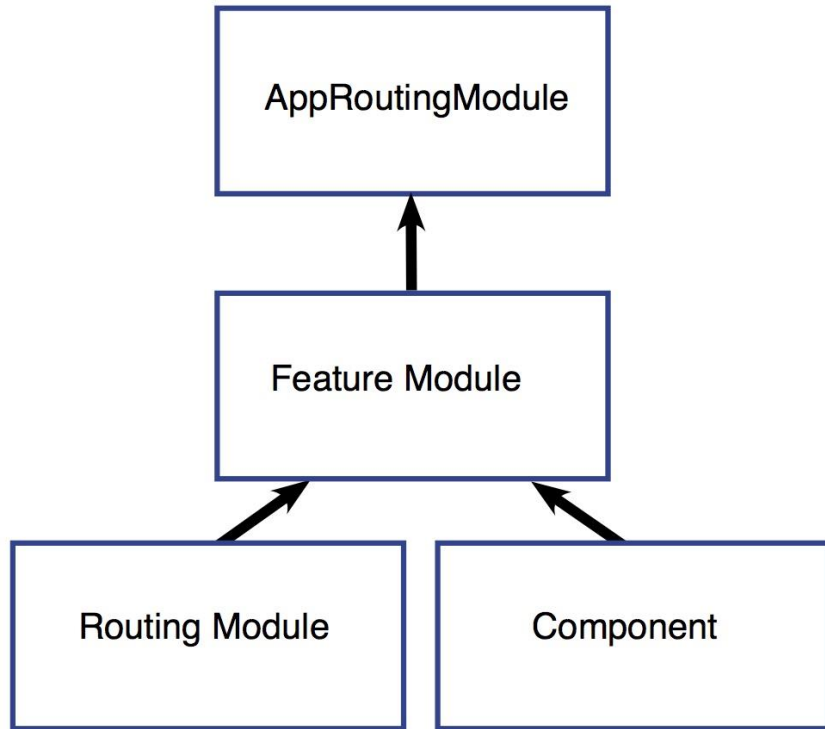
- Then we launch the application

```
ng serve
```

Create lazy-loaded modules

- The two modules, OrdersModule and CustomersModule, will be connected to the AppRoutingModule in order for the router to be aware of them.

- Each submodule acts as a gateway through the router.

- In AppRoutingModule the routes to the sub modules are configured. This way, the router knows to go to the appropriate submodule.

- The submodule then connects the AppRoutingModule to the CustomersRoutingModule or the OrdersRoutingModule.

- These routing modules tell the router where to go to load the relevant components.

Create lazy-loaded modules

- Therefore the **routes at application level** a will be:

- The first two routes are the routes to the CustomersModule and OrdersModule, respectively.

- Note that the **lazy-loading** syntax uses **loadChildren** followed by a string that is the path to the module.

```typescript
const routes: Routes = [
  {
    path: 'customers',
    loadChildren: () =>
      import('./customers/customers.module').then(
        (m) => m.CustomersModule
      ),
  },
  {
    path: 'orders',
    loadChildren: () =>
      import('./orders/orders.module').then(
        (m) => m.OrdersModule
      ),
  },
  {
    path: '',
    redirectTo: '',
    pathMatch: 'full',
  },
];
```

Create lazy-loaded modules

- The **routes at the submodule level** will be

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

import { CustomerListComponent } from './customer-list/customer-list.component';

const routes: Routes = [
  {
    path: '', component: CustomerListComponent
  }
];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class CustomersRoutingModule { }
```

- The path is set to an empty string (`path='' `).
  - This is because the route in the AppRoutingModule is already configured, so this route in the CustomersRoutingModule is already inside the customer context.
- Each route in this routing module is a **relative route**.

Shared services

- The steps to create a shared service are
  - Create the shared module
  - Various options:
    - Use providedIn:'root'
    - Generate a provided servide with foorRoot() method.
    - https://angular.io/guide/singleton-services

  - Import the submodule into the main module (in the case of provided forRoot)

  - In other submodules import the submodule without the forRoot

```typescript
@Injectable({
  providedIn: 'root'
})
export class SharedService {}
```

```typescript
@NgModule({})
export class SharedModule {
  static forRoot(): ModuleWithProviders<SharedModule> {
    return {
      ngModule: SharedModule,
      providers: [SharedService],
    };
  }
}
```

```typescript
@NgModule({
  declarations: [...],
  imports: [
    SharedModule.forRoot(), ...
  ],
  providers: [...],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

```typescript
@NgModule({
  declarations: [...],
  imports: [
    SharedModule, ...
  ],
  providers: [...],
  bootstrap: [AppComponent],
})
export class FeatureModule {}
```

Let's put it into practice: Tasks/Projects App

- Separate the app login into a submodule

- Likewise, the authorization verification service

The build for Production

- The project generated by Angular CLI comes with a simple **ng build**.
- To generate artifacts for production we have to do a bit of customization ourselves adding a script sntry to our **package.json**.

```
...
"scripts": {
    "build:prod": "ng build --aot --vendor-chunk --common-chunk --delete-output-path --build-optimizer --output-hashing=all --source-map=false --named-chunks=false"
}
...
```

- --aot: enable ahead-of-time compilation.
- --output-hashing=all : hashes the contents of the generated files and append the hash to the filename to make it easier to clear the browser cache (any change to the file contents will result in a different hash and thus the browser will be forced to load a new version of the file)
- --extract-css=true: extract all css into a separate style sheet file
- --sourcemaps=false: disable source map generation
- --named-chunks=false: disable the use of human-readable names for the chunk and use numbers instead
- --build-optimizer: new feature that results in smaller packages but much longer build times, (should also be enabled by default in the future)
- --vendor-chunk: extract all vendor (library) code into a separate chunk
- https://angular.io/cli/build

Aliases for the app and the environment

- Creating aliases for our folders and environments will allow us to **implement clean imports** that will be consistent throughout our application.

- For example, a typical situation would be:
  - We are working on a component that is three folders deep in feature A and we want to import the service from core that is two folders deep.
  - This would lead to the import statement:

```
import {SomeService} from '../../../core/subpackage1/subpackage2/some.service'
```

- In order to use aliases we need to add the **baseUrl** and **paths** properties to our **tsconfig.json** file.

```json
{
  "compilerOptions": {
    "...": "reduced for brevity",

    "baseUrl": "src",
    "paths": {
      "@app/*": ["app/*"],
      "@env/*": ["environments/*"]
    }
  }
}
```

Aliases for the app and the environment

- With these changes we can import the services and environment variables in this way:

```typescript
import { Component, OnInit } from '@angular/core';
import { Observable } from 'rxjs/Observable';

/* globally accessible app code (in every feature module) */
import { SomeSingletonService } from '@app/core';
import { environment } from '@env/environment';

/* localy accessible feature module code, always use relative path */
import { ExampleService } from './example.service';

@Component({
  /* ... */
})
export class ExampleComponent implements OnInit {
  constructor(
    private someSingletonService: SomeSingletonService,
    private exampleService: ExampleService
  ) {}
}
```

Aliases for the app and the environment

- We're importing entities (like SomeSingletonService in the example above) directly from `@app/core` instead of `@app/core/some-package/some-singleton.service` .

- This is made possible by re-exporting all public entities in the main index.ts file. We create one index.ts file per package (folder):

```
export * from './core.module';
export * from './auth/auth.service';
export * from './user/user.service';
export * from './some-singleton-service/some-singleton.service';
```

- In most applications, the components and services of a particular feature module will generally only have access to the services of the CoreModule and the SharedModule components.

- Sometimes this may not be enough to solve particular business cases and we will also need some sort of "shared feature module" that provides functionality for a limited subset of other feature modules.

- In this case, we'll end up with something like import {SomeService} from '@ app/shared-feature' ; Similar to core, the shared function is also accessed using @app aliases.

Optimization tips

Use OnPush

- By default, Angular runs change detection on all components whenever something changes in the application -- from a click event, to data received from an ajax call. (user events, timers, xhr, promises, etc.)

- We can help Angular and give you a better indication of when to check out our component.

- We can set the **ChangeDetectionStrategy** of our component to **ChangeDetectionStrategy.OnPush**.

  - https://angular.io/api/core/ChangeDetectionStrategy

- This tells Angular that the component only depends on its Inputs (aka pures) and should only be checked in the following cases:
  - The Input reference changes.
  - An event occurred in the component or one of its children.
  - Run change detection explicitly by calling detectChanges()/tick()/markForCheck() .

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class AppComponent {
  title = 'Products store';
}
```

Optimization tips

Use TrackBy

- If at some point, we need to change the data in the collection, perhaps as a result of an API request, we'll run into a problem, because Angular can't keep track of the items in the collection and has no knowledge of which items have been changed. removed or added.
    - As a result, Angular needs to remove all DOM elements associated with the data and create them anew. This can mean a lot of DOM manipulations, especially in the case of a large collection. And, as we know, DOM manipulations are expensive.
    - If we provide a trackBy function, Angular can track which items have been added or removed according to the unique identifier and only create or destroy things that have changed.

```
@Component({
  selector: 'my-app',
  template: `
    <ul>
      <li *ngFor="let item of collection;trackBy: trackByFn">{{item.id}}</li>
    </ul>
    <button (click)="getItems()">Refresh items</button>
  `,
})
export class AppComponent {
  title = 'Products store';
}
```

Optimization tips

Avoid calculating values in the template

- Sometimes you need to transform a value that comes from the server into something that you can display in the UI.
    - The problem is that because Angular needs to re-run your function on every change detection cycle, if the function performs expensive tasks, it can be expensive.

```html
<table>
  <tr *ngFor="let skill of skills">{{skill.calcSomething(skill)}}</tr>
</table>
```

- If the value doesn't change dynamically at runtime, a better solution would be:
    - Use pure pipes: Angular executes a pure pipe only when it detects a pure change in the input value.
    - Create a new property and set the value once

```javascript
this.skills = this.skills.map(skill => (
  { ...skill, percentage: calcSomething(skill)  }
);
```

Optimization tips

Disable change detection

- Let's imagine that we have a component that depends on data that is constantly changing, many times per second. Updating the UI every time new data arrives can be expensive.

- A more efficient way would be to check and update the UI every X seconds. We can do this by separating the change listener from the component and doing a local check every x seconds.

```
@Component({
  selector: 'giant-list',
  template: `
    <li *ngFor="let d of dataProvider.data">Data {{d}}</lig>
  `,
})
class GiantList {
  constructor(private ref: ChangeDetectorRef, private dataProvider: DataProvider) {
    ref.detach();
    setInterval(() => {
      this.ref.detectChanges();
    }, 5000);
  }
}
```

Optimization tips

Use lazy loading

- Lazy-loading is one of Angular's most powerful features.

- By default, Webpack will generate all the application code in one big bundle. Lazy-loading provides the ability to optimize application load time by splitting the application into feature modules and loading them on demand.

- Angular makes this process almost transparent.

```
{
  path: 'customers',
  loadChildren: () =>
    import('./customers/customers.module').then(
      (m) => m.CustomersModule
    ),
},
```

- We can even prevent entire modules from loading based on some condition.

- For example, don't load the admin pack if the user is not an admin (route guard **canLoad**)
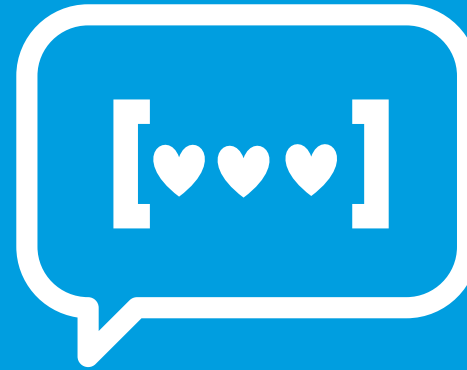
Let's put it into practice: Tasks/Projects App

- Review the application and apply the optimization tips

Next steps

We would like to know your opinion!

Please, let us know what you think about the content.
From Netmind we want to say thank you, we appreciate time
and effort you have taking in answering all of that is
important in order to improve our training plans so that you
will always be satisfied with having chosen us
quality@netmind.es

Thanks!

Follow us: