

UNIT 2

Angular



Part 3 **Http and Observables.** **Navigation and routing.**

Client-side Web Development
2nd course – DAW
IES San Vicente 2020/21
Author: Arturo Bernal Mayordomo

Index

| | |
|--|----|
| Getting data with HttpClient service..... | 3 |
| RxJS and Observables..... | 3 |
| Handling responses and errors with observables..... | 4 |
| Transforming our data in the Service..... | 5 |
| Other HTTP methods (post, put, delete)..... | 7 |
| Sending authentication token in Angular..... | 7 |
| Using an interceptor to send the token..... | 8 |
| Using an interceptor to set the base url for services..... | 9 |
| Routing basics..... | 10 |
| Changing the page's title..... | 11 |
| Advanced routing..... | 13 |
| Passing parameters to a route..... | 13 |
| Showing async object data..... | 14 |
| Activate a route with code..... | 14 |
| Guard navigation to a route with CanActivate..... | 15 |
| Guard navigation from a route with CanDeactivate..... | 16 |
| Pre-fetch data before activating a route with Resolve..... | 18 |

Getting data with HttpClient service

In a real world web application, data is usually retrieved from a web server using HTTP requests (like we saw in previous units using AJAX). Angular provides a service called **HttpClient** that does that job. We need to import in our **app.module.ts** file Angular's **HttpClientModule**.

```
import { HttpClientModule } from '@angular/common/http';
...
@NgModule({
  imports: [
    ...
    HttpClientModule
  ],
  ...
})
export class AppModule { }
```

Now, we want to get our products from a web server instead of having an static array in our **product.service.ts** file. First of all, we'll need to inject the **HttpClient** service into our **ProductService** class (yes, a service can be injected into another service).

```
import { Injectable } from '@angular/core';
import { IProduct } from '../interfaces/i-product';
import { Observable } from 'rxjs';
import { map } from 'rxjs/operators';
import { HttpClient } from '@angular/common/http';

@Injectable({
  providedIn: 'root'
})
export class ProductsService {
  private productURL = 'http://localhost:3000/products';

  constructor(private http: HttpClient) { }

  getProducts(): Observable<IProduct[]> {
    return this.http.get<{products: IProduct[]}>(this.productURL).pipe(
      map(response => response.products)
    );
  }
}
```

RxJS and Observables

When doing HTTP requests with Angular, we could work with **Promises** if we wanted to (AngularJS 1.x works with promises). But instead, by default, Angular 2+ works with **Observables**, which can be seen as a more advanced version of a Promise, with more features. In order to do so, Angular uses an additional library called [RxJS](#).

These are the main differences between Promises and Observables:

- A Promise provides a single future value. A single Observable can provide multiple values over time.
- A Promise starts when its created. An Observable only starts when you

subscribe to it (lazy loading).

- A Promise can't be canceled while an Observable will be canceled when **unsubscribing**.
- Observables have useful methods such as **map**, **filter**, **reduce** and many more. Promises just use the generic method **then** for processing data.

Any call to the HttpClient (**get**, **post**, **put**, **delete**) service returns an Observable. In order to get the data we have to subscribe to it. However, we can process the data that comes from the Observable before the subscription using intermediary operations such as **map**, **filter**, **do**, **reduce** and more.

These are what those functions do (very similar to Array's methods). For now, we'll only use the **map** function when necessary.

- **map** → Takes the data returned from an Observable<Data>, transforms it and returns the transformed data (Observable<TransformedData>).
- **tap** → Used normally for debugging purposes, to print the data on the console, etc. It doesn't return anything. Returns an observable with the same type and data it receives.
- **filter** → Filters the data coming from the observable and only allow it to pass if it meets certain conditions. It has to return true or false (if false that data will be immediately discarded). Returns an observable with the same type it processes.
- **reduce** → Like filter, it doesn't make sense if the observable returns only one value (like an Http call). It accumulates the results (adding, multiplying, etc..) returning an observable with the final accumulated result when the observable is closed.

Handling responses and errors with observables

When creating a new observable, we can process it with [many useful methods](#) (a few of them already mentioned). Those chained methods are **intermediary** methods and execute just after the observable emits a value.

An observable doesn't start until we subscribe to it. The method **subscribe** is a **final method**. This means that other methods return a new Observable object but this one doesn't, so it must be the last method chained to an Observable.

subscribe can receive 3 (optional) parameters, which are functions. The first function will receive the final returned result (if any) when everything was successful (no errors). The second function will receive an error if the observable or any intermediary methods throw an error, and the third function will execute always at the end, no matter if there was an error or not (like the **finally** block in a **try...catch**)

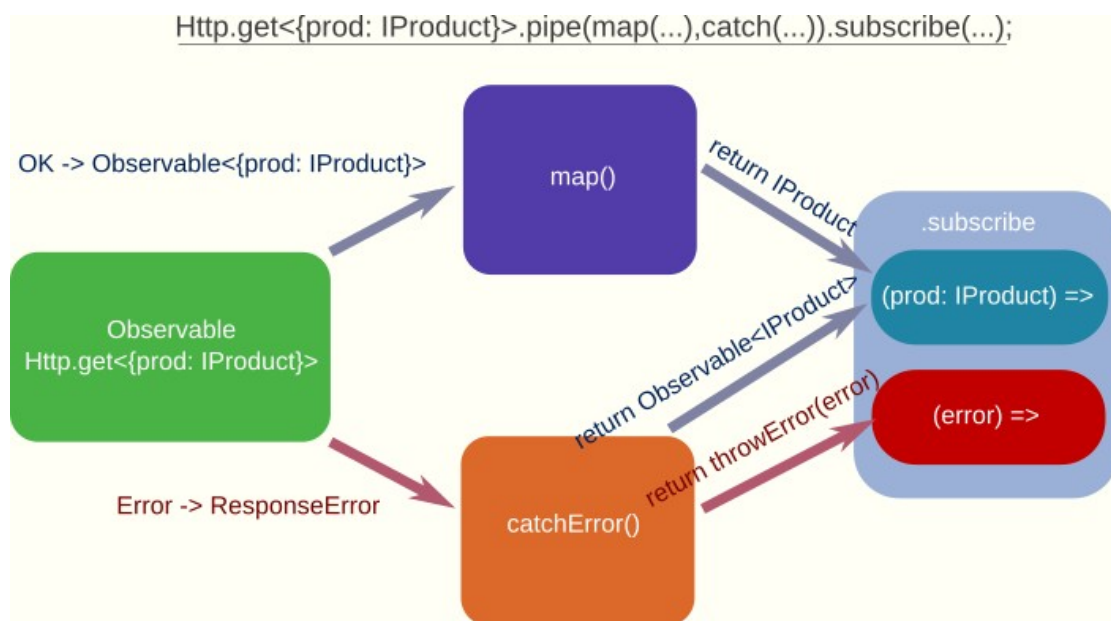
```
Observable.pipe( map(...), catchError(...) ).subscribe(  
  (result) => // Do something with the returned data  
  (error) => // Do something with the error (inform the user, etc...),  
  () => // This will execute always at the end (error or not)  
)
```

However, sometimes we'll want to recover from an error or even process that

error (and maybe return another error with our own format). The **catchError** function does that. If you chain a **catchError** function between the Observable and the subscribe method, it will execute if there's an error before. This method can return a new Observable with faked data, or calling a web service again (error recovery) or throw an error instead with another format → **throwError(newError)**.

The next diagram shows an example of an Http call (returns an Observable), and the possibilities if there's any error. In the **catch** method you can return a new Observable (recover → will go to the map method) returning a new Http call (Observable) or creating a fake response (**return of({prod: IProduct})**). Or you can throw a new Observable error which can be a string, object, anything...

When an error is thrown and there's no catch method after that, the error will be received in the second function of the subscribe method.



There's another useful method that will let us restart the Observable if there's an error. This method is called **retry**, and receives the number of attempts to start again if there's any error. When the maximum number of retries is reached, if there's another error, it will be thrown normally.

```

getProducts(): Observable<IProduct[]> {
  return this.http.get<{products: IProduct[]}>(this.productURL).pipe(
    map(response => response.products),
    catchError((resp: HttpResponse) => throwError(`Error getting
products. Status: ${resp.status}. Message: ${resp.message}`))
  );
}

```

In this example, it will make the Http GET call 3 times (maximum), if there's an error with that call. If there's still an error, it will go to the `catchError` function.

```
http.get(...).pipe( retry(3), map(...), catchError(...) ).subscribe(...);
```

Transforming our data in the Service

From the server we'll get a JSON object, and usually the data we want is inside that object. Sometimes we'll also need to make some transformations to the data

received. That's what the **map** function is for.

It's also a good practice to create interfaces that represent the server's response:

```
export interface IProduct {
  id?: number;
  description: string;
  price: number;
  available: string;
  imageUrl: string;
  rating: number;
}

export interface ResponseProducts {
  products: IProduct[];
}
```

We are writing code in TypeScript, so it's helpful that when we get that data, we cast it to the type we want before returning it (using **map**). In this case, we'll want to return an Observable with an array of IProduct objects:

```
...
import { Observable, throwError } from 'rxjs';
import { map, catchError, retry } from 'rxjs/operators';
...
export class ProductService {




  private productURL = 'http://SERVER/products-angular/products';

  constructor(private http: HttpClient) { }

  getProducts(): Observable<IProduct[]> {
    return this.http.get<ResponseProducts>(this.productURL).pipe(
      map(response => response.products),
      catchError((resp: HttpResponse) => throwError(`Error getting
products. Status: ${resp.status}. Message: ${resp.message}`))
    );
  }
}
```

Finally, in the **ProductListComponent** class, we subscribe to the observable returned by the service and assign the array when it's returned. We can also define a function that will be executed when the Observable throws any error and a third function that will be executed always even if there was an error (for example to indicate with a boolean that the request has finished).

```
ngOnInit() {
  this.productService.getProducts()
    .subscribe(
      prods => this.products = prods, // Success function
      error => console.error(error), // Error function (optional)
      () => console.log("Products loaded") // Finally function (optional)
    );
}
```

| Hide images | Product | Price | Available | Rating |
|---|------------------------|---------|------------|--------|
|  | DDR3 1600 8GB 2x4GB | €49.95 | 14/11/2016 | ★★★★☆ |
|  | Toshiba SSD Q300 480GB | €119.00 | 02/11/2016 | ★★★★★ |
|  | Sony PS4 500GB | €249.95 | 14/11/2016 | ★★★★☆ |

Other HTTP methods (post, put, delete)

The difference between GET/DELETE and POST/PUT, is that the latter HTTP methods require data to be sent in the body of the request. We must send a JSON object as the second parameter after the URL.

In the following example we are going to change a product's rating in the server by calling `'/products/rating/:id'` by **PUT** and sending the new rating in a JSON object `{rating: newRating}`. Let's add the necessary method in the `ProductService` class.

```
export class ProductService {
  ...

  changeRating(idProduct: number, rating: number): Observable<boolean> {
    return this.http.put<void>(this.productURL + '/rating/' + idProduct,
      {rating: rating}).pipe(
        catchError((resp: HttpResponse) => throwError(`Error changing rating.
Status: ${resp.status}. Message: ${resp.message}`)))
  };
}
```

Next, from the **ProductItemComponent**, we call the service when we detect that the rating has changed and subscribe to the observable. When the service responds, if there's no error, we change the product's rating. Try to reload the page to see that the change is permanent on the server.

```
export class ProductItemComponent implements OnInit {
  ...
  changeRating(rating: number) {
    this.productService.changeRating(this.product.id, rating)
      .subscribe(
        () => this.product.rating = rating,
        error => console.error(error)
      );
  }
  ...
}
```

If you need to send to the server a partial object, instead of setting in the interface a lot of properties to optional '?', you can use TypeScript [Partial](#) type.

Sending authentication token in Angular

The simplest way for sending a JWT authentication token is, in every service's method that does an HTTP call (and needs to send the token), sending a **HttpHeaders** object inside any http call. Don't forget to store the token when you log in.

```
export class ProductService {
  ...
  getProducts(): Observable<IProduct[]> {
    let options = {
      headers: new HttpHeaders().set('Authorization',
        localStorage.getItem('ap-token'))
    };
    return this.http.get(this.productsURL, options).pipe(...)
  }
  ...
}
```

Using an interceptor to send the token

[Interceptors](#) are services used to process every incoming (request) or outgoing (response) HTTP communication. To create an interceptor, just create a service and implement the [HttpInterceptor](#) interface. This will require a method called **intercept**, that will receive and process every HTTP request.

ng g interceptor interceptors/auth

```
@Injectable()
export class AuthInterceptor implements HttpInterceptor {
  constructor() {}

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    const token = localStorage.getItem('token');

    if (token) { // Clone the request to add the new header.
      const authReq = req.clone({
        headers: req.headers.set('Authorization', token)});
      // Pass on the cloned request instead of the original request.
      return next.handle(authReq);
    }
    return next.handle(req);
  }
}
```

You will receive 2 parameters:

- **req: HttpRequest** → This is the original request before sending it to the server. You can modify what you send by **cloning** this request and setting a new headers, etc...
- **next: HttpHandler** → If you have only one interceptor, this will be the [Angular Backend](#) service. By calling handle, you will pass the modified (or original) request to this service which will send it to the server.

To tell Angular it must use this interceptor, just declare it as a service in your App module, but with an especial syntax:

```
import {HTTP_INTERCEPTORS} from '@angular/common/http';
...

@NgModule({
  providers: [{
    provide: HTTP_INTERCEPTORS,
    useClass: AuthInterceptor,
    multi: true,
  }],
})
export class AppModule {}
```

By implementing an interceptor for usual tasks like sending a token, you only need to do it once (in the intercept method) instead of doing it in every method that sends an HTTP request.

Using an interceptor to set the base url for services

Inside **src/environments** we can find 2 different files: **environment.ts** which is loaded in development mode and **environment.prod.ts**, which is loaded in production mode. In these files, there's an object called **environment**. We'll use this object to put

properties we want to be different when we are in production or development mode.

For example, the services url could be different, as we won't want to use production services when we are developing our app. Let's create a property (in **both** files) called **baseUrl**.

```
// environment.ts
export const environment = {
  production: false,
  baseUrl: 'http://localhost:3000'
};

// environment.prod.ts
export const environment = {
  production: true,
  baseUrl: 'http://localhost:3000' // Here should go the production url
};
```

Now, we are going to create an interceptor that prepends this base url to the partial url of the service we are calling (don't use full urls in angular services now).

ng g interceptor interceptors/base-url

```
import { Injectable } from '@angular/core';
import { HttpInterceptor, HttpRequest, HttpHandler, HttpEvent } from
'@angular/common/http';
import { Observable } from 'rxjs';
import { environment } from 'src/environments/environment';

@Injectable()
export class BaseUrlInterceptor implements HttpInterceptor {
  intercept(
    req: HttpRequest<any>,
    next: HttpHandler
  ): Observable<HttpEvent<any>> {
    const reqClone = req.clone({
      url: `${environment.baseUrl}/${req.url}`
    });
    return next.handle(reqClone);
  }

  constructor() {}
}
```

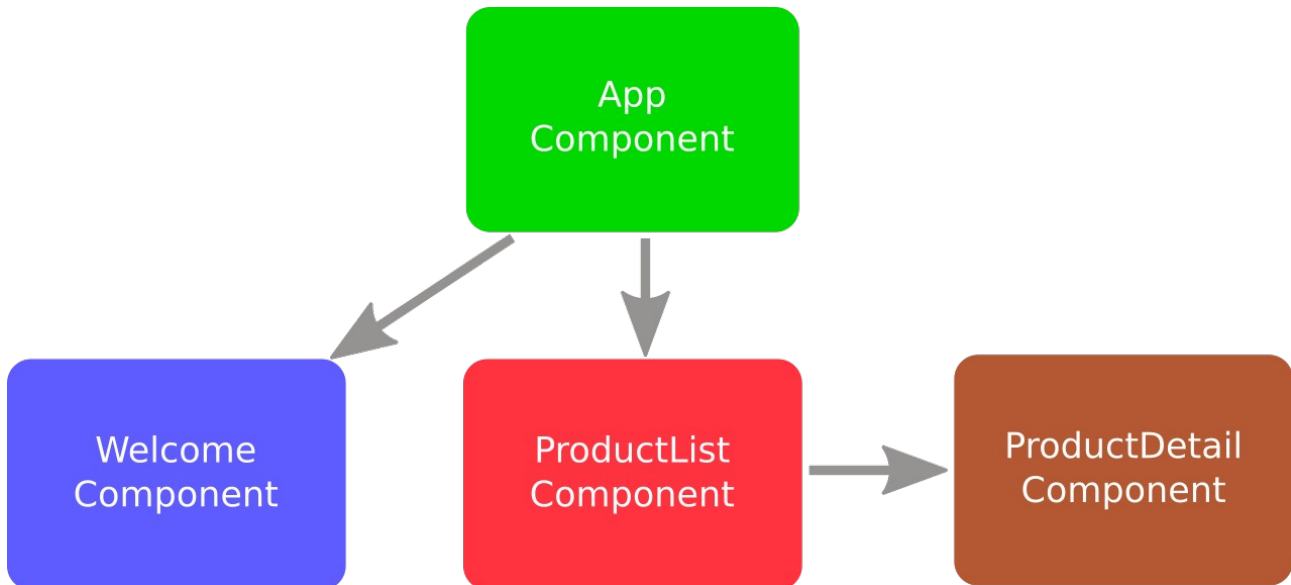
And don't forget to include the interceptor in your AppModule:

```
@NgModule({
  ...
  providers: [
    {
      provide: HTTP_INTERCEPTORS,
      useClass: BaseUrlInterceptor,
      multi: true,
    },
    ...
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Routing basics

As we said earlier, with Angular, we are building a SPA (Single Page Application). This means that index.html is the **only** HTML page downloaded from the server and all our application is handled from this document. But, how can we navigate to another view in our application?. This is handled by the **Angular Router**.

In our example application, we are going to add 2 more views to display, apart from the products list. Those views are the Welcome view (loaded in the first place), and the ProductDetail view (loaded when we click on a product's name in the list).



First of all, we are going to create the two components missing (src/app):

```
ng g component welcome
```

```
ng g component product-detail
```

Secondly, we are going to create a separate module for app routing instead of declaring the routes in the main app module (cleaner code). If we create a project (or another module) with --routing option, it creates a separate module with routes. As we didn't create it before, we'll do it now manually.

```
ng generate module app-routing --module app --flat
```

Before defining routes, we need to import the Router module from Angular in our AppRoutingModuleModule class.

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

@NgModule({
  declarations: [],
  imports: [
```

```

    CommonModule,
    RouterModule
  ]
})
export class AppRoutingModuleModule { }

```

However, in order to define routes, we need to call a method on the `RouterModule` object. This method is **`forRoot()`**, and we'll pass the URLs our application will have, and which component should be loaded on each case.

Our routes will be:

- **`/welcome`** → Should load the **`WelcomeComponent`** that welcomes us.
- **`/products`** → Should load the **`ProductListComponent`** that shows the list of products in our application.
- **`/products/:id`** → Should load the **`ProductDetailComponent`** that shows detailed information about a single product. This route will have one parameter (the product's id).
- By default and when the route is not correct, the **`WelcomeComponent`** route will be loaded.

These are the changes that we should make to the **`app-routing.module.ts`** file (this module is used only to define the application routes):

```

const routes: Routes = [
  { path: 'welcome', component: WelcomeComponent },
  { path: 'products', component: ProductListComponent },
  // :id is a parameter (product's id)
  { path: 'products/:id', component: ProductDetailComponent },
  // Default route (empty) -> Redirect to welcome page
  { path: '', redirectTo: '/welcome', pathMatch: 'full' },
  // Doesn't match any of the above
  { path: '**', redirectTo: '/welcome', pathMatch: 'full' }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModuleModule { }

```

There's still one more thing to do, or the application won't work. We have to include a **`<router-outlet>`** element in our main **`AppComponent`** template. Inside this element, the router will load the different views / components. We'll also include a navigation bar that allows us to go to the **`welcome`** and **`products`** routes.

To make those links work, we use the directive **`[routerLink]`** instead of **`href`**.

`app.component.html`

```

<nav class="navbar navbar-expand navbar-dark bg-dark mb-4">
  <a class="navbar-brand" href="#">{{title}}</a>
  <ul class="navbar-nav mr-auto">
    <li class="nav-item">
      <a class="nav-link" [routerLink]="['/welcome']">

```

```

        [routerLinkActive]="['active']">Welcome</a>
    </li>
    <li class="nav-item">
        <a class="nav-link" [routerLink]="['/products']"
            [routerLinkActive]="['active']">Products</a>
    </li>
</ul>
</nav>
<div class="container-fluid">
    <router-outlet></router-outlet>
</div>

```

The **[routerLinkActive]** directive, is used to assign on or more classes to an element when that route is active (so we can see better in which section we are in).

Changing the page's title

Page's title is located in the **index.html** file. In this file is where all libraries and styles are loaded, and also the AppComponent. Angular only has direct control of what happens inside the **<app-root>** tag (it can have other name). So, the **<title>** element **cannot** be set with something like **<title>{{title}}</title>**.

However, Angular has a built-in service called **Title** that can be used to change the page's title whenever we need (for example, when we navigate to another view). This service's class must be imported from **'@angular/platform-browser'**.

app.module.ts

```

import { BrowserModule, Title } from '@angular/platform-browser';
...

@NgModule({
    ...
    providers: [Title, ProductService],
    ...
})
export class AppModule { }

```

welcome.component.ts

```

import { Title } from '@angular/platform-browser';
...

@Component({
    ...
})
export class WelcomeComponent implements OnInit {

    constructor(private titleService: Title) { }

    ngOnInit() {
        this.titleService.setTitle('Welcome to Angular Products!');
    }

}

```

Advanced routing

In this section we are going to see how to pass parameters to a route (for example, a product's id), go to a route by code instead of using a link, detect when we are leaving the actual page (to save changes for example), and protect a route from being accessed under some conditions (not having logged in).

Passing parameters to a route

For example, when we want to go to the product's detail page, we have to indicate which product are we going to see. If you remember, the route defined for this view is **path: 'products/:id'**. Names starting with ':' are parameters passed to this route (in the browser you'll see something like **product/4** where **4** is the product's **id**).

Let's see how to call this route defining the product's description as a **link** to go there, passing the current product's id:

```
# product-item.component.html
...
<div class="col-4">
  <a [routerLink]="['/products', product.id]">
    {{ product.description }}
  </a>
</div>
...
```

As you can see, after the route name, the next element in the array is the first and only route parameter (id). If a route has more than one parameter, we just need to include them in order inside the array of the [routerLink] directive.

Now, we are going to receive that parameter (product's id) in the destination component: **ProductDetailComponent**. First of all, we need to inject the Angular's **ActivatedRoute** service, which allows us to access the route's parameters (id).

```
import { IProduct } from '../product-item/iproduct';
import { ProductService } from '../shared/product.service';
import { ActivatedRoute } from '@angular/router';
import { Component, OnInit } from '@angular/core';

...
export class ProductDetailComponent implements OnInit {
  product: IProduct;

  constructor(private route: ActivatedRoute,
               private productService: ProductService) { }

  ngOnInit() {
    const id = +this.route.snapshot.params['id'];
    this.productService.getProduct(id)
      .subscribe(
        p => this.product = p,
        error => console.error(error)
      );
  }
}
```

Showing async object data

For example, when we go to our product's detail page, we'll get those product's details from the server. This can be seen as async data. Before we get that product from the Observable (web service call), we have an undefined value (**product**). If angular tries to show a property from the object (ex: **product.description**) in the view before the data has been received, it will throw an **error**.

We could create an object with "empty" properties (empty strings, number = 0, ...), and that would solve the problem. Those empty properties would be shown before getting the product. After receiving the product, the real properties will be shown.

Another solution is to use the optional operator '?' after the object's name (example: **product?.description**). This will tell Angular to get that object's property only when the object exists (not undefined or null). We could also use ***ngIf="product"** instead (in fact, when another component depends on that object it's better to use this solution).

product-detail.component.html

```
<div class="card">
  <div class="card-header bg-primary text-white">
    {{product?.description}}
  </div>
  <div class="card-block p-3 text-center">
    <img [src]="product?.imageUrl" alt="">
    <div>Price: {{ product?.price | currency:'EUR':'symbol' }}</div>
    <div>Available since: {{ product?.available | date:'dd/MM/y' }}</div>
    <div>
      <ap-star-rating *ngIf="product" [rating]="product.rating"
        (ratingChanged)="changeRating($event)"></ap-star-rating>
    </div>
    <div>
      <button type="button" class="btn btn-default" (click)="goBack()">
        Go back
      </button>
    </div>
  </div>
</div>
```

Activate a route with code

We have added a "Go back" button to the product details page. Instead of using **[routerLink]**, we'll activate the route with TypeScript in the component's class. For that purpose, we have to inject the Router service in the component:

```
...
import { ActivatedRoute, Router } from '@angular/router';
...
export class ProductDetailComponent implements OnInit {
  ...
  constructor(private router: Router, ...) { }
  ...
  goBack() {
    this.router.navigate(['/products']);
  }
}
```

As you can see, activating a route with code is very similar to using the **[routerLink]** directive. It uses an array and the first element is the route. If that route needed parameters, we'll just have to pass them in the array.

Guard navigation to a route with CanActivate

Angular can control if we can go to a route or leave it using guards. Guards are special classes (services) used in combination with the Angular Router and we'll see soon how to use them. In this section, we are going to see how to prevent the user from going to a specific route if, for example, the user is not logged in. To achieve this, we'll use the **CanActivate** guard.

Guards are **services**, so they implement the **@Injectable** decorator. In this case, a CanActivate guard service must implement the **CanActivate interface**. This makes necessary to create a method called **canActivate** that returns a boolean (if we can go to that route or not).

First of all, we'll create the service (in the shared folder for example:

ng g guard guards/product-detail

Now, we rename the class to **ProductDetailGuard**, and implement the **CanActivate** interface. For now, we'll return true, so it always goes to that route.

```
import { CanActivate } from '@angular/router';
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class ProductDetailGuard implements CanActivate {
  constructor() { }

  canActivate(): boolean {
    return true;
  }
}
```

In order to use this service, we must register it in the AppModule's providers, and also in the route we want to guard (in this case the route to ProductDetailComponent). To guard a route, we use the property **canActivate**. It's an array because we can use many CanActivate guards in one route (if any fails, it won't go to that route):

```
...
{
  path: 'products/:id',
  canActivate: [ProductDetailGuard],
  component: ProductDetailComponent
},
...
```

Now, we are going to implement the **canActivate** method. This method receives 2 parameters (the second parameter is usually not used and can be omitted). In this method we'll check that the id is a number. If we also wanted to check if user is logged in (we'll need to create a login page or it won't make any sense), we should create another CanActivate guard and reuse it in more routes.

```
import { ActivatedRouteSnapshot, CanActivate, Router, RouterStateSnapshot } from
 '@angular/router';
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class ProductDetailGuard implements CanActivate {

  constructor(private router: Router) { }

  canActivate(route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): boolean | UrlTree {
    let id = +route.params['id'];
    if(isNaN(id) || id < 1) {
      alert("Invalid product id!");
      return this.router.createUrlTree(['/products']); // Go to products page
    }
    return true;
  }
}
```

The `canActivate` method, apart from a **boolean** value, it can also return a **Promise<boolean>** or **Observable<boolean>**. The Angular route will subscribe automatically to that observable or promise and go to that route (if everything ok) when the promise or observable returns a true value.

Starting since Angular 7.2, guards can also return a `UrlTree` to force navigation to another page when the route couldn't be activated.

Guard navigation from a route with `CanDeactivate`

A `CanDeactivate` guard prevents the user from leaving the current page if certain conditions are met. For example, the user is editing some product's information and hasn't saved the changes. We can use this guard to show her/him a message telling to save (or cancel) changes before leaving the route.

We're going to create a new page (`/product/edit/:id` → `ProductEditComponent`) that will allow us to edit a product's information. When the user tries to leave this page, a confirmation message will be shown.

```
{
  path: 'product/edit/:id',
  canActivate: [ProductDetailGuard],
  component: ProductEditComponent
}
```

In the product's detail page (**`product-detail.component.ts`**), we'll add a button that will send us to this newly created page:

```
<button type="button" class="btn btn-default" (click)="edit()">
  Edit product
</button>
```

And this is the method this button will call:

```
edit() {
  this.router.navigate(['/products/edit', this.product.id]);
}
```


To implement this guard, we must create a service like before. This time, we'll have to implement the `CanDeactivate` interface. This interface will need a component as the generic type \rightarrow `CanDeactivate<Component>`.

ng g guard guards/can-deactivate

```
import { ActivatedRouteSnapshot, CanDeactivate, RouterStateSnapshot } from
 '@angular/router';
import { Injectable } from '@angular/core';

import { ProductEditComponent } from '../product-edit/product-edit.component';

@Injectable({
  providedIn: 'root'
})
export class CanDeactivateGuard implements CanDeactivate<ProductEditComponent> {

  constructor() { }

  canDeactivate(component: ProductEditComponent, route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot) {
    // We can call any component method and access any public property
    return confirm("Do you want to leave this page?. Changes can be lost");
  }
}
```

The `canDeactivate` method must return a **boolean**, a **Promise<boolean>** or an **Observable<boolean>**. And in the routes configuration, we'll have to tell Angular to use this guard when the user tries to leave the corresponding route:

```
{
  path: 'products/edit/:id',
  canActivate: [ProductDetailGuard],
  canDeactivate: [CanDeactivateGuard],
  component: ProductEditComponent
}
```

As you can see, if we have to specify the component for this guard, we can only use it in one route. But there's a way to create a reusable `CanDeactivate` guard. We'll create an interface called **CanComponentDeactivate** that every component will implement with a method called `canDeactivate`. Instead of using a component as the type, we'll use this interface, so it can be used with any component that implements it.

```
import { Observable } from 'rxjs/Observable';
import { ActivatedRouteSnapshot, CanDeactivate, RouterStateSnapshot } from
 '@angular/router';
import { Injectable } from '@angular/core';

import { ProductEditComponent } from '../product-edit/product-edit.component';

export interface CanComponentDeactivate {
  canDeactivate: () => Observable<boolean> | Promise<boolean> | boolean;
}

@Injectable()
export class CanDeactivateGuard implements CanDeactivate<CanComponentDeactivate>
{

  constructor() { }

  canDeactivate(component: CanComponentDeactivate, route:
    ActivatedRouteSnapshot,
    state: RouterStateSnapshot) {
```

```

    // If the component implements the interface's method, we'll call it.
    return component.canDeactivate? component.canDeactivate() : true;
  }
}

```

This is a better solution if we want to reuse this guard with more than 1 component. We'll just implement the **CanComponentDeactivate** interface in the components we want and use this guard on those routes.

Pre-fetch data before activating a route with Resolve

What if we want to get data (for example a product), before going to a route that needs it? (like the product's detail page). Instead of getting the product from the id in the ProductDetailComponent, we'll get it before loading the route using a **Resolve guard**. This way, we'll know that when the component is loaded, the product will be immediately available.

ng g resolver resolvers/product-detail

This Resolve guard can return primitive data, a promise or an observable (the Angular router will **subscribe** to it automatically). We'll also protect our service against errors using the Observable's **catch** method. In this catch method we'll return an empty observable and redirect to the product's list, if there was an error getting the product. Let's take a look at **product-detail-resolve.service.ts**:

```

import { Observable, of, EMPTY } from 'rxjs';
import { catchError } from 'rxjs/operators';
...

@Injectable()
export class ProductDetailResolve implements Resolve<Product>{

  constructor(private productService: ProductService,
               private router: Router) { }

  resolve(route: ActivatedRouteSnapshot): Observable<Product> | Resolve<never> {
    return this.productService.getProduct(route.params['id']).pipe(
      catchError(error => {
        this.router.navigate(['/products']);
        return EMPTY;
      }));
  }
}

```

If there's any error, we'll be redirected to the 'products' page. If everything goes well, we will get a product from the server before going to the product's detail page. This is how we implement this resolve in the route:

```

{
  path: 'products/:id',
  canActivate: [ProductDetailGuard],
  component: ProductDetailComponent,
  resolve: {
    product: ProductDetailResolve
  }
}

```

In the ProductDetailComponent's class, we can get the product from the router like this (there's no need anymore to get that data from the server here, it will already

be in the route's data).

```
...
export class ProductDetailComponent implements OnInit {
  ...
  ngOnInit() {
    this.product = this.route.snapshot.data['product'];
  }
  ...
}
```