МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра комп'ютерних наук

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

на тему: **«РОЗРОБКА ВЕБЗАСТОСУНКУ ДЛЯ ЕЛЕКТРОННОЇ КОМЕРЦІЇ З ВИКОРИСТАННЯМ ПРИНЦИПУ ІНВЕРСІЇ КЕРУВАННЯ»**

Виконав: студент 4 курсу, групи 6.1229
спеціальності 122 комп'ютерні науки
(шифр і назва спеціальності)
освітньої програми комп'ютерні науки
(назва освітньої програми)
Н.В. Вовк
(ініціали та прізвище)
Керівник завідувач кафедри комп'ютерних наук, доцент, д.т.н. Шило Г. М.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)
Рецензент завідувач кафедри програмної інженерії, доцент, Лісняк А. О.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2023

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра комп'ютерних наук

Рівень вищої освіти бакалавр

Спеціальність 122 комп'ютерні науки
(шифр і назва)

Освітня програма комп'ютерні науки

ЗАТВЕРДЖУЮ
Завідувач кафедри комп'ютерних наук,
д.т.н., професор

_____ Чопоров С.В.
(підпис)

" ____ " _____ 2023 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Вовку Назарію Володимировичу
(прізвище, ім'я та по-батькові)

1. Тема роботи Розробка вебзастосунку для електронної комерції з використанням принципу інверсії керування.

керівник роботи Шило Галина Миколаївна, професор кафедри комп'ютерних наук, д.т.н., доцент
(прізвище, ім'я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 26 » січня 2023 року № 102-с

2. Строк подання студентом роботи 07.06.2023

3. Вихідні дані до роботи 1.Технічне завдання розробки вебзастосунку;
2. Статті та книги про паттерни інверсії контролю та ін'єкції залежностей;
3. Документація програмних інструментів розробки

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Постановка задачі.
2. Основні теоретичні відомості.
3. Розробка вебзастосунку.
4. Тестування вебзастосунку.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) презентація

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання 30.01.2023

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	30.01.2023	
2.	Збір вихідних даних.	01.02.2023	
3.	Обробка методичних та теоретичних	13.02.2023	
	джерел.		
4.	Розробка першого та другого розділу.	01.03.2023	
5.	Розробка практичної частини та третього розділу.	06.03.2023	
6.	Оформлення та нормоконтроль	24.05.2023	
	кваліфікаційної роботи бакалавра.		
7.	Захист кваліфікаційної роботи.	20.06.2023	

Студент _____ Н. В. Вовк
(підпис) (ініціали та прізвище)

Керівник роботи _____ Г. М. Шило
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ О. Г. Спиця
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота бакалавра «Розробка вебзастосунку для електронної комерції з використанням принципу інверсії керування»: 43 с., 30 рис., 14 джерел.

ВЕБЗАСТОСУНОК, ІНВЕРСІЯ ІН’ЄКЦІЯ ЗАЛЕЖНОСТЕЙ, КОНТРОЛЮ, МОДУЛЬНА АРХІТЕКТУРА, СТАТИЧНА ГЕНЕРАЦІЯ.

Об’єкт дослідження: паттерн інверсії контролю в розробці веб-застосунків.

Мета роботи: розробити веб-застосунок для електронної комерції використовуючи архітектуру інверсії контролю.

Структура роботи складається з вступу, трьох розділів, висновків та списку літератури. Результатом роботи є вебсайт, доступний у мережі інтернет, опис архітектури додатку та аналіз переваг обраної архітектури.

SUMMARY

Bachelor’s qualifying thesis «Development of the E-commerce Web Application Using Inversion of Control Programming Principle»: 43 pages, 30 pictures, 14 sources.

Inversion of control, dependency injection, web application, static generation, modular architecture. Аналогічно і відповідно укр реферату

The subject of research is the Inversion of control pattern in the development of web applications.

The aim of the work is to develop an ecommerce web application for a publishing house, using the inversion of control programming architecture principle. The result is a working website, available on the internet, a description of application architecture, and an analysis of advantages of the chosen architecture. The thesis consists of an introduction, three sections, takeaways and a list of sources.

ЗМІСТ

ВСТУП

При відкритті бізнесу, орієнтованого на розповсюдження продукції через інтернет, в першу чергу постає питання розробки вебсайту. Ще 15-20 років тому це означало великі витрати на розробку, адміністрацію, хостинг. Останніми ж роками комерція в Інтернеті помітно спростилась. Інструменти на кшталт Shopify та Squarespace дозволяють швидко створити вебсайт навіть без знань програмування. А популярніша серед більших інтернет-магазинів зв'язка системи менеджменту контенту WordPress та плагіну WooCommerce займає 38.74% ринку електронної комерції.

Розробляючи інтернет-магазин видавництва, що є поставленою задачею, було проведено аналіз вебсайтів українських видавництв. Виходячи з економічних міркувань, було б доцільно піти шляхом аналогічних підприємств, і побудувати інтернет-магазин використовуючи готові інструменти. Та з спрощенням для власників і операторів платформ прийшли недоліки для кінцевих користувачів. Поки вебтехнології роками розвивались і в інших сферах вебсайти переходили на сучасніші інструменти, мови програмування та фреймворки, в електронній комерції відбулася певна стагнація. Це призвело до того, що за сучасними мірками багато сайтів, побудованих за допомогою вищезгаданих інструментів помітно відстають у функціоналі та швидкості роботи від інших типів вебсайтів. Тому було прийнято рішення побудувати систему на основі найактуальніших інструментів, що використовуються у веброзробці на даний момент.

Мета роботи: розробити вебсайт для інтернет-магазину видавництва, використовуючи сучасні технології та інструменти, що забезпечать високу швидкість роботи, гнучкість та зручність використання для кінцевого користувача.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Постановка проблеми

Продовжуючи порівняння із згаданими у вступі інструментами, основна їх проблема не стільки в застарілих технологіях, а в тому, що вони дуже міцно до них прив’язані. Таким чином, наприклад, WordPress прив’язаний до бази даних MySQL та свого власного формату бази. І якщо з часом виникає потреба змінити бізнес-логіку, місце чи/та провайдера будь-якого сервісу, що використовується у вебзастосунку – це доводиться робити в рамках обмежень монолітної системи.

Основу проблеми складає аксіома: передбачити всіх майбутніх вимог бізнесу і змін застосунку неможливо. Тому найкращою стратегією є розробка застосунку з урахуванням майбутніх змін.

В цьому випадку при побудові системи допомагає архітектурний принцип Інверсії Контролю (Inversion of Control). Він дозволяє заміняти модулі програми, при цьому не змінюючи компоненти системи, які ці модулі безпосередньо використовують. Це дозволяє не тільки створити систему, що готова до майбутніх змін, але й забезпечити гнучкість її розвитку та адаптації до нових вимог бізнесу.

На рис. 1.1 схематично показані відмінності архітектури інверсії контролю та традиційного підходу. Діаграми 2 та 3 ілюструють один і той самий додаток у різних конфігураціях або на різних етапах розвитку.

Використання інверсії контролю сприяє впровадженню модульної архітектури, коли система розділена на незалежні компоненти, що можуть бути розроблені, тестовані та оновлені окремо. Це дає змогу зосередитись на розвитку окремих елементів системи, швидше відповідати на зміни вимог до продукту та забезпечувати високу якість коду.

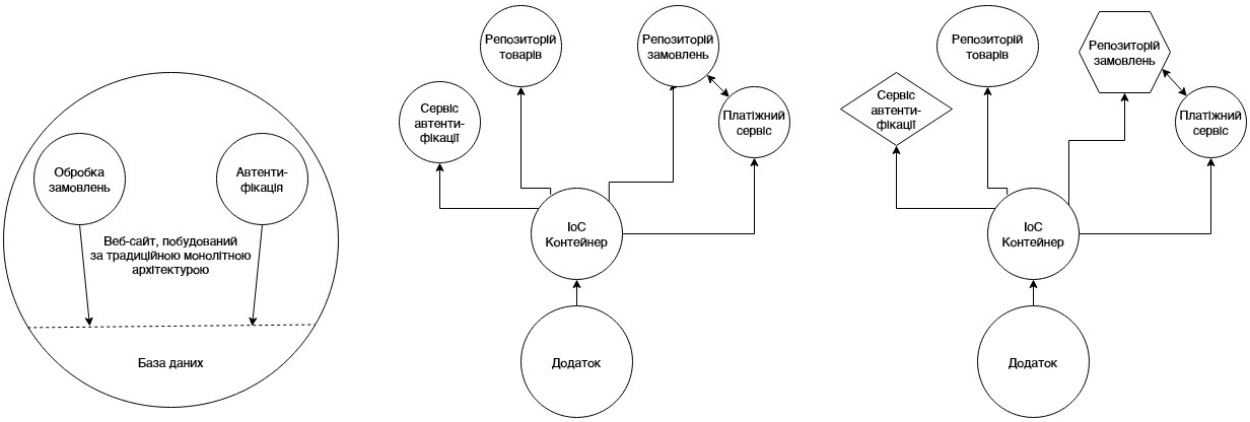


Рисунок 1.1 – Порівняння традиційної архітектури та паттерну інверсії контролю

1.2 Теоретичні основи паттерну Інверсії Контролю

Основна ідея Інверсії Контролю полягає у зміні відносин між модулями: замість того, щоб активний модуль сам створював і викликав інший модуль, він пропонує свої послуги через інтерфейс, і центральний компонент системи (контейнер IoC) забезпечує зв'язок між модулями. Під модулями зазвичай маються на увазі інстанції классів (об’єкти) в об’єктно-орієнтованому програмуванні, хоча можуть бути представлені і інкаше, в залежності від можливостей мови програмування.

Один з найрозповсюдженіших способів реалізації інверсії контролю – використання Ін’єкції Залежностей (Dependency Injection або DI). Цей принцип полягає у передачі залежностей об'єкта (тобто інших об'єктів, з якими він взаємодіє) ззовні, замість інстанціювання залежностей всередині самого об'єкта.

На прикладі псевдо-коду це виглядає так (рис. 1.2):

```
class ServiceA {
    getData() {
    }
}

class ServiceB {
    constructor(container) {
        this.container = container
    }
    printData() {
        data = this.container.serviceA.getData()
        console.table(data)
    }
}

container = new Container({
    serviceA: ServiceA,
    serviceB: ServiceB,
})

container.serviceB.printData() // Надрукує дані з сервісу A
```

Рисунок 1.2 – Приклад паттерну інверсії контролю

1.2.1 Недоліки Інверсії Контролю

Хоча інверсія контролю має значні переваги, також існують деякі недоліки, які варто враховувати. Інверсія контролю може спонукати до створення надмірно універсального коду, який ускладнює розуміння та підтримку. У таких ситуаціях, важливо знайти оптимальний баланс між гнучкістю та простотою. Треба застосовувати інверсію контролю з розумом, не допускаючи перекладення зайвої відповідальності на контейнер або надмірної загальності інтерфейсів.

Також, через незалежність модулів, може бути знижена продуктивність та швидкість роботи системи. Наприклад, якщо модулі в проєкті залежать від різних баз даних чи сервісів, то запити можуть займати довший час порівняно з монолітною системою виключно через фізичну відстань.

1.3 Вимоги до вебсайту

Враховуючи аналіз предметної області та проблеми, що можуть виникнути при використанні традиційних підходів, було сформульовано основні цілі та завдання розробки інтернет-магазину видавництва:

- розробка вебсайту інтернет-магазину на основі сучасних технологій, фреймворку та мови програмування, що відповідають актуальним вимогам та стандартам веброзробки;
- впровадження принципу інверсії контролю для полегшення заміни та інтеграції модулів системи, забезпечення гнучкості архітектури та адаптації до нових вимог;
- реалізація необхідного функціоналу для комфортного використання сайту як клієнтами, так і адміністрацією видавництва, включаючи каталог товарів, систему замовлень, корзину, модуль обробки платежів та інші відповідні функції;
- забезпечення високого рівня безпеки, швидкодії та стабільності роботи інтернет-магазину, а також оптимізація сайту.

Вебзастосунок має мати такий функціонал:

а) каталог товарів;
б) фунціонал кошика;
в) обробка платежів;
г) авторизація, як за допомогою email та паролю, так і за допомогою сторонніх сервісів (Facebook, Google, і т.п.);
д) особистий кабінет;
е) система замовлень.

Найважливіші модулі додатку, як-от обробка платежів, мають мати автоматизовані тести, а весь додаток – ретельно протестований вручну.

2 ВИБІР ТЕХНОЛОГІЙ ДЛЯ РЕАЛІЗАЦІЇ ПРОЄКТУ

2.1 Мова програмування

Для розробки вебзастосунку мовою програмування було обрано TypeScript. Це надбудова над мовою JavaScript, розроблена компанією Microsoft, що дозволяє створювати складні вебзастосунки, які легше зберігати та розвивати завдяки можливості використання типів даних, інтерфейсів та інших інструментів. TypeScript не є інтерпритованою мовою, що значить, що програми написані на ній не виконуються напряму, а спочатку транспілюються у Javascript.

JavaScript традиційно використовується на стороні клієнта у браузері. Проте, за допомогою Node.js – серверного оточення для JavaScript, він використовується на стороні сервера. В екосистемі Node.js побудовано безліч бібліотек та фреймворків, що допоможуть нам в якості фундаменту додатку, і імплементацій конкретних модулів.

TypeScript є особливо корисним для великих проектів, де потрібно розробити багато коду та докладно протестувати його. Він допомагає зменшити кількість помилок на етапі розробки, що знижує час та кошти на виправлення помилок пізніше.

2.2 Фреймворк

Вибір фреймворку в екосистемі JavaScript – задача з безліччю альтернатив, що частіше за все зводиться до особистих вподобань. Та все ж, враховуючи архітектуру Інверсії Контролю, деякі фреймворки є більш прийнятними. Нижче наведено список розглянутих варіантів:

а) NuxtJS – це фреймворк, заснований на Vue.js, який спрощує створення універсальних і серверно-рендерених застосунків. Він пропонує модульну архітектуру, автоматичну оптимізацію коду та підтримку зручних плагінів. Попри це, через недостатню ознайомленість з Vue.js цей фреймворк було відкинуто;

б) Svelte – інноваційний фреймворк, який відрізняється від традиційних фреймворків тим, що він компілює компоненти і забезпечує низькі розміри бандлів та швидкість виконання. Основним недоліком фреймворку є відносно молода спільнота з меншими ресурсами та підтримкою, порівняно з іншими фреймворками, а також обмежений набір плагінів та інструментів, що можуть потребувати самостійної реалізації деяких рішень;

в) Remix – новий фреймворк, заснований на React, який фокусується на покращенні продуктивності розробки та швидкості вебзастосунків. Він надає гнучке API, оптимальні стратегії завантаження даних та підтримку серверного рендерення.

Будучи майже ідеальним варіантом, Remix має одне обмеження, що є критичним при проєктуванні з використанням Інверсії Контролю – відсутність підтримки Статичного Генерування (Static Site Generation або SSG). Адже не маючи прив’язки до джерела даних, та маючи модулі з різними джерелами, генерація сторінки під час запиту може ставати неприпустимо довгою.

Розглянувши ці варіанти, було прийнято рішення зупинитись на Next.js. Це один з найпопулярніших фреймворків, заснованих на React, що має велику спільноту, активну підтримку та розвиток, а також гнучкість у виборі допоміжних інструментів.

Основні переваги Next.js:

- вбудована підтримка серверної, статичної та гібридної моделей генерації сторінок;

– розділення серверної та клієнтської частини, що дозволяє оптимізувати та мінімізувати код, що передається на клієнтську сторону, таким чином прискорюючи продуктивність системи.

Недоліки Next.js:

– можлива складність у конфігурації, особливо для нових користувачів;
– незначне сповільнення розробки через додаткові абстракції та концепції;
– відносно важка інтеграція з компонентами сторонніх розробників, які не розроблені спеціально для Next.js.

Загалом, Next.js виявився оптимальним варіантом для проекту, оскільки він поєднує переваги React-екосистеми з продуктивністю та швидкістю, що надається серверним та статичним генеруванням сторінок.

2.3 Контейнер Інверсії Контролю

Ін’єкція Залежностей не є найпопулярнішою архітектурою в екосистемі JavaScript, та попри це існує декілька популярних бібліотек, що широко використовуються, таких як Awilix та InversifyJS. Проте, InversifyJS дуже сильно спирається на спеціальні анотації та нестандартні функції мови, як декоратори. А з Awilix я мав широкий досвід роботи, та більш детально знайомий з його недоліками, такими як:

– невідповідність типів до реальних значень залежностей, що призводить до компіляції коду, що насправді не працює;
– розширення контейнерів відбувається за принципом клонування, а не пов’язаної ієрархії. Це призводить до повторних інстаціювань та залишкових неутилізованих залежностей;
– заточеність під внутрішній імпорт файлів з модулями засобами самої бібліотеки, що в деяких випадках призводить до

неправильної компіляції та runtime-помилок, через використання нестандартних компіляторів (таких як SWC, що використовується в Next.js).

З огляду на ці недоліки, деякий час тому, поза контекстом цього проєкту, мною було розроблено мінімальну IoC-бібліотеку під назвою Holoscope. Приклад роботи з цією бібліотекою (рис. 2.1):

```
const scope = new Scope({
  currentTime: asFunction(() => Date.now()),
})

scope.container.currentTime // Завжди повертає поточний час
```

Рисунок 2.1 – Приклад створення контейнеру за допомогою Holoscope

На прикладі сервісу, що отримує дані з стороннього API (рис. 2.2):

```
class ExampleService {
  constructor(private container: { env: Environment }) {}

  public async getData() {
    const apiKey = this.container.env.get('API_KEY')
    const data = await fetchData('https://example.com', {
      headers: {
        apiKey,
      },
    })
    return data
  }
}

const scope = new Scope({
  env: asClass(Environment, { cached: true }),
  exampleService: asClass(ExampleService, { cached: true }),
})
```

Рисунок 2.2 – Приклад реалізації сервісу, що має залежність

Тут, для надсилання запиту необхідний ключ API. Завдяки Ін'єкції Залежностей:

а) залежність env буде інстанційована тільки тоді, коли вона потрібна іншій залежності (в цьому випадку в методі getEnv сервісу ExampleService);

б) всі залежності будуть інстанційовані лише 1 раз, завдяки параметру cached, що дозволяє зберегти ресурси;

в) залежність env може мати будь-яку реалізацію до тих пір поки вона відповідає інтерфейсу з публічним методом get. Наприклад, значення можуть бути отримані з параметрів оточення, або з значень, переданих при інстаціюванні (рис. 2.3).

```
class ProcessEnvironment {
  public get(key: string): string {
    return process.env[key]
  }
}

const scope = new Scope({
  env: asClass(ProcessEnvironment, { cached: true }),
  exampleService: asClass(ExampleService, { cached: true }),
})

// або:

class MemoryEnvironment {
  constructor(private values: Record<string, string>) {}
  public get(key: string): string {
    return this.values[key]
  }
}

const scope = new Scope({
  env: new MemoryEnvironment({ API_KEY: '123' }),
  exampleService: asClass(ExampleService, { cached: true }),
})
```

Рисунок 2.3 – Приклади реалізації взаємозамінних залежностей

Щоб не «збирати» контейнер у кожному місці застосуку, де він потрібен, імпортуючи всі модулі, використовується наслідування класів, та реєстрація залежностей (модулів) у конструкторі (рис. 2.4):

```
interface ExampleContainer {
  env: Environment
  exampleService: ExampleService
}

class ExampleScope extends Scope<ExampleContainer> {
  constructor() {
    super({
      env: asClass(Environment, { cached: true }),
      exampleService: new MemoryEnvironment({ API_KEY: '123' }),
    })
  }
}


// потім, в місці використання:

const scope = new ExampleScope()
const data = await scope.container.exampleService.getData()
```

Рисунок 2.4 – Практичний приклад створення контейнера залежностей

2.4 Допоміжні інструменти

2.4.1 TailwindCSS

Tailwind – це CSS фреймворк, що дозволяє приміняти до елементів стилі у вигляді окремих класів. Це прибирає необхідність в окремих файлах з таблицями стилей. Для прикладу, ідентичний код (рис. 2.5):

```
<!-- Стилі CSS -->
<style>
  .red {
    color: red;
  }
</style>
<div class="red">Hello</div>

<!-- Tailwind CSS -->
<div class="text-red-800">Hello</div>
```

Рисунок 2.5 – Порівняння CSS та Tailwind

2.4.2 Formik та Yup

Formik – це бібліотека для будування інтерактивних форм у React.JS. Yup – це бібліотека валідації даних. Використовується в основному для валідації форм та перевірки формату запитів сторонніх сервісів.

3 РОЗРОБКА ВЕБЗАСТОСУНКУ

Застосунок доступний за адресою https://coliiir.com

Репозиторій проєкту – https://github.com/nazarvovk/third-color

3.1 Проектування архітектури

В схематичному вигляді, основний функціонал системи можна зобразити наступним чином (рис. 3.1):

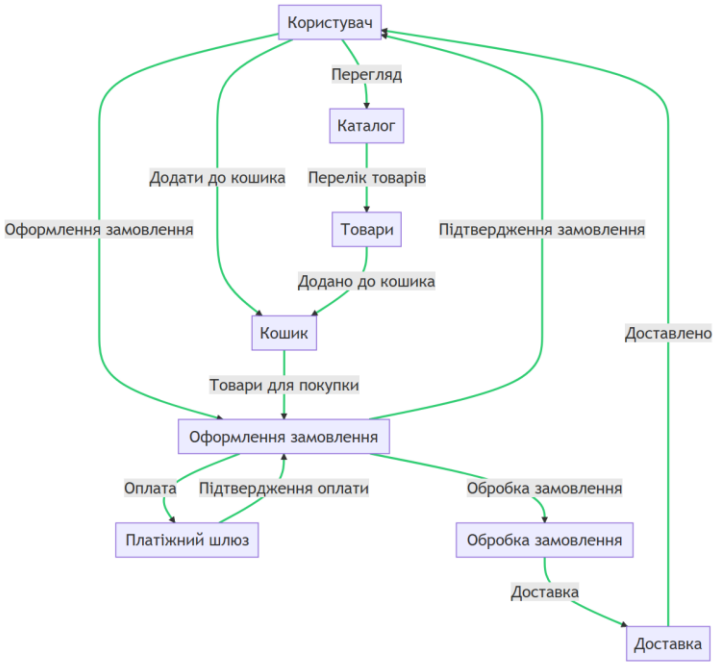


Рисунок 3.1 – Схематичне зображення циклу використання застосунку

Для реалізації цього функціоналу, маємо наступну програмну архітектуру застосунку (рис. 3.2):

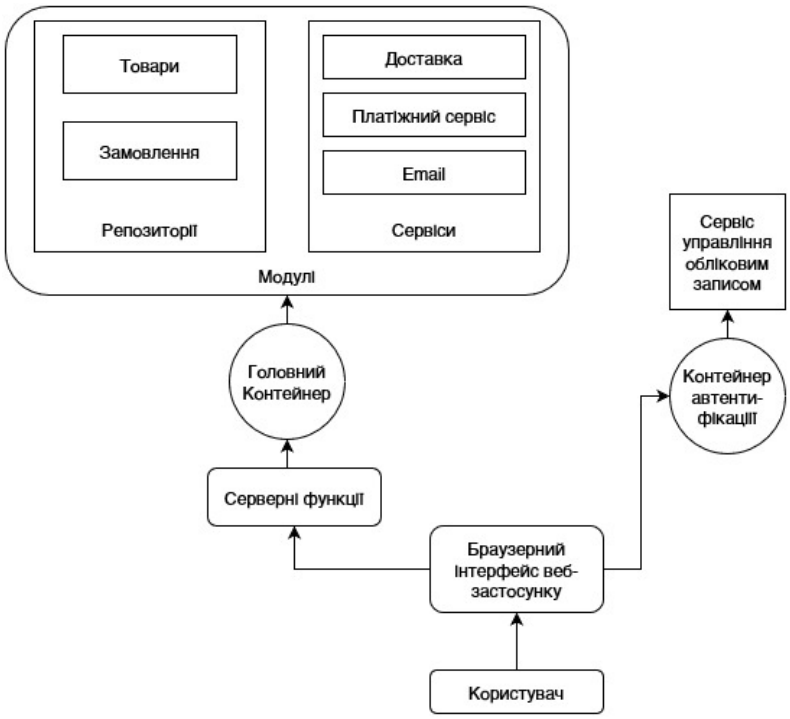


Рисунок 3.2 – Схематичне зображення програмної архітектури застосунку

Данна архітектура дозволяє розділити обов’язки модулів, відділити бізнес-логіку у серверні функції, та мінімізувати логіку на стороні клієнта. Як видно, частина логіки, а саме управління обліковим записом, виконується у бразері через окремий контейнер. Це є компромісом задля полегшення та пришвидшення роботи.

3.2 Дизайн застосунку

Перед початком розробки, за допомогою Figma було розроблено дизайн майбутнього вебсайту (рис. 3.3):

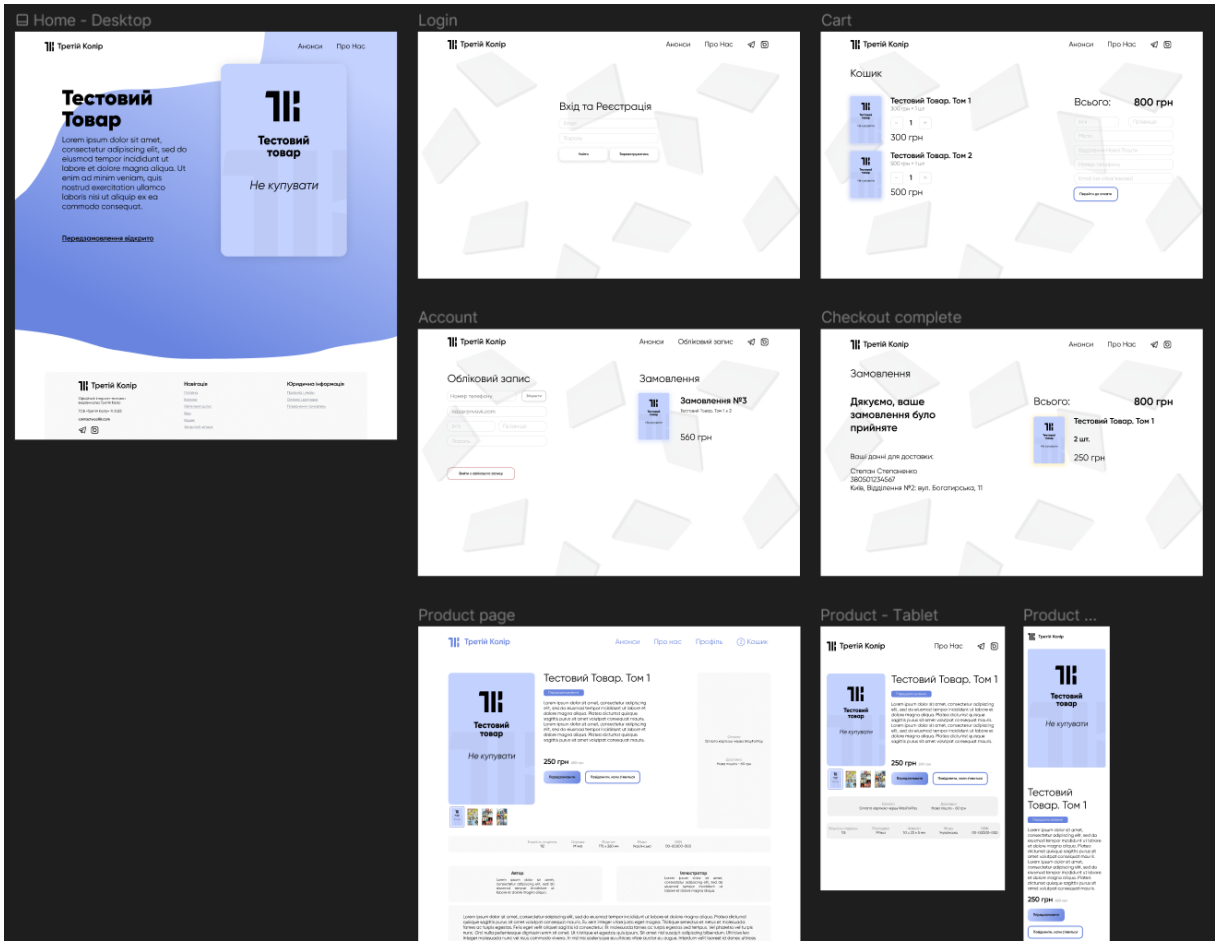


Рисунок 3.3 – Макет дизайну застосунку

3.3 Структура проєкту

Логічна структура проєкту прив’язана до файлової, тож можна почати ознайомлення з проєктом, оглянувши основні директорії і файли.

Файлова структура проєкту включає наступні ключові елементи:

– pages/ – директорія, що містить компоненти-сторінки, де ім'я файлу стає маршрутом вебдодатку (наприклад, index.ts стає кореневим маршрутом /);

– pages/api/ – піддиректорія в каталозі pages, призначений для серверних функцій API;

– public/ – директорія для статичних ресурсів, таких як зображення, файли CSS та JavaScript, які будуть доступні безпосередньо через URL;
– components/ – директорія, де зберігаються повторно використовувані компоненти, які можна імпортувати в сторінки та інші компоненти;
– lib/ – ключова директорія, в якій розташовані інтерфейси модулів, їх імплементації, а також IoC контейнери.

3.3.1 Директорія pages

Директорія pages містить наступну ієрархію сторінок (рис. 3.4):

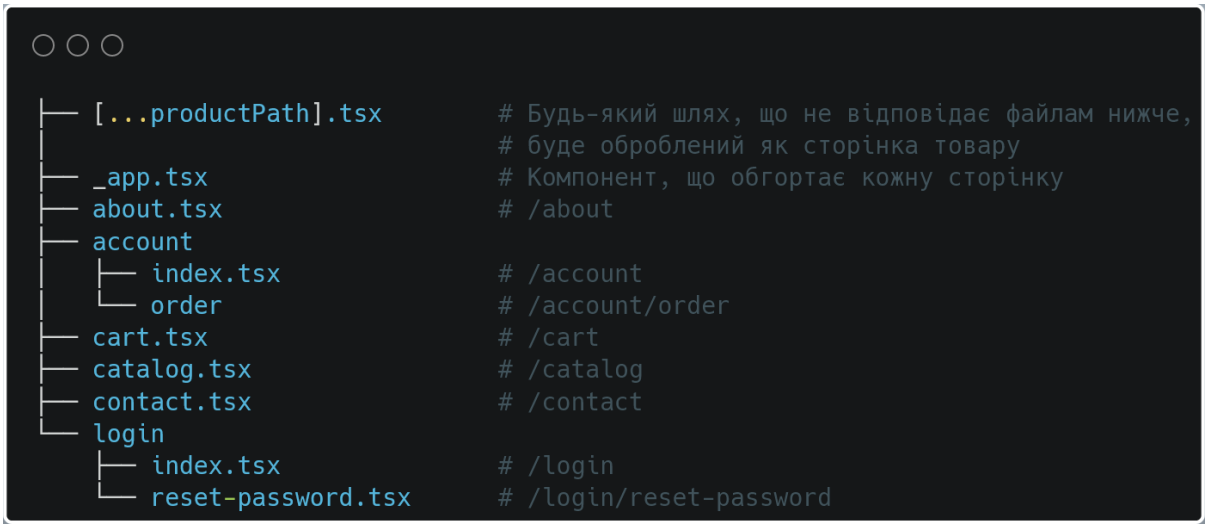


Рисунок 3.4 – Файли у директорії сторінок

Кожен файл сторінки може експортувати або метод getStaticProps, або getServerSideProps, що визначає як буде генеруватись сторінка – статично, або при кожному запиті користувача.

3.3.2 Директорія lib. Розділення на репозиторії та сервіси

Модулі, що використовуються в контейнері IoC, розподілені у дві категорії:

- репозиторії – зберігають данні, що є безпосередньою частиною застосунку. Наприклад:
  - репозиторій товарів;
  - репозиторій замовлень.
- сервіси – взаємодіють із сторонніми сервісами, виконують операції, що напряму не пов’язані зі збереженням даних. Наприклад:
  - сервіс опрацювання платежів – відповідальний за створення оплати, та опрацювання зміни стану оплати, як-от успішна оплата, скасування та повернення, чи помилки при оплаті;
  - сервіс доставки – отримання актуальних даних про можливі міста та відділення доставки.

Деякі модулі, що не підходять для жодної з категорій, віднесені до загальної директорії utils.

3.3.3 Директорія lib. Контейнери

Директорія контейнерів має наступний вигляд (рис. 3.5):

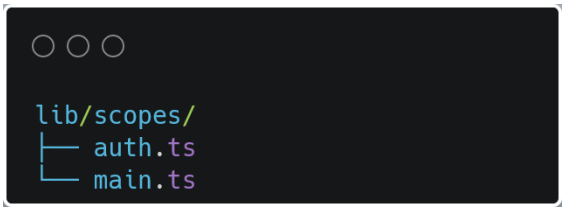


Рисунок 3.5 – Файли контейнерів застосунку (див. рис. 3.2)

У файлі main.ts визначено основний контейнер, що об’єднує модулі, які використовуються на стороні серверу при генерації статичних сторінок, генерації даних під час запиту сторінки, чи шляхів API.

Файл auth.ts визначає контейнер, що використовується на стороні браузера, і містить сервіс, що відповідає за авторизацію, зміну користувача, та допоміжні методи. Окремий контейнер є необхідним, адже якщо використовувати основний MainScope, то в браузерний код буде додано багато невикористовуваних модулів, що критично збільшить розмір клієнтських файлів та уповільнить роботу застосунку.

3.3.4 Використання IoC Контейнеру на прикладі сторінки товару

Поглянемо детальніше на метод, експортований з сторінки товару […productPath].tsx, що вибирає дані про товар з репозиторію, генерує допоміжні дані, як-от стилі, та передає їх для генерації сторінки (рис. 3.6).

MainScope – це основний контейнер залежностей. В контексті цієї сторінки, для генерації необхідно отримати товар з репозиторію, згідно з шляхом товару. Таким чином шлях /test-product/1 відповідає товару з id «test-product.1». Репозиторій визначено інтерфейсом (рис. 3.7).

Всі модулі визначені таким чином, абстрагуючись від імплементацій. Врешті, в визначенні MainScope реєструються конкретні модулі, що імплементують визначений інтерфейсом контейнер (рис. 3.8).

```
export async function getStaticProps({ params }: { params: { productPath: string[] } }) {
  const { productPath } = params

  const productId = productPath.join('.')

  const scope = new MainScope()
  const { productRepository } = scope.container

  let product: Product
  try {
    product = await productRepository.getProductById(productId)
  } catch (error) {
    if (error instanceof Error && error.message.includes('No product')) {
      return { notFound: true }
    }
    throw error
  }

  const productDto = ProductMapper.toDto(product)

  const color = product.color ? new Color(product.color) : BLACK
  const saturatedColor = color.saturate()

  const props: ProductPageProps = {
    productDto,
    style: {
      primaryColor: color.hex,
      primaryBackground: getGradient(color, saturatedColor),
      accentColor: color.contrastBW.hex,
    },
  }

  await scope.dispose()

  return {
    props,
    revalidate: 60, // 1 minute
  }
}
```

Рисунок 3.6 – Функція, що реалізує отримання статичних параметрів, необхідних для відображення сторінки товару

```
import { Product } from '~/lib/entities/product'

export interface ProductRepository {
  listProducts(): Promise<Product[]>
  getProductById(productId: string): Promise<Product>
}
```

Рисунок 3.7 – Інтерфейс репозиторію товарів

```
export interface MainContainer extends ContainerWithEnv {
  productRepository: ProductRepository
  orderRepository: OrderRepository
  deliveryService: DeliveryService
  paymentService: PaymentService
}

export class MainScope extends Scope<MainContainer> {
  constructor() {
    super({
      productRepository: asClass(FirebaseAdminProductRepository, {
        cached: true,
      }),
      // ... інші модулі
    })
  }
}
```

Рисунок 3.8 – Головний контейнер інверсії контролю

FirebaseAdminProductRepository імплементує ProductRepository, а ззовні контейнера productRepository типізується як базовий інтерфейс (приклад на Рис. 11). Звернімо увагу, що методи повертаюсь інстанції класу Product, а це значить що імплементація репозиторію може зберігати дані у будь-якому форматі, і навіть за допомогою різних сервісів (баз даних, провайдерів сховищ, тощо).

3.4 Імплементація модулів

3.4.1 Вибір основної бази даних

Для основних репозиторіїв товарів та замовлень було обрано хмарний сервіс Firebase, який працює на базі Google Cloud. Основні переваги:

– NoSQL база даних Firestore, що підтримує гнучкі структури даних та швидке масштабування;

– Firebase Storage – сховище файлів, яке ми можемо використовувати для хостингу зображень;

– сервіс автентифікації, що спрощує управління обліковими записами;

– наявність зручного вебзастосунку для адміністраторів бази даних, а значить немає необхідності у розробці додаткового вебінтерфейсу для працівників видавництва.

3.4.2 Імплементація репозиторію товарів

Імплементація репозиторію товарів використовує Firestore для збереження даних про товар, та Firebase Storage для збереження зображень (рис. 3.9). Для цього репозиторій очікує, що контейнер залежностей, переданий при інстаціюванні має інстанції відповідних SDK.

При цьому, інверсія контролю дозволяє параллельно мати інші імплементації, як-от FilesystemProductRepository, що зчитує товари з локального диску (рис. 3.10).

Це дозволяє реєструвати цей репозиторій замість основного в MainScope у development середовищі, що значно полегшує розробку додатку (рис. 3.11).

```
interface FirebaseProductRepositoryContainer {
  firestore: Firestore
  firebaseStorage: FirebaseStorage
}

export class FirebaseProductRepository implements ProductRepository {
  constructor(protected readonly container: FirebaseProductRepositoryContainer) {}

  public async listProducts(): Promise<Product[]> {
    const products = await getDocsFromServer(this.productsCollection)
    const dtos = products.docs.map((doc) => doc.data())

    return Promise.all(
      dtos.map(async (dto) => {
        const images = await this.getProductImageUrls(dto.id)
        return ProductMapper.fromDto({ ...dto, images })
      }),
    )
  }

  public async getProductById(productId: string): Promise<Product> {
    const productDoc = doc(this.productsCollection, productId)
    const [productSnapshot, images] = await Promise.all([
      getDocFromServer(productDoc),
      this.getProductImageUrls(productId),
    ])
    if (!productSnapshot.exists()) {
      throw new Error(`No product with id "${productId}" found`)
    }
    const dto = productSnapshot.data()
    return ProductMapper.fromDto({ ...dto, images })
  }

  // ... приватні методи productsCollection, getProductImageUrls, productConverter
}
```

Рисунок 3.9 – Імплементація репозиторію товарів, що зберігає товари у базі даних Firebase Firestore

```
export class FilesystemProductRepository implements ProductRepository {
  public async listProducts(): Promise<Product[]> {
    return this.readProducts()
  }

  public async getProductById(productId: string): Promise<Product> {
    const products = await this.readProducts()
    const product = products.find((product) => product.id === productId)
    if (!product) {
      throw new Error(`No product with id "${productId}" found`)
    }
    return product
  }

  private async readProducts(): Promise<Product[]> {
    const jsonData = await fs.readFile(this.filePath)
    const objectData: { products: ProductDTO[] } = JSON.parse(jsonData.toString())
    return Promise.all(
      objectData.products
        .map((product) => ProductMapper.fromDto(product))
        .map(async (product) => {
          product.images = await this.getProductImageUrls(product.id)
          return product
        }),
    )
  }
}
```

Рисунок 3.10 – Імплементація репозиторію товарів, що зберігає товари у локальному json-файлі

```
export class MainScope extends Scope<MainContainer> {
  constructor() {
    // ... попередні реєстрації модулів

    if (process.env.NODE_ENV === 'development') {
      this.register({
        productRepository: asClass(FilesystemProductRepository, {
          cached: true,
        }),
      })
    }
  }
}
```

Рисунок 3.11 – Підстановка репозиторію в середовищі розробки

За цим приципом так само реалізовано і репозиторій замовлень, з таким базовим інтерфейсом (рис. 3.12).

```
import type { Order, OrderDTO } from '~/lib/entities/order'
import type { User } from '~/lib/entities/user'

export type CreateOrderProperties = Omit<OrderDTO, 'id' | 'createdAt'>

export interface OrderRepository {
  createOrder(properties: CreateOrderProperties): Promise<Order>
  updateOrder(order: Order): Promise<void>
  getOrderById(orderId: string): Promise<Order>
  getOrdersByUser(user: User, limit: number): Promise<Order[]>
}
```

Рисунок 3.12 – Інтерфейс репозиторію замовлень

Звернімо увагу, що окрім класу Order використовується інтерфейс OrderDTO. Це – патерн Data Transfer Object, що використовує прості об’єкти для перенесення сутностей (товар, замовлення, користувач) між процесами. Наприклад, використовується при передачі даних між сервером та клієнтом. В данному випадку часткове DTO використовується для створення замовлення, коли не всі данні наявні. Це зроблено для того, щоб імплементація мала контроль над нумерацією замовлень. Так,

FirebaseOrderRepository створює порядкові номери, а локальний FilesystemOrderRepository використовує генерацію UUID.

3.5 Загальний контекст застосунку

В директорії pages визначено файл _app.tsx. Це спеціальний React-компонент, що «обгортає» кожну сторінку додатку. Відповідно, в ньому розташовується логіка, яка присутня на кожній сторінці (рис. 3.13).

```tsx
import '../styles/globals.css'
import type { AppProps } from 'next/app'
import Head from 'next/head'
import useSystemTheme from 'use-system-theme'
import { AuthProvider } from '~/lib/utils/use-auth'
import { CartProvider } from '~/lib/utils/cart'
import { Background } from '~/components/background'
import { LoadingIndicator } from '~/components/loading-indicator'

function MyApp({ Component, pageProps }: AppProps) {
  // change favicon to dark one if the user prefers dark mode
  const theme = useSystemTheme()
  const favicon = theme === 'dark' ? '/favicon-dark.png' : '/favicon.png'

  return (
    <CartProvider initialCart={pageProps.cartDto}>
      <AuthProvider initialUserDto={pageProps.user}>
        <div className='overflow-hidden min-h-screen'>
          <Head>
            <title>Третій Колір - Видавництво коміксів, мальописів і не тільки</title>
            <meta name='description' content='Видавництво коміксів, мальописів і не тільки' />
            <link rel='icon' href={favicon} type='image/png' />
          </Head>
          <LoadingIndicator />
          <Component {...pageProps} />
          {pageProps.withBackground && <Background />}
        </div>
      </AuthProvider>
    </CartProvider>
  )
}

export default MyApp
```

Рисунок 3.13 – Загальний контекст застосунку

Основні функції цього компоненту:

– підключення глобальних стилів та елементів дизайну;

– рендер компоненту індикації завантаження;

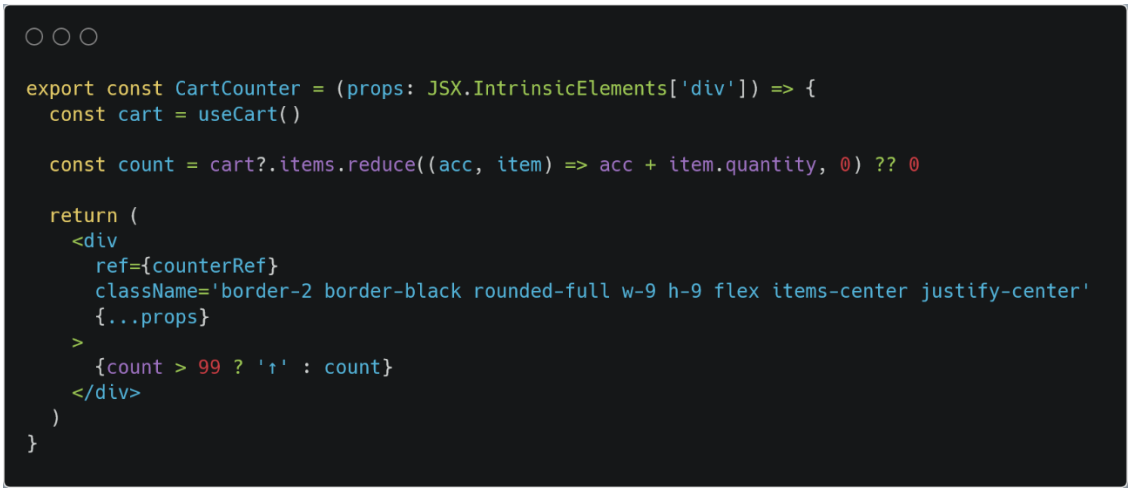
– провайдери контексту для кошику та облікового запису.

Зупинимося детальніше на останньому.

3.5.1 Кошик

Компонент CartProvider надає доступ до контексту з поточним станом кошика, а також методи маніпуляції ним. Стан кошика зберігається у cookie, таким чином він автоматично надсилається з кожним запитом, наприклад запитом сторінки /cart, чи API-шлях оформлення замовлення.

Будь-який компонент в застосунку може використовувати контекст кошику, наприклад лічильник у Header (рис. 3.14).



```
export const CartCounter = (props: JSX.IntrinsicElements['div']) => {
  const cart = useCart()

  const count = cart?.items.reduce((acc, item) => acc + item.quantity, 0) ?? 0

  return (
    <div
      ref={counterRef}
      className='border-2 border-black rounded-full w-9 h-9 flex items-center justify-center'
      {...props}
    >
      {count > 99 ? '↑' : count}
    </div>
  )
}
```

Рисунок 4.14 – Використання контексту кошика на прикладі лічильника товарів

3.5.2 Провайдер автентифікації

AuthProvider – компонент, що відповідає за управління станом користувача (рис. 3.15).

```tsx
export const AUTH_UNINITIALIZED = Symbol.for('user_uninitialized')
type UserState = User | null | typeof AUTH_UNINITIALIZED

type AuthContext = {
  user: UserState
  container: AuthContainer
}
const AuthContext = createContext<AuthContext>()


export const getAuthTokenCookie = (req: GetServerSidePropsContext['req']) => {
  const value = getCookie(AUTH_TOKEN_COOKIE, { req })?.toString()
  return typeof value === 'string' ? value : undefined
}

export const AuthProvider = (props: AuthProviderProps) => {
  const { asPath: currentPath } = useRouter()
  const { children, initialUserDto } = props
  const [user, setUser] = useState<UserState>(() => {
    if (initialUserDto) {
      const { id, ...userProps } = initialUserDto
      return new User(userProps, id)
    }
    return AUTH_UNINITIALIZED
  })

  const scope = useMemo(() => new AuthScope(), [])
  const { container } = scope
  const { authService } = container

  useEffect(() => {
    // return to unsubscribe on unmount
    return authService.onAuthStateChanged((user) => {
      // ... збереження токену
      return setUser(user)
    })
  }, [authService])

  useEffect(() => {
    // Save current path to redirect to after login
    if (!currentPath.startsWith('/login')) {
      setCookie(LOGIN_REDIRECT_COOKIE, currentPath, { sameSite: 'lax' })
    }
  }, [currentPath])

  return (
    <AuthContext.Provider
      value={{
        user,
        container,
      }}
    >
      {children}
    </AuthContext.Provider>
  )
}

export const useAuth = () => useContext(AuthContext)
```

Рисунок 3.15 – Провайдер автентифікації

Тут створюється стан користувача та IoC контейнер AuthScope через який відбувається взаємодія з сервісом автентифікації. useEffect використовується для підписки на зміни стану авторизації та оновлення користувача та токенів авторизації. Таким чином будь-який компонент в застосунку може змінити стан автентифікації, і це відобразиться у всіх

інших компонентах. Також він зберігає поточний шлях, щоб повернути користувача на попередню сторінку після входу.

3.6 Сторінка оформлення замовлення

На відміну від більшості інтернет-магазинів, де сторінка кошику та оформлення замовлення – окремі, було прийняте рішення зробити цей функціонал одною сторінкою, що пришвидшує процес замовлення для кінцевого користувача. Ця сторінка (Рис. 3.16) – найбільш складна та комплексна, адже має логіку валідації значень, динамічні елементи, що змінюються в залежності від контексту, та завантаження міст та відділень Нової Пошти.

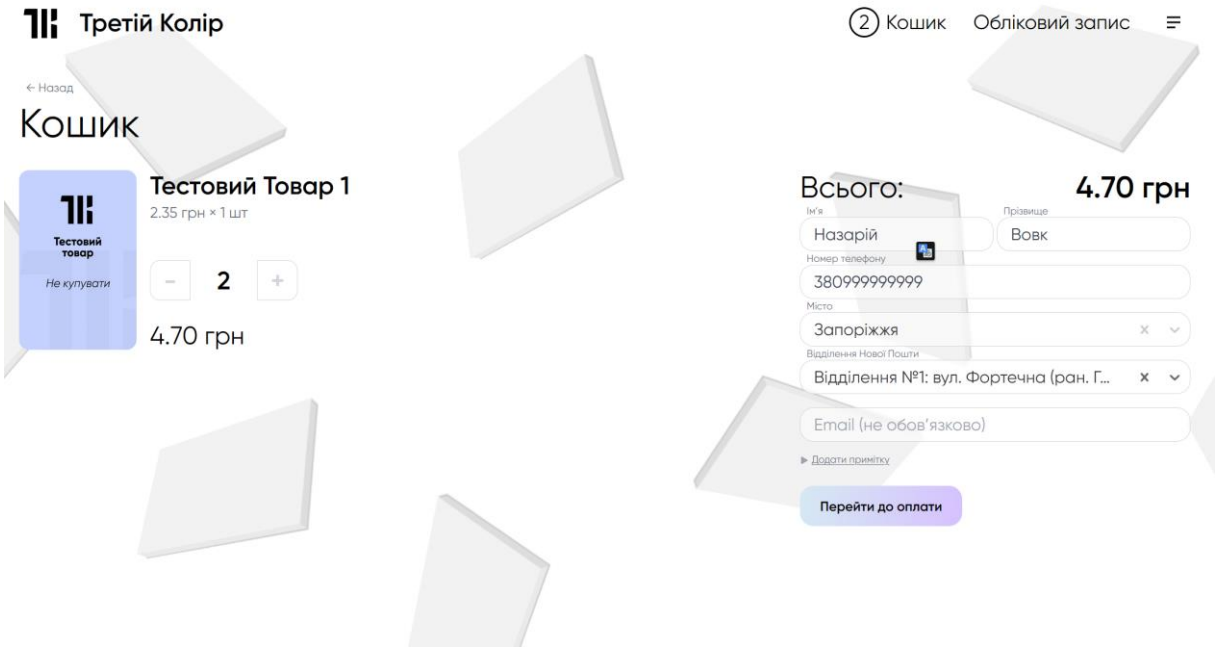


Рисунок 3.16 – Сторінка додатку «Кошик»

3.6.1 Динамічний пошук доставки

Поглянемо детальніше на останнє, на прикладі API-шляху, що дозволяє динамічно завантажувати список міст в залежності від пошуку користувача (рис. 3.17).

```
const requestBodySchema = object().shape({
  query: string().defined().nonNullable(),
})
export default async function handler(req: NextApiRequest, res: NextApiResponse) {
  const { query } = requestBodySchema.validateSync(req.body)
  const scope = new MainScope()
  const { deliveryService } = scope.container
  const dtos = await deliveryService.getCitiesByQuery(query)
  const cities = dtos.map((city) => ({
    value: city.Ref,
    label: city.Description,
  }))
  const responseBody = {
    cities,
  }
  res.status(200).json(responseBody)
  await scope.dispose()
}
```

Рисунок 3.17 – API-шлях для пошуку міст

Тут, знову ж таки, використовується основний контейнер IoC та модуль сервісу доставки. API-шлях запитує міста, що відповідають пошуку, формує опції в мінімальному вигляді задля економії та пришвидшення передачі даних, та повертає їх клієнту.

3.6.2 Генерація замовлення та оплати

За оформлення замовлення відповідає API-шлях checkout (рис. 3.18).

На прикладі CheckoutFormValidator можна побачити, що модуль не обов’язково має бути зареєстрований на контейнері, щоб використовувати його. Так, можливе і ручне створення, з передачею будь-якого об’єкту, що відповідає інтерфейсу необхідного модулю контейнеру (хоч у данному

прикладі передається головний контейнер, задля запобігання повторення коду).

```
export default async function handler(req: NextApiRequest, res: NextApiResponse) {
  const scope = new MainScope()
  // ...
  const validator = new CheckoutFormValidator(scope.container)
  let values: CheckoutValues = await validator.validate(req.body)
  // validate that every item has a valid quantity
  for (const item of cart.items) {
    if (item.quantity < 1) {
      throw new Error(`Invalid quantity ${item.quantity} for item ${item.id}`)
    }
  }
  // this also ensures that products exist in the repository
  const items = await Promise.all(
    cart.items.map(async (item) => {
      const product = await productRepository.getProductById(item.id)
      return {
        name: product.name,
        id: product.id,
        price: product.price,
        quantity: item.quantity,
      }
    }),
  )

  const order = await orderRepository.createOrder({
    ...values,
    items,
    paymentStatus: 'pending',
    deliveryStatus: 'awaiting_payment',
  })
  // ...
}
```

Рисунок 3.18 – Реалізація API-шляху оформлення замовлення

Перед тим як створити замовлення і оплату, проводиться декілька перевірок:

– всі данні форми заповнені і вірні;
– місто та відділення доставки доступні та відповідають одне одному;
– кошик – вірно сформований. Товари в кошику доступні для замовлення та мають вірну кількість.
– актуальні ціни на товари, а також інші дані, вибираються з бази даних.

Всі ці перевірки захищають від можливої експлуатації системи, адже так як система буде мати публічний доступ – будь хто з належними знаннями може спробувати атакувати систему некоректно сформованими даними запитів.

3.7 Платіжний модуль

При розробці сервісу оплати було розглянуто декілька платіжних шлюзів через які може прийматись оплата, таких як Fondy, LiqPay та WayForPay. Вибір було зроблено на користь останнього, і відповідно розроблено імплементацію за наступним інтерфейсом (рис. 3.19).

```
export interface PaymentService {
  createPaymentURL: (payment: CreatePaymentOptions) => Promise<string>
  getPaymentFromServiceRequest: (requestBody: unknown) => Promise<HandleServiceRequestReturn>
}
```

Рисунок 3.19 – Інтерфейс платіжного модуля

Реалізуються два методи. Перший – створює оплату замовлення та повертає посилання на яке потрібно перенаправити користувача. Другий метод – опрацьовує оновлення статусу оплати, приймаючи запити від платіжного шлюзу.

Проте, невдовзі після розробки модуля, що використовує WayForPay, 02.05.2023 платіжний шлюз припинив прийом платежів через припинення ліцензії, виданої Національним Банком України. Ця ситуація є саме тим, де інверсія контролю проявляє себе найкраще.

Вимушено, було розроблено модуль оплати, що використовує LiqPay. Завдяки архітектурі IoC код, написаний до цього, не було змінено, окрім підстановки LiqpayPaymentService в головному контейнері. Таким чином, якщо в майбутньому WayForPay відновить свою роботу, ми

зможемо повернутись до нього, не змінюючи коду, що використовує платіжний модуль. А інверсія контролю дозволяє мінімізувати негативні наслідки від ситуацій, які власник системи безпосередньо не контролює.

3.8 Ін’єкція залежностей у автоматизованому тестуванні

Зважаючи на розмір та цілі застосунку, написання юніт-тестів для всіх модулів не є доцільним. Проте, для деяких аспектів, як-от оформлення замовлень чи обробка платежів, це є необхідним. Для юніт-тестів використовується бібліотека Jest. Розглянемо тести на прикладі API-шляху, що оновлює дані оплати після оформлення замовлення (рис. 3.20).

```
describe(UpdateOrderPaymentUseCase.name, () => {
  const orderRepository = new OrderRepositoryMock()
  const paymentService = new PaymentServiceMock()
  const container = {
    orderRepository,
    paymentService,
  }
  const useCase = new UpdateOrderPaymentUseCase(container)
  // ...
  it('should update order delivery status to "ready_to_ship" if payment is paid', async () => {
    mockOrderStatusesInRepo('awaiting_payment', 'pending')
    paymentService.getPaymentFromServiceRequest.mockResolvedValueOnce({
      payment: {
        id: TEST_ORDER_ID,
        status: 'paid',
      },
      responseBody: { message: 'success' },
    })

    const response = await useCase.execute(req)

    expect(paymentService.getPaymentFromServiceRequest).toHaveBeenCalledWith(req.body)
    const updateCall = orderRepository.updateOrder.mock.calls[0]
    expect(updateCall).toMatchSnapshot()
    const order: Order = updateCall[0]
    expect(order.deliveryStatus).toBe('ready_to_ship')
    expect(order.paymentStatus).toBe('paid')
    expect(response).toStrictEqual({ message: 'success' })
  })
  // ...
})
```

Рисунок 3.20 – Приклад юніт-тестування

В тестах розкривається одна з найбільших переваг Інверсії Контролю і Ін’єкції Залежностей – підстановка одних модулів для тестування іншого. Тут, OrderRepositoryMock та PaymentServiceMock – це модулі з

підставними методами, що дозволяють відстежувати коли і з якими аргументами їх було викликано. При цьому контейнеру не обов'язково бути інстанцією Scope, адже достатньо щоб тип збігався з типом аргумента. Все це значно спрощує логіку тестування.

3.9 Зачистка контейнеру і модулів

Ще одна перевага Ін'єкції Залежностей – централізований контроль життєвого циклу модулів. Без DI, якщо в кінці життєвого циклу потрібно виконати якість дії (як розповсюджений приклад – закрити підключення до бази даних), то це потрібно робити вручну для кожного модуля. З DI же, якщо модуль зареєстровано у IoC контейнері, то є можливість додати метод disposer. На прикладі Firebase (Рис. 3.21):

```
this.register({
  // ...
  firebase: asFunction(firebaseFactory, {
    cached: true,
    disposer: (firebase: FirebaseApp) => deleteApp(firebase),
  }),
})
```

Рисунок 3.21 – Залежність із визначеним disposer

І тоді, в кінці життєвого циклу **контейнера,** викликається метод dispose(), що зачищає всі зареєстровані модулі, та очищує створені інстанції. Приклади – на рисунках 3.6, 3.17.

3.10 Тестування застосунку

В ході розробки було написано 59 автоматизованих тестів, для 16 модулів (рис. 3.22).

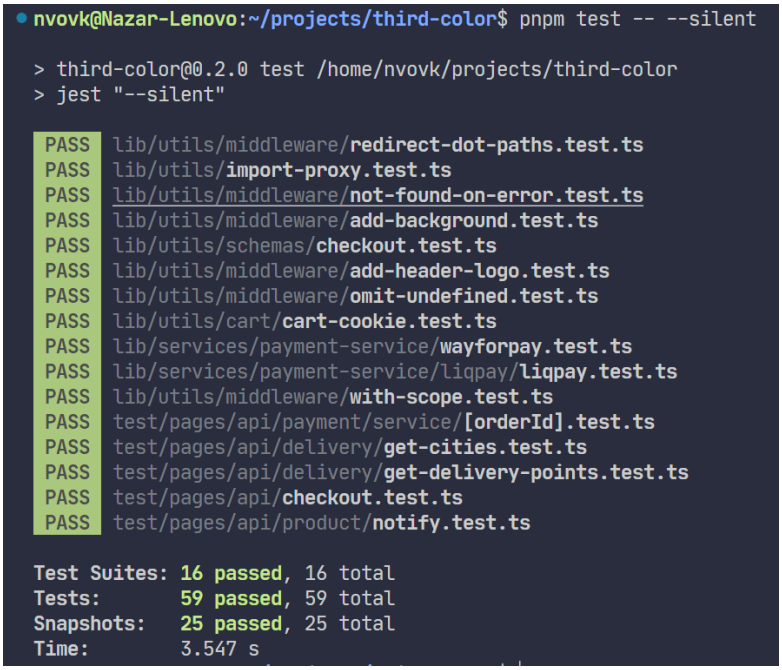


Рисунок 3.22 – Результати юніт-тестування

Також, проведене ретельне тестування вручну, та за допомогою таких інструментів як Google Chrome Lighthouse, що показує 100% за шкалою швидкодії (рис. 3.23).

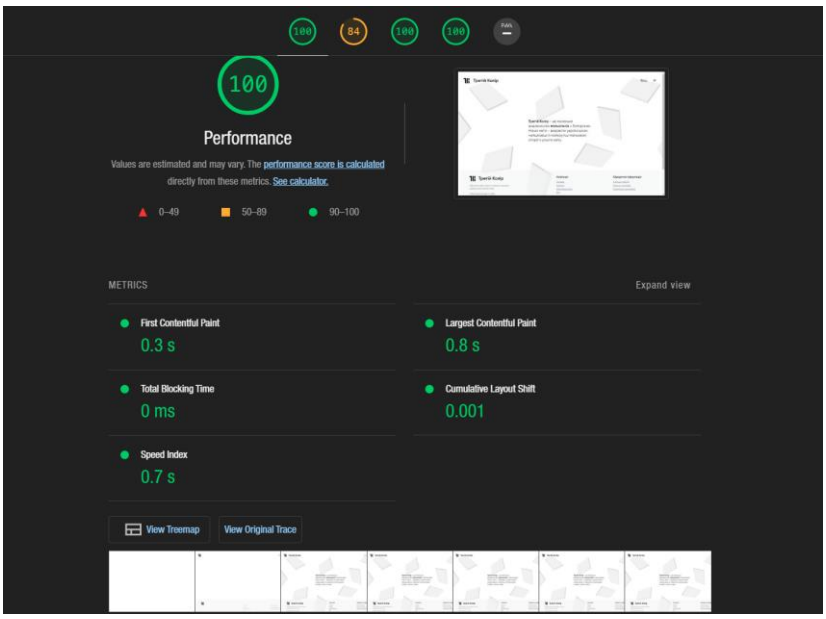


Рисунок 3.23 – Результат Google Chrome Lighthouse

3.11 Налаштування хостингу проєкту, та сервісів-залежностей.

У якості хостингу обрано Vercel, який має відмінну підтримку фреймворку Next.js. Було проведене налаштування сервісів Firebase, Нової Пошти, smtp-сервера провайдера email, а також платіжної системи. Задля майбутнього спрощення налаштування проєкту, всі необхідні данні, які необхідно визначити через змінні оточення викладено у файлі README.

Хоч фреймворк Next.js і належить компанії Vercel, він не обов’язково має бути запущений на їх хостингу. За необхідності змінити платформу, застосунок запускається наступною командою:

```
pnpm i && pnpm build && pnpm start
```

ВИСНОВКИ

В ході роботи було розроблено мультифункціональний вебсайт та інтернет-магазин. Використано сучасні технології, що дозволило забезпечити високу швидкодію та безпеку. Імплементовано архітектуру інверсії контролю, що дозволяє забезпечити масштабованість та розширюваність проєкту, високу тестовість та низьку залежність від конкретних технологій. Розроблено альтернативні імплементації модулів, що використовують різні технології, що дозволяє змінювати технології, не змінюючи бізнес-логіки. Виконано та імплементовано сучасний дизайн застосунку, з використанням анімацій, адаптивного дизайну, та загальних принципів проєктування користувацього досвіду (UX, англ. – User Experience).

Можна стверджувати, що поставлені цілі були досягнуті, а проєкт виконаний успішно. За адресою https://coliiir.com/ можна ознайомитись з результатом роботи.

ПЕРЕЛІК ПОСИЛАНЬ

1. Eric Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional. 2003.

2. Martin Fowler, with Dave Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, and Randy Stafford. Patterns of Enterprise Application Architecture. Addison-Wesley Professional. 2002.

3. Khalil Stemmler. Dependency Injection & Inversion Explained | Node.js w/ TypeScript. URL : https://khalilstemmler.com/articles/tutorials/dependency-injection-inversion-explained/ (дата звернення: 19.04.2023).

4. Market share of leading e-commerce software platforms and technologies worldwide as of March 2023. URL : https://www.statista.com/statistics/710207/worldwide-ecommerce-platforms-market-share/ (дата звернення: 19.04.2023).

5. Dependency Injection in Node.js – 2016 edition. URL : https://medium.com/@Jeffijoe/dependency-injection-in-node-js-2016-edition-f2a88efdd427 (дата звернення: 19.04.2023).

6. Репозиторій Awilix на платформі GitHub. URL : https://github.com/jeffijoe/awilix (дата звернення: 19.04.2023).

7. Formik Docs. URL : https://formik.org/docs/overview (дата звернення: 19.04.2023).

8. Firebase Documentation. URL : https://firebase.google.com/docs/web (дата звернення: 19.04.2023).

9. Пакет holoscope в репозиторії npm. URL : https://www.npmjs.com/package/holoscope (дата звернення: 19.04.2023).

10. Jest Delightful JavaScript Testing URL : https://jestjs.io (дата звернення: 19.04.2023).

11. Next.js Documentation. URL : https://nextjs.org/docs (дата звернення: 19.04.2023).

12. React.js Documentation. URL : https://reactjs.org/docs (дата звернення: 19.04.2023).

13. TailwindCSS. URL : https://tailwindcss.com/ (дата звернення: 19.04.2023).

14. The truth behind Inversion of Control – Part II – Inversion of Control. URL : https://www.sebaslab.com/the-truth-behind-inversion-of-control-part-ii-inversion-of-control/ (дата звернення: 19.04.2023).