МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Сумський державний університет

Факультет електроніки та інформаційних технологій

Кафедра комп'ютерних наук

«До захисту допущено»

В.о. завідувача кафедри

Ігор ШЕЛЕХОВ

(підпис)

01 червня 2024 р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня бакалавра

зі спеціальності 122 – Комп'ютерні науки,

освітньо-професійної програми «Інформатика»

на тему: «Адаптивний інтернет-магазин з доставки їжі»

здобувача групи ІН – 03 Шевченка Олексія Сергійовича

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

Олексій ШЕВЧЕНКО

(підпис)

Керівник,

асистент кафедри комп'ютерних наук

кандидат фізико-математичних наук Ольга ШУТИЛЄВА

(підпис)

Суми – 2024

Сумський державний університет
Факультет електроніки та інформаційних технологій
Кафедра комп'ютерних наук

«Затверджую»
В.о. завідувача кафедри
_____ Ігор ШЕЛЕХОВ
(підпис)

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня бакалавра

зі спеціальності 122 – Комп'ютерні науки, освітньо-професійної програми «Інформатика»
здобувача групи ІН-03 Шевченка Олексія Сергійовича

1. Тема роботи: «Адаптивний інтернет-магазин з доставки їжі» затверджена наказом по СумДУ від *«22» квітня 2024 р. № 0414-VI*
2. Термін здачі здобувачем кваліфікаційної роботи *до 01 червня 2024 року*
3. Вхідні дані до кваліфікаційної роботи
4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити)
1) Аналіз проблеми та актуальності розробки інтернет-магазину, аналогів та цільової аудиторії, постановка й формування завдань дослідження. 2) Огляд та вибір програмних засобів. 3) Проєктування та розроблення адаптивного інтернет-магазину з доставки їжі. 4) Аналіз отриманих результатів.
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)_____
6. Консультанти до проєкту (роботи), із значенням розділів проєкту, що стосується їх

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання «08» квітня 2024 р.

Керівник

Завдання прийняв до виконання _____ _____
(підпис) (підпис)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Термін виконання	Примітка
1	*Аналіз проблеми та актуальності розробки інтернет-магазину, аналогів та цільової аудиторії, постановка й формування завдань дослідження.*	06.05.2024–08.05.2024	
2	*Огляд та вибір програмних засобів.*	09.05.2024–10.05.2024	
3	*Проєктування та розроблення адаптивного інтернет-магазину з доставки їжі.*	11.05.2024–14.05.2024	
4	*Аналіз отриманих результатів.*	15.05.2024–16.05.2024	
5	*Оформлення пояснювальної записки до кваліфікаційної роботи.*	17.05.2024–20.05.2024	

Здобувач вищої освіти _____ Керівник _____
(підпис) (підпис)

АНОТАЦІЯ

Записка: 99 сторінок., 68 рисунків, 8 таблиць, 3 додатки, 19 використаних джерел.

Обґрунтування актуальності теми роботи – тема кваліфікаційної роботи є дуже актуальною в наші часи тому, що індустрія доставки їжі переживає бурхливе зростання. Споживачі все частіше шукають способи насолоджуватися ресторанними стравами вдома або на роботі. Зі зростанням залежності від онлайн-платформ для повсякденних потреб, значення адаптивних вебсайтів важко переоцінити. Крім того, у світлі останніх подій, що підкреслюють важливість віддалених послуг, актуальність інтернет-магазинів для доставки їжі стала ще більш вираженою.

Об'єкт дослідження – інтернет-магазини з доставки їжі.

Предмет дослідження – методи розробки та проєктування вебсервісів доставки їжі.

Мета роботи – розробка адаптивної інтернет-платформи з доставки їжі, що надає інноваційний та ефективний користувацький досвід, пропонуючи зручність, доступність та персоналізацію загальних вподобань.

Методи дослідження – аналіз та застосування різноманітних інструментів, які сприяють полегшенню розробки вебсервісів, а саме NestJS, ReactJS, MongoDB, Redux, TypeScript, Axios.

Результати – розроблено інтернет-магазин з доставки їжі, який надає користувача ефективний користувацький досвід, пропонуючи користувачам широкий вибір ресторанів, систему відгуків, персоналізувати вебсайт під свої вподобання, робити замовлення та надає змогу адміністраторам ефективно керувати вмістом вебсайту використовуючи панель керування.

ІНТЕРНЕТ-МАГАЗИН, ДОСТАВКА ЇЖІ, АДАПТИВНИЙ ДИЗАЙН,
REACTJS, NESTJS, MONGODB, TYPESCRIPT, REDUX, AXIOS

ЗМІСТ

ВСТУП

Актуальність. Тема кваліфікаційної роботи є дуже актуальною в наші часи тому, що індустрія доставки їжі переживає бурхливе зростання. Споживачі все частіше шукають способи насолоджуватися ресторанними стравами вдома або на роботі. Зі зростанням залежності від онлайн-платформ для повсякденних потреб, значення адаптивних вебсайтів важко переоцінити. Крім того, у світлі останніх подій, що підкреслюють важливість віддалених послуг, актуальність інтернет-магазинів для доставки їжі стала ще більш вираженою.

Об'єкт дослідження. Інтернет-магазини з доставки їжі.

Предмет дослідження. Методи розробки та проєктування вебсервісів, які спеціалізуються на доставці їжі.

Гіпотеза. Принципи адаптивного дизайну та використання сучасних технологій призведе до розробки високоефективного та зручного для користувача інтерфейсу. Вебсайт буде динамічно адаптуватися до різних розмірів екрану користувача, забезпечуючи безперешкодний досвід. Ретельне тестування та аналіз результатів призведене до покращення задоволеності клієнтів, сприятиме успіху та конкурентоспроможності інтернет-магазину з доставки їжі на цифровому ринку.

Новизна. Розроблене програмне забезпечення дозволить не лише забезпечити адаптивність під різні пристрої та екрани, але також використовувати передові технології такі як TypeScript для забезпечення статичної типізації даних, що призведе до підвищення ефективності та надійності програмного рішення. Використання NestJS дозволить створити високоефективну серверну частину додатку, що забезпечить високу швидкість відповіді на запити користувача. Використання принципів об'єктно орієнтованого програмування дозволить легку та довгострокову підтримку інтернет-магазину.

Структура. Дана робота складається зі вступу, аналітичного огляду, постановки задачі, вибір методу розв'язання поставленої задачі, опису програмного забезпечення інформаційної системи, висновків, списку використаних джерел та додатків.

1 АНАЛІТИЧНИЙ ОГЛЯД

1.1 Актуальність проблеми

У сучасному ритмі життя, коли час є одним з найважливіших ресурсів, вебсайт для закладу з доставки їжі стає невід'ємною складовою успішного бізнесу. Він відкриває безмежні можливості для користувачів, які прагнуть насолодитися смачною їжею.

Вебсайт надає можливість ретельно обирати страви з широкого асортименту, враховуючи дієтичні обмеження та смакові вподобання. Вебсайт дає можливість відчути себе в ресторані, не покидаючи домів, офісів чи інших місць. Таким чином, вебсайт для закладу з доставки їжі перетворює звичайне харчування в високоефективний та комфортний процес. Він відкриває нові горизонти для гастрономічних насолод та сприяє ефективному розвитку сучасних бізнесів, задовольняючи потреби сучасного споживача.

Згідно зі звітом Grand View Research, у 2022 році обсяг світового ринку онлайн-доставки їжі оцінювався в 221,65 мільярда доларів і, як очікується, зростатиме зі середньорічним темпом приросту у 10%, у період з 2023 по 2030 рік [1]. Гарний прогноз підкреслює потужний потенціал для бізнесу, що працює у сфері доставки їжі онлайн. Однак, це призведе до високого конкурентного середовища до якого треба адаптуватися впроваджуючи інновації.

З кожним днем кількість мобільних користувачів Інтернету зростає. Відсоток користувачів мобільних пристроїв перевищив кількість користувачів настільних ПК у всьому світі в Інтернеті [2]. У 2019 році Google підрахувала, що 61% усіх пошукових запитів надходили від людей, які шукали на своїх телефонах. Справа в тому, що Google дивиться на мобільну версію вашого вебсайту, щоб визначити, яке місце він повинен мати в результатах пошукової системи [3].

Коли мова йде про вебдизайн, простота – найкращий вибір. Багато вебсайтів страждають від надскладних елементів, таких як навігація. Потрібно

думати про те, як вебсайт буде сприйматися користувачем, а коли мова йде про користувача, є просте правило – якщо він не зрозуміє, він піде. Ось чому зараз, як ніколи, важливо привернути увагу користувача, використовуючи адаптивний, простий і зрозумілий макет сайту.

1.2 Аналіз існуючих аналогів

Сучасний ринок, насичений різноманітними сервісами з доставки їжі, пропонує широкий спектр послуг та можливостей. Багато аналогічних ресурсів займають своє особливе місце на ринку, мають власні підходи до обслуговування клієнтів, а також відмінності у різних аспектах бізнесу, включаючи продуктову пропозицію, ціноутворення та маркетингові стратегії. Кожен з них має свої переваги та недоліки, що робить аналіз конкурентів ключовим етапом для розуміння подальших шляхів розвитку.

Було обрано 2 сервіси з доставки їжі:

- Glovo – іспанська компанія, яка пропонує сервіс доставки їжі через мобільний додаток та вебсайт. Є одним з найпопулярніших сервісів в Україні.
- JoJo – сервіс з доставки їжі в Сумах.

Для аналізу переваг та недоліків розглянемо таблицю 1.1.

Таблиця 1.1 – Аналіз існуючих аналогів

Сервіс	**Опис**	**Переваги**	**Недоліки**
1	**2**	**3**	**4**
Glovo	Навігація	Легка та зрозуміла навігація	-
JoJo		Легка та зрозуміла навігація	-
Glovo	Адаптивність	Присутня	-
JoJo		Присутня	-

Продовження табл. 1.1

1	2	3	4
Glovo	Система відгуків	Присутня	Наявний лише загальний рейтинг закладів
JoJo		-	Система відгуків відсутня
Glovo	Наявність історії замовлень	Присутня	-
JoJo		-	Історія замовлень відсутня
Glovo	Можливість замовити їжу з декількох закладів одночасно	-	Відсутня можливість
JoJo		-	Відсутня можливість

Отже, при розробці інтернет-магазину з доставкою їжі потрібно враховувати переваги та недоліки аналогічних рішень, зокрема систему відгуків, розширивши її можливості до можливості ставити оцінку та залишати коментарі, наявність історії замовлень, що дозволить користувачам переглядати останні транзакції, та можливість замовлення їжі з декількох закладів одночасно.

1.3 Аналіз предметної області

Адаптивні сайти розраховані на нескінчену кількість розмірів екрану. Що є дуже гарною новиною, оскільки пристрої з новими розмірами екрану випускаються регулярно.

Краще позиціонування в пошукових системах. Адаптивні сайти краще позиціонуються в пошукових системах, оскільки вони вважаються зручними для

мобільних пристроїв. Адаптивний вебсайт вимагає від розробника більше роботи. Навіть коли UX-дизайнер і розробник тісно співпрацюють, щоб переконатися, що макет можна використовувати на якомога більшій кількості пристроїв, неможливо контролювати макет на кожному окремому пристрої. Як наслідок, на певних пристроях вебсторінка може бути не налаштована таким чином, щоб забезпечити найкращу взаємодію з користувачем.

Повільне завантаження. Оскільки код для всього вебсайту завантажується незалежно від того, з якого пристрою доступ до сайту, адаптивний сайт може завантажуватися повільніше. Це також може поставити під загрозу взаємодію з користувачем.

Розробка адаптивного дизайну – лише один аспект успішного проєкту. Серверна частина також являється важливою частиною розробки інтернет-магазину. Дані мають першорядне значення у сфері веброзробки. Використання бази даних забезпечує безпечне зберігання та ефективний доступ до збережених даних, що є фундаментальним компонентом будь-якого додатку.

Значення баз даних у розробці вебдодатків неможливо переоцінити. Вони є наріжним каменем у побудові програми, що вимагає всебічного розуміння перед інтеграцією. Дизайн бази даних відіграє ключову роль у функціональності вебсайту, пропонуючи ефективність, цілісність даних і безпеку. Приклад бази даних можна побачити на рисунку 1.1.

Термін "база даних" був введений Пітером Науром у 1960 році, щоб описати свій підхід до розробки програмних систем. Наур дав таке визначення: "Файл можна розглядати як логічний запис фактів або ідей, тоді як база даних містить інформацію, організовану таким чином, щоб її можна було легко і гнучко використовувати"[4]. На початку розвитку комп’ютерів бази даних мали однакове значення з файлами на диску. Цей термін часто використовується в такому значенні коли люди називають свій жорсткий диск ‘основною базою даних’.

Дані – це основа кожного вебдодатка. Користувач вводить свою пошту та пароль для реєстрації – це дані, які треба зберігати для того, щоб в подальшому

не приходилось кожен раз знову реєструватися на вебсайті. База даних – це центральне сховище для всіх цих даних [4]. Вебдодатки використовують різні бази даних для зберігання даних, такі як реляційні бази даних, об'єктно-реляційні бази даних і бази даних NoSQL. Кожен тип бази даних має свої переваги та недоліки при зберіганні та пошуку даних.

```
dvdrental=# select title, release_year, length, replacement_cost from film
dvdrental-#   where length > 120 and replacement_cost > 29.50
dvdrental-#   order by title desc;
          title          | release_year | length | replacement_cost
-------------------------+--------------+--------+------------------
 West Lion               |         2006 |    159 |            29.99
 Virgin Daisy            |         2006 |    179 |            29.99
 Uncut Suicides          |         2006 |    172 |            29.99
 Tracy Cider             |         2006 |    142 |            29.99
 Song Hedwig             |         2006 |    165 |            29.99
 Slacker Liaisons        |         2006 |    179 |            29.99
 Sassy Packer            |         2006 |    154 |            29.99
 River Outlaw            |         2006 |    149 |            29.99
 Right Cranes            |         2006 |    153 |            29.99
 Quest Mussolini         |         2006 |    177 |            29.99
 Poseidon Forever        |         2006 |    159 |            29.99
 Loathing Legally        |         2006 |    140 |            29.99
 Lawless Vision          |         2006 |    181 |            29.99
 Jingle Sagebrush        |         2006 |    124 |            29.99
 Jericho Mulan           |         2006 |    171 |            29.99
 Japanese Run            |         2006 |    135 |            29.99
 Gilmore Boiled          |         2006 |    163 |            29.99
 Floats Garden           |         2006 |    145 |            29.99
 Fantasia Park           |         2006 |    131 |            29.99
 Extraordinary Conquerer |         2006 |    122 |            29.99
 Everyone Craft          |         2006 |    163 |            29.99
 Dirty Ace               |         2006 |    147 |            29.99
 Clyde Theory            |         2006 |    139 |            29.99
 Clockwork Paradise      |         2006 |    143 |            29.99
 Ballroom Mockingbird    |         2006 |    173 |            29.99
(25 rows)
```

Рисунок 1.1 – Приклад бази даних

Проєктування та розробка бази даних не є достатніми для забезпечення зв'язку між клієнтом та сервером. Для цього необхідно створити API (Application Programming Interface), який буде використовуватися для взаємодії між клієнтською та серверною частинами програмного забезпечення.

API – набір правил, інструментів, які дозволяють різним програмам взаємодіяти одна з одною. Наприклад, метеорологічна система містить щоденні дані про погоду у різних частинах світу. Вебдодаток «спілкується» з цією системою через API і відображає щоденні оновлення погоди на вашому телефоні або комп’ютері. Кожен раз, коли користувач відвідує будь-яку сторінку в мережі, він взаємодіє з API віддаленого сервера. API-це складова частина сервера, яка

отримує запити і відправляє відповіді [5]. Цей процес можна відобразити на рисунку 1.2.

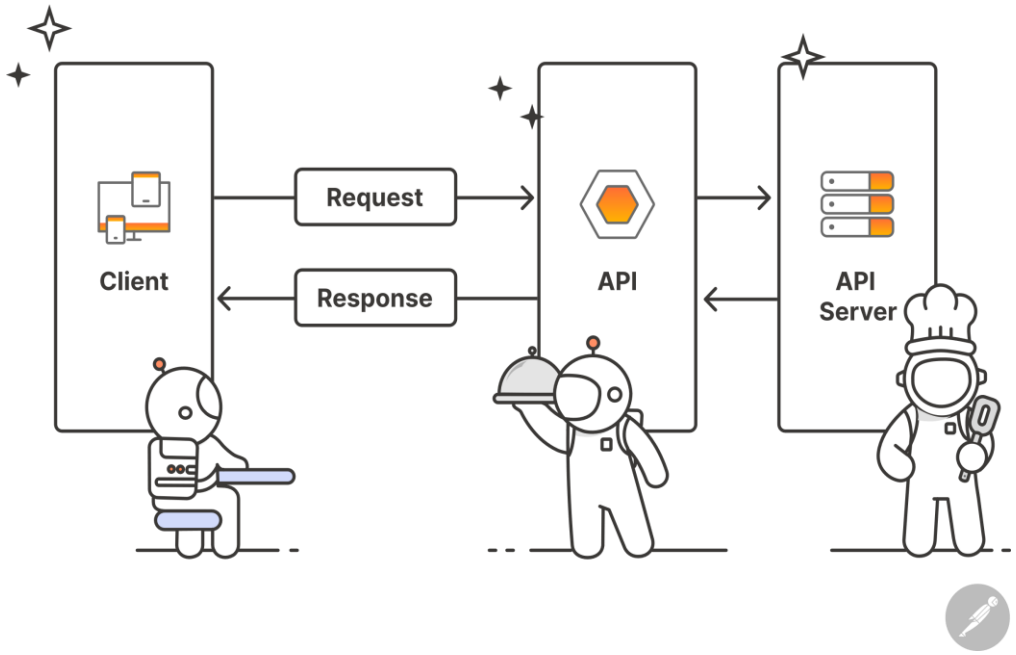


Рисунок 1.2 – Як працює API

Як можна побачити з рисунка 1.2, користувач надсилає запит до сервера, API отримує цей запит та надсилає його на сервер з базою даних. Сервер, знайшовши необхідні файли, надсилає відповідь клієнту. Якщо сервер не знайшов необхідних даних, він надсилає відповідний статус-код відповіді, який розробник оброблює та відображає зрозумілу для звичайного користувача помилку. Більш детально про статус-коди буде розглянуто у підрозділі 1.4.

Цей процес можна порівняти з відвіданням ресторану. У ролі користувача виступає гість ресторану, API – це офіціант, а сервер з базою даних – це кухар. Офіціант, тобто API, приймає замовлення та передає його до кухаря, який починає приготування страви. Після того, як кухар завершив приготування, він передає замовлення до офіціанта, який в свою чергу доставляє його гостю.

1.4 Статус-коди

Код стану HTTP – це тризначне число, з якого починається будь-яка

відповідь сервера на запит по протоколу HTTP. Код коротко повідомляє суть відповіді – чи був виконаний запит або виникла помилка [6].

Код стану є корисним інструментом для розробників, оскільки дозволяє їм розуміти, де і чому виникла помилка під час надсилання запиту до сервера. Однак для користувача такі коди можуть бути не зрозумілі, особливо якщо розробник не обробив помилку і показав лише трицифрове число. Наприклад, при реєстрації користувач вводить пароль, який не відповідає вимогам, що описані у базі даних. У цьому випадку сервер поверне статус-код 400 – Bad Request. Більш детально про основні статус-коди можна дізнатися з таблиці 1.2 – Основні статус-коди. Користувач скоріше за все не зрозуміє, в чому полягає його помилка і може залишити ваш вебсайт, не зробивши замовлення.

Коди відповідей можна розділити на п’ять класів. У кожному класі є різна кількість статус-кодів і всі вони призначені для своєї ситуації [6]. Наведемо коротку інформацію про кожен з п’яти класів.

- 1xx – інформаційні коди. Повідомляють про процес виконання запиту. На практиці дуже рідко використовується.
- 2xx – коди про успішне виконання. Повідомляють про успішне виконання запиту;
- 3xx – коди перенаправлень. Повідомляють користувача про те, що запитувана сторінка переїхала і потрібно зробити запит до нового URL;
- 4xx – коди помилок зі сторони клієнта. Повідомляють про помилку користувача, який надсилає запит. Наприклад, не правильно введений пароль, або будь яке інше значення;
- 5xx – коди помилок зі сторони сервера. Повідомляють про помилку сервера, на який надсилається запит. Наприклад, сервер не відповідає.

Існує близько 40 статус-кодів. Основні з яких, які найчастіше використовуються, можна побачити у таблиці 1.2.

Таблиця 1.2 – Основні статус-коди

Статус-код	Опис
200 OK	Сервер успішно обробив запит користувача.
201 Created	Сервер успішно обробив запит і створив новий ресурс.
301 Moved Permanently	Запитуваний ресурс переїхав назавжди, та доступний за іншим URL.
400 Bad Request	Сервер відмовив обробити запит, бо у користувач ввів некоректні дані.
401 Unauthorized	Сервер відмовив обробити запит, бо користувач не авторизувався.
403 Forbidden	Ресурс заборонено. Сервер відмовив обробити запит, бо у користувача немає прав.
404 Not Found	Сторінку не знайдено.
500 Internal Server Error	Внутрішня помилка сервера.
502 Bad Gateway	Невірний шлюз. Сервер, який працює як шлюз або проксі, отримав некоректну відповідь від іншого сервера.

1.5 Постановка задачі

Метою кваліфікаційної роботи є розробка адаптивного інтернет-магазину, який надасть користувачам зручну та ефективну платформу для замовлення їжі. Цей магазин буде відповідати різним розмірам екранів і пристосовуватися до різних пристроїв, що забезпечить безперервний доступ користувачів до сервісу незалежно від використовуваного пристрою. Крім того, метою є створення інтуїтивно зрозумілого інтерфейсу, який спростить процес вибору страв, оформлення замовлення та взаємодії з магазином для забезпечення максимальної задоволеності користувачів.

Розроблене програмне забезпечення повинно задовольняти наступні вимоги:

- адаптивний дизайн;
- мінімалістичний інтерфейс;
- фільтрування ресторанів за типом кухні;
- фільтрування страв за їхнім типом;
- система відгуків;
- вибір мови інтерфейсу;
- вибір теми вебсайту, світла або темна;
- система реєстрації та авторизації;
- наявність адмін-панелі з можливістю додавати, видаляти або переглядати наявні ресторани, страв та користувачів;
- надати можливість користувачу видалити свій акаунт з бази даних;
- можливість додавати страви з різних ресторанів до кошика;
- можливість оформити замовлення;
- можливість перегляду історії замовлень;
- наявність сторінки профілю з інформацією про користувача;
- забезпечити безпеку додатку та відхиляти запити на несанкціонований доступ до адмін-панелі.

Для досягнення мети необхідно виконати наступні кроки:

1) Провести аналіз предметної області та зрозуміти актуальність;
2) Провести аналіз інструментів розробки;
3) Обрати технології для розробки програмного забезпечення;
4) Реалізувати серверну частину інтернет-магазину з доставки їжі;
5) Провести навантажувальне тестування серверної частини та тестування на працездатність;
6) Розробити унікальний дизайн вебдодатку, який задовольняє базові потреби користувачів;
7) Реалізувати клієнтську частину додатку, яка забезпечує адаптивність сторінок до різних розмірів екранів;
8) Провести тестування адаптивності;
9) Проаналізувати отримані результати.

2 ВИБІР МЕТОДУ РІШЕННЯ

2.1 Вибір бази даних

При плануванні нового проєкту, часто виникає питання про вимоги до бази даних. В наш час дані можна поділити на два типи:

- Оперативні – використовуються для щоденних транзакцій і повинні бути в актуальному стані;
- Аналітичні – зазвичай використовуються бізнесом для отримання інформації про поведінку клієнтів, продуктивність продуту та прогнозування. Вони включають дані, зібрані протягом певного періоду часу.

Бази даних - це найефективніший спосіб постійного зберігання та отримання оперативних та аналітичних *даних* у цифровому вигляді [7]. Виходячи з вимог до проєкту, розробникам потрібно обрати базу даних, яка зможе:

- Зберігати всі типи даних;
- Мати швидкий доступ до необхідних даних;
- Отримувати інформацію швидко для прийняття стратегічних бізнес рішень.

Існує багато типів баз даних, але розглянемо реляційні та нереляційні. Реляційні бази даних зберігає інформацію у таблицях. Часто, ці таблиці мають спільну інформацію між собою, що призводить до утворення зв’язків між ними. Звідси і походить назва реляційної бази даних. Приклад такої бази даних можна побачити на рисунку 2.1.

Найпоширенішим способом взаємодії з реляційними системами є використання мови SQL (Structured Query Language). Розробники пишуть SQL-запити для виконання CRUD-операцій (Create, Read, Update, Delete). Приклад Read-запиту можна побачити на рисунку 2.2.

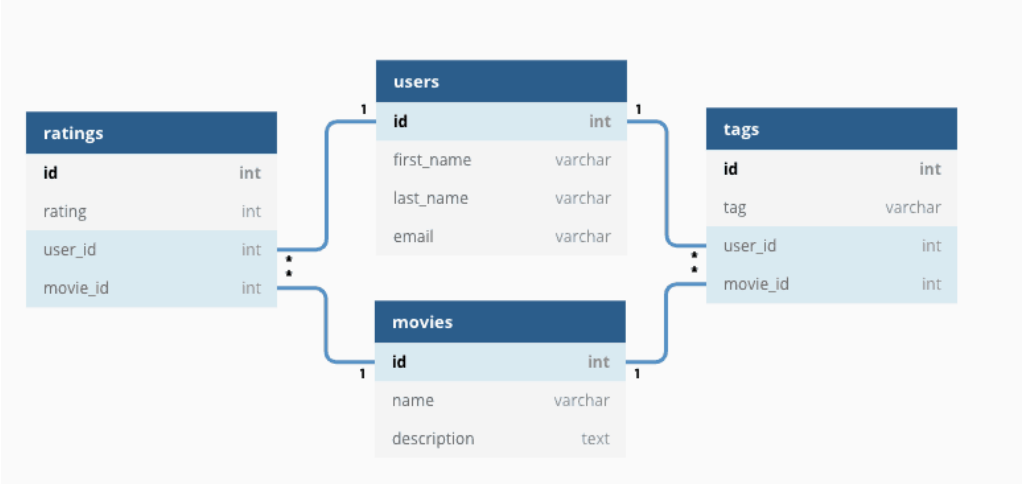


Рисунок 2.1 – Приклад реляційної бази даних

```
SELECT PRODUCT_NAME, PRICE FROM PRODUCT WHERE PRODUCT _ID = 23;
```

Рисунок 2.2 – Приклад SQL-запиту

Переваги реляційних баз даних – це використання первинних та зовнішніх ключів, що дозволяє забезпечити відсутність дублювання інформації. Ще одна перевага – це простота використання. SQL бази даних існують дуже давно, тому велика кількість інструментів та ресурсів були розроблені, які допомагають розпочати розробку з реляційними базами даних та взаємодіяти з ними.

Також існує головний недолік, це продуктивність. Продуктивність бази даних тісно пов'язана зі складністю таблиць - їх кількістю, а також обсягом даних у кожній таблиці. Зі збільшенням цього показника збільшується і час, необхідний для виконання запитів.

Нереляційні бази даних, іноді вони називаються NoSQL (Not only SQL). Це система яка не використовує концепцію структурованих таблиць. Нереляційні бази даних були розроблені з урахуванням хмарних технологій, що робить їх ідеальними для горизонтального масштабування [7]. Існує кілька різних груп, які зберігають інформацію по-різному, але зупинимось на Document баз даних, яку було використано для створення серверної частини.

Document бази даних зберігають дані, які є JSON-подібними структурами, що підтримують різні типи даних. Ці типи включать рядки, числа, масиви, об’єкти та навіть інші документи з даними. Дані зберігаються парами ключ – значення. Приклад можна побачити на рисунку 2.3.

Рисунок 2.3 – Зберігання даних в документній нереляційній базі даних

На рисунку 2.4 наведено приклад запиту для отримання назви товару та ціни за заданим productid за допомогою Mongo Query API.

```
db.product.find({"_id": 23}, {productName: 1, price: 1})
```

Рисунок 2.4 – Приклад запиту у MongoDB

Завдяки тому, що документи мають формат JSON, їх набагато легше читати і розуміти користувачеві. Дані організовані, а також легко переглядаються. Немає необхідності звертатися до декількох документів або колекцій, щоб переглянути дані одного клієнта [7]. Отже, можна скласти порівняльну таблицю реляційних та нереляційних баз даних, таблиця 2.1.

Таблиця 2.1 – Порівняльна таблиця баз даних

Особливості	**Нереляційні**	**Реляційні**
Доступність	Висока	Висока
Горизонтальне масштабування	Високе	Низьке
Вертикальне масштабування	Високе	Високе
Зберігання даних	Оптимізовано для великих обсягів даних	Оптимізовано для середньої кількості даних
Продуктивність	Висока	Низька
Гнучкість	Висока	Низька

Дивлячись на таблицю 2.1, можна сказати, що MongoDB – нереляційна база даних, яка пропонує масштабованість, високу продуктивність, надійність та гнучкість, є найкращим вибором.

2.2 Вибір фреймворку для серверної частини

Фреймворк – це набір інструментів, бібліотек та правил, який використовується для створення програмних додатків [8]. Цей набір інструментів визначає як компоненти програми повинні взаємодіяти між собою. Використання фреймворку значно спрощує розробку, так як надає вже готові рішення для багатьох випадків.

Вибір фреймворку це важливий етап у процесі розробки. Хоча існує можливість вибору будь-якого набору інструментів на будь якій мові програмування, проте зупинимось на JavaScript фреймворках так як маю досвід роботи з ним. Існує величезна кількість JS фреймворків для розробки серверної частин. Деякі з них наведено у списку:

- ExpressJS – пропонує мінімалістичний фреймворк, який спрощує маршрутизацію, управління проміжним програмним забезпеченням (middleware) та конфігурацію серверів. Ця простота прискорює розробку без шкоди для функціональності;
- NestJS – фреймворк для створення ефективних, масштабованих серверних додатків на Node.js. Він використовує сучасний JavaScript, побудований за допомогою TypeScript і поєднує в собі елементи ООП (об'єктно-орієнтованого програмування), ФП (функціонального програмування) та ФРП (функціонально-реактивного програмування) [9];
- NuxtJS - фреймворк для створення серверних застосунків за допомогою Vue.js;
- BackboneJS – хоча Backbone.js частіше асоціюється з фронтенд-розробкою, його також можна використовувати на стороні сервера. Це фреймворк, який може допомогти структурувати ваш додаток і надати необхідні інструменти для створення вебсервісів.

Для розробки серверної частини інтернет-магазину був обраний NestJS, використовуючи TypeScript. Це рішення обумовлено тим, що цей фреймворк з кожним днем стає більш популярним. Якщо порівняти кількість поставлених запитань на stack overflow серед NestJS та NuxtJS, то можна побачити, що nest випереджає у своїй популярності nuxt, але поступається express (рис. 2.5).

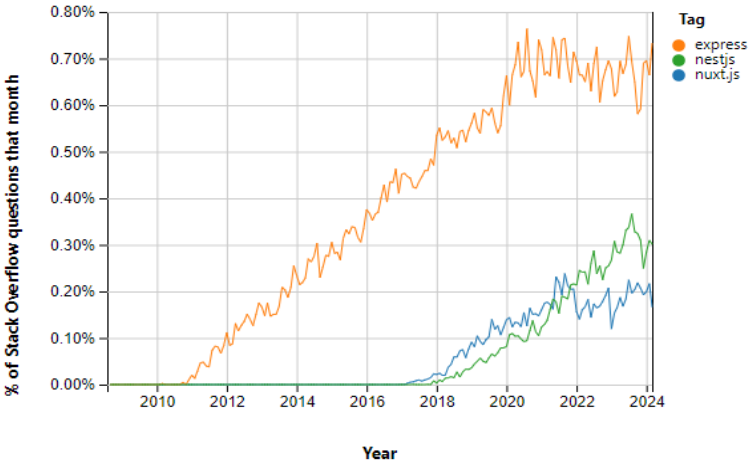


Рисунок 2.5 – Кількість поставлених питань на Stack Overflow

Також використання TypeScript замість JS обумовлене тим, що TypeScript надає додаткові переваги, які підвищують якість коду. Деякі з цих переваг включають:

Статична типізація - дозволяє визначати типи даних змінних, параметрів та повертаємих значень функцій. Це допомагає виявляти помилки ще на етапі розробки та покращує читабельність коду;

Підтримка класів та модулів – що спряє структурованому коду та полегшує його перевикористання.

Отже, використання TypeScript разом з NestJS допомагає підвищити ефективність та якість розробки серверної частини інтернет-магазину.

2.3 Вибір фреймворку для клієнтської частини

Як вже було сказано раніше існує велика кількість JS фреймворків. Такі популярні фреймворки, як React, Angular, Vue, то здобувають, то втрачають велику кількість прихильників, але їм вдається триматися в перших місцях в рейтингу найкращих фреймворків. Згідно статистики Stack Overflow [10] у 2024 React посідає перше місце серед найпопулярніших JavaScript фреймворків, рисунок 2.6.

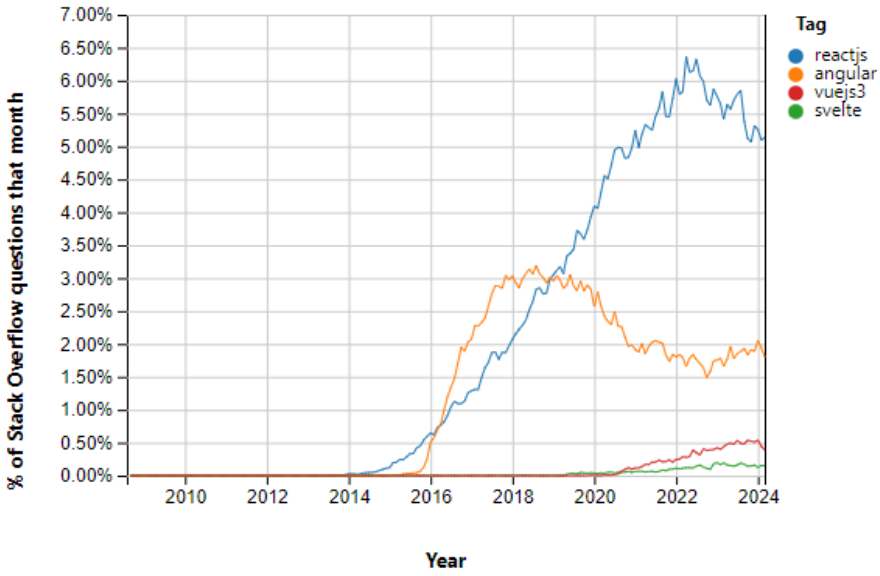


Рисунок 2.6 – Найпопулярніші фреймворки

На другому місці можна побачити Angular, який відомий своєю технологією двостороннім зв’язуванням даних. Третє місце посідає VueJS, відомий своєю простотою у використані. Для розробки клієнтської частини було обрано фреймворк React. Цей вибір обумовлений тим, що:

- Використання JSX синтаксису дозволяє писати JavaScript код прямо у HTML розмітці, що прискорює розробку;
- Віртуальний DOM - створює віртуальне представлення або копію DOM під назвою Virtual DOM або vDOM. React порівнює віртуальний DOM з реальним DOM, щоб перезавантажити тільки ті компоненти, які змінилися, замість того, щоб перезавантажувати всю сторінку. Це ключ до блискавичної продуктивності React [11].

2.4 Інструмент для створення адаптивного дизайну

TailwindCSS – це CSS фреймворк з відкритим вихідним кодом [12]. Даний інструмент наповнений готовими CSS класами, які можна комбінувати для створення будь-якого дизайну безпосередньо у розмітці HTML.

Традиційно, коли потрібно щось стилізувати на вебсайті, треба писати CSS, рисунок 2.7.

```
.chat-notification {
  display: flex;
  align-items: center;
  max-width: 24rem;
  margin: 0 auto;
  padding: 1.5rem;
  border-radius: 0.5rem;
  background-color: #fff;
  box-shadow: 0 20px 25px -5px rgba(0, 0, 0, 0.1), 0 10px 10px -5px rgba(0, 0, 0, 0
}
.chat-notification-logo-wrapper {
  flex-shrink: 0;
}
.chat-notification-logo {
  height: 3rem;
  width: 3rem;
}
.chat-notification-content {
  margin-left: 1.5rem;
}
```

Рисунок 2.7 – Приклад CSS коду

За допомогою TailwindCSS можна стилізувати елементи, застосовуючи вже існуючі класи, рисунок 2.8.

```
<div class="p-6 max-w-sm mx-auto bg-white rounded-xl shadow-lg flex items-center
  <div class="shrink-0">
    <img class="h-12 w-12" src="/img/logo.svg" alt="ChitChat Logo">
  </div>
  <div>
    <div class="text-xl font-medium text-black">ChitChat</div>
    <p class="text-slate-500">You have a new message!</p>
  </div>
</div>
```

Рисунок 2.8 – Приклад TailwindCSS коду

Такий підхід написання стилів дозволяє реалізувати повність кастомний дизайн без написання жодного рядка CSS коду. Це надає такі переваги:

- Не потрібно постійно придумувати нові назви класів;
- CSS код перестає збільшуватися;
- Впроваджувати стилі стає безпечніше – CSS код є глобальним, і при внесенні будь яких змін не можна бути впевненим, що щось не ‘попливе’ у стилях. Готові компоненти у TailwindCSS належать тільки тому тегу, де вони написані, тому при внесені змін можна не турбуватися про те, що щось буде порушено.

Ще одна перевага TailwindCSS – це його продуктивність. Він генерує найменший можливий файл CSS, генеруючи лише той код, який насправді використовується у проєкті. Прикладом може слугувати компанія Neflix, яка використовує Tailwind на своїй сторінці ‘Топ 10 фільмів від Netflix’. Ця сторінка генерує лише 6.5 кілобайтів CSS коду [13].

TailwindCSS дозволяє легко забезпечити адаптивність користувацького інтерфейсу. Кожен клас можна застосовувати у різних контрольних точках. Існує 5 контрольних точок за замовчуванням, які мають найпоширеніші роздільні здатності пристроїв, рисунок 2.9.

Breakpoint prefix	Minimum width
`sm`	640px
`md`	768px
`lg`	1024px
`xl`	1280px
`2xl`	1536px

Рисунок 2.9 – Контрольні точки

Для малих екранів використовується префікс sm з мінімальною шириною екрану у 640 пікселів. Для середніх екранів md з мінімальною шириною 768 пікселів і так далі.

TailwindCSS працює за принципом mobile-first. Це означає, що класи написані без префіксів (наприклад: uppercase) діють на всіх розмірах екранів, тоді як клас з префіксом (наприклад, md:uppercase) діє лише у вказаній контрольній точці і вище.

2.5 Вибір репозиторія для зберігання коду

Репозиторій – централізоване місце для зберігання та керування версіями програмного коду. Замість того, щоб зберігати код на жорсткому диску, його можна тримати у віддаленому репозиторії. Це має багато переваг:

Забезпечення безпеки даних – код, збережений на жорсткому диску комп’ютера, може бути втрачений або пошкоджений внаслідок непередбачуваних обставин. У випадку збереження у віддаленому репозиторії дані захищені від таких ризиків, оскільки вони зберігаються на сервері, який може мати резервне копіювання та інші заходи забезпечення безпеки.

Інша перевага це зручний доступ до даних. Коли код збережений у віддаленому репозиторії, розробник може отримати до нього доступ з будь-якої точки світу та працювати з ним.

Остання перевага яку б хотілось навести, це керування історією змін. Віддалені репозиторії, такі як Git, дозволяють відслідковувати історію змін у коді, відновлювати попередні версії та керувати гілками розвитку проєкту. Це робить процес розробки більш систематизованим та контрольованим.

Вибір репозиторія пав на GitHub через його популярність та легкість використання. GitHub є одним з найпопулярніших сервісів для збереження коду, який має велику спільноту розробників та активну підтримку. Його інтерфейс є дружнім до користувача, а функціональність дозволяє ефективно керувати репозиторіями, відстежувати зміни, створювати гілки та злиття, також проводити перегляди коду та робити коментарі до нього.

3 ПРАКТИЧНА РЕАЛІЗАЦІЯ

3.1 Проєктування бази даних

Перед початком проєктування бази даних потрібно визначитися з основними вимогами до інтернет-магазину з доставки їжі. Функціональні вимоги визначають, які функції повинен мати продукт і як вони повинні працювати. Нефункціональні вимоги визначають якості продукту, такі як безпека, надійність, продуктивність та інші аспекти, які не стосуються конкретно функціональності.

Функціональні вимоги до інтернет-магазину включають:

- Реєстрація та авторизація – можливість створення облікового запису та входу в систему для кожного користувача;
- Перегляд переліку ресторанів – користувач може переглядати список доступних ресторанів;
- Перегляд переліку блюд кожного ресторану – користувач може переглядати список доступних страв у кожному з ресторанів;
- Залишення відгуків про ресторан – користувач може залишити відгук про ресторан та поставити свою оцінку;
- Створення замовлень – користувач може створювати замовлення з різних ресторанів;
- Налаштування загальних вподобань користувача – користувач може налаштувати загальні вподобання до сайту, а саме вибір теми (темна або світла), вибір мови вебсайту (англійська або українська).

Нефункціональні вимоги включають:

- Безпека – забезпечення безпеки даних користувача;
- Надійність – забезпечення надійності серверної частини та відсутності відмов;
- Продуктивність – забезпечення швидкої роботи системи та швидке реагування на запити користувача.

Визначившись з вимогами, можна починати створювати схеми у базі даних. Схеми можна побачити у таблицях 3.1 – 3.5.

Таблиця 3.1 – Схема користувача

Користувач	
Назва поля	**Тип даних**
Ім’я	String
Прізвище	String
Адреса	String
Номер телефону	String
Пошта	String
Пароль	String
Роль (за замовчуванням USER)	USER | ADMIN | DELIVERYMAN

Таблиця 3.2 – Схема ресторанів

Ресторан	
Назва поля	**Тип даних**
Назва	String
Тип кухні	String
Адреса	String
Час початку роботи	String
Час закінчення роботи	String
Середній рейтинг (формується на основі відгуків користувачів)	Number
Відгуки	Review[]
Номер телефону	String

Таблиця 3.3 – Схема відгуків

Відгук	
Назва поля	**Тип даних**
User_ID	String
Restaurant_ID	String
Рейтинг (від 1 до 5)	Number
Коментар	String

Таблиця 3.4 – Схема страв

Страва	
Назва поля	**Тип даних**
Restaurant_ID	String
Назва	String
Вага	Number
Опис	String
Інгредієнти	Array of strings[]
Калорії	Number
Категорія	String
Ціна	Number

Таблиця 3.5 – Схема замовлення

Замовлення	
Назва поля	**Тип даних**
User_ID	String
Dish_ID	String
Restaurant_ID	String
Статус	PROCESSING | COOKING | DELIVERY | DELIVERED
Середній час очікування	Number
Дата	Date
Загальна ціна	Number

Визначившись з вимогами та таблицями, можна приступати до розробки.

3.2 Розробка серверної частини

Перед початком розробки потрібно створити базу даних. Для цього заходимо на офіційний вебсайт MongoDB, рисунок 3.1, далі реєструємось.

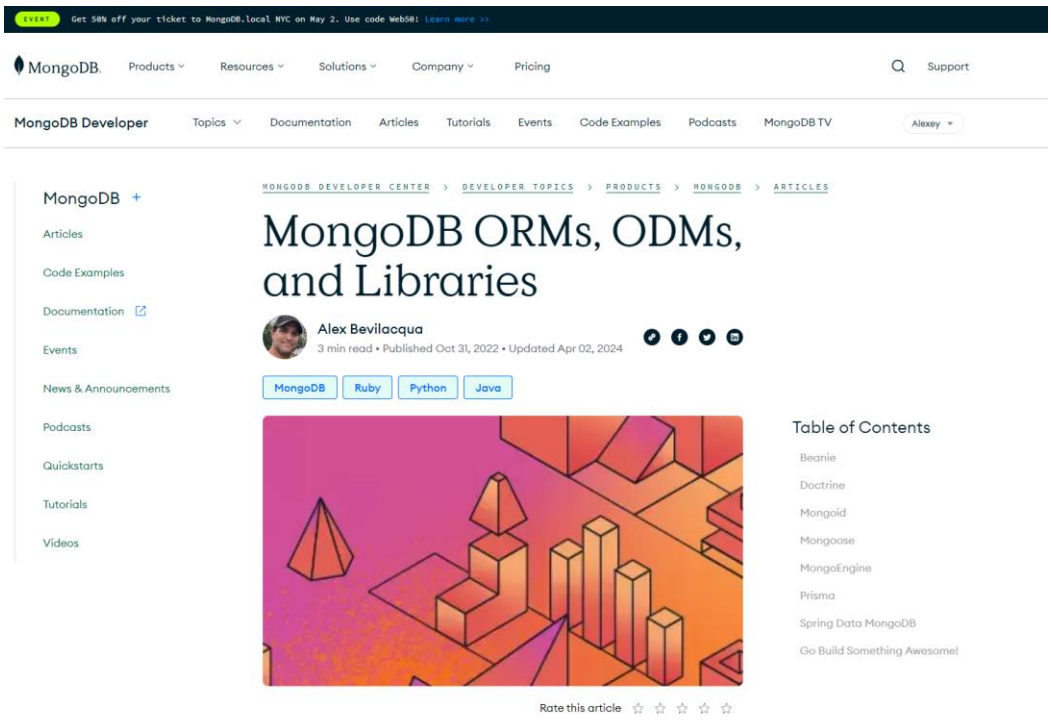


Рисунок 3.1 – Офіційний сайт MongoDB

Після реєстрації потрібно створити проєкт, задаємо назву та налаштовуємо провайдера, регіон розташування сервера, рисунок 3.2.

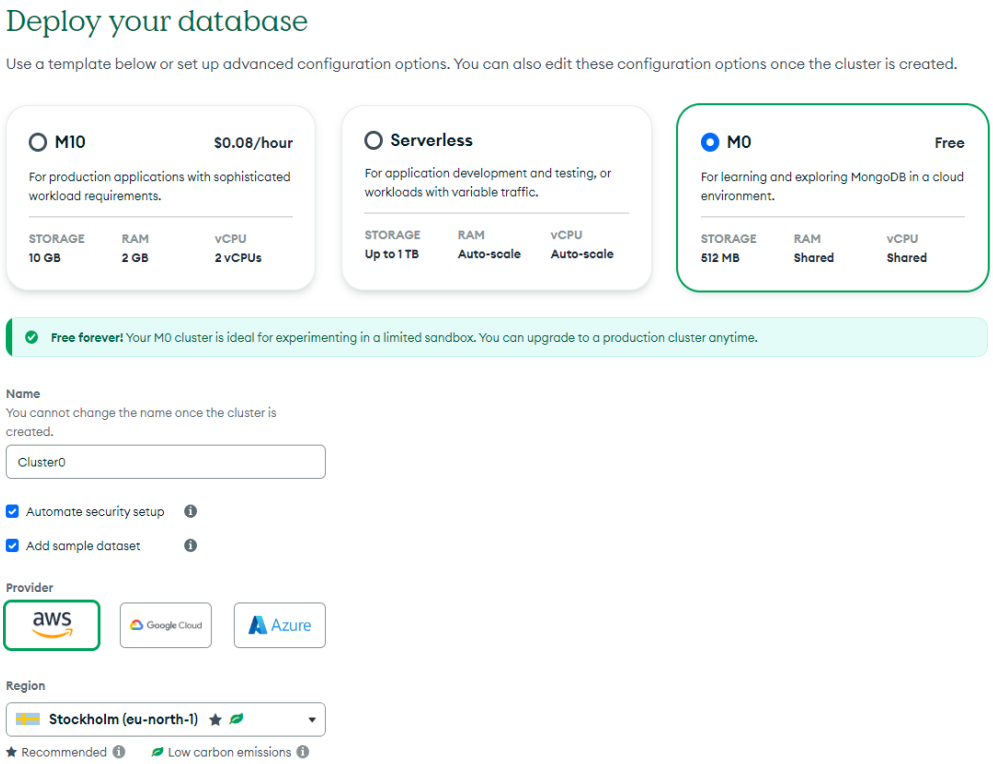


Рисунок 3.2 – Налаштування сервера

Далі отримуємо посилання, завдяки якому можна під’єднатися до сервера. Структуру бази даних можна побачити на рисунку 3.3

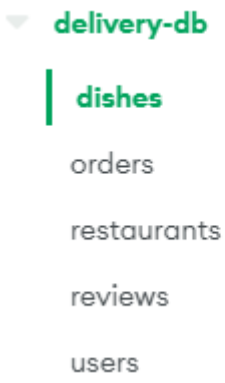


Рисунок 3.3 – Структура бази даних

Створюємо проєкт у будь якій IDE, яка підтримує NestJS. У нашому випадку це Visual Studio Code. Задаємо загальну структуру папок, рисунок 3.4

- Папка auth зберігає у собі весь код пов’язаний з авторизацією;
- Папка order містить код для роботи з замовленнями;
- Папка restaurant містить код для роботи з ресторанами;
- Папка review містить код для роботи з відгуками;
- Папка user містить код для роботи з користувачами;
- Папка shared містить файли для налаштування бази даних;
- Все інше – це системні файли, які створюються автоматично при налаштування проєкту.

Файл main.ts – це точка старту проєкту. У цьому файлі ми задаємо, на якому порту повинен «слухати» наш сервер. Також тут створюється Swagger Document, який дозволяє описати структуру API, що спрощує розуміння структури запитів для фронтенд розробників, рисунки 3.5 – 3.6.

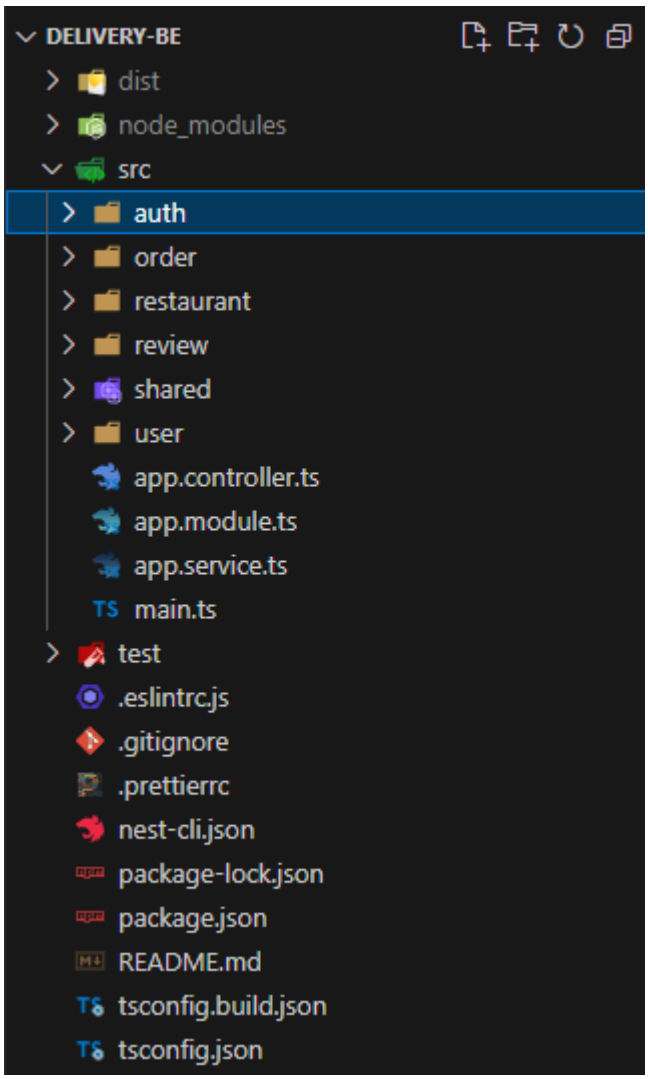


Рисунок 3.4 – Структура папок проєкту

Рисунок 3.5 – Файл main.ts

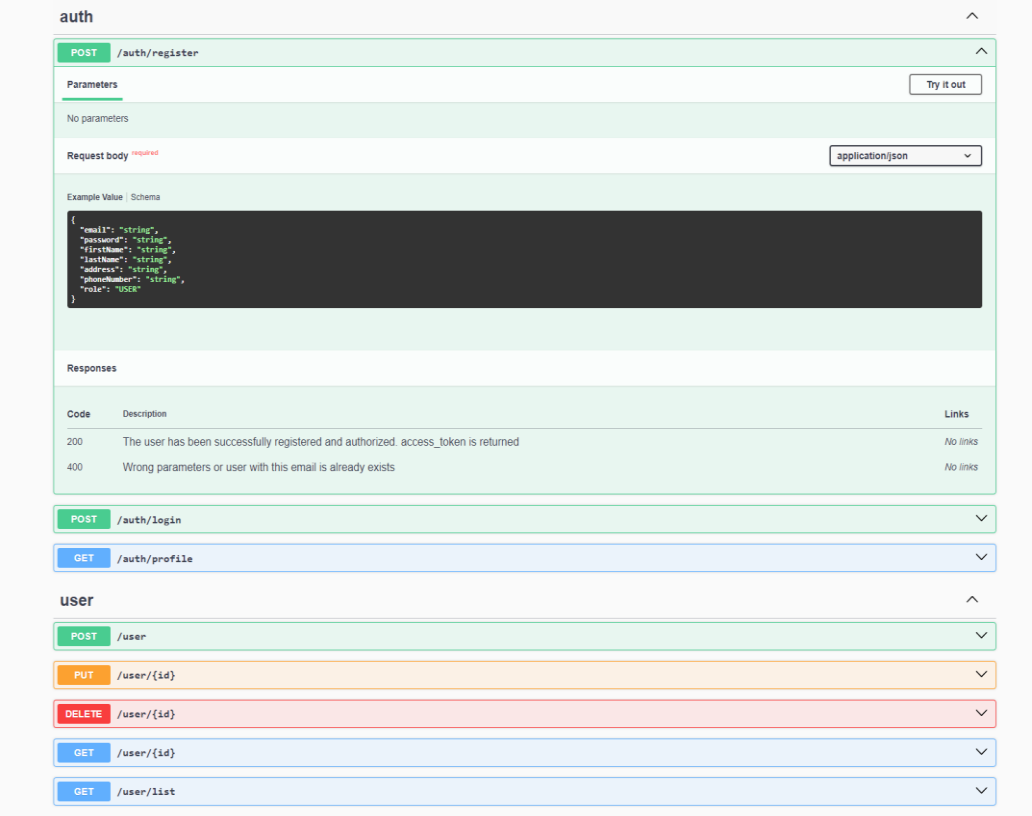


Рисунок 3.6 – Swagger Document

Почнемо розробку зі схеми користувача. У папці user створимо іншу папку schemas, яка містить файл user.schema.ts, у якому створюємо схему користувача, рисунок 3.7.

```
import { Prop, Schema, SchemaFactory } from '@nestjs/mongoose';  131.5k (gzipped: 27.7k)
import { HydratedDocument } from 'mongoose';  883.9k (gzipped: 237k)
import { Role } from 'src/auth/constants';

@Schema()
export class User {
  @Prop()
  id: string;

  @Prop()
  email: string;

  @Prop()
  firstName: string;

  @Prop()
  lastName: string;

  @Prop()
  address: string;

  @Prop()
  phoneNumber: string;

  @Prop()
  password: string;

  @Prop()
  role: Role;
}

export const UserSchema = SchemaFactory.createForClass(User);

export type UserDocument = HydratedDocument<User>;
```

Рисунок 3.7 – Схема користувача

Створюємо файли user.service.ts, user.controller.ts, user.module.ts. У service файлі було створено кінцеві точки. Кінцева точка – це місце, з якого API може отримати доступ до ресурсів, необхідних для виконання своїх функцій. Були створені такі кінцеві точки:

- createUser – дозволяє створити користувачу обліковий запис;
- checkIfEmailExist – перевіряє чи існує користувач з такою поштою;
- getUserData – повертає інформацію про користувача;
- deleteUser – видаляє обліковий запис користувача;
- getUserList – повертає список зареєстрованих користувачів;
- updateUser – дозволяє оновити дані користувача;
- getUserById – повертає дані про користувача по його ID.

Файл controller містить код, який повертає статус-коди по завершенню виконання запиту.

Файл module збирає контроллери, сервіси та експортує їх. Всі створені файли module зібрано у головний файл app.module.ts, рисунок 3.8.

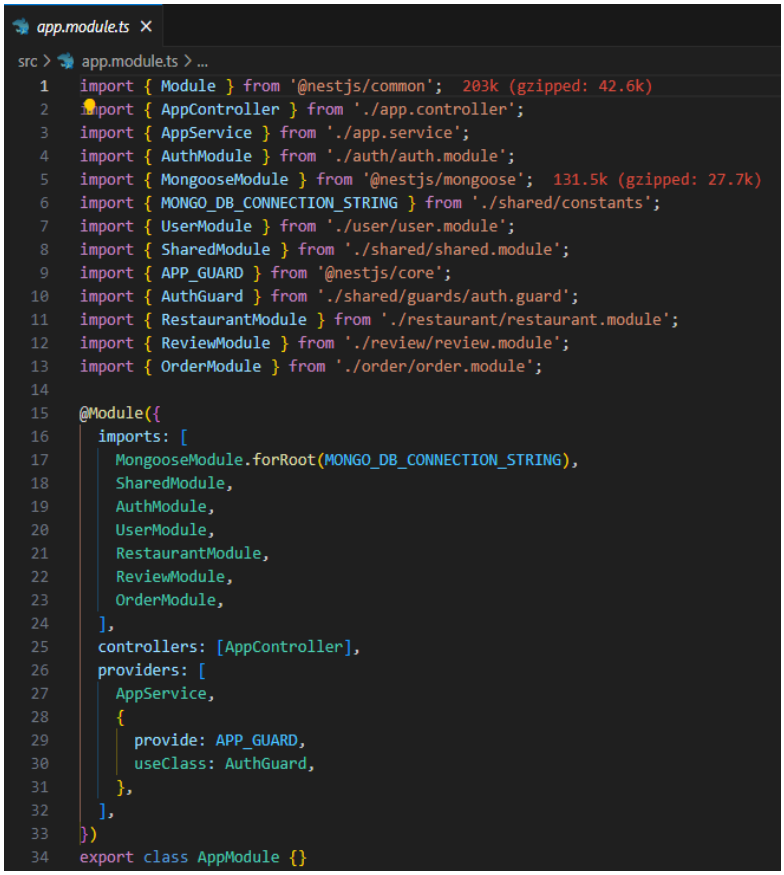


Рисунок 3.8 – Файл app.module.ts

По аналогії розробляємо інші схеми, контроллери та сервіси. Програмний код можна побачити у додатку А.

Біль детально розглянемо реєстрацію та авторизацію, які використовують JWT токен. JWT – JSON Web Token стандарт токена доступу, який використовується для передачі даних для аутентифікації [14]. Приклад токену та інформації, яку він містить можна побачити на рисунку 3.9. JWT може зберігати в собі різну інформацію, в нашому випадку він зберігає:

- sub – використовується для визначення об'єкта, якому належить токен;
- id – id користувача;
- email – пошта користувача;
- firstName – ім’я користувача;
- lastName – прізвище користувача;
- address – адреса користувача;
- role – роль користувача;
- phoneNumber – номер телефону користувача.

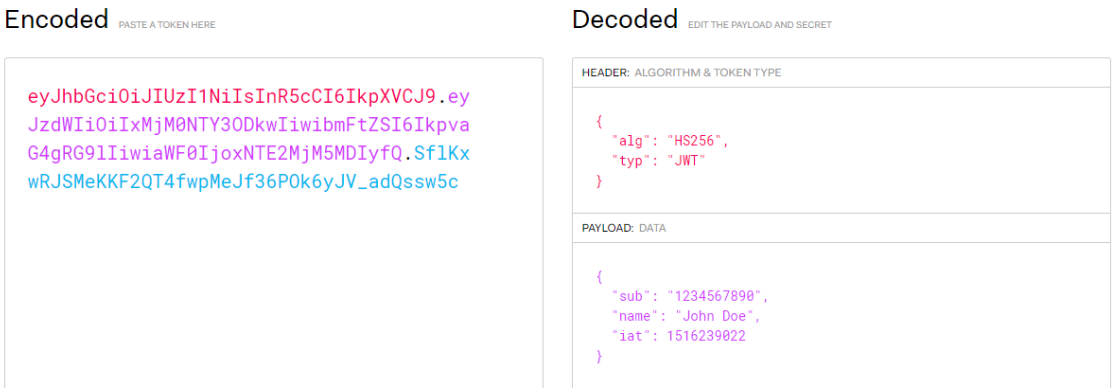


Рисунок 3.9 – Приклад JWT токену

Після успішної реєстрації або авторизації користувачу надається токен доступу. Без нього він не зможе використовувати розроблений вебсайт. Токен розшифровується за допомогою ключа JWT, який потрібно самостійно задати, і розробник отримує інформацію про роль користувача, на основі якої користувачу дозволяються або забороняються деякі дії. Наприклад, доступ до адмін-панелі надається лише користувачу з роллю ADMIN. Пароль користувача

зберігається у зашифрованому виді у базі даних, для його ж безпеки, рисунки 3.10 – 3.11.

Рисунок 3.10 – Запис про користувача у базі даних

```
private encryptPassword(password: string): string {
  return Buffer.from(password).toString('base64');
}
```

Рисунок 3.11 – Шифрування паролю

Програмний код був розроблений з дотриманням принципів ООП, що дозволяє довгострокову підтримку та легке розширення функціоналу. Після розробки потрібно провести тестування працездатності та навантажувальне тестування серверної частини.

3.3 Тестування серверної частини

Тестування будемо проводити за допомогою програми Postman. Postman - дозволяє створювати і виконувати запити, документувати й проводити моніторинг сервісів в одному місці [15]. Інтерфейс програми можна побачити на рисунку 3.12.

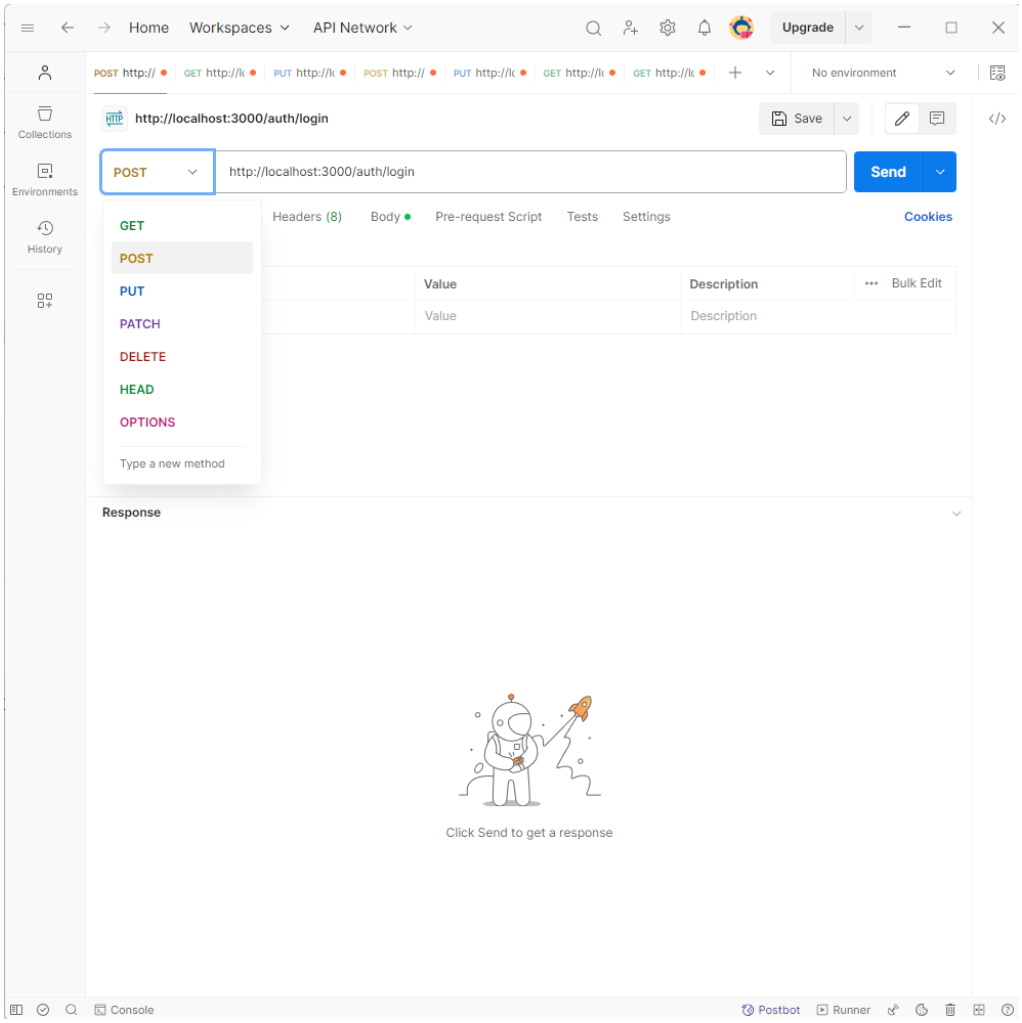


Рисунок 3.12 – Інтерфейс програми Postman

Почнемо з реєстрації та авторизації. Ці запити називаються POST – запитами, тому обираємо метод POST. Задаємо посилання на створену кінцеву точку, задаємо поля у Body у форматі JSON та натискаємо Send, рисунок 3.13.

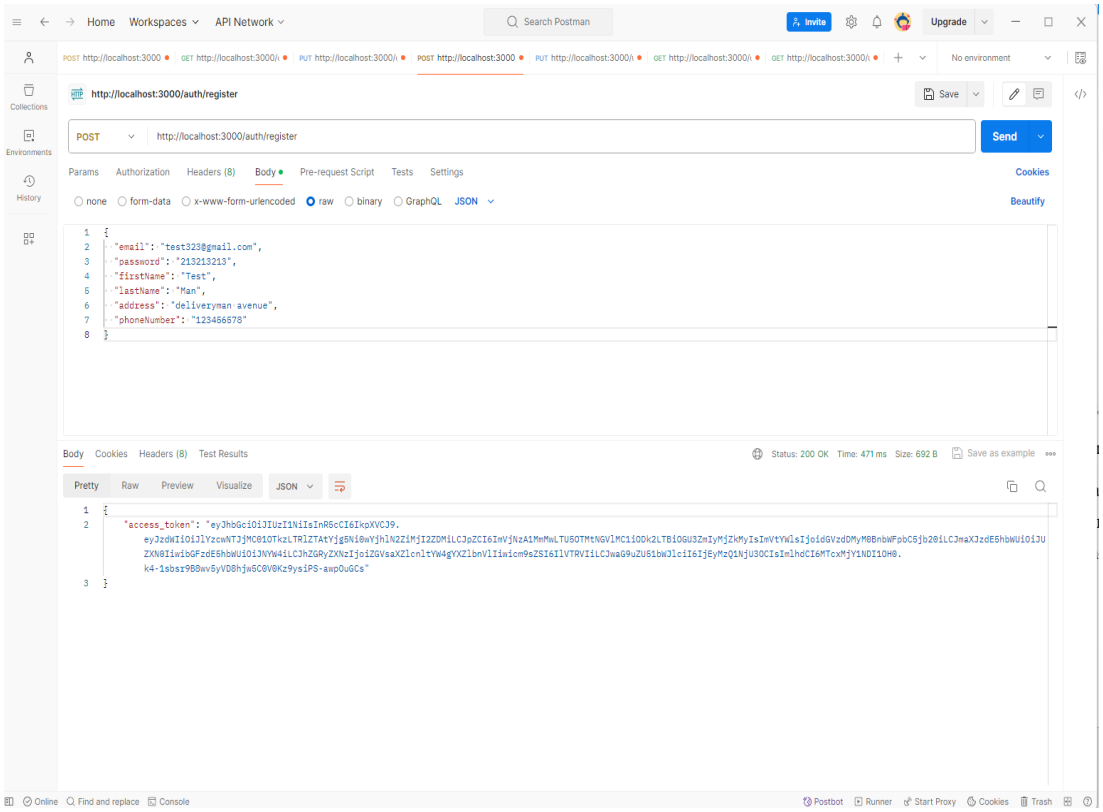


Рисунок 3.13 – Тестування реєстрації

Як можна побачити з рисунку 3.13, реєстрація працює та у відповідь надсилається JWT токен. Перевіримо авторизацію. Змінюємо посилання на кінцеву точку авторизації, задаємо необхідні поля та натискаємо Send (рис. 3.14).

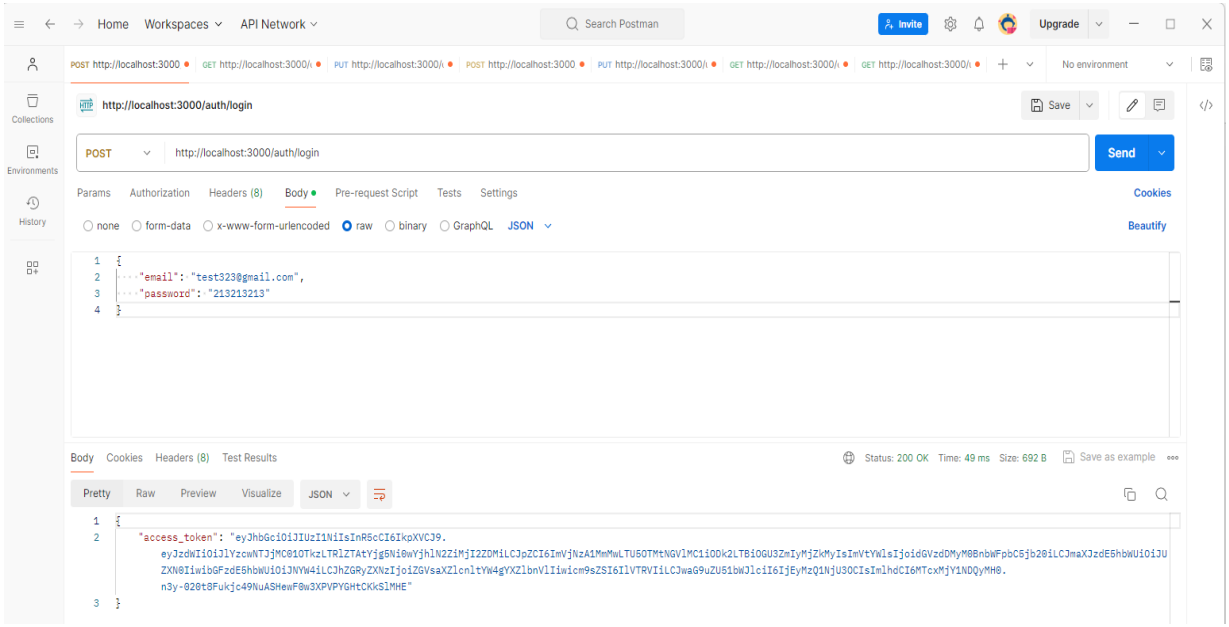


Рисунок 3.14 – Тестування авторизації

Бачимо, що авторизація також працює і JWT токен надсилається у відповідь.

Перевіримо безпеку деяких кінцевих точок, а саме отримання списку зареєстрованих користувачів. Цей запит може надсилати користувач з роллю ADMIN. Для цього змінюємо посилання на кінцеву точку і метод ставимо GET. Для того, щоб сервер зміг перевірити, що користувач має роль ADMIN потрібно задати заголовок Authorization, який містить JWT токен. Без нього сервер повинен повернути статус-код 401 Unauthorized, рисунки 3.15 – 3.16.

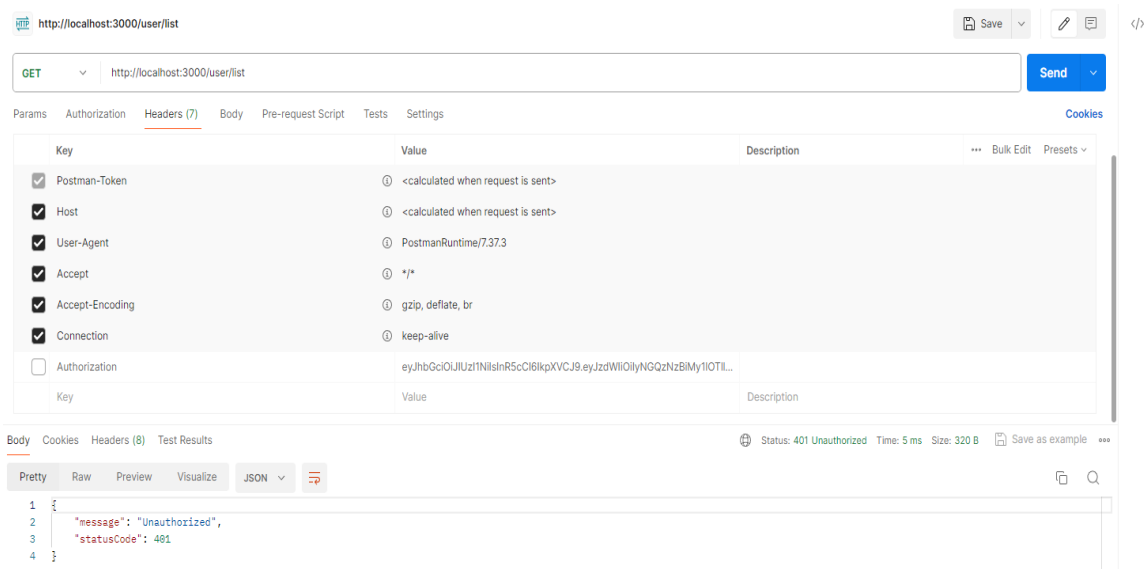


Рисунок 3.15 – Надсилання запиту без заголовку Authorization

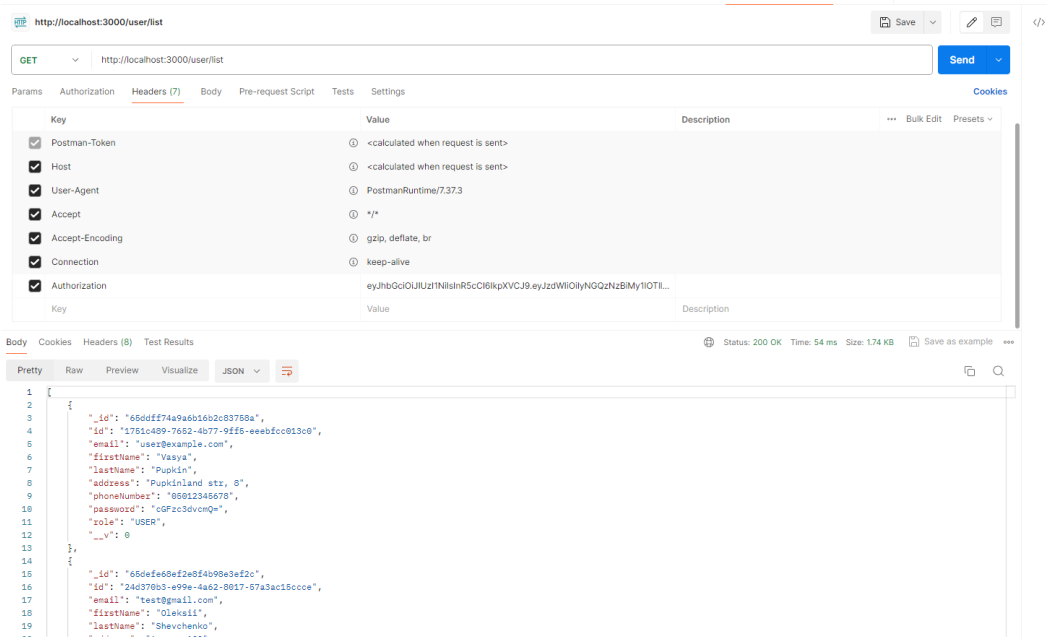


Рисунок 3.16 – Надсилання запиту з заголовком Authorization

Як бачимо всі дані користувачів зберігаються у безпеці.

Проведемо навантажувальне тестування на розроблений API, для перевірки стресостійкості та ефективності. Симулюємо ситуацію коли багато користувачів намагаються зробити різні запити одночасно. Для цього налаштуємо колекцію запитів, рисунок 3.17-3.18. Задаємо 30 віртуальних користувачів та тривалість тесту 2 хвилини.

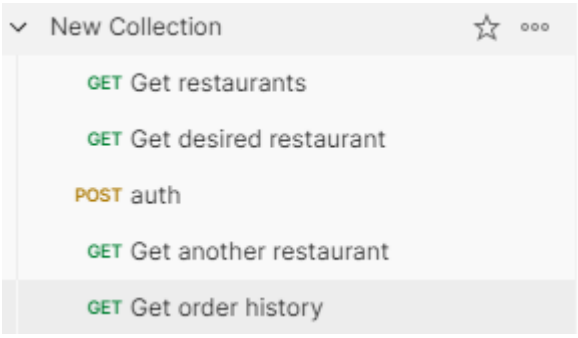


Рисунок 3.17 – Налаштування колекції запитів

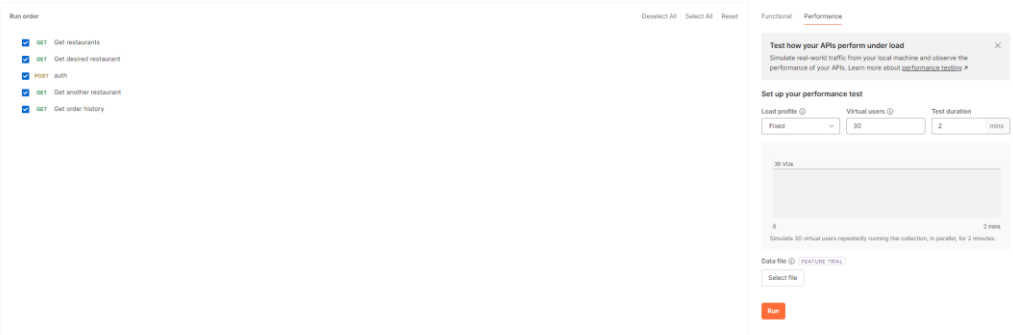


Рисунок 3.18 – Налаштування тесту

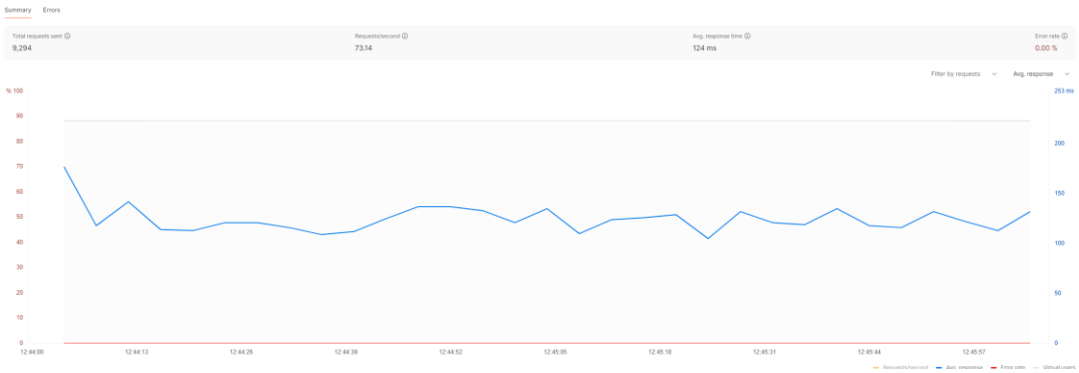


Рисунок 3.19 – Результати тесту

Як можна побачити з рисунка 3.19, кількість помилок та відмов дорівнює 0, що є дуже гарним показником стресостійкості серверної частини. Загальна кількість надісланих запитів становить 9.294, при цьому середня швидкість надсилання запитів складає 73.14 запитів за секунду. Середній час відповіді дорівнює 124 мілісекунди. Більш детальну інформацію по кожній з кінцевих точок можна побачити на рисунку 3.20.

Performance details for total duration

Request	Total requests	Requests/s	Resp. time (Avg. ms)	Min (ms)	Max (ms)	90th (ms)	Error %
GET Get restaurants	1,864	14.67	122	38	739	346	0.00
GET Get desired restaurant	1,860	14.64	142	74	1,419	423	0.00
POST auth	1,860	14.64	87	38	685	327	0.00
GET Get another restaurant	1,856	14.60	149	73	769	454	0.00
GET Get order history	1,854	14.59	120	39	652	456	0.00

Рисунок 3.20 – Детальна інформація по кожній з кінцевих точок

На основі цих даних можна зробити висновок, що серверна частина є досить ефективною та стресостійкою. Відсутність помилок і відмов свідчить про стабільну роботу системи, а швидка відповідь сервера та низька середня тривалість обробки запитів підтверджують високу продуктивність та швидкість роботи сервера.

3.4 Створення дизайну клієнтської частини

Дизайн відіграє ключову роль у розробці будь-якого вебдодатка, формуючи візуальну ідентичність та користувацький досвід. Дизайн охоплює функціональність, зручність використання та безперешкодну інтеграцію потреб користувачів. Ретельно продуманий макет, підібрані кольори та елементи інтерфейсу мають на меті створити інтуїтивно зрозумілий досвід для користувача. У сучасному світі дизайну існує безліч інструментів для створення макетів, а саме:

- Photoshop;

- Sketch;
- Figma.

Всі ці програми мають свої плюси та мінуси, але для роботи було обрано Figma. Figma – це хмарний сервіс для дизайнерів інтерфейсів і web-розробників, з яким можна працювати безпосередньо в браузері [16]. Використовуючи цей інструмент можна створювати:

- Іконки, кнопки, форми;
- Векторні зображення;
- Ілюстрації;
- Прототипи сторінок.

Почнемо розробку макетів сторінок авторизації та реєстрації, вони є найважливішими, бо при переході на вебсайт це перші сторінки які бачить користувач. Форма реєстрації повинна містити поля: ім’я, прізвище, адреса, номер телефону, пошта, пароль. Форма авторизації повинна містити поля: пошта, пароль. Було розроблено макет форми реєстрації та авторизації, рисунки 3.21 – 3.22.

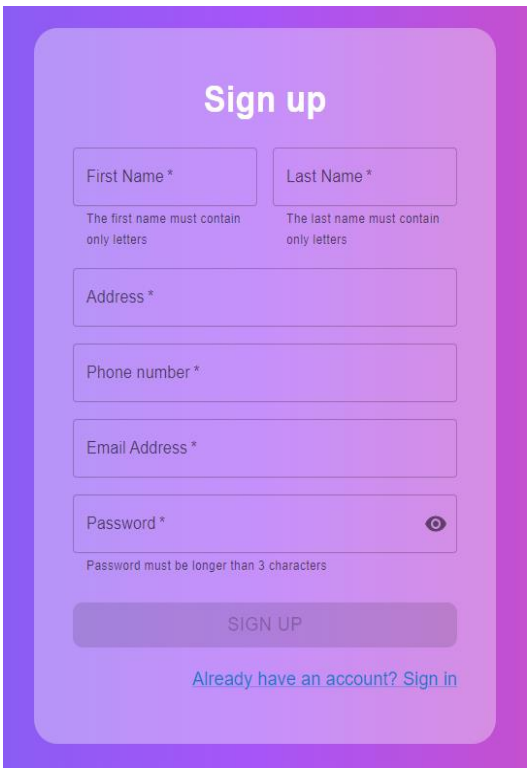


Рисунок 3.21 – Макет форми для реєстрації

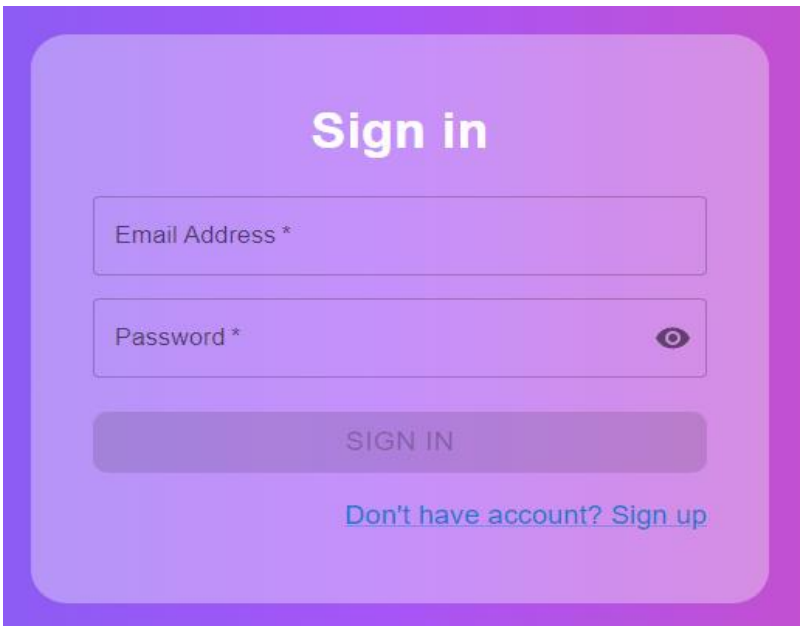


Рисунок 3.22 – Макет форми для авторизації

Для гарного досвіду користувача була додана можливість перегляду введеного паролю. Натиснувши на іконку ока, можна побачити введений пароль, приклад на рисунку 3.23.

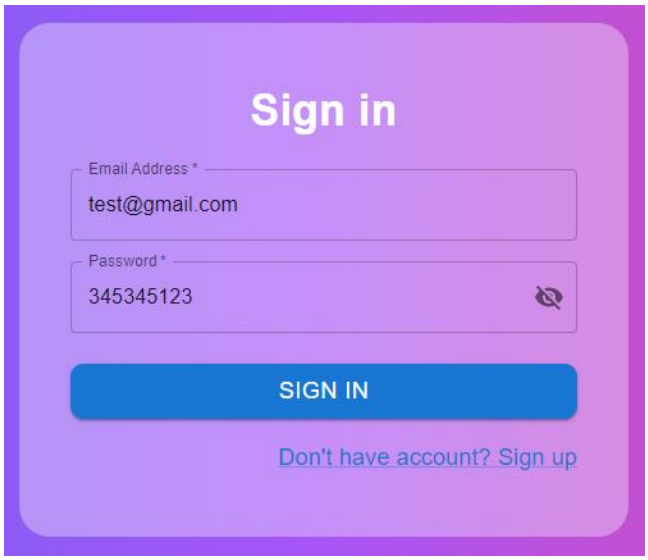


Рисунок 3.23 – Перегляд введеного паролю

Після реєстрації або авторизації, користувач опиняється на головній сторінці інтернет-магазину. На цій сторінці присутні такі елементи:

- Навігаційна панель – за допомогою якої, можна потрапити на інші сторінки вебдодатку, змінити тему на світлу або темну, вибір мови та відкрити

випадаюче меню з швидкими налаштуваннями;

- Блок фільтрації – дозволяє користувачу фільтрувати заклади за типом кухні;
- Список закладів – дозволяє побачити робочі години, назву, тип кухні, адресу закладу та номер телефону. При натисканні на карточку закладу, йде перенаправлення на сторінку закладу з доступними стравами.

Розроблені макети можна побачити на рисунках 3.24 – 3.26.

Рисунок 3.24 – Навігаційна панель

Рисунок 3.25 –Блок фільтрації

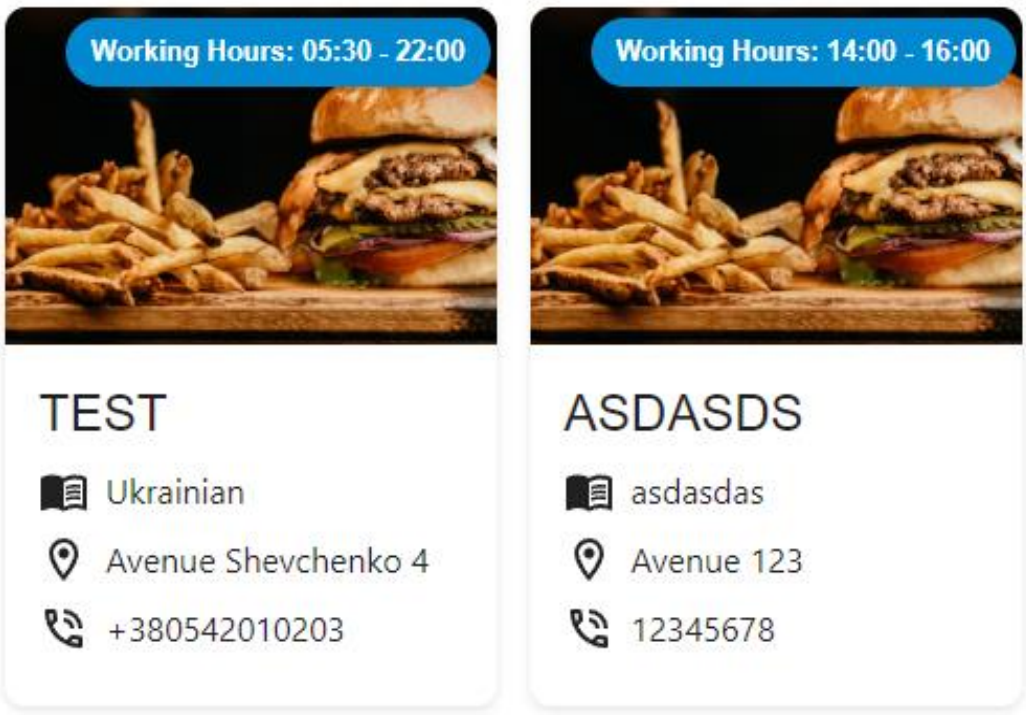


Рисунок 3.26 – Список закладів

При виборі будь-якого з доступних закладів, користувач потрапляє до сторінки ресторану, на якому можна побачити:

- Навігаційне меню;
- Назву закладу;
- Блок фільтрації – дозволяє користувачу фільтрувати страви за їхнім типом;
- Список страв з назвою, типом, інгредієнтами, ціною та можливістю додати до кошика;
- Блок відгуків з кількістю коментарів, середньою оцінкою закладу, можливістю опублікувати відгук або переглянути наявні відгуки.

Макет сторінки закладу можна побачити на рисунках 3.27 – 3.28.

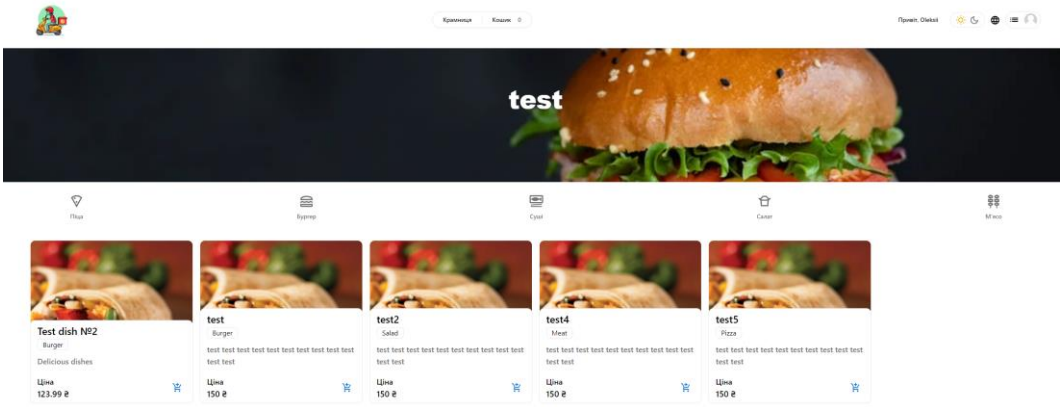


Рисунок 3.27 – Сторінка закладу

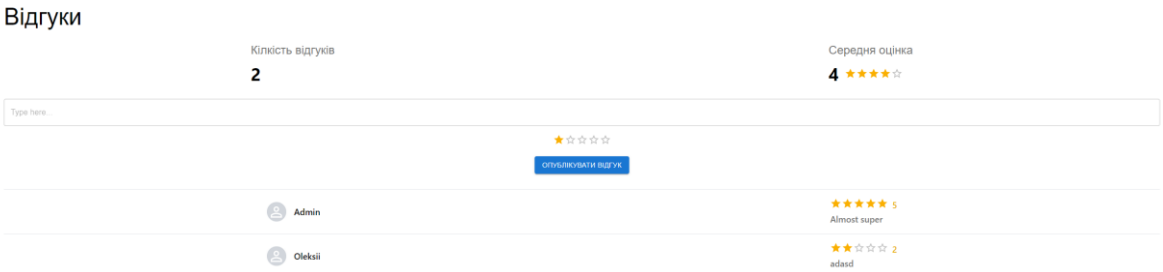


Рисунок 3.28 –Блок відгуків

Після того, як користувач обрав страви він потрапляє до сторінки кошика. На цій сторінці можна побачити список обраних страв з можливістю видалення з кошика та блок з ціною та кнопкою ‘Оформити замовлення’. Також, для

вдосконалення користувацького досвіду, є можливість перегляду кількості страв у навігаційній панелі. Сторінку кошика можна переглянути на рисунку 3.29.

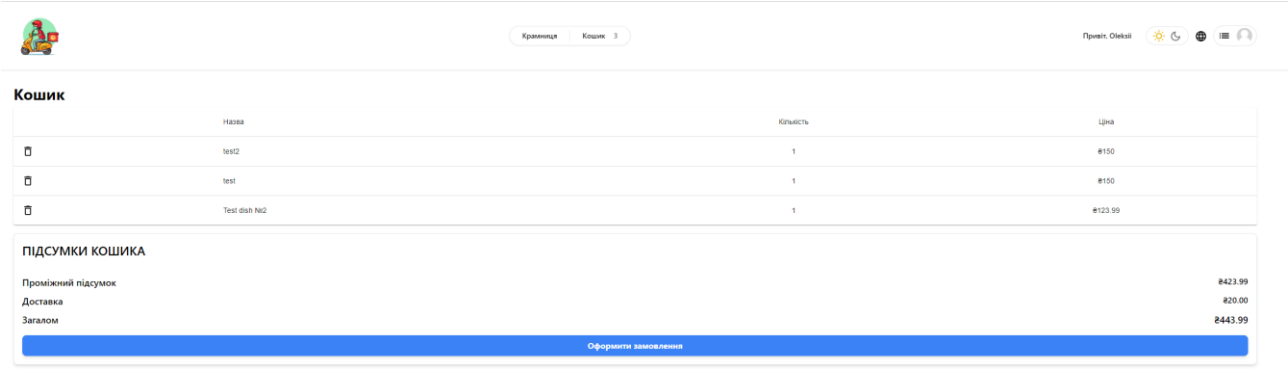


Рисунок 3.29 – Сторінка кошика

Для перегляду особистих даних, налаштувань загальних вподобань, деактивації акаунту, доступу до панелі адміністратора (тільки для користувача з роллю ADMIN), перегляду створених замовлень та доступних замовлень (для користувача з роллю ADMIN або DELIVERYMAN) було створено макет сторінки профілю, доступ до якого можливий через випадаюче меню у навігаційній панелі. Макет можна переглянути на рисунку 3.30.

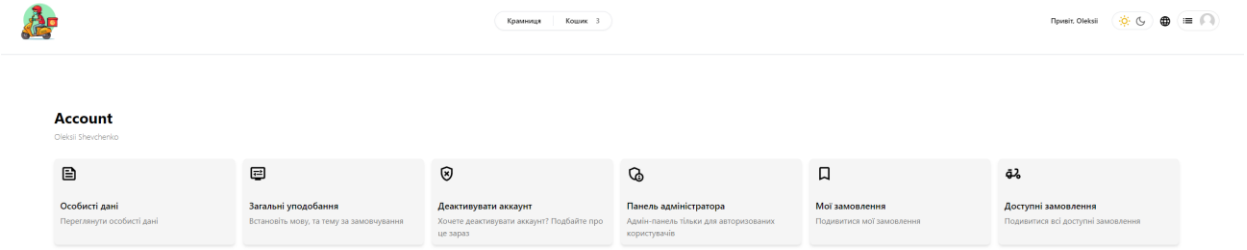


Рисунок 3.30 – Макет сторінки профілю

Кожна наявна карточка у списку переадресовує користувача на власну сторінку. Також, було вирішено, що демонструвати карточки ‘Панель адміністратора’ та ‘Доступні замовлення’ не є безпечно, тому користувач з роллю USER не бачить цих карток у списку, рисунок 3.31.

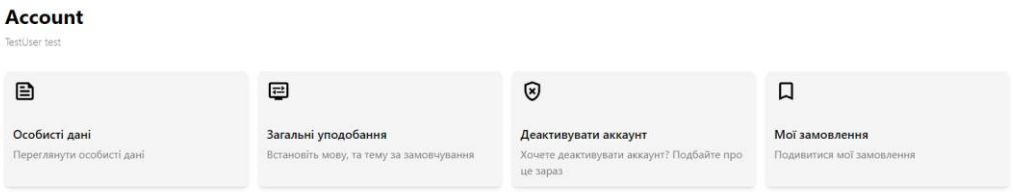


Рисунок 3.31 – Список налаштувань для користувача з роллю USER

Макети доступних сторінок наявні на рисунках 3.32 – 3.37.

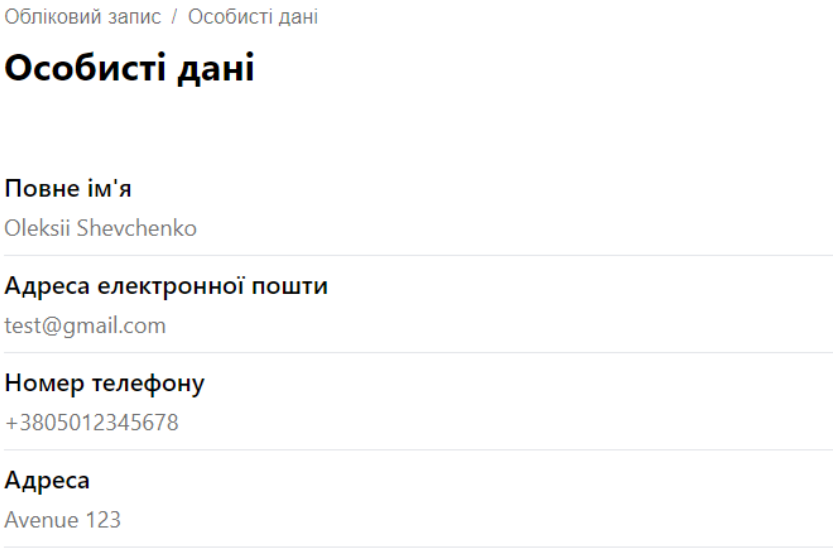


Рисунок 3.32 – Сторінка особистих даних

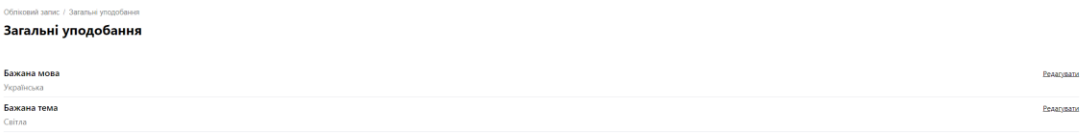


Рисунок 3.33 – Сторінка загальних вподобань

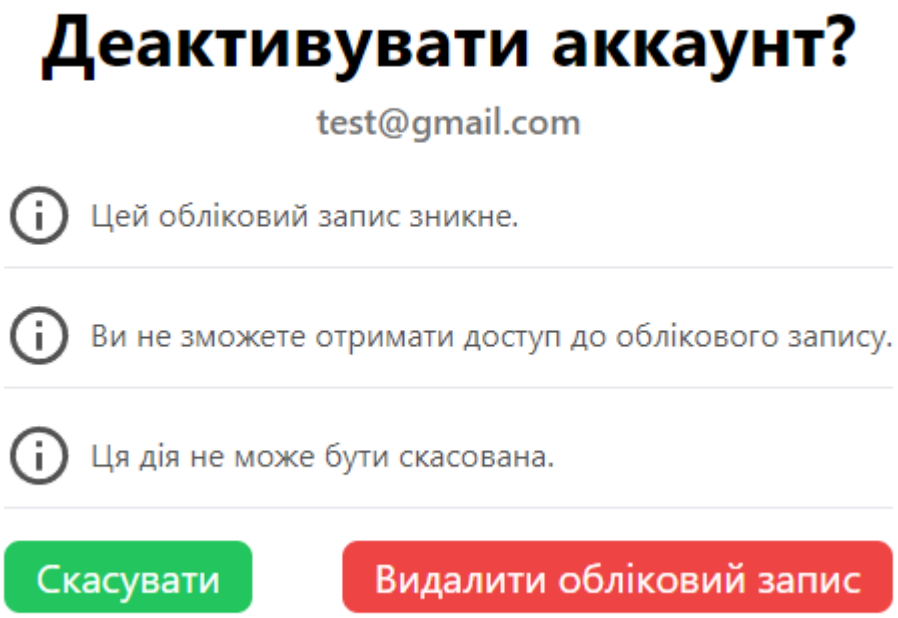


Рисунок 3.34 – Сторінка деактивації акаунту

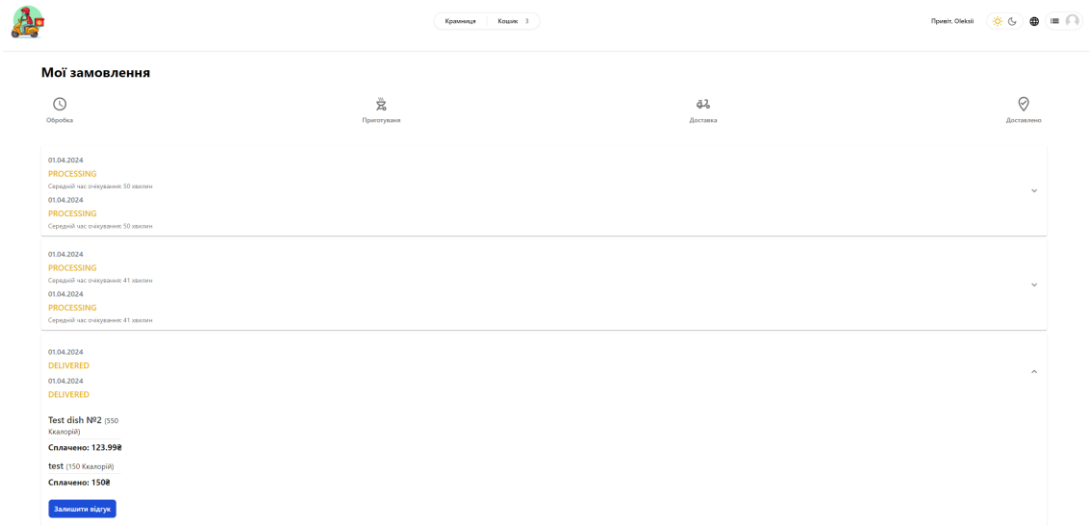


Рисунок 3.35 – Сторінка ‘Мої замовлення’

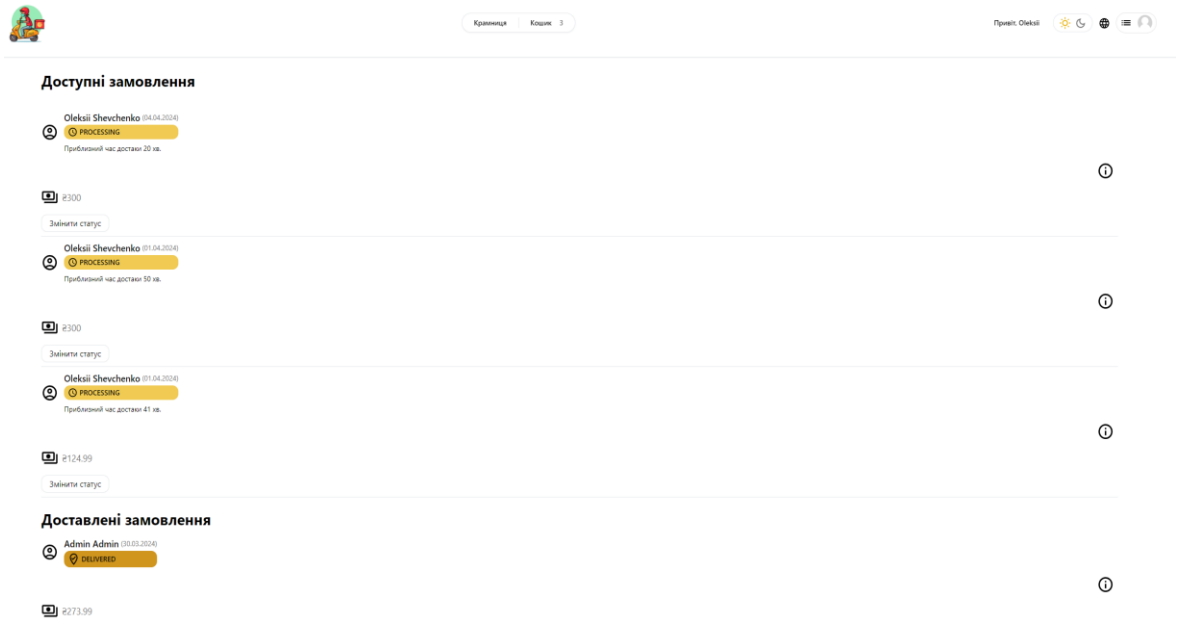


Рисунок 3.36 – Сторінка доступних замовлень

Сторінка адмін-панелі ділиться на 3 сторінки:

● Сторінка з користувачам – на цій сторінці є можливість подивитися список всіх користувачів та, деяку, інформацію про них ім’я, прізвище, адреса, номер телефону, роль. Адміністратор може видаляти, додавати користувачів, а також є можливість скачати таблицю у форматі csv;

● Сторінка з ресторанами - на цій сторінці є можливість подивитися список всіх ресторанів та інформацію про них назва, тип кухні, адреса, робочі години, номер телефону. Адміністратор може видаляти, додавати ресторани, а також є можливість скачати таблицю у форматі csv. При натискані на назву ресторану, адміністратор може побачити список меню з можливістю перегляду інформації, додавання або видалення страв;

● Сторінка загальної інформації – на цій сторінці можна переглянути загальну кількість користувачів, ресторанів, категорій та замовлень. Також, є можливість перегляду графіку кількості створених замовлень у конкретний день.

Макет можна побачити на рисунках 3.37 – 3.40.

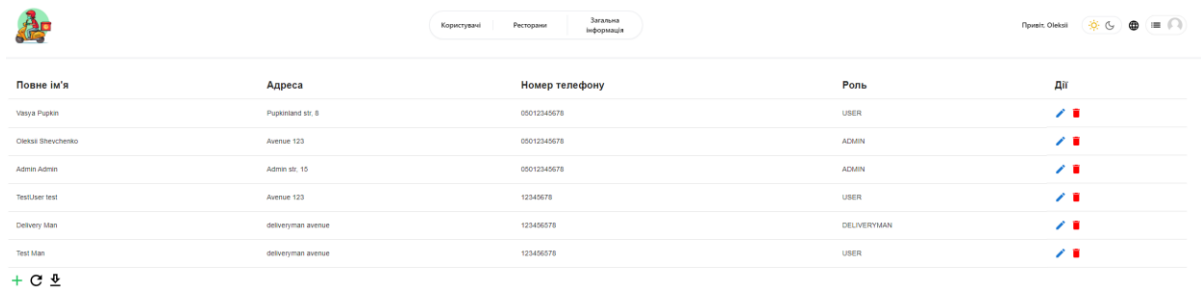


Рисунок 3.37 – Список користувачів

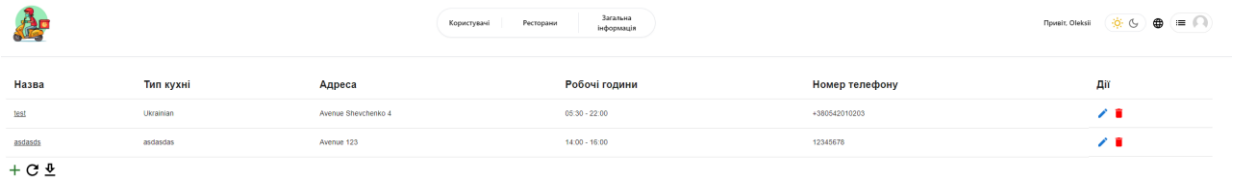


Рисунок 3.38 – Список ресторанів

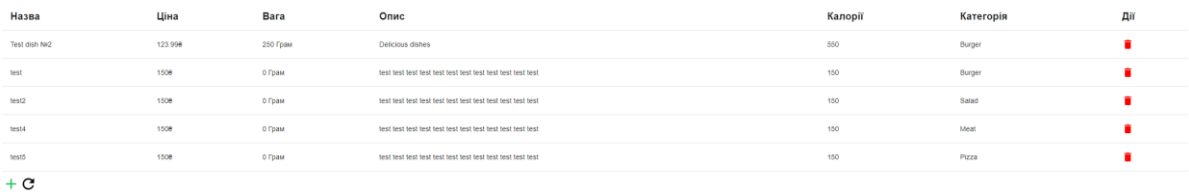


Рисунок 3.39 – Список страв закладу під назвою ‘test’

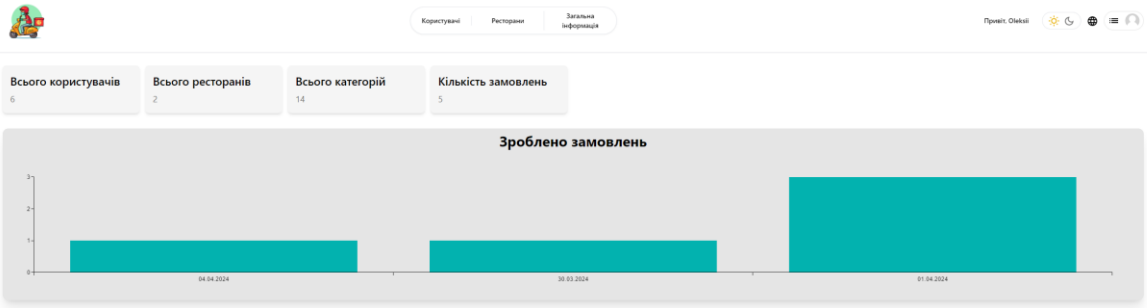


Рисунок 3.40 – Загальна інформація

Для перегляду детальної інформації про страву та вибору мови, було створено модальні вікна, які можна побачити у додатку Б.

Отже, створивши унікальний дизайн, який задовольняє базові потреба користувача та надає унікальний користувацький досвід, можна приступати до

написання програмного коду.

3.5 Розробка клієнтської частини

Перед початком розробки, треба визначитися з архітектурою вебдодатку, рисунок 3.41.

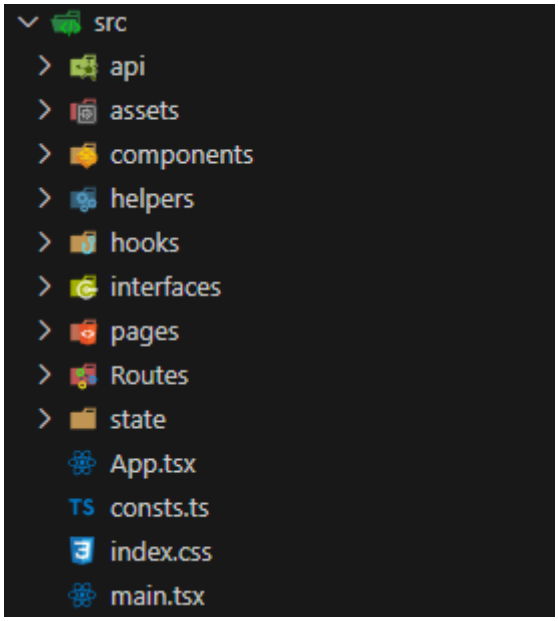


Рисунок 3.41 – Архітектура інтернет-магазину

- Папка api – містить запити до серверної частини, які не треба зберігати у глобальному стані;
- Папка assets – містить рисунки та ілюстрації;
- Папка components – містить будівельні блоки, які використовуються для побудови вебдодатку. Деякі з цих блоків можна використовувати на різних сторінка, а саме навігаційна панель, контейнер, повідомлення про помилку та скелетон вебсторінки;
- Папка helpers – містить допоміжні функції, які використовуються у різних частинах додатку;
- Папка hooks – містить власне створені хуки;
- Папка interfaces – містить інтерфейси, які задають статичну типізацію об’єктам, які були надіслані з серверної частини. Це дозволяє забезпечити безпеку та ефективність;

- Папка pages – містить сторінки інтернет-магазину;
- Папка routes – містить об’єкти, які комбінують у собі сторінки та маршрути до них;
- Папка state – зберігає глобальний стан, за допомогою якого можна звертатися до об'єкта, надісланого з серверної частини, у будь-якому місці додатка;
- Файл consts.ts – містить константні зміні. Наприклад, маршрути до сторінок або посилання для звернення до серверної частини.

Було вирішено, що користувач не зможе використовувати додаток без реєстрації або авторизації. Це рішення обумовлене тим, що користувач зможе робити замовлення, але не буде відомо ніякої інформації про користувача, а саме ім’я, номер телефону та адреса. Перше, що необхідно розробити – це форми авторизації та реєстрації. Для цього було створено файли Login.ts та Registration.ts у папці pages, додано маршрути у файлі consts.ts та додано об’єкт з маршрутом та сторінкою до папки routes, рисунки 3.42 – 3.44.

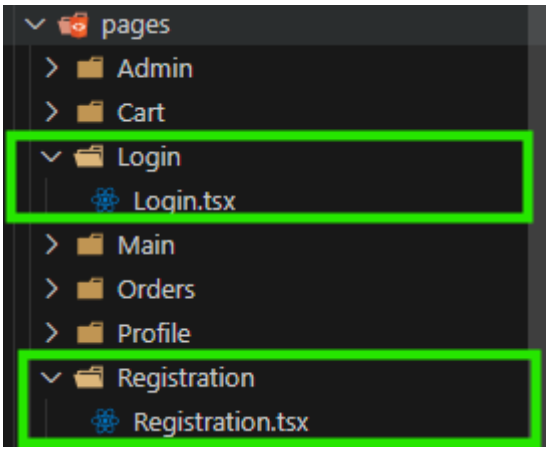


Рисунок 3.42 – Сторінки авторизації та реєстрації

```
2   export const DEFAULT_ROUTE: string = "/login";
3   export const REGISTRATION_ROUTE: string = "/registration";
```

Рисунок 3.43 – Маршрути до сторінок авторизації та реєстрації

```
export const publicRoutes: RouteItem[] = [
  {
    path: REGISTRATION_ROUTE,
    Component: Registration,
  },
  {
    path: DEFAULT_ROUTE,
    Component: Login,
  },
];
```

Рисунок 3.44 – Об’єкт загальнодоступних сторінок

Далі, було створено слайси для реєстрації та авторизації, які звертаються до серверної частини передаючи дані користувача, для реєстрації: ім’я, прізвище, адреса, номер телефону, пошта та пароль, для авторизації: пошта та пароль та зберігаються отримані дані з бази даних у глобальному стані. Для цього використовується об’єкт Promise, який використовується для асинхронних операцій. Promise - забезпечують надійний спосіб обробки результатів роботи асинхронного коду, будь то успішне завершення запиту на отримання даних або помилка при отриманні даних з API [17]. Асинхронні запити – це команди, які виконуються тривалий час [18] і для того, щоб не блокувати поток виконання коду, використовуються Promise. Стани Promise можна побачити на рисунку 3.45.

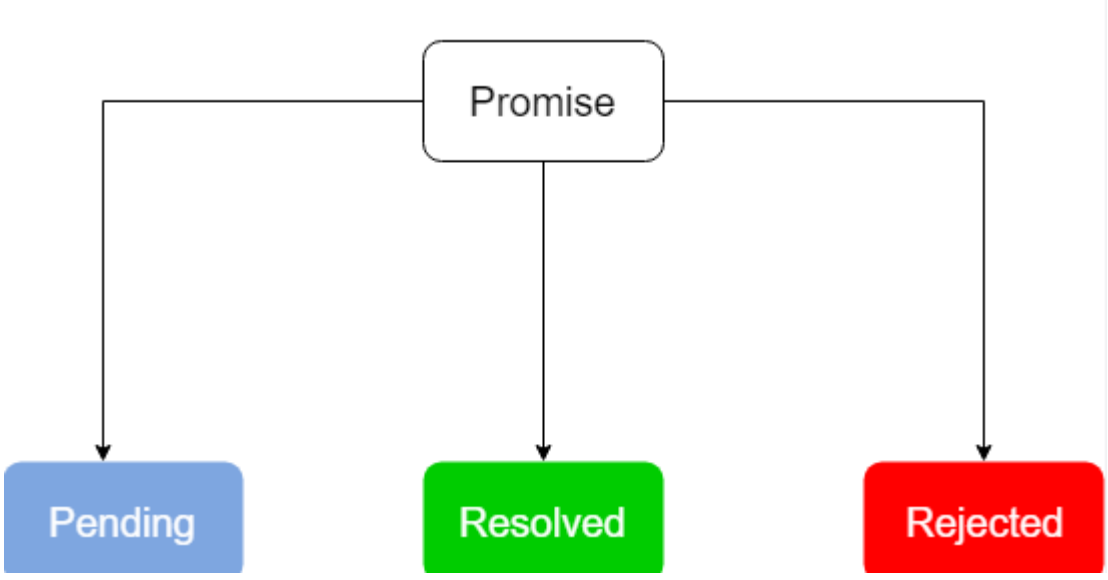


Рисунок 3.45 – Стани виконання Promise

- Pending - початковий стан, Promise не виконаний та не відхилений, тобто очкує виконання;
- Resolved – виконання успішно завершене;
- Rejected – операція виконана з помилкою.

На основі цієї інформації було створено реєстрацію та авторизацію. На початку робимо запит до серверної частини, а потім оброблюємо кожен зі станів. Якщо операція закінчена успішно, то зберігаємо дані у глобальному стані. У іншому випадку оброблюємо помилку та відображаємо користувачу, рисунки 3.46 – 3.47.

```
24  export const login = createAsyncThunk(
25    "auth/login",
26    async (credentials: { email: string; password: string }) => {
27      try {
28        const response = await instance.post<Token>(
29          "auth/login",
30          JSON.stringify(credentials),
31          {
32            headers: {
33              "Content-Type": "application/json",
34            },
35          }
36        );
37        return response.data;
38        // eslint-disable-next-line @typescript-eslint/no-explicit-any
39      } catch (error: any) {
40        if (
41          error.response &&
42          error.response.status >= 400 &&
43          error.response.status < 500
44        ) {
45          throw new Error(error.response?.data.message);
46        } else {
47          return error.response?.data.message || "Network error";
48        }
49      }
50    }
51  );
```

Рисунок 3.46 – Запит до API

```
61  const registerSlice = createSlice({
62    name: "auth",
63    initialState,
64    reducers: {},
65    extraReducers: (builder) => {
66      builder
67        .addCase(register.pending, (state) => {
68          state.status = "loading";
69          state.isAuthenticated = false;
70          state.error = null;
71        })
72        .addCase(register.fulfilled, (state, action) => {
73          state.status = "succeeded";
74          state.isAuthenticated = true;
75          state.access_token = action.payload.access_token;
76
77          sessionStorage.setItem("access_token", action.payload.access_token);
78        })
79        .addCase(register.rejected, (state, action) => {
80          state.status = "failed";
81          state.isAuthenticated = false;
82          state.error = action.error.message || "Network error";
83
84          sessionStorage.removeItem("access_token");
85        });
86    },
87  });
```

Рисунок 3.47 – Обробка станів Promise

Протестуємо авторизацію. При успішному виконані, API повинен повернути JWT токен авторизації, рисунок 3.48. При виникненні помилки, повідомляємо користувача, рисунок 3.49.

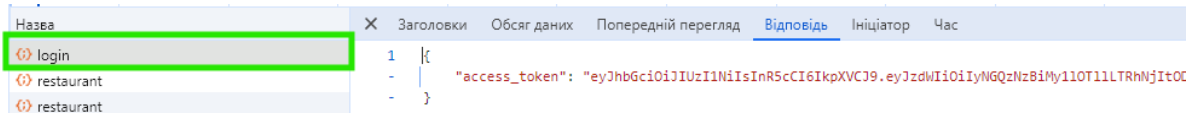


Рисунок 3.48 – Успішне виконання авторизації

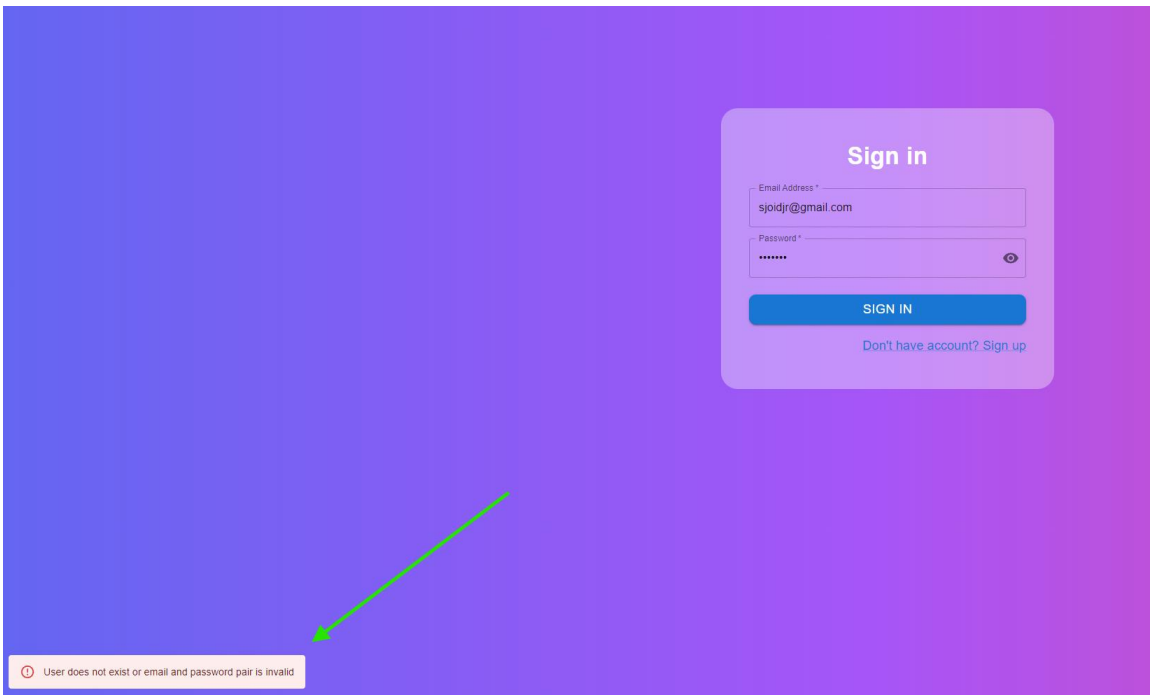


Рисунок 3.49 – Помилка при авторизації

За аналогією було зроблено всі інші запити, відрізняється лише дані які було надіслано та отримано.

Для забезпечення безпеки та надійності додатку було розроблено захищені маршрути. Захищені маршрути – це маршрути, які доступні лише авторизованим користувача. Зазвичай реалізовуються за допомогою механізмів авторизації, таких як JWT, сесійні токена та інше. Коли користувач намагається отримати доступ до захищеного маршруту, програма перевіряє, чи користувач авторизований. Якщо так, то йому надається доступ до маршруту. В іншому випадку користувача перенаправляються на сторінку входу.

Так як, у створеному інтернет-магазині з доставки їжі заборонено переглядати товари та взаємодіяти з налаштуваннями акаунту, потрібно захистити ці маршрути. Для цього було створено компонент, який при переході на захищений маршрут намагається отримати JWT токен з локального сховища. Якщо токен наявний, користувачу надається право переглядати вміст вебсайту. Якщо ні, то переадресовує користувача на сторінку логіну, рисунок 3.50. Локальне сховище, або localStorage – об’єкт, який дозволяє зберігати дані на локальну сховищі браузера [19].

```tsx
import { Navigate } from "react-router-dom";  5.2k (gzipped: 2.2k)
import { DEFAULT_ROUTE, HOME_ROUTE } from "../consts";

interface ProtectedRouteProps {
  children: React.ReactNode;
}

export const ProtectedRoute: React.FC<ProtectedRouteProps> = ({ children }) => {
  const token = sessionStorage.getItem("access_token");

  if (!token) {
    return <Navigate to={DEFAULT_ROUTE} />;
  }

  return children;
};
```

Рисунок 3.50 – Захищений маршрут

3.6 Тестування адаптивності

Найважливішим аспектом тестування адаптивності є оцінка того, як макет контенту вебсайту реагує на зміну розміру та орієнтацію екрану. Тестування має на меті зрозуміти, чи забезпечує читабельність і зручність використання вебдодатку на різних пристроях. Ефективна навігація має вирішальне значення для переміщення користувачів, незалежно від того, який пристрій вони використовують. Тестування адаптивності передбачає оцінку інтуїтивності та функціональності навігаційних меню, щоб вони залишалися доступними і простими у використанні як на великих екранах, так і на маленьких сенсорних екранах.

У тестуванні було використано вбудовані інструменти у браузер Google Chrome. Для переходу до панелі інструментів пристрою необхідно натиснути CTRL + SHIFT + M, рисунок 3.51.

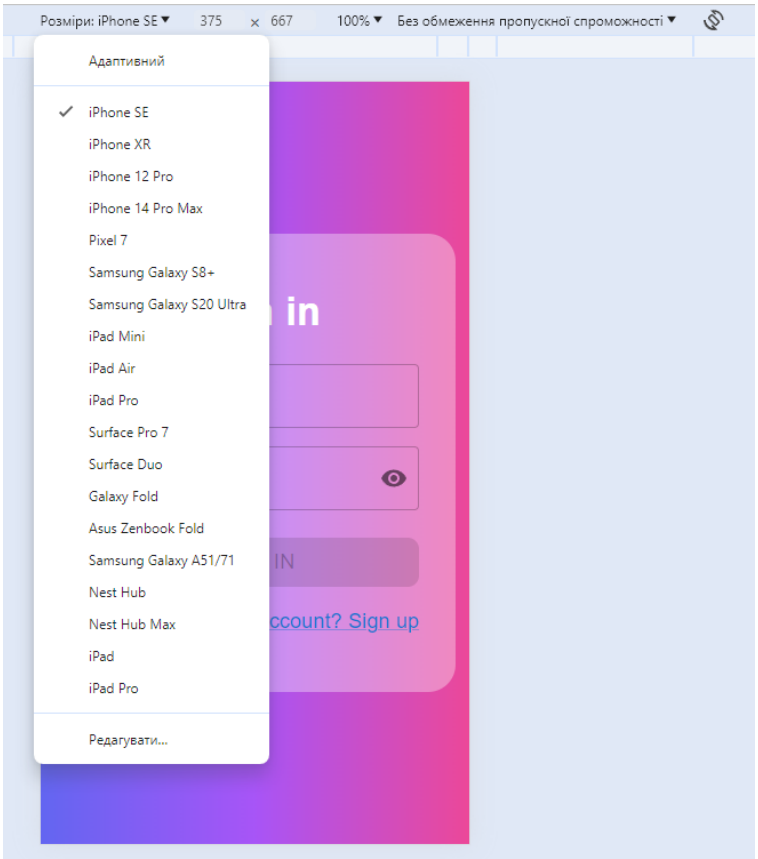


Рисунок 3.51 - Панель інструментів пристрою

На цій панелі можна обрати різні розміри екрана або зробити власний обравши пункт ‘Адаптивний’. Також можна обрати горизонтальну або вертикальну орієнтацію пристрою.

Для тестування було обрано пристрої: iPhone SE та iPad Pro, які мають роздільні здатності 375 x 667 та 1024 x 1366.

Тестування сторінок реєстрації та авторизації, рисунки 3.52 – 3.55.

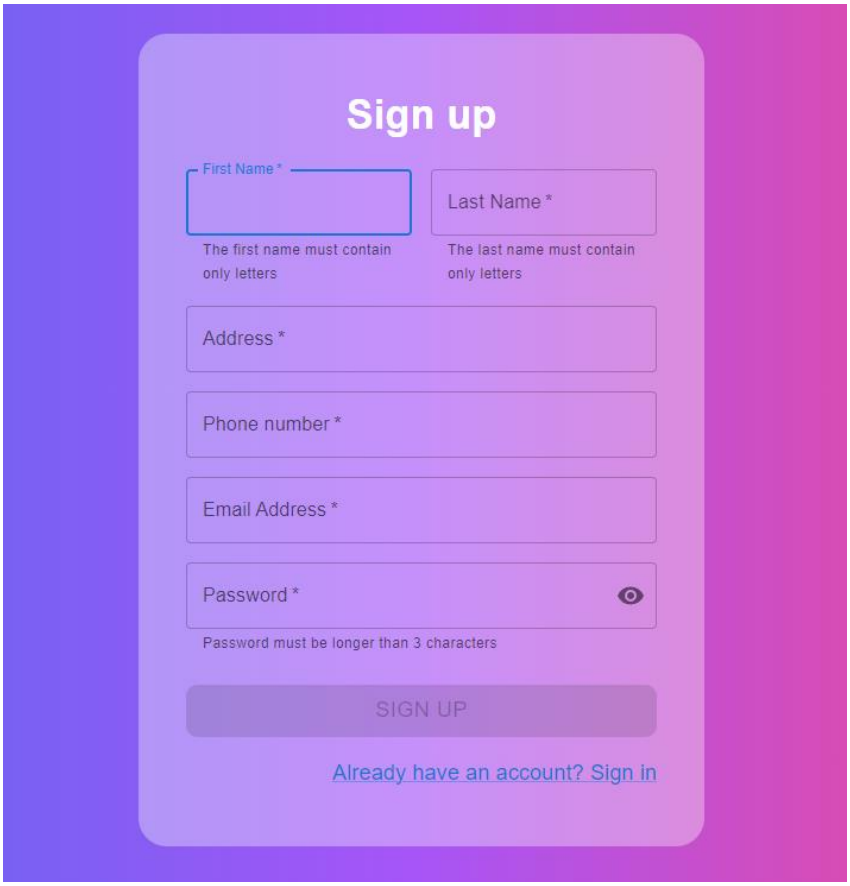


Рисунок 3.52 – Сторінка реєстрації на пристрої iPad Pro

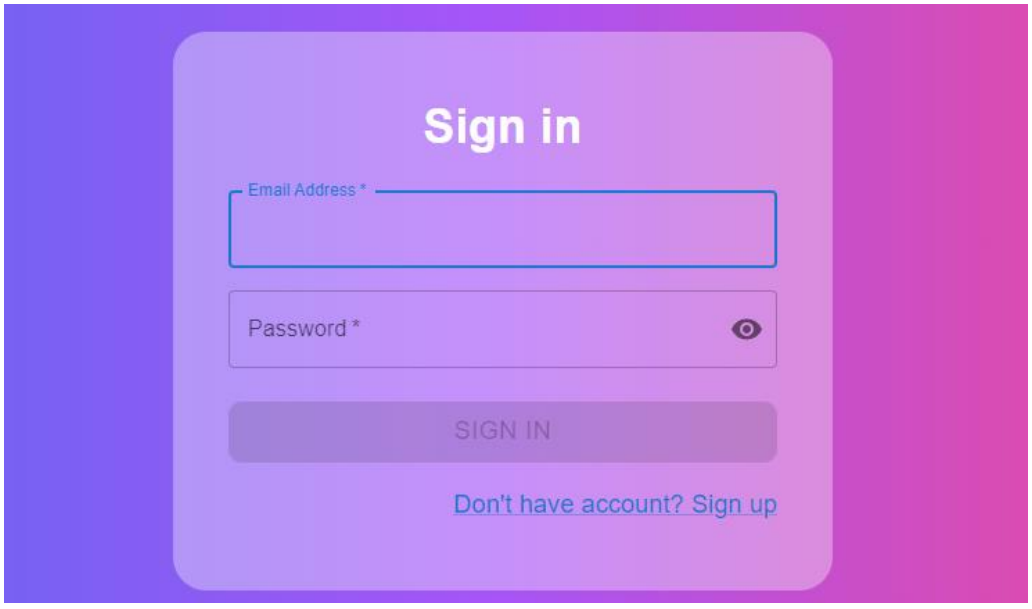


Рисунок 3.53 – Сторінка авторизації на пристрої iPad Pro

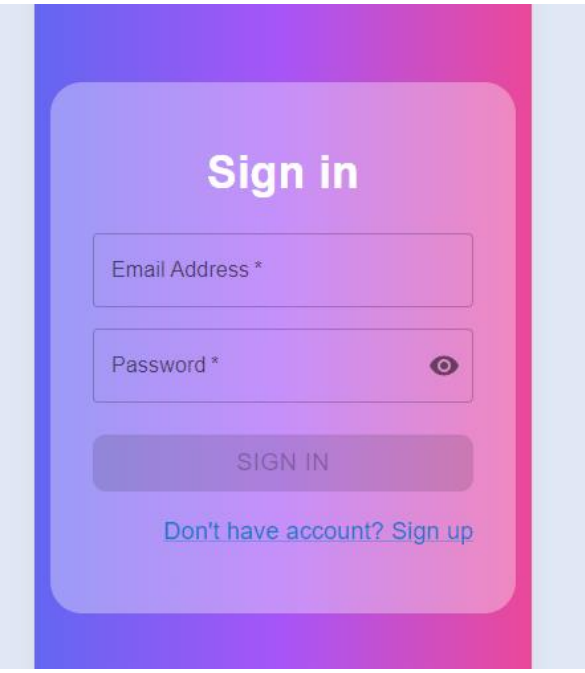


Рисунок 3.54 – Сторінка авторизації на пристрої iPhone SE

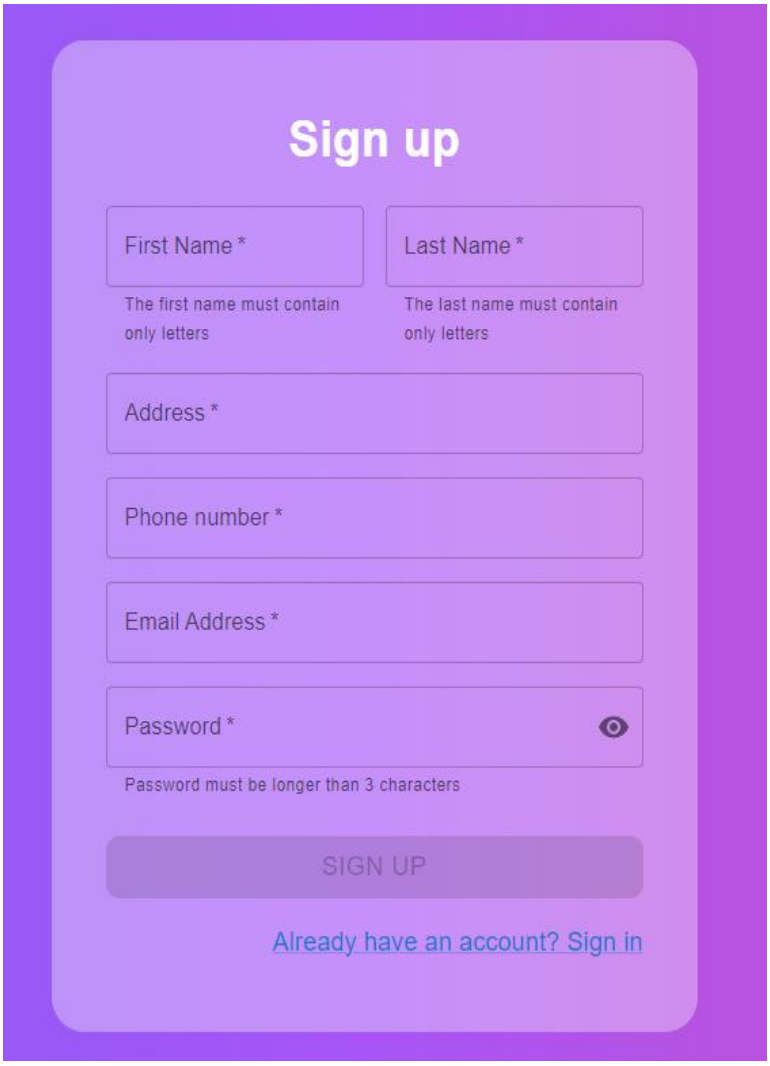


Рисунок 3.55 – Сторінка реєстрації на пристрої iPhone SE

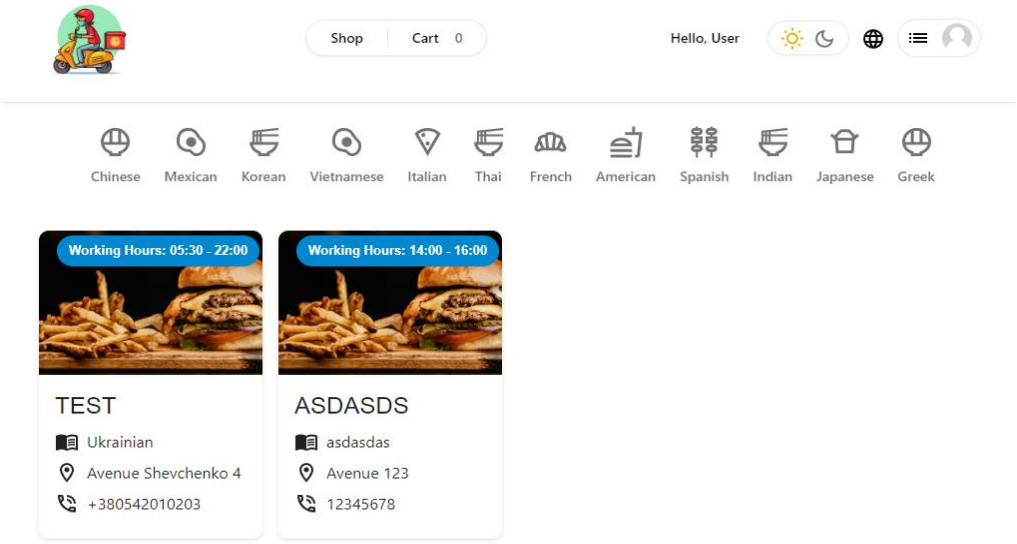


Рисунок 3.56 – Головна сторінка на пристрої iPad Pro

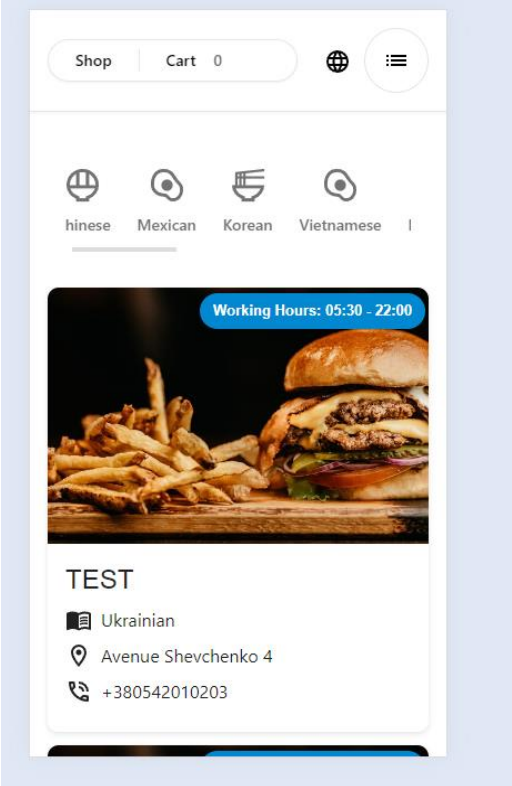


Рисунок 3.57 – Головна сторінка на пристрої iPhone SE

Як можна побачити на рисунках 3.52 – 3.57, адаптивний вебсайт було успішно розроблено та протестовано. Рисунки всіх сторінок інтернет-магазину можна побачити у додатку В.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи було розроблено та протестовано інтернет-магазин з доставки їжі. Було розглянуто різноманітні інструменти, зазначено їхні плюси та мінуси, обрано оптимальні та ефективні фреймворки для виконання поставленої мети. Також, під час розробки було визначено функціональні та нефункціональні вимоги до програмного продукту.

У ході виконання кваліфікаційної роботи було успішно виконано наступні завдання:

1. Виконано аналіз предметної області, визначено актуальність;
2. Проведено порівняння інструментів розробки;
3. Обрано оптимальні та ефективні інструменти для виконання поставленої мети;
4. Реалізовано серверну частину, яка задовольняє визначені функціональні та нефункціональні вимоги;
5. Проведено навантажувальне тестування серверної частини та тестування на працездатність;
6. Розроблено унікальний дизайн вебдодатку, який задовольняє базові потреби користувачів надаючи ефективний користувацький досвід;
7. Реалізовано клієнтську частину додатку, яка забезпечує адаптивність сторінок до різних розмірів екранів пристроїв;
8. Проведено тестування адаптивності;
9. Проаналізовано отримані результати та зроблено висновки.

Можна зробити висновок, що створений інтернет-магазин задовольняє потреби користувачів надаючи унікальний користувацький досвід, дозволяючи проводити реєстрацію, авторизацію, створювати замовлення з різних ресторанів, залишати відгук, а адміністраторам ефективно керувати вмістом вебдодатку.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Grand view research, inc. Market Research Reports & Consulting | Grand View Research, Inc. URL: https://www.grandviewresearch.com/ (дата звернення: 08.04.2024).

2. Desktop vs mobile vs tablet market share worldwide | statcounter global stats. StatCounter Global Stats. URL: https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet/worldwide (дата звернення: 08.04.2024).

3. Mobile-first indexing best practices | google search central | documentation | google for developers. Google for Developers. URL: https://developers.google.com/search/docs/crawling-indexing/mobile/mobile-sites-mobile-first-indexing (дата звернення: 09.04.2024).

4. Databases in web application development | ramotion branding agency. Web Design, UI/UX, Branding, and App Development Blog. URL: https://www.ramotion.com/blog/database-in-web-app-development/ (дата звернення: 09.04.2024).

5. Що таке API? Просте пояснення від Петра Газарова. dev.ua. URL: https://dev.ua/news/chto-takoe-api-prostym-yazykom (дата звернення: 09.04.2024).

6. Коди помилок HTTP та інші відповіді сервера | HOSTiQ Wiki. HOSTiQ Wiki. URL: https://hostiq.ua/wiki/ukr/http-status-codes/ (дата звернення: 10.04.2024).

7. Relational Vs. Non-Relational Databases | MongoDB. MongoDB. URL: https://www.mongodb.com/compare/relational-vs-non-relational-databases (дата звернення: 10.04.2024).

8. Що таке фреймворк - пояснюємо простими словами ⊳ вебстудія Brainlab. Brainlab. URL: https://brainlab.com.ua/uk/blog-uk/shho-take-frejmvork-poyasnyuyemo-prostymy-slovamy#title_1 (дата звернення: 11.04.2024).

9. GitHub - nestjs/nest: A progressive Node.js framework for building efficient, scalable, and enterprise-grade server-side applications with TypeScript/JavaScript. GitHub. URL: https://github.com/nestjs/nest (дата звернення: 11.04.2024)

10. Stack overflow. Stack Overflow Insights - Developer Hiring, Marketing, and User Research. URL: https://insights.stackoverflow.com/trends?tags=reactjs,angular,svelte,vuejs3 (дата звернення: 12.04.2024).

11. 20 best javascript frameworks for 2024 | lambdatest. LambdaTest. URL: https://www.lambdatest.com/blog/best-javascript-frameworks/ (дата звернення: 12.04.2024).

12. Contributors to Wikimedia projects. Tailwind CSS - wikipedia. Wikipedia, the free encyclopedia. URL: https://en.wikipedia.org/wiki/Tailwind_CSS (дата звернення: 12.04.2024).

13. Optimizing for production - tailwind CSS. Tailwind CSS - Rapidly build modern websites without ever leaving your HTML. URL: https://tailwindcss.com/docs/optimizing-for-production (дата звернення: 14.04.2024).

14. Учасники проєктів Вікімедіа. JSON web token – вікіпедія. Вікіпедія. URL: https://uk.wikipedia.org/wiki/JSON_Web_Token (дата звернення: 14.04.2024).

15. Знайомтесь, Postman - must have для QA. QualityAssuranceGroup. URL: https://qagroup.com.ua/publications/znajomtes-postman-must-have-dlia-qa/ (дата звернення: 15.04.2024).

16. Омельчук Є. Що таке Figma: функції, інструменти та переваги - академія Wezom. Академия Wezom - Обучаем IT технологиям с нуля. URL: https://wezom.academy/ua/chto-takoe-figma-funktsii-instrumenty-ipreimuschestva/ (дата звернення: 15.04.2024).

17. Mastering asynchronous code with react promise. DhiWise: App Development Platform for High Productivity. URL: https://www.dhiwise.com/post/how-to-handle-asynchronous-code-with-react-promise (дата звернення: 15.04.2024).

18. Community J. Асинхронність в JS. Medium. URL: https://medium.com/@jstify.community/асинхронність-в-js-2eba549b2d1c (дата

звернення: 16.04.2024).

19. LocalStorage - JavaScript довідка. JavaScript - JavaScript довідка. URL: http://xn--80adth0aefm3i.xn--j1amh/localstorage (дата звернення: 16.04.2024).

Додаток А

```
import { Prop, Schema, SchemaFactory } from '@nestjs/mongoose';
import mongoose, { HydratedDocument } from 'mongoose';
import { Review } from 'src/review/schemas/review.schema';

@Schema()
export class Restaurant {
  @Prop()
  id: string;

  @Prop()
  title: string;

  @Prop()
  cuisineType: string;

  @Prop()
  address: string;

  @Prop()
  openHours: string;

  @Prop()
  closeHours: string;

  @Prop()
  phoneNumber: string;

  @Prop({ type: mongoose.Schema.Types.Buffer })
  image?: Buffer;

  @Prop()
  imageMimeType?: string;

  averageRating?: number;

  reviews?: Review[];
}

export const RestaurantSchema = SchemaFactory.createForClass(Restaurant);

export type RestaurantDocument = HydratedDocument<Restaurant>;
```

```
import { Prop, Schema, SchemaFactory } from '@nestjs/mongoose';
import mongoose, { HydratedDocument } from 'mongoose';

@Schema()
export class Dish {
  @Prop()
  id: string;

  @Prop({ type: mongoose.Schema.Types.String, ref: 'restaurant' })
  restaurantId: string;

  @Prop()
  title: string;

  @Prop()
  weight: number;

  @Prop()
  description: string;

  @Prop([String])
  ingredients: string[];

  @Prop()
  calories: number;

  @Prop()
  category: string;

  @Prop()
  price: number;
}

export const DishSchema = SchemaFactory.createForClass(Dish);

export type DishDocument = HydratedDocument<Dish>;
```

```
import {
  Controller,
  Get,
  Post,
  Body,
  Param,
  Delete,
  Query,
  Put,
  UploadedFile,
  UseInterceptors,
  BadRequestException,
} from '@nestjs/common';
import { RestaurantService } from './restaurant.service';
import { CreateRestaurantDto } from './dto/create-restaurant.dto';
```

```
import { UpdateRestaurantDto } from './dto/update-restaurant.dto';
import { ApiOperation, ApiQuery, ApiResponse, ApiTags } from '@nestjs/swagger';
import { Auth } from 'src/shared/decorators/auth.dectorator';
import { Role } from 'src/auth/constants';
import { Restaurant } from './schemas/restaurant.schema';
import { SkipAuth } from 'src/shared/decorators/skip-auth.decorator';
import { FileInterceptor } from '@nestjs/platform-express';

const MAX_FILE_SIZE = 1024000;

@ApiTags('restaurant')
@Controller('restaurant')
export class RestaurantController {
  constructor(private readonly restaurantService: RestaurantService) {}

  @Post()
  @Auth(Role.admin)
  @ApiOperation({ summary: 'Create restaurant. For ADMIN role only' })
  @ApiResponse({
    status: 200,
    description: 'The created record',
    type: Restaurant,
  })
  create(
    @Body() createRestaurantDto: CreateRestaurantDto
  ): Promise<Restaurant> {
    return this.restaurantService.createRestaurant(createRestaurantDto);
  }

  @Get()
  @ApiOperation({ summary: 'Get restaurants list' })
  @ApiQuery({
    description: 'Optionalliy filters query params can be passed.',
    example: '?title=Sazha&address=Bandera',
  })
  @SkipAuth()
  @ApiResponse({
    status: 200,
    description: 'The found records',
    type: Array<Restaurant>,
  })
  findAll(@Query() filters?: any): Promise<Restaurant[]> {
    return this.restaurantService.getRestaurantList(filters);
  }

  @Get(':id')
  @ApiOperation({ summary: 'Get restaurant details by id' })
  @SkipAuth()
  @ApiResponse({
    status: 200,
    description: 'The found record',
    type: Restaurant,
  })
  findOne(@Param('id') id: string): Promise<Restaurant> {
```

```
    return this.restaurantService.getRestaurant(id);
  }

  @Put(':id')
  @Auth(Role.admin)
  @ApiOperation({
    summary: 'Updates restaurant details by id. For ADMIN role only',
  })
  @ApiResponse({
    status: 200,
    description: 'The updated record.',
    type: Restaurant,
  })
  update(
    @Param('id') id: string,
    @Body() updateRestaurantDto: UpdateRestaurantDto
  ): Promise<Restaurant> {
    return this.restaurantService.updateRestaurant(id, updateRestaurantDto);
  }

  @Delete(':id')
  @ApiOperation({
    summary: 'Delete restaurant details by id. For ADMIN role only',
  })
  @Auth(Role.admin)
  remove(@Param('id') id: string) {
    return this.restaurantService.removeRestaurant(id);
  }

  @UseInterceptors(FileInterceptor('image'))
  @Post('image/:id')
  @Auth(Role.admin)
  uploadFile(
    @Param('id') id: string,
    @UploadedFile() file: Express.Multer.File
  ) {
    if (file?.size > MAX_FILE_SIZE) {
      throw new BadRequestException({
        code: '0002',
        message: `Max file size of "${MAX_FILE_SIZE}" bytes exceeded`,
      });
    }
    if (!file?.buffer) {
      throw new BadRequestException({
        code: '0003',
        message: `Wrong file format`,
      });
    }
    this.restaurantService.updateRestaurant(id, {
      id,
      image: file.buffer,
      imageMimeType: file.mimetype,
    });
  }
```

```
}
```

```
import { Module, forwardRef } from '@nestjs/common';
import { RestaurantService } from './restaurant.service';
import { RestaurantController } from './restaurant.controller';
import { Restaurant, RestaurantSchema } from './schemas/restaurant.schema';
import { MongooseModule } from '@nestjs/mongoose';
import { ReviewModule } from 'src/review/review.module';
import { Dish, DishSchema } from './schemas/dish.schema';
import { DishController } from './dish/dish.controller';
import { DishService } from './dish/dish.service';

@Module({
  imports: [
    MongooseModule.forFeature([
      { name: Restaurant.name, schema: RestaurantSchema },
    ]),
    MongooseModule.forFeature([{ name: Dish.name, schema: DishSchema }]),
    forwardRef(() => ReviewModule),
  ],
  controllers: [RestaurantController, DishController],
  providers: [RestaurantService, DishService],
  exports: [RestaurantService, DishService],
})
export class RestaurantModule {}
```

```
import { Inject, Injectable, forwardRef } from '@nestjs/common';
import * as uuid from 'uuid';
import { CreateRestaurantDto } from './dto/create-restaurant.dto';
import { UpdateRestaurantDto } from './dto/update-restaurant.dto';
import { RepositoryAbstract } from 'src/shared/db-tools/repository-abstract';
import { Restaurant, RestaurantDocument } from './schemas/restaurant.schema';
import { FilterQuery, Model } from 'mongoose';
import { InjectModel } from '@nestjs/mongoose';
import { ReviewService } from 'src/review/review.service';

@Injectable()
export class RestaurantService extends RepositoryAbstract<
  Restaurant,
  RestaurantDocument
> {
  constructor(
    @InjectModel(Restaurant.name) restaurantModel: Model<RestaurantDocument>,
    @Inject(forwardRef(() => ReviewService))
    private reviewService: ReviewService
  ) {
    super(restaurantModel);
  }

  createRestaurant(
```

```
    createRestaurantDto: CreateRestaurantDto
  ): Promise<Restaurant> {
    return this.create({
      ...createRestaurantDto,
      id: uuid.v4(),
    });
  }

  getRestaurantList(filters?: FilterQuery<any>): Promise<Restaurant[]> {
    return this.findMultiple(filters);
  }

  async getRestaurant(id: string): Promise<Restaurant> {
    const restaurant = await this.findOne({ id });
    if (restaurant) {
      const reviews = await this.reviewService.findAllReviews({
        restaurantId: id,
      });
      let rating = 0;
      restaurant.reviews = [...reviews];
      reviews.forEach((review) => (rating += review.rating));
      restaurant.averageRating = Number((rating / reviews.length).toFixed(2));
    }
    return restaurant;
  }

  updateRestaurant(
    id: string,
    updateRestaurantDto: UpdateRestaurantDto
  ): Promise<Restaurant> {
    const data = { ...updateRestaurantDto };
    delete data.id;
    return this.update({ id }, data);
  }

  async removeRestaurant(id: string): Promise<unknown> {
    await this.reviewService.removeReview(id);
    return this.remove({ id });
  }
}
```

```
import {
  Body,
  Controller,
  Delete,
  Get,
  Param,
  Post,
  Put,
} from '@nestjs/common';
import { ApiOperation, ApiParam, ApiResponse, ApiTags } from '@nestjs/swagger';
import { DishService } from './dish.service';
```

```
import { SkipAuth } from 'src/shared/decorators/skip-auth.decorator';
import { CreateDishDto } from '../dto/create-dish.dto';
import { Auth } from 'src/shared/decorators/auth.dectorator';
import { Role } from 'src/auth/constants';
import { UpdateDishDto } from '../dto/update-dish.dto';
import { Dish } from '../schemas/dish.schema';

@ApiTags('restaurant')
@Controller('restaurant/:restaurantId/dish')
export class DishController {
  constructor(private readonly dishService: DishService) {}

  @Get('')
  @SkipAuth()
  @ApiOperation({ summary: 'Get dishes list for the restaurant' })
  @ApiParam({
    name: 'restaurantId',
    type: String,
    description: 'Restaurant ID',
  })
  @ApiResponse({
    status: 200,
    type: Dish,
    isArray: true,
  })
  public getDishes(
    @Param('restaurantId') restaurantId: string
  ): Promise<Dish[]> {
    return this.dishService.getAllDishes(restaurantId);
  }

  @Get(':id')
  @SkipAuth()
  @ApiOperation({
    summary: 'Get dish data by Dish ID',
  })
  @ApiResponse({
    status: 200,
    type: Dish,
    description: 'Dish information',
  })
  @ApiParam({
    name: 'restaurantId',
    type: String,
    description: 'Restaurant ID',
  })
  @ApiParam({
    name: 'id',
    type: String,
    description: 'Dish ID',
  })
  public getDish(@Param('id') dishId: string): Promise<Dish> {
    return this.dishService.getDishById(dishId);
  }
```

```
@Post()
@Auth(Role.admin)
@ApiOperation({
  summary:
    'Add dish data for the provided restaurant. Available for ADMIN only',
})
@ApiResponse({
  status: 200,
  type: Dish,
  description: 'Dish information',
})
@ApiResponse({
  status: 404,
  description: 'Restaurant with provided restaurantId was not found',
})
public addDish(
  @Param('restaurantId') restaurantId: string,
  @Body() dishParams: CreateDishDto
): Promise<Dish> {
  return this.dishService.createDish(restaurantId, dishParams);
}

@Put(':id')
@Auth(Role.admin)
@ApiOperation({
  summary:
    'Updates dish data for the provided restaurant. Available for ADMIN only',
})
@ApiResponse({
  status: 200,
  type: Dish,
  description: 'Dish information',
})
@ApiResponse({
  status: 404,
  description: 'Restaurant with provided restaurantId was not found',
})
public updateDish(
  @Param('restaurantId') restaurantId: string,
  @Param('id') dishId: string,
  @Body() dishParams: UpdateDishDto
): Promise<Dish> {
  return this.dishService.updateDish(restaurantId, dishId, dishParams);
}

@Delete(':id')
@Auth(Role.admin)
@ApiOperation({
  summary:
    'Deletes dish data for the provided restaurant. Available for ADMIN only',
})
@ApiResponse({
  status: 200,
```

```
  })
  @ApiResponse({
    status: 404,
    description: 'Restaurant with provided restaurantId was not found',
  })
  public deleteDish(
    @Param('restaurantId') restaurantId: string,
    @Param('id') dishId: string
  ): Promise<any> {
    return this.dishService.deleteDish(restaurantId, dishId);
  }
}
```

```
import { Injectable, NotFoundException } from '@nestjs/common';
import { InjectModel } from '@nestjs/mongoose';
import { Model } from 'mongoose';
import * as uuid from 'uuid';
import { RepositoryAbstract } from 'src/shared/db-tools/repository-abstract';
import { RestaurantService } from '../restaurant.service';
import { Dish, DishDocument } from '../schemas/dish.schema';
import { CreateDishDto } from '../dto/create-dish.dto';
import { UpdateDishDto } from '../dto/update-dish.dto';

@Injectable()
export class DishService extends RepositoryAbstract<Dish, DishDocument> {
  constructor(
    @InjectModel(Dish.name) dishModel: Model<DishDocument>,
    private restaurantService: RestaurantService
  ) {
    super(dishModel);
  }

  private async checkIfRestaurantExists(id: string): Promise<boolean> {
    const restaurant = await this.restaurantService.getRestaurant(id);
    if (!restaurant) {
      throw new NotFoundException(`Restaurant with ID ${id} not found`);
    }
    return true;
  }

  public getAllDishes(restaurantId: string): Promise<Dish[]> {
    return this.findMultiple({ restaurantId });
  }

  public getDishById(id: string): Promise<Dish> {
    return this.findOne({ id });
  }

  public async createDish(
    restaurantId: string,
    dishData: CreateDishDto
  ): Promise<Dish> {
```

```
    await this.checkIfRestaurantExists(restaurantId);
    return this.create({
      id: uuid.v4(),
      restaurantId,
      title: dishData.title,
      weight: dishData.weight,
      calories: dishData.calories,
      category: dishData.category,
      description: dishData.description,
      ingredients: dishData.ingredients,
      price: dishData.price,
    });
  }

  public async updateDish(
    restaurantId: string,
    dishId: string,
    dishData: UpdateDishDto
  ): Promise<Dish> {
    await this.checkIfRestaurantExists(restaurantId);
    delete dishData['id'];
    delete dishData['restaurantId'];
    return this.update(
      {
        restaurantId,
        id: dishId,
      },
      dishData
    );
  }

  public async deleteDish(restaurantId: string, dishId: string): Promise<any> {
    await this.checkIfRestaurantExists(restaurantId);
    return this.remove({ restaurantId, id: dishId });
  }

  public getDishesById(dishesId: string[]): Promise<Dish[]> {
    return this.dataModel.find({ id: dishesId });
  }
}
```

```
import { Prop, Schema, SchemaFactory } from '@nestjs/mongoose';
import mongoose, { HydratedDocument } from 'mongoose';

@Schema()
export class Review {
  @Prop({ type: mongoose.Schema.Types.String, ref: 'User' })
  userId: string;

  @Prop({ type: mongoose.Schema.Types.String, ref: 'Restaurant' })
  restaurantId: string;
```

```
  @Prop()
  rating: number;

  @Prop()
  comment: string;
}

export const ReviewSchema = SchemaFactory.createForClass(Review);

export type ReviewDocument = HydratedDocument<Review>;
```

```
import {
  Controller,
  Get,
  Post,
  Body,
  Param,
  Delete,
  Put,
  Request,
  Query,
} from '@nestjs/common';
import { ReviewService } from './review.service';
import { CreateReviewDto } from './dto/create-review.dto';
import { UpdateReviewDto } from './dto/update-review.dto';
import { SkipAuth } from 'src/shared/decorators/skip-auth.decorator';
import { Role } from 'src/auth/constants';
import { ReviewResponseDto } from './dto/review-response.dto';
import { Review } from './schemas/review.schema';
import {
  ApiOperation,
  ApiParam,
  ApiQuery,
  ApiResponse,
  ApiTags,
} from '@nestjs/swagger';
import { Auth } from 'src/shared/decorators/auth.dectorator';

@ApiTags('review')
@Controller('review')
export class ReviewController {
  constructor(private readonly reviewService: ReviewService) {}

  @ApiOperation({
    summary: 'Creates or updates review record for specified restaurant',
  })
  @ApiResponse({
    status: 200,
    description: 'The created record',
    type: Review,
  })
  @ApiResponse({
```

```
    status: 404,
    description:
      'Not found exception. The provided restaurantId has not been found.',
  })
  @Post()
  create(@Body() createReviewDto: CreateReviewDto, @Request() req) {
    return this.reviewService.createReview({
      ...createReviewDto,
      userId: req.user.id,
    });
  }

  @Get()
  @SkipAuth()
  @ApiOperation({ summary: 'Get the list of review records' })
  @ApiResponse({
    status: 200,
    description: 'The list of records',
    type: ReviewResponseDto,
  })
  @ApiQuery({
    name: 'restaurantId',
    required: true,
    type: String,
  })
  findAll(
    @Query('restaurantId') restaurantId: string
  ): Promise<ReviewResponseDto[]> {
    return this.reviewService.findAllReviews({ restaurantId });
  }

  @Put()
  @ApiOperation({ summary: 'Updates specified review record' })
  @ApiResponse({
    status: 200,
    description: 'The updated review',
    type: Review,
  })
  @Auth(Role.admin, Role.user)
  update(@Body() updateReviewDto: UpdateReviewDto, @Request() req) {
    const role = req.user?.role;
    const userId = req.user?.id;

    return this.reviewService.updateReview({
      ...updateReviewDto,
      userId: role === Role.admin ? updateReviewDto.userId || userId : userId,
    });
  }

  @Delete(':restaurantId')
  @ApiOperation({ summary: 'Deletes specific review record' })
  @ApiResponse({
    status: 200,
    description: 'The deleted review',
```

```
    type: Review,
  })
  @ApiParam({
    name: 'restaurantId',
    type: String,
  })
  @Auth(Role.admin, Role.user)
  remove(@Param('restaurantId') restaurantId: string, @Request() req) {
    return this.reviewService.removeReview(req.user?.id, restaurantId);
  }
}
```

```
import { Module, forwardRef } from '@nestjs/common';
import { ReviewService } from './review.service';
import { ReviewController } from './review.controller';
import { MongooseModule } from '@nestjs/mongoose';
import { Review, ReviewSchema } from './schemas/review.schema';
import { RestaurantModule } from 'src/restaurant/restaurant.module';

@Module({
  imports: [
    MongooseModule.forFeature([{ name: Review.name, schema: ReviewSchema }]),
    forwardRef(() => RestaurantModule),
  ],
  controllers: [ReviewController],
  providers: [ReviewService],
  exports: [ReviewService],
})
export class ReviewModule {}
```

```
import {
  Inject,
  Injectable,
  NotFoundException,
  forwardRef,
} from '@nestjs/common';
import { UpdateReviewDto } from './dto/update-review.dto';
import { RepositoryAbstract } from 'src/shared/db-tools/repository-abstract';
import { Review, ReviewDocument } from './schemas/review.schema';
import { InjectModel } from '@nestjs/mongoose';
import { Model } from 'mongoose';
import { RestaurantService } from 'src/restaurant/restaurant.service';
import { ReviewResponseDto } from './dto/review-response.dto';

@Injectable()
export class ReviewService extends RepositoryAbstract<Review, ReviewDocument> {
  constructor(
    @InjectModel(Review.name) private reviewModel: Model<ReviewDocument>,
    @Inject(forwardRef(() => RestaurantService))
    private restaurantService: RestaurantService
```

```
  ) {
    super(reviewModel);
  }

  private async checkIfRestaurantExists(id: string): Promise<boolean> {
    const restaurant = await this.restaurantService.getRestaurant(id);
    if (!restaurant) {
      throw new NotFoundException('Restaurant with provided id was not found');
    }
    return true;
  }

  private async checkIfCommentExists(
    restaurantId: string,
    userId: string
  ): Promise<boolean> {
    const comment = await this.findOne({ restaurantId, userId });
    if (comment) {
      return true;
    }
    return false;
  }

  async createReview(createReviewData: Review): Promise<Review> {
    await this.checkIfRestaurantExists(createReviewData.restaurantId);
    const commentExists = await this.checkIfCommentExists(
      createReviewData.restaurantId,
      createReviewData.userId
    );
    if (commentExists) {
      return this.updateReview({ ...createReviewData });
    }
    return this.create(createReviewData);
  }

  findAllReviews(filters?: any): Promise<ReviewResponseDto[]> {
    return this.reviewModel.aggregate([
      {
        $match: {
          ...filters,
        },
      },
      {
        $lookup: {
          from: 'users',
          localField: 'userId',
          foreignField: 'id',
          as: 'userData',
        },
      },
      {
        $project: {
          restaurantId: 1,
          userId: 1,
```

```
          rating: 1,
          comment: 1,
          userFirstName: '$userData.firstName',
          userLastName: '$userData.lastName',
        },
      },
      {
        $unwind: '$userFirstName',
      },
      {
        $unwind: '$userLastName',
      },
    ]);
  }

  async updateReview(updateReviewDto: UpdateReviewDto): Promise<Review> {
    await this.checkIfRestaurantExists(updateReviewDto.restaurantId);
    return this.update(
      {
        userId: updateReviewDto.userId,
        restaurantId: updateReviewDto.restaurantId,
      },
      {
        comment: updateReviewDto.comment,
        rating: updateReviewDto.rating,
      }
    );
  }

  removeReview(restaurantId: string, userId?: string): Promise<any> {
    let params = {};
    if (userId) {
      params = { restaurantId, userId };
    } else {
      params = { restaurantId };
    }
    return this.remove(params);
  }
}
```

```
import { Prop, Schema, SchemaFactory } from '@nestjs/mongoose';
import mongoose, { HydratedDocument } from 'mongoose';
import { OrderStatus } from '../shared/constants';

@Schema()
export class Order {
  @Prop()
  orderId: string;

  @Prop({ type: mongoose.Schema.Types.String, ref: 'dish' })
  dishId: string;
```

```
  @Prop({ type: mongoose.Schema.Types.String, ref: 'user' })
  userId: string;

  @Prop()
  status: OrderStatus;

  @Prop()
  averageWaitingTimeMinutes: number;

  @Prop()
  orderTimestamp: number;
}

export const OrderSchema = SchemaFactory.createForClass(Order);

export type OrderDocument = HydratedDocument<Order>;
```

```
import {
  Body,
  Controller,
  Get,
  Param,
  Post,
  Put,
  Query,
  Request,
} from '@nestjs/common';
import { OrderService } from './order.service';
import { Auth } from 'src/shared/decorators/auth.dectorator';
import { Role } from 'src/auth/constants';
import { CreateOrderDto } from './dto/create-order.dto';
import { OrderStatus } from './shared/constants';
import { OrderResponseDto } from './dto/order-response.dto';
import { UpdateOrderStatusDto } from './dto/update-order-status.dto';
import { ApiOkResponse, ApiOperation, ApiTags } from '@nestjs/swagger';
import { UserHistoryResponseDto } from './dto/user-history-response.dto';

@Controller('order')
@ApiTags('order')
export class OrderController {
  constructor(private readonly orderService: OrderService) {}

  @Post('create-order')
  @Auth(Role.user, Role.admin)
  @ApiOperation({
    summary: 'Creates order for the entier user with provided dishes IDs',
    description: 'Available for users with role USER and ADMIN only',
  })
  @ApiOkResponse({
    description: 'Order records',
    type: OrderResponseDto,
    isArray: true,
```

```
  })
  public createOrder(
    @Body() createOrderDto: CreateOrderDto,
    @Request() req
  ): Promise<OrderResponseDto[]> {
    const userId = req.user.id;
    return this.orderService.makeOrder(createOrderDto.dishesId, userId);
  }

  @Get(':id')
  @Auth(Role.user, Role.admin, Role.deliveryMan)
  @ApiOperation({
    summary: 'Get order detailed info by provided order ID',
    description:
      'Available for users with role DELIVERYMAN and ADMIN only. Additionally
"status" query param can be provided. Example: "order/abc3213-3231-2323za-
3231hg?status=COOKING"',
  })
  @ApiOkResponse({
    description: 'Order records',
    type: OrderResponseDto,
    isArray: true,
  })
  public getOrderData(
    @Param('id') orderId: string,
    @Query('status') status?: OrderStatus
  ): Promise<OrderResponseDto[]> {
    return this.orderService.getOrderDetails(orderId, status);
  }

  @Put(':id/status')
  @Auth(Role.admin, Role.deliveryMan)
  @ApiOperation({
    summary: 'Updates order status by provided Order ID',
    description: 'Available for users with role USER and DELIVERYMAN only',
  })
  @ApiOkResponse({
    description: 'Order records',
    type: OrderResponseDto,
    isArray: true,
  })
  public updateStatus(
    @Param('id') orderId: string,
    @Body() updateOrderStatusDto: UpdateOrderStatusDto
  ) {
    return this.orderService.updateOrderStatus(
      orderId,
      updateOrderStatusDto.status
    );
  }

  @Get('')
  @Auth(Role.admin, Role.deliveryMan)
  @ApiOperation({
```

```
    summary:
      'Get list of unique orders. Additionally it can be filtered by status',
    description:
      'Available for users with role DELIVERYMAN and ADMIN only. Additionally
"status" query param can be provided. Example: "order?status=PROCESSING"',
  })
  @ApiOkResponse({
    description: 'Order records',
    type: OrderResponseDto,
    isArray: true,
  })
  public getOrderList(@Query('status') status?: OrderStatus) {
    return this.orderService.getOrderList(status);
  }

  @Get('history/:userId')
  @Auth(Role.admin, Role.user)
  @ApiOperation({
    summary:
      'Get list of user orders. Additionally it can be filtered by status',
    description:
      'Available for users with role USER and ADMIN only. Additionally "status"
query param can be provided. Example: "order/history/312iuy3-231kjh3-
2312?status=PROCESSING"',
  })
  @ApiOkResponse({
    description: 'Order history records',
    type: UserHistoryResponseDto,
    isArray: true,
  })
  public getOrderHistoryForUser(
    @Param('userId') userId: string,
    @Query('status') status: OrderStatus,
    @Request() req
  ) {
    let getUserId = userId;
    if (req.user.role === Role.user) {
      getUserId = req.user.id;
    }
    return this.orderService.getUserOrders(getUserId, status);
  }
}
```

```
import { Module, forwardRef } from '@nestjs/common';
import { OrderService } from './order.service';
import { OrderController } from './order.controller';
import { MongooseModule } from '@nestjs/mongoose';
import { Order, OrderSchema } from './schemas/order.schema';
import { RestaurantModule } from 'src/restaurant/restaurant.module';
import { UserModule } from 'src/user/user.module';

@Module({
```

```
  imports: [
    MongooseModule.forFeature([{ name: Order.name, schema: OrderSchema }]),
    forwardRef(() => RestaurantModule),
    forwardRef(() => UserModule),
  ],
  controllers: [OrderController],
  providers: [OrderService],
})
export class OrderModule {}
```

```
import { Inject, Injectable, forwardRef } from '@nestjs/common';
import { Order, OrderDocument } from './schemas/order.schema';
import { InjectModel } from '@nestjs/mongoose';
import { Model } from 'mongoose';
import { RepositoryAbstract } from 'src/shared/db-tools/repository-abstract';
import { UserService } from 'src/user/user.service';
import { DishService } from 'src/restaurant/dish/dish.service';
import * as uuid from 'uuid';
import { OrderStatus } from './shared/constants';

const MIN_DATE_RANGE = 20;
const MAX_DATE_RANGE = 60;

@Injectable()
export class OrderService extends RepositoryAbstract<Order, OrderDocument> {
  constructor(
    @InjectModel(Order.name) orderModel: Model<OrderDocument>,
    @Inject(forwardRef(() => DishService))
    private dishesService: DishService,
    @Inject(forwardRef(() => UserService))
    private userService: UserService
  ) {
    super(orderModel);
  }

  public async makeOrder(dishesId: string[], userId: string) {
    const averageWaitingTimeMinutes = Math.round(
      Math.random() * (MAX_DATE_RANGE - MIN_DATE_RANGE) + MIN_DATE_RANGE
    );
    const orderId = uuid.v4();
    const orderTimestamp = Date.now();
    await Promise.all(
      dishesId.map((id) => {
        return this.create({
          dishId: id,
          orderId,
          averageWaitingTimeMinutes,
          status: OrderStatus.Processing,
          userId,
          orderTimestamp,
        });
      })
    );
    return this.getOrderDetails(orderId);
  }

  public async getOrderDetails(orderId: string, status?: OrderStatus) {
    const match = {
      orderId,
    };
    if (status) {
      match['status'] = status;
    }
```

```
    return this.dataModel.aggregate([
      {
        $match: match,
      },
      {
        $lookup: {
          from: 'users',
          localField: 'userId',
          foreignField: 'id',
          as: 'userData',
        },
      },
      {
        $lookup: {
          from: 'dishes',
          localField: 'dishId',
          foreignField: 'id',
          as: 'dishData',
        },
      },
      {
        $project: {
          orderId: 1,
          dishId: 1,
          userId: 1,
          status: 1,
          averageWaitingTimeMinutes: 1,
          orderTimestamp: 1,
          userFirstName: '$userData.firstName',
          userLastName: '$userData.lastName',
          dishTitle: '$dishData.title',
          dishPrice: '$dishData.price',
        },
      },
      {
        $unwind: '$userFirstName',
      },
      {
        $unwind: '$userLastName',
      },
      {
        $unwind: '$dishTitle',
      },
      {
        $unwind: '$dishPrice',
      },
    ]);
  }

  public async updateOrderStatus(
    orderId: string,
    newStatus: OrderStatus
  ): Promise<Order[]> {
    await this.dataModel.updateMany(
```

```
    {
      orderId,
    },
    { status: newStatus }
  );
  return this.getOrderDetails(orderId, newStatus);
}

public async getOrderList(status?: OrderStatus) {
  const filter = {};
  if (status) {
    filter['status'] = status;
  }
  const list = await this.dataModel.aggregate([
    {
      $match: {
        ...filter,
      },
    },
    {
      $lookup: {
        from: 'users',
        localField: 'userId',
        foreignField: 'id',
        as: 'userData',
      },
    },
    {
      $project: {
        orderId: 1,
        dishId: 1,
        userId: 1,
        status: 1,
        averageWaitingTimeMinutes: 1,
        orderTimestamp: 1,
        userFirstName: '$userData.firstName',
        userLastName: '$userData.lastName',
      },
    },
    {
      $unwind: '$userFirstName',
    },
    {
      $unwind: '$userLastName',
    },
    {
      $group: {
        _id: '$orderId',
        doc: { $first: '$$ROOT' },
      },
    },
    {
      $replaceRoot: {
        newRoot: '$doc',
```

```
      },
    },
  ]);
  return list;
}

public getUserOrders(userId: string, status: OrderStatus) {
  const filter = {
    userId,
  };
  if (status) {
    filter['status'] = status;
  }
  return this.dataModel.aggregate([
    {
      $match: {
        ...filter,
      },
    },
    {
      $lookup: {
        from: 'dishes',
        localField: 'dishId',
        foreignField: 'id',
        as: 'dishData',
      },
    },
    {
      $project: {
        orderId: 1,
        orderTimestamp: 1,
        status: 1,
        averageWaitingTimeMinutes: 1,
        dishId: '$dishData.id',
        dishTitle: '$dishData.title',
        dishPrice: '$dishData.price',
        dishCalories: '$dishData.calories',
      },
    },
    {
      $unwind: '$dishId',
    },
    {
      $unwind: '$dishTitle',
    },
    {
      $unwind: '$dishPrice',
    },
    {
      $unwind: '$dishCalories',
    },
    {
      $group: {
        _id: '$orderId',
```

```
          entries: {
            $push: {
              orderId: '$orderId',
              userId: '$userId',
              orderTimestamp: '$orderTimestamp',
              status: '$status',
              averageWaitingTimeMinutes: '$averageWaitingTimeMinutes',
              dishId: '$dishId',
              dishTitle: '$dishTitle',
              dishPrice: '$dishPrice',
              dishCalories: '$dishCalories',
            },
          },
        },
      },
      {
        $project: {
          _id: 0,
          orderId: '$_id',
          entries: '$entries',
        },
      },
    ]);
  }
}
```

```
import { CreateUserDto } from '../user/dto/create-user.dto';
import {
  Body,
  Controller,
  Get,
  HttpCode,
  HttpStatus,
  Post,
  Request,
} from '@nestjs/common';
import { AuthService } from './auth.service';
import { SignInDto } from './dto/sign-in.dto';
import { User } from './interfaces/user.interface';
import { ApiHeader, ApiResponse, ApiTags } from '@nestjs/swagger';
import { SkipAuth } from 'src/shared/decorators/skip-auth.decorator';

@ApiTags('auth')
@Controller('auth')
export class AuthController {
  constructor(private authService: AuthService) {}

  @HttpCode(HttpStatus.OK)
  @SkipAuth()
  @Post('register')
```

```
  @ApiResponse({
    status: 200,
    description:
      'The user has been successfully registered and authorized. access_token is
returned',
  })
  @ApiResponse({
    status: 400,
    description: 'Wrong parameters or user with this email is already exists',
  })
  public register(
    @Body() registerDto: CreateUserDto
  ): Promise<{ access_token: string }> {
    return this.authService.registerUser(registerDto);
  }

  @HttpCode(HttpStatus.OK)
  @SkipAuth()
  @Post('login')
  @ApiResponse({
    status: 200,
    description: 'access_token is returned',
  })
  @ApiResponse({
    status: 401,
    description: 'User with provided email and password does not exists',
  })
  public signIn(
    @Body() signInDto: SignInDto
  ): Promise<{ access_token: string }> {
    return this.authService.signIn(signInDto);
  }

  @Get('profile')
  @ApiHeader({
    name: 'authorization',
    description: 'Custom header with access token provided',
  })
  public getProfile(@Request() request): Promise<Omit<User, 'password'>> {
    return request.user;
  }
}
```

```
import { Module } from '@nestjs/common';
import { AuthService } from './auth.service';
import { AuthController } from './auth.controller';
import { JwtModule } from '@nestjs/jwt';
import { JWT_SECRET } from 'src/shared/constants';
import { UserModule } from 'src/user/user.module';

@Module({
  imports: [
```

```
    JwtModule.register({
      global: true,
      secret: JWT_SECRET,
    }),
    UserModule,
  ],
  providers: [AuthService],
  controllers: [AuthController],
})
export class AuthModule {}
```

```
import {
  BadRequestException,
  Injectable,
  UnauthorizedException,
} from '@nestjs/common';
import { User } from './interfaces/user.interface';
import { JwtService } from '@nestjs/jwt';
import { CreateUserDto } from '../user/dto/create-user.dto';
import { UserService } from 'src/user/user.service';
import { SignInDto } from './dto/sign-in.dto';
import { Role } from './constants';

export interface AccessTokenInfo {
  access_token: string;
}

@Injectable()
export class AuthService {
  constructor(
    private jwtService: JwtService,
    private userService: UserService
  ) {}

  private async authenticate(user: User): Promise<AccessTokenInfo> {
    const payload: Omit<User, 'password'> & { sub: string } = {
      sub: user.id,
      id: user.id,
      email: user.email,
      firstName: user.firstName,
      lastName: user.lastName,
      address: user.address,
      role: user.role,
      phoneNumber: user.phoneNumber,
    };
    const accessToken = await this.jwtService.signAsync(payload);
    return {
      access_token: accessToken,
    };
  }

  private encryptPassword(password: string): string {
```

```
    return Buffer.from(password).toString('base64');
  }

  async registerUser(
    registrationData: CreateUserDto
  ): Promise<{ access_token: string }> {
    const isExists = await this.userService.checkIfEmailExists(
      registrationData.email
    );
    if (isExists) {
      throw new BadRequestException({
        code: '0001',
        message: 'User with the same email is already exists',
      });
    }
    const newUser = await this.userService.createUser({
      ...registrationData,
      role: Role.user,
    });
    return this.authenticate(newUser);
  }

  async signIn(signInData: SignInDto): Promise<AccessTokenInfo> {
    const user = await this.userService.getUserData(
      signInData.email,
      signInData.password
    );
    if (!user) {
      throw new UnauthorizedException({
        code: '0002',
        message: 'User does not exist or email and password pair is invalid',
      });
    }
    return this.authenticate(user);
  }
}
```

```
import Button from "@mui/material/Button";
import CssBaseline from "@mui/material/CssBaseline";
import TextField from "@mui/material/TextField";
import Link from "@mui/material/Link";
import Grid from "@mui/material/Grid";
import Box from "@mui/material/Box";
import Typography from "@mui/material/Typography";
import Container from "@mui/material/Container";
import { Alert, IconButton, InputAdornment, Snackbar } from "@mui/material";
import { Visibility, VisibilityOff } from "@mui/icons-material";
import { useState } from "react";
import { useNavigate } from "react-router-dom";
import { HOME_ROUTE, REGISTRATION_ROUTE } from "../../consts";
import { useDispatch, useSelector } from "react-redux";
import { login } from "../../state/auth/auth-slice";
import { AppDispatch, RootState } from "../../state/store";
```

```
const SignIn = () => {
  const navigate = useNavigate();
  const dispatch = useDispatch<AppDispatch>();
  const errorMessage: string | null = useSelector<RootState, string | null>(
    (state) => state.auth.error
  );
  const [showPassword, setShowPassword] = useState(false);
  const [isDisabled, setIsDisabled] = useState(true);

  const handleClickShowPassword = () => setShowPassword((show) => !show);

  const handleSubmit = async (event: React.FormEvent<HTMLFormElement>) => {
    event.preventDefault();

    const formData = new FormData(event.currentTarget);
    const email = formData.get("email") as string;
    const password = formData.get("password") as string;

    const credentials = {
      email,
      password,
    };

    try {
      const actionResult = await dispatch(login(credentials));

      if (actionResult.payload) {
        // Successful login
        navigate(HOME_ROUTE);
      }
    } catch (error) {
      console.error(error);
    }
  };

  const handleChange = (event: React.FormEvent<HTMLFormElement>) => {
    const formData = new FormData(event.currentTarget);

    const password = formData.get("password") as string;

    password.length > 3 ? setIsDisabled(false) : setIsDisabled(true);
  };

  return (
    <div className="h-screen flex items-center justify-center">
      <div className="bg-white/35 py-12 px-4 rounded-3xl m-3">
        <Container component="main" maxWidth="xs">
          <CssBaseline />
          <Box
            sx={{
              display: "flex",
              flexDirection: "column",
              alignItems: "center",
```

```
          }}
        >
          <Typography
            sx={{ color: "white", fontWeight: 600 }}
            component="h1"
            variant="h4"
          >
            Sign in
          </Typography>
          <Box
            component="form"
            noValidate
            onSubmit={handleSubmit}
            onChange={handleChange}
            sx={{ mt: 3 }}
          >
            <Grid container spacing={2}>
              <Grid item xs={12}>
                <TextField
                  required
                  fullWidth
                  id="email"
                  label="Email Address"
                  name="email"
                  autoComplete="email"
                  autoFocus
                />
              </Grid>
              <Grid item xs={12}>
                <TextField
                  required
                  fullWidth
                  name="password"
                  label="Password"
                  type={showPassword ? "text" : "password"}
                  id="password"
                  autoComplete="new-password"
                  InputProps={{
                    endAdornment: (
                      <InputAdornment position="end">
                        <IconButton
                          aria-label="toggle password visibility"
                          onClick={handleClickShowPassword}
                          edge="end"
                        >
                          {showPassword ? <VisibilityOff /> : <Visibility />}
                        </IconButton>
                      </InputAdornment>
                    ),
                  }}
                />
              </Grid>
            </Grid>
            <Button
```

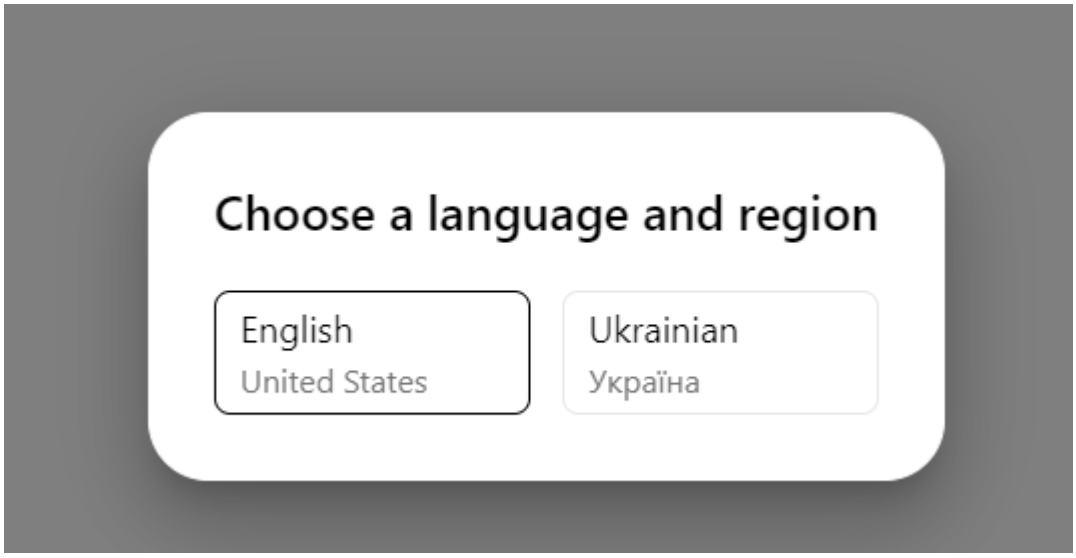
```
                type="submit"
                fullWidth
                variant="contained"
                disabled={isDisabled}
                sx={{ mt: 3, mb: 2, borderRadius: "10px", fontSize: "18px" }}
              >
                Sign In
              </Button>
              <Grid container justifyContent="flex-end">
                <Grid item>
                  <Link variant="body2">
                    <span
                      className="cursor-pointer text-lg"
                      onClick={() => navigate(REGISTRATION_ROUTE)}
                    >
                      Don't have account? Sign up
                    </span>
                  </Link>
                </Grid>
              </Grid>
            </Box>
          </Box>
        </Container>
      </div>
      {errorMessage && (
        <Snackbar open={errorMessage.length > 0}>
          <Alert severity="error">{errorMessage}</Alert>
        </Snackbar>
      )}
    </div>
  );
};

export default SignIn;
```

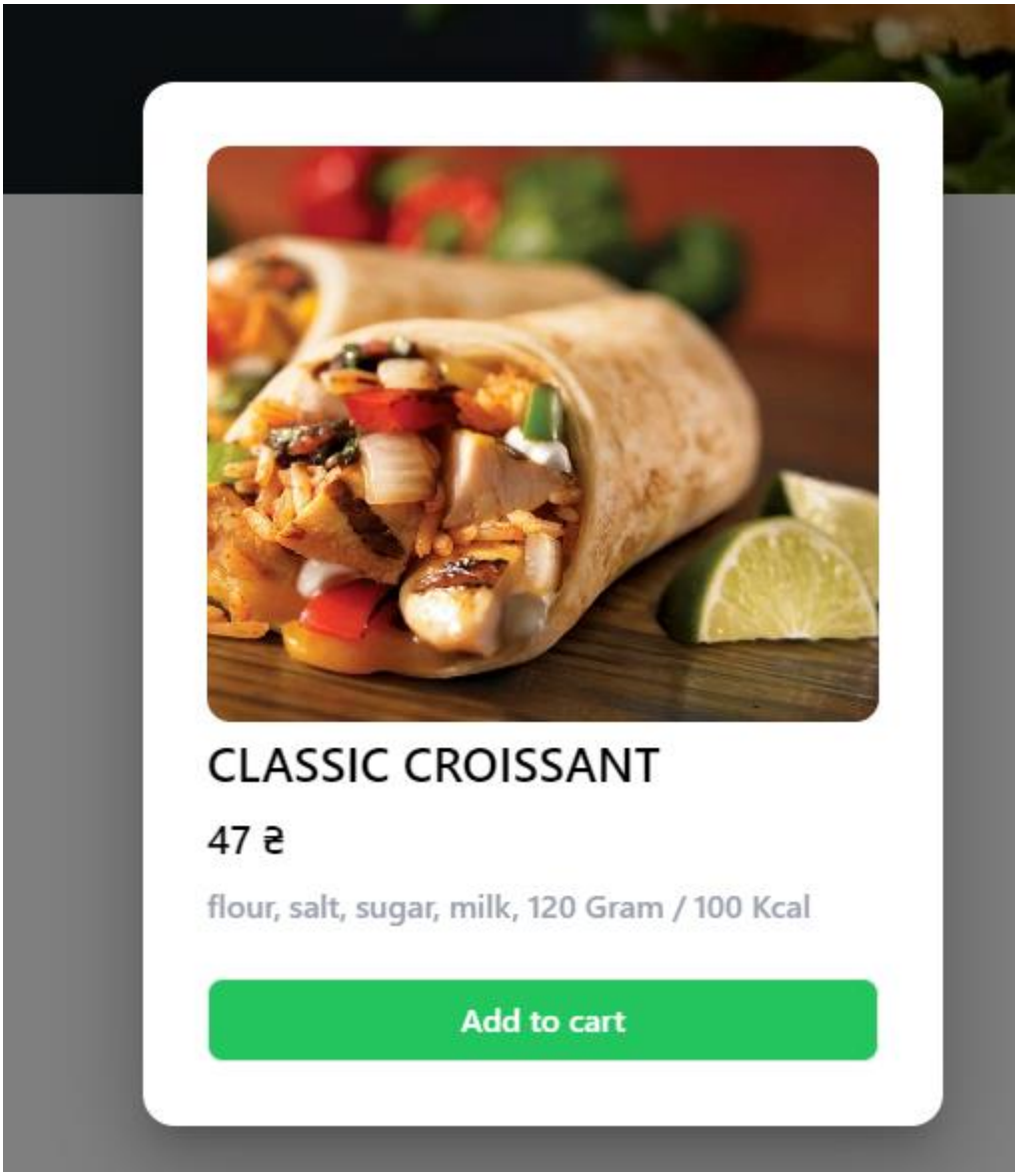
```
import { combineReducers, configureStore } from "@reduxjs/toolkit";
import authReducer from "./auth/auth-slice";
import registerReducer from "./register/register-slice";
import getProfileInfoReducer from "./profile/profile-slice";
import getUserTableReducer from "./admin/get-users-slice";
import deleteUserReducer from "./admin/delete-user-slice"; // remove
import addUserReducer from "./admin/add-user-slice"; // remove
import getRestaurantsReducer from "./restaurant/restaurant-slice";
import findRestaurantByIdReducer from "./restaurant/getRestaurantById-slice";
import deleteRestaurantReducer from "./admin/delete-restaurant-slice"; // remove
import addRestaurantReducer from "./admin/add-restaurant-slice"; // remove
import addReviewReducer from "./review/add-review.slice"; // remove
import getDishesReducer from "./restaurant/dish/dish-slice";
import themeSwitchReducer from "./theme/theme-switcher-slice";
import languageSelectReducer from "./language/select-language-slice";
import cartReducer from "./order/cartSlice";
import orderReducer from "./order/orderSlice";
import orderHistoryReducer from "./order/orderHistorySlice";
```

```
const rootReducer = combineReducers({
  auth: authReducer,
  register: registerReducer,
  profileInfo: getProfileInfoReducer,
  userTable: getUserTableReducer,
  deleteUser: deleteUserReducer,
  add: addUserReducer,
  getRestaurants: getRestaurantsReducer,
  findRestaurantById: findRestaurantByIdReducer,
  deleteRestaurant: deleteRestaurantReducer,
  addRestaurant: addRestaurantReducer,
  addReview: addReviewReducer,
  getDishes: getDishesReducer,
  theme: themeSwitchReducer,
  setLanguage: languageSelectReducer,
  cart: cartReducer,
  orders: orderReducer,
  orderHistory: orderHistoryReducer,
});

export const store = configureStore({
  reducer: rootReducer,
});

export type RootState = ReturnType<typeof store.getState>;
export type AppDispatch = typeof store.dispatch;
```

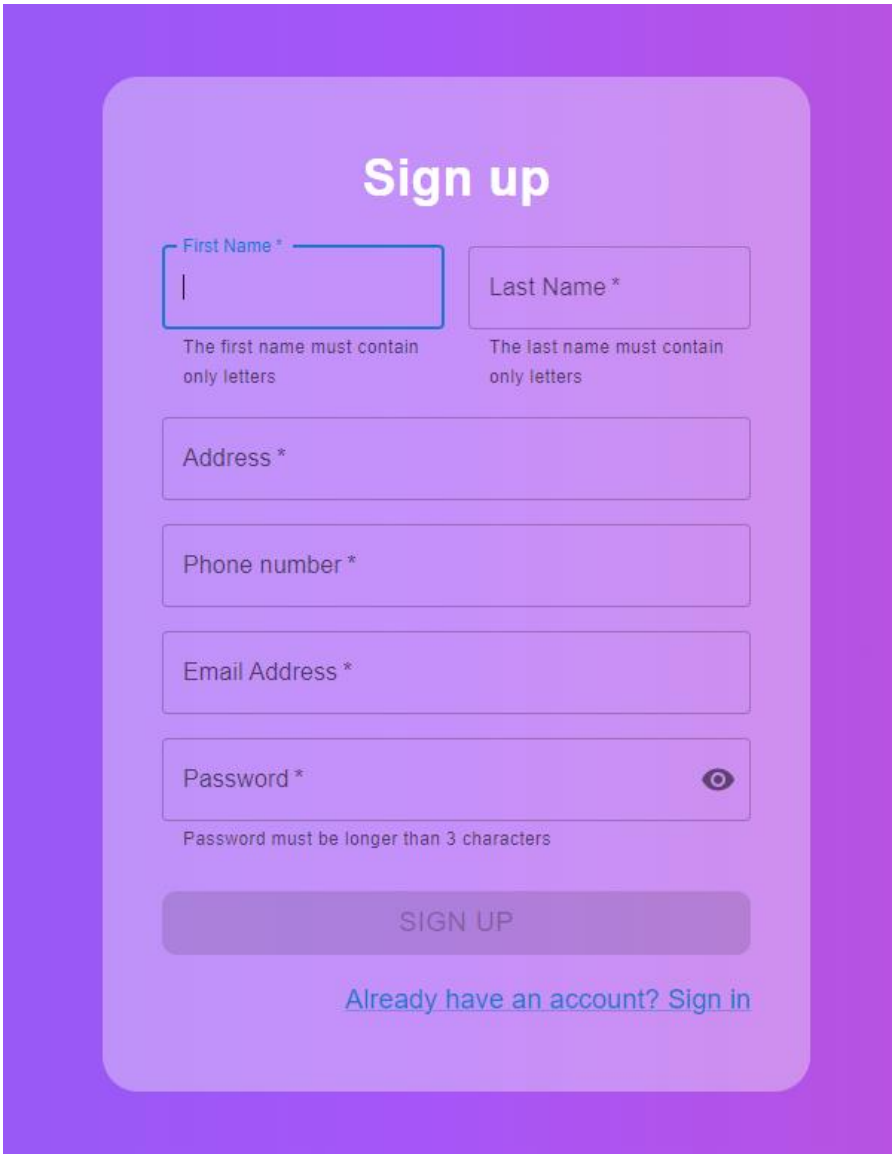
```
import en from "./en.json";
import ua from "./ua.json";

export const translate = (key: string, language: string): string => {
  let langData: { [key: string]: string } = {};

  if (language === "English") {
    langData = en;
  } else if (language === "Ukrainian") {
    langData = ua;
  }

  return langData[key];
};
```

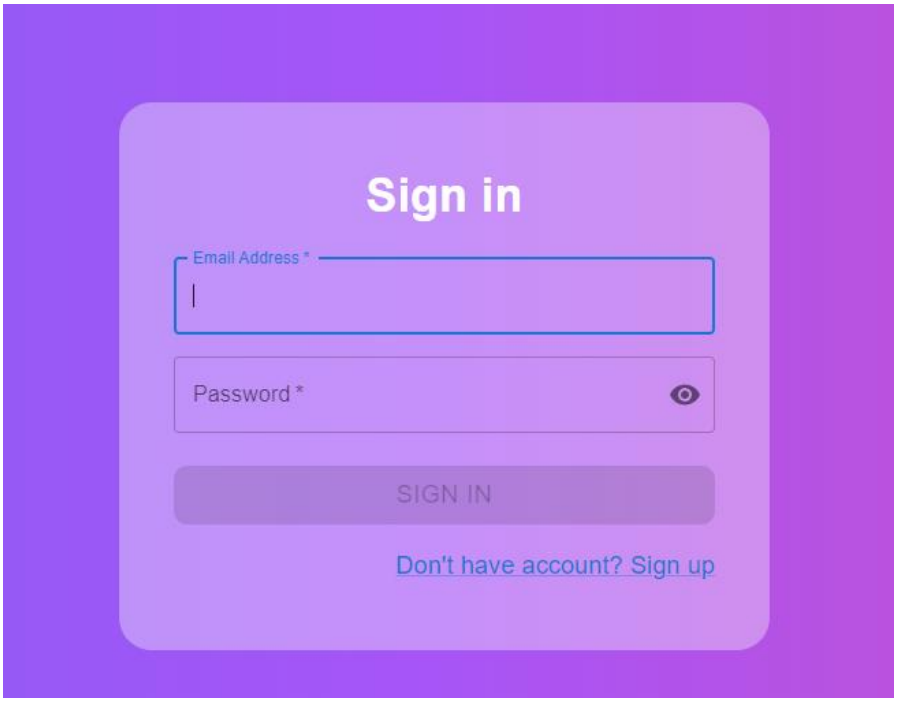
Додаток Б

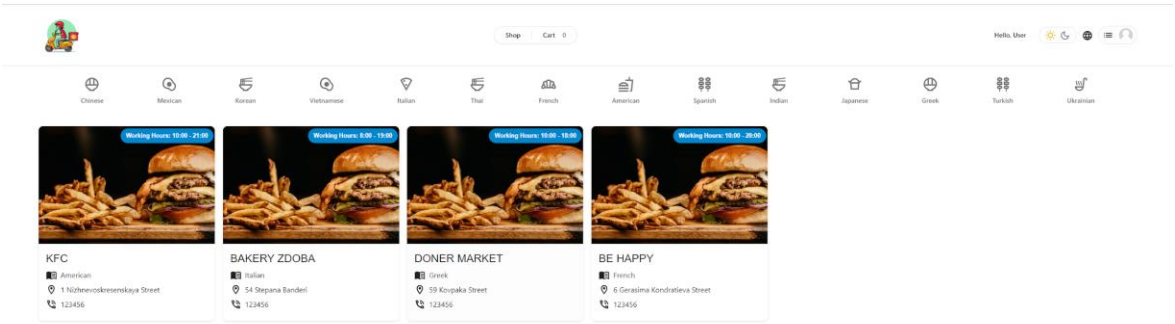

Choose a language and region
English
United States
Ukrainian
Україна

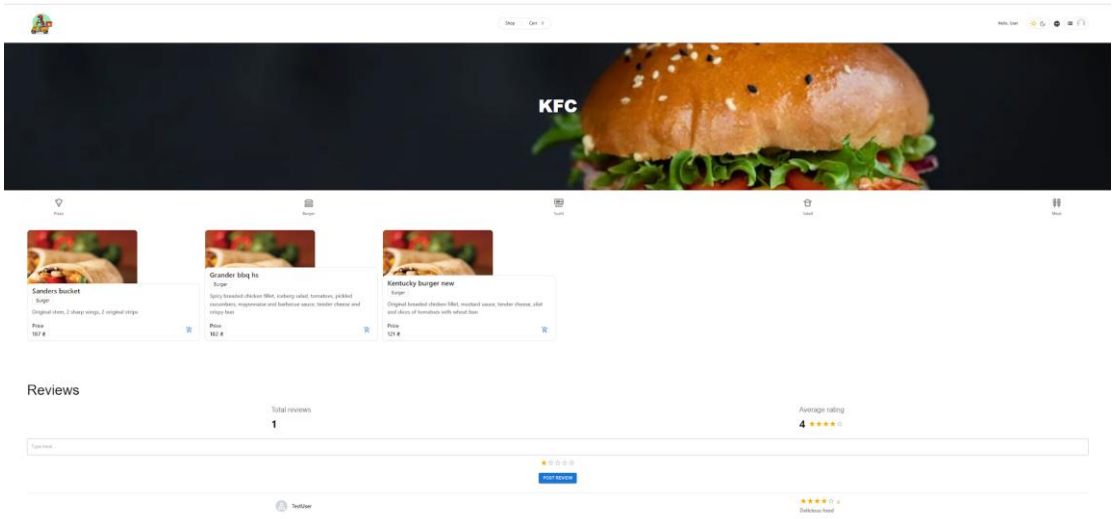

CLASSIC CROISSANT
47 ₴
flour, salt, sugar, milk, 120 Gram / 100 Kcal
Add to cart

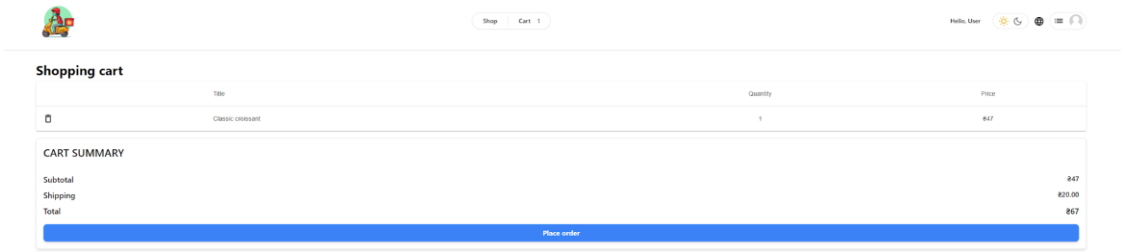
Додаток В

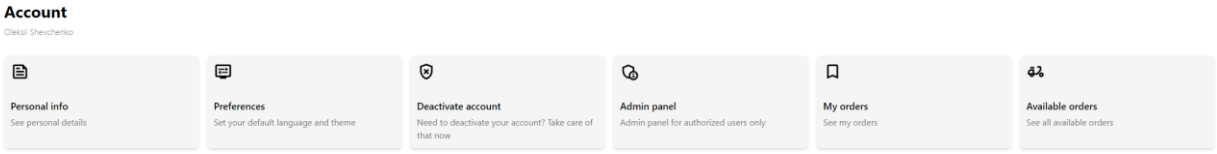

Sign up
First Name *
Last Name *
The first name must contain only letters
The last name must contain only letters
Address *
Phone number *
Email Address *
Password *
Password must be longer than 3 characters
SIGN UP
Already have an account? Sign in

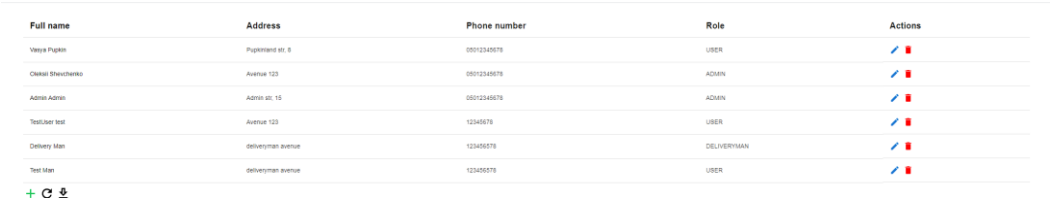

Sign in
Email Address *
Password *
SIGN IN
Don't have account? Sign up

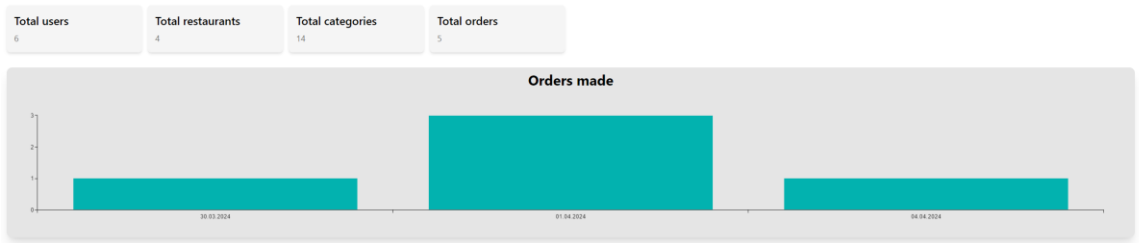

KFC
American
1 Nizhnevoskresenskaya Street
123456
BAKERY ZDOBA
Italian
54 Stepana Bandery
123456
DONER MARKET
Greek
59 Kovpaka Street
123456
BE HAPPY
French
6 Gerasima Kondratieva Street
123456

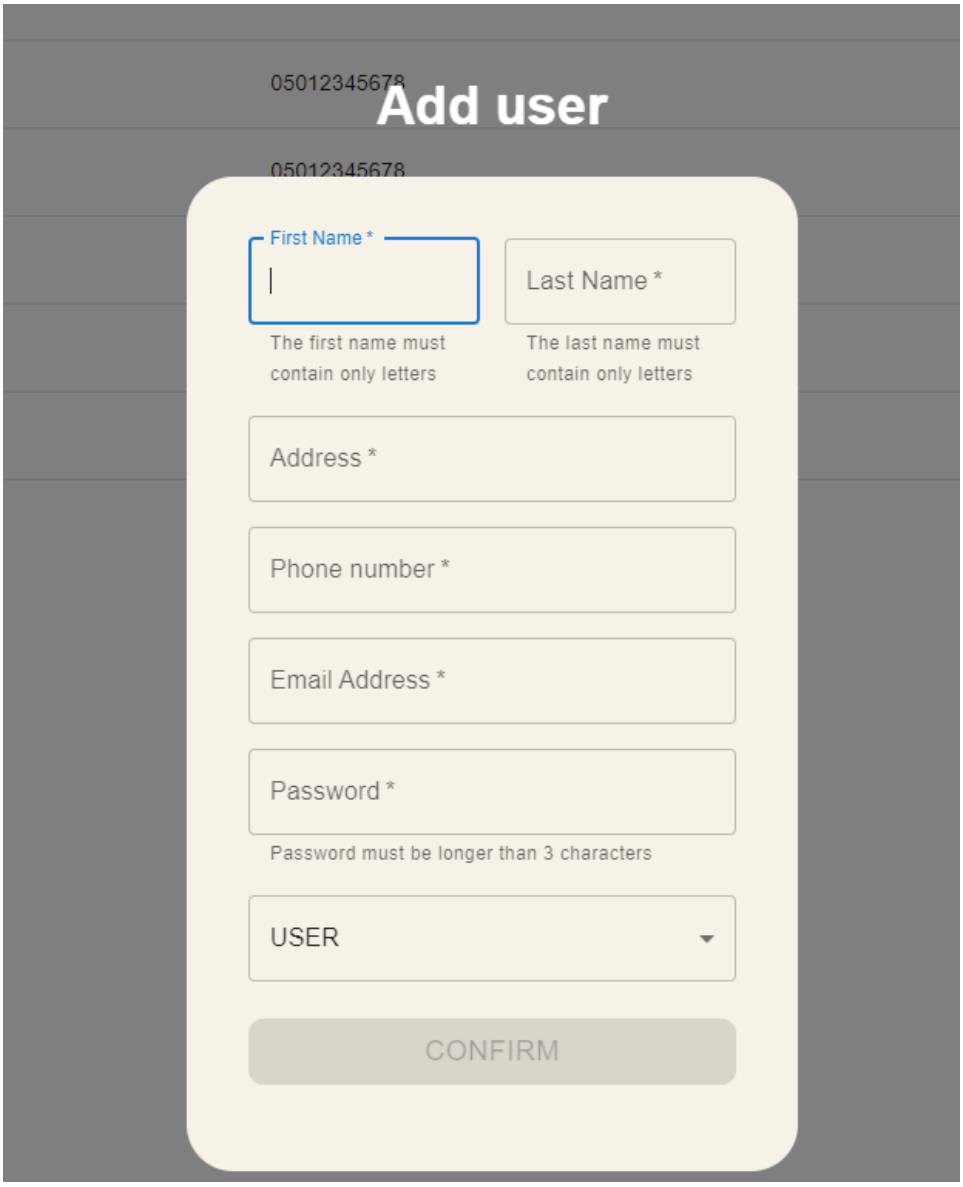

KFC
Sanders bucket
Grander bbq hs
Kentucky burger new
Reviews
Total reviews
1
Average rating
4
POST REVIEW


Shop
Cart 1
Hello, User
Shopping cart
Title
Quantity
Price
Classic croissant
1
₴47
CART SUMMARY
Subtotal
₴47
Shipping
₴20.00
Total
₴67
Place order


Account
Oleksii Shevchenko
Personal info
See personal details
Preferences
Set your default language and theme
Deactivate account
Need to deactivate your account? Take care of that now
Admin panel
Admin panel for authorized users only
My orders
See my orders
Available orders
See all available orders


Full name
Address
Phone number
Role
Actions
Vasya Pupkin
Pupkinland str, 8
0501234567
USER
Oleksii Shevchenko
Avenue 123
0501234567
ADMIN
Admin Admin
Admin str, 15
0501234567
ADMIN
TestUser test
Avenue 123
12345678
USER
Delivery Man
deliveryman avenue
123456578
DELIVERYMAN
Test Man
deliveryman avenue
123456578
USER


Total users
6
Total restaurants
4
Total categories
14
Total orders
5
Orders made
30.03.2024
01.04.2024
04.04.2024


05012345678
Add user
05012345678
First Name *
Last Name *
The first name must contain only letters
The last name must contain only letters
Address *
Phone number *
Email Address *
Password *
Password must be longer than 3 characters
USER
CONFIRM