МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Острозька академія»
Навчально-науковий інститут інформаційних технологій та бізнесу
Кафедра інформаційних технологій та аналітики даних

КВАЛІФІКАЦІЙНА РОБОТА
на здобуття освітнього ступеня бакалавра

на тему: «**Розробка веб-додатку для управління меню ресторану TomYum на основі React та FastAPI»**

Виконав: студент 4 курсу, групи КН-41
першого (бакалаврського) рівня вищої освіти
спеціальності 122 Комп'ютерні науки
освітньо-професійної програми «Комп'ютерні науки»
Кляйн Олексій Юрійович

Керівник: *викладач кафедри інформаційних технологій та аналітики даних*
Місай Володимир Віталійович

Рецензент: *Розробник програмного забезпечення в ТОВ ОБНОВА-ЄВРОШОП, викладач кафедри інформаційних технологій та аналітики даних НаУОА*
Шведюк Володимир Володимирович

РОБОТА ДОПУЩЕНА ДО ЗАХИСТУ
Завідувач кафедри інформаційних технологій та аналітики даних _____
(проф., д.е.н. Кривицька О.Р.)
Протокол №10 від 28 травня 2025 р.

Острог, 2025

АНОТАЦІЯ
кваліфікаційної роботи
на здобуття освітнього ступеня бакалавра

Тема: Розробка веб-додатку для управління меню ресторану TomYum на основі React та FastAPI

Автор:Кляйн Олексій Юрійович

Науковий керівник: викладач кафедри ІТАД, Місай Володимир Віталійович

Захищена «……»…… 20__ року.

Пояснювальна записка до кваліфікаційної роботи: *54с., 6 рис., 2 табл., 0 додатків, 20 джерел.*

Ключові слова: *веб-додаток, FastAPI, React, меню ресторану, авторизація, база даних, ролі користувачів.*

Короткий зміст праці:

Кваліфікаційна робота присвячена розробці веб-додатку для управління меню ресторану TomYum, який забезпечує ефективну взаємодію між адміністраторами закладу та клієнтами. У роботі реалізовано клієнт-серверну архітектуру з використанням React.js для клієнтської частини та FastAPI для серверної логіки. Особливу увагу приділено побудові зручного, адаптивного інтерфейсу, системі ролей із розмежуванням прав доступу (адміністратор/клієнт), а також CRUD-функціоналу для категорій та страв. Архітектура проєкту структурована відповідно до принципів чистого коду, що забезпечує модульність, масштабованість і можливість подальшого розвитку системи.Розроблений веб-застосунок є практичним рішенням для автоматизації роботи з меню в закладах харчування.

_____

ANNOTATION

of qualification paper

for bachelor's degree

Theme: Development of a Web Application for Restaurant Menu Management TomYum Based on React and FastAPI

Author: Kliain Oleksii

Scientific supervisor: Lecturer of the Department of ITAD, Misai Volodymyr Vitalievich

Defended: «……»…… 20__ year.

Explanatory note to the qualification work: *54 pages, 6 figures, 2 tables, 0 appendices, 20 sources.*

Keywords: *web application, FastAPI, React, restaurant menu, authorization, database, user roles.*

Abstract:

The qualification thesis is devoted to the development of a web application for managing the menu of the TomYum restaurant, which ensures effective interaction between restaurant administrators and clients. The system is implemented using a client-server architecture, with React.js used for the client side and FastAPI for the server logic. Special attention is paid to the creation of a user-friendly, adaptive interface, a role-based access system (admin/client), and CRUD functionality for categories and dishes. The project architecture is structured according to clean code principles, which ensures modularity, scalability, and the possibility of further system development. The developed web application is a practical solution for automating menu management in food service establishments.

_____

Зміст

ВСТУП

Актуальність обраної теми зумовлена стрімким розвитком веб-технологій і постійним зростанням попиту на зручні, масштабовані та функціональні веб-додатки для автоматизації бізнес-процесів. Веб-рішення дозволяють ефективно організувати взаємодію між користувачами та цифровими платформами у режимі реального часу, забезпечуючи високий рівень доступності, безпеки й адаптивності до потреб конкретного бізнесу. Зокрема, для ресторанної сфери актуальним є впровадження веб-додатків, які дають змогу швидко та зручно управляти меню, відображати актуальні позиції для клієнтів і адмініструвати контент через захищений інтерфейс.

Мета дослідження полягає у створенні повнофункціонального веб-додатку для управління меню ресторану з авторизацією користувачів, розмежуванням прав доступу, зручним інтерфейсом і можливістю редагування контенту.

Завдання дослідження охоплюють аналіз предметної області, проєктування інформаційної системи, а також вибір інструментарію, методів реалізації та тестування програмного продукту, що забезпечує ефективну реалізацію поставленої мети.

Об'єктом дослідження є веб-додаток, який виступає як повноцінна інформаційна система для цифрового управління меню ресторану. Така система створюється з метою забезпечення ефективного адміністрування контенту — додавання, редагування та видалення страв і категорій — з боку адміністратора закладу. Окрім того, веб-додаток забезпечує зручну взаємодію з користувачами, зокрема надання клієнтам можливості переглядати актуальне меню в режимі реального часу.

Таким чином, об'єкт дослідження охоплює не лише технічну реалізацію веб-інтерфейсу, а й логіку обробки даних, організацію доступу для різних категорій користувачів, підтримку адаптивного дизайну та інтеграцію з базою даних для зберігання інформації про страви.

Предметом дослідження виступають інструментальні засоби розробки веб-додатків, включаючи математичні моделі, методи й інформаційні технології, що використовуються для побудови та дослідження функціонування об’єкта роботи.

Методи дослідження, що були застосовані в межах виконання кваліфікаційної роботи, включають аналіз літературних джерел і вивчення аналогічних рішень, використання системного та функціонального підходів до проєктування, застосування об’єктно-орієнтованого програмування, а також використання сучасних інструментів тестування та розгортання веб-додатків .

РОЗДІЛ 1
ТЕОРЕТИЧНІ ОСНОВИ СТВОРЕННЯ ВЕБ-ДОДАТКІВ

1.1. Сучасні технології у розробці веб-додатків

Розробка сучасних веб-додатків ґрунтується на використанні взаємопов’язаних технологій, які забезпечують функціональність, швидкодію, безпеку й масштабованість. Архітектура поділяється на клієнтську (frontend) та серверну (backend) частини, кожна з яких має специфічні інструменти, бібліотеки та фреймворки. До вибору технологій розробники підходять з урахуванням багатьох критеріїв: простота інтеграції, продуктивність, активність спільноти, документація, підтримка тестування тощо.

На боці клієнта ключову роль відіграють JavaScript-фреймворки. Найпопулярнішими серед них є React, Angular і Vue.js. React, створений компанією Meta (Facebook), є бібліотекою для побудови інтерфейсів користувача, яка відзначається гнучкістю, ефективністю завдяки Virtual DOM, компонентною архітектурою та великим вибором інструментів для інтеграції (React Router, Redux, Next.js) **[1]**. Angular — це повноцінний фреймворк, розроблений Google, який забезпечує строго структуровану архітектуру додатків, двобічну прив’язку даних (two-way binding), підтримку TypeScript та великої кількості вбудованих сервісів **[2]**. Vue.js, розроблений незалежною командою, — легкий фреймворк, який дозволяє швидко створювати інтерактивні інтерфейси, добре підходить для малих і середніх проєктів завдяки своїй простоті та низькому порогу входження **[3]**.

Кожен із цих фреймворків має свої переваги й використовується в залежності від специфіки проєкту, технічних вимог та уподобань команди розробників.

Таблиця 1.1

Порівняльна характеристика фреймворків

Характеристика	React	Angular	Vue.js
Тип	Бібліотека	Фреймворк	Фреймворк
Підтримка	Meta	Google	Спільнота
Навчання	Помірне	Високе	Легке
Архітектура	Відкрита	Строга	Проста
Розмір проєктів	Універсальний	Великі	Малі/середні

На боці сервера вибір фреймворків також великий: Flask, Django, FastAPI (Python); Express (Node.js) **[6]**; Spring Boot (Java) тощо. У межах цієї роботи використовується FastAPI , який забезпечує високу продуктивність завдяки асинхронній обробці запитів, автоматичну генерацію документації за стандартом OpenAPI **[10]**, строгість типізації (з використанням Python type hints), гнучку інтеграцію з SQLAlchemy і Pydantic **[4]**. На відміну від Django **[5]**, FastAPI не нав’язує структуру проєкту і дає розробникові більше свободи у виборі рішень.

Це особливо зручно для побудови масштабованих API, де важливо контролювати архітектуру і залежності вручну. Завдяки своїй гнучкості FastAPI добре підходить як для невеликих сервісів, так і для великих систем мікросервісної архітектури. У поєднанні з Alembic для міграцій бази даних та Uvicorn для запуску серверу, FastAPI дозволяє створити повноцінний, продуктивний та легко підтримуваний бекенд.

Таблиця 1.2

Порівняльна характеристика backend-фреймворків

Фреймворк	Мова	Продуктивність	Документація	Структурованість
FastAPI	Python	Висока	Автоматична	Гнучка
Django	Python	Середня	Вбудована	Висока
Express	Node.js	Висока	Ручна	Середня

Ще одним важливим компонентом веб-додатку є база даних. У нашій роботі використовується PostgreSQL — потужна реляційна система керування базами даних, яка забезпечує транзакційність, підтримку JSON, гнучке налаштування індексів, масштабування, надійність і велику спільноту **[7]**. У порівнянні з MySQL, PostgreSQL краще підходить для складних реляційних моделей, а у порівнянні з MongoDB **[8]** забезпечує цілісність та структурованість даних, що є необхідним для реалізації коректної логіки у веб-додатках із розмежуванням доступу, CRUD-операціями та міграціями.

Таким чином, вибір стеку React + FastAPI + PostgreSQL у цій роботі базується на співвідношенні гнучкості, продуктивності, підтримки документації, розширюваності та активності спільноти.

1.2. Архітектура клієнт-серверної взаємодії

Клієнт-серверна архітектура — це одна з базових концепцій побудови веб-додатків, яка передбачає поділ програмного забезпечення на дві окремі частини: клієнтську (frontend) та серверну (backend). Клієнт відповідає за взаємодію з

користувачем, відображення інтерфейсу, надсилання запитів до сервера. Сервер, у свою чергу, обробляє запити, виконує бізнес-логіку, звертається до бази даних і повертає відповідь клієнту. Цей підхід забезпечує гнучкість у реалізації, дозволяє розділяти відповідальність між компонентами та масштабувати систему при зростанні навантаження.

У класичній клієнт-серверній взаємодії обмін даними відбувається через протокол HTTP або HTTPS. Клієнт може надсилати запити з використанням таких методів, як GET (отримання даних), POST (створення ресурсу), PUT (оновлення ресурсу) або DELETE (видалення ресурсу) [9]. Сервер обробляє запити та повертає відповіді, зазвичай у форматі JSON або XML.

Важливою частиною сучасної клієнт-серверної взаємодії є реалізація REST-архітектури. REST (Representational State Transfer) — це стиль архітектури, що базується на принципах взаємодії клієнта і сервера без збереження стану (stateless), використанні унікальних ресурсів (URI) і стандартних HTTP-методів. Такий підхід дозволяє зробити API додатку логічним, зрозумілим, добре структурованим і придатним до масштабування.

У межах цієї роботи клієнтська частина реалізована за допомогою React. Вона надсилає HTTP-запити до серверного API, реалізованого у FastAPI. Сервер перевіряє запит, взаємодіє з базою даних PostgreSQL через SQLAlchemy та формує відповідь у форматі JSON, яка повертається клієнту. Наприклад, коли користувач відкриває сторінку з меню ресторану, клієнт надсилає запит GET до /dishes, сервер витягує інформацію про страви з бази, структурує її і надсилає клієнту для відображення.

Архітектура дозволяє використовувати додаткові технології, такі як токен-базована автентифікація (JWT), кешування на рівні клієнта або проксі, а також захист від атак (наприклад, CSRF або XSS). Крім того, реалізація CORS (Cross-Origin Resource Sharing) дозволяє безпечно здійснювати запити між доменами, що особливо важливо при хостингу фронтенду й бекенду на різних серверах.

Таким чином, клієнт-серверна архітектура є гнучкою, масштабованою та ефективною моделлю взаємодії між компонентами веб-додатку, що дозволяє реалізувати складні інформаційні системи з розподіленими обов’язками, чіткою структурою та можливістю подальшого розвитку.

1.3. Валідація даних у веб-додатках

У процесі взаємодії користувача з веб-додатком валідація введених даних відіграє критично важливу роль, оскільки дозволяє гарантувати, що дані, які надходять на сервер, є коректними, структурованими та безпечними. Це особливо важливо в умовах динамічного веб-середовища, де ризик обробки некоректної або шкідливої інформації досить високий. Завдяки валідації забезпечується не лише цілісність зібраної інформації, а й ефективний захист від потенційних загроз, зокрема SQL-ін’єкцій, введення даних у хибному форматі або спроб обходу бізнес-логіки додатку.

У веб-розробці зазвичай розрізняють три рівні валідації. Перший — це фронтендна валідація, яка виконується безпосередньо на стороні клієнта, за допомогою HTML5-атрибутів, JavaScript або спеціалізованих бібліотек, таких як Formik чи Yup. Такий підхід забезпечує миттєвий зворотний зв’язок для користувача, дозволяючи швидко реагувати на помилки ще до надсилання форми.

Другий рівень — бекендна валідація, що відбувається на серверній стороні. Вона є обов’язковою, оскільки тільки сервер може остаточно підтвердити достовірність і безпеку даних перед їх обробкою та збереженням. Цей тип валідації запобігає зловмисним діям і порушенню цілісності системи.

Третій рівень — валідація на рівні бізнес-логіки. Вона перевіряє, чи відповідають вхідні дані конкретним правилам предметної області. Наприклад, чи не є ціна страви від’ємною, або чи не перевищує назва дозволену кількість символів. Цей рівень валідації дозволяє підтримувати логічну узгодженість даних і дотримання специфіки конкретного бізнесу.

У FastAPI валідація даних реалізується за допомогою бібліотеки Pydantic **[13]**. Вона дозволяє описувати структури даних у вигляді Python-класів, в яких вказуються типи даних і обмеження для кожного поля. Це значно спрощує процес валідації, робить його декларативним і зручним для підтримки.

Крім перевірки типів і обмежень, Pydantic дозволяє використовувати регулярні вирази для перевірки формату email-адрес, номерів телефонів, складних шаблонів рядків тощо. Також підтримується створення вкладених моделей для складних структур, наприклад, списків об’єктів, дат з певним форматом або значень за умовчанням.

Таким чином, ефективна реалізація валідації є не лише запорукою безпеки, а й необхідною умовою для стабільної та передбачуваної роботи веб-додатку, забезпечуючи комфортне користувацьке середовище й захист внутрішньої логіки системи.

Для забезпечення коректної обробки даних, що вводяться користувачем, необхідна валідація — перевірка відповідності даних заданим правилам. У FastAPI вона реалізується через Pydantic-моделі. Це дозволяє гарантувати типізацію, межі значень, обов’язковість полів тощо.

Це забезпечує автоматичну перевірку даних при кожному запиті, підвищує безпеку та стабільність додатку.

1.4. Побудова REST API

Одним із найважливіших аспектів побудови сучасних веб-застосунків є реалізація інтерфейсу прикладного програмування (API), який забезпечує взаємодію між клієнтською та серверною частинами. Одним із найпоширеніших архітектурних стилів для реалізації API є REST (Representational State Transfer). Цей підхід базується на використанні стандартних методів протоколу HTTP та чітких принципів, які забезпечують просту, масштабовану та ефективну комунікацію між компонентами системи.

Основна концепція архітектурного стилю REST полягає в тому, що кожен ресурс у системі представляється як об’єкт з унікальним ідентифікатором — URI (Uniform Resource Identifier). Наприклад, шлях /dishes може вказувати на колекцію страв у додатку. Доступ до цих ресурсів і взаємодія з ними здійснюється за допомогою стандартних HTTP-методів: GET використовується для отримання одного ресурсу або списку, POST — для створення нового, PUT чи PATCH — для оновлення наявного, а DELETE — для його видалення.

Архітектура REST базується на кількох ключових принципах, які сприяють її ефективності. По-перше, це клієнт-серверна модель, що передбачає чіткий розподіл обов’язків: клієнт займається інтерфейсом, а сервер — обробкою запитів і збереженням даних. По-друге, REST вимагає безстанової взаємодії (statelessness), що означає: кожен запит повинен містити всю необхідну інформацію, оскільки сервер не зберігає попередній стан клієнта.

Ще одним важливим принципом є універсальність інтерфейсу — застосування уніфікованих способів доступу до ресурсів. REST також підтримує кешування відповідей, що дозволяє зменшити навантаження на сервер і пришвидшити обробку повторних запитів. Крім того, архітектура допускає багаторівневу реалізацію — використання проміжних серверів (наприклад, проксі) для підвищення продуктивності чи безпеки. Нарешті, REST допускає можливість кодування на вимогу, тобто передачу з сервера виконуваного коду або скриптів, які клієнт може запускати для розширення функціональності.

У рамках цього проєкту реалізація REST API здійснювалася з використанням фреймворку FastAPI, який є сучасним інструментом для створення асинхронних веб-додатків на мові Python. FastAPI побудований на базі стандарту ASGI та використовує можливості типізації Python для автоматичної валідації вхідних даних, генерації документації, обробки запитів та формування відповідей.

Створення маршруту для отримання списку страв у FastAPI реалізується за допомогою функціонального синтаксису. Наприклад:

Лістинг 1.4.1. Ендпоінт для отримання списку всіх страв

```
@router.get("/", response_model=List[DishRead])
async def read_dishes(session: AsyncSession = Depends(get_async_session)):
    return await dish_repo.get_all_dishes(session)
```

У наведеному прикладі за допомогою декоратора @router.get("/") створюється маршрут для обробки HTTP-запиту типу GET. Параметр response_model вказує на те, що відповідь має відповідати Pydantic-моделі DishRead, що забезпечує автоматичну серіалізацію та перевірку структури відповіді. Сама функція read_dishes є асинхронною і використовує залежність Depends(get_async_session) для отримання сесії бази даних, з якою здійснюється взаємодія через відповідний репозиторій.

FastAPI також забезпечує валідацію вхідних параметрів за допомогою Pydantic-моделей, що дозволяє автоматично перевіряти коректність і типізацію даних, які надходять у запитах. Крім того, фреймворк підтримує обробку стандартних HTTP-статусів, зокрема таких як 200 OK, 201 Created, 404 Not Found, 422 Unprocessable Entity, що дозволяє коректно реагувати на різні ситуації під час виконання запитів. Важливою функцією є також підтримка як query-, так і path-параметрів, що забезпечує гнучкість в організації маршрутів. FastAPI дає можливість реалізовувати авторизацію та автентифікацію, зокрема з використанням JWT-токенів **[11]**, що підвищує рівень безпеки додатків. Крім того, фреймворк дозволяє легко інтегрувати сторонні сервіси, такі як Cloudinary **[12]**, що використовується для зберігання та обробки зображень. Однією з ключових переваг FastAPI є автоматична генерація інтерактивної документації у форматі OpenAPI, доступної через інтерфейси Swagger UI та ReDoc, що значно полегшує процес розробки, тестування та інтеграції API.

У межах реалізації даного веб-додатку REST API дозволив реалізувати всі необхідні операції для управління сутностями «страва» та «категорія», зокрема:

- створення, оновлення та видалення страв (тільки для суперкористувача),
- отримання списку всіх страв або конкретної страви,
- пошук страв за назвою,
- отримання списку страв певної категорії,
- фільтрація та перевірка вхідних даних,
- контроль доступу відповідно до ролей користувачів.

Таким чином, REST API, побудоване за допомогою FastAPI, є ефективною основою для взаємодії між клієнтським та серверним компонентами веб-додатку. Такий підхід дозволяє забезпечити масштабованість, розширюваність та зручність у використанні, що є критично важливим для проєктів, пов’язаних з керуванням контентом, зокрема у ресторанній сфері.

1.5. Робота з базами даних

База даних є центральним елементом будь-якого веб-додатку, оскільки саме вона відповідає за збереження, пошук, оновлення та видалення даних. У сучасній веб-розробці використовуються як реляційні, так і нереляційні системи керування базами даних (СКБД). Вибір конкретної СКБД залежить від вимог проєкту до структурованості, транзакційності, масштабованості, типів даних, що обробляються, і способу доступу до них.

У рамках реалізації цього проєкту було обрано реляційну СКБД PostgreSQL. Основною перевагою PostgreSQL є повна підтримка ACID-властивостей, що гарантує надійність збереження даних у багатокористувацькому середовищі. Вона також підтримує складні типи даних (JSON, масиви, UUID), надає потужні засоби індексації (B-Tree, Hash, GIN, GiST), дозволяє будувати гнучкі SQL-запити та оптимізує продуктивність завдяки планувальнику запитів.

Для взаємодії веб-додатку з базою даних у Python застосовується ORM-бібліотека SQLAlchemy **[14]**, яка забезпечує об’єктно-реляційне відображення. Це означає, що таблиці бази даних представляються як класи Python, а записи — як об’єкти, з якими можна працювати за допомогою знайомих інструментів об’єктно-орієнтованого програмування. Наприклад, створення таблиці страв у SQLAlchemy виглядає так:

Лістинг 1.5.1. Модель страви

```
class Dish(Base):
   __tablename__ = "dishes"
   id = Column(String, primary_key=True, index=True)
   name = Column(String, nullable=False)
   description = Column(String, nullable=True)
   price = Column(Float, nullable=False)
   image_url = Column(String, nullable=True)
   category_id = Column(String, ForeignKey("categories.id"))
   category = relationship("Category")
```

Для застосування змін до структури бази даних використовується утиліта Alembic, яка інтегрується з SQLAlchemy. Alembic дозволяє створювати, застосовувати та відслідковувати міграції, тобто зміни в схемі бази даних, що необхідні під час еволюції проєкту **[15]**. Це забезпечує контроль версій бази даних і мінімізує ризик втрати даних при оновленнях.

Таким чином, використання PostgreSQL у поєднанні з SQLAlchemy і Alembic дозволяє створити стабільну, розширювану та підтримувану систему зберігання даних, що повністю відповідає вимогам до реалізації надійного веб-додатку з багатою структурою, суворими правилами доступу та складною бізнес-логікою.

Бази даних є основою веб-додатку. PostgreSQL забезпечує реляційне зберігання інформації з підтримкою транзакцій, складних запитів, індексації. Зв’язок з Python реалізовано через ORM SQLAlchemy, що дозволяє працювати з моделями як з об’єктами мови Python.

Міграції реалізуються через Alembic — це дозволяє змінювати структуру бази без втрати даних, підтримувати цілісність під час розробки. PostgreSQL обрано також за надійність, активну підтримку спільнотою та можливість роботи в хмарному середовищі.

РОЗДІЛ 2
ПРОЄКТУВАННЯ ВЕБ-ДОДАТКУ

2.1. Аналіз предметної області та вимог

У сучасному ресторанному бізнесі ефективність обслуговування клієнтів значною мірою залежить від оперативності оновлення меню, зручності його представлення та легкості взаємодії з інформацією про страви. Ручне або застаріле управління меню уповільнює процеси, ускладнює оновлення інформації та знижує загальну конкурентоспроможність закладу. Саме тому виникає потреба в автоматизації та цифровізації цього процесу шляхом впровадження веб-застосунку для управління меню.

Метою проєкту є створення зручної, функціональної та безпечної інформаційної системи, що забезпечує двосторонню взаємодію між адміністративним персоналом ресторану та кінцевими користувачами (клієнтами). Основна ідея полягає у розробці веб-додатку, який дозволить адміністраторам легко керувати стравами, категоріями меню, їхнім описом та зображеннями, а клієнтам — швидко отримувати актуальну інформацію про наявні страви в зручному форматі, доступному з будь-якого пристрою.

Ключові ролі користувачів

У межах предметної області було визначено дві основні групи користувачів, кожна з яких має свої функціональні обов’язки та рівень доступу до системи. Адміністраторами виступають менеджери ресторану, які відповідають за наповнення та оновлення меню, контроль за його структурою, змістом і правильним відображенням на сайті. Вони мають повний доступ до адміністративної панелі й можуть здійснювати всі CRUD-операції. Іншу групу користувачів становлять клієнти — відвідувачі ресторану або користувачі веб-сайту, які отримують доступ до візуального представлення меню. Вони можуть ознайомлюватися з переліком страв, їхнім описом, цінами та зображеннями перед тим, як зробити замовлення. Для

кожної з цих ролей були проаналізовані їхні потреби, основні функції та специфіка взаємодії із системою, що дозволило врахувати вимоги до доступу та функціоналу під час реалізації веб-додатку.

Можливості адміністратора

Користувач із роллю адміністратора повинен мати доступ до повного набору інструментів для керування вмістом меню. Основні функції включають:

- створення нових позицій меню з повною інформацією (назва, опис, ціна, категорія, зображення);
- редагування вже наявних страв;
- видалення страв у разі необхідності;
- створення, редагування та видалення категорій меню;
- завантаження зображень страв із подальшим зберіганням через зовнішній хмарний сервіс (наприклад, Cloudinary);
- контроль за структурою меню та наповненням у реальному часі через захищену адміністративну панель.

Ці функції є критично важливими для підтримки актуальності даних, зручної модерації вмісту та організації щоденної роботи персоналу ресторану.

Можливості клієнта

На противагу адміністраторам, клієнти мають обмежений, але достатній рівень доступу. Основною метою є ознайомлення з меню та формування враження про заклад. Відповідно, користувач із цією роллю повинен мати змогу:

- переглядати список усіх доступних страв, структурованих за категоріями;
- бачити назву кожної страви, її короткий опис, ціну та зображення;
- здійснювати пошук за назвою чи перегляд страв певної категорії;
- отримувати актуальну інформацію незалежно від пристрою — з мобільного телефону, планшета чи комп’ютера.

Редагування, додавання або видалення даних клієнтам заборонено. Це відповідає принципу розмежування доступу згідно з ролями.

Системні вимоги

Розробка веб-додатку повинна відповідати чітко визначеним вимогам, які поділяються на функціональні та нефункціональні. Функціональні вимоги описують, що саме система повинна виконувати — її поведінку, логіку та можливості, пов’язавзаємні з одією користувача з інтерфейсом. Нефункціональні вимоги, своєю чергою, визначають якісні характеристики системи, тобто її продуктивність, безпеку, масштабованість, адаптивність тощо.

Функціональні вимоги

На підставі аналізу предметної області було визначено перелік основних функціональних вимог, які необхідно реалізувати для забезпечення повноцінної роботи веб-застосунку. Однією з ключових вимог є підтримка ролей користувачів. Система повинна реалізовувати рольову модель, що передбачає наявність двох основних рівнів доступу: адміністратора (superuser) та клієнта. Адміністратор отримує повний доступ до інструментів управління вмістом, тоді як клієнт має змогу лише переглядати інформацію. Це забезпечує безпеку системи та контроль над діями користувачів.

Наступною важливою вимогою є реалізація механізмів авторизації та автентифікації. Користувачі повинні мати змогу здійснювати вхід до системи з подальшим визначенням своїх прав доступу. Для цього застосовуються технології JWT (JSON Web Token) або готові рішення, як-от FastAPI Users. Завдяки цьому доступ до адміністративної панелі отримують виключно авторизовані особи.

Також критичною функціональністю є підтримка CRUD-операцій для основних сутностей меню. Система повинна дозволяти адміністраторам створювати, редагувати, переглядати та видаляти записи про страви й категорії, щоб оперативно оновлювати меню відповідно до потреб закладу.

Для підвищення зручності користування меню, необхідно реалізувати пошук, фільтрацію та сортування. Користувачі повинні мати можливість здійснювати пошук страв за назвою, фільтрувати їх за категоріями або сортувати за такими параметрами, як ціна чи алфавітний порядок. Це значно полегшує навігацію.

Крім того, усі вхідні дані, що надходять до системи через форми або API-запити, мають бути обов’язково перевірені. Валідація даних гарантує відповідність установленим правилам, таким як типи полів, допустима довжина рядків, числові межі, а також обов’язковість окремих значень. Це підвищує стабільність сервісу та знижує ризик помилок.

Ще однією важливою вимогою є інтеграція із зовнішніми сервісами для зберігання зображень, зокрема з Cloudinary. Це дозволяє розвантажити сервер, зменшити обсяг локального сховища та забезпечити швидке відображення зображень на стороні клієнта.

Нарешті, API, який забезпечує зв’язок між клієнтською та серверною частинами, повинен автоматично генерувати документацію у форматі OpenAPI. Це спрощує процес розробки, тестування та інтеграції з іншими сервісами, дозволяючи розробникам швидше орієнтуватися в структурі запитів і відповідей.

Нефункціональні вимоги

Окрім реалізації функціональних можливостей, веб-додаток повинен також відповідати низці нефункціональних вимог, які визначають його якість, надійність та відповідність сучасним технологічним стандартам. Однією з ключових характеристик є масштабованість. Архітектура системи має бути гнучкою та легко адаптованою для впровадження в інших закладах харчування — як у межах окремих ресторанів, так і в рамках цілих мереж. Це вимагає чіткої модульної структури проєкту та використання стандартизованого API, що дозволяє безперешкодно розширювати функціонал без порушення роботи існуючих компонентів.

Ще одним важливим аспектом є безпека системи. Уся передача даних між клієнтською та серверною частинами повинна здійснюватися виключно через захищене з’єднання за протоколом HTTPS. Особливу увагу слід приділяти обробці персональних даних користувачів, автентифікаційним токенам та адміністративним функціям. Для забезпечення безпеки необхідно впроваджувати механізми хешування паролів, захист від SQL-ін’єкцій та перевірку прав доступу до кожного ресурсу.

Також веб-додаток повинен бути адаптивним і доступним для користувачів з різних пристроїв. Це означає, що інтерфейс має коректно відображатися на екранах різних розмірів — від великих моніторів до мобільних телефонів. Для досягнення цього застосовується адаптивна верстка (responsive design), що гарантує зручність перегляду контенту незалежно від типу пристрою.

Крім того, важливою вимогою є дотримання принципів REST під час побудови серверної частини системи. Реалізація REST API передбачає логічну та послідовну організацію ендпоінтів, використання стандартних HTTP-методів (GET, POST, PUT, DELETE), а також підтримку відповідних статус-кодів. Такий підхід значно спрощує інтеграцію з іншими сервісами, прискорює розробку клієнтських застосунків та полегшує тестування.

У сукупності всі ці вимоги формують цілісну концепцію веб-застосунку, який відповідає сучасним вимогам до якості програмного забезпечення, забезпечує надійну та ефективну роботу в умовах реального ресторанного бізнесу, а також є придатним до масштабування та подальшого розвитку.

2.2. Проєктування архітектури веб-додатку

Архітектура веб-додатку TomYum ґрунтується на принципах модульності, розділення відповідальності та багаторівневої взаємодії між компонентами. Це забезпечує надійність, зручність супроводу, розширюваність та відповідність сучасним стандартам веб-розробки. Головною ідеєю є реалізація схеми “тонкий

клієнт — потужний сервер", де бізнес-логіка повністю реалізується на бекенді, а клієнт займається лише взаємодією з користувачем.

Розробка відбувалася з урахуванням найкращих практик, зокрема принципів чистої архітектури, яка передбачає поділ проєкту на незалежні шари. Це дозволяє зменшити зв'язаність між модулями, спрощує тестування та забезпечує гнучкість при внесенні змін.

Клієнтська частина

Клієнтський інтерфейс побудований на основі React — популярної JavaScript-бібліотеки, що дозволяє створювати односторінкові додатки (SPA). Основна роль клієнта полягає у відображенні вмісту, забезпеченні навігації по сторінках, обробці взаємодії користувача (натискання, введення, перемикання) та передачі запитів до серверної частини за допомогою HTTP-протоколу.

У проєкті використано компонентно-орієнтований підхід, тобто весь інтерфейс поділено на окремі незалежні компоненти: заголовки, списки страв, картки з описом, модальні вікна для авторизації тощо. Кожен компонент має власну логіку, стилізацію (через Tailwind CSS) та життєвий цикл. Це дозволяє легко повторно використовувати компоненти, масштабувати систему, оновлювати окремі елементи без впливу на інші частини інтерфейсу.

Для збереження стану застосунку використано React-хуки (useState, useEffect тощо), а також бібліотеки для маршрутизації (react-router-dom), що дозволяє реалізовувати динамічну зміну контенту без перезавантаження сторінки.

Також клієнт реалізує:

- адаптивний дизайн для підтримки мобільних пристроїв;
- зручну навігацію;
- форми для входу/реєстрації;
- інтеграцію з backend API для отримання меню.

Серверна частина

Серверна частина веб-додатку реалізована з використанням FastAPI — сучасного веб-фреймворку для мови Python, який підтримує асинхронну обробку запитів, високу продуктивність і строгість типізації. FastAPI є одним із найбільш ефективних інструментів для створення RESTful API, оскільки дозволяє швидко розгортати маршрути для обробки HTTP-запитів, описувати вхідні та вихідні дані за допомогою моделей Pydantic, реалізовувати механізми авторизації й автентифікації, а також автоматично генерувати інтерактивну документацію за стандартом OpenAPI у форматах Swagger UI та ReDoc.

Проєкт побудований за принципом модульності, що забезпечує чітке розмежування відповідальностей між різними частинами системи. Зокрема, логіка застосунку структурована на окремі функціональні блоки. Роутери (routers) відповідають за обробку вхідних HTTP-запитів і визначають маршрути (наприклад, для створення, оновлення, видалення або отримання інформації). Схеми (schemas) описують формат даних, які приймаються від користувача та повертаються у відповідь, і реалізуються на основі Pydantic для забезпечення автоматичної валідації. Репозиторії (repositories) містять функції взаємодії з базою даних, виконуючи всі необхідні запити до сховища за допомогою ORM SQLAlchemy.

Бізнес-логіка зосереджена в сервісах (services), які виконують складні операції, зокрема обробку умов, роботу із зовнішніми сервісами, такими як Cloudinary, і об’єднання кількох дій у межах одного сценарію. Опис структури бази даних винесено до окремого модуля models, де визначено таблиці та зв’язки між ними. Конфігураційні налаштування системи, зокрема підключення до бази даних, змінні середовища та утилітарні функції, зібрані в модулі core.

Такий структурований підхід дозволяє досягти високої якості коду, полегшує тестування, розширення проєкту та підтримку на етапі експлуатації.

База даних

Для зберігання даних про користувачів, страви, категорії та інші об'єкти у веб-додатку використовується реляційна система керування базами даних PostgreSQL. Вибір саме цієї СКБД зумовлений її високою надійністю, стабільною продуктивністю, підтримкою транзакцій, складних SQL-запитів, індексації та чітким механізмом побудови зв’язків між таблицями, що критично важливо для структурованих даних у складних інформаційних системах.

Для взаємодії з базою даних на стороні сервера застосовується ORM-бібліотека SQLAlchemy, яка дозволяє моделювати таблиці у вигляді Python-класів. Завдяки цьому програміст оперує об’єктами замість сирих SQL-запитів, що значно спрощує розробку, підвищує читабельність коду та знижує ризик помилок. Усі основні операції — створення, оновлення, видалення та отримання даних — реалізуються через об’єктно-орієнтований підхід.

Для керування міграціями, тобто змінами у структурі бази даних у процесі розробки, використовується Alembic — спеціалізований інструмент, який інтегрується з SQLAlchemy. Він дозволяє фіксувати історію змін у вигляді окремих версій, автоматизує процес оновлення схеми бази даних і допомагає уникнути втрати даних під час модифікацій.

У структурі бази даних передбачено окремі таблиці для зберігання інформації про користувачів, категорії страв і самі страви, які містять такі поля, як назва, опис, ціна та посилання на зображення. Усі ці таблиці логічно пов’язані між собою за допомогою зовнішніх ключів, що забезпечує цілісність та узгодженість даних.

Структура проєкту

Серверна частина проєкту має структуру, яка відображає розподіл функціональності за каталогами. Такий підхід дозволяє чітко визначати місце кожного модуля у системі та спрощує підтримку коду.

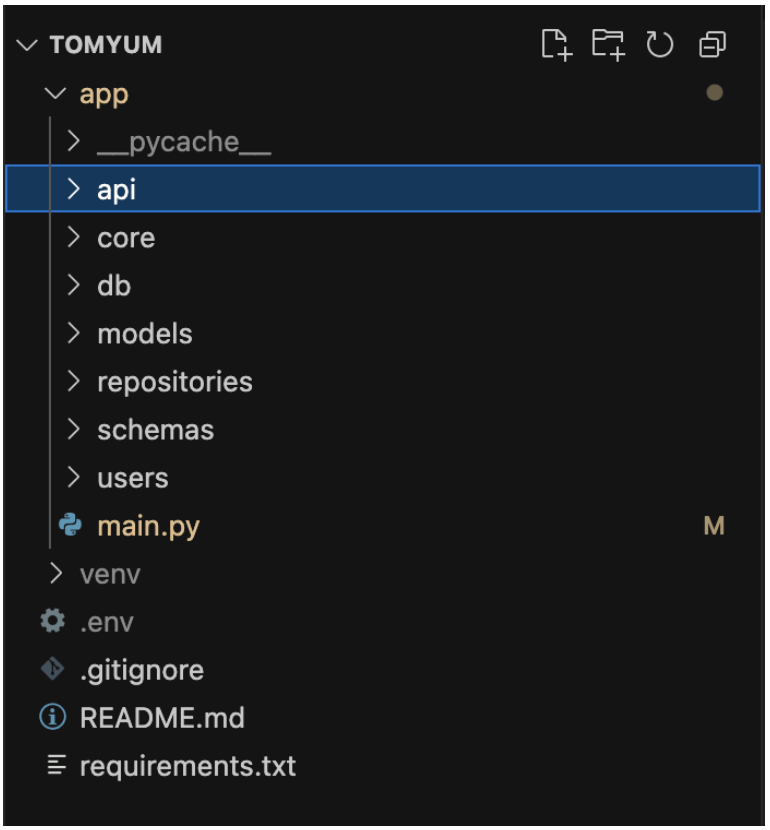

Рисунок 2.1. Структура серверної частини проєкту TomYum

Джерело: розроблене автором

Типова послідовність взаємодії користувача з веб-додатком TomYum реалізується за класичною схемою клієнт-серверної архітектури. Усе починається з того, що користувач відкриває веб-додаток у браузері, після чого клієнтська частина, реалізована на базі React, завантажується із сервера. React відповідає за ініціалізацію інтерфейсу та надання користувачу доступу до функціональних елементів, таких як меню, кнопки навігації, вікна авторизації тощо.

Після ініціалізації інтерфейсу клієнт надсилає HTTP-запити до серверної частини веб-додатку. Наприклад, якщо користувач відкриває сторінку з меню ресторану, React автоматично формує запит до API-сервера (наприклад, на ендпоінт /dishes), щоб отримати актуальний список страв.

На сервері цей запит приймається FastAPI — асинхронним фреймворком для Python. Якщо доступ до відповідного ресурсу потребує авторизації (наприклад, для

адміністративних функцій), FastAPI виконує перевірку токену користувача, що засвідчує його особу та роль (через JWT або інші механізми). Після цього сервер обробляє бізнес-логіку, пов’язану з даним запитом, і звертається до бази даних.

Дані, що зберігаються у PostgreSQL, запитуються за допомогою ORM-бібліотеки SQLAlchemy. Вона дозволяє зчитувати дані у вигляді Python-об’єктів, які потім серіалізуються у формат JSON і повертаються у відповіді на запит.

Ця відповідь у форматі JSON надсилається назад на клієнтську частину. React отримує дані, оновлює відповідні компоненти інтерфейсу й відображає користувачу оновлену інформацію — наприклад, список страв у меню, деталі обраної страви або повідомлення про успішне виконання дії.

Таким чином, весь процес — від запиту користувача до відображення результату — побудований на чіткій та ефективній взаємодії між клієнтом і сервером, що дозволяє забезпечити швидкодію, надійність та зручність користування додатком.

Переваги обраної архітектури

Архітектура веб-додатку, що базується на принципах модульності, REST-підходу та асинхронної обробки запитів, має низку важливих переваг, які визначають її ефективність, гнучкість та придатність до реального використання в умовах сучасного ресторанного бізнесу.

По-перше, така архітектура забезпечує масштабованість. Завдяки чіткому розділенню функціональних блоків між клієнтською та серверною частинами, у проєкт можна легко додавати нові модулі або масштабувати окремі компоненти (наприклад, збільшувати обчислювальні ресурси на боці сервера чи змінювати інтерфейс користувача без потреби втручання у бекенд).

По-друге, архітектура відзначається гнучкістю, оскільки кожен її компонент розроблено як незалежну частину, що дозволяє змінювати, оновлювати чи навіть

повністю замінювати окремі модулі без ризику порушення загальної цілісності системи. Це особливо важливо при довготривалому життєвому циклі проєкту.

Важливою перевагою є також безпека. Централізоване управління авторизацією та правами доступу, реалізація захищеної передачі даних через HTTPS, а також використання сучасних механізмів автентифікації на основі токенів (JWT) дозволяють гарантувати захист персональних даних і захист від несанкціонованого доступу.

Ще однією суттєвою перевагою є відповідність сучасним стандартам розробки веб-сервісів, зокрема таким як REST, SOLID-принципи об'єктно-орієнтованого програмування, та відкритий стандарт OpenAPI для документації. Це полегшує інтеграцію з іншими системами, забезпечує зрозумілу структуру API та сприяє прозорості розробки.

У сукупності всі ці переваги роблять обрану архітектурну модель надійною основою для створення стабільного, масштабованого та зручного у використанні веб-додатку для управління ресторанним меню.

2.3. Моделювання бази даних

Проєктування бази даних є одним з найважливіших етапів під час створення будь-якого інформаційного веб-додатку, адже саме від структури збереження даних залежить стабільність роботи системи, швидкість обробки запитів, цілісність інформації та можливість масштабування у майбутньому. У випадку розробки застосунку для управління ресторанним меню, база даних має забезпечувати швидкий доступ до даних про страви, їхні категорії, а також ефективне управління правами доступу через систему користувачів.

На основі попереднього аналізу предметної області та функціональних вимог було визначено перелік основних сутностей, які мають бути представлені в структурі бази даних: користувачі , категорії та страви. Для кожної з них були визначені

необхідні атрибути, а також встановлено логічні зв’язки, які дозволяють ефективно організувати взаємодію між таблицями.

У веб-додатку TomYum користувач є ключовою сутністю, адже саме він визначає рівень доступу до функціоналу системи. Усі дані про користувачів зберігаються в таблиці User, структура якої передбачає зберігання унікального ідентифікатора (id), електронної пошти (email) як логіна для авторизації, захешованого пароля (hashed_password) для безпечного зберігання, а також логічних полів is_active та is_superuser, які вказують, відповідно, на активний статус облікового запису та наявність розширених прав адміністратора. Така модель забезпечує гнучке управління доступом, дозволяючи адміністраторам мати повний контроль над вмістом меню, а звичайним користувачам — лише переглядати доступну інформацію.

Іншою важливою сутністю є категорія (Category), яка служить для логічного групування страв — наприклад, за типом: «Супи», «Основні страви», «Напої» тощо. У таблиці зберігаються унікальний ідентифікатор категорії (id) та її назва (name), яка відображається у клієнтському інтерфейсі. Категорії організовують меню у зручний для користувача вигляд і дозволяють ефективно структурувати перелік страв.

Центральне місце в системі займає сутність страви (Dish), оскільки саме з нею найчастіше взаємодіє користувач. Таблиця Dish включає поля для унікального ідентифікатора (id), назви страви (name), її опису (description), вартості (price), посилання на зображення (image_url), яке зберігається в хмарному сховищі (наприклад, Cloudinary), а також зовнішнього ключа category_id, що забезпечує зв’язок із відповідною категорією.

Між зазначеними сутностями реалізовано низку логічних зв’язків. Основним є зв’язок "один до багатьох" між категорією і стравами: кожна категорія може містити кілька страв, але кожна страва належить лише до однієї категорії. Крім того, логіка роботи з ролями користувача реалізована в межах таблиці User завдяки полям is_superuser та is_active, що дозволяє ефективно керувати доступом без створення

окремої таблиці для ролей. Сутності User, Category та Dish водночас є незалежними — це забезпечує модульність структури бази даних, спрощує її підтримку та дає змогу розвивати функціонал окремо.

Всі зазначені сутності реалізовано за допомогою ORM SQLAlchemy — це дозволяє описувати таблиці як Python-класи, працювати з ними через об’єктно-орієнтований підхід, а також забезпечує автоматизацію роботи з базою даних без потреби у прямому написанні SQL-запитів. Такий підхід гарантує баланс між зручністю розробки, гнучкістю розширення та стабільністю збереження даних, а також забезпечує повну підтримку цілісності інформації та контроль її структури на всіх етапах життєвого циклу додатку.

Використання ORM-моделей

На основі вищенаведеної логічної структури реалізуються класи-моделі в SQLAlchemy, які використовуються для роботи з базою даних у серверній частині застосунку. Кожна модель відповідає окремій таблиці та містить опис полів, типів, обмежень (nullable, unique, foreign key) тощо. Це дозволяє взаємодіяти з базою даних на рівні Python-коду, не використовуючи безпосередньо SQL-запити.

Такий підхід дозволяє досягти балансу між гнучкістю налаштування та стабільністю структури даних. Крім того, завдяки ORM-підходу розробники можуть швидко змінювати структуру даних, створювати тестові записи, виконувати складні запити й забезпечувати цілісність системи.

2.4. Проєктування API та логіки доступу

Після моделювання структури бази даних одним із найважливіших етапів створення веб-додатку є розробка API — інтерфейсу прикладного програмування, який слугує посередником між клієнтською та серверною частинами застосунку. Саме API визначає, як клієнт може отримувати, передавати, змінювати або видаляти дані, не маючи прямого доступу до бази даних. У цьому проєкті реалізація API

відбулася за допомогою фреймворку FastAPI, який дозволяє створювати сучасні, швидкі та надійні REST-сервіси.

Загальний підхід до побудови API

У веб-додатку TomYum інтерфейс прикладного програмування (API) розроблено відповідно до архітектурного стилю REST, що дозволяє ефективно реалізувати всі основні операції над даними через стандартизовані HTTP-методи. Такі методи, як GET, POST, PUT і DELETE, використовуються для виконання CRUD-операцій над ключовими сутностями системи: стравами, категоріями та користувачами. Кожен маршрут або ендпоінт API відповідає окремій операції, що може бути виконана над певним ресурсом, наприклад, отримання списку страв, створення нової категорії або редагування наявного запису.

Особливістю реалізованого API є врахування рівнів доступу користувачів. У системі чітко розділено функціональність між адміністраторами (суперкористувачами) та звичайними клієнтами. Зокрема, публічний доступ надається всім користувачам — як авторизованим, так і неавторизованим — і дозволяє лише читати дані: переглядати список страв, деталі окремих позицій меню або категорій. Натомість адміністративний доступ доступний лише для користувачів із правами суперкористувача й дозволяє виконувати такі операції, як створення, редагування та видалення записів, а також завантаження зображень.

Забезпечення цього розмежування досягається завдяки механізмам авторизації у FastAPI, зокрема через залежності, реалізовані за допомогою Depends(). Ці залежності перевіряють, чи має поточний користувач достатньо прав для виконання певної дії, перш ніж дозволити доступ до відповідного ендпоінту.

Кожна сутність у системі має реалізований повний набір CRUD-функцій. Для страв, наприклад, реалізовано такі маршрути: запит GET /dishes повертає список усіх страв; GET /dishes/{id} дозволяє отримати детальну інформацію про конкретну страву; POST /dishes створює нову страву (доступне лише адміністратору); PUT /dishes/{id} оновлює наявну інформацію; DELETE /dishes/{id} видаляє обрану

страву. Аналогічно, для категорій реалізовано маршрути GET для отримання списку та окремої категорії, а також POST для створення нової (з адміністративними правами).

Таким чином, API у веб-додатку TomYum забезпечує повноцінну взаємодію між клієнтською та серверною частинами, підтримує гнучке керування правами доступу й реалізує всі необхідні функції для повсякденної роботи як адміністратора, так і кінцевого користувача.

Приклад реалізації ендпоінту створення нової страви подано нижче:

Лістинг 2.4.1. Ендпоінт створення страви

```
@router.post("/", response_model=DishRead,
dependencies=[Depends(require_superuser)])
async def create_dish(
   name: str = Form(...),
   description: str = Form(None),
   price: float = Form(...),
   category_id: str = Form(...),
   image: UploadFile = File(...),
   session: AsyncSession = Depends(get_async_session)
):
   image_url = upload_image(image)
   dish_data = DishCreate(
       name=name,
       description=description,
       price=price,
       image_url=image_url,
       category_id=category_id
   )
   return await dish_repo.create_dish(session, dish_data)
```

У наведеному прикладі реалізовано обробку POST-запиту для створення нової страви у веб-додатку. Доступ до цього ендпоінту надається виключно користувачам із правами суперкористувача, що реалізується за допомогою залежності Depends(require_superuser), яка перевіряє роль поточного користувача.

Параметри нової страви, такі як назва, опис, ціна та ідентифікатор категорії, передаються на сервер через об’єкти типу Form(...), тоді як зображення надсилається як файл через File(...). Отримане зображення обробляється функцією upload_image, яка завантажує файл у хмарний сервіс (наприклад, Cloudinary) і повертає URL, що зберігається у базі даних разом з іншими даними страви.

Усі отримані дані об’єднуються у схему DishCreate, реалізовану через Pydantic, яка автоматично перевіряє правильність типів, обов’язковість полів та інші обмеження. Після успішної валідації дані передаються в репозиторій, де за допомогою SQLAlchemy здійснюється створення нового запису в базі даних.

Такий підхід дозволяє адміністратору зручно створювати нові страви за допомогою форми із прикріпленням зображення, водночас забезпечуючи ізоляцію логіки обробки медіафайлів, чіткий контроль доступу та дотримання принципу єдиної відповідальності (Single Responsibility Principle) згідно з SOLID-підходом до програмування **[19]**.

Репозиторії та сервіси

Для підтримання чистоти коду та дотримання принципів розділення відповідальностей (Separation of Concerns), логіка доступу до бази даних винесена у репозиторії (директорія repositories/). Тут зберігаються функції, які безпосередньо виконують SQL-запити або взаємодіють із моделями SQLAlchemy. Наприклад:

- створення нового запису;
- отримання списку елементів;
- фільтрація за категорією;
- оновлення чи видалення.

Окремо від репозиторіїв розміщено сервіси (директорія services/), які відповідають за бізнес-логіку. Сервіси можуть комбінувати кілька дій репозиторію, перевіряти умови (наприклад, існування категорії), обробляти зображення,

взаємодіяти з Cloudinary та інше. Таким чином, у коді маршруту залишаються лише найнеобхідніші виклики, що значно підвищує читабельність і тестованість.

Валідація через Pydantic-схеми

У веб-додатку обмін даними між клієнтською та серверною частинами організовано за допомогою схем, створених на основі бібліотеки Pydantic. Ці схеми зберігаються у директорії schemas/ і відіграють ключову роль у визначенні структури як вхідних, так і вихідних даних. Кожна схема чітко описує, які поля мають бути присутніми у запиті або відповіді, які з них є обов’язковими, які повинні бути унікальними або відповідати певному типу (наприклад, числовому значенню).

Завдяки цьому підходу валідація даних виконується автоматично. Якщо клієнт надсилає запит, що не відповідає визначеним у схемі правилам, FastAPI самостійно перехоплює цей запит і повертає відповідь з описом помилки, пояснюючи, які саме дані є некоректними.

У проєкті застосовуються такі схеми: DishCreate — для опису структури даних, що надсилаються при створенні нової страви; DishRead — для структурування відповіді, яку клієнт отримує після запиту про страву; CategoryCreate і CategoryRead — для аналогічної роботи з категоріями. Така організація забезпечує надійність, передбачуваність і прозорість у взаємодії між клієнтом і сервером

2.5. Інтеграція з хмарними сервісами та робота з зображеннями

Одним із важливих елементів у веб-додатку для ресторану є зображення страв. Візуальне представлення підвищує привабливість меню, полегшує сприйняття інформації користувачами та дозволяє швидше приймати рішення щодо замовлення. Для цього важливо реалізувати не лише збереження посилання на зображення, а й повноцінну обробку файлів: завантаження, перевірку формату, збереження у зовнішньому середовищі, а також доступ до них через URL.

У межах даного проєкту було обрано інтеграцію з **Cloudinary** — популярним хмарним сервісом для обробки зображень і відео. Цей сервіс забезпечує зручне API для завантаження файлів, їх оптимізації, масштабування, а також автоматичну генерацію URL-адрес для віддаленого збереження. Такий підхід дозволяє не зберігати важкі медіафайли безпосередньо на сервері, що суттєво знижує навантаження та забезпечує високу швидкість доступу до зображень із будь-якої точки світу.

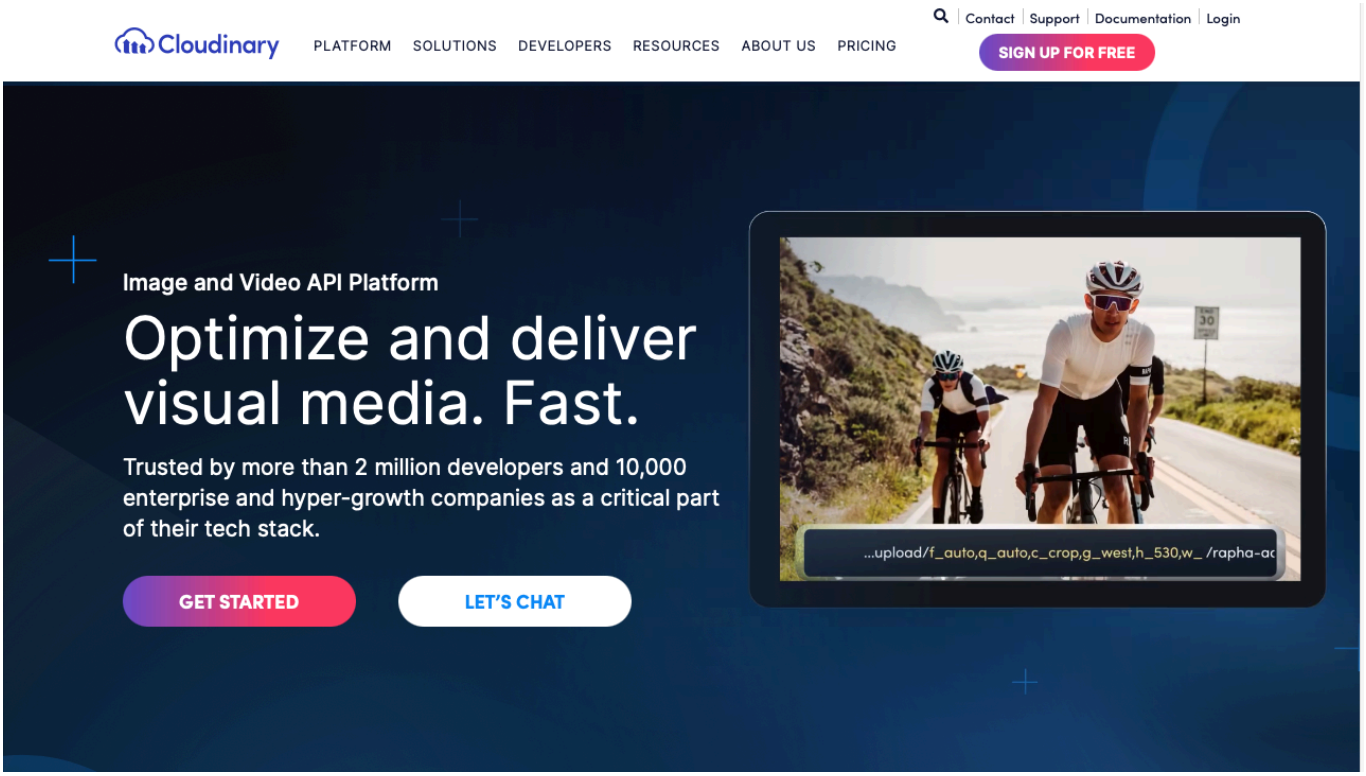

Рисунок 2.2. Головна сторінка Cloudinary

Джерело: https://cloudinary.com

Інтеграція веб-додатку з сервісом Cloudinary відкриває низку значних переваг, які стосуються зберігання, обробки та доставки зображень. Насамперед, Cloudinary надає можливість зберігати всі медіафайли у хмарному середовищі, а не локально на сервері. Це зменшує навантаження на інфраструктуру застосунку, підвищує продуктивність та дозволяє зосередити ресурси на обробці бізнес-логіки.

Після завантаження кожне зображення автоматично отримує унікальне посилання (URL), яке зберігається у базі даних. Це забезпечує швидкий і зручний доступ до файлу з будь-якої точки світу без необхідності зберігати його фізично в системі. Такий підхід спрощує структуру даних, а також дозволяє легко відображати зображення в інтерфейсі користувача.

Крім того, Cloudinary самостійно виконує оптимізацію зображень. Сервіс автоматично стискає файли без суттєвої втрати якості, що значно зменшує їхній обсяг. Це, у свою чергу, скорочує час завантаження сторінок, підвищує швидкість роботи додатку та покращує загальний користувацький досвід.

Безпека також є важливою перевагою Cloudinary. Усі дані передаються через захищений протокол HTTPS, що гарантує конфіденційність і цілісність медіаконтенту під час передачі між клієнтом і сервером.

І, нарешті, масштабованість. Cloudinary легко адаптується як до потреб невеликих проєктів, так і до вимог великих систем з високим навантаженням. Це робить його ефективним інструментом для інтеграції в будь-які веб-рішення, включаючи ресторанні застосунки, де важливо швидко, безпечно та якісно обробляти візуальний контент.

2.6. Структура клієнтської частини веб-додатку

Клієнтська частина є важливим компонентом веб-додатку TomYum, оскільки саме вона безпосередньо взаємодіє з кінцевим користувачем — адміністратором або клієнтом. Основна мета клієнтської частини полягає у відображенні структури меню, забезпеченні зручної навігації, надсиланні запитів до API та відображенні відповідей у вигляді зручного й адаптивного інтерфейсу.

Для реалізації клієнтського інтерфейсу обрано React — сучасну бібліотеку JavaScript, яка дозволяє будувати масштабовані інтерфейси на основі компонентної моделі. React є однією з найпопулярніших технологій у сфері frontend-розробки,

оскільки забезпечує швидке оновлення вмісту сторінки без перезавантаження, високу продуктивність та розширювану архітектуру.

Архітектурні особливості клієнтської частини

Клієнтський код організовано у вигляді окремих компонентів, кожен з яких виконує визначену роль. Наприклад, окремі компоненти відповідають за:

- відображення навігаційного меню;
- показ списку страв за категоріями;
- авторизацію/реєстрацію користувача;
- модальні вікна для створення нових записів;
- картки окремих страв;
- фільтрацію, пошук та пагінацію меню.

Кожен компонент є ізольованим, повторно використовуваним та стилізованим за допомогою Tailwind CSS **[16]**, що дозволяє швидко створювати адаптивний інтерфейс без надлишкового CSS-коду.

Логіка взаємодії клієнтської частини з сервером побудована на основі HTTP-запитів, які надсилаються до API, реалізованого у FastAPI. Для цього використовуються як вбудовані інструменти JavaScript (наприклад, fetch), так і спеціалізовані обгортки, що спрощують процес взаємодії з бекендом. Запити надсилаються до відповідних маршрутів API, таких як /api/dishes, /api/categories, /api/auth/login, а відповіді у форматі JSON використовуються для оновлення інтерфейсу у реальному часі.

У React-компонентах активно застосовуються React-хуки, зокрема useState, useEffect і useContext. Вони дозволяють зберігати поточний стан компонента, реагувати на зміни та синхронізувати дані з сервером. Крім того, useContext використовується для збереження глобальної інформації, наприклад, авторизаційного токена користувача (JWT), що спрощує контроль доступу до приватних маршрутів інтерфейсу.

Адаптивність дизайну веб-додатку є важливою складовою користувацького досвіду. Завдяки використанню Tailwind CSS і компонентної архітектури, інтерфейс автоматично підлаштовується під розміри різних пристроїв. Наприклад, навігаційне меню перетворюється у мобільний формат, коли сторінка відкривається на смартфоні, а компоненти — як-от списки страв чи кнопки — змінюють своє розташування і розмір відповідно до ширини екрана. Це дозволяє забезпечити однаково зручне користування веб-додатком як на настільних комп'ютерах, так і на мобільних пристроях.

Файлова структура клієнтської частини організована за функціональними принципами, де кожен блок коду відповідає за окремий компонент чи логіку інтерфейсу. Такий підхід сприяє зручності розробки, масштабуванню додатку та підтримці чистої й зрозумілої архітектури.

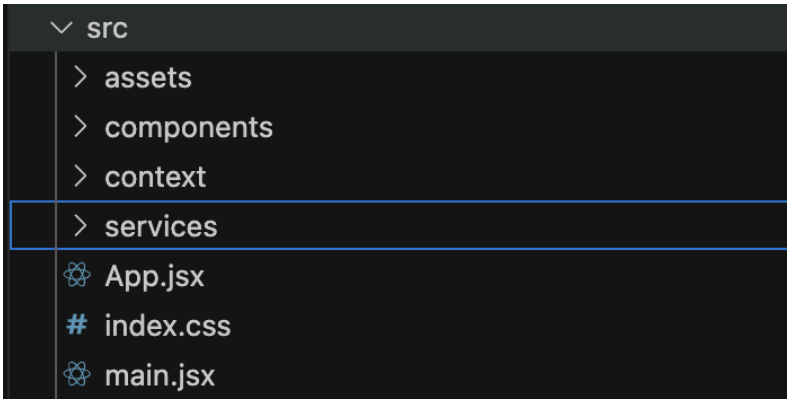

Рисунок 2.3. Структура клієнтської частини веб-додатку

Джерело: розроблене автором

Завдяки впорядкованій файловій структурі клієнтської частини веб-додатку розробник має змогу швидко знаходити, редагувати або розширювати окремі функціональні компоненти, не зачіпаючи загальну логіку додатку. Це значно полегшує підтримку проєкту та впровадження нових можливостей у майбутньому.

Що стосується авторизації користувачів у React, вона реалізована через механізм JWT (JSON Web Token). Після успішного входу в систему сервер генерує

токен, який клієнтська частина зберігає у локальному сховищі браузера (localStorage) або у глобальному контексті застосунку. Цей токен автоматично додається до заголовків усіх подальших HTTP-запитів до API, що дозволяє ідентифікувати користувача та перевіряти його права доступу без необхідності повторної автентифікації.

Інтерфейс веб-додатку динамічно адаптується відповідно до ролі авторизованого користувача. Якщо користувач має статус клієнта, йому доступна лише функція перегляду меню. Натомість, якщо користувач є адміністратором (superuser), у нього з’являється розширена панель керування, яка дозволяє створювати, редагувати та видаляти категорії й страви. Такий підхід забезпечує гнучке керування правами доступу і підвищує безпеку системи.

РОЗДІЛ 3
РЕАЛІЗАЦІЯ ВЕБ-ДОДАТКУ

3.1. Розгортання середовища розробки

Розробка веб-додатку TomYum здійснювалася в умовах, максимально наближених до практичного застосування сучасних технологій у реальних проєктах. На першому етапі було створено ізольоване середовище для зручної роботи з залежностями, базою даних та клієнтською частиною.

Серверна частина веб-додатку була реалізована з використанням мови програмування Python та сучасного фреймворку FastAPI. Такий вибір зумовлений його високою швидкодією, підтримкою асинхронного програмування, а також можливістю автоматичної генерації документації до API, що значно спрощує розробку та тестування. Клієнтська частина побудована за допомогою бібліотеки React, яка дозволяє створювати динамічний та модульний інтерфейс користувача. Для створення SPA (односторінкового застосунку) використано інструмент збірки Vite **[17]**, який забезпечує швидку ініціалізацію проєкту та ефективне управління ресурсами.

До складу системи входять й інші важливі компоненти. У якості основної реляційної бази даних застосовується PostgreSQL, яка забезпечує надійне збереження інформації та підтримку складних SQL-запитів. Для зручної взаємодії з базою даних у коді Python використано бібліотеку SQLAlchemy, що реалізує ORM-підхід. Контроль за змінами структури бази здійснюється за допомогою утиліти Alembic, яка дозволяє створювати та застосовувати міграції.

Зображення страв зберігаються у хмарному середовищі через сервіс Cloudinary, що дозволяє зменшити навантаження на сервер і забезпечити швидкий доступ до медіа. Стилізація інтерфейсу реалізована з використанням Tailwind CSS — утилітарного фреймворку для адаптивної верстки. Для організації автентифікації та

авторизації користувачів інтегровано бібліотеку FastAPI Users **[18]**, яка підтримує роботу з JWT-токенами, хешування паролів та розмежування прав доступу.

3.2. Реалізація серверної частини веб-додатку

Серверна частина веб-додатку TomYum реалізована з використанням FastAPI, сучасного асинхронного фреймворку для мови Python, що забезпечує високу швидкодію, просту структуру коду та автоматичну генерацію документації API. Основними завданнями серверної частини є: обробка HTTP-запитів від клієнта, виконання CRUD-операцій над сутностями (страви, категорії, користувачі), контроль доступу відповідно до ролей, валідація даних та взаємодія з базою даних.

Підключення FastAPI і створення застосунку

Серверна частина веб-додатку TomYum має чітко структуровану точку входу файл main.py, у якому ініціалізується об’єкт FastAPI, підключаються основні маршрути, конфігурації CORS і middleware. Це забезпечує коректне завантаження застосунку, а також налаштування середовища для обробки запитів.

У фрагменті нижче наведено приклад реалізації основного файлу запуску:

Лістинг 3.2.1. Код файлу main.py

```
@asynccontextmanager
async def lifespan(app: FastAPI):
    await create_db_and_tables()
    yield

app = FastAPI(lifespan=lifespan)

app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:5173"],
    allow_methods=["*"],
    allow_headers=["*"],
    allow_credentials=True,
)
```

```
app.include_router(fastapi_users.get_auth_router(auth_backend),
prefix="/auth/jwt", tags=["auth"])
app.include_router(fastapi_users.get_register_router(UserRead, UserCreate),
prefix="/auth", tags=["auth"])
app.include_router(fastapi_users.get_users_router(UserRead, UserUpdate),
prefix="/users", tags=["users"])
app.include_router(dishes.router)
app.include_router(categories.router)
```

Усі маршрути згруповані у модулі api/routers/, де кожен файл відповідає за окрему частину функціональності (наприклад, dishes.py — для страв, categories.py — для категорій, auth.py — для автентифікації).

Реалізація моделей

Для опису структури бази даних використано ORM-моделі бібліотеки SQLAlchemy. Кожна таблиця в базі даних представлена у вигляді Python-класу, що містить поля, типи даних, зв’язки між таблицями та інші обмеження. Наприклад, модель Dish описує інформацію про страву:

Лістинг 3.2.2. Код файлу dish.py

```
from sqlalchemy import Column, String, ForeignKey, Float
from sqlalchemy.orm import relationship
from app.db.session import Base

class Dish(Base):
    __tablename__ = "dishes"
    id = Column(String, primary_key=True, index=True)
    name = Column(String, nullable=False)
    description = Column(String, nullable=True)
    price = Column(Float, nullable=False)
    image_url = Column(String, nullable=True)
    category_id = Column(String, ForeignKey("categories.id"))
    category = relationship("Category")
```

Модель включає в себе основні атрибути страви — назву, опис, ціну, зображення та зв’язок з категорією, до якої вона належить.

Для кожної сутності створено відповідні Pydantic-схеми, які визначають, які поля є обов’язковими або необов’язковими, а також відповідають за автоматичну валідацію даних, що надходять у запитах.

Лістинг 3.2.3. Код файлу schemas/dish.py

```
class DishBase(BaseModel):
    name: str
    description: Optional[str] = None
    price: float
    image_url: Optional[str] = None
    category_id: str
```

Ці схеми дозволяють FastAPI не тільки проводити перевірку вхідних даних, а й автоматично генерувати документацію у форматі OpenAPI, що суттєво пришвидшує інтеграцію та тестування.

Щоб забезпечити чистоту коду та логічне розділення відповідальностей, усі функції, пов’язані з базою даних, винесені в окремі модулі — репозиторії. Це дозволяє зберігати, оновлювати, видаляти чи фільтрувати дані, не дублюючи логіку в контролерах.

Лістинг 3.2.4. Код файлу repositories/dish.py

```
async def create_dish(session: AsyncSession, data: DishCreate) -> Dish:
    new_dish = Dish(
        id=str(uuid4()),
        name=data.name,
        description=data.description,
        price=data.price,
        image_url=data.image_url,
        category_id=data.category_id
    )
    session.add(new_dish)
    await session.commit()
    await session.refresh(new_dish)
    return new_dish
```

Використання асинхронної сесії `AsyncSession` дозволяє забезпечити високу продуктивність і паралельну обробку запитів.

Роутери та ендпоінти

Кожна функціональність має власний модуль маршрутів. Наприклад, у routers/dishes.py знаходяться всі ендпоінти, пов'язані зі стравами. Один із них — видалення страви — реалізовано наступним чином:

Лістинг 3.2.5. Код файлу routes/dish.py

```
@router.delete("/{dish_id}", status_code=status.HTTP_204_NO_CONTENT,
dependencies=[Depends(require_superuser)])
async def delete_dish(dish_id: str, session: AsyncSession =
Depends(get_async_session)):
   deleted = await dish_repo.delete_dish(session, dish_id)
   if not deleted:
       raise HTTPException(status_code=404, detail="Dish not found")
```

Захист маршруту здійснюється за допомогою залежності require_superuser, що гарантує доступ лише користувачам із відповідними правами.

Реалізація ролей і авторизації у веб-додатку здійснена за допомогою бібліотеки FastAPI Users, яка надає зручні інструменти для роботи з автентифікацією через JWT-токени, зберіганням захешованих паролів і підтримкою ролей користувачів. У моделі користувача використовується логічне поле is_superuser, що дозволяє відокремити адміністратора від звичайного користувача. На основі цього механізму в системі реалізовані захищені маршрути: доступ до CRUD-операцій над даними мають лише адміністратори, тоді як звичайні клієнти можуть лише переглядати інформацію.

Щодо обробки зображень, система передбачає можливість завантаження фото для кожної страви під час її створення або редагування. Цей процес ініціюється адміністратором, який надсилає файл зображення, що обробляється сервером і

передається у хмарний сервіс Cloudinary. Після завантаження Cloudinary повертає URL, який зберігається у базі даних у полі image_url. Завдяки цьому зображення відображаються у клієнтському інтерфейсі, а сам додаток залишається легким і не перевантаженим файлами **[20]**.

Важливою функцією FastAPI є автоматична генерація документації API згідно з форматом OpenAPI. Завдяки цьому розробники та тестувальники можуть отримати повну інтерактивну документацію без потреби вручну її створювати. Веб-додаток надає доступ до двох типів документації: /docs — з інтерактивним інтерфейсом Swagger UI та /redoc — у форматі ReDoc. Це значно полегшує перевірку авторизаційних механізмів і доступів на етапах розробки та тестування.

3.3. Реалізація клієнтської частини веб-додатку

Клієнтська частина веб-додатку TomYum реалізована з використанням React — популярної бібліотеки JavaScript, яка дає змогу створювати сучасні динамічні веб-інтерфейси. Основне завдання фронтенду — надати користувачу зручний і адаптивний інтерфейс для ознайомлення з меню, взаємодії з категоріями, а також виконання авторизації та роботи з даними через API.

Проєкт клієнтської частини був ініціалізований за допомогою Vite — сучасного та швидкого інструменту для збірки React-додатків. Для стилізації інтерфейсу обрано Tailwind CSS, яка забезпечує адаптивний дизайн, гнучкість у компонуванні елементів інтерфейсу та дозволяє уникнути створення окремих CSS-файлів завдяки утилітарному підходу.

Архітектура клієнтської частини має чітке поділення на компоненти відповідно до їх функціонального призначення. Це дозволяє зберігати логіку інтерфейсу структурованою та спрощує подальший розвиток проєкту. Серед основних компонентів слід виділити такі:

- Navbar — верхня навігаційна панель, яка містить логотип, навігаційні посилання, а також кнопки входу та виходу;

- DishCard — компонент для відображення окремої страви з усією інформацією: назвою, описом, ціною та зображенням;
- DishModal — модальне вікно, яке відкривається під час створення або редагування страви. Доступне лише користувачам із правами адміністратора;
- MenuNavigation — інтерфейс для фільтрації страв за категоріями;
- Login та Signup — форми, які забезпечують автентифікацію та реєстрацію користувача;
- AuthRoute — компонент, що виступає захисником маршруту: він перевіряє, чи автентифікований користувач, перш ніж надати доступ до сторінки або функції.

Усі ці компоненти взаємодіють з API-сервером, отримують або надсилають дані та забезпечують повноцінну функціональність веб-застосунку з боку клієнта.

Робота з API

Взаємодія клієнтської частини веб-додатку з сервером організована через окремий модуль services/api.js, який містить усі функції для виконання HTTP-запитів до API. У цьому модулі використовуються функції fetch, які відправляють запити до відповідних маршрутів серверної частини. Наприклад, для отримання списку всіх страв реалізована функція fetchAllDishes(), яка звертається до ендпоінту /dishes/. Вона передає відповідні заголовки, включаючи токен авторизації (якщо такий є), і повертає відповідь у форматі JSON. У разі помилки користувач отримує повідомлення про неможливість завантаження даних.

Лістинг 3.3.4. Код файлу services/api.js

```
export async function fetchAllDishes() {
 const res = await fetch(`${BASE}/dishes/`, {
   headers: headers(false),
 });
 if (!res.ok) throw new Error('Не вдалося завантажити страви');
 return res.json();
```

Отримані з API дані зберігаються у локальному стані React-компонентів за допомогою хука useState, після чого виводяться на сторінці через JSX. Такий підхід забезпечує динамічне оновлення інтерфейсу у відповідь на зміну стану даних.

Керування станом та авторизація

Для збереження інформації про авторизацію користувача використовується контекст React — UserContext. У ньому зберігається JWT-токен, а також роль користувача (клієнт або адміністратор). Після успішної автентифікації токен зберігається в localStorage, і надалі автоматично додається до заголовків усіх запитів. Це дозволяє підтримувати сесію навіть після перезавантаження сторінки та забезпечує динамічне керування доступом. Наприклад, інтерфейс змінюється залежно від ролі користувача: клієнт бачить лише публічну частину меню, тоді як адміністратор отримує доступ до інструментів редагування або додавання нових страв.

Адаптивність та UX

За допомогою Tailwind CSS усі компоненти мають адаптивну верстку, що дозволяє коректно відображати інтерфейс на будь-якому пристрої — від мобільних телефонів до великих моніторів. Наприклад, меню автоматично перетворюється на випадаючий список у мобільному форматі, а елементи інтерфейсу — такі як шрифти, кнопки, блоки — змінюють свої розміри та розташування відповідно до ширини екрана. Tailwind CSS надає набір утилітарних класів, які дозволяють гнучко налаштовувати стилі без написання окремих CSS-файлів, забезпечуючи швидкість розробки та однорідність зовнішнього вигляду.

Крім того, фреймворк дозволяє легко реалізувати темну тему, медіа-запити, керування розмірами шрифтів, кольорами, відступами та положенням елементів. Завдяки цьому веб-додаток зберігає професійний і зручний для користувача вигляд незалежно від розміру екрана, а також гарантує, що користувацький досвід буде однаково приємним як на смартфоні, так і на великому екрані ноутбука або ПК.

3.4. Результати реалізації веб-додатку

У процесі реалізації веб-додатку TomYum було створено повноцінну систему для управління ресторанним меню, яка відповідає сучасним вимогам до веб-рішень як з боку клієнтської взаємодії, так і з боку архітектури серверної частини. У результаті реалізації вдалося досягти всіх поставлених функціональних цілей і забезпечити стабільну роботу системи в умовах локального середовища.

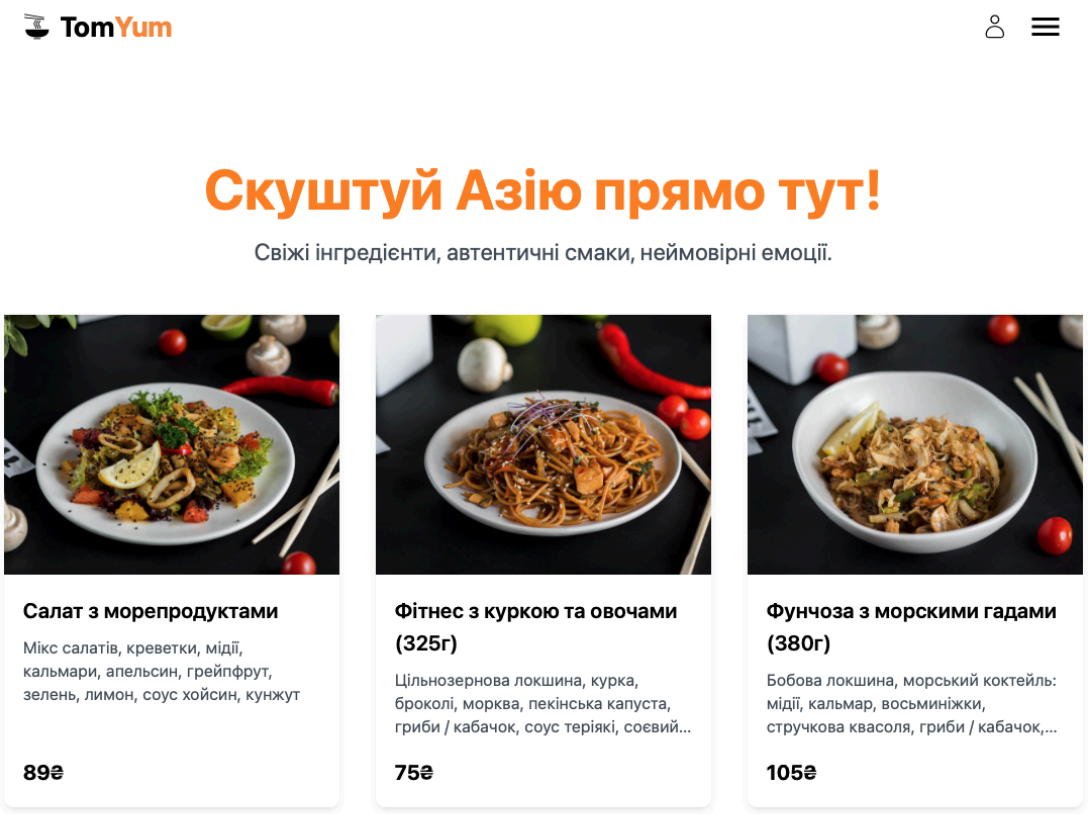

Рисунок 3.1. Головна сторінка клієнтської частини веб-додатку TomYum

Джерело: розроблено автором

Серверна частина веб-додатку реалізована на основі FastAPI з дотриманням принципів REST. Вона забезпечує роботу з API, підтримує авторизацію та розмежування прав доступу, дозволяє виконувати CRUD-операції для основних сутностей — страв і категорій — та забезпечує інтеграцію з хмарним сервісом Cloudinary для зберігання зображень. Уся логіка реалізована в асинхронному режимі

з використанням бази даних PostgreSQL, у якій збережено структуровану інформацію про користувачів, категорії та страви із відповідними зв'язками між ними.

Рисунок 3.2. Автоматично згенерована документація API

Джерело: розроблено автором

Реалізована рольова модель доступу дозволяє розмежовувати функціональність між адміністраторами та звичайними користувачами: суперкористувач має повний доступ до створення, редагування та видалення інформації, тоді як інші користувачі можуть лише переглядати меню.

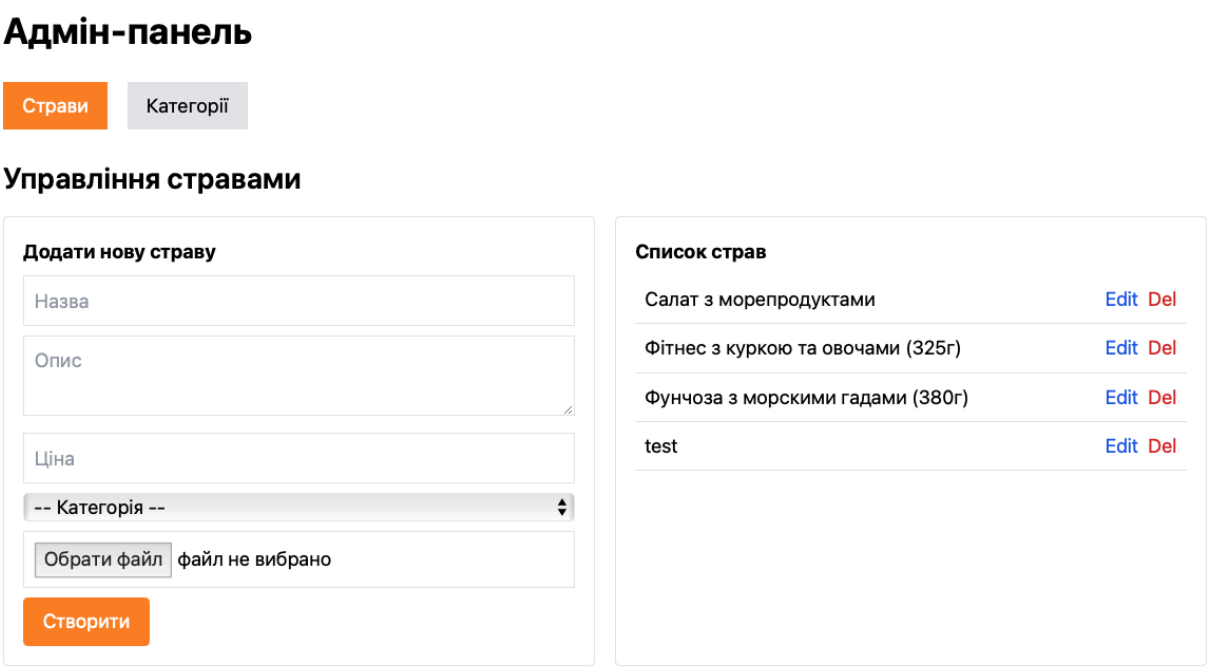

Рисунок 3.3. Інтерфейс адміністратора з можливістю редагування страв

Джерело: розроблено автором

Клієнтська частина побудована з використанням React, що дозволило реалізувати сучасний динамічний інтерфейс з компонентною архітектурою. Вона підтримує адаптивність для мобільних і десктопних пристроїв, забезпечує зручну навігацію, авторизацію користувачів, а також взаємодію з бекендом через HTTP-запити. У клієнті реалізовано логіку відображення меню за категоріями, перегляд детальної інформації про страви, а також окремі екрани для адміністративних дій.

Завдяки успішній інтеграції клієнта з API-сервером, користувачі можуть переглядати актуальне меню, а адміністратори — швидко оновлювати вміст у зручному інтерфейсі. Уся система функціонує як єдиний цілісний застосунок і є готовою до подальшого масштабування або адаптації для інших закладів громадського харчування.

Таким чином, реалізований веб-додаток виконує всі поставлені задачі — як з точки зору технічного виконання, так і з позиції кінцевого користувача. Система є стабільною, розширюваною, технологічно актуальною та орієнтованою на реальне практичне застосування.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було здійснено повний цикл проєктування та реалізації веб-додатку для управління ресторанним меню на основі сучасних веб-технологій. Основною метою дослідження було створення ефективного, зручного у використанні та технічно надійного програмного продукту, який дозволяє адміністраторам керувати вмістом меню, а клієнтам — переглядати актуальну інформацію про страви в інтерактивному форматі.

На етапі аналітичної роботи було визначено основні функціональні та нефункціональні вимоги до системи, а також досліджено предметну область, що дозволило чітко сформулювати архітектурне рішення. В основі проєкту лежить принцип «тонкий клієнт — потужний сервер», що реалізовано за допомогою FastAPI на стороні бекенду та React на стороні фронтенду.

Серверна частина забезпечує обробку HTTP-запитів, реалізацію бізнес-логіки, захист маршруту за ролями користувачів, інтеграцію з хмарним сервісом Cloudinary для зображень та взаємодію з реляційною базою даних PostgreSQL через ORM SQLAlchemy. Клієнтська частина реалізована за допомогою компонентної архітектури, що дозволяє зручно працювати з інтерфейсом, адаптувати його під різні пристрої, динамічно оновлювати контент та виконувати авторизацію користувачів.

У результаті було створено стабільний веб-застосунок, який повністю відповідає заданим вимогам. Система підтримує повний набір CRUD-операцій для ключових сутностей, рольову модель доступу, авторизацію, обробку зображень та адаптивний дизайн. Архітектура проєкту є масштабованою та готовою до подальшого впровадження у практичну діяльність ресторанного бізнесу.

Отримані результати демонструють доцільність використання вибраного технологічного стеку, а також підтверджують ефективність реалізованого підходу до побудови веб-додатків. Надалі проєкт може бути розширений за рахунок додавання модуля замовлень, системи повідомлень або повної панелі керування для адміністрації.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. React. React Documentation. URL: https://reactjs.org/docs/getting-started.html (дата звернення: 03.05.2025).
2. Angular. Angular Official Guide. URL: https://angular.io/guide/what-is-angular (дата звернення: 04.05.2025).
3. Vue.js. Vue.js Documentation. URL: https://vuejs.org/guide/introduction.html (дата звернення: 04.05.2025).
4. FastAPI. FastAPI Documentation. URL: https://fastapi.tiangolo.com/ (дата звернення: 05.05.2025).
5. Django. Django Documentation. URL: https://docs.djangoproject.com/en/stable/ (дата звернення: 05.05.2025).
6. Node.js. Node.js Documentation. URL: https://nodejs.org/en/docs/ (дата звернення: 05.05.2025).
7. PostgreSQL. PostgreSQL Official Documentation. URL: https://www.postgresql.org/docs/ (дата звернення: 06.05.2025).
8. MongoDB. MongoDB Manual. URL: https://www.mongodb.com/docs/manual/ (дата звернення: 06.05.2025).
9. Mozilla. HTTP Methods. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods (дата звернення: 07.05.2025).
10. OpenAPI Initiative. OpenAPI Specification. URL: https://spec.openapis.org/oas/latest.html (дата звернення: 08.05.2025).
11. JSON Web Tokens. JWT.io Introduction. URL: https://jwt.io/introduction (дата звернення: 09.05.2025).
12. Cloudinary. Cloudinary Documentation. URL: https://cloudinary.com/documentation (дата звернення: 10.05.2025).
13. Pydantic. Pydantic Documentation. URL: https://docs.pydantic.dev/ (дата звернення: 11.05.2025).
14. SQLAlchemy. SQLAlchemy Documentation. URL: https://docs.sqlalchemy.org/ (дата звернення: 12.05.2025).

15. Alembic. Alembic Documentation. URL: https://alembic.sqlalchemy.org/en/latest/ (дата звернення: 13.05.2025).
16. Tailwind CSS. Tailwind CSS Documentation. URL: https://tailwindcss.com/docs (дата звернення: 14.05.2025).
17. Vite. Vite Documentation. URL: https://vitejs.dev/guide/ (дата звернення: 15.05.2025).
18. FastAPI Users. GitHub Repository. URL: https://github.com/fastapi-users/fastapi-users (дата звернення: 16.05.2025).
19. SOLID Principles. URL: https://web.archive.org/web/20230209070718/https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design (дата звернення: 17.05.2025).
20. GitHub. FastAPI Cloudinary Example. URL: https://github.com/topics/fastapi-cloudinary (дата звернення: 18.05.2025).