

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ
СІКОРСЬКОГО»**

Факультет інформатики та обчислювальної техніки
Кафедра інформатики та програмної інженерії

Звіт по лабораторній роботі № 5
Створення проекту «SportShop». Розробка
основних структурних блоків та моделі даних. Вибір товарів та
оформлення замовлень.
з дисципліни: «Реактивне програмування»

Студент: Головня Олександр Ростиславович
Група: ІІІ-11
Дата захисту роботи: _____
Викладач: доц. Полупан Юлія Вікторівна
Захищено з оцінкою: _____

Київ, 2024

Зміст

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»	1
I) Частина 1: Створення проекту «SportShop»	5
Підготовка проекту	5
Створення структури папок	5
Встановлення додаткових пакетів NPM	5
Додавання CSS стилів до проекту	5
Підготовка REST – сумісної веб-служби	6
Підготовка файлу HTML	8
Запуск прикладу	8
Запуск REST-сумісної веб-служби	8
Підготовка проекту Angular	9
Оновлення кореневого компонента	9
Перевірка кореневого модуля	10
Аналіз файлу початкового завантаження	11
Початок роботи над моделлю даних	11
Створення класів моделі	11
Створення фіктивного джерела даних	12
Створення репозиторію моделі	13
Створення функціонального модуля	14
Створення сховища	15
Створення компоненту магазину та шаблону	15
Створення функціонального модуля сховища	17
Оновлення кореневого компонента та кореневого модуля	18
Додавання функціональності: докладна інформація про товари. Виведення докладної інформації про товари	19
Додавання вибору категорій	21
Посторінне виведення списку товарів	23
Створення нестандартної директиви	26
II) Частина 2: Вибір товарів та оформлення замовлень. Підготовка програми	29
Створення кошика	29
Створення моделі кошика	29
Створення компонентів для зведеної інформації кошика	31
Інтеграція кошика у додаток	33
Маршрутизація URL	36
Створення та застосування конфігурації маршрутизації	38
Навігація у додатку	39
Захисники маршрутів	41
Завершення виведення вмісту кошика	44

Обробка замовлень	46
Розширення моделі	47
Оновлення репозиторію та джерела даних	48
Оновлення функціонального модуля	49
Отримання інформації про замовлення	49
Використання REST-сумісної веб-служби	53
Застосування джерела даних	55
Посилання на додатки:	57
Висновок:	57
Список використаних джерел:	Error! Bookmark not defined.

Мета: Мета: Навчитися створювати Angular-додатки, які містять сервіси, pipes, директиви та використовувати бібліотеку RxJS.

Завдання:

Створити два Angular-додатки під назвою Service1 та Service2.

Частина 1:

Створити проект «SportShop». Встановити та налаштувати засоби розробника, створити кореневі структурні блоки для проекту (модель даних, фіктивне джерело даних, репозиторій моделі, сховище, компоненти магазину та шаблони).

Вивести дані фіктивної моделі даних на головну сторінку магазину, реалізувати розбивку на сторінки та фільтрацію товарів за категоріями. Створити нестандартну директиву для пагінації. Додаток, отриманий в результаті виконання

Частина 2:

Розробити додаткову логіку в додатку «SportShop» для вибору товарів та оформлення замовлень. Реалізувати підтримку кошика для вибору товарів користувачем та процесу оформлення замовлення. Фіктивне джерело даних у проекті замінити джерелом, що надсилає запити до HTTP REST-сумісної веб-служби. Роботу додатку «SportShop», отриманого в результаті виконання Частини 1 та Частини 2, продемонструвати в режимі відеоконференції.

Зробити звіт по роботі. Звіт повинен включати: титульний лист, зміст, основну частину, список використаних джерел.

Angular-додаток SportShop, отриманий в результаті виконання Частини 1 завдання, розгорнути на платформі Firebase у проекті з ім'ям «ПрізвищеГрупаLaba5-1», наприклад «KovalenkoIP01Laba5-1».

I) Частина 1: Створення проекту «SportShop»

Підготовка проекту

Спочатку я підготував новий проєкт командою:

```
ng new SportShop --routing false --style css --skip-git --skip-tests
```

Створення структури папок

Структура папок:

Папка	Опис
SportShop/src/app/model	Папка для коду моделі даних
SportShop/src/app/shop	Папка для базової функціональності здійснення покупок
SportShop/src/app/admin	Папка для функціональності адміністрування

Встановлення додаткових пакетів NPM

Додаткові пакети потрібні для проекту SportShop на додаток до основних пакетів Angular. Виконав наведені нижче команди, щоб додати необхідні пакети:

```
npm install bootstrap@5.1.3
```

```
npm install @fortawesome/fontawesome-free@6.0.0
```

```
npm install --save-dev json-server@0.17.0
```

```
npm install --save-dev jsonwebtoken@8.5.1
```

Додавання CSS стилів до проекту

Додав стилі до проєкту:

```
"styles": [
  "src/styles.css",
  "node_modules/@fortawesome/fontawesome-free/css/all.min.css",
  "node_modules/bootstrap/dist/css/bootstrap.min.css"
],
```

```
PS C:\Users\Саша Головня\Desktop\Reactive programming\HolovniaIP11Lab5\sportshop>
ng config projects.SportShop.architect.build.options.styles
[
  "src/styles.css",
  "node_modules/@fortawesome/fontawesome-free/css/all.min.css",
  "node_modules/bootstrap/dist/css/bootstrap.min.css"
]
```

Підготовка REST – сумісної веб-служби

Програма SportShop використовуватиме асинхронні HTTP-запити, щоб отримати дані моделі, надані REST-сумісною веб-службою. До проекту було додано пакет jsonserver. Це чудовий пакет для створення веб-сервісів із даних JSON або коду JavaScript.

Додав оператор, показаний на рисунку, до розділу сценаріїв у файлі package.json, щоб пакет json-server можна було запустити з командного рядка.

```
"scripts": {  
  "ng": "ng",  
  "start": "ng serve",  
  "build": "ng build",  
  "watch": "ng build --watch --configuration development",  
  "test": "ng test",  
  "json": "json-server data.js -p 3500 -m authMiddleware.js",  
}
```

Щоб забезпечити пакет json-server даними для роботи, додав файл під назвою data.js у папку SportShop і додав код, показаний нижче, який забезпечить доступність тих самих даних кожного разу, коли запускається пакет json-server.

```
HoloVnialP11Lab5 > SportShop > JS data.js > <unknown> > exports > products  
1 module.exports = function () {  
2   return {  
3     products: [  
4       {  
5         id: 1, name: "Rubber boat", category: "Watersports",  
6         description: "A boat for two persons", price: 285  
7       },  
8       {  
9         id: 2, name: "Fishing rod", category: "Watersports",  
10        description: "For winter fishing", price: 55  
11      },  
12      {  
13        id: 3, name: "Soccer Ball", category: "Soccer",  
14        description: "FIFA-approved size and weight", price: 19.50  
15      },  
16      {  
17        id: 4, name: "Corner Flags", category: "Soccer",  
18        description: "Give your playing field a professional touch",  
19        price: 34.95  
20      },  
21      {  
22        id: 5, name: "Stadium", category: "Soccer",  
23        description: "Flat-packed 35,000-seat stadium", price: 79500  
24      },  
25      {  
26        id: 6, name: "Thinking Cap", category: "Chess",  
27        description: "Improve brain efficiency by 75%", price: 16  
28      },  
29      {  
30        id: 7, name: "Unsteady Chair", category: "Chess",  
31        description: "Secretly give your opponent a disadvantage",  
32        price: 29.95  
33      },  
34      {  
35        id: 8, name: "Human Chess Board", category: "Chess",  
36        description: "A fun game for the family", price: 75  
37      },  
38      {  
39        id: 9, name: "Bling King", category: "Chess",  
40        description: "Gold-plated, diamond-studded King", price: 1200  
41      }  
42    ],  
43  }  
44 }
```

Дані, що зберігаються REST-сумісною веб-службою, необхідно захистити, щоб звичайні користувачі не могли змінювати опис товарів або стан своїх замовлень. Пакет `json-server` не містить вбудованих засобів аутентифікації, тому було створено файл з ім'ям `authMiddleware.js` в папці `SportShop` з наступним кодом:

```
HolovniaIP11Lab5 > SportShop > JS authMiddleware.js > ...
1  const jwt = require("jsonwebtoken");
2  const APP_SECRET = "myappsecret";
3  const USERNAME = "admin";
4  const PASSWORD = "secret";
5  const mappings = {
6    get: ["/api/orders", "/orders"],
7    post: ["/api/products", "/products", "/api/categories", "/categories"]
8  }
9  function requiresAuth(method, url) {
10     return (mappings[method.toLowerCase()] || []).
11         .find(p => url.startsWith(p)) !== undefined;
12 }
13 module.exports = function (req, res, next) {
14     if (req.url.endsWith("/login") && req.method == "POST") {
15         if (req.body && req.body.name == USERNAME &&
16             req.body.password == PASSWORD) {
17             let token = jwt.sign({ data: USERNAME, expiresIn: "1h" }, APP_SECRET);
18             res.json({ success: true, token: token });
19         } else {
20             res.json({ success: false });
21         }
22         res.end();
23         return;
24     } else if (requiresAuth(req.method, req.url)) {
25         let token = req.headers["authorization"] || "";
26         if (token.startsWith("Bearer<")) {
27             token = token.substring(7, token.length - 1);
28             try {
29                 jwt.verify(token, APP_SECRET);
30                 next();
31                 return;
32             } catch (err) { }
33         }
34         res.statusCode = 401;
35         res.end();
36         return;
37     }
38     next();
39 }
```

Цей код перевіряє запити HTTP, надіслані REST-сумісній веб-службі, та реалізує найпростіші засоби безпеки. Це серверний код, що не пов'язаний безпосередньо з розробкою додатків Angular.

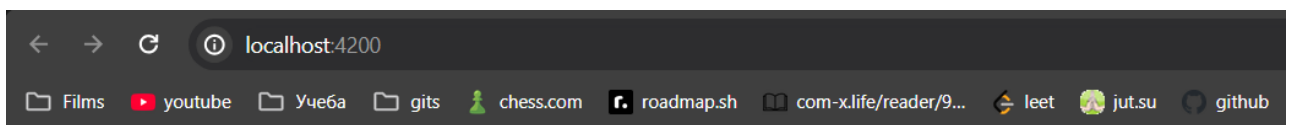
Підготовка файлу HTML

Кожна веб-програма Angular має файл HTML, який завантажується браузером і виконує завантаження і запуск програми. Відредагував файл `index.html` в папці `SportShop/src` та додав до нього елементи нижче.

```
HolovniatP11Lab5 > SportShop > src > < index.html > html > body.p-2
1  <!doctype html>
2  <html lang="en">
3  <head>
4    <meta charset="utf-8">
5    <title>SportShop</title>
6    <base href="/">
7    <meta name="viewport" content="width=device-width, initial-scale=
8    <link rel="icon" type="image/x-icon" href="favicon.ico">
9  </head>
10 <body class="p-2">
11   <app>SportShop is run</app>
12 </body>
13 </html>
14
```

Документ HTML містить елемент `link` для завантаження таблиці стилів Bootstrap та елемент `app` що резервує місце для функціональності SportShop.

Запуск прикладу



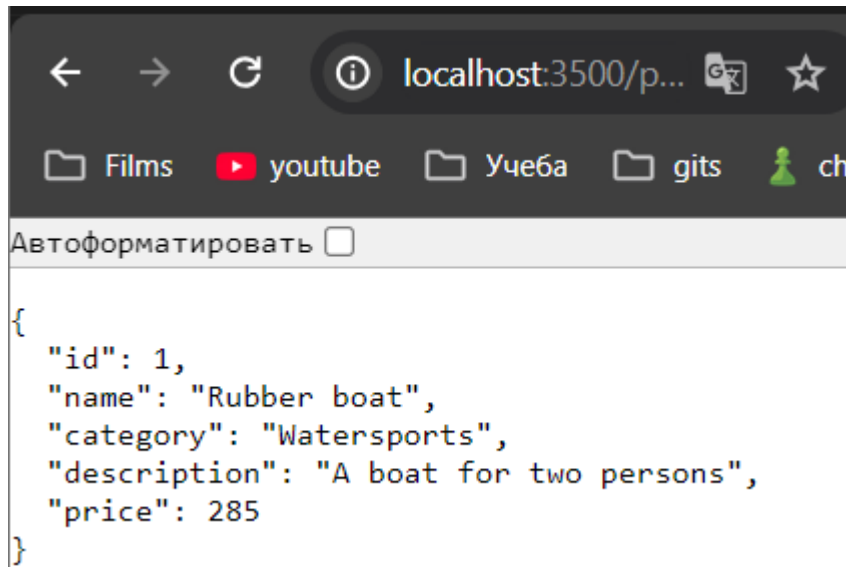
Hello, SportShop

Congratulations! Your app is running. 🎉

Запуск REST-сумісної веб-служби

Команда: `npm run json`

Бачимо результат:



Підготовка проекту Angular

У наступних розділах ми закладемо основу, на якій будуватиметься програма SportShop.

Оновлення кореневого компонента

Почнемо з кореневого компонента – структурного блоку Angular, який керуватиме елементом app у документі HTML. Додаток може містити кілька компонентів, але серед них завжди є кореневий компонент, що відповідає за відображення контенту верхнього рівня. Відредагував файл `app.component.ts` в папці `SportShop/src/app` та включив в нього код нижче:

```
HolovniaIP11Lab5 > SportShop > src > app > TS app.component.ts > AppComponent
1  import { Component } from "@angular/core";
2  @Component({
3    selector: "app",
4    template: `<div class="bg-success p-2 text-center text-white">
5    This is SportShop
6    </div>`
7  })
8  export class AppComponent { }
```

Перевірка кореневого модуля

Модулі Angular поділяються на дві категорії: функціональні модулі та кореневий модуль.

Функціональні модулі використовуються для угруповання взаємопов'язаної функціональності програми, щоб спростити керування програмою. Ми створимо функціональні модулі для всіх основних функціональних областей програми, включаючи модель даних, інтерфейс магазину користувача і інтерфейс адміністрування. Кореневий модуль передає опис програми для Angular. В описі зазначено, які функціональні модулі необхідні для запуску додатку, які нестандартні можливості слід завантажити та як називається кореневий компонент.

Традиційно файлу кореневого компонента надається ім'я `app.module.ts`. Перевірів файл із таким ім'ям у папці `SportShop/src/app`. Цей файл повинен включати код нижче

```
HolovniaIP11Lab5 > SportShop > src > app > TS app.module.ts > ...
1  import { NgModule } from "@angular/core";
2  import { BrowserModule } from "@angular/platform-browser";
3  import { AppComponent } from "../app.component";
4  @NgModule({
5      imports: [BrowserModule],
6      declarations: [AppComponent],
7      providers: [],
8      bootstrap: [AppComponent]
9  })
10 export class AppModule { }
11
```

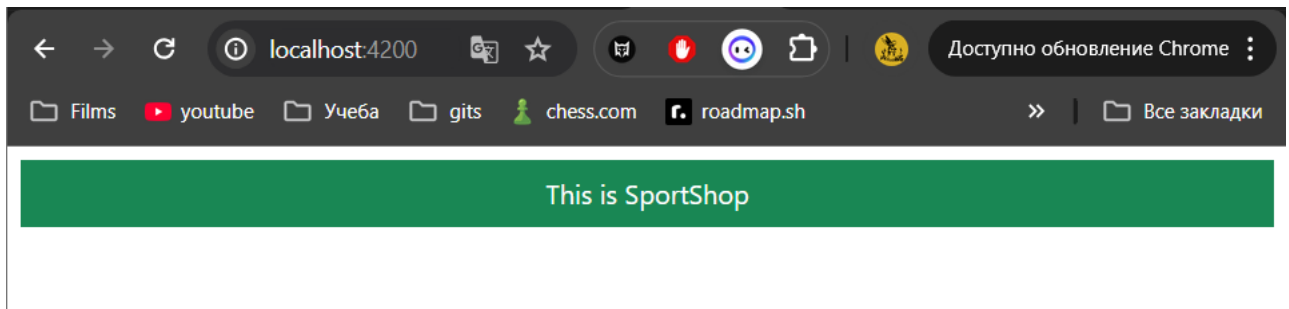
За аналогією з корневим компонентом клас кореневого модуля не містить код. Річ у тім, що кореневий модуль існує лише для передачі інформації через декоратор `@NgModule`. Властивість `imports` наказує Angular завантажити функціональний модуль `BrowserModule` з усією основною функціональністю Angular, необхідною для вебпрограми. Властивість `declarations` наказує Angular завантажити кореневий компонент, а властивість `bootstrap` повідомляє, що корневим компонентом є клас `AppModule`. Надалі буде додана інформація до властивостей цього декоратора.

Аналіз файлу початкового завантаження

Наступний блок службового коду – файл початкового завантаження, який запускає програму. Файл початкового завантаження використовує браузерну платформу Angular для завантаження кореневого модуля та запуску програми. Створив файл з ім'ям `main.ts` в папці `SportShop/src/app` і додав код нижче:

```
HolovniatP11Lab5 > SportShop > src > app > TS main.ts
1  import { enableProdMode } from '@angular/core';
2  import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
3
4  import { AppModule } from './app.module';
5  import { environment } from './environments/environment';
6
7  if (environment.production) {
8    enableProdMode();
9  }
10
11 platformBrowserDynamic().bootstrapModule(AppModule);
```

Результат:



Початок роботи над моделлю даних

Створення класів моделі

Кожній моделі даних необхідні класи для опису типів даних, що входять до моделі даних. У додатку SportShop це класи з описом товарів, що продаються в інтернетмагазині, та замовлення, отримані від користувачів. Для початку роботи програми SportShop достатньо можливості опису товарів; інші класи моделей будуть створюватися для підтримки розширеної функціональності в міру їхньої реалізації. Створив файл з ім'ям `product.model.ts` в папці `SportShop/src/app/model` та включив код нижче:

```
HolovniatP11Lab5 > SportShop > src > app > model > TS product.model.ts > ...
1  export class Product {
2      constructor(
3          public id?: number,
4          public name?: string,
5          public category?: string,
6          public description?: string,
7          public price?: number) { }
8  }
9
```

Клас Product визначає конструктор, який отримує властивості id, name, category, description і price. Ці властивості відповідають структурі даних, які використовуються для заповнення REST-сумісної веб-служби у лістингу 8.5. Знаки запитання (?) за іменами параметрів вказують, що це необов'язкові параметри, які можуть бути опущені під час створення нових об'єктів з використанням класу Product; це може бути зручно при розробці програм.

Створення фіктивного джерела даних

Щоб підготувати перехід від фіктивних даних до реальних, ми будемо передавати дані з джерела даних. Решта коду програми не знає, звідки надійшли дані, і перехід на отримання даних із запитів HTTP пройде прозоро. Створив файл static.datasource.ts в папці SportShop/src/app/model та включив визначення класу нижче:

```
HolovniatP11Lab5 > SportShop > src > app > model > TS static.datasource.ts > ...
1  import { Injectable } from "@angular/core";
2  import { Product } from "../product.model";
3  import { Observable, from } from "rxjs";
4  @Injectable()
5  export class StaticDataSource {
6      private products: Product[] = [
7          new Product(1, "Product 1", "Category 1", "Product 1 (Categ
8          new Product(2, "Product 2", "Category 1", "Product 2 (Categ
9          new Product(3, "Product 3", "Category 1", "Product 3 (Categ
10         new Product(4, "Product 4", "Category 1", "Product 4 (Categ
11         new Product(5, "Product 5", "Category 1", "Product 5 (Categ
12         new Product(6, "Product 6", "Category 2", "Product 6 (Categ
13         new Product(7, "Product 7", "Category 2", "Product 7 (Categ
14         new Product(8, "Product 8", "Category 2", "Product 8 (Categ
15         new Product(9, "Product 9", "Category 2", "Product 9 (Categ
16         new Product(10, "Product 10", "Category 2", "Product 10 (Ca
17         new Product(11, "Product 11", "Category 3", "Product 11 (Ca
18         new Product(12, "Product 12", "Category 3", "Product 12 (Ca
19         new Product(13, "Product 13", "Category 3", "Product 13 (Ca
20         new Product(14, "Product 14", "Category 3", "Product 14 (Ca
21         new Product(15, "Product 15", "Category 3", "Product 15 (Ca
22     ];
23     getProducts(): Observable<Product[]> {
24         return from([this.products]);
25     }
26 }
```

Клас `StaticDataSource` визначає метод з ім'ям `getProducts`, що повертає фіктивні дані. Виклик методу `getProducts` повертає результат `Observable` - реалізацію `Observable` для отримання масивів об'єктів `Product`. Клас `Observable` надається пакетом `Reactive Extensions`, який використовується `Angular` для обробки змін стану проекту.

Об'єкт `Observable` схожий на об'єкт `JavaScriptPromise`: він представляє асинхронне завдання, яке в майбутньому має повернути результат. `Angular` розкриває використання об'єктів `Observable` для деяких своїх функцій, включаючи роботу із запитамі `HTTP`; саме тому метод `getProducts` повертає `Observable` замість повернення даних - простого синхронного або з використанням `Promise`. Декоратор `@Injectable` застосовується до класу `StaticDataSource`. Цей декоратор повідомляє `Angular`, що цей клас буде використовуватися як служба, що дозволяє іншим класам звертатися до його функціональності через механізм впровадження залежностей.

Створення репозиторію моделі

Джерело даних має надати додатку запитувані дані, але звернення до даних зазвичай відбувається через посередника (репозиторій), що відповідає за передачу цих даних окремим структурним блокам програми, щоб подробиці отримання даних залишалися прихованими.

Створив файл `product.repository.ts` в папці `SportShop/src/app/model` та визначив в ньому клас нижче:

```

HolovniaIP11Lab5 > SportShop > src > app > model > TS product.repository.ts > ...
1  import { Injectable } from "@angular/core";
2  import { Product } from "../product.model";
3  import { StaticDataSource } from "../static.datasource";
4  @Injectable()
5  export class ProductRepository {
6      private products: Product[] = [];
7      private categories: string[] = [];
8      constructor(private dataSource: StaticDataSource) {
9          dataSource.getProducts().subscribe(data => {
10              this.products = data;
11              this.categories = data.map(p => p.category ?? "(None)")
12                  .filter((c, index, array) => array.indexOf(c) == index).sort();
13          });
14      }
15      getProducts(category?: string): Product[] {
16          return this.products
17              .filter(p => category == undefined || category == p.category);
18      }
19      getProduct(id: number): Product | undefined {
20          return this.products.find(p => p.id == id);
21      }
22      getCategories(): string[] {
23          return this.categories;
24      }
25  }

```

Коли Angular буде потрібно створити новий екземпляр репозиторію, Angular аналізує клас і бачить, що для виклику конструктора ProductRepository та створення нового об'єкта йому потрібен об'єкт StaticDataSource. Конструктор репозиторію викликає метод getProducts джерела даних, після чого використовує метод subscribe об'єкта Observable, що повертається для отримання цих товарів.

Створення функціонального модуля

Зараз визначимо функціональну модель Angular для моделі даних, яка дозволить легко використовувати функціональність моделі даних у будь-якій точці програми. Створив файл з ім'ям model.module.ts в папці SportShop/app/model та визначив клас, наведений нижче:

```
HolovniatP11Lab5 > SportShop > src > app > model > TS model.module.ts > Mod
1  import { NgModule } from "@angular/core";
2  import { ProductRepository } from "../product.repository";
3  import { StaticDataSource } from "../static.datasource";
4  @NgModule({
5    providers: [ProductRepository, StaticDataSource]
6  })
7  export class ModelModule { }
```

Декоратор `@NgModule` використовується для створення функціональних модулів, а його властивості повідомляють Angular про те, як повинен використовуватися модуль. В даному випадку модуль містить лише одну властивість `providers`, яка повідомляє, які класи повинні використовуватися як служби для механізму впровадження залежностей.

Створення сховища

Модель даних готова, і ми можемо переходити до побудови функціональності магазину: перегляд списку товарів та оформлення замовлень. У магазині використовуватиметься двостовпцевий макет, список товарів фільтруватиметься за допомогою кнопок категорій, а самі товари виводитимуться в таблиці.

Створення компоненту магазину та шаблону

Відправною точкою для функціональності магазину стане новий компонент - клас, що надає дані та логіку для шаблону HTML, що містить прив'язки даних динамічного генерування контенту. Створив файл з ім'ям `shop.component.ts` в папці `SportShop/src/app/shop` та додав визначення класу нижче:

```

HolovniaIP11Lab5 > SportShop > src > app > shop > TS shop.component.ts > ShopComponent
1  import { Component } from "@angular/core";
2  import { Product } from "../model/product.model";
3  import { ProductRepository } from "../model/product.repository";
4  @Component({
5      selector: "shop",
6      templateUrl: "shop.component.html"
7  })
8  export class ShopComponent {
9      constructor(private repository: ProductRepository) { }
10     get products(): Product[] {
11         return this.repository.getProducts();
12     }
13     get categories(): string[] {
14         return this.repository.getCategories();
15     }
16 }

```

До класу ShopComponent застосовується декоратор @Component, який повідомляє Angular, що клас є компонентом. Властивості декоратора вказують Angular, як застосовувати компонент до HTML контенту (з використанням елемента з ім'ям shop) і де знаходиться шаблон компонента (у файлі з ім'ям shop.component.html). Клас ShopComponent надає логіку, яка забезпечує отримання контенту шаблоном. Конструктор класу отримує об'єкт ProductRepository в аргументі, що передається через механізм застосування залежностей. Компонент визначає властивості products і categories, які будуть використовуватися для генерування контенту HTML у шаблоні на підставі даних, одержаних з репозиторію.

Щоб реалізувати шаблон компонента, створив файл shop.component.html в папці SportShop/src/app/shop та додав контент HTML нижче:


```
HolovniaIP11Lab5 > SportShop > src > app > shop > <> shop.component.html > div.c
1 <div class="container-fluid">
2   <div class="row">
3     <div class="bg-dark text-white p-2">
4       <span class="navbar-brand ml-2">SportShop</span>
5     </div>
6   </div>
7   <div class="row text-white">
8     <div class="col-3 bg-info p-2">
9       {{categories.length}} Categories
10    </div>
11    <div class="col-9 bg-success p-2">
12      {{products.length}} Products
13    </div>
14  </div>
15 </div>
```

Початкова версія шаблону проста. Більшість елементів надає структуру для макета магазину та застосування деяких CSS-класів Bootstrap. На даний момент використовуються лише дві прив'язки даних Angular, позначені символами `{{i}}`. Це прив'язки до рядкової інтерполяції; вони наказують Angular обчислити вираз прив'язки і вставити результат в елемент. Вирази у цих прив'язках виводять кількість продуктів та категорій, що надаються компонентом сховища.

Створення функціонального модуля сховища

Щоб створити функціональний модуль Angular для функціональності магазину, створив файл з ім'ям `shop.module.ts` в папці `SportShop/src/app/shop` та додав код нижче:

```
HolovniaIP11Lab5 > SportShop > src > app > shop > TS shop.module.ts > ShopModule
1 import { NgModule } from "@angular/core";
2 import { BrowserModule } from "@angular/platform-browser";
3 import { FormsModule } from "@angular/forms";
4 import { ModelModule } from "../model/model.module";
5 import { ShopComponent } from "./shop.component";
6 @NgModule({
7   imports: [ModelModule, BrowserModule, FormsModule],
8   declarations: [ShopComponent],
9   exports: [ShopComponent]
10 })
11 export class ShopModule {}
```

Декоратор `@NgModule` налаштовує модуль; при цьому властивість `imports` використовується для передачі Angular інформації про те, що модуль магазину залежить від модуля моделі, а також модулів `Browser Module` і `FormsModule`, що містять стандартні функції Angular для веб-застосунків і роботи з елементами форм HTML. Декоратор використовує властивість `declarations` для передачі Angular інформації про клас `ShopComponent`, який (як повідомляє властивість `exports`) може використовуватися в інших частинах програми, це важливо, тому що він буде використовуватися кореневим модулем.

Оновлення кореневого компонента та кореневого модуля

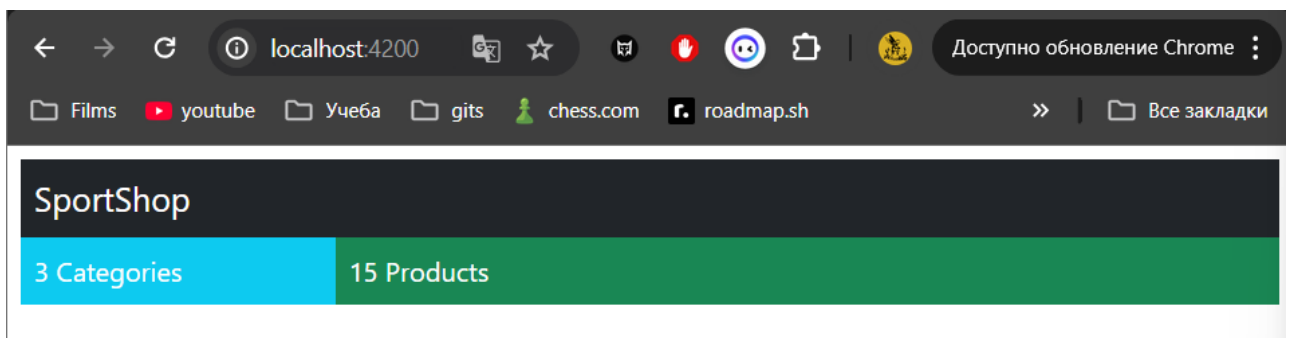
Застосування базової функціональності магазину та моделі вимагає оновлення кореневого модуля програми: він повинен імпортувати два функціональні модулі, а також оновити шаблон кореневого модуля для додавання елемента HTML, до якого буде застосовуватися компонент модуля магазину. Нижче представлені зміни шаблону кореневого компонента.

```
HolovniaIP11Lab5 > SportShop > src > app > TS app.component.ts > AppComponent
1  import { Component } from "@angular/core";
2  @Component({
3    selector: "app",
4    standalone: true,
5    template: "<shop></shop>"
6  })
7  export class AppComponent { }
```

Елемент `shop` замінює попередній контент у шаблоні кореневого компонента та відповідає значенню властивості `selector` декоратора `@Component`. Нижче показані зміни, які необхідно внести до кореневого модуля, щоб середовище Angular завантажувало функціональний модуль з функціональністю магазину.

```
HolovniatP11Lab5 > SportShop > src > app > TS app.module.ts > AppModule
1  import { NgModule } from '@angular/core';
2  import { BrowserModule } from '@angular/platform-browser';
3  import { AppComponent } from './app.component';
4  import { ShopModule } from '../shop/shop.module';
5  @NgModule({
6    declarations: [AppComponent],
7    imports: [BrowserModule, ShopModule],
8    providers: [],
9    bootstrap: [AppComponent]
10 })
11 export class AppModule {}
```

Angular матиме всю інформацію, необхідну для завантаження програми та відображення контенту з модуля магазину.



Всі створені структурні блоки спільно працюють для відображення контенту, який показує, скільки в магазині товарів і на скільки категорій вони діляться.

Додавання функціональності: докладна інформація про товари. Виведення докладної інформації про товари

Очевидна відправна точка для роботи над магазином — виведення докладної інформації про товари, щоб користувач бачив, що йому пропонують. Нижче у шаблон компонента магазину додаються елементи HTML з прив'язками даних, що генерують контент для кожного товару, що надається компонентом.

```

HolovniaIP11Lab5 > SportShop > src > app > shop > <> shop.component.html > div.container-fluid > div.row.text-w
1  <div class="container-fluid">
2    <div class="row">
3      <div class="bg-dark text-white p-2">
4        <span class="navbar-brand ml-2">SportShop</span>
5      </div>
6    </div>
7    <div class="row text-white">
8      <div class="col-3 bg-info p-2">
9        {{categories.length}} Categories
10     </div>
11     <div class="col-9 p-2 text-dark">|
12       <div *ngFor="let product of products" class="card m-1 p-1 bg-light">
13         <h4>
14           {{product.name}}
15           <span class="badge rounded-pill bg-primary" style="float:right">
16             {{ product.price | currency:"USD":"symbol":"2.2-2" }}
17           </span>
18         </h4>
19         <div class="card-text bg-white p-1">{{product.description}}</div>
20       </div>
21     </div>
22   </div>
23 </div>

```

Більшість елементів керує макетом та зовнішнім виглядом контенту. Найважливіша зміна – додавання виразу прив'язки даних Angular.

```
<div *ngFor="let product of products" class="card m-1 p-1 bg-light">
```

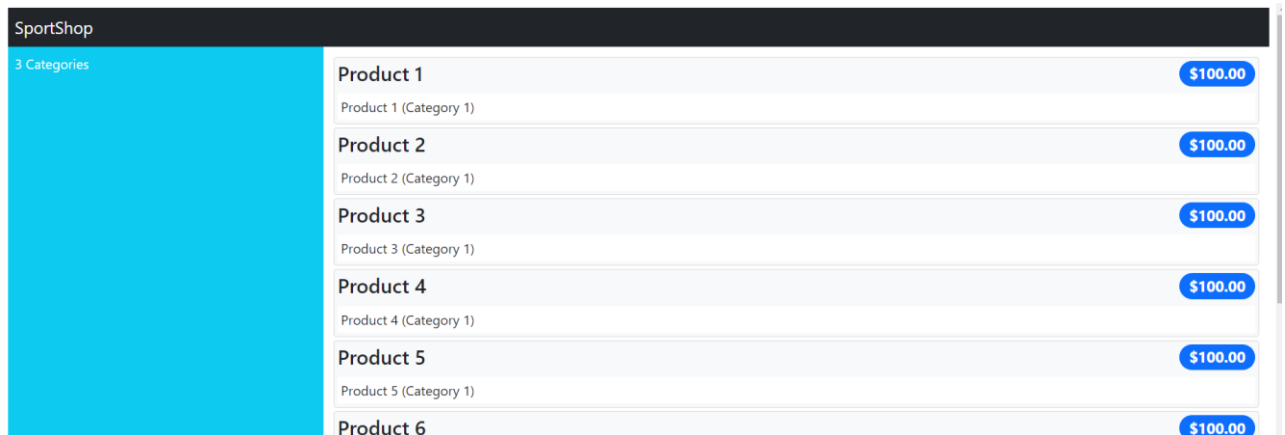
Перед вами приклад директиви, що трансформує елемент HTML, до якого вона застосовується. Ця конкретна директива `ngFor` перетворює елемент `div` дублюючи його для кожного об'єкта, що повертається властивістю `products` компонента. Angular включає низку вбудованих директив для вирішення більшості типових завдань. При дублюванні елемента `div` поточний об'єкт надається змінною з ім'ям `product`, що дозволяє легко посилатися на нього з інших прив'язок даних - як у наступному прикладі, де значення властивості `description` поточного товару вставляється як контент елемента `div`:

```
<div class="card-text bg-white p-1">{{product.description}}</div>
```

Не всі дані моделі даних проекту можуть виводитися безпосередньо для користувача. Angular включає механізм каналів (`pipes`), що використовуються класами для перетворення або підготовки значень для прив'язування даних. Angular містить кілька вбудованих каналів, включаючи канал `currency`, що форматує числові значення у грошовому форматі:

```
{{ product.price | currency:"USD":"symbol":"2.2-2" }}
```

Вираз у цій прив'язці наказує Angular відформатувати властивість `price` поточного продукту з використанням каналу `currency` за правилами форматування грошових величин, прийнятими в США. Збережіть зміни у шаблоні. Список товарів з моделі даних виводиться у вигляді довгого списку.



Додавання вибору категорій

Щоб додати підтримку фільтрації списку товарів за категоріями, необхідно підготувати компонент магазину. Він повинен стежити за тим, яка категорія була обрана користувачем, та змінювати механізм вибірки даних для використання обраної категорії

```
HolovniyIP11Lab5 > SportShop > src > app > shop > TS shop.component.ts > ShopComponent
1  import { Component } from "@angular/core";
2  import { Product } from "../model/product.model";
3  import { ProductRepository } from "../model/product.repository";
4  @Component({
5    selector: 'shop',
6    templateUrl: "shop.component.html",
7  })
8  export class ShopComponent {
9    selectedCategory: string | undefined;
10   constructor(private repository: ProductRepository) { }
11   get products(): Product[] {
12     return this.repository.getProducts(this.selectedCategory);
13   }
14   get categories(): string[] {
15     return this.repository.getCategories();
16   }
17   changeCategory(newCategory?: string) {
18     this.selectedCategory = newCategory;
19   }
20 }
```

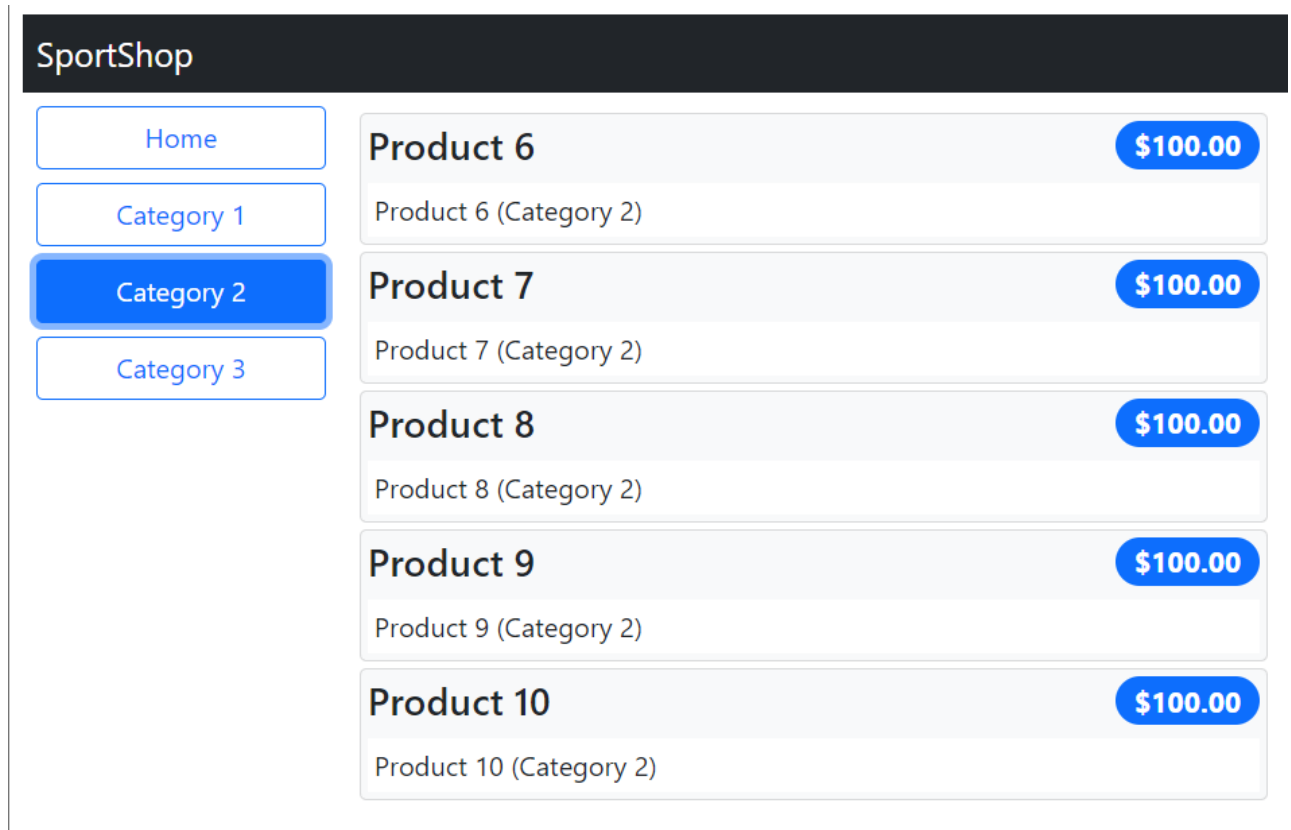
Властивості `selectedCategory` надається обрана користувачем категорія (null- Усі категорії); ця властивість використовується в методі `updateData` як аргумент методу `getProducts` так що фільтрація делегується джерелу даних. Метод `change Category` об'єднує ці значення у методі, який може викликатись при виборі категорії користувачем. Нижчн представлені відповідні зміни шаблону компонента. Шаблон повинен відображати набір кнопок для зміни вибраної категорії та показувати, яка категорія обрана зараз.

```
HolovniatP11Lab5 > SportShop > src > app > shop > shop.component.html > div.container-fluid
1  <div class="container-fluid">
2    <div class="row">
3      <div class="bg-dark text-white p-2">
4        <span class="navbar-brand ml-2">SportShop</span>
5      </div>
6    </div>
7    <div class="row text-white">
8      <div class="col-3 p-2">
9        <div class="d-grid gap-2">
10         <button class="btn btn-outline-primary" (click)="changeCategory()">
11           Home
12         </button>
13         <button *ngFor="let cat of categories" class="btn btn-outline-primary"
14           [class.active]="cat == selectedCategory" (click)="changeCategory(cat)">
15           {{cat}}
16         </button>
17       </div>
18     </div>
19     <div class="col-9 p-2 text-dark">
20       <div *ngFor="let product of products" class="card m-1 p-1 bg-light">
21         <h4>
22           {{product.name}}
23           <span class="badge rounded-pill bg-primary" style="float:right">
24             {{ product.price | currency:"USD":"symbol":"2.2-2" }}
25           </span>
26         </h4>
27         <div class="card-text bg-white p-1">{{product.description}}</div>
28       </div>
29     </div>
30   </div>
31 </div>
```

У шаблоні з'явилися два нових елементи `button`. Перша – кнопка `Home` - має прив'язку події, яка викликає метод `changeCategory` компонента при натисканні на кнопці. Метод не отримує аргументу, що рівнозначне призначенню категорії `null` та вибору всіх товарів. Прив'язка `ngFor` застосовується до іншого елементу `button` з виразом, який повторює елемент для кожного значення в масиві, що повертається властивістю `categories` компонента. Кнопці також призначено прив'язку події `click`, вираз якої викликає метод

changeCategory для вибору поточної категорії; це призведе до фільтрації списку товарів, що виводяться для користувача. Також є прив'язка class яка додає елемент button до активного класу, коли категорія, пов'язана з кнопкою, збігається з обраною категорією.

Таким чином, забезпечується візуальний зворотний зв'язок для користувача при фільтрації за категоріями



Посторінне виведення списку товарів

Фільтрування продуктів за категоріями спрощує роботу зі списком товарів, але в більш типовому рішенні список розбивається на менші фрагменти, і кожен фрагмент виводиться на окремій сторінці з навігаційними кнопками для переміщення між сторінками. Нижче у компонент магазину вносяться зміни, щоб у ньому зберігалася поточна сторінка і кількість елементів на сторінці.

```

HolovniaIP11Lab5 > SportShop > src > app > shop > TS shop.component.ts > ...
1  import { Component } from "@angular/core";
2  import { Product } from "../model/product.model";
3  import { ProductRepository } from "../model/product.repository";
4  @Component({
5      selector: "shop",
6      templateUrl: "shop.component.html"
7  })
8  export class ShopComponent {
9      selectedCategory: string | undefined;
10     productsPerPage = 4;
11     selectedPage = 1;
12     constructor(private repository: ProductRepository) { }
13     get products(): Product[] {
14         let pageIndex = (this.selectedPage - 1) * this.productsPerPage
15         return this.repository.getProducts(this.selectedCategory)
16             .slice(pageIndex, pageIndex + this.productsPerPage);
17     }
18     get categories(): string[] {
19         return this.repository.getCategories();
20     }
21     changeCategory(newCategory?: string) {
22         this.selectedCategory = newCategory;
23     }
24     changePage(newPage: number) {
25         this.selectedPage = newPage;
26     }
27     changePageSize(newSize: number) {
28         this.productsPerPage = Number(newSize);
29         this.changePage(1);
30     }
31     get pageNumbers(): number[] {
32         return Array(Math.ceil(this.repository
33             .getProducts(this.selectedCategory).length / this.productsPerPage))
34             .fill(0).map((x, i) => i + 1);
35     }
36 }
37

```

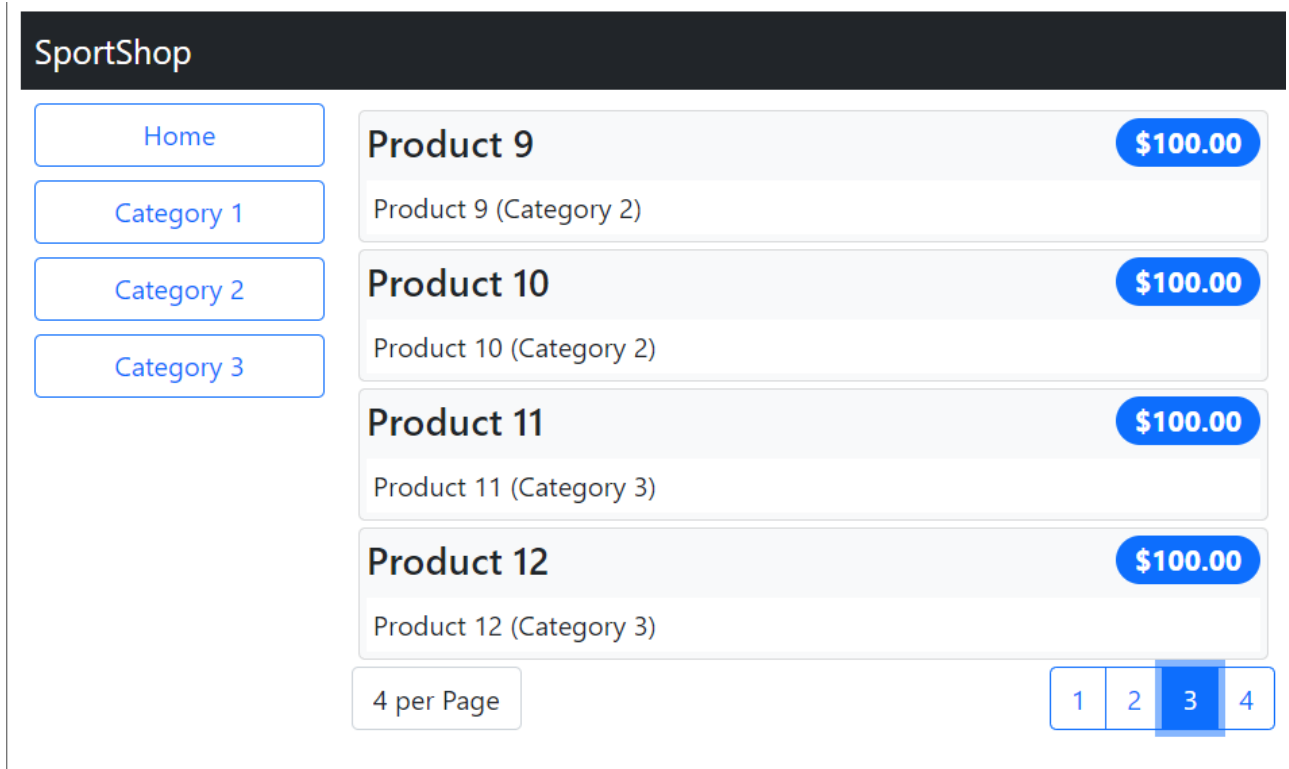
У цьому лістингу реалізовано дві нові можливості: отримання сторінки з інформацією про товари та зміну розміру сторінок (зі зміною кількості товарів, що відображаються на кожній сторінці). Тут є одна дивина, на яку компоненту доводиться використовувати обхідне рішення. Вбудована директива `ngFor`, що надається Angular, дозволяє генерувати контент тільки для об'єктів із масиву або колекції (без використання лічильника). Так як нам потрібно згенерувати пронумеровані кнопки навігації між сторінками, доводиться створювати масив із потрібними числами:


```
return Array(Math.ceil(this.repository
  .getProducts(this.selectedCategory).length / this.productsPerPage))
  .fill(0).map((x, i) => i + 1);
```

Ця команда створює новий масив, заповнює його значенням 0, а потім за допомогою методу `map` генерує новий масив із числовою послідовністю. Таке рішення досить добре працює у реалізації сторінкового виводу, але виглядає досить незграбно; у наступному розділі буде продемонстровано більш вдале рішення. Нижче наведено зміни у шаблоні компонента магазину, необхідні для реалізації сторінкового виводу.

```
HolovniaIP11Lab5 > SportShop > src > app > shop > shop.component.html > div.container-fluid
11 | | | | | Home
12 | | | | | </button>
13 | | | | | <button *ngFor="let cat of categories" class="btn btn-outline-primary"
14 | | | | |   [class.active]="cat == selectedCategory" (click)="changeCategory(cat)"
15 | | | | |   {{cat}}
16 | | | | | </button>
17 | | | | | </div>
18 | | | | | </div>
19 | | | | | <div class="col-9 p-2 text-dark">
20 | | | | |   <div *ngFor="let product of products" class="card m-1 p-1 bg-light">
21 | | | | |     <h4>
22 | | | | |       {{product.name}}
23 | | | | |       <span class="badge rounded-pill bg-primary" style="float:right">
24 | | | | |         {{ product.price | currency:"USD": "symbol":"2.2-2" }}
25 | | | | |       </span>
26 | | | | |     </h4>
27 | | | | |     <div class="card-text bg-white p-1">{{product.description}}</div>
28 | | | | |   </div>
29 | | | | |   <div class="form-inline float-start mr-1">
30 | | | | |     <select class="form-control" [value]="productsPerPage"
31 | | | | |       (change)="changePageSize($any($event).target.value)"
32 | | | | |       <option value="3">3 per Page</option>
33 | | | | |       <option value="4">4 per Page</option>
34 | | | | |       <option value="6">6 per Page</option>
35 | | | | |       <option value="8">8 per Page</option>
36 | | | | |     </select>
37 | | | | |   </div>
38 | | | | |   <div class="btn-group float-end">
39 | | | | |     <button *ngFor="let page of pageNumbers" (click)="changePage(page)" class="btn btn-outline-primary"
40 | | | | |       [class.active]="page == selectedPage"
41 | | | | |       {{page}}
42 | | | | |     </button>
43 | | | | |   </div>
44 | | | | | </div>
45 | | | | | </div>
46 | | | | | </div>
```

До розмітки додається елемент `select`, що дозволяє змінювати розмір сторінки, та набір кнопок для переходу між сторінками товарів. Нові елементи містять прив'язки даних, що пов'язують їх із властивостями та методами, що надаються компонентом. В результаті ми отримуємо список товарів, з яким зручніше працювати



Створення нестандартної директиви

У цьому розділі ми створимо нестандартну директиву, щоб нам не доводилося генерувати масив заповнений числами для створення кнопок навігації. Angular надає хороший набір вбудованих директив, але розробник може відносно просто створювати власні директиви для вирішення завдань, притаманних його додатку, або для підтримки можливостей, які відсутні у вбудованих директивах. Створіть файл `counter.directive.ts` в папці `SportShop/app/shop` та використайте його для визначення класу

```

HolovniaIP11Lab5 > SportShop > src > app > shop > TS counter.directive.ts > CounterDirective
1  import {
2      Directive, ViewContainerRef, TemplateRef, Input, SimpleChanges
3  } from "@angular/core";
4  @Directive({
5      selector: "[counterOf]"
6  })
7  export class CounterDirective {
8      constructor(private container: ViewContainerRef,
9                  private template: TemplateRef<Object>) { }
10     @Input("counterOf")
11     counter: number = 0;
12     ngOnChanges(changes: SimpleChanges) {
13         this.container.clear();
14         for (let i = 0; i < this.counter; i++) {
15             this.container.createEmbeddedView(this.template,
16                 new CounterDirectiveContext(i + 1));
17         }
18     }
19 }
20 class CounterDirectiveContext {
21     constructor(public $implicit: any) { }
22 }

```

Це приклад структурної директиви. Такі директиви застосовуються до елементів через властивість `counter` та використовують спеціальні засоби Angular для багаторазового створення контенту (за аналогією із вбудованою директивою `ngFor`). У цьому випадку, замість того, щоб повертати кожен об'єкт у колекції, нестандартна директива повертає серію чисел, які можуть використовуватися для створення навігаційних кнопок між сторінками. Щоб використати директиву, її необхідно додати до властивості `declarations` функціонального модуля, як показано нижче:

```
HolovniaIP11Lab5 > SportShop > src > app > shop > TS shop.module.ts > ShopMo
1  import { NgModule } from "@angular/core";
2  import { BrowserModule } from "@angular/platform-browser";
3  import { FormsModule } from "@angular/forms";
4  import { ModelModule } from "../model/model.module";
5  import { ShopComponent } from "./shop.component";
6  import { CounterDirective } from "./counter.directive";
7  @NgModule({
8      imports: [ModelModule, BrowserModule, FormsModule],
9      declarations: [ShopComponent, CounterDirective],
10     exports: [ShopComponent]
11 })
12 export class ShopModule {}
```

Після того, як директива була зареєстрована, вона може використовуватися в шаблоні компонента магазину для заміни директиви ngFor, як показано нижче:

```
</div>
<div class="btn-group float-end">
  <button *counter="let page of pageCount" (click)="changePage(page)" class="btn btn-outline-primary"
    [class.active]="page == selectedPage">
    {{page}}
  </button>
</div>
</div>
```

Нова прив'язка даних залежить від налаштування нестандартної директиви з використанням властивості pageCount. Нижче масив чисел замінюється простим значенням number, що представляє результат виразу.

```
31  // get pageNumbers(): number[] {
32  // return Array(Math.ceil(this.repository
33  // .getProducts(this.selectedCategory).length |
34  //this.productsPerPage))
35  // .fill(0).map((x, i) => i + 1);
36  // }
37  get pageCount(): number {
38      return Math.ceil(this.repository
39          .getProducts(this.selectedCategory).length /
40          this.productsPerPage)
41  }
42  }
```

В додатку SportShop зовні нічого не змінилося, але продемонстрував використання нестандартної директиви для демонстрації того, як вбудовані функції Angular можуть розширюватись спеціалізованим кодом, адаптованим

для потреб конкретного проекту. В цій лабораторній роботі ми розпочали роботу над проектом SportShop.

Ми «заклали фундамент» для проекту: установили та налаштували засоби розробника, створили кореневі структурні блоки для застосування та початку роботи над функціональними модулями. Вивели дані фіктивної моделі даних, реалізували розбивку на сторінки та фільтрацію товарів за категоріями. Також створили нестандартну директиву для демонстрації того, як вбудовані функції Angular можуть розширюватись спеціалізованим кодом.


II) Частина 2: Вибір товарів та оформлення замовлень. Підготовка програми

Створення кошика

Користувачеві знадобиться кошик, в якому розміщуються продукти для подальшого оформлення замовлення. У цій роботі додамо функціональність кошика в додаток і інтегруємо його в магазин, щоб користувач міг вибрати товари, що його цікавлять.

Створення моделі кошика

Вихідною точкою для функціональності кошика стане новий клас моделі, який використовуватиметься для збирання товарів, обраних користувачем. Створимо файл з ім'ям `cart.model.ts` в папці `SportShop/src/app/model` та включимо у нього визначення класу нижче:

HolovniaIP11Lab5 > SportShop > src > app > model > TS cart.model.ts >  CartLine

```
1  import { Injectable } from "@angular/core";
2  import { Product } from "../product.model";
3  @Injectable()
4  export class Cart {
5      public lines: CartLine[] = [];
6      public itemCount: number = 0;
7      public cartPrice: number = 0;
8      addLine(product: Product, quantity: number = 1) {
9          let line = this.lines.find(line => line.product.id == product.id);
10         if (line != undefined) {
11             line.quantity += quantity;
12         } else {
13             this.lines.push(new CartLine(product, quantity));
14         }
15         this.recalculate();
16     }
17     updateQuantity(product: Product, quantity: number) {
18         let line = this.lines.find(line => line.product.id == product.id);
19         if (line != undefined) {
20             line.quantity = Number(quantity);
21         }
22         this.recalculate();
23     }
24     removeLine(id: number) {
25         let index = this.lines.findIndex(line => line.product.id == id);
26         this.lines.splice(index, 1);
27         this.recalculate();
28     }
29     clear() {
30         this.lines = [];
31         this.itemCount = 0;
32         this.cartPrice = 0;
33     }
```

```
34  private recalculate() {
35      this.itemCount = 0;
36      this.cartPrice = 0;
37      this.lines.forEach(l => {
38          this.itemCount += l.quantity;
39          this.cartPrice += l.lineTotal;
40      })
41  }
42  }
43  export class CartLine {
44      constructor(public product: Product,
45                  public quantity: number) { }
46      get lineTotal() {
47          return this.quantity * (this.product.price ?? 0);
48      }
49  }
```

Вибрані продукти є масивом об'єктів `CartItem`, кожен з яких містить об'єкт `Product` та кількість одиниць товару. В класі `Cart` зберігається загальна кількість обраних товарів та їх загальна вартість, яка відображатиметься у процесі покупки. У всьому додатку повинен використовуватися лише один об'єкт `Cart`, який гарантує, що будь-яка частина програми зможе отримати інформацію про товари, обрані користувачем. Для цього ми оформимо `Cart` у вигляді глобальної служби; це означає, що Angular буде відповідати за створення екземпляра класу `Cart` і використовувати його, коли потрібно створити компонент з аргументом конструктора `Cart`.

Це ще один приклад використання механізму впровадження залежностей Angular, який може використовуватися для спільного доступу до об'єктів у додатку. Декоратор `@Injectable`, який застосовується до класу `Cart` у лістингу означає, що клас буде використовуватися як служба. (Строго кажучи, декоратор `@Injectable` обов'язковий лише за наявності у класі власних аргументів конструктора; проте його краще застосовувати завжди, тому що він сигналізує, що клас призначений для використання як служба.) реєструє клас `Cart` як службу у властивості `providers` функціонального модуля моделі.

```
HolovniyP11Lab5 > SportShop > src > app > model > TS model.module.ts > ModelModule
1  import { NgModule } from "@angular/core";
2  import { ProductRepository } from "../product.repository";
3  import { StaticDataSource } from "../static.datasource";
4  import { Cart } from "../cart.model";
5  @NgModule({
6    providers: [ProductRepository, StaticDataSource, Cart]
7  })
8  export class ModelModule {}
```

Створення компонентів для зведеної інформації кошика

Компоненти є основними структурними блоками у додатках Angular, тому що вони дозволяють легко створювати ізольовані блоки коду та контенту. Програма `SportShop` виводить зведену інформацію про вибрані товари в заголовку сторінки; для реалізації цієї функціональності треба створити

компонент. Створив файл з ім'ям `cartSummary.component.ts` в папці `SportShop/src/app/shop` та визначив в ньому компонент

```
HolovniatP11Lab5 > SportShop > src > app > shop > TS cartSummary.component.ts >
1  import { Component } from "@angular/core";
2  import { Cart } from "../model/cart.model";
3  @Component({
4    selector: "cart-summary",
5    templateUrl: "cartSummary.component.html"
6  })
7  export class CartSummaryComponent {
8    constructor(public cart: Cart) { }
9  }
```

Коли потрібно створити екземпляр цього компонента, середовище Angular має надати об'єкт `Cart` як аргумент конструктора з використанням служби, налаштованої раніше (додаванням класу `Cart` у властивість `providers` функціонального модуля). У варіанті поведінки за замовчуванням один об'єкт `Cart` буде створено та використано у додатку, хоча доступні різні варіанти поведінки служб.

Щоб надати компоненту шаблон, створив файл HTML з ім'ям `cartSummary.component.html` в одній папці з файлом класу компонента та додав розмітку

```
HolovniatP11Lab5 > SportShop > src > app > shop > <> cartSummary.component.html > div.float-end
1  <div class="float-end">
2    <small class="fs-6">
3      Your cart:
4      <span *ngIf="cart.itemCount > 0">
5        {{ cart.itemCount }} item(s)
6        {{ cart.cartPrice | currency:"USD":"symbol":"2.2-2" }}
7      </span>
8      <span *ngIf="cart.itemCount == 0">
9        (empty)
10     </span>
11   </small>
12   <button class="btn btn-sm bg-dark text-white" [disabled]="cart.itemCount == 0">
13     <i class="fa fa-shopping-cart"></i>
14   </button>
15 </div>
```

Шаблон використовує об'єкт `Cart`, наданий компонентом, для виведення кількості товарів у кошику та їх загальної вартості. Також передбачена кнопка для запуску процесу оформлення замовлення, який буде додано до програми

пізніше. Код нижче реєструє новий компонент у функціональному модулі магазину, щоб підготуватися до його використання пізніше.

```
HolovniatP11Lab5 > SportShop > src > app > shop > TS shop.module.ts > ShopModule
1  import { NgModule } from "@angular/core";
2  import { BrowserModule } from "@angular/platform-browser";
3  import { FormsModule } from "@angular/forms";
4  import { ModelModule } from "../model/model.module";
5  import { ShopComponent } from "./shop.component";
6  import { CounterDirective } from "./counter.directive";
7  import { CartSummaryComponent } from "./cartSummary.component";
8  @NgModule({
9      imports: [ModelModule, BrowserModule, FormsModule],
10     declarations: [ShopComponent, CounterDirective,
11                   CartSummaryComponent],
12     exports: [ShopComponent]
13 })
14 export class ShopModule { }
```

Інтеграція кошика у додаток

Shop компонент відіграє ключову роль в інтеграції кошика та його віджету у додаток. Код нижче оновлюється компонент Shop: до нього додається конструктор з параметром Cart, а також визначається метод додавання товару в кошик.

```
HolovniatP11Lab5 > SportShop > src > app > shop > TS shop.component.ts > ShopComponent
1  import { Component } from "@angular/core";
2  import { Product } from "../model/product.model";
3  import { ProductRepository } from "../model/product.repository";
4  import { Cart } from "../model/cart.model";
5  @Component({
6      selector: "shop",
7      templateUrl: "shop.component.html"
8  })
9  export class ShopComponent {
10     selectedCategory: string | undefined;
11     productsPerPage = 4;
12     selectedPage = 1;
13     constructor(private repository: ProductRepository, private cart: Cart) { }
14     get products(): Product[] {
15         let pageIndex = (this.selectedPage - 1) * this.productsPerPage
16         return this.repository.getProducts(this.selectedCategory)
17             .slice(pageIndex, pageIndex + this.productsPerPage);
18     }
19     get categories(): string[] {
20         return this.repository.getCategories();
21     }
22     changeCategory(newCategory?: string) {
23         this.selectedCategory = newCategory;
24     }
25     changePage(newPage: number) {
26         this.selectedPage = newPage;
27     }
28     changePageSize(newSize: number) {
29         this.productsPerPage = Number(newSize);
30         this.changePage(1);
31     }

```

```

25     changePage(newPage: number) {
26         this.selectedPage = newPage;
27     }
28     changePageSize(newSize: number) {
29         this.productsPerPage = Number(newSize);
30         this.changePage(1);
31     }
32     get pageCount(): number {
33         return Math.ceil(this.repository
34             .getProducts(this.selectedCategory).length / this.productsPerPage)
35     }
36     addProductToCart(product: Product) {
37         this.cart.addLine(product);
38     }
39 }

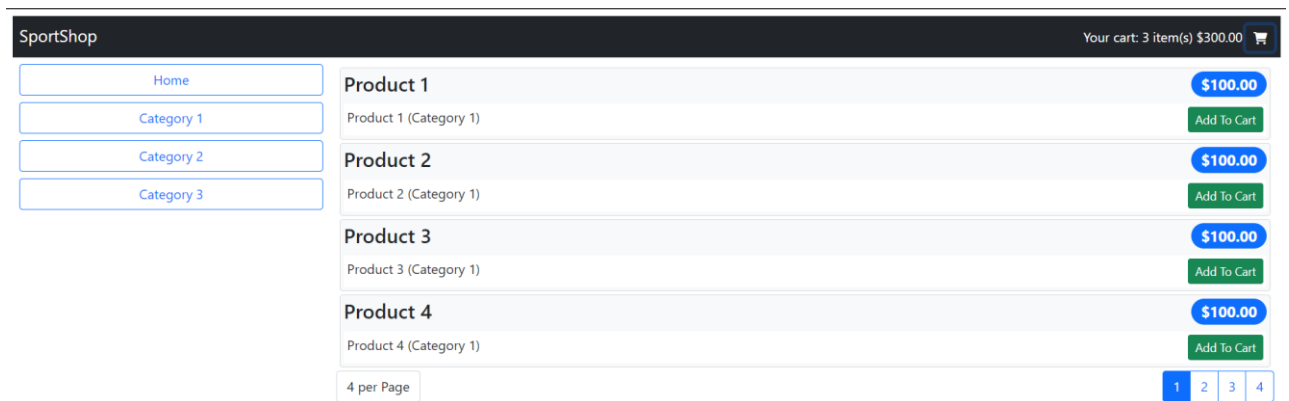
```

Щоб завершити інтеграцію кошика до компоненту магазину, далі додається елемент, який застосовує компонент зі зведеною інформацією кошика

до шаблону компонента магазину та додає в опис кожного товару кнопку з прив'язкою події для виклику методу `addProductToCart`.

```
HolovniatP11Lab5 > SportShop > src > app > shop > shop.component.html > div.container-fluid
1  <div class="container-fluid">
2    <div class="row">
3      <div class="bg-dark text-white p-2">
4        <span class="navbar-brand ml-2">
5          SportShop
6          <cart-summary></cart-summary>
7        </span>
8      </div>
9    </div>
10   <div class="row text-white">
11     <div class="col-3 p-2">
12       <div class="d-grid gap-2">
13         <button class="btn btn-outline-primary" (click)="changeCategory()">
14           Home
15         </button>
16         <button *ngFor="let cat of categories" class="btn btn-outline-primary"
17           [class.active]="cat == selectedCategory" (click)="changeCategory(cat)">
18           {{cat}}
19         </button>
20       </div>
21     </div>
22     <div class="col-9 p-2 text-dark">
23       <div *ngFor="let product of products" class="card m-1 p-1 bg-light">
24         <h4>
25           {{product.name}}
26           <span class="badge rounded-pill bg-primary" style="float:right">
27             {{ product.price | currency:'USD':"symbol":"2.2-2" }}
28           </span>
29         </h4>
30       <div class="card-text bg-white p-1">
31         {{product.description}}
32         <button class="btn btn-success btn-sm float-end" (click)="addProductToCart(product)">
33           Add To Cart
34         </button>
35       </div>
36     </div>
37     <div class="form-inline float-start mr-1">
38       <select class="form-control" [value]="productsPerPage"
39         (change)="changePageSize($any($event).target.value)">
40         <option value="3">3 per Page</option>
41         <option value="4">4 per Page</option>
42         <option value="6">6 per Page</option>
43         <option value="8">8 per Page</option>
44       </select>
45     </div>
46     <div class="btn-group float-end">
47       <button *counter="let page of pageCount" (click)="changePage(page)" class="btn btn-outline-primary"
48         [class.active]="page == selectedPage">
49         {{page}}
50       </button>
51     </div>
52   </div>
53 </div>
54 </div>
```

У результаті для кожного товару створюється кнопка додавання до корзини. Повноцінна підтримка кошика ще не реалізована, але наслідки кожного додавання товару відображаються у зведенні у верхній частині сторінки.



Зверніть увагу: при натисканні однієї з кнопок Add To Cart вміст компонента зведення змінюється автоматично. Це стало можливим завдяки тому, що один об'єкт Cart спільно використовується двома компонентами і зміни, що вносяться одним компонентом, відображаються при обчисленні виразів прив'язок даних в іншому компоненті.

Маршрутизація URL

Багато програм відображають різний контент у різні моменти часу. У програмі SportShop при натисканні однієї з кнопок Add To Cart користувач повинен побачити докладний опис обраних товарів, а також отримати можливість запустити процес оформлення замовлення. Angular підтримує механізм маршрутизації URL, який використовує поточну URLадресу в браузері для вибору компонентів, що відображаються для користувача [46, 47, 48]. Цей механізм спрощує створення додатків, в яких компоненти не мають жорсткого зчеплення та легко змінюються без необхідності внесення змін до інших місць.

Маршрутизація URL також дозволяє легко змінити шлях, яким користувач взаємодіє з додатком. У програмі SportShop додамо підтримку трьох різних URL-адрес з табл. 9.1.

Таблиця 9.1.URL-адреси, які підтримує програма SportShop

URL	Опис
/shop	URL для виведення списку товарів
/cart	URL для виведення кошика
/checkout	URL для процесу оформлення замовлення

```
HolovniaIP11Lab5 > SportShop > src > app > shop > TS cartDetail.component.ts > CartDetailComponent
1 import { Component } from "@angular/core";
2 @Component({
3   template: `<div><h3 class="bg-info p-1 text-white">Cart Detail Component</h3></div>`
4 })
5 export class CartDetailComponent {}
```

Потім створимо файл checkout.component.ts в папці SportShop/src/app/shop і додамо визначення компонента:

```
HolovniaIP11Lab5 > SportShop > src > app > shop > TS checkout.component.ts > CheckoutComponent
1 import { Component } from "@angular/core";
2 @Component({
3   template: `<div><h3 class="bg-info p-1 text-white">Checkout Component</h3></div>`
4 })
5 export class CheckoutComponent {}
```

Компонент побудований за тією ж схемою, що і компонент кошика: він виводить тимчасове повідомлення, яке показує, який компонент відображається. У коді компоненти реєструються у функціональному модулі shop і включаються у властивість exports, щоб вони могли використовуватись в інших місцях програми.

```
HolovniaIP11Lab5 > SportShop > src > app > shop > TS shop.module.ts > ShopModule
1 import { NgModule } from "@angular/core";
2 import { BrowserModule } from "@angular/platform-browser";
3 import { FormsModule } from "@angular/forms";
4 import { ModelModule } from "../model/model.module";
5 import { ShopComponent } from "./shop.component";
6 import { CounterDirective } from "./counter.directive";
7 import { CartSummaryComponent } from "./cartsummary.component";
8 import { CartDetailComponent } from "./cartDetail.component";
9 import { CheckoutComponent } from "./checkout.component";
10 @NgModule({
11   imports: [ModelModule, BrowserModule, FormsModule],
12   declarations: [ShopComponent, CounterDirective, CartSummaryComponent,
13     CartDetailComponent, CheckoutComponent],
14   exports: [ShopComponent, CartDetailComponent, CheckoutComponent]
15 })
16 export class ShopModule {}
```

Створення та застосування конфігурації маршрутизації

Тепер, коли ми маємо набір компонентів, на наступному кроці створюється конфігурація маршрутизації, яка описує відповідності між URL і компонентами. Кожна відповідність між URL і компонентом називається маршрутом URL, або просто маршрутом (route).

При створенні більш складних конфігурацій маршрутизації, маршрути визначаються в окремому файлі, але в цьому проекті використаємо інше рішення - визначення маршрутів у декораторі `@NgModule` кореневого модуля програми (лістинг 9.11). Механізм маршрутизації Angular вимагає присутності в документі HTML елемента `base`, що визначає базову URL-адресу, до якої застосовуються маршрути. Цей елемент був доданий раніше, коли ми створювали проект SportShop. Якщо елемент пропущено, Angular повідомить про помилку та не зможе застосувати маршрути.

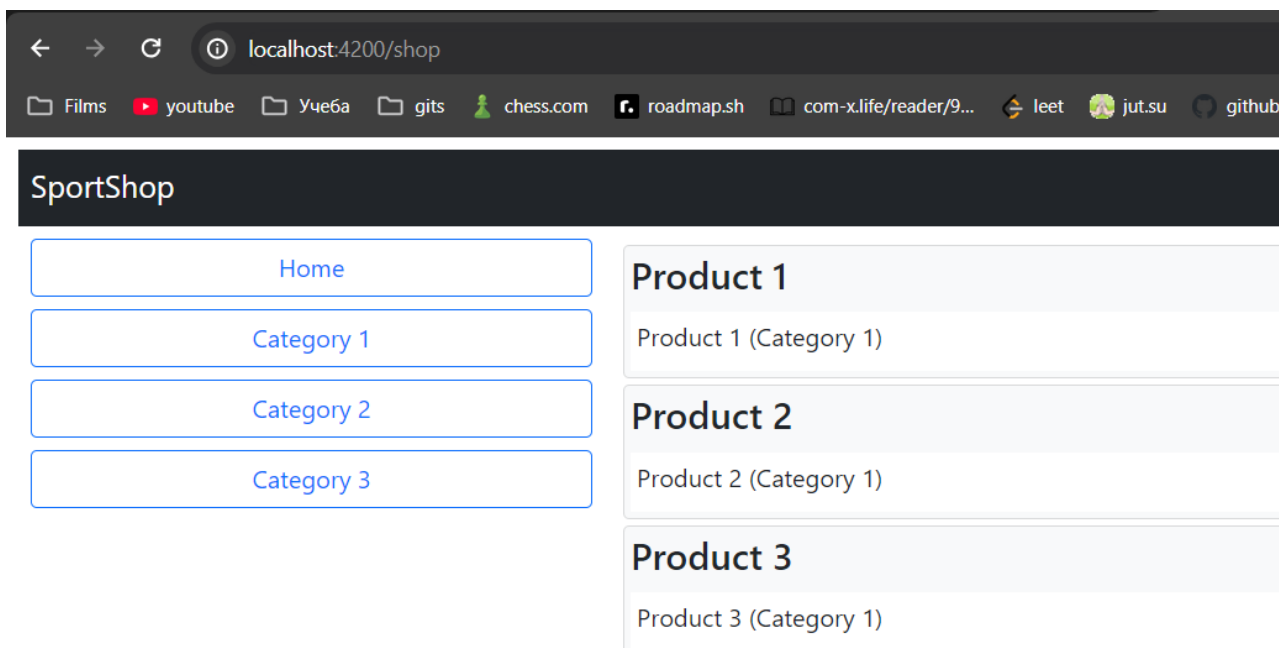
```
TS app.module.ts X
HolovniaIP11Lab5 > SportShop > src > app > TS app.module.ts > AppModule
1  import { NgModule } from "@angular/core";
2  import { BrowserModule } from "@angular/platform-browser";
3  import { AppComponent } from "../app.component";
4  import { ShopModule } from "../shop/shop.module";
5  import { ShopComponent } from "../shop/shop.component";
6  import { CheckoutComponent } from "../shop/checkout.component";
7  import { CartDetailComponent } from "../shop/cartDetail.component";
8  import { RouterModule } from "@angular/router";
9  @NgModule({
10     declarations: [AppComponent],
11     imports: [BrowserModule, ShopModule,
12             RouterModule.forRoot([
13                 { path: "shop", component: ShopComponent },
14                 { path: "cart", component: CartDetailComponent },
15                 { path: "checkout", component: CheckoutComponent },
16                 { path: "**", redirectTo: "/shop" }
17             ])],
18     providers: [],
19     bootstrap: [AppComponent]
20 })
21 export class AppModule { }
```

Методу `RouterModule.forRoot` передається набір маршрутів, кожен із яких пов'язує URL з компонентом. Перші три маршрути у лістингу відповідають

URL. Останній маршрут є універсальним — він перенаправляє будь-яку іншу URL на /shop, що відображає ShopComponent. При використанні механізму маршрутизації Angular шукає елемент router-outlet, що визначає місце для пошуку компонента, що відповідає поточному URL. У лістингу 9.12 елемент shop шаблону кореневого компонента замінюється елементом router-outlet.

```
HolovniaIP11Lab5 > SportShop > src > app > TS app.component.ts > AppComponent
1  import { Component } from "@angular/core";
2  @Component({
3    selector: "app",
4    template: "<router-outlet></router-outlet>"
5  })
6  export class AppComponent { }
```

Angular застосовує конфігурацію маршрутизації, коли ви зберігаєте зміни, а браузер перезавантажує HTML документ. Контент, що відображається у вікні браузера, не змінився, але в адресному рядку браузера видно, що конфігурація маршрутизації успішно застосована.



Навігація у додатку

Коли конфігурація маршрутизації буде налаштована, можна переходити до підтримки навігації між компонентами зміною URL-адреси у браузері. Механізм маршрутизації URL залежить від JavaScript API, що надається

браузером; це означає, що користувач не може просто ввести цільову URL-адресу в адресному рядку браузера. Натомість навігація повинна виконуватися програмою — або з використанням коду JavaScript у компоненті (або іншому структурному блоці), або з додаванням атрибутів до елементів HTML у шаблоні.

Коли користувач клацає на одній із кнопок Add To Cart, повинен відображатись компонент з інформацією кошика; це означає, що програма має перейти за URL-адресою /cart. У коді нижче навігація додається у метод компонента, який викликається при натисканні кнопки користувачем.

```
5 import { Router } from "@angular/router";
6 @Component({
7   selector: 'app-cart-summary',
8   templateUrl: './cart-summary.component.html',
9   styleUrls: ['./cart-summary.component.css'],
10  constructor(private repository: ProductRepository,
11               private cart: Cart,
12               private router: Router) { }
13
14  addProductToCart(product: Product) {
15    this.cart.addLine(product);
16    this.router.navigateByUrl("/cart");
17  }
18 })
```

Конструктор отримує параметр Router, що надається Angular через механізм впровадження залежностей при створенні нового екземпляра компонента. У методі addProductToCart метод Router.navigateByUrl використовується для переходу через URL /cart.

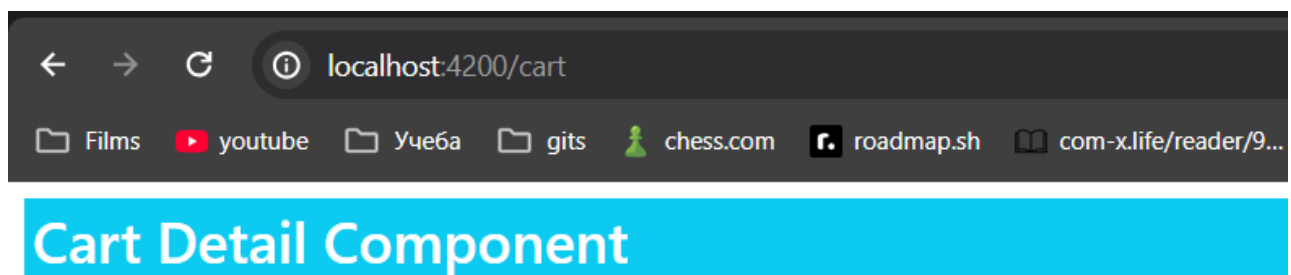
Навігація також може здійснюватися додаванням атрибуту routerLink елементу в шаблоні. У коді атрибут routerLink застосовується до кнопки у шаблоні компонента зведеної інформації кошика.

```
HolovniolP11Lab5 > SportShop > src > app > shop > cartSummary.component.html > div.float-end
1 <div class="float-end">
2   <small class="fs-6">
3     Your cart:
4     <span *ngIf="cart.itemCount > 0">
5       {{ cart.itemCount }} item(s)
6       {{ cart.cartPrice | currency:"USD":"symbol":"2.2-2" }}
7     </span>
8     <span *ngIf="cart.itemCount == 0">
9       (empty)
10    </span>
11  </small>
12  <button class="btn btn-sm bg-dark text-white" [disabled]="cart.itemCount == 0" routerLink="/cart">
13    <i class="fa fa-shopping-cart"></i>
14  </button>
15 </div>
```


Як значення атрибуту `routerLink` вказується URL-адреса, за якою має переходити додаток при натисканні на кнопки. Ця кнопка блокується при порожньому кошику, так що перехід буде відбуватися тільки після додавання товару в кошик користувачем. Для додавання підтримки атрибуту `routerLink` необхідно імпортувати модуль `RouterModule` у функціональний модуль, як показано нижче:

```
HolovniaIP11Lab5 > SportShop > src > app > shop > TS shop.module.ts > ShopModule
1  import { NgModule } from "@angular/core";
2  import { BrowserModule } from "@angular/platform-browser";
3  import { FormsModule } from "@angular/forms";
4  import { ModelModule } from "../model/model.module";
5  import { ShopComponent } from "./shop.component";
6  import { CounterDirective } from "./counter.directive";
7  import { CartSummaryComponent } from "./cartsummary.component";
8  import { CartDetailComponent } from "./cartDetail.component";
9  import { CheckoutComponent } from "./checkout.component";
10 import { RouterModule } from "@angular/router";
11 @NgModule({
12   imports: [ModelModule, BrowserModule, FormsModule, RouterModule],
13   declarations: [ShopComponent, CounterDirective, CartSummaryComponent,
14     |   CartDetailComponent, CheckoutComponent],
15   exports: [ShopComponent, CartDetailComponent, CheckoutComponent]
16 })
17 export class ShopModule {}
```

Щоб побачити, як навігація працює, збережемо зміни у файлах, а після того як браузер перезавантажить документ HTML, клацнімо на одній з кнопок `Add To Cart`. Браузер переходить за URL-адресою `/cart`



Захисники маршрутів

Пам'ятайте, що навігація може виконуватись лише програмою. Якщо змінити URLадресу прямо в адресному рядку браузера, то браузер запросить

введену URL-адресу у веб-сервера. Сервер розробки Angular, що відповідає на запити HTTP, відповідає на будь-який запит, що не відповідає файлу, повертаючи вміст index.html. Зазвичай така поведінка зручна, тому що вона запобігає помилці HTTP при натисканні на кнопки оновлення в браузері. Але така поведінка також може створити проблеми, якщо програма очікує, що користувач буде переходити в додатку певним шляхом. Наприклад, якщо клацнути на одній із кнопок Add To Cart, а потім клацнути на кнопці оновлення у браузері, то сервер HTTP поверне вміст файлу index.html, а Angular негайно перейде до компоненту вмісту кошика і пропустить частину програми, що дозволяє користувачеві вибирати продукти. У деяких додатках можливість починати з різних URL-адрес має сенс, а для інших випадків Angular підтримує захисників маршрутів (route guards), що використовуються для керування системою маршрутизації.

Щоб програма не могла починати з URL /cart або /order, додамо файл shopFirst.guard.ts у папку SportShop/app та визначимо клас:

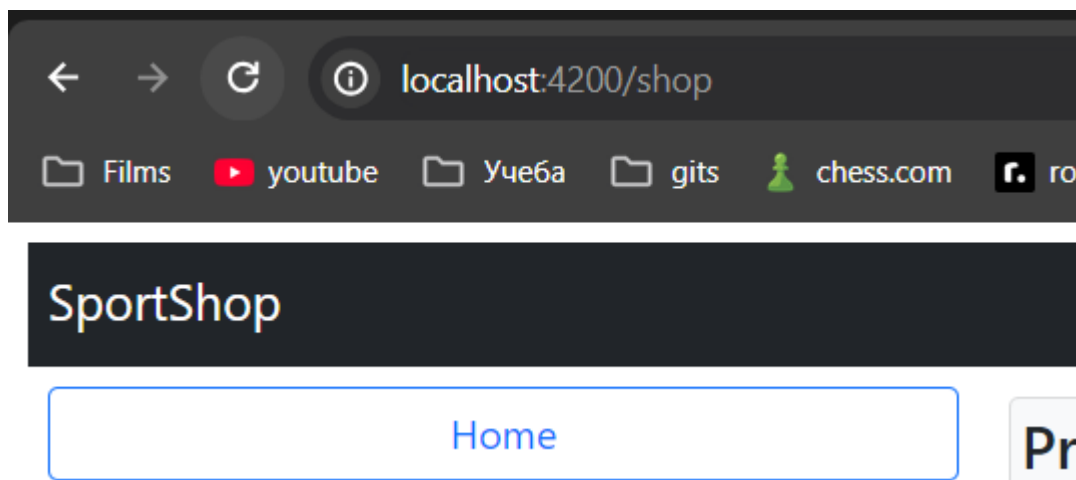
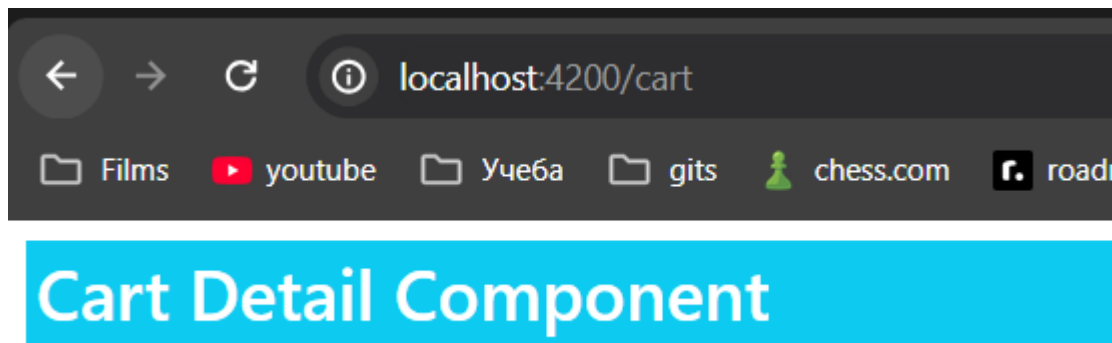
```
HolovniaIP11Lab5 > SportShop > src > app > TS shopFirst.guard.ts > ShopFirstGuard
1  import { Injectable } from "@angular/core";
2  import {
3      ActivatedRouteSnapshot, RouterStateSnapshot,
4      Router
5  } from "@angular/router";
6  import { ShopComponent } from "../shop/shop.component";
7  @Injectable()
8  export class ShopFirstGuard {
9      private firstNavigation = true;
10     constructor(private router: Router) { }
11     canActivate(route: ActivatedRouteSnapshot,
12         state: RouterStateSnapshot): boolean {
13         if (this.firstNavigation) {
14             this.firstNavigation = false;
15             if (route.component !== ShopComponent) {
16                 this.router.navigateByUrl("/");
17                 return false;
18             }
19         }
20         return true;
21     }
22 }
```

Існують різні способи захисту маршрутів; цей різновид захисника запобігає активізації маршруту. Він реалізується класом, що визначає метод `canActivate`. Реалізація методу використовує об'єкти контексту, що надаються Angular; по ним він перевіряє, чи є цільовим компонентом `ShopComponent`. Якщо метод `canActivate` викликається вперше і використовуватись повинен інший компонент, то метод `Router.navigateByUrl` використовується для переходу до кореневої URL-адреси. У лістингу застосовується декоратор `@Injectable` тому, що захисники маршрутів є службами.

Захисник реєструється як служба за допомогою якості `providers` кореневого модуля та захищає кожен маршрут за допомогою властивості `canActivate`.

```
HolovniatP11Lab5 > SportShop > src > app > TS app.module.ts > ...
1  import { NgModule } from '@angular/core';
2  import { BrowserModule } from '@angular/platform-browser';
3  import { AppComponent } from './app.component';
4  import { ShopModule } from './shop/shop.module';
5  import { ShopComponent } from './shop/shop.component';
6  import { CheckoutComponent } from './shop/checkout.component';
7  import { CartDetailComponent } from './shop/cartDetail.component';
8  import { RouterModule } from '@angular/router';
9  import { ShopFirstGuard } from './shopFirst.guard';
10 @NgModule({
11   declarations: [AppComponent],
12   imports: [BrowserModule, ShopModule,
13     RouterModule.forRoot([
14       {
15         path: "shop", component: ShopComponent,
16         canActivate: [ShopFirstGuard]
17       },
18       {
19         path: "cart", component: CartDetailComponent,
20         canActivate: [ShopFirstGuard]
21       },
22       {
23         path: "checkout", component: CheckoutComponent,
24         canActivate: [ShopFirstGuard]
25       },
26       { path: "**", redirectTo: "/shop" }
27     ])],
28   providers: [ShopFirstGuard],
29   bootstrap: [AppComponent]
30 })
31 export class AppModule { }
32
```

Якщо оновити браузер після клацання на одній із кнопок Add To Cart, побачимо, що браузер автоматично перенаправляється у безпечний стан



Завершення виведення вмісту кошика

Природний процес розробки програм Angular переходить від підготовки інфраструктури (наприклад, маршрутизації URL) до функцій, видимих користувачеві. Тепер, коли в програмі реалізована підтримка навігації, настав час відобразити подання з докладним вмістом кошику.

У лістингу нижче вбудований шаблон виключається з компонента кошика, замість нього призначається зовнішній шаблон з того ж каталогу, а у конструктор додається параметр `Cart`, який буде доступний у шаблоні через властивість з ім'ям `cart`.

```
HolovniaIP11Lab5 > SportShop > src > app > shop > TS cartDetail.component.ts >
1 import { Component } from "@angular/core";
2 import { Cart } from "../model/cart.model";
3 @Component({
4   moduleId: module.id,
5   templateUrl: "cartDetail.component.html"
6 })
7 export class CartDetailComponent {
8   constructor(public cart: Cart) { }
9 }
10
```

Щоб завершити функціональність кошика, створимо файл HTML з ім'ям cartDetail.component.html в папці SportShop/src/app/shop та додамо контент:

```
HolovniaIP11Lab5 > SportShop > src > app > shop > cartDetail.component.html > div.container-fluid
1 <div class="container-fluid">
2   <div class="row">
3     <div class="bg-dark text-white p-2">
4       <span class="navbar-brand ml-2">SportShop</span>
5     </div>
6   </div>
7   <div class="row">
8     <div class="col mt-2">
9       <h2 class="text-center">Your Cart</h2>
10      <table class="table table-bordered table-striped p-2">
11        <thead>
12          <tr>
13            <th>Quantity</th>
14            <th>Product</th>
15            <th class="text-end">Price</th>
16            <th class="text-end">Subtotal</th>
17          </tr>
18        </thead>
19        <tbody>
20          <tr *ngIf="cart.lines.length == 0">
21            <td colspan="4" class="text-center">
22              Your cart is empty
23            </td>
24          </tr>
25          <tr *ngFor="let line of cart.lines">
26            <td>
27              <input type="number" class="form-control-sm" style="width:5em" [value]="line.quantity"
28                (change)="cart.updateQuantity(line.product,
29                $any($event).target.value)" />
30            </td>
31            <td>{{line.product.name}}</td>
32            <td class="text-end">
33              {{line.product.price | currency:"USD": "symbol": "2.2-2"}}
34            </td>
35            <td class="text-end">
36              {{(line.lineTotal) | currency:"USD": "symbol": "2.2-2" }}
37            </td>
38            <td class="text-center">
39              <button class="btn btn-sm btn-danger" (click)="cart.removeLine(line.product.id ?? 0)">
40                Remove
41              </button>
42            </td>
43          </tr>
44        </tbody>
45      </table>
46    </div>
47  </div>
48</div>
```

```

45 |         <tfoot>|
46 |         <tr>
47 |             <td colspan="3" class="text-end">Total:</td>
48 |             <td class="text-end">
49 |                 {{cart.cartPrice | currency:"USD":"symbol":"2.2-2"}}
50 |             </td>
51 |         </tr>
52 |     </tfoot>
53 | </table>
54 | </div>
55 | </div>
56 | <div class="row">
57 |     <div class="col">
58 |         <div class="text-center">
59 |             <button class="btn btn-primary m-1" routerLink="/shop">
60 |                 Continue Shopping
61 |             </button>
62 |             <button class="btn btn-secondary m-1" routerLink="/checkout" [disabled]="cart.lines.length == 0">
63 |                 Checkout
64 |             </button>
65 |         </div>
66 |     </div>
67 | </div>
68 | </div>

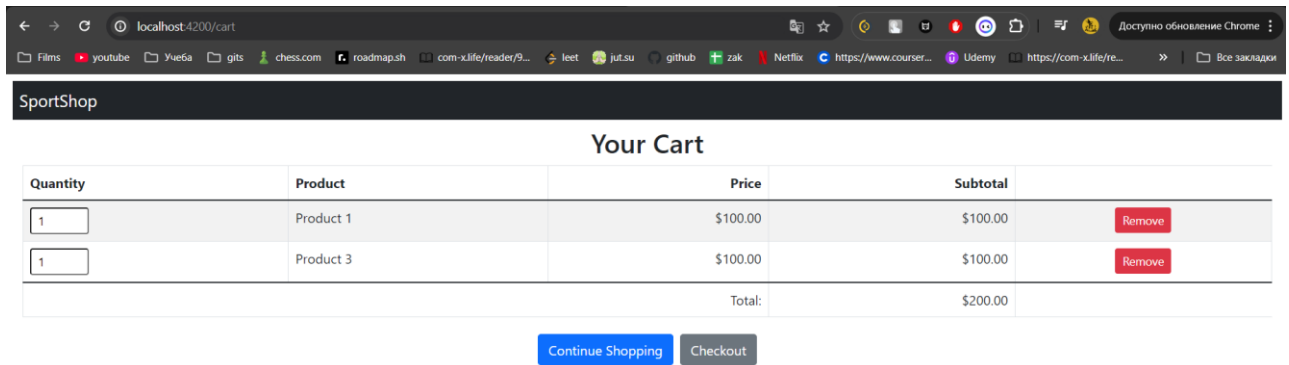
```

Шаблон виводить таблицю із товарами, обраними користувачем. Для кожного товару створюється елемент `input`, який може використовуватися для зміни кількості одиниць, та кнопка `Remove` для видалення товару із кошика. Також створюються дві навігаційні кнопки для повернення до списку товарів та переходу до оформлення замовлення.

Поєднання прив'язок даних Angular та загального об'єкта `Cart` означає, що будь-які зміни, що вносяться в кошик, призводять до негайного перерахунку ціни, а якщо ви натиснете на кнопки `Continue Shopping`, зміни відображаються в компоненті зі зведеною інформацією кошика, що відображається над списком товарів

Обробка замовлень

Процес обробки замовлень від клієнтів – найважливіший аспект інтернет-магазину. Далі у додатку буде реалізовано підтримку введення додаткової інформації користувачем та оформлення замовлення. Для простоти ми не будемо займатися подробицями взаємодії з платіжними системами, які зазвичай є службовими підсистемами, що не належать до додатків Angular.



Розширення моделі

Для опису замовлень від користувачів, створимо файл із ім'ям `order.model.ts` в папці `SportShop/src/app/model` і додамо код нижче:

```
HolovniakIP11Lab5 > SportShop > src > app > model > TS order.model.ts > Order
1  import { Injectable } from "@angular/core";
2  import { Cart } from "../cart.model";
3  @Injectable()
4  export class Order {
5      public id?: number;
6      public name?: string;
7      public address?: string;
8      public city?: string;
9      public state?: string;
10     public zip?: string;
11     public country?: string;
12     public shipped: boolean = false;
13     constructor(public cart: Cart) { }
14     clear() {
15         this.id = undefined;
16         this.name = this.address = this.city = undefined;
17         this.state = this.zip = this.country = undefined;
18         this.shipped = false;
19         this.cart.clear();
20     }
21 }
```

Клас `Order` також буде оформлено у вигляді служби; це означає, що він існуватиме лише в одному екземплярі, який спільно використовуватиметься в межах програми. Коли середовище Angular створює об'єкт `Order`, воно виявляє параметр конструктора `Cart` і надає один і той самий об'єкт `Cart`.

Оновлення репозиторію та джерела даних

Для обробки замовлень у додатку необхідно розширити репозиторій та джерело даних, щоб вони могли отримувати об'єкти Order. У коді нижче до джерела даних додається метод отримання замовлення. Оскільки джерело даних поки що залишається фіктивним, метод просто створює на підставі замовлення рядок JSON і виводить його на консоль JavaScript.

У наступному розділі ми створимо джерело даних, що використовує запити HTTP для взаємодії з REST-сумісними веб-службами.

```
4  import { Order } from "../order.model";
5
6  @Injectable()
7  export class StaticDataSource {
8      private products: Product[] = [
9          new Product(1, "Product 1", "Category 1", "Pr
10         new Product(2, "Product 2", "Category 1", "Pr
11         new Product(3, "Product 3", "Category 1", "Pr
12         new Product(4, "Product 4", "Category 1", "Pr
13         new Product(5, "Product 5", "Category 1", "Pr
14         new Product(6, "Product 6", "Category 2", "Pr
15         new Product(7, "Product 7", "Category 2", "Pr
16         new Product(8, "Product 8", "Category 2", "Pr
17         new Product(9, "Product 9", "Category 2", "Pr
18         new Product(10, "Product 10", "Category 2", "
19         new Product(11, "Product 11", "Category 3", "
20         new Product(12, "Product 12", "Category 3", "
21         new Product(13, "Product 13", "Category 3", "
22         new Product(14, "Product 14", "Category 3", "
23         new Product(15, "Product 15", "Category 3", "
24     ];
25     getProducts(): Observable<Product[]> {
26         return from([this.products]);
27     }
28     saveOrder(order: Order): Observable<Order> {
29         console.log(JSON.stringify(order));
30         return from([order]);
31     }
32 }
```

На даний момент репозиторій містить лише один метод, але його функціональність буде розширена далі, коли ми займемося створенням засобів адміністрування. Використовувати різні репозиторії для різних типів моделі у додатку необов'язково. Але рекомендується робити так, тому що один клас, який

використовується для декількох типів моделей, стає надто складним і незручним у супроводі.

```
HolovniaIP11Lab5 > SportShop > src > app > model > TS order.repository.ts > ...
1  import { Injectable } from "@angular/core";
2  import { Observable } from "rxjs";
3  import { Order } from "../order.model";
4  import { StaticDataSource } from "../static.datasource";
5  @Injectable()
6  export class OrderRepository {
7      private orders: Order[] = [];
8      constructor(private dataSource: StaticDataSource) {
9          getOrders(): Order[] {
10             return this.orders;
11         }
12         saveOrder(order: Order): Observable<Order> {
13             return this.dataSource.saveOrder(order);
14         }
15     }
}
```

Оновлення функціонального модуля

У коді нижче клас Order і новий репозиторій реєструються як служби за допомогою якості providers у функціональному модулі моделі.

```
HolovniaIP11Lab5 > SportShop > src > app > model > TS model.module.ts > ModelModule
1  import { NgModule } from "@angular/core";
2  import { ProductRepository } from "../product.repository";
3  import { StaticDataSource } from "../static.datasource";
4  import { Cart } from "../cart.model";
5  import { Order } from "../order.model";
6  import { OrderRepository } from "../order.repository";
7  @NgModule({
8      providers: [ProductRepository, StaticDataSource, Cart, Order,
9                  OrderRepository]
10 })
11 export class ModelModule {}
```

Отримання інформації про замовлення

Наступним кроком має стати отримання від користувача додаткової інформації, необхідної для завершення замовлення. Angular включає вбудовані директиви для роботи з формами HTML та перевірки їхнього вмісту.

У коді нижче відбувається підготовка компонента оформлення замовлення, перемикання на зовнішній шаблон, отримання об'єкта Order у параметрі конструктора та реалізація додаткової підтримки роботи шаблону.

```
HolovniaIP11Lab5 > SportShop > src > app > shop > TS checkout.component.ts > CheckoutComponent
1  import { Component } from "@angular/core";
2  import { NgForm } from "@angular/forms";
3  import { OrderRepository } from "../model/order.repository";
4  import { Order } from "../model/order.model";
5  @Component({
6    templateUrl: "checkout.component.html",
7    styleUrls: ["checkout.component.css"]
8  })
9  export class CheckoutComponent {
10    orderSent: boolean = false;
11    submitted: boolean = false;
12    constructor(public repository: OrderRepository,
13      public order: Order) { }
14    submitOrder(form: NgForm) {
15      this.submitted = true;
16      if (form.valid) {
17        this.repository.saveOrder(this.order).subscribe(order => {
18          this.order.clear();
19          this.orderSent = true;
20          this.submitted = false;
21        });
22      }
23    }
24  }
```

Метод submitOrder буде викликатись при відправленні даних формою, яка представляється об'єктом NgForm. Якщо дані, що містяться у формі, проходять перевірку, то об'єкт Order буде переданий методу saveOrder репозиторія, а дані в кошику та в замовленні скидаються. Властивість styleUrls декоратора @Component використовується для завдання однієї чи кількох стилевих таблиць CSS, які мають застосовуватися до контенту шаблону компонента.

Щоб надати зворотній зв'язок перевірки даних для значень, введених користувачем в елементах форми HTML, створимо файл з ім'ям checkout.component.css в папці SportShop/app/shop та визначимо стилі:

```
HolovniaIP11Lab5 > SportShop > src > app > shop > # checkout.component.css > ...
1  input.ng-dirty.ng-invalid { border: 2px solid ■ #ff0000 }
2  input.ng-dirty.ng-valid { border: 2px solid ■ #6bc502 }
3  |
```

Angular додає елементи до класів `ng-dirty`, `ng-validating-valid` для визначення статусу перевірки. Стили додають зелену рамку до елементів `input`, що містить перевірені дані, та червону рамку - до недійсних елементів. Останній фрагмент мозаїки — шаблон компонента, який надає користувачеві поля форми, необхідні для заповнення властивостей об'єкта `Order`

```
HolovniaIP11Lab5 > SportShop > src > app > shop > checkout.component.html > form.m-2
1 <div class="container-fluid">
2   <div class="row">
3     <div class="bg-dark text-white p-2">
4       <span class="navbar-brand ml-2">SportShop</span>
5     </div>
6   </div>
7 </div>
8 <div *ngIf="orderSent" class="m-2 text-center">
9   <h2>Thanks!</h2>
10  <p>Thanks for placing your order.</p>
11  <p>We'll ship your goods as soon as possible.</p>
12  <button class="btn btn-primary" routerLink="/shop">Return to Shop</button>
13 </div>
14 <form *ngIf="!orderSent" #form="ngForm" novalidate (ngSubmit)="submitOrder(form)" class="m-2">
15   <div class="form-group">
16     <label>Name</label>
17     <input class="form-control" #name="ngModel" name="name" [(ngModel)]="order.name" required />
18     <span *ngIf="submitted && name.invalid" class="text-danger">
19       Please enter your name
20     </span>
21   </div>
22   <div class="form-group">
23     <label>Address</label>
24     <input class="form-control" #address="ngModel" name="address" [(ngModel)]="order.address" required />
25     <span *ngIf="submitted && address.invalid" class="text-danger">
26       Please enter your address
27     </span>
28   </div>
```

```

29     <div class="form-group">
30         <label>City</label>
31         <input class="form-control" #city="ngModel" name="city" [(ngModel)]="order.city" required />
32         <span *ngIf="submitted && city.invalid" class="text-danger">
33             Please enter your city
34         </span>
35     </div>
36     <div class="form-group">
37         <label>State</label>
38         <input class="form-control" #state="ngModel" name="state" [(ngModel)]="order.state" required />
39         <span *ngIf="submitted && state.invalid" class="text-danger">
40             Please enter your state
41         </span>
42     </div>
43     <div class="form-group">
44         <label>Zip/Postal Code</label>
45         <input class="form-control" #zip="ngModel" name="zip" [(ngModel)]="order.zip" required />
46         <span *ngIf="submitted && zip.invalid" class="text-danger">
47             Please enter your zip/postal code
48         </span>
49     </div>
50     <div class="form-group">
51         <label>Country</label>
52         <input class="form-control" #country="ngModel" name="country" [(ngModel)]="order.country" required />
53         <span *ngIf="submitted && country.invalid" class="text-danger">
54             Please enter your country
55         </span>
56     </div>
57     <div class="text-center">
58         <button class="btn btn-secondary m-1" routerLink="/cart">Back</button>
59         <button class="btn btn-primary m-1" type="submit">Complete Order</button>
60     </div>
61 </form>

```

Елементи форми `input` у цьому шаблоні використовують засоби Angular, щоб переконатися, що користувач ввів значення для кожного поля, і надають візуальний зворотний зв'язок, якщо користувач клацнув на кнопці `Complete Order` без заповнення форми. Одна частина цього зворотного зв'язку забезпечується застосуванням стилів, а інша – елементами `span`, які залишаються прихованими, поки користувач не спробує надіслати недійсну форму. Перевірка наявності обов'язкових значень - лише один із способів перевірки полів форми в Angular. Розробник також може легко реалізувати власну, нестандартну перевірку даних.

Щоб побачити, як працює цей процес, почнемо зі списку товарів та клацнемо на одній із кнопок `Add To Cart`, щоб додати товар у кошик. Далі на кнопку `Checkout`; на екрані з'являється форма HTML. Спробуємо натиснути на кнопку `Complete Order`, не вводячи текст у жодному полі, і отримаємо повідомлення про помилку перевірки даних.

Заповнимо форму та клацнемо на кнопку `Complete Order`; з'явиться підтверджуюче повідомлення

SportShop

Name

Address

City

State

Zip/Postal Code

Country

Back Complete Order

SportShop

Thanks!

Thanks for placing your order.

We'll ship your goods as soon as possible.

Return to Shop

На консолі JavaScript у браузері виводиться подання замовлення у форматі JSON:

```
{
  "cart": {
    "lines": [
      {
        "product": {
          "id": 1,
          "name": "Product 1",
          "category": "Category 1",
          "description": "Product 1 (Category 1)",
          "price": 100,
          "quantity": 1
        },
        "product": {
          "id": 4,
          "name": "Product 4",
          "category": "Category 1",
          "description": "Product 4 (Category 1)",
          "price": 100,
          "quantity": 1
        }
      ],
      "itemCount": 2,
      "cartPrice": 200,
      "shipped": false,
      "name": "Holovnia Oleksandr",
      "address": "5Avenu",
      "city": "Kyiv",
      "state": "dadas",
      "zip": "12321312",
      "country": "Ukraine"
    }
  }
}
```

Використання REST-сумісної веб-служби

Тепер, коли базова функціональність SportShop підготовлена, настав час замінити фіктивне джерело даних іншим, яке отримує дані з REST-сумісної веб-служби, яка була створена під час підготовки проекту раніше.

Щоб створити джерело даних, створимо файл `rest.datasource.ts` в папці `SportShop/src/app/model` та додамо наступний код:

```

HolovniatP11Lab5 > SportShop > src > app > model > TS rest.datasource.ts > ...
1  import { Injectable } from "@angular/core";
2  import { HttpClient } from "@angular/common/http";
3  import { Observable } from "rxjs";
4  import { Product } from "../product.model";
5  import { Order } from "../order.model";
6  const PROTOCOL = "http";
7  const PORT = 3500;
8  @Injectable()
9  export class RestDataSource {
10     baseUrl: string;
11     constructor(private http: HttpClient) {
12         this.baseUrl = `${PROTOCOL}://${location.hostname}:${PORT}/`;
13     }
14     getProducts(): Observable<Product[]> {
15         return this.http.get<Product[]>(this.baseUrl + "products");
16     }
17     saveOrder(order: Order): Observable<Order> {
18         return this.http.post<Order>(this.baseUrl + "orders", order);
19     }
20 }
21

```

Angular надає вбудовану службу `HttpClient`, яка використовується для створення запитів HTTP. Конструктор `RestDataSource` отримує службу `Http` та використовує глобальний об'єкт `location`, наданий браузером, для визначення URL-адреси для надсилання запитів; він відповідає порту 3500 на тому хості, з якого було завантажено додаток. Методи, що визначаються класом `RestDataSource`, відповідають методам, що визначаються статичним джерелом даних, та реалізуються викликом методу `sendRequest`, який використовує службу `HttpClient`.

При отриманні даних через HTTP може статися так, що мережевий затор або підвищене навантаження на сервер затримують обробку запиту і користувач буде дивитися на програму, яка не отримала даних. Щоб такого не було треба відповідним чином настроїти систему маршрутизації для запобігання таким проблемам.

Застосування джерела даних

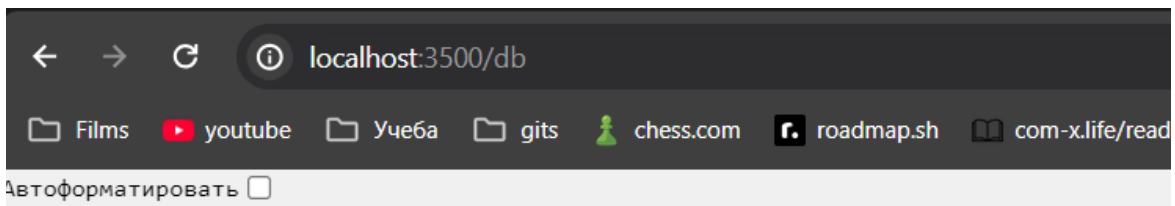
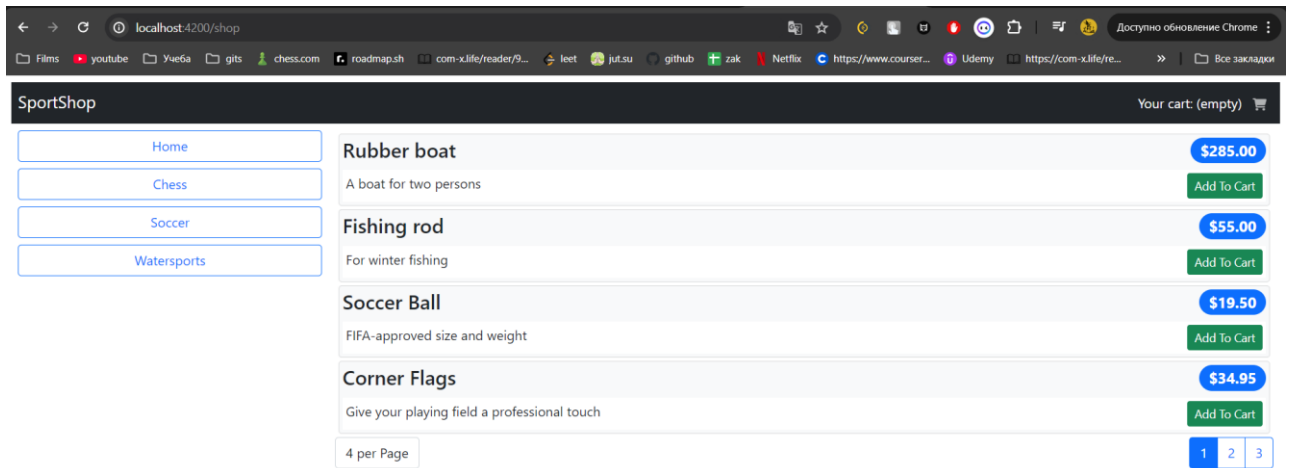
Тепер застосуємо REST-сумісне джерело даних. Для цього перебудуємо додаток, щоб перехід з фіктивних даних на дані REST здійснювався змінами лише в одному файлі. У коді нижче поведінка джерела даних змінюється у функціональному модулі моделі.

```
HolovniaIP11Lab5 > SportShop > src > app > model > TS model.module.ts > ...
1  import { NgModule } from "@angular/core";
2  import { ProductRepository } from "../product.repository";
3  import { StaticDataSource } from "../static.datasource";
4  import { Cart } from "../cart.model";
5  import { Order } from "../order.model";
6  import { OrderRepository } from "../order.repository";
7  import { RestDataSource } from "../rest.datasource";
8  import { HttpClientModule } from "@angular/common/http";
9  @NgModule({
10     imports: [HttpClientModule],
11     providers: [ProductRepository, StaticDataSource, Cart, Order,
12                OrderRepository,
13                { provide: StaticDataSource, useClass: RestDataSource } ]
14 })
15 export class ModelModule { }
```

Властивість `imports` використовується для оголошення залежності від функціонального модуля `HttpModule`, який надає службу `Http`. Зміна у властивості `providers` повідомляє Angular, що коли потрібно створити екземпляр класу з параметром конструктора `StaticDataSource` замість нього слід використовувати `RestDataSource`.

Так як обидва об'єкти визначають однакові методи, завдяки динамічній системі типів JavaScript заміна повинна пройти гладко. Після того, як усі зміни будуть збережені, а браузер перезавантажить програму, фіктивні дані замінюються даними, отриманими через HTTP. Якщо пройти процедуру вибору товарів та оформлення замовлення, ви зможете переконатися, що джерело даних записало замовлення до веб-служби.

Перейдемо за наступною URL-адресою:



```
{
  "products": [
    {
      "id": 1,
      "name": "Rubber boat",
      "category": "Watersports",
      "description": "A boat for two persons",
      "price": 285
    },
    {
      "id": 2,
      "name": "Fishing rod",
      "category": "Watersports",
      "description": "For winter fishing",
      "price": 55
    },
    {
      "id": 3,
      "name": "Soccer Ball",
      "category": "Soccer",
      "description": "FIFA-approved size and weight",
      "price": 19.5
    },
    {
      "id": 4,
      "name": "Corner Flags",
      "category": "Soccer",
      "description": "Give your playing field a professional touch",
      "price": 34.95
    },
    {
      "id": 5,
      "name": "Stadium",
      "category": "Soccer",
      "description": "Flat-packed 35,000-seat stadium",
      "price": 79500
    },
    {
      "id": 6,
      ..
    }
  ]
}
```


В цій роботі ми продовжили розширювати функціональність програми SportShop: до програми була додана підтримка кошика для вибору товарів користувачем та процесу оформлення замовлення, що завершує процес покупки. Також у цій частині роботи фіктивне джерело даних було замінено джерелом, що надсилає запити HTTP RESTсумісній веб-службі.

Посилання на додатки:

<https://holovniaip11sportshop1.web.app/>

Висновок:

Під час виконання комп'ютерного практикуму я навчився створювати Angular-додатки, які містять сервіси, pipes, директиви та використовують бібліотеку RxJS. Створив проект, що складається з двох частин:

Встановив та налаштував засоби розробника, створив кореневі структурні блоки для проекту (модель даних, фіктивне джерело даних, репозиторій моделі, сховище, компоненти магазину та шаблони). Вивів дані фіктивної моделі даних на головну сторінку магазину, реалізував розбивку на сторінки та фільтрацію товарів за категоріями. Створив нестандартну директиву для пагінації.

Додаток, отриманий в результаті виконання Частини 1 розгорнутий на платформі Firebase.

Далі розробив додаткову логіку в додатку «SportShop» для вибору товарів та оформлення замовлень. Реалізував підтримку кошика для вибору товарів користувачем та процесу оформлення замовлення. Фіктивне джерело даних у проекті замінив джерелом, що надсилає запити до HTTP REST-сумісної веб-служби.