Navigating through Application with Routing

In previous chapters, we did a great job of separating concerns and adding different layers of abstraction to increase the maintainability of an Angular application. However, we have barely concerned ourselves with the user experience that we provide throughout the application.

Currently, our user interface is bloated, with components scattered across a single screen. We need to provide a better navigational experience and a logical way to intuitively change the application's view. Now is the right time to incorporate routing and split the different areas of interest into pages, interconnected by a grid of links and URLs.

So, how do we deploy a navigation scheme between components of an Angular application? We use the Angular router, which was built with componentization in mind, and create custom links for our components to react to.

In this chapter, we will do the following:

- Introduce the Angular router
- Create an Angular application with routing
- Create feature routing modules
- Pass parameters to routes
- Enhance navigation with advanced features

Technical requirements

The chapter contains various code samples to walk you through the concept of routing in the Angular framework. You can find the related source code in the `ch09` folder of the following GitHub repository:

https://www.github.com/PacktPublishing/Learning-Angular-Fourth-Edition

Introducing the Angular router

In traditional web applications, when we wanted to change from one view to another, we needed to request a new page from the server. The browser would create a URL for the view and send it to the server. The browser would then reload the page as soon as the client received a response. It was a process that resulted in round trip time delays and a bad user experience for our applications:

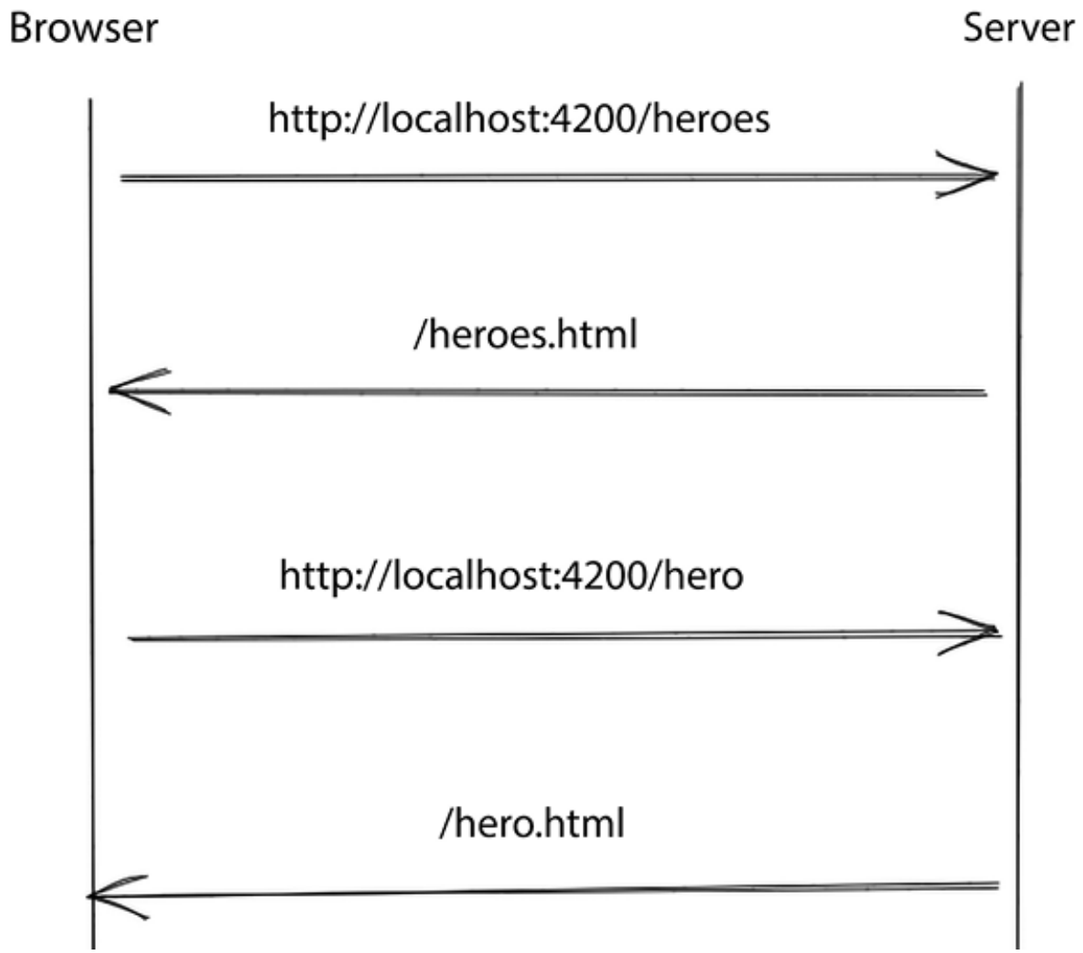
Browser
Server
http://localhost:4200/heroes
/heroes.html
http://localhost:4200/hero
/hero.html

Figure 9.1: Routing in traditional web applications

Modern web applications using JavaScript frameworks such as Angular follow a different approach. They handle changes between views or components on the client side without bothering the server. They contact the server only once during bootstrapping to get the main HTML file. Any subsequent URL changes are intercepted and handled by the router on the client. These types of applications are called **Single-Page Applications** (**SPAs**) because they do not cause a full reload of a page:

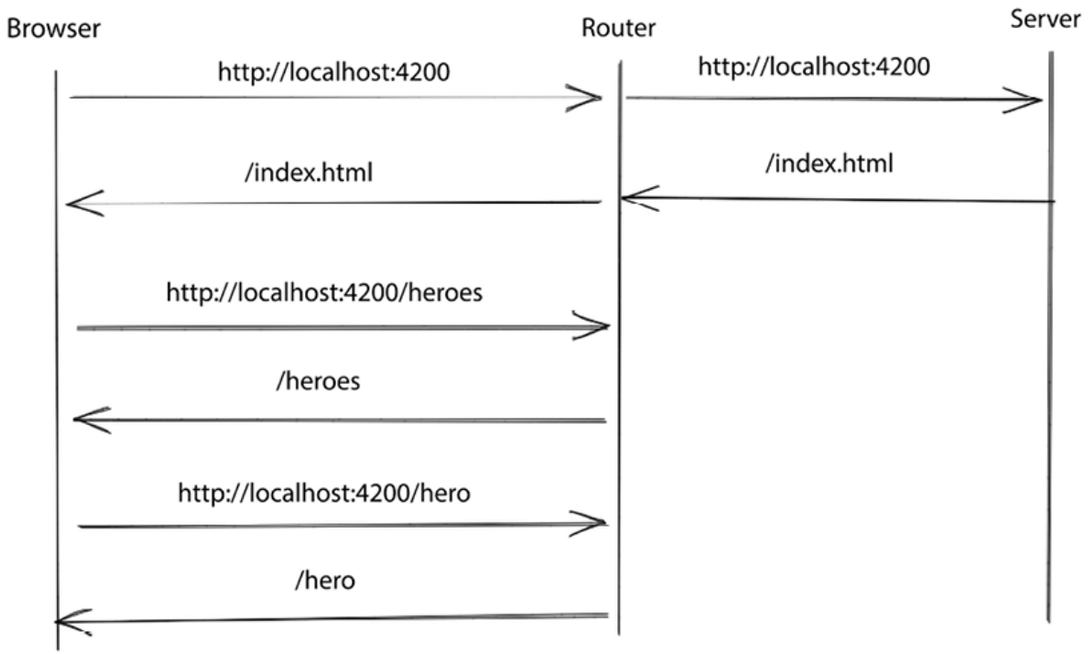

Figure 9.2: SPA architecture

The Angular framework provides the `@angular/router` `npm` package, which we can use to navigate between different components in an Angular application. Adding routing in an Angular application involves the following steps:

1. Specify the base path for the Angular application.
2. Use an appropriate Angular module from the `@angular/router` package.
3. Configure different routes for the Angular application.
4. Decide where to render components upon navigation.

In the following sections, we will learn the basics of Angular routing before diving deeper into hands-on examples.

Specifying a base path

As we have already seen, modern and traditional web applications react differently when a URL changes inside the application. The architecture of each browser plays an essential part in this behavior. Older browsers initiate a new request to the server when the URL changes. Modern browsers, also known as **evergreen** browsers, can change the URL and the history of the browser when navigating in different views without sending a request to the server, using a technique called **pushState**.

HTML5 `pushState` allows in-app navigation without causing a full reload of a page and is supported by all modern browsers.

An Angular application must set the `base` HTML tag in the `index.html` file to enable pushState routing:

```
<!doctype html>
  <html lang="en">
  <head>
    <meta charset="utf-8">
    <title>MyApp</title>
    <base href="/">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="icon" type="image/x-icon" href="favicon.ico">
  </head>
  <body>
    <app-root></app-root>
  </body>
  </html>
```

The `href` attribute informs the browser about the path it should follow when attempting to load external resources, such as media or CSS files, once it goes deeper into the URL hierarchy.

The Angular CLI automatically adds the tag when creating a new Angular application and sets the `href` value to the application root, `/`. If your application resides in a different folder than the `src\app`, you should name it after that folder.

Importing the router module

The Angular router library contains `RouterModule`, an Angular module that we need to import into our application to start using the routing features:

```
import { RouterModule } from '@angular/router';
```

We import `RouterModule` in the main application module, `AppModule`, using the `forRoot` method:

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

```
  imports: [
    BrowserModule,
    RouterModule.forRoot(routes)
  ],
  providers: [],
  bootstrap: [AppComponent]
})
```

The `forRoot` pattern is used when a module defines services and other declarable artifacts such as components and pipes. If we try to import it normally, we will end up with multiple instances of the same service, thereby violating the singleton pattern. It works similarly to when we provide a service to the `root` injector of the application.

The `forRoot` method of `RouterModule` returns an Angular module that contains a set of Angular artifacts related to routing:

- Services to perform common routing tasks such as navigation
- Directives that we can use in our components to enrich them with navigation logic

It accepts a single parameter, which is the route configuration of the application.

Configuring the router

The `routes` variable that we pass in the `forRoot` method is a list of `Routes` objects that specify what routes exist in the application and what components should respond to a specific route. It can look like the following:

```
const routes: Routes = [
    { path: 'products', component: ProductListComponent },
    { path: '**', component: PageNotFoundComponent }
  ];
```

Each route definition object contains a `path` property, which is the URL path of the route, and a `component` property that defines which component will be loaded when the application navigates to that path.

The value of a `path` property should not contain a leading `/`.

Navigation in an Angular application can occur either manually by changing the browser URL or by navigating using in-app links. In the first scenario, the browser will cause the application to reload, while the second will instruct the router to navigate along a route path in the application code. In our case, when the browser URL contains the `/products` path, the router creates an instance of `ProductListComponent` and displays its template on the page. On the contrary, when the application navigates to `/products` by code, the router follows the same procedure and additionally updates the URL of the browser.

If the user tries to navigate to a URL that does not match any route, Angular activates a custom type of route called a **wildcard**. The wildcard route has a `path` property with two asterisks and matches any URL. The `component` property is usually an application-specific `PageNotFoundComponent` or the main component of the application.

Rendering components

One of the directives the Angular router makes available in an Angular application is `router-outlet`. It is used as an Angular component and a placeholder for components activated with routing.

Typically, the main component of an Angular application is used only for providing the main layout of the application and orchestrating all other components. We should write it once, forget it, and not modify it when we want to add a new feature to our application. So, a typical example of the `app.component.html` file is the following:

```
<app-header></app-header>
  <router-outlet></router-outlet>
  <app-footer></app-footer>
```

In the preceding HTML template, the <`app-header`> and <`app-footer`> elements are layout components. The <`router-outlet`> element is where all other components will be rendered using routing. In reality, these components are rendered as a sibling element of the `router-outlet` directive.

We have already covered the basics and provided a minimal router setup. In the next section, we will look at a more realistic example and further expand our knowledge of the routing module and how it can help us.

Creating an Angular application with routing

Whenever we have created a new Angular application through the course of this book so far, the Angular CLI has asked us whether we wanted to add routing, and we have always replied *no*. Well, it is time to respond positively and enable routing to our Angular application! In the following sections, we will put into practice all the basics that we have learned about routing:

- Scaffolding an Angular application with routing
- Adding route configuration to our Angular application
- Navigating to application routes

At the end of this section, we will have built a simple application with complete routing capabilities.

Scaffolding an Angular application with routing

We will use the Angular CLI to create a new Angular application from scratch. Run the following Angular CLI command to create a new Angular application named `my-app`:

```
ng new my-app --routing --style=css
```

The preceding command uses runtime options to skip the interactive workflow of the Angular CLI that asks questions about the application we want to build. It uses the following command-line options:

- `--routing`: Imports the Angular router to configure navigation for our application
- `--style=css`: Configures our application to use CSS for component styling

The previous Angular CLI command generates roughly the same files as usual but with one exception, the `app-routing.module.ts` file:

```
import { NgModule } from '@angular/core';
  import { RouterModule, Routes } from '@angular/router';
  const routes: Routes = [];
  @NgModule({
    imports: [RouterModule.forRoot(routes)],
    exports: [RouterModule]
  })
```

```
export class AppRoutingModule { }
```

The `app-routing.module.ts` file is the routing module of the main application module. The Angular CLI names the routing module file after the actual module, appending the `-routing` suffix. It is a convention that helps us to quickly identify whether a module has routing enabled and what the respective routing module is. The name of the TypeScript class of the routing module also follows a similar convention.

The preceding file exports the `AppRoutingModule` class, an Angular module used to configure and enable the router in our application. It imports `RouterModule` using the `forRoot` method, as we have already learned in the previous section. It also re-exports `RouterModule` so that components of other modules that import `AppRoutingModule` have access to router services and directives. By default, `AppModule` imports `AppRoutingModule` in the `app.module.ts` file, so all the components of our application are enabled with routing capabilities:

```
import { NgModule } from '@angular/core';
  import { BrowserModule } from '@angular/platform-browser';
  import { AppRoutingModule } from './app-routing.module';
  import { AppComponent } from './app.component';
```

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

We mentioned in the previous section that `AppModule` imports `RouterModule` directly. We could have followed that approach for a minimal route configuration, but we suggest creating a separate routing module for the following reasons:

- We can change the route configuration of the application anytime, independent of the Angular module that imports it.
- We can easily test the Angular module without enabling routing. Routing is difficult to manage in

unit tests.
- From the existence of a routing module, we can quickly understand that an Angular module supports routing.

Routing modules are used not only in the main application module but also in feature modules, as we will learn in the following section.

Configuring routing in our application

When we start designing the architecture of an Angular application with routing, it is easier to think about its main features as links that users can click to access. Products and shopping carts are basic features of the e-shop application we are currently building. Adding links and configuring them to activate certain features of an Angular application is part of the route configuration of the application.

To set up the route configuration of our application, we need to follow the steps below:

1. Run the following command to create a new Angular component named `cart`:

```
ng generate component cart
```

2. Run the following command to create a new Angular component named `products`:

```
ng generate component products
```

3. Open the `app-routing.module.ts` file and add the following `import` statements:

```
import { CartComponent } from './cart/cart.component';
  import { ProductsComponent } from './products/products.component';
```

4. Add two route definition objects in the `routes` variable:

```
const routes: Routes = [
    { path: 'products', component: ProductsComponent },
    { path: 'cart', component: CartComponent }
  ];
```

In the preceding snippet, the `products` route will activate `ProductsComponent`, and the `cart` route

will activate `CartComponent`.

5. Open the `app.component.html` file and replace its content with the following HTML template:

```
<a routerLink="/products">Products</a>
  <a routerLink="/cart">Cart</a>
  <router-outlet></router-outlet>
```

In the preceding template, we use two router directives to perform navigation in our application, the `router-outlet` directive we have already seen and the `routerLink`. We apply the `routerLink` directive to anchor HTML elements, and we assign the route path in which we want to navigate as a value. Notice that the path should start with `/` as opposed to the `path` property in the route definition object.

6. Open the `styles.css` file and add the following CSS styles:

```
a {
    color: #1976d2;
    text-decoration: none;
    margin: 5px;
```

```
}
a:hover {
  opacity: 0.8;
}
```

We are now ready to preview our Angular application. Run the `ng serve` command and click on the **Products** link. The application should display the template of `ProductsComponent`. It should also update the URL of the browser to match the path of the route.

Now try to do the opposite. Navigate to the `root` path at `http://localhost:4200` and append the `/products` path at the end of the URL. The application should behave the same as before:

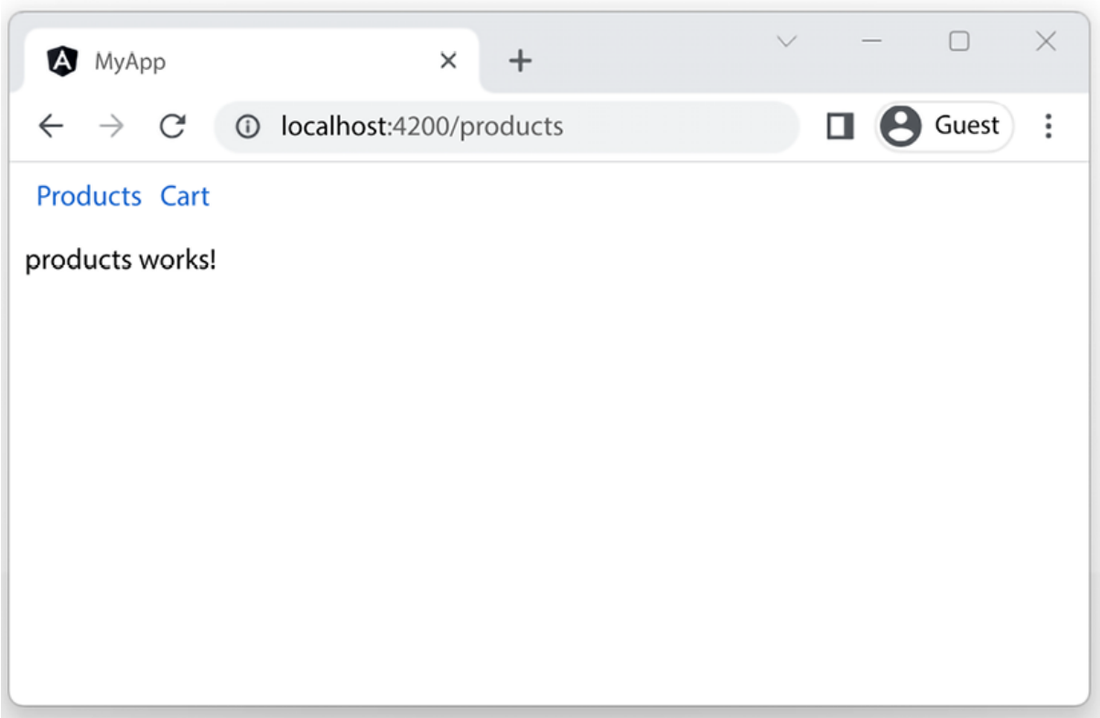

Figure 9.3: Products route

Congratulations! Your Angular application now supports in-app navigation. We have barely scratched the surface of routing in Angular. There are many router features for us to investigate waiting in

the following sections. For now, let's try to move our components to a separate feature module so that we can manage it independently of the main application module.

Creating feature routing modules

At this point, we have set up the route configuration so that routing works the way it should. However, this approach doesn't scale so well. As our application grows, more and more routes may be added to the routing module of the main application module. To overcome this problem, we should create a separate feature module for our components that will also have a dedicated routing module. We already created the products module in the previous chapter. We will use it in our application so that any product-related functionality is contained inside that module:

1. Copy the global CSS styles from the code samples of *Chapter 8*, *Communicating with Data Services over HTTP*, inside the `styles.css` file of the current Angular project.
2. Delete the `src\app\products` folder from the current Angular CLI project.
3. Copy the `products` and `auth` folders from *Chapter 8*, *Communicating with Data Services over*

HTTP, into the `src\app` folder of the current Angular project.

4. Remove any references to the `ProductsComponent` class from the `app.module.ts` file and import `ProductsModule` and `HttpClientModule`:

```
import { HttpClientModule } from '@angular/common/http';
  import { NgModule } from '@angular/core';
  import { BrowserModule } from '@angular/platform-browser';
  import { AppRoutingModule } from './app-routing.module';
  import { AppComponent } from './app.component';
  import { CartComponent } from './cart/cart.component';
  import { ProductsModule } from './products/products.module';
  @NgModule({
    declarations: [
      AppComponent,
      CartComponent
    ],
    imports: [
      BrowserModule,
      AppRoutingModule,
      ProductsModule,
      HttpClientModule
    ],
    providers: [],
```

```
    bootstrap: [AppComponent]
  })
  export class AppModule { }
```

5. Open the `app-routing.module.ts` file and modify the `products` path so that it loads `ProductListComponent`:

```
import { NgModule } from '@angular/core';
  import { RouterModule, Routes } from '@angular/router';
  import { CartComponent } from './cart/cart.component';
  import { ProductListComponent } from './products/product-list/product-list.component';
  const routes: Routes = [
    { path: 'products', component: ProductListComponent },
    { path: 'cart', component: CartComponent }
  ];
  @NgModule({
    imports: [RouterModule.forRoot(routes)],
    exports: [RouterModule]
  })
  export class AppRoutingModule { }
```

If we run the Angular application using the `ng serve` command and click on the **Products** link, it should show the product list from the Fake Store API:

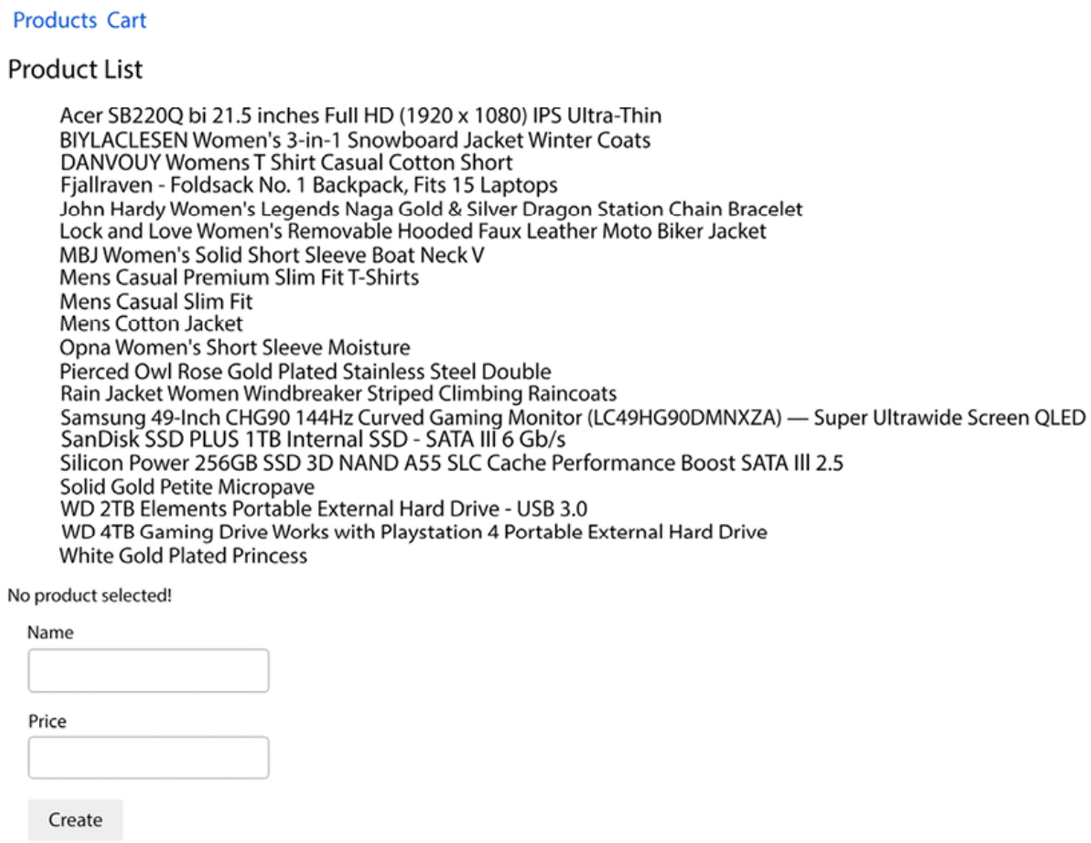

Figure 9.4: Product list

The route configuration for the `products` feature is still tied to the main application module. We will move the configuration into its own routing module inside the `src\app\products` folder:

1. Navigate to the `src\app\products` folder and create a new `products-routing.module.ts` file:

```
import { NgModule } from '@angular/core';
  import { RouterModule, Routes } from '@angular/router';
  import { ProductListComponent } from './product-list/product-list.component';
  const routes: Routes = [
    { path: 'products', component: ProductListComponent }
  ];
  @NgModule({
    imports: [RouterModule.forChild(routes)],
    exports: [RouterModule]
  })
  export class ProductsRoutingModule { }
```

In the previous snippet, you may have noticed that we do not import `RouterModule` using the `forRoot` method as we did before. Instead, we use the `forChild` method to import it. The `forChild` method is used when we want to register routes in a feature module. You should call

the `forRoot` method *only* in the routing module of the main application module.

2. Open the `products.module.ts` file and add the following `import` statement:

```
import { ProductsRoutingModule } from './products-routing.module';
```

3. Add the `ProductsRoutingModule` class in the `imports` array of the `@NgModule` decorator:

```
@NgModule({
    declarations: [
      ProductListComponent,
      ProductDetailComponent,
      SortPipe,
      ProductViewComponent,
      ProductCreateComponent
    ],
    imports: [
      CommonModule,
      ProductsRoutingModule
    ],
    exports: [ProductListComponent]
  })
```

4. Open the `app-routing.module.ts` file and remove the route definition object of the `products` path.
5. Open the `app.module.ts` file and change the location of `ProductsModule` in the `@NgModule` decorator so that it is imported before `AppRoutingModule`:

```
@NgModule({
    declarations: [
      AppComponent,
      CartComponent
    ],
    imports: [
      BrowserModule,
      ProductsModule,
      AppRoutingModule,
      HttpClientModule
    ],
    providers: [],
    bootstrap: [AppComponent]
  })
```

The order that we import routing modules does matter. The router selects a route with a first-match-wins strategy. We place feature routing modules, which contain more specific routes, before the main application routing module, which contains more generic routes. Thus, we want to force the router to search through our specific route paths and then fall back to an application-specific one.

If we run the Angular application using the `ng serve` command, we will see that it is working as before. We have not introduced any new features or done anything fancy, but we have paved the way to separating our route configurations effectively. The router combines the routes of our feature module, `ProductsModule`, with those of the main application module, `AppModule`. Thus, we can continue to work with routing in our feature module without modifying the main route configuration.

Currently, the route configuration of our application is pretty straightforward. However, there are scenarios that we need to take into account when working with routing in a web application, such as the following:

- Do we want to display a specific view when we bootstrap our application?
- What will happen if we try to navigate to a non-existing route path?

In the following section, we will explore how to handle the last case so that we do not break our

application.

Handling unknown route paths

We have already come across the concept of unknown routes in the *Introducing the Angular router* section. We set up a wildcard route to display `PageNotFoundComponent` when our application tries to navigate to a route path that does not exist. Now it is time to add that component for real:

1. Use the `generate` command of the Angular CLI to create a new component named `page-not-found`:

   ```
   ng generate component page-not-found
   ```

   Our application will display the newly generated component when navigating to an unknown route path.

2. Open the `page-not-found.component.html` file and replace its content with a meaningful HTML template:

```
<h3>Ooops!</h3>
  <p>The requested page was not found</p>
```

3. Open the `app-routing.module.ts` file and add the following `import` statement:

```
import { PageNotFoundComponent } from './page-not-found/page-not-found.component';
```

4. Add a new route definition object in the `routes` variable. Set the `path` property to double asterisks and the `component` property to the new component that we created:

```
const routes: Routes = [
    { path: 'cart', component: CartComponent },
    { path: '**', component: PageNotFoundComponent }
  ];
```

It is better to define the wildcard route in the `app-routing.module.ts` file. The wildcard route applies to the whole application, and thus it is not tied to a specific feature. Additionally, the wildcard route must be the last entry in the route list because the application should only reach it if there are no matching

routes.

If we run our application using the `ng serve` command and navigate to an unknown path, our application will display the following output:

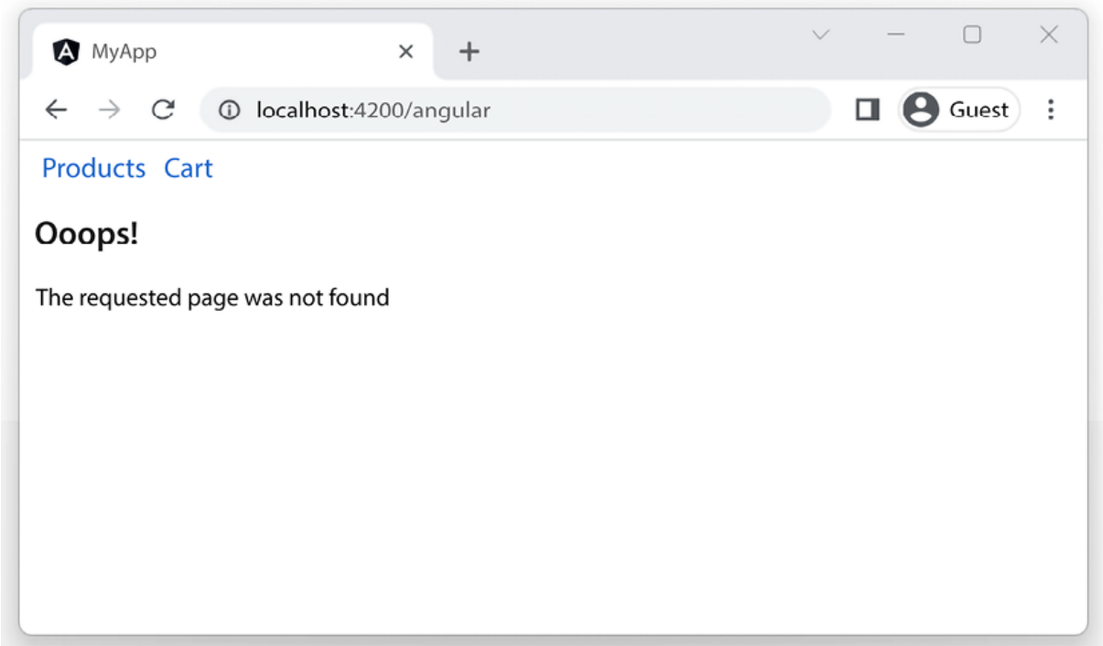

Figure 9.5: Page not found route

When the router encounters an unknown route, it navigates to the wildcard route, but the browser still points to the invalid URL.

Try to navigate to the root path of the application, `http://localhost:4200`, and you will notice that the template of `PageNotFoundComponent` is still visible on the screen. We have accidentally broken our application! How did this happen?

The `href` attribute of the `base` tag in the `index.html` file is the location at which an Angular application starts, as we learned in the *Introducing the Angular router* section.

The Angular CLI sets the value of `href` to `/` by default when creating a new Angular application. We have also learned that a route does not contain `/` in its `path` property. So, when our application bootstraps, it loads in an empty route path. According to our route configurations, we have not defined such a path. Thus, the router falls back to the wildcard route. We need to define a default route for our Angular application, which brings us to the first scenario we described: how to define a default route path when our application bootstraps.

Setting a default path

We set the `path` property of a route to an empty string to indicate that the route is the default one for an Angular application. In our case, we want the default route path to display the product list. Open the `products-routing.module.ts` file and add a new route definition object *at the end* of the existing routes:

```
import { NgModule } from '@angular/core';
  import { RouterModule, Routes } from '@angular/router';
  import { ProductListComponent } from './product-list/product-list.component';
  const routes: Routes = [
    { path: 'products', component: ProductListComponent },
    { path: '', redirectTo: '/products', pathMatch: 'full' }
  ];
  @NgModule({
    imports: [RouterModule.forRoot(routes)],
    exports: [RouterModule]
  })
  export class ProductsRoutingModule { }
```

In the preceding snippet, we tell the router to redirect to the `/products` path when the application navigates to the default route. The `pathMatch` property tells the router how to match the URL to the route path property. In this case, the router redirects to the `/products` path only when the URL matches the root path, which is the empty string.

If we run the application, we will notice that when the browser URL points to the root path of our

application, we are redirected to the `products` path, and the product list is displayed on the screen.

We added the empty route path after all other routes because, as we have already learned, the order of the routes is important. We want more specific routes before less specific ones. In the following diagram, you can see the order in which the router resolves paths in our application:

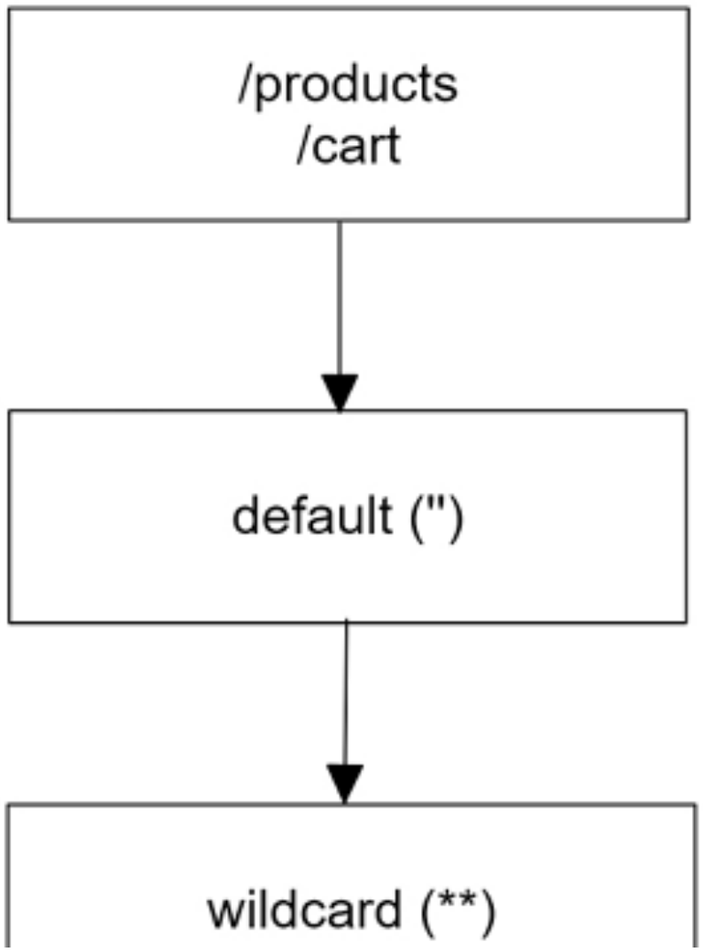
/products
/cart
default ('')
wildcard (**)

Figure 9.6: Route path resolve process

We have already learned how to navigate our application using the `routerLink` directive. It is the preferred way when using anchor elements in a component template. However, in a real-world application, we also use buttons for navigation. In the following section, we will learn how to navigate to a route path **imperatively** using a `<button>` element.

Navigating imperatively to a route

When we navigate to a wildcard route, the template of the `component` property is displayed on the screen. However, as we have seen, the address bar of the browser stays on the invalid URL. So, we need to provide a way for the user to escape from this route:

1. Open the `page-not-found.component.html` file and add a `<button>` HTML element:

```
<h3>Ooops!</h3>
  <p>The requested page was not found</p>
  <button (click)="goHome()">Home</button>
```

In the preceding HTML template, we have added an event binding to the `click` event of the `<button>` element that points to the `goHome` component method, which does not exist yet.

2. Open the `page-not-found.component.ts` file and add the following `import` statement:

```
import { Router } from '@angular/router';
```

3. Inject the `Router` service in the `constructor` of the `PageNotFoundComponent` class:

```
constructor(private router: Router) { }
```

4. Add the following `goHome` method:

```
goHome() {
    this.router.navigate(['/']);
  }
```

In the preceding method, we call the `navigate` method of the `Router` service to navigate into the root path of the application. It accepts a **link parameters array** containing the destination route path we want to navigate.

It is worth noting that the link parameters array syntax can also be used with the `routerLink` directive. For example, we could have written the `app.component.html` file as follows:

```
<a [routerLink]="['/products']">Products</a>
  <a [routerLink]="['/cart']">Cart</a>
  <router-outlet></router-outlet>
```

It is perfectly fine to use imperative navigation with an anchor element and a `routerLink` directive with a `<button>` element. However, using them, as suggested in this book, is more semantically correct in HTML. The `routerLink` directive modifies the behavior of the target element and adds an `href` attribute, which targets only anchor elements.

Until now, we have relied on the address bar of the browser to indicate which route path is active at any given time. We could improve the user experience by using CSS styling, as we will learn in the

following section.

Decorating router links with styling

The module of the Angular router exports the `routerLinkActive` directive, which we can use to change the style of a route when it is active. It works similarly to the class binding we learned about in *Chapter 4*, *Enabling User Experience with Components*. It accepts a list of class names or a single class that is added when the link is active and removed when it becomes inactive.

To use it in our Angular application, we need first to create a class for active links in the `styles.css` file:

```
.active {
    color: black;
  }
```

We can then add the `routerLinkActive` directive to both links in the `app.component.html` file and set it with the `active` class name:

```
<a routerLink="/products" routerLinkActive="active">Products</a>
  <a routerLink="/cart" routerLinkActive="active">Cart</a>
  <router-outlet></router-outlet>
```

Now, when we click on a link in our application, its color turns to `black` to denote that the link is active.

We have already learned that we can navigate to a route with a static path value. In the next section, we will learn how to do this when the path changes dynamically using route parameters.

Passing parameters to routes

A common scenario in enterprise web applications is to have a list of items and when you click on one of them, the page changes the current view and displays details of the selected item. The previous approach resembles a master-detail browsing functionality, where each generated URL on the master

page contains the identifiers required to load each item on the detail page.

We can represent the previous scenario with two routes navigating to different components. One component is the list of items, and the other is the details of an item. So, we need to find a way to create and pass dynamic item-specific data from one route to the other.

We are tackling double trouble here: creating URLs with dynamic parameters at runtime and parsing the value of these parameters. No problem: the Angular router has our back, and we will see how with a real example.

Building a detail page using route parameters

The product list in our application currently displays a list of products. When we click on a product, the product details appear below the list. We need to refactor the previous workflow so that the component responsible for displaying product details is rendered on a different page from the list. We will use the Angular router to redirect the user to the new page upon clicking on a product from the list.

Currently, the product list component passes the `id` of the selected product via input binding. Input binding cannot be used if the component is activated with routing. We will use the Angular router to

pass the product `id` as a route parameter:

1. Open the `products-routing.module.ts` file and add a new route definition:

```
import { NgModule } from '@angular/core';
  import { RouterModule, Routes } from '@angular/router';
  import { ProductDetailComponent } from './product-detail/product-detail.component';
  import { ProductListComponent } from './product-list/product-list.component';
  const routes: Routes = [
    { path: 'products', component: ProductListComponent },
    { path: 'products/:id', component: ProductDetailComponent },
    { path: '', redirectTo: '/products', pathMatch: 'full' }
  ];
  @NgModule({
    imports: [RouterModule.forRoot(routes)],
    exports: [RouterModule]
  })
  export class ProductsRoutingModule { }
```

The colon character denotes `id` as a route parameter in the new route definition object. If a route has more than one parameter, we separate them with `/`. The parameter name is important when we

want to consume its value in our components, as we will learn later.

2. Open the `product-list.component.html` file and modify its HTML content so that it uses the new route definition:

```
<ul>
    <li *ngFor="let product of products | sort">
      <a [routerLink]="['/products', product.id]">{{product.name}}</a>
    </li>
  </ul>
  <app-product-create (added)="onAdd($event)"></app-product-create>
```

The preceding version of the `product-list.component.html` file is much different from the previous one because we have removed the notion of the selected product. The product list does not need to keep the selected product in its local state because we are navigating away from the list upon selecting a product.

We have added an anchor element inside the `<li>` element and attached the `routerLink` directive to it. The `routerLink` directive uses property binding to set its value in a link parameters array. We

pass the `id` of the `product` template reference variable as a second parameter in the array. The `routerLink` directive requires property binding when dealing with dynamic routes. It will create an `href` attribute that contains the `/products` path, followed by the value of its `id` property.

3. Open the `product-detail.component.ts` file and import the `OnInit`, `ActivatedRoute`, and `switchMap` artifacts:

```
import { Component, OnInit, Input, Output, EventEmitter, OnChanges } from '@angular/core';
  import { ActivatedRoute } from '@angular/router';
  import { Observable, switchMap } from 'rxjs';
```

The Angular router exports the `ActivatedRoute` service, which we can use to retrieve information about the currently active route, including any parameters.

4. Inject the `ActivatedRoute` service into the `constructor` of the `ProductDetailComponent` class:

```
constructor(
    private productService: ProductsService,
```

```
    public authService: AuthService,
    private route: ActivatedRoute
  ) { }
```

5. Add the `OnInit` interface to the list of implemented interfaces of the `ProductDetailComponent` class:

```
export class ProductDetailComponent implements OnInit, OnChanges {
```

6. Create the following `ngOnInit` method:

```
ngOnInit(): void {
    this.product$ = this.route.paramMap.pipe(
      switchMap(params => {
        return this.productService.getProduct(Number(params.get('id')));
      })
    );
  }
```

The `ActivatedRoute` service contains the `paramMap` observable, which we can use to subscribe and get route parameter values. We use the `switchMap` RxJS operator to pipe the `id` parameter from the `paramMap` observable to the `getProduct` method of the `ProductsService` class.

It is also worth noting the following:

- The `paramMap` observable returns an object of the `ParamMap` type. We can use the `get` method of the `ParamMap` object and pass the name of the parameter we defined in the route configuration to access its value.
- We convert the value of the `id` parameter to a number because route parameter values are always strings.

If we run the application using the `ng serve` command and click on a product from the list, the application navigates us to the **Product Details** page:

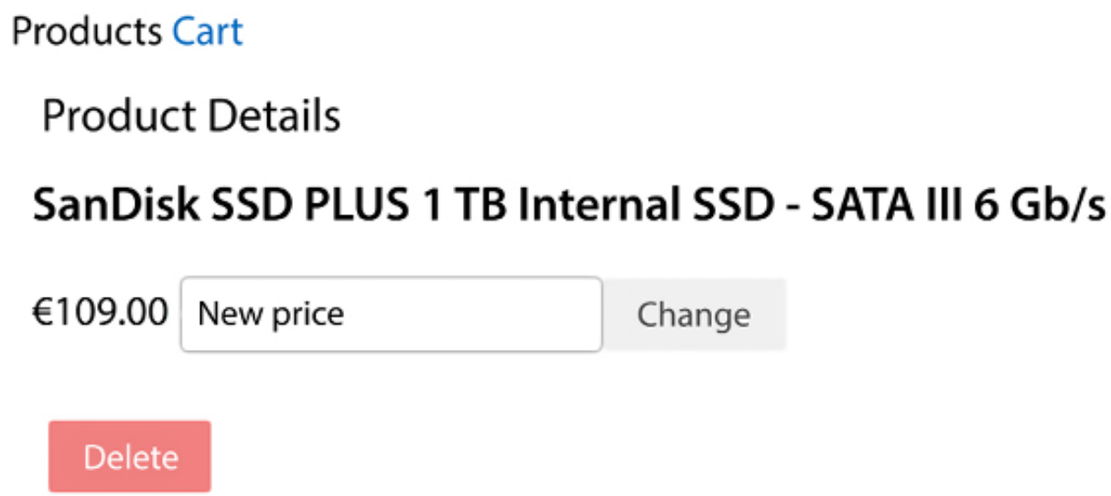

Figure 9.7: Product Details page

In the previous example, we used the `paramMap` property to get route parameters as an observable. So, ideally, our component could be notified of new values during its lifetime. But the component is destroyed each time we want to select a different product from the list, and so is the subscription to the `paramMap` observable. So, what's the point of using it after all?

The router can reuse the instance of a component as soon as it remains rendered on the screen during consecutive navigations. We can achieve this behavior using child routes, as we will learn in the

following section.

Reusing components using child routes

Using child routes is a perfect solution when we want to have a landing page component that will provide routing to other components in a feature module. It should contain a `<router-outlet>` element in which child routes will be loaded. Suppose that we want to define the layout of our Angular application like the following:

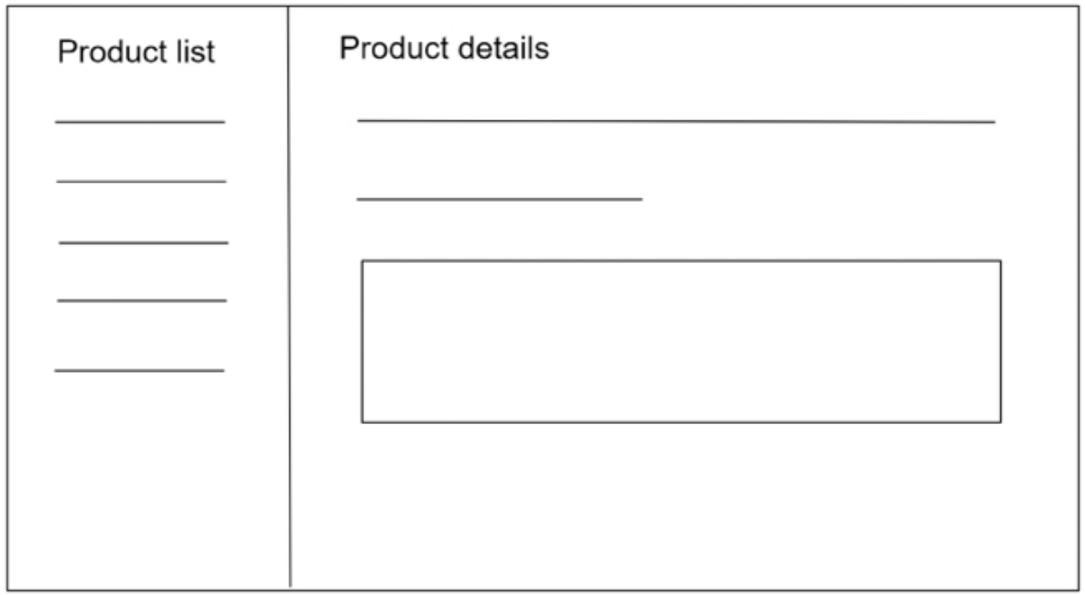

Figure 9.8: Master-detail layout

The scenario described in the previous diagram requires the product list component to contain a `<router-outlet>` element that will render the product details component when the related route is activated.

The product details component is rendered in the `router-outlet` of the product list component and not in the `router-outlet` of the main application component.

The product details component is not destroyed when we navigate from one product to another. Instead, it remains in the DOM tree, and its `ngOnInit` method is called once, the first time we select a product. When we select a new product from the list, the `paramMap` observable emits the `id` of the new product. The new product is fetched using the `ProductsService` class, and the component template is refreshed to reflect the new changes.

The route configuration of the products module, in this case, would be as follows:

```
const routes: Routes = [
    {
      path: 'products',
      component: ProductListComponent,
      children: [
        { path: ':id', component: ProductDetailComponent },
      ]
    },
    { path: '', redirectTo: '/products', pathMatch: 'full' }
```

```
];
```

We use the `children` property of a route definition object to define child routes, which contain a list of route definition objects. Notice also that we removed the word `products` from the `path` property of the product route. We wanted to clarify that it is a child of the `products` route and should be accessed using the `/products/:id` path.

We must also change the `routerLink` directive of the anchor elements in the `product-list.component.html` file so that our application will work correctly:

```
<a [routerLink]="['./', product.id]">{{product.name}}</a>
```

Notice that we replaced `/products` with `./`. What is this strange syntax? It is called **relative navigation** and tells the router to navigate to a specific route relative to the current activated route. It is the opposite of the current syntax we have used so far, **absolute navigation**.

For example, the `./` path indicates to navigate relative to the current level, which is `/products`, in our case. If the route we wanted to navigate was one level above the products route, we would have used `../` as a path. You can think of it as navigating between folders using the command line. The same syntax applies to imperative navigation also:

```
this.router.navigate(['./', product.id], { relativeTo: this.route });
```

In this case, we pass an additional object after the link parameters array that defines the `relativeTo` property pointing to the current activated route.

Relative navigation is considered a better choice over absolute navigation because it is easier to refactor. It decouples hardcoded links by defining paths relative to the current route. Imagine moving a bunch of components around, and suddenly all your hardcoded paths are wrong. Navigation inside a feature module works as expected, even if you decide to change the parent route.

We have learned how to take advantage of the `paramMap` observable in Angular routing. In the following section, we will discuss an alternative approach using route snapshots.

Taking a snapshot of route parameters

Currently, in our application when we select a product from the list, the product list component is removed from the DOM tree, and the product details component is added. To select a different product, we need to click on either the **Products** link or the back button of our browser. Consequently, the product details component is removed from the DOM, and the product list component is added. So, we are in a situation where only one component is displayed on the screen at any time.

When the product details component is destroyed, so is its `ngOnInit` method and the subscription to the `paramMap` observable. So, we do not benefit from using observables at this point. Alternatively, we could use the `snapshot` property of the `ActivatedRoute` service to get values for route parameters as follows:

```
ngOnInit(): void {
    const id = this.route.snapshot.params['id'];
    this.product$ = this.productService.getProduct(id);
  }
```

The `snapshot` property always represents the current value of a route parameter, which also happens to be the initial value. It contains the `params` property, an object of route parameter key-value pairs, which we can access as we would access a plain object in TypeScript.

If you are sure your component will not be reused, you should use the snapshot approach since it is also more readable.

So far, we have dealt with routing parameters in the form `products/:id`. We use these parameters when we want to route to a component that requires the parameter to work correctly. In our case, the product details component requires the `id` parameter to get details of a specific product. However, there is another type of route parameter that is considered optional, as we will learn in the following section.

Filtering data using query parameters

Query parameters are considered optional because they aim to provide optional services such as sorting or filtering data. Some examples are as follows:

- `/products?sortOrder=asc`: Sorts a list of products in ascending order
- `/products?page=3&pageSize=10`: Splits a list of products into pages of 10 records and gets the third

page

Query parameters are recognized in a route by the `?` character. We can combine multiple query parameters by chaining them with an ampersand (`&`) character. The `ActivatedRoute` service contains a `queryParamMap` observable that we can subscribe to in order to get query parameter values. It returns a `ParamMap` object, similar to the `paramMap` observable, which we can query to get parameter values. For example, to retrieve the value of a `sortOrder` query parameter, we would use it as follows:

```
ngOnInit(): void {
    this.route.queryParamMap.subscribe(params => {
      console.log(params.get('sortOrder'));
    });
  }
```

The `queryParamMap` property is also available when working with snapshot routing to get query parameter values.

Now that we have learned how to pass parameters during navigation, we have covered all the essential

information we need to start building Angular applications with routing. In the following sections, we will focus on advanced practices that enhance the user experience when using in-app navigation in our Angular applications.

Enhancing navigation with advanced features

So far, we have covered basic routing, with route parameters as well as query parameters. The Angular router is quite capable, though, and able to do much more, such as the following:

- Controlling access to a route
- Preventing navigation away from a route
- Prefetching data to improve the UX
- Lazy-loading routes to speed up the response time

In the following sections, we will learn about all these techniques in more detail.

Controlling route access

When we want to prevent unauthorized access to a particular route, we use a specific Angular concept called a **guard**. An Angular guard can be of the following types:

- **canActivate**: Controls whether a route can be activated.
- **canActivateChild**: Controls access to child routes of a route.
- **canDeactivate**: Controls whether a route can be deactivated. Deactivation happens when we navigate away from a route.
- **canLoad**: Controls access to a route that loads a lazy-loaded module.
- **canMatch**: Controls access to the same route path based on application conditions.

To create a guard that will allow access to a route depending on whether the user is authenticated or not, we will run the following steps:

1. Create a file named `auth.guard.ts` inside the `src\app\auth` folder.
2. Add the following `import` statements at the top of the file:

```
import { inject } from '@angular/core';
```

```
import { CanActivateFn, Router } from '@angular/router';
import { AuthService } from './auth.service';
```

3. Create an `authGuard` function:

```
export const authGuard: CanActivateFn = () => {
    const authService = inject(AuthService);
    const router = inject(Router);
    if (authService.isLoggedIn) { return true; }
    return router.parseUrl('/');
  };
```

In the preceding function, we use the `inject` method to inject the `AuthService` and `Router` services into the function. The `inject` method behaves the same as if we have injected both services into the constructor of a TypeScript class. We then check the value of the `isLoggedIn` property. If it is `true`, the application can navigate to the specified route. Otherwise, we use the `parseUrl` method of the `Router` service to navigate to the root path of the Angular application. The `parseUrl` method returns a `UrlTree` object, which effectively cancels the previous navigation and redirects the user to the URL passed in the parameter.

4. Open the `app-routing.module.ts` file and add the following `import` statement:

```
import { authGuard } from './auth/auth.guard';
```

5. Add the `authGuard` function in the `canActivate` array of the `cart` route:

```
const routes: Routes = [
    {
      path: 'cart',
      component: CartComponent,
      canActivate: [authGuard]
    },
    { path: '**', component: PageNotFoundComponent }
  ];
```

Only authenticated users can now access the shopping cart. If you run the application using the `ng serve` command and click on the **Cart** link, you will notice that nothing happens.

When you try to access the shopping cart from the product list, you always remain on the same page.

This seems to be because the redirection that occurs due to the authentication guard does not have any effect when you are already in the redirected route. Select a product from the list before clicking the **Cart** link to understand how the guard works.

To be able to access the shopping cart, we need to restore the login functionality from *Chapter 8*, *Communicating with Data Services over HTTP*:

1. Open the `app.module.ts` file and add the following `import` statement:

```
import { AuthModule } from './auth/auth.module';
```

2. Add the `AuthModule` class in the `imports` array of the `@NgModule` decorator:

```
@NgModule({
    declarations: [
      AppComponent,
      CartComponent,
      PageNotFoundComponent
    ],
    imports: [
      BrowserModule,
```

```
    ProductsModule,
    AppRoutingModule,
    HttpClientModule,
    AuthModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
```

3. Open the `app.component.html` file and add the `<app-auth>` component at the top of the file:

```
<app-auth></app-auth>
  <a routerLink="/products" routerLinkActive="active">Products</a>
  <a routerLink="/cart" routerLinkActive="active">Cart</a>
  <router-outlet></router-outlet>
```

If we run the application now, we can use the **Login** button to authenticate with the API and access the cart functionality.

Preventing navigation away from a route

Guards are used not only to prevent access to a route but also to prevent navigation away from it. A guard that controls if a route can be deactivated is a function of the `CanDeactivateFn` type. We will learn how to use it by implementing a guard that notifies the user of pending products in the cart:

1. Create a file named `checkout.guard.ts` inside the `src\app` folder.
2. Add the following `import` statements at the top of the file:

```
import { CanDeactivateFn } from '@angular/router';
  import { CartComponent } from './cart/cart.component';
```

3. Create the following `checkoutGuard` function:

```
export const checkoutGuard: CanDeactivateFn<CartComponent> = () => {
    const confirmation = confirm('You have pending items in your cart. Do you want to cont
    return confirmation;
  };
```

In the preceding function, we set the type of
the `CanDeactivateFn` function to `CartComponent` because we want to check whether the user navigates away from this component only.

In a real-world scenario, you may need to create a generic guard to support additional components.

We then use the global `confirm` method to display a confirmation dialog before navigating away from the cart component.

4. Open the `app-routing.module.ts` file and add the following `import` statement:

```
import { checkoutGuard } from './checkout.guard';
```

5. A route definition object contains a `canDeactivate` array similar to `canActivate`. Add the `checkoutGuard` function to the `canDeactivate` array of the `cart` route:

```
const routes: Routes = [
    {
      path: 'cart',
```

```
    component: CartComponent,
    canActivate: [authGuard],
    canDeactivate: [checkoutGuard]
  },
  { path: '**', component: PageNotFoundComponent }
];
```

For such a simple scenario, we could have written the logic of the `checkoutGuard` function inline and avoided the creation of the `checkout.guard.ts` file:

```
{
    path: 'cart',
    component: CartComponent,
    canActivate: [authGuard],
    canDeactivate: [() => confirm('You have pending items in your cart. Do you want to continue
  }
```

Run the application using the `ng serve` command and click the **Cart** link after you have logged in. If

you then click on the **Products** link or press the back button of the browser, you should see the following dialog:

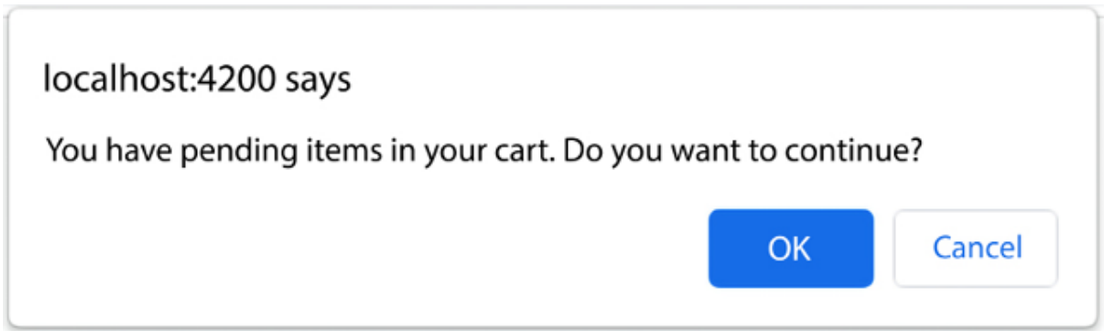

Figure 9.9: Pending items dialog

If you click the **Cancel** button, the navigation is canceled, and the application remains in its current state. If you click the **OK** button, you will be redirected to the root path of the application.

Prefetching route data

You may have noticed that when you select a product from the list and navigate to the product details

component, there is a delay in displaying the product details data. It is reasonable since we are making an HTTP request to the backend API. However, there is flickering in the user interface, which is bad for the user experience. Thankfully, the Angular router can help us to fix that. We can use a **resolver** to prefetch the details of a product so that they are available when activating the route and displaying the component.

A resolver can be handy when we want to handle possible errors before activating a route. In our case, it would be more appropriate not to navigate to the product details component if the `id` we pass as a route parameter does not exist in the backend.

Let's create a route resolver for our product details component:

1. Create a file named `product-detail.resolver.ts` inside the `src\app\products` folder.
2. Add the following `import` statements at the top of the file:

```
import { inject } from '@angular/core';
  import { ActivatedRouteSnapshot, ResolveFn } from '@angular/router';
  import { Product } from './product';
  import { ProductsService } from './products.service';
```

3. Create the following `productDetailResolver` function:

```
export const productDetailResolver: ResolveFn<Product> = (route: ActivatedRouteSnapshot) =
    const productService = inject(ProductsService);
    const id = Number(route.paramMap.get('id'));
    return productService.getProduct(id);
  };
```

A resolver is a function of the `ResolveFn<T>` type, where `T` is the resolved data type. It can return resolved data either synchronously or asynchronously. In our case, since we are communicating with a backend API using the HTTP client, we need to return an observable of a `Product` object.

The `productDetailResolver` function injects the `ProductsService` class, gets the value of the `id` route parameter, and converts it into a number. It then calls the `getProduct` method of the `ProductsService` class, passing the `id` as a parameter.

4. Open the `products-routing.module.ts` file and add the following `import` statement:

```
import { productDetailResolver } from './product-detail.resolver';
```

5. Add a `resolve` property to the route definition object that activates `ProductDetailComponent`:

```
const routes: Routes = [
    { path: 'products', component: ProductListComponent },
    {
      path: 'products/:id',
      component: ProductDetailComponent,
      resolve: {
        product: productDetailResolver
      }
    },
    { path: '', redirectTo: '/products', pathMatch: 'full' }
  ];
```

The `resolve` property is an object that contains a unique name as a key and the TypeScript class of the resolver as a value. The name of the key is important because we will use that in our components to access the resolved data.

6. Open the `product-detail.component.ts` file and import the `of` operator from the `rxjs` npm package:

```
import { Observable, of, switchMap } from 'rxjs';
```

7. Modify the `ngOnInit` method so that it subscribes to the `data` property of the `ActivatedRoute` service:

```
ngOnInit(): void {
    this.product$ = this.route.data.pipe(
      switchMap(data => of(data['product']))
    );
  }
```

In the preceding snippet, the `data` observable emits an object where the value of the requested product exists in the `product` key of the object. Notice that we use the `switchMap` operator to return the product in a new observable.

If you run the application now, you will notice no flickering when navigating to the product details

component, and data are displayed at once. However, you may notice a slight delay upon selecting the product from the list. It is the delay introduced by the HTTP request to the backend API that originates from the resolver.

Lazy-loading routes

At some point, our application may grow in size, and the amount of data we put into it may also grow. The result is that the application may take a long time to start initially, or certain parts can take a long time to start. To overcome these problems, we can use a technique called **lazy loading**.

Lazy loading means that we don't load all parts of our application initially. When we refer to parts, we mean Angular modules. Application modules can be separated into chunks that are only loaded when needed. There are many advantages of lazy loading a module in an Angular application:

- Feature modules can be loaded upon request from the user.
- Users that visit certain areas of your application can significantly benefit from this technique.
- We can add more features in a lazy-loaded module without affecting the overall application bundle size.

To understand how lazy loading in Angular works, we will create a new module with a component that displays information about our e-shop application:

1. Run the following command to create an Angular module with routing enabled:

```
ng generate module about --routing
```

2. Create a component named `about-info` inside the `src\app\about` folder:

```
ng generate component about-info
```

3. Open the `about-routing.module.ts` file and add the following `import` statement:

```
import { AboutInfoComponent } from './about-info/about-info.component';
```

4. Add a new route definition object in the `routes` variable to activate `AboutInfoComponent`:

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

```
```

In the preceding snippet, we set the `path` property to an empty string so that the route is activated by default.

5. Add a new anchor element to the `app.component.html` file that links to the newly created route:

```
<app-auth></app-auth>
  <a routerLink="/products" routerLinkActive="active">Products</a>
  <a routerLink="/cart" routerLinkActive="active">Cart</a>
  <a routerLink="/about" routerLinkActive="active">About Us</a>
  <router-outlet></router-outlet>
```

6. Finally, open the `app-routing.module.ts` file and add a new route definition object:

```
const routes: Routes = [
    {
      path: 'cart',
      component: CartComponent,
      canActivate: [authGuard],
      canDeactivate: [checkoutGuard]
```

```
  },
  {
    path: 'about',
    loadChildren: () => import('./about/about.module').then(m => m.AboutModule)
  },
  { path: '**', component: PageNotFoundComponent }
];
```

The `loadChildren` property of a route definition object is used to lazy load Angular modules. It returns an arrow function that uses a dynamic import statement to lazy load the module.
The `import` function accepts the relative path of the module we want to import, returning a promise object that contains the TypeScript class of the Angular module we want to load.

We did not import `AboutModule` in the main application module. Otherwise, it would have been loaded twice, eagerly from the main application module and lazily from the `about` route.

Run the application using the `ng serve` command and open the browser's developer tools. Click the **About Us** link, and inspect the requests in the **Network** tab:

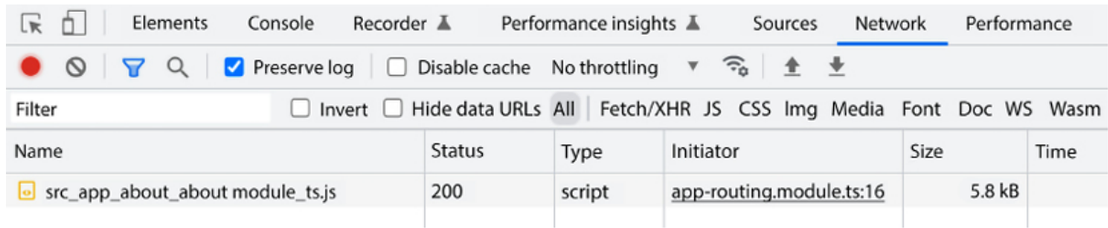

Figure 9.10: Lazy-loaded module

The application initiates a new request to the `src_app_about_about_module_ts.js` file, which is the bundle of the `about` module. The Angular framework creates a new bundle for each module that is lazy-loaded and does not include it in the main application bundle.

If you navigate away and click on the **About Us** link again, you will notice that the application does not make a new request to load the module. As soon as a lazy-loaded module is requested, it is kept in memory and can be used for subsequent requests.

A word of caution, however. As we learned in *Chapter 6*, *Managing Complex Tasks with Services*, an Angular service is registered with the root injector of the application using the `providedIn` property of the `@Injectable` decorator.

Lazy-loaded modules create a separate injector that is an immediate child of the root application injector. If you use an Angular service registered with the root application injector in a lazy-loaded module, you will end up with a separate service instance in both cases. So, we must be cautious as to how we use services in lazy-loaded modules.

Lazy-loaded modules are standard Angular modules, so we can control their access using guards.

Protecting a lazy-loaded module

We can control unauthorized access to a lazy-loaded module similar to how we can do so in eagerly loaded ones. However, our guards need to implement a different interface for this case, the `CanLoad` interface.

We will extend our authentication guard for use with lazy-loaded modules.

1. Open the `auth.guard.ts` file and import `CanLoadFn` from the `@angular/router` npm package:

```
import { CanActivateFn, CanLoadFn, Router } from '@angular/router';
```

2. Add the `CanLoadFn` type to the `authGuard` function:

```
export const authGuard: CanActivateFn | CanLoadFn = () => {
    const authService = inject(AuthService);
    const router = inject(Router);
    if (authService.isLoggedIn) { return true; }
    return router.parseUrl('/');
  };
```

3. As with all previous guards, we must register the `authGuard` function with the lazy-loaded route using the `canLoad` array of the route definition object. Open the `app-routing.module.ts` file and add the `authGuard` function in the `canLoad` array of the `about` route:

```
{
    path: 'about',
    loadChildren: () => import('./about/about.module').then(m => m.AboutModule),
    canLoad: [authGuard]
  }
```

If we now run the application and click on the **About Us** link, we will notice that we cannot navigate to

the **About** page unless we are authenticated.

We have already learned about standalone components in *Chapter 4*, *Enabling User Experience with Components*. Standalone components are not registered with an Angular module but can be lazy loaded, as we will see in the following section.

Lazy loading components

We have already learned how to create standalone components, pipes, and directives in Angular applications. Before the standalone option, it was cumbersome and complicated to load a component dynamically. It required many lines of code and advanced Angular techniques from the developer's perspective. However, with the introduction of standalone APIs from Angular, developers now have a powerful tool at their disposal to satisfy even the most complex business needs in their Angular applications.

The Angular router can lazy load not only Angular modules but also standalone components. The route definition object contains a `loadComponent` property, similar to `loadChildren` for modules, allowing us to pass an Angular component for lazy loading. We will learn more by converting our `about` component into a standalone one and lazy loading it:

1. Open the `about-info.component.ts` file and add the `standalone` property in the `@Component` decorator:

```
import { Component } from '@angular/core';
  @Component({
    selector: 'app-about-info',
    templateUrl: './about-info.component.html',
    styleUrls: ['./about-info.component.css'],
    standalone: true
  })
  export class AboutInfoComponent {}
```

2. Open the `about.module.ts` file and remove any occurrences of the `AboutInfoComponent` class:

```
import { NgModule } from '@angular/core';
  import { CommonModule } from '@angular/common';
  import { AboutRoutingModule } from './about-routing.module';
  @NgModule({
    imports: [
      CommonModule,
      AboutRoutingModule
    ]
```

```
})
export class AboutModule { }
```

We need to remove it specifically from the `declarations` array of the `@NgModule` decorator. Otherwise, it will not be a standalone component.

3. Open the `app-routing.module.ts` file and modify the `about` route so that it uses the `loadComponent` property to lazily load `AboutInfoComponent`:

```
{
    path: 'about',
    loadComponent: () => import('./about/about-info/about-info.component').then(c => c.Abo
  }
```

Run the application using the `ng serve` command and click the **About Us** link. The component is still loaded on the page, but a different chunk appears in the **Network** tab of the browser console:

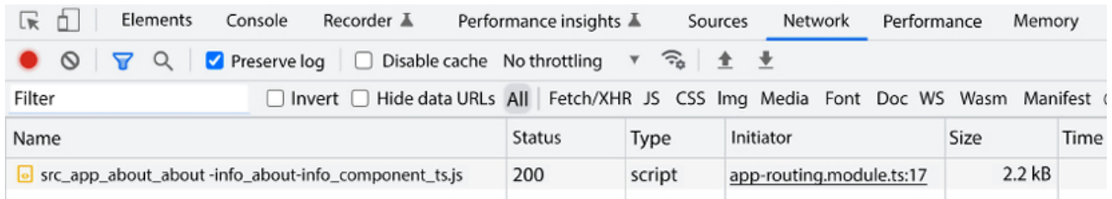

Figure 9.11: Lazy-loaded component

The application initiates a new request to the `src_app_about_about-info_about-info_component_ts.js` file this time. Everything related to lazy loading we saw in the module example also applies here.

Summary

We have now uncovered the power of the Angular router, and we hope you have enjoyed this journey into the intricacies of this library. One of the things that shines in the Angular router is the vast number of options and scenarios we can cover with such a simple but powerful implementation.

We have learned the basics of setting up routing and handling different types of parameters. We have also learned about more advanced features, such as child routing. Furthermore, we have learned how to protect our routes from unauthorized access. Finally, we have shown the full power of routing and how you can improve response time with lazy loading and prefetching.

In the next chapter, we will beef up our application components to showcase the mechanisms underlying web forms in Angular and the best strategies to grab user input, with form controls.

End of Chapter 9