Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

Кафедра Програмного забезпечення комп'ютерних систем
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня

бакалавра
(назва освітньо-кваліфікаційного рівня)

студента *Величка Ігоря Юрійовича*
(ПІБ)

академічної групи *122-20-3*
(шифр)

спеціальності *122 Комп'ютерні науки*
(код і назва спеціальності)

освітньої програми *Комп'ютерні науки*
(назва освітньої програми)

на тему: *Розробка web-застосунку для сервісу доставки їжі за допомогою технологій React JS та ASP.NET Core*

Керівники	**Прізвище, ініціали**	**Оцінка за шкалою**		**Підпис**
		рейтинговою	**інституційною**	
кваліфікаційної роботи	*доц. Кабак Л.В.*			
розділів:				
спеціальний	*доц. Кабак Л.В.*			
економічний	*доц. Касьяненко Л.В.*			
Рецензент				
Нормоконтролер	*доц. Гуліна І.Г.*			

Дніпро
2024

Міністерство освіти і науки України
НТУ «Дніпровська політехніка»

ЗАТВЕРДЖЕНО:
завідувач кафедри
програмного забезпечення комп'ютерних систем
(повна назва)

М.О. Алексєєв
(підпис) (прізвище, ініціали)

« » 2024 року

ЗАВДАННЯ
на кваліфікаційну роботу

бакалавра
(назва освітньо-кваліфікаційного рівня)

студента *122-20-3* *Величка Ігоря Юрійовича*
(група) (прізвище та ініціали)

тема кваліфікаційної роботи *Розробка web-застосунку для сервісу доставки їжі за допомогою технологій React JS та ASP.NET Core*

затверджена наказом ректора НТУ «ДП» від *23.05.2024* № *470-с*

Розділ	Зміст виконання	Термін виконання
Спеціальний	***На основі матеріалів проєктно-технологічної практики та інших науково-технічних джерел провести аналіз стану рішення проблеми та постановку задачі. Обґрунтувати вибір та здійснити реалізацію методів вирішення проблеми***	***19.06.2024 р.***
Економічний	***Провести розрахунок трудомісткості розробки програмного забезпечення, витрат на створення ПЗ й тривалості його розробки***	***22.06.2024 р.***

Завдання видав _____ *Доц. Кабак Л.В.*
(підпис) (посада, прізвище, ініціали)

Завдання прийняв до виконання _____ *Величко І.Ю.*
(підпис) (прізвище, ініціали)

Дата видачі завдання: *14.01.2024 р.*

Термін подання кваліфікаційної роботи до ЕК: *24.06.2024 р.*

ЗМІСТ

РЕФЕРАТ

Пояснювальна записка: 109 с., 90 рис., 2 табл., 3 дод., 21 джерел.

Об'єкт розробки: web-застосунок для сервісу доставки їжі.

Мета дипломного проекту: розробка web-застосунку для сервісу доставки їжі за допомогою технологій React JS та ASP.NET Core.

У вступі розглядається аналіз та сучасний стан проблеми, конкретизується мета кваліфікаційної роботи та галузь її застосування, наведено обґрунтування актуальності теми та уточнюється постановка завдання.

У першому розділі проведено аналіз предметної області, визначено актуальність завдання та призначення розробки, розроблена постановка завдання, задані вимоги до програмної реалізації, технологій та програмних засобів.

У другому розділі виконано аналіз існуючих рішень, обрано вибір платформи для розробки, виконано проектування і розробка програми, наведено опис алгоритму і структури функціонування системи, визначені вхідні і вихідні дані, наведені характеристики складу параметрів технічних засобів, описаний виклик та завантаження застосунку, описана робота програми.

В економічному розділі визначено трудомісткість розробленої інформаційної підсистеми, проведений підрахунок вартості роботи по створенню застосунку та розраховано час на його створення.

Актуальність даного програмного забезпечення визначається необхідністю клієнтів закладів харчування в зручному веб-застосунку для створення замовлень та перегляду їх в особистому кабінеті.

Список ключових слів: ВЕБ-ЗАСТОСУНОК, REACT, C#, .NET, TYPESCRIPT, CSS, HTML, JSON, DOM, BOOTSTRAP, SQL SERVER.

ABSTRACT

Explanatory note: 109 p., 90 figures, 2 tables, 3 app., 21 sources.

Object of development: a web-application for the food delivery service.

The purpose of the diploma project: development of the web-application for the food delivery service using technologies React JS and ASP.NET Core.

The introduction considers the analysis and the current state of the problem, specifies the purpose of the qualification work and the field of its application, provides a justification for the relevance of the topic and clarifies the problem.

In the first section the analysis of the subject area is carried out, the urgency of the task and purpose of development is defined, the statement of the task is developed, requirements to software realization, technologies and software are set.

The second section analyzes the existing solutions, selects the platform for development, performs design and development of the program, describes the algorithm and structure of the system, determines the input and output data, provides the characteristics of the parameters of hardware, describes the call and application load, describes the program.

In the economic section, the complexity of the developed information subsystem is determined, the cost of work on creating the application is calculated and the time for its creation is calculated.

The relevance of this software is determined by the need of the food service clients to have a convenient web-application for creating orders and viewing their status using a personal cabinet.

List of keywords: WEB-APPLICATION, REACT, C#, .NET, TYPESCRIPT, CSS, HTML, JSON, DOM, BOOTSTRAP, SQL SERVER.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

CRUD – операції створення, читання, оновлення та видалення різного роду даних.

WEB API – програмний інтерфейс веб-застосунку.

React – відкрита JavaScript бібліотека для створення інтерфейсу користувача.

React Router – бібліотека для реалізації маршрутизації запитів у програмному забезпеченні, що використовує React.

C# – об'єктно-орієнтована мова програмування з безпечною системою типізації для платформи .NET. Є розробкою компанії Microsoft.

.NET – безкоштовний крос-платформний фреймворк з керованим кодом підтримуваний на Windows, Linux і Mac OSX. Підтримує мови C#, F# та Visual Basic.

ASP.NET Core – вільне та відкрите програмне забезпечення для створення веб-застосунків. Розроблене корпорацією Microsoft і співтовариством.

TypeScript – мова програмування, представлена Microsoft восени 2012; позиціонується як засіб розробки веб-застосунків, що розширює можливості JavaScript.

HTML – стандартизована мова розмітки документів для перегляду вебсторінок у браузері.

CSS – це спеціальна мова стилю сторінок, що використовується для опису їхнього зовнішнього вигляду.

JSON – текстовий формат обміну даними.

JWT – стандарт токена доступу на основі JSON.

Axios – бібліотека для мов програмування JavaScript та TypeScript для зручного надсилання HTTP запитів до Web API.

SQL – мова запитів, що використовується для маніпуляції реляційними базами даних.

LINQ – інтегрована мова запитів на платформі .NET, яка дозволяє маніпулювати колекціями даних.

DI – це шаблон проектування програмного забезпечення, який реалізує інверсію контролю для вирішення залежностей.

Cross-Origin Resource Sharing (CORS) – механізм безпеки сучасних браузерів, який дозволяє веб-сторінкам використовувати дані, що знаходяться на інших доменах.

СКБД – система керування базами даних.

ВСТУП

Розроблена інформаційна система призначена для застосування у сфері обслуговування покупців закладів харчування.

Кожна людина має потребу в харчуванні, але не в кожного є час, бажання або вміння приготувати смачні та поживні страви. Заклади харчування вирішують цю проблему, надаючи клієнтам можливість купити собі свіжозроблені страви на їх вибір.

Заклади харчування, зазвичай, надають можливість зробити замовлення у самому приміщенні або дистанційно, як правило, за допомогою дзвінків на мобільний телефон сервісу, але не кожен клієнт знаходить такий підхід зручним.

З розвитком інформаційних технологій, все більше людей почали користуватись інтернетом для пошуку інформації через пошукові системи, такі як Google, Bing, DuckDuckGo та інші. Відповідно, користувачам було б зручно не тільки шукати потрібний харчовий заклад в інтернеті, а ще й мати можливість здійснювати замовлення без дзвінків, використовуючи зручний користувацький інтерфейс веб-сайту.

Харчовий заклад також зацікавлений у побудові веб-застосунку разом з базою даних для збільшення ефективності та надійності ведення бізнесу.

Тому проблема, розглянута в даній кваліфікаційній роботі, є актуальною та має широке практичне значення.

Метою даної роботи є розробка веб-застосунку для закладів харчування з можливістю створення замовлень.

Інформаційна система має такі задачі в сфері обслуговування клієнтів:

- створення додаткового способу формування замовлень для покупців;
- покращення взаємодії між закладом та клієнтом через зручний користувацький інтерфейс веб-сайту;
- зменшення навантаження на операторів, що відповідають за оформлення замовлень клієнтів через дзвінки на мобільний телефон;

– побудова єдиної інформаційної бази даних страв, користувачів та замовлень для усунення проблеми ведення ручної документації даних.

В цілому, система забезпечує ефективність та надійність ведення комерційної діяльності закладом харчування, а також покращує зручність взаємодії потенційних клієнтів із відповідним закладом харчування. Розроблену інформаційну систему можуть використовувати всі користувачі, що бажають створити замовлення, а також адміністратори, що мають права для зміни стану бази даних.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ПОСТАНОВКА ЗАВДАННЯ

1.1. Загальні відомості з предметної галузі

В кожній країні існує безліч різних закладів харчування, що надають послуги покупки різних видів готової їжі. Компанії повинні попіклуватись про впровадження різних видів комунікацій між клієнтами та закладом для покращення клієнтського досвіду.

Перший варіант комунікації є телефонний дзвінок, що не є найзручнішим варіантом для деяких клієнтів і самого бізнесу, через велике навантаження мобільних операторів. Другим варіантом є використання веб або мобільних застосунків, що є більш привабливим як для великої кількості користувачів, через зручний інтерфейс, так і для бізнесу. В інтернет просторі вже існує декілька служб, що вирішують проблему потреби у застосунку. Наприклад, є такі служби доставляння: Glovo, Bolt Food та Loko, за допомогою яких користувачі можуть замовити бажані страви в обмеженій кількості закладів харчування (рис. 1.1 – 1.3).

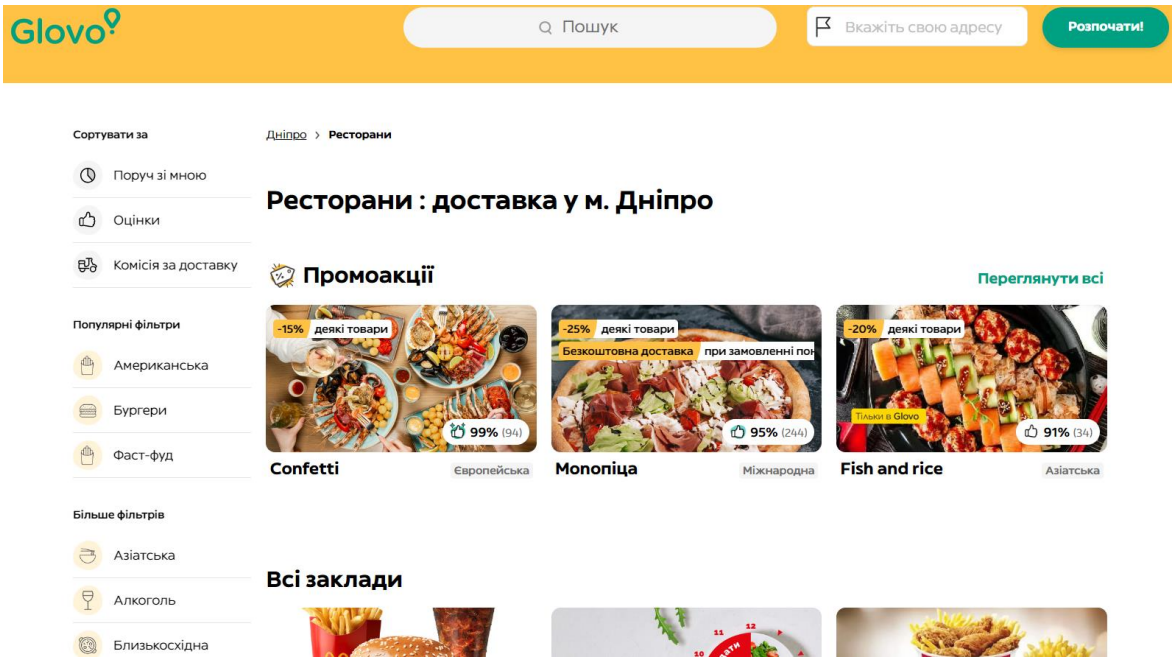


Рис. 1.1. Інтерфейс веб-застосунку сервісу Glovo.

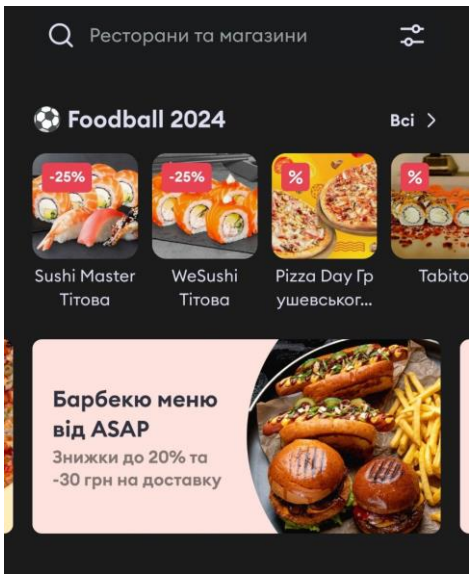


Рис. 1.2. Інтерфейс мобільного застосунку сервісу Bolt Food.

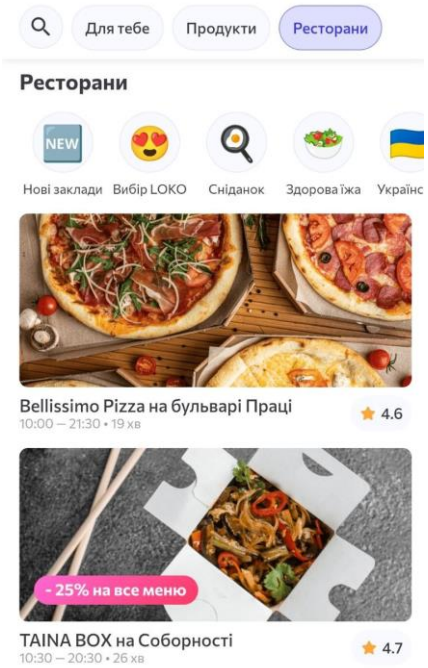


Рис. 1.3. Інтерфейс мобільного застосунку сервісу Loko.

Такі сервіси доставляння є зручним і готовим рішенням для бізнесу і для користувачів, через централізацію великої кількості інших харчових закладів в одному місці. Проте такі служби не можуть підійти усім та мають свої недоліки: Glovo надає можливість замовлення через веб-сайт або мобільний застосунок, а взаємодія з Bolt Food та Loko відбувається тільки через відповідні

мобільні застосунки; такі служби можуть не працювати в тих регіонах, де знаходиться сам харчовий заклад, роблячи взаємну працю неможливою.

Також у компаній з’являється проблема залежності від інших сервісів, котрі можуть раптово збанкрутувати або припинити свою роботу з інших непередбачуваних причин, що поставить компанію в неприємне становище.

1.2. Призначення розробки та галузь застосування

Темою бакалаврської кваліфікаційної роботи виступає: «Розробка web-застосунку для сервісу доставки їжі за допомогою технологій React JS та ASP.NET Core». Проаналізувавши потребу закладів харчування в додатковому способі комунікації з клієнтом та недоліки використання вже існуючих спільних служб для доставки, було прийнято рішення створити власне програмне забезпечення для конкретного підприємства з назвою «DokWok».

Система може бути використана для:

- огляду наявних позицій меню;
- додавання позицій меню до кошику;
- створення замовлення;
- створення особистого кабінету користувача;
- перегляду стану замовлень користувача;
- керування базою даних позицій меню, користувачів, замовлень та закладів підприємства адміністратором.

1.3. Підстави для розробки

Підставами для розробки та виконання кваліфікаційної роботи є:

- освітня програма 122 Комп’ютерні науки;
- навчальний план та графік навчального процесу;
- наказ ректора Національного технічного університету «Дніпровська політехніка» № 470-с від 23.05.2024 р;

– завдання на кваліфікаційну роботу на тему «Розробка web-застосунку для сервісу доставки їжі за допомогою технологій React JS та ASP.NET Core».

1.4. Постановка завдання

Метою кваліфікаційної роботи є розробка web-застосунку для сервісу доставки їжі, з можливістю формувати замовлення, створювати особистий кабінет користувача, слідкувати за станами своїх замовлень в створеному кабінеті, переглядати контактні дані з закладом харчування, інформацію про працюючи заклади підприємства.

Програмне забезпечення повинно мати такі веб-сторінки: загальне меню, меню за категоріями страв, контакти, про заклад, кошик, створення замовлення, особистий кабінет та адміністрація для керування базою даних системи.

Таблиця 1.1 містить сутності користувачів веб-застосунку.

Таблиця 1.1

Сутності користувачів веб-застосунку

Тип користувача	Доступний функціонал
Не авторизований	Огляд меню і контактів; додавання позицій до кошику; формування замовлення.
Покупець	Огляд меню і контактів; додавання позицій до кошику; формування замовлення; авторизація або реєстрація особистого кабінету; перегляд створених замовлень в кабінеті; зміна особистих даних.
Адміністратор	Огляд меню і контактів; додавання позицій до кошику; формування замовлення; авторизація або реєстрація особистого кабінету; перегляд створених замовлень в кабінеті; зміна особистих даних; керування записами продуктів, користувачів, замовлень та працюючих магазинів у базі даних.

Таблиця 1.2 містить сутності, з якими буде працювати веб-застосунок під час своєї роботи. Ці об’єкти будуть взяті як основа для створення сутностей бази даних та класових моделей програмного забезпечення.

Таблиця 1.2

Сутності для використання у веб-застосунку

Сутність	Властивості
Користувач	Ім’я, логін, електронна пошта, номер телефону, пароль
Замовлення	Ім’я покупця, номер телефону, електронна пошта, необов’язкова адреса доставлення, тип оплати, ціна замовлення, дата створення, статус, необов’язковий зв’язок з користувачем, необов’язковий зв’язок з магазином
Лінія замовлення	Зв’язок з замовленням, зв’язок з продуктом, кількість продуктів, ціна лінії замовлення
Продукт	Назва, ціна, вага, одиниця виміру ваги, опис, зв’язок з категорією продукту
Категорія продукту	Назва
Магазин	Вулиця, номер будівлі, час відкриття, час закриття

1.5. Вимоги до програми або програмного виробу

1.5.1. Вимоги до функціональних характеристик

Веб-застосунок повинен мати клієнтську та серверну частини.

Клієнтська частина повинна бути побудована за допомогою інструментів Vite, React та мови TypeScript. Бібліотека React Router має бути використаною для реалізації маршрутизації запитів. Веб-сторінки мають бути побудованими за допомогою HTML для структурування та CSS для стилізації. Дозволяється використання інструментів Material UI та Bootstrap з Sass для побудови окремих елементів або повноцінних сторінок клієнтської частини.

Серверна частина має бути RESTful Web API та створена за допомогою мови C# з використанням технології ASP.NET Core на платформі .NET. Для забезпечення зручного керування користувачами, потрібно використати ASP.NET Core Identity. Клієнтська частина надсилатиме HTTP запити з методами GET, POST, PUT або DELETE до серверної частини, щоб отримати відповідь, використовуючи бібліотеку «axios» для JavaScript і TypeScript. У якості системи керування базами даних буде використана Microsoft SQL Server. Entity Framework Core був обраний як ORM технологія для зв'язку сутностей бази даних та об'єктів класів серверу для зручного виконання операцій з базою даних, використовуючи LINQ. Взаємодія між сервісами серверу та Entity Framework Core відбувається через класи репозиторії, впроваджуючи використання Repository pattern [1]. Всі компоненти серверної частини повинні бути слабо залежними один від одного, спираючись на абстракції, та реалізовувати Dependency Injection (DI) pattern.

1.5.2. Вимоги до інформаційної безпеки

Веб-застосунок повинен мати сторінки з обмеженим доступом для не авторизованих користувачів. Користувачі матимуть змогу зареєструватись в системі з роллю «Покупець» та отримати доступ до особистого кабінету, де зможуть керувати своїми контактними даними та переглядати свої замовлення. Користувачі з роллю «Адміністратор» створюються поза користувацьким інтерфейсом, а дані для входу надаються адміністраторам, котрі будуть мати доступ до особистого кабінету та сторінки адміністрації з можливістю керувати записами бази даних.

Паролі користувачів повинні зберігатись у хешованому вигляді, використовуючи криптографічну «сіль», для запобігання отримання зловмисниками паролів клієнтів у випадку витоку бази даних.

Авторизація та аутентифікація повинні бути реалізовані, використовуючи сесії серверу та JWT [2, 16, 17]. Після аутентифікації даних користувача

відкривається сесія, генерується JWT з даними користувача та цей токен додається до сховища сесії, а клієнт отримує cookie з ID сесії у HTTP відповіді. У подальших запитах користувача буде використаний cookie сесії для авторизації до сторінок з обмеженим доступом.

1.5.3. Вимоги до складу та параметрів технічних засобів

Програмне забезпечення є веб-застосунком, тому може бути використаним на любому пристрої за умови наявності браузера.

Для забезпечення надійного функціонування програмного забезпечення на настільному комп’ютері він повинен відповідати таким вимогам:

- доступ до мережі Internet;
- наявність 4 Гб вільного місця на жорсткому диску;
- наявність щонайменше 4 Гб оперативної пам’яті;
- рідкокристалічний монітор з діагоналлю не менше 15" для відображення контенту;
- процесор з мінімальною частотою в 2 ГГц для коректної роботи браузеру;
- маніпулятор «миша»;
- клавіатура.

Для забезпечення надійного функціонування програмного забезпечення на мобільному телефоні він повинен відповідати таким вимогам:

- доступ до мережі Internet;
- наявність 4 Гб вільного місця на жорсткому диску;
- наявність щонайменше 4 Гб оперативної пам’яті;
- екран не менше 6" для відображення контенту;
- процесор з мінімальною частотою в 2 ГГц для коректної роботи браузеру;
- працюючий сенсорний екран;

Вище наведені характеристики є мінімальними для коректної та надійної роботи програмного забезпечення.

1.5.4. Вимоги до інформаційної та програмної сумісності

Для правильної роботи веб-застосунку на настільному комп’ютері він повинен відповідати таким вимогам:

- наявність браузера Google Chrome, Firefox, Opera, Microsoft Edge або іншого;
- встановлена операційна система Windows 7/10/11.

Для правильної роботи веб-застосунку на мобільному пристрої він повинен відповідати таким вимогам:

- наявність браузера Google Chrome, Firefox, Opera, Microsoft Edge, Safari або іншого;
- встановлена операційна система Android 13 і вище, або IOS 15 і вище.

Веб-застосунок повинен бути побудованим з клієнтської та серверної частини.

Клієнтська частина повинна бути реалізованою за допомогою бібліотек React, React Router, Axios, Bootstrap та Material UI з використанням TypeScript як мови програмування. HTML 5 та CSS 3 будуть виконувати роль структуризації та стилізації веб-сторінок застосунку. Для збірки React проєкту та його запуску буде використовуватись інструмент Vite.

Серверна частина повинна бути реалізованою на платформі .NET за допомогою фреймворків ASP.NET Core, Entity Framework Core, ASP.NET Core Identity та бібліотеки AutoMapper з використанням C# як мови програмування.

РОЗДІЛ 2
ПРОЄКТУВАННЯ ТА РОЗРОБКА ІНФОРМАЦІЙНОЇ СИСТЕМИ

2.1. Функціональне призначення системи

В ході виконання кваліфікаційної роботи буде розроблено веб-застосунок для створення замовлень із закладу харчування «DokWok». Метою створення даного застосунку є надання можливості покупцям закладу «DokWok» робити замовлення і відстежувати їх у персональному кабінеті через зручний інтерфейс веб-сайту. Додатково, бізнесу потрібно мати змогу зберігати всі дані у централізованій базі даних разом з користувацьким інтерфейсом для керування нею.

Користувачі веб-сайту зможуть передивлятись меню продуктів, додавати продукти до кошику та формувати замовлення надаючи контактні дані. За бажанням, користувач має можливість зареєструватись в системі та переглядати статус та інформацію про всі свої створені замовлення у персональному кабінеті. Особисті дані, включаючи пароль, можуть бути зміненими у персональному кабінеті у відповідній вкладці.

Адміністратор отримає свій особистий акаунт з відповідними правами: матиме доступ до сторінки адміністрації, де він зможе керувати станом бази даних системи, а саме записами продуктів, користувачів, замовлень та магазинів закладу.

2.2. Опис застосованих математичних методів

Математичні методи не були використані під час розробки програмного забезпечення, тому що особливості предметної області розв'язуваного завдання не передбачають застосування математичних методів, при розробці веб-застосунку для створення замовлень користувачами та керування базою даних системи.

2.3. Опис використаних технологій та мов програмування

Веб-застосунок для кваліфікаційної роботи складається з клієнтської та серверної частин. Серверна частина побудована за допомогою технологій ASP.NET Core, Entity Framework Core та ASP.NET Core Identity з використанням мови програмування C#. Microsoft SQL Server використовується як система керування базами даних.

ASP.NET Core [3] – це швидкий, надійний, крос-платформний фреймворк з відкритим кодом, що надає можливість будувати повноцінні або API веб-застосунки. Компанія розробник Microsoft постійно, з кожною новою версією, намагається максимально збільшити вже високу швидкість роботи фреймворку, що є великою перевагою під час вибору платформи розробки.

C# [4] – це об’єктно-орієнтована мова програмування, але вона може підтримувати й інші парадигми. C# використовує строгу типізацію, що підвищує надійність та зручність підтримки написаного програмного коду. C# є компільованою мовою та може бути використана для створення веб, настільних, мобільних та ігрових застосунків.

Microsoft SQL Server – це система керування базами даних, котра використовує Transact-SQL як мову запитів. Перевага цієї СКБД в тому, що вона має високу інтеграцію з іншими продуктами корпорації Microsoft, котрі використані для написання кваліфікаційної роботи.

Entity Framework Core [11] – це ORM технологія для зв’язку сутностей бази даних та об’єктів класів серверу для зручного виконання операцій з базою даних, використовуючи інтегровану мову запитів платформи .NET – LINQ [12]. Перевага такої технології закладається в тому, що вона дозволяє розробнику не використовувати мову запитів SQL для спілкування з базою даних, а користуватись LINQ для виконання CRUD запитів і міграціями для керування станом таблиць або інших елементів бази даних.

ASP.NET Core Identity [13] – це потужний інструмент керування користувачами та їх ролями. Перевага даної системи є в тому, що розробнику

не потрібно самому розробляти базу даних користувачів та реалізовувати хешування паролів з використанням криптографічної «солі» – весь цей функціонал одразу доступний після встановлення.

Клієнтська частина побудована за допомогою технологій React, React Router, Vite, HTML, CSS, Bootstrap та Material UI з використанням мови TypeScript, котра компілюється в JavaScript.

JavaScript – це прототипна інтерпретована мова програмування з динамічною типізацією, що здебільшого використовується у веб-розробці.

TypeScript [6] – це строго типізована мова програмування, що будується над JavaScript, роблячи створене програмне забезпечення більш надійним, на відміну від JavaScript.

React [7] – це відкрита бібліотека для JavaScript/TypeScript, що призначена для створення чуйного та реактивного користувацького інтерфейсу. Також бібліотека дозволяє створювати одно-сторінкові застосунки, де користувач працює в контексті одного HTTP запиту, а контент сторінки змінюється динамічно. React є зручним та легким для навчання, що є перевагою під час вибору інструмента розробки користувацького інтерфейсу.

React Router [8] – це бібліотека, що додає маршрутизацію запитів до програмного забезпечення побудованого за допомогою React.

Vite – це швидкий інструмент збірки, що дозволяє зручно запускати веб-застосунки, що були зроблені за допомогою технологій React, Vue, Svelte, Solid або інших.

HTML [9] – це стандартизована мова маркування, яку використовують для структуризації та відображення веб-сторінок.

CSS [10] – це мова стилю сторінок, що використовується для стилізації HTML документів.

Bootstrap [14] – це бібліотека, що надає користувачам готові CSS стилі та JS скрипти, котрі можна персоналізувати для своїх потреб, для пришвидшення побудови клієнтської частини застосунків. Bootstrap був використаний для

стилізації форм та деяких навігаційних меню у веб-застосунку, що розробляється. Приклад Bootstrap форми зображений на рис. 2.1.

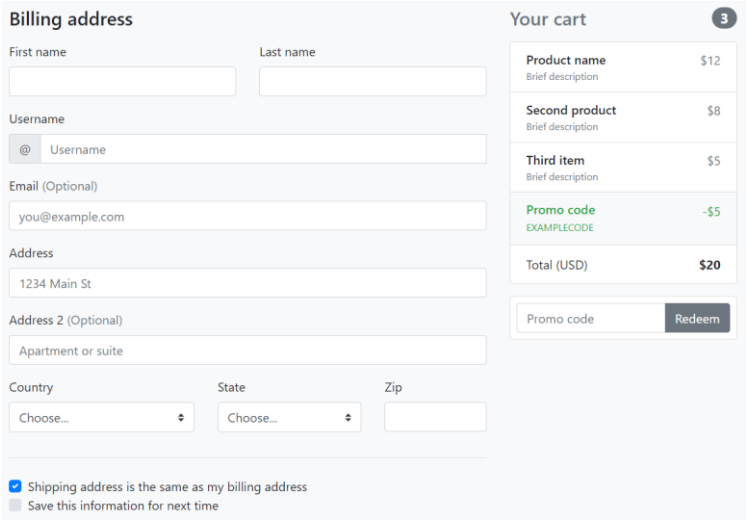


Рис. 2.1. Приклад форми створеної бібліотекою Bootstrap.

Material UI [15] – це бібліотека готових і стилізованих React компонентів, котрі можна персоналізувати за потребою, що дозволяє зекономити час на розробку компонентів користувацького інтерфейсу. Material UI був використаний для створення бічної панелі навігації та кнопки входу до особистого кабінету. Приклади Material UI компонентів наведені на рис. 2.2 – 2.3.

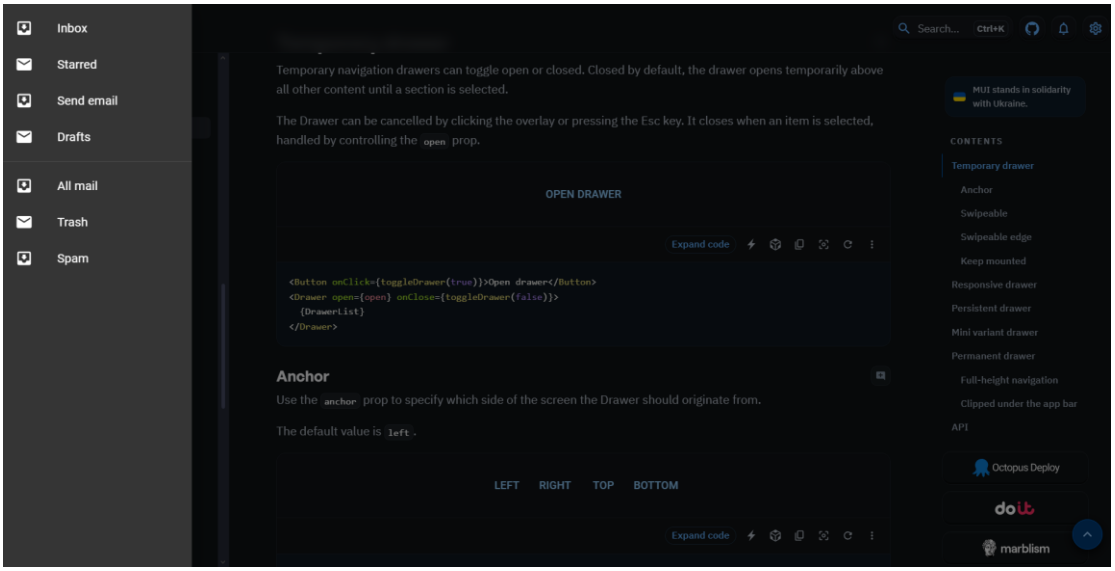


Рис. 2.2. Приклад бічної панелі навігації створеної бібліотекою Material UI.

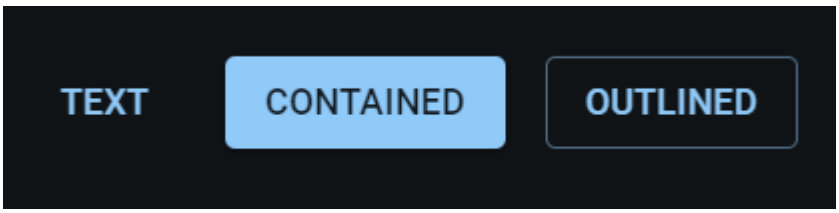


Рис. 2.3. Приклад кнопок створених бібліотекою Material UI.

Серверна частина являє собою RESTful Web API [5], тому для коректної розробки був використаний інструмент Swagger. Він допомагає створювати Open API Specification файл, котрий описує всі кінцеві точки API, і додатково генерувати візуальний інтерфейс для можливості тестування роботи кінцевих точок застосунку. Приклад вигляду користувацького інтерфейсу, що був побудований за допомогою Swagger, зображений на рис. 2.4.

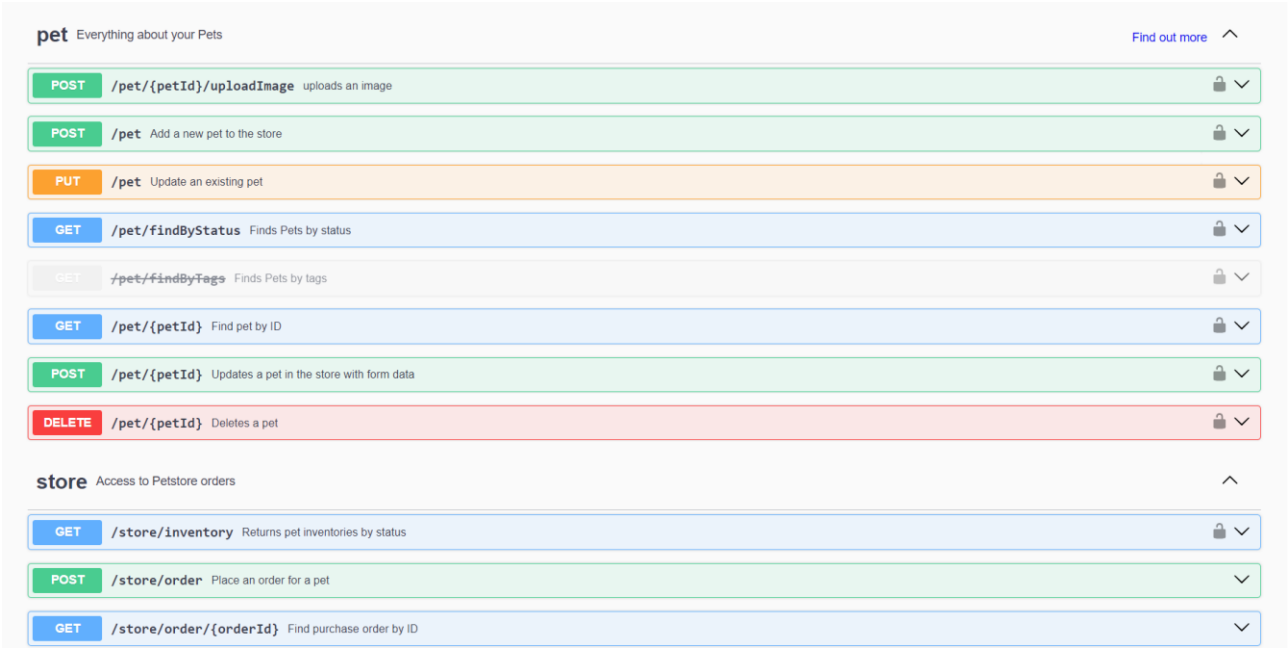


Рис. 2.4. Приклад створеного візуального інтерфейсу за допомогою Swagger.

2.4. Опис структури системи та алгоритмів її функціонування

Веб-застосунок, що розробляється, складається з двох частин: клієнтської та серверної. Клієнтською частиною являється React Project, а серверною – ASP.NET Core Web API, котра спілкується з базою даних. На рис. 2.5 зображена структура всього програмного забезпечення.

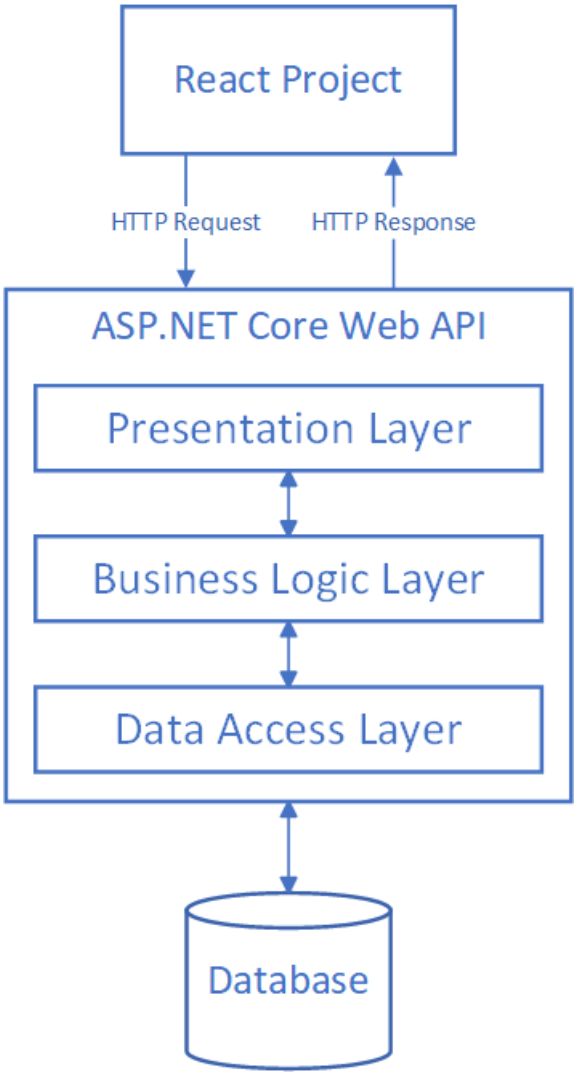


Рис. 2.5. Структура програмного забезпечення.

Спочатку розглянемо серверну частину. ASP.NET Core застосунок має трьох-шарову архітектуру і складається з таких елементів: шар доступу до даних (Data Access Layer), шар бізнес логікі (Business Logic Layer) та шар презентації (Presentation Layer). Кожен шар має свою функцію: Data Access

Layer відповідає за зв’язок з базою даних та виконання операцій над нею, не знає про існування інших шарів; Business Logic Layer відповідає за всю бізнес логіку застосунку, знає про існування Data Access Layer; Presentation Layer опрацьовує вхідні HTTP запити та формує відповідь на них, знає про існування Business Logic Layer. Багатошарова архітектура дозволяє розподілити відповідальності за кожну функціональність в окремий шар. Для зменшення залежності між шарами, класи реалізують дизайн патерн ін’єкції залежностей, що дозволяє класам спиратись на абстракції залежностей, а не на конкретні їх реалізації.

Data Access Layer повинен забезпечувати доступ до бази даних та надавати можливість маніпулювати нею. Для вирішення цього завдання шар використовує Entity Framework Core, ASP.NET Core Identity та патерн репозиторію.

Entity Framework Core підтримує дві моделі побудова системи: «code first», що надає можливість спочатку написати доменні класи, встановити між ними зв’язки і потім створити нову базу даних на основі цих класів за допомогою міграцій; «database first», де ви створюєте доменні класи, виставляєте між ними зв’язки спеціально під існуючу базу даних. Веб-застосунок використовує перший підхід «code first». Для роботи з Entity Framework Core програма потребує клас контексту бази даних (рис. 2.6), котрий описує саму базу даних, а саме надає доступ до таблиць та їх конфігурації.

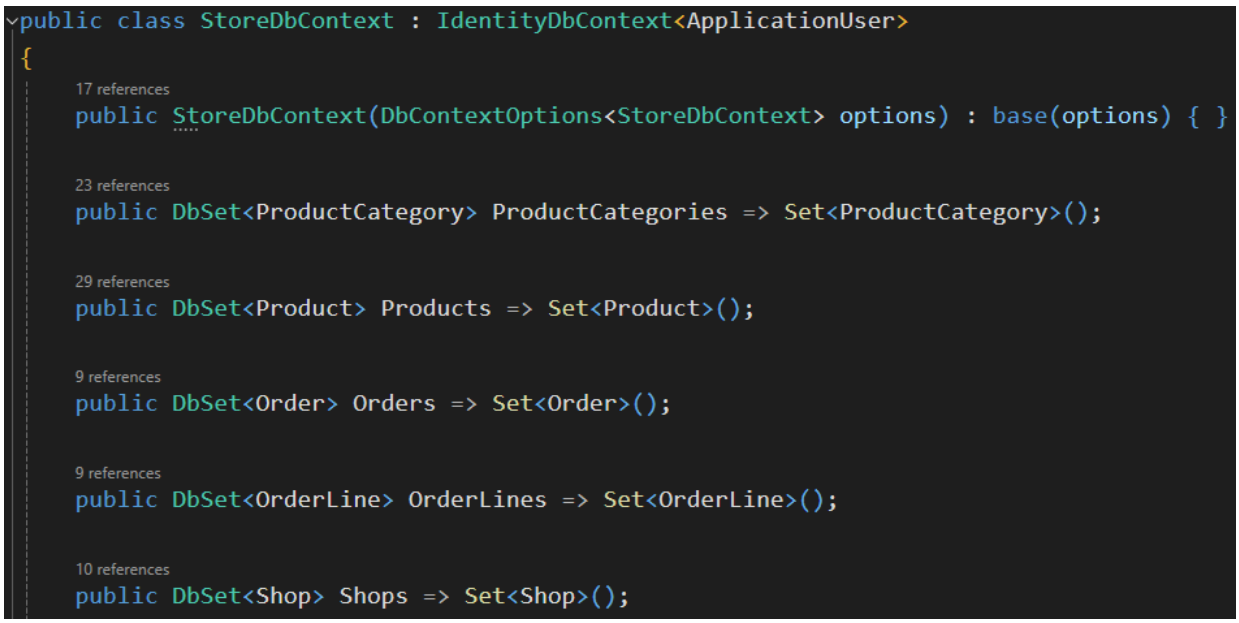


Рис. 2.6. Контекст бази даних.

На рис. 2.6 зображений клас, котрий виступатиме контекстом бази даних. Для цього він наслідує IdentityDbContext<ApplicationUser> клас, що не тільки робить його контекстом бази даних, а й додатково створить всі необхідні таблиці для керування користувачами ApplicationUser за допомогою інструменту ASP.NET Core Identity. Властивості цього класу типу DbSet описують одну таблицю бази даних та надають зручний інтерфейс для роботи з цією таблицею, тобто ми можемо звертатись до цих властивостей для виконання CRUD операцій над конкретною таблицею бази даних. Властивості DbSet також приймають тип доменного класу (рис. 2.7), котрий буде представляти сутність бази даних.

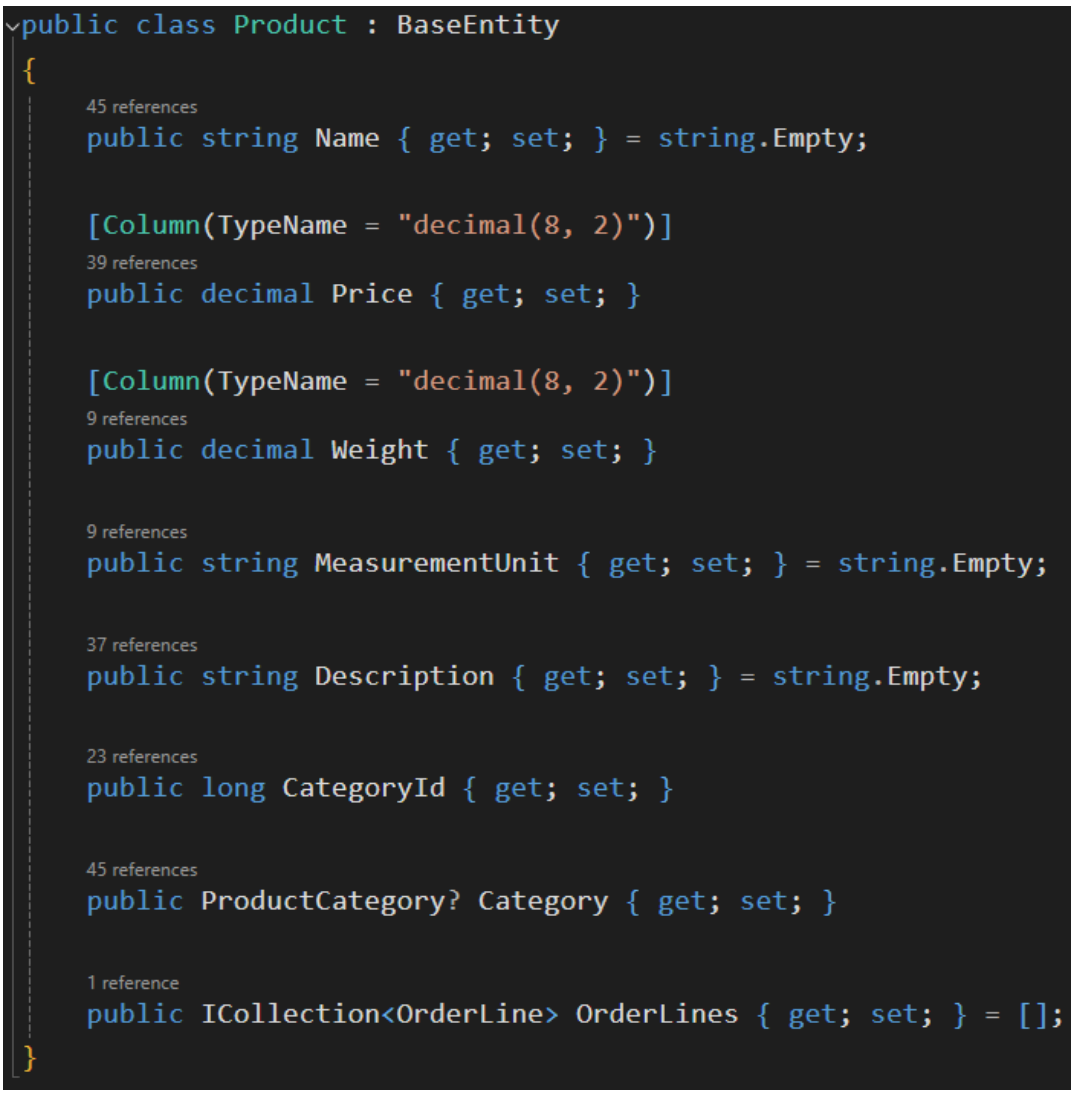


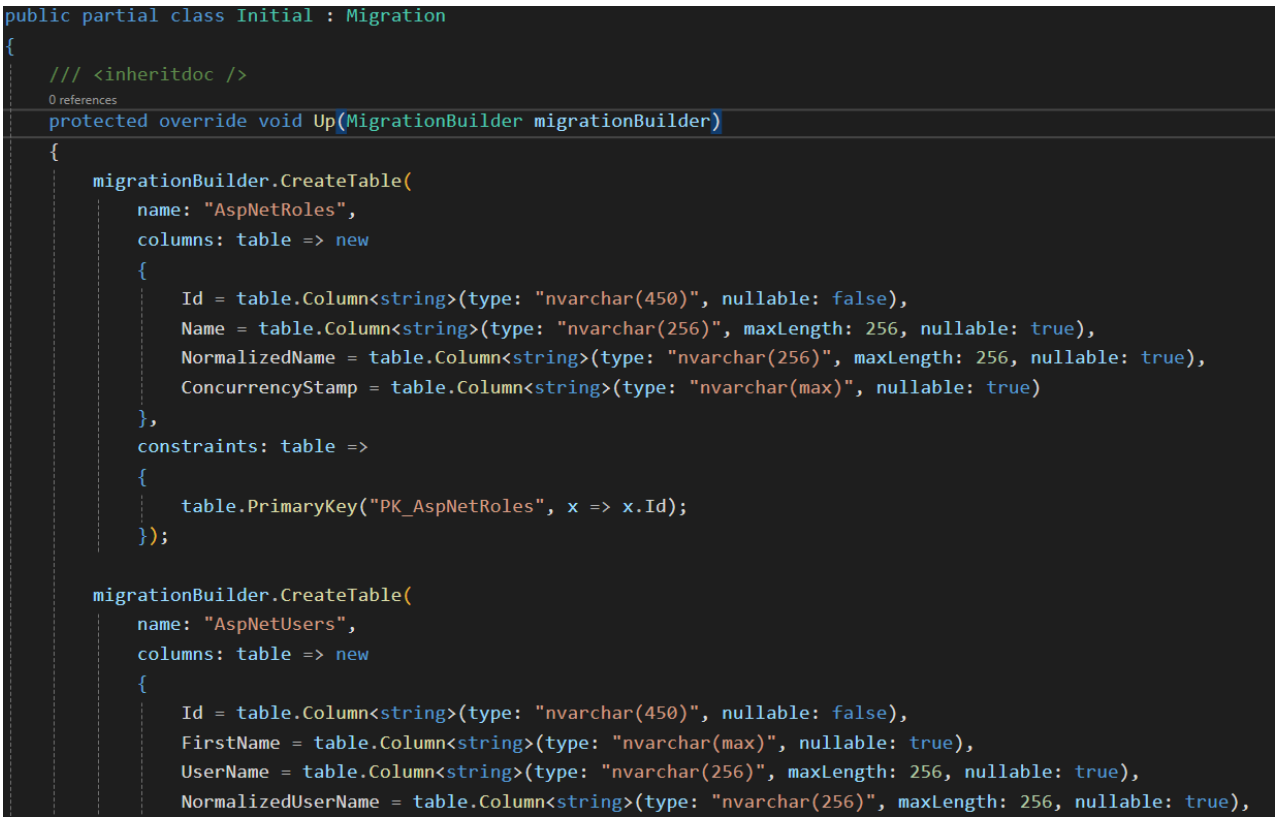
Рис. 2.7. Вигляд створеного доменного класу для Entity Framework Core.

Налаштування залежностей та обмежень сутностей виконується у спеціальному перевизначеному методі в класі контексту під назвою OnModelCreating, використовуючи Fluent API, котрий дозволяє ланцюжковим викликом методів виконати конфігурацію. На рис. 2.8 можна побачити вигляд конфігурації частини сутностей бази даних, котрим налаштовується зв’язок One-to-Many, обирається foreign key та встановлюється тип видалення залежних записів у разі видаленні головної сутності, а саме обмежений тип, де видалення головної сутності заборонено у випадку наявності залежних записів, або каскадний тип, де всі залежні записи видаляються разом із головною сутністю.

```
protected override void OnModelCreating(ModelBuilder builder)
{
    base.OnModelCreating(builder);

    builder.Entity<Product>()
        .HasMany(p => p.OrderLines)
        .WithOne(ol => ol.Product)
        .HasForeignKey(ol => ol.ProductId)
        .OnDelete(DeleteBehavior.Restrict);

    builder.Entity<ProductCategory>()
        .HasMany(c => c.Products)
        .WithOne(p => p.Category)
        .HasForeignKey(p => p.CategoryId)
        .OnDelete(DeleteBehavior.Restrict);

    builder.Entity<Order>()
        .HasMany(o => o.OrderLines)
        .WithOne(ol => ol.Order)
        .HasForeignKey(ol => ol.OrderId)
        .OnDelete(DeleteBehavior.Cascade);

    builder.Entity<ApplicationUser>()
        .HasMany(u => u.Orders)
        .WithOne(o => o.User)
        .HasForeignKey(o => o.UserId)
        .OnDelete(DeleteBehavior.Restrict);
```

Рис. 2.8. Вигляд конфігурації сутностей бази даних.

Після налаштування контексту бази даних, потрібно створити міграцію. Міграції описують структуру бази даних, використовуючи конфігурацію контексту. Щоб створити міграцію потрібно використати .NET CLI у командному рядку, а саме ввести команду встановлення інструментів Entity Framework «dotnet tool install --global dotnet-ef» і потім зробити міграцію за допомогою команди «dotnet ef migrations add Initial». Дана операція створить нову міграцію з назвою Initial. На рис. 2.9 відображений створений клас міграції з двома методами Up і Down. Метод Up описує інструкції, що повинні бути виконані щодо бази даних під час запуску міграції, а метод Down описує інструкції для виконання щодо бази даних під час відкату до попередньої міграції. Для виконання кваліфікаційної роботи було створена тільки одна міграція з назвою Initial.



```
public partial class Initial : Migration
{
    /// <inheritdoc />
    0 references
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "AspNetRoles",
            columns: table => new
            {
                Id = table.Column<string>(type: "nvarchar(450)", nullable: false),
                Name = table.Column<string>(type: "nvarchar(256)", maxLength: 256, nullable: true),
                NormalizedName = table.Column<string>(type: "nvarchar(256)", maxLength: 256, nullable: true),
                ConcurrencyStamp = table.Column<string>(type: "nvarchar(max)", nullable: true)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_AspNetRoles", x => x.Id);
            });

        migrationBuilder.CreateTable(
            name: "AspNetUsers",
            columns: table => new
            {
                Id = table.Column<string>(type: "nvarchar(450)", nullable: false),
                FirstName = table.Column<string>(type: "nvarchar(max)", nullable: true),
                UserName = table.Column<string>(type: "nvarchar(256)", maxLength: 256, nullable: true),
                NormalizedUserName = table.Column<string>(type: "nvarchar(256)", maxLength: 256, nullable: true),
```

Рис. 2.9. Вигляд створеного класу міграції.

Створені міграції потрібно виконати щоб вони змінили стан бази даних. Виконати міграцію можна за допомогою виклику команди «dotnet ef database update» у командному рядку, або, як реалізовано в проєкті, за допомогою метода «Migrate», котрий перевірить наявність нових міграцій і у випадку знаходження нових міграцій вони будуть виконані. У застосунку був створений клас з методом SeedDatabaseAsync (рис. 2.10), котрий викликає нові міграції методом Migrate та додає штучні записи до таблиць бази даних за умови, що вони пусті. Заповнення таблиць штучними записами було додано в контексті розробки, тому для випускної версії дані мають бути прибрані.

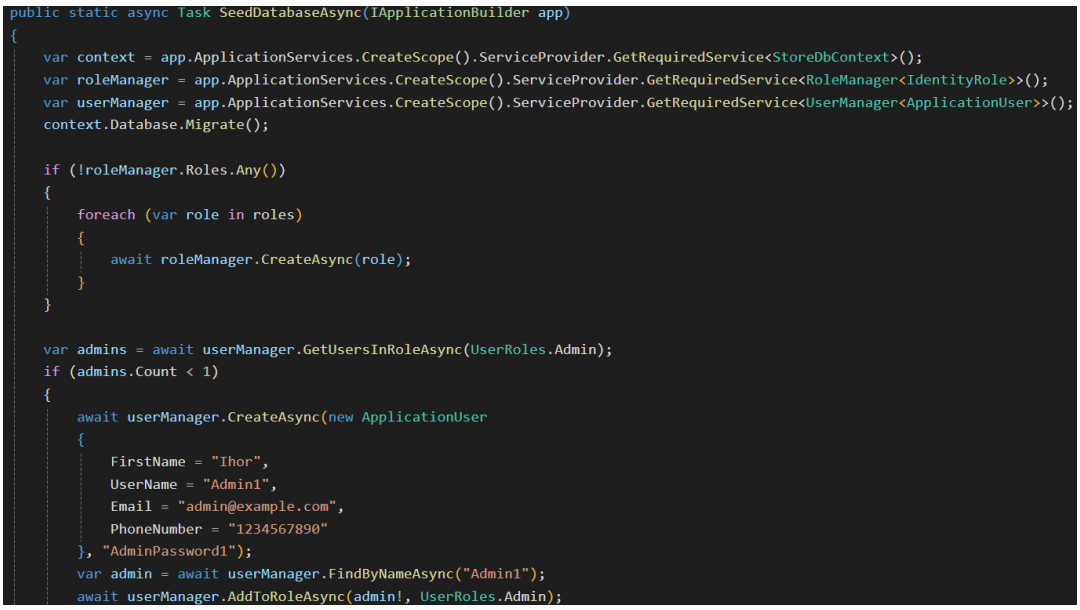


Рис. 2.10. Місце виконання міграцій та заповнення бази даних штучними даними.

Data Access Layer реалізує патерн репозиторію, тому для кожної сутності був створений відповідний клас репозиторію (рис. 2.11), котрий реалізує відповідний інтерфейс. Всі інтерфейси репозиторію наслідують один IRepository інтерфейс (рис. 2.12), котрий визначає базові CRUD операції, і, за бажанням, можуть додати свої додаткові методи. Всю структуру шару доступу до даних зображено на рис. 2.13.

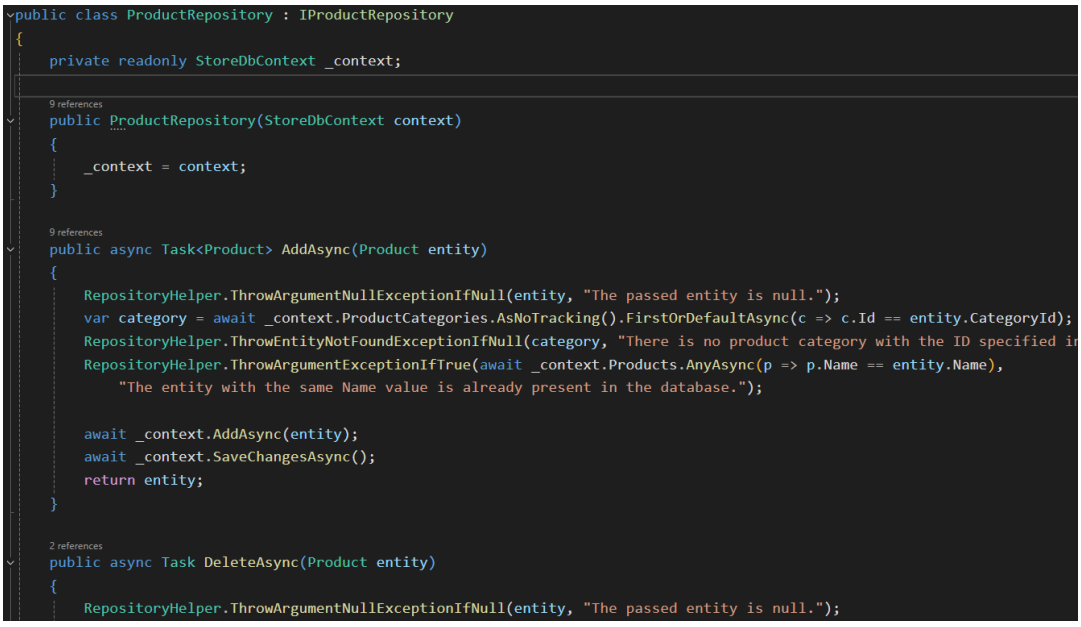


Рис. 2.11. Вигляд класу репозиторію.

```
public interface IRepository<TEntity> where TEntity : BaseEntity
{
    12 references
    IQueryable<TEntity> GetAll();

    12 references
    Task<TEntity?> GetByIdAsync(long id);

    14 references
    Task<TEntity> AddAsync(TEntity entity);

    16 references
    Task<TEntity> UpdateAsync(TEntity entity);

    7 references
    Task DeleteAsync(TEntity entity);

    14 references
    Task DeleteByIdAsync(long id);
}
```

Рис. 2.12. Спільні можливості всіх репозиторіїв у інтерфейсі IRepository.

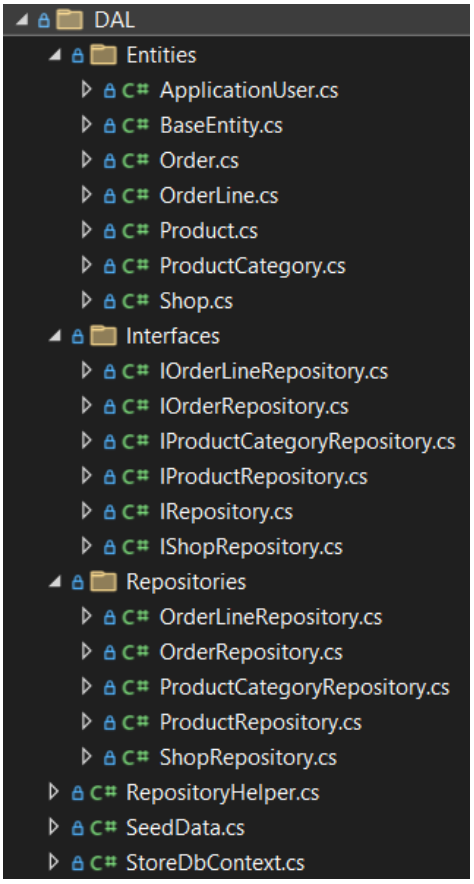


Рис. 2.13. Структура шару доступу до даних у файловій системі.

Business Logic Layer повинен містити основну логіку програми. Сервіси цього шару реалізують дизайн патерн ін’єкції залежностей, щоб класи застосунку було слабо залежними. Такий підхід супроводжується інверсією

створення об’єктів залежностей поза клас та отримання цих об’єктів через конструктор класу, властивість або метод. Класи також повинні спиратись на абстракції, а не на конкретні реалізації для збільшення гнучкості програми та можливості заміни типу реалізації на інший без потреби переробляти залежні класи.

У шарі бізнес логіки були створені класи сервіси, котрі визначають потрібний для споживача Web API функціонал, а саме сервіси для виконання CRUD операцій з базою даних, сервіс кошику, сервіс токенів безпеки і сервіс керування користувачами та їх авторизацією. Всі сервіси реалізовують свій відповідний інтерфейс. Частина інтерфейсів сервісів, задача котрих полягає в роботі з базою даних, наслідують інтерфейс ICrud (рис. 2.14), що визначає базові CRUD операції.

```
public interface ICrud<TModel> where TModel : class
{
    12 references
    Task<IEnumerable<TModel>> GetAllAsync();

    15 references
    Task<TModel?> GetByIdAsync(long id);

    12 references
    Task<TModel> AddAsync(TModel model);

    12 references
    Task<TModel> UpdateAsync(TModel model);

    12 references
    Task DeleteAsync(long id);
}
```

Рис. 2.14. Інтерфейс ICrud, що визначає базові CRUD операції.

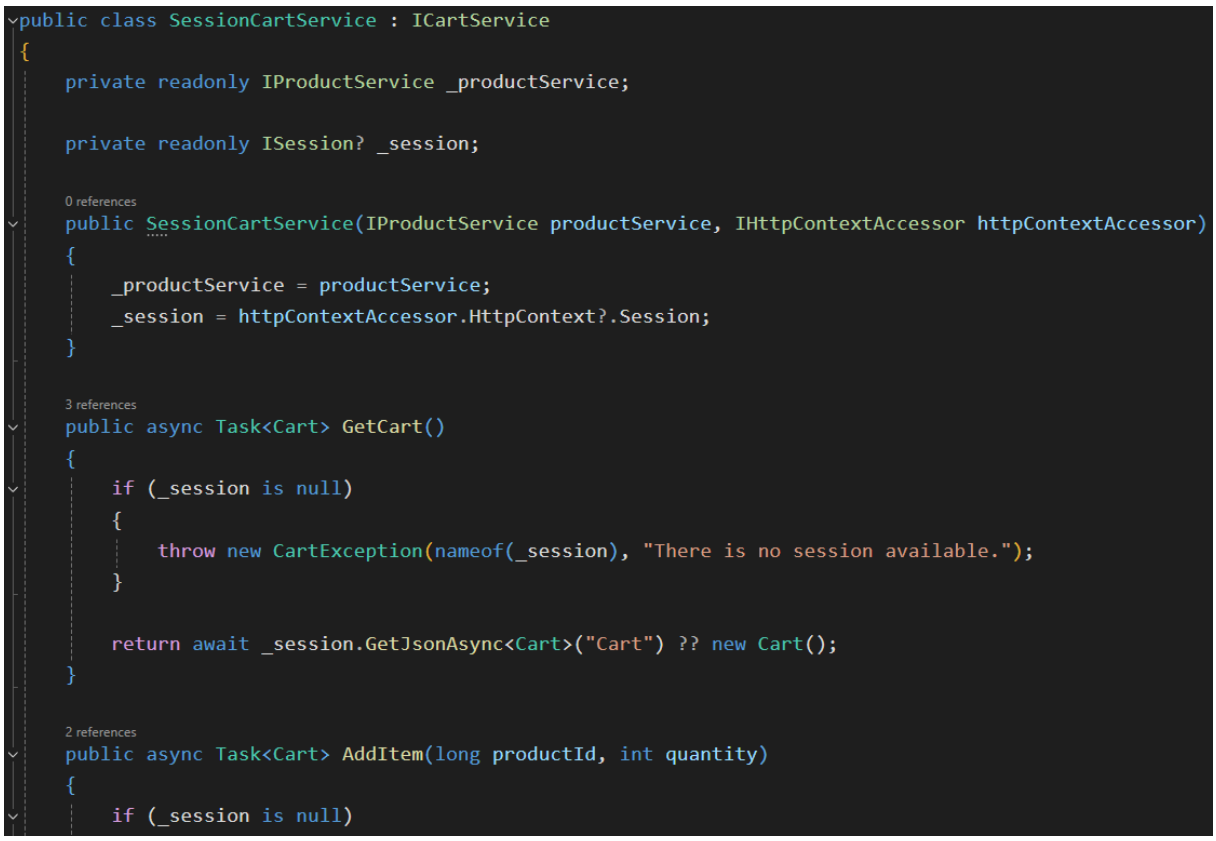
Сервіси, що потребують доступ до бази даних, будуть отримувати ін’єкцію потрібних репозиторіїв та використовувати їх через інтерфейси. Через те, що класи-сутності для Entity Framework Core та моделі шару бізнес логіки відрізняються, була використана бібліотека AutoMapper для зручного налаштування схем конвертації об’єктів одних типів у інші. На рис. 2.15

зображений сервіс, котрий отримує ін’єкції двох залежностей – об’єкти репозиторію та маппера, котрий надає зручний інтерфейс для конвертації об’єктів бізнес логіки у об’єкти сутностей Entity Framework Core.

```
public class ProductService : IProductService
{
    private readonly IProductRepository _repository;

    private readonly IMapper _mapper;

    5 references
    public ProductService(IProductRepository repository, IMapper mapper)
    {
        _repository = repository;
        _mapper = mapper;
    }

    6 references
    public async Task<ProductModel> AddAsync(ProductModel model)
    {
        ServiceHelper.ThrowArgumentNullExceptionIfNull(model, "The passed model is null.");
        var entity = _mapper.Map<Product>(model);
        var addedEntity = await _repository.AddAsync(entity);
        var addedEntityWithDetails = await _repository.GetByIdWithDetailsAsync(addedEntity.Id);
        return _mapper.Map<ProductModel>(addedEntityWithDetails);
    }

    7 references
    public async Task DeleteAsync(long id)
    {
        await _repository.DeleteByIdAsync(id);
    }
```

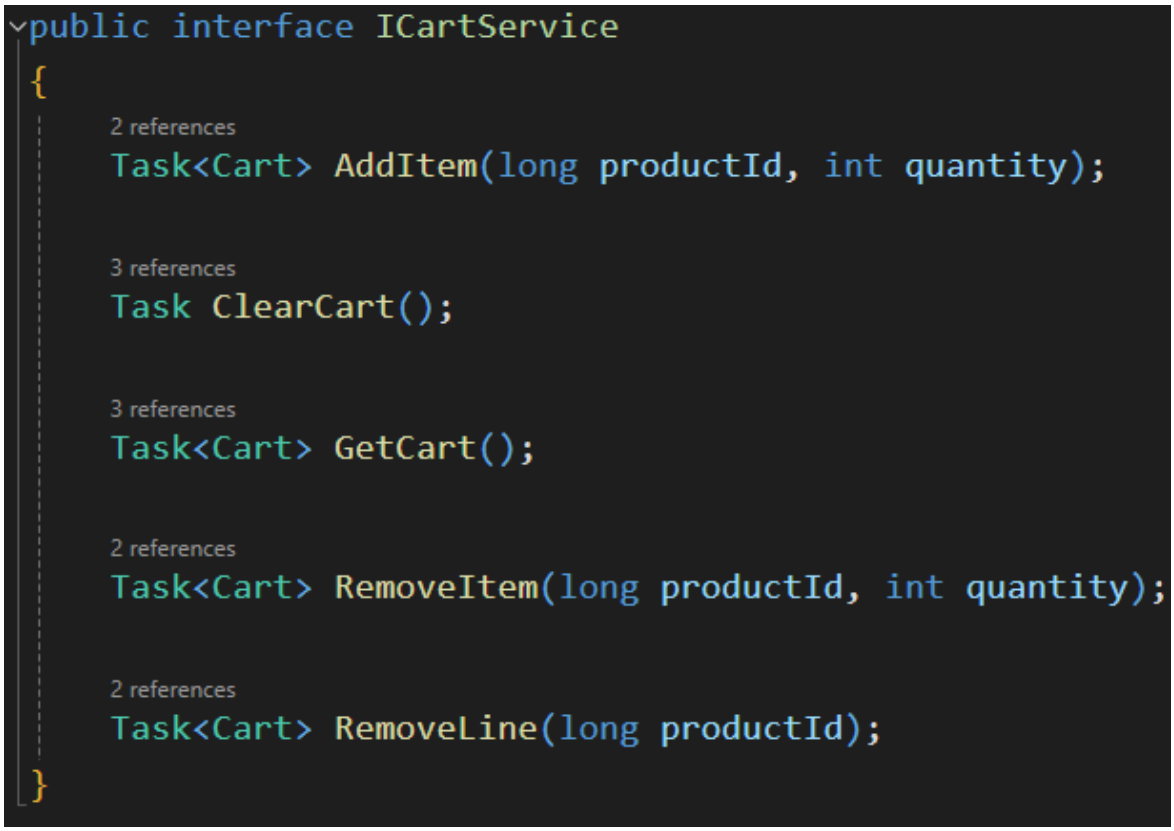
Рис. 2.15. Вигляд сервісу шару бізнес логіки.

Сервіс кошику SessionCartService (рис. 2.16) надає можливість додавати і видаляти продукти кошику, отримувати об’єкт кошику та очищувати його. Кошик зберігається у сериалізованому вигляду в сховищі сесії. Цей клас потребує об’єкт продуктового сервісу та об’єкт для роботи з сесією. Сервіс реалізує інтерфейс ICartService (рис. 2.17), що визначає його функціонал.



```
public class SessionCartService : ICartService
{
    private readonly IProductService _productService;

    private readonly ISession? _session;

    0 references
    public SessionCartService(IProductService productService, IHttpContextAccessor httpContextAccessor)
    {
        _productService = productService;
        _session = httpContextAccessor.HttpContext?.Session;
    }

    3 references
    public async Task<Cart> GetCart()
    {
        if (_session is null)
        {
            throw new CartException(nameof(_session), "There is no session available.");
        }

        return await _session.GetJsonAsync<Cart>("Cart") ?? new Cart();
    }

    2 references
    public async Task<Cart> AddItem(long productId, int quantity)
    {
        if (_session is null)
```

Рис. 2.16. Сервіс кошику.



```
public interface ICartService
{
    2 references
    Task<Cart> AddItem(long productId, int quantity);

    3 references
    Task ClearCart();

    3 references
    Task<Cart> GetCart();

    2 references
    Task<Cart> RemoveItem(long productId, int quantity);

    2 references
    Task<Cart> RemoveLine(long productId);
}
```

Рис. 2.17. Інтерфейс функціоналу сервісу кошику.

Оскільки в авторизації та автентифікації задіяний JWT, то був створений відповідний JwtService клас, що реалізує інтерфейс ISecurityTokenService (рис. 2.18), котрий дозволяє створювати токени для авторизованих користувачів та перевіряти токени на дійсність. Сервіс має два методи: CreateToken (рис. 2.19), що дозволяє створити JWT за допомогою секретного ключа та даних переданого користувача; ValidateToken (рис. 2.20), що дозволяє виконати перевірку JWT на дійсність, використовуючи той самий секретний ключи, котрий був використаний під час створення JWT. Всі потрібні дані для побудови JWT записані у файл конфігурації appsettings.json (рис. 2.21), а саме об'єкт «Jwt» з властивістю «Key», котрий буде кодуватись у формат UTF8 як послідовність байтів, а також властивостями «Issuer», «Audience» та «Subject», що є обов'язковою частиною у побудові та перевірці токену. Окрім JWT даних у файлі конфігурації знаходиться налаштування логування; рядок підключення до бази даних та дозволені URL, з яких можна можна надсилати HTTP запити до цього Web API. Ці дані доступні через сервіс IConfiguration та будуть використані у Presentation Layer, пізніше у цьому розділі.

```
public interface ISecurityTokenService<in TUser, out IToken>
    where TUser : class
    where IToken : SecurityToken
{
    4 references
    string CreateToken(TUser user);

    4 references
    IToken ValidateToken(string token);
}
```

Рис. 2.18. Інтерфейс функціоналу JWT сервісу.

```
public string CreateToken(UserModel user)
{
    const int ExpirationDays = 1;
    DateTime expiration = DateTime.UtcNow.AddDays(ExpirationDays);

    var encodedKey = Encoding.UTF8.GetBytes(_configuration["Jwt:Key"]!);
    SymmetricSecurityKey securityKey = new(encodedKey);
    SigningCredentials tokenSigningCredentials = new(securityKey, SecurityAlgorithms.HmacSha256);

    Claim[] claims = [
        new(JwtRegisteredClaimNames.Sub, _configuration["Jwt:Subject"]!),
        new(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString()),
        new(JwtRegisteredClaimNames.Iat, DateTime.UtcNow.ToString()),
        new("id", user.Id ?? string.Empty),
        new("username", user.UserName ?? string.Empty),
    ];

    var token = new JwtSecurityToken(
        _configuration["Jwt:Issuer"]!,
        _configuration["Jwt:Audience"]!,
        claims,
        expires: expiration,
        signingCredentials: tokenSigningCredentials);

    var tokenHandler = new JwtSecurityTokenHandler();
    return tokenHandler.WriteToken(token);
}
```

Рис. 2.19. Метод для створення JWT.

```
public JwtSecurityToken ValidateToken(string token)
{
    var tokenHandler = new JwtSecurityTokenHandler();
    var key = Encoding.UTF8.GetBytes(_configuration["Jwt:Key"]!);
    tokenHandler.ValidateToken(token, new TokenValidationParameters
    {
        ValidateIssuer = true,
        ValidateAudience = true,
        ValidateLifetime = true,
        ValidateIssuerSigningKey = true,
        ValidIssuer = _configuration["Jwt:Issuer"],
        ValidAudience = _configuration["Jwt:Audience"],
        IssuerSigningKey = new SymmetricSecurityKey(key),
        ClockSkew = TimeSpan.Zero
    }, out SecurityToken validatedToken);

    var jwtToken = (JwtSecurityToken)validatedToken;
    return jwtToken;
}
```

Рис. 2.20. Метод для перевірки JWT на дійсність.

Рис. 2.21. Файл конфігурації застосунку.

Сервіс користувачів UserService (рис. 2.22) надає можливість створювати, оновлювати, видаляти та отримувати об’єкти користувачів, а також функціонал авторизації, реєстрації та перевірки чи є користувач авторизованим. Даний сервіс потребує чотири залежності: маппер для конвертації об’єктів, менеджер користувачів, наданий ASP.NET Core Identity, сервіс токенів безпеки та об’єкт сесії. UserService реалізує інтерфейс IUserService (рис. 2.23), що визначає його функціонал.

```
public class UserService : IUserService
{
    private readonly UserManager<ApplicationUser> _userManager;

    private readonly IMapper _mapper;

    private readonly ISecurityTokenService<UserModel, JwtSecurityToken> _securityTokenService;

    private readonly ISession? _session;

    0 references
    public UserService(UserManager<ApplicationUser> userManager, IMapper mapper,
        ISecurityTokenService<UserModel, JwtSecurityToken> securityTokenService,
        IHttpContextAccessor httpContextAccessor)
    {
        _userManager = userManager;
        _mapper = mapper;
        _securityTokenService = securityTokenService;
        _session = httpContextAccessor.HttpContext?.Session;
    }

    3 references
    public async Task<UserModel> AddAsync(UserModel model, string password)
    {
        model = ServiceHelper.ThrowArgumentNullExceptionIfNull(model, "The passed model is null.");
        ServiceHelper.ThrowUserExceptionIfTrue(model.PhoneNumber is null, "The phone number is null");
        bool isPhoneNumberTaken = await _userManager.Users.AnyAsync(u => u.PhoneNumber == model.PhoneNumber);
        ServiceHelper.ThrowUserExceptionIfTrue(isPhoneNumberTaken, "The phone number is already taken");
```

Рис. 2.22. Вигляд сервісу користувачів.

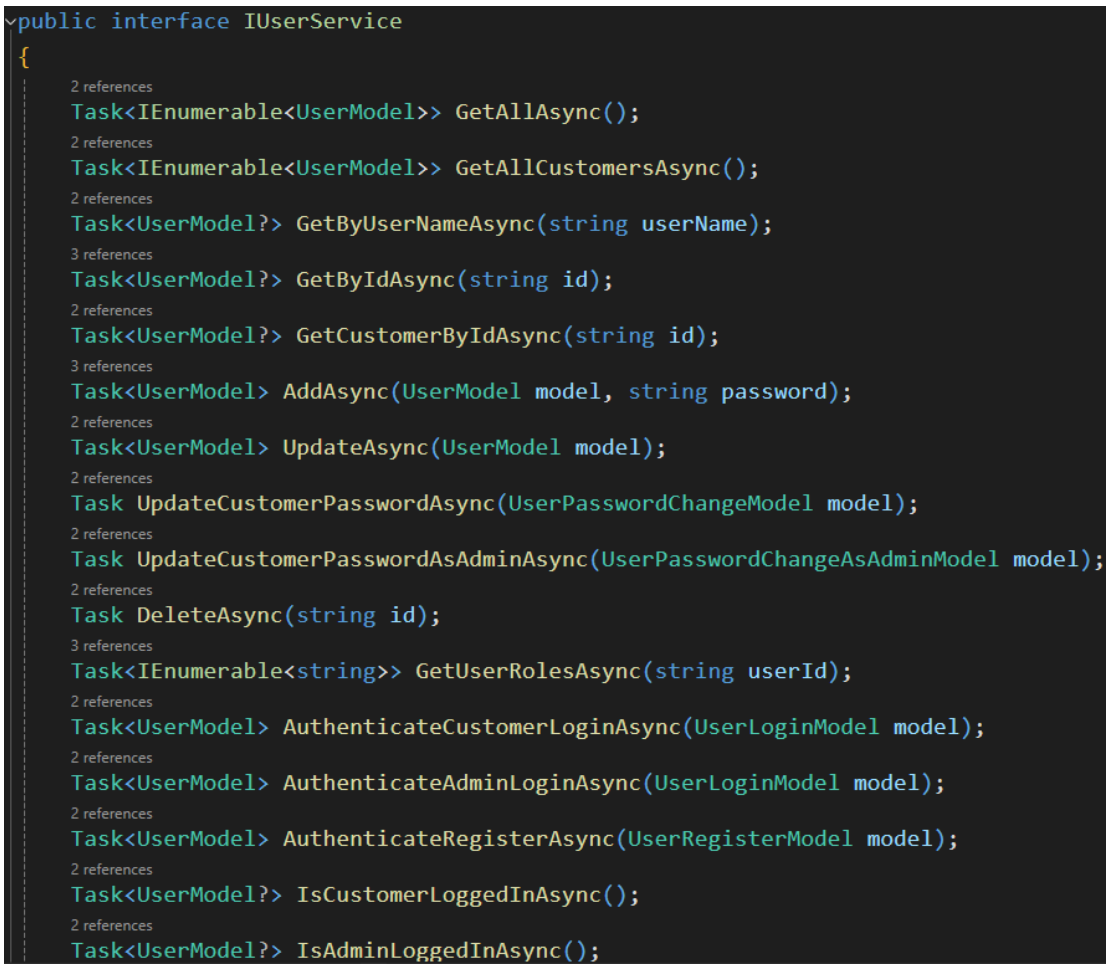


Рис. 2.23. Інтерфейс функціоналу сервісу користувачів.

Процес авторизації виконаний у двох методах сервісу: AuthenticateCustomerLoginAsync (рис. 2.24), для авторизації користувачів з роллю «Покупець» та «Адміністратор», та AuthenticateAdminLoginAsync (рис. 2.25), для авторизації користувачів з роллю «Адміністратор». У випадку коректних даних користувача – він авторизується та JWT, створений для нього, додається до сесії, інакше – в доступі буде відмовлено.

```
public async Task<UserModel> AuthenticateCustomerLoginAsync(UserLoginModel model)
{
    model = ServiceHelper.ThrowArgumentNullExceptionIfNull(model, "The passed model is null.");

    var user = await _userManager.FindByNameAsync(model.UserName);
    user = RepositoryHelper.ThrowEntityNotFoundExceptionIfNull(user, "The credentials are wrong.");

    bool isCustomer = await _userManager.IsInRoleAsync(user, UserRoles.Customer);
    bool isAdmin = await _userManager.IsInRoleAsync(user, UserRoles.Admin);
    ServiceHelper.ThrowUserExceptionIfUserIsNotValid(isCustomer || isAdmin, "The user is not allowed.");

    var isValidPassword = await _userManager.CheckPasswordAsync(user, model.Password);
    ServiceHelper.ThrowUserExceptionIfUserIsNotValid(isValidPassword, "The credentials are wrong.");

    var userModel = _mapper.Map<UserModel>(user);
    var token = _securityTokenService.CreateToken(userModel);
    if (_session is null)
    {
        throw new SessionException(nameof(_session), "The session is null");
    }

    await _session.SetStringAsync("userToken", token);
    return userModel;
}
```

Рис. 2.24. Метод авторизації покупців та адміністраторів.

```
public async Task<UserModel> AuthenticateAdminLoginAsync(UserLoginModel model)
{
    model = ServiceHelper.ThrowArgumentNullExceptionIfNull(model, "The passed model is null.");

    var user = await _userManager.FindByNameAsync(model.UserName);
    user = RepositoryHelper.ThrowEntityNotFoundExceptionIfNull(user, "The credentials are wrong.");

    bool isAdmin = await _userManager.IsInRoleAsync(user, UserRoles.Admin);
    ServiceHelper.ThrowUserExceptionIfUserIsNotValid(isAdmin, "The user is not allowed.");

    var isValidPassword = await _userManager.CheckPasswordAsync(user, model.Password);
    ServiceHelper.ThrowUserExceptionIfUserIsNotValid(isValidPassword, "The credentials are wrong.");

    var userModel = _mapper.Map<UserModel>(user);
    var token = _securityTokenService.CreateToken(userModel);
    if (_session is null)
    {
        throw new SessionException(nameof(_session), "The session is null");
    }

    await _session.SetStringAsync("userToken", token);
    return userModel;
}
```

Рис. 2.25. Метод авторизації тільки адміністраторів.

Реєстрація виконується за допомогою метода AuthenticateRegisterAsync (рис. 2.26). Якщо користувача з наданими даними не існує і самі дані є коректними, то користувач реєструється в системі та додається до бази даних із хешованим паролем.

```
public async Task<UserModel> AuthenticateRegisterAsync(UserRegisterModel model)
{
    model = ServiceHelper.ThrowArgumentNullExceptionIfNull(model, "The passed model is null.");

    var userModel = _mapper.Map<UserModel>(model);
    userModel = await AddAsync(userModel, model.Password!);

    var token = _securityTokenService.CreateToken(userModel);
    if (_session is null)
    {
        throw new SessionException(nameof(_session), "The session is null");
    }

    await _session.SetStringAsync("userToken", token);
    return userModel;
}
```

Рис. 2.26. Метод реєстрації користувачів.

Для виходу авторизованого користувача із системи існує метод LogOutAsync (рис. 2.27), котрий видаляє JWT користувача зі сховища сесії.

Повна структура шару бізнес логіки зображена на рис. 2.28.

```
public async Task LogOutAsync()
{
    if (_session is null)
    {
        throw new SessionException(nameof(_session), "The session is null");
    }

    await _session.RemoveAsync("userToken");
}
```

Рис. 2.27. Метод виходу користувача із системи.

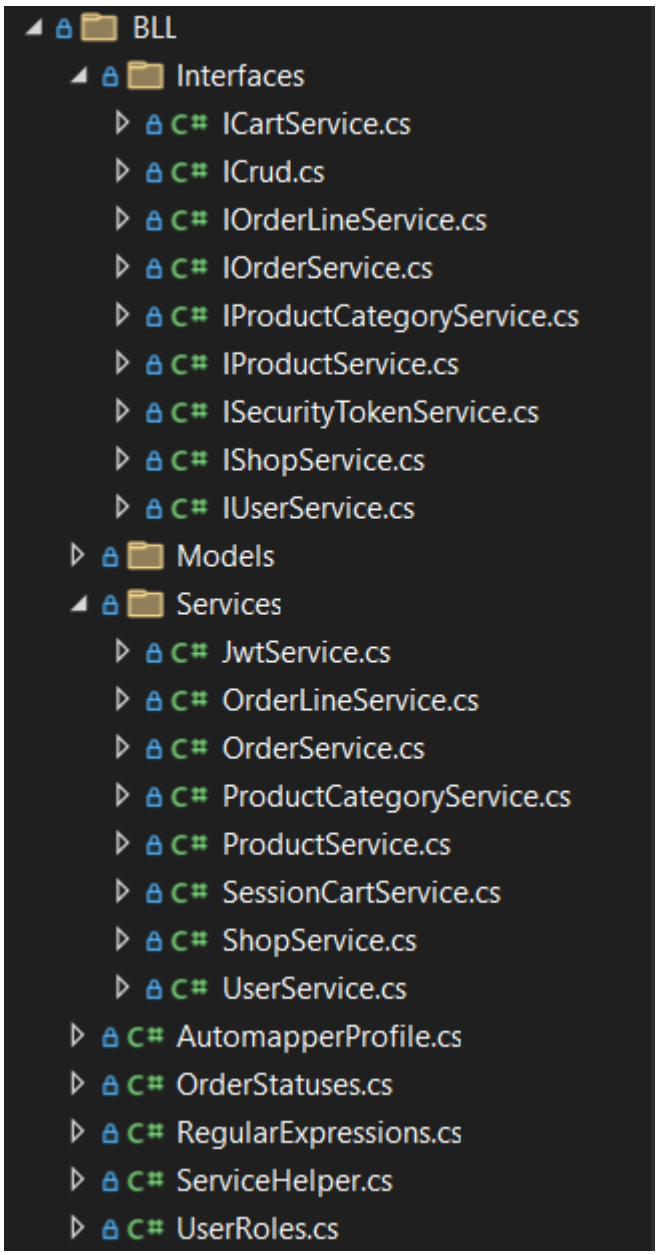


Рис. 2.28. Структура шару бізнес логіки у файловій системі.

Presentation Layer повинен надавати можливість клієнтам Web API отримувати та надсилати дані за допомогою HTTP запитів. Ця частина побудована на технології ASP.NET Core, котра надає контролери для керування вхідними HTTP запитами, має конфігурований канал запитів та надає контейнер інверсії контролю для автоматичної ін’єкції залежностей об’єктам цього контейнеру. Контролери використовують створені сервіси в Business Logic Layer, що були описані раніше в цьому розділі.

Контролери це класи, що визначають action методи для обробки вхідних HTTP запитів. Для пов’язання контролерів та їх action методів з запитами використовуються маршрути. Маршрут – це шаблон URL адреси та кінцева точка, що є обробником запиту на зазначений шаблон URL адреси. Встановлюється маршрут за допомогою атрибуту класу контролера [Route], що приймає базовий шаблон URL адреси на цей контролер, і атрибутів action методів [HttpGet], [HttpPost], [HttpPut], [HttpPatch], [HttpDelete], котрі вказують тип HTTP методу і можуть доповнювати базовий URL шаблону контролера. Для виконання завдання кваліфікаційної роботи було створено п’ять контролерів: для роботи з кошиком продуктів, для роботи з замовленнями, для роботи з продуктами, для роботи з магазинами та для роботи з користувачами, їх авторизацією, реєстрацією.

На рис. 2.29 зображений клас контролера UsersController з атрибутами [ApiController], котрий спрощує написання action методів для Web API, та [Route], що задає базовий шаблон URL «api/[controller]», де [controller] заміниться на Users, тобто назву контролера. Клас буде вважатись контролером тільки тоді, коли його назва закінчується на «Controller». ControllerBase клас, котрий наслідує контролер, надає зручні методи для формування відповідей на запити та інші можливості.

```
[ApiController]
[Route("api/[controller]")]
1 reference
public class UsersController : ControllerBase
```

Рис. 2.29. Приклад налаштування класу контролера.

На рис. 2.30 зображений авторизований action метод контролера з атрибутами [HttpGet], що вказує на прийнятний HTTP метод запиту для обробки, [Authorize], що був створений для обмеження доступу не авторизованим користувачам або користувачам з недозволеною роллю, та

[ProducesResponseType], що допомагає інструменту Swagger коректно описати кінцеву точку в OpenAPI Specification.

```
[Authorize(UserRoles.Admin)]
[HttpGet]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
0 references
public async Task<ActionResult<IEnumerable<UserModel>>> GetAllUsers()
{
    try
    {
        var users = await _userService.GetAllAsync();

        return Ok(users);
    }
    catch (Exception ex)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
    }
}
```

Рис. 2.30. Приклад авторизованого action методу контролеру.

ASP.NET Core має канал запитів, що складається з middleware. Middleware – це елемент каналу запитів, що може модифікувати HTTP запит і відповідь на нього. На рис. 2.31 зображені все middleware каналу запитів, що визначаються у головному файлі застосунку Program.cs. Порядок middleware є важливим і вони додані в такій послідовності: Routing, Cors, Session, Jwt, Swagger, SwaggerUI, Endpoints. Routing та Endpoints додаються неявним чином, а виклик MapControllers методу пов’язує систему маршрутизації з контролерами. CorsMiddleware налаштовує політику спільного використання ресурсів різного походження, потім SessionMiddleware перевіряє cookie сесії в HTTP запиті користувача і налаштовує сховище сесії, потім JwtMiddleware перевіряє наявність JWT у сховищі сесії та, у разі успіху, додає дані користувача до словника HTTP контексту. У кінці каналу запитів виконуються

Swagger та SwaggerUI middlewares, котрі створюють OpenAPI Specification та користувацький інтерфейс для візуалізації кінцевих точок Web API.

На рис. 2.32 зображений метод, що використовує JwtMiddleware для отримання токену з сесії та перевірки дійсності цього токену. Якщо токен був знайдений та успішно перевірений, то йде пошук користувача за його ідентифікатор та додавання даних цього користувача до словнику HTTP контексту, котрий буде використаний атрибутом [Authorize].

```
var app = builder.Build();

app.UseCors(policyName);
app.UseSession();
app.UseMiddleware<JwtMiddleware>();
app.MapControllers();

app.UseSwagger();
app.UseSwaggerUI(options => {
    options.SwaggerEndpoint("/swagger/v1/swagger.json", "WebApp");
});
```

Рис. 2.31. Всі елементи middleware каналу запитів.

```
private async Task AttachUserToContext(HttpContext context, IUserService userService,
    ISecurityTokenService<UserModel, JwtSecurityToken> securityTokenService, string token)
{
    try
    {
        var jwtSecurityToken = securityTokenService.ValidateToken(token);
        var userId = jwtSecurityToken.Claims.FirstOrDefault(x => x.Type == "id")?.Value;
        var user = userId is not null ? await userService.GetByIdAsync(userId) : null;
        if (user is not null)
        {
            var roles = await userService.GetUserRolesAsync(user.Id!);
            context.Items["User"] = user;
            context.Items["UserRoles"] = roles;
            _logger.LogDebug("Jwt validation passed successfully. User '{UserName}' was added to the HTTP context items", user.UserName);
        }
    }
    catch(Exception ex)
    {
        _logger.LogDebug(message: "The security token had not passed the validation", exception: ex);
    }
}
```

Рис. 2.32. Метод класу JwtMiddleware, що перевіряє наявність користувача в сесії.

На рис. 2.33 зображений атрибут, що використовується для авторизації користувачів до елементів з обмеженим доступом. Він приймає в конструкторі всі ролі користувачів, котрим можна надати доступ. Для отримання даних використовується словник контексту Items, що заповнюється у JwtMiddleware. Якщо користувач не був авторизованим до системи або він має невідповідну роль, то йому відмовляється в доступі та формується відповідь з кодом 401 [18], тобто вказує що користувач не авторизований для доступу до бажаних даних. Користувач отримає доступ до обмеженого ресурсу тільки якщо він пройде перевірку.

Рис. 2.33. Створений атрибут для обмеження доступу не авторизованим користувачам.

Program.cs файл це головна точка входу програми, в якій встановлюється конфігурація ASP.NET Core застосунку. Різні вбудовані та створені мною компоненти реалізують дизайн патерн ін’єкції залежностей, тому в цьому файлі за допомогою контейнера інверсії контролю можна налаштувати потрібні залежності й інші аспекти веб-застосунку. Був доданий сервіс з CORS політикою (рис. 2.34). Були додані сервіси для роботи з сесією, контекстом бази даних та сервіси, що потрібні для функціонування ASP.NET Core Identity (рис. 2.35). Були додані сервіси для роботи маппера, Swagger, репозиторії, створені в Data Access Layer, та сервіси, створені в Business Logic Layer (рис. 2.36).

Елементи middleware, що були встановлені каналу запитів у файлі Program.cs, зображені на рис. 2.31.

На рис. 2.37 зображений вигляд структури всього ASP.NET Core проєкту, що був розроблений для виконання завдання кваліфікаційної роботи.

```
const string policyName = "ReactProjectCorsPolicy";

builder.Services.AddCors(opts =>
{
    opts.AddPolicy(policyName, policy =>
    {
        policy.WithOrigins(builder.Configuration["AllowedCorsUrls:ReactHttpProject"]!,
            builder.Configuration["AllowedCorsUrls:ReactHttpsProject"]!)
            .AllowAnyHeader()
            .AllowAnyMethod()
            .AllowCredentials();
    });
});
```

Рис. 2.34. Налаштування CORS сервісу.

```
builder.Services.AddDistributedMemoryCache();
builder.Services.AddSession(opts =>
{
    opts.IdleTimeout = TimeSpan.FromMinutes(30);
    opts.Cookie.Name = "DokWokApi.Session";
    opts.Cookie.IsEssential = true;
    opts.Cookie.HttpOnly = true;
});

builder.Services.AddDbContext<StoreDbContext>(opts =>
{
    opts.UseSqlServer(builder.Configuration["ConnectionStrings:FoodStoreConnection"]);
});

builder.Services.AddIdentity<ApplicationUser, IdentityRole>().AddEntityFrameworkStores<StoreDbContext>();
builder.Services.Configure<IdentityOptions>(opts => {
    opts.Password.RequiredLength = 6;
    opts.Password.RequireNonAlphanumeric = false;
    opts.Password.RequireLowercase = false;
    opts.Password.RequireUppercase = true;
    opts.Password.RequireDigit = true;
    opts.User.RequireUniqueEmail = true;
});
```

Рис. 2.35. Налаштування сервісів сесії та контексту бази даних.

```
var mapperConfig = new MapperConfiguration(mc => mc.AddProfile(new AutomapperProfile()));
IMapper mapper = mapperConfig.CreateMapper();
builder.Services.AddSingleton(mapper);

builder.Services.AddHttpContextAccessor();

builder.Services.AddScoped<IProductCategoryRepository, ProductCategoryRepository>();
builder.Services.AddScoped<IProductRepository, ProductRepository>();
builder.Services.AddScoped<IOrderRepository, OrderRepository>();
builder.Services.AddScoped<IOrderLineRepository, OrderLineRepository>();
builder.Services.AddScoped<IShopRepository, ShopRepository>();

builder.Services.AddScoped<IProductCategoryService, ProductCategoryService>();
builder.Services.AddScoped<IProductService, ProductService>();
builder.Services.AddScoped<IOrderService, OrderService>();
builder.Services.AddScoped<IOrderLineService, OrderLineService>();
builder.Services.AddScoped<IShopService, ShopService>();
builder.Services.AddScoped<ICartService, SessionCartService>();
builder.Services.AddScoped<IUserService, UserService>();
builder.Services.AddScoped<ISecurityTokenService<UserModel, JwtSecurityToken>, JwtService>();

builder.Services.AddSwaggerGen(c => {
    c.SwaggerDoc("v1", new OpenApiInfo
    {
        Title = "DokWokApi",
        Version = "v1"
    });
});
```

Рис. 2.36. Налаштування сервісів мапперу, репозиторіїв, власних сервісів та Swagger.

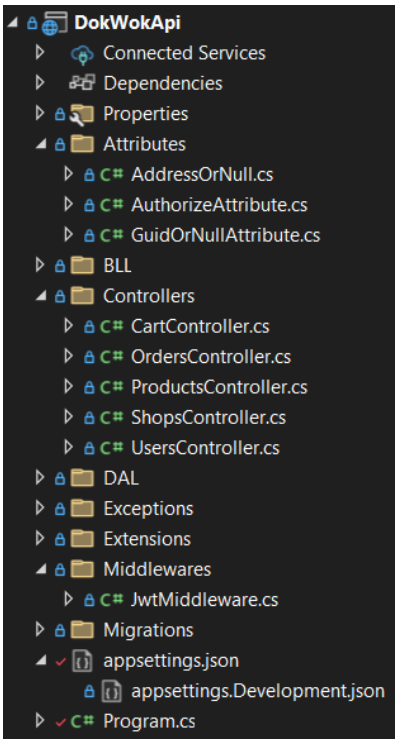


Рис. 2.37. Структура всього ASP.NET Core проєкту у файловій системі.

Клієнтську частину представляє React проєкт. Структура проєкту зображена на рис. 2.38. Каталог «node_modules» тримає в собі всі файли та сторонні пакунки необхідні для роботи серверного середовища Node.js. Всі файли розробки знаходяться в каталозі «src». Каталог «assets» зберігає всі зображення та іконки, що використовує програма. Каталог «components» (рис. 2.39) зберігає всі файли з розширенням tsx, що виступають React компонентами. Ці файли є фундаментом побудови візуального інтерфейсу застосунку. React компоненти можуть входити в склад інших компонентів, або виступати в ролі повноцінної сторінки. Каталог «css» (рис. 2.40) зберігає всі стилі, що були написані для інтерфейсу користувача. Каталог «functions» тримає в собі файли з функціями виконання запитів до серверної частини (рис. 2.41). Каталог «helpers» має файли з константами та інтерфейсами, що використовуються у застосунку. Каталог «hooks» тримає в собі файл зі створеним «гачком», що являє собою функцію-обгортку навколо існуючої функції бібліотеки React. На рис. 2.42 зображена структура каталогів «functions», «helpers» та «hooks». Каталог «scss» має файл з приєднаними потрібними компонентами бібліотеки Bootstrap (рис. 2.43), для того щоб уникнути конфлікту стилів. Останній каталог «validation» зберігає файли з функціями перевірки введених даних до полів форм користувацького інтерфейсу.

Файл App.tsx описує логіку маршрутизації з використанням бібліотеки React Router, main.tsx тримає в собі головний компонент React проєкту, а index.html виступає у якості корінного документу всіх веб-сторінок. Всі інші файли виконують важливу конфігурацію проєкту.

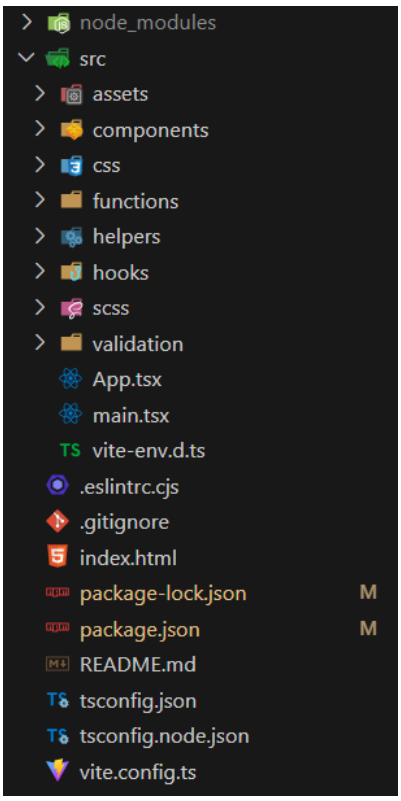


Рис. 2.38. Структура React проєкту.

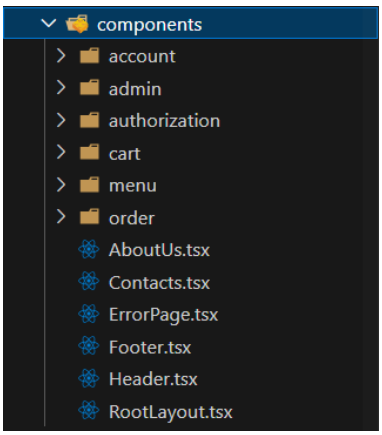


Рис. 2.39. Структура каталогу «components».

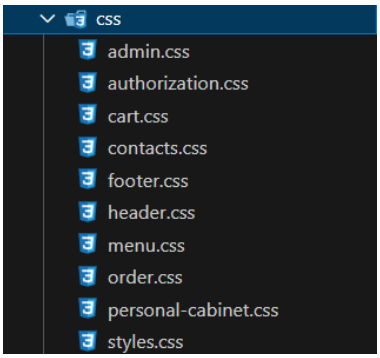


Рис. 2.40. Структура каталогу «css».

```ts
export async function customerLogin(
  loginUser: LoginUserProp
): Promise<AuthUserProp | null> {
  const axiosInstance = getAxiosInstance();
  try {
    const response = await axiosInstance.post(
      "users/customers/login",
      loginUser
    );
    return response.data;
  } catch (error) {
    if (axios.isAxiosError(error)) {
      if (error.response) {
        if (error.response.status === 400 || error.response.status === 404) {
          return null;
        } else {
          throw new Error("There was a server-side error.");
        }
      } else {
        throw new Error("The request never reached the server.");
      }
    } else {
      throw new Error("There was a non-axios related error.");
    }
  }
}
```

Рис. 2.41. Приклад функції, що надсилає HTTP запити до серверу.

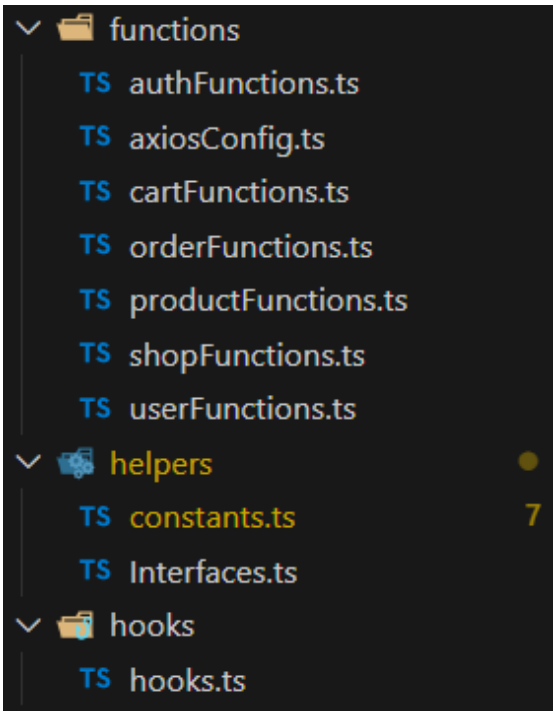


Рис. 2.42. Структура каталогів «functions», «helpers» та «hooks»

```scss
@import "/node_modules/bootstrap/scss/functions";
@import "/node_modules/bootstrap/scss/variables";
$my-form-color: #e2520d;

$input-focus-border-color: $my-form-color;
$input-focus-box-shadow: 0 0 0 .25rem rgba($my-form-color, .25);

$form-select-focus-border-color: $my-form-color;
$form-select-focus-box-shadow: 0 0 0 .25rem rgba($my-form-color, .25);

$form-check-input-focus-border: $my-form-color;
$form-check-input-checked-color: $my-form-color;
$form-check-input-checked-bg-color: $my-form-color;
$form-check-input-checked-border-color: $my-form-color;
$form-check-input-focus-box-shadow: 0 0 0 .25rem rgba($my-form-color, .25);

@import "/node_modules/bootstrap/scss/variables-dark";
@import "/node_modules/bootstrap/scss/maps";
@import "/node_modules/bootstrap/scss/mixins";
@import "/node_modules/bootstrap/scss/root";
@import "/node_modules/bootstrap/scss/bootstrap-utilities.scss";
@import "/node_modules/bootstrap/scss/containers";
@import "/node_modules/bootstrap/scss/grid";
@import "/node_modules/bootstrap/scss/navbar";
@import "/node_modules/bootstrap/scss/buttons";
@import "/node_modules/bootstrap/scss/tables";
@import "/node_modules/bootstrap/scss/forms";
```

Рис. 2.43. Файл підключення компонентів бібліотеки Bootstrap.

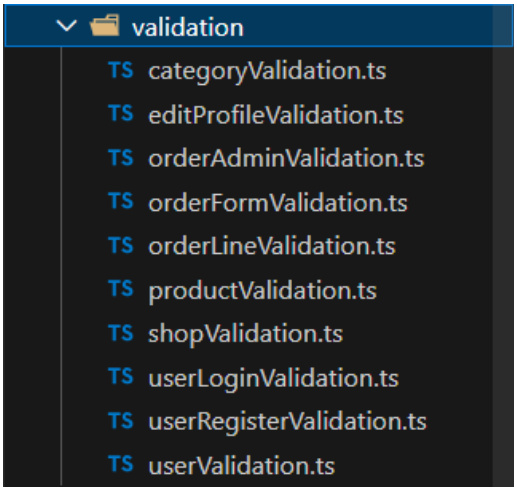


Рис. 2.44. Структура каталогу «validation».

Для роботи системи «DokWok» була побудована реляційна база даних з дванадцяти таблиць: Products (продукти), ProductCategories (категорії продуктів), OrderLines (лінії замовлення), Orders (замовлення), Shops (магазини), AspNetUsers (користувачі), AspNetUserRoles (ролі користувачів), AspNetUserTokens (токени користувачів), AspNetUserLogins (входження

користувачів у систему), AspNetUserClaims (твердження користувачів), AspNetRoles (ролі), AspNetRoleClaims (твердження ролей). Таблиці, чия назва починається на «AspNet», були створені інструментом ASP.NET Core Identity для керування користувачами. На рис. 2.45 зображена діаграма зв’язків сутностей бази даних.

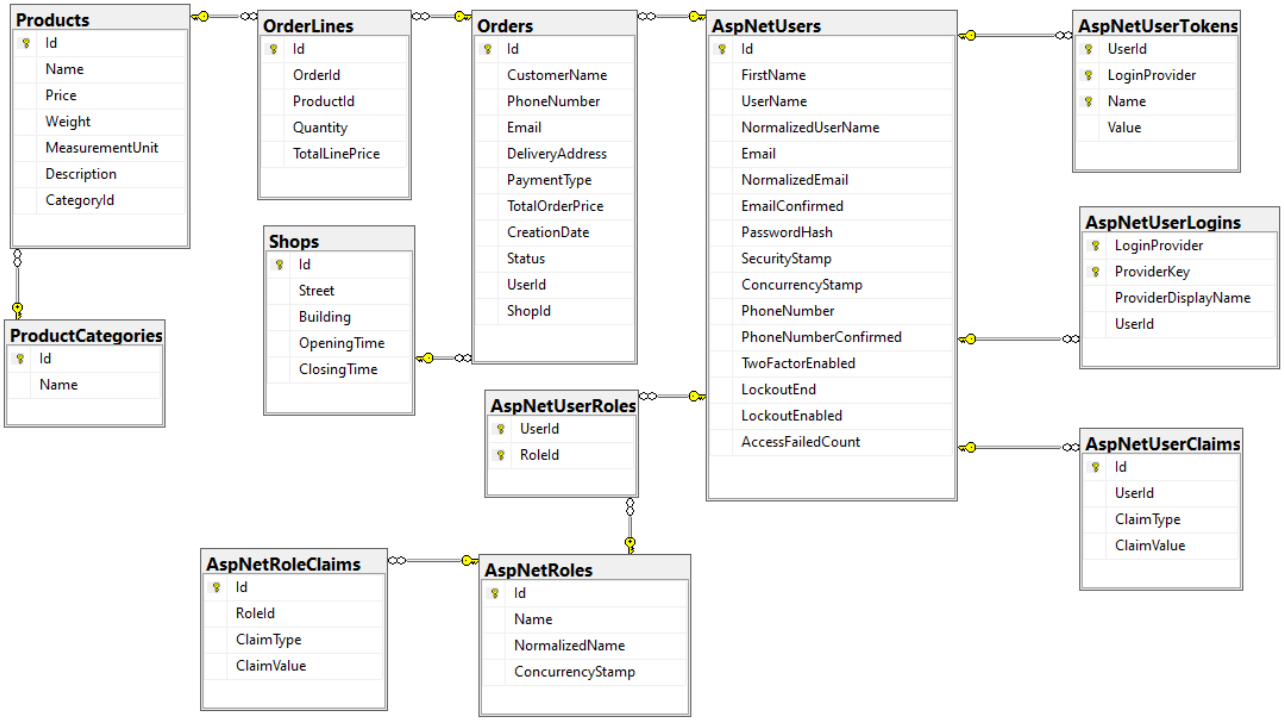


Рис. 2.45. ER-діаграма бази даних системи «DokWok».

2.5. Обґрунтування та організація вхідних та вихідних даних програми

Зв’язок між клієнтською та серверною частиною відбувається через надсилання HTTP запитів від клієнта до сервера та, відповідно, відправка HTTP відповіді від сервера клієнту. Клієнт і сервер можуть також обмінюватись даними у форматі JSON, що були записані у тіло HTTP запиту або відповіді. ASP.NET Core автоматично виконує серіалізацію C# об’єктів у JSON формат при відправленні даних та десеріалізацію JSON об’єктів у C# формат при отриманні даних.

Клієнтська частина надає користувачам програмного забезпечення можливість надсилати дані на сервер за допомогою форм авторизації, реєстрації, оформлення замовлення та зміни особистих даних. Адміністратори мають доступ до модифікації стану бази даних і також використовуватимуть форми для створення та оновлення записів таблиць. Всі поля форм користувацького інтерфейсу проходять попередню перевірку даних перед відправкою їх на сервер. Після успішної перевірки формується запит, в котрий додається об’єкт з даними форми, та відправляється на сервер, котрий також проводить додаткову перевірку.

Серверна частина, за запитом клієнта, може записувати об’єкти з даними до тіл відповідей та надсилати їх клієнту.

Звичайні класи, що використовуються як сутності бази даних, не підходять для обміну даними між клієнтом та сервером, тому для цього були створені окремі класи-моделі на рівні шару бізнес логіки. Наприклад, сутність продукту має властивість ідентифікатора, але для створення продукту ця властивість непотрібна, тому вона буде відсутня в класі-моделі для зручності та уникнення непорозуміння зі сторони клієнта, котрий може думати, що ідентифікатор потрібен для створення нового запису в базі даних. Також є випадки коли нам треба відправити тільки частину даних сутності, тому для кожного можливого варіанту обміну даними була створена своя класова модель, що чітко підходить під конкретний випадок (рис. 2.46). Деякі створені класи, наприклад Cart, не пов’язані з сутностями бази даних і використовуються сервісами незалежно від них. На рис. 2.47 зображений приклад вигляду класової моделі обміну даних, а на рис. 2.48 можна побачити вигляд класу сутності бази даних. Сервіси та контролери будуть використовувати інструмент AutoMapper для зручної та швидкої конвертації об’єктів сутностей у об’єкти класових моделей і навпаки. AutoMapper для використання потребує лише один раз прописані інструкції конвертації одних типів у інші.

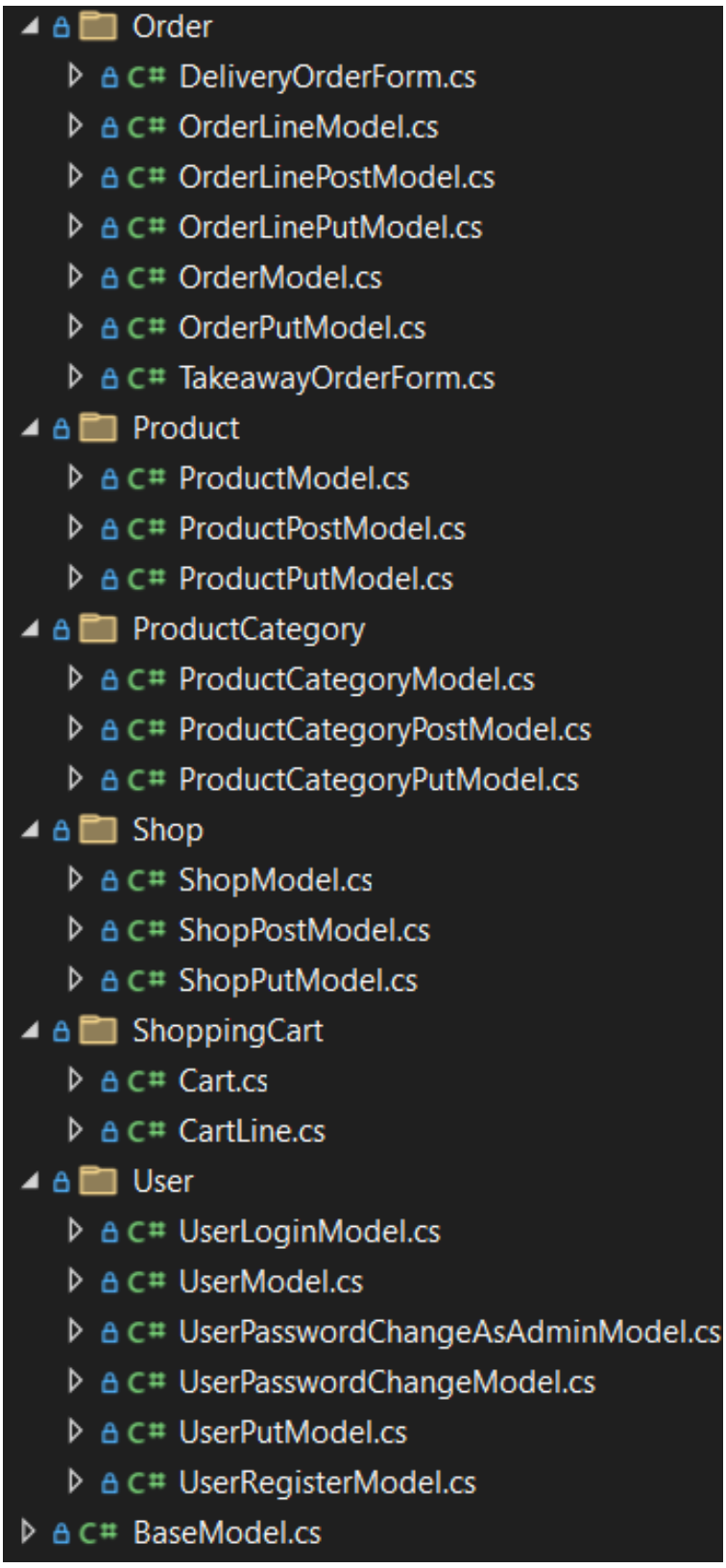


Рис. 2.46. Створені класи-моделі.

```
public class ProductModel : BaseModel
{
    14 references
    public string Name { get; set; } = string.Empty;

    15 references
    public decimal Price { get; set; }

    0 references
    public decimal Weight { get; set; }

    0 references
    public string MeasurementUnit { get; set; } = string.Empty;

    13 references
    public string Description { get; set; } = string.Empty;

    13 references
    public long CategoryId { get; set; }

    14 references
    public string CategoryName { get; set; } = string.Empty;
}
```

Рис. 2.47. Вигляд класової моделі обміну даних.

```
public class Product : BaseEntity
{
    45 references
    public string Name { get; set; } = string.Empty;

    [Column(TypeName = "decimal(8, 2)")]
    39 references
    public decimal Price { get; set; }

    [Column(TypeName = "decimal(8, 2)")]
    9 references
    public decimal Weight { get; set; }

    9 references
    public string MeasurementUnit { get; set; } = string.Empty;

    37 references
    public string Description { get; set; } = string.Empty;

    23 references
    public long CategoryId { get; set; }

    45 references
    public ProductCategory? Category { get; set; }

    1 reference
    public ICollection<OrderLine> OrderLines { get; set; } = [];
}
```

Рис. 2.48. Вигляд класу сутності бази даних.

2.6. Опис розробленої системи

2.6.1. Використані технічні засоби

Клієнтське обладнання не потребує великої потужності компонентів для роботи веб-застосунку, оскільки для взаємодії використовується браузер.

Для забезпечення надійного функціонування програмного забезпечення на настільному комп’ютері клієнта він повинен відповідати таким вимогам:

- доступ до мережі Internet;
- наявність 4 Гб вільного місця на жорсткому диску;
- наявність щонайменше 4 Гб оперативної пам’яті;
- рідкокристалічний монітор з діагоналлю не менше 15" для відображення контенту;
- процесор з мінімальною частотою в 2 ГГц для коректної роботи браузеру;
- маніпулятор «миша»;
- клавіатура.

Для забезпечення надійного функціонування програмного забезпечення на мобільному телефоні клієнта він повинен відповідати таким вимогам:

- доступ до мережі Internet;
- наявність 4 Гб вільного місця на жорсткому диску;
- наявність щонайменше 4 Гб оперативної пам’яті;
- екран не менше 6" для відображення контенту;
- процесор з мінімальною частотою в 2 ГГц для коректної роботи браузеру;
- працюючий сенсорний екран.

Вище наведені характеристики є мінімальними для коректної та надійної роботи програмного забезпечення.

Для розробки був використаний ноутбук з такими технічними характеристиками:

- процесор Intel Core i5-1035G1з тактовою частотою від 1.0 до 3.6 ГГц;
- 8 Гб оперативної пам’яті;
- внутрішня пам’ять типу SSD обсягом 256 Гб;
- екран діагоналлю 15,6";
- маніпулятор «миша»;
- клавіатура.

Такої обчислювальної машини, з характеристиками наведеними вище, достатньо щоб вести розробку програмного забезпечення, запускати його на локальному сервері, витримувати базове навантаження запитами користувачів та використовувати повноцінну систему керування базами даних.

2.6.2. Використані програмні засоби

Для використання веб-застосунку зі сторони клієнта потрібні такі програмні засоби:

- браузер Google Chrome, Firefox, Opera, Microsoft Edge, Safari або інший;
- операційна система Windows 7/10/11 для персональних комп’ютерів;
- операційна система Android 13 і вище, або IOS 15 і вище для мобільних пристроїв.

Під час розробки програмного забезпечення були використані такі програмні засоби:

- IDE Microsoft Visual Studio Community 2022 для створення серверної частини;
- редактор коду Microsoft Visual Studio Code для створення клієнтської частини;
- Microsoft SQL Server Management Studio 19 створення серверу бази даних;

– браузер Opera GX.

Клієнтська частина була реалізована за допомогою бібліотек React, React Router, Axios, Bootstrap та Material UI з використанням TypeScript як мови програмування. HTML 5 та CSS 3 виконують роль структуризації та стилізації веб-сторінок застосунку. Для збірки React проєкту та його запуску використовується інструмент Vite.

Серверна частина була реалізована на платформі .NET 8 за допомогою фреймворків ASP.NET Core, Entity Framework Core, ASP.NET Core Identity та бібліотеки AutoMapper з використанням C# як мови програмування.

2.6.3. Виклик та завантаження програми

Клієнту для використання веб-застосунку не потрібно його скачувати, а за допомогою браузера перейти за URL посиланням місцезнаходження веб-застосунку. Розробники повинні передчасно розгорнути програмне забезпечення за допомогою сервісу веб-хостингу з обраним доменом веб-сайту, або іншим бажаним способом.

Розробник може завантажити клієнтську та серверну частини на локальну машину за допомогою клонування публічних репозиторіїв на сервісі GitHub.

Для запуску клієнтською частини потрібно встановити серверне середовище Node.js. Далі, знаходячись в директорії проєкту, потрібно виконати в терміналі команду «npm install» для встановлення необхідних залежностей та команду «npm run dev» для запуску React проєкту. Веб-сайт буде доступним в браузері за посиланням «localhost:3000».

Клієнтська частина залежить від серверної, тому наступним кроком треба запустити ASP.NET Core застосунок. Для запуску серверної частини потребується встановлений .NET 8 SDK. Далі, знаходячись в директорії проєкту, виконати команду терміналу «dotnet run» для запуску .NET проєкту. Сервер стане доступним для звертання за посиланням «localhost:5000». Встановлення SDK та запуск проєкту може відбуватись за допомогою IDE

Microsoft Visual Studio. Слід зауважити, що завантажується тільки код програми без бази даних, тому треба попіклуватись про налаштування серверу бази даних на локальній машині та змінити за потребою рядок підключення до бази даних у файлі appsettings.json.

2.6.4. Опис інтерфейсу користувача

Головна сторінка веб-сайту відображає всі пропозиції закладу харчування, що можна замовити (рис. 2.49). Заголовок тримає логотип, що є посиланням до головної сторінки, навігаційне меню між категоріями продуктів, іконка кошика, що переправляє на сторінку з кошиком, та кнопку «Увійти» (рис. 2.50) для не авторизованих користувачів, що переправляє до сторінки авторизації, або іконку людини (рис. 2.51) для авторизованих користувачів, що переправляє до особистого кабінету. Нижній колонтитул сторінки (рис. 2.52) містить логотип, такий самий як і заголовку, контактні номера, перелік партнерів підприємства і посилання на сторінки «Контакти» та «Про нас».

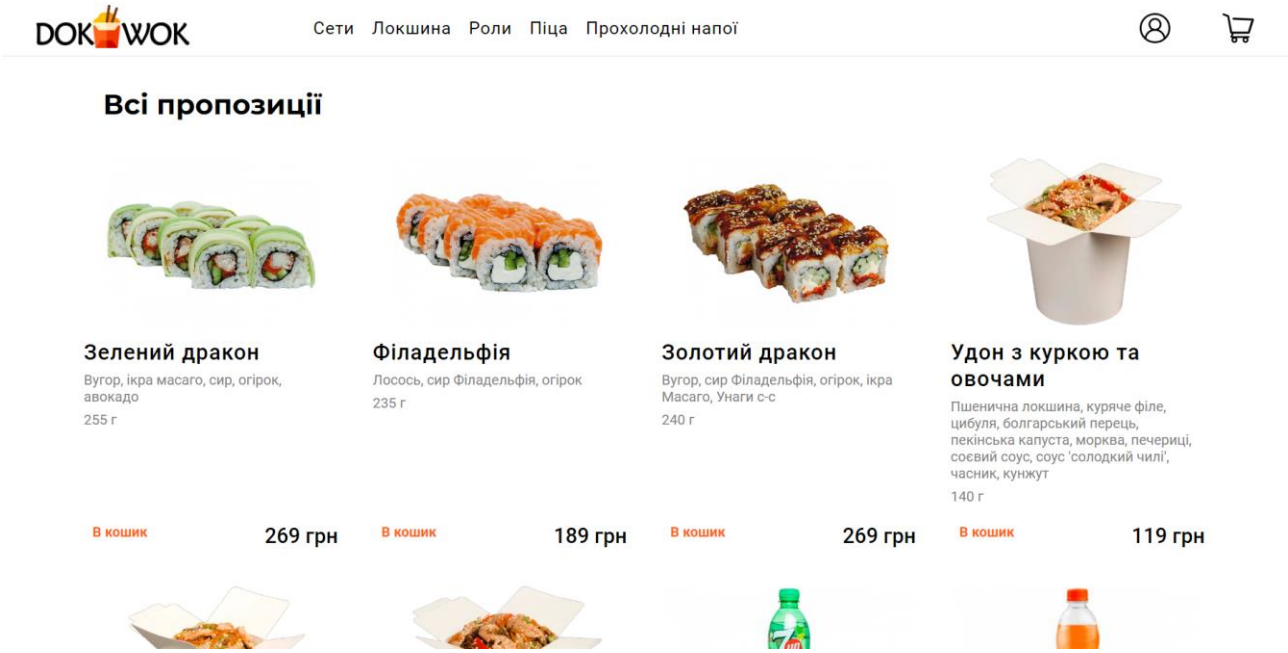


Рис. 2.49. Головна сторінка веб-застосунку «DokWok».

Рис. 2.50. Кнопка для авторизації користувача.

Рис. 2.51. Ікона людини для переходу до особистого кабінету.

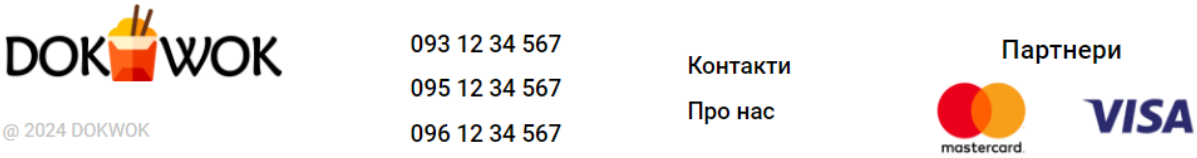


Рис. 2.52. Ніжній колонтитул веб-сторінки.

На рис. 2.53 зображена сторінка з продуктами обраної категорії «Роли» через навігаційне меню заголовку сторінки.

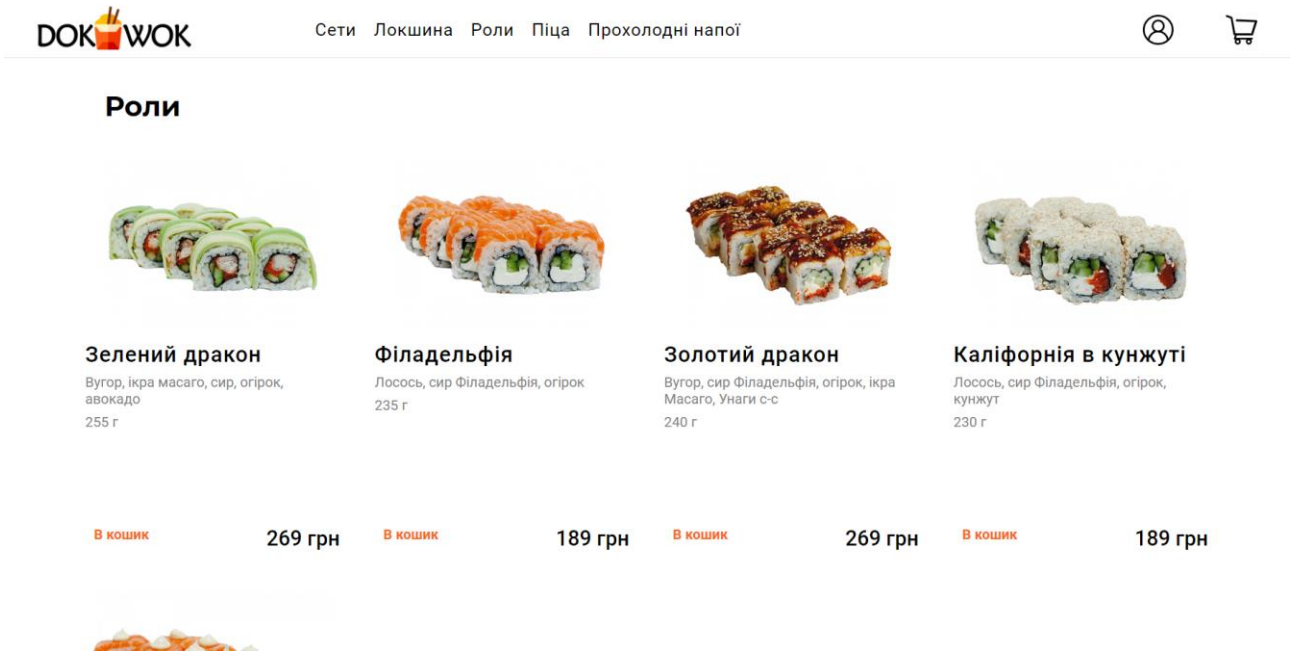


Рис. 2.53. Веб-сторінки з продуктами категорії «Роли».

Продукт меню відображає його назву, зображення, опис, вагу, ціну та кнопку для додавання продукту до кошика. З продуктами меню можна взаємодіяти (рис. 2.54). Наведення курсором миші додає тінь навколо об’єкта та змінює колір опису продукту на чорний. Наведення на кнопку «В кошик» змінює її колір на помаранчевий, а текст на білий, і натискання на кнопку додає одну одиницю товару до кошику. Іконка кошика відображає кількість одиниць товару в ньому (рис. 2.55). Якщо кошик пустий, то сторінка кошика буде відображати відповідне повідомлення (рис. 2.56).

Рис 2.54. Взаємодія з продуктом меню.

Рис. 2.55. Вигляд кошику з доданим товаром.

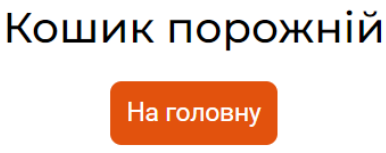


Рис. 2.56. Повідомлення про пустий кошик.

Після додавання продуктів до кошику їх можна знайти на відповідній веб-сторінці, якщо натиснути на іконку кошика (рис. 2.57). Кожен продукт кошика виводить такі елементи: назву, вагу, кнопку видалення продукту з кошику, кількість одиниць доданого продукту в штуках, кнопки для змінення кількості одиниць продукту та загальну вартість цього продукту. Після переліку всіх продуктів виводиться загальна вартість покупки всіх елементів кошика та кнопку для оформлення замовлення.

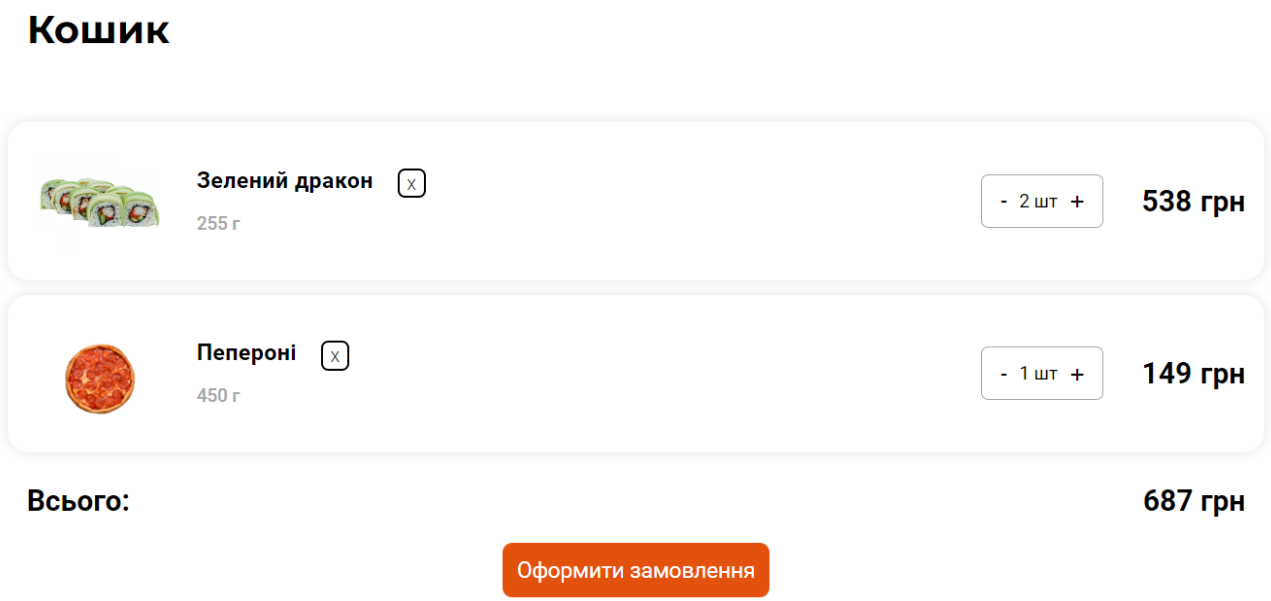


Рис. 2.57. Веб-сторінка кошика з доданими продуктами.

Створювати замовлення у закладі харчування «DokWok» можуть не тільки авторизовані клієнти, а й гості, але вони не зможуть зберегти історію своїх замовлень та слідкувати за станом їх виконання. Для зручності буде продемонстрований процес створення замовлення авторизованим користувачем, для подальшого його перегляду в особистому кабінеті.

Для переходу до веб-сторінки авторизації потрібно натиснути кнопку заголовку «Увійти» (рис. 2.50). У відкритій сторінці буде форма авторизації та кнопка «Зареєструйтесь», що перенаправить користувача до форми реєстрації, якщо в нього немає власного акаунту (рис. 2.58). Для реєстрації користувач вводить до форми свої особисті дані: логін, пароль, ім’я, номер телефону та

електронну пошту (рис. 2.59). Уведені дані всіх форм веб-застосунку перевіряються на наявність та на коректність (рис. 2.60).

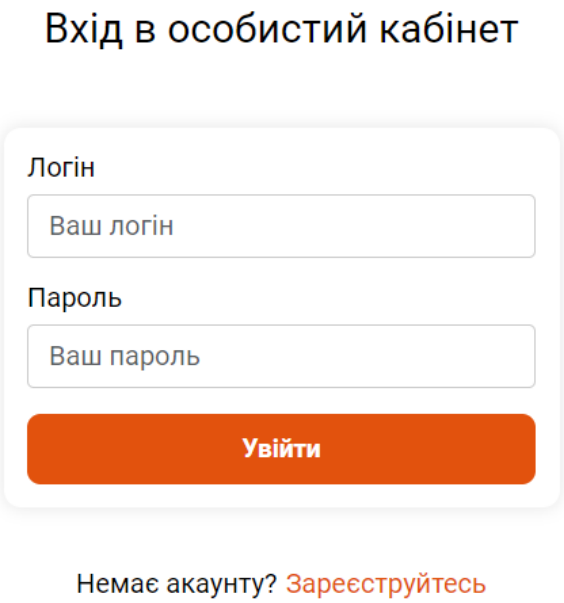


Рис. 2.58. Форма авторизації існуючого користувача.

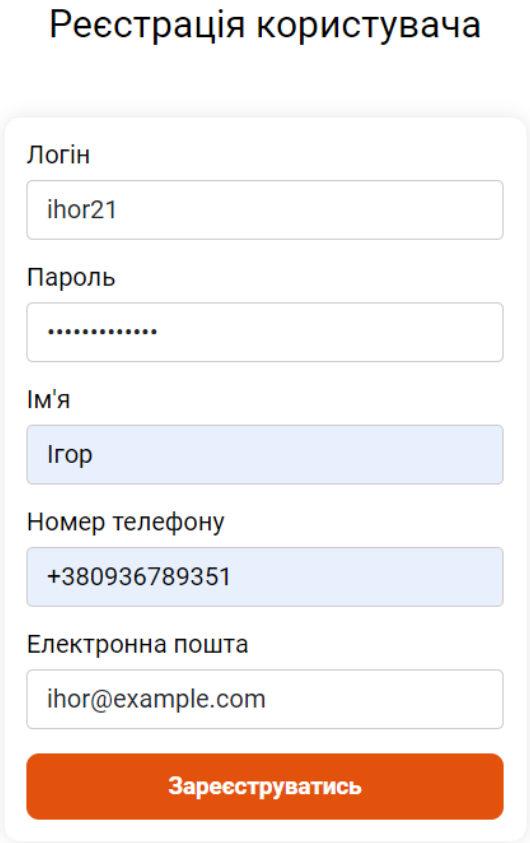


Рис. 2.59. Заповнена форма реєстрації.

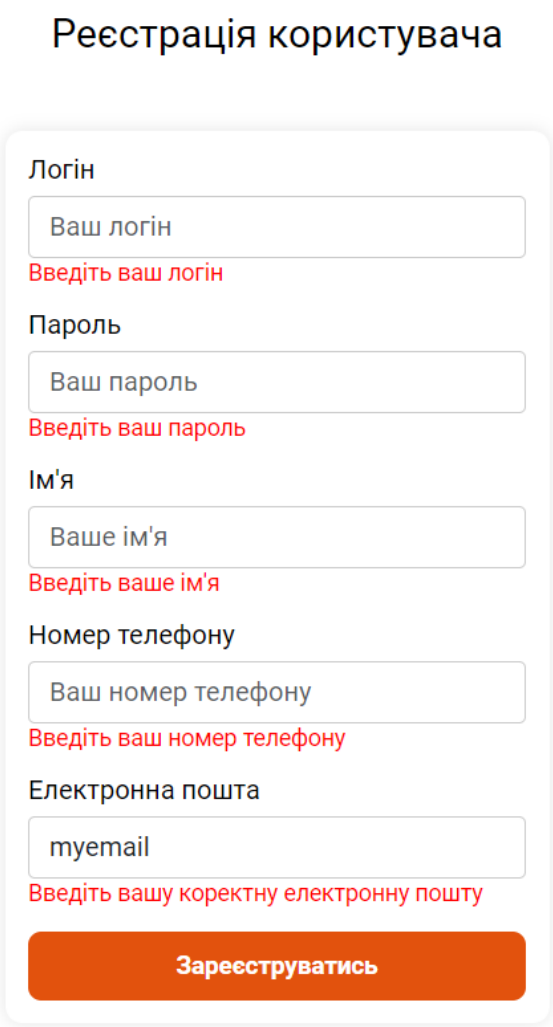


Рис. 2.60. Форма реєстрації з відсутніми та некоректними даними користувача.

Після реєстрації, користувача буде перенаправлено до веб-сторінки особистого кабінету (рис. 2.61), що відображає навігаційне меню з кнопками «Профіль», котра відобразить особисті дані користувача, «Історія замовлень», котра відобразить інформацію про всі замовлення користувача, та «Вийти», котра виконає вихід поточного користувача з системи. Як видно на рис. 2.61, особистий кабінет за замовчуванням відобразить блок «Профіль», що містить особисті дані користувача з кнопкою зміни цих даних. Якщо користувач забажає змінити свої дані, то він може натиснути кнопку «Змінити особисті дані» та використати дві форми: зміни особистих даних та зміни паролю (рис. 2.62). Кнопка «Історія замовлень» перенаправить до відповідного блоку, але він буде пустим, оскільки користувач ще не створював замовлень (рис. 2.63).

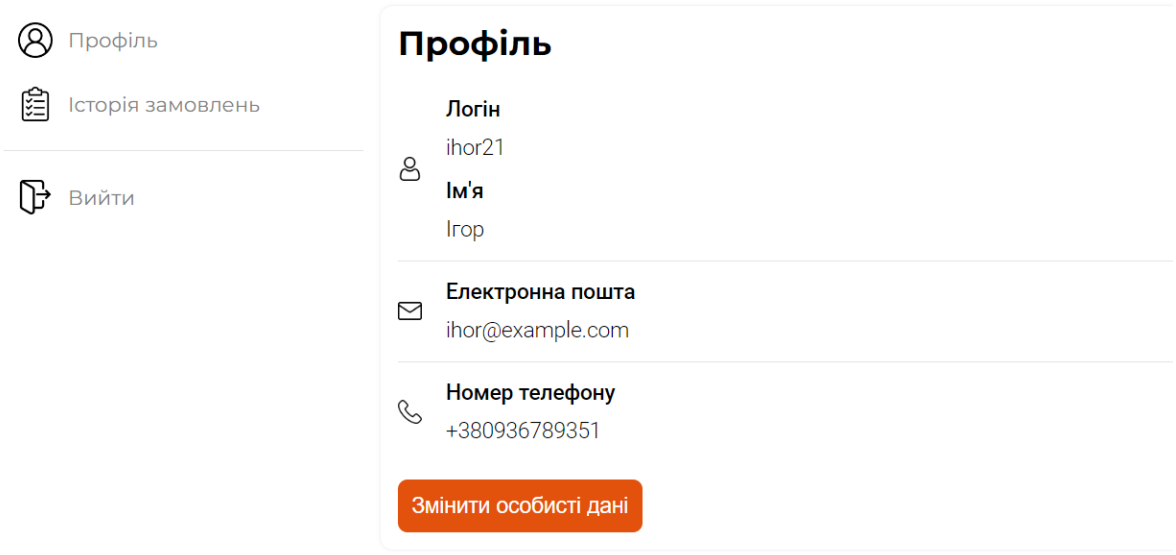


Рис. 2.61. Особистий кабінет авторизованого користувача.

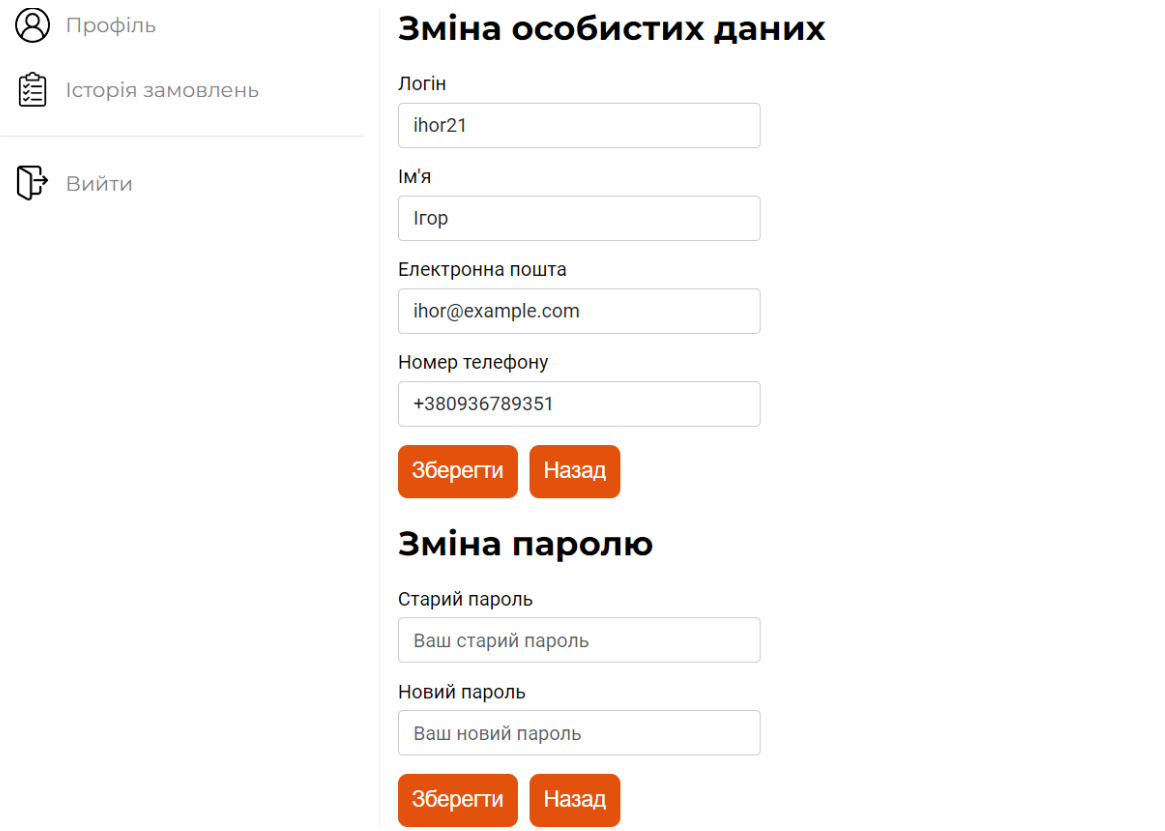


Рис. 2.62. Форми зміни особистих даних користувача.

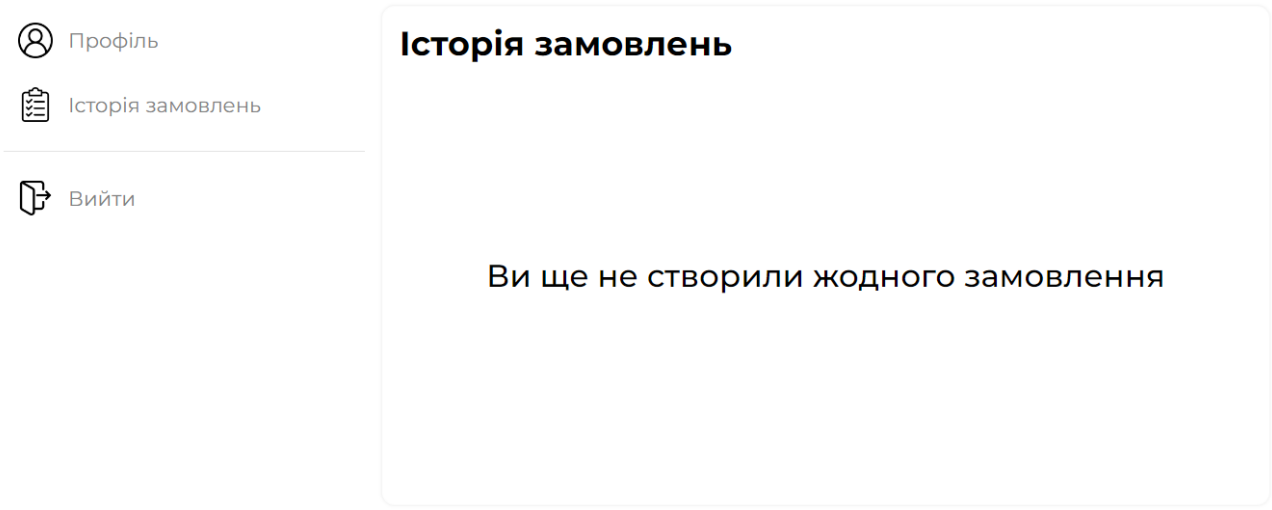


Рис. 2.63. Пустий блок з історією замовлень користувача.

Натиснувши кнопку «Оформити замовлення» на сторінці кошика (рис. 2.57), користувач перейде до сторінки з формою замовлення (рис. 2.64).

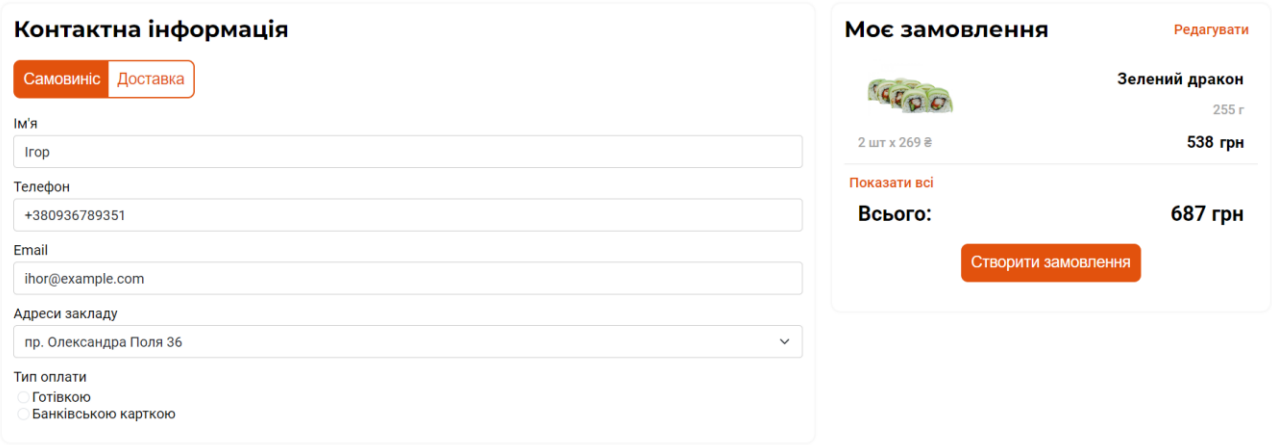


Рис. 2.64. Веб-сторінка оформлення замовлення.

Зліва сторінки знаходиться блок з формою замовлення. Можна обрати тип замовлення кнопками «Самовиніс» та «Доставка». На рис. 2.65 зображена форма замовлення типу самовиніс, що потребує ім’я, телефон та email замовника, а також тип оплати і адресу закладу забору замовлення (рис. 2.66). На рис. 2.67 зображена форма замовлення типу доставка, що потребує ті самі дані, що і попередня форма, але замість адреси закладу треба ввести адресу доставки замовлення.

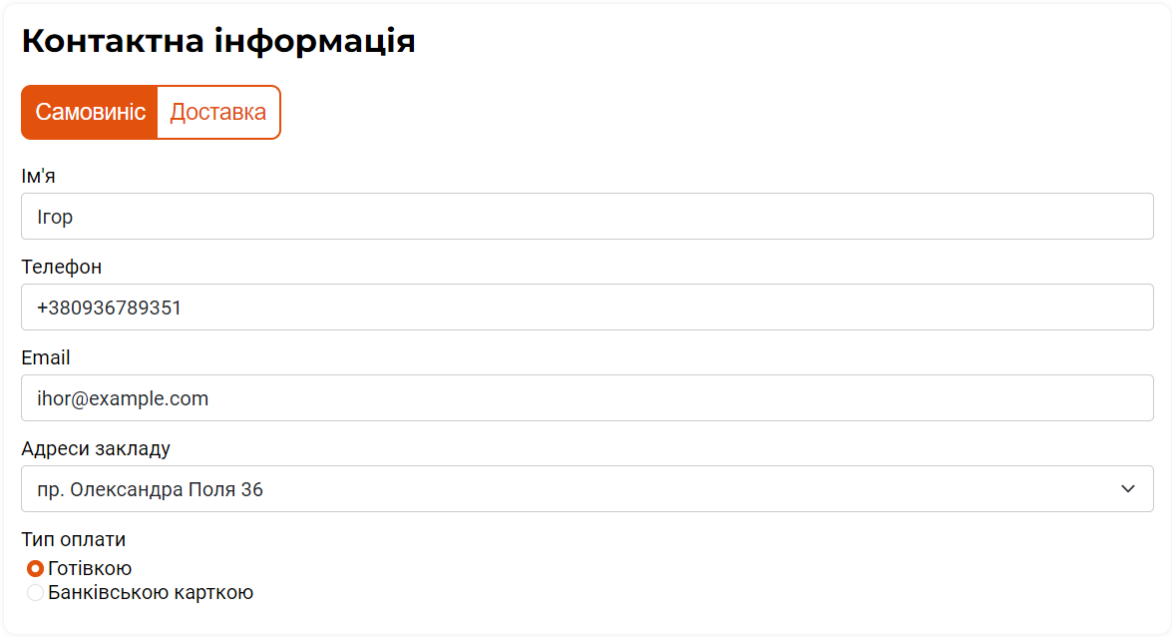


Рис. 2.65. Оформлення замовлення типу самовиніс.

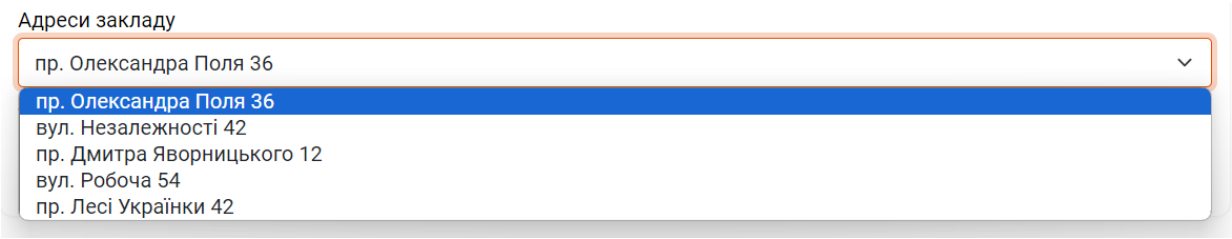


Рис. 2.66. Вибір точки закладу харчування як місця забору замовлення.

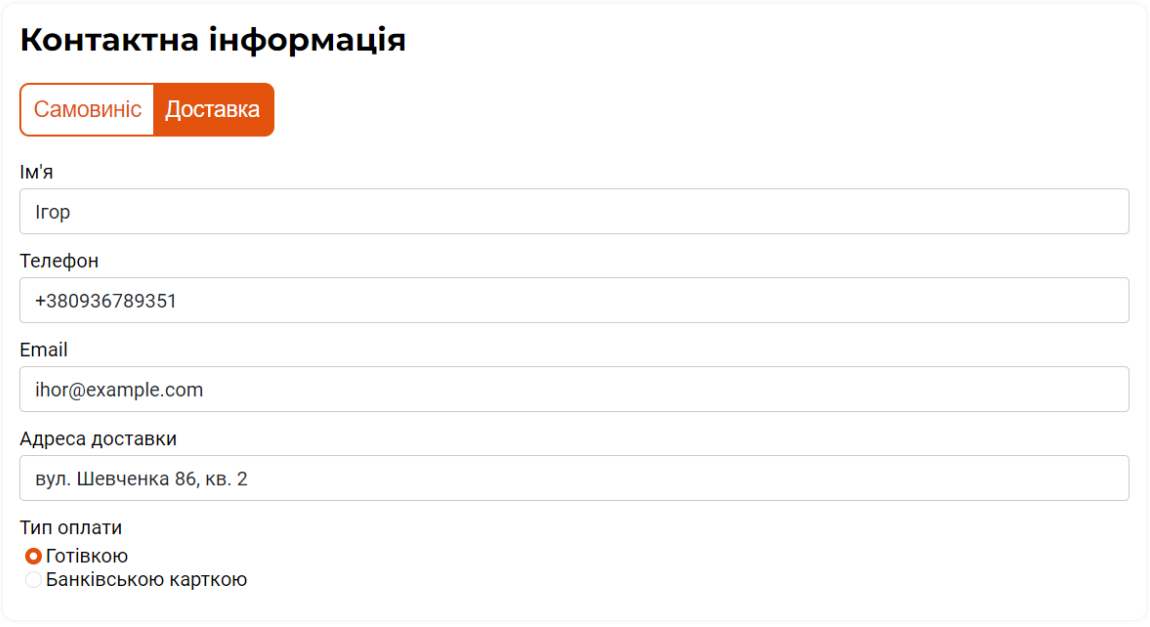


Рис. 2.67. Оформлення замовлення типу доставка.

Справа сторінки знаходиться блок «Моє замовлення», що містить складений список продуктів замовлення (рис. 2.68), кнопку «Показати всі», загальну вартість замовлення та кнопку для створення замовлення. Якщо натиснути на кнопку «Показати всі», то розкриється список продуктів та кнопка змінить напис на «Показати менше» (рис. 2.69). Продукт списку відображає такі дані: назву, зображення, вагу, кількість одиниць продукту разом з ціною одиниці та загальну вартість цього продукту.

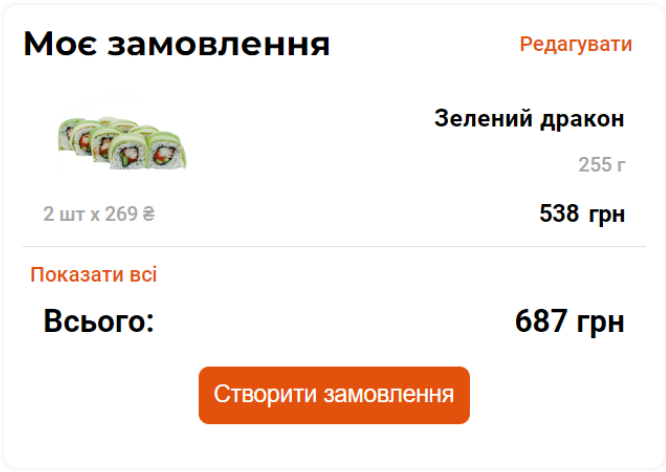


Рис. 2.68. Блок «Моє замовлення» зі складеним списком продуктів.

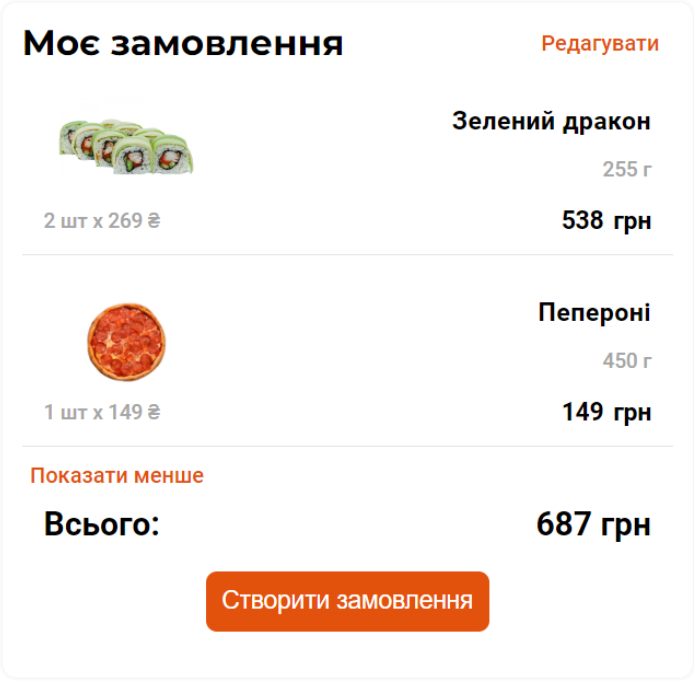


Рис. 2.69. Блок «Моє замовлення» з розкритим списком продуктів.

На рис. 2.70 зображений невдалий результат спроби надсилання форми з некоректними даними. Якщо дані форми були введені коректно, то після натискання кнопки «Створити замовлення» буде надісланий запит до серверу та буде виведене повідомлення результату створення замовлення, разом з його номером у разі успіху (рис. 2.71).

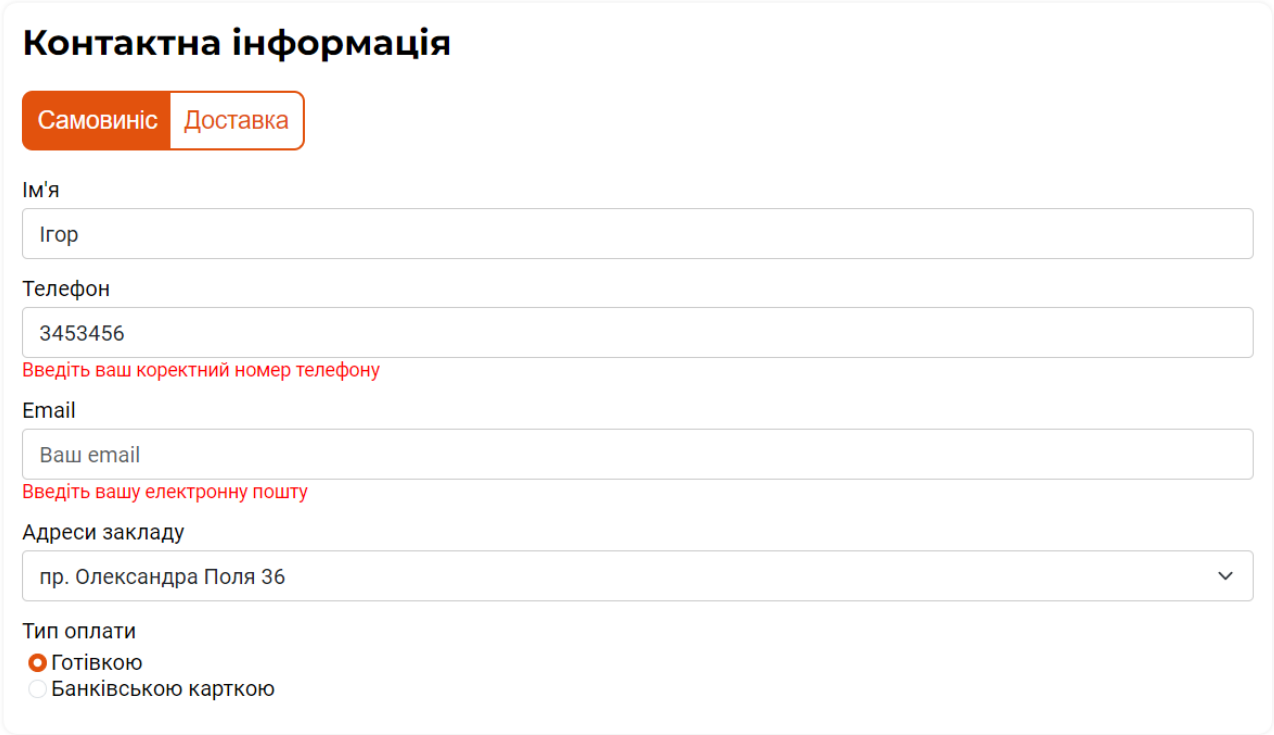


Рис. 2.70. Результат спроби відправки форми з некоректними або пустими полями.

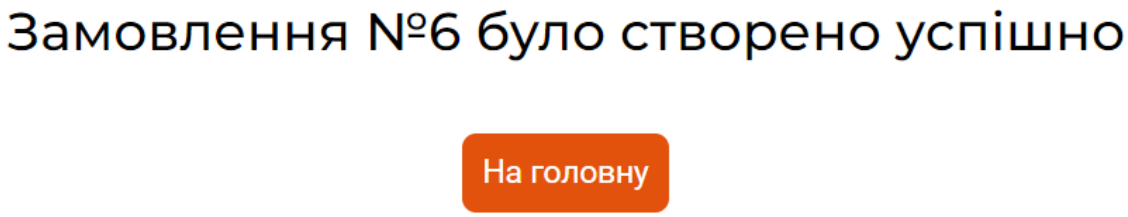


Рис. 2.71. Результат створення замовлення.

Після створення замовлення користувач може перейти до свого особистого кабінету та обрати блок історії замовлень. Там він побачить своє створене замовлення у розкритому вигляді (рис. 2.72), тому що воно тільки одне. Якщо замовлень більше ніж один, то вони відображаються у складеному вигляді за замовчуванням (рис. 2.73). В залежності від типу замовлення в ньому буде зображена адреса доставки замовлення (рис. 2.72) або адреса закладу отримання замовлення (рис. 2.74). Продукти замовлення відображають ту саму інформацію, що і під час створення замовлення.

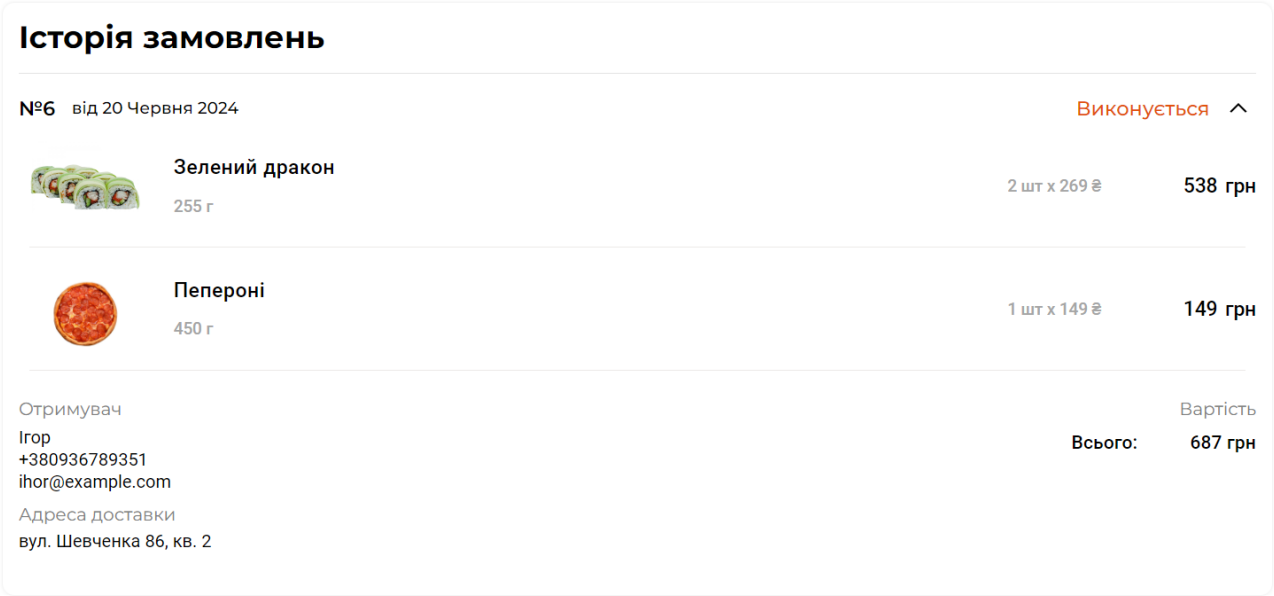


Рис. 2.72. Створене розкрите замовлення користувача типу доставка в блоці «Історія замовлень».

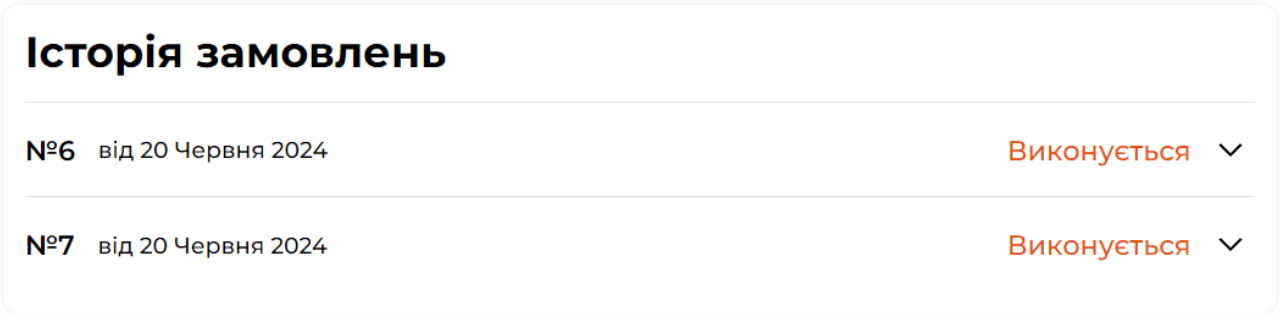


Рис. 2.73. Вигляд складеного замовлення.

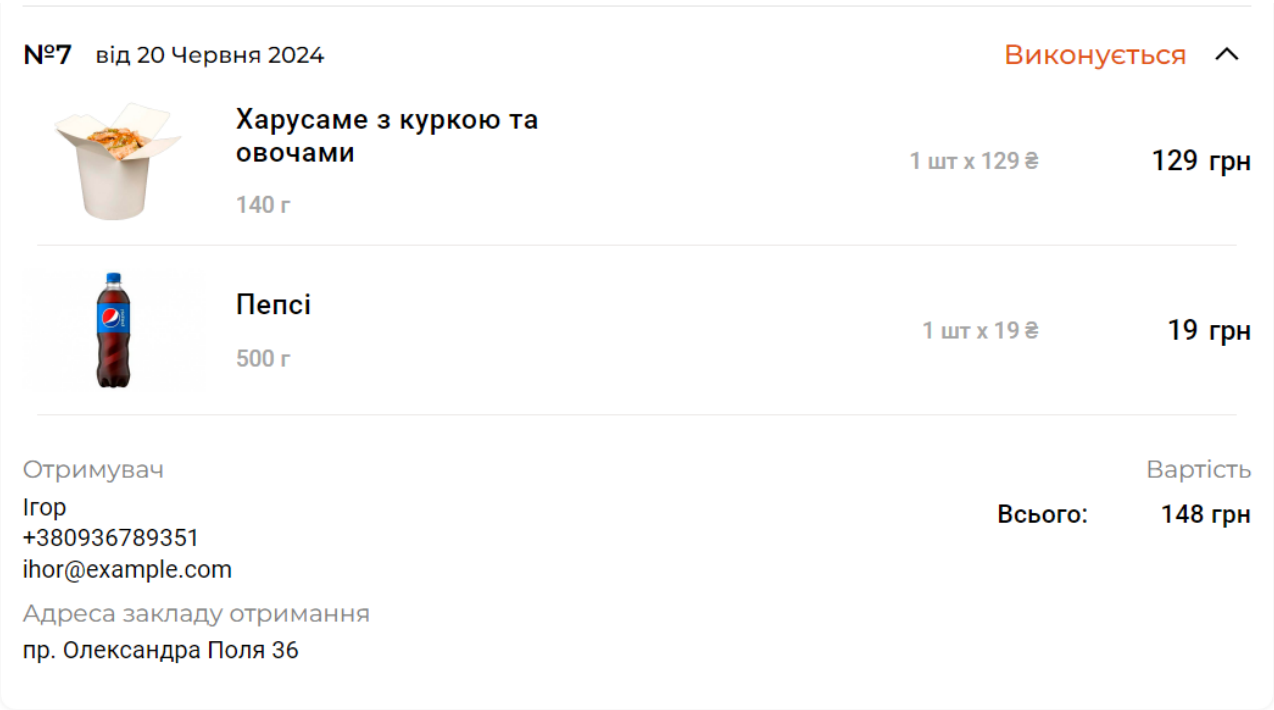


Рис. 2.74. Створене розкрите замовлення користувача типу самовиніс в блоці «Історія замовлень».

На рис. 2.75 зображена веб-сторінка «Контакти» з контактними даними закладу та переліком всіх ресторанів з їх місцезнаходженням і часом роботи. На рис. 2.76 зображена сторінка «Про нас», де заклад може охарактеризувати свою діяльність або місію.

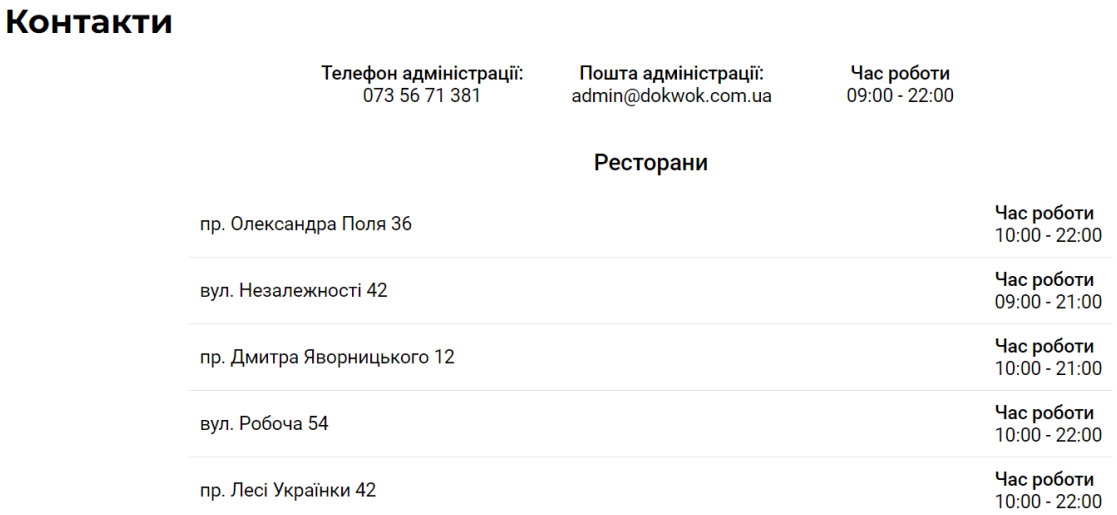


Рис. 2.75. Веб-сторінка «Контакти».

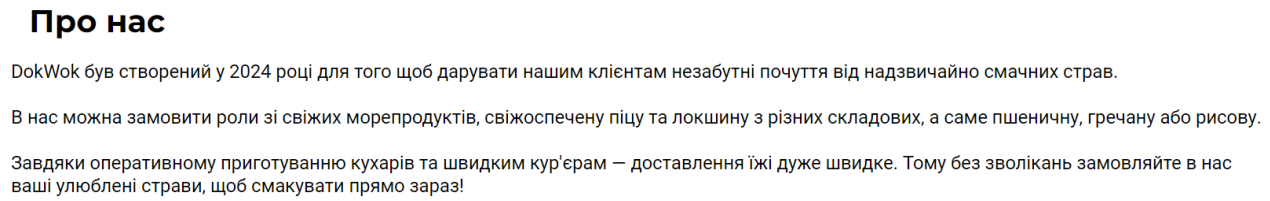


Рис. 2.76. Веб-сторінка «Про нас».

Макет веб-сайту є адаптивним та пристосовується до розширення зображення користувача. Наприклад, при маленькому розширені екрану навігаційне меню зникає та з’являється кнопка відкриття висувного бічного меню, а іконка кошика переноситься до правого нижнього кута (рис. 2.77). На рис. 2.78 зображене висувне навігаційне меню.

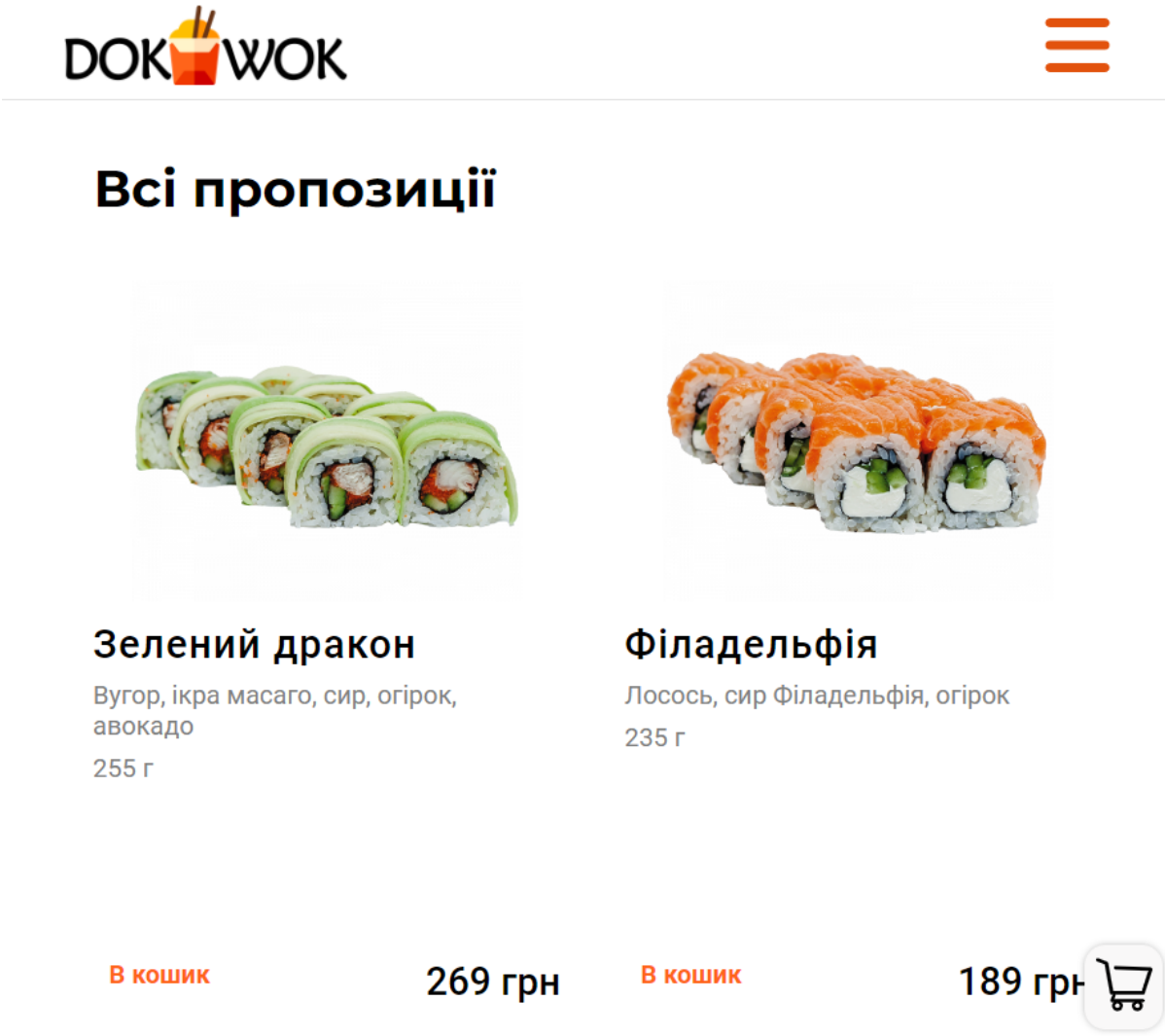


Рис. 2.77. Головна сторінка при маленькому розширенні екрану.

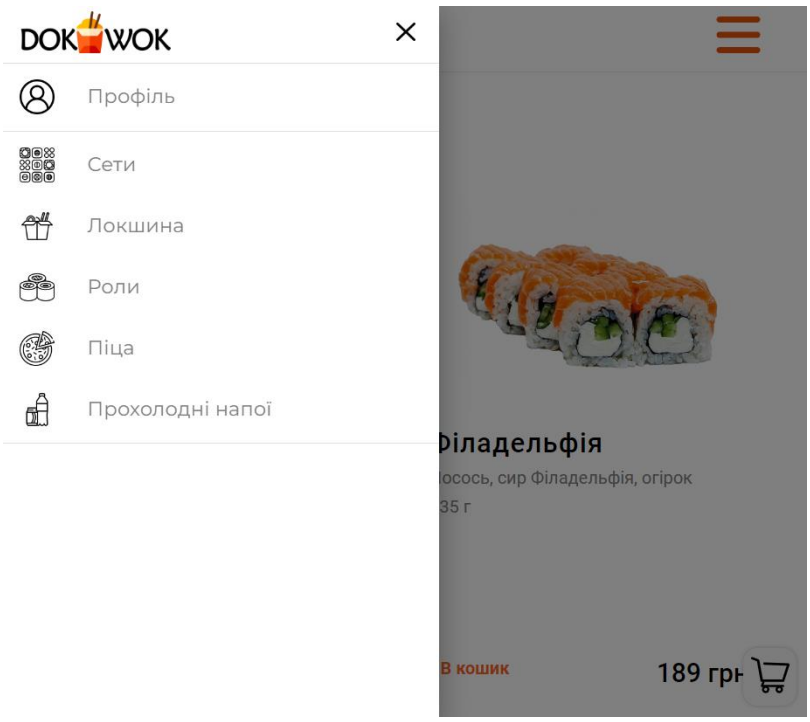


Рис. 2.78. Вигляд висувного навігаційного меню.

Для доступу до адміністрації немає елементів навігації, бо користувачам не потрібно бачити інформацію про її наявність. За шляхом «/admin» в адресному рядку можна спробувати перейти до адміністрації, але якщо користувач з роллю адміністратор не авторизований, то він буде перенаправлений до веб-сторінки авторизації (рис. 2.79). Якщо уведені дані користувача є правильними та цей користувач має роль адміністратора, то він буде допущений до головної сторінки адміністрації (рис. 2.80).

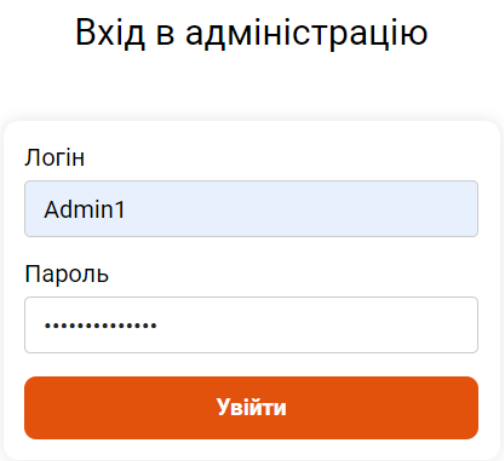


Рис. 2.79. Форма авторизації адміністратора.

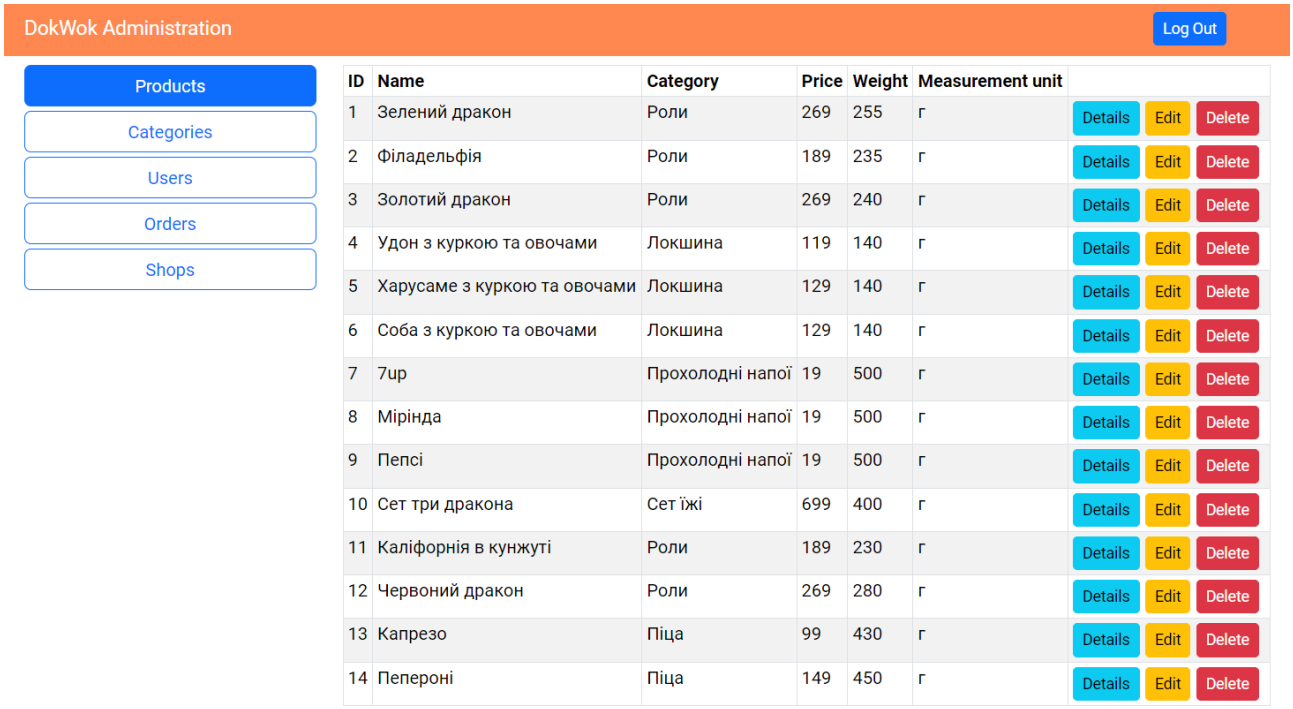


Рис. 2.80. Веб-сторінка адміністрації.

Веб-сторінка адміністрації з лівого боку має навігаційне меню для переходу між різними типами сутностей бази даних, а саме Products (продукти), Categories (категорії продуктів), Users (користувачі), Orders (замовлення) та Shops (магазини). Всі сутності мають на своїх сторінках таблицю зі всіма записами, приклад такої таблиці зображений на рис. 2.80. Під таблицями знаходиться кнопка «Create» (рис. 2.81), що відкриває форму для створення нового запису сутності (рис. 2.82). Якщо введені дані у форму не є коректними, то будуть відображені повідомлення про відсутність або некоректність даних та відправки форми не відбудеться (рис. 2.83). У разі передачі правильної інформації про запис, він буде доданий до бази даних та буде відображений у таблиці сутності (рис. 2.84).

Рис. 2.81. Кнопка переходу до форми створення запису.

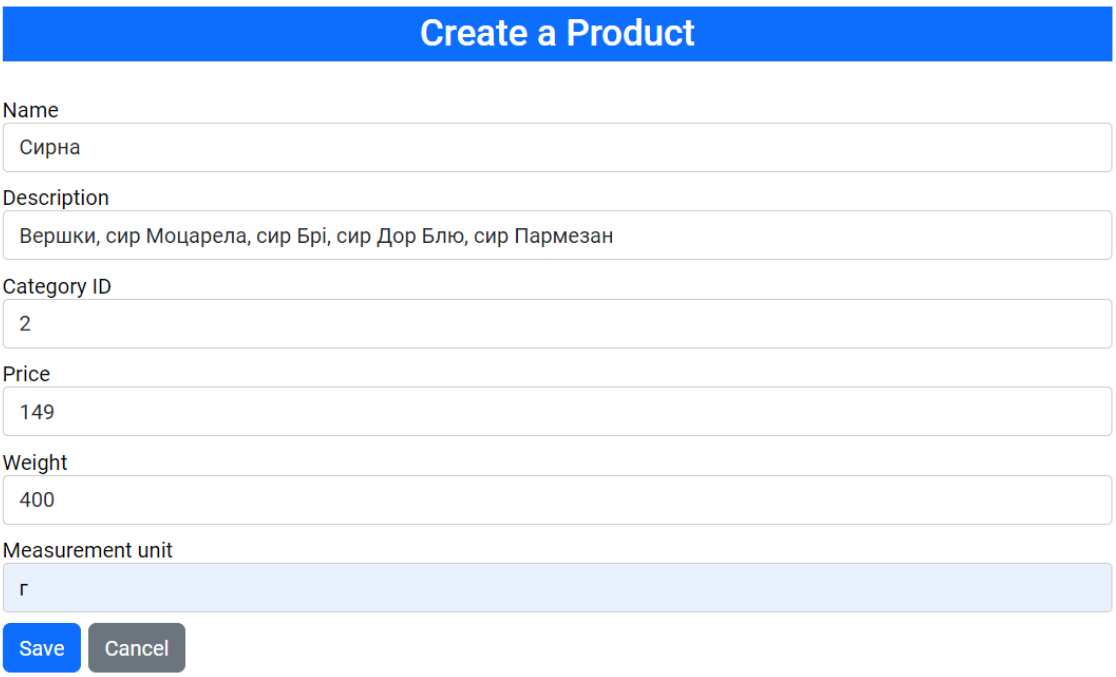


Рис. 2.82. Коректно заповнена форма створення запису.

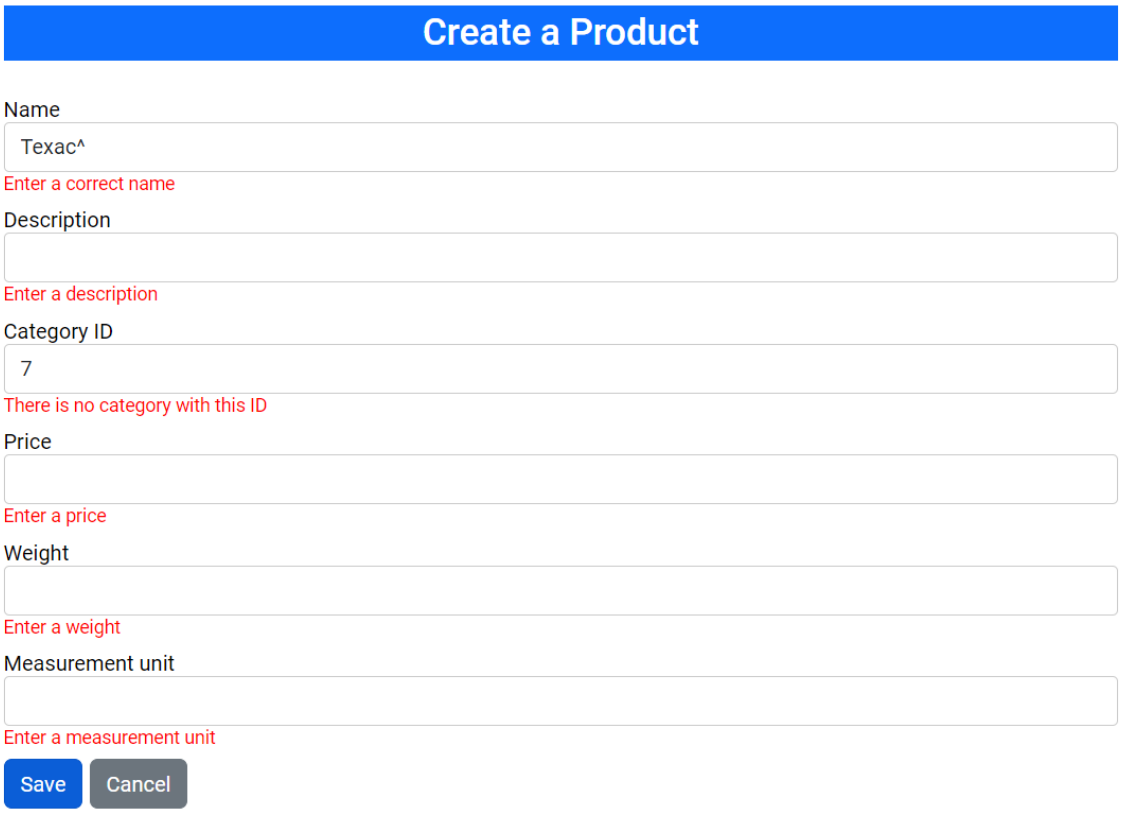


Рис. 2.83. Надсилання форми адміністрації з некоректними даними.

Рис. 2.84. Створений запис у таблиці.

Останній стовпець для кожного запису в таблиці має три кнопки: «Details», «Edit» та «Delete». Кнопка «Details» відкриває сторінку зі всіма подробицями запису (рис. 2.85). Кнопка «Delete» остаточно видаляє запис з таблиці бази даних. Кнопка «Edit» відкриває форму для зміни даних запису. Ця форма також виконує перевірку даних перед їх відправкою. На рис. 2.86 зображена форма зміни запису з оновленим полем ціни, а на рис. 2.87 видна зміна значення цього поля в таблиці сутності.

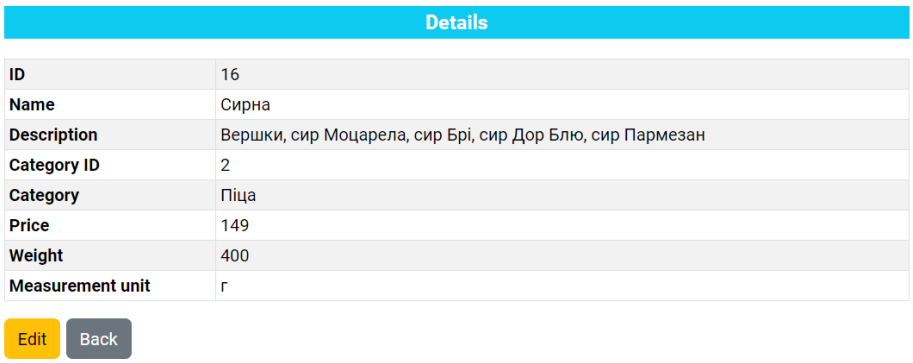


Рис. 2.85. Сторінка з деталями запису.

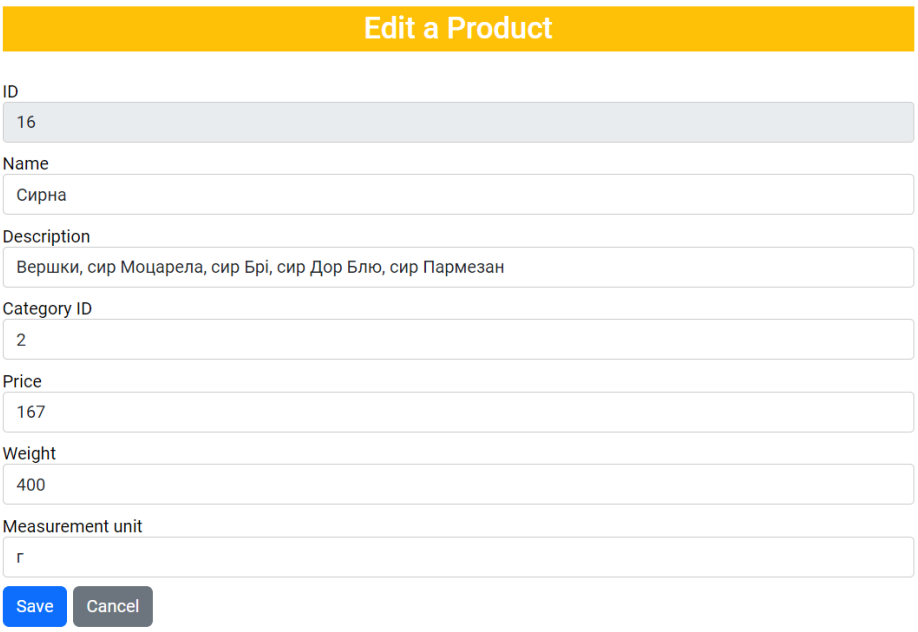


Рис. 2.86. Форма зміни запису з оновленою ціною продукту.

Рис. 2.87. Змінений запис продукту в таблиці.

РОЗДІЛ 3
ЕКОНОМІЧНИЙ РОЗДІЛ

3.1. Розрахунок трудомісткості та вартості розробки програмного продукту.

Вихідні дані:

1. Передбачуване число операторів програми – 2000;
2. Коефіцієнт складності програми – 1,6;
3. Коефіцієнт корекції програми в ході її розробки – 0,06;
4. Годинна заробітна плата програміста – 131 грн/год;

Значення цього показника були взяті з веб-сайту з робочими вакансіями jooble [19].

5. Коефіцієнт збільшення витрат праці внаслідок недостатнього опису задачі – 1,3;
6. Коефіцієнт кваліфікації програміста, обумовлений від стажу роботи з даної спеціальності – 1;
7. Вартість машино-години ЕОМ – 0,489 грн/год;

Ноутбук, що використовувався для виконання кваліфікаційної роботи, споживає приблизно 80 Вт/год. Згідно з даними офіційного сайту постачальника електроенергії «Yasno» [20], на момент розробки програмного забезпечення ціна за електроенергію жителям України становила 2,64 грн кВт/год з урахуванням ПДВ. Тоді витрати на електроенергію ноутбуком становлять 0,08 * 2,64 = 0,2112 грн/год. Витрати за використання послуг інтернет-провайдера «Domonet» [21] становлять 200 грн/міс, з них 195 грн/міс за тариф та 5 грн/міс за оренду конвертера. Погодинна вартість користування інтернетом становить 200 / 720 = 0,278 грн/год. Таким чином, вартість машино-години становила 0,2112 + 0,278 = 0,489 грн/год.

Нормування праці в процесі створення ПЗ істотно ускладнено в силу творчого характеру праці програміста. Тому трудомісткість розробки ПЗ може бути розрахована на основі системи моделей з різною точністю оцінки.

Трудомісткість розробки ПЗ можна розрахувати за формулою:

$$t = t_{\text{о}} + t_{\text{и}} + t_{\text{а}} + t_{\text{п}} + t_{\text{отл}} + t_{\text{д}}, \text{ людино-годин,} \quad (3.1)$$

де t_o – витрати праці на підготовку й опис поставленої задачі (приймається 50);

t_u – витрати праці на дослідження алгоритму рішення задачі;

t_a – витрати праці на розробку блок-схеми алгоритму;

t_n – витрати праці на програмування по готовій блок-схемі;

$t_{отл}$ – витрати праці на налагодження програми на ЕОМ;

$t_д$ – витрати праці на підготовку документації.

Складові витрати праці визначаються через умовне число операторів у ПЗ, яке розробляється.

Умовне число операторів (підпрограм):

$$Q = q * C * (1 + p), \quad (3.2)$$

де q – передбачуване число операторів;

C – коефіцієнт складності програми;

p – коефіцієнт корекції програми в ході її розробки.

Розрахунок умовного числа операторів з підставкою відповідних значень:

$$Q = 2000 * 1{,}6 * (1 + 0{,}06) = 3392$$

Витрати праці на вивчення опису задачі t_u визначається з урахуванням уточнення опису і кваліфікації програміста:

$$t_{\text{и}} = \frac{Q * B}{(75..85) * k}, \text{ людино-годин,} \quad (3.3)$$

де B – коефіцієнт збільшення витрат праці внаслідок недостатнього опису задачі;

k – коефіцієнт кваліфікації програміста, обумовлений від стажу роботи з даної спеціальності.

Розрахунок витрат праці на вивчення опису задачі з підставкою відповідних значень:

$$t_{\text{и}} = \frac{3392 * 1{,}3}{85 * 1} = 51{,}9 \text{ людино-годин.}$$

Витрати праці на розробку алгоритму рішення задачі:

$$t_{\text{а}} = \frac{Q}{(20..25) * k} \text{, людино-годин.} \quad (3.4)$$

Розрахунок витрат праці на розробку алгоритму рішення задачі з підставкою відповідних значень:

$$t_{\text{а}} = \frac{3392}{25 * 1} = 135{,}68 \text{ людино-годин.}$$

Витрати на складання програми по готовій блок-схемі:

$$t_{\text{п}} = \frac{Q}{(20..25) * k} \text{, людино-годин.} \quad (3.5)$$

Розрахунок витрат на складання програми по готовій блок-схемі з підставкою відповідних значень:

$$t_{\text{п}} = \frac{3392}{24 * 1} = 141{,}3 \text{ людино-годин.}$$

Витрати праці на налагодження програми на ЕОМ:

– за умови автономного налагодження одного завдання:

$$t_{\text{отл}} = \frac{Q}{(4..5) * k} \text{, людино-годин.} \quad (3.6)$$

Розрахунок формули з підставкою відповідних значень:

$$t_{\text{отл}} = \frac{3392}{5 * 1} = 678{,}4 \text{ людино-годин.}$$

– за умови комплексного налагодження завдання:

$$t_{\text{отл}}^{k} = 1{,}5 * t_{\text{отл}} \text{, людино-годин.} \quad (3.7)$$

Розрахунок формули з підставкою відповідних значень:

$$t_{\text{отл}}^{k} = 1{,}5 * 678{,}4 = 1017{,}6 \text{ людино-годин.}$$

Витрати праці на підготовку документації:

$$t_{\text{д}} = t_{\text{др}} + t_{\text{до}} \text{, людино-годин,} \quad (3.8)$$

де $t_{др}$ – трудомісткість підготовки матеріалів і рукопису;

$$t_{\text{др}} = \frac{Q}{(15..20) * k} \text{, людино-годин.} \quad (3.9)$$

$t_{до}$ – трудомісткість редагування, печатки й оформлення документації.

$$t_{\text{до}} = 0{,}75 * t_{\text{др}} \text{ , людино-годин.} \quad (3.10)$$

Розрахунок формул з підставкою відповідних значень:

$$t_{\text{др}} = \frac{3392}{20 * 1} = 169{,}6 \text{ людино-годин.}$$

$$t_{\text{до}} = 0{,}75 * 169{,}6 = 127{,}2 \text{ людино-годин.}$$

$$t_{\text{д}} = 169{,}6 + 127{,}2 = 296{,}8 \text{ людино-годин.}$$

Розрахунок трудомісткості розробки ПЗ за формулою (3.1):

$t = 50 + 51{,}9 + 135{,}68 + 141{,}3 + 678{,}4 + 296{,}8 = 1354{,}08$ людино-годин.

3.2. Розрахунок витрат на створення програми.

Витрати на створення ПЗ *Кпо* включають витрати на заробітну плату виконавця програми *Зз/п* і витрат машинного часу, необхідного на налагодження програми на ЕОМ:

$$\text{К}_{\text{ПО}} = \text{З}_{\text{ЗП}} + \text{З}_{\text{МВ}} \text{ , грн.} \quad (3.11)$$

Заробітна плата виконавців визначається за формулою:

$$\text{З}_{\text{ЗП}} = t * \text{С}_{\text{ПР}} \text{ , грн,} \quad (3.12)$$

де t – загальна трудомісткість, людино-годин;

$C_{пр}$ – середня годинна заробітна плата програміста, грн/година.

Розрахунок заробітної плати виконавців з підставкою відповідних значень:

$$\text{З}_{\text{ЗП}} = 1354{,}08 * 131 = 177384{,}48 \text{ грн.}$$

Вартість машинного часу, необхідного для налагодження програми на ЕОМ:

$$\text{З}_{\text{МВ}} = t_{\text{отл}} * \text{С}_{\text{мч}} \text{ , грн,} \quad (3.13)$$

де $t_{отл}$ – трудомісткість налагодження програми на ЕОМ, год;

$C_{мч}$ – вартість машино-години ЕОМ, грн/год.

Розрахунок вартості машинного часу з підставкою відповідних значень:

$$З_{МВ} = 678{,}4 * 0{,}489 = 331{,}74 \text{ грн.}$$

Розрахунок витрат на створення ПЗ за формулою (3.11):

$$К_{ПО} = 177384{,}48 + 331{,}74 = 177716{,}22 \text{ грн.}$$

Очікуваний період створення ПЗ:

$$T = \frac{t}{B_k * F_p}, \text{ міс,} \quad (3.14)$$

де B_k – число виконавців;

F_p – місячний фонд робочого часу (при 40 годинному робочому тижні F_p=176 годин).

Розрахунок очікуваного періоду створення ПЗ:

$$T = \frac{1354{,}08}{1 * 176} = 7{,}69 \text{ міс.}$$

Висновок: Відповідно до проведених розрахунків, трудомісткість розробки web-застосунку для сервісу доставки їжі за допомогою технологій React JS та ASP.NET Core дорівнює 1354,08 людино-годинам. Вартість розробки даного програмного забезпечення досягає 177716,22 грн, а час потрібний для виконання роботи становить 7,69 місяці.

ВИСНОВКИ

Під час виконання кваліфікаційної роботи був успішно розроблений web-застосунок для сервісу доставки їжі за допомогою технологій React JS та ASP.NET Core. У якості системи керування базами даних була обрана Microsoft SQL Server, а Entity Framework Core обрана як ORM технологія для зв’язку сутностей бази даних та об’єктів класів серверу. Взаємодія з базою даних виконана за допомогою патерну репозиторій. Всі компоненти серверу реалізують патерн ін’єкції залежностей (DI). Клієнтська і серверна частина викладені у окремі публічні репозиторії на платформі GitHub.

Створене програмне забезпечення призначено для того, щоб користувачі могли легко та зручно переглядати меню закладу харчування, додавати бажані продукти в кошик, формувати замовлення, створювати особисті кабінети та переглядати там стан всіх своїх замовлень. Клієнти також мають доступ до перегляду контактів та місцезнаходження закладів. Адміністратори можуть керувати станом бази даних на сторінці адміністрації.

Були виконані такі задачі під час виконання кваліфікаційної роботи:

- проаналізовано проблематику поставленої задачі;
- спроєктовано макет клієнтської частини веб-застосунку;
- визначена архітектура всього програмного забезпечення;
- побудована клієнтська частина веб-застосунку;
- побудована серверна частина веб-застосунку.

Програмне забезпечення складається з двох мало залежних частин: клієнтської та серверної, котрі створені за допомогою різних технологій. Така архітектура додає гнучкості у виборі інструментів побудови застосунку, і, відповідно, кожна з частин програмного забезпечення може бути переробленою або повністю заміненою на іншу технологію, не руйнуючи зв’язок з іншою частиною архітектури.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Repository pattern. [Електронний ресурс] URL: https://dotnettutorials.net/lesson/repository-design-pattern-csharp/ (дата звернення: 06.05.2024).
2. JWT. [Електронний ресурс] URL: https://jwt.io (дата звернення: 06.05.2024).
3. ASP.NET Core. [Електронний ресурс] URL: https://learn.microsoft.com/en-us/aspnet/core/?view=aspnetcore-8.0 (дата звернення: 06.05.2024).
4. C#. [Електронний ресурс] URL: https://learn.microsoft.com/en-us/dotnet/csharp/ (дата звернення: 06.05.2024).
5. Архітектурний стиль REST. [Електронний ресурс] URL: https://restfulapi.net (дата звернення: 06.05.2024).
6. TypeScript. [Електронний ресурс] URL: https://www.typescriptlang.org/docs/ (дата звернення: 06.05.2024).
7. React. [Електронний ресурс] URL: https://react.dev (дата звернення: 06.05.2024).
8. React Router. [Електронний ресурс] URL: https://reactrouter.com/en/main (дата звернення: 06.05.2024).
9. HTML. [Електронний ресурс] URL: https://www.w3schools.com/html/default.asp (дата звернення: 07.05.2024)
10. CSS. [Електронний ресурс] URL: https://www.w3schools.com/css/default.asp (дата звернення: 07.05.2024).
11. Entity Framework Core. [Електронний ресурс] URL: https://learn.microsoft.com/en-us/ef/core/ (дата звернення: 07.05.2024).
12. LINQ. [Електронний ресурс] URL: https://learn.microsoft.com/en-us/dotnet/csharp/linq/ (дата звернення: 07.05.2024).
13. ASP.NET Core Identity. [Електронний ресурс] URL: https://learn.microsoft.com/en-

us/aspnet/core/security/authentication/identity?view=aspnetcore-8.0&tabs=visual-studio (дата звернення: 08.05.2024).

14. Bootstrap. [Електронний ресурс] URL: https://getbootstrap.com/docs/5.3/getting-started/introduction/ (дата звернення: 09.05.2024).

15. Material UI. [Електронний ресурс] URL: https://mui.com/material-ui/getting-started/ (дата звернення: 10.05.2024).

16. JWT аутентифікація в .NET. [Електронний ресурс] URL: https://jasonwatmore.com/post/2021/12/14/net-6-jwt-authentication-tutorial-with-example-api (дата звернення: 11.05.2024).

17. Аутентифікація в ASP.NET Core Web APIs. [Електронний ресурс] URL: https://www.endpointdev.com/blog/2022/06/implementing-authentication-in-asp.net-core-web-apis/ (дата звернення: 12.05.2024).

18. HTTP статус коди. [Електронний ресурс] URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status (дата звернення: 12.05.2024).

19. Програміст: середня зарплата. [Електронний ресурс] URL: https://ua.jooble.org/salary/програміст#hourly (дата звернення: 25.05.2024).

20. Базовий тариф електроенергії для населення. [Електронний ресурс] URL: https://yasno.com.ua/bazovij-tarif-dlya-naselennya-2024 (дата звернення: 25.05.2024).

21. Тарифи інтернету від провайдера «Domonet». [Електронний ресурс] URL: https://dnipro.domonet.ua/uk/prices/apartment (дата звернення: 25.05.2024).

ДОДАТОК А

КОД ПРОГРАМИ

Program.cs // файл точки входу ASP.NET Core застосунку.

```
using AutoMapper;
using DokWokApi.BLL;
using DokWokApi.BLL.Interfaces;
using DokWokApi.BLL.Models.User;
using DokWokApi.BLL.Services;
using DokWokApi.DAL;
using DokWokApi.DAL.Entities;
using DokWokApi.DAL.Interfaces;
using DokWokApi.DAL.Repositories;
using DokWokApi.Middlewares;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Microsoft.OpenApi.Models;
using System.IdentityModel.Tokens.Jwt;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllers();
const string policyName = "ReactProjectCorsPolicy";
builder.Services.AddCors(opts =>
{
   opts.AddPolicy(policyName, policy =>
   {
      policy.WithOrigins(builder.Configuration["AllowedCorsUrls:ReactHttpProject"]!,
         builder.Configuration["AllowedCorsUrls:ReactHttpsProject"]!)
         .AllowAnyHeader()
         .AllowAnyMethod()
         .AllowCredentials();
   });
});
builder.Services.AddDistributedMemoryCache();
builder.Services.AddSession(opts =>
{
   opts.IdleTimeout = TimeSpan.FromMinutes(30);
   opts.Cookie.Name = "DokWokApi.Session";
   opts.Cookie.IsEssential = true;
   opts.Cookie.HttpOnly = true;
});
builder.Services.AddDbContext<StoreDbContext>(opts =>
{
   opts.UseSqlServer(builder.Configuration["ConnectionStrings:FoodStoreConnection"]);
});
builder.Services.AddIdentity<ApplicationUser, IdentityRole>().AddEntityFrameworkStores<StoreDbContext>();
builder.Services.Configure<IdentityOptions>(opts => {
   opts.Password.RequiredLength = 6;
   opts.Password.RequireNonAlphanumeric = false;
   opts.Password.RequireLowercase = false;
   opts.Password.RequireUppercase = true;
   opts.Password.RequireDigit = true;
   opts.User.RequireUniqueEmail = true;
});
var mapperConfig = new MapperConfiguration(mc => mc.AddProfile(new AutomapperProfile()));
IMapper mapper = mapperConfig.CreateMapper();
builder.Services.AddSingleton(mapper);
builder.Services.AddHttpContextAccessor();
builder.Services.AddScoped<IProductCategoryRepository, ProductCategoryRepository>();
```

```
builder.Services.AddScoped<IProductRepository, ProductRepository>();
builder.Services.AddScoped<IOrderRepository, OrderRepository>();
builder.Services.AddScoped<IOrderLineRepository, OrderLineRepository>();
builder.Services.AddScoped<IShopRepository, ShopRepository>();
builder.Services.AddScoped<IProductCategoryService, ProductCategoryService>();
builder.Services.AddScoped<IProductService, ProductService>();
builder.Services.AddScoped<IOrderService, OrderService>();
builder.Services.AddScoped<IOrderLineService, OrderLineService>();
builder.Services.AddScoped<IShopService, ShopService>();
builder.Services.AddScoped<ICartService, SessionCartService>();
builder.Services.AddScoped<IUserService, UserService>();
builder.Services.AddScoped<ISecurityTokenService<UserModel, JwtSecurityToken>, JwtService>();
builder.Services.AddSwaggerGen(c => {
   c.SwaggerDoc("v1", new OpenApiInfo
   {
      Title = "DokWokApi",
      Version = "v1"
   });
});
var app = builder.Build();
app.UseCors(policyName);
app.UseSession();
app.UseMiddleware<JwtMiddleware>();
app.MapControllers();
app.UseSwagger();
app.UseSwaggerUI(options => {
   options.SwaggerEndpoint("/swagger/v1/swagger.json", "WebApp");
});
await SeedData.SeedDatabaseAsync(app);
app.Run();
```

CartController.cs // контролер обробки запитів для роботи з кошиком.

```
using DokWokApi.BLL.Interfaces;
using DokWokApi.BLL.Models.ShoppingCart;
using DokWokApi.Exceptions;
using Microsoft.AspNetCore.Mvc;

namespace DokWokApi.Controllers;

[ApiController]
[Route("api/[controller]")]
public class CartController : ControllerBase
{
   private readonly ICartService _cartService;
   public CartController(ICartService cartService)
   {
      _cartService = cartService;
   }
   [HttpGet]
   [ProducesResponseType(StatusCodes.Status200OK)]
   [ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
   public async Task<ActionResult<Cart>> GetCart()
   {
      try
      {
         var cart = await _cartService.GetCart();
         return Ok(cart);
      }
      catch (CartException ex)
      {
         return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
      }
```

```
    catch (Exception ex)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
    }
}
[HttpPost("item")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType<string>(StatusCodes.Status400BadRequest)]
[ProducesResponseType<string>(StatusCodes.Status404NotFound)]
[ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
public async Task<ActionResult<Cart>> AddProductToCart(long productId, int quantity)
{
    if (quantity <= 0)
    {
        return BadRequest("The quantity value must be greater than 0");
    }

    try
    {
        var modifiedCart = await _cartService.AddItem(productId, quantity);
        return Ok(modifiedCart);
    }
    catch (CartException ex)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
    }
    catch (EntityNotFoundException ex)
    {
        return NotFound(ex.Message);
    }
    catch (Exception ex)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
    }
}
[HttpDelete("item")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType<string>(StatusCodes.Status400BadRequest)]
[ProducesResponseType<string>(StatusCodes.Status404NotFound)]
[ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
public async Task<ActionResult<Cart>> RemoveProductFromCart(long productId, int quantity)
{
    if (quantity <= 0)
    {
        return BadRequest("The quantity value must be greater than 0");
    }

    try
    {
        var modifiedCart = await _cartService.RemoveItem(productId, quantity);
        return Ok(modifiedCart);
    }
    catch (EntityNotFoundException ex)
    {
        return NotFound(ex.Message);
    }
    catch (CartNotFoundException ex)
    {
        return NotFound(ex.Message);
    }
    catch (CartException ex)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
```

```
        }
        catch (Exception ex)
        {
            return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
        }
    }
    [HttpDelete("line")]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType<string>(StatusCodes.Status404NotFound)]
    [ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
    public async Task<ActionResult<Cart>> RemoveLineFromCart(long productId)
    {
        try
        {
            var modifiedCart = await _cartService.RemoveLine(productId);
            return Ok(modifiedCart);
        }
        catch (EntityNotFoundException ex)
        {
            return NotFound(ex.Message);
        }
        catch (CartNotFoundException ex)
        {
            return NotFound(ex.Message);
        }
        catch (CartException ex)
        {
            return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
        }
        catch (Exception ex)
        {
            return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
        }
    }
    [HttpDelete]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
    public async Task<ActionResult> ClearCart()
    {
        try
        {
            await _cartService.ClearCart();
            return Ok("The cart was cleared successfully.");
        }
        catch (CartException ex)
        {
            return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
        }
        catch (Exception ex)
        {
            return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
        }
    }
}
```

OrdersController.cs // контролер обробки запитів для роботи з замовленнями.

```
using AutoMapper;
using DokWokApi.Attributes;
using DokWokApi.BLL;
using DokWokApi.BLL.Interfaces;
using DokWokApi.BLL.Models.Order;
using DokWokApi.Exceptions;
```

```
using Microsoft.AspNetCore.Mvc;

namespace DokWokApi.Controllers;

[ApiController]
[Route("api/[controller]")]
public class OrdersController : ControllerBase
{
    private readonly IOrderService _orderService;
    private readonly IOrderLineService _orderLineService;
    public OrdersController(IOrderService orderService, IOrderLineService orderLineService)
    {
        _orderService = orderService;
        _orderLineService = orderLineService;
    }
    [Authorize(UserRoles.Customer, UserRoles.Admin)]
    [HttpGet]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
    public async Task<ActionResult<IEnumerable<OrderModel>>> GetAllOrders(string? userId)
    {
        try
        {
            var orders = userId is null ?
                await _orderService.GetAllAsync() :
                await _orderService.GetAllByUserIdAsync(userId);

            return Ok(orders);
        }
        catch (Exception ex)
        {
            return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
        }
    }
    [Authorize(UserRoles.Customer, UserRoles.Admin)]
    [HttpGet("{id}")]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType<string>(StatusCodes.Status404NotFound)]
    [ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
    public async Task<ActionResult<OrderModel>> GetOrderById(long id)
    {
        try
        {
            var order = await _orderService.GetByIdAsync(id);
            if (order is null)
            {
                return NotFound("The order was not found.");
            }

            return Ok(order);
        }
        catch (Exception ex)
        {
            return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
        }
    }
    [HttpPost("delivery")]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType<string>(StatusCodes.Status400BadRequest)]
    [ProducesResponseType<string>(StatusCodes.Status404NotFound)]
    [ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
    public async Task<ActionResult<OrderModel>> AddDeliveryOrder(DeliveryOrderForm form, [FromServices]
IMapper mapper)
```

```
    {
        try
        {
            var model = mapper.Map<OrderModel>(form);
            var addedModel = await _orderService.AddOrderFromCartAsync(model);
            return Ok(addedModel);
        }
        catch (ArgumentNullException ex)
        {
            return BadRequest(ex.Message);
        }
        catch (OrderException ex)
        {
            return BadRequest(ex.Message);
        }
        catch (EntityNotFoundException ex)
        {
            return NotFound(ex.Message);
        }
        catch (Exception ex)
        {
            return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
        }
    }
    [HttpPost("takeaway")]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType<string>(StatusCodes.Status400BadRequest)]
    [ProducesResponseType<string>(StatusCodes.Status404NotFound)]
    [ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
    public async Task<ActionResult<OrderModel>> AddTakeawayOrder(TakeawayOrderForm form, [FromServices]
IMapper mapper)
    {
        try
        {
            var model = mapper.Map<OrderModel>(form);
            var addedModel = await _orderService.AddOrderFromCartAsync(model);
            return Ok(addedModel);
        }
        catch (ArgumentNullException ex)
        {
            return BadRequest(ex.Message);
        }
        catch (OrderException ex)
        {
            return BadRequest(ex.Message);
        }
        catch (EntityNotFoundException ex)
        {
            return NotFound(ex.Message);
        }
        catch (Exception ex)
        {
            return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
        }
    }

    [Authorize(UserRoles.Admin)]
    [HttpPut]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType<string>(StatusCodes.Status400BadRequest)]
    [ProducesResponseType<string>(StatusCodes.Status404NotFound)]
    [ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
```

```
    public async Task<ActionResult<OrderModel>> UpdateOrder(OrderPutModel putModel, [FromServices] IMapper
mapper)
    {
        try
        {
            var model = mapper.Map<OrderModel>(putModel);
            var updatedModel = await _orderService.UpdateAsync(model);
            return Ok(updatedModel);
        }
        catch (ArgumentNullException ex)
        {
            return BadRequest(ex.Message);
        }
        catch (EntityNotFoundException ex)
        {
            return NotFound(ex.Message);
        }
        catch (Exception ex)
        {
            return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
        }
    }

    [Authorize(UserRoles.Admin)]
    [HttpDelete("{id}")]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType<string>(StatusCodes.Status404NotFound)]
    [ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
    public async Task<ActionResult> DeleteOrder(long id)
    {
        try
        {
            await _orderService.DeleteAsync(id);
            return Ok();
        }
        catch (EntityNotFoundException ex)
        {
            return NotFound(ex.Message);
        }
        catch (Exception ex)
        {
            return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
        }
    }

    [Authorize(UserRoles.Customer, UserRoles.Admin)]
    [HttpGet("lines")]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
    public async Task<ActionResult<IEnumerable<OrderLineModel>>> GetAllOrderLines(long? orderId)
    {
        try
        {
            var orderLines = orderId.HasValue ?
                await _orderLineService.GetAllByOrderIdAsync(orderId.Value) :
                await _orderLineService.GetAllAsync();

            return Ok(orderLines);
        }
        catch (Exception ex)
        {
            return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
        }
```

```
    }

    [Authorize(UserRoles.Customer, UserRoles.Admin)]
    [HttpGet("lines/{id}")]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType<string>(StatusCodes.Status404NotFound)]
    [ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
    public async Task<ActionResult<OrderLineModel>> GetOrderLineById(long id)
    {
        try
        {
            var orderLine = await _orderLineService.GetByIdAsync(id);
            if (orderLine is null)
            {
                return NotFound("The order line was not found.");
            }

            return Ok(orderLine);
        }
        catch (Exception ex)
        {
            return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
        }
    }

    [Authorize(UserRoles.Customer, UserRoles.Admin)]
    [HttpGet("lines/{orderId}/{productId}")]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType<string>(StatusCodes.Status404NotFound)]
    [ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
    public async Task<ActionResult<OrderLineModel>> GetOrderLineByOrderAndProductIds(long orderId, long
productId)
    {
        try
        {
            var orderLine = await _orderLineService.GetByOrderAndProductIdsAsync(orderId, productId);
            if (orderLine is null)
            {
                return NotFound("The order line was not found.");
            }

            return Ok(orderLine);
        }
        catch (Exception ex)
        {
            return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
        }
    }

    [Authorize(UserRoles.Admin)]
    [HttpPost("lines")]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType<string>(StatusCodes.Status400BadRequest)]
    [ProducesResponseType<string>(StatusCodes.Status404NotFound)]
    [ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
    public async Task<ActionResult<OrderLineModel>> AddOrderLine(OrderLinePostModel postModel,
[FromServices] IMapper mapper)
    {
        try
        {
            var model = mapper.Map<OrderLineModel>(postModel);
            var addedModel = await _orderLineService.AddAsync(model);
            return Ok(addedModel);
```

```
        }
        catch (ArgumentNullException ex)
        {
            return BadRequest(ex.Message);
        }
        catch (ArgumentException ex)
        {
            return BadRequest(ex.Message);
        }
        catch (EntityNotFoundException ex)
        {
            return NotFound(ex.Message);
        }
        catch (Exception ex)
        {
            return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
        }
    }

    [Authorize(UserRoles.Admin)]
    [HttpPut("lines")]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType<string>(StatusCodes.Status400BadRequest)]
    [ProducesResponseType<string>(StatusCodes.Status404NotFound)]
    [ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
    public async Task<ActionResult<OrderLineModel>> UpdateOrderLine(OrderLinePutModel putModel,
[FromServices] IMapper mapper)
    {
        try
        {
            var model = mapper.Map<OrderLineModel>(putModel);
            var updatedModel = await _orderLineService.UpdateAsync(model);
            return Ok(updatedModel);
        }
        catch (ArgumentNullException ex)
        {
            return BadRequest(ex.Message);
        }
        catch (ArgumentException ex)
        {
            return BadRequest(ex.Message);
        }
        catch (EntityNotFoundException ex)
        {
            return NotFound(ex.Message);
        }
        catch (Exception ex)
        {
            return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
        }
    }

    [Authorize(UserRoles.Admin)]
    [HttpDelete("lines/{id}")]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType<string>(StatusCodes.Status404NotFound)]
    [ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
    public async Task<ActionResult> DeleteOrderLine(long id)
    {
        try
        {
            await _orderLineService.DeleteAsync(id);
            return Ok();
```

```
        }
        catch (EntityNotFoundException ex)
        {
            return NotFound(ex.Message);
        }
        catch (Exception ex)
        {
            return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
        }
    }
}
```

ProductsController.cs // контролер обробки запитів для роботи з продуктами.

```
using AutoMapper;
using DokWokApi.Attributes;
using DokWokApi.BLL;
using DokWokApi.BLL.Interfaces;
using DokWokApi.BLL.Models.Product;
using DokWokApi.BLL.Models.ProductCategory;
using DokWokApi.Exceptions;
using Microsoft.AspNetCore.Mvc;

namespace DokWokApi.Controllers;

[ApiController]
[Route("api/[controller]")]
public class ProductsController : ControllerBase
{
    private readonly IProductService _productService;

    private readonly IProductCategoryService _categoryService;

    public ProductsController(IProductService productService, IProductCategoryService categoryService)
    {
        _productService = productService;
        _categoryService = categoryService;
    }

    [HttpGet]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
    public async Task<ActionResult<IEnumerable<ProductModel>>> GetAllProducts(long? categoryId)
    {
        try
        {
            var products = categoryId.HasValue ?
                await _productService.GetAllByCategoryIdAsync(categoryId.Value) :
                await _productService.GetAllAsync();

            return Ok(products);
        }
        catch (Exception ex)
        {
            return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
        }
    }

    [HttpGet("{id}")]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType<string>(StatusCodes.Status404NotFound)]
    [ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
    public async Task<ActionResult<ProductModel>> GetProductById(long id)
```

```
    {
        try
        {
            var product = await _productService.GetByIdAsync(id);
            if (product is null)
            {
                return NotFound("The product was not found.");
            }

            return Ok(product);
        }
        catch (Exception ex)
        {
            return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
        }
    }

    [Authorize(UserRoles.Admin)]
    [HttpPost]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType<string>(StatusCodes.Status400BadRequest)]
    [ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
    public async Task<ActionResult<ProductModel>> AddProduct(ProductPostModel postModel, [FromServices]
IMapper mapper)
    {
        try
        {
            var model = mapper.Map<ProductModel>(postModel);
            var addedModel = await _productService.AddAsync(model);
            return Ok(addedModel);
        }
        catch (ArgumentNullException ex)
        {
            return BadRequest(ex.Message);
        }
        catch (ArgumentException ex)
        {
            return BadRequest(ex.Message);
        }
        catch (Exception ex)
        {
            return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
        }
    }

    [Authorize(UserRoles.Admin)]
    [HttpPut]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType<string>(StatusCodes.Status400BadRequest)]
    [ProducesResponseType<string>(StatusCodes.Status404NotFound)]
    [ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
    public async Task<ActionResult<ProductModel>> UpdateProduct(ProductPutModel putModel, [FromServices]
IMapper mapper)
    {
        try
        {
            var model = mapper.Map<ProductModel>(putModel);
            var updatedModel = await _productService.UpdateAsync(model);
            return Ok(updatedModel);
        }
        catch (ArgumentNullException ex)
        {
            return BadRequest(ex.Message);
```

```
    }
    catch (EntityNotFoundException ex)
    {
        return NotFound(ex.Message);
    }
    catch (ArgumentException ex)
    {
        return BadRequest(ex.Message);
    }
    catch (Exception ex)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
    }
}

[Authorize(UserRoles.Admin)]
[HttpDelete("{id}")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType<string>(StatusCodes.Status404NotFound)]
[ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
public async Task<ActionResult> DeleteProduct(long id)
{
    try
    {
        await _productService.DeleteAsync(id);
        return Ok();
    }
    catch (EntityNotFoundException ex)
    {
        return NotFound(ex.Message);
    }
    catch (Exception ex)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
    }
}


[HttpGet("isNameTaken/{name}")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType<string>(StatusCodes.Status400BadRequest)]
[ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
public async Task<ActionResult> IsProductNameTaken(string name)
{
    try
    {
        var isTaken = await _productService.IsNameTaken(name);
        return new JsonResult(new { isTaken }) { StatusCode = StatusCodes.Status200OK };
    }
    catch (ArgumentNullException ex)
    {
        return BadRequest(ex.Message);
    }
    catch (Exception ex)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
    }
}

[HttpGet("categories")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
public async Task<ActionResult<IEnumerable<ProductCategoryModel>>> GetAllCategories()
```

```
    {
        try
        {
            var categories = await _categoryService.GetAllAsync();
            return Ok(categories);
        }
        catch (Exception ex)
        {
            return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
        }
    }

    [HttpGet("categories/{id}")]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType<string>(StatusCodes.Status404NotFound)]
    [ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
    public async Task<ActionResult<ProductCategoryModel>> GetCategoryById(long id)
    {
        try
        {
            var category = await _categoryService.GetByIdAsync(id);
            if (category is null)
            {
                return NotFound("The product category was not found.");
            }

            return Ok(category);
        }
        catch (Exception ex)
        {
            return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
        }
    }

    [Authorize(UserRoles.Admin)]
    [HttpPost("categories")]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType<string>(StatusCodes.Status400BadRequest)]
    [ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
    public async Task<ActionResult<ProductCategoryModel>> AddCategory(ProductCategoryPostModel postModel,
[FromServices] IMapper mapper)
    {
        try
        {
            var model = mapper.Map<ProductCategoryModel>(postModel);
            var addedModel = await _categoryService.AddAsync(model);
            return Ok(addedModel);
        }
        catch (ArgumentNullException ex)
        {
            return BadRequest(ex.Message);
        }
        catch (ArgumentException ex)
        {
            return BadRequest(ex.Message);
        }
        catch (Exception ex)
        {
            return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
        }
    }

    [Authorize(UserRoles.Admin)]
```

```
    [HttpPut("categories")]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType<string>(StatusCodes.Status400BadRequest)]
    [ProducesResponseType<string>(StatusCodes.Status404NotFound)]
    [ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
    public async Task<ActionResult<ProductCategoryModel>> UpdateCategory(ProductCategoryPutModel putModel,
[FromServices] IMapper mapper)
    {
        try
        {
            var model = mapper.Map<ProductCategoryModel>(putModel);
            var updatedModel = await _categoryService.UpdateAsync(model);
            return Ok(updatedModel);
        }
        catch (ArgumentNullException ex)
        {
            return BadRequest(ex.Message);
        }
        catch (EntityNotFoundException ex)
        {
            return NotFound(ex.Message);
        }
        catch (ArgumentException ex)
        {
            return BadRequest(ex.Message);
        }
        catch (Exception ex)
        {
            return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
        }
    }

    [Authorize(UserRoles.Admin)]
    [HttpDelete("categories/{id}")]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType<string>(StatusCodes.Status404NotFound)]
    [ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
    public async Task<ActionResult> DeleteCategory(long id)
    {
        try
        {
            await _categoryService.DeleteAsync(id);
            return Ok();
        }
        catch (EntityNotFoundException ex)
        {
            return NotFound(ex.Message);
        }
        catch (Exception ex)
        {
            return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
        }
    }

    [HttpGet("categories/isNameTaken/{name}")]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType<string>(StatusCodes.Status400BadRequest)]
    [ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
    public async Task<ActionResult> IsCategoryNameTaken(string name)
    {
        try
        {
            var isTaken = await _categoryService.IsNameTaken(name);
```

```
            return new JsonResult(new { isTaken }) { StatusCode = StatusCodes.Status200OK };
        }
        catch (ArgumentNullException ex)
        {
            return BadRequest(ex.Message);
        }
        catch (Exception ex)
        {
            return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
        }
    }
}
```

UsersController.cs // контролер обробки запитів для роботи з користувачами.

```
using AutoMapper;
using DokWokApi.BLL.Interfaces;
using DokWokApi.Exceptions;
using Microsoft.AspNetCore.Mvc;
using DokWokApi.BLL.Models.User;
using DokWokApi.Attributes;
using DokWokApi.BLL;

namespace DokWokApi.Controllers;

[ApiController]
[Route("api/[controller]")]
public class UsersController : ControllerBase
{
    private readonly IUserService _userService;

    public UsersController(IUserService userService)
    {
        _userService = userService;
    }
    [Authorize(UserRoles.Admin)]
    [HttpGet]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
    public async Task<ActionResult<IEnumerable<UserModel>>> GetAllUsers()
    {
        try
        {
            var users = await _userService.GetAllAsync();

            return Ok(users);
        }
        catch (Exception ex)
        {
            return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
        }
    }
    [Authorize(UserRoles.Admin)]
    [HttpGet("customers")]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
    public async Task<ActionResult<IEnumerable<UserModel>>> GetAllCustomers()
    {
        try
        {
            var customers = await _userService.GetAllCustomersAsync();

            return Ok(customers);
```

```
    }
    catch (Exception ex)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
    }
}
[Authorize(UserRoles.Customer, UserRoles.Admin)]
[HttpGet("username/{userName}")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType<string>(StatusCodes.Status401Unauthorized)]
[ProducesResponseType<string>(StatusCodes.Status403Forbidden)]
[ProducesResponseType<string>(StatusCodes.Status404NotFound)]
[ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
public async Task<ActionResult<UserModel>> GetUserByUserName(string userName, bool isAdmin = false)
{
    if (!isAdmin)
    {
        if (HttpContext.Items["User"] is not UserModel authenticatedUser)
        {
            return Unauthorized("Unauthorized access.");
        }
        else if (authenticatedUser.UserName != userName)
        {
            return StatusCode(StatusCodes.Status403Forbidden, "Action is not allowed.");
        }
    }

    try
    {
        var user = await _userService.GetByUserNameAsync(userName);
        if (user is null)
        {
            return NotFound("The user was not found.");
        }

        return Ok(user);
    }
    catch (Exception ex)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
    }
}
[Authorize(UserRoles.Customer, UserRoles.Admin)]
[HttpGet("id/{id}")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType<string>(StatusCodes.Status401Unauthorized)]
[ProducesResponseType<string>(StatusCodes.Status403Forbidden)]
[ProducesResponseType<string>(StatusCodes.Status404NotFound)]
[ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
public async Task<ActionResult<UserModel>> GetUserById(string id, bool isAdmin = false)
{
    if (!isAdmin)
    {
        if (HttpContext.Items["User"] is not UserModel authenticatedUser)
        {
            return Unauthorized("Unauthorized access.");
        }
        else if (authenticatedUser.Id != id)
        {
            return StatusCode(StatusCodes.Status403Forbidden, "Action is not allowed.");
        }
    }
```

```
        try
        {
            var user = await _userService.GetByIdAsync(id);
            if (user is null)
            {
                return NotFound("The user was not found.");
            }

            return Ok(user);
        }
        catch (Exception ex)
        {
            return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
        }
    }
    [Authorize(UserRoles.Customer, UserRoles.Admin)]
    [HttpGet("customers/id/{id}")]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType<string>(StatusCodes.Status401Unauthorized)]
    [ProducesResponseType<string>(StatusCodes.Status403Forbidden)]
    [ProducesResponseType<string>(StatusCodes.Status404NotFound)]
    [ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
    public async Task<ActionResult<UserModel>> GetCustomerById(string id, bool isAdmin = false)
    {
        if (!isAdmin)
        {
            if (HttpContext.Items["User"] is not UserModel authenticatedUser)
            {
                return Unauthorized("Unauthorized access.");
            }
            else if (authenticatedUser.Id != id)
            {
                return StatusCode(StatusCodes.Status403Forbidden, "Action is not allowed.");
            }
        }

        try
        {
            var user = await _userService.GetCustomerByIdAsync(id);
            if (user is null)
            {
                return NotFound("The customer was not found.");
            }

            return Ok(user);
        }
        catch (Exception ex)
        {
            return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
        }
    }
    [Authorize(UserRoles.Admin)]
    [HttpPost]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType<string>(StatusCodes.Status400BadRequest)]
    [ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
    public async Task<ActionResult<UserModel>> AddUser(UserRegisterModel postModel, [FromServices] IMapper
mapper)
    {
        try
        {
            var model = mapper.Map<UserModel>(postModel);
            var addedModel = await _userService.AddAsync(model, postModel.Password!);
```

```
            return Ok(addedModel);
        }
        catch (ArgumentNullException ex)
        {
            return BadRequest(ex.Message);
        }
        catch (UserException ex)
        {
            return BadRequest(ex.Message);
        }
        catch (Exception ex)
        {
            return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
        }
    }
    [Authorize(UserRoles.Customer, UserRoles.Admin)]
    [HttpPut]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType<string>(StatusCodes.Status400BadRequest)]
    [ProducesResponseType<string>(StatusCodes.Status404NotFound)]
    [ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
    public async Task<ActionResult<UserModel>> UpdateUser(UserPutModel putModel, [FromServices] IMapper
mapper, bool isAdmin = false)
    {
        if (!isAdmin)
        {
            if (HttpContext.Items["User"] is not UserModel authenticatedUser)
            {
                return Unauthorized("Unauthorized access.");
            }
            else if (authenticatedUser.Id != putModel.Id)
            {
                return StatusCode(StatusCodes.Status403Forbidden, "Action is not allowed.");
            }
        }

        try
        {
            var model = mapper.Map<UserModel>(putModel);
            var updatedModel = await _userService.UpdateAsync(model);
            return Ok(updatedModel);
        }
        catch (ArgumentNullException ex)
        {
            return BadRequest(ex.Message);
        }
        catch (UserException ex)
        {
            return BadRequest(ex.Message);
        }
        catch (EntityNotFoundException ex)
        {
            return NotFound(ex.Message);
        }
        catch (Exception ex)
        {
            return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
        }
    }
    [Authorize(UserRoles.Customer)]
    [HttpPut("password")]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType<string>(StatusCodes.Status400BadRequest)]
```

```
[ProducesResponseType<string>(StatusCodes.Status404NotFound)]
[ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
public async Task<ActionResult> UpdateCustomerPassword(UserPasswordChangeModel model)
{
    if (HttpContext.Items["User"] is not UserModel authenticatedUser)
    {
        return Unauthorized("Unauthorized access.");
    }
    else if (authenticatedUser.Id != model.UserId)
    {
        return StatusCode(StatusCodes.Status403Forbidden, "Action is not allowed.");
    }

    try
    {
        await _userService.UpdateCustomerPasswordAsync(model);
        return Ok();
    }
    catch (ArgumentNullException ex)
    {
        return BadRequest(ex.Message);
    }
    catch (UserException ex)
    {
        return BadRequest(ex.Message);
    }
    catch (EntityNotFoundException ex)
    {
        return NotFound(ex.Message);
    }
    catch (Exception ex)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
    }
}
[Authorize(UserRoles.Admin)]
[HttpPut("password/asAdmin")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType<string>(StatusCodes.Status400BadRequest)]
[ProducesResponseType<string>(StatusCodes.Status404NotFound)]
[ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
public async Task<ActionResult> UpdateCustomerPasswordAsAdmin(UserPasswordChangeAsAdminModel model)
{
    try
    {
        await _userService.UpdateCustomerPasswordAsAdminAsync(model);
        return Ok();
    }
    catch (ArgumentNullException ex)
    {
        return BadRequest(ex.Message);
    }
    catch (UserException ex)
    {
        return BadRequest(ex.Message);
    }
    catch (EntityNotFoundException ex)
    {
        return NotFound(ex.Message);
    }
    catch (Exception ex)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
```

```
    }
}
[Authorize(UserRoles.Admin)]
[HttpDelete("{id}")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType<string>(StatusCodes.Status400BadRequest)]
[ProducesResponseType<string>(StatusCodes.Status404NotFound)]
[ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
public async Task<ActionResult> DeleteUserById(string id)
{
    try
    {
        await _userService.DeleteAsync(id);
        return Ok();
    }
    catch (EntityNotFoundException ex)
    {
        return NotFound(ex.Message);
    }
    catch (UserException ex)
    {
        return BadRequest(ex.Message);
    }
    catch (Exception ex)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
    }
}
[HttpPost("customers/login")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType<string>(StatusCodes.Status400BadRequest)]
[ProducesResponseType<string>(StatusCodes.Status404NotFound)]
[ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
public async Task<ActionResult<UserModel>> LoginCustomer(UserLoginModel loginModel)
{
    try
    {
        var loggedInUser = await _userService.AuthenticateCustomerLoginAsync(loginModel);
        return Ok(loggedInUser);
    }
    catch (ArgumentNullException ex)
    {
        return BadRequest(ex.Message);
    }
    catch (UserException ex)
    {
        return BadRequest(ex.Message);
    }
    catch (EntityNotFoundException ex)
    {
        return NotFound(ex.Message);
    }
    catch (Exception ex)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
    }
}
[HttpPost("admins/login")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType<string>(StatusCodes.Status400BadRequest)]
[ProducesResponseType<string>(StatusCodes.Status404NotFound)]
[ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
public async Task<ActionResult<UserModel>> LoginAdmin(UserLoginModel loginModel)
```

```
{
    try
    {
        var loggedInUser = await _userService.AuthenticateAdminLoginAsync(loginModel);
        return Ok(loggedInUser);
    }
    catch (ArgumentNullException ex)
    {
        return BadRequest(ex.Message);
    }
    catch (UserException ex)
    {
        return BadRequest(ex.Message);
    }
    catch (EntityNotFoundException ex)
    {
        return NotFound(ex.Message);
    }
    catch (Exception ex)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
    }
}
[HttpPost("register")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType<string>(StatusCodes.Status400BadRequest)]
[ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
public async Task<ActionResult<UserModel>> RegisterUser(UserRegisterModel registerModel)
{
    try
    {
        var registeredUser = await _userService.AuthenticateRegisterAsync(registerModel);
        return Ok(registeredUser);
    }
    catch (ArgumentNullException ex)
    {
        return BadRequest(ex.Message);
    }
    catch (UserException ex)
    {
        return BadRequest(ex.Message);
    }
    catch (Exception ex)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
    }
}
[HttpGet("customers/isloggedin")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType<string>(StatusCodes.Status401Unauthorized)]
[ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
public async Task<ActionResult<UserModel>> IsCustomerLoggedIn()
{
    try
    {
        var user = await _userService.IsCustomerLoggedInAsync();
        if (user is null)
        {
            return Unauthorized("Unauthorized.");
        }

        return Ok(user);
    }
```

```
    catch (Exception ex)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
    }
}
[HttpGet("admins/isloggedin")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType<string>(StatusCodes.Status401Unauthorized)]
[ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
public async Task<ActionResult<UserModel>> IsAdminLoggedIn()
{
    try
    {
        var user = await _userService.IsAdminLoggedInAsync();
        if (user is null)
        {
            return Unauthorized("Unauthorized.");
        }

        return Ok(user);
    }
    catch (Exception ex)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
    }
}
[Authorize(UserRoles.Customer, UserRoles.Admin)]
[HttpGet("logout")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
public async Task<ActionResult> LogOutUser()
{
    try
    {
        await _userService.LogOutAsync();
        return Ok();
    }
    catch (Exception ex)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
    }
}
[Authorize(UserRoles.Admin)]
[HttpGet("roles")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType<string>(StatusCodes.Status404NotFound)]
[ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
public async Task<ActionResult<IEnumerable<string>>> GetUserRoles(string userId)
{
    try
    {
        var roles = await _userService.GetUserRolesAsync(userId);
        return Ok(roles);
    }
    catch (EntityNotFoundException ex)
    {
        return NotFound(ex.Message);
    }
    catch (Exception ex)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
    }
}
```

```
    [HttpGet("customers/isUserNameTaken/{userName}")]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType<string>(StatusCodes.Status400BadRequest)]
    [ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
    public async Task<ActionResult> IsUserNameTaken(string userName)
    {
        try
        {
            var isTaken = await _userService.IsUserNameTaken(userName);
            return new JsonResult(new { isTaken }) { StatusCode = StatusCodes.Status200OK };
        }
        catch (ArgumentNullException ex)
        {
            return BadRequest(ex.Message);
        }
        catch (Exception ex)
        {
            return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
        }
    }
    [HttpGet("customers/isEmailTaken/{email}")]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType<string>(StatusCodes.Status400BadRequest)]
    [ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
    public async Task<ActionResult> IsEmailTaken(string email)
    {
        try
        {
            var isTaken = await _userService.IsEmailTaken(email);
            return new JsonResult(new { isTaken }) { StatusCode = StatusCodes.Status200OK };
        }
        catch (ArgumentNullException ex)
        {
            return BadRequest(ex.Message);
        }
        catch (Exception ex)
        {
            return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
        }
    }
    [HttpGet("customers/isPhoneTaken/{phoneNumber}")]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType<string>(StatusCodes.Status400BadRequest)]
    [ProducesResponseType<string>(StatusCodes.Status500InternalServerError)]
    public async Task<ActionResult> IsPhoneNumberTaken(string phoneNumber)
    {
        try
        {
            var isTaken = await _userService.IsPhoneNumberTaken(phoneNumber);
            return new JsonResult(new { isTaken }) { StatusCode = StatusCodes.Status200OK };
        }
        catch (ArgumentNullException ex)
        {
            return BadRequest(ex.Message);
        }
        catch (Exception ex)
        {
            return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
        }
    }
}
```

Увесь вихідний код застосунку надається окремо на диску.

ДОДАТОК Б

ВІДГУК

керівника економічного розділу

на кваліфікаційну роботу бакалавра

на тему:

«Розробка web-застосунку для сервісу доставки їжі за допомогою технологій React JS та ASP.NET Core»

студента групи 122-20-3 Величка Ігоря Юрійовича

Керівник економічного розділу
доцент каф. ПЕП та ПУ, к.е.н **Л. В. Касьяненко**

ДОДАТОК В

ПЕРЕЛІК ДОКУМЕНТІВ НА ДИСКУ

Ім’я файлу	**Опис**
Пояснювальні документи	
КР_Величко_І.Ю.docx	Пояснювальна записка до кваліфікаційної роботи. Документ Word.
КР_Величко_І.Ю.pdf	Пояснювальна записка до кваліфікаційної роботи у форматі PDF.
Програма	
dokwok.zip	Архів. Містить коди програми і скомпільовану програму.
Презентація	
Презентація_Величко_І.Ю.pptx	Презентація кваліфікаційної роботи