ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра інженерії програмного забезпечення

Пояснювальна записка

до бакалаврської роботи
на ступінь вищої освіти бакалавр

на тему: «**РОЗРОБКА ДОДАТКУ ДЛЯ ОПТИМІЗАЦІЇ ПРОЦЕСУ ОБСЛУГОВУВАННЯ В ЗАКЛАДАХ ГРОМАДСЬКОГО ХАРЧУВАННЯ МОВОЮ C#**»

Виконав: студент 4 курсу, групи ПД–41
спеціальності
121 Інженерія програмного забезпечення
(шифр і назва спеціальності/спеціалізації)

Матвійчук А. М.
(прізвище та ініціали)

Керівник Жебка В.В.
(прізвище та ініціали)

Рецензент _____
(прізвище та ініціали)

Київ –2022

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра Інженерії програмного забезпечення
Ступінь вищої освіти -«Бакалавр»
Спеціальність підготовки – 121 «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____Негоденко О.В.
"____" _____2022 року

З А В Д А Н Н Я
НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТА

МАТВІЙЧУКУ АРТЕМУ МИКОЛАЙОВИЧУ

(прізвище, ім'я, по батькові)

1. Тема роботи: «Розробка додатку для оптимізації процесу обслуговування в закладах громадського харчування мовою C#»

Керівник роботи: Жебка В.В. д.т.н., доц.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом вищого навчального закладу від «18» лютого 2022 року №__
2. Строк подання студентом роботи «3» червня 2022 року
3. Вхідні дані до роботи
   3.1 Положення побудови клієнт-серверної архітектури додатків;
   3.2 Методи побудови клієнт-серверної архітектури;
   3.3 Існуючі додатки для оптимізації процесу обслуговування в закладах громадського харчування
   3.4 Науково-технічна література

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити).
   4.1 Огляд існуючих засобів оптимізації процесу обслуговування в закладах громадського харчування
   4.2 Вимоги та оцінка якості системи.

4.3Опис проектування системи.

4.4Опис використаних технологій.

4.5Висновки

5. Перелік демонстраційного матеріалу

5.1 Титульний слайд

5.2 Мета, об’єкт та предмет роботи

5.3 Аналіз аналогів

5.4 Технічне завдання

5.5 Спеціалізовані технології

5.6 Архітектура серверу

5.7 Схема автентифікації та авторизації

5.8 Процес замовлення

5.9 Приклад використання

5.10 Наукова новизна та практична значимість

5.11 Апробація

5.12 Висновки

6. Дата видачі завдання «11» квітня 2022

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів роботи	Примітка
1	Підбір науково-технічної літератури	16.02-21.02	Виконано
2	Розробка вимог до системи	22.02-24.02	Виконано
3	Підбір технологій розробки та створення структури додатку	25.02-04.03	Виконано
4	Розробка програмного забезпечення	05.03-25.03	Виконано
5	Тестування програмних модулів	28.03-05.04	Виконано
6	Вступ, висновки, реферат	06.04-20.04	Виконано
7	Розробка обов‘язкових демонстраційних матеріалів	21.04-26.04	Виконано
8	Попередній захист роботи		
9	Здача роботи		

Студент ____ _____ Матвійчук А. М.
(підпис) (прізвище та ініціали)

Керівник роботи _____ _____
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Текстова частина бакалаврської роботи 78100 с., 3 табл., 52 рис., 2 додатки., 20 джерел.

ОПТИМІЗАЦІЯ БІЗНЕС ПРОЦЕСІВ, ГРОМАДСЬКЕ ХАРЧУВАННЯ, КЛІЄНТ-СЕРВЕРНА АРХІТЕКТУРА, ВЕБ ДОДАТОК

Об'єкт дослідження – процес замовлення у закладах громадського харчування.

Предмет дослідження – веб-додаток для автоматизації процесу замовлення.

Мета роботи – автоматизація процесу замовлення у закладах громадського харчування, за допомогою розробленого додатку.

Методи дослідження – методи оптимізації бізнес процесів за допомогою програмного забезпечення, емпіричні методи.

У роботі проведено аналіз існуючих додатків, таких як Glovo, Bolt Food, Domino's Pizza та ін. методів перенесення взаємодії між клієнтами та закладами громадського харчування.

Загальною проблемою цих додатків є неорієнтованість на малий бізнес та неможливість робити замовлення для прийому їжі у закладі, бронювання столиків та ін.

Описано архітектуру та основні принципи розробки. Додаток розроблений на платформі .NET з використанням мови C# для серверу і фреймворку Angular на мові TypeScript для клієнтської частини.

В якості серверу баз даних було взято Microsoft SQL Server та бібліотеку Entity Framework Core для взаємодії з нею.

Отже, розроблено та описано веб-додаток, завданням якого є можливість взаємодії клієнтів та надавачів послуг (заклади громадського харчування) із мінімальною фізичною взаємодією.

Даний додаток може бути використано будь-якими закладами, які хочуть мати підключення до централізованого сервісу для замовлень.

Галузь використання – сфера громадського харчування

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

REST – Representational State Transfer

API – Application Programming Interface

JSON – Java Script Object Notation

JWT – JSON Web Token

ПЗ – програмне забезпечення

ORM – Object-Relation Mapping

СУБД – Система Управління Базами Даних

TSC – TypeScript Compiler

UI – User Interface

UX – User Experience

ВСТУП

Обґрунтування вибору теми та її актуальність: клієнти закладів громадського харчування витрачають велику кількість часу в чергах, щоб зробити замовлення, та в очікуванні приготування замовлення. При необхідності зробити передчасне замовлення, клієнт мусить шукати сайти або номери телефонів бажаних закладів.

Для оптимізації витрат часу необхідно розробити інформаційну систему, за допомогою якої можна швидко робити замовлення у різних закладах, незалежно від їх розміру та популярності.

Ступінь вивчення проблеми: існуючі методи оптимізації процесу обслуговування клієнтів переважно орієнтовані на доставку замовлень від великого та середнього бізнесу, рідше – на виніс. Прикладами таких систем є Glovo, Bolt Food, Domino’s Pizza тощо. В цих системах відсутні можливості замовлення у малих закладів та замовлення для прийому їжі на місці.

Об’єктом дослідження є процес замовлення у закладах громадського харчування.

Предметом роботи є веб-додаток для автоматизації процесу замовлення.

Метою роботи є автоматизація процесу замовлення у закладах громадського харчування, за допомогою розробленого додатку.

Завданням роботи розробка веб-додатку для автоматизації процесу обслуговування у закладах громадського харчування.

Методика дослідження Перш за все, потрібно було визначити ролі користувачів системи, необхідних для них функціонал, обмеження та обов’язки, вивчити процес їх взаємодії між собою. Далі, вибрати архітектуру майбутнього додатку та стек технологій для забезпечення необхідного функціоналу та працездатності продукту, за можливості підтримки його розробником з метою ефективного масштабування додатку або виправлення виявлених недоліків в майбутньому.

Враховуючи вимоги до специфіки програмного продукту, найкращим рішенням є веб-додаток, що має клієнт-серверну архітектуру, де клієнтська сторона

відповідає за взаємодію з клієнтом програмного продукту та графічне відображення даних, а серверна – за бізнес логіку, обчислення та зберігання даних. Такий тип технології дає можливість маніпулювати системою централізовано.

Наукова новизна роботи: Таким чином, наукова новизна полягає у створенні нового виду взаємодії клієнтів закладів громадського харчування з підприємствами та розширення таких на заклади малого бізнесу.

Практична значущість результатів: Даний продукт допоможе користувачам системи, як бізнесу так і клієнтам, економити час та краще ним розпоряджатися. Крім того, створена інформаційна система може слугувати концептом для створення нової компанії, яка займатиметься подальшим розвитком та підтримкою додатку, або інтеграції такого функціоналу в уже існуючі додатки.

1. ОГЛЯД ПРОЦЕСУ ОБСЛУГОВУВАННЯ В ЗАКЛАДАХ ГРОМАДСЬКОГО ХАРЧУВАННЯ

1.1 Аналіз процесу обслуговування в закладах громадського харчування

Відповідно до закону України, закладами громадського харчування називають заклади, незалежно від їх знаходження і доступності, що забезпечує харчуванням невизначену кількість осіб [1]. Основними видами закладів є ресторани, бари, кафе, їдальні, піцерії тощо. Незалежно від виду, класу та орієнтованості, всі вони мають схожий алгоритм взаємодії з клієнтами, наведений нижче.

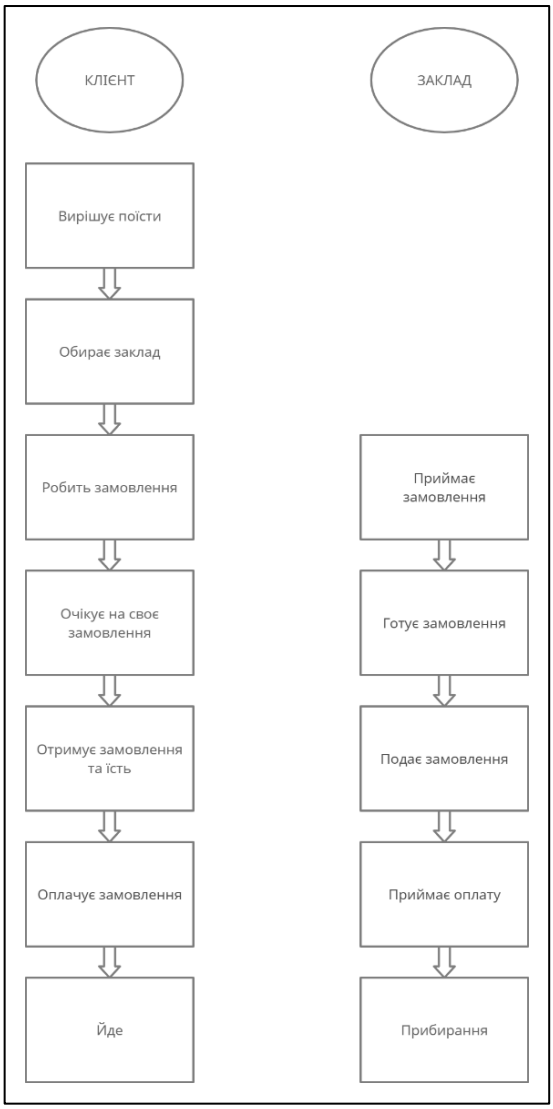


Рис 1.1.1. - Взаємодія клієнтів та закладів громадського харчування

З даної схеми наглядно видно, що підприємство не може впливати на взаємодію напряму, доки клієнт сам не обере заклад, а для цього необхідно вкладатися в маркетинг та заохочувати клієнта усілякими бонусами.

Перед підприємцями закладів харчування завжди існує багато проблем на шляху до якісного обслуговування, та, як наслідок, до фінансового успіху. Для забезпечення потреб клієнта сьогодні потрібно винаходити нові способи дистрибуції. Бізнес мусить йти у ногу з часом та підлаштовуватися під сучасний ритм життя своїх клієнтів. Як зазначається на офіційному сайті Міністерства з питань стратегічних галузей промисловості України, за даними про результати досліджень однією з експертних сесій NRA Show [2], 78% людей віком від 25 до 40 років шукають меню закладу в інтернеті, для 71% клієнтів важлива можливість замовити їжу на виніс, 47% розраховують на можливість попереднього замовлення. Отже, перед підприємцями стоїть вибір як саме надавати людям такий функціонал.

Одним з можливих варіантів є створення цілого підрозділу, який буде займатися прийомом дзвінків від клієнтів, які будуть вчасно відповідати та створювати замовлення, яке будуть опрацьовувати інші робітники. Проте цей варіант не найкращий, він має велику кількість недоліків. Створення цілого підрозділу може створити велику кількість проблем на підприємстві. Пошук кадрів може зайняти час, достатній для того, щоб втратити клієнтську базу. Щороку кількість працездатного населення України зменшується, наприклад, через трудову міграцію, а втримати робітника стає все тяжче. Та найважливіше, це те що дозволити таке собі може лише великий бізнес, не враховуючи часу та ресурсів на маркетинг ідеї. Малий же бізнес, такий як сімейна пекарня, не має такої змоги взагалі. Як показує практика, для рішення цієї проблеми потрібні інші методи.

Найкраще автоматизувати процес отримання та оброблення замовлень, виключаючи використання додаткових людських ресурсів, та зробити це зручно і зрозуміло можна через інтернет та мобільні додатки [3]. Це призведе не тільки до покращення сервісу, та також і допоможе підприємцям оптимізувати використання наявних людських ресурсів, на основі даних вести детальні технологічні та економічні карти, що дає змогу краще розуміти та контролювати процеси на

підприємстві. За даними 2020-го року, вище згаданої NRA Show, близько 41% закладів громадського харчування США планують використовувати автоматизовані способи обслуговування клієнтів. Подібні методи прийому замовлень також мінімізують можливість помилок при обробці та виробництві продукції, через відсутність людського фактору у ланцюзі між покупцем та підприємцем, більш того, дає зручний спосіб взаємодії персоналу між собою.

До того ж, у закладах громадського харчування через нерівномірність клієнтського потоку впродовж дня, середня продуктивність праці працівників, та як наслідок пропускна здатність закладу, падає. Така нерівномірність виражається в чергах та затримках, а за даними тайванських дослідників Фанг-Пеі Ніє та Чінь-Юнь Понг, які вони надали в своїй роботі «Key success factors in catering industry management» [4], час очікування замовлення клієнтом входить в п’ятірку ключових факторів успішного бізнесу. Можливість зробити завчасне замовлення на виніс або на місці розвантажує черги, що неабияк важливо у часи пандемії, та дає змогу робітникам більш ефективно використовувати час. Як наслідок, значно зменшуються шанси на перевтому у окремих працівників, та збільшення загальної працездатності.

Якщо більш детально розглядати ефективність цієї системи з точки зору клієнта, то немає сумнівів, що вона має критичні переваги над іншими способами обслуговування. Можливість дистанційно переглянути меню закладу, та замовити потрібне, на виніс, на місці, чи із доставкою – зручно та ефективно. Це неабияк збільшує доступність улюблених закладів харчування для кожного відвідувача. Більш того, автоматизована система індивідуально підходить до кожного клієнта, та дає можливість обрати усе необхідне на стадії прийому замовлення, та дає можливість відстежування на стадії обробки замовлення, знаючи, коли замовлення буде готовим, та зайти за ним і забрати, не чекаючи у черзі. Слід зазначити, що така система ідеально працює у тандемі з інтернет платіжними системами. За даними німецької компанії Statista, що спеціалізується на ринкових та споживчих даних, у 2020 році загальна кількість користувачів Apple Pay, Google Pay та Samsung Pay сягнула майже 500 мільйонів людей по всьому світі [5].

Отже, перехід на систему замовлень через мобільні додатки став життєво важливим для більшості закладів громадського харчування, особливо під час пандемії. Він допоміг і надалі допомагає підприємцям не тільки зберігати свій бізнес, а і надавати звичні послуги набагато більш ефективно. Автоматизація більшості процесів стала невід'ємною частиною сьогоднішньої культури споживання. Кожна зі сторін має з цього вигоду: підприємець спрощує роботу персоналу, відмовляється від деяких послуг, оптимізовує витрати та підвищує швидкість обслуговування, з чого має прибуток, а клієнти у свою чергу мають просту та зрозумілу систему замовлення та оплати, без черг та очікування.

1.2 Аналіз програмного забезпечення та інших засобів оптимізації процесу обслуговування в закладах громадського харчування

У сфері громадського харчування існують кілька моделей взаємодії клієнтів з власне бізнесом:

- звичайна - клієнт робить замовлення та їсть на місці
- доставка - клієнт робить замовлення дистанційно і воно доставляється службою доставки в необхідне місце
- самовивіз - клієнт робить замовлення, але забирає товар з собою (можна розділити на ще два – де замовлення відбувається прямо у закладі та коли клієнт робить замовлення дистанційно)

Для звичайної взаємодії можливості оптимізації обмежені, адже взаємодія відбувається фізично. Головна задача в такому випадку – прискорити швидкість людей у черзі та збільшити продуктивність працівників. До прикладу, якщо це фаст-фуд з великою кількістю працівників на кухні, де їжа готується швидко, можна зробити більше кас. Проблема такого рішення у тому, що воно матиме успіх при великій та постійній кількості клієнтів, що не є можливим, адже клієнтський потік не є рівномірним.

Інший підхід до такого роду автоматизації полягає у виключенні працівника від прямої взаємодії з клієнтом при прийнятті замовлення, доприкладу – термінали

самообслуговування. Це доволі популярне рішення для великих мереж фаст-фудів, які можуть дозволити собі встановлювати такі пристрої.

За допомогою такого терміналу, клієнт має змогу обрати будь-які бажані опції меню, кількість, скористатися промокодами компанії та оплатити замовлення. Після оплати, користувач отримує чек з номером замовлення, та очікує появи його номеру на спеціальному екрані всередині закладу. Далі він має змогу підійти на касу та забрати своє замовлення. Це не погане рішення для зменшення черг та розвантаження персоналу, хоча також передбачує взаємодію та очікування у самому закладі. Крім того, це не таке дешеве рішення, тому невеличкі кав'ярні не можуть собі такого дозволити.

Головна проблема живого замовлення – необхідність чекати доки їжа готується, а також те, що йдучи у заклад, немає впевненості чи достатньо там місця та чи заклад має необхідні продукти для конкретних страв. Хоча в багатьох кафе та ресторанах можна бронювати столик, проте взаємодія відбувається напряму з закладом, що створює незручності у разі, якщо необхідно розглянути декілька варіантів.

Сфера доставки має набагато кращі можливості для оптимізації та на сьогодні вже є достатньо діджиталізованою. Такі сервіси популярні у всьому світі , тому їх по всьому світі та десятки в Україні. У період пандемії, такі сервіси можуть сприяти стримуванню поширення захворювання, адже клієнти замість скупченню в чергах можуть безпечно перебувати вдома та отримувати бажані продукти харчування. Одними з найпопулярніших в Україні є міжнародні компанії Glovo та Bolt Food.

Glovo — іспанська компанія, яка пропонує сервіс доставки через мобільний додаток. На даний час компанія надає свої послуги у більш, ніж 20 країнах світу. Принцип його роботи такий: через мобільний додаток клієнти можуть замовити будь-який товар (з переліку) невеликого розміру (40x40x30см) та вагою до 9 кг. Після оформлення замовлення користувач має можливість відстежувати переміщення кур'єра на карті у смартфоні в режимі реального часу. Сервіс доступний через десктоп, IOS та Android. Оплата можлива як за допомогою банківської карти так і готівкою при отриманні. Найбільший попит сервісу

припадає на їжу, ліки та документи. У Європі, наприклад, 85 % замовлень складає доставка їжі.

Додаток має зручний та сучасний інтерфейс. Для створення замовлення спочатку потрібно:

- Уточнити категорію

  Оскільки компанія доволі велика, має змогу приймати замовлення на доставку не лише їжі. Glovo пропонує великий асортимент категорій для замовлення, такі як їжа, аптеки, супермаркети тощо.

- Вибрати конкретну організацію

  Конкретний ресторан, аптеку, супермаркет або ін., в залежності від обраної вами категорії. З компанією співпрацює велика кількість закладів громадського харчування, супермаркетів тощо, тому асортименту товарів зазвичай достатньо.

- Вибрати предмет доставки

  Необхідно обрати бажаний товар або продукт та додати його в кошик. Також можна обрати кількість товару, порції та модифікації.

- Підтвердити замовлення

У кошику зберігаються всі обрані вами товари та одразу відображається ціна з послугами кур'єра. Тут необхідно обрати спосіб оплати, вказати номер телефону, за наявності також можна застосувати промокод на знижку або додатковий товар. Із запропонованих способів оплати – картою по передплаті або готівкою при отриманні.

Bolt Food – сервіс доставки їжі від естонської компанії Bolt (раніше Taxify). Принцип роботи аналогічний компанії Glovo, але це сервіс орієнтований лише на доставку їжі, а також оплата можлива лише за допомогою банківської картки онлайн.

Першим етапом замовлення є вибір точки доставки, після чого користувачу відображаються найпопулярніші заклади громадського харчування поблизу. Після вибору, додаток відображає інтерактивне меню з фото, де користувач може подивитися доступні варіанти та ціни на них. Далі користувач також додає товари

в кошик, робить замовлення і оплачує його. Мінімальна сума замовлення становить сто гривень.

Можливості самовивозу також є доволі розвиненими. У згаданих вище додатках така функція можлива, проте працює з обмеженою кількістю закладів бо сервіс орієнтований саме на доставку, а самовивіз є додатковою функцією. Основні зусилля закладів громадського харчування у сфері взаємодії з клієнтом спрямовані на створення та обслуговування особистих додатків. Такий підхід може бути зручним з точки зору керування для бізнесу, але для користувачів це може бути складним принципом.

Як приклад такого додатку буде використано сервіс піцерії Domino’s – міжнародної мережі піцерій, однієї з найбільших у світі мереж ресторанів піци, працює більше у 85 країнах.

Головна сторінка додатку – це меню, яке представляє компанія. Воно розділене на окремі категорії, такі як піца, напої, десерти. Після вибору товарів ви маєте змогу коригувати розміри та добавки продуктів. Будь-яка опція доступна у всіх закладах, що робить процес замовлення зручним, адже після вибору піци, ви маєте змогу обрати заклад для самовивозу, або адресу доставки. Оплата може бути здійснена за допомогою банківської карти на сайті або готівкою кур’єру при доставці чи оплата на місці при самовивозі.

Загалом, такі додатки мають є доволі зручними для великих компаній з великим меню та великою мережею закладів, адже для кав’ярень або малих фастфудів такий додаток не матиме необхідного ефекту.

У таблиці 1.2.1 наведені зведені результати аналізу характеристик згаданих рішень для оптимізації процесу обслуговування у закладах громадського харчування по трьом моделям взаємодії клієнта з бізнесом (класична, доставка, самовивіз).

Таблиця 1.2.1 – Зведені результати аналізу характеристик програмних рішень для оптимізації процесу обслуговування у закладах громадського харчування

Показник	**Glovo**	**Bolt Food**	**Domino's**	**Майбутній додаток**
Платформи	Android, IOS, WEB	Android, IOS, WEB	Android, IOS, WEB	WEB
Замовлення для їжі на місці	ні	ні	ні	так
Доставка	так	так	так	ні (в перспективі)
Самовивіз	так	так	так	так
Власний бізнес	ні	ні	так	ні
Інтеграція з малим бізнесом	ні	ні	-	так
Інтеграція з середнім та великим бізнесом	так	так	-	так
Оплата готівкою	так	ні	так	так
Оплата картою	так	так	так	так
Інтерактивна карта для пошуку закладів	ні	ні	так	так

1.3 Огляд засобів реалізації додатків для оптимізації процесу обслуговування в закладах громадського харчування

Для початку необхідно окреслити основні архітектурні та технічні рішення, що будуть застосовані в розробці. Оскільки додаток використовуватиме для своєї роботи мережу Інтернет, матиме різних користувачів, які повинні будуть взаємодіяти між собою та окремо з даними, для зберігання яких необхідна база даних, архітектура додатку безсумнівно повинна бути клієнт-серверною [6].

Сервер (в цьому випадку веб-сервер) – програма, яка використовує протокол HTTP для обробки запитів від клієнтів. Сервери чекають надходження запитів від клієнтів, а потім відповідають на них. В ідеалі сервер надає клієнтам стандартизований прозорий інтерфейс, щоб клієнтам не потрібно було знати про особливості системи (тобто апаратного та програмного забезпечення), яка надає послугу [7].

Клієнт – програма, що надає користувачу взаємодіяти з сервісом через (в основному) графічний інтерфейс, використовуючи різноманітні протоколи передачі даних (в цьому випадку HTTP).

Для створення додатку, було обрано модульну платформу для розробки програмного забезпечення від компанії Microsoft .NET, найактуальнішої LTS (long term support) версії 6.0 на даний час. Ця платформа дозволяє створювати кросплатформенні додатки, що можуть запускатися на різних операційних системах, таких як Windows, Mac, Linux тощо. Така гнучкість дає змогу розміщувати цю частину додатку на різноманітних серверах.

В якості мови програмування обрано мову C#, найпопулярнішу мову для цієї платформи і одну з найбільш поширених в усьому світі. Це сучасна мова програмування, за допомогою якої можна створювати не лише серверне програмне забезпечення, а й мобільні та комп’ютерні додатки. C# є мультипарадигмальною мовою, тому допускає об’єктно-орієнтоване програмування, функціональне програмування, в тому числі, лямбда-вирази. В стандартну бібліотеку мови входить великий пакет готових пакетів коду, що значно спрощує розробку, беручи на себе значну кількість алгоритмічної роботи. До прикладу, інтегрована мова запитів LINQ дозволяє виконувати велику кількість операцій з колекціями, такі як сортування, групування, фільтрація, приведення до інших реалізацій колекцій та переведення типу колекцій, лише в кілька рядків коду.

У якості СУБД було вибрано Microsoft SQL Server та, оскільки це проста, швидка і водночас потужна система, яка до того ж має глибоку інтеграцію з іншими сервісами та платформами компанії.

Для зручної роботи з БД обраний найпопулярніший для цього фреймворк на мові C# - Entity Framework Core. Цей самий функціональний ORM інструмент. ORM - відображення даних у вигляді об’єктів відповідних класів. Наприклад, якщо розробник безпосередньо працює з базами даних, програміст повинен думати про підключення, підготовку SQL і параметрів SQL, як надсилати запити та транзакції. А за допомогою Entity Framework Core все це робиться автоматично – розробник працює безпосередньо з класами .NET.

Клієнтська частина пердставлятиме собою веб-сайт. Це дозволяє одразу покривати велику кількість користувачів на різних операційних системах всього одним додатком та зменшити час розробки. Для створення клієнтської частини, було обрано TypeScript front-end фреймворк Angular [8].

TypeScript - мова програмування від Microsoft, що позиціонується як засіб розробки веб-застосунків. Вона побудована на базі JavaScript та розширює його функціонал. Головною відмінністю TypeScript від JavaScript є можливість статичної типізації, а це в свою чергу спрощує розробку та підтримання коду. Також, варто сказати, що на відміну від JavaScript, TypeScript не може бути інтерпретовано браузером на пряму, тому для його використання необхідний TSC для компіляції в JavaScript.

Angular — це платформа JavaScript з відкритим вихідним кодом, написана на TypeScript, підтримувана Google. Його основна мета — розробляти односторінкові програми. Як фреймворк, Angular має очевидні переваги над конкурентами, як можливість статичної типізації та модульність програми, а також надає стандартну структуру для роботи розробників. Це дозволяє користувачам створювати великі програми, що легко підтримувати. Цей фреймворк дозволяє розробляти додатки, для роботи з якими не потрібно перезавантажувати веб-сторінку, тобто розробляти сторінки з динамічним вмістом.

2. ВИМОГИ ТА ТЕХНОЛОГІЇ РОЗРОБКИ СИСТЕМИ

2.1 Аналіз видів користувачів системи

Оскільки програмний продукт спрямований на велику кількість користувачів та їх взаємодію між собою, система матиме кілька видів користувачів. В загальному вигляді їх можна описати як клієнтів, агентів бізнесу та адміністрування додатку.

До перших входять пересічні користувачі, які використовуватимуть додаток для замовлення їжі в закладах громадського харчування.

До агентів бізнесу відносяться два види користувачів:

– компанія, тобто юридична особа, яка володіє закладами громадського харчування;
– заклад, який власне буде отримувати та обробляти замовлення, стежити за відповідністю доступних продуктів та їх ціною.

До облікових записів адміністрування додатку входять два види: адміністратор та служба підтримки. Перші будуть повинні стежити за обліковими записами компаній та клієнтів, створювати та видаляти аккаунти служби підтримки тощо. Служба підтримки повинна обробляти скарги користувачів та бути на з’язку у разі, якщо у клієнтів з’явиться питання.

Таким чином, система повинна мати п’ять видів користувачів: клієнт, компанія, заклад громадського харчування, адміністратор та служба підтримки. До того ж варто зазначити, що лише два з них (клієнт та компанія) матимуть змогу вільно реєструватися, усі інші – будуть генеруватися автоматично, відповідно потреб.

Таблиця 2.1.1 – Класифікація облікових закладів.

Назва користувача	Опис	Призначення
Компанія	Юридична особа, володіє закладами громадського харчування	Менеджмент закладів громадського харчування, реквізити для оплати послуг

Продовження таблиці 2.1.1

Назва користувача	Опис	Призначення
Клієнт	Фізична особа, що розміщує замовлення	Створення та оплата замовлень
Заклад громадського харчування	Заклад громадського харчування, що отримує замовлення	Отримання, приготування та видача замовлень клієнтів
Адміністратор	Працівник для управління користувачами додатку	Перевірка даних компаній та адміністрування всіх облікових записів системи
Служба підтримки	Працівник для обробки скарг та запитань від клієнтів	Спілкування з користувачами системи, обробка скарг клієнтів

2.2 Опис основних вимог до функціоналу користувача типу «Клієнт»

Готова інформаційна система повинна надавати змогу клієнтам закладів громадського харчування знаходити заклади поблизу їх місцезнаходження в першу чергу, та можливість перегляду інших закладів на карті. Крім того, необхідно реалізувати пошук закладів по ключовим словам, щоб клієнтам було легше знаходити бажані страви та напої. Користувач повинен мати змогу обирати заклад зі списку запропонованих та переглядати його меню, обирати необхідні модифікації та порції, вибирати кількість продукту, додавати позиції меню до корзини.

Для забезпечення кращого UX, корзина повинна утворювати групи доданих до неї страв за закладом харчування, адже до кожного закладу формується своє замовлення. У корзині клієнт повинен мати змогу видаляти непотрібні позиції та змінювати їх кількість. Кожен продукт мусить відображати свою ціну в залежності від порції та модифікацій, помноженій на кількість, а кожна група страв – відображати загальну ціну потенційного замовлення. Для кожної групи потрібна кнопка, для створення замовлення, після натискання якої клієнт повинен мати змогу обрати спосіб оплати (готівка або оплата картою) та додати коментар до

замовлення, якщо він, наприклад, хоче зазначити бажану дату або час або вилучити окремий інгредієнт зі страви.

У вкладці «Замовлення» клієнт матиме змогу переглядати історію замовлень, де відображатимуться їх статуси в режимі реального часу, номери та інша довідкова інформація. Користувач повинен мати змогу відмінити замовлення, оплатити, якщо оплата картою та поскаржитися у випадку неналежного обслуговування. Після отримання замовлення, клієнту необхідно буде підтвердити його завершення. Також, для захисту від фальшивих замовлень, варто обмежити кількість незакритих замовлень, щоб уберегти компанії від зловмисників.

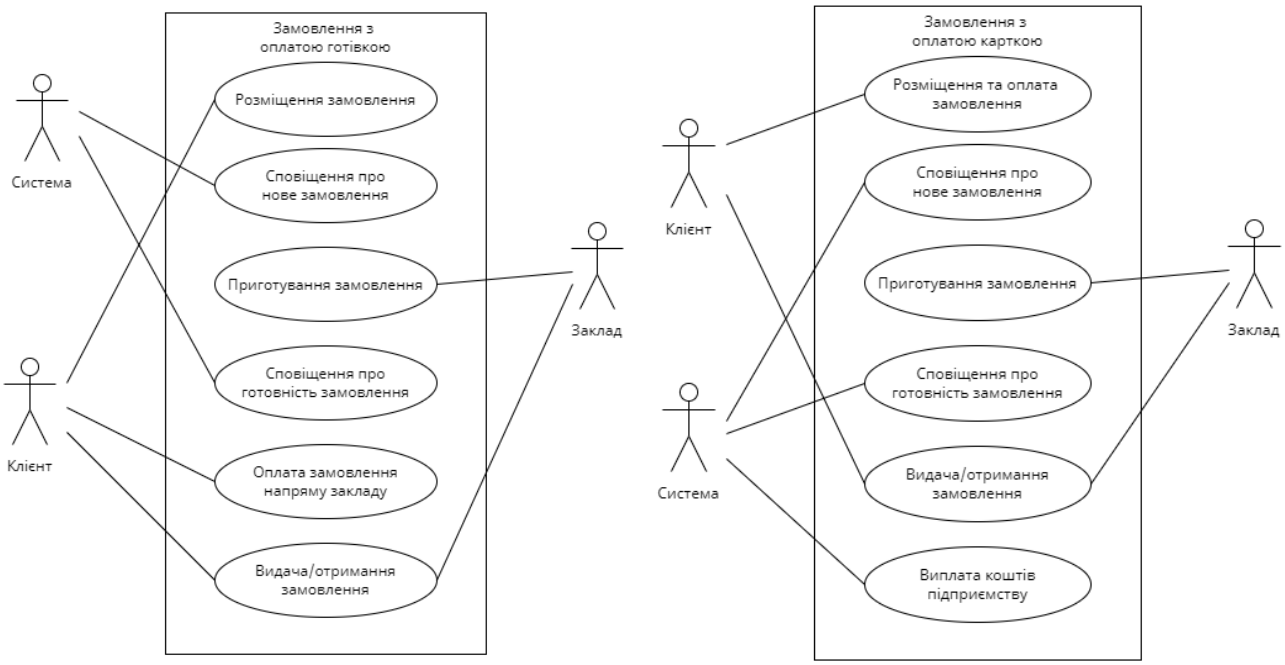


Рис. 2.2.1 – Діаграми прецедентів при розміщенні замовлень з різними видами оплати

2.3 Опис основних вимог до функціоналу користувача типу «Компанія»

Програмний продукт для компаній повинен включати в себе такий функціонал як зазначення назви та адреси юридичної особи, платіжної інформації, опису і логотипу компанії. Проте, основний функціонал для цього виду облікових

записів полягає у створенні та адмініструванні закладів громадського харчування, підпорядкованих взятій компанії.

До адміністрування закладів входить:

– додавання нових закладів громадського харчування;
– зміна існуючих закладів компанії;
– видалення закладів.

Компанія повинна мати змогу налаштовувати меню закладів, існуючих порцій товарів та їх модифікацій для економії та оптимізації часу, адже у випадку мережевих закладів громадського харчування меню вони мають схоже.

Облікові записи компаній повинні ставати активними лише після перевірки адміністраторами та укладанням необхідних угод, відповідно до законодавства України. З цього випливає необхідність адміністраторів системи, котрі будуть брати на себе потрібну відповідальність та допускати компанії до роботи в додатку лише після закінчення бюрократичних процедур з реєстрації.

Рис. 2.3 – Процес створення закладів громадського харчування

2.4 Опис основних вимог до функціоналу користувача типу «Заклад громадського харчування»

Заклад громадського харчування виконуватиме всю обробку замовлень клієнтів, тож повинен мати такі функції:

- можливість змінювати позиції меню (ціна та доступність), адже в окремих випадках існує можливість зміни цих характеристик в залежності від обставин, наприклад, закінчилися великі паперові стаканчики для кави;
- можливість змінювати модифікації меню (по тій же причині);
- можливість переглядати замовлення в режимі реального часу та обробляти їх.

До процесу обробки замовлень відносяться зміна статусів замовлення, перегляд замовлення та коментарів до нього, а також можливість зв’язатися з клієнтом у разі, якщо щось не так.

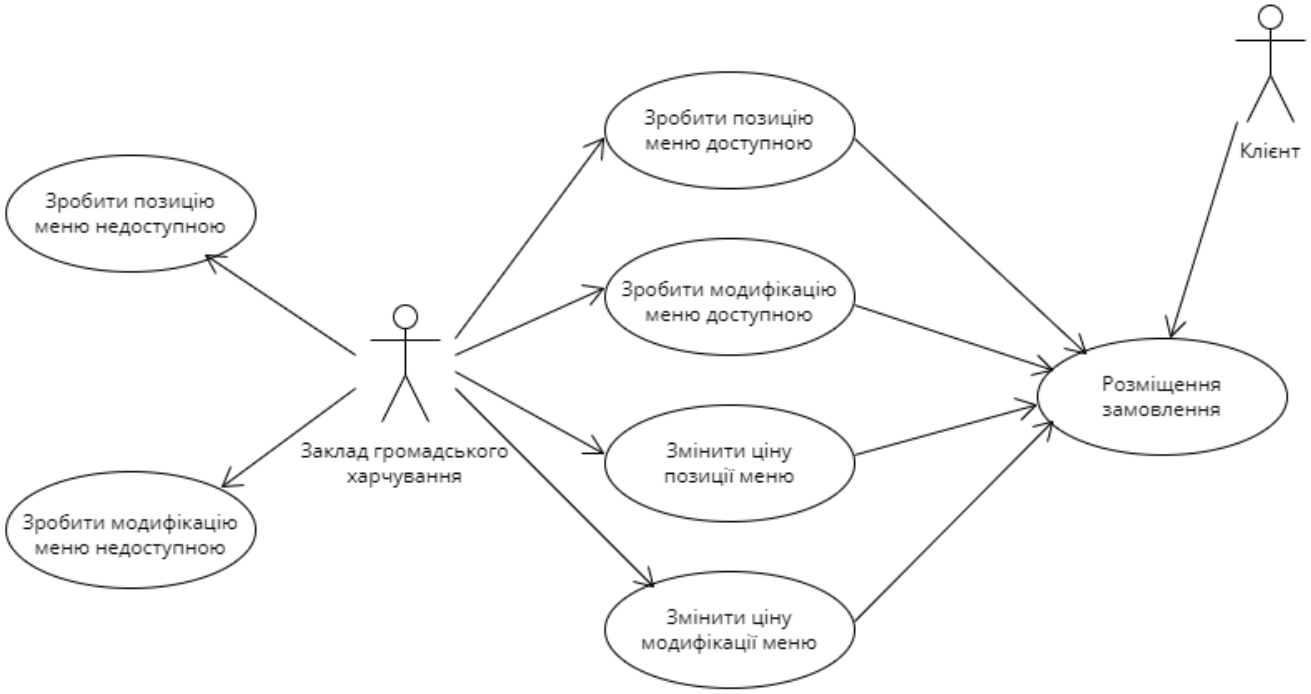


Рис. 2.4.1 – Можливості закладів громадського харчування змінювати доступні опції для замовлення

2.5 Опис основних вимог до функціоналу користувача типу «Адміністратор»

Адміністратор – головний вид облікового запису для управління всіма користувачами системи. Він повинен мати три головних напрямки управління: компанії, клієнти та сервісні аккаунти.

До функціоналу управління компаніями повинні бути включені такі функції як перегляд платіжної інформації та можливість дозволити або заборонити компанії вести свою діяльність за допомогою сервісу. Для перевірки реквізитів та укладання угод, а також для ведення переговорів повинні використовуватися сторонні сервіси, такі як, наприклад, електронна пошта або телефон, адже це є більш надійним та зручним способом комунікацій.

До функціоналу управління клієнтами мусять входити такі можливості як блокування та розблокування. У разі, якщо клієнт веде себе недобросовісно, наприклад, створює велику кількість замовлень, а потім їх відміняє, намагається обдурювати заклади громадського харчування, не оплачує замовлення або пише огидні речі в коментарях до замовлень тощо, адміністратор матиме змогу заборонити такому клієнту створювати нові замовлення. Ці обмеження не діятимуть на перегляд карти закладів та їх меню, але замовлення можна буде зробити лише фізично знаходячись в бажаному закладі громадського харчування.

Для управління сервісними обліковими записами, адміністратор матиме такі функції як їх створення та видалення. Наприклад, у разі необхідності розширення штабу працівників компанії, що займатиметься підтримкою та подальшим розвитком інформаційної системи, кількість працівників служби підтримки або менеджерів (адміністраторів) системи буде збільшуватися, тому необхідність цих облікових записів зростатиме. Очевидно, що облікові записи такого виду не можна дозволяти створювати з форми реєстрації. Для цих цілей і існуватимуть такі функції у адміністраторів, котрі по натисканні кнопки та заповненню необхідних полів, наприклад, ім’я та прізвище працівника, за яким буде закріплено обліковий запис, автоматично генеруватиме пароль та логін для входу в сервісний аккаунт. У

разі втрати паролю, адміністратор матиме змогу згенерувати новий надійний пароль на надати його працівнику. Якщо один з працівників звільниться, адміністратор зможе видалити непотрібний обліковий запис.

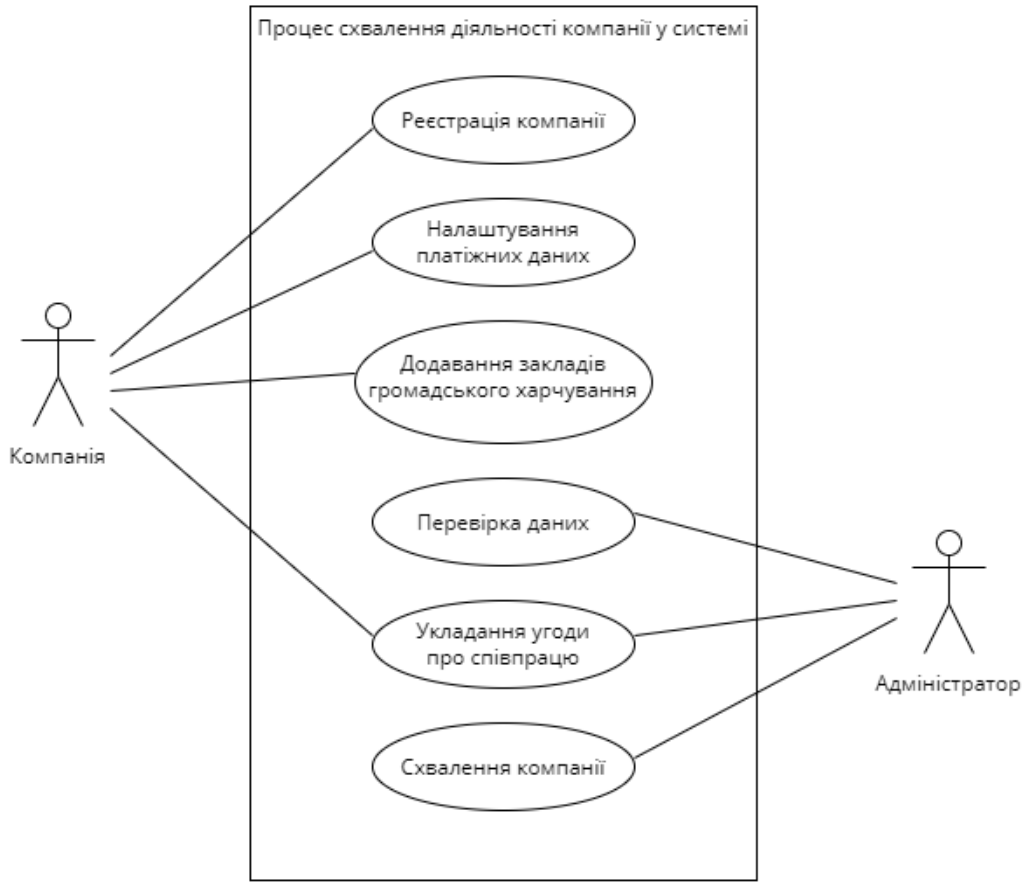


Рис. 2.5.1 – Процес схвалення адміністратором діяльності компанії у системі

2.6 Опис основних вимог до функціоналу користувача типу «Служба підтримки»

Для облікових записів служби підтримки необхідний мінімальний функціонал, адже для зручності працівники використовуватимуть листування електронною поштою та мобільні дзвінки, у разі необхідності. До обов’язкових функцій служби підтримки входитиме обробка скарг та пропозицій клієнтів, тож перш за все їм потрібна можливість змінювати статуси скарг та довідкові дані для встановлення зв’язку з клієнтами та компанією, номер замовлення та заклад громадського харчування, на який з’явилася скарга. Після обробки службою

підтримки, такі скарги можуть бути однією з причин виключення окремих закладів та компаній з системи адміністратором.

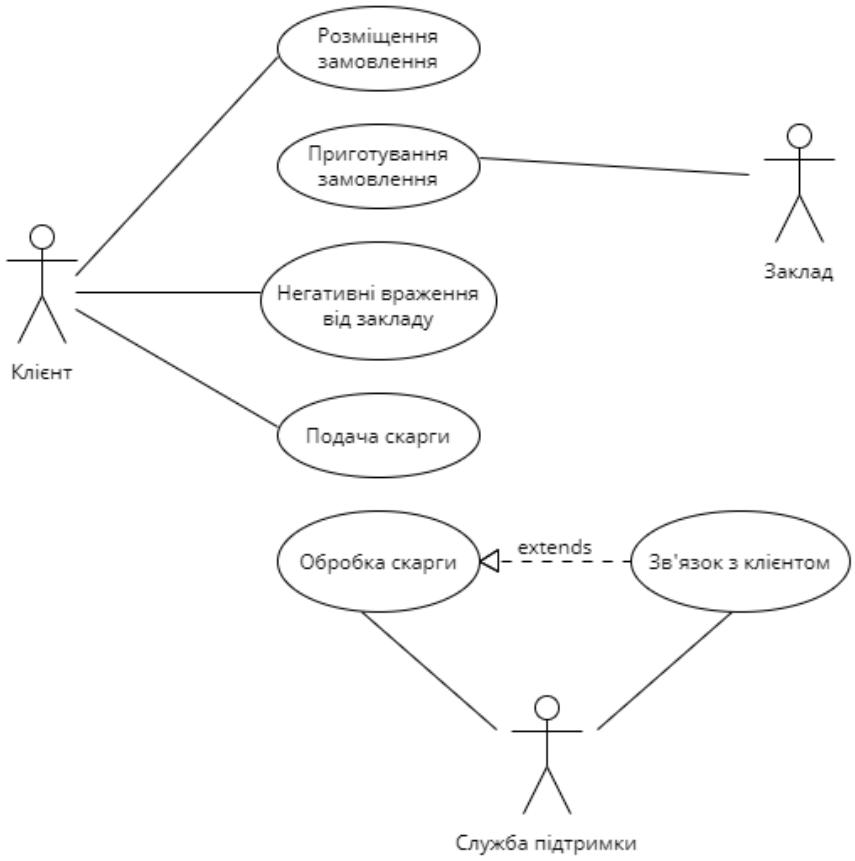


Рис. 2.6.1 – Процес подачі та обробки скарг

2.7 Опис використовуваних технологій та структури додатку

Перед початком розробки необхідно окреслити структуру додатку та описати технології, протоколи передачі даних та принципи, котрі будуть закладені в інформаційну систему. Оскільки додаток матиме клієнт-серверну архітектуру, потрібно провести аналіз продукту у вигляді окремих програмних модулів та частин як клієнту, так і серверу.

Додаток матиме трирівневу архітектуру, а саме фізичне та логічне розподілення на базу даних, сервер та клієнт. За даними компанії IBM, трирівнева архітектура є переважною архітектурою програмного забезпечення для

традиційних клієнт-серверних додатків [9]. До того ж вона має такі переваги як логічне та фізичне розподілення функціональності. Кожен з рівнів має свій виділений або віртуальний сервер (веб-сервер, сервер додатків та сервер бази даних) тому послуги кожного рівня можна налаштувати та оптимізувати без впливу на інші рівні, що сильно впливає на стійкість системи та можливість подальшої розробки. Також варто зазначити, що це дає інші корисні аспекти:

– пришвидшена розробка, оскільки розробку клієнту та серверу, операції з базами даних можна проводити паралельно;
– покращене масштабування, бо кожен з рівнів можна масштабувати окремо;
– підвищена надійність, адже збій на окремому рівні менше впливає на інші;
– покращена безпека, адже клієнтський рівень та рівень бази даних не взаємодіють напряму, тому стає неможливим створення шкідливих SQL ін’єкцій та інших дій.

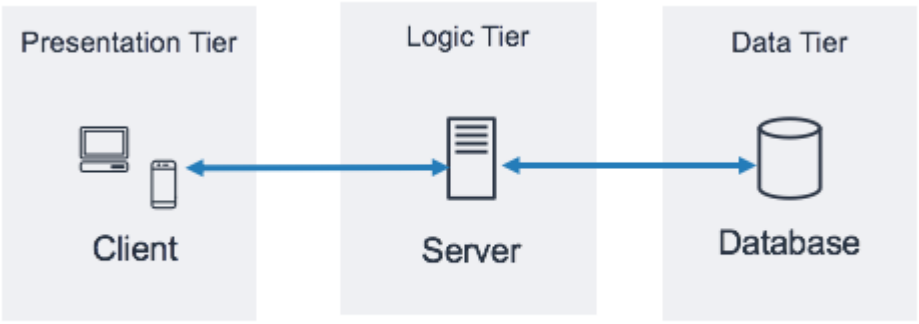


Рис. 2.7.1 – Трирівнева архітектура

Для взаємодії з сервером клієнт використовуватиме HTTP протокол передачі даних, який широко використовується в мережі Інтернет. Цей протокол може передавати текст, зображення, аудіо та відео та інші види даних. За замовчуванням використовує TCP порт 80 [10]. Для запиту на отримання або зміну даних, зазвичай використовують чотири основні види запитів, а саме:

- GET – для запиту на отримання даних з серверу;
- POST – для запиту з відправкою даних, зазвичай на створення чогось у базі даних;
- PUT – для зміни якихось даних;
- DELETE – для видалення даних.

Хоча насправді всі види взаємодій можна виконувати лише за допомогою лише одного методу, наприклад POST, таке розділення функціоналу дозволяє створювати API, які набагато легше розуміти та розвивати.

В свою чергу, серверна частина інформаційної системи матиме триярусну архітектуру, що означає розділення функціоналу на три логічних рівні:

- DAL (Data Access Layer), рівень доступу до даних;
- BLL (Business Logic Layer), рівень бізнес логіки;
- API (Application Programming Interface), рівень взаємодії з клієнтом.

Як було сказано вище, у якості СУБД буде використано Microsoft SQL Server. При розробці інформаційної системи, на рівні бази даних не потрібні будуть ніякі маніпуляції, адже вона буде створена автоматично на базі автоматично згенерованого коду. Такий підхід до БД називають Code-First. Він дозволяє починати роботу з додатком без попереднього налаштування бази даних, що значно спрощує розгортання додатку на сервері.

На рівні DAL знаходяться всі необхідні інструменти для роботи з базою даних, а саме моделі (entity), конфігурації моделей, контекст бази даних, репозиторії для роботи з даними та міграції БД. Фреймворком для створення програмної бази для роботи з БД є Entity Framework.

Entity Framework — це ORM платформа для додатків .NET, яку підтримує Microsoft. Вона дозволяє розробникам працювати з даними, використовуючи об’єкти класів, не зосереджуючи увагу на таблицях і стовпцях бази даних, де ці дані зберігаються. Використовуючи Entity Framework, розробники мають змогу працювати на більш високому рівні абстракції, коли вони мають справу з об’єктами моделей, а не результатом виклику SQL запиту, як у інших засобів.

Моделі – це класи C# з описом полів таблиць, які потім будуть використані Entity Framework для створення таблиць в базі даних. Конфігурації моделей містять детальний опис для EF щодо створення таблиць, наприклад, обмеження довжини рядка або специфіку зв'язків між окремими таблицями. Репозиторії для роботи з даними – це абстракція даних, за допомогою якої розробник має змогу використовувати базу даних, використовуючи низку простих методів, без необхідності мати справу з проблемами БД, такими як підключення, команди, курсори або зчитувачі. Міграції – це засіб керування станом бази даних, використовуючи програмний код. При додаванні чи зміні нових сутностей або властивостей, схеми бази даних повинні бути відповідним чином обновлені для синхронізації з додатком. За допомогою пакету Microsoft.EntityFrameworkCore.Tools генерація та оновлення схеми можна проводити за допомогою простих команд.

На рівні бізнес логіки знаходяться об'єкти, які називаються сервісами, саме в них знаходиться вся логіка додатку. Це найбільша частина серверного додатку, адже всі правила, можливості та процеси обробки даних, які надходять від користувачів системи обробляються тут. Сервіси використовують репозиторії DAL для маніпулювання даними з бази даних. Рівень бізнес-логіки існує тому, що програма вимагає більше, ніж просто отримання та оновлення даних. Саме на цьому рівні відбуваються такі процеси як реєстрація, автентифікація, створення замовлення, надсилання сповіщень та таке інше.

Рівень API існує для відокремлення прийняття HTTP запитів від власне логіки додатку. На цьому рівні зазначаються всі кінцеві точки серверу, які клієнтська частина додатку може використовувати при своїй роботі. Саме тут зазначаються правила, за якими користувачі матимуть змогу звертатися до серверу, приймаються запити клієнта, викликається обробка даних з рівня бізнес логіки та формується відповідь на запит.

При розробці API буде дотримано архітектурний стиль REST, головним правилом якого є те, що сервер повинен відправляти у відповідь на запит лише дані у форматах JSON (найчастіше) або XML через мережевий протокол HTTP [11].

Таким чином, клієнт сам вирішує, який вигляд матимуть ті чи інші дані. Цей підхід до розробки веб-API вважається правильним та «чистим» стилем програмування.

Крім трьох основних рівнів додатку варто виділити четвертий – допоміжний, який буде використовуватися у всіх інших. Це рівень, де зберігаються константи та різноманітні DTO (Data Transfer Object). Зберігання констант необхідне для централізованого управління даними, такими як назви полів у файлі конфігурацій, деякі постійні значеннями з бази даних, адже назва змінної у порівнянні зі значенням рядка або цифрою дає більш конкретне розуміння принципів та правил їх роботи при розробці. DTO – це простий клас C#, який використовується для передачі даних між клієнтом і сервером. Такі моделі необхідні, адже користувачу не завжди потрібні або дозволені всі дані з таблиць БД, до того ж під час процесу, що називається маппинг, з моделі бази даних можна створювати об’єкт, який включатиме в себе більше інформації.

Також, серверна частина повинна мати підтримку логування непередбачених помилок, щоб процес їх знаходження та виправлення займав менше часу. Досягається це за допомогою так званого middleware, або проміжного обробника, сутності, яка має доступ до об’єкту запиту на рівні між клієнтом та обробником на стороні серверу. Також тут виконується серіалізація та десеріалізація запитів та даних, які вони несуть.

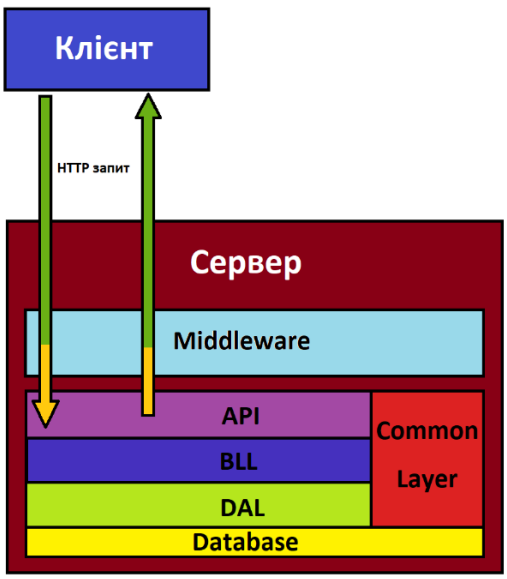


Рис. 2.7.2 – Загальна структура серверної частини додатку

Клієнтська частина додатку, що буде написана на Angular, основуватиметься на модулях, що є структурною одиницею додатку на даному фреймворку [12]. Модулі в свою чергу містять компоненти, які визначають представлення, котрі являють собою набори елементів екрану, змінюючи які у відповідності до логіки додатку Angular створює веб-сторінки. Самі компоненти використовують сервіси, які являють собою набори методів, не пов'язаних напряму з відображенням даних. Сервіси впроваджуються у компоненти за допомогою процесу DI (Dependency Injection), що відповідає шаблону проектування IoC (Inversion of Control), який голосить про те, що класи повинні мати мінімальну залежність один від одного, тобто мати слабку зв'язність, оскільки це дозволяє легше тестувати та підтримувати код [13]. Для маркування класів як модулів, компонентів або сервісів використовуються декоратори – спеціальні конструкції, які допомагають системі визначати призначення класів, створювати та управляти ними відповідно. Будь-який додаток Angular має як-найменше один модуль та компонент, так звані кореневі модуль і компонент, на їх базі створюються всі інші.

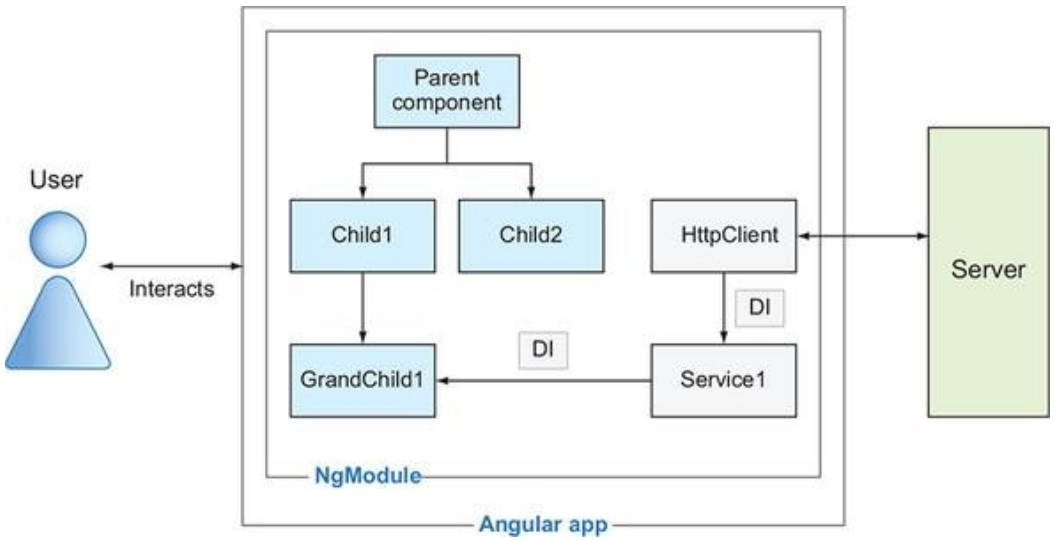


Рис. 2.7.3 – Структура додатку Angular

Готовий додаток повинен мати якнайменше вісім модулів, а саме кореневий, спільний, модуль для реєстрації та входу і по одному для кожного виду облікових записів. За допомогою такого розподілення можна полегшити додаток для

користувачів, адже при роботі з системою, користувач не буде підвантажувати інші компоненти та створювати об’єкти сервісів, що позитивно позначиться на продуктивності системи.

Сервіси поділятимуться на три види:

- для доступу до даних з серверу, використовуючи протокол HTTP за допомогою стандартних класів бібліотеки angular/core;
- для зберігання стану системи, як, наприклад, обрана сторінка відображення;
- для виконання стандартних дій, як виклик вікна для підтвердження дії, показу помилок тощо.

Для створення приємного UI (інтерфейсу користувача) використовуватимуться готові збалансовані бібліотеки Bootstrap та Angular Material.

Bootstrap – це безкоштовний інтерфейсний фреймворк для простої та швидкої розробки веб-розробки. Він включає в себе шаблони дизайну на основі HTML і CSS для тексту, таблиць, кнопок, навігації тощо. Крім того, має додаткові плагіни на JavaScript для створення, наприклад, каруселей зображень. Також, Bootstrap дає можливість просто створювати адаптивний дизайн, що буде особливо цінним для створення даної інформаційної системи, адже різні види користувачів використовуватимуть різні операційні системи для доступу до додатку.

Angular Material – це бібліотека компонентів інтерфейсу користувача, яку розробники можуть використовувати в додатках, написаних за допомогою Angular, для пришвидшення розробки приємних інтерфейсів. Як і сам фреймворк, розробляється командою Google, тому завжди має максимальну сумісність версій.

Також, для додавання мапи з закладами громадського харчування буде використаний Google Maps API, за допомогою якого мапа на веб-сторінці буде генеруватися автоматично. Це простий сервіс від компанії Google, який дозволяє використовувати їх додатки прямо з коду.

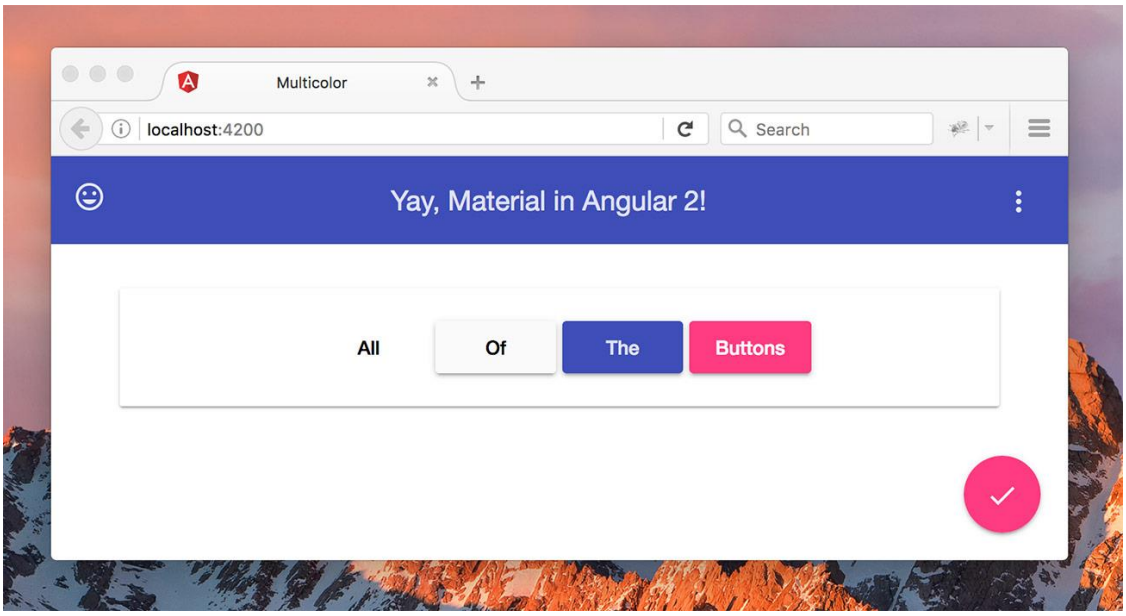


Рис. 2.7.4 – Приклад інтерфейсу Angular Material

3. ОПИС РОЗРОБКИ ІНФОРМАЦІЙНОЇ СИСТЕМИ

3.1 Створення структури серверної частини програмного продукту

Перед початком власне розробки необхідно створити проєкт та налаштувати його. Середовищем розробки для C# було обрано Microsoft Visual Studio, адже самец я компанія займається розробкою та підтримкою як платформи .NET, та мови C#, так і розробкою більшості бібліотек та фреймворків для розробки, тому ця IDE (Integrated Developer Environment) є найпопулярнішою для даної мови програмування.

Відкриємо Microsoft Visual Studio та створимо новий проект типу «ASP.NET Core Web API», що містить необхідну структуру для створення веб-API. Необхідно обрати назву проєкту та рішенню. Оскільки рішення – це назва всієї серверної архітектури та міститиме всі яруси програми, назва у нього буде загальна. Я вирішив назвати додаток «Contactless Order» (безконтактне замовлення). Для проєкту необхідно задати більш конкретне ім’я – ContactlessOrder.Api, оскільки саме цей проект буде запускати систему та приймати запити від клієнту.

Після нього необхідно додати до рішення проєкти, які відображатимуть решту ярусів серверу. Вони мають тип «Бібліотека класів» та називаються відповідно: загальна назва рішення з приставкою назви ярусу.

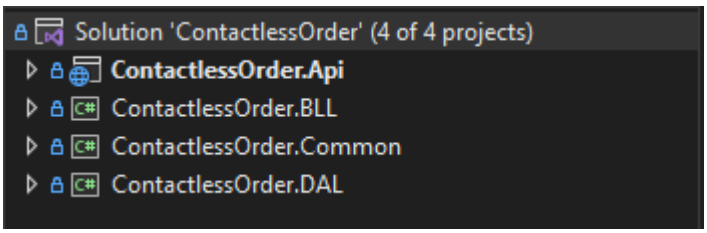


Рис. 3.1.1 – Яруси серверного додатку

Після створення пустих проєктів, необхідно розробити їх структуру. Варто зазначити, що проектування системи опиралося на принципи SOLID, тому усі сервіси та репозиторії повинні реалізовувати інтерфейси, під обгорткою яких ми і

будемо користуватися їх функціоналом. Тому шари DAL та BLL матимуть папку Interfaces для кращого розподілення структури.

Таким чином, шар DAL матиме такі папки:

– EF для усього необхідного для функціонування EntityFramework;
– Entities для збереження моделей бази даних;
– Interfaces для інтерфейсів репозиторіїв;
– Migrations, автоматично згенерована папка, в якій містяться міграції бази даних;
– Repositories, яка містить реалізації репозиторіїв.

Шар зі спільними елементами матиме дві папки: Constants і Dto. В першій міститимуться статичні класи для зберігання констант, а в другій знаходитимуться моделі транспортування даних.

BLL матиме у своєму складі чотири папки з рядом підпапок, адже саме тут розташована логіка програми. Це такі папки як Services, Interfaces, Infrastructure та HubConnections. Папка Infrastructure необхідна для винесення допоміжних сервісів, які виконують якусь логіку, проте не потребують для себе ніяких особливих структур даних. Папка HubConnections слугуватиме місцем, де будуть зберігатися всі сутності для забезпечення роботи пакету SignalR, який слугуватиме мостом між сервером в напрямку клієнта, за допомогою якого будуть надсилатися сповіщення про зміну статусу замовлення та ін.

API проект матиме складнішу структуру, адже тут розміщатимуться різноманітні обробники запитів. Головному папкою тут є Controllers, яка містить контроллери, котрі власне і оголошують методи, прив’язані до адреси. Також тут є такі папки:

– Hubs – конфігураційні файли для SignalR;
– JsonConverters – контератори різних типів до формату JSON та навпаки;
– Middleware – проміжкові обробники;
– Validators – файли з логікою валідації моделей.

Реалізації окремих файлів та логіки буде розібрано трохи нижче.

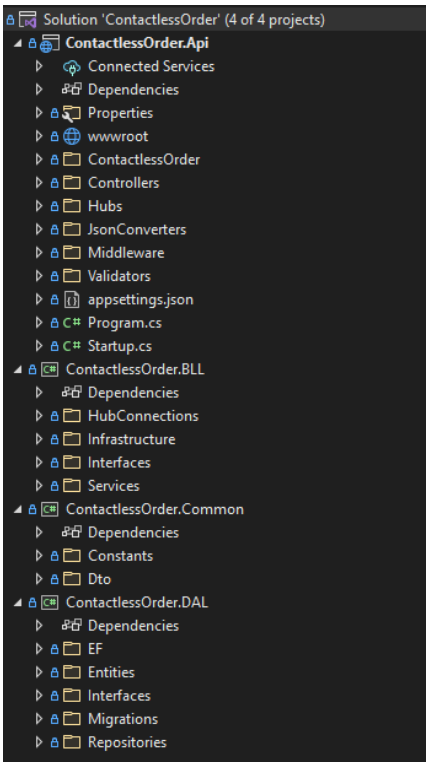


Рис. 3.1.2 – Загальна структура серверу

3.2 Розробка реєстрації та входу в додаток на стороні серверу

Для початку необхідно визначити, які таблиці необхідні для створення користувачів системи. Для реалізації функціоналу в зазначених вище вимогах потрібно чотири таблиці, деталі яких зазначені у таблиці 3.2.1. Варто уточнити, що це не всі стовці в таблицях, а лише необхідні для ведення обліку користувачів.

Таблиця 3.2.1 – Таблиці, необхідні для реалізації користувачів системи

Назва	**Назва поля, тип**	**Опис поля**	**Призначення таблиці**
Users	FirstName, string	Ім’я	Кожен запис у таблиці відображає окремого користувача системи
	LastName, string	Прізвище	
	Email, string	Електронна пошта	
	PhoneNumber, string	Телефонний номер	
	PasswordHash, string	Хеш паролю	
	EmailConfirmed, bool	Чи підтверджена адреса електронної пошти	
	RoleId, int	Посилання на роль	
	RegistrationDate, DateTime	Дата реєстрації	

Продовження таблиці 3.2.1

Назва	Назва поля, тип	Опис поля	Призначення таблиці
Roles	Name, string	Назва	Містить всі ролі користувачів системи
	Value, int	Константне значення для розрізнення окремих ролей	
Companies	Name, string	Назва	Зберігає дані компаній
	Address, string	Адреса	
	Description, string	Опис	
	UserId, int	Посилання на користувача, прив'язаного до компанії	
Caterings	Name, string	Назва	Зберігає дані закладів громадського харчування
	Services, string	Короткий опис послуг	
	UserId, int	Посилання на користувача	
	CompanyId, int	Посилання на компанію	

Для наочного зображення необхідних зв'язків у базі даних, потрібно створити діаграму БД для цих таблиць.

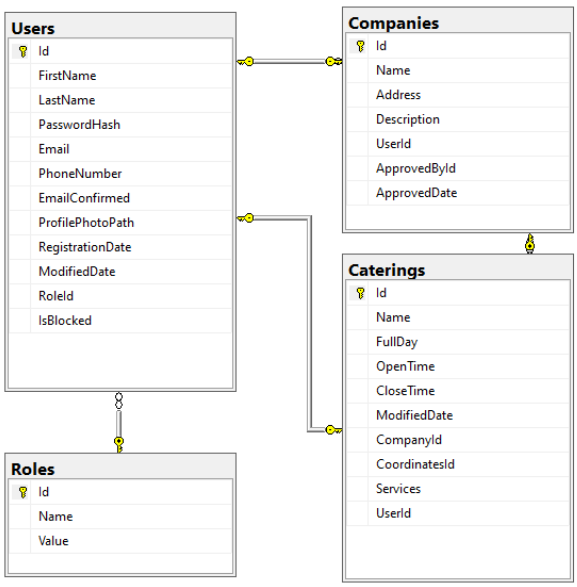


Рис. 3.2.1 – Діаграма класів, необхідних для ведення облікових записів користувачів системи

Як зазначалося раніше, для створення таблиць у базі даних з використанням Entity Framework необхідно створювати модель для кожної таблиці. Таким чином, необхідно створити чотири класи з відповідними назвами у правильних кладених папках Entities. Потім, необхідно заповнити їх відповідно до стовбців спроектованих таблиць та завдяки можливості фреймворку для роботи з БД можна додати так звані навігаційні властивості – властивості типу, який втілює якусь таблицю в базі даних, тобто Entity Framework має змогу витягувати додаткові дані з інших таблиць за допомогою одного запиту. Так, наприклад, навігаційними властивостями класу User є Role, Company, Catering.

```
3
4   namespace ContactlessOrder.DAL.Entities.Users
5   {
        44 references
6       public class User
7       {
            7 references
8           public int Id { get; set; }
            10 references
9           public string FirstName { get; set; }
            10 references
10          public string LastName { get; set; }
            10 references
11          public string PasswordHash { get; set; }
            18 references
12          public string Email { get; set; }
            10 references
13          public string PhoneNumber { get; set; }
14
            3 references
15          public bool IsBlocked { get; set; }
            11 references
16          public bool EmailConfirmed { get; set; }
            7 references
17          public string ProfilePhotoPath { get; set; }
18
            8 references
19          public DateTime RegistrationDate { get; set; }
            3 references
20          public DateTime? ModifiedDate { get; set; }
21
            6 references
22          public int RoleId { get; set; }
            15 references
23          public Role Role { get; set; }
24
            12 references
25          public Company Company { get; set; }
            1 reference
26          public Catering Catering { get; set; }
27      }
28  }
```

Рис. 3.2.2 –Модель БД на прикладі класу User

Після створення всіх моделей, необхідно додати власне сам пакет Microsoft.EntityFrameworkCore до проекту DAL та декілька допоміжних, які дозволять працювати з базою даних SQL Server (Microsoft.EntityFrameworkCore.SqlServer) та розширити можливості і полегшити

розробку (Microsoft.EntityFrameworkCore.Tools). Для цього потрібно перейти у меню «Управління пакетами NuGet» для DAL та за допомогою пошуку знайти ці розширення.

Після їх встановлення перейдемо до папки, створеної для Entity Framework та додамо сюди клас, який являтиме собою контекст бази даних. Це простий клас, який наслідується від класу DbContext та реалізовує конструктор і кілька простих методів. Всю іншу роботу з підключення та виконання різних операцій фреймворк бере на себе. У цій же папці EF створимо ще одну – EntityConfigurations, яка міститиме конфігурації моделей в базі даних. У цій папці створимо класи, які описуватимуть певні аспекти таблиць. Вони повинні реалізовувати інтерфейс IEntityTypeConfiguration<T>, де T – це тип моделі [14]. Таким чином, ми позначаємо ці класи такими, які впливають на схему бази даних, тому при створенні міграції такі зміни будуть прочитані та відповідним чином описані оновленою схемою.

```
8   namespace ContactlessOrder.DAL.EF.EntityConfigurations.Users
9   {
        0 references
10      public class UserConfiguration : IEntityTypeConfiguration<User>
11      {
            0 references
12          public void Configure(EntityTypeBuilder<User> builder)
13          {
14              builder.ToTable("Users");
15
16              builder.Property(e => e.FirstName).IsRequired().HasMaxLength(250);
17              builder.Property(e => e.LastName).IsRequired().HasMaxLength(250);
18              builder.Property(e => e.PasswordHash).IsRequired().HasMaxLength(250);
19              builder.Property(e => e.Email).IsRequired().HasMaxLength(250);
20              builder.Property(e => e.PhoneNumber).IsRequired().HasMaxLength(250);
21              builder.Property(e => e.ProfilePhotoPath).HasMaxLength(500);
22
23              builder.Property(e => e.RegistrationDate)
24                  .HasConversion(d => d, d => DateTime.SpecifyKind(d, DateTimeKind.Utc));
25
26              builder.Property(e => e.ModifiedDate)
27                  .HasConversion(d => d, d => DateTime.SpecifyKind(d.Value, DateTimeKind.Utc));
28
29              builder.HasOne(e => e.Role).WithMany().HasForeignKey(e => e.RoleId).OnDelete(DeleteBehavior.Restrict);
30          }
31      }
32  }
33
```

Рис. 3.2.3 – Конфігурація моделі User

Після опису всіх моделей, необхідно задати рядок підключення до бази даний. Це правильно робити за допомогою файлу конфігурацій appsettings.json. До

нього потрібно додати розділ ConnectionStrings, в якому задати ім’я рядку та його значення:

```
{
  “ConnectionStrings”: {
    “COLocal”: “Server=localhost;Database=ContactlessOrder;Trusted_Connection=True;”
  },
  …
}
```

Після цього варто створити файл AppConstants у проекті Common і додати туди константу назви рядка підключення до бази даних. Далі можна налаштувати підключення до БД у коді, що при запуску додатку він зміг її використовувати. Для цього необхідно додати наступний рядок до класу Startup у метод ConfigureServices:

```
services.AddDbContext<ContactlessOrderContext>(options =>
    options.UseSqlServer(Configuration.GetConnectionString(AppConstants.DBNameLocal)));
```

Тепер можна перейти до додавання міграції, яка створить базу даних. Для цього необхідно відкрити Package Manager Console у Visual Studio та ввести команду Add-Migration AccountsMigration, після чого буде додано міграцію з назвою AccountsMigration, у якій буде описаний процес створення БД та таблиць у ній. Також, за допомогою SQL скипту варто додати код, який створює п’ять ролей та одного користувача з роллю адміністратора. Тепер після запуску програми ми можемо побачити, що створилася база даних та таблиці, відповідно до створених та описаних моделей.

Далі потрібно створити репозиторій для роботи з базою даних. Оскільки всі репозиторії будуть працювати з контекстом та мати схожі методи, було вирішено створити базовий інтерфейс репозиторію IRepositoryBase та його абстрактну реалізацію RepositoryBase, яка приймає контекст БД у якості аргумернту конструктора. Репозиторій включає в себе операції додавання, отримання та видалення даних з бази даних та дозволяє зберігати зміни окремим методом. Отримання рядку з таблиці БД може бути виконане як по унікальному ідентифікатору (Id), так і по спеціальній лямбда-функції – предикаті, яка приймає

об’єкт з таблиці та повертає логічне значення true або false, яке означає чи підходить об’єкт для вибірки (Рис. 3.2.4).

Після цього створюємо UserRepository, який власне і буде використано при реєстрації та автентифікації користувача. Сюди варто додати два методи – для отримування користувача по адресі електронної пошти та по ідентифікатору. Варто зазначити, що e-mail – це унікальне поле, перевірку унікальності якого, як і номеру телефону, буде реалізовано далі. Також, варто створити можливість входити по імені користувача, яке буде дорівнювати адресі електронної пошти до знаку «@»:

```
public async Task<User> GetUser(string email)
{
    return await Context.Set<User>()
        .Include(e => e.Company)
        .Include(e => e.Catering)
        .Include(e => e.Role)
        .FirstOrDefaultAsync(e => e.Email == email
          || (e.Email != null &&  e.Email.StartsWith($"{email}@")));
}
```

Таким чином ми будемо перевіряти наявність користувача при вході або реєстрації.

Перед переходом до рівня бізнес логіки, до цього проекту варто додати AutoMapper – пакет, який містить необхідну інфраструктуру для зручного та швидкого перенесення інформації з моделей бази даних у DTO та навпаки. Такий процес називається маппинг (від англ. mapping). Цей пакет дозволяє створювати складні правила перенесення інформації за допомогою простих методів, у разі якщо необхідні маніпуляції з даними, або взагалі лише через оголошення, якщо імена властивостей співпадають [15]. Створення правил переносу відбувається у спеціальних класах, які називаються Mapping Profile. Вони повинні унаслідуватися від абстрактного класу Profile, який постачається в пакеті AutoMapper та реалізовує методи для створення правил перетворення між типами. При запуску програми, всі класи, що унаслідуються від Profile будуть зібрані разом та по вказаним в них правилам буде створено об’єкт мапперу, що і буде введено для всіх користувачів за допомогою DI.

Для уникнення повторів електронних пошт та номерів телефону, необхідно створити сервіс, який дозволить зручно перевіряти такі умови, не задумуючись про

саму реалізацію методів. Такий сервіс матиме назву ValidationService, та крім валідації пошти та номеру тут будуть реалізовані методи повної валідації користувача та компанії.

```
3 references
public async Task<string> ValidatePhoneNumber(string phoneNumber, int? id = null)
{
    if (string.IsNullOrWhiteSpace(phoneNumber))
    {
        return "Телефонний номер не може бути пустим";
    }

    var user = await _repository.Get<User>(e => e.Id != id && e.PhoneNumber == phoneNumber);
    if (user != null)
    {
        return "Телефонний номер вже використовується";
    }

    return string.Empty;
}
```

Рис. 3.2.4 – Метод перевірки унікальності номеру телефону

Тепер можна створювати AuthService – сервіс, який буде працювати з обліковими записами. В інтерфейсі сервісу необхідно об’явити п’ять методів:

- Authenticate – автентифікація за допомогою адреси електронної пошти та паролю;
- GoogleAuthenticate – автентифікація за допомогою облікового запису Google;
- Register – реєстрація в системі користувача;
- RegisterCompany – реєстрація в системі компанії;
- ConfirmEmail – підтвердження адреси електронної пошти користувача.

Сам процес авторизації буде заснований на JWT (JSON Web Token), що широко використовується сьогодні по всьому світі [16]. Вона полягає в тому, що разом з усіма запитами, користувач також повинен надавати зашифрований код (токен) у розділі заголовків HTTP. Цей код генерується при автентифікації користувача та може містити в собі такі дані як ідентифікатор користувача, ім’я, роль тощо, але найголовніше це кінцева дата придатності, після якої використання

токену заборонено. Токен шифрується за допомогою ключового рядку, який також зберігається на сервері у файлі конфігурацій.

Першим буде реалізовано метод реєстрації, який приймає об’єкт класу UserRegisterRequestDto, який містить всі обов’язкові поля користувача. В першу чергу відбувається перевірка всіх унікальний полів на наявність у БД. Після цього, за допомогою маппера виконується перетворення UserRegisterRequestDto у User, задається роль клієнта, дата реєстрації, зазначається, що електронна пошта не підтверджена та встановлюється хеш паролю. База даних не містить паролі напряму, адже це дуже небезпечно у разі, якщо вона потрапить до рук зловмисників. Тому тут зберігається лише його хеш – рядок, який є його зашифрованою версією. Після цього, за допомогою репозиторію новостворений користувач додається у таблицю користувачів та на зазначену адресу електронної пошти відправляється лист з посиланням для її підтвердження. Після натискання на посилання, користувач повинен буде перейти до додатку клієнта, який надішле запит до серверу для підтвердження адреси.

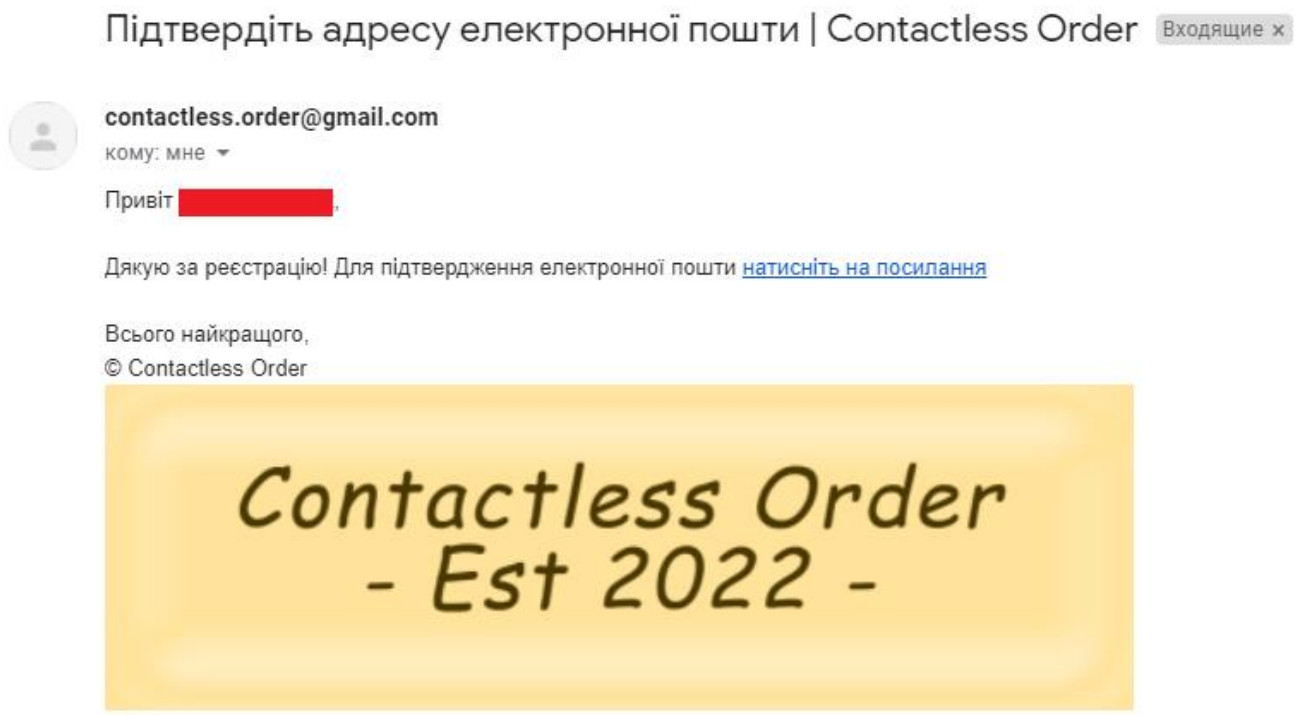


Рис. 3.2.5 – Приклад листа для підтвердження адреси електронної пошти

Для входу користувачі використовуватимуть логін або адресу електронної пошти та пароль від облікового запису. Метод простий та перевіряє базові умови доступу, після чого, у разі успіху, генерує та відправляє токен авторизації.

```
2 references
public async Task<ResponseDto<string>> Authenticate(UserLoginRequestDto dto)
{
    var passwordHash = CryptoHelper.GetMd5Hash(dto.Password);
    var user = await _repository.GetUser(dto.Email);

    if (user == null)
    {
        return new ResponseDto<string>() { ErrorMessage = "Користувач не знайдений" };
    }
    else if (user.PasswordHash != passwordHash)
    {
        return new ResponseDto<string>() { ErrorMessage = "Невірний пароль" };
    }
    else if (!user.EmailConfirmed)
    {
        return new ResponseDto<string>() { ErrorMessage = "Перевірте скриньку електронної пошти для підтвердження адреси" };
    }

    return new ResponseDto<string>() { Response = GenerateToken(user) };
}
```

Рис. 3.2.6 – Метод входу до системи

Після реалізації інших методів, які мають схожу логіку, можна переходити до створення першого контролеру, за допомогою якого клієнтська частина зможе використовувати можливості входу та реєстрації за допомогою протоколу HTTP. Контролер матиме назву AuthController, матиме атрибути ApiController та Route зі стандартним шляхом «api/[controller]». Всі REST контролери повинні унаслідуватися від класу ControllerBase, адже він надає можливість надсилати відповіді на запити з потрібними HTTP статусами (200 OK, 400 BadRequest, тощо).

За допомогою DI, в контролер буде введено вже розроблені сервіси для максимальної реалізації на front-end. Таким чином при реєстрації, ValidationService буде використовуватися для перевірки адреси електронної пошти на унікальність в режимі реального часу.

Розглянемо метод автентифікації в систему. Він має атрибут HttpPost, що означає, що для його застосування клієнт має відправити запит типу POST на необхідну адресу. Оскільки точний шлях не вказано, метод використовуватиме адресу контролера «api/Auth». Метод приймає об’єкт типу UserLoginRequestDto з

атрибутом FromBody, що означає, що він очікує отримати цей об’єкт з тіла HTTP запиту. У самому методі відбувається виклик методу з AuthService та в залежності від результату формується відповідь клієнту.

```
[HttpPost]
0 references
public async Task<IActionResult> Authenticate([FromBody] UserLoginRequestDto dto)
{
    var response = await _authService.Authenticate(dto);

    if (!string.IsNullOrEmpty(response.ErrorMessage))
    {
        return BadRequest(new { message = response.ErrorMessage });
    }

    return Ok(new { Token = response.Response });
}
```

Рис. 3.2.7 – Метод автентифікації контролера

Після реалізації всіх інших методів цього контролера, можна переходити до налаштування Dependency Injection та тестування методів. Для того щоб задати конкретну реалізацію інтерфейсів репозиторіїв та сервісів, які потрібно використовувати у ході виконання програми, у методі ConfigureServices класу Startup необхідно створити пари за допомогою методу AddTransient:

```
services.AddTransient<IValidationService, ValidationService>();
services.AddTransient<IAuthService, AuthService>();

services.AddTransient<IUserRepository, UserRepository>();
```

Рис. 3.2.8 – Налаштування Dependency Injection

Також, за допомогою методу AddControllers система додає до свого списку шляхів всі класи, які наслідуються від ControllerBase, у цьому випадку контролери з папки Controllers. Крім того, у цілях безпеки, в налаштуваннях проекту ContactlessOrder.API необхідно задати використання SSL, що надасть можливість приймати HTTP запити по захищеному протоколу HTTPS [17], який не дозволяє зловмисникам перехоплювати інформацію, що міститься у запиті.

Тепер необхідно протестувати створений контролер. Це відбуватиметься за допомогою програми Postman – додатку, призначеного для перевірки HTTP/HTTPS запитів з клієнта на сервер та отримання відповіді від нього.

Після запуску серверу необхідно відкрити додаток, відкрити нову сторінку та обрати тип запиту. Оскільки тестування буде проводитися від реєстрації до входу, спочатку буде протестовано реєстрацію. Для цього обхідно обрати тип запиту POST, вказати адресу «https://localhost:5001/api/Auth/Register» перейти у розділ Body, обрати raw, а потім JSON та описати модель UserRegisterRequestDto у JSON форматі та натиснути кнопку Send.

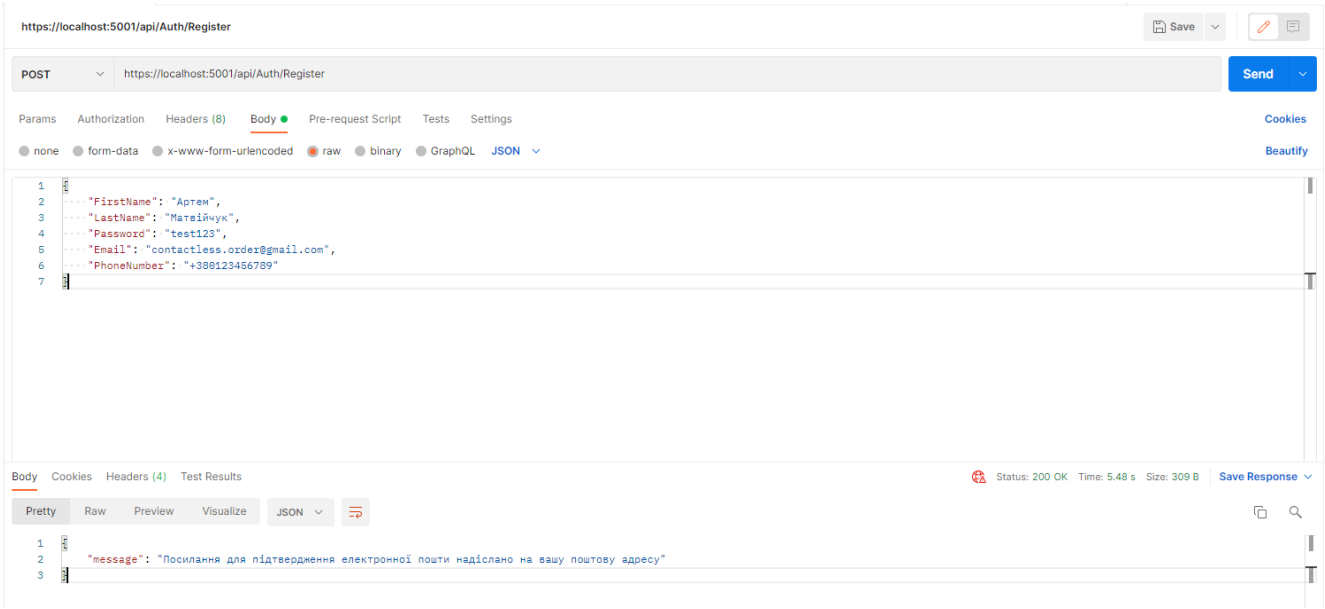


Рис. 3.2.9 – Реєстрація за допомогою Postman

Як видно з відповіді серверу, обліковий запис зареєстровано. Для підтвердження електронної пошти необхідно розробити клієнт, який буде надсилати відповідний запит, тому вхід можна протестувати на тому користувачу, який було створено під час міграції. Надіславши дані для входу, сервер надав згенерований токен авторизації, який буде використано для доступу до контролерів, які потребують авторизації. Також тут можна протестувати методи на перевірку унікальності електронної пошти, номеру телефону тощо.

3.3 Створення структури клієнтської частини програмного продукту

Для роботи з Angular в першу чергу необхідно встановити Node.js. Це платформа для роботи фреймворків, працюючих на JavaScript [18]. Після цього необхідно встановити Angular за допомогою команди «npm install -g @angular/cli@latest», введеної у терміналі Microsoft Visual Studio Code – простої та зручної IDE для розробки front-end. Коли фреймворк буде встановлено, командою «ng new ContactlessOrder.UI» створюється новий проект, який виступатиме клієнтом додатку. У меню вибору формату стилів таблиць необхідно обрати SCSS – удосконалений варіант CSS з більш зручним стилем коду та більшими можливостями, який однак всеодно при запуску додатку переводиться у CSS.

Після створення додатку необхідно у папку src/app додати сім підпапок – по одній на основний модуль програми, а саме:

- admin для модуля адміністраторів;
- catering для модуля закладів громадського харчування;
- client для модуля клієнтів;
- company для модуля компаній;
- login для модуля входу, реєстрації та підтвердження адреси електронної пошти;
- shared для модуля, в якому буде зібраний спільний функціонал;
- support для модуля служби підтримки.

У кожній з папок необхідно створити кілька файлів: module, routing-module, service та component. Перший – це власне модуль, в ньому описується всі компоненти, які він оголошує та модулі і сервіси, які потребує. У другому знаходяться правила перенаправлення URL адреси, який саме компонент чи модуль повинен відображатися в конктретному місці програми. Третій файл оголошуватиме сервіс, за допомогою якого модуль буде мати змогу надсилати запити на сервер. Четвертий потрібен саме для можливості перенаправлення routing-module та відображення вмісту, який буде розроблений згодом.

Після створення усіх структурних компонентів модулів, необхідно задати правила переходів між ними у кореневому app-routing.module.

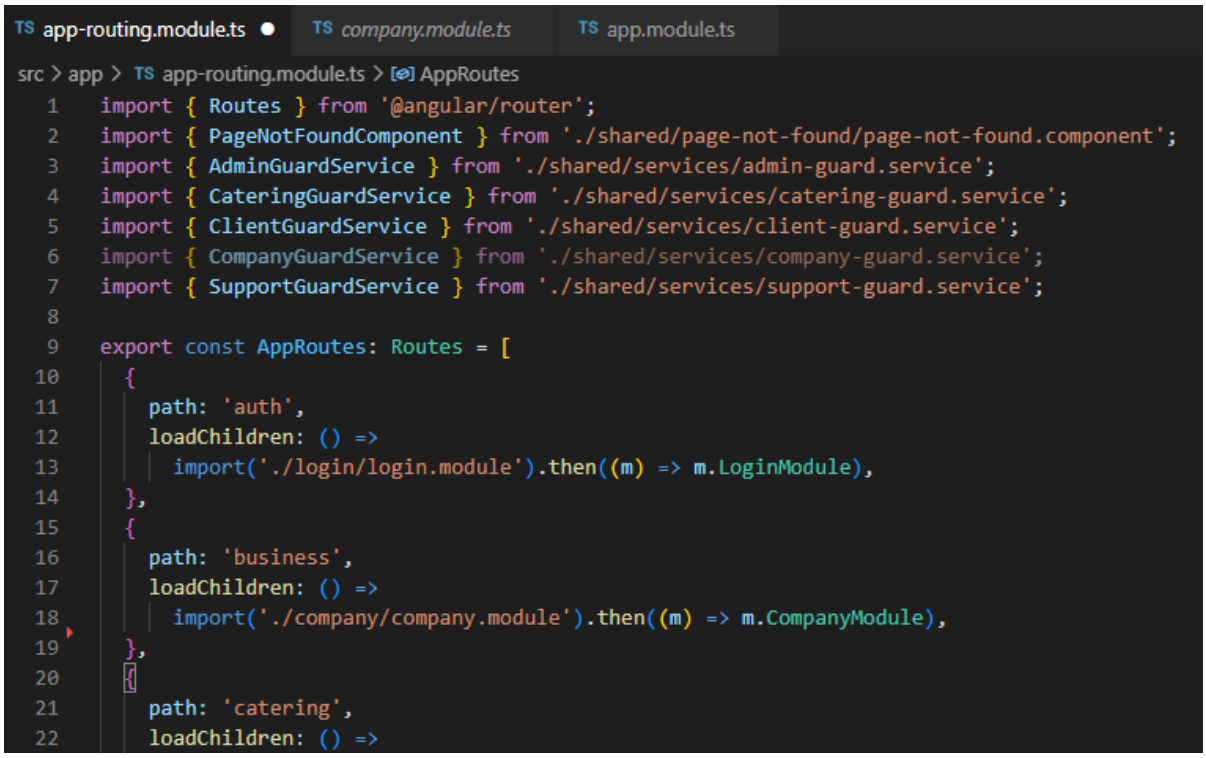


Рис. 3.3.1 – Приклад вмісту app-routing.module

Після цього час реалізувати перший сервіс, від якого будуть наслідуватися всі інші. Це буде AuthService з модуля shared. Для його створення методом DI система вводитиме класи Router та HttpClient. Router – стандартний клас Angular, який дозволяє розробнику перенаправляти користувача між сторінками. HttpClient – також клас, що входить до стандартних бібліотек Angular і дозволяє надсилати HTTP запити. Крім того, тут знаходитиметься URL адреса серверної частини додатку. Насправді, як константне значення, вона знаходитиметься у глобальних змінних environment та environment.prod, які виступають у ролі тримачів констант для debug та release версій програми відповідно.

У цьому сервісі необхідно створити методи login, googleLogin, register, registerCompany, confirmEmail, validateEmail, validatePhoneNumber та validateCompanyName, які будуть використовувати HttpClient, а також ряд допоміжних для інший цілей. Наприклад, метод, що дозволить отримувати токен

авторизації, який після успішної автентифікації буде зберігатися у localStorage браузеру користувача.

```
@Injectable({
  providedIn: 'root',
})
export class AuthService {
  public url = environment.apiURL;

  constructor(
    protected readonly _router: Router,
    private readonly _httpClient: HttpClient
  ) {}

  public async login(email, password): Promise<any> {
    return this._httpClient
      .post<any>(this.url + '/api/Auth', {
        email,
        password,
      })
      .toPromise();
  }

  public googleLogin(sendData: any) {
    return this._httpClient
      .post<any>(this.url + '/api/Auth/GoogleLogin', sendData)
      .toPromise();
  }

  public register(formValue: any) {
    return this._httpClient
      .post<any>(this.url + '/api/Auth/Register', formValue)
      .toPromise();
  }
```

Рис. 3.3.2 – Приклад реалізації AuthService

Тепер необхідно забезпечити недоступність різних видів користувачів до чужих модулів по URL. Для цього у Angular існують так звані Guard, які можна застосовувати на конктретний шлях, для якого буде виконано метод, що поверне логічне значення true або false, що означає чи може користувач звернутися до цього ресурсу. Так, наприклад, авторизований користувач не повинен мати змоги перейти на сторінку реєстрації. Для цього необхідно створити LoginGuardService та у login.module задати його як єдиний елемент масиву властивості canActivate для необхідного шляху. Таким чином необхідно створити необхідні обробники для інших модулів, що будуть перевіряти роль користувача в системі.

```
@Injectable({
  providedIn: "root",
})
export class LoginGuardService implements CanActivate {
  constructor(private route: Router, private authService: AuthService) {}

  canActivate() {
    if (this.authService.isLoggedIn()) {
      this.route.navigateByUrl("dashboard");
      return false;
    }
    return true;
  }
}
```

Рис. 3.3.3 – Клас LoginGuardService

3.4 Розробка реєстрації та входу в додаток на стороні клієнта

Першим етапом до створення функціоналу для реєстрації входу та виходу із системи є конфігурація системи для використання JWT. Першим етапом є додавання бібліотеки @auth0/angular-jwt до проекту за допомогою команди «npm install @auth0/angular-jwt» введеної у терміналі Visual Studio Code. Після цього необхідно додати JwtModule до AppModule, зазначивши постачальника налаштувань JWT. Для цього за шаблоном проектування «Фабрика» необхідно створити метод, який буде повертати збережений токен. Це метод tokenGetter з AuthService, тому тут варто додати лише кілька рядків.

```
export function jwtOptionsFactory(authservice) {
  return {
    tokenGetter: () => {
      return authservice.tokenGetter();
    },
    whitelistedDomains: [environment.apiURL],
  };
}

@NgModule({
  declarations: [AppComponent],
  imports: [
    ...
    JwtModule.forRoot({
      jwtOptionsProvider: {
        provide: JWT_OPTIONS,
        useFactory: jwtOptionsFactory,
        deps: [AuthService],
      },
    }),
    ...
```

Таким чином, отримується контроль над токеном та можливість прочитувати дані, які зашифровані в ньому. Останнє, що варто додати до AppModule є AppHttpInterceptor, який буде перевіряти всі запити HTTP від клієнта та у разі, якщо додаток має токен авторизації, автоматично додавати його до запиту, щоб не робити це щоразу. Цей клас реалізовує інтерфейс HttpInterceptor, що має один метод intercept, який у якості аргуметів приймає HTTP запит та обробник.

Тепер можна перейти до реалізації компонентів. Як було зазначено у розділі 3.2, для реєстрації користувача необхідні такі дані, як ім'я, прізвище, адреса електронної пошти, номер телефону та пароль. У цілях безпеки, на форму реєстрації також буде додане поле повторення паролю. Щоб створити необхідний компонент, необхідно три файли: розширення .ts для коду і логіки, розширення .html для створення зовнішнього вигляду та .scss для стиізації контенту. У .ts файлі необхідно створити клас та задати йому декоратор @Component, де потрібно вказати шляхи до файлів розмітки та стилів, а також задати HTML селектор для вмісту цього компоненту.

```
@Component({
  selector: 'app-register',
  templateUrl: './register.component.html',
  styleUrls: ['./register.component.scss'],
  changeDetection: ChangeDetectionStrategy.OnPush,
})
export class RegisterComponent implements OnInit, OnDestroy {
```

Рис. 3.4.1 – Приклад заповнення декоратору @Component

Інтерфейси OnInit та OnDestroy, які реалізовує компонент, зобов'язують його реалізувати два методи, які викликаються протягом життєвого циклу компоненту, після ініціалізації всіх властивостей, пов'язаних з даними [19] та під час знищення об'єкту в пам'яті [20] відповідно.

Для створення форми реєстрації необхідно оголосити властивість типу FormGroup та ініціалізувати її за допомогою FormBuilder, з пакету @angular/forms. Під час ініціалізації необхідно задати всі валідатори полів, як обов'язковість,

вимоги до довжини або зобов’язаність слідувати якомусь шаблону. Також, сюди можна додати перевірки на унікальність з серверу.

```
this.registerForm = fb.group({
  firstName: ['', Validators.required],
  lastName: ['', Validators.required],
  email: [
    '',
    [Validators.required, Validators.email],
    this.uniqueValidator((value) =>
      this.authService.validateEmail(null, value)
    ),
  ],
  phoneNumber: [
    '+380',
    [Validators.required, Validators.pattern(/\+380\d{9}/)],
    this.uniqueValidator((value) =>
      this.authService.validatePhoneNumber(null, value)
    ),
  ],
  password: ['', [Validators.required, Validators.minLength(8)]],
  repeatPassword: ['', Validators.required],
});
```

Рис. 3.4.2 – Ініціалізація форми реєстрації

Після цього можна перейти до створення елементів UI. У файлі формату .html необхідно додати тег form із властивістю [formGroup]=”registerForm”. Це зв’язує форму представлення з її програмним відображенням у компоненті. За допомогою пакету @angular/material легко та швидко можна створити приємний та функціональний дизайн полів.

Після додавання всіх необхідних даних форми потрібно додати кнопки. Перша кнопка слугуватиме для відправки форми на сервер та подальшої реєстрації, друга – для переходу на сторінку входу і третя, для входу в систему за допомогою облікового запису Google. Для цього необхідно зареєструватися у Google Console, додати додаток та створити OAuth 2.0 Client ID, який дозволить безпечно передавати дані від Google до цього додатку. Після натискання на цю кнопку та отримання дозволу на вхід, клієнт повинен відправити запит на сервер, у спеціально створений метод GoogleAuthenticate, який отримає всі необхідні дані та створить обліковий запис або увійде до нього за нявності. Підтвердження адреси

електронної пошти при реєстрації за допомогою Google по зрозумілим причинам непотрібне.

Після розміщення всіх необхідних елементів на веб-сторінці та проведення необхідної стилізації, можна запустити програму.

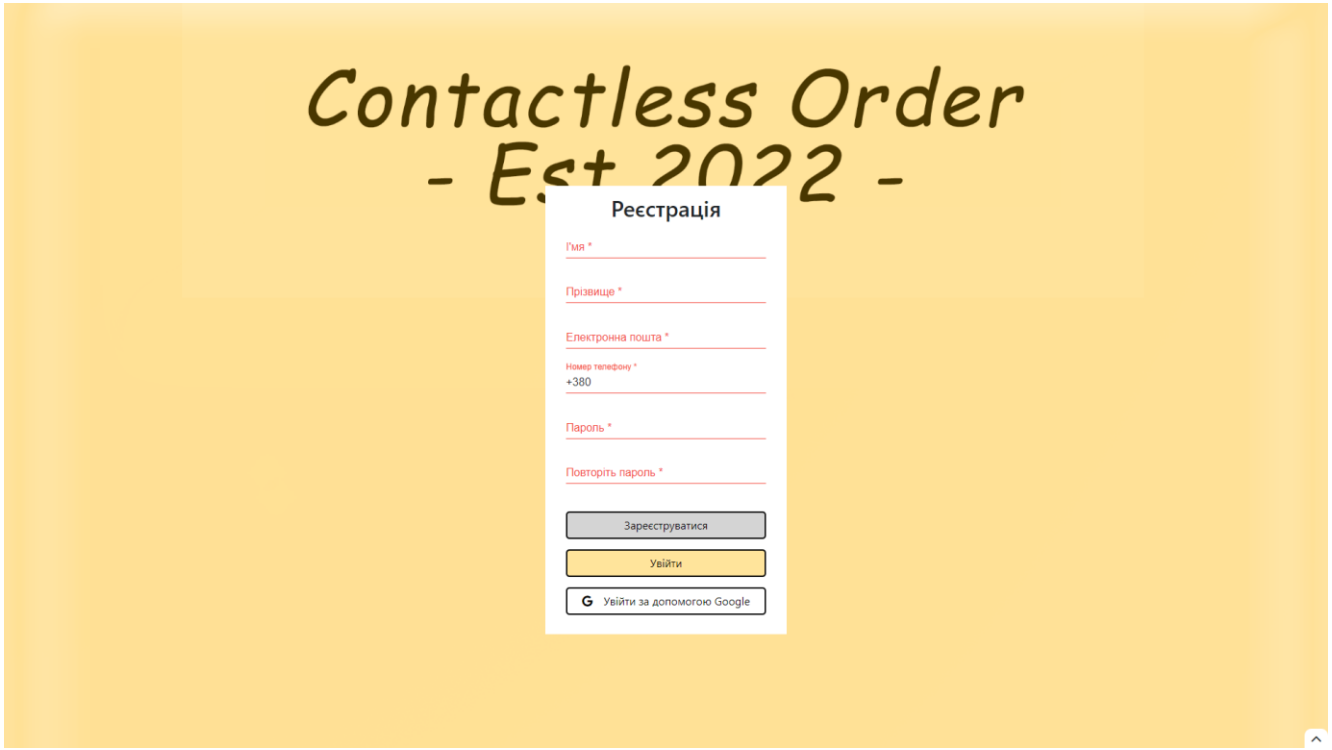


Рис. 3.4.3 – Форма реєстрації

Також, у правому нижньому куті знаходиться кнопка, яка при натисканні відкриває меню з додатковою функцією – реєстрація компанії, яка має такий же вигляд, але поля «Ім’я» та «Прізвище» замінені на «Назва компанії».

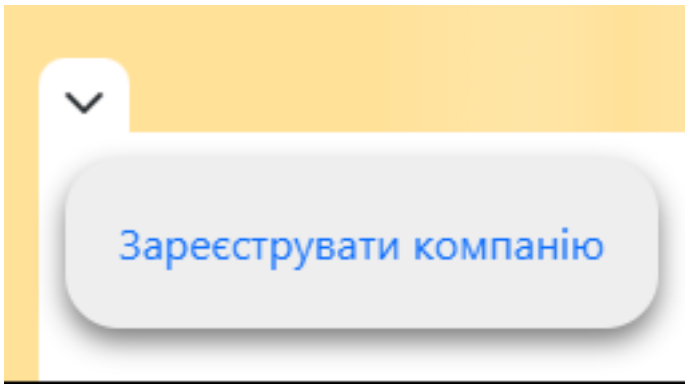


Рис. 3.4.4 – Кнопка для переходу на форму реєстрації компанії

При спробі зарєстувати існуючу адресу електронної пошти або номер телефону, форма підказує користувачу в чому проблема.

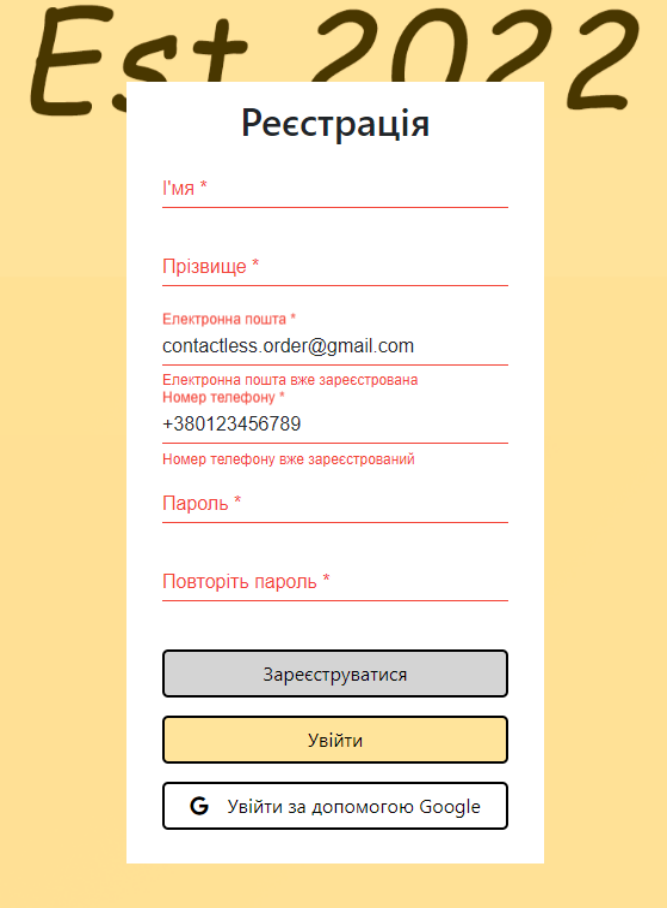


Рис. 3.4.5 – Перевірка унікальності email та номеру телефону

За таким же принципом розробляється форма входу. Виключенням з модулю login є компонент для підтвердження адреси електронної пошти, адже він не має форми та слугує лише для відправки запиту на сервер, а після отримання відповіді, перенаправляє користувача на головну сторінку системи.

3.5 Розробка функціоналу для користувача типу «Компанія»

Після реєстрації користувача Компанія та підтвердження адреси електронної пошти, час розпочати розробку усіх функцій, пов’язаних з компаніями. Для початку необхідно створити головну сторінку, де користувачі зможуть переглядати

та змінювати інформацію, в тому числі платіжну. Також, оскільки платіжну інформацію краще не виводити на головну сторінку, потрібно зробити кнопку, за допомогою якої можна виводити діалогове вікно для введення і виведення інформації. Оскільки платіжна інформація є обов'язкова для можливості компанії працювати в системі, необхідно додати нагадування, у разі якщо вона відсутня.

У другій половині сторінки необхідно створити повний функціонал для відображення та управління закладами громадського харчування. Для цього буде використавно AgGrid - це повнофункціональна таблиця даних JavaScript з можливістю налаштування. Він забезпечує чудову продуктивність, не має залежностей від сторонніх розробників і плавно інтегрується з усіма основними фреймворками JavaScript, в тому числі Angular.

Тут будуь розміщені три таблиці: з закладами, з позиціями меню та з добавками до них. Оскільки предметів меню та їх модифікацій немає у базі даних, необхідно на серверній частині створити відповідні моделі, додати їх конфігурації та створити міграцію. Після цього додати репозиторій, сервіс та контролер для роботи з даними компаній.

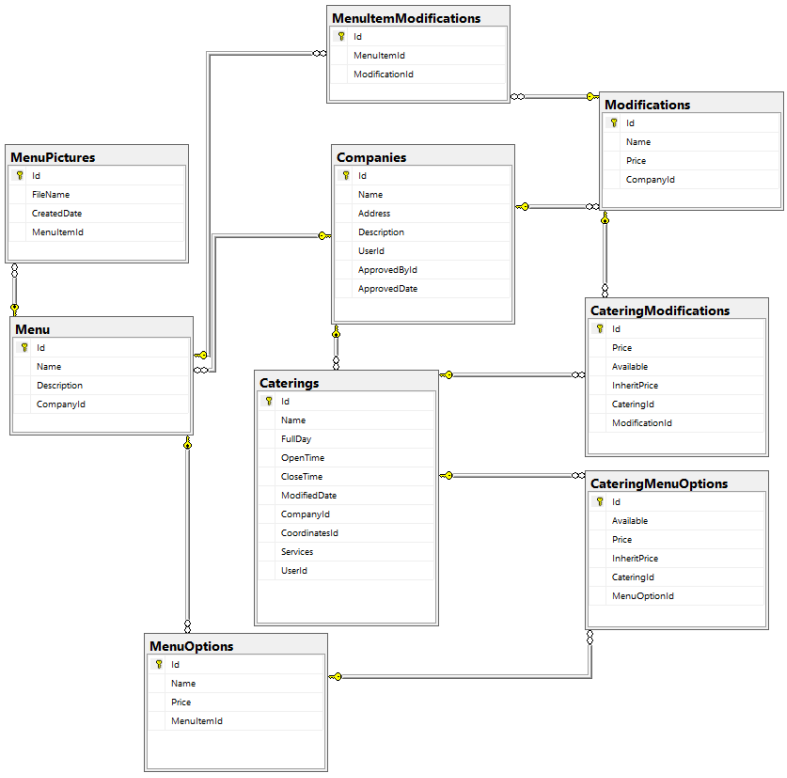


Рис. 3.5.1 – Схема БД для меню їх модифікацій та добавок

Добавка має два поля – назва та ціна, наприклад, «Кокосовий сироп, 10 грн.». Вони можуть бути використані для створення меню, наприклад, кава американо зможе мати доступну модифікацію «кокосовий сироп». Крім назви та модифікацій елементи меню мають такі поля як опис та доступні порції. Порції в свою чергу мають назву та ціну, наприклад, «S, 20грн». Також, до позиції меню можна додавати картинки, щоб користувач міг бачити який вигляд має меню.

Після цього можна створити заклад громадського хрчування. Для його створення компанія повинна вказати назву закладу, опис його послуг, вказати робочі години, обрати меню, яке подається у закладі та вказати його місцезнаходження на мапі. Для зберігання позиції закладу, до бази даних буде збережена географічна довгота та ширина, яку клієнт буде отримувати разом зі списком закладів.

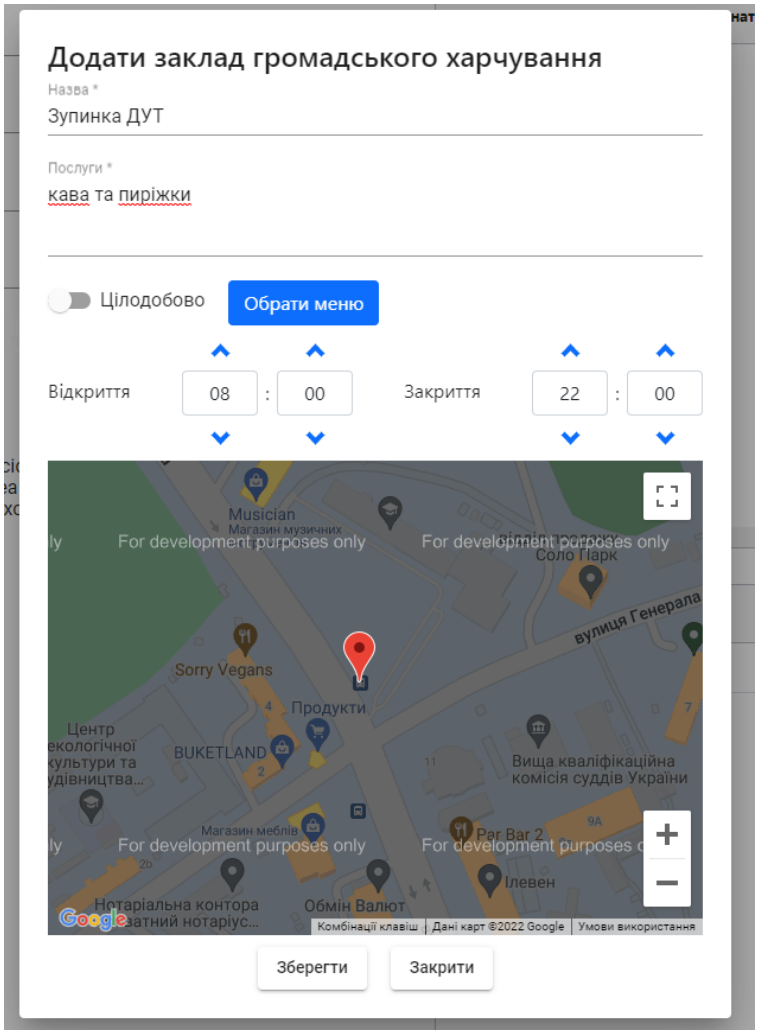


Рис. 3.5.2 – Додавання закладу громадського харучування

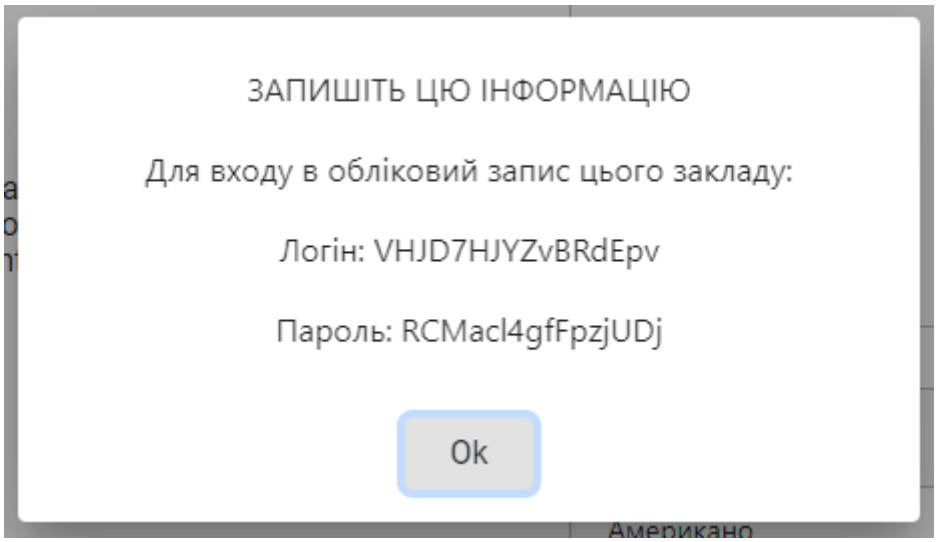


Рис. 3.5.3 – Результат створення закладу

Для відображення мапи необхідно підключити Google Maps, а для цього потрібно у Google Console створити додаток та додати до нього дозвіл на використання Maps API, яке працює лише за протоколом HTTPS, тому окрім SSL сертифікату для серверу, необхідно встановити такий ще і на клієнт. Зробити це можна просто за допомогою програми OpenSSL, яка згенерує ключ та сертифікат, який потрібно помістити до папки проекту та задати необхідні параметри у angular.json.

Рис. 3.5.4 – Додавання SSL сертифікату до клієнту

Також, для виведення довідкової інформації до таблиці з даними було розроблено спеціальний компонент, що являє собою значок з літерою «і», при наведенні на який з’являється контейнер з текстом, що містить короткі підказки по використанню таблиці.

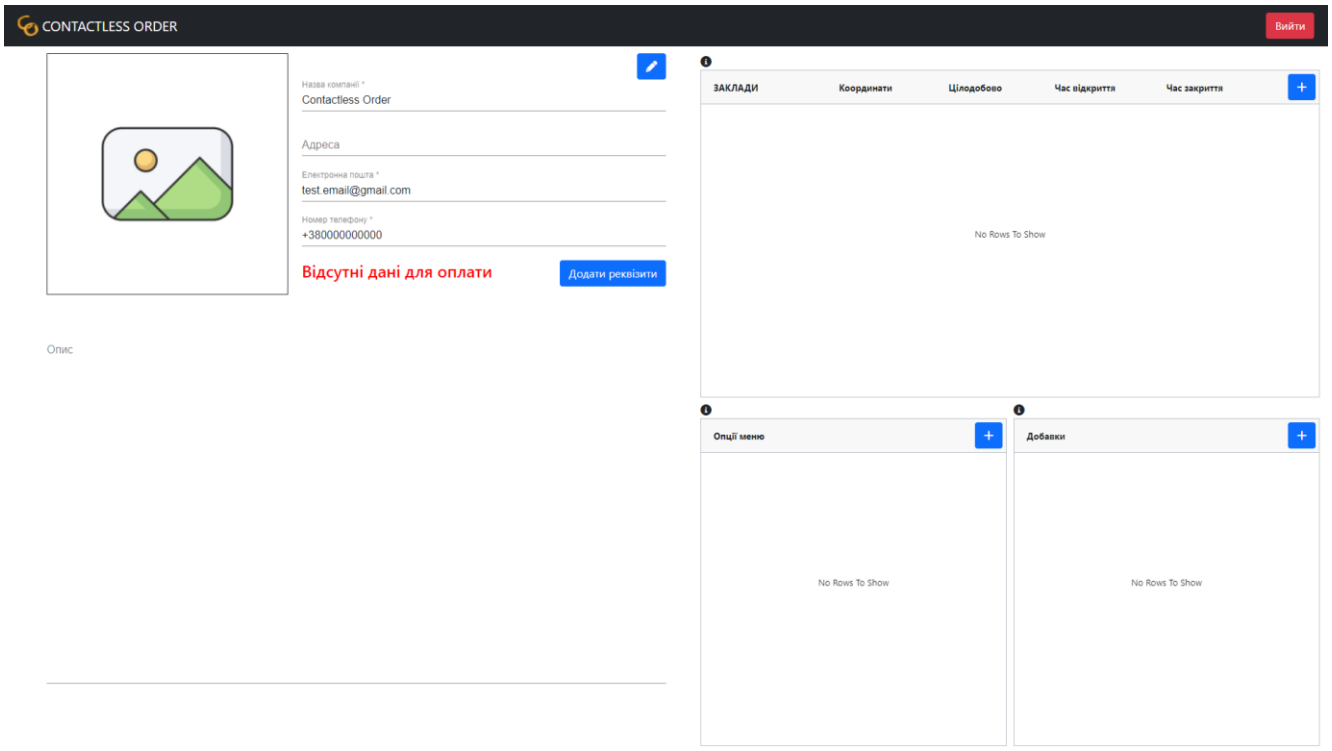


Рис. 3.5.5 – Сторінка управління компанією

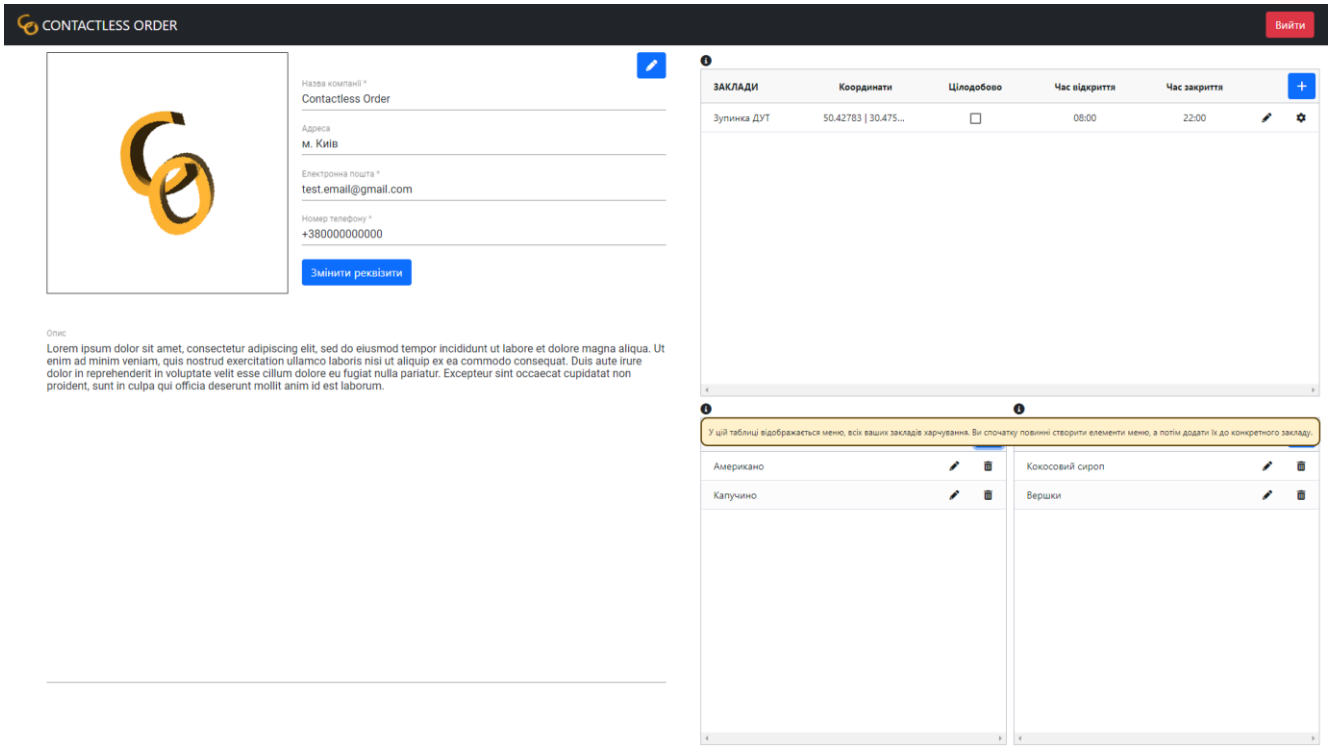


Рис. 3.5.6 – Заповнена сторінка компанії

3.6 Розробка функціоналу для користувача типу «Заклад громадського харчування»

Усі необхідні таблиці у базі даних для управління даними закладів громадського харчування було створено у попередньому підпункті, тому з точки зору серверної частини додатку необхідно створити таблиці для обробки замовлень, репозиторій, сервіс, контролер та створити між їх інтерфейсами та реалізаціями з'язки для Dependency Injection. Крім цього, в розробці знадобиться надсилати сповіщення, тому таким за таким же принципом необхідно створити NotificationService для інкапсуляції функціоналу їх надсилання.

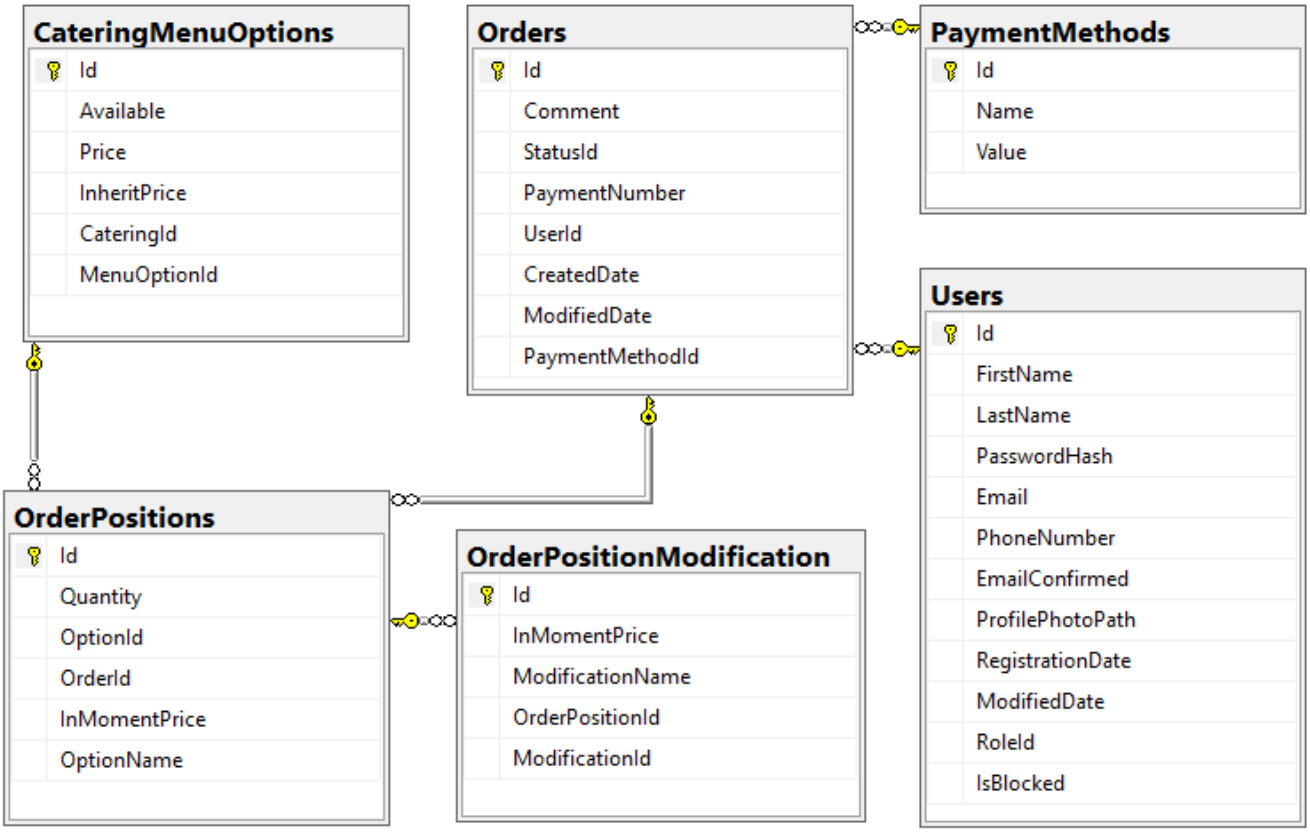


Рис.3.6.1 – Схема бази даних замовлень

У сервісі необхідно реалізувати методи отримання меню та модифікацій, зміни можливості їх замовлення та методи обробки замовлень.

UI закладів складатиметься з трьох таблиць, де буде відображене меню та добавки, з можливістю зміни, і замовлення, з можливістю зміни статусу та перегляду детальної інформації.

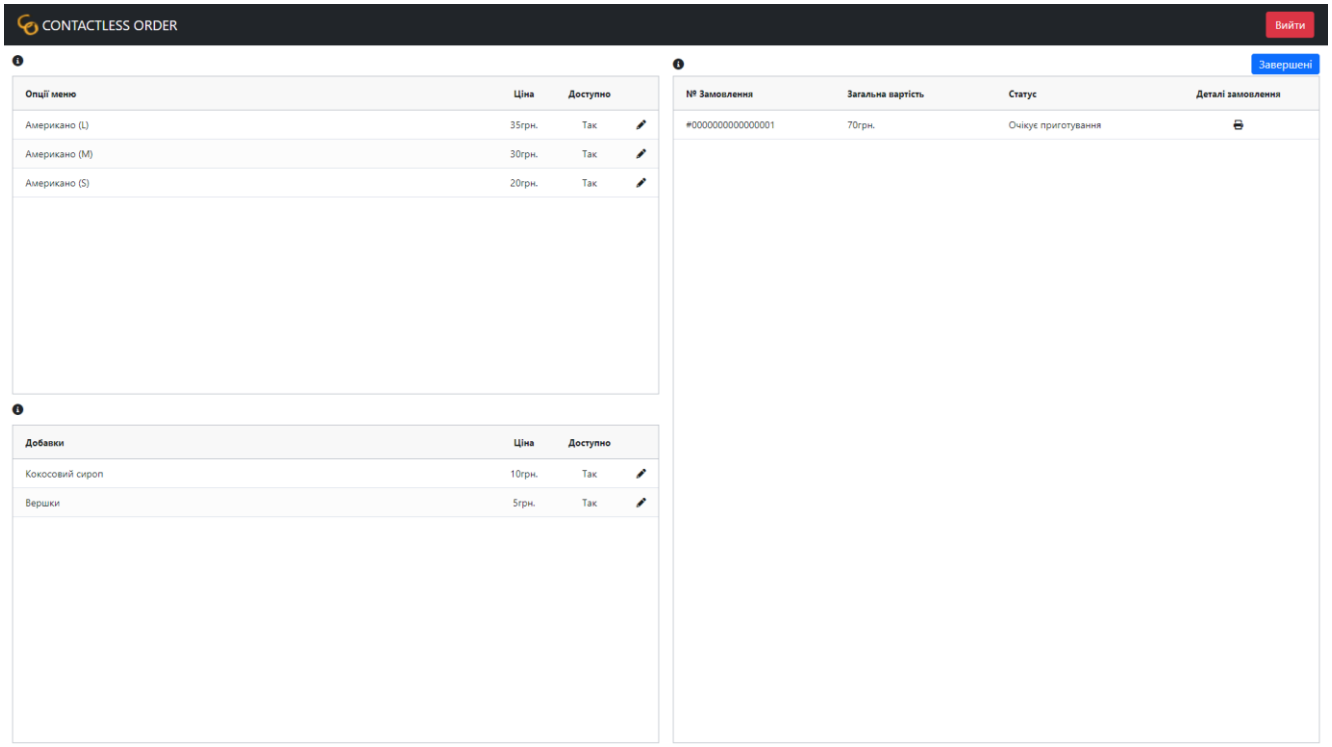


Рис. 3.6.2 – Сторінка закладу громадського харчування

Дані для таблиць зліва отримуються із заданого компанією меню та добавок. Права таблиця слугує для опрацювання замовлень. За замовчуванням, вона відображає ті замовлення, що потребують обробки. Для перегляду завершених замовлень у правому верхньому куті екрану існує кнопка «Завершені», що переводить таблицю у режим відображення відмінених або завершених замовлень. Для того, щоб заклад мав завжди актуальний список замовлень, необхідно створити додавання та видалення замовлень у режимі реального часу.

Для створення такого функціоналу буде використано бібліотеки SignalR від Microsoft. Вони дозволяють створювати події на стороні серверного додатку та підписуватися на них з клієнтського додатку. Для їх функціонування необхідно створити інтерфейс, який описуватиме сигнатури подій, та клас-вузол, що наслідується від Hub<T>, де T – це описаний інтерфейс. Після цього, в класі Startup

в методі Configure необхідно описати їх конфігурацію за допомогою методу UseEndpoints аргуметну app.

```
app.UseEndpoints(c =>
{
    c.MapControllers();
    c.MapHub<OrdersHub>("/orders", opt => opt.Transports = HttpTransportType.WebSockets);
});
```

Рис. 3.6.3 – Конфігурування SignalR

Після цього можна створити спеціальний сервіс у клієнтській частині додатку та підписатися на описані події. Для зручності створення наступних сервісів для аднімінстраторів та служби підтримки, спочатку необхідно створити базовий клас з методами підключення та відключення до серверу, а також абстрактним методом пісписок на події. Коли такий сервіс створено, можна перейти до побудови конртентого сервісу для замовлень.

```
@Injectable()
export class CateringNotificationService extends SignalrService {
  private orderUpdated = new Subject<any>();
  public onOrderUpdated = (): Observable<any> => this.orderUpdated;

  private orderPaid = new Subject<any>();
  public onOrderPaid = (): Observable<any> => this.orderPaid;

  private orderRejected = new Subject<any>();
  public onOrderRejected = (): Observable<any> => this.orderRejected;

  private orderCompleted = new Subject<any>();
  public onOrderCompleted = (): Observable<any> => this.orderCompleted;

  constructor(
    protected readonly _router: Router,
    private readonly _http: HttpClient
  ) {
    super('orders', _router, _http);
  }

  protected registerMethods(): void {
    this.connection.on('OrderUpdated', (order) => this.orderUpdated.next(order));
    this.connection.on('OrderPaid', (order) => this.orderPaid.next(order));
    this.connection.on('OrderRejected', (order) => this.orderRejected.next(order));
    this.connection.on('OrderCompleted', (order) => this.orderCompleted.next(order));
  }
}
```

Рис. 3.6.4 – Сервіс для роботи SignalR

У компоненті закладу громадського харчування виконується підписка на створені події та їх обробка, як додавання, оновлення або видалення запису з таблиці. Зміна статусу замовлення відбуватиметься при натисканні на клітинку статусу, після чого з’являтиметься список доступних статусів для замовлення. У разі, якщо необхідно уточнити деталі замовлення, заклад матиме змогу зателефонувати замовнику через вікно деталей замовлення. При натисканні на кнопку «Зателефонувати», номер користувача буде скопійовано до буферу обміну.

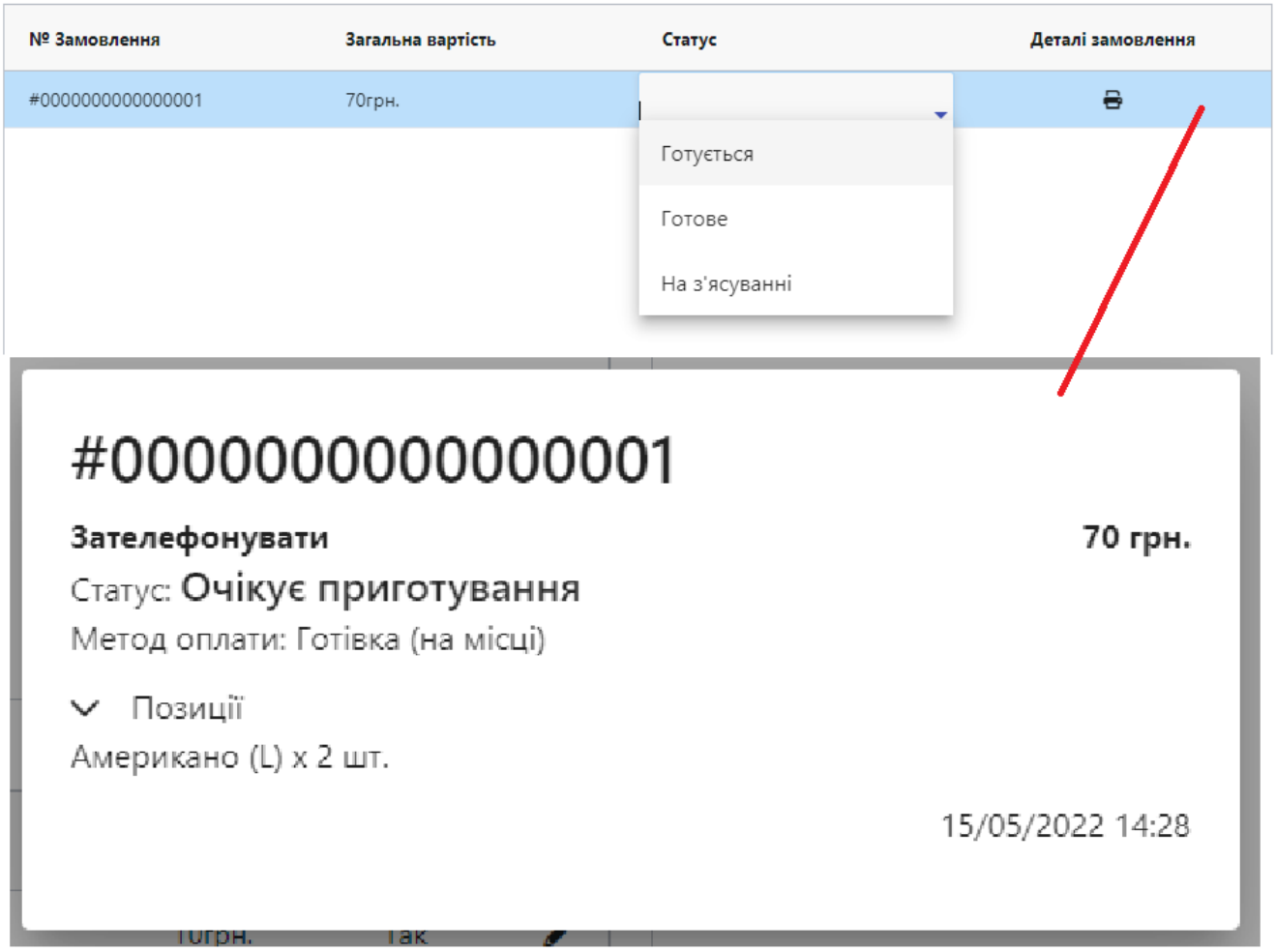


Рис. 3.6.5 – Обробка та деталі замовлення

3.7 Розробка функціоналу для користувача типу «Клієнт»

Відповідно до вимог до функціоналу, зазначених у пункті 2.2 необхідно розробити кілька сторінок для користувача типу «Клієнт»:

1. Головна сторінка, для пошуку і відображення закладів громадського харчування;
2. Корзина, для перегляду обраних товарів та створення замовлень;
3. Замовлення, для перегляду активних замовлень у режимі реального часу, а також історії замовлень.

Оскільки клієнти використовуватимуть додаток найчастіше з мобільного телефону, необхідно розробити адаптивний дизайн, який буде однаково зручним як для повної версії сайту, так і для мобільної.

Пошук закладів громадського харчування на карті проходитиме за таким принципом: отримати всі відкриті заклади, які по своєму розміщенню знаходяться на обраній частині карти, та у разі, якщо рядок пошуку не пустий, фільтрувати отримані заклади.

Після натискання на обраний заклад громадського харчування, позначеного маркером на карті, відбуватиметься його перегляд. У перегляді зазначається назва і опис закладу, години його роботи та меню. Додаткову інформацію, таку як опис, можна сховати, натиснувши на напис «Про заклад».

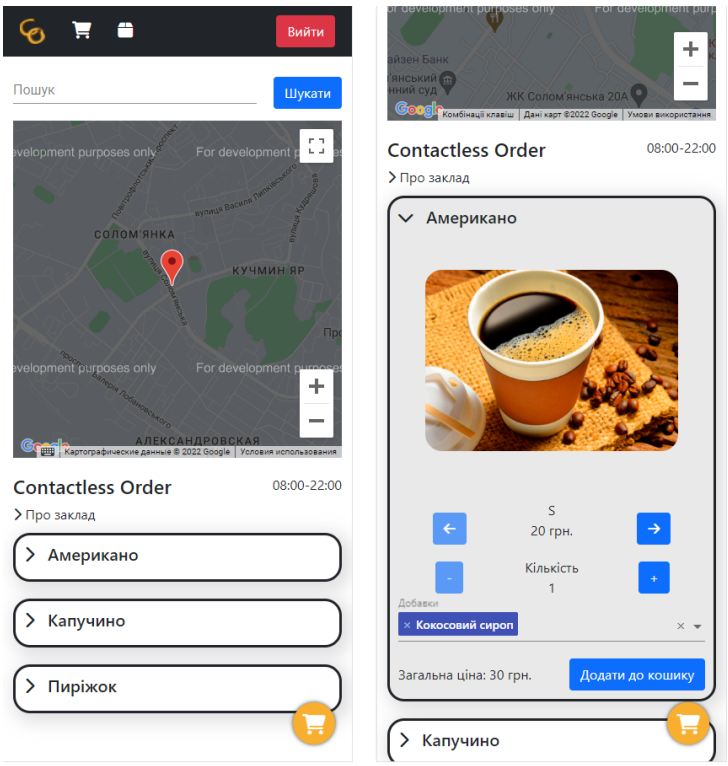


Рис. 3.7.1 – Перегляд закладу громадського харчування

У разі, якщо обрана порція є недоступно, користувачу буде показано відповідний напис, якщо недоступна добавка, то вона буде недоступною для вибору. Після додавання бажаних товарів у кошик, при натисканні на відповідну кнопку в нижньому правому куті екрану або на панелі зверху, клієнт має змогу перейти на сторінку кошику, звідки має змогу зробити замовлення, попередньо внесши, при необхідності, зміни.

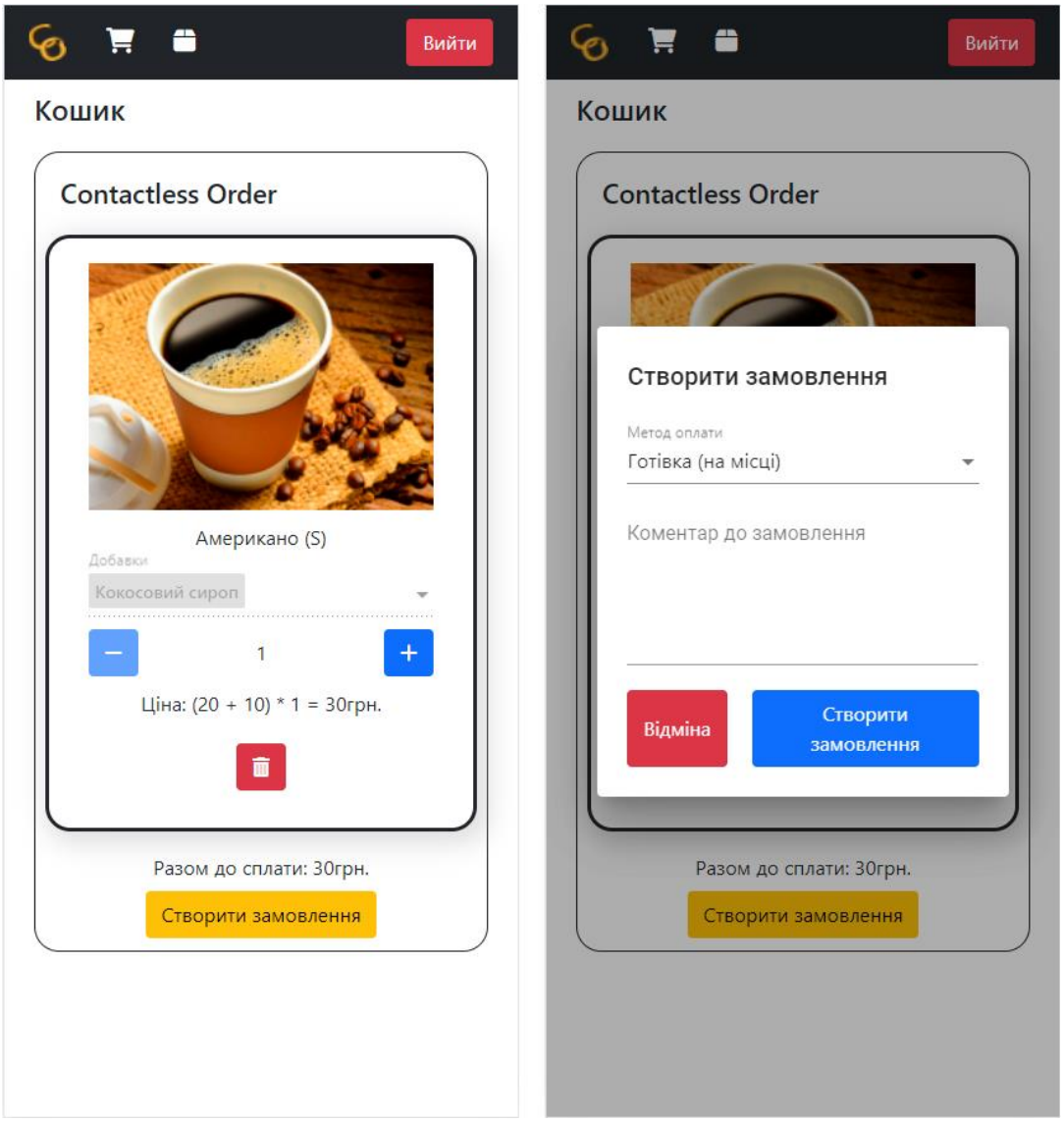


Рис. 3.7.2 – Кошик та створення замовлення

Після створення замовлення, користувач перейде до сторінки замовлень і у разі, якщо було обрано оплату карткою, клієнту буде відкрито вікно для оплати. Також, тут знаходитиметься кнопка для повернення до оплати, якщо користувач

відмінив оплату вперше. При оплаті готівкою, заклад отримає сповіщення про нове замовлення одразу, а у разі якщо картою – лише після оплати клієнтом.

Також, на кожному замовленні є можливість переглянути місцезнаходження закладу, у якого воно було здійснене, переглянути замовлені позиції, відмінити замовлення, якщо заклад ще не почав його готувати та можливість поскаржитися на замовлення.

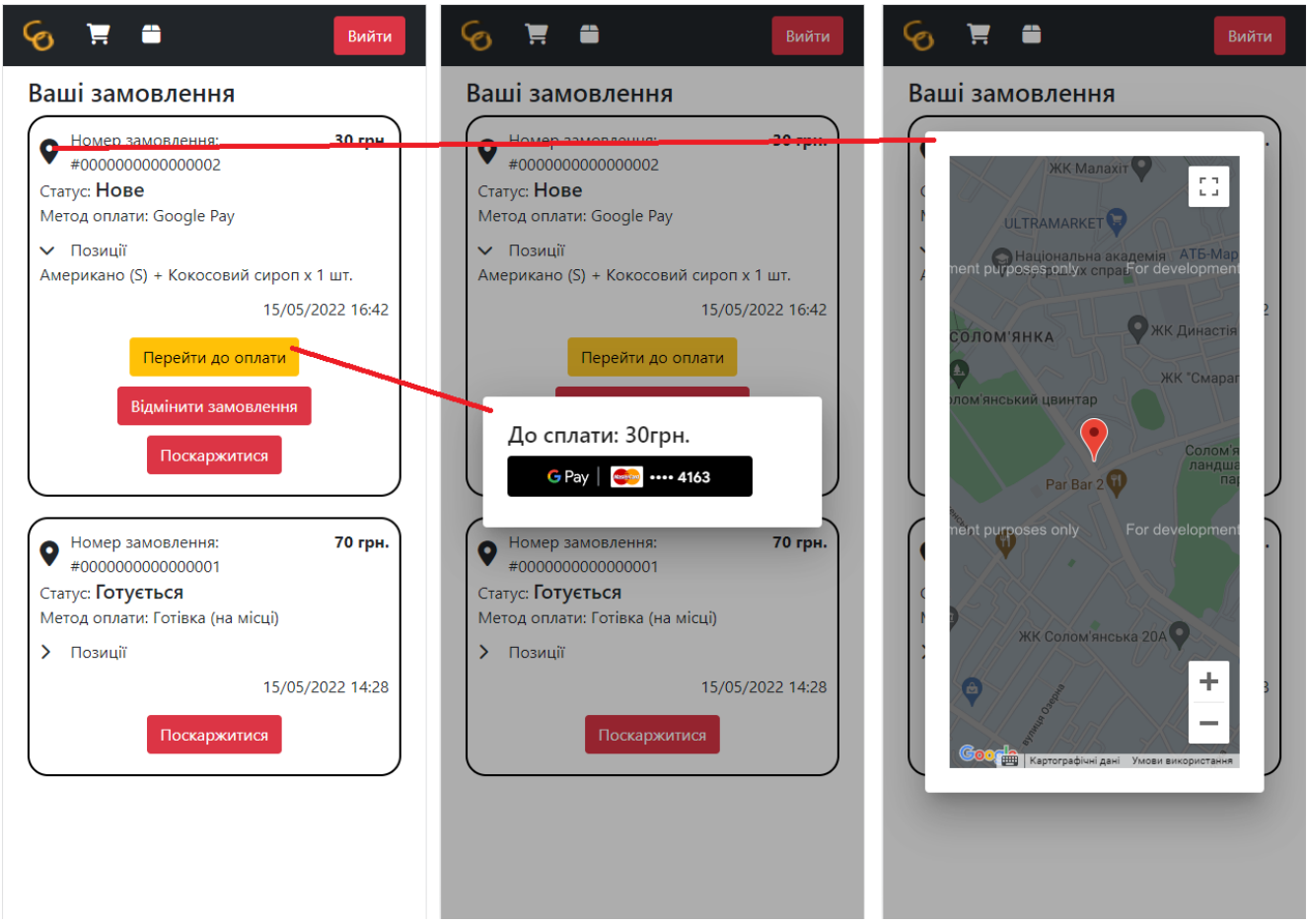


Рис. 3.7.3 – Сторінка замовлень

Для сторінки замовлень також необхідно розробити функціонал для оновлення статусу в режимі реального часу. Він розробляється за допомогою нового сервісу SignalR, за прикладом такого для закладів громадського харчування. Після того, як заклад позначить замовлення як готове, користувач отримає сповіщення зі звуковим супроводженням. Після цього користувач має підтвердити отримання, якщо була обрана оплата карткою, якщо ж була обрана оплата готівкою,

оплату повинен підтвердити заклад, адже у разі оплати картою, компанія отримує кошти лише після підтвердження клієнта.

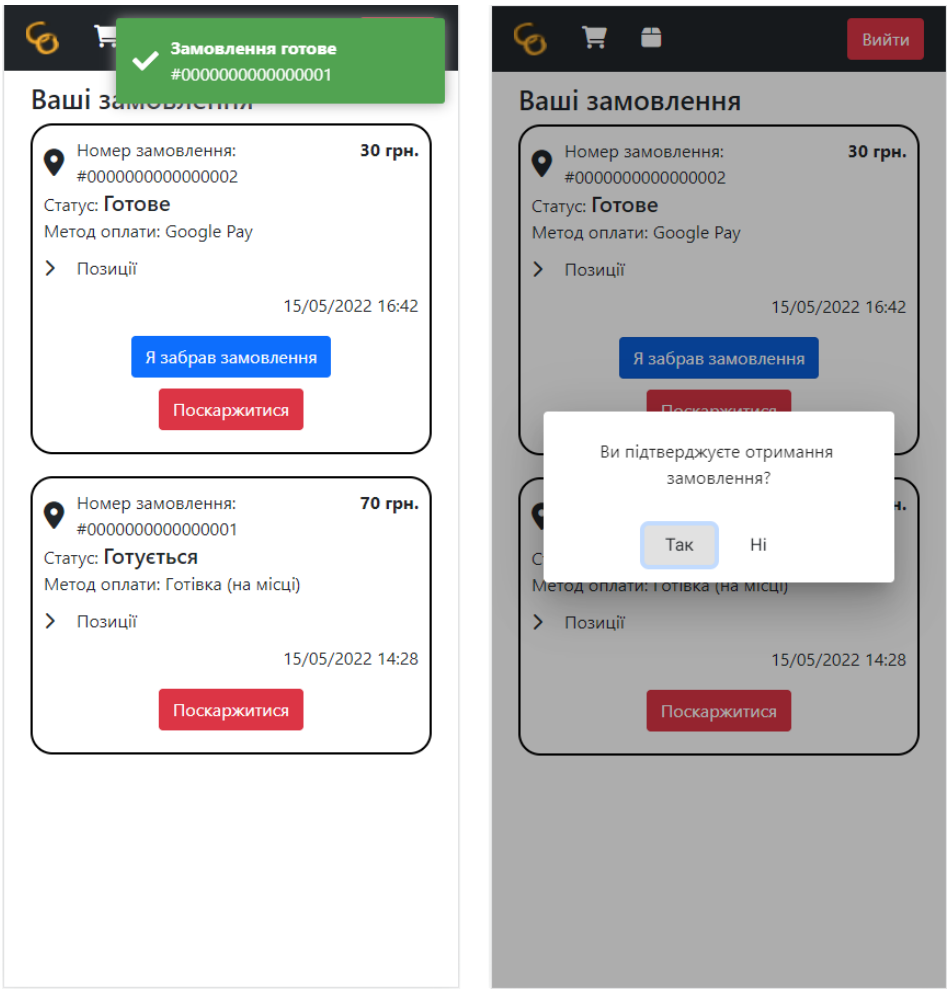


Рис. 3.7.4 – Завершення замовлення

3.8 Розробка функціоналу для користувача типу «Адміністратор»

Оскільки адміністратор повинен адмініструвати всіх інших користувачів, однієї сторінки для всього функціоналу не вистачить. Тому необхідно створити три компонента для керування компаніями, користувачами та робочими обліковими записами (адміністратор та служба підтримки). Для перемикання між ними до панелі вгорі сторінки буде додано перемикач з вивобором бажаної сторінки. Цей перемикач зберігатиме останню обрану опцію у пам’ять браузеру, а під час завантаження сторінки завантажувати збережені дані.

Сторінка керування компаніями являтиме собою таблицю, в якій будуть показані не схвалені компанії. Справа вгорі міститиметься перемикач, за допомогою якого можна перевести сторінку в режим відображення схвалених компаній. Кожен стовпець таблиці матиме можливість фільтрації вмісту за пошуковим запитом та іншими налаштуваннями, що входять у набір фільтрів AgGrid.

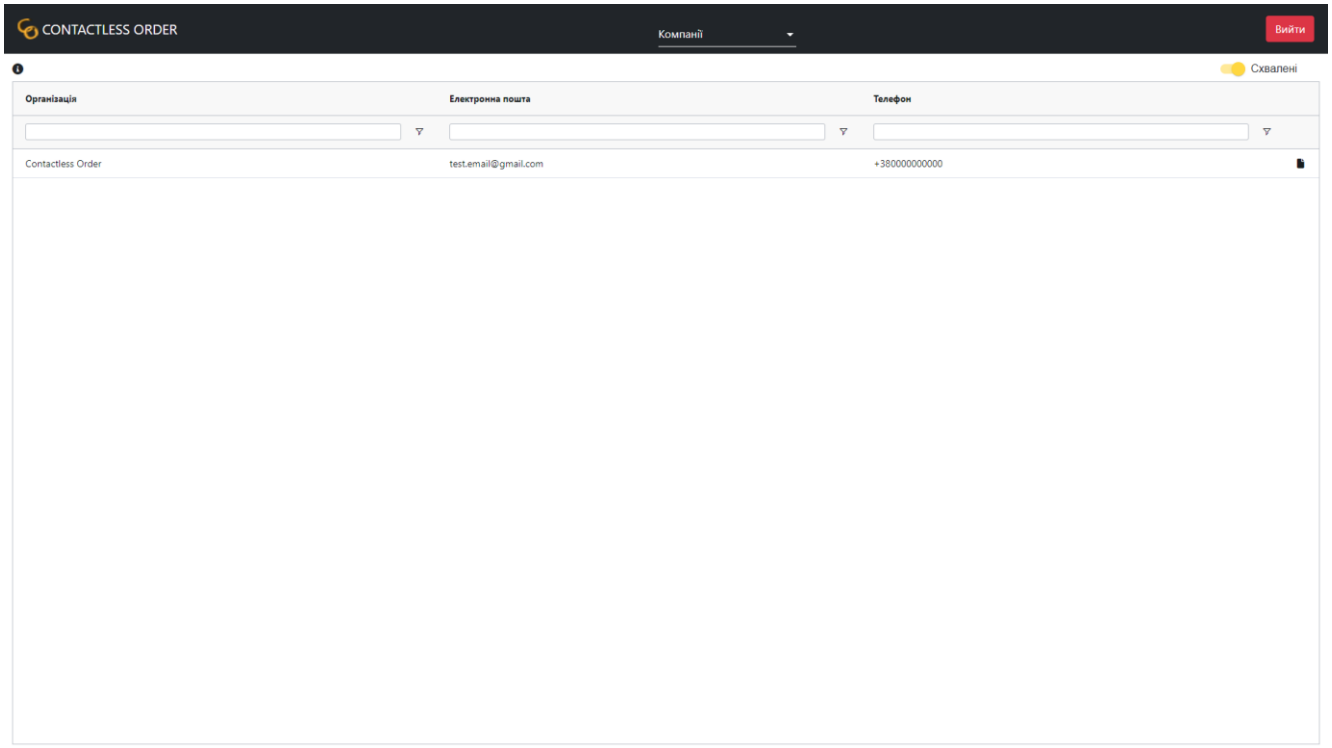


Рис. 3.8.1 – Сторінка управління компанією

При натисканні на клітинку з номером телефону або адресою електронної пошти, відповідне значення буде скопійовано до буферу обміну. При натисканні на значок документу у правому кінці кожного запису можна відкрити детальну інформацію про компанію.

У вікні детальної інформації можна перевірити дані, які не входять до складу таблиці, а саме адресу, дату реєстрації та опис. Крім цього при натисканні на напис «Заклади» з’являється список усіх закладів громадського харчування компанії, місцезнаходження яких можна переглянути на карті по кліку миші. Внизу знаходяться кнопки для закриття вікна, для перевірки реквізитів для оплати коштів

компанії за замовлення, для схвалення компанії, якщо вона не схвалена, та для відзову схвалення, якщо вона була схвалена раніше.

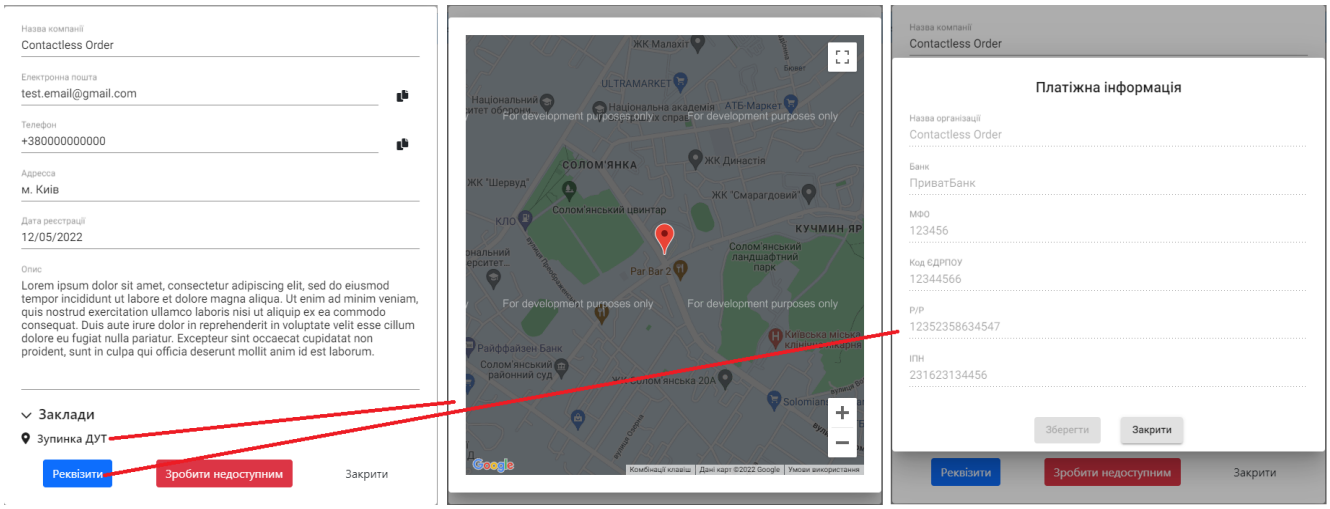


Рис. 3.8.2 – Детальна інформація про компанію

Після схвалення компанії адміністратором, її заклади громадського харчування почнуть відображатися клієнтам на карті при пошуку.

На сторінці управління користувачами також відображена таблиця з користувачами з можливістю фільтрації результатів за ключовими словами. Адміністратор може блокувати та розблоковувати клієнтів додатку у разі потреби, наприклад, якщо на користувача надходили скарги з боку інших.

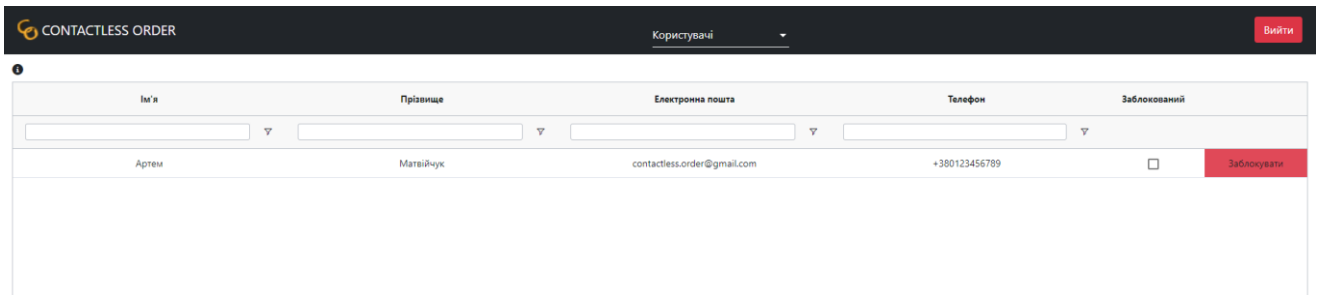


Рис. 3.8.3 – Сторінка управління клієнтами

Сторінка управління командою додатку міститься дві таблиці – з адміністраторами та користувачами служби підтримки. Оскільки реєстрація таких облікових записів неможлива, то адміністратор повинен керувати ними, до того ж,

перший аккаунт з правами адміністратора було створено при першій міграції бази даних за допомогою SQL скрипту.

За домомогою кнопок зі знаком плюс користувачі цієї ролі можуть створювати нові облікові записи відповідних можливостей. Для цього необхідно лише вказати ім’я, прізвище та номер телефону нової посадової особи. Після натискання кнопки «Зберегти» на сервер буде виконано запит на створення користувача в системі. Надійний логін та пароль для входу буде згенеровано автоматично та виведено на екран. Після їх створення, адміністратор може за потреби видаляти облікові записи або генерувати новий пароль у разі втрати старого. Проте, користувач не може видалити власний аккаунт, що дозволяє забезпечити нормальну роботу додатку.

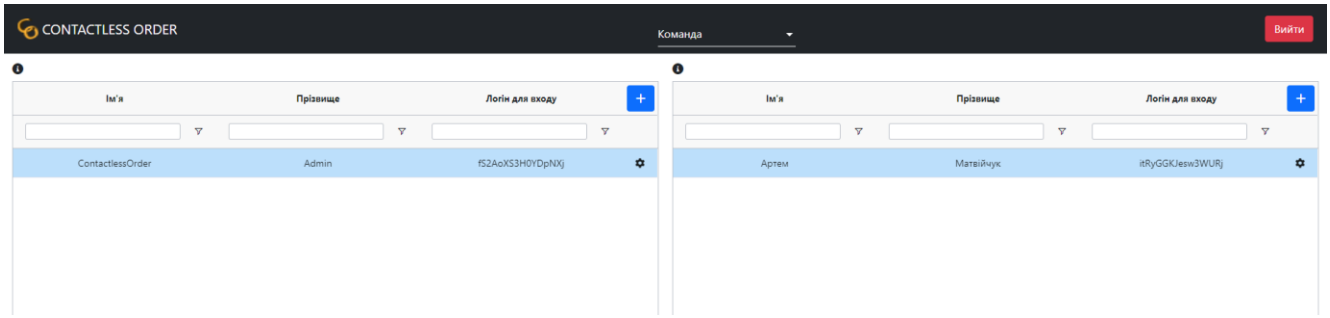


Рис. 3.8.4 – Сторінка управління командною додатку

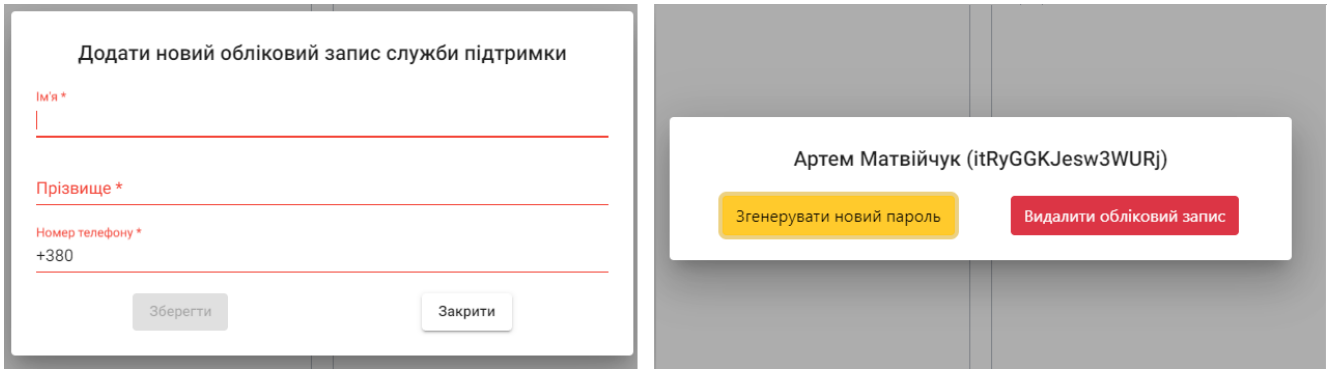


Рис. 3.8.5 – Додавання та модифікація облікових записів команди

Варто зазначити, що всі дані в цих таблицях оновлюються в режимі реального часу за допомогою нового сервісу, що працює з SignalR, для адміністраторів.

3.9 Розробка функціоналу для користувача типу «Служба підтримки»

Сторінка працівника служби підтримки міститиме таблицю з основною інформацією про скаргу та перемикачем, якого саме статусу відображати скарги в даний момент. Такий перемикач взятий з пакету @ng-module та називається NgSelect. Це зручний, простий у використанні і водночас гнучкий компонент, що дозволяє створювати випадаючі списки.

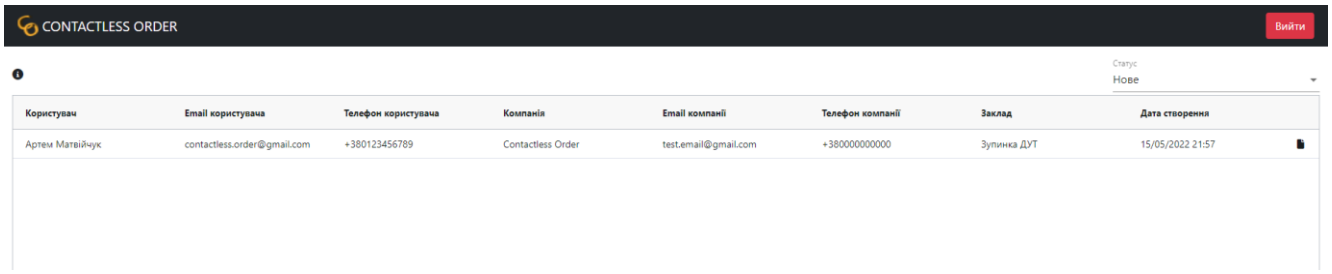


Рис. 3.9.1 – Сторінка служби підтримки

При натисканні на будь-який номер телефону або адресу електронної пошти, дані буде скопійовано до буферу обміну та з’явиться відповідний напис. Це дозволить працівникам швидше та зручніше обробляти скарги клієнтів на неякісне обслуговування.

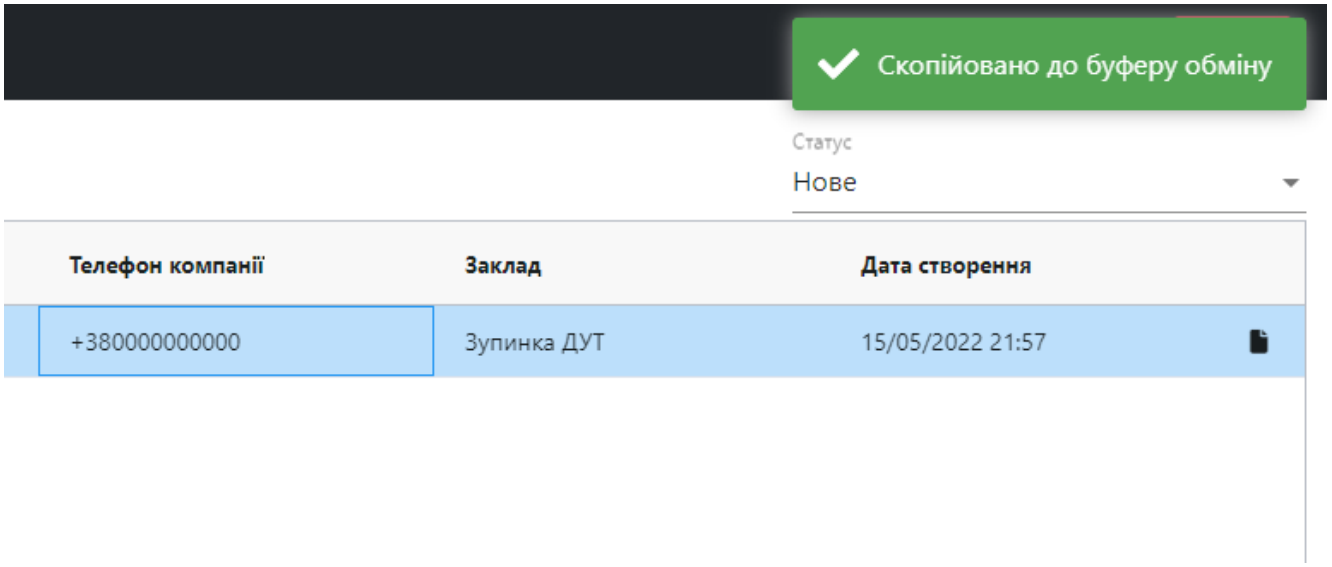


Рис. 3.9.2 – Копіювання даних

При натисканні на значок документу, працівник матиме змогу переглянути номер замовлення на коментар, залишений клієнтом. За допомогою випадаючого списку справа вгорі, працівник може змінити статус скарги.

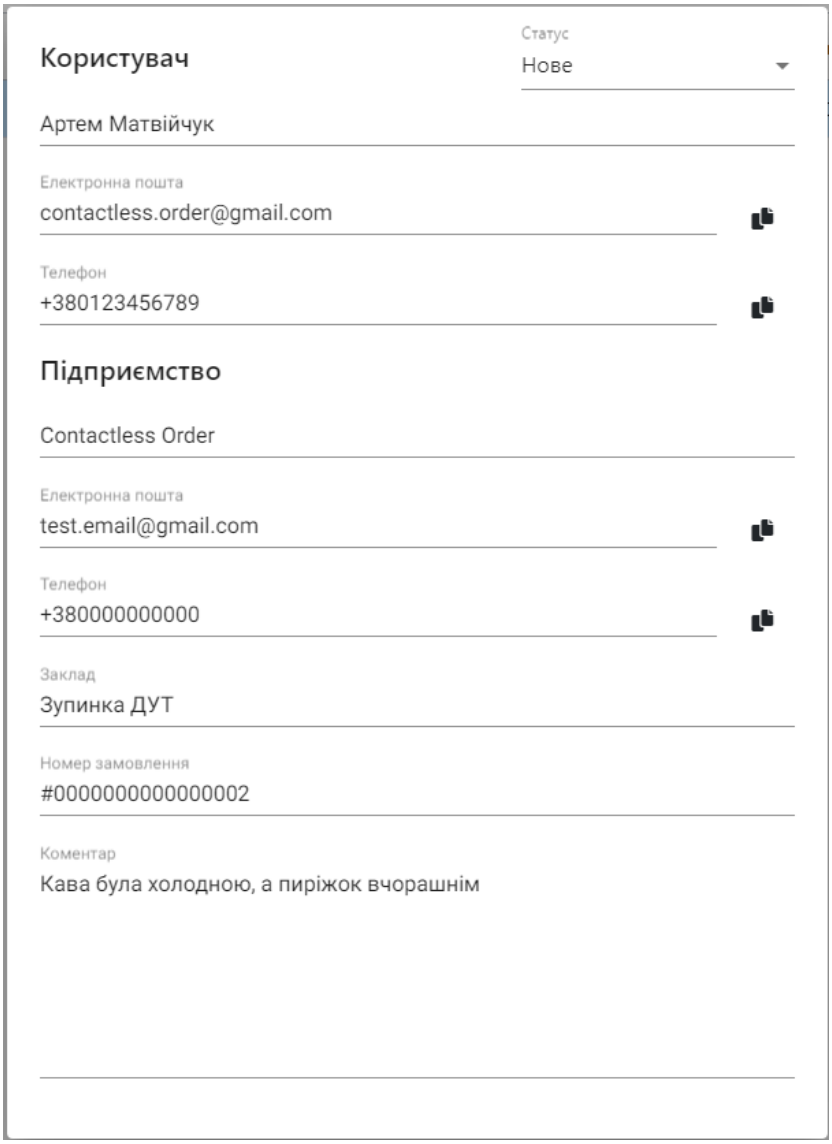


Рис. 3.9.3 – Деталі скарги

Отже, розроблена система направлена на допомогу закладам громадського харчування та їх клієнтам для швидкого прийому та оплати замовлень. Програмний продукт може підвищити продуктивність працівників закладів громадського харчування та полегшити процес замовлення для їх клієнтів.

ВИСНОВКИ

Дана робота була спрямована на проектування та реалізацію інструменту для оптимізації та автоматизації деяких бізнес процесів при обслуговуванні у закладах громадського харчування.

1. Було обґрунтовано актуальність розробленої системи та її наукову новизну шляхом аналізу існуючих засобів оптимізації процесу обслуговування в закладах громадського харчування. Було досліджено типові задачі сучасної індустрії та виявлено, що вона потребує інструментів, які можуть зменшувати час витрачений на пошук необхідного закладу та очікування замовлення, одночасно збільшуючи потенційну продуктивність працівників.
2. Було розроблено клієнт-серверну архітектуру веб-додатку, оскільки це обумовлено функціональною специфікою програми та одразу дозволяє максимальній кількості користувачів користуватися нею.

Було обрано інструменти та технології для створення системи, такі як Microsoft SQL Server, Entity Framework Core, .NET, SignalR, Angular, обґрунтовано їх практичність та існуючи переваги над аналогами.

Для розробки успішного продукту, було проаналізовано типові задачі, які виконують учасники взаємодії та їх послідовність та створено набір вимог у вигляді UML діаграм.

3. Описано програмні засоби та інструменти, котрі було застосовано для розробки програмного забезпечення. Виявлено, що для побудови клієнтської та серверної частин додатку краще використовувати різні IDE. Підібрано допоміжні інструменти розробки, такі як Postman для тестування розробленого API. Також було проведено роботу з Google API, для забезпечення деяких функцій додатку.
4. Розроблена система цілеспрямована на допомогу закладам громадського харчування та їх клієнтам для швидкого прийому та оплати замовлень. Визначено, що програмний продукт може підвищити продуктивність працівників закладів громадського харчування та полегшити процес замовлення для їх клієнтів.

Результати дослідження бакалаврської роботи апробовані на всеукраїнських науково-технічних конференціях: "Застосування програмного забезпечення в

інфокомунікаційних технологіях" та «Сучасні інтелектуальні інформаційні технології в науці та освіті».

ПЕРЕЛІК ПОСИЛАНЬ

1. Про основні принципи та вимоги до безпечності та якості харчових продуктів. Офіційний веб-портал парламенту України. [Електронний ресурс]: [Веб-сайт]. – електронні дані. – Режим доступу: https://zakon.rada.gov.ua/laws/show/771/97-вр#Text – Дата звернення: 16.05.2022.
2. Технології та інновації, які змінюють ресторанний бізнес. Міністерство з питань стратегічних галузей промисловості України. [Електронний ресурс]: [Веб-сайт]. – електронні дані. – Режим доступу: https://sfii.gov.ua/tehnologii-ta-innovacii-yaki-zminjujut-restorannij-biznes/ – Дата звернення: 16.05.2022.
3. Матвійчук А. М. Оптимізація процесу обслуговування в закладах громадського харчування. *Застосування програмного забезпечення в інфокомунікаційних технологіях*: матеріали наук.-тех. конф., м. Київ, 20 квітня 2022 р. / Державний університет телекомунікацій, кафедра інженерії програмного забезпечення. Київ, 2022
4. Ніє Ф.-П. Key success factors in catering industry management / Фанг-Пеі Ніє, Чінь-Юнь Понг // Актуальні проблеми економіки. – 2012. – № 4. – С. 423–430.
5. Statista Research Department. Number of Apple Pay, Samsung Pay and Google Pay contactless payment users in 2018, with a forecast for 2020. Statista. [Електронний ресурс]: [Веб-сайт]. – електронні дані. – Режим доступу: https://www.statista.com/statistics/722213/user-base-of-leading-digital-wallets-nfc/ – Дата звернення: 16.05.2022.
6. Oluwatosin H. S. Client-Server Model / Haroon Shakirat Oluwatosin // IOSR Journal of Computer Engineering. – 2014. – Т. 16, № 1. – С. 67–71.
7. The Editors of Encyclopaedia Britannica. Client-server architecture. Encyclopedia Britannica | Britannica. [Електронний ресурс]: [Веб-сайт]. – електронні дані. – Режим доступу: https://www.britannica.com/technology/client-server-architecture. – Дата звернення: 16.05.2022.

8. Матвійчук А. М. Оптимізація процесу обслуговування в закладах громадського харчування. *Сучасні інтелектуальні інформаційні технології в науці та освіті*: матеріали наук.-тех. конф., м. Київ, 5 квітня 2022 р. / Державний університет телекомунікацій, кафедра штучного інтелекту. Київ, 2022
9. What is Three-Tier Architecture. IBM. [Електронний ресурс]: [Веб-сайт]. – електронні дані. – Режим доступу: https://www.ibm.com/cloud/learn/three-tier-architecture – Дата звернення: 16.05.2022.
10. Hypertext Transfer Protocol -- HTTP/1.1. IETF. Datatracker. [Електронний ресурс]: [Веб-сайт]. – електронні дані. – Режим доступу: https://datatracker.ietf.org/doc/html/rfc2616 – Дата звернення: 16.05.2022.
11. Fielding Roy. Architectural Styles and the Design of Network-based Software Architectures [Електронний ресурс] / Fielding Roy - 2000 – Режим доступу: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm – Дата звернення: 16.05.2022.
12. Introduction to Angular concepts. Офіційний сайт Angular. [Електронний ресурс]: [Веб-сайт]. – електронні дані. – Режим доступу: https://angular.io/guide/architecture – Дата звернення: 16.05.2022.
13. What is Inversion of Control?. Educative: Interactive Courses for Software Developers. [Електронний ресурс]: [Веб-сайт]. – електронні дані. – Режим доступу: https://www.educative.io/edpresso/what-is-inversion-of-control – Дата звернення: 16.05.2022.
14. IEntityTypeConfiguration. Microsoft Docs. [Електронний ресурс]: [Веб-сайт]. – електронні дані. – Режим доступу: https://docs.microsoft.com/ru-ru/dotnet/api/microsoft.entityframeworkcore.ientitytypeconfiguration-1?view=efcore-6.0 – Дата звернення: 16.05.2022.
15. Getting Started Guide. Офіційний сайт AutoMapper. [Електронний ресурс]: [Веб-сайт]. – електронні дані. – Режим доступу:

https://docs.automapper.org/en/latest/Getting-started.html – Дата звернення: 16.05.2022.

16. Shingala, K.. Json web token (jwt) based client authentication in message queuing telemetry transport (MQTT). arXiv preprint arXiv:1903.02895, 2018.
17. Naylor, David, et al. The cost of the "s" in https. *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. 2014. с. 133-140
18. Про Node.js. Офіційний сайт Node.js. [Електронний ресурс]: [Веб-сайт]. – електронні дані. – Режим доступу: https://nodejs.org/uk/about/ – Дата звернення: 16.05.2022.
19. OnInit. Офіційний сайт Angular. [Електронний ресурс]: [Веб-сайт]. – електронні дані. – Режим доступу: https://angular.io/api/core/OnInit – Дата звернення: 16.05.2022.
20. OnDestroy. Офіційний сайт Angular. [Електронний ресурс]: [Веб-сайт]. – електронні дані. – Режим доступу: https://angular.io/api/core/OnDestroy – Дата звернення: 16.05.2022.

Додаток А

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
Кафедра Інженерії програмного забезпечення

РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ОПТИМІЗАЦІЇ ПРОЦЕСУ ОБСЛУГОВУВАННЯ В ЗАКЛАДАХ ГРОМАДСЬКОГО ХАРЧУВАННЯ

Виконавець: студент 4 курсу,
групи ПД–41
Матвійчук Артем Миколайович
Керівник роботи:
Жебка Вікторія Вікторівна

Київ 2022

Мета, об’єкт та предмет роботи

- ***Мета роботи*** – автоматизація процесу замовлення у закладах громадського харчування, за допомогою розробленого додатку.
- ***Об'єкт дослідження*** – процес замовлення у закладах громадського харчування.
- ***Предмет дослідження*** – веб-додаток для автоматизації процесу замовлення.

Аналіз аналогів

Переваги

- Можливість доставки.

Недоліки

- Неможливість робити замовлення для прийому їжі на місці.
- Відсутність інтеграції з малим бізнесом.
- Відсутність інтерактивної карти для замовлень.

Технічне завдання

- Клієнт-серверна архітектура
- Реєстрація, автентифікація, авторизація
- Функціонал для п’яти видів користувачів (клієнт, компанія, заклад, адміністратор, служба підтримки)
- Оновлення даних у режимі реального часу

Спеціалізовані технології

- СУБД – Microsoft SQL Server Management Studio
- Платформи розробки – .NET, Node.js.
- Мови програмування – C#, TypeScript.
- Фреймворки – Angular, Entity Framework Core.
- Середовища розробки – Microsoft Visual Studio, Microsoft Visual Studio Code.
- Протокол передачі даних – HTTPS
- Авторизація – JWT

Архітектура серверу

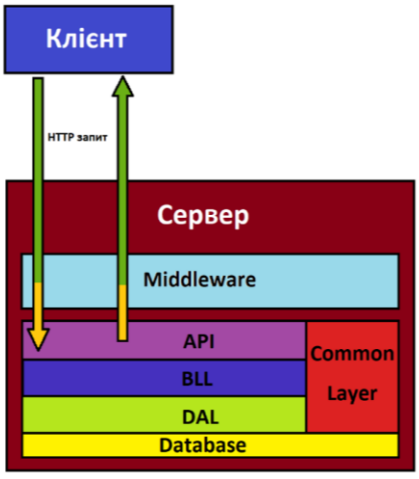
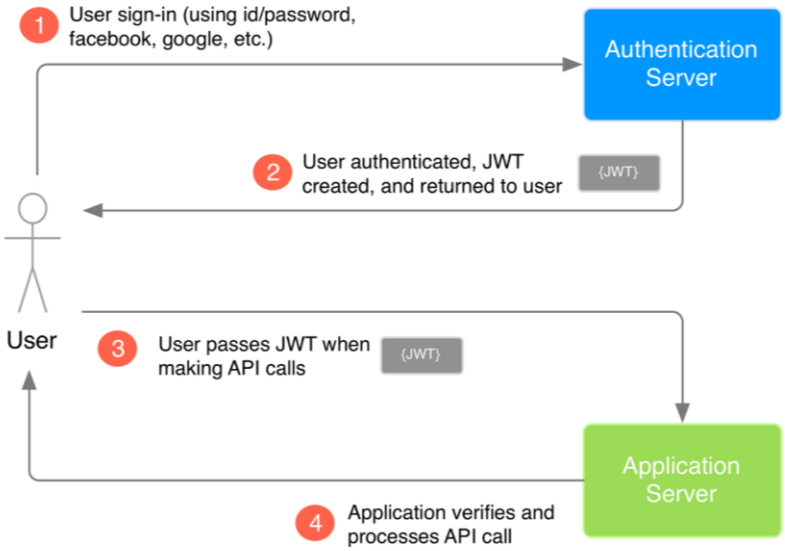
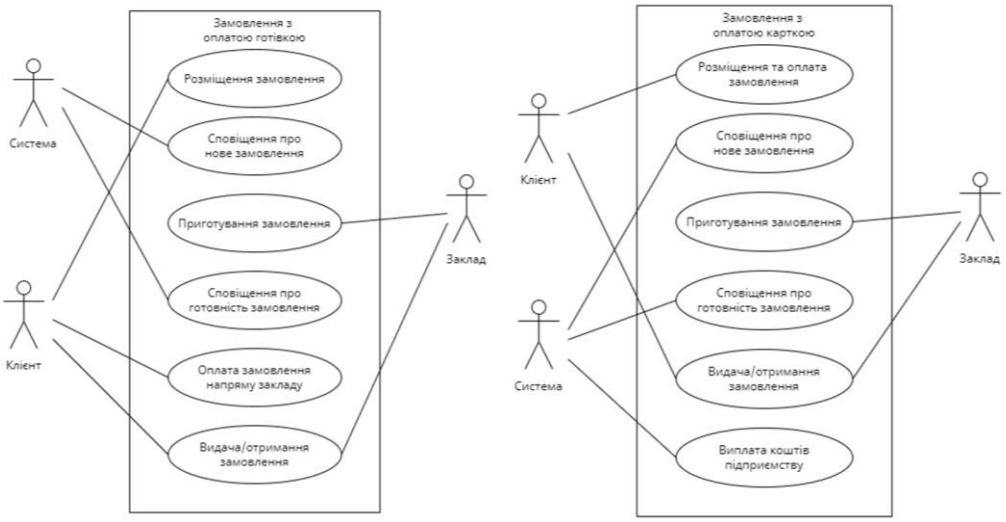


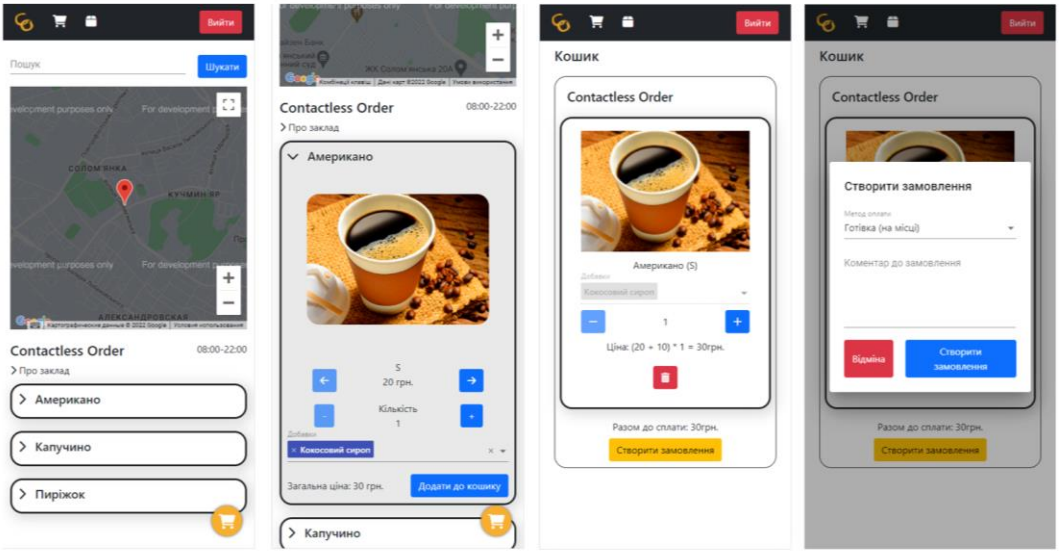
Схема автентифікації та авторизації



Процес замовлення



Приклад використання



Наукова новизна та практична значимість

Наукова новизна дослідження полягає у створенні нового виду взаємодії клієнтів закладів громадського харчування з підприємствами та розширення таких на заклади малого бізнесу.

Практична значимість дослідження полягає в економії часу бізнесу та клієнтів за рахунок автоматизації деяких процесів. Крім того, створена інформаційна система може слугувати концептом для створення нової компанії, яка займатиметься подальшим розвитком та підтримкою додатку, або інтеграції такого функціоналу в уже існуючі додатки.

Апробація

Результати дослідження бакалаврської роботи апробовані на всеукраїнських науково-технічних конференціях:

1. Матвійчук А. М. Оптимізація процесу обслуговування в закладах громадського харчування. *Застосування програмного забезпечення в інфокомунікаційних технологіях*: матеріали наук.-тех. конф., м. Київ, 20 квітня 2022 р. / Державний університет телекомунікацій, кафедра інженерії програмного забезпечення. Київ, 2022

2. Матвійчук А. М. Оптимізація процесу обслуговування в закладах громадського харчування. *Сучасні інтелектуальні інформаційні технології в науці та освіті*: матеріали наук.-тех. конф., м. Київ, 5 квітня 2022 р. / Державний університет телекомунікацій, кафедра штучного інтелекту. Київ, 2022

Висновки

Дана робота була спрямована на проектування та реалізацію інструменту для оптимізації та автоматизації деяких бізнес процесів при обслуговуванні у закладах громадського харчування.

Захват та обробка кадру обличчя займає декілька секунд.

1. Було обґрунтовано актуальність розробленої системи та її наукову новизну шляхом аналізу існуючих засобів оптимізації процесу обслуговування в закладах громадського харчування
2. Для розробки успішного продукту, було проаналізовано типові задачі, які виконують учасники взаємодії та їх послідовність та створено набір вимог у вигляді UML діаграм.
3. Описано програмні засоби та інструменти, котрі було застосовано для розробки програмного забезпечення. Підібрано та використано допоміжні інструменти розробки, такі як Postman та Google API.
4. Розроблена система цілеспрямована на допомогу закладам громадського харчування та їх клієнтам для швидкого прийому та оплати замовлень.

Дякую за увагу!

Додаток Б

Сервер:

Startup.cs:

```
using System;
using System.Linq;
using System.Net;
using System.Text;
using System.Threading.Tasks;
using ContactlessOrder.Api.Hubs;
using ContactlessOrder.Api.JsonConverters;
using ContactlessOrder.Api.Middleware;
using ContactlessOrder.Api.Validators;
using ContactlessOrder.BLL.Infrastructure;
using ContactlessOrder.BLL.Infrastructure.MappingProfiles;
using ContactlessOrder.BLL.Interfaces;
using ContactlessOrder.BLL.Services;
using ContactlessOrder.BLL.HubConnections.Hubs;
using ContactlessOrder.Common.Constants;
using ContactlessOrder.Common.Dto.Auth;
using ContactlessOrder.DAL.EF;
using ContactlessOrder.DAL.Interfaces;
using ContactlessOrder.DAL.Repositories;
using FluentValidation;
using FluentValidation.AspNetCore;
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http.Connections;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.SignalR;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.IdentityModel.Tokens;
using Serilog;
using Serilog.Events;
using Serilog.Filters;
using Serilog.Sinks.Email;

namespace ContactlessOrder.Api
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddControllers().AddFluentValidation()
                .AddJsonOptions(opt =>
                {
                    opt.JsonSerializerOptions.Converters.Add(new StringConverter());
                    opt.JsonSerializerOptions.Converters.Add(new DateTimeConverter());
                });

            services.AddDbContext<ContactlessOrderContext>(options =>
               options.UseSqlServer(Configuration.GetConnectionString(AppConstants.DBNameLocal)));

            services.AddTransient<IValidationService, ValidationService>();
            services.AddTransient<IAuthService, AuthService>();
            services.AddTransient<IUserService, UserService>();
            services.AddTransient<ICompanyService, CompanyService>();
            services.AddTransient<ICateringService, CateringService>();
            services.AddTransient<IClientService, ClientService>();
```

```
            services.AddTransient<INotificationService, NotificationService>();
            services.AddTransient<ICommonService, CommonService>();
            services.AddTransient<IAdminService, AdminService>();
            services.AddTransient<ISupportService, SupportService>();

            services.AddTransient<IUserRepository, UserRepository>();
            services.AddTransient<ICompanyRepository, CompanyRepository>();
            services.AddTransient<ICateringRepository, CateringRepository>();
            services.AddTransient<IClientRepository, ClientRepository>();
            services.AddTransient<IAdminRepository, AdminRepository>();
            services.AddTransient<ISupportRepository, SupportRepository>();

            services.AddTransient<EmailHelper>();
            services.AddTransient<FileHelper>();

            services.AddAutoMapper(typeof(Startup), typeof(UserMappingProfile));

            services.AddTransient<IValidator<UserDto>, UserValidation>();

            services.Configure<ApiBehaviorOptions>(options =>
            {
                options.InvalidModelStateResponseFactory = (context) =>
                {
                    var error = context.ModelState.Values.SelectMany(x => x.Errors.Select(p =>
p.ErrorMessage)).FirstOrDefault();
                    var result = new
                    {
                        Message = error
                    };
                    return new BadRequestObjectResult(result);
                };
            });

            services.AddCors(options =>
            {
                options.AddDefaultPolicy(builder =>
                {
                    builder.WithOrigins(Configuration[AppConstants.CorsOrigin])
                        .AllowAnyMethod()
                        .AllowAnyHeader()
                        .AllowCredentials();
                });
            });

            var appSettingsSection = Configuration.GetSection("AppSettings");
            services.Configure<AppSettings>(appSettingsSection);

            var appSettings = appSettingsSection.Get<AppSettings>();

            var key = Encoding.ASCII.GetBytes(appSettings.Secret);

            services.AddAuthentication(x =>
            {
                x.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
                x.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
            })
                .AddJwtBearer(x =>
                {
                    x.RequireHttpsMetadata = false;
                    x.SaveToken = true;
                    x.TokenValidationParameters = new TokenValidationParameters
                    {
                        ValidateIssuerSigningKey = true,
                        IssuerSigningKey = new SymmetricSecurityKey(key),
                        ValidateIssuer = false,
                        ValidateAudience = false
                    };

                    x.Events = new JwtBearerEvents
                    {
                        OnMessageReceived = context =>
                        {
                            var accessToken = context.Request.Query["access_token"];

                            if (!string.IsNullOrEmpty(accessToken))
                            {
                                context.Token = accessToken;
```

```
                        }

                        return Task.CompletedTask;
                    }
                };
            });

            Log.Logger = new LoggerConfiguration()
                .Enrich.FromLogContext()
                .MinimumLevel.Information()
                .MinimumLevel.Override("Microsoft", LogEventLevel.Warning)
                .WriteTo.File(Configuration.GetValue<string>(AppConstants.LogPath))
                .WriteTo.Logger(l => l
                    .Filter.ByIncludingOnly(Matching.FromSource<ExceptionMiddleware>())
                    .WriteTo.Email(new EmailConnectionInfo()
                    {
                        FromEmail = Configuration.GetValue<string>(AppConstants.ErrorFromEmail),
                        ToEmail = Configuration.GetValue<string>(AppConstants.ErrorToEmail),
                        EmailSubject = Configuration.GetValue<string>(AppConstants.ErrorMailSubject) + "
{Domain}",
                        NetworkCredentials = new NetworkCredential(
                            Configuration.GetValue<string>(AppConstants.SmtpErrorUserName),
                            Configuration.GetValue<string>(AppConstants.SmtpErrorPassword)),
                        Port = Configuration.GetValue<int>(AppConstants.SmtpErrorPort),
                        MailServer = Configuration.GetValue<string>(AppConstants.SmtpErrorServer),
                    }))
                .CreateLogger();

            services.AddSingleton<IUserIdProvider, UserIdProvider>();
            services.AddSignalR(opt =>
            {
                opt.ClientTimeoutInterval = TimeSpan.FromMinutes(2);
                opt.KeepAliveInterval = TimeSpan.FromMinutes(1);
            });

            services.AddMemoryCache();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            using (var scope = app.ApplicationServices.GetService<IServiceScopeFactory>().CreateScope())
            {
                scope.ServiceProvider.GetRequiredService<ContactlessOrderContext>().Database.Migrate();

                (Configuration as IConfigurationRoot)?.Reload();
            }

            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseHsts();
            }

            app.UseMiddleware<ExceptionMiddleware>();
            app.UseHttpsRedirection();

            app.UseStaticFiles();

            app.UseRouting();

            app.UseCors();

            app.UseAuthentication();
            app.UseAuthorization();

            app.UseEndpoints(c =>
            {
                c.MapControllers();
                c.MapHub<OrdersHub>("/orders", opt => opt.Transports = HttpTransportType.WebSockets);
                c.MapHub<SupportHub>("/support", opt => opt.Transports = HttpTransportType.WebSockets);
                c.MapHub<AdminHub>("/admin", opt => opt.Transports = HttpTransportType.WebSockets);
            });
        }
    }
```

```
}
```

AuthController.cs:

```
using System.Security.Claims;
using System.Threading.Tasks;
using ContactlessOrder.BLL.Interfaces;
using ContactlessOrder.Common.Constants;
using ContactlessOrder.Common.Dto.Common;
using ContactlessOrder.Common.Dto.Auth;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace HE.Material.Api.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class AuthController : ControllerBase
    {
        private readonly IAuthService _authService;
        private readonly IValidationService _validationService;

        public AuthController(IAuthService authService, IValidationService validationService)
        {
            _authService = authService;
            _validationService = validationService;
        }

        [HttpPost]
        public async Task<IActionResult> Authenticate([FromBody] UserLoginRequestDto dto)
        {
            var response = await _authService.Authenticate(dto);

            if (!string.IsNullOrEmpty(response.ErrorMessage))
            {
                return BadRequest(new { message = response.ErrorMessage });
            }

            return Ok(new { Token = response.Response });
        }

        [HttpPost("GoogleLogin")]
        public async Task<IActionResult> GoogleAuthenticate(GoogleRegisterRequestDto dto)
        {
            var response = await _authService.GoogleAuthenticate(dto);

            if (!string.IsNullOrEmpty(response.ErrorMessage))
            {
                return BadRequest(new { message = response.ErrorMessage });
            }

            return Ok(new { Token = response.Response });
        }

        [HttpPost("Register")]
        public async Task<IActionResult> Register(UserRegisterRequestDto dto)
        {
            var response = await _authService.Register(dto);

            if (!string.IsNullOrEmpty(response.ErrorMessage))
            {
                return BadRequest(new { message = response.ErrorMessage });
            }

            return Ok(new { message = response.Response });
        }

        [HttpPost("RegisterCompany")]
        public async Task<IActionResult> RegisterCompany(CompanyRegisterRequestDto dto)
        {
            var response = await _authService.RegisterCompany(dto);

            if (!string.IsNullOrEmpty(response.ErrorMessage))
            {
                return BadRequest(new { message = response.ErrorMessage });
            }
```

```
            return Ok(new { message = response.Response });
        }

        [HttpPost("ValidateEmail")]
        public async Task<IActionResult> ValidateEmail(ValidateValueDto dto)
        {
            var message = await _validationService.ValidateEmail(dto.Value, dto.Id);

            return Ok(new { message });
        }

        [HttpPost("ValidatePhoneNumber")]
        public async Task<IActionResult> ValidatePhoneNumber(ValidateValueDto dto)
        {
            var message = await _validationService.ValidatePhoneNumber(dto.Value, dto.Id);

            return Ok(new { message });
        }

        [HttpPost("ValidateCompanyName")]
        public async Task<IActionResult> ValidateCompanyName(ValidateValueDto dto)
        {
            var message = await _validationService.ValidateCompanyName(dto.Value, dto.Id);

            return Ok(new { message });
        }

        [Authorize]
        [HttpPost("ConfirmEmail")]
        public async Task<IActionResult> ConfirmEmail()
        {
            var userId = int.Parse(User.FindFirstValue(TokenProperties.Id));
            var message = await _authService.ConfirmEmail(userId);

            if (!string.IsNullOrEmpty(message))
            {
                return BadRequest(new { message });
            }

            return Ok(new { message });
        }
    }
}
```

AuthService.cs:

```
using System;
using System.Collections.Generic;
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using System.Text;
using System.Threading.Tasks;
using AutoMapper;
using ContactlessOrder.BLL.Infrastructure;
using ContactlessOrder.BLL.Interfaces;
using ContactlessOrder.Common.Constants;
using ContactlessOrder.Common.Dto.Common;
using ContactlessOrder.Common.Dto.Auth;
using ContactlessOrder.DAL.Entities.Users;
using ContactlessOrder.DAL.Interfaces;
using Google.Apis.Auth;
using Microsoft.Extensions.Configuration;
using Microsoft.IdentityModel.Tokens;
using ContactlessOrder.DAL.Entities.Companies;

namespace ContactlessOrder.BLL.Services
{
    public class AuthService : IAuthService
    {
        private readonly IValidationService _validationService;
        private readonly IUserRepository _repository;
        private readonly IMapper _mapper;
        private readonly EmailHelper _emailHelper;
        private readonly INotificationService _notificationService;
        private readonly string _secret;
```

```
        private readonly string _googleClientId;

        public AuthService(IUserRepository repository, IConfiguration configuration, IMapper mapper,
            EmailHelper emailHelper, IValidationService validationService, INotificationService
notificationService)
        {
            _secret = configuration["AppSettings:Secret"];
            _googleClientId = configuration["GoogleAuthSettings:clientId"];
            _repository = repository;
            _mapper = mapper;
            _emailHelper = emailHelper;
            _validationService = validationService;
            _notificationService = notificationService;
        }

        public async Task<ResponseDto<string>> Authenticate(UserLoginRequestDto dto)
        {
            var passwordHash = CryptoHelper.GetMd5Hash(dto.Password);
            var user = await _repository.GetUser(dto.Email);

            if (user == null)
            {
                return new ResponseDto<string>() { ErrorMessage = "Користувач не знайдений" };
            }
            else if (user.PasswordHash != passwordHash)
            {
                return new ResponseDto<string>() { ErrorMessage = "Невірний пароль" };
            }
            else if (!user.EmailConfirmed)
            {
                return new ResponseDto<string>() { ErrorMessage = "Перевірте скриньку електронної пошти
для підтвердження адреси" };
            }

            return new ResponseDto<string>() { Response = GenerateToken(user) };
        }

        public async Task<ResponseDto<string>> GoogleAuthenticate(GoogleRegisterRequestDto dto)
        {
            try
            {
                var settings = new GoogleJsonWebSignature.ValidationSettings()
                {
                    Audience = new List<string>() { _googleClientId }
                };
                var payload = await GoogleJsonWebSignature.ValidateAsync(dto.IdToken, settings);

                var user = await _repository.GetUser(payload.Email);

                if (user == null)
                {
                    var role = await _repository.Get<Role>(e => e.Value == UserRoles.ClientValue);

                    user = new User()
                    {
                        Email = payload.Email,
                        FirstName = payload.GivenName,
                        LastName = payload.FamilyName,
                        PasswordHash = string.Empty,
                        PhoneNumber = dto.PhoneNumber,
                        ProfilePhotoPath = payload.Picture,
                        RegistrationDate = DateTime.UtcNow,
                        EmailConfirmed = true,
                        RoleId = role.Id,
                    };

                    await _repository.Add(user);
                    await _repository.SaveChanges();

                    user.Role = role;
                }

                return new ResponseDto<string>() { Response = GenerateToken(user) };
            }
            catch (Exception)
            {
```

```
                return new ResponseDto<string>() { ErrorMessage = "Невірна спроба зовнішньої
автентифікації" };
            }
        }

        public async Task<ResponseDto<string>> Register(UserRegisterRequestDto dto)
        {
            var error = await _validationService.ValidateUser(dto);
            if (!string.IsNullOrEmpty(error))
            {
                return new ResponseDto<string>() { ErrorMessage = error };
            }

            var user = _mapper.Map<User>(dto);

            var role = await _repository.Get<Role>(e => e.Value == UserRoles.ClientValue);

            user.RoleId = role.Id;
            user.RegistrationDate = DateTime.UtcNow;
            user.EmailConfirmed = false;
            user.PasswordHash = CryptoHelper.GetMd5Hash(dto.Password);

            await _repository.Add(user);
            await _repository.SaveChanges();

            user.Role = role;

            await _emailHelper.SendConfirmEmail(user.Email, $"{user.FirstName} {user.LastName}",
GenerateToken(user));

            return new ResponseDto<string>() { Response = "Посилання для підтвердження електронної пошти
надіслано на вашу поштову адресу" };
        }

        public async Task<ResponseDto<string>> RegisterCompany(CompanyRegisterRequestDto dto)
        {
            var error = await _validationService.ValidateCompany(dto);
            if (!string.IsNullOrEmpty(error))
            {
                return new ResponseDto<string>() { ErrorMessage = error };
            }

            var role = await _repository.Get<Role>(e => e.Value == UserRoles.CompanyValue);

            var user = new User()
            {
                FirstName = string.Empty,
                LastName = string.Empty,
                PasswordHash = CryptoHelper.GetMd5Hash(dto.Password),
                Email = dto.Email,
                PhoneNumber = dto.PhoneNumber,
                RegistrationDate = DateTime.UtcNow,
                EmailConfirmed = false,
                RoleId = role.Id,
                Company = new Company()
                {
                    Name = dto.Name,
                }
            };

            await _repository.Add(user);
            await _repository.SaveChanges();

            user.Role = role;

            await _emailHelper.SendConfirmEmail(user.Email, dto.Name, GenerateToken(user));

            return new ResponseDto<string>() { Response = "Посилання для підтвердження електронної пошти
надіслано на вашу поштову адресу" };
        }

        public async Task<string> ConfirmEmail(int userId)
        {
            var user = await _repository.GetUser(userId);

            if (user == null)
            {
```

```
                return "Користувач не знайдений";
            }
            else if (user.EmailConfirmed)
            {
                return "Електронна адреса вже підтвердженна";
            }

            user.EmailConfirmed = true;
            await _repository.SaveChanges();

            if (user.Company != null)
            {
                await _notificationService.NotifyCompanyAdded(userId);
            }

            return string.Empty;
        }

        private string GenerateToken(User user)
        {
            var tokenHandler = new JwtSecurityTokenHandler();
            var key = Encoding.ASCII.GetBytes(_secret);

            var claims = new List<Claim>
            {
                new Claim(TokenProperties.Id, user.Id.ToString()),
                new Claim(TokenProperties.Email, user.Email),
                new Claim(TokenProperties.FullName, GetUserName(user)),
                new Claim(TokenProperties.Role, UserRoles.GetName(user.Role.Value)),
                new Claim(TokenProperties.RoleValue, user.Role.Value.ToString()),
            };

            var tokenDescription = new SecurityTokenDescriptor
            {
                Subject = new ClaimsIdentity(claims),
                Expires = DateTime.UtcNow.AddDays(10),
                SigningCredentials = new SigningCredentials(new SymmetricSecurityKey(key),
                     SecurityAlgorithms.HmacSha256Signature)
            };

            var token = tokenHandler.WriteToken(tokenHandler.CreateToken(tokenDescription));
            return token;
        }

        private string GetUserName(User user)
        {
            return user.Company?.Name ?? user.Catering?.Name ?? $"{user.FirstName} {user.LastName}";
        }
    }
}
```

ContactlessOrderContext.cs:

```
using Microsoft.EntityFrameworkCore;

namespace ContactlessOrder.DAL.EF
{
    public partial class ContactlessOrderContext : DbContext
    {
        public ContactlessOrderContext(DbContextOptions<ContactlessOrderContext> options)
            : base(options)
        {
        }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.ApplyConfigurationsFromAssembly(GetType().Assembly);

            OnModelCreatingPartial(modelBuilder);
        }

        partial void OnModelCreatingPartial(ModelBuilder modelBuilder);
    }
}
```

SupportRepository.cs:

```
using ContactlessOrder.DAL.EF;
using ContactlessOrder.DAL.Entities.Users;
using ContactlessOrder.DAL.Interfaces;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContactlessOrder.DAL.Repositories
{
    public class SupportRepository : RepositoryBase, ISupportRepository
    {
        public SupportRepository(ContactlessOrderContext context)
            : base(context)
        {
        }

        public async Task<IEnumerable<Complain>> GetComplains(int statusValue)
        {
            return await Context.Set<Complain>()
                .Include(e => e.User)
                .Include(e => e.Catering)
                .ThenInclude(e => e.Company.User)
                .Include(e => e.Order)
                .Where(e => (int)e.Status == statusValue)
                .ToListAsync();
        }

        public async Task<Complain> GetComplain(int id)
        {
            return await Context.Set<Complain>()
                .Include(e => e.User)
                .Include(e => e.Catering)
                .ThenInclude(e => e.Company.User)
                .Include(e => e.Order)
                .FirstOrDefaultAsync(e => e.Id == id);
        }

        public async Task<User> GetUser(int id)
        {
            return await Context.Set<User>()
                .Include(e => e.Role)
                .FirstOrDefaultAsync(e => e.Id == id);
        }
    }
}
```

Клієнт:

company-setting.component.ts:

```
import { ChangeDetectorRef, Component, OnDestroy, OnInit } from '@angular/core';
import {
  AbstractControl,
  FormBuilder,
  FormGroup,
  Validators,
} from '@angular/forms';
import { MatDialog, MatDialogConfig } from '@angular/material/dialog';
import { DomSanitizer } from '@angular/platform-browser';
import { map, Observable, Subject, takeUntil, tap } from 'rxjs';
import { PLACEHOLDER_IMAGE } from 'src/app/shared/constants/images';
import { SharedService } from 'src/app/shared/services/shared.service';
import { CompanySettingsService } from '../company-settings.service';
import { ChangePaymentDataComponent } from '../../../../shared/change-payment-data/change-payment-
data.component';

@Component({
  selector: 'app-company-settings',
```

```
  templateUrl: './company-settings.component.html',
  styleUrls: ['./company-settings.component.scss'],
})
export class CompanySettingsComponent implements OnInit, OnDestroy {
  public form: FormGroup;

  public logo: any;
  public defaultLogo = PLACEHOLDER_IMAGE;

  public editMode = false;

  public company: any;
  private newLogo: any;

  private onDestroy$ = new Subject<void>();

  constructor(
    fb: FormBuilder,
    private dialog: MatDialog,
    private companySettingsService: CompanySettingsService,
    private sanitizer: DomSanitizer,
    private sharedService: SharedService,
    private cdr: ChangeDetectorRef
  ) {
    this.form = fb.group({
      name: [
        null,
        Validators.required,
        this.uniqueValidator((value) =>
          this.companySettingsService.validateCompanyName(
            this.company.userId,
            value
          )
        ),
      ],
      address: [null],
      email: [
        '',
        [Validators.required, Validators.email],
        this.uniqueValidator((value) =>
          this.companySettingsService.validateEmail(this.company.userId, value)
        ),
      ],
      phoneNumber: [
        '+380',
        [Validators.required, Validators.pattern(/\+380\d{9}/)],
        this.uniqueValidator((value) =>
          this.companySettingsService.validatePhoneNumber(
            this.company.userId,
            value
          )
        ),
      ],
      description: [null],
    });
  }

  public async ngOnInit() {
    await Promise.all([this.getCompany(), this.getCompanyLogo()]);
    this.sharedService.validateFormFields(this.form);

    this.setData();
    this.subscribeToChanges();
  }

  public ngOnDestroy() {
    this.onDestroy$.next();
    this.onDestroy$.complete();
  }

  public async changePaymentData() {
    const config = new MatDialogConfig();
    config.width = '500px';
    config.data = { ...this.company };

    if (this.company.paymentDataId) {
      try {
```

```
      config.data.paymentData =
        await this.companySettingsService.getPaymentData();
    } catch (error) {
      this.sharedService.showRequestError(error);
      return;
    }
  }

  this.dialog
    .open(ChangePaymentDataComponent, config)
    .afterClosed()
    .subscribe((result) => {
      if (result?.success) {
        this.getCompany();
      }
    });
}

public startEdit() {
  this.editMode = true;
  this.cdr.markForCheck();
}

public changeLogo(event) {
  const file = event.target.files[0];

  if (file) {
    this.logo = this.sanitizer.bypassSecurityTrustUrl(
      URL.createObjectURL(file)
    );

    this.newLogo = file;
    this.cdr.markForCheck();
  }
}

public removeLogo() {
  this.logo = this.defaultLogo;
  this.newLogo = null;

  this.cdr.markForCheck();
}

public async saveChanges() {
  this.editMode = false;

  const data = {
    ...this.form.value,
    removeLogo: this.logo === this.defaultLogo,
    logo: this.newLogo,
  };

  try {
    await this.companySettingsService.updateCompanyData(data);
  } catch (error) {
    this.sharedService.showRequestError(error);
  }

  this.cdr.markForCheck();
}

private async getCompany() {
  try {
    this.company = await this.companySettingsService.getCompany();
  } catch (error) {
    this.sharedService.showRequestError(error);
  }
}

private async getCompanyLogo() {
  try {
    const logo = await this.companySettingsService.getCompanyLogo();
    this.logo = this.sanitizer.bypassSecurityTrustResourceUrl(
      URL.createObjectURL(logo)
    );
  } catch (error) {
    this.logo = this.defaultLogo;
```

```
    }

    this.cdr.markForCheck();
  }

  private setData() {
    this.form.patchValue(this.company);
    this.cdr.markForCheck();
  }

  private subscribeToChanges() {
    this.form.controls.phoneNumber.valueChanges
      .pipe(takeUntil(this.onDestroy$))
      .subscribe((value) => {
        if (!value.startsWith('+380')) {
          this.form.controls.phoneNumber.patchValue('+380');
        }
      });
  }

  private uniqueValidator =
    (validate: (value) => Observable<any>) => (control: AbstractControl) =>
      validate(control.value).pipe(
        map((result) =>
          result?.message ? { notUnique: result.message } : null
        ),
        tap(() => this.cdr.markForCheck())
      );
}
```

company-settings.component.scss:

```
.logo-container {
  width: 100%;
  max-width: 350px;
  height: 350px;
  display: flex;
  justify-content: center;
  align-items: center;
  position: relative;
  border: 1px solid;

  .company-logo {
    max-width: calc(100% - 30px);
    max-height: calc(100% - 30px);
  }

  .edit-logo-container {
    position: absolute;
    bottom: 10px;
    right: 10px;
  }
}

.main-company-info {
  display: flex;
  margin-bottom: 50px;
}

.company-grids-container {
  flex: 1;
  display: flex;
  flex-direction: column;
  justify-content: space-between;
  height: calc(100vh - 80px);
  row-gap: 10px;
}

@media screen and (max-width: 600px) {
  .main-company-info {
    display: block;
  }

  .logo-container {
    margin-bottom: 20px;
```

```
  }
}

@media screen and (min-width: 768px) {
  form {
    margin: 0 50px;
  }

  .logo-container {
    margin-right: 20px;
  }
}

@media screen and (min-width: 950px) {
  .full-company-container {
    display: flex;
  }
}
```

company-settings.component.html:

```
<block-ui>
  <div class="full-company-container">
    <form [formGroup]="form" style="flex: 1">
      <div class="main-company-info">
        <div class="logo-container">
          <img [src]="logo" class="company-logo" />

          <div *ngIf="editMode" class="edit-logo-container">
            <button class="btn btn-primary" (click)="fileDialog.click()">
              <i class="fa fa-pencil" aria-hidden="true"></i>

              <input
                #fileDialog
                style="display: none"
                type="file"
                accept="image/*"
                (change)="changeLogo($event)"
              />
            </button>

            <button
              *ngIf="logo !== defaultLogo"
              class="btn btn-danger"
              style="margin-left: 10px"
              (click)="removeLogo()"
            >
              <i class="fa fa-trash" aria-hidden="true"></i>
            </button>
          </div>
        </div>

        <div>
          <div class="d-flex justify-content-end">
            <button
              *ngIf="!editMode"
              class="btn btn-primary"
              (click)="startEdit()"
            >
              <i class="fa fa-pencil" aria-hidden="true"></i>
            </button>

            <button
              *ngIf="editMode"
              class="btn btn-primary"
              (click)="saveChanges()"
              [disabled]="!form.valid"
            >
              Зберегти
            </button>
          </div>

          <mat-form-field>
            <input
              matInput
```

```
        placeholder="Назва компанії"
        formControlName="name"
        [readonly]="!editMode"
      />

      <mat-error *ngIf="form.controls.name.hasError('notUnique')">
        {{ form.controls.name.errors.notUnique }}
      </mat-error>
    </mat-form-field>

    <mat-form-field>
      <input
        matInput
        placeholder="Адреса"
        formControlName="address"
        [readonly]="!editMode"
      />
    </mat-form-field>

    <mat-form-field>
      <input
        matInput
        placeholder="Електронна пошта"
        formControlName="email"
        [readonly]="!editMode"
      />

      <mat-error *ngIf="form.controls.email.hasError('notUnique')">
        {{ form.controls.email.errors.notUnique }}
      </mat-error>

      <mat-error *ngIf="form.controls.email.hasError('email')">
        Невірна електронна пошта
      </mat-error>
    </mat-form-field>

    <mat-form-field>
      <input
        matInput
        placeholder="Номер телефону"
        formControlName="phoneNumber"
        [readonly]="!editMode"
      />

      <mat-error *ngIf="form.controls.phoneNumber.hasError('notUnique')">
        {{ form.controls.phoneNumber.errors.notUnique }}
      </mat-error>

      <mat-error *ngIf="form.controls.phoneNumber.hasError('pattern')">
        Невірний номер телефону
      </mat-error>
    </mat-form-field>

    <div class="d-flex align-items-center">
      <div *ngIf="company && !company.paymentDataId" style="flex: 1;color:red">
        <h4>Відсутні дані для оплати</h4>
      </div>

      <button class="btn btn-primary" (click)="changePaymentData()">
        {{
          company?.paymentDataId
            ? "Змінити реквізити"
            : "Додати реквізити"
        }}
      </button>
    </div>
  </div>
</div>

<mat-form-field>
  <textarea
    matInput
    placeholder="Опис"
    formControlName="description"
    rows="27"
    style="resize: none"
    [readonly]="!editMode"
```

```
        ></textarea>
      </mat-form-field>
    </form>

    <div class="company-grids-container">
      <app-caterings style="flex: 1"></app-caterings>

      <div class="d-flex" style="column-gap: 10px; flex: 1">
        <app-menu style="flex: 1"></app-menu>
        <app-modifications style="flex: 1"></app-modifications>
      </div>
    </div>
  </div>
</block-ui>
```