МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
Факультет комп‘ютерних наук та технологій
Кафедра Комп’ютерних інформаційних технологій

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри
_____Аліна САВЧЕНКО
«___»_____2023 р.

КВАЛІФІКАЦІЙНА РОБОТА
(ДИПЛОМНА РОБОТА, ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ **«МАГІСТР»**
ЗА ОСВІТНЬО-ПРОФЕСІЙНОЮ ПРОГРАМОЮ
«ІНФОРМАЦІЙНІ УПРАВЛЯЮЧІ СИСТЕМИ ТА ТЕХНОЛОГІЇ»

Тема: «Система управління бізнесом з аналітикою та оптимізацією на базі ШІ»

Виконавець: студент групи УС-211М Луцький Іван Максимович

Керівник: к.т.н., доцент Холявкіна Тетяна Володимирівна

Нормоконтролер:_____ Ігор РАЙЧЕВ

Київ 2023

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет *Факультет комп'ютерних наук та технологій*

Кафедра *Комп'ютерних інформаційних технологій*

Галузь знань, спеціальність, освітньо-професійна програма: 12 «Інформаційні технології», 122 «Комп'ютерні науки», «Інформаційні управляючі системи та технології»

ЗАТВЕРДЖУЮ
Завідувач випускової кафедри
_____Аліна САВЧЕНКО
«_____»_____ 2023 р.

ЗАВДАННЯ

на виконання кваліфікаційної роботи студента

Луцького Івана Максимовича
(прізвище, ім'я, по батькові)

1. Тема роботи: «Система управління бізнесом з аналітикою та оптимізацією на базі ШІ» затверджена наказом ректора від «29» вересня 2023 р. за №1976/ст.

2. Термін виконання роботи: 02.10.2023 – 18.12.2023р.

3. Вихідні дані до роботи: система управління бізнесом з аналітикою та оптимізацією на базі ШІ.

4. Зміст пояснювальної записки: вступ, дослідження предметної області та та постановка задачі, дослідження існуючих рішень та потреб клієнтів, огляд технологій створення клієнт-сервених систем, опис процесу створення клієнт-сервеного веб-сервісу, тестування програмного продукту, аналіз виконаного проектування, висновки.

5. Перелік обов'язкового ілюстративного матеріалу: Схема архітектури клієнт-серверного додатку, варіанти дизайну системи, структури та вмісти директорій, файлів, та модулів, інтерфейс бази даних, сторінки клієнтського інтерфейсу системи.

6. Календарний план-графік

№ п/п	*Завдання*	*Термін виконання*	*Підпис керівника*
1.	Проаналізувати літературу та джерела за темою дипломної роботи	02.10.23 – 08.10.23р.	
2.	Розроблення та затвердження плану дипломної роботи	09.10.23 – 11.10.23р.	
3.	Привести консультації з науковим керівником щодо створення першого розділу	12.10.23 – 16.10.23р.	
4.	Розробка розділу 1	17.10.23 – 28.10.23р.	
5.	Розробка розділу 2	29.10.23 – 19.11.23р.	
6.	Розробка розділу 3	20.11.23 – 01.11.23р.	
8.	Висновки та оформлення пояснювальної записки дипломної роботи	02.12.23 – 13.12.23р.	
9.	Підписання необхідних документів у встановленому порядку	14.12.22 –16.12.23р.	
10.	Підготовка до захисту та попередній захист дипломного проекту на випусковій кафедрі дипломної роботи	16.12.23 – 18.12.23р.	

7. Дата видачі завдання: «02» жовтня 2023 р.

Керівник дипломної роботи _____ Тетяна ХОЛЯВКІНА
(підпис керівника) (П.І.Б.)

Завдання прийняв до виконання _____ Іван ЛУЦЬКИЙ
(підпис випускника) (П.І.Б.)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи «Система управління бізнесом з аналітикою та оптимізацією на базі ШІ» містить 102 сторінки, 24 рисунки, 7 бібліографічних посилань.

Об'єктом дослідження є: система управління взаємодіями з клієнтами для бізнесів.

Предметом дослідження є: система, що дозволяє керувати товарами бізнесу, аналізувати та працювати з замовленнями від клієнтів, аналізувати фінансові транзакції та процеси, а також використовувати сервіси математичних моделей штучного інтелекту для роботи з даними та побудови графіків чи діаграм для глибокої аналітики бізнес-процесів.

Мета роботи: створити повністю керовану, та адаптивну під унікальні процеси різних бізнесів, систему, яка довзолить не просто керувати усима процесами, що пов’язані із взаємодіями з клієнтами, а й налаштувати дану систему повністю під бізнес-замовника, додавши унікальні можливості аналітики за допомогою моделей штучного інтелекту.

Для досягнення поставленої мети необхідно виконати **наступні завдання**:

- Проаналізувати існуюючі рішення подібних систем, виявляючи недоліки та додаючи новітні можливості в нашу систему.
- Зформувати каркас дизайну, спираючись на вже існуючі схожі системи.
- Побудова архітектури: від загальної для всієї системи, до окремо структур серверу, клієнтської частини, та бази даних.
- Написання коду системи із додатковим залученням сервісів штучного інтелекту для більш швидкого написання коду.
- Інтеграція сервісів штучного інтелекту.
- Інтеграція системи управління взаємодіями з клієнтами з реальним бізнесом.

Методи дослідження: проведення аналогій і відмінностей зі схожими системами, пояснення способів реалізації даної системи, моделювання системи, опис

та аналіз використаних технологічних рішень, застосування системи до реального бізнес-проекту.

Ключові слова: JAVASCRIPT, NODE JS, MONGO DB, NEST JS, CHAT GPT, УПРАВЛІННЯ БІЗНЕС-ПРОЦЕСАМИ, КЛІЄНТ-СЕРВЕРНИЙ ДОДАТОК, ШІ, CRM, REACT.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

Клієнт-серверна архітектура - це один із архітектурних шаблонів програмного забезпечення та є домінуючою концепцією у створенні розподілених мережних застосунків і передбачає взаємодію та обмін даними між ними [5];

Web-додаток - це програмне забезпечення, яке запускається у вашому веб-браузері. Бізнеси використовують їх щоб обмінюватися інформацією та надавати послуги віддалено. Вони використовують веб-додатки для зручного та безпечного зв'язку з клієнтами. Найпоширеніші функції веб-сайтів, такі як кошики для покупок, пошук і фільтрація товарів, обмін миттєвими повідомленнями та стрічки новин у соціальних мережах, за своєю суттю є веб-додатками. Вони дозволяють отримати доступ до складної функціональності без встановлення чи налаштування програмного забезпечення [6];

Фреймворк - це програмне середовище, яке спрощує та прискорює створення програмного забезпечення. За використання фреймворків ви пишете лише код, який реалізує логіку, специфічну для вашого продукту. Вам не доводиться самостійно забезпечувати роботу з базою даних, автентифікацію, підтримку сеансів тощо. Все це реалізовано у фреймворках [7];

HTML - це стандартна мова розмітки для створення та відображення веб-сторінок. Вона використовується

для опису структури та вмісту веб-документа, і є однією з основних технологій для розробки веб-сайтів разом із CSS (Cascading Style Sheets) та JavaScript. HTML використовує теги (tags) для визначення елементів сторінки, таких як заголовки, абзаци, зображення, посилання і багато інших. Кожен тег має свою функцію та впливає на те, як вміст відображається на сторінці. Теги зазвичай включають в себе початковий тег <tag> та заключний тег </tag>, які обмежують вміст, що належить до цього тегу;

CSS - це мова опису стилів, яка використовується для задання вигляду та форматування веб-документів, написаних мовою HTML або XML. Основна мета CSS - відокремити зміст сторінки від її представлення, що дозволяє розробникам легко змінювати вигляд сторінок без зміни їх структури;

JS - це високорівнева мова програмування, яка використовується для надання динамічності та взаємодії на веб-сторінках. Вона дозволяє розробникам створювати скрипти, які виконуються в браузері клієнта, контролюючи поведінку сторінки під час взаємодії з користувачем або відповіді на події;

CRM - це стратегія управління взаємодією з клієнтами, а також система, що допомагає компаніям в цьому управлінні. Основною метою CRM є покращення відносин і взаєморозуміння між компанією і її клієнтами;

MVC - це архітектурний шаблон програмування, який використовується для розробки програмних додатків. Цей шаблон допомагає розділити компоненти

програми на три основні частини: Model (Модель), View (Вигляд) та Controller (Контролер). Кожна з цих частин виконує певні функції і взаємодіє з іншими для забезпечення ефективного управління додатком;

JSON - це легкий формат обміну даними, який є текстовим та легко читається людьми. Вперше цей формат був пов'язаний з JavaScript, але тепер він є незалежним від мови стандартом для обміну даними між різними мовами програмування;

Webpack - це інструмент для збірки (бандлінгу) веб-додатків. Він призначений для об'єднання і оптимізації ресурсів, таких як JavaScript, CSS, зображення та інші, щоб зробити їх ефективними для розгортання на веб-сервері. Webpack дозволяє створювати оптимізовані версії файлів, а також працювати з різними модулями та залежностями в проекті;

CRA - це інструмент командного рядка, який дозволяє швидко створювати та налаштовувати нові проекти React без необхідності вручну налаштовувати конфігурацію та інші деталі проекту. CRA допомагає спростити процес розробки, забираючи від розробника багато рутинних задач та конфігураційних деталей;

URL - це адреса, що вказує на розташування ресурсу в Інтернеті. URL визначає шлях до ресурсу та протокол зв'язку, який використовується для доступу до цього ресурсу;

URI - це ідентифікатор ресурсу, що використовується для ідентифікації та взаємодії з різноманітними ресурсами в Інтернеті. URI включає дві підкатегорії: URL

(Uniform Resource Locator) та URN (Uniform Resource Name).

HTTP - є протоколом передачі даних, який використовується для обміну інформацією між клієнтами та серверами в Інтернеті. HTTP є основним протоколом для передачі даних в World Wide Web;

REST - це архітектурний стиль для розробки веб-служб та взаємодії між компонентами системи. Архітектурний стиль REST базується на принципах, які забезпечують стандартизований та простий спосіб спілкування між розподіленими компонентами. Ідеологія REST часто використовується для розробки веб-служб, особливо в контексті веб-додатків та API;

AJAX - це техніка, яка дозволяє асинхронно взаємодіяти з сервером і оновлювати частину сторінки без повного оновлення веб-сторінки. AJAX використовує комбінацію JavaScript, XMLHttpRequest об'єкта та HTML (або інших форматів даних, таких як JSON чи XML) для взаємодії з сервером та оновлення сторінок без перезавантаження;

CLI - це інтерфейс командного рядка, що дозволяє користувачам взаємодіяти з комп'ютером за допомогою текстових команд. Користувач вводить команди в командному рядку, і операційна система або програма обробляє ці команди та виконує відповідні дії;

Heroku Dyno - У контексті платформи Heroku термін "Dyno" вказує на віртуальний контейнер, в якому запускається та виконується один екземпляр вашого додатка. Dyno - це основна одиниця обчислення на платформі Heroku.

ВСТУП

Дана робота присвячена розробці системи управління бізнесом з аналітикою та оптимізацією на базі ШІ.

Метою дипломної роботи є створення менеджерської системи управління бізнес-проектами, на базі CRM-системи, зокрема клієнтськими B2C сервіси, магазини, тощо.

Актуальність роботи полягає у нищевикладених пунктах:

✔*Зростання конкуренції на ринку* - сфера e-commerce і інтернет-сервісів постійно зростає, і конкуренція серед підприємств і магазинів у цій галузі стає все більш жорсткою. Власники бізнесів мають бути готові до конкурентного середовища та вміти швидко реагувати на зміни.

✔*Вимоги споживачів до обслуговування* - споживачі стають більш вимогливими і очікують швидкого обслуговування, зручних інтерфейсів та персоналізованих пропозицій. Забезпечення такого рівня обслуговування може бути складним завданням без спеціалізованої системи управління бізнесом.

✔*Потреба в аналітиці та інсайтах* - аналітика грає ключову роль у прийнятті стратегічних рішень в бізнесі. Інтернет-магазини та сервіси повинні аналізувати дані про продажі, клієнтів, логістику та інші аспекти діяльності для того, щоб покращувати ефективність та реагувати на зміни на ринку.

✔*ШІ як інноваційна технологія* - штучний інтелект (ШІ) розвивається стрімко і вже зараз знаходить широке застосування в різних галузях. Використання ШІ в системі управління бізнесом дозволить автоматизувати багато процесів, забезпечити персоналізацію та підвищити ефективність роботи.

✔*Споживачі очікують більшого відповідального бізнесу* - сучасні споживачі все більше звертають увагу на соціальну відповідальність бізнесу. Система управління бізнесом може допомогти власникам підприємств контролювати та

вдосконалювати свої практики стосовно сталого розвитку та екологічної відповідальності.

✔*Потреба в оптимізації внутрішніх процесів* - управління замовленнями, складом, логістикою, фінансами та іншими аспектами бізнесу вимагає ефективної оптимізації, особливо в умовах зростаючого обсягу роботи.

✔*Зменшення ризиків і помилок* - використання системи управління бізнесом з ШІ допомагає знизити ризики помилок в процесах і приймати обгрунтовані рішення на підставі аналізу даних. Це особливо важливо для фінансової стійкості бізнесу та збереження довіри клієнтів.

✔*Глобальний доступ та мобільність* - завдяки інтернет-технологіям та мобільним додаткам, система управління бізнесом може бути доступною для власників і менеджерів з будь-якої точки світу. Це дозволяє збільшити продуктивність та реагувати на події в реальному часі.

✔*Потенціал для росту* - інтернет-бізнеси мають потенціал для швидкого росту. Важливо мати систему управління, яка може масштабуватися та адаптуватися до зростаючого обсягу діяльності без великих інвестицій у нові ресурси.

✔Відкритий доступ до аналітичних засобів - інтеграція системи з різноманітними ШІ-сервісами для аналітики, оптимізації та інтелектуального прогнозування надає бізнесу конкурентну перевагу. Власники можуть використовувати дані для розробки стратегій, підвищення ефективності рекламних кампаній і зменшення витрат.

✔*Забезпечення безпеки даних* - з огляду на значущість обробки особистих даних клієнтів та фінансових інформацій, система управління бізнесом повинна гарантувати найвищий рівень безпеки. Інтегровані заходи захисту даних допомагають уникнути порушень та санкцій.

✔*Постійне оновлення та розвиток* - багато компаній змушені витрачати значні ресурси на розробку та підтримку власних систем управління бізнесом. Використання готової системи на базі ШІ дозволяє власникам фокусуватися на своєму бізнесі, а не на технічних питаннях.

Загальна актуальність проекту "Система управління бізнесом з аналітикою та оптимізацією на базі ШІ" очевидна. Вона відповідає насущним потребам сучасного бізнесу, який стикається зі складними завданнями конкурентоспроможності, аналізу даних, оптимізації та сталого росту. Система, що поєднує в собі сучасні інформаційні технології та штучний інтелект, може стати ключовим інструментом для досягнення цих цілей та забезпечення успіху бізнесу в епоху цифрових можливостей.

Завдання полягає у створенні конкурентоспроможної системи переважно завдяки доданню інноваційного функціоналу та розширення можливостей для бізнесу.

Об'єкт – система управління взаємодіями з клієнтами для бізнесів

Предмет – система, що дозволяє керувати товарами бізнесу, аналізувати та працювати з замовленнями від клієнтів, аналізувати фінансові транзакції та процеси, а також використовувати сервіси математичних моделей штучного інтелекту для роботи з даними та побудови графіків чи діаграм для глибокої аналітики бізнес-процесів.

Методи дослідження – проведення аналогій і відмінностей зі схожими системами, пояснення способів реалізації даної системи, моделювання системи, опис та аналіз використаних технологічних рішень, застосування системи до реального бізнес-проекту.

РОЗДІЛ 1. ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ТА ПОСТАНОВКА ЗАДАЧІ

1.1. Постановка задачі

Проект "Система управління бізнесом з аналітикою та оптимізацією на базі ШІ" має на меті розробити і впровадити інноваційну систему управління для інтернет-магазинів та різноманітних сервісів. Головною метою цього проекту є створення високоефективного та конкурентоздатного інструменту, спрямованого на оптимізацію ключових бізнес-процесів та забезпечення конкурентних переваг для користувачів.

Задачі, які стоять перед проектом, включають:

1. Розробка функціональності. Створення системи, яка включатиме в себе інструменти для ефективного керування замовленнями, акаунтами користувачів, товарним асортиментом, а також забезпечення аналітичного та оптимізаційного функціоналу. Метою є створення універсального інструменту, який дозволить користувачам ефективно керувати всіма аспектами свого бізнесу.

2. Інтеграція з ШІ-сервісами. Забезпечення можливості інтеграції з різноманітними ШІ-сервісами для аналізу продажів, оптимізації логістики та фінансової аналітики. Це дозволить користувачам отримувати глибокий інсайт в їх бізнес та приймати обгрунтовані рішення на основі аналізу даних.

3. Забезпечення безпеки. Розробка високого рівня захисту даних користувачів та підприємства є однією з найважливіших задач проекту. Ми покладаємо особливу увагу на конфіденційність та відповідність нормативам з охорони особистих даних, щоб забезпечити безпеку інформації.

4. Масштабованість. Система має бути створена таким чином, щоб легко масштабуватися та відповідати на зростаючий обсяг діяльності клієнтів. Це дозволить підприємствам з різних галузей та розмірів використовувати систему без обмежень.

5. Глобальний доступ. Забезпечення можливості доступу до системи з будь-якої точки світу за допомогою інтернет-підключення та мобільних пристроїв. Це надасть користувачам гнучкість та зручність у використанні системи.

6. Навчання та підтримка. Розробка навчальних матеріалів та надання технічної підтримки для користувачів системи. Мета полягає в тому, щоб користувачі мали доступ до необхідних ресурсів для оптимального використання системи.

Завдання проекту полягають у створенні інтелектуальної, високоефективної та інтуїтивно зрозумілої системи, яка допоможе власникам бізнесів управляти своєю діяльністю з використанням сучасних технологій та забезпечити стійкий розвиток у конкурентному середовищі. Проект визначається як амбіційний та інноваційний, спрямований на створення цінного рішення для бізнес-спільноти.

Проект також передбачає впровадження інтерактивних інструментів звітності та візуалізації даних, що допоможе користувачам легше розуміти та аналізувати інформацію. Такі інструменти стануть потужними засобами для прийняття обґрунтованих стратегічних рішень та вдосконалення ефективності діяльності. Крім того, проект передбачає розробку інтерфейсу, що відповідає сучасним стандартам щодо зручності та доступності. Інтуїтивно зрозумілий інтерфейс сприятиме швидшому навчанню користувачів та підвищить загальний комфорт використання системи.

Зусилля проекту спрямовані на створення комплексного рішення, яке задовольнить потреби найвимогливіших користувачів. Враховуючи широкий спектр функціональності та можливостей, система стане незамінним інструментом для бізнесу будь-якого масштабу та напрямку.

1.2. Аналіз існуючих рішень

На ринку існує кілька основних конкурентів, які надають системи управління бізнесом для інтернет-магазинів та сервісів. Давайте розглянемо деякі з них, їхні рівні конкуренції, а також переваги та недоліки:

1. Salesforce CRM

a. Опис: Salesforce - одна з провідних CRM-систем, яка надає широкий спектр функціоналу для управління клієнтами та продажами [1].

b. Рівень конкуренції: Високий, Salesforce володіє значним ринковим паєм і численними клієнтами у всьому світі.

c. Переваги: Розширений функціонал, велика спільнота користувачів, інтеграція з іншими бізнес-системами.

d. Недоліки: Високі витрати на впровадження та підтримку, складне налаштування.

2. Zoho CRM

a. Опис: Zoho CRM - інша популярна CRM-система з широким функціоналом для управління клієнтами та продажами [2].

b. Рівень конкуренції: Високий, особливо серед малих і середніх підприємств.

c. Переваги: Відмінна цінова доступність, інтеграція з іншими інструментами Zoho, користувачів форуми для підтримки.

d. Недоліки: Обмежена функціональність у порівнянні з Salesforce, менший ринковий досвід.

3. HubSpot CRM [3]

a. Опис: HubSpot CRM - це безкоштовна CRM-система з акцентом на інбаунд-маркетинг та продажі.

b. Рівень конкуренції: Середній, HubSpot стає популярним в середовищі стартапів та малих компаній.

c. Переваги: Безкоштовна версія, інтеграція з іншими інструментами HubSpot, простий інтерфейс.

d. Недоліки: Обмежена функціональність порівняно з іншими CRM-системами, платна версія має обмеження.

4. Bitrix24

a. Опис: Bitrix24 - це комплексна платформа для управління бізнесом, яка включає в себе CRM, управління проектами, комунікацію та інше [4].

b. Рівень конкуренції: Середній, особливо в регіоні Східної Європи та СНД.

c. Переваги: Різноманітний функціонал, можливість вибору хмарної або локальної версії, доступність для малих підприємств.

d. Недоліки: Складний інтерфейс, можливість переплати за невикористований функціонал.

Висновок*:* Ринок систем управління бізнесом має велику конкуренцію з численними гравцями. Вибір системи повинен бути обґрунтованим і залежати від потреб і можливостей конкретного бізнесу. Проект "Система управління бізнесом з аналітикою та оптимізацією на базі ШІ" повинен зосередитися на поєднанні широкого функціоналу, ефективної інтеграції з ШІ-сервісами, а також високої доступності та безпеки, щоб конкурувати на ринку та задовольняти потреби сучасних бізнесів.

1.3. Вимоги до проекту

Визначення вимог є критичним етапом у розробці системи управління бізнесом з аналітикою та оптимізацією на базі ШІ. Нижче ми детально розглянемо вимоги до проекту, розподілені на різні категорії:

1.3.1. Функціональні вимоги

Управління замовленнями:

- Створення, редагування та відстеження замовлень.
- Автоматичне підтвердження та відправлення замовлень клієнтам.
- Можливість внесення додаткових опцій та знижок до замовлень.

Управління клієнтськими акаунтами:

- Реєстрація клієнтів та збереження їхніх особистих даних.
- Відстеження історії покупок та зв'язку з клієнтами.
- Підтримка системи лояльності та програм збільшення продажів.

Управління товарами та асортиментом:

- Додавання, редагування та видалення товарів.
- Можливість класифікації товарів за категоріями та характеристиками.
- Відстеження кількості товарів на складі та автоматичне оновлення інформації.

Аналітика продажів:

- Збір та аналіз даних про продажі, включаючи звіти та графіки.
- Прогнозування попиту на товари та послуги.
- Відстеження ефективності маркетингових кампаній.

Оптимізація логістики та постачання:

- Автоматичне управління запасами та поповнення складських запасів.
- Маршрутизація доставки та відстеження руху товарів.
- Оптимізація постачальницьких ланцюгів.

Фінансова аналітика:

- Ведення обліку фінансових операцій та доходів.
- Генерація звітів про прибуток та витрати.
- Підтримка різних валют та обмінних курсів.

1.3.2. Інтерфейсні вимоги

Зручний інтерфейс користувача:

- Інтуїтивно зрозумілий інтерфейс, доступний для користувачів з різним рівнем технічної компетентності.
- Можливість персоналізації інтерфейсу та налаштувань.

Мобільна сумісність:

- Доступ до системи з мобільних пристроїв (смартфони, планшети).
- Адаптивний дизайн для різних розмірів екранів.

Мультиплатформенність:

- Підтримка різних операційних систем (Windows, macOS, Linux) та браузерів.

1.3.3. Безпека і захист даних

Захист особистих даних:

- Забезпечення конфіденційності особистих даних клієнтів та бізнес-інформації.

Захист від кібератак:

- Виявлення та запобігання кібератакам, включаючи захист від вторгнень та вірусів.

Резервне копіювання та відновлення:

- Регулярне резервне копіювання даних і можливість їхнього відновлення в разі аварій.

1.3.4. Інтеграція та масштабованість

Інтеграція з іншими системами:

- Можливість легко інтегрувати систему з іншими програмними рішеннями, включаючи бухгалтерські програми та електронну комерцію.

Масштабованість:

- Здатність системи масштабуватися для відповіді на зростаючий обсяг діяльності клієнтів без втрати продуктивності.

1.3.5. Підтримка та навчання

Технічна підтримка:

- Надання користувачам доступу до технічної підтримки та консультантів для вирішення питань та вирішення проблем.

Навчання користувачів:

- Постачання навчальних матеріалів та онлайн-курсів для користувачів.

Оновлення та підтримка продукту:

- Регулярні оновлення системи для виправлення помилок та додавання нових функцій.

Вимоги до проекту обґрунтовані і визначають специфікації для розробки системи управління бізнесом. Дотримання цих вимог допоможе забезпечити високу якість продукту та задовольнення потреб користувачів.

1.4. Технології для створення ПЗ CRM системи

Побудова ефективної CRM системи для управління бізнесом з використанням сучасних технологій є важливою складовою успіху проекту. У даному розділі ми розглянемо ключові технології, які будуть використовуватися для розробки нашої CRM системи.

Використання стеку React для фронтенду, NestJS для бекенду та MongoDB для бази даних є найкращим вибором для нашого проекту з декількох важливих причин.

1. Висока продуктивність: React - це потужна бібліотека для розробки користувацьких інтерфейсів, яка дозволяє створювати ефективні та швидкі веб-додатки. NestJS і MongoDB також відомі своєю продуктивністю та швидкодією.

2. Масштабованість: Всі три технології легко масштабуються, що дозволить нашій CRM-системі зростати разом з розвитком бізнесу клієнтів.

3. Спільнота та підтримка: React та NestJS мають велику спільноту розробників і регулярно оновлюються, що забезпечує актуальність та безпеку наших додатків. MongoDB також має активну спільноту та широкі можливості для розширення.

4. Гнучкість та модульність: NestJS базується на TypeScript і пропонує структурований та модульний підхід до розробки серверної частини. React надає можливість створювати компоненти, що легко перевикористовувати. MongoDB дозволяє гнучко зберігати дані в документах, що підходить для потреб CRM-системи.

5. Інтеграція з Штучним Інтелектом (ШІ): React та NestJS можуть легко інтегруватися з різноманітними ШІ-сервісами, забезпечуючи можливість використовувати інтелектуальний аналіз даних та оптимізацію процесів.

Цей стек технологій гармонійно поєднує високу продуктивність, масштабованість, гнучкість та підтримку спільноти, що робить його найкращим

вибором для створення нашого інноваційного проекту CRM-системи з аналітикою та оптимізацією. Більш детальний опис кожної технології:

1. ***Фронтенд:***

a. React: React - це одна з найпопулярніших бібліотек для розробки інтерфейсу користувача. Вона відома своєю ефективністю, компонентною архітектурою та зручною системою керування станом додатку.

b. Typescript: Typescript надасть нам можливість сильно типізувати наш код, що полегшить відлагодження та підтримку додатку на великій протязій терміну.

c. Redux: Redux - це бібліотека для керування станом додатку. Вона дозволяє ефективно організовувати стан додатку та забезпечувати прозорий потік даних.

d. Material-UI: Material-UI надасть нам багато готових компонентів та стилів, що спростять розробку користувацького інтерфейсу та забезпечать його сучасним та привабливим виглядом.

2. ***Бекенд:***

a. Node.js: Node.js - це середовище виконання JavaScript на серверній стороні. Воно відоме своєю швидкістю та легкістю використання.

b. NestJS: NestJS - це фреймворк для розробки серверних додатків на Node.js, який надає структуру та організацію для проекту. Він сприяє створенню масштабованих та добре організованих додатків.

c. MongoDB: MongoDB - це NoSQL база даних, яка дозволяє зберігати та опрацьовувати дані у форматі JSON. Вона відома своєю швидкістю та гнучкістю в роботі з даними.

3. ***Інші технології та інструменти:***

a. Docker: Docker дозволяє контейнеризувати наш додаток, що спростить розгортання та масштабування.

b. Git: Git - це система контролю версій, яка дозволяє спільно працювати над кодом та відстежувати зміни.

c. Swagger: Swagger дозволить автоматично генерувати документацію для нашого API, що полегшить розуміння та використання API.

d. Webpack: Webpack - це інструмент для збирання та оптимізації нашого фронтенд-коду.

В результаті матимемо наступну структуру проєкту:

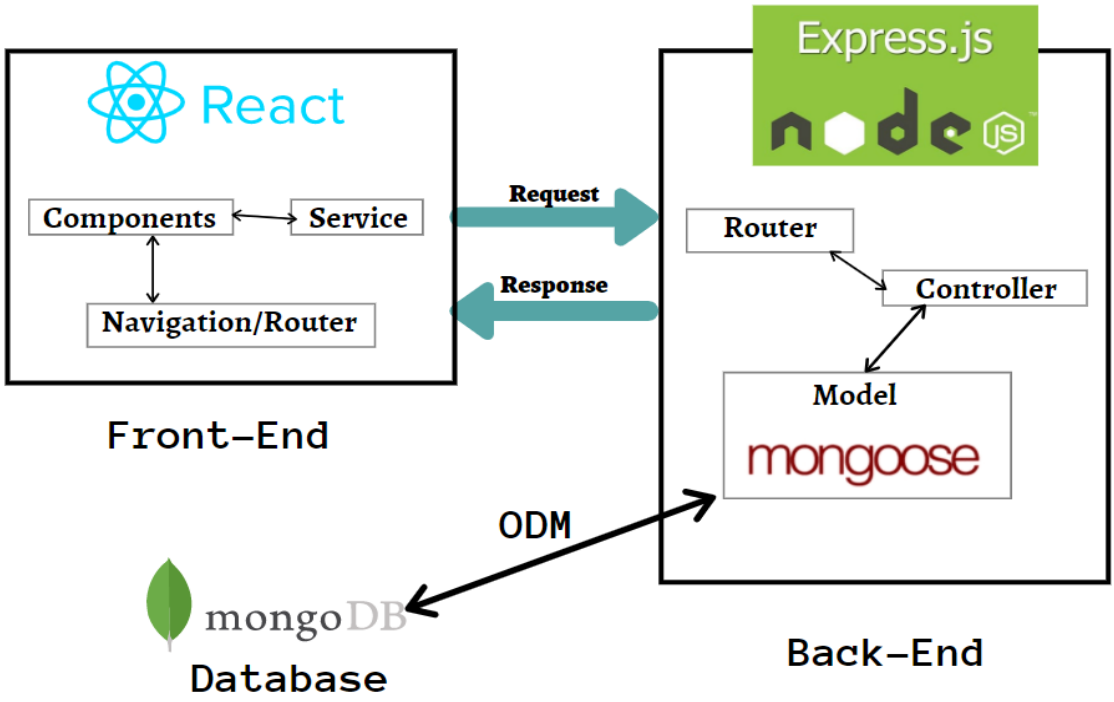

Рис.1.1. Схема архітектури клієнт-серверного додатку

Використання цих технологій дозволить нам створити сучасну, ефективну та масштабовану CRM систему, яка відповідає потребам користувачів і забезпечує високий рівень якості та безпеки. Правильний вибір технологій є ключовим фактором у успішному розвитку проекту та досягненні поставлених цілей.

1.5. Задача проекту

Метою проекту "Система управління бізнесом з аналітикою та оптимізацією на базі ШІ" є створення і впровадження інноваційної та інтегрованої інформаційної системи, яка надасть підприємствам та підприємцям набір потужних інструментів для ефективного управління різними аспектами їхньої діяльності. Проект розглядається

як відповідь на зростаючу потребу ринку в комплексних рішеннях, які поєднують у собі управління замовленнями, аналітику продажів, оптимізацію логістики, фінансовий облік та використання сучасних ШІ-технологій.

Задачі, які стоять перед проектом, можна розглядати наступним чином:

1. **Створення комплексного рішення:**

Основною задачею проекту є розробка та впровадження універсальної інформаційної системи, яка охоплює всі ключові аспекти управління бізнесом. Від управління замовленнями та клієнтськими акаунтами до аналітики продажів та оптимізації логістики - це комплексне рішення має забезпечити повний контроль та ефективність у всіх сферах діяльності.

2. Інтеграція з ШІ-сервісами:

Для забезпечення передового аналізу даних, автоматизації прийняття рішень та прогнозування попиту, система має бути інтегрована з сучасними ШІ-сервісами. Це відкриває нові можливості для користувачів у використанні машинного навчання, обробки природної мови, аналізу великих обсягів даних та інших передових технологій.

3. Забезпечення безпеки та конфіденційності:

Враховуючи важливість даних користувачів та бізнес-інформації, проект має гарантувати високий рівень захисту даних та конфіденційності. Шифрування, автентифікація та моніторинг безпеки - це складові, які мають бути на вищому рівні.

4. Підвищення продуктивності бізнесу:

Однією з ключових задач є оптимізація бізнес-процесів, що дозволить підприємствам підвищити ефективність та продуктивність. Автоматизація рутинних операцій, керування запасами, ефективне використання ресурсів - це завдання, які стоять перед системою.

5. Аналітика для прийняття рішень:

Система має надавати користувачам інструменти для збору, аналізу та візуалізації даних. Це допоможе власникам бізнесу приймати обгрунтовані рішення на основі великої кількості інформації. Аналіз продажів, звіти про фінансову

продуктивність та прогнозування попиту - це лише кілька аспектів аналітики, які може надати система.

6. Масштабованість і гнучкість:

Система повинна бути готовою відповідати на зростаючий обсяг діяльності клієнтів та бути гнучкою для врахування індивідуальних потреб кожного бізнесу. Масштабування та адаптація до змін вимог - це важливі характеристики системи.

7. Підтримка і навчання:

Забезпечення доступу до технічної підтримки та навчальних ресурсів допоможе користувачам максимально використовувати потенціал системи. Навчальні матеріали, онлайн-курси та консультації - це компоненти, які допоможуть користувачам вивчити та оптимально використовувати систему.

8. Оновлення і розвиток продукту:

Постійне оновлення та розвиток системи є важливим завданням для того, щоб вона залишалася актуальною та відповідала сучасним вимогам ринку. Впровадження нових функцій, виправлення помилок та адаптація до змін - це процес, який повинен продовжуватися протягом усього життєвого циклу системи.

Загалом, проект "Система управління бізнесом з аналітикою та оптимізацією на базі ШІ" визначається як інноваційний та амбіційний, спрямований на створення комплексного інструменту для підприємств та підприємців. Перед ним стоїть завдання надати користувачам засоби для ефективного управління, аналізу даних та прийняття обгрунтованих рішень у сучасному конкурентному бізнес-середовищі.

ВИСНОВОК ДО РОЗДІЛУ 1

У даному розділі ми ретельно проаналізували і представили концепцію проекту "Система управління бізнесом з аналітикою та оптимізацією на базі ШІ". В процесі створення цього плану я вклав всіх можливих зусиль для того, щоб детально розкрити всі аспекти проекту та довести його актуальність та цінність.

Актуальність проекту обумовлена швидкими змінами в сучасному бізнес-середовищі та зростаючими вимогами до ефективного управління. Інтернет-магазини та інші бізнеси потребують інтегрованих та інноваційних рішень, які поєднують у собі управління замовленнями, аналітику продажів, оптимізацію логістики, фінансовий облік та використання сучасних ШІ-технологій.

Задачі, які стоять перед проектом, можна розглядати наступним чином:

- Створення комплексного рішення
- Інтеграція з ШІ-сервісами
- Забезпечення безпеки та конфіденційності:
- Підвищення продуктивності бізнесу
- Аналітика для прийняття рішень
- Масштабованість і гнучкість
- Підтримка і навчання
- Оновлення і розвиток продукту

Також було визначено та описано стек технологій, який буде найоптимальнішим і збалансованим рішенням для створення данної системи.

РОЗДІЛ 2. ПРОЦЕС РОЗРОБКИ СИСТЕМИ УПРАВЛІННЯ БІЗНЕСОМ І ВІДНОСИНАМИ З КЛІЄНТАМИ

2.1. Вибір, аналіз, і потреби клієнтської частини бізнесу

Для даного проєкту системи управління бізнесом я обрав інтернет-магазин жіночого одягу. Проект вже має готову клієнтську частину, товари, що продаються, користувачів, тощо. Даний інтернет-магазин наразі потребує власну CRM-систему, яка буде покривати усі необхідні задачі, що постають перед бізнесом. А саме:

1. Бачення загальної динаміки: CRM-система повинна надавати можливість детального аналізу та візуалізації статистики щодо різних аспектів бізнесу, включаючи продажі, конверсію, витрати та доходи. Це допоможе виробити стратегічні рішення на основі цих даних.

2. Управління товарами: Окрім вже згаданих параметрів, система повинна мати можливість автоматично оновлювати інформацію про товари, включаючи нові надходження та зміни цін. На додачу, система повинна підтримувати керування запасами товарів, щоб уникнути нестачі або перевищення їх необхідної кількостів.

3. Аналіз користувацьких даних: CRM-система повинна бути здатна аналізувати дані користувачів, враховуючи їхню історію покупок, взаємодію з сайтом та інші параметри. Ця інформація допоможе нам створити більш персоналізований досвід для наших клієнтів і підвищити їхню лояльність.

4. Управління та моніторинг замовлень: Важливо мати можливість в режимі реального часу відстежувати стан замовлень, а також здійснювати їх сортування та фільтрацію залежно від різних критеріїв, таких як статус, дата і інші. Покращена система керування замовленнями сприятиме оптимізації процесу обробки та виконання замовлень.

5. Надання доступу працівникам: CRM-система повинна підтримувати індивідуальні облікові записи для різних працівників і надавати можливість делегувати завдання та повноваження між менеджерами. Контроль доступу і створення груп користувачів буде важливим аспектом для ефективного управління.

6. Аналіз фінансових звітів: CRM-система повинна забезпечувати можливість генерувати фінансові звіти, які відображають прибуток, витрати та інші фінансові показники. Ця інформація допоможе бізнесу зробити обґрунтовані фінансові рішення та планування на майбутнє.

Ці розширені вимоги для CRM-системи допоможуть забезпечити більш комплексний та ефективний управління бізнесом і підвищити конкурентоспроможність інтернет-магазину.

Інтернет-магазин складається із наступних сторінок:

- Головна сторінка
- “Магазин” із представленими товарами за обраною категорією
- “Про нас”
- Фотожурнал минулих колекцій
- Контакти з формою зворотнього зв’язку
- Сторінки з інформацією по доставці й оплаті, з описаною політикою конфіденційності, та сторінка публічного договору
- Сторінка конкретного товару з можливістю переглянути фото, опис товару, можливістю вибору розміру та додання в кошик для подальшого оформлення замовлення
- Кошик
- Форма оформлення замовлення
- Та сторінка, що надається платіжною системою, де відбується оплата товару.

Усі взаємодії клієнтської частини та CRM відображаються на сторінці “Магазин”, такі як наявність товару, ціна, назва; на сторінці конкретної речі - стосовно будь-якого параметру товару; та кошику, де буде змінюватись ціна товару,

застосовуватись знижка та промокод. Також буде додано управління через систему керування взаємовідносинами з клієнтами зі сторінками: Головна (завдяки завантаженню нових зображень на головну сторінку), “Про нас” (через зміну тексту в даному розділі), сторінок з інформацією по доставці й оплаті, з описаною політикою конфіденційності, та сторінка публічного договору.

Окрім управління безпосередньо клієнтською частиною інтернет-магазину буде створено розділі з управлінням та аналізом внутрішньо бізнесових параметрів.

2.2. Дизайн системи

Придумувати інноваційний дизайн CRM-системи немає сенсу, адже, на мій погляд, це суто практична система, яка має, насамперед, добре виконувати свою роботу, і не має на меті залучення клієнтів. Тож я вирішив взяти приклад з уже існуючих аналогічних систем управління або з вже готових дизайнів, та ось приклади наступних:

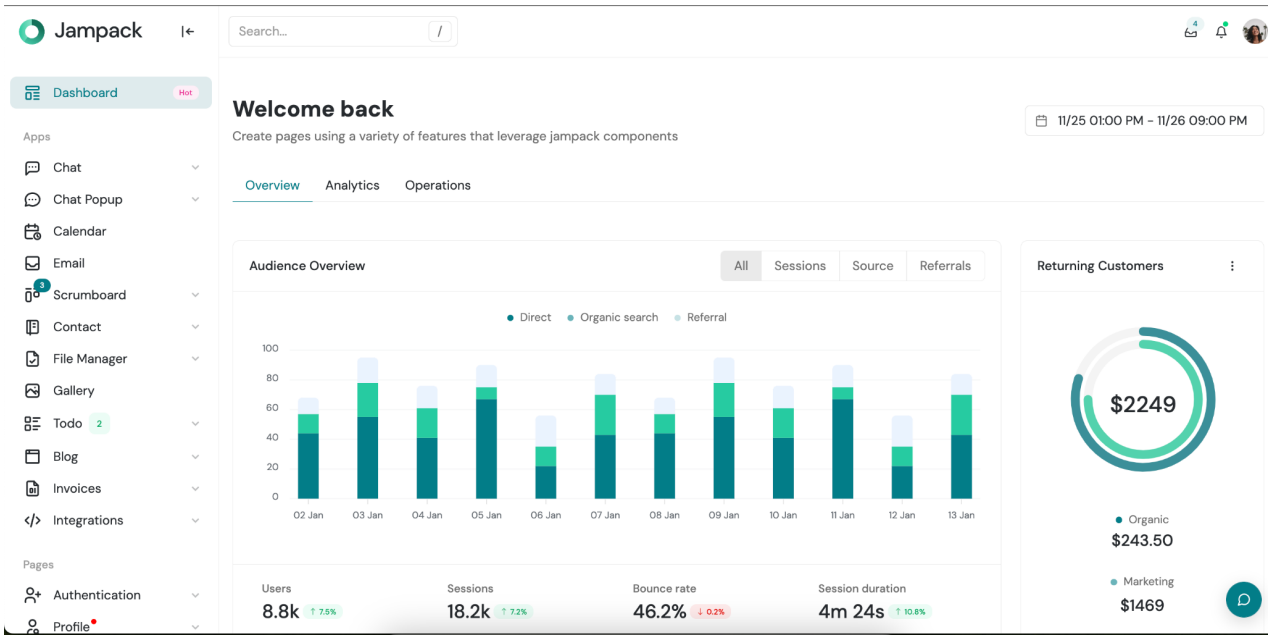

Рис.2.1. Варіант дизайну системи 1

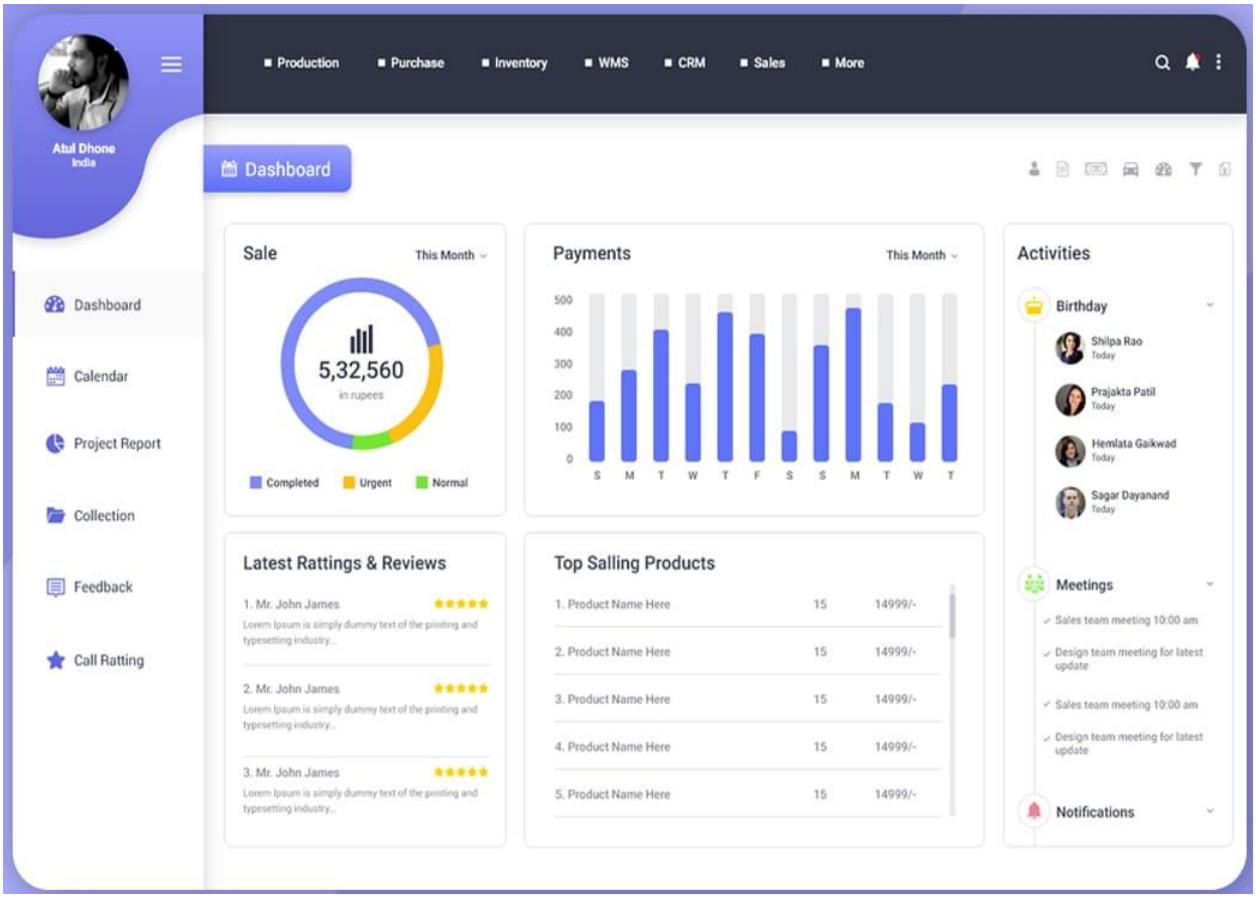

Рис.2.2. Варіант дизайну системи 2

Як можна побачити, усі вони, в більшості, мають дуже простий і зрозумілий дизайн – це бічна панель з меню, звідки відбувається перехід між сторінками та розділами системи, ну й, звичайно ж, логотип. Бічну панель завжди видно, вона займає до 20% екрану сторінки. З правої ж сторони бачимо ще дві складові, перша – це “шапка” сайту CRM, яку також завжди видно, вона дозволяє відкривати додаткові модальні та діалогові вікна, а також звідти відбувається перехід на сторінку профілю активного користувача, у розділ налаштувань, вікна повідомлень, тощо; і остання, та ключова, секція системи – це самі сторінки системи управління взаємодією з клієнтами, де відбувається керування, моніторинг, аналіз усіх складових бізнесу, які ми створимо.

Тож систему буде створено за схожим, вже перевіреним, дизайном.

2.3. Створення каркасу системи

Тепер давайте перейдемо до процесу створення каркасу проекту, спираючись на обрані технології та інструменти.

2.3.1. Ініціалізація React клієнтської частини

Почнемо з ініціалізації React клієнтської частини нашого проекту. Для цього використовуємо команду `npx create-react-app client`, яка створює новий проект з використанням стандартного шаблону React. Після цього переходимо в папку `client`, де будемо розробляти клієнтську частину.

Однак перед початком розробки, нам потрібно підключити деякі додаткові бібліотеки. Використовуючи `npm`, ми встановлюємо бібліотеки Typescript (`npm install typescript`), Material UI (`npm install @mui/material @mui/icons`), та Redux (`npm install redux react-redux @reduxjs/toolkit`).

Після встановлення необхідних бібліотек, ми можемо розпочати створення компонентів, роботу зі станом додатка та розробку користувацького інтерфейсу з використанням Material UI. Додавши компоненти та розробивши деяку логіку, наша клієнтська частина стає готовою до використання.

2.3.2. Ініціалізація Nest.js серверної частини

Після успішної ініціалізації React клієнтської частини, переходимо до ініціалізації серверної частини нашого проекту з використанням Nest.js. Найперше, створюємо папку для сервера за допомогою команди `mkdir server`, а потім переходимо в цю папку командою `cd server`.

Далі, ми встановлюємо Nest.js, використовуючи команду `npm install -g @nestjs/cli`, що дозволяє нам створити новий проект на основі Nest.js. Створюємо новий проект за допомогою `nest new server`, де `server` - назва нашого серверу.

Після створення проекту, ми встановлюємо додаткові бібліотеки, такі як `@nestjs/mongoose` для роботи з MongoDB, `@nestjs/passport` для автентифікації, і `@nestjs/config` для роботи з конфігурацією. Ці бібліотеки допомагають нам створити потужний та гнучкий сервер для нашого CRM-рішення.

Завдяки Nest.js ми можемо створити контролери та сервіси для обробки запитів, налаштувати маршрутизацію та використовувати Middleware для обробки запитів та відповідей. Надійний фреймворк Nest.js допомагає організувати наш сервер і забезпечити його ефективну роботу.

2.3.3. Робота з базою даних MongoDB та MongoDB Atlas

Щоб реалізувати зберігання даних, ми використовуємо MongoDB - NoSQL базу даних. Для роботи з нею ми встановлюємо `mongoose`, бібліотеку, яка надає ORM (Object-Relational Mapping) для MongoDB.

MongoDB Atlas - це хмарна версія MongoDB, яку ми обрали для забезпечення безпеки, доступності та масштабованості нашої бази даних. Для роботи з MongoDB Atlas, ми створюємо обліковий запис, створюємо проект та підключаємо нашу серверну частину до бази даних через URI-рядок підключення.

Разом із `@nestjs/mongoose`, ми створюємо моделі для сутностей в нашому додатку та налаштовуємо взаємодію з базою даних. MongoDB Atlas забезпечує надійне зберігання даних та автоматичне резервне копіювання, що важливо для забезпечення надійності та безпеки даних користувачів.

На початку роботи над проектом наша структура файлів виглядає наступним чином:

```
project-root/
  ├─ client/
  │   ├─ public/
  │   │   └─ index.html
  │   ├─ src/
```

```
│   │   ├─ components/
│   │   │   ├─ App.tsx
│   │   │   ├─ ...
│   │   ├─ reducers/
│   │   │   ├─ index.ts
│   │   │   ├─ ...
│   │   └─ ...
├─ server/
│   ├─ src/
│   │   ├─ controllers/
│   │   │   ├─ ...
│   │   ├─ modules/
│   │   │   ├─ ...
│   │   ├─ app.module.ts
│   │   ├─ main.ts
├─ ...
```

1. Client: Ця директорія містить клієнтську частину нашого проекту, яка відповідає за візуальний інтерфейс та взаємодію з користувачами.

a. public: Тут розміщується вихідний HTML файл, який відображає основну структуру сторінки.

b. src: В цій папці розміщені компоненти, редуктори, екшени, та інші файлы, відповідні за клієнтську частину.

2. Server: Директорія серверної частини, яка відповідає за обробку запитів та бізнес-логіку.

a. controllers: Тут розміщені контролери, які обробляють HTTP-запити.

b. modules: В цій папці розміщені модулі, які містять сервіси та роблять всяку логіку.

c. app.module.ts: Головний модуль Nest.js додатка, де проводиться налаштування всього додатка.

d. main.ts: Головний файл, який запускає сервер.

Із структурою файлів відповідно пов'язані обов'язки кожної частини системи:

1. Client (Клієнтська частина):

a. components: У цій папці розміщені React компоненти, які створюють інтерфейс користувача.

b. reducers: Редуктори Redux відповідають за управління станом клієнтської частини.

2. Server (Серверна частина):

a. controllers: Контролери обробляють HTTP-запити та взаємодіють з відповідними сервісами.

b. modules: Модулі містять сервіси, які виконують різні завдання, такі як робота з базою даних та бізнес-логіка.

c. app.module.ts: Головний модуль, де налаштовується вся конфігурація додатка та підключення модулів.

d. main.ts: Цей файл відповідає за запуск сервера.

Завдяки такому розподілу обов'язків між клієнтською та серверною частинами, ми забезпечуємо модульність та зручність розробки проекту. Кожна частина відповідає за свою функціональність та легко підтримується та розширюється в майбутньому. Завдяки описаним ініціалізаційним крокам, ми створюємо надійну основу для нашого CRM-проекту. React клієнтська частина, Nest.js серверна частина та MongoDB Atlas база даних - всі ці компоненти інтегруються разом, дозволяючи нам створити потужне та функціональне CRM-рішення для нашого інтернет-магазину. У наступних розділах ми продовжимо розглядати розробку та розширення нашого проекту, надаючи користувачам широкий функціонал та зручний досвід користування.

2.3.4. Налаштування базової файлової структури React-додатку

На верхньому рівні файлової структури після ініціалізації додатку за допомогою Command Line Interface “Create React App” було створено наступні директорії:

- .git: Директорія .git - це одна з найважливіших частин кожного репозиторію Git і відповідає за зберігання інформації, пов'язаної з репозиторієм та його історією. Основні складові директорії .git та їх призначення:

a. objects: Ця директорія містить стислі файли, які представляють коміти, дерева та файли, які були збережені в репозиторії. Кожен коміт із всіма змінами зберігається як об'єкт в цій директорії.

b. refs: В цій директорії зберігаються вказівники на коміти та гілки. Наприклад, файли в refs/heads представляють гілки, а файли в refs/tags - теги.

c. HEAD: Цей файл вказує на поточний коміт або гілку, на якій знаходиться ваш репозиторій. Він допомагає Git знаходити поточний стан.

d. config: Файл конфігурації репозиторію, який містить налаштування, такі як ім'я та email користувача, а також інші параметри.

e. hooks: В цій директорії можна розміщати сценарії-гачки, які виконуються перед або після певних подій в Git, таких як коміт чи отримання пул-запиту.

f. index: Цей файл представляє індекс (staging area), де зберігаються зміни, готові до коміту. Він містить вказівки до змінених та нових файлів.

g. logs: В цій директорії зберігаються журнали подій, такі як коміти, гілкові зміни тощо.

h. hooks: Директорія для власних сценаріїв-гачків, які виконуються в певних подіях Git.

i. info: Загальна інформація про репозиторій.

Директорія .git дозволяє Git відстежувати зміни в проекті, забезпечуючи історію, можливість відновлення попередніх станів та багато інших функцій. Не

рекомендується редагувати файли в .git вручну, якщо ви не розумієте їхньої структури. Ви повинні користуватися командами Git або інтерфейсами Git-клієнтів для управління репозиторієм.

- node_modules: Директорія node_modules - це спеціальна директорія у проектах, що використовують Node.js і зазвичай пов'язана з управлінням залежностями та пакетами, які використовуються в проекті. Ось докладні відповіді на ваші запитання:

a. Що таке node_modules? node_modules - це директорія, де зберігаються всі залежності і пакети (модулі), які використовуються в проекті, розроблених з використанням Node.js. Це може включати в себе бібліотеки, фреймворки, плагіни, та інші ресурси, які потрібні для правильної роботи проекту.

b. Для чого використовується node_modules? node_modules використовується для організації та управління залежностями проекту. Коли ви використовуєте Node.js для розробки, ви часто підключаєте сторонні бібліотеки та пакети до свого проекту. node_modules забезпечує структуроване зберігання цих залежностей та дозволяє проекту легко знаходити і використовувати їх.

c. Як працює node_modules? Після встановлення Node.js та ініціалізації проекту з package.json, ви можете використовувати npm або yarn (інші менеджери пакетів) для встановлення залежностей. При цьому, відповідні пакети завантажуються і зберігаються в директорії node_modules. Коли ви запускаєте свій проект, Node.js автоматично знаходить і використовує ці залежності, дозволяючи вам використовувати їх функціональність в своєму коді.

Отже, node_modules - це важлива частина проекту, яка спрощує управління залежностями та допомагає забезпечити, що ваш проект використовує необхідні бібліотеки та пакети для його правильної роботи.

- public: Директорія public у React додатку - це спеціальна директорія, яка використовується для зберігання статичних ресурсів, таких як HTML-файли, зображення, шрифти та інші файли, які повинні бути доступні з клієнтської сторони додатку. Ось кілька ключових аспектів щодо директорії public в React:

a. Головний HTML-файл: У директорії public зазвичай розміщується головний HTML-файл, який буде відображатися в браузері при завантаженні додатку. Цей файл містить кореневий елемент, до якого приєднується весь React-код під час ініціалізації.

b. Статичні ресурси: У public можна зберігати різні статичні ресурси, такі як зображення, шрифти, CSS файли тощо. Ці ресурси можуть бути використані в компонентах React і завантажені клієнтом безпосередньо з цієї директорії.

c. Публічно доступні файли: Все, що розміщується в директорії public, доступне для прямого завантаження клієнтами. Це дозволяє створювати посилання на ці файли і використовувати їх з клієнтського коду без необхідності обробки сервером.

d. Параметри середовища: В директорії public можуть бути розміщені файли для налаштування параметрів середовища, які використовуються під час розробки та збірки додатку.

Загалом, директорія public грає важливу роль у React додатку, забезпечуючи доступ до статичних ресурсів та визначаючи початковий HTML-файл, який відображається в браузері. Вона допомагає структурувати та організувати роботу зі статичними ресурсами та публічно доступними файлами у вашому додатку.

- src: Директорія src у React додатку є однією з найважливіших і використовується для зберігання вихідного коду вашого додатку. Ось докладніша інформація про цю директорію:

a. Головна точка входу: В директорії src зазвичай міститься головний файл, який використовується як точка входу у вашому додатку. Цей файл може називатися, наприклад, index.js або App.js. Він ініціює запуск вашого React додатку та рендерить кореневий компонент, який буде відображений в браузері.

b. Компоненти React: У директорії src зазвичай розміщуються всі компоненти React, які ви створюєте для вашого додатку. Це включає в себе файли JavaScript або JSX, які містять компоненти, їх стилі, інші ресурси, пов'язані з інтерфейсом користувача.

c. Модулі та функціонал: Усі файли, які містять логіку вашого додатку, також зазвичай розміщуються в директорії src. Це можуть бути модулі, функції, сервіси та інші складові, які забезпечують роботу додатку.

d. Ресурси та зображення: Якщо у вас є статичні ресурси, такі як зображення, JSON файли чи інші файли, вони також можуть бути розміщені в директорії src, або створені окремі піддиректорії для кращої організації.

e. Стилі: Файли CSS, SASS, SCSS, або інші файли стилів, які використовуються для оформлення вашого додатку, також зазвичай знаходяться в директорії src.

f. Роутинг та стейт-менеджмент: Якщо ви використовуєте бібліотеки для роутингу (наприклад, React Router) або стейт-менеджменту (наприклад, Redux), файли, пов'язані з цими бібліотеками, також можуть бути розміщені в директорії src.

g. Тести та інші ресурси: Якщо ви пишете тести для вашого додатку або маєте інші ресурси, вони також можуть бути організовані у директорії src або в окремих піддиректоріях.

Загалом, директорія src служить як основне місце для розробки вашого React додатку. Весь вихідний код та ресурси повинні бути структуровані та організовані в цій директорії для зручності розробки та управління проектом.

А також файли:

- .gitignore: Файл .gitignore є важливою складовою керування версіями за допомогою Git. Цей файл використовується для визначення файлів та директорій, які повинні бути проігноровані Git під час збору інформації для коміту. Ось докладніше про файл .gitignore:

a. Ігноровані файли та директорії: Ви можете вказати у файлі .gitignore файли, директорії та шаблони, які не повинні бути включені в репозиторій Git. Наприклад, це можуть бути тимчасові файли, які створюються під час розробки, файли налаштувань, файли журналів, файли залежностей тощо.

b. Зручність розробки: Використовуючи .gitignore, ви можете полегшити процес розробки, оскільки не потрібно включати незначущі файли в коміти. Це

допомагає підтримувати репозиторій чистим та зменшує обсяг даних, які потрібно синхронізувати між розробниками.

c. Конфіденційність і безпека: Ви можете використовувати .gitignore, щоб уникнути включення конфіденційних або безпекових даних, таких як паролі, ключі API, файли з налаштуваннями, в репозиторій. Це допомагає утримувати ці дані в приватності та уникає можливих порушень безпеки.

d. Автоматизація та стандартизація: Файл .gitignore може бути використаний для стандартизації ігнорованих файлів у вашому проекті. Ви можете використовувати загальні шаблони для різних типів проектів та мов програмування.

Приклад вмісту файлу .gitignore:

```
# Ігнорувати всі файли з розширенням .log
*.log

# Ігнорувати всі файли у директорії temp/
/temp/*

# Ігнорувати файли з налаштуваннями
config.json

# Ігнорувати всі файли у директорії node_modules/
/node_modules/

# Ігнорувати файли, які містять паролі
*password*
```

Рис.2.3. Вміст файлу .gitignore

Файл .gitignore дозволяє вам керувати тим, які файли та директорії будуть включені або виключені з репозиторію Git, що робить його потужним інструментом для керування версіями та збереження конфіденції ваших проектів.

- README.md: цей файл є стандартним файлом документації для проектів, особливо популярний у репозиторіях Git. Він призначений для надання короткого та

зрозумілого опису проекту, його функціональності та інструкцій щодо його використання.

- tsconfig.json: використовується для налаштування компілятора TypeScript та визначення параметрів компіляції для проекту

- package.json: Файл package.json є основним конфігураційним файлом для Node.js проектів, який містить інформацію про проект та його залежності.

- package-lock.json: створюється автоматично при встановленні пакетів через npm і містить докладну інформацію про версії та хеш-суми кожного встановленого пакета та його залежностей. Це гарантує стабільність та репродукованість залежностей при розгортанні проекту на інших системах або у різних середовищах. Коли ви спільно працюєте над проектом або розгортаєте його на інших системах, package-lock.json допомагає забезпечити, що всі залежності встановлюються відповідно до конкретних версій, зазначених у файлі, та не виникають конфлікти між різними версіями.

Після чого вже в ручному режимі було додано наступні файли:

- .env: Файл .env - це текстовий файл, який містить конфіденційні змінні середовища для додатків. Ці змінні використовуються для зберігання конфіденційної інформації, такої як ключі API, паролі або інші параметри, які не повинні бути відкриті для громадськості. Файл .env зазвичай використовується в розробці програмного забезпечення для зберігання цих конфіденційних даних, і зазвичай не публікується в репозиторіях коду для забезпечення безпеки. Примітка: Файл .env має специфічний формат, де змінні середовища визначаються у формі "ЗМІННА=ЗНАЧЕННЯ", кожен запис в новому рядку. Наприклад:

SECRET_KEY=my_secret_key321

DATABASE_URL=mongodb://login:password123@localhost:5000/mydatabase

API_KEY=blabla992255

- .eslintrc.json та .prettierrc: Файли .eslintrc.json і .prettierrc використовуються для налаштування правил лінтера та форматера відповідно для проекту на JavaScript.

Файл .eslintrc.json (ESLint Configuration): Це конфігураційний файл для ESLint - інструмента для аналізу та виявлення помилок в коді JavaScript. У цьому файлі ви визначаєте правила, які визначають стиль і які допомагають уникнути потенційних помилок в коді. Він може містити такі налаштування, як встановлені правила, ігноровані файли, розширення файлів та інші параметри. Файл .prettierrc (Prettier Configuration): Це конфігураційний файл для Prettier - інструмента для автоматичного форматування коду. Ви визначаєте у цьому файлі стилі форматування, такі як розміри відступів, розміри та використання одинарних чи подвійних лапок для рядків тощо. Prettier допомагає забезпечити однорідний стиль коду в проекті.

- jest.config.ts: це конфігураційний файл для Jest, який є популярним фреймворком для тестування JavaScript-коду. Цей файл дозволяє вам налаштувати параметри виконання тестів та визначити, як Jest повинен проводити тестування вашого проекту.

2.4. Робота над інтерфейсом основних розділів CRM-системи

2.4.1. Налаштування файлів директорії public

Для початку згорегуємо кореневий HTML файл, прибравши з нього непотрібні теги та атрибути, а також додамо налаштування, що будуть нам необхідні, назву і логотип сайту, посилання на зовнішні стилі, тощо. Кінцевий варіант коду кореневого HTML-файлу index.html виглядає наступним чином:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>FRÉO Admin</title>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-scale=1.0, user-scalable=0" />
    <meta name="theme-color" content="white" />
```

```
    <meta name="title" content="FRÉO Admin" />
    <meta name="description" content="FREO CRM System" />
    <meta property="og:image" content="%PUBLIC_URL%/Logo.png" />
    <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
    <link rel="preconnect" href="https://fonts.googleapis.com" />
    <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin />
    <link
href="https://fonts.googleapis.com/css2?family=Montserrat:wght@300;400;700&display=
swap" rel="stylesheet" />
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
  </body>
</html>
```

А також додамо в ту ж директорію наступні файли:

- .htaccess - файл із налаштуваннями, які нам знадобляться під час розміщення на хостингу

- замінено файл з логотипом сайту favicon.ico

- а також додано директорію locales/ в якій зберігатимуться файли у форматі JSON в яких будуть прописані усі слова, які є на сайті, на різних мовах для роботи системи перекладу

Ось такий вигляд має файлова структура у вікні редактора коду:

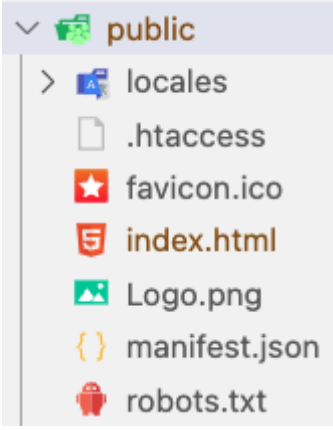

Рис.2.4. Структура клієнтської директорії public/

Наразі роботу з директорією public/ завершено.

2.4.2. Налаштування кореневої директорії src

Файлова структура директоріїї src складатиметься з файлів:

- index.tsx: Файл index.tsx на сайті з React використовується як основний файл, який вказує, як весь React-додаток повинен бути зібраний та рендеритися в браузері. Основні функції файлу index.tsx включають в себе:

a. Запуск React-додатка: В цьому файлі зазвичай ініціалізується React-додаток, ініціалізується кореневий компонент і встановлюються параметри для рендерингу.

b. Рендеринг в DOM: Файл index.tsx зазвичай містить виклик ReactDOM.render, де вказується кореневий компонент, який повинен бути відображений, і DOM-елемент, в який цей компонент повинен бути вбудований.

c. Підключення CSS та інших ресурсів: Також цей файл може включати імпорти CSS-файлів або інших ресурсів, необхідних для стилізації та роботи React-додатку.

d. Налаштування середовища: Він також може містити налаштування середовища для розробки, такі як підключення гарячої перезагрузки (Hot Module Replacement) або інших розширень для зручної розробки та тестування.

Отже, файл index.tsx - це головний файл, який визначає точку входу та налаштовує рендеринг React-додатка в браузері.

- index.css: Файл index.css на сайті з React використовується для визначення стилів і зовнішнього вигляду вашого React-додатку. Цей файл містить CSS-правила, які будуть застосовуватися до всього додатку. Основні функції файлу index.css включають в себе:

a. Глобальні стилі: Ви можете визначити глобальні стилі, які застосовуються до всіх елементів на вашому сайті. Це може включати налаштування шрифтів, колірів, відступів, вирівнювань та інших стилів, які ви бажаєте застосувати до всього додатку.

b. Стилі для кореневого компонента: Ви також можете використовувати index.css для визначення стилів для кореневого компонента вашого React-додатку, який зазвичай рендериться в файлі index.tsx. Це допомагає встановити загальний вигляд та макет для всього додатку.

c. Стилі для компонентів: Ви можете імпортувати файл index.css в компонентах та використовувати його для стилізації окремих компонентів. Це дозволяє вам розділити стилі на більше дрібні частини та забезпечити модульність та підтримку стилів для кожного компонента.

d. Підключення і збірка стилів: Файл index.css зазвичай підключається до основного файлу index.tsx за допомогою імпорту. Спільно з іншими інструментами збірки, такими як Webpack або Create React App, цей файл може бути об'єднаним та оптимізованим перед розгортанням на продакшн-сервері.

- react-app-env.d.ts: Файл react-app-env.d.ts використовується для встановлення та налаштування середовища розробки для проектів, створених за допомогою Create React App (CRA). Він дозволяє визначити типи та налаштування, які пов'язані з середовищем CRA і впливають на роботу TypeScript в вашому проекті. Основні функції react-app-env.d.ts включають:

a. Підтримка глобальних змінних: Ви можете використовувати цей файл для визначення глобальних змінних або об'єктів, які доступні у вашому додатку без необхідності явного оголошення типів для них в кожному файлі. Наприклад, ви

можете визначити глобальні змінні для сторонніх бібліотек або конфігураційні параметри, які використовуються в багатьох частинах проекту.

b. Автоматичні типи CRA: react-app-env.d.ts включає в себе автоматично генеровані типи, які визначаються CRA, включаючи типи для середовища та реактивний контекст.

c. Можливість розширення: Ви можете розширити цей файл, додавши власні оголошення типів та налаштування, специфічні для вашого проекту. Це дозволяє налаштовувати типи та середовище розробки для ваших потреб.

- i18nextConf.js: даний файл вказує на налаштування для бібліотеки i18next, яка використовується для міжнародного перекладу тексту у вашому додатку. Цей файл допомагає налаштовувати та конфігурувати багатомовний інтерфейс вашого додатку. Основні функції i18nextConf.js включають:

a. Конфігурація i18next: Файл i18nextConf.js зазвичай містить налаштування для i18next, такі як мовні файли, налаштування локалізації, ключі перекладу та інші параметри. Ви можете визначити, яким чином i18next має виконувати переклад тексту та які мовні ресурси використовувати.

b. Інтеграція зі збіркою: Якщо ви використовуєте інструменти збірки, такі як Webpack або Babel, файл i18nextConf.js може містити налаштування для інтеграції i18next з вашою збіркою. Це може включати налаштування плагінів або завдань для автоматичної генерації мовних файлів або оптимізації локалізації.

c. Підключення до додатку: Файл i18nextConf.js зазвичай імпортується в вашому додатку та використовується для налаштування інфраструктури міжнародного перекладу. Ви можете використовувати його для ініціалізації i18next та підключення інтерфейсу міжнародного перекладу до вашого додатку.

- custom.d.ts: використовується для визначення власних типів (custom types) в проекті, коли вам потрібно додати додаткові типи даних, які TypeScript не розпізнає автоматично. Цей файл дозволяє вам розширити систему типізації TypeScript та створити власні типи, інтерфейси або псевдоніми типів, що використовуються в вашому проекті.

- reportWebVitals.ts: використовується в додатках, створених за допомогою Create React App (CRA), для моніторингу та звітування про показники продуктивності вашого веб-додатка в реальному часі. Цей файл дозволяє відстежувати важливі показники, такі як час завантаження сторінки та час відгуку сервера, і передавати цю інформацію на сервери для подальшого аналізу. Основні функції файлу reportWebVitals.ts включають:

a. Збір та вимірювання показників продуктивності: Файл reportWebVitals.ts містить функції для вимірювання показників продуктивності, такі як час завантаження сторінки (LCP - Largest Contentful Paint), час відгуку сервера (TTFB - Time to First Byte), час рендерингу (FCP - First Contentful Paint) та інші. Ці показники допомагають вам визначити, наскільки ефективно працює ваш веб-додаток.

b. Відправка даних на сервер: Після вимірювань, отриманих від браузера, reportWebVitals.ts відправляє ці дані на сервери аналізу або моніторингу, де вони можуть бути оброблені та відображені в зручному для аналітики форматі.

c. Легка інтеграція в додаток: Цей файл легко інтегрується з додатком, створеним за допомогою CRA, і не вимагає значних змін в самому додатку. Процес включення та налаштування reportWebVitals.ts відбувається автоматично під час створення проекту CRA.

- setupTests.ts: даний файл використовується для налаштування оточення тестування (testing environment) в проектах, які використовують бібліотеку для тестування, таку як Jest, у поєднанні з Create React App або іншими інструментами розробки. Цей файл дозволяє вам визначити спеціальні налаштування тестів або виконувати певні дії перед та після запуску тестів. Основні функції файлу setupTests.ts включають:

a. Підключення допоміжних бібліотек: Ви можете використовувати цей файл для підключення додаткових бібліотек або утиліт, які потрібні для тестування, таких як enzyme для тестування React-компонентів або sinon для співпраці з мокованими об'єктами.

b. Налаштування середовища тестування: setupTests.ts дозволяє вам налаштовувати середовище тестування, наприклад, встановлювати глобальні налаштування або підключати функції, які потрібні для тестування (наприклад, налаштування для підтримки браузерної роботи).

c. Запуск коду перед та після тестами: Ви можете використовувати цей файл для виконання певних дій перед або після виконання всіх тестів. Наприклад, ви можете налаштувати підключення до фейкового сервера для тестування API-запитів або ініціалізувати фіксований стан для вашого додатку перед тестами.

d. Загальні налаштування для всіх тестів: Ви можете визначити налаштування, які будуть застосовуватися до всіх тестів у вашому проекті.

Та директорій:

- app: в даній директорії розміщується другий кореневий файл react-додатку App.tsx, адже не всі налаштування та ініціалізації зручно і доречно робити у файлі index.tsx

- assets: Папка src/assets/ використовується для зберігання різноманітних ресурсів та активів, таких як зображення, шрифти, стилі, відео, аудіо, інші медіафайли та статичні файли, які використовуються у вашому веб-додатку. Основна мета цієї папки - це організація та управління ресурсами, які використовуються для візуального та аудіовізуального змісту вашого додатку.

- components: Папка components використовується для організації та розміщення React компонентів у вашому проекті. Кожен компонент в цій папці зазвичай відповідає за обраний фрагмент інтерфейсу вашого додатку та може бути повторно використаним на різних сторінках або частинах додатку. Основні функції папки components включають:

a. Організація коду: Використання папки components допомагає вам структурувати код вашого додатку, робити його більш організованим та легким для розуміння. Кожен компонент розміщений в окремому файлі або підпапці, що полегшує пошук та управління ними.

b. Повторне використання коду: Компоненти в папці components можуть бути повторно використані в різних частинах додатку, що дозволяє вам ефективно використовувати однаковий код для відображення аналогічних елементів інтерфейсу.

c. Семантичність та читабельність: Розміщення компонентів в окремій папці під назвою components робить ваш код більш семантичним та легко зрозумілим. Відразу зрозуміло, що ці файли містять компоненти, які відповідають за відображення елементів інтерфейсу.

d. Легка робота у команді: Якщо ви працюєте у команді, то розділення компонентів на окремі файли в папці components спрощує спільну роботу та розвиток проекту.

e. Тестування: Розділення компонентів на окремі файли полегшує тестування, оскільки ви можете тестувати кожен компонент окремо.

- configs: використовується для зберігання конфігураційних файлів та параметрів, які впливають на поведінку вашого додатку. Ця папка допомагає вам розділити налаштування від іншого коду та зробити його більш підтримуваним і налаштовуваним.

- hooks: використовується для зберігання власних React hooks, які надають додатку певну логіку та функціональність, яка може бути повторно використана в різних частинах вашого проекту. React hooks - це спосіб розділити та повторно використовувати логічну частину компонентів в React.

- interfaces: використовується для зберігання інтерфейсів (interfaces) в вашому проекті на мові програмування TypeScript. Ці інтерфейси визначають структуру та формат даних, що використовуються в вашому додатку, і допомагають вам забезпечити типову безпеку та відслідковуваність в процесі розробки.

- layout: використовується для організації та розміщення макетів сторінок (page layouts) у вашому проекті. Макети сторінок визначають загальну структуру та розмітку сторінок вашого додатку і можуть включати заголовки, навігацію, підвал та інші елементи, які повторюються на багатьох сторінках.

- pages: використовується для організації та розміщення окремих сторінок (page components) у вашому проекті. Кожен файл у цій папці відповідає за одну конкретну сторінку або маршрут вашого веб-додатку. Сторінки визначають, як виглядає та веде себе окрема сторінка вашого додатку.

- Router: може використовуватися для організації налаштування маршрутизації (routing) в вашому веб-додатку. Вона містить налаштування та компоненти, які визначають, які сторінки або компоненти мають бути відображені при різних URL-адресах чи маршрутах. Основні функції папки Router включають:

a. Налаштування маршрутів: У цій папці можна визначити маршрути та URL-шляхи, які відповідають певним сторінкам або компонентам вашого додатку.

b. Керування навігацією: Ви можете визначити логіку переходу між сторінками та обробку URL-параметрів в цій папці.

c. Організація структури маршрутизації: Ця папка допомагає вам розділити та організувати налаштування маршрутизації в окремому місці, роблячи ваш проект більш організованим.

d. Зручне додавання нових сторінок: Додавання нових сторінок або маршрутів вимагає додавання або редагування компонентів у цій папці, що полегшує розширення додатку.

e. Спільна логіка навігації: Якщо ваш додаток включає спільну логіку навігації, таку як захист сторінок від неавторизованих користувачів, ця папка може містити відповідні компоненти та логіку.

f. Тестування маршрутів: Можливість тестувати маршрутизацію і переконатися, що вони працюють правильно.

- services: використовується для організації та розміщення сервісів (або модулів), які взаємодіють з зовнішніми джерелами даних, такими як сервери API, бази даних, сторонні сервіси і інші зовнішні ресурси. Ці сервіси використовуються для виконання операцій з даними, отримання інформації та обробки даних, що подалі використовуються в вашому додатку.

- store: використовується для організації та управління станом додатку у застосуваннях, які використовують підхід до керування станом, в яких використовуються бібліотеки стану, такі як Redux або Mobx. Ця папка містить всі необхідні елементи для роботи зі станом додатку, включаючи дії (actions), редюсери (reducers), ініціалізацію та конфігурацію стору.

- styles: використовується для організації та зберігання стилів, які стосуються вашого веб-додатку. Ця папка містить файли CSS, SASS, LESS, або інші файли стилів, які використовуються для задання зовнішнього вигляду та макету компонентів та сторінок вашого додатку.

- utils: Папка utils (скорочення від "utilities" або "утиліти") використовується для організації та зберігання допоміжних функцій та утиліт, які можуть бути використані в різних частинах вашого проекту. Ці функції та утиліти розробляються для виконання загальних завдань, які можуть бути використані в різних компонентах, сервісах або модулях вашого додатку.

В результаті файлова структура директорії src у вікні редактора коду виглядає наступним чином:

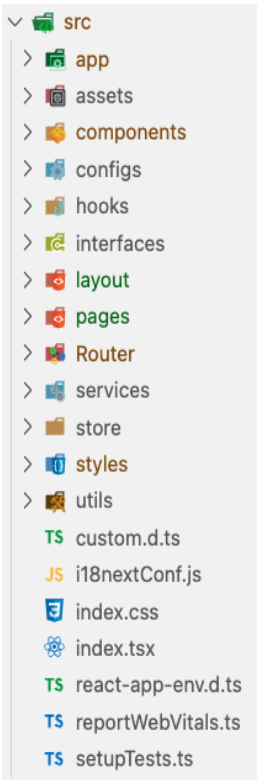

Рис.2.5. Файлова структура директорії src

2.4.3. Робота над ключовим компонентом App

Для початку, напишемо код для ініціалізації базової логіки основних вузлів проекту у файлі App.tsx:

```
import { BrowserRouter } from 'react-router-dom';
import { FC, Suspense, useCallback, useEffect } from 'react';
import i18next from 'i18next';
import {
  CircularProgress,
  StyledEngineProvider,
  ThemeProvider,
} from '@mui/material';
import { useDispatch, useSelector } from 'react-redux';
```

```
import styled from 'styled-components';

import theme from '../styles/theme';
import { RootState } from '../store/store';
import { addShopItems } from '../store/reducers/shopItemsSlice';
import { API_BASE_URL } from '../services/api';
import { NavMenu } from '../components';
import { Header } from '../layout';

const RootSuspenseFallbackContainer = styled.div`
  width: 100vw;
  height: 100vh;
  display: flex;
  align-items: center;
  justify-content: center;
`;

const RootSuspenseFallback = () => (
  <RootSuspenseFallbackContainer>
    <CircularProgress />
  </RootSuspenseFallbackContainer>
);

const App: FC = () => {
  const dispatch = useDispatch();

  const fetchShopItems = useCallback(async () => {
    const res = await fetch(`${API_BASE_URL}/shop-items`, {
      method: 'GET',
```

```
  });
  const data = await res.json();
  dispatch(addShopItems(data));
}, [dispatch]);

const { lang } = useSelector((state: RootState) => state.core);

useEffect(() => {
  i18next.changeLanguage(lang);
}, [lang]);

useEffect(() => {
  fetchShopItems();
}, [fetchShopItems]);

return (
  <StyledEngineProvider injectFirst>
    <BrowserRouter>
      <Suspense fallback={RootSuspenseFallback}>
        <ThemeProvider theme={theme}>
          <AppContainer data-testid="app-root-container">
            <Header />
            <NavMenu />
          </AppContainer>
        </ThemeProvider>
      </Suspense>
    </BrowserRouter>
  </StyledEngineProvider>
);
```

```
};

const AppContainer = styled.div`
  display: flex;
  flex-direction: column;
  justify-content: space-between;
  flex: 1;
  background-color: white;
  min-height: 100vh;
`;

export default App;
```

У даному коді ми:

1. Імпортували всі необхідні бібліотеки та компоненти
2. Прописали стилі верхньорівневого контейнера
3. У методі fetchShopItems прописали запит на сервер для отримання списку усіх речей, що є в інтернет-магазині
4. Прописуємо рендеринг усих необхідних нам компонентів-”обгорток”, які дадуть доступ до глобальних функцій, це:

a. StyledEngineProvider та ThemeProvider - для коректної роботи стилів бібліотеки Material UI

b. BrowserRouter - для роботи роутингу бібліотеки react-router-dom

c. Suspense - компонент, який відображатиме процес завантаження, якщо сайт буде завантажувати будь-які дані

Надалі більша частина коду буде писатися в директоріях components, layout, та pages:

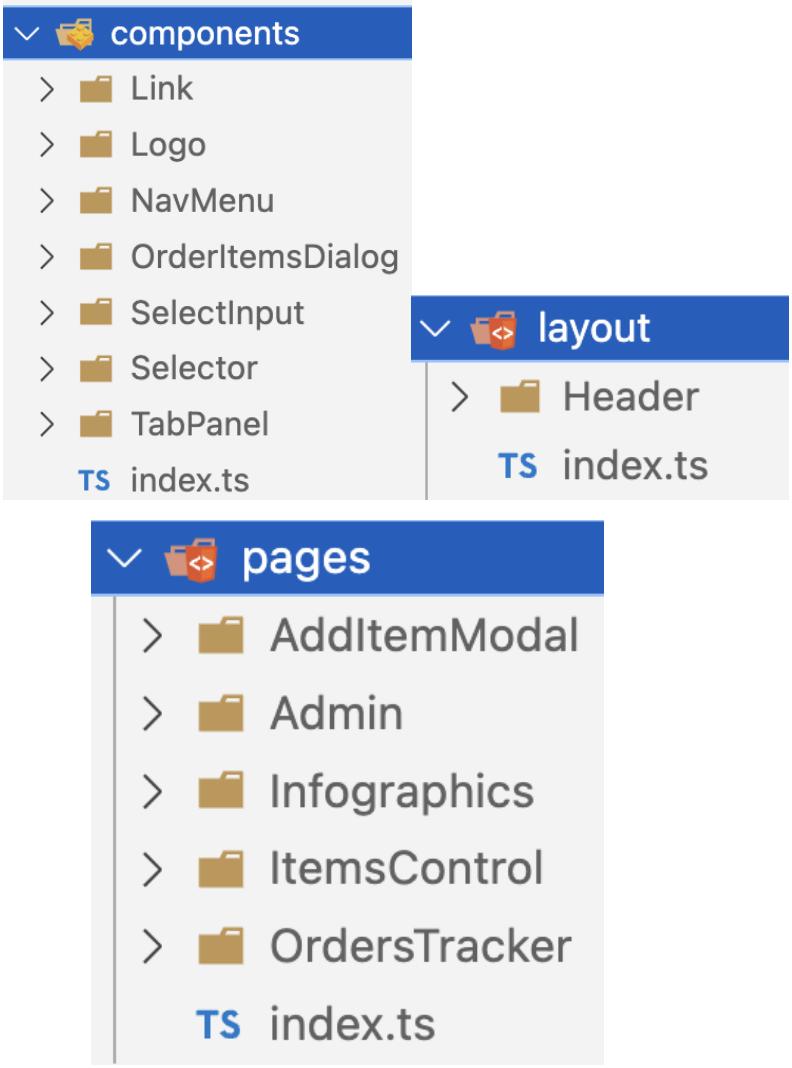

Рис.2.6. Файлова структура компонентів

В них ми створюватимемо спочатку найменші компоненти, які потім компонуватимуться в складених компонентах “layouts”, а ті, в свою чергу, будуть компонуватися в повноцінні сторінки, які будуть створені в директорії pages.

2.4.4. Атомарні компоненти

Атомарні компоненти зберігаються в директорії components і являються відносно незалежними і базовими. Створюємо, наприклад, компонент селектор для форм, яке відповідатиме потребам нашої системи:

```
import {
  SelectChangeEvent,
  FormControl,
  InputLabel,
  Select,
  MenuItem,
  FormHelperText,
} from '@mui/material';
import { FC } from 'react';

interface ISelectInputProps<ValueType> {
  disabled?: boolean;
  label: string;
  id: string;
  value: ValueType;
  onChange: (event: SelectChangeEvent<any>) => void;
  elements: ValueType[];
  error?: string;
}
const SelectInput: FC<ISelectInputProps<any>> = ({
  disabled,
  label,
  id,
  value,
  onChange,
  elements,
  error,
}) => (
  <FormControl fullWidth disabled={disabled}>
```

```
    <InputLabel>{label}</InputLabel>
    <Select
      id={id}
      name={id}
      value={value}
      label={label}
      onChange={onChange}
      error={!!error}
    >
      {elements.map((it, i) => (
        <MenuItem key={i} value={it}>
          {it}
        </MenuItem>
      ))}
    </Select>

    <FormHelperText error={!!error}>{error}</FormHelperText>
  </FormControl>
);
export default SelectInput;
```

В даному компоненті ми:

1. Імпортуємо залежності та бібліотеки
2. Прописуємо інтерфейс параметрів компоненту:

a. disabled - заблокований він чи ні

b. label - текстовий підпис, щоб користувач бачив що за дані він має вибирати

c. id - ключ-ідентифікатор поля селектору

d. value - обране в селекторі значення

e. onChange - функція-обробник, яка визначає як обробляти вибране значення

f. elements - варіанти значень вибору в селекторі

g. error - помилка, яку буде створювати форма, якщо в полі було вибрано неправильне значення

2.4.5. Компоненти компоновки layouts

Компоненти компоновки зберігаються у директорії layout. Ці компоненти об’єднують в собі менші, зачсту тільки атомарні, компоненти. Наприклад розділ графіків Infographics:

```
import { useEffect, useState } from 'react';
import {
  LineChart,
  Line,
  XAxis,
  YAxis,
  CartesianGrid,
  Tooltip,
  ResponsiveContainer,
} from 'recharts';
import styled from 'styled-components';

import { OrderUADto, OrderWWDto } from '../../interfaces/order';
import { getAllOrders } from '../../services/orders';
import {
  calculateOrderCounts,
  calculateMoneyCounts,
} from '../../utils/inforgraphics';

type OrderData = {
```

```
  date: string;
  amount: number;
};

const Container = styled.div`
  min-height: 70vh;
  display: flex;
  flex: 1;
  justify-content: space-around;
`;

const ChartContainer = styled.div`
  width: 45%;
  display: flex;
  flex-direction: column;
  align-items: center;
`;

const ChartTitle = styled.h3`
  margin-bottom: 10px;
`;

const Infographics = () => {
  const [orders, setOrders] = useState<(OrderUADto | OrderWWDto)[]>([]);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const { data } = await getAllOrders();
```

```
      setOrders(data);
    } catch (error) {
      console.error('Error fetching order data:', error);
    }
  };

  fetchData();
}, []);

const ordersCountData: OrderData[] = calculateOrderCounts(orders);
const moneyCountData: OrderData[] = calculateMoneyCounts(orders);

return (
  <Container>
    <ChartContainer>
      <ChartTitle>Orders by Month</ChartTitle>
      <ResponsiveContainer width="100%" height="80%">
        <LineChart data={ordersCountData}>
          <CartesianGrid strokeDasharray="3 3" />
          <XAxis dataKey="date" />
          <YAxis />
          <Tooltip />
          <Line type="monotone" dataKey="amount" stroke="#8884d8" />
        </LineChart>
      </ResponsiveContainer>
    </ChartContainer>

    <ChartContainer>
      <ChartTitle>Amount in EUR by Month</ChartTitle>
```

```
        <ResponsiveContainer width="100%" height="80%">
          <LineChart data={moneyCountData}>
            <CartesianGrid strokeDasharray="3 3" />
            <XAxis dataKey="date" />
            <YAxis />
            <Tooltip />
            <Line type="monotone" dataKey="amount" stroke="#8884d8" />
          </LineChart>
        </ResponsiveContainer>
      </ChartContainer>
    </Container>
  );
};
export default Infographics;
```

В даному компоненті ми так само спочатку імпортуємо залежності та бібліотеки, а також наші інтерфейси типів. Після цього прописуємо стилі за допомогою бібліотеки styled-components, яка дозволяє створювати React-компоненти з css-стилів. Завдяки цій бібліотеці не потрібно створювати окремі css-файли та використовувати стилі через css класи, ідентифікатори, селектори, тощо. В інфраструктурі React-додатків це вважається дуже сучасним, зручним, та потужним інструментом.

Після того, як ми підготували усі стилі та імпортували сторонні бібліотеки та інтерфейси, створюємо сам компонент. Компонент графіків має отримувати дані про всі замовлення інтернет-магазину задля побудови цих самих графіків. Для цього спочатку прописуємо внутрішній “стан” компоненту, тобто дані, які зберігатимуться в компоненті до моменту його “видалення” з DOM-дерева, в нашому випадку - коли ми переходитимемо зі сторінки графіків на будь-яку іншу. Ініціалізуємо стан orders,

який за замовчуванням являється пустим масивом, та функцію setOrders, яка буде записувати нові дані в масив замовлень, зокрема при першій ініціалізації.

При ініціалізації компоненту, тобто при відкритті сторінки графіків, ми за допомогою функції useEffect(()=>{}, []) відловлюємо цей момент і викликаємо функцію fetchData, в якій відбувається асинхронний виклик методу getAllOrders, який звертається на сервер, і після отримання даних, тобто усих замовлень, ми викликаємо функцію setOrders, яка записує в стан компоненту отримані дані. Після цього розраховуємо кількість замовлень, вартості по місяцях, тобто підготовлюємо дані до відображенні на графіках. І вже тоді відбувається рендеринг компоненту.

2.4.6. Створення сторінок сайту

Сторінки це найвищий рівень компонентів, в якому ми об’єднуємо атомарні компоненти та їх компоновки в повноцінні сторінки. У випадку нашої системи відбувається невелике відхилення від цього підходу розподілу компонентів, бо ми маємо, доприкладу, бічне меню, яке є компонентом компоновки (layout), але в ньому ми перемикаємо сайт між сторінками (pages).

2.5. Створення серверної частини CRM-системи

2.5.1. Опис фреймворку

Наш проект буде використовувати серверну частину на основі Node.js технології, що являє собою виконавче середовище (runtime environment), яке дозволяє виконувати JavaScript код на серверному боці. Node.js базується на движку JavaScript V8, розробленому компанією Google, і відкритий для використання та розробки спільнотою. Основні особливості та характеристики Node.js включають:

- Серверна розробка: Node.js дозволяє розробникам створювати серверні застосування та мережеві додатки з використанням JavaScript. Це особливо корисно для розробки веб-серверів, API, чат-серверів, трансляції даних тощо.

- Невербальний (non-blocking) та асинхронний: Однією з ключових особливостей Node.js є його невербальний та асинхронний підхід до обробки запитів. Це означає, що Node.js може обробляти багато запитів одночасно без блокування потоку виконання.

- Побудований на подіях (event-driven): Node.js використовує подій для обробки подій та відгуку на запити. Це робить його дуже ефективним для мережевого програмування та обробки багатьох одночасних підключень.

- Широкий вибір модулів: В Node.js існує велика кількість модулів та бібліотек, що дозволяє розробникам легко додавати функціональність до своїх додатків.

- Платформонезалежність: Node.js підтримується на різних платформах, таких як Windows, macOS, Linux, що дозволяє розробникам розгортати свої додатки на різних серверах та обчислювальних платформах.

- Активна спільнота: Node.js має велику та активну спільноту розробників, що сприяє постійному розвитку та оновленню.

Node.js добре підходить для розробки веб-додатків, особливо тих, які вимагають обробки багатьох одночасних підключень та подій. Він також корисний для створення інструментів командного рядка, роботи з базами даних, створення API та інших завдань на серверному боці.

Але використовувати “чистий” Node.js не дуже зручно і досить довго. Для швидкого, гнучкого, та не менш потужного функціоналу створення серверної частини проекту розробники використовують додаткові бібліотеки та/або фреймворки. Я обрав один із найпопулярніших фреймворків для серверної розробкик - NestJS.

NestJS – це фреймворк для розробки серверних застосувань, побудованих з використанням мови програмування TypeScript і заснованих на архітектурному

шаблоні "Сервер для сервера" (Server-Side) або "Node.js фреймворк для масштабованих додатків". NestJS надає структуру та інструменти для створення ефективних та масштабованих серверних додатків.Основні особливості та характеристики NestJS включають:

- Використання TypeScript: NestJS побудований на TypeScript, що дозволяє використовувати статичну типізацію та сучасний синтаксис для розробки додатків, що полегшує відладку та підтримку.
- Модульна структура: Додатки в NestJS будуються на основі модульної структури, яка сприяє організації коду та розділенню функціональності на окремі модулі.
- Вбудована обробка HTTP-запитів: NestJS надає вбудовану підтримку для обробки HTTP-запитів, включаючи створення маршрутів, контролери та фільтри.
- Завдання (Task) та Розкладка (Scheduler): NestJS дозволяє легко створювати завдання, які можуть бути запущені за розкладом або як реакція на певні події.
- Система обробки помилок: Надійна система обробки помилок та журналізації, що сприяє відстеженню та вирішенню помилок в додатку.
- Легке тестування: NestJS підтримує тестування, що допомагає вам переконатися, що ваш додаток працює правильно.
- Підтримка WebSocket: NestJS надає підтримку WebSocket, що дозволяє створювати інтерактивні та реального часу додатки.
- Спільнота та розширення: NestJS має активну спільноту розробників і багато розширень для спрощення розробки певних функцій.
- Масштабованість: NestJS дозволяє побудовувати масштабовані додатки завдяки своїй архітектурі та підтримці мікросервісів.

Загалом, NestJS - це потужний та гнучкий фреймворк, який допомагає розробникам створювати надійні та масштабовані серверні додатки з використанням TypeScript та сучасних підходів до розробки.

2.5.2. Ініціалізація серверної частини проекту

Для ініціалізації NestJS потрібно виконати лише декілька простих кроків. По-перше, встановлюємо глобальний інтерфейс для терміналу nestjs/cli за допомогою команди: npm i -g @nestjs/cli. Після чого нам в терміналі стають доступні методи даного інтерфейсу. Використовуючи метод “new” ініціалізуємо проект: nest new <project-name>. Дана команда створює повноцінний і майже завершений каркас проекту, після якого, у більшості випадків, не потрібно нічого додатково конфігурувати. Після ініціалізації структура нашого проект має наступний вигляд:

```
project-name/
  ├── dist/
  ├── node_modules/
  ├── src/
  │   ├── main.ts
  │   ├── app.module.ts
  │   ├── app.controller.ts
  │   ├── app.service.ts
  ├── test/
  ├── .eslintrc.js
  ├── .prettierrc
  ├── jest.config.js
  ├── package.json
  ├── README.md
  ├── tsconfig.build.json
  ├── tsconfig.json
```

Основні елементи структури проекту NestJS включають:

- dist/: Каталог, де зберігається скомпільований код TypeScript після збірки проекту. Цей каталог автоматично створюється під час роботи з фреймворком.

- node_modules/: Каталог, де зберігаються залежності проекту, такі як бібліотеки та пакети.

- src/: Основний каталог для розробки вашого додатку, який містить наступні файли та папки:

  - main.ts: Головний файл, з якого починається виконання додатку.
  - app.module.ts: Головний модуль додатку, де визначається основна логіка та завдання додатку.
  - app.controller.ts: Приклад контролера, який обробляє HTTP-запити.
  - app.service.ts: Приклад служби (service), яка надає функціональність для контролера.

- test/: Каталог, в якому можна зберігати тести для вашого додатку.

- .eslintrc.js: Файл конфігурації ESLint для налаштування стилевих перевірок коду.

- .prettierrc: Файл конфігурації Prettier для форматування коду.

- jest.config.js: Конфігурація для фреймворку тестування Jest.

- package.json: Файл з описом проекту та його залежностями.

- README.md: Документація проекту, яка містить опис та інструкції.

- tsconfig.build.json та tsconfig.json: Файли конфігурації TypeScript для компіляції та розробки проекту відповідно.

Ця структура є загальною для проектів NestJS, і далі ми будемо додавати власні файли, модулі та ресурси, відповідно до потреб нашого додатку. NestJS надає можливість створювати різні модулі, контролери, служби та фільтри для організації та розвитку вашого серверного додатку.

2.5.3. Створення серверного функціоналу системи

Ключовим принципом розробки нашого серверного функціоналу є створення так званих REST API-points. REST API-точки (або REST API-ресурси) представляють собою URL-шляхи або URI, які визначають ресурси, які можуть бути доступні через RESTful API. Кожна точка в API відповідає певному ресурсу або колекції ресурсів та дозволяє клієнту здійснювати різні операції з цими ресурсами, такі як створення, читання, оновлення та видалення (операції CRUD).

REST (Representational State Transfer) - це архітектурний стиль, який використовує HTTP-протокол для взаємодії між клієнтом і сервером. В рамках цього стилю, кожен ресурс представлений у вигляді унікального URI, і клієнти можуть використовувати HTTP-методи (GET, POST, PUT, DELETE) для взаємодії з цими ресурсами.

Основні характеристики REST API-точок включають:

- Унікальні URI: Кожен ресурс має унікальний URI, який ідентифікує його на сервері. Наприклад, /users може представляти колекцію користувачів.

- HTTP-методи: REST API використовує стандартні HTTP-методи для взаємодії з ресурсами. Наприклад, GET для отримання даних, POST для створення нового ресурсу, PUT для оновлення існуючого ресурсу, та DELETE для видалення ресурсу.

- Представлення ресурсів: Ресурси можуть бути представлені у різних форматах, таких як JSON, XML або HTML, в залежності від потреб клієнта та сервера.

- Безстандартність (Statelessness): Кожен запит клієнта до сервера повинен містити всю необхідну інформацію для обробки запиту. Сервер не зберігає інформацію про стан клієнта між запитами.

- Самопописність (Self-descriptiveness): REST API повинен бути самоописним, тобто клієнт може розуміти, як взаємодіяти з API, завдяки інформації, яка надається відповідями сервера.

Приклади REST API-точок включають /users для роботи з користувачами, /products для управління продуктами, /orders для замовлень тощо. Кожна точка

дозволяє клієнтам отримувати, створювати, оновлювати та видаляти відповідні ресурси на сервері, надаючи стандартизований та послідовний спосіб взаємодії з додатком.

Для нашого проекту ключовими будуть API-точки для ресурсів "orders", "shop-items", "promos" та "support" за допомогою NestJS включає наступні кроки:

1. Створення модулів: Для кожного ресурсу створюємо власний модуль. Наприклад, orders.module.ts, shop-items.module.ts, promos.module.ts та support.module.ts. В кожному модулі визначаємо контролери, служби та інші компоненти, пов'язані з цим ресурсом.

2. Створення контролерів: Для кожного ресурсу створюємо контролер, наприклад, orders.controller.ts, shop-items.controller.ts, promos.controller.ts та support.controller.ts. У контролерах визначаємо обробники запитів HTTP (GET, POST, PUT, DELETE) для ресурсу.

3. Створення служб: Для кожного ресурсу створюємо службу, наприклад, orders.service.ts, shop-items.service.ts, promos.service.ts та support.service.ts. У службах реалізовуємо бізнес-логіку для роботи з даними цих ресурсів.

4. Визначення маршрутів: В кожному модулі визначаємо маршрути, за допомогою яких клієнти зможуть взаємодіяти з ресурсом. Використовуємо декоратори @Get(), @Post(), @Put() та @Delete() для визначення маршрутів та викликів контролера.

5. Підключення модулів: Після створення модулів включаємо їх до головного модуля нашого додатку (зазвичай це app.module.ts). Для цього використовуємо декоратор @Module() та імпортуємо наші модулі.

6. Захист маршрутів: Якщо потрібно буде захистити деякі API-точки, можна розглянути можливість використання middleware або гвардіанів для автентифікації та авторизації.

7. Тестування

8. Документація: Рекомендується створити документацію для API, щоб інші розробники могли легко розуміти, як взаємодіяти з ресурсами.

Node.js сервер на фреймворку NestJS починається з файлу main.ts:

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.enableCors();
  await app.listen(process.env.PORT || 8000);
}
bootstrap();
```

В даному файлі імпортуються кореневий модуль проекту та клас NestFactory з фреймворку NestJS, який створює серверну логіку. Ми прописуємо метод ініціалізації серверу bootstrap(), який ми одразу й викликаємо. В даному методі ми створюємо константу app, яка являє собою комплексний інтерфейс нашого серверу, який створює метод create() класу NestFactory.

Після створення ключового інтерфейсу app ми вмикаємо його cors-налаштування. Метод app.enableCors() у файлі main.ts у NestJS проекті вмикає CORS (Cross-Origin Resource Sharing) для вашого додатку. CORS - це механізм, який дозволяє веб-серверам вказувати, з яких джерел можуть завантажуватися ресурси на веб-сторінку в браузері.

Коли ми викликаємо app.enableCors() в головному файлі вашого додатку, це дозволяє іншим веб-додаткам (наприклад, клієнтським додаткам, які запитують ресурси через AJAX) зробити запити на наш сервер, навіть якщо ці додатки розташовані на інших доменах чи портах. Без увімкнення CORS, браузери зазвичай блокують такі запити для безпеки (Same-Origin Policy).

app.enableCors() надає серверу інформацію про те, які джерела (домени або IP-адреси) мають доступ до ресурсів на сервері. Ми можемо додатково налаштувати

параметри CORS, вказавши джерела, яким можна довіряти, HTTP-заголовки, дозволені методи тощо.

Наприклад:

```
app.enableCors({
  origin: 'http://example.com', // Домен, з якого дозволено запити
  methods: 'GET,POST,PUT,DELETE', // Дозволені HTTP-методи
  allowedHeaders: 'Content-Type,Authorization', // Дозволені HTTP-заголовки
});
```

І тепер ми можемо запустити наш сервер командою app.listen(), до якої ми параметром передаємо порт, на якому має запуститись наш сервер.

Далі ми створюємо той самий модуль app.module.ts:

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { ShopItemsModule } from './shop-items/shop-items.module';
import { ServeStaticModule } from '@nestjs/serve-static';
import { join } from 'path';
import { OrderModule } from './order/order.module';
import { MongooseModule } from '@nestjs/mongoose';
import { ConfigModule } from '@nestjs/config';
import { SupportModule } from './support/support.module';
import { PromosModule } from './promos/promos.module';
import { CertificatesModule } from './certificates/certificates.module';

@Module({
  controllers: [AppController],
  imports: [
    ServeStaticModule.forRoot({
      rootPath: join(__dirname, '..', 'public'),
```

```
      exclude: ['/*'],
    }),
    MongooseModule.forRoot(process.env.MONGO_URI),
    ConfigModule.forRoot(),
    ShopItemsModule,
    OrderModule,
    SupportModule,
    PromosModule,
    CertificatesModule,
  ],
})
export class AppModule {}
```

Файл app.module.ts є головним модулем нашого NestJS додатку і відіграє центральну роль у визначенні структури та компонентів. У цьому файлі визначається основний модуль, який об'єднує всі інші модулі та компоненти, які використовуються в додатку. Ось що відбувається у файлі app.module.ts:

- Імпорт залежностей: У файлі app.module.ts імпортуємо всі необхідні модулі та компоненти, які нам потрібні для роботи додатку. Це включає модулі для роботи з HTTP-запитами, базами даних, сервісами, контролерами тощо.
- Визначення декоратора @Module: Цей декоратор вказує, які компоненти і модулі пов'язані з основним модулем, і включає такі параметри як imports, controllers, providers та інші.
- Підключення компонентів: Ми додаємо компоненти до основного модулю за допомогою параметрів controllers, providers та інших. Наприклад, контролери, які обробляють HTTP-запити, та служби, які надають функціональність для додатку.
- Конфігурація середовища: тут ми можемо вказати параметри конфігурації, такі як порт сервера, базу даних, налаштування CORS та інші параметри.

- Завантаження файлів конфігурації: Якщо нам потрібно додати файли конфігурації, вони можуть бути завантажені та використані тут для налаштування додатку.

- Запуск додатку: Зазвичай у кінці app.module.ts ви викликаєте метод app.listen(), який запускає сервер на визначеному порту і починає обробку запитів.

Все це робить файл app.module.ts центральною точкою NestJS додатку, де визначається, які компоненти та модулі об'єднуються, як вони налаштовані і як запускається додаток.

А далі наше завдання створювати окремі ресурси зі своїми модулями, контролерами, сервісами, моделями та схемами даних. Ось так, наприклад, виглядає структура модулю замовлень:

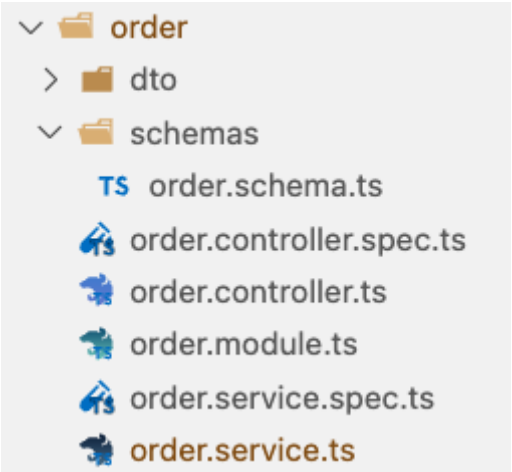

Рис.2.7. Файлова структура модулю замовлень

Даний ресурс складається з наступних файлів:

- order.module.ts - це корінний файл модулю замовлень. Його код виглядає наступним чином:

```
@Module({
  controllers: [OrderController],
```

```
  providers: [OrderService],
  imports: [
    MongooseModule.forFeature([
      { name: OrderUA.name, schema: OrderUASchema },
      { name: OrderWW.name, schema: OrderWWSchema },
      { name: ShopItem.name, schema: ShopItemSchema },
    ]),
  ],
})
export class OrderModule {}
```

В ньому ми підключаємо наш контролер, сервіс-провайдер, та схеми для бази даних, використовуючи бібліотеку для роботи з NoSQL БД MongoDB - mongoose.

- order.controller.ts - файл контролеру відповідає за першочергову обробку HTTP-запитів:

```
@Controller('order')
export class OrderController {
  constructor(private readonly orderService: OrderService) {}
  @Get()
  getAllOrders() {
    return this.orderService.getAll();
  }
  @Post()
  async createOrder(@Body() order: CreateOrderDto) {
    try {
      return await this.orderService.createOrder(order);
    } catch (error) {
      throw new HttpException('Error', error.status);
```

```
  }
 }
 @Post('/email/test')
 testEmail() {
  return this.orderService.testEmail();
 }
 @Post('/approve')
 @HttpCode(HttpStatus.OK)
 approveOrder(@Body() data: IPaymentResponse) {
  return this.orderService.approveOrder(data);
 }
 @Post('success')
 @Redirect('https://www.***.com.ua/success', 301)
 successOrder() {
  return { url: 'https://www.***.com.ua/success' };
 }
}
```

Як видно з коду, контролер за допомогою декораторів http-методів обробляє конкретні методи запитів на конкретні шляхи модулю order. Було створено обробку методів GET ‘/’ для отримання усих замовлень, методу POST ‘/’ з даними нового замовлення в тілі запиту, POST ‘/email/test’ для тестування відправки електронної пошти з інформацією по замовленню, POST ‘/approve’ який викликається платіжною системою для підтвердження замовлення після оплати, а також POST ‘/success’ який відправляє користувача на сторінку сайту з інформацією про успішне замовлення.

- order.controller.spec.ts – це файл для тестів контролеру
- order.service.ts – сервіс-провайдер відповідає за роботу з даними. До

його методів звертається контролер після первинної обробки запиту:

```
@Injectable()
```

```
export class OrderService {
  constructor(
    @InjectModel(OrderUA.name) private orderUaModel: Model<OrderUADocument>,
    @InjectModel(OrderWW.name) private orderWwModel:
Model<OrderWWDocument>,
    @InjectModel(ShopItem.name) private shopItemModel: Model<ShopItemDocument>,
  ) {}
          async getAll(): Promise<(OrderUA | OrderWW)[]> {
            const ordersUa = await this.orderUaModel.find().exec();
            const ordersWw = await this.orderWwModel.find().exec();
            return [...ordersUa, ...ordersWw];
          }

          async createOrder(orderDto: CreateOrderDto): Promise<any> {
            const { items } = orderDto;
            try {
              for (const item of items) {
                const shopItem = await this.shopItemModel.findOne({ id: item.id });
                if (!shopItem) {
                  throw new Error(`Shop item with ID ${item.id} not found`);
                }
                const amount = shopItem.amount[item.size];
                if (amount <= 0) {
                  throw new Error(
                    `No ${item.size}-size items of item with ID ${item.id} left`,
                  );
                }
              }
              const modelData = {
```

```
      ...orderDto,
      ...orderDto.orderData,
    };
    const newOrder =
      orderDto.orderType === 'ukraine'
        ? new this.orderUaModel(modelData)
        : new this.orderWwModel(modelData);

    return newOrder.save();
  } catch (error) {
    throw new InternalServerErrorException();
  }
}

testEmail() {
        sendOrderWwEmail('test@gmail.com','Test',Test','+xxxxxxxxxxx','City','Cou
ntry','Postal code','Address',
    [
      {
        id: 'xxxxxx',
        title: 'Shop item',
        size: 'XS',
      },
    ],
    1000,null,'uah',
  );
}

async approveOrder(paymentResponse: IPaymentResponse) {
```

```
if (
  paymentResponse.response_status !== 'success' ||
  paymentResponse.order_status !== 'approved'
) {
  return;
}

const [uaCandidate, wwCandidate] = await Promise.all([
  this.orderUaModel.findOne({ orderId: paymentResponse.order_id }),
  this.orderWwModel.findOne({ orderId: paymentResponse.order_id }),
]);

if (uaCandidate && !uaCandidate.approved) {
  for (const item of uaCandidate.items) {
    const shopItem = await this.shopItemModel.findOne({ id: item.id });
    if (!shopItem) {
      throw new Error(`Shop item with ID ${item.id} not found`);
    }
    const amount = shopItem.amount[item.size];
    if (amount <= 0) {
      throw new Error(
        `No ${item.size}-size items of item with ID ${item.id} left`,
      );
    }
    shopItem.amount[item.size] = amount - 1;
    await shopItem.save();
  }

  await this.orderUaModel.findOneAndUpdate(
```

```
      { orderId: paymentResponse.order_id },
      { approved: true },
    );

    sendOrderUaEmail(...uaCandidate);
  }

  if (wwCandidate && !wwCandidate.approved) {
    for (const item of wwCandidate.items) {
      const shopItem = await this.shopItemModel.findOne({ id: item.id });
      if (!shopItem) {
        throw new Error(`Shop item with ID ${item.id} not found`);
      }
      const amount = shopItem.amount[item.size];
      if (amount <= 0) {
        throw new Error(
          `No ${item.size}-size items of item with ID ${item.id} left`,
        );
      }
      shopItem.amount[item.size] = amount - 1;
      await shopItem.save();
    }
    await this.orderWwModel.findOneAndUpdate(
      { orderId: paymentResponse.order_id },
      { approved: true },
    );
    sendOrderWwEmail(...wwCandidate);
  }
}
```

}

В цьому файлі відбувається ключова логіка серверу, яка визначає як обробляти дані по кожному із запитів, контактує з базою даних, тощо. Всі ці методи ми бачили як викликаються у контролері.

- order.service.spec.ts – це файл для тестів сервіс-провайдеру
- oder.schema.ts – це файл-схема для бази даних, де прописуються всі поля та їх типи для екземпляру замовлень. Ось так, наприклад, виглядає схема замовлення, яке було замовлено для клієнта в Україні:

```
@Schema()
export class OrderUA {
 @Prop()
 approved: boolean;
 @Prop()
 orderId: string;
 @Prop([
  {
   id: { type: String },
   size: { type: String },
   title: { type: String },
   additionalParams: [{ type: String }],
  },
 ])
 items: {
  id: string;
  size: string;
  title: string;
  additionalParams?: string[];
 }[];
```

```
@Prop()
price: number;
@Prop()
orderType: 'ukraine';
@Prop()
email: string;
@Prop()
fullName: string;
@Prop()
phone: string;
@Prop()
city: string;
@Prop()
novaPoshta: string;
@Prop()
agreement: boolean;
@Prop({
  type: {
    name: { type: String },
    discount: { type: Number },
  },
})
promo: { name: string; discount: number };
@Prop()
createdDateString: string;
@Prop()
delivery: number;
@Prop()
currency: 'uah' | 'eur';
```

```
}
```

- order.dto.ts – цей файл також визначає поля і їх типи, але трохи інакшим чином і вже не для бази даних, а для внутрішніх операцій на сервері. Ось наприклад інтерфейс замовлення прописаний в даному файлі:

```
export type OrderUADto = {
  email: string;
  fullName: string;
  phone: string;
  city: string;
  novaPoshta: string;
  agreement: boolean;
  promo: CreatePromoDto | null;
  currency: 'eur' | 'uah';
};
```

2.5.4. Підключення бази даних

Базу даних я вирішив використовувати MongoDB та користуватись її хмарною версією MongoDB Atlas, що значно спрощує налаштування та не потребує додатково її розміщувати на орендованих серверах.
Для підключення вашої бази даних з MongoDB Atlas в NestJS ми можемо використовувати офіційний драйвер MongoDB для Node.js, такий як mongoose. Ось кроки, які нам потрібно виконати:

1. Встановлення mongoose: Встановлюємо mongoose в нашому проекті за допомогою npm або yarn: *npm install mongoose*.

2. Імпорт mongoose: У нашому файлі app.module.ts або в будь-якому іншому місці, де ми плануємо використовувати базу даних, імпортуємо mongoose: *import * as mongoose from 'mongoose'*.

3. Підключення до бази даних: Додайте код для підключення до вашої бази даних. Ми можемо зробити це у файлі app.module.ts або створити окремий сервіс для керування підключенням до бази даних. Ось приклад підключення:

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  await mongoose.connect(
‘mongodb+srv://<username>:<password>@<cluster>.mongodb.net/<database>’,
{
    useNewUrlParser: true,
    useUnifiedTopology: true,
  });
  await app.listen(3000);
}
bootstrap();
```

Замість <username>, <password>, <cluster> та <database> потрібно вказати наші дані доступу та назву нашої бази даних з MongoDB Atlas.

4. Використання бази даних: Тепер, коли ми підключилися до бази даних, ми можемо створювати схеми, моделі та взаємодіяти з даними з допомогою mongoose. Нам потрібно буде створити моделі для сутностей, які ми плануємо зберігати в базі даних, і визначити їх структуру.

Це загальний підхід до підключення MongoDB Atlas до NestJS. Пам'ятаймо також про забезпечення безпеки та налаштування прав доступу до бази даних у нашому обліковому записі MongoDB Atlas.

Інтерфейс хмарного середовища бази даних MongoDB виглядає наступним чином:

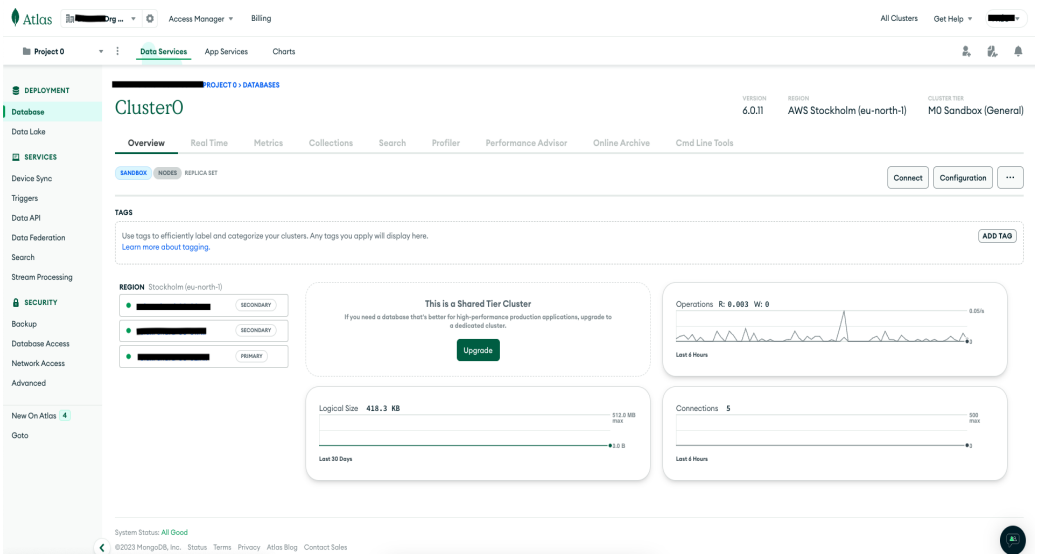

Рис.2.8. Інтерфейс головної сторінки MongoDB Atlas

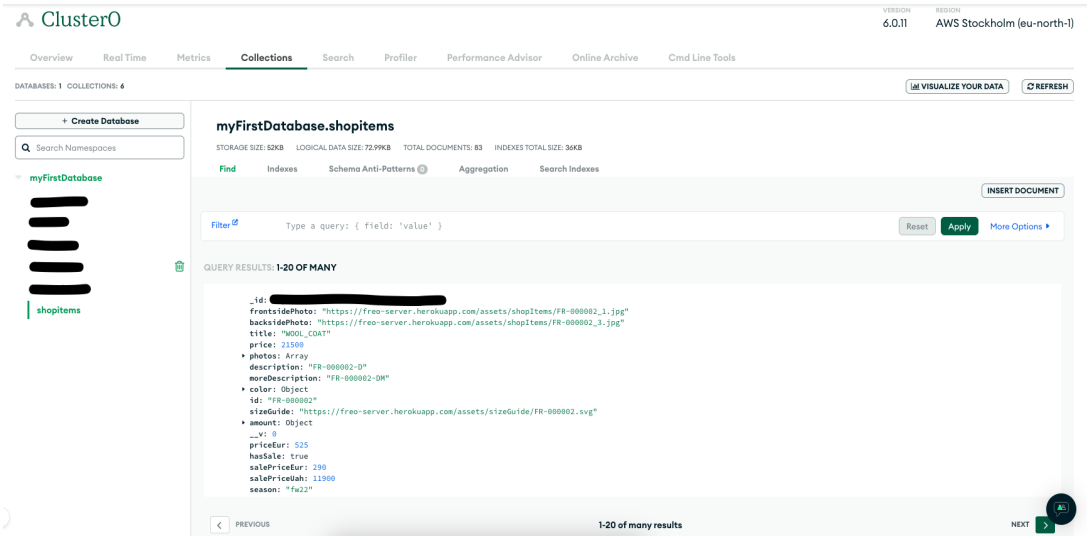

Рис.2.9. Інтерфейс сторінки колекції товарів у MongoDB Atlas

Це дві сторінки онлайн-інтерфейсу, перша – це головна сторінка із загальними даними, а друга це колекції бази даних, в якій я відкрив колекцію речей інтернет-магазину. Частину даних було приховано задля безпеки даних існуючого інтернет-магазину.

2.6. Налаштування хостингу та реліз проєкту

Проект буде розміщуватися на серверах сервісу Heroku. Heroku - це платформа для розгортання та хостингу веб-додатків. Вона дозволяє розробникам розгортати, керувати та масштабувати свої додатки в хмарному середовищі з меншим рівнем зусиль.

Розміщення нашого проекту на хостингу Heroku включає декілька кроків:

1. Створюємо обліковий запис на Heroku:

Якщо ще немає облікового запису на Heroku, то спочатку треба зареєструватися на їхньому веб-сайті (https://www.heroku.com/).

2. Встановити Heroku CLI:

Нам знадобиться Heroku Command Line Interface (CLI) для взаємодії з Heroku з нашого локального комп'ютера. Встановімо Heroku CLI та авторизуймося в ньому за допомогою команди heroku login.

3. Підготуймо наш проект React-сайту:

Впакуймо React-проект в директорію, яка готова для розгортання. Переконаймося, що еаш проект має файл package.json, в якому описані залежності та команда start, яка запускає наш додаток.

4. Ініціалізуймо Git репозиторій:

5. Якщо наш проект ще не має Git репозиторію, ініціалізуємо його за допомогою git init.

6. Створюємо файл Procfile:

В корені нашого проекту створімо файл з назвою Procfile (без розширення), в якому ми визначаємо команду, яку Heroku повинен виконати для запуску нашого додатку. Наприклад, якщо ми використовуємо Create React App, наш Procfile виглядатиме так:
web: npm start

7. Додаємо та фіксуємо зміни в Git:

Додаємо всі файли нашого проекту до Git репозиторію та робимо коміт.

8. Розгортаймо додаток на Heroku:

Використовуємо Heroku CLI, щоб створити новий додаток на Heroku і розгорнути його: heroku create, та git push heroku master

9. Відкриймо додаток:

Наш сайт тепер розгорнутий на Heroku. Ми можемо відкрити його у браузері за допомогою команди: heroku open

Також ми можемо легко масштабувати наш додаток на Heroku, змінюючи кількість "Dynos" за допомогою команд Heroku CLI, додаючи кількість потужностей.

2.7. Застосування сервісів штучного інтелекту в процесі написання коду

2.7.1. Використання ChatGPT для створення CRM-системи

Ось декілька прикладів того, як ChatGPT може допомогти розробникові при створенні CRM-системи:

- Користувацький Чат: Використання ChatGPT для реалізації чат-інтерфейсу, де користувачі можуть задавати питання та отримувати інформацію щодо їхніх замовлень, статусу доставки тощо.

```
// Приклад виклику API ChatGPT для обробки користувацьких запитань
const handleUserChat = async (userInput) => {
  const response = await callChatGPTAPI(userInput);
  // Обробка відповіді та відображення користувачеві
  displayChatResponse(response);
};
```

- Обробка Інструкцій Користувача: Використовуючи ChatGPT для аналізу інструкцій користувача або текстових коментарів, щоб автоматично розуміти та виконувати відповідні дії.

```
// Аналіз інструкцій користувача з використанням ChatGPT
const processUserInstructions = async (instructions) => {
  const action = await analyzeInstructions(instructions);
  // Виконання відповідної дії в CRM-системі
  performAction(action);
};
```

- Автоматична Обробка Запитань про Замовлення: Використання ChatGPT для автоматичної обробки питань про статус замовлення, інформацію про товари та інше.

```
// Обробка запитань про замовлення за допомогою ChatGPT
const processOrderInquiries = async (orderID) => {
  const orderInfo = await getOrderInformation(orderID);
  // Відображення інформації про замовлення користувачеві
  displayOrderInformation(orderInfo);
};
```

- Оптимізація Користувацького Досвіду: Використання ChatGPT для аналізу користувацьких відгуків та пропозицій для подальшої оптимізації і покращення функціональності CRM.

```
// Аналіз відгуків користувачів з використанням ChatGPT
const analyzeUserFeedback = async (userFeedback) => {
  const insights = await gatherUserInsights(userFeedback);
  // Використання отриманих інсайтів для покращення функціональності CRM
  improveCRMFunctionality(insights);
};
```

Важливо зауважити, що при використанні ChatGPT важливо враховувати його обмеження і брати до уваги безпеку, особливо коли обробляються конфіденційні дані.

А також при розробці можна і було використано:

- Побудова бізнес-плану системи

- Домомога в плануванні архітектури
- Написання частин коду
- Побудова аналітики, графіків, адаптація даних

2.7.2. Використання GitHub Copilot для написання коду CRM-системи

GitHub Copilot - це інструмент, який створений для автоматичної генерації коду на основі коментарів та контексту нашої роботи. Це може значно полегшити процес розробки, зокрема написання фрагментів коду та рутинних задач. Однак важливо враховувати, що результати, які Copilot надає, повинні бути перевірені та адаптовані відповідно до наших потреб. Ось кілька кроків, як використовувати GitHub Copilot для написання частин системи:

- Встановлення та Налаштування:
  - Встановлюємо Copilot в редакторі коду (наприклад, у Visual Studio Code).
  - Слідуємо інструкціям Copilot для налаштування.
- Створення Файлу або Коментарів:
  - Створюючи новий файл або вибравши існуючий, до якого ми хочемо додати код.
  - В блоці коментарів вводимо опис того, що нам потрібно.

    // Опис функціоналу CRM-системи:

    // Вхідні дані - замовлення користувачів.

    // Вихідні дані - аналіз та обробка даних для різних частин системи.
- Використання Copilot для Генерації Коду:
  - Починаючи писати коментарі чи опис коду Copilot буде намагатися автоматично доповнювати наш код.

    // Створення функції для аналізу замовлень:

    function analyzeOrders(orders: Order[]): AnalyticsResult {

     // Copilot може запропонувати автоматичне завершення коду для аналізу замовлень.

```
  // ...
}
```

- Перевірка та Адаптація:
    - Завжди необхідно перевіряти та адаптувати код, який генерує Copilot, оскільки він може не завжди правильно враховувати наші потреби коректно.

```
// Адаптація функції для аналізу замовлень під конкретні вимоги системи.
function analyzeOrders(orders: Order[]): AdvancedAnalyticsResult {
  // Наш власний код для аналізу замовлень.
  // ...
}
```

- Поступова Розробка:
    - Розробка системи може бути поступовою. Використовуймо Copilot для швидкого написання базових блоків коду, а потім доповнюємо та розширюємо їх відповідно до потреб проекту.

```
// Розширення функціоналу для аналізу замовлень з урахуванням нового фільтру.
function analyzeOrders(orders: Order[], filter: OrderFilter): AdvancedAnalyticsResult {
  // Додавання нового фільтру в аналіз замовлень.
  // ...
}
```

ВИСНОВОК ДО РОЗДІЛУ 2

У даному розділі ми детально прописали усі етапи створення менеджерської системи управління бізнес-проектами на прикладі співпраці з інтернет-магазином. Було вибрано необхідні технології, такі як React для клієнтського інтерфейсу, NestJS для серверного функціоналу, та базу даних MongoDB. Було ретельно пропрацьовано усі складові проекту:

1. Обрано дизайн системи.
2. Ініціалізовано клієнтський інтерфейс системи з використанням бібліотеки React.
3. Створено усі необхідні компоненти сайту, такі як: найменші атомарні компоненти, як наприклад поле вводу; компоненти компоновки layout; та високорівневі компоненти цілих сторінок системи.
4. Ініціалізовано серверну інфраструктуру за допомогою сучасного Node.js фреймворку – NestJS.
5. Створено усі API вузли для обробки HTTP-запитів з клієнтського інтерфейсу
6. Описано роботу з даними в базі даних MongoDB
7. Та запущено проект в загальний доступ, розмістивши його на хостинг-сервісі Heroku

РОЗДІЛ 3. ОПИС ТА АНАЛІЗ ГОТОВОГО ПРОЕКТУ

3.1. Аналіз роботи системи замовлень

У клієнта-замовника на веб-сайті його інтернет-магазину ми можемо оформити замовлення на будь-які речі. Для цього користувачі заходять на веб-сайт у розділ “Магазин” і обирає категорію товарів “Брюки”, обирає один із представлених товарів обраної категорії:

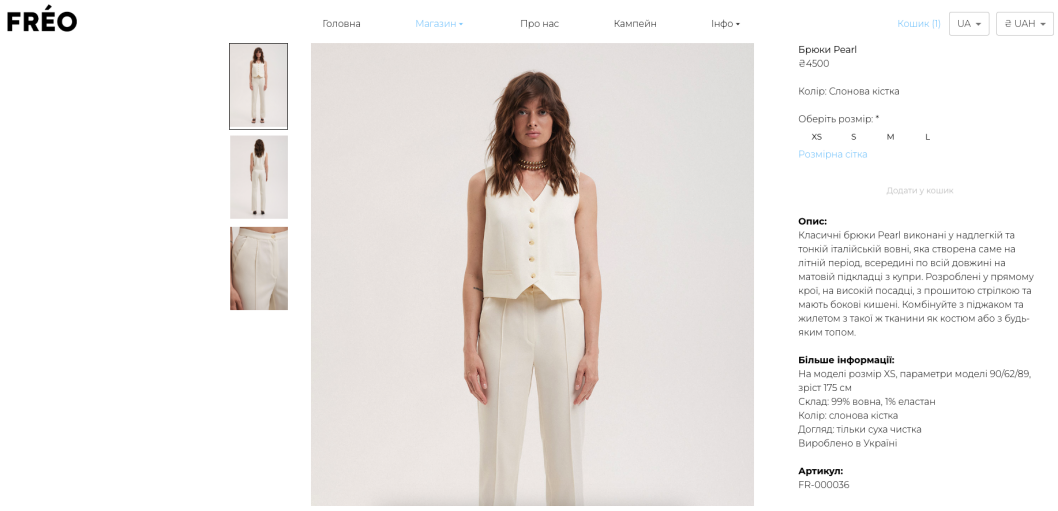

Рис. 3.1. Сторінка сайту інтернет-магазину клієнта

Для оформлення замовлення оберемо розмір речі, та додамо його в кошик:

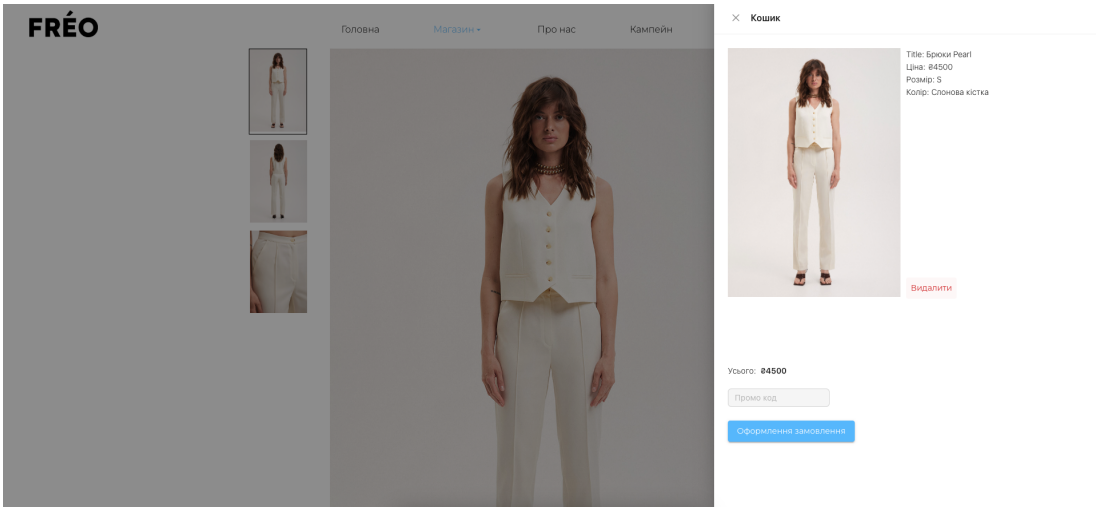

Рис. 3.2. Кошик інтернет-магазину клієнта

Після чого клієнт заповнює форму замовлення і натискає кнопку “Submit”:

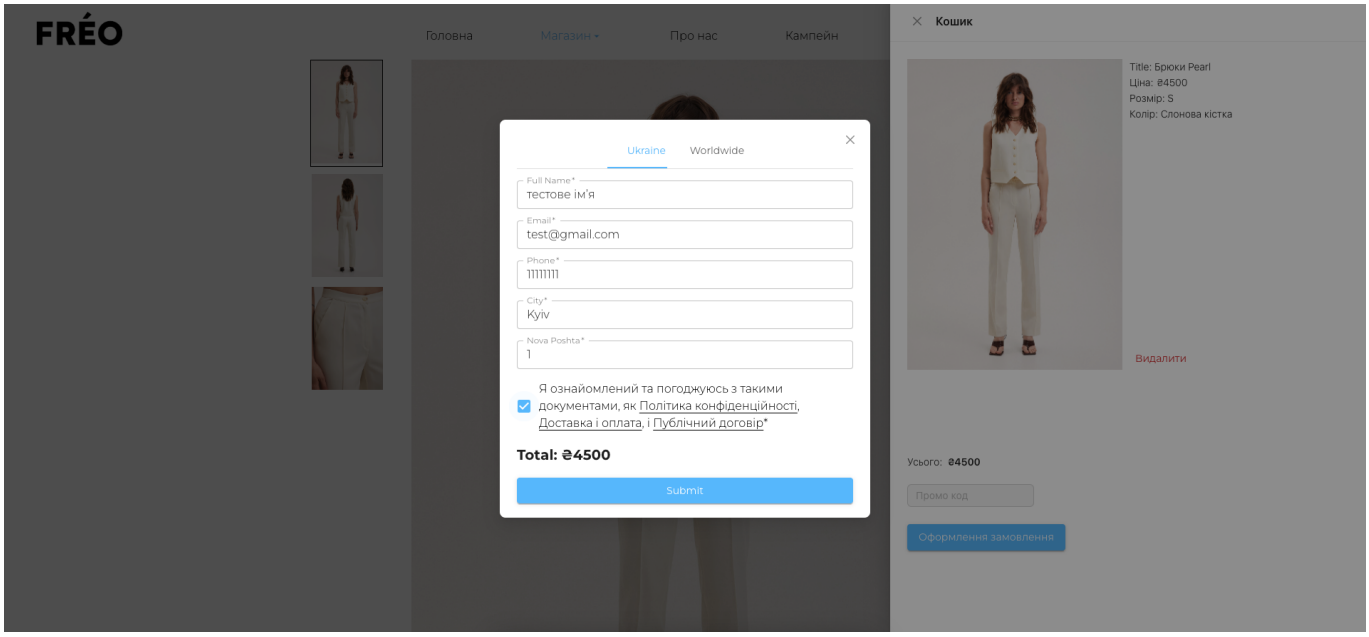

Рис. 3.3. Форма даних покупця у інтернет-магазині клієнта

Після цього сервер нашої системи управління взаємодіями з клієнтами у бізнесі отримує HTTP запит методом POST для створення нового замовлення:

POST /order HTTP/1.1

Accept: */*

Connection: keep-alive

Content-Length: 444

Content-Type: application/json

Host: localhost:8000

Origin: http://localhost:3000


```
approved: false
createdDateString: "2023-11-24T09:35:54.701Z"
currency: "uah"
delivery: 4500
items: [{id: "FR-000036", size: "S", title: "Брюки Pearl", additionalParams: []}]
  0: {id: "FR-000036", size: "S", title: "Брюки Pearl", additionalParams: []}
orderData: {fullName: "тестове ім'я", email: "test@gmail.com", phone: "11111111", city: "Kyiv", novaPoshta: "1",…}
  agreement: true
  city: "Kyiv"
  currency: "uah"
  email: "test@gmail.com"
  fullName: "тестове ім'я"
  novaPoshta: "1"
  phone: "11111111"
orderId: "d443635d-bd6f-48c0-b804-8ec68cbc45e9"
orderType: "ukraine"
price: 4500
promo: null
```


Рис. 3.4. Стуктура даних тіла запиту на створення нового замовлення

Після чого сервер нашої системи записує дане замовлення в базу даних:


```
approved: false
orderId: "d443635d-bd6f-48c0-b804-8ec68cbc45e9"
items: Array (1)
price: 4500
orderType: "ukraine"
email: "test@gmail.com"
fullName: "тестове ім'я"
phone: "11111111"
city: "Kyiv"
novaPoshta: "1"
agreement: true
promo: null
createdDateString: "2023-11-24T09:35:54.701Z"
delivery: 4500
currency: "uah"
__v: 0
```


Рис. 3.5. Стуктура даних замовлення в базі даних MongoDB Atlas

І тепер ми можемо побачити дане замовлення в нашій системі управління взаємодіями з клієнтами:

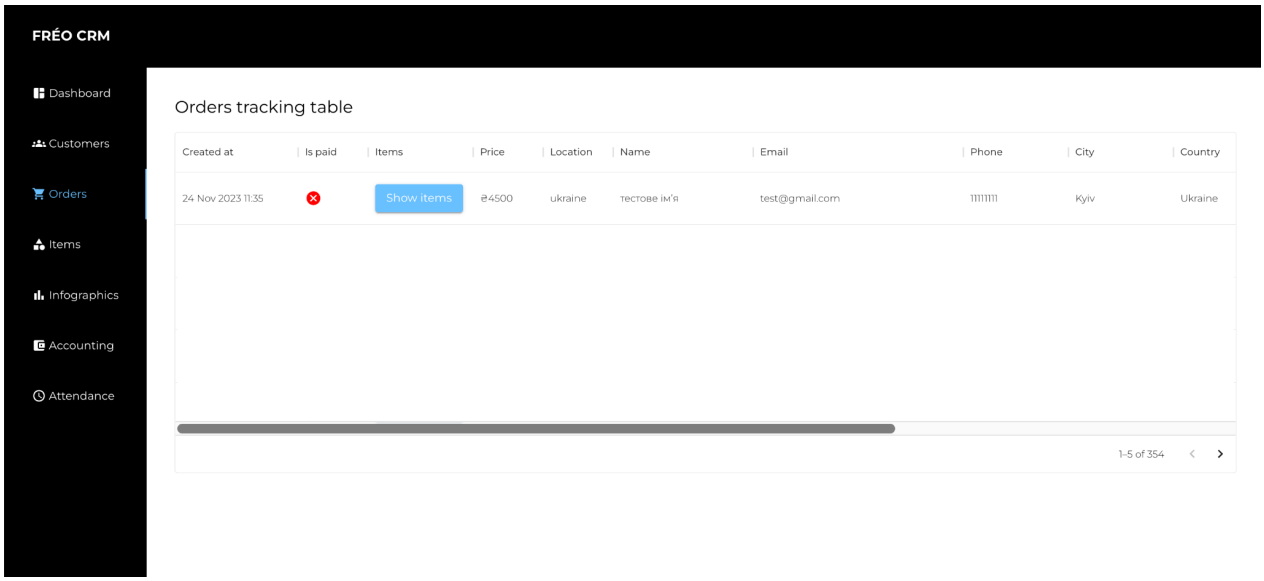

Рис. 3.6. Розділ замовлень системи управління взаємодіями з клієнтами

Таблиця замовлень в CRM системі інтернет-магазину виконує кілька важливих функцій, що сприяють ефективній роботі та управлінню бізнесом. Ось кілька ключових причин, чому ця таблиця є важливою:

- Відстеження замовлень: Таблиця замовлень дозволяє вам легко відстежувати всі замовлення, які роблять ваші клієнти. Це полегшує контроль над процесом замовлення від початку до кінця.
- Аналіз ефективності: Збір та аналіз даних з таблиці замовлень може надати корисну інформацію щодо того, як ефективно працює ваш інтернет-магазин. Ви можете вивчати популярні товари, розпізнавати тенденції попиту та визначати, які маркетингові стратегії найбільше ефективні.
- Взаємодія з клієнтами: Ваша CRM система може автоматично виводити дані з таблиці замовлень для покращення обслуговування клієнтів. Наприклад, вона може надсилати сповіщення про статус замовлення, підтвердження оплати, інформацію про відправлення тощо.

- Управління запасами: З даної таблиці можна отримати інформацію про кількість продуктів, які були замовлені. Це може бути корисним для управління запасами, забезпечення належного рівня товарів і уникнення ситуацій, коли товар вийшов з обігу.
- Аналітика та звітність: Дані з таблиці можуть використовуватися для створення різноманітних аналітичних звітів. Це дозволяє керівництву отримувати зрозумілу інформацію щодо продажів, прибутковості, конверсій і інших ключових показників.
- Покращення обробки замовлень: Інтеграція таблиці замовлень з іншими системами дозволяє автоматизувати процес обробки замовлень, зменшуючи можливість помилок та оптимізуючи час виконання замовлення.

3.2. Застосування математичних моделей штучного інтелекту до системи контролю замовлень

Для застосування методів штучного інтелекту в проекті використовується ChatGPT API. Для аналітики замовлень необхідно передати вибірку даних замовлень до ChatGPT API наступним чином:

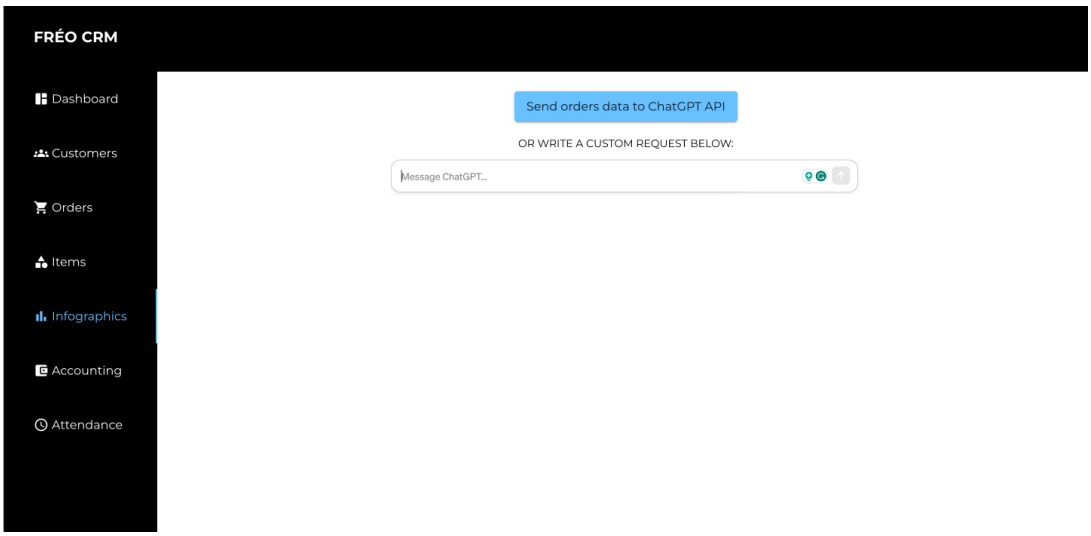

Рис. 3.7. Розділ аналітики CRM-системи

Настискаючи на кнопку відправлення даних замовлень у математичну модель ШІ вона, попередньо запрограмована на підготовку даних, надсилає нам дані для аналітичних графіків і наша система оновлюється та відображає графіки:

Рис. 3.8.1. Аналітичні дані замовлень у графіках

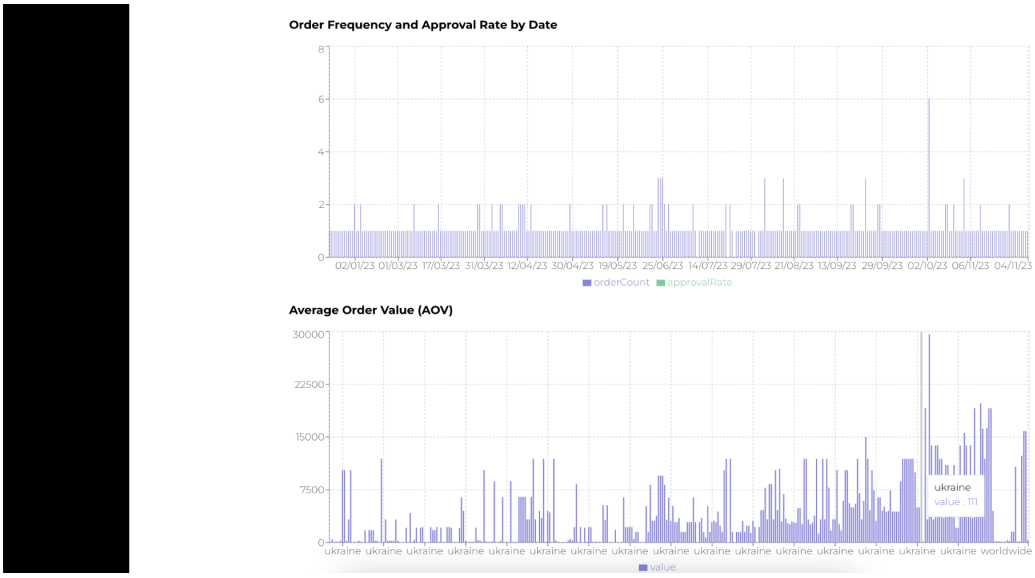

Рис. 3.8.2. Аналітичні дані замовлень у графіках

На майбутнє можна все більше і точніше робити запити до математичної моделі ШІ, уточнюючі дані, змінюючи вибірку, тощо.

3.3. Огляд головної сторінки системи Dashboard

Головна сторінка Dashboard грає дуже важливу роль, адже на неї виводиться ключова агрегована інформація, яку можна використати не шукаючи її в “глибинах” системи розміщену по інших розділах.

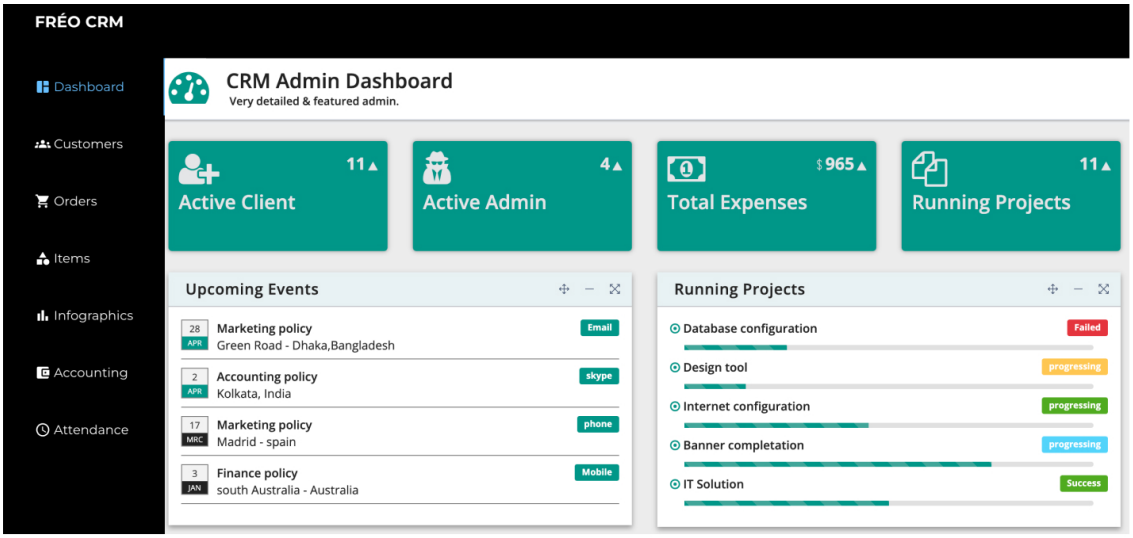

Рис.3.9.1. Головна сторінка CRM-системи 1

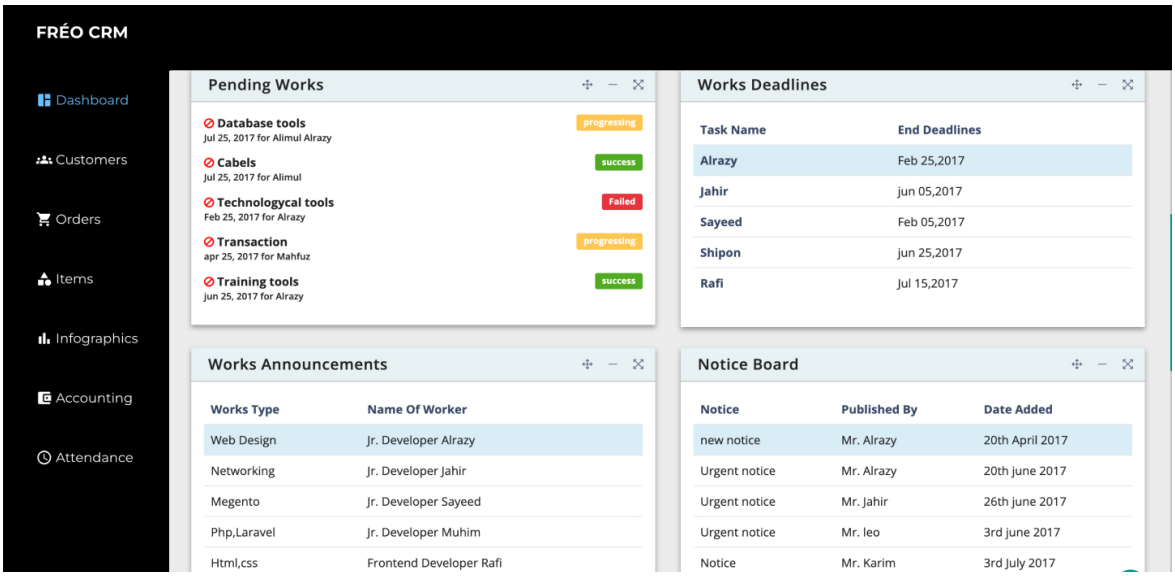

Рис.3.9.2. Головна сторінка CRM-системи 2

Ця сторінка є повністю керованою і на неї можна виводити будь-які підрозділи, які бізнесу було б зручно мати разом на одній сторінці. Як-от наприклад за замовчуванням я відобразив розділи блоки з кількістю активних клієнтів на сайті бізнесу, кількості адміністраторів в системі, загальні витрати бізнесу, та активні завдання. А також активні події, проекти, дедлайни, тощо. Тобто загальні відомості з різних розділів системи.

3.4. Розділ фінансового обліку

В розділі фінансового обліку ми можемо слідкувати та аналізувати все, що стосується фінансових операцій між клієнтами бізнесу та бізнесом. Тобто аналіз і менеджмент прибутків від продажів, успішні або неуспішні транзакції, тощо. Ось так виглядає даний розділ в нашій системі:

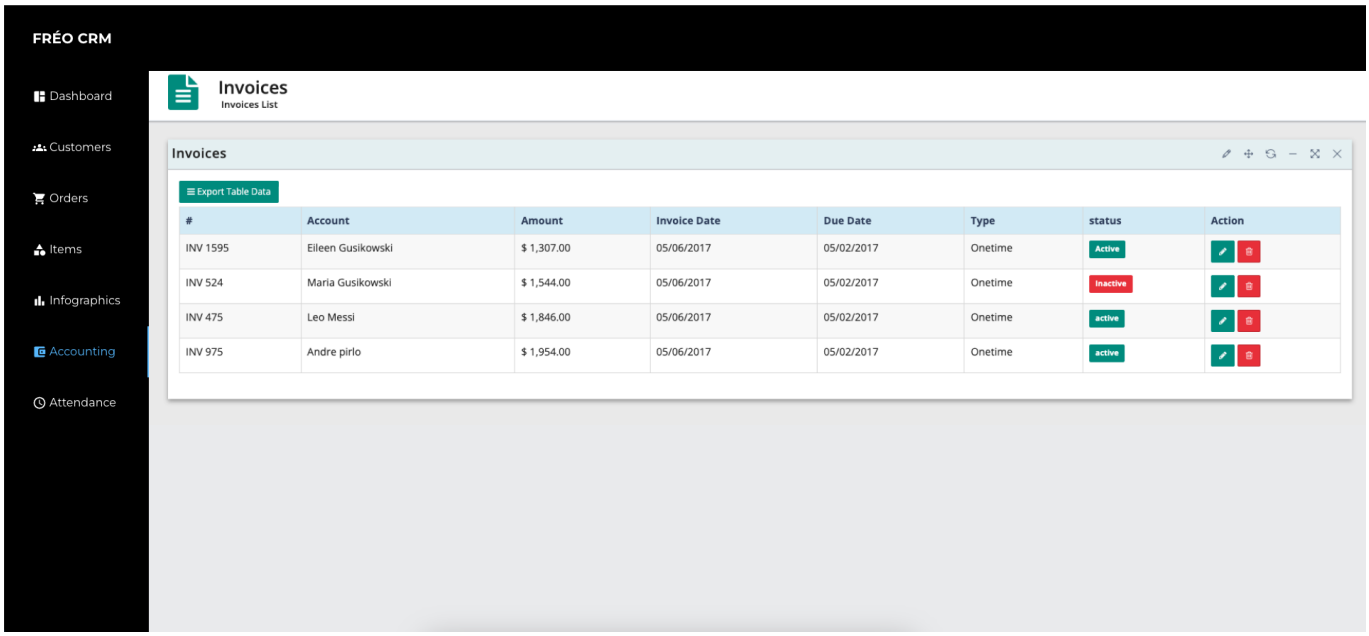

Рис. 3.10. Розділ фінансового обліку

Фінансові дані для даного розділу взяті тестові, адже справжні дані - це закрита інформація, доступна тільки для співробітників бізнесу.

3.5. Розділ управління товарами

Також є важливою можливість створювати та змінювати будь-які дані про товари бізнесу, до якого ми підключаємо нашу систему. В нашому випадку це регулювання цін на товари інтернет-магазину, з яким ми співпрацюємо, назв товарів, регулювання знижок, зміна категорії товару, кількість товарів, чи закінчився товар і нових не буде, чи товару немає в наявності але скоро він поступить в продаж, тощо. А також важлива можливість створювати нові товари на сайті, аби не доводилося вручну їх створювати в базі даних. Даний розділ в нашій системі реалізовано наступним чином:

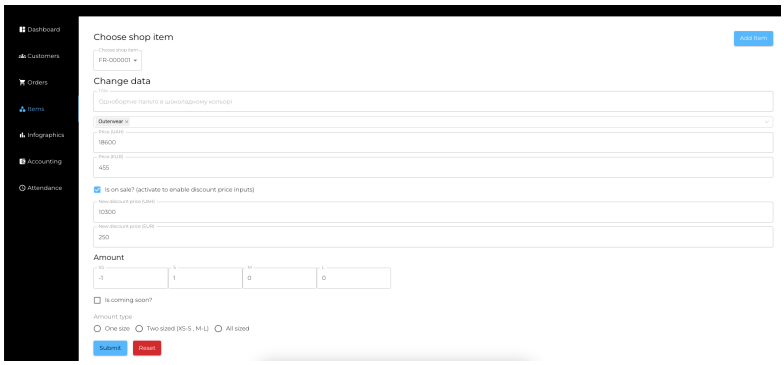

Рис. 3.11.1. Розділ управління товарами

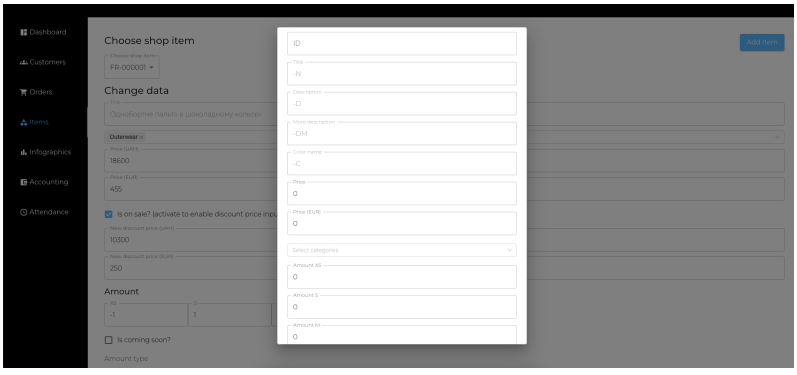

Рис. 3.11.2. Форма створення нових товарів

ВИСНОВОК ДО РОЗДІЛУ 3

У даному розділі було проілюстровано та розглянуто результат роботи над нашою системою. Було описано процес взаємодії з бізнесом-замовником, починаючи від того, коли клієнт бізнесу створив нове замовлення на покупку товарів, демонстрацією, як ці замовлення потрапляють в нашу базу даних, тощо, та завершуючи тим, як менеджери бізнесу бачать та взаємодіять із замовленнями в нашій системі, а саме:

- Отримують списки всих замовлень
- Мають можливість відфільтрувати сплачені від несплачених
- Можуть видаляти замовлення
- Аналізувати та передавати замовлення на доставку

Також було розглянуто розділи системи, такі як: головна сторінка для швидкого доступу до ключових частин окремих розділів, розміщених на одній сторінці; розділу фінансового обліку з усима транзакціями; та розділ управління та створення товарів, де менеджери можуть змінювати параметри наявних у бізнеса товарів, та створювати нові.

Також було наведено приклад, як в даній системі можна застосовувати математичні моделі штучного інтелекту для роботи з неопрацьованими даними про замовлення або ж фінансові транзакції, та як адаптуючи ці дані можна отримувати графіки для роботи з аналітикою бізнесу.

ВИСНОВКИ

У підсумку, проведений аналіз та розгляд різноманітних аспектів проекту "Система управління бізнесом з аналітикою та оптимізацією на базі ШІ" дозволяє зробити висновок про його високий рівень актуальності та значущості в контексті сучасного бізнес-середовища.

Перш за все, проаналізованою концепцією було визначено, що управління бізнесом потребує інтегрованих інноваційних рішень, спрямованих на покращення різноманітних аспектів, включаючи управління замовленнями, аналітику продажів, оптимізацію логістики та використання сучасних ШІ-технологій.

Другий аспект, пов'язаний із створенням менеджерської системи управління бізнес-проектами для інтернет-магазину, демонструє ретельно розроблений план дій, вибір оптимальних технологій (React, NestJS, MongoDB) та етапи реалізації проекту. Зазначені завдання включають створення комплексного рішення, інтеграцію з ШІ-сервісами, забезпечення безпеки та конфіденційності, що визначає високий рівень вимог до продуктивності та аналітики для прийняття рішень.

Нарешті, третій етап ілюструє результати роботи системи в контексті співпраці з бізнесом. Відзначено ефективність взаємодії з бізнес-замовником, від початкового створення замовлення до його обробки в системі та подальшого аналізу та управління. Розглянуто розділи системи, зокрема головну сторінку, фінансовий облік та управління товарами, де надано приклад використання математичних моделей штучного інтелекту для аналізу даних.

Загалом, проект демонструє свою вагому цінність, розроблену інфраструктуру та готовність до впровадження в реальні умови, що робить його перспективним та конкурентоспроможним у сучасному бізнес-середовищі.

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ

1. Документація системи Salesforce CRM [електронний ресурс] 2023. – Режим доступу: https://www.salesforce.com/crm/
2. Документація системи Zoho CRM [електронний ресурс] 2023. – Режим доступу: https://www.zoho.com/crm/
3. Документація системи HubSpot CRM [електронний ресурс] 2023. – Режим доступу: https://www.hubspot.com/products/crm
4. Документація системи Bitrix24 [електронний ресурс] 2023. – Режим доступу: https://www.bitrix24.eu/tools/crm/
5. Клієнт-серверна архітектура [електронний ресурс] 2023. – Режим доступу: https://en.wikipedia.org/wiki/Client%E2%80%93server_model
6. What is a Web App? by AWS [електронний ресурс] 2023. – Режим доступу: https://aws.amazon.com/what-is/web-application/#:~:text=A%20web%20application%20is%20software,with%20customers%20conveniently%20and%20securely.
7. Стаття “Фреймворки у веб-розробці — що це, які існують і для чого потрібні” електронного журналу Highload [електронний ресурс] 2023. – Режим доступу: https://highload.today/uk/frejmvorki-u-veb-rozrobtsi-shho-tse-yaki-isnuyut-i-dlya-chogo-potribni/