Лабораторне заняття №8. Створення проекту «SportShop». Вибір товарів та оформлення замовлення

Мета: Навчитися створювати та використовувати сервіси та pipes у реальному проекті Angular. Навчитися використовувати HTTP REST-сумісну веб-службу в проектах Angular.

Завдання:

I) Частина 1: Створення проекту «SportShop». Створити проект Angular під назвою «SportShop». Встановити та налаштувати засоби розробника, створити кореневі структурні блоки для проекту (модель даних, джерело даних, репозиторій моделі, сховище, компоненти магазина та шаблони).

II) Частина 2: Вибір товарів та оформлення замовлення. Розширити функціональність проекту «SportShop», реалізувавши 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.

III) Зробити звіт по роботі.

IV) Angular-додаток SportShop (Частина 1) розгорнути на платформі Firebase у проекті з ім'ям «ПрізвищеГрупаLaba8», наприклад «KovalenkoIP01Laba8».

I) Частина 1: Створення проекту «SportShop». Створення проекту Angular під назвою «SportShop». Встановлення та налаштування засобів розробника, створення кореневих структурних блоків для проекту (модель даних, джерело даних, репозиторій моделі, сховище, компоненти магазина та шаблони).

У нашому додатку, який називається SportShop, буде використано класичний підхід, що реалізується в інтернет-магазинах по всьому світу. Ми створимо електронний каталог товарів, який клієнти можуть переглядати за категоріями та сторінками; кошик, до якого користувачі можуть додавати чи видаляти товари; та процедуру оформлення замовлення, в якій клієнти зможуть вводити дані для доставки та розміщувати замовлення. Також буде створено адміністративну панель, яка включає операції створення, читання, оновлення та видалення (CRUD) для управління каталогом, та організуємо систему захисту, щоб зміни могли вноситись лише адміністраторами з підтвердженими повноваженнями. В кінці необхідно буде підготувати та розгорнути додаток Angular.

Аспекти інтеграції із зовнішніми системами (наприклад, базами даних) по можливості спрощені, інші (наприклад, обробка платежів) повністю опущені.

У Angular багато взаємодіючих компонентів. Програма SportShop повинна показати, як вони взаємодіють.

Підготовка проекту

Щоб створити проект SportShop, відкрийте командний рядок, перейдіть у зручне місце та виконайте наступну команду:

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

Пакет angular-cli створює новий проект для розробки Angular із файлами конфігурації, тимчасовим контентом та іншими засобами розробника. Процес підготовки проекту може зайняти деякий час через завантаження та встановлення великої кількості пакетів NPM.

Створення структури папок

Відправною точкою для будь-якої програми Angular стає створення структури папок. Команда ng new створює структуру, в якій усі файли програми знаходяться в папці src, а файли Angular - у папці src/app. Щоб визначити додаткову структуру проекту, створіть папки, наведені в таблиці. 2.1.

Таблиця 1.1. Додаткові папки, необхідні для проекту SportShop

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

Встановлення додаткових пакетів NPM

Додаткові пакети потрібні для проекту SportsShope на додаток до основних пакетів Angular. Виконайте наведені нижче команди, щоб перейти до папки SportsShope та додайте необхідні пакети:

```
cd SportsShope
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
```

Важливо використовувати номери версій, указані в списку. Деякі з пакетів встановлено за допомогою аргументу --save-dev, який вказує, що вони використовуються під час розробки та не будуть частиною додатку SportsShope.

Додавання CSS стилів до проекту

Після встановлення пакетів виконайте команду, показану в лістингу 1-1, у папці SportsShope, щоб додати до проекту таблицю стилів Bootstrap CSS.

Лістинг 1-1. Зміна конфігурації програми

```
ng config projects.SportsShope.architect.build.options.styles \
'["src/styles.css",'\
'"node_modules/@fortawesome/fontawesome-free/css/all.min.css",'\
'"node_modules/bootstrap/dist/css/bootstrap.min.css"]'
```

Якщо ви використовуєте Windows, скористайтеся підказкою PowerShell, щоб запустити команду, наведену в лістингу 1-2 у зазначену папку.

Лістинг 1-2. Зміна конфігурації програми за допомогою PowerShell

```
ng config projects.SportsShope.architect.build.options.styles `
'[""src/styles.css"",
""node_modules/@fortawesome/fontawesome-free/css/all.min.css"",
""node_modules/bootstrap/dist/css/bootstrap.min.css""]'
```

Виконайте команду, наведену в лістингу 1-3, у папці SportsShope, щоб переконатися, що конфігурація була змінена правильно.

Лістинг 1-3. Перевірка змін конфігурації

```
ng config projects.SportsShope.architect.build.options.styles
```

Результат цієї команди повинен містити три файли, перелічені в лістингу 1-1 і лістингу 1-2. Якщо все зроблено правильно, то результат має бути таким:

```
[
"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 SportsShope використовуватиме асинхронні HTTP-запити, щоб отримати дані моделі, надані REST-сумісною веб-службою. До проекту було додано пакет json-server. Це чудовий пакет для створення веб-сервісів із даних JSON або коду JavaScript. Додайте оператор, показаний у лістингу 1-4, до розділу сценаріїв у файлі package.json, щоб пакет json-server можна було запустити з командного рядка.

Лістинг 1-4. Додавання сценарію у файл package.json у папці SportShope

```
...
"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, додайте файл під назвою data.js у папку SportShope і додайте код, показаний у лістингу 1-5, який забезпечить доступність тих самих даних кожного разу, коли запускається пакет json-server.

Лістинг 1-5. Вміст файлу data.js у папці SportShope

```
module.exports = function () {
return {
products: [
{ id: 1, name: "Kayak", category: "Watersports",
description: "A boat for one person", price: 275 },
{ id: 2, name: "Lifejacket", category: "Watersports",
description: "Protective and fashionable", price: 48.95 },
{ id: 3, name: "Soccer Ball", category: "Soccer",
description: "FIFA-approved size and weight", price: 19.50 },
{ 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, name: "Thinking Cap", category: "Chess",
description: "Improve brain efficiency by 75%", price: 16 },
{ id: 7, name: "Unsteady Chair", category: "Chess",
description: "Secretly give your opponent a disadvantage",
price: 29.95 },
{ id: 8, name: "Human Chess Board", category: "Chess",
description: "A fun game for the family", price: 75 },
{ id: 9, name: "Bling King", category: "Chess",
description: "Gold-plated, diamond-studded King", price: 1200 }
],
orders: []
}
}
```

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 (лістинг 1.6).

Лістинг 1.6. Вміст файлу authMiddleware.js у папці SportShop

```
const jwt = require("jsonwebtoken");
const APP_SECRET = "myappsecret";
const USERNAME = "admin";
const PASSWORD = "secret";
const mappings = {
get: ["/api/orders", "/orders"],
post: ["/api/products", "/products", "/api/categories", "/categories"]
}
function requiresAuth(method, url) {
return (mappings[method.toLowerCase()] || [])
.find(p => url.startsWith(p)) !== undefined;
}
module.exports = function (req, res, next) {
if (req.url.endsWith("/login") && req.method == "POST") {
if (req.body && req.body.name == USERNAME && req.body.password == PASSWORD) {
let token = jwt.sign({ data: USERNAME, expiresIn: "1h" }, APP_SECRET);
res.json({ success: true, token: token });
} else {
res.json({ success: false });
}
res.end();
return;
} else if (requiresAuth(req.method, req.url)) {
let token = req.headers["authorization"] || "";
if (token.startsWith("Bearer<")) {
token = token.substring(7, token.length - 1);
try {
jwt.verify(token, APP_SECRET);
next();
return;
} catch (err) { }
}
res.statusCode = 401;
res.end();
return;
}
next();
}
```

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. Відредагуйте файл index.html в папці SportShop/src та додайте до нього елементи з лістингу 1.7.

Лістинг 1.7. Вміст файлу index.html у папці SportShop/src

```
<!doctype html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>SportShope</title>
<base href="/">
<meta name="viewport" content="width=device-width, initial-scale=1">
<link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body class="p-2">
<app>SportShope Will Go Here</app>
</body>
</html>
```

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

Запуск прикладу

Переконайтеся, що всі файли були збережені, і виконайте наступну команду з папки SportShop:

```
ng serve --open
```

Команда запускає ланцюжок інструментів розробки, який налаштований angular-cli, і автоматично перекомпільовує та перебудовує код та файли контенту з папки src при виявленні будь-яких змін. Відкривається нове вікно браузера з контентом на рис. 1.1.

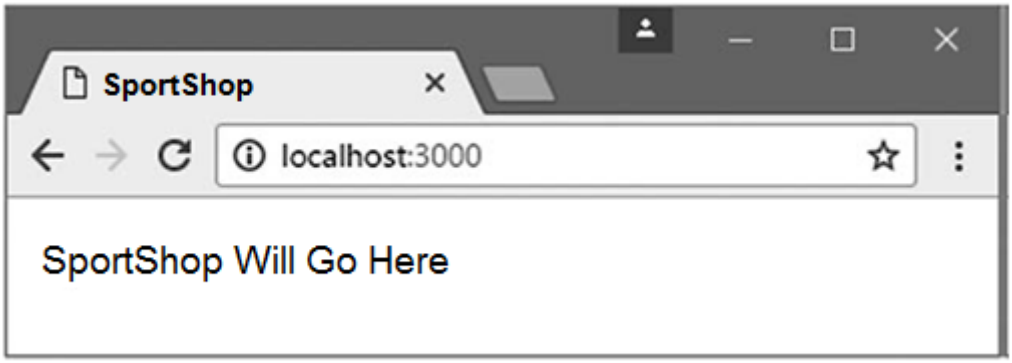

Рис. 1.1. Запуск програми

Веб-сервер розробки запускатиметься на порті 4200, тому URL-адреса програми буде http://localhost:4200. Вам не потрібно включати ім’я документа HTML, оскільки index.html є файл за замовчуванням, яким сервер відповідає.

Запуск REST-сумісної веб-служби

Щоб запустити REST-сумісну веб-службу, відкрийте нове вікно командного рядка, перейдіть до папки SportShop та виконайте наступну команду:

```
npm run json
```

REST-сумісна веб-служба налаштована для роботи на порті 3500. Щоб протестувати запит до веб-служби, введіть у браузері URL http://localhost:3500/products/1. Браузер виводить подання одного з товарів, визначених у лістингу 1.5, у форматі JSON:

```
{
"id": 1,
"name": "Kayak",
"category": "Watersports",
"description": "A boat for one person",
"price": 275
}
```

Підготовка проекту Angular

Кожен проект Angular вимагає певної базової підготовки - просто для переходу в стан, в якому програма завантажується та запускається браузером. 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. Відредагуйте файл app.component.ts в папці SportShop/src/app та включіть в нього код з лістингу 1.8.

Лістинг 1.8. Вміст файлу app.component.ts у папці SportShop/src/app

```
import { Component } from "@angular/core";
@Component({
selector: "app",
template: `<div class="bg-success p-2 text-center text-white">
This is SportShope
</div>`
})
```

```
export class AppComponent { }
```

Декоратор @Component повідомляє Angular, що клас AppComponent є компонентом, а його властивості описують застосування цього компонента. Компонент може мати великий набір властивостей, але три властивості, наведені в лістингу 1.8, є основними і часто застосовуваними на практиці. Властивість selector повідомляє Angular, як слід застосовувати компонент у документі HTML, а властивість template визначає контент, який відображатиметься компонентом. Компоненти можуть визначати вбудовані шаблони, як у даному випадку, або використовувати зовнішні HTML файли, які спрощують управління складним контентом.

Клас AppComponent не містить коду, тому що кореневий компонент у проекті Angular існує тільки для керування контентом, що відображається для користувача. На початковій стадії ми керуватимемо контентом, що відображається кореневим компонентом, вручну. Далі в інших лабораторних роботах буде представлений механізм маршрутизації URL для автоматичної адаптації контенту в залежності від дій користувача.

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. Перевірте файл із таким ім'ям у папці SportShop/src/app. Цей файл повинен включати код з лістингу 1.9.

Лістинг 1.9. Вміст файлу app.module.ts у папці SportShop/src/app

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { AppComponent } from "./app.component";
@NgModule({
imports: [BrowserModule],
declarations: [AppComponent],
bootstrap: [AppComponent]
})
export class AppModule { }
```

Якщо при створенні проекту ви вказували необхідність підтримки Routing, то файл app.module.ts також буде включати модуль AppRoutingModule.

За аналогією з кореневим 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 веб-програми.

Властивість declarations наказує Angular завантажити кореневий компонент, а властивість bootstrap повідомляє, що кореневим 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. Створіть файл з ім'ям main.ts в папці SportShop/src/app і додайте код з лістингу 1.10.

Лістинг 1.10.Вміст файлу main.ts у папці SportShop/src

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app/app.module';
import { environment } from './environments/environment';
if (environment.production) {
enableProdMode();
}
platformBrowserDynamic().bootstrapModule(AppModule);
```

Інструменти розробки виявляють зміни у файлі проекту, компілюють файли з кодом та автоматично перезавантажують браузер із виведенням контенту, зображеного на рис. 1.2.

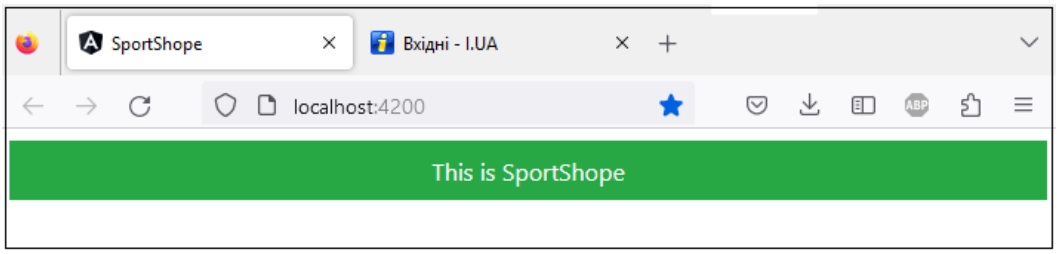

Рис. 1.2. Запуск програми SportShop

Переглядаючи модель DOM у браузері, ви побачите, що тимчасовий контент із шаблону кореневого компонента був вставлений між початковим та кінцевим тегами елемента app:

```
<body class="p-2">
<app>
<div class="bg-success p-2 text-center text-white
">
This is SportShop
</div>
</app>
</body>
```

Початок роботи над моделлю даних

Роботу над будь-яким новим проектом найкраще розпочинати з моделі даних. Щоб якнайшвидше продемонструвати деякі можливості Angular у дії, ми почнемо з реалізації базової функціональності на фіктивних даних. Потім ці дані будуть використані для створення інтерфейсної частини, а далі ми повернемося до моделі даних та зв'яжемо її з REST-сумісною веб-службою.

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 інтернет-магазині, 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. Створіть файл з ім'ям product.model.ts в папці SportShop/src/app/model та включіть код з лістингу 1.11.

Лістинг 1.11. Вміст файлу product.model.ts із папки SportShop/src/app/model

```
export class Product {
constructor(
public id?: number,
public name?: string,
public category?: string,
public description?: string,
public price?: number) { }
}
```

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 1.5. Знаки запитання (?) за іменами параметрів вказують, що це необов'язкові параметри, які можуть бути опущені під час створення нових об'єктів з використанням классу 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.

Створіть файл static.datasource.ts в папці SportShop/src/app/model та включіть визначення класу з лістингу 1.12.

Лістинг 1.12. Вміст файлу static.datasource.ts із папки SportShop/src/app/model

```
import { Injectable } from "@angular/core";
import { Product } from "./product.model";
import { Observable } from "rxjs/Observable";
import "rxjs/add/observable/from";
@Injectable()
export class StaticDataSource {
private products: Product[] = [
new Product(1, "Product 1", "Category 1", "Product 1 (Category 1)", 100),
new Product(2, "Product 2", "Category 1", "Product 2 (Category 1)", 100),
new Product(3, "Product 3", "Category 1", "Product 3 (Category 1)", 100),
new Product(4, "Product 4", "Category 1", "Product 4 (Category 1)", 100),
new Product(5, "Product 5", "Category 1", "Product 5 (Category 1)", 100),
new Product(6, "Product 6", "Category 2", "Product 6 (Category 2)", 100),
new Product(7, "Product 7", "Category 2", "Product 7 (Category 2)", 100),
new Product(8, "Product 8", "Category 2", "Product 8 (Category 2)", 100),
new Product(9, "Product 9", "Category 2", "Product 9 (Category 2)", 100),
new Product(10, "Product 10", "Category 2", "Product 10 (Category 2)", 100),
new Product(11, "Product 11", "Category 3", "Product 11 (Category 3)", 100),
new Product(12, "Product 12", "Category 3", "Product 12 (Category 3)", 100),
new Product(13, "Product 13", "Category 3", "Product 13 (Category 3)", 100),
new Product(14, "Product 14", "Category 3", "Product 14 (Category 3)", 100),
new Product(15, "Product 15", "Category 3", "Product 15 (Category 3)", 100),
];

getProducts(): Observable<Product[]> {
return Observable.from([this.products]);
}
}
```

Клас StaticDataSource визначає метод з ім'ям getProducts, що повертає фіктивні дані. Виклик методу getProducts повертає результат Observable<Product[]> - 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 запитами HTTP; саме тому метод getProducts повертає Observable<Product[]> замість повернення даних - простого синхронного або з використанням Promise.

Декоратор @Injectable застосовується до классу 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. Створіть файл product.repository.ts в папці SportShop/src/app/model та визначте в ньому клас з лістингу 1.13.

Лістинг 1.13. Вміст файлу product.repository.ts із папки SportShop/app/model

```
import { Injectable } from "@angular/core";
import { Product } from "./product.model";
import { StaticDataSource } from "./static.datasource";
@Injectable()
export class ProductRepository {
private products: Product[] = [];
private categories: string[] = [];
constructor(private dataSource: StaticDataSource) {
dataSource.getProducts().subscribe(data => {
this.products = data;
this.categories = data.map(p => p.category ?? "(None)")
.filter((c, index, array) => array.indexOf(c) == index).sort();
});
}
getProducts(category?: string): Product[] {
return this.products
.filter(p => category == undefined || category == p.category);
}
getProduct(id: number): Product | undefined {
```

```
return this.products.find(p => p.id == id);
}
getCategories(): string[] {
return this.categories;
}
}
```

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. Створіть файл з ім'ям model.module.ts в папці SportShop/app/model та визначте клас, наведений у лістингу 1.14.

Лістинг 1.14. Вміст файлу model.module.ts із папки SportShop/src/app/model

```
import { NgModule } from "@angular/core";
import { ProductRepository } from "./product.repository";
import { StaticDataSource } from "./static.datasource";
@NgModule({
providers: [ProductRepository, StaticDataSource]
})
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 (рис. 1.3).

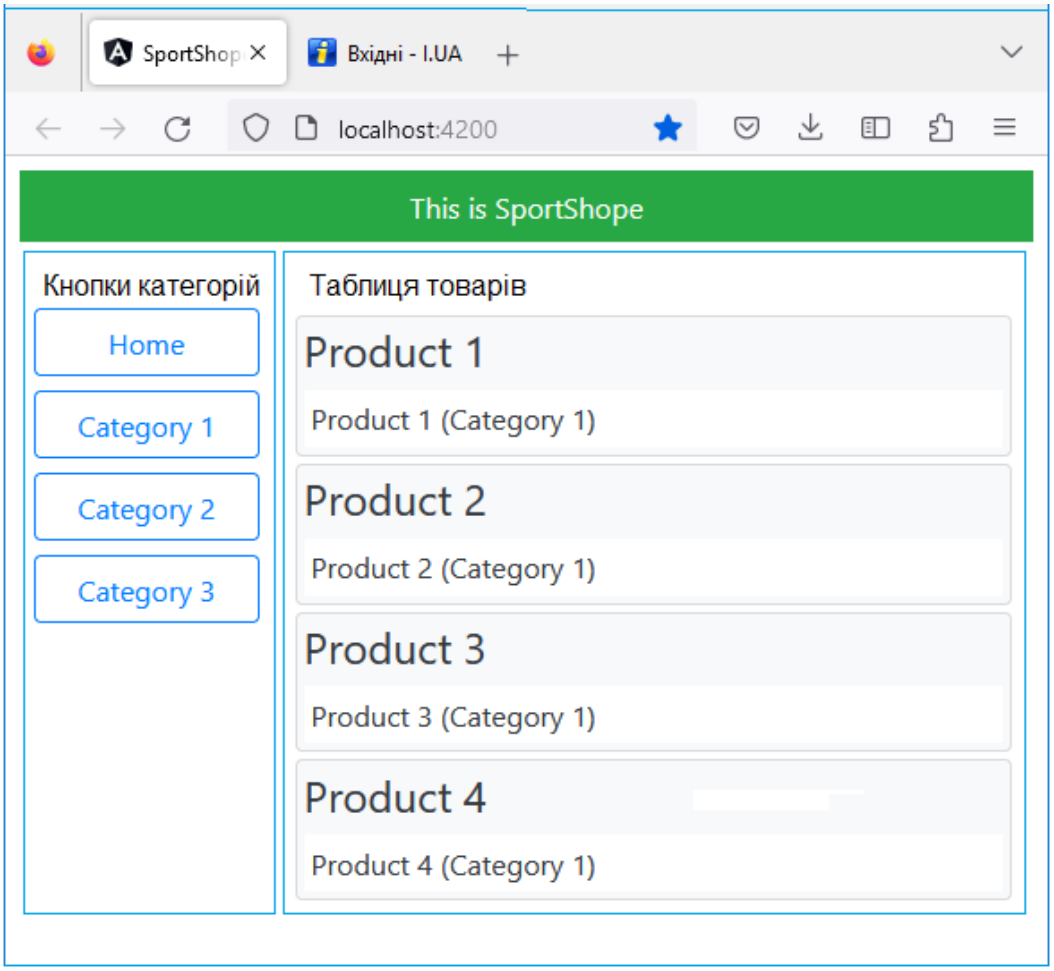

Рис. 1.3. Базова структура магазину

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. Створіть файл з ім'ям store.component.ts в папці SportShop/src/app/store та додайте визначення класу з лістингу 1.15.

Лістинг 1.15. Вміст файлу store.component.ts із папки SportShop/src/app/store

```
import { Component } from "@angular/core";
import { Product } from "../model/product.model";
import { ProductRepository } from "../model/product.repository";
@Component({
selector: "store",
templateUrl: "store.component.html"
})
export class StoreComponent {
constructor(private repository: ProductRepository) { }
get products(): Product[] {
return this.repository.getProducts();
```

```
}
get categories(): string[] {
return this.repository.getCategories();
}
}
```

До класу StoreComponent 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 store) і де знаходиться шаблон компонента (у файлі з ім'ям store.component.html).

Клас StoreComponent надає логіку, яка забезпечує отримання контенту шаблону.

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, створіть файл store.component.html в папці SportShop/src/app/store та додайте контент HTML з лістингу 1.16.

Лістинг 1.16. Вміст файлу store.component.html із папки SportShop/src/app/store

```
<div class="container-fluid">
<div class="row">
<div class="bg-dark text-white p-2">
<span class="navbar-brand ml-2">SPORTS STORE</span>
</div>
</div>
<div class="row text-white">
<div class="col-3 bg-info p-2">
{{categories.length}} Categories
</div>
<div class="col-9 bg-success p-2">
{{products.length}} Products
</div>
</div>
</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, створіть файл з ім'ям store.module.ts в папці SportShop/src/app/store та додайте код з лістингу 1.17.

Лістинг 1.17. Вміст файлу store.module.ts із папки SportShop/src/app/store

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { StoreComponent } from "./store.component";
@NgModule({
imports: [ModelModule, BrowserModule, FormsModule],
declarations: [StoreComponent],
exports: [StoreComponent]
})
export class StoreModule { }
```

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 StoreComponent, 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. У лістингу 1.18 представлені зміни шаблону кореневого компонента.

Лістинг 1.18. Додавання компонента до файлу app.component.ts

```
import { Component } від "@angular/core";
@Component({
selector: "app",
template: "<store></store>"
})
export class AppComponent { }
```

Елемент store замінює попередній контент у шаблоні кореневого компонента та відповідає значенню властивості selector декоратора @Component у лістингу 1.15. У лістингу 1.19 показані зміни, які необхідно внести до кореневого модулю, щоб середовище Angular завантажувало функціональний модуль з функціональністю магазину.

Лістинг 1.19. Імпортування функціональних модулів у файл app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { StoreModule } from "./store/store.module";
@NgModule({
declarations: [AppComponent],
imports: [BrowserModule, StoreModule],
providers: [],
bootstrap: [AppComponent]
})
export class AppModule { }
```

Коли ви збережете зміни в кореневому модулі, Angular матиме всю інформацію, необхідну для завантаження програми та відображення контенту з модуля магазину (рис. 1.4).

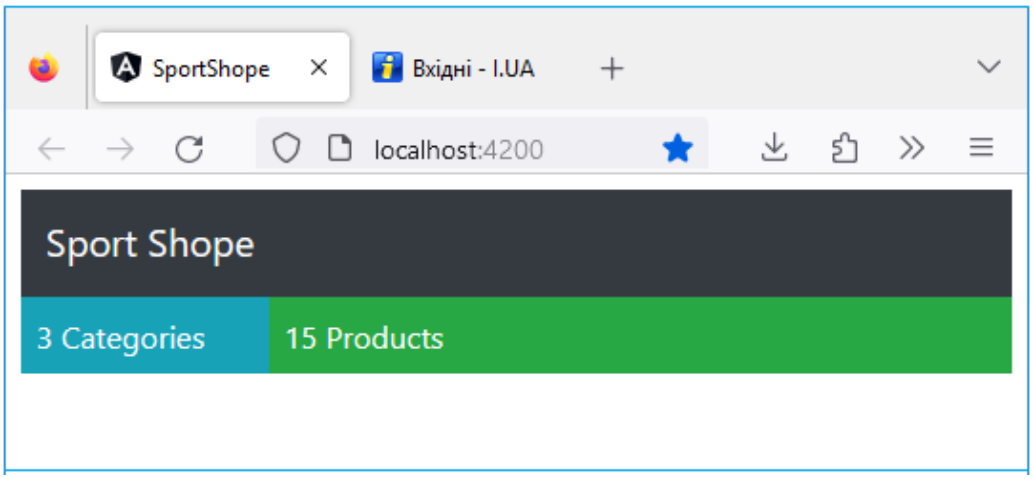

Рис. 1.4. Базова функціональність програми SportShop

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. У лістингу 1.20 у

шаблон компонента магазину додаються елементи HTML з прив'язками даних, що генерують контент для кожного товару, що надається компонентом.

Лістинг 1.20. Додавання елементів до файлу store.component.html

```
<div class="container-fluid">
<div class="row">
<div class="bg-dark text-white p-2">
<span class="navbar-brand ml-2">SPORTS STORE</span>
</div>
</div>
<div class="row text-white">
<div class="col-3 bg-info p-2">
{{categories.length}} Categories
</div>
<div class="col-9 p-2 text-dark">
<div *ngFor="let product of products" class="card m-1 p-1 bg-light">
<h4>
{{product.name}}
<span class="badge rounded-pill bg-primary" style="float:right">
{{ product.price | currency:"USD":"symbol":"2.2-2" }}
</span>
</h4>
<div class="card-text bg-white p-1">{{product.description}}</div>
</div>
</div>
</div>
</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 card-outline-primary">
...
```

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:

```
...
```

{{product.description}}

...

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":true:"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 (рис. 1.5).

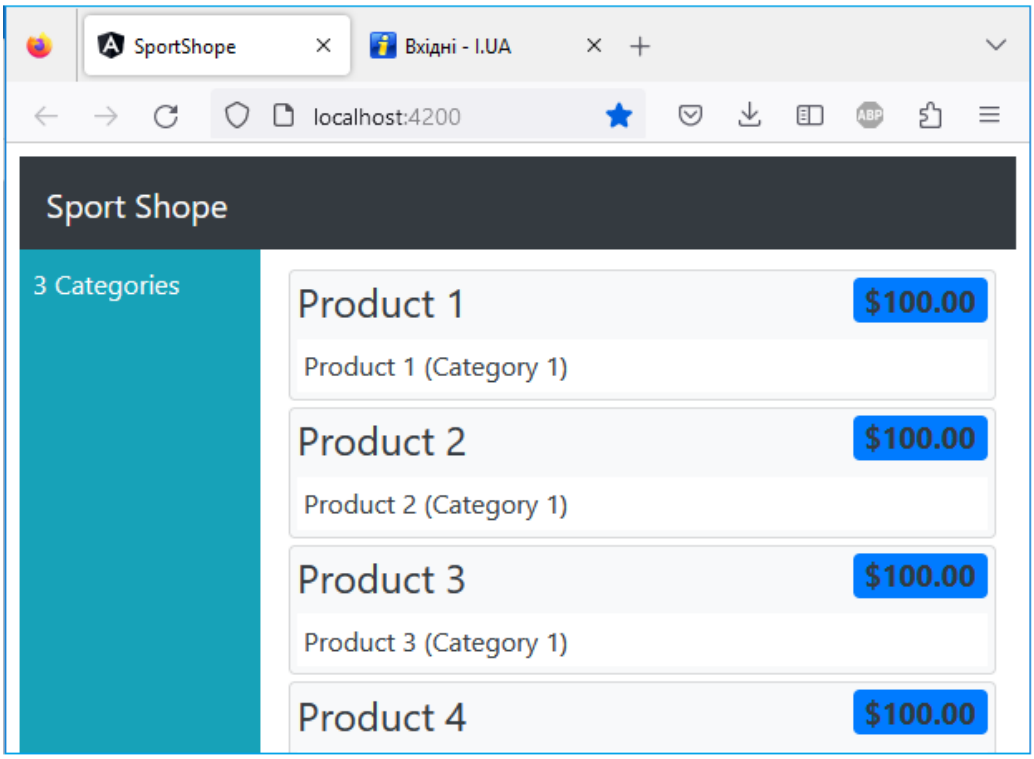

Рис. 1.5. Виведення інформації про товар

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 (листинг 1.21).

Лістинг 1.21. Додавання фільтрації за категоріями у файл store.component.ts

```
import { Component } from "@angular/core";
import { Product } from "../model/product.model";
import { ProductRepository } from "../model/product.repository";
@Component({
selector: "store",
templateUrl: "store.component.html"
})
export class StoreComponent {
selectedCategory: string | undefined;
constructor(private repository: ProductRepository) { }
get products(): Product[] {
return this.repository.getProducts(this.selectedCategory);
}
get categories(): string[] {
return this.repository.getCategories();
}
changeCategory(newCategory?: string) {
this.selectedCategory = newCategory;
}
}
```

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.

У лістингу 1.22 представлені відповідні зміни шаблону компонента. Шаблон повинен відображати набір кнопок для зміни вибраної категорії та показувати, яка категорія обрана зараз.

Лістинг 1.22. Додавання кнопок категорій у файлі store.components.html

```
<div class="container-fluid">
<div class="row">
<div class="bg-dark text-white p-2">
<span class="navbar-brand ml-2">SPORTS STORE</span>
</div>
</div>
<div class="row text-white">
<div class="col-3 p-2">
<div class="d-grid gap-2">
<button class="btn btn-outline-primary" (click)="changeCategory()">
Home
</button>
<button *ngFor="let cat of categories"
class="btn btn-outline-primary"
```

```
[class.active]="cat == selectedCategory"
(click)="changeCategory(cat)">
{{cat}}
</button>
</div>
</div>
<div class="col-9 p-2 text-dark">
<div *ngFor="let product of products" class="card m-1 p-1 bg-light">
<h4>
{{product.name}}
<span class="badge rounded-pill bg-primary" style="float:right">
{{ product.price | currency:"USD":"symbol":"2.2-2" }}
</span>
</h4>
<div class="card-text bg-white p-1">{{product.description}}</div>
</div>
</div>
</div>
</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'язка 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 (рис. 1.6).

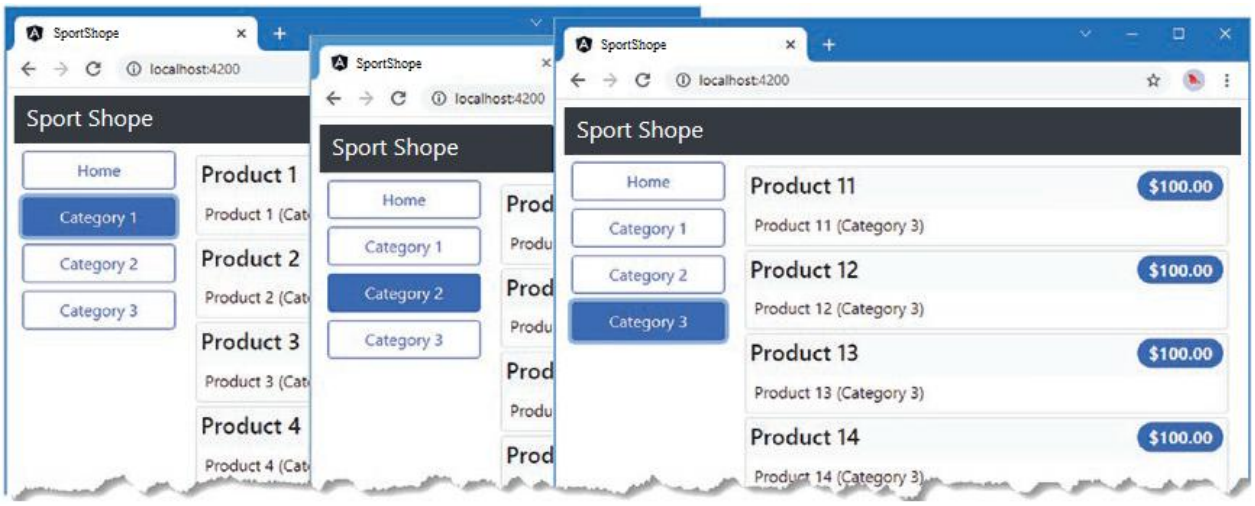

Рис. 1.6. Вибір категорії товарів

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.

У лістингу 1.23 у компонент магазину вносяться зміни, щоб у ньому зберігалася поточна сторінка і кількість елементів на сторінці.

Лістинг 1.23. Додавання розбивки на сторінки до файлу store.component.ts

```
import { Component } from "@angular/core";
import { Product } from "../model/product.model";
import { ProductRepository } from "../model/product.repository";
@Component({
selector: "store",
templateUrl: "store.component.html"
})
export class StoreComponent {
selectedCategory: string | undefined;
productsPerPage = 4;
selectedPage = 1;
constructor(private repository: ProductRepository) { }
get products(): Product[] {
let pageIndex = (this.selectedPage - 1) * this.productsPerPage
return this.repository.getProducts(this.selectedCategory)
.slice(pageIndex, pageIndex + this.productsPerPage);
}
get categories(): string[] {
return this.repository.getCategories();
}
```

```
changeCategory(newCategory?: string) {
this.selectedCategory = newCategory;
}
changePage(newPage: number) {
this.selectedPage = newPage;
}
changePageSize(newSize: number) {
this.productsPerPage = Number(newSize);
this.changePage(1);
}
get pageNumbers(): number[] {
return Array(Math.ceil(this.repository
.getProducts(this.selectedCategory).length / this.productsPerPage))
.fill(0).map((x, i) => i + 1);
}
}
```

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. У лістингу 1.24 наведено зміни у шаблоні компонента магазину, необхідні для реалізації сторінкового виводу.

Лістинг 1.24. Реалізація сторінкового виводу у файлі store.component.html

```
<div class="container-fluid">
<div class="row">
<div class="bg-dark text-white p-2">
<span class="navbar-brand ml-2">SPORTS STORE</span>
```

```
</div>
</div>
<div class="row text-white">
<div class="col-3 p-2">
<div class="d-grid gap-2">
<button class="btn btn-outline-primary" (click)="changeCategory()">
Home
</button>
<button *ngFor="let cat of categories"
class="btn btn-outline-primary"
[class.active]="cat == selectedCategory"
(click)="changeCategory(cat)">
{{cat}}
</button>
</div>
</div>
<div class="col-9 p-2 text-dark">
<div *ngFor="let product of products" class="card m-1 p-1 bg-light">
<h4>
{{product.name}}
<span class="badge rounded-pill bg-primary" style="float:right">
{{ product.price | currency:"USD":"symbol":"2.2-2" }}
</span>
</h4>
<div class="card-text bg-white p-1">{{product.description}}</div>
</div>
<div class="form-inline float-start mr-1">
<select class="form-control" [value]="productsPerPage"
(change)="changePageSize($any($event).target.value)">
<option value="3">3 per Page</option>
<option value="4">4 per Page</option>
<option value="6">6 per Page</option>
<option value="8">8 per Page</option>
</select>
</div>
<div class="btn-group float-end">
<button *ngFor="let page of pageNumbers" (click)="changePage(page)"
class="btn btn-outline-primary"
[class.active]="page == selectedPage">
{{page}}
</button>
</div>
</div>
</div>
</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 (рис. 1.7).

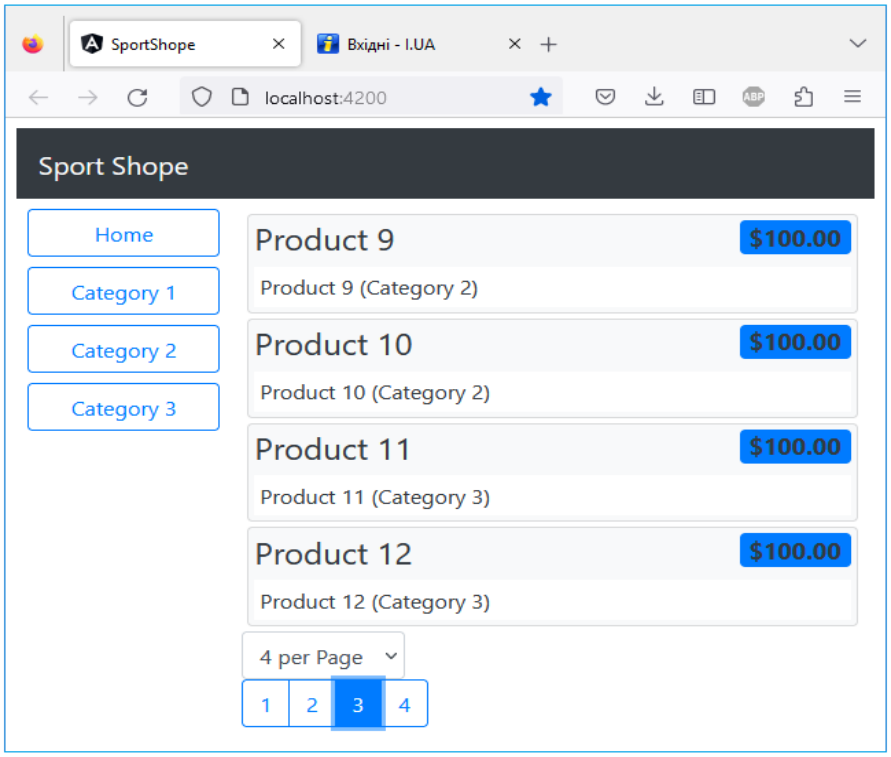

Рис. 1.7. Посторінне виведення списку товарів

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/store та використайте його для визначення класу з лістингу 1.25.

Лістинг 1.25. Вміст файлу counter.directive.ts у папці SportShop/src/app/store

```
import {
Directive, ViewContainerRef, TemplateRef, Input, SimpleChanges
} from "@angular/core";
@Directive({
selector: "[counterOf]"
})
export class CounterDirective {
constructor(private container: ViewContainerRef,
private template: TemplateRef<Object>) {
```

```
}
@Input("counterOf")
counter: number = 0;
ngOnChanges(changes: SimpleChanges) {
this.container.clear();
for (let i = 0; i < this.counter; i++) {
this.container.createEmbeddedView(this.template,
new CounterDirectiveContext(i + 1));
}
}
}
class CounterDirectiveContext {
constructor(public $implicit: any) { }
}
```

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 у лістингу 1.26.

Лістинг 1.26. Реєстрація нестандартної директиви у файлі store.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { StoreComponent } from "./store.component";
import { CounterDirective } from "./counter.directive";
@NgModule({
imports: [ModelModule, BrowserModule, FormsModule],
declarations: [StoreComponent, CounterDirective],
exports: [StoreComponent]
})
export class StoreModule { }
```

Після того, як директива була зареєстрована, вона може використовуватися в шаблоні компонента магазину для заміни директиви ngFor, як показано у лістингу 1.27.

Лістинг 1.27. Заміна вбудованої директиви у файлі store.component.html

```
<div class="container-fluid">
<div class="row">
<div class="bg-dark text-white p-2">
```

```
<span class="navbar-brand ml-2">SPORTS STORE</span>
</div>
</div>
<div class="row text-white">
<div class="col-3 p-2">
<div class="d-grid gap-2">
<button class="btn btn-outline-primary" (click)="changeCategory()">
Home
</button>
<button *ngFor="let cat of categories"
class="btn btn-outline-primary"
[class.active]="cat == selectedCategory"
(click)="changeCategory(cat)">
{{cat}}
</button>
</div>
</div>
<div class="col-9 p-2 text-dark">
<div *ngFor="let product of products" class="card m-1 p-1 bg-light">
<h4>
{{product.name}}
<span class="badge rounded-pill bg-primary" style="float:right">
{{ product.price | currency:"USD":"symbol":"2.2-2" }}
</span>
</h4>
<div class="card-text bg-white p-1">{{product.description}}</div>
</div>
<div class="form-inline float-start mr-1">
<select class="form-control" [value]="productsPerPage"
(change)="changePageSize($any($event).target.value)">
<option value="3">3 per Page</option>
<option value="4">4 per Page</option>
<option value="6">6 per Page</option>
<option value="8">8 per Page</option>
</select>
</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>
</div>
</div>
```

Нова прив'язка даних залежить від настроювання нестандартної директиви з використанням властивості pageCount. У лістингу 1.28 масив чисел замінюється простим значенням number, що представляє результат виразу.

Лістинг 1.28. Підтримка нестандартної директиви у файлі store.component.ts

```
import { Component } from "@angular/core";
import { Product } from "../model/product.model";
import { ProductRepository } from "../model/product.repository";
@Component({
selector: "store",
templateUrl: "store.component.html"
})
export class StoreComponent {
selectedCategory: string | undefined;
productsPerPage = 4;
selectedPage = 1;
constructor(private repository: ProductRepository) { }
get products(): Product[] {
let pageIndex = (this.selectedPage - 1) * this.productsPerPage
return this.repository.getProducts(this.selectedCategory)
.slice(pageIndex, pageIndex + this.productsPerPage);
}
get categories(): string[] {
return this.repository.getCategories();
}
changeCategory(newCategory?: string) {
this.selectedCategory = newCategory;
}
changePage(newPage: number) {
this.selectedPage = newPage;
}
changePageSize(newSize: number) {
this.productsPerPage = Number(newSize);
this.changePage(1);
}
// get pageNumbers(): number[] {
// return Array(Math.ceil(this.repository
// .getProducts(this.selectedCategory).length / this.productsPerPage))
// .fill(0).map((x, i) => i + 1);
// }
get pageCount(): number {
return Math.ceil(this.repository
.getProducts(this.selectedCategory).length / this.productsPerPage)
}
}
```

В додатку SportShop зовні нічого не змінилося, але лістинг 1.28 продемонстрував використання нестандартної директиви для демонстрації того, як вбудовані функції Angular можуть розширюватись спеціалізованим кодом, адаптованим для потреб конкретного проекту.

В цій частині ми розпочали роботу над проектом SportShop. Частина 1 була присвячена «закладанню фундаменту» для проекту: установці та налаштуванню засобів розробника, створенню кореневих структурних блоків для застосування та початку роботи над функціональними модулями. Коли підготовку було завершено, ми швидко 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 замовлення. Розширення функціональності проекту «SportShop», реалізувавши 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.

Підготовка програми

Виконайте наступну команду з папки SportShope, щоб запустити REST-сумісну веб-службу:

```
npm start json
```

Відкрийте друге вікно командного рядка та введіть наступну команду з папки SportShope, щоб запустити інструменти розробки та сервер HTTP:

```
ng serve –open
```

Створення кошика

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

Створення моделі кошика

Вихідною точкою для функціональності кошика стане новий клас моделі, який використовуватиметься для збирання товарів, обраних користувачем.

Створіть файл з ім'ям cart.model.ts в папці SportShope/src/app/model та включіть у нього визначення класу з лістингу 2.1.

Лістинг 2.1. Вміст файлу cart.model.ts у папці SportShope/app/model

```
import { Injectable } from "@angular/core";
import { Product } from "./product.model";
@Injectable()
export class Cart {
public lines: CartLine[] = [];
public itemCount: number = 0;
public cartPrice: number = 0;
addLine(product: Product, quantity: number = 1) {
let line = this.lines.find(line => line.product.id == product.id);
if (line != undefined) {
line.quantity += quantity;
} else {
this.lines.push(new CartLine(product, quantity));
}
this.recalculate();
}
updateQuantity(product: Product, quantity: number) {
let line = this.lines.find(line => line.product.id == product.id);
if (line != undefined) {
line.quantity = Number(quantity);
}
this.recalculate();
}
removeLine(id: number) {
let index = this.lines.findIndex(line => line.product.id == id);
this.lines.splice(index, 1);
this.recalculate();
}
clear() {
this.lines = [];
this.itemCount = 0;
this.cartPrice = 0;
}
private recalculate() {
this.itemCount = 0;
this.cartPrice = 0;
this.lines.forEach(l => {
this.itemCount += l.quantity;
this.cartPrice += l.lineTotal;
})
}
}
export class CartLine {
```

```
constructor(public product: Product,
public quantity: number) {}
get lineTotal() {
return this.quantity * (this.product.price ?? 0);
}
}
```

Вибрані продукти є масивом об'єктів CartLine, 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.) Лістинг 2.2 реєструє клас Cart як службу у властивості providers функціонального модуля моделі.

Лістинг 2.2. Реєстрація Cart як служби у файлі model.module.ts

```
import { NgModule } від "@angular/core";
import { ProductRepository } від "./product.repository";
import { StaticDataSource } from "./static.datasource";
import { Cart } from "./cart.model";
@NgModule({
providers: [ProductRepository, StaticDataSource, Cart]
})
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 SportShope виводить зведену інформацію про вибрані товари в заголовку сторінки; для реалізації цієї функціональності треба створити компонент. Створіть файл з ім'ям cartSummary.component.ts в папці SportShope/src/app/store та визначте в ньому компонент з лістингу 2.3.

Лістинг 2.3. Вміст файлу cartSummary.component.ts у папці SportShope/src/app/store

```
import { Component } from "@angular/core";
import { Cart } from "../model/cart.model";
@Component({
selector: "cart-summary",
templateUrl: "cartSummary.component.html"
})
export class CartSummaryComponent {
constructor(public cart: Cart) { }
}
```

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, створіть файл HTML з ім'ям cartSummary.component.html в одній папці з файлом класу компонента та додайте розмітку з лістингу 2.4.

Лістинг 2.4. Вміст файлу cartSummary.component.html у папці SportShope/src/app/store

```
<div class="float-end">
<small class="fs-6">
Your cart:
<span *ngIf="cart.itemCount > 0">
{{ cart.itemCount }} item(s)
{{ cart.cartPrice | currency:"USD":"symbol":"2.2-2" }}
</span>
<span *ngIf="cart.itemCount == 0">
(empty)
</span>
</small>
<button class="btn btn-sm bg-dark text-white" [disabled]="cart.itemCount == 0">
<i class="fa fa-shopping-cart"></i>
</button>
</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.

Лістинг 2.5 реєструє новий компонент у функціональному модулі магазину, щоб підготуватися до його використання пізніше.

Лістинг 2.5.Реєстрація компонента у файлі store.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { StoreComponent } from "./store.component";
import { CounterDirective } from "./counter.directive";
import { CartSummaryComponent } from "./cartSummary.component";
@NgModule({
imports: [ModelModule, BrowserModule, FormsModule],
declarations: [StoreComponent, CounterDirective, CartSummaryComponent],
exports: [StoreComponent]
})
export class StoreModule { }
```

Інтеграція кошика у додаток

Компонент store відіграє ключову роль в інтеграції кошика та його віджету у додаток. У лістингу 2.6 оновлюється компонент store: до нього додається конструктор з параметром Cart, а також визначається метод додавання товару в кошик.

Лістинг 2.6.Додавання підтримки кошика до файлу store.component.ts

```
import { Component } from "@angular/core";
import { Product } from "../model/product.model";
import { ProductRepository } from "../model/product.repository";
import { Cart } from "../model/cart.model";
@Component({
selector: "store",
templateUrl: "store.component.html"
})
export class StoreComponent {
selectedCategory: string | undefined;
productsPerPage = 4;
selectedPage = 1;
constructor(private repository: ProductRepository,
private cart: Cart) { }
get products(): Product[] {
let pageIndex = (this.selectedPage - 1) * this.productsPerPage
return this.repository.getProducts(this.selectedCategory)
.slice(pageIndex, pageIndex + this.productsPerPage);
}
get categories(): string[] {
return this.repository.getCategories();
```

```
}
changeCategory(newCategory?: string) {
this.selectedCategory = newCategory;
}
changePage(newPage: number) {
this.selectedPage = newPage;
}
changePageSize(newSize: number) {
this.productsPerPage = Number(newSize);
this.changePage(1);
}
get pageCount(): number {
return Math.ceil(this.repository
.getProducts(this.selectedCategory).length / this.productsPerPage)
}
addProductToCart(product: Product) {
this.cart.addLine(product);
}
}
```

Щоб завершити інтеграцію кошика до компоненту магазину, у лістингу 2.7 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.

Лістинг 2.7. Застосування компонента у файлі store.component.html

```
<div class="container-fluid">
<div class="row">
<div class="bg-dark text-white p-2">
<span class="navbar-brand ml-2">
SportShope
<cart-summary></cart-summary>
</span>
</div>
</div>
<div class="row text-white">
<div class="col-3 p-2">
<div class="d-grid gap-2">
<button class="btn btn-outline-primary" (click)="changeCategory()">
Home
</button>
<button *ngFor="let cat of categories"
class="btn btn-outline-primary"
[class.active]="cat == selectedCategory"
(click)="changeCategory(cat)">
{{cat}}
```

```
</button>
</div>
</div>
<div class="col-9 p-2 text-dark">
<div *ngFor="let product of products" class="card m-1 p-1 bg-light">
<h4>
{{product.name}}
<span class="badge rounded-pill bg-primary" style="float:right">
{{ product.price | currency:"USD":"symbol":"2.2-2" }}
</span>
</h4>
<div class="card-text bg-white p-1">
{{product.description}}
<button class="btn btn-success btn-sm float-end"
(click)="addProductToCart(product)">
Add To Cart
</button>
</div>
</div>
<div class="form-inline float-start mr-1">
<select class="form-control" [value]="productsPerPage"
(change)="changePageSize($any($event).target.value)">
<option value="3">3 per Page</option>
<option value="4">4 per Page</option>
<option value="6">6 per Page</option>
<option value="8">8 per Page</option>
</select>
</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>
</div>
</div>
```

У результаті для кожного товару створюється кнопка додавання до корзини (рис. 2.1). Повноцінна підтримка кошика ще не реалізована, але наслідки кожного додавання товару відображаються у зведенні у верхній частині сторінки.

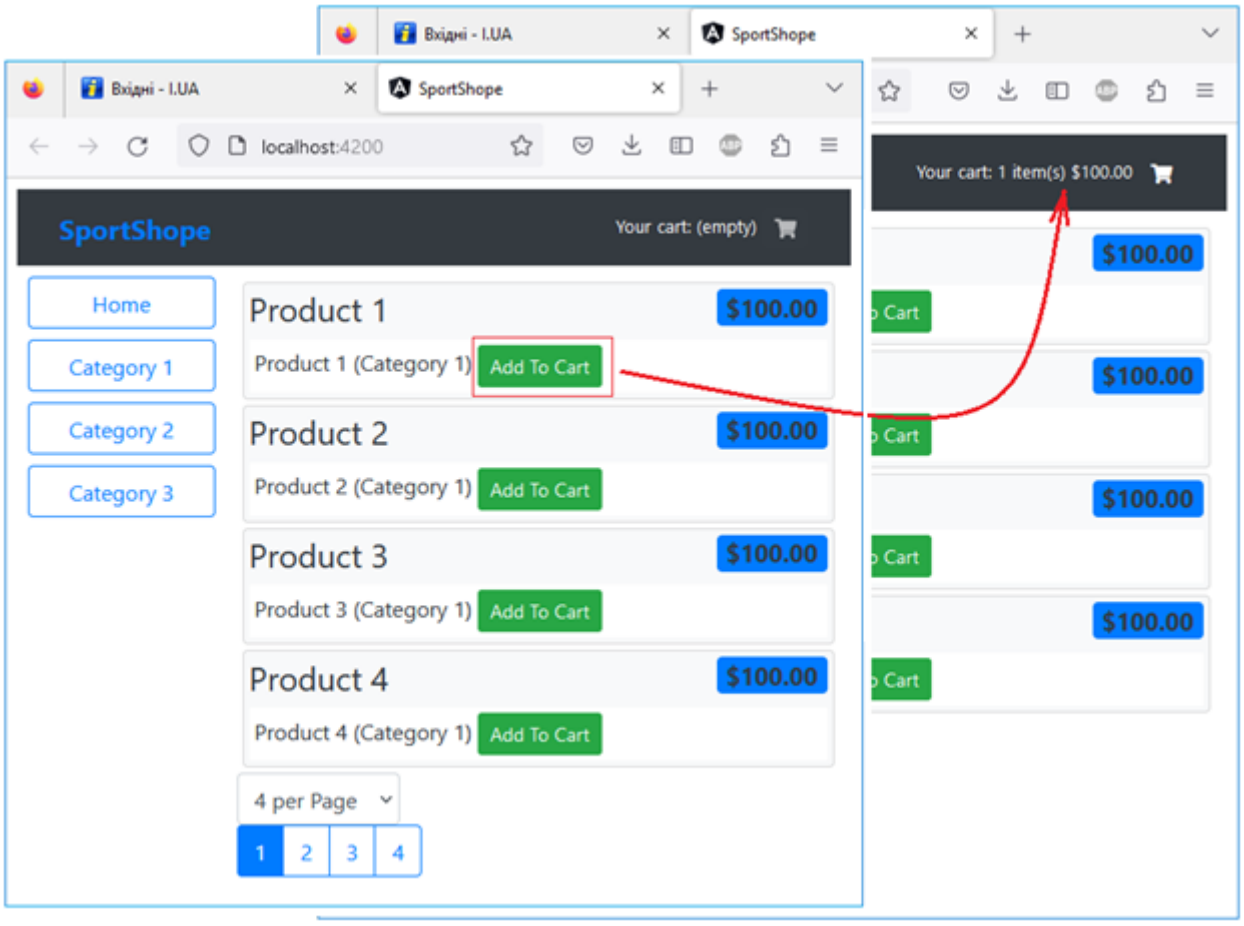

Рис. 2.1. Додавання підтримки кошика до програми SportShope

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 SportShope 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 URL-адресу в браузері для вибору компонентів, що відображаються для користувача. 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 SportShope додамо підтримку трьох різних URL-адрес з табл. 2.1.

Таблиця 2.1.URL-адреси, які підтримує програма SportShope

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

Лістинг 2.8. Вміст файлу cartDetail.component.ts у папці SportShope/app/store

```
import { Component } from "@angular/core";
@Component({
template: `<div><h3 class="bg-info p-1 text-white">Cart Detail Component</h3></div>`
})
export class CartDetailComponent {}
```

Потім створіть файл checkout.component.ts в папці SportShope/src/app/store і дододайте визначення компонента з лістингу 2.9.

Лістинг 2.9. Вміст файлу checkout.component.ts у папці SportShope/app/store

```
import { Component } from "@angular/core";
@Component({
template: `<div><h3 class="bg-info p-1 text-white">Checkout Component</h3></div>`
})
export class CheckoutComponent { }
```

Компонент побудований за тією ж схемою, що і компонент кошика: він виводить тимчасове повідомлення, яке показує, який компонент відображається. У лістингу 2.10 компоненти реєструються у функціональному модулі store і включаються у властивість exports, щоб вони могли використовуватись в інших місцях програми.

Лістинг 2.10.Реєстрація компонентів у файлі store.module.ts

```
import { NgModule } від "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { StoreComponent } from "./store.component";
import { CounterDirective } from "./counter.directive";
import { CartSummaryComponent } from "./cartsummary.component";
import { CartDetailComponent } from "./cartDetail.component";
import { CheckoutComponent } from "./checkout.component";
@NgModule({
imports: [ModelModule, BrowserModule, FormsModule],
declarations: [StoreComponent, CounterDirective, CartSummaryComponent,
CartDetailComponent, CheckoutComponent],
exports: [StoreComponent, CartDetailComponent, CheckoutComponent]
```

```
})
export class StoreModule { }
```

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 2.11).

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 SportShope. Якщо елемент пропущено, Angular повідомить про помилку та не зможе застосувати маршрути.

Лістинг 2.11.Створення конфігурації маршрутизації у файлі app.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { AppComponent } from "./app.component";
import { StoreModule } from "./store/store.module";
import { StoreComponent } from "./store/store.component";
import { CheckoutComponent } from "./store/checkout.component";
import { CartDetailComponent } from "./store/cartDetail.component";
import { RouterModule } from "@angular/router";
@NgModule({
imports: [BrowserModule, StoreModule,
RouterModule.forRoot([
{ path: "store", component: StoreComponent },
{ path: "cart", component: CartDetailComponent },
{ path: "checkout", component: CheckoutComponent },
{ path: "**", redirectTo: "/store" }
])],
declarations: [AppComponent],
bootstrap: [AppComponent]
})
export class AppModule { }
```

Методу RouterModule.forRoot передається набір маршрутів, кожен із яких пов'язує URL з компонентом. Перші три маршрути у лістингу відповідають URL із табл. 2.1. Останній маршрут є універсальним — він перенаправляє будь-яку іншу URL на /store, що відображає StoreComponent.

При використанні механізму маршрутизації Angular шукає елемент router-outlet, що визначає місце для пошуку компонента, що відповідає поточному URL. У лістингу 2.12 елемент store шаблону кореневого компонента замінюється елементом router-outlet.

Лістинг 2.12.Визначення мети маршрутизації у файлі app.component.ts

```
import { Component } від "@angular/core";
@Component({
selector: "app",
template: "<router-outlet></router-outlet>"
})
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 (рис. 2.2).

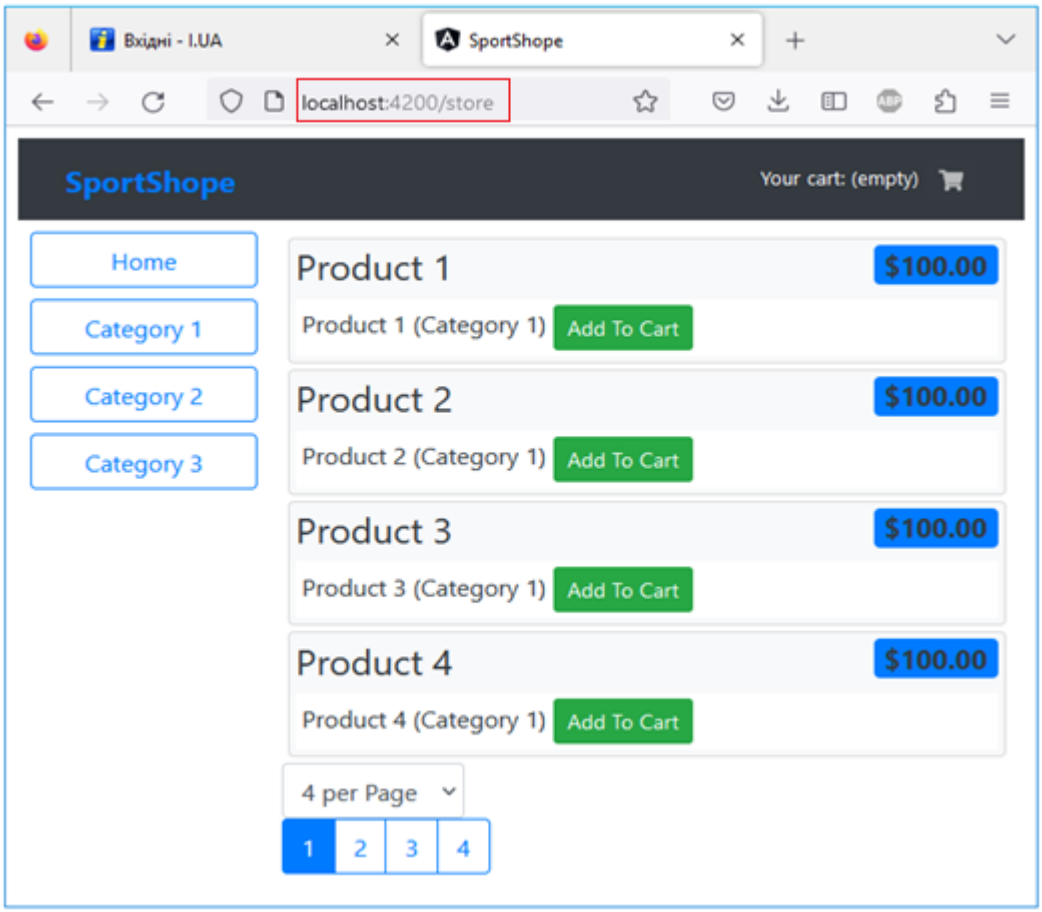

Рис. 2.2. Ефект маршрутизації URL

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 лістингу 2.13 навігація додається у метод компонента, який викликається при натисканні кнопки користувачем.

Лістинг 2.13. Навігація з використанням JavaScript у файлі store.component.ts

```
import { Component } from "@angular/core";
import { Product } from "../model/product.model";
import { ProductRepository } from "../model/product.repository";
import { Cart } from "../model/cart.model";
import { Router } from "@angular/router";
@Component({
selector: "store",
templateUrl: "store.component.html"
})
export class StoreComponent {
selectedCategory: string | undefined;
productsPerPage = 4;
selectedPage = 1;
constructor(private repository: ProductRepository,
private cart: Cart,
private router: Router) { }
get products(): Product[] {
let pageIndex = (this.selectedPage - 1) * this.productsPerPage
return this.repository.getProducts(this.selectedCategory)
.slice(pageIndex, pageIndex + this.productsPerPage);
}
get categories(): string[] {
return this.repository.getCategories();
}
changeCategory(newCategory?: string) {
this.selectedCategory = newCategory;
}
changePage(newPage: number) {
this.selectedPage = newPage;
}
```

```
changePageSize(newSize: number) {
this.productsPerPage = Number(newSize);
this.changePage(1);
}
get pageCount(): number {
return Math.ceil(this.repository
.getProducts(this.selectedCategory).length / this.productsPerPage)
}
addProductToCart(product: Product) {
this.cart.addLine(product);
this.router.navigateByUrl("/cart");
}
}
```

Конструктор отримує параметр Router, що надається Angular через механізм впровадження залежностей при створенні нового екземпляра компонента. У методіaddProductToCartметодRouter.navigateByUrlвикористовується для переходу через URL /cart.

Навігація також може здійснюватися додаванням атрибутуrouterLinkелементи в шаблоні. У лістингу 2.14 атрибут routerLink застосовується до кнопки у шаблоні компонента зведеної інформації кошика.

Лістинг 2.14. Додавання навігаційного атрибуту до файлу cartSummary.component.html

```
<div class="float-end">
<small class="fs-6">
Your cart:
<span *ngIf="cart.itemCount > 0">
{{ cart.itemCount }} item(s)
{{ cart.cartPrice | currency:"USD":"symbol":"2.2-2" }}
</span>
<span *ngIf="cart.itemCount == 0">
(empty)
</span>
</small>
<button class="btn btn-sm bg-dark text-white"
[disabled]="cart.itemCount == 0" routerLink="/cart">
<i class="fa fa-shopping-cart"></i>
</button>
</div>
```

Як значення атрибуту routerLink вказується URL-адреса, за якою має переходити додаток при натисканні на кнопці. 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 у лістингу 2.15.

Лістинг 2.15. Імпортування модуля маршрутизації у файл store.module.ts

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { StoreComponent } from "./store.component";
import { CounterDirective } from "./counter.directive";
import { CartSummaryComponent } from "./cartsummary.component";
import { CartDetailComponent } from "./cartDetail.component";
import { CheckoutComponent } from "./checkout.component";
import { RouterModule } from "@angular/router";
@NgModule({
imports: [ModelModule, BrowserModule, FormsModule, RouterModule],
declarations: [StoreComponent, CounterDirective, CartSummaryComponent,
CartDetailComponent, CheckoutComponent],
exports: [StoreComponent, CartDetailComponent, CheckoutComponent]
})
export class StoreModule { }
```

Щоб побачити, як навігація працює, збережіть зміни у файлах, а після того як браузер перезавантажить документ HTML, клацніть на одній з кнопок Add To Cart. Браузер переходить за URL-адресою /cart (рис. 2.3).

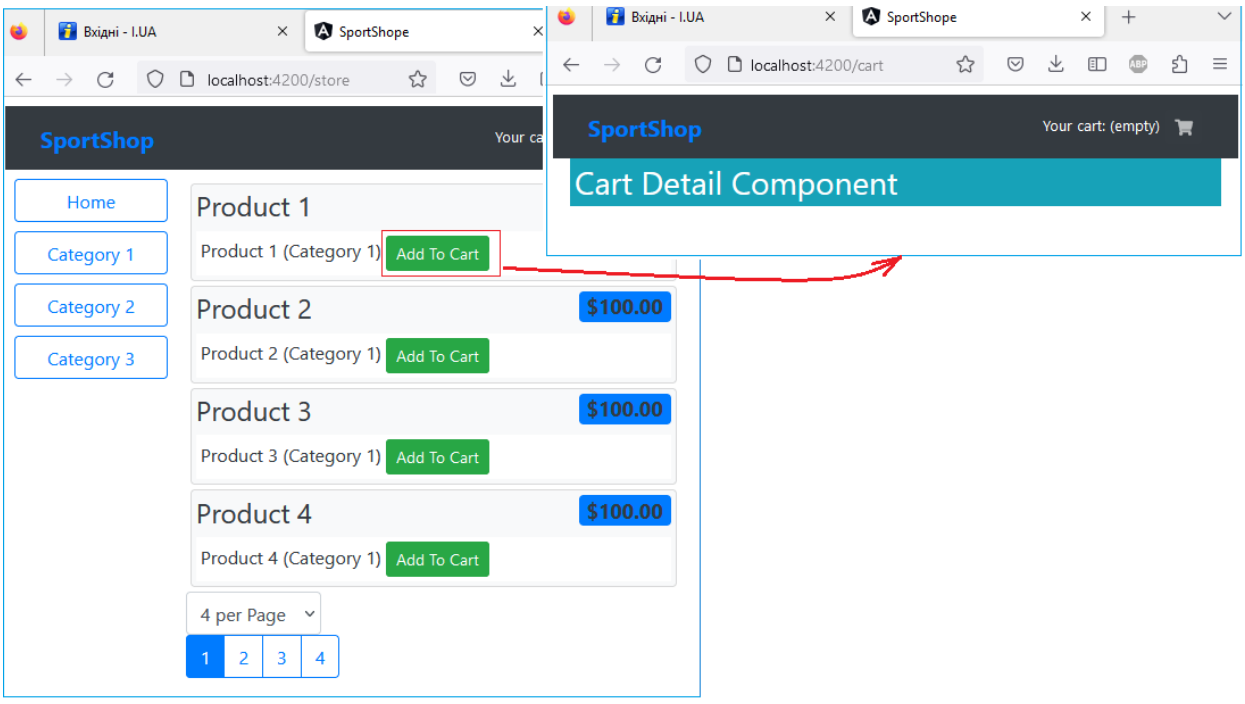

Рис. 2.3. Маршрутизація URL

Захисники маршрутів

Пам'ятайте, що навігація може виконуватись лише програмою. Якщо змінити URL-адресу 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 кнопці 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, додайте файл storeFirst.guard.ts у папку SportShope/app та визначте клас з лістингу 2.16.

Лістинг 2.16. Вміст файлу storeFirst.guard.ts у папці SportShope/src/app

```
import { Injectable } from "@angular/core";
import {
ActivatedRouteSnapshot, RouterStateSnapshot,
Router
} from "@angular/router";
import { StoreComponent } from "./store/store.component";
@Injectable()
export class StoreFirstGuard {
private firstNavigation = true;
constructor(private router: Router) { }
canActivate(route: ActivatedRouteSnapshot,
state: RouterStateSnapshot): boolean {
if (this.firstNavigation) {
this.firstNavigation = false;
if (route.component != StoreComponent) {
this.router.navigateByUrl("/");
return false;
}
}
return true;
}
}
```

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 StoreComponent. 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. У лістингу 2.17 захисник реєструється як служба за допомогою якості providers кореневого модуля та захищає кожен маршрут за допомогою властивості canActivate.

Лістинг 2.17. Захист маршрутів у файлі app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { StoreModule } from "./store/store.module";
import { StoreComponent } from "./store/store.component";
import { CheckoutComponent } from "./store/checkout.component";
import { CartDetailComponent } from "./store/cartDetail.component";
import { RouterModule } from "@angular/router";
import { StoreFirstGuard } from "./storeFirst.guard";
@NgModule({
declarations: [AppComponent],
imports: [BrowserModule, StoreModule,
RouterModule.forRoot([
{
path: "store", component: StoreComponent,
canActivate: [StoreFirstGuard]
},
{
path: "cart", component: CartDetailComponent,
canActivate: [StoreFirstGuard]
},
{
path: "checkout", component: CheckoutComponent,
canActivate: [StoreFirstGuard]
},
{ path: "**", redirectTo: "/store" }
])],
providers: [StoreFirstGuard],
bootstrap: [AppComponent]
})
export class AppModule { }
```

Якщо оновити браузер після клацання на одній із кнопок Add To Cart, ви побачите, що браузер автоматично перенаправляється у безпечний стан (рис. 2.4).

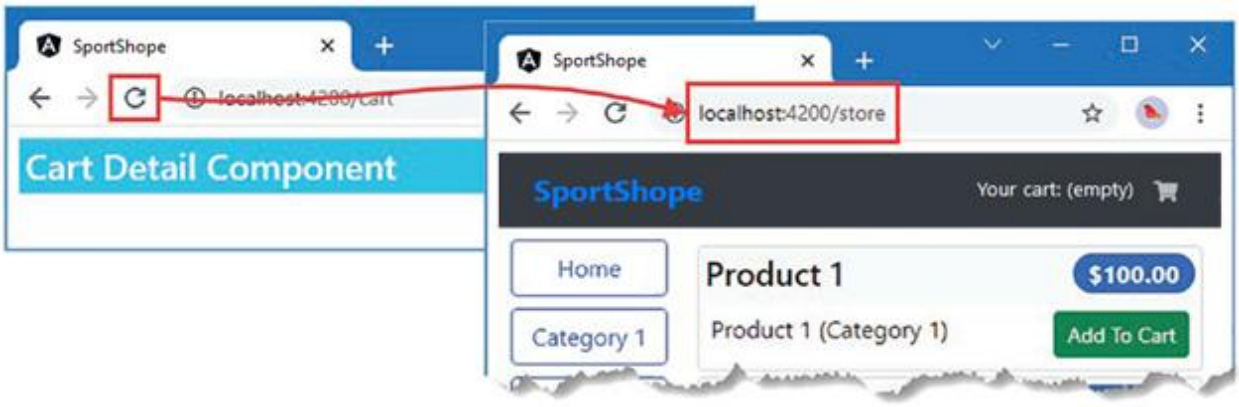

Рис. 2.4. Захист маршрутів

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 2.18 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.

Лістинг 2.18. Зміна шаблону у файлі cardDetail.component.ts

```
import { Component } від "@angular/core";
import { Cart } from "../model/cart.model";
@Component({
moduleId: module.id,
templateUrl: "cartDetail.component.html"
})
export class CartDetailComponent {
constructor(public cart: Cart) {}
}
```

Щоб завершити функціональність кошика, створіть файл HTML з ім'ям cartDetail.component.html в папці SportShope/src/app/store та додайте контент із лістингу 2.19.

Лістинг 2.19. Вміст файлу cardDetail.component.html у папці SportShope/src/app/store

```
<div class="container-fluid">
```

```
<div class="row">
<div class="bg-dark text-white p-2">
<span class="navbar-brand ml-2">SPORTS STORE</span>
</div>
</div>
<div class="row">
<div class="col mt-2">
<h2 class="text-center">Your Cart</h2>
<table class="table table-bordered table-striped p-2">
<thead>
<tr>
<th>Quantity</th>
<th>Product</th>
<th class="text-end">Price</th>
<th class="text-end">Subtotal</th>
</tr>
</thead>
<tbody>
<tr *ngIf="cart.lines.length == 0">
<td colspan="4" class="text-center">
Your cart is empty
</td>
</tr>
<tr *ngFor="let line of cart.lines">
<td>
<input type="number" class="form-control-sm"
style="width:5em" [value]="line.quantity"
(change)="cart.updateQuantity(line.product,
$any($event).target.value)" />
</td>
<td>{{line.product.name}}</td>
<td class="text-end">
{{line.product.price | currency:"USD":"symbol":"2.2-2"}}
</td>
<td class="text-end">
{{(line.lineTotal) | currency:"USD":"symbol":"2.2-2" }}
</td>
<td class="text-center">
<button class="btn btn-sm btn-danger"
(click)="cart.removeLine(line.product.id ?? 0)">
Remove
</button>
</td>
</tr>
</tbody>
<tfoot>
<tr>
```

```
<td colspan="3" class="text-end">Total:</td>
<td class="text-end">
{{cart.cartPrice | currency:"USD":"symbol":"2.2-2"}}
</td>
</tr>
</tfoot>
</table>
</div>
</div>
<div class="row">
<div class="col">
<div class="text-center">
<button class="btn btn-primary m-1" routerLink="/store">
Continue Shopping
</button>
<button class="btn btn-secondary m-1" routerLink="/checkout"
[disabled]="cart.lines.length == 0">
Checkout
</button>
</div>
</div>
</div>
</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 (рис. 2.5). 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 кнопці 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.

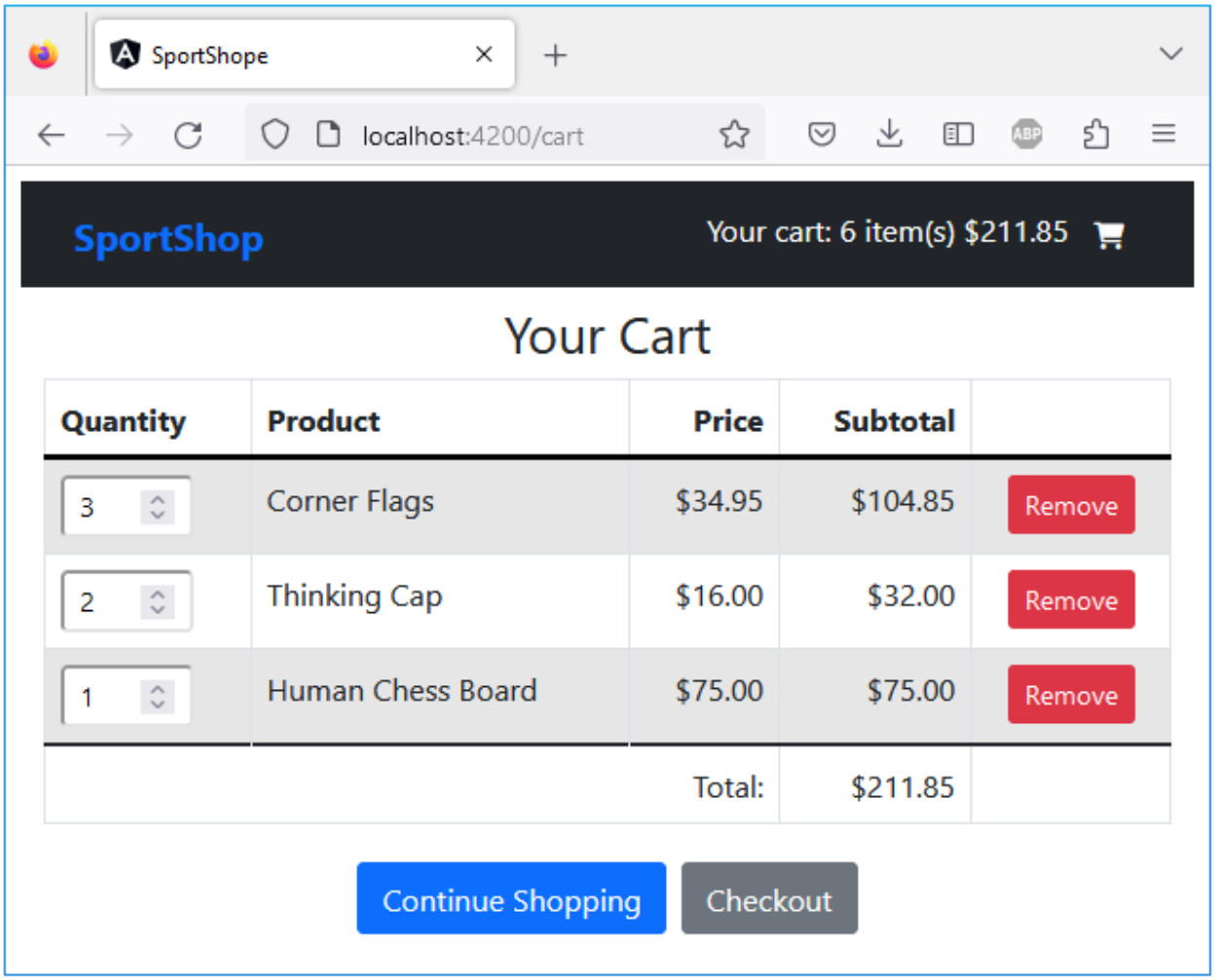

Рис. 2.5. Готова функціональність кошика

Розширення моделі

Для опису замовлень, розміщених користувачами, створіть файл із ім'ям order.model.ts в папці SportShope/src/app/model і додайте код з лістингу 2.20.

Лістинг 2.20. Вміст файлу order.model.ts у папці SportShope/src/app/model

```
import { Injectable } from "@angular/core";
import { Cart } from "./cart.model";
@Injectable()
export class Order {
public id?: number;
public name?: string;
public address?: string;
public city?: string;
public state?: string;
public zip?: string;
public country?: string;
public shipped: boolean = false;
constructor(public cart: Cart) {}
clear() {
```

```
this.id = undefined;
this.name = this.address = this.city = undefined;
this.state = this.zip = this.country = undefined;
this.shipped = false;
this.cart.clear();
}
}
```

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 лістингу 2.21 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.

Лістинг 2.21. Обробка замовлень у файлі static.datasource.ts

```
import { Injectable } from "@angular/core";
import { Product } from "./product.model";
import { Observable, from } from "rxjs";
import { Order } from "./order.model";
@Injectable()
export class StaticDataSource {
private products: Product[] = [
new Product(1, "Product 1", "Category 1", "Product 1 (Category 1)", 100),
new Product(2, "Product 2", "Category 1", "Product 2 (Category 1)", 100),
new Product(3, "Product 3", "Category 1", "Product 3 (Category 1)", 100),
new Product(4, "Product 4", "Category 1", "Product 4 (Category 1)", 100),
new Product(5, "Product 5", "Category 1", "Product 5 (Category 1)", 100),
new Product(6, "Product 6", "Category 2", "Product 6 (Category 2)", 100),
new Product(7, "Product 7", "Category 2", "Product 7 (Category 2)", 100),
new Product(8, "Product 8", "Category 2", "Product 8 (Category 2)", 100),
new Product(9, "Product 9", "Category 2", "Product 9 (Category 2)", 100),
new Product(10, "Product 10", "Category 2", "Product 10 (Category 2)", 100),
new Product(11, "Product 11", "Category 3", "Product 11 (Category 3)", 100),
new Product(12, "Product 12", "Category 3", "Product 12 (Category 3)", 100),
new Product(13, "Product 13", "Category 3", "Product 13 (Category 3)", 100),
new Product(14, "Product 14", "Category 3", "Product 14 (Category 3)", 100),
new Product(15, "Product 15", "Category 3", "Product 15 (Category 3)", 100),];
```

```
getProducts(): Observable<Product[]> {
return from([this.products]);
}

saveOrder(order: Order): Observable<Order> {
   console.log(JSON.stringify(order));
   return from([order]);
   }
}
```

Для керування замовленнями створіть файл order.repository.ts в папці SportShope/app/model та використовуйте його для визначення класу з лістингу 2.22. 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.

Лістинг 2.22. Вміст файлу order.repository.ts у папці SportShope/app/model

```
import { Injectable } from "@angular/core";
import { Observable } from "rxjs";
import { Order } from "./order.model";
import { StaticDataSource } from "./static.datasource";
@Injectable()
export class OrderRepository {
private orders: Order[] = [];
constructor(private dataSource: StaticDataSource) {}
getOrders(): Order[] {
return this.orders;
}
saveOrder(order: Order): Observable<Order> {
return this.dataSource.saveOrder(order);
}
}
```

Оновлення функціонального модуля

У лістингу 2.23 клас Order і новий репозиторій реєструються як служби за допомогою якості providers у функціональному модулі моделі.

Лістинг 2.23. Реєстрація служб у файлі model.module.ts

```
import { NgModule } from "@angular/core";
import { ProductRepository } from "./product.repository";
import { StaticDataSource } from "./static.datasource";
import { Cart } from "./cart.model";
```

```
import { Order } from "./order.model";
import { OrderRepository } from "./order.repository";
@NgModule({
providers: [ProductRepository, StaticDataSource, Cart, Order, OrderRepository]
})
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 лістингу 2.24 відбувається підготовка компонента оформлення замовлення, перемикання на зовнішній шаблон, отримання об'єкта Order у параметрі конструктора та реалізація додаткової підтримки роботи шаблону.

Лістинг 2.24. Підготовка форми у файлі checkout.component.ts

```
import { Component } from "@angular/core";
import { NgForm } from "@angular/forms";
import { OrderRepository } from "../model/order.repository";
import { Order } from "../model/order.model";
@Component({
templateUrl: "checkout.component.html",
styleUrls: ["checkout.component.css"]
})
export class CheckoutComponent {
orderSent: boolean = false;
submitted: boolean = false;
constructor(public repository: OrderRepository,
public order: Order) {}
submitOrder(form: NgForm) {
this.submitted = true;
if (form.valid) {
this.repository.saveOrder(this.order).subscribe(order => {
this.order.clear();
this.orderSent = true;
this.submitted = false;
});
}
}
```

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 стильових таблиць CSS, які мають застосовуватися до контенту шаблону компонента. Щоб надати зворотній зв'язок перевірки даних для значень, введених користувачем в елементах форми HTML, створіть файл з ім'ям checkout.component.css в папці SportShope/app/store та визначте стилі у лістингу 2.25.

Лістинг 2.25.Вміст файлу checkout.component.css у папці SportShope/app/store

```
input.ng-dirty.ng-invalid { border: 2px solid #ff0000 }
input.ng-dirty.ng-valid { border: 2px solid #6bc502 }
```

Angular додає елементи до класів ng-dirty, ng-validing-valid для визначення статусу перевірки. Стилі в лістингу 2.25 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 (лістинг 2.26).

Лістинг 2.26.Вміст файлу checkout.component.html

```
<div class="container-fluid">
<div class="row">
<div class="bg-dark text-white p-2">
<span class="navbar-brand ml-2">SPORTS STORE</span>
</div>
</div>
</div>
<div *ngIf="orderSent" class="m-2 text-center">
<h2>Thanks!</h2>
<p>Thanks for placing your order.</p>
<p>We'll ship your goods as soon as possible.</p>
<button class="btn btn-primary" routerLink="/store">Return to Store</button>
</div>
<form *ngIf="!orderSent" #form="ngForm" novalidate
(ngSubmit)="submitOrder(form)" class="m-2">
<div class="form-group">
<label>Name</label>
<input class="form-control" #name="ngModel" name="name"
[(ngModel)]="order.name" required />
<span *ngIf="submitted && name.invalid" class="text-danger">
Please enter your name
</span>
</div>
<div class="form-group">
```

```
<label>Address</label>
<input class="form-control" #address="ngModel" name="address"
[(ngModel)]="order.address" required />
<span *ngIf="submitted && address.invalid" class="text-danger">
Please enter your address
</span>
</div>
<div class="form-group">
<label>City</label>
<input class="form-control" #city="ngModel" name="city"
[(ngModel)]="order.city" required />
<span *ngIf="submitted && city.invalid" class="text-danger">
Please enter your city
</span>
</div>
<div class="form-group">
<label>State</label>
<input class="form-control" #state="ngModel" name="state"
[(ngModel)]="order.state" required />
<span *ngIf="submitted && state.invalid" class="text-danger">
Please enter your state
</span>
</div>
<div class="form-group">
<label>Zip/Postal Code</label>
<input class="form-control" #zip="ngModel" name="zip"
[(ngModel)]="order.zip" required />
<span *ngIf="submitted && zip.invalid" class="text-danger">
Please enter your zip/postal code
</span>
</div>
<div class="form-group">
<label>Country</label>
<input class="form-control" #country="ngModel" name="country"
[(ngModel)]="order.country" required />
<span *ngIf="submitted && country.invalid" class="text-danger">
Please enter your country
</span>
</div>
<div class="text-center">
<button class="btn btn-secondary m-1" routerLink="/cart">Back</button>
<button class="btn btn-primary m-1" type="submit">Complete Order</button>
</div>
</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, визначених у лістингу 2.25, 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, почніть зі списку товарів та клацніть на одній із кнопок Add To Cart, щоб додати товар у кошик. Клацніть на кнопці Checkout; на екрані з'являється форма HTML, зображена на рис. 2.6. Спробуйте натиснути на кнопкуComplete Order, не вводячи текст у жодному полі, і ви отримаєте повідомлення про помилку перевірки даних. Заповніть форму та клацніть на кнопці Complete Order; з'явиться підтверджуюче повідомлення (рис. 2.6).

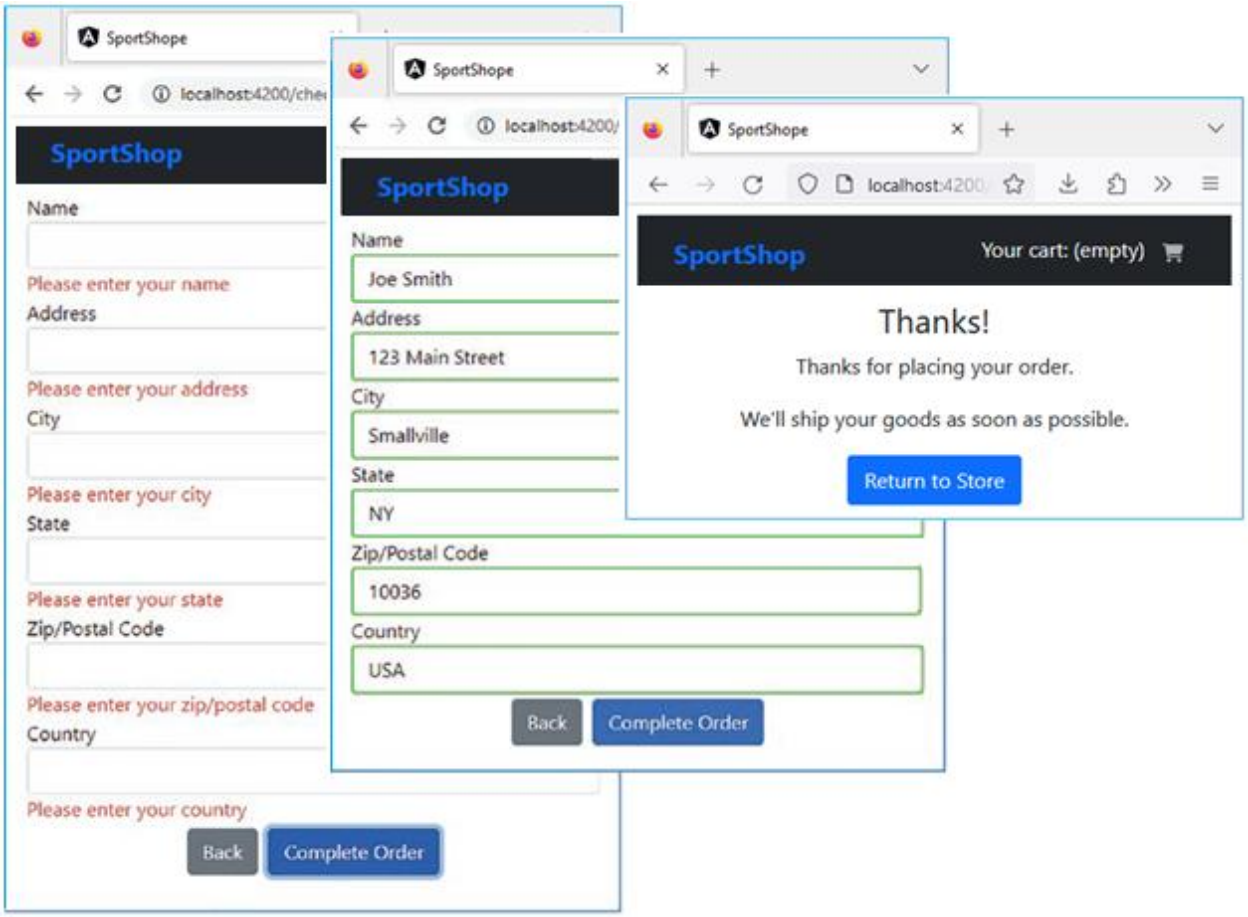

Рис. 2.6. Завершення замовлення

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}],
"itemCount":1,"cartPrice":100},
"shipped":false,
"name":"Joe Smith","address":"123 Main Street",
"city": "Smallville", "state": "NY", "zip": "10036", "country": "USA"
}

Використання REST-сумісної веб-служби

Тепер, коли базова функціональність SportShope 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, створіть файл rest.datasource.ts в папці SportShope/src/app/model та додайте код у лістингу 2.27.

Лістинг 2.27. Вміст файлу rest.datasource.ts у папці SportShope/app/model

```
import { Injectable } from "@angular/core";
import { HttpClient } from "@angular/common/http";
import { Observable } from "rxjs";
import { Product } from "./product.model";
import { Order } from "./order.model";
const PROTOCOL = "http";
const PORT = 3500;
@Injectable()
export class RestDataSource {
baseUrl: string;
constructor(private http: HttpClient) {
this.baseUrl = `${PROTOCOL}://${location.hostname}:${PORT}/`;
}
getProducts(): Observable<Product[]> {
return this.http.get<Product[]>(this.baseUrl + "products");
}
saveOrder(order: Order): Observable<Order> {
return this.http.post<Order>(this.baseUrl + "orders", order);
}
}
```

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 затримають обробку запиту і користувач буде дивитися на програму, яка не отримала даних. Щоб такого не було треба выдповыдним 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 лістингу 2.28 поведінка джерела даних змінюється у функціональному модулі моделі.

Лістинг 2.28.Зміна конфігурації служби у файлі model.module.ts

```
import { NgModule } від "@angular/core";
import { ProductRepository } від "./product.repository";
import { StaticDataSource } from "./static.datasource";
import { Cart } from "./cart.model";
import {Order} від "./order.model";
import { OrderRepository } від "./order.repository";
import { RestDataSource } from "./rest.datasource";
import { HttpClientModule } from "@angular/common/http";
@NgModule({
imports: [HttpClientModule],
providers: [ProductRepository, StaticDataSource, Cart, Order, OrderRepository,
{ provide: StaticDataSource, useClass: RestDataSource }]
})
export class ModelModule { }
```

Властивість imports використовується для оголошення залежності від функціонального модуля HttpModule, який надає службу Http, що використовується у лістингу 2.27. 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 (рис. 2.7).

Якщо пройти процедуру вибору товарів та оформлення замовлення, ви зможете переконатися, що джерело даних записало замовлення до веб-служби. Перейдіть за наступною URL-адресою:

http://localhost:3500/db

На екрані з'являється повний вміст бази даних, включаючи колекцію замовлень. Ви не зможете видати запит з URL /orders, тому що він вимагає автентифікації, налаштуванням якої ми займемося в наступній роботі.

Не забудьте, що дані, надані REST-сумісною веб-службою, скидаються під час перезапуску npm, з поверненням до початкових даних.

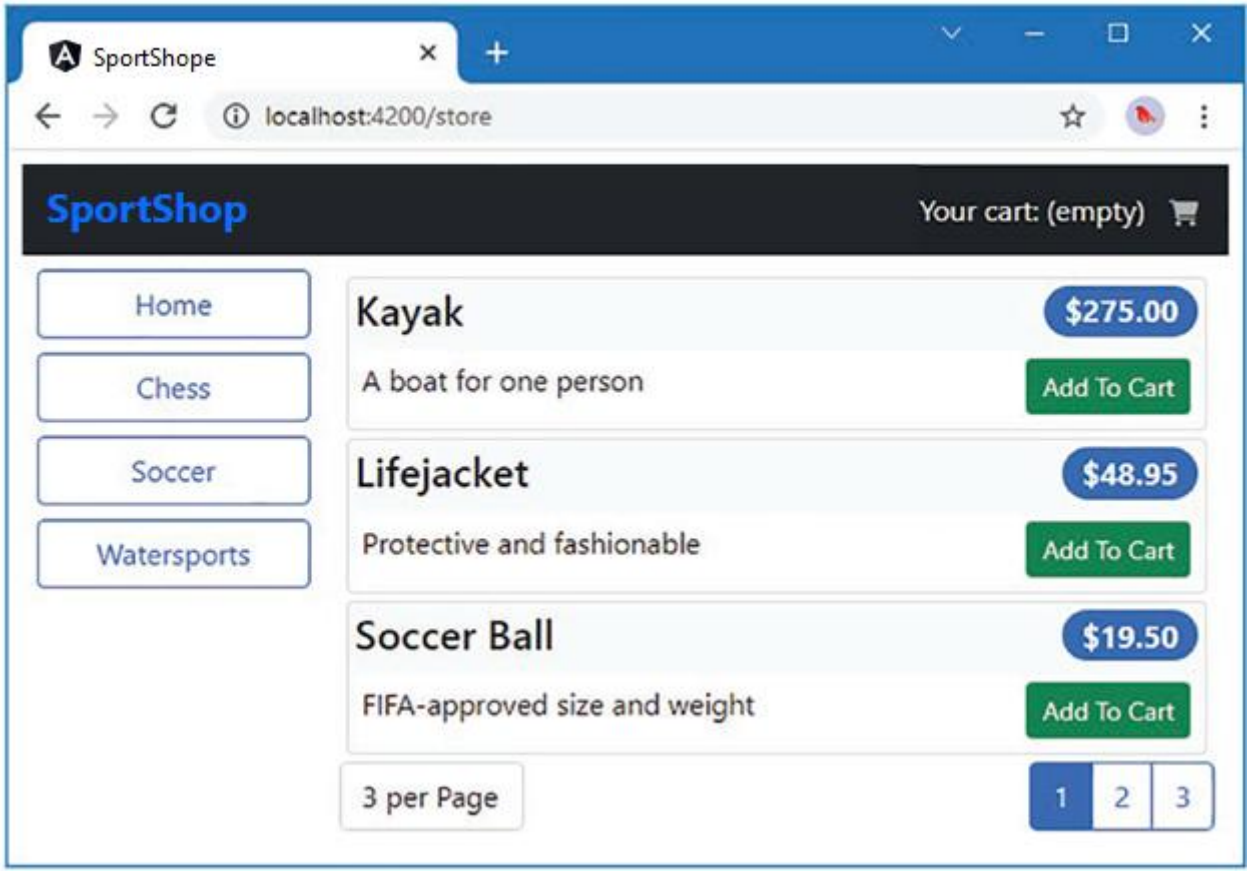

Рис. 2.7. Використання REST-сумісної веб-служби

В цій частині роботи ми продовжили розширювати функціональність програми SportShope: 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 REST-сумісній веб-службі.

III) Зробити звіт по роботі. Звіт повинен бути не менше 12 сторінок без титульного аркуша (шрифт Times New Roman, 14, полуторний інтервал). Титульний аркуш приводиться у додатку. Звіт повинен містити наступні розділи:

a) Огляд моделі даних Angular-додатку SportShop;
b) Огляд фіктивного джерела даних у додатку;
c) Детальний огляд компонентів магазину та шаблонів;
d) Огляд існуючих компонентів в додатку. Призначення, використання;
e) Огляд існуючих сервісів в додатку. Призначення, використання;
f) Огляд існуючих директив в додатку. Призначення, використання;
g) Можливі способи фільтрації товарів у додатку за категоріями;
h) Маршрутизація у фреймворку Angular;
i) Навігація у додатку SportShop;
j) REST-сумісні веб-служби: призначення, використання.

IV) Angular-додаток SportShop (Частина 1) розгорнути на платформі Firebase у проекті з ім’ям «ПрізвищеГрупаLaba8», наприклад «KovalenkoIP01Laba8».

ДОДАТОК

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

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Звіт по лабораторній роботі №____

_____

назва лабораторної роботи

з дисципліни: «Реактивне програмування»

Студент: _____

Група: _____

Дата захисту роботи: _____

Викладач: доц. Полупан Юлія Вікторівна

Захищено з оцінкою: _____

Київ, 2023