БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

15.03 — КР. 2249 “С” 2024.12.16 16 ПЗ

ДУМАНСЬКИЙ ВЛАДЛЕН ЮРІЙОВИЧ

2025 р.

НАЦІОНАЛЬНИЙ УНІВЕРСТИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ

Завідувач кафедри
комп'ютерних наук
(назва кафедри)

_____ Голуб Б. Л.
(підпис) (ПІБ)

“___” _____ 20 р.

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА
на тему

«CRM-система для малого і середнього бізнесу»

Спеціальність 121 – «Інженерія програмного забезпечення»

Гарант освітньої програми
к.н.т., доцент _____ Вайганг Г.О
(науковий ступінь та вчене звання) (підпис) (ПІБ)

Керівник бакалаврської кваліфікаційної роботи
ст. викладач _____ Василюк-Зайцева С. В.
(науковий ступінь та вчене звання) (підпис) (ПІБ)

Виконав _____ Думанський Владлен Юрійович
(підпис) (ПІБ студента)

КИЇВ – 2025

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
Факультет інформаційних технологій

ЗАТВЕРДЖУЮ
Завідувач кафедри
Комп'ютерних наук
к.н.т., доцент _____ Голуб Б.Л.
(науковий ступінь, вчене звання) (підпис) (ПІБ)
"_3" червня 2025 р.

ЗАВДАННЯ
на виконання бакалаврської кваліфікаційної роботи студенту
Думанський Владлен Юрійович
(прізвище, ім'я, по батькові)

Спеціальність 121 – «Інженерія програмного забезпечення»

Тема бакалаврської кваліфікаційної роботи CRM-система для малого і середнього бізнесу

Затверджена наказом ректора НУБіП України від "16" грудня 2024 р. №2249 С

Термін подання завершеної роботи на кафедру 2025.05.25
(рік, місяць, число)

Вихідні дані до бакалаврської кваліфікаційної роботи
Опис предмету дослідження, опис програмного забезпечення

Перелік питань, які потрібно розробити:
Системний аналіз предметної області
Аналіз предметної області
Розробка програмного забезпечення
Рекомендації щодо впровадження та експлуатації системи

Дата видачі завдання "16" грудня 2024 р.

Керівник бакалаврської кваліфікаційної роботи
_____ Василюк-Зайцева С. В.
(підпис) (прізвище та ініціали)

Завдання прийняв до виконання _____ Думанський В.Ю.
(підпис) (прізвище та ініціали студента)

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ПЗ - Програмне забезпечення.

ІС - Інформаційна система.

СУБД - Система управління базою даних.

БД - База даних.

SQL - Structured Query Language, мова запитів до баз даних.

API - Application Programming Interface, програмний інтерфейс.

UI - User Interface, інтерфейс користувача.

JSON - JavaScript Object Notation, формат обміну даними.

REST - Representational State Transfer, архітектурний стиль API.

JWT - JSON Web Token, формат токена автентифікації.

UML - Unified Modeling Language, мова моделювання.

IDE - Integrated Development Environment, середовище розробки.

CI/CD - Continuous Integration / Continuous Deployment, безперервна інтеграція та розгортання.

SPA - Single Page Application, односторінковий застосунок.

TS/JS - TypeScript / JavaScript, мови програмування

Nuxt / Vue - веб-фреймворки на базі мови JavaScript

ВСТУП

У сучасному світі більшість бізнес-процесів активніше переходить в діджитал-формат. Особливо це стосується підприємств малого та середнього бізнесу, для яких ефективність внутрішньої організації, контроль за роботою з клієнтами та автоматизація рутинних дій є критичними факторами виживання і зростання. В той самий час, більшість існуючих CRM-систем є занадто громіздкими. Вони орієнтовані на великі підприємства або мають погану гнучкість у налаштуванні під специфіку конкретного бізнесу. Все це створює значний розрив між реальними потребами малого бізнесу та можливостями доступних програмних рішень.

Актуальність даної розробки полягає в неохідності створення адаптивної, простої у використанні та в той самий час функціонально розширюваної CRM-системи, що дозволяє малому бізнесу керувати не лише базою клієнтів, але й внутрішніми процесами без залучення технічних фахівців та додаткового програмування. Основна особливість запропонованого рішення - це гнучка структура, коли користувач має можливість самостійно створювати типи сутностей, визначити поля, татуси, логіку автоматизації та спостерігати історію змін. І все це в межах однієї системи без сторонніх інтеграцій.

Мета даного проєкту полягає у створенні програмної системи, яка буде орієнтована на потреби малого та середнього бізнесу, забезпечить універсальний підхід до побудови облікових процесів. CRM-система зможе значно пришвидшити бізнес-процеси, а також дозволить керувати взаємодією клієнтами, замовленнями, угодами тощо. Будь-які бізнес-одиниці створюються просто та швидко. Серед ключових елементів системи можна зазначити модульну архітектуру, вбудовану систему автоматизації дій (на основі тригерів і умов), журнал змін для прозорого

контролю, а також можливість підключення зовнішніх сервісів через webhook-інтеграції.

Поточний проєкт реалізовано у межах науково-дослідної діяльності кафедри комп’ютерних наук факультету інформаційних технологій, відповідно до затвердженого плану тематичних робіт.

Розробку програмного забезпечення було поділено на декілька основних етапів: аналіз предметної області, моделювання інформаційної структури, проєктування архітектури системи, розробка функціональних модулей та тестування. Для об’єктного моделювання було використано діаграми класів, кооперацій, компонентів, пакетів і логічну ER-модель. У процесі проєктування особливо уважно було вивчено аспект універсальності моделі збереження даних (через динамічне поле json) та масштабованості архітектури.

Пояснювальна записка має структуру з чотирьох основних розділів:

1) Перший розділ містить системний аналіз предметної області та постановка завдання.
2) Другий розділ описує моделювання, архітектуру та логіку.
3) Третій розділ містить інформацію про розробку системи, вибір технології та реалізацію модулів.
4) Четвертий розділ охоплює тестування, оцінку якості системи, апаратні вимоги та висновки.

1 СИСТЕМНИЙ АНАЛІЗ УПРАВЛІННЯ БІЗНЕС-ПРОЦЕСАМИ У МАЛОМУ ТА СЕРЕДНЬОМУ БІЗНЕСІ

1.1 Опис предметної області

Малий та середній бізнес у XXI сторіччі потребує ефективних технологічних рішень для ведення робочих процесів, клієнтського обліку, контролю процесів продажів, постановки завдань працівниками та аналізу ефективності. CRM-системи (системи управління взаємовідносинами клієнтами - Customer Relationship Management) - це надзвичайно важливі системи, які забезпечують стабільний розвиток, а також підвищують продуктивність команди, допомагають масштабувати бізнес та збільшити прибуток [1]. Але більшість сучасних CRM-систем, які орієнтовані великі компанії та підприємства, мають багато надлишкового функціоналу, дуже складну структуру та фіксовану логіку роботи з клієнтами, товарами, угодами, та будь-якими іншими об’єктами. Досить часто це не задовольняє потреби малого та середнього бізнесу (МСБ), у якому потрібна гнучкість та простота в адаптації до специфіки бізнесу та його процесів.

У рамках бакалаврської кваліфікаційної роботи створюється CRM-система нового формату. Вона не прив’язана до конкретних типів об’єктів (наприклад контактів, задач, лідів, товарів чи угод). Насамперед система надає можливість користувачам власноруч створювати універсальні типи сутностей (Entity Types), які будуть відповідати специфіці їхнього бізнесу. Всі сутності мають певний набір користувацьких полів (User Fields) - (число, текст, email, дата та час, зв’язок з іншими сутностями, логічні значення тощо). Окремо типи сутностей повинні мати набір власних статусів, бо статус сутності - це першочергова річ, від якої залежить автоматизація. Зручна CRM-система повинна мати підтримку канбан

подання даних або списочного формату, фільтрацію даних, аналітику та історію змін, а також зручну і просту автоматизацію бізнес процесів.

До проблематики даної предметної області відноситься відсутність гнучкості у сучасних CRM-системах. Велика частина систем примушують прив'язуватися до певних моделей даних, які часто неможливо чітко адаптувати під свої потреби. В свою чергу це змушує компанії використовувати недостатньо зручні системи, або навпаки, використовувати забагато надлишкового функціоналу. Також можна зазначити занадто високу складність існуючих CRM-систем. У багатьох випадках бізнесу доводиться адаптуватися під логіку їхньої системи, хоча повинно бути навпаки - система має бути адаптована під бізнес. Користувачам потрібен гнучкий функціонал, який надає можливості для масштабування без зайвого функціоналу та швидких інтеграцій.

При розробці системи брався курс на мінімалізм, масштабованість та гнучкість. Адміністратор CRM-системи може:

1) Створювати безліч будь-яких типів сутностей (наприклад “Лід”, “Товар”, “Контакт”, “Холодні клієнти”).
2) Створити статуси та обрати для них кольори (Наприклад: Новий, Опрацювання, Завершено).
3) Додавати безліч полів різних типів.
4) Контролювати права доступу (адміністратор / менеджер).
5) Додавати користувачів / видаляти.

Основні

1) Універсальні сутності - створення будь-яких типів із довільними полями та статусами.
2) Канбан - візуальне представлення записів із групуванням по статусам.
3) Аналітика - графічні дані та метрики.

4) Система контролю доступу - розмежування доступу до певних дій між адміністраторами та менеджерами.
5) Інтерфейс автоматизацій - можливість автоматизації процесів та інтеграції з різними сервісами.

Запропонована CRM-система пропонує вирішення ключових проблем для малого та середнього бізнесу. Вона досить проста, має зрозумілу логіку та при цьому універсальна. Система надає можливість будувати власну структуру управління задачами, клієнтами, угодами та в цілому бізнес-процесами без необхідності у прив’язці до заздалегідь створених шаблонів. Це робить систему дуже ефективною, зручною та масштабованою для бізнесу незалежно від його галузі.

1.2. Аналіз вимог до програмної системи

Створена CRM-система повинна забезпечити універсальність, гнучкість, швидкість та зручність у побудуванні бізнес-процесів без прив’язки до заздалегідь визначених типів сутностей. Системні вимоги можна поділити на функціональні та нефункціональні.

Функціональні вимоги:

1) Автентифікація та реєстрація користувачів - Користувачі системи повинні зареєструватися, увійти до системи та мати змогу керувати власним профілем. Необхідність підтримки двох рівнів доступу: адміністраторський та менеджерський.
2) Управління типами сутностей - Адміністратор повинен мати змогу створювати будь-які типи сутностей та назначати власні назви, візуальне зображення для інтуїтивного розуміння назви (іконка) та опис сутності. Кожен тип сутності може мати безліч полів різних типів (текст, число, дата, email, прив’язка до іншої сутності, логічне значення тощо).
3) Управління полями типів сутностей - Кожне поле даних у сутності повинно мати свою кодову назву, назву українською

мовою, налаштування обов’язковості та плейсхолдер. Поля з типом select повинно мати опціональні значення для вибору. Поля з типом прив’язування до інших сутностей повинні мати налаштування, до якого саме типу сутностей, серед існуючих, воно прив’язується.

4) Управління статусами - Кожен тип сутностей повинен мати певний набір статусів із текстовою назвою та кольором. Один із статусів для кожного типу сутностей повинен бути за замовчуванням. Статуси повинні будувати використовуватися як основа для побудування канбану та мати помітне відображення у режимі списку. У режимі канбану кожен статус повинен мати окрему кнопку створення сутності з цим статусом задля швидкого та зручного управління даними.
5) Створення та редагування записів (сутностей) - Сутності повинні відображатися у вигляді канбану та списку. Менеджери та адміністратори повинні мати змогу додавати нові записи на основі визначених типів сутностей. Форми створення / редагування сутностей повинні формуватися автоматично на основі створених адміністратором полів.
6) Аналітика - Доступна базова аналітика по кількості записів для кожного типу сутностей, по кількості сутностей по кожному статусу. Наявність зручних та інтуїтивно простих графіків, які відображають динаміку створення та редагування сутностей, а також групування по базовим типам полів за вибором користувача.
7) Інтеграції та автоматизація - Зручний та дуже мінімалістичний інтерфейс для автоматизації процесів, що дозволяє будувати прості бізнес-процеси залежно від певних тригерів (створення, редагування тощо), задавати певні умови виконання цього

процесу та визначати певну дію (наприклад зміна значення поля або виклик зовнішнього ресурса через webhook). Архітектура для викликів webhook, що дозволяє надсилати “кастомні” сповіщення у будь-який сервіс (із залученням розробників для додаткової інтеграції). Лог активності - Простий лог активності, який зберігає в собі історію змін даних у системі. Інтеграція логу активності всередину карточки сутності у вигляді таймлайну.

Нефункціональні вимоги

1) Веб-інтерфейс - Система повинна мати адаптивний та інтуїтивно зрозумілий інтерфейс, який буде добре працювати на ПК та планшетах. UI системи повинен бути приємним та зручним для користувача.
2) Продуктивність - Списки сутностей і канбан мають підтримувати віртуальне підвантаження даних частинами, яке не буде помітним для користувача. Система повинна забезпечити швидке завантаження інтерфейсу та роботу без помітних для користувача затримок.
3) Безпека - Всі запити до бази даних мають бути реалізовані на серверній частині та підтримувати захист від екранування. Доступ адміністратора може надаватися лише довіреній особі та інші користувачі не можуть впливати на зміну даних у адміністративній частині. Всі дані між клієнтом та сервером передаються за допомогою SSL, паролі шифруються, а дані користувачів надійно захищаються у рамках сервера.
4) Масштабованість - Архітектура повинна мати підтримку створення нових типів полів, сутностей та статусів без змін у базовому коді ПЗ.

5) Сумісність - Система повинна працювати в середовищі сучасних веб-браузерів (Firefox, Chrome, Edge).

Обмеження: Наразі система тимчасово не має власного модуля реєстрації для всієї компанії, тому припускається використання одного проектного оточення для однієї компанії-власника.

1.3. Моделювання предметної області

Предметна область проєкту охоплює автоматизацію управління бізнес-процесами у малому та середньому бізнесі через створення гнучкої CRM-системи, яка не буде ніяк прив’язуватися до фіксованих структур даних. У рамках моделювання були виділені ключові об’єкти, певні взаємозв’язки та дії, які відбуваються між ними.

Визначити правильну модель інформаційної системи допоможе **діаграма прецедентів**, яку можна зробити за допомогою мови моделювання UML. Діаграма прецедентів (*англ. Use Case Diagram*) - це один із основних елементів моделювання об’єктної системи. Діаграма надає можливість графічно відобразити взаємодію між учасниками (акторами системи) у зовнішньому полі та функціональними можливостями програмної системи (тобто прецедентами), які вони можуть виконувати [2]. Цей підхід надає можливість наочно уявити, які користувачі будуть взаємодіяти із системою в рамках основних бізнес-сценаріїв.

У рамках поточного проєкту діаграма прецедентів використовується для опису ключових дій, які доступні користувачам CRM-системи. Вона є важливою не тільки на етапі вимог, а й є хорошим інструментом для всієї команди розробки. На основі цієї діаграми можна формулювати специфікації функціональних модулів, будувати тестові сценарції та масштабувати систему в майбутньому.

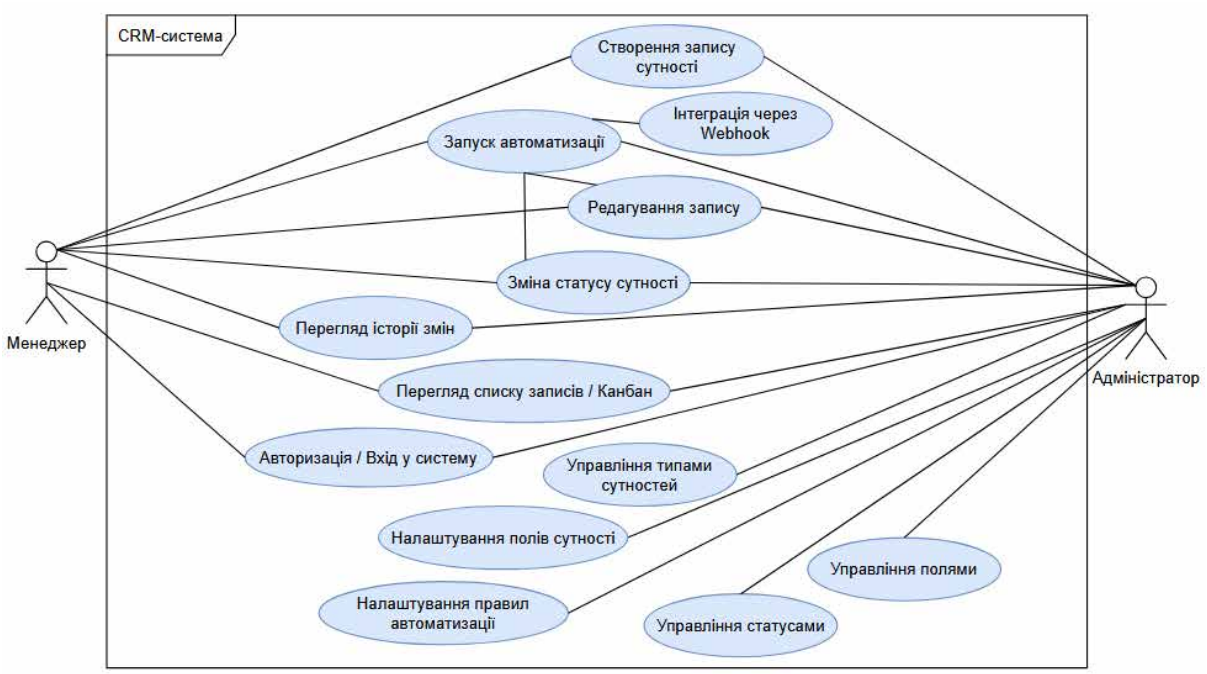

Рис. 1 Діаграма прецедентів

Система має наступних акторів:

1) **Менеджер (Manager):**

Користувач системи, який має базовий рівень доступу до управління сутностями. Може вносити дані у системи, ініціювати автоматизації за допомогою зміни даних, але не може впливати на систему в цілому.

2) **Адміністратор (Administrator)**

Користувач системи, який є адміністратором, тобто має всі права менеджера, та в той самий час може напряму впливати на систему через управління типами сутностей, статусами, полями, автоматизаціями, користувачами системи тощо.

Бізнес-процеси:

Актори мають певні можливості, тобто їх основні логічні можливості в контексті взаємодії з системою (прецеденти).

1) Адміністратор створює тип сутності та додає до нього поля та статуси.
2) Менеджери створюють записи сутностей через динамічні форми, зазначені адміністратором, які генеруються автоматично.

3) У час створення або зміни даних сутності можуть автоматично запускатися певні дії, які визначені у правилах автоматизації.
4) Дані мають два види відображення, канбан або список.
5) Усі зміни даних сутностей зберігаються у журналі активності.

Обрана модель архітектури: Система побудована з урахуванням правил розробки модульних систем та підтримує масштабованість. Сутності використовують канбан та списочну моделі, які дозволяють швидко візуалізувати стан об’єктів у системи. Разом з цим, автоматизація дозволяє розширити функціонал без необхідності додаткового програмування системи та залучення розробників ПЗ.

Модель предметної області орієнтована на забезпеченні гнучкоті та масштабованості та інтеграційності. Обрана структура дозволяє користувачам самостійно будувати CRM-процеси, які будуть адаптивними до специфіки бізнесу та не будуть прив’язуватися до певних жорстких шаблонів.

1.4. Огляд інформаційних джерел та існуючих рішень

Під час дослідження предметної області було розглянуто низку існуючих CRM-систем, як популярні міжнародні продукти, так і деякі менш відомі альтернативи з відкритим кодом (self-hosted). За мету під час аналізу було поставлено завдання для виявлення головних переваг та обмежень систем, які треба буде враховувати при розробці нової системи для малого та середнього бізнесу.

Zoho CRM можна виділити як одну з доступних рішень. Це дуже потужна система з надзвичайно широким спектром можливостей, інтеграцій та налаштувань. Вона пропонує великий набір функцій, проте має дуже складний інтерфейс та мають деякі обмеження у автоматизації. Також негативним фактором виступає занадто висока ціна, що може стати проблемою для малих компаній [3].

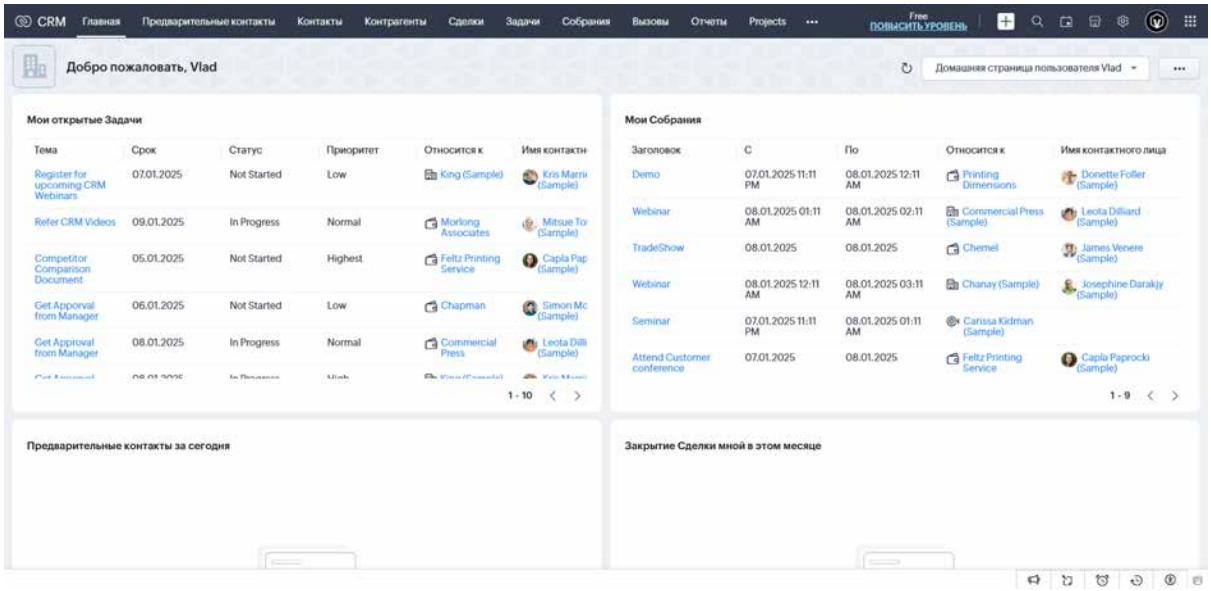

Рис. 2 CRM-система - Zoho CRM

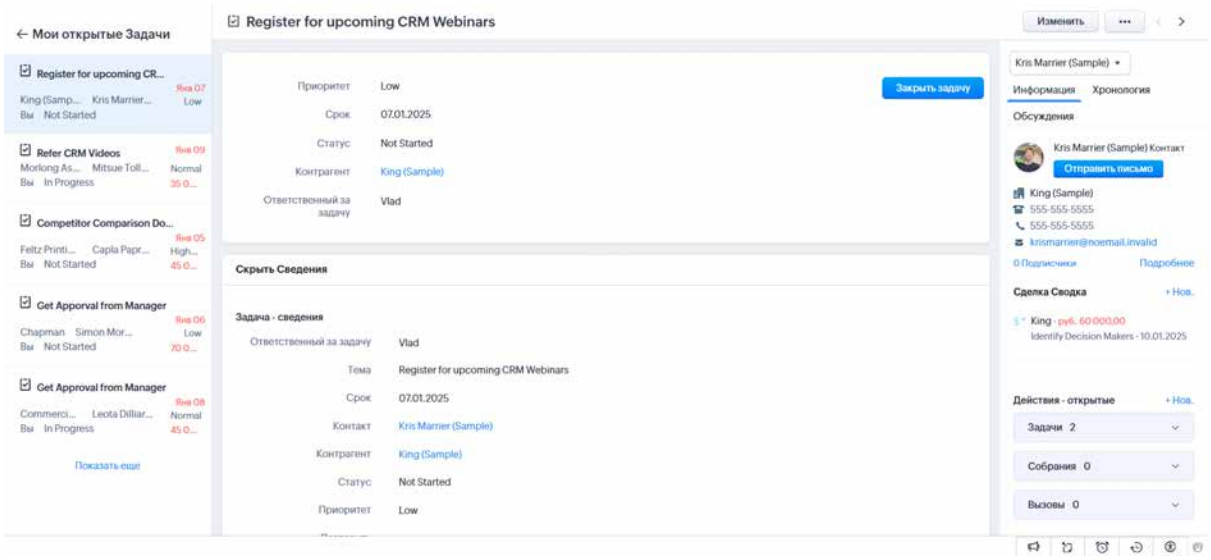

Рис. 3 Сторінка задач CRM-системи - Zoho CRM

HubSpot CRM - ще одне відоме рішення. У безкоштовній версії система дозволяє організовувати роботу з клієнтами, проте має велику кількість обмежень щодо кастомізації структур даних. Розширення функціоналу вимагає додаткових фінансових витрат на різні модулі. HubSpot надає досить зручний API-інтерфейс для розширення можливостей системи, але це вимагає залучення кваліфікаційного працівника [3].

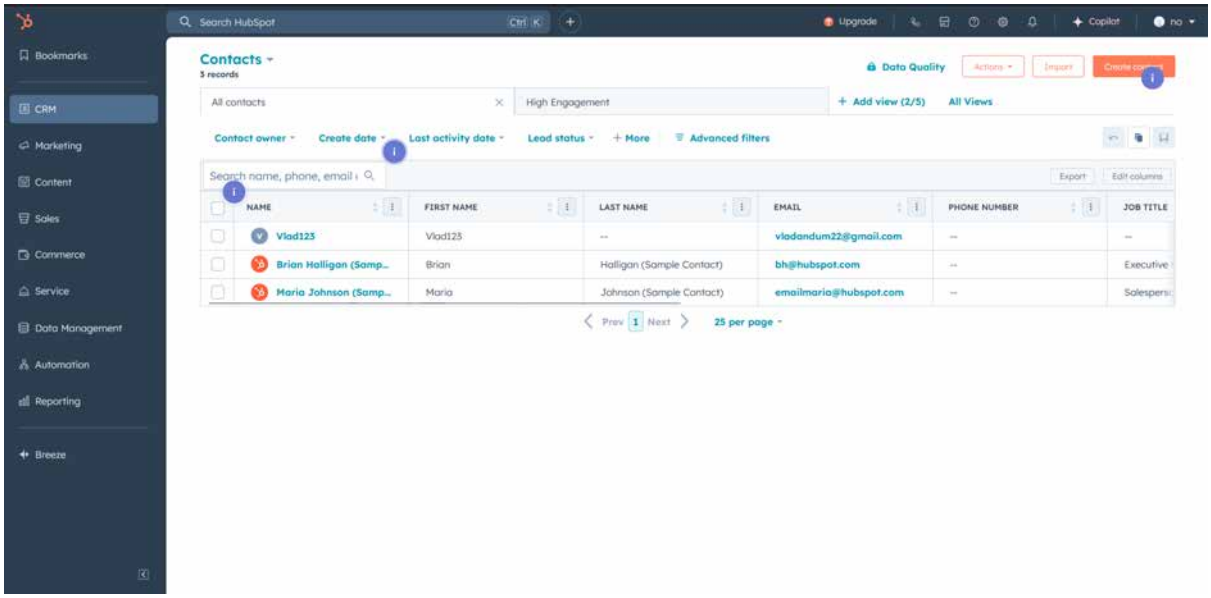

Рис. 4 CRM-система - HubSpot

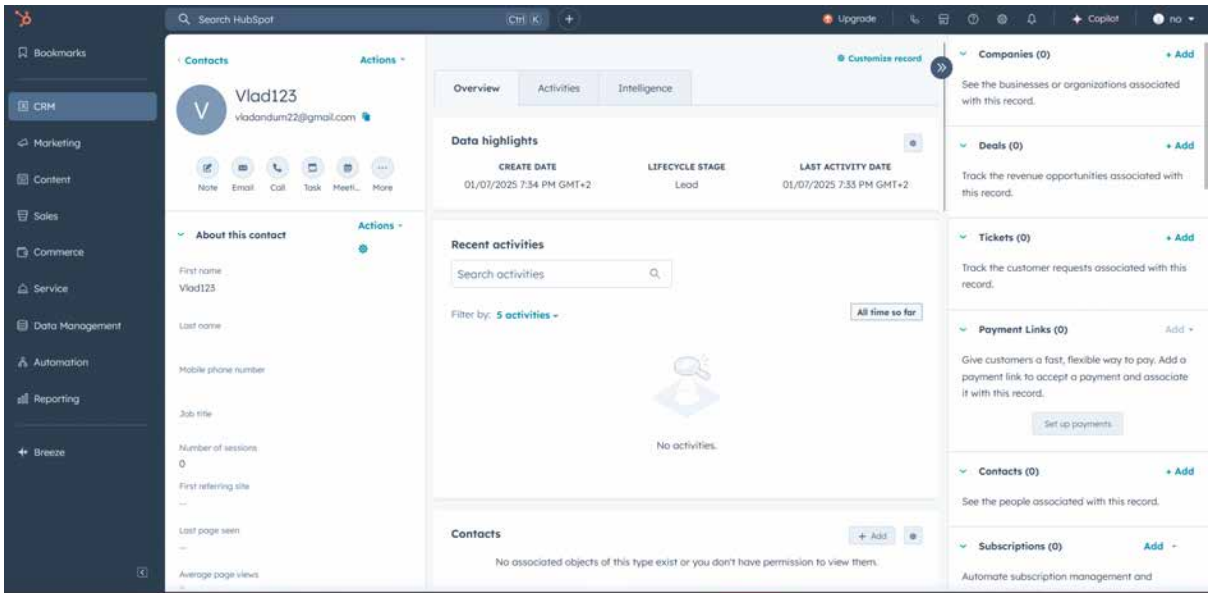

Рис. 5 Управління контактами CRM-системи - HubSpot

ERPNext та EspoCRM - це некомерційні системи. Перша є комплексною ERP-системою з можливістю розгортання на власному сервері. Надмірна складність робить її неадаптованою до вузьких бізнес-процесів та затратною в плані підтримки. Високий поріг входу робить її неконкурентноздатною з розроблювальною системою. Та всі вони так само прив’язані до конкретних сутностей. Не дивлячись на те, що в цих системах є великий набір жорстко встановлених сутностей та можливість працювати з “воронкми продажів”, це створює надлишкову кількість як візуального,

так і ресурсного навантаження на систему. Малому та середньому бізнесу необхідно концентруватися на швидкості та гнучкості бізнес-процесів.

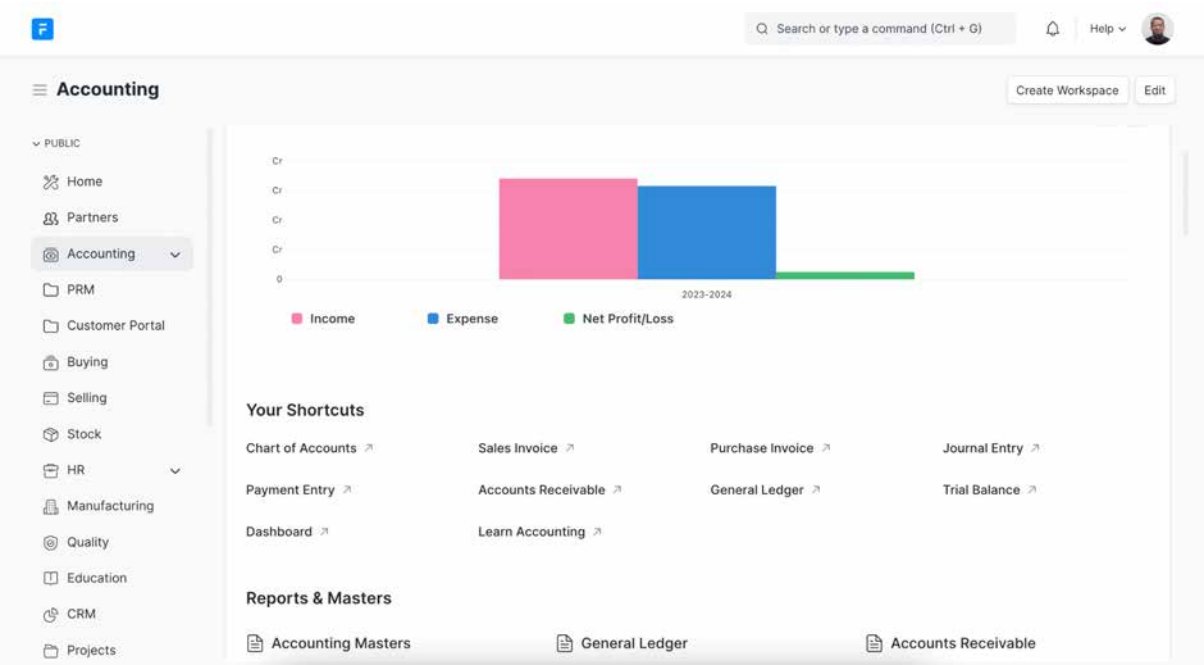

Рис. 6 Управління рахунками CRM-системи - ERPNext

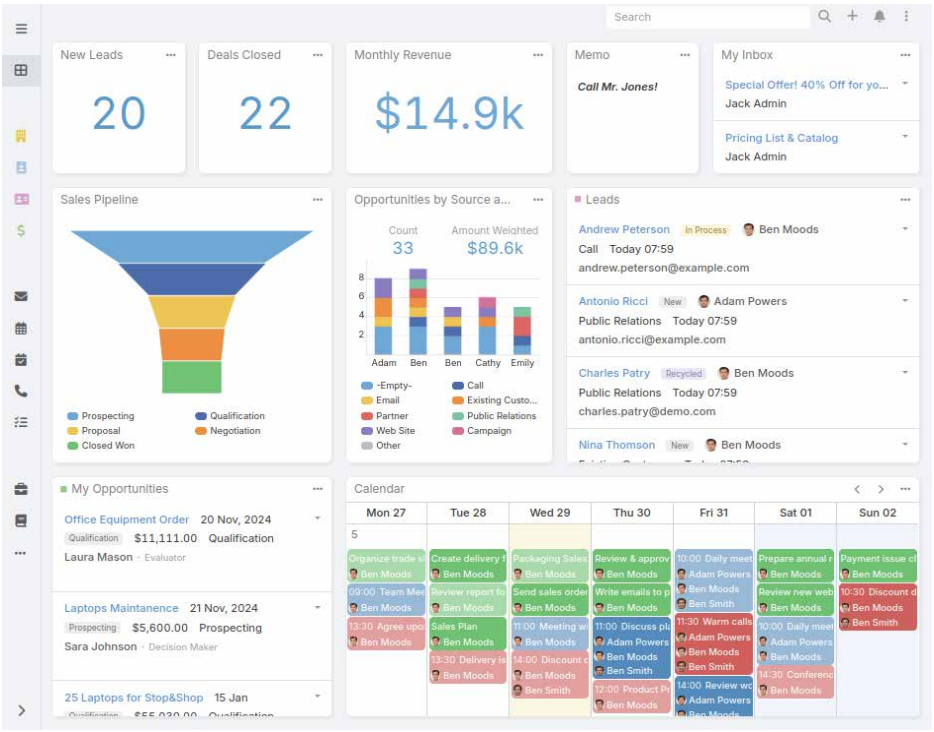

Рис. 7 Аналітичний дашборд CRM-системи - EspoCRM

EspoCRM - легка та гнучка, однак вона так само вимагає розробницьких навичок для внесення змін у логіку роботи або структуру

об’єктів. Всі перераховані системи підкреслюють загальну проблематику: навіть відкриті рішення часто мають жорстко визначену модель даних.

Таблиця 1

Порівняння існуючих CRM-систем

Параметр / Система	Zoho CRM	HubSpot CRM	ERP Next	Espo CRM	Пропонована система
Гнучкість структури даних	-	-	+	+ -	+
Простота інтерфейсу	-	+	-	+ -	+
Автоматизація без коду	+ -	-	-	+ -	+
Кастомізація без програміста	+ -	-	-	+ -	+
Наявність інтеграцій (API)	+	+ -	+ -	+ -	+
Орієнтація на МСБ	+ -	+ -	-	+ -	+
Безкоштовне використання	-	+ -	+	+	+

Аналіз показав, що жодна з розглянутих CRM-систем не може повністю поєднати гнучку структуру даних, простий інтерфейс та зручну автоматизацію без залучення розробника ПЗ. Запропонована система має на меті вирішити ці проблеми, надаючи адаптивний інтерфейс, універсальну модель даних та орієнтацію саме на потреби малого та середнього бізнесу. Всі існуючі рішення можуть задовольнити класичний крупний бізнес та класичний середній, який чітко регламентований та всі процеси не виходять за рамки визначених вимог. Але потреби такого бізнесу вже давно повністю покриті існуючими рішеннями. Зараз, особливо у період нестабільності, люди хочуть сподіватися на самих себе та

намагаються відкривати власний бізнес, тому всі існуючи системи є надважкими.

Під час аналізу були визначені також аналітичні матеріали з джерел на кшталт DOU, Medium, а також деякі порівняльні огляди з платформи G2 та документаційних сайтів систем. За власним досвідом можна сказати, що використання даних рішень у невеликих та навіть середніх проектах не є достатньо гнучким та зручним.

1.5. Постановка завдання

За мету у кваліфікаційній роботі бралась розробка веб-орієнтованої CRM-системи нового покоління, яка дозволить малому та середньому бізнесу ефективно та зручно керувати своїми процесами на базовому рівні без обмежень типової логіки, яка притаманна більшості існуючих CRM-рішень.

Основне завдання полягає у тому, щоб створити гнучку платформу для управління даними, яка дозволить користувачам самостійно формувати структуру сутностей, налаштовувати поля до них та визначати статуси, а також автоматизувати базові бізнес-процеси бе необхідності у залученні технічних спеціалістів чи розробників. Для досягнення поставленої мети треба вирішити такі задачі:

1) Створити систему аутентифікації та ролей користувачів (менеджер та адміністратор).
2) Надати можливість створення та редагування типів сутностей із довільними полями та статусами.
3) Побудувати механізм автоматичного формування форм для створення та редагування сутностей.
4) Розробити інтерфейс канбану та списочного формату.
5) Додати модуль базової аналітики даних та сутностей.
6) Реалізувати інтерфейс налаштування правил автоматизації на основі певних подій (створення, оновлення, зміна статусу, видалення).

7) Забезпечити можливість виклику webhook-запитів.
8) Створити журнал змін, в якому можна буде відстежувати історію змін даних.
9) Побудувати адаптивний веб-інтерфейс і використання сучасного стеку (Next 3, Fastify, PostgreSQL).

Очікуваним результатом буде програмна система, яка дозволить користувачу самостійно створювати CRM-процеси, адаптовані під його конкретну бізнес-модель, без необхідності залучення технічних спеціалістів, а також без необхідності використання фіксованих шаблонів чи надлишкового функціоналу.

2 ПРОЄКТУВАННЯ ІНФОРМАЦІЙНОГО ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1. Логічна модель даних у вигляді ER-діаграми

Для побудови ефективної CRM-системи з можливістю гнучкого керування типами сутностей, полями, статусами та користувачами, необхідно створити логічну модель даних, у якій будуть відображені ключові об’єкти предметної області та зв’язки між ними. Ця модель реалізується у вигляді ER-діаграми (Entity-Relationship Diagram), яка описує структуру даних на рівні логічних сутностей та їх атрибутів.

Метою побудови ER-діаграми є візуалізація основних типів сутностей, що використовуються у системі. Також візуалізація трибутів кожної сутності, зв’язків між сутностями (один-до-одного, один-до-багатьох, багато-до-багатьох), кардинальність та обов’язковість зв’язків, логічні обмеження, які треба враховувати під час реалізації фізичної моделі БД [6].

ER-діаграма має декілька основних елементів, а саме сутності, їх атрибути, відношення між сутностями та ключі. Сутності - це об’єкти, які мають певні характеристики. Атрибути є властивостями сутностей. Вони описують її. Відношення відображають залежності між декількома сутностями, насамперед існує декілька видів зв’язків: один до одного, багато до багатьох, один до багатьох. Ключі є атрибутами, вони необхідні для ідентифікації сутності. У якості інструмента створення діаграми було обрано онлайн сервіс “DB diagram”, який є зручним, потужним та в той самий час надає простий інтерфейс для створення логічної моделі.

На рис. 8 зображено діаграму з восьма сутностями. У створювальній системі виділяються такі основні логічні сутності:

1) **User (Користувач)** - Визначає обліковий запис користувача системи. Має атрибути: id, email, name, last_name, password_hash, role, created_at, is_active. Один користувач (адміністратор) може створювати різні типи сутностей, поля, статуси, правила автоматизації, менеджери можуть створювати сутності.

2) **EntityType (Тип сутності)** - Універсальна структура даних, яка визначає бізнес-об’єкти. Має атрибути: id, name, description, icon, user_id, created_at. Один користувач може може створювати безліч типів сутностей.

3) **EntityField (Поле типу сутності)** - Визначає схему структури даних для записів певного типу сутності. Атрибути: id, entity_type_id, field_key, label, type, is_required, options, created_at. Один тип сутності може мати багато полів.

4) **EntityStatus (Статус)** - Представляє можливі стани записів певного типу, Серед атрибутів можна визначити: id, entity_type_id, label, color, is_default. Один тип сутності має власний набір статусів.

5) **EntityRecord (Запис сутності)** - Це окремий об’єкт у системі, який має певний власний тип, статус та значення полів. Атрибути: id, entity_type_id, user_id, status_id, data (JSON), created_at, updated_at. Один тип сутності може мати багато записів. Кожен запис може бути створений або змінений користувачем.

6) **AutomationRule (Правило автоматизації)** - Правила автоматизації містять логіку, що виконується при певних діях (створення, оновлення тощо). Атрибути: id, entity_type_id, trigger, condition (JSON), action (JSON), is_active. Зв’язується з типом сутності.

7) **AtivityLog (Журнал активності)** - Містить історію змін записів. Атрибути: id, entity_record_id, user_id, action_type, payload, timestamp. Кожен запис може мати багато різних подій у журналі. Відображається у історії (у проєкті він буде називатися таймлайн) кожної сутності. Ця історія

надає повне розуміння про зміни поточного запису навіть тоді, коли поле запису було видалено, що допомагає зберегти цілісність інформації.

8) **AutomationLog (Журнал запуску бізнес-процесів)** - Містить історію запусків бізнес-процесів. Атрибути: id, ttiggered_by, entity_record_id, result, rule_id, payload, timestamp. Кожен бізнес-процес може мати багато подій у журналі.

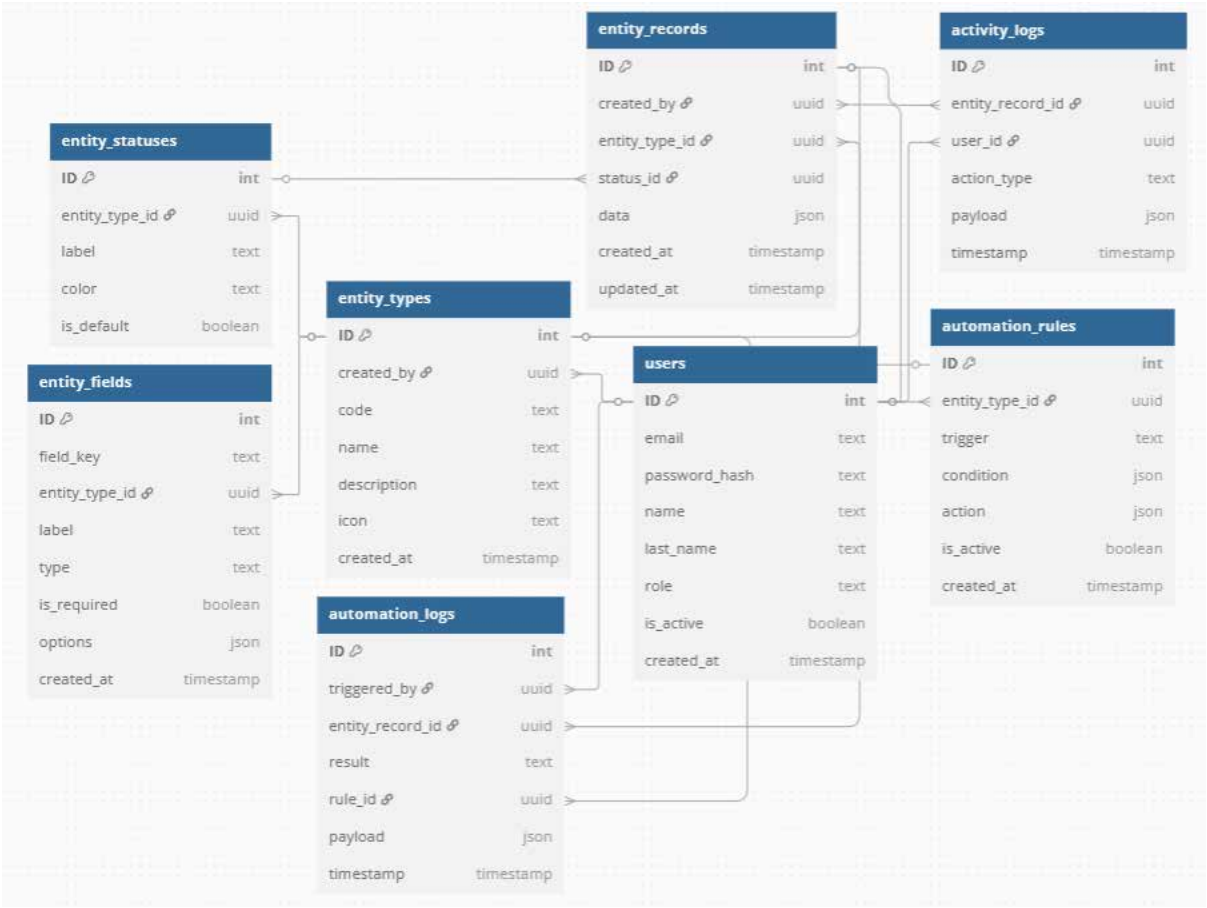

Рис. 8 Логічна модель даних (ER-діаграма)

Кардинальність та обов’язковість

1) Кожен EntityRecord пов’язаний з обов’язковими EntityType і обов’язковими EntityStatus.
2) Кожен EntityType може не мати полів або статусу на момент створення, бо користувач сам визначає їх кількість.

3) AutomationRule не є обов’язковим для роботи системи, але надає можливість розширювати функціонал системи та автоматизувати деякі бізнес-процеси.

ER-діаграма для даного проєкту відображає розширену структуру даних, яка орієнтована на гнучкість, адаптивність та масштабованість. Цей підхід надає можливість підтримки динамічної побудови сутностей, а також налаштовувати логіку роботи системи без втручання у структуру БД та програмний код. Це забезпечує основу для майбутнього розширення функціоналу системи.

2.2. Діаграма класів та кооперацій

Для кращої деталізації архітектури розроблювальної системи та візуального відображення об’єктно-орієнтованих зв’язків між елементами системи, були розроблені діаграми кооперацій та класів. Дані діаграми надають можливість описати не тільки логічні взаємодії об’єктів під час виконання бізнес-процесів, але й відображають структуру класів, атрибути, методи та зв’язки, а також послідовність викликів між ними.

Діаграма класів - це структурна діаграма, вона відображає головні компоненти логіки програми, що включає моделі та сервіси [8]. У програмній системі центральне місце займають класи, які пов’язані зі створенням та обробкою універсальних сутностей. Клас User описує облікові записи користувачів і містить основну інформацію: ім’я, роль, стан та активності. EntityType є шаблоном. на основі якого формуються конкретні типи об’єктів, а EntityField визначає структуру їхніх полів. Всі типи мають певний набір потенційних статусів, які описуються класом EntityStatus.

Ключовою частиною системи є клас EntityRecord, що є представленням конкретного запису сутності. Там зберігаються дані, статус, а також зв’язок і користувачем та типом сутності. Для реалізації автоматизації передбачений клас AutomationRule, який містить певні умови

виконання та відповідні дії, що можуть бути передані до будь-яких зовнішніх сервісів через Webhook. Результати записуються через AutomationLog. Журнал змін реалізується у вигляді класу ActivityLog.

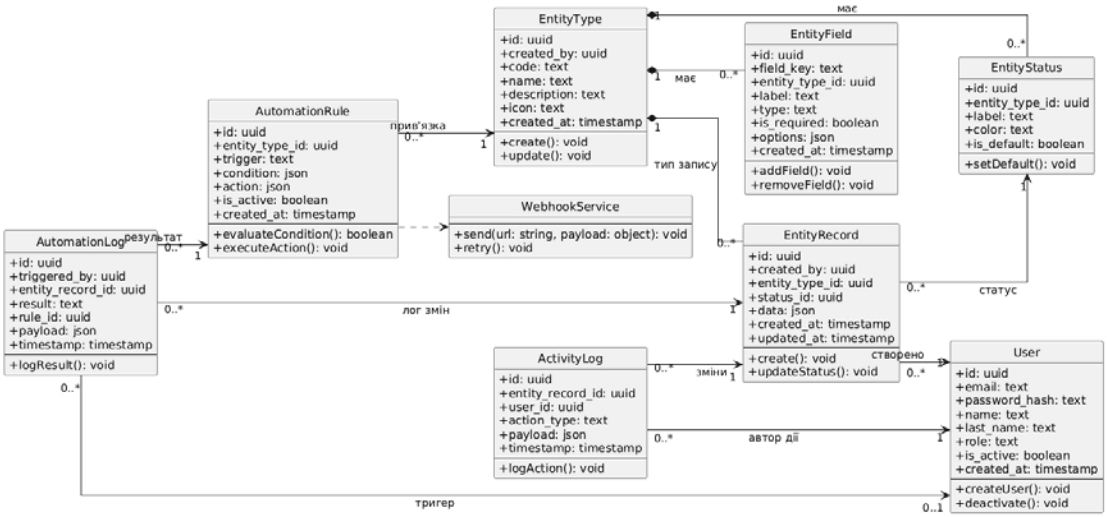

Рис. 9 Діаграма класів

На рисунку 9 зображена діаграма класів. В ній використовуються різні типи зв’язків. Використовується композиція, яка відображає сильну залежність, тобто одна частина не може існувать без цілого. Також використовуються напрямлена асоціація, в якій один клас посилається на інший. Використовується залежність, в якій клас використовує інший, але в цілому не володіє ним.

Діаграма кооперацій відображає взаємодію об’єктів в контексті певного сценарію. Для цього вибрано типовий випадок - створено нового запису сутності з подальшим запуском правила автоматизації [2]. У цьому процесі об’єкти виконують роль учасників, що послідовно передають один одному повідомлення. Зображена на рисунку 10. Взаємодія починається з контроллера UserController, він займається обробкою запиту клієнта. Після цього створюється EntityService, також створюється ActivityLog. Запускається перевірка в AutomationRule. Якщо умови виконані, то запускається WebhookService, або EntityUpdaterService. Всі зміни

зберігаються в AutomationLog, ActivityLog і при необхідності цикл повторюється знову.

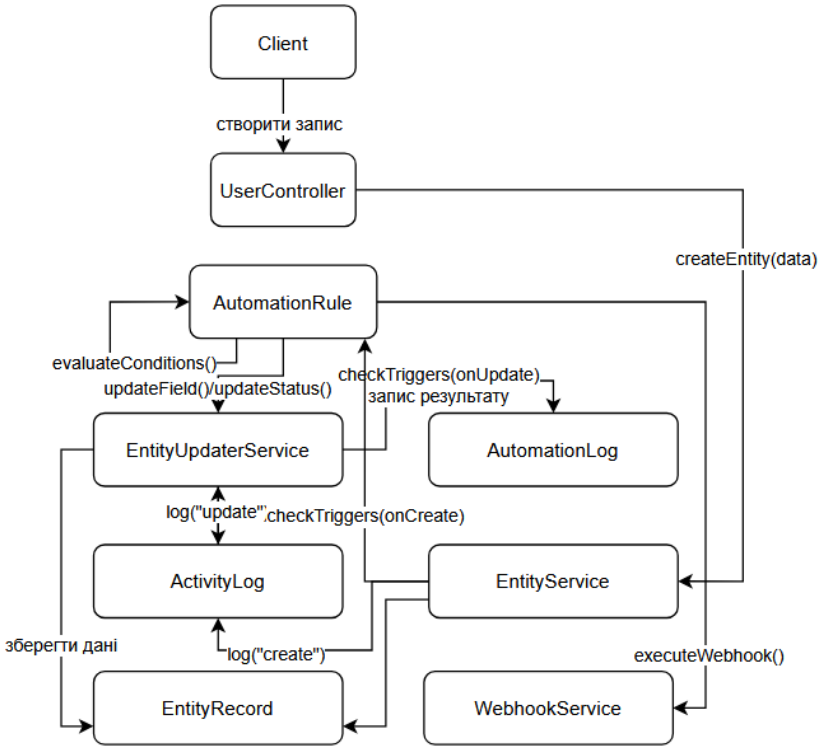

Рис. 10 Діаграма кооперацій

У випадку, якщо дія - це внутрішня модифікація, наприклад зміна статусу чи оновлення значення певного поля, це відбувається через EntityUpdaterService, що оновлює сутність і викликає повторну перевірку автоматизації, якщо це потрібно (onUpdate). Якщо це зовнішня дія - то її оброблює зовнішній сервіс - WebhookService.

2.3. Діаграма пакетів

Package Diagram (Діаграма пакетів) - це один з ключових елементів статичних структур UML діаграм. Діаграма пакетів надає реальну можливість описати організацію програмного забезпечення у модульному вигляді. Ця діаграма показує, як саме логічно згруповані класи, сервіси та інтерфейси у вигляді модулів, тобто окремих пакетів і які саме залежності

є між цими модулями. Ця діаграма дозволяє краще сприйняття загальної структури системи та її підтримуваності та масштабованості [2].

У рамках розробленої CRM-системи пакети відповідають окремим логічним компонентам, а саме: керування сутностями, робота з користувачами, автоматизація, журналювання та інтеграції. Кожен з пакетів має власні класи, які пов’язані загальною функціональністю. В свою чергу це спрощує розширення та супровід коду. Дана діаграма зображена на рисунку 11.

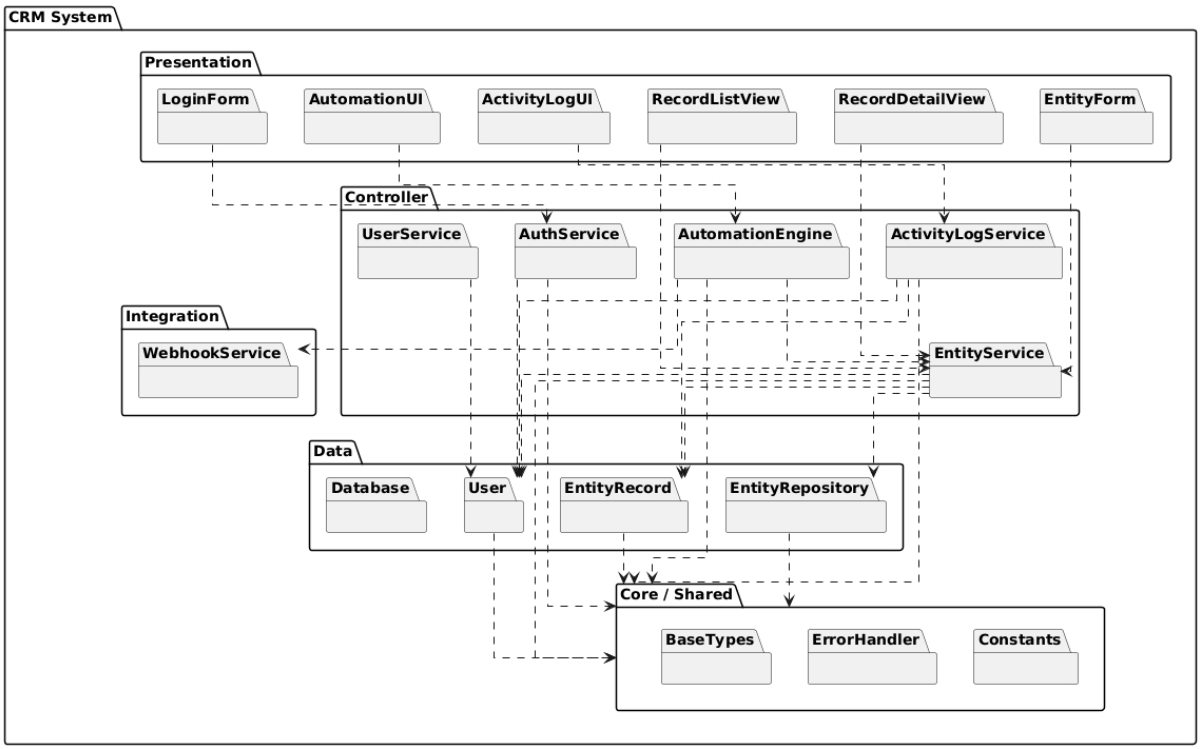

Рис. 11 Діаграма пакетів

Система складається з декількох основних пакетів: презентаційний рівень, який відповідає за представлення даних. Контроллер має на мені управління даними та бізнес-логікою продукту. Інтеграційний рівень пов’язаний із зовнішніми сервісами через webhook. Рівень даних займається обробкою користувацьких даних, даних сутності та власне базою даних. Рівень Core / Shared забезпечує загальний набір класів, констант та обробку помилок всієї системи.

Система умовно поділяється на такі основні пакети:

1) Auth - пакет містить логіку аутентифікації користувачів системи, а також керування сесіями, авторизацію та перевірку ролей (адміністратор та менеджер). Класи у цьому пакеті відповідають за реєстрацію, логін, валідацію токенів тощо.
2) Users - пакет включає класи моделі користувача, сервіси редагування та створення профілю, а також контролери та схеми валідації. Це незалежний пакет, хоча він використовується в інших частинах системи через залежності.
3) Entity - один з найбільших та найскладніших пакетів, який містить моделі EntityType, EntityField, EntityStatus, EntityRecord, а також сервіси для обробки логіки створення типів сутностей, записів, канбану, списку тощо. Пакет тісно взаємодіє з автоматизацією та активностями (automation, activity).
4) Automation - цей пакет реалізує правила автоматизації (AutomationRule), механізм виконання (AutomationEngine), а також обробку умов і дій. Також ведення логів через AutomationLog. Пакет залежить від entity a integration.
5) Integration - це пакет, який відповідає за інтеграцію із різними зовнішніми сервісами. На поточному етапі система реалізує виклики webhook, який є універсальним способом взаємодії з зовнішніми сервісами. Основним класом пакета є WebhookService.
6) Activity - це пакет для логів, у якому журналюються події у системі. Цей пакет містить ActivityLog та сервіс ActivityLogService. Він залежить від entity та users.
7) Core/shared - це допоміжний пакет з утилітами, константами та деякими загальними типами. Також у пакеті зберігається обробники помилок та інтерфейси. Є базовим для більшості інших модулів.

Залежність між пакетами:

1) Entity залежить від users, тому що записи створюються користувачами.
2) Automation залежить від entity, через перевірку умов по сутностях, а також від integration, через виконання дій.
3) Activity залежить від entity та users, задля того, щоб фіксувати події по певних сутностях, які були ініційовані користувачами системи.
4) Integration - інфраструктурний, не залежить від зовнішніх сервісів, але впливає на них.
5) Auth - це ізольований пакет, хоча він використовується в users для аутентифікації.

Таким чином цей розподіл надає чітку та структуровану логіку для системи, дозволяє забезпечити слабке зв’язування між модулями, які є важливим принципом розробки підтримуваних та масштабованих рішень.

Діаграма пакетів відображає логічну декомпозицію системи та її архітектурну цілісність. Чітко визначене розмежування функціоналу за модулями дозволяє мінімізувати складність проєкту, а також полегшити супровід коду. Разом з цим з'являється можливість виконувати незалежну розробку окремих частин системи, що позитивно впливає на хід розробки, швидкість, зручність контроля версій, масштабованість у подальшому розвитку мікросервісної архітектури (архітектура, в основі якої лежить розподілення компонентів серверної частини на повністю незалежні один від одної частини) тощо. Така структура надає чудову можливість для розширення ролей та особливо інтеграцій, бо кожна інтеграція - повністю незалежна частина, яка потенційно може бути не лише зав’язана на webhook, а я доповнюватися іншими більш складними сервісами.

2.4. Діаграма компонентів

Діаграма компонентів (*Component Diagram від англ.*) є одним з типів діаграм UML. Ця діаграма використовується для моделювання фізичної структури програмної системи, зокрема для відображення основних компонентів, модулів, бібліотек чи пакетів та взаємозв’язками між ними. Діаграма компонентів орієнтована на макрорівень, тобто як саме частини системи співпрацюють як окремі незалежні блоки, які реалізують свої контракти через чітко визначені інтерфейси, коли діаграма класів показує саме внутрішню логіку [2].

У контексті розроблювальної програмної системи, компонентна архітектура базується на принципах розділення відповідальностей, слабкому зв’язувані та інтерфейсній взаємодії між модулями. Це дозволить зробити кожен функціональний блок окремо, з можливістю подальшої заміни, тестування та масштабування системи.

Основними компонентами системи можна зазначити:

1. Frontend Application
   Веб-інтерфейс, що реалізований на базі Nuxt.js (на основі Vue.js), відповідає за візуальне представлення додатку, за перевірку та валідацію введення, маршрутизацію, інтеграції з API та, як ключовий фактор, взаємодію з користувачем. Компонент має залежність лише від Backend API, спілкування з яким реалізується через протокол HTTP з захистом через шифрування HTTPS. Окремим підкомпонентом є UI-каталог на базі Tailwind CSS.
2. Backend API (Фреймворк Fastify на базі Node.js)
   Компонент є ядром всієї бізнес-логіки CRM-системи. Компонент реалізує REST API, аутентифікацію користувачів, обробку запитів, логіку керування сутностями, правилами, журналами тощо. Компонент складається з підкомпонентів і є центральним вузлом для взаємодії з базою даних, сторонніми сервісами та з фронтендом.

3. Auth Module
   Компонент відповідає за авторизацію користувачів. Створення та валідацію токенів, захист маршрутів (middleware для перевірки прав). Реалізує інтерфейси для: /login, /register, /me.
4. Entity Management Module
   Це компонент, що містить логіку CRUD для типів сутностей, полів, статусів та записів. Це ядро системи, яка дозволяє користувачеві створювати структури під власні бізнес-процеси. Реалізує інтерфейси: /entity-types, /entity-records, /fields, /statuses.
5. Automation Engine Module
   Компонент обробляє запуск правил автоматизації на основі певних тригерів, наприклад onCreate, onUpdate, onStatusChange. Компонент реалізує логіку перевірки умов та виконання дій, внутрішніх чи зовнішніх. Має залежність від модуля сутностей, оскільки він діє на їх основі.
6. Webhook Integration Module
   Відповідає за виконання зовнішніх дій, таких як HTTP POST-запити до сторонніх систем. Він має чіткий інтерфейс виклику та забезпечує універсальність інтеграцій через конфігурування webhook-и.
7. Activity Logging Module
   Записує події користувачів і системи, включаючи створення записів, зміни статусу, запуски автоматизації тощо. Компонент має власний інтерфейс таблиці та сховища: logEvent(entityId, actionTye, payload).
8. Automation Logging Module:
   Реалізує ведення журналу результатів виконання автоматизації. Дозволяє відстежувати успіх або помилки під час виконання дій. Залежить від механізму автоматизації та бази даних.
9. Database (PostgreSQL)
   Центральне сховище інформації, з яким працює API Backend. Доступ

до бази даних здійснюється виключно через моделі та репозиторії. Вона реалізує інтерфейси для читання, запису, фільтрації та оновлення даних.

Компоненти між собою формують такі зв'язки:

1) Frontend пов'язаний з Backend API. HTTP REST-взаємодія через захищені маршрути.
2) Backend API взаємодіє з PostgreSQL.
3) Automation Engine викликає методи EntityService, WebhookService, EntityUpdaterService.
4) Webhook Module не має зворотних залежностей, бо його задача лише надіслати запит на зовнішній сервіс, (інфраструктурний).
5) ActivityLog та AutomationLog - це слухачі (observer) змін у сутностях та автоматизації відповідно.

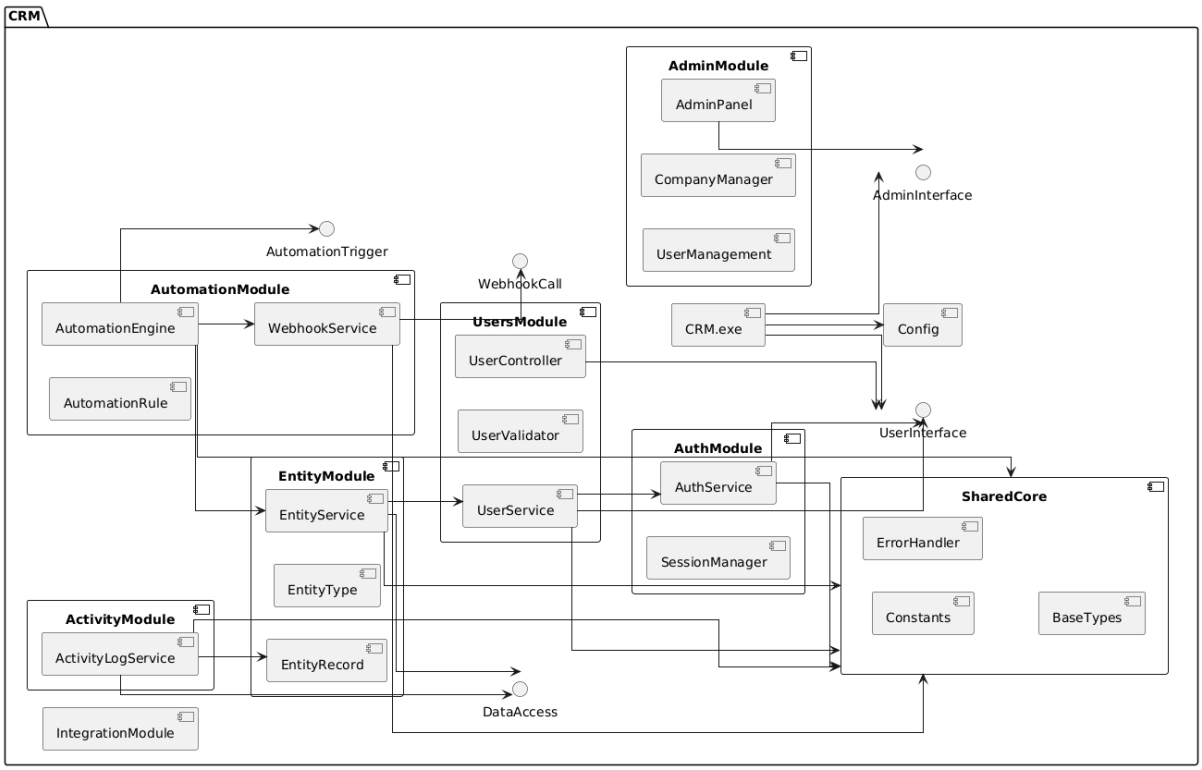

Рис. 12 Діаграма компонентів

Діаграма компонентів відображає архітектурну структуру CRM-системи у вигляді незалежних модулів. Всі ці модулі взаємодіють між

собою, але всі вони належать центральному елементу - CRM.exe, який взаємодіє з конфігураційним файлом Config і використовує інтерфейси користувача та адміністратора.

Інтерфейси - це узгоджені точки взаємодії між компонентами системи. Інтерфейси визначають, які саме функції та дії будуть доступні зовнішнім модулям. У той самий час вони не розкривають внутрішню реалізацію компонента. Наприклад, інтерфейс UserInterface описує те, як зовнішні модулі можуть звертатися до функцій, що пов’язані з користувачами. Інтерфейси надають можливість чітко розмежувати відповідальність модулів, уникнути прямої залежності між компонентами системи, добре розділити зони відповідальності модулів та спростити зміну або оновлення частин системи без зміни інших частин.

3 РОЗРОБКА ІНФОРМАЦІЙНОГО ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1. Система управління інформаційною базою

Інформаційна база - це центральний компонент будь якої інформаційної системи, бо вона надає можливість збереження, обробки, логічну цілісність, безпеку даних та доступ до них [4]. У розроблювальному проєкті інформаційна база побудована на основі гнучкої, динамічної моделі з можливістю налаштування структури даних без зміни самої бази даних. Для інтеграції цього підходу критично важливо правильно спроєктувати систему управління інформаційною базою.

Обрана СУБД - PostgreSQL. У якості системи управління базами даних (СУБД) було обрано PostgreSQL. Це дуже потужна об’єктно-реляційна СУБД з відкритим кодом. Вона надає високий рівень надійності, гнучкості, безпеки, розширюваності та підтримки складних типів даних. Всі ці фактори можуть дуже сильно знадобитися у майбутньому, та навіть на поточному етапі добре задовольняють вимоги системи. PostgreSQL підтримує CTE-запити, JSONB, індекси GIN, обмеження, транзакції та розширення. Це робить її ідеальним варіантом для створення системи з динамічною структурою полів, як у випадку розроблювальної CRM-системи. Обрана СУБД ідеально підходить для реалізації CRM-системи з підтримкою динамічних структур даних, завдяки поєднанню реляційного ядра з підтримкою напівструктурованих даних (jsonb) [4].

СУБД та обрана модель дозволяє забезпечити одночасно сувору цілісність та гнучкість, необхідну для зберігання записів довільної структури. PostgreSQL підтримує складні індекси, транзакції, каскадні зв’язки та має розширення, що можуть бути використані у майбутньому для оптимізації пошуку та аналізу в текстових полях.

Інформаційна модель системи розроблена з дотриманням принципів нормалізації бази даних до третьої нормальної форми (ЗНФ). Кожна логічна сутність винесена в окрему таблицю, що дозволяє уникати дублювання даних та забезпечує логічну цілісність. У той же час, для досягнення гнучкості в частині збереження значень полів записів (які відрізняються за структурою), використано комбінацію нормалізованої схеми з динамічним полем data: json, що дозволяє реалізувати змішану модель, тобто баланс між чіткою нормалізацією та реальною та гнучкою адаптивністю до потреб користувача.

Структура інформаційної бази

Інформаційна модель базується на універсальному підході до зберігання та організації бізнес-даних. Користувачі будуть мати змогу самостійно створювати типи сутностей, додавати до них поля, статуси та визначати правила взаємодії. Це дозволяє системі адаптуватися до практично будь-яких бізнес-процесів, при цьому немає потреби втручатися в код або в структуру БД.

Загальна структура БД включає вісім основних логічних блоків, всі вони реалізовані у вигляді таблиць:

1. **users - Користувачі системи**

   Таблиця користувачів містить базовий набір інформації про користувачів системи, які з нею працюють. Кожен запис містить унікальний ID, адресу електронної пошти, хеш пароля, ім’я, прізвище, роль (admin, manager) та мітку про активність користувача (наприклад, користувач може бути відключений від системи без можливості входу, хоча фактично його аккаунт та записи з БД не будуть видалені для збереження цілісності даних). Всі ключові сутності в системі (entity_types, entity_records, activiy_logs) мають зв’язки із цією таблицею для збереження автора запису чи виконавця конкретної дії.

2. **entity_types - Типи сутностей**
   Таблиця визначає концептуальні одиниці обліку, які можуть створювати користувачі з роллю адміністратора. Кожен тип має код, назву, опис, іконку та прив’язку до свого автора (created_by). Це дозволяє реалізувати багатооб'єктну модель, коли у рамках однієї бази можуть зберігатися, наприклад, клієнти, заявки, продукти тощо - всі як окремі типи сутностей, при цьому вони універсальні, мають абсолютно ідентичну системну основу.
3. **entity_fields - Поля типу сутност**
   Таблиця entity_fields - це важлива частина для реалізації гнучкої структури, яка дозволяє описати, які саме поля має кожен тип сутності. У полі type вказується тип значення (text, number, boolean, select, date, relation). Поле options у форматі JSON дозволяє описати список можливих значень, наприклад для select, або вказати налаштування для зв’язків з іншими типами у полі relation. Таким чином, структура полів може змінюватися в будь-який момент без зміни схеми таблиць.
4. **entity_statuses - Статуси сутностей**
   Кожен тип сутності має власний набір статусів. Вони дозволяють реалізувати “етапність” у бізнес-процесах сутності. Наприклад: “Новий”, “В роботі”, “Закрито”). Статус має назву, колір та ознаку статусу за замовчуванням. Статуси використовуються для відображення канбан-дошки, фільтрації даних і, також, у ролі тригерів в автоматизації.
5. **entity_records - Записи сутностей.**
   Це таблиця, у якій зберігається всі об’єкти, що були створені користувачами. Кожен запис містить entity_type_id, status_id, created_by, часові мітки та поле data з типом json, що є головним, оскільки воно зберігає основну інформацію про поточний об’єкт, які

були зазначені при його створені на основі полів типу цієї сутності. Такий підхід дозволяє зберігати будь-яку структуру без необхідності створення окремої таблиці кожного типу.

6. **automation_rules - Правила автоматизації**
   Таблиця містить в собі правила, які прикріплені до певного типу сутності та викликаються за подіями onCreate, onUpdate, onStatusChange. У полі condition зберігається логіка перевірки умови (у вигляді JSON), в той час у полі ation зберігається опис поточної дії: зміна значення поля або статусу, виклик вебхука тощо. Таким чином, автоматизація стає гнучкою та керованою з інтерфейсу користувача.
7. **activity_logs - Журнал активності**
   Таблиця містить в собі історію дій користувачів і системи, а саме історії змін полів сутностей, що дозволяє видстежувати історію в зручному форматі. Для кожної події створюється запис сутності, виконавець, тип дії (наприклад:create, update, status_change, automation), а також JSON-представлення змінених даних. Це дозволяє реалізувати аудит, прозорість зін і відображення історії змін у вигляді таймлайну для кожного об’єкту, що може забезпечити як зручність, так і надійність доступу до інформації.
8. **automation_logs - Журнал автоматизації**
   Це окрема таблиця для реєстрації результатів виконання автоматизованих правил. Таблиця зберігає правило, що спрацювало, для якого запису, чи було воно ініційоване користувачем чи системою (бо поле triggered_by може бути NULL, а значить явного користувача-ініціатора не існує, тож процес системний), а також результати виконання у вигляді статусу (success, failed, pending) і payload для зберігання додаткових корисних даних.

Ключовий елемент даної моделі - це використання поля data з типом json, яке зберігає значення, що відповідають динамічно створеним полям. Це дозволяє підтримувати необмежену кількість типів об’єктів і їх структур. Без потреби у створенні нових таблиць. Кожне значення валідується та обробляються на рівні застосунку згідно з метаданими таблиці entity_fields, тож вони є невід’ємною частиною одне одного з точки зору логічної структури. Для забезпечення продуктивності планується використання GIN-індексів по JSONB-полям, а також є плани по створенню зовнішніх індексів для полів, які часто використовуються для фільтрації (created_by, status_id, created_at).

Кожен ключовий зв’язок між таблицями зазначено через зовнішні ключі з каскадним оновленням або видаленням (ON DELETE CASCADE), що дозволяє уникнути пошкодження структури при видалені типів, статусів або користувачів. Інформаційна база також передбачає можливість масштабованості у майбутньому. Наприклад, розділення по схемах (shema per tenant) у разі реалізації багатокомпанійної архітектури в рамках SaaS-додатку, додавання розмежування прав доступу через RLS (Row-Level Security), інтеграції з чергами задач (RabbitMQ, Redis Streams, BullMQ тощо) для асинхронних автоматизацій виконання довгих задач.

Система управління інформаційною базою на основі PostgreSQL є гнучкою, модульною та адаптованою до змін. Вона повністю відповідаю вимогам, що були передбачені при постановці задачі до розроблювальної інформаційної системи. Така архітектура передбачає створення динамічних структур даних без змін схеми БД, створити універсальне рішення для CRM-потреб малого та середнього бізнесу, яке здатне еволюціонувати, інтегруватися з іншими сервісами без втрати стабільності та до масштабованості.

3.2. Розробка інформаційної бази

Процес розробки інформаційної бази є одним із ключових етапів створенні інформаційної системи. Від структури бази даних та правильно визначених взаємозв'язків залежить ефективність зберігання, обробки та відображення даних у системі. У межах даного проєкту інформаційна база була спроєктована як модульна, гнучка та масштабована структура, яка буде здатна адаптуватися до динамічних вимог користувачів.

На поточному етапі була реалізована логічна структура БД у вигляді SQL таблиць відповідними зв'язками, обмеженнями, первинними та зовнішніми ключами. В основу були закладені принципи нормалізації, дотримання вимог консистентності та реалізовано механізм динамічного зберігання даних з використанням типу даних json.

Процес проєктування інформаційної бази складався поділяється на:

1. Формування логічної моделі даних
   Для початку було визначено основні об'єкти предметної області, тобто користувачі, типи сутностей, записи, статуси, поля, правила автоматизації та історії змін. Усі ці сутності описані у вигляді логічної ER-моделі з подальшим перетворенням у структуру таблиць
2. Опрацювання типів зв'язків між таблицями
   Було встановлено типи зв'язків між сутностями (один-до-багатьох, багато-до-одного), використання зовнішніх ключів та каскадного видалення повністю обдумане, що забезпечує цілісність.
3. Проєктування гнучкої структури записів
   Замість створення окремих таблиць під кожен тип сутності реалізовано універсальну модель зберігання даних (entity_records) із полем data: json, що динамічно адаптується до вибору полів, описаних у таблиці entity_fields.
4. Опис правил автоматизації
   Правила автоматизації розроблено у вигляді окремої таблиці, де

кожне правило містить тригер, умову у форматі JSON (визначається у бізнес-логіці додатку) і дію (також у форматі JSON). Все це дозволяє централізовано керувати автоматизацією без втручання в логіку коду.

5. Додавання логування подій

   Для збереження історичності та аудиту даних, створено дві додаткові таблиці- activity_logs та automation_logs, що дозволяють зберігати інформацію про дії користувачів і виконання автоматизованих дій відповідно.

Було створено фізичну структуру БД у вигляді SQL-таблиць за результатами логічної моделі. Вісім основних таблиць, що описані у попередньому підрозділі, з чітко визначеними типами даних, зовнішніми ключами та індексами для пришвидшення доступу.

```
CREATE DATABASE crm_system;
```

Рис. 13 Створення бази даних

Також було створено таблицю користувачів, яка містить основні дані про користувача, дані авторизації та його роль.

```
CREATE TABLE users (
        id UUID PRIMARY KEY,
        email TEXT NOT NULL UNIQUE,
        password_hash TEXT NOT NULL,
        name TEXT,
        last_name TEXT,
        role TEXT CHECK (role IN ('admin', 'manager', 'user')),
        is_active BOOLEAN DEFAULT TRUE,
        created_at TIMESTAMP WITHOUT TIME ZONE
DEFAULT CURRENT_TIMESTAMP
);
```

Рис. 14 Створення таблиці користувачів CRM-системи

Напевно ключовою таблицею всієї системи є таблиця типів сутностей, яка містить в собі всю інформацію про потенційно створений запис. Кодова назва дозволяю визначити зрозумілий код до сутності, що набагато зручніше, ніж її ID. Цей код можна буде використовувати в URL. Іконка - це візуальне зображення, яке буде використовуватися для кращого інтуїтивного розуміння значення сутності.

```
CREATE TABLE entity_types (
        id UUID PRIMARY KEY,
        created_by UUID REFERENCES users(id) ON DELETE
SET NULL,
        code TEXT NOT NULL UNIQUE,
        name TEXT NOT NULL,
        description TEXT,
        icon TEXT,
        created_at TIMESTAMP WITHOUT TIME ZONE
DEFAULT CURRENT_TIMESTAMP
);
```

Рис. 15 Створення таблиці типів сутностей

У процесі реалізації враховано деякі аспекти: ID генеруються на рівні СУБД, що забезпечує унікальність записів. Для таблиць, що зберігають JSON (entity_records, automation_rules, activity_logs, automation_logs), використовуються json/jsonb залежно від потреб у фільтрації. Для критичних зв’язків використано ON DELETE CASCADE, що забезпечує автоматичне видалення залежних записів, наприклад при видалені типу сутності видаляються всі записи та статуси.

Для забезпечення швидкості навіть при зростанні обсягу даних, було застосовано індексацію по ключових полях:

1) created_by для фільтрації записів по користувачах.
2) entity_type_id, status_id для фільтрації у списках та канбані.
3) created_at для сортування та аналітики.

Також враховано, що в майбутньому система може бути доповнена механізмами кешування, аналітичними шардами.

Було створено користувача-адміністратора:

```
INSERT INTO users (email, password_hash, name, last_name, role,
is_active, created_at)
VALUES (
        'ipz22-v.dumanskyi@nubip.edu.ua',
        'ХЕШ_ПАРОЛЬ',
        'Владлен',
        'Думанський',
        'admin',
        TRUE,
        CURRENT_TIMESTAMP
);
```

Рис. 16 Створення користувача для таблиці users

Після створення схеми було проведено початкове тестування наповненням тестовими даними, створення сутностей, записів, логуванням та внесення записів про автоматизації. Всі операції проходили через бізнес-логіку застосунку з подальшою перевірку відповідності збережених даних у БД. Це дозволило перевірити правильність структури таблиці полів сутностей і відповідність JSON-полів у таблиці записів сутностей. Також це дозволило перевірити стабільність тригерів автоматизації, реєстрацію змін у таблиці логування сутностей та надійність зберігання результатів виконання автоматизації.

Інформаційна база розробленої CRM-системи - це гнучка і добре нормалізована основа, що забезпечує структурованість і адаптивність одночасно. Використання PostgreSQL повністю аргументовано та відповідає всім вимогам. Підтримка динамічних полів, продумане логування та інструменти автоматизації роблять БД придатною як для

невеликих бізнесів, так і для середнього бізнесу з подальшою можливістю масштабування в складні корпоративні середовища.

3.3 Вибір інструментарію для створення прикладного програмного забезпечення

Вибір технічного стеку та інструментальних засобів має надзвичайно важливе значення для ефективності розробки прикладного програмного забезпечення. У рамках реалізації CRM-системи нового типу, яка має забезпечити гнучкість, простоту інтерфейсу, масштабованість та підтримку складних логічних зв’язків, було прийнято рішення використати сучасні веб-орієнтовані технології, які одночасно дозволяють реалізувати як серверну, так і клієнтську частину системи з високим рівнем узгодженості.

Вибраний стек технологій дозволяє досягти повної декапсуляції фронтенду та бекенду. Це спрощує масштабування. Також це дозволяє підвищити продуктивність а рахунок асинхронної обробки запитів, ручної типізації та підтримки масштабних проєктів. Ефективне CD/CD і розгортання на хмарних платформах повністю підходить під обраний стек.

3.3.1 Клієнтська частина (Frontend)

Фреймворк: Nuxt.js 3 на базі Vue.js 3.

Інші інструменти: TypeScript, Pinia, Tailwind CSS, Vite.

Nuxt 3 є сучасним фреймворком для створення продуктивних, реактивних та швидких веб-застосунків. Він забезпечує SSR (server-side rendering) за потреби, підтримку статичної генерації, модульну архітектуру веб-додатку та дуже зручну та гнучку інтеграції із власним бекендом за допомогою API [5]. Nuxt має ряд переваг у даному проєкті:

1. Розділення інтерфейсу та логіки
2. Швидкий старт, легкість у конфігураціях
3. Сильна типізація за рахунок використання TypeScript (мова на основі JavaScript, яка дозволяє використовувати типи, що покращує практику написання якісного коду та тестування) [7].

4. Можливість потенційно у майбутньому створення SEO-адаптованих веб-сторінок.
5. Компонентний підхід, який закладено у Vue.js , який дозволяє відокремлювати UI елементи, взаємодіяти з їхнім життєвим циклом та використовувати безліч разів.

Сучасні веб-браузери використовують механізм DOM-дерева для відображення елементів веб-сайту та контролю за їх станом. Це базовий підхід, але середовище ізольовано та працює синхронно, тому воно не може бути “динамічним”. Завдяки Vue ця проблема вирішилася за рахунок віртуалізації DOM-дерева, що дозволило ефективно оновлювати інтерфейс (рендеринг). Цей фреймворк є аналогом популярного React.js, але більш мінімалістичний та “приємний” з точки зору досвіду розробки. Але разом з цим постає питання зберігання даних на клієнтській частині, дуже ефективно це дозволяє робити state-manager (контролер стану) Pinia, що є більш сучасною альтернативою Vuex із високою консистентністю та офіційною рекомендацією команди Vue.

3.3.2 Серверна частина (Backend)

Платформа: Node.js

Фреймворк: Fastify

Мова: TypeScript

Для створення серверної логіки використано високопродуктивний фреймворк Fastify. Цей фреймворк є більш досконалою версією Express. Він дозволяє забезпечити обробку HTTP-запитів с мінімальними витратами на ресурси, що значно підвищує RPS (Request per second - [кількість] запити за секунду). Фреймворк має підтримку плагінів, обробок помилок, нативну підтримку схем валідації, що дозволяє валідувати дані як на вході, так і на виході, та асинхонну логіку. Використання TypeScript дозволяє уникнути помилок на етапі розробки та полегшує підтримку великого проєкту за рахунок типізації. Хоча TypeScript не виконується в runtime (під час

запуску), а лише в режимі розробки, він все одно добре вирішує поставлене перед собою завдання.

Fastify взявся за основу для реалізації REST AI з маршрутами для:

1) аутентифікації
2) керування типами сутностей, записами, статусами
3) обробки правил автоматизації
4) ведення журналі дій

3.3.3. База даних

СУБД: PostgreSQL

PostgreSQL є основною базою даних завдяки своїй гнучкості, продуктивності, підтримці реляційних та JSON-структур.

3.3.4. Інтеграція та автоматизація

Використані компоненти:

1) Webhook-сервіс - для виклику зовнішніх URL при виконання автоматизацій.
2) Власна реалізація тригерної системи (AutomationEngine).
3) Сервіси логування подій (ActivityLogServie, AutomationLogService).

Всі інтеграції розроблені у вигляді сервісів, що дозволяє підключати нові зовнішні системи, типи дій для автоматизацій тощо, без втручання в коріння системи.

3.3.5. Супровід, тестування, розгортання

У проєкті використовується:

1) VS Code - основне середовище розробки
2) Система контролю версій Git та Github з GitHub Actions - для контролю версій проєкту та CI/CD (автоматичного тестування, розгортання та доставлення).
3) Postman - для тестування API маршрутів серверної частини проєкту.

Також проєкт спроєктовано з урахуванням можливості автоматичного тестування за допомогою Vitest і подальшого CI-

деплойменту на хмарні платформи, було обрано Fly.io за рахунок вигідної тарифікації та зручних інструментів роботи з сервером, але це опціонально і прямого впливу на роботу розроблювальної системи не має.

Вибраний інструментарій дозволяє створити сучасну, ефективну та масштабовану CRM-систему, що може бути швидко розгорнута, адаптована до малого чи середнього бізнесу та підтримувати високі навантаженні за рахунок асинхронності та надійної роботи з БД. Поєднання Nuxt, Fastify та PostgreSQL надає гнучкість на рівні інтерфейсу, швидкий та масштабований API та структури даних. А використання типізованого підходу та модульної архітектури дозволяє легко підтримувати, розширювати та тестувати функціонал системи у майбутньому.

3.4. Алгоритмізація та програмування програмних модулів

Розробка прикладного ПЗ передбачала реалізацію як клієнтської, так і серверної частини застосунку, з чітким розділенням відповідальностей між модулями та дотриманням принципів модульної архітектури. Для реалізації логіки взаємодії між користувачем, системою та БД були використані сучасні підходи, включаючи REST API, подієву автоматизацію, динамічну обробку даних та облік історичних змін.

Адміністратор перш за все має створити інших користувачів системи. Він сам може вирішувати, кому які доступи надавати: адміністратор чи менеджер. Алгоритм створення користувачів досить простий, він має певний перелік валідацій, хоча основна перевірка - це перевірка на існування електронної пошти.

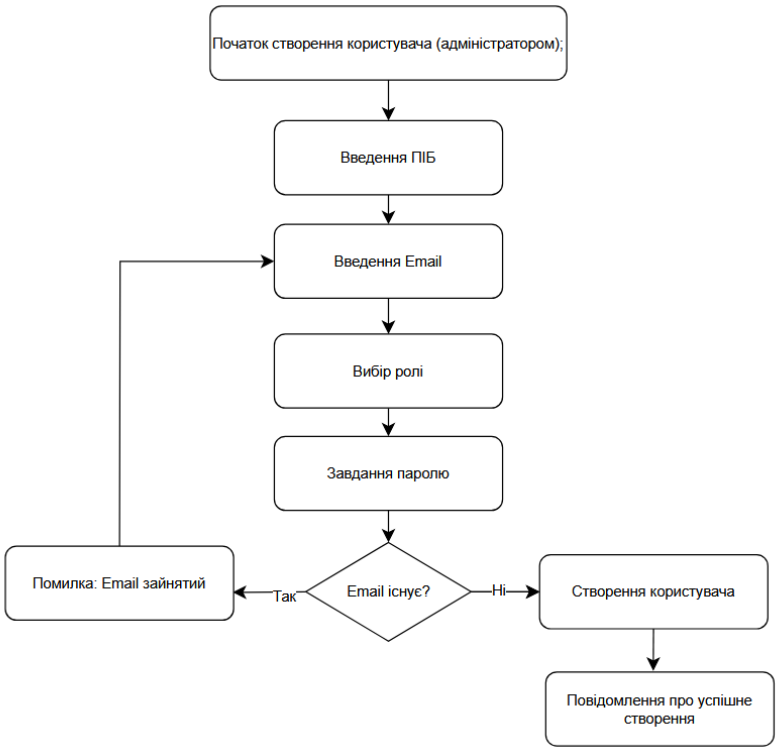

Рис. 17 Алгоритм створення користувача адміністратором

```
async function createUser(client, user) {
        const query = `INSERT INTO users (id,
email, password_hash, name, last_name, role,
is_active, created_at) VALUES ($1, $2, $3, $4, $5,
$6, true, NOW()) `;
        const values = [
                        user.id,
                        user.email,
                        user.password_hash,
                        user.name,
                        user.last_name,
```

Рис. 18 Програмний код створення користувача

Не менш цікавим етапом є створення типу сутності. Цей алгоритм включає в себе саму ідею розробки, оскільки основна ціль - це гнучкість. При створенні типу сутності перш за все треба визначити, за що відповідає ця сутність. Чому вона є саме окремою сутністю, які атрибути (поля) повинна мати та які статуси.

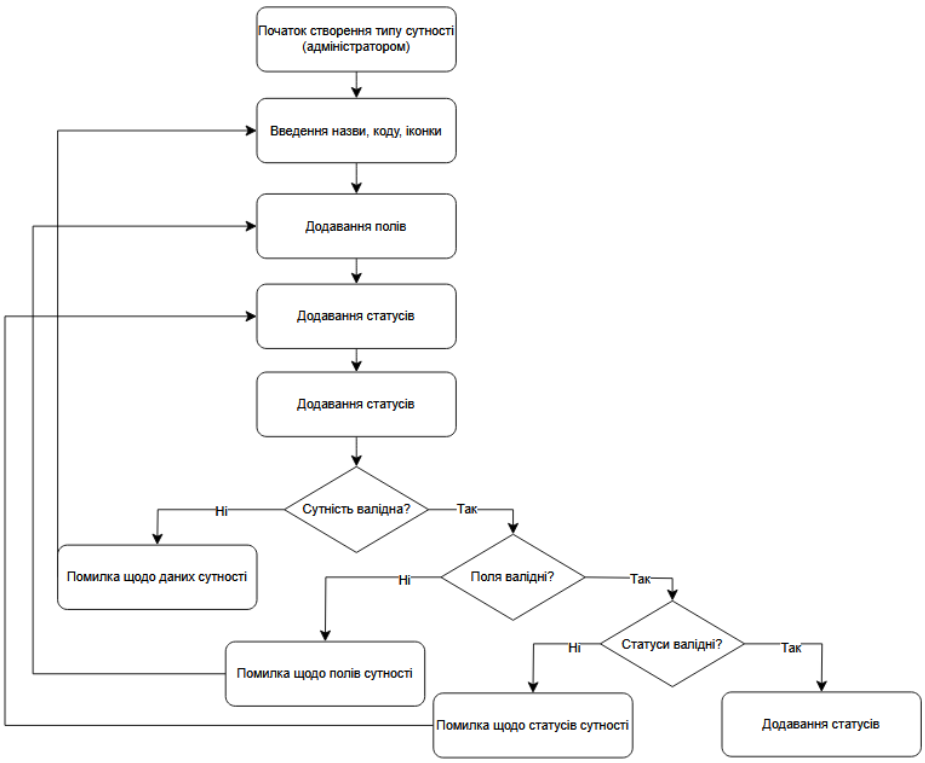

Рис. 19 Алгоритм створення типу сутності адміністратором

```
async function createEntityType(client, entityType)
{
        const query = `
                INSERT INTO entity_types (id,
        created_by, code, name, description, icon,
        created_at)
                VALUES ($1, $2, $3, $4, $5, $6,
        NOW())`;
          const values = [
        entityType.id,
        entityType.created_by,
```

Рис. 20 Програмний код створення типу сутності

Не дивлячись на те, що створення типу сутностей об’єднано із створенням полів та статусів на клієнтській частині, на сервеній частині - це окремі логічні структури. На стороні клієнта у будь-який момент можна буде просто розмежувати функціонал, поки що він досить мінімалістичний та у цьому немає необхідності. Як зображено на рисунку 19, всі функції розділені, вони належать відповідним сервісам, які так чи інакше взаємодіють між собою.

Програмна система розроблена за класичною багаторівневою структурою (tree-tier). Вона включає в себе:

1) Клієнтський рівень (Презентаційний шар). Він відповідає за взаємодію з користувачем, його реалізовано у вигляді односторінкового застосунку (SPA) з використанням Nuxt 3 та Tailwind CSS.
2) Серверна логіка (Шар додатку). Він відповідає за обробку запитів, реалізацію всієї бізнес логіки проєкту, роботу з базою даних, запуск та виконання автоматизацій, спілкування з інтеграціями за допомогою Webhook-сервіса. Реалізовано з використанням Fastify на базі Node.js та TypeScript.
3) Рівень даних (Шар даних). Він реалізований у вигляді СУБД PostgreSQL. Цей рівень забезпечує зберігання всіх даних застосунку: типів сутностей, записів, статусів, полів, журналів, автоматизацій, користувачів тощо.

Модуль створення сутності відповідає за створення запису сутності на основі типу, визначеного користувачем. На вхід приймається структура entity_type_id, об’єкт data, а також status_id. Перед записом відбувається валідація даних згідно з описом полів у entity_fields. Після цього створюється запис у таблиці entity_records, до якого прив’яується користувач, що його створив, а результат повертається клієнту.

```
await entityService.createRecord({
   entityTypeId,
   data,
   statusId,
   createdBy: user.id,
});
```

Рис. 21 Програмний код створення запису сутності

Одразу після створення запускається модуль автоматизації, якщо вказано trigger = onCreate.

Модуль запуску автоматизації сканує automation_rules за entity_type_id і заданим тригером створення, оновлення, зміни статусу тощо. Для кожного правила перевіряється умова (condition), бережена у форматі JSON. Якщо умова виконується, викликається відповідна дія - внутрішня або зовнішня.

```
await automationEngine.handleEvent(entityRecord,
```

Рис. 22 Програмний код виклику події “onCreate”

Модуль побудований на основі централізованого сервісу AutomationEngine, що забезпечує масштабованість і розширюваність логіки.

Модуль обробки дій (внутрішніх і webhook) виконується після проходження умови правило передає свою action подію на виконання. Дія може бути внутрішньою або зовнішньою. Внутрішні дії - це зміна статусу (changeStatus) і зміна поля даних (setField). До зовнішність поки відноситься одна дія - виклик зовнішнього сервісу (callWebhook).

Внутрішні дії виконуються у рамках EntityUpdaterService, який змінює відповідний запис. Зовнішні - чере WebhookService, який відправляє POST-запит на задану URL.

```
switch (action.type) {
   case 'changeStatus':
      await entityService.updateStatus(entityRecord,
action.status);
      break;
   case 'setField':
      await entityService.setField(entityRecord, action.field,
action.value);
      break;
   case 'callWebhook':
```

Рис. 23 Програмний код викликів різних типів автоматизацій

Усі події (створення, оновлення, автоматизації) записуються в таблиці модулю логування:

1. activity_logs - для дій користувачів
2. automation_logs - для результатів виконання правил

Це дає можливість побудувати історію змін та аналітику.

Виклик функцій логування також виконано досить мінімалістично. Все розподілено по окремим сервісам та дозволяє дуже зручно викликати код з будь-якого міся програми на серверній частині.

```
await activityLogService.log({
   entityRecordId: entity.id,
   userId: user.id,
   actionType: 'update',
   payload: data
});
```

Рис. 24 Програмний код викликів різних типів автоматизацій

Спілкування з сервером через REST API:

На клієнтський частині формується форма на основі entity_fields, з полями типу input, select, chackbox тощо. Дані збираються у v-model та передаються на бекенд через HTTP-запит. Фрагмент коду запиту даних з сервера (Vue з використанням useFetch) зображено на рисунку 23. Використовується стандартний метод Nuxt - useFetch. Це дуже зручний

інструмент, який надає можливість мінімалістично спілкуватися з власним бекендом.

```
const { data, error } = await useFetch('/api/entity-
records', {
    method: 'POST',
    body: formData,
});
```

Рис. 25 Програмний код запиту на сервер з клієнтської частини

У разі успіху відображається повідомлення та переходить на сторінку перегляду. При помилках виконується вивід помилок валідації.

4 РЕКОМЕНДАЦІЇ ЩОДО ВПРОВАДЖЕННЯ ТА ЕКСПЛУАТАЦІЇ СИСТЕМИ

4.1 Тестування системи

Тестування - це надзвичайно важливий етап життєвого циклу розробки ПЗ. Тестування надає можливість знайти помилки та перевірити відповідність функціональності вимогам та забезпечити стабільну й надійну роботу системи. У розроблювальному проєкті тестування проводилося в ручному режимі та частково автоматично.

Метою тестування була перевірка на відповідність вимогам створених функцій. Також необхідно виявити логічні помилки у процесах створення, обробки та збереження сутностей. Важливо забезпечити надійність журналювання подій в системі та забезпечити обробку помилок і стабільність при взаємодії з API.

Об’єкти тестування:

1) Тестування охоплювало таки ключові модулі:
2) Створення типу сутності, полів, статусів.
3) Створення запису сутності з динамічними полями.
4) Автоматичне спрацювання правил при створенні або оновлені запису.
5) Оновлення статусів та значень полів через автоматизацію.
6) Логування подій у activity_logs і automation_logs.
7) Клієнтський інтерфейс.

Використовувалися різні типи тестування, як модульне тестування, так і функціональне. Модульне тестування було ручним та частково автоматизованим. Воно проводилося для окремих методів EntityService, AutomationEngine та WebhookService. Для них було створено внутрішні перевірки на коректність параметрів, які були винесені у вигляді умов та обробки помилок. Наприклад, надзвичайно важливим є перевірка типу

JSON-даних перед збереженням у відповідне поле, бо вони мають гнучку структуру та повинні відповідати певним правилам, які з часом можуть розширюватися, тому необхідно підтримувати зворотню сумісність (підтримка застарілих версій продукту у більш нових версіях). До функціонального тестування відноситься перевірка всіх основних сценаріїв використання системи, наприклад створення типу сутності, додавання полів та статусів, створення записів, виконання автоматизації, перегляд журналів.

Тестування API виконувалось за допомогою Postman (додаток для тестування API з підтримкою REST, GraphQL, WebSocket тощо). За основу була взята валідація даних, обробка помилок (наприклад при відсутноті поля), відповідність структурі статусів HTTP-запитів (200 - успіх, 400 - некоректність запиту зі стороні клієнта, 403 - доступ заборонено).

Також проводилося інтеграційне тестування, у якому перевіряється робота взаємодії між модулями. Інтеграційне тестування - це перевірка того, як частини розроблювальної системи працюють одна з одною. При створені сутності та створення відповідного запису у БД, при запуску правила та зміни відповідних логів. На рисунку 24 зображення приклад одного з інтеграційних тестів, який створює нового користувача.

```
const request = require('supertest');
const app = require('../app');
describe('User integration', () => {
  it('створює нового користувача', async () => {
        const res = await request(app.server)
        .post('/api/users')
        .send({
        email: 'test@example.com',
        password: 'password123',
        name: 'Іван',
        last_name: 'Іванов',
        role: 'user'
        });

        expect(res.statusCode).toBe(201);
        expect(res.body).toHaveProperty('id');
        expect(res.body.email).toBe('test@example.com')
;
  });
```

Рис. 26 Програмний код інтеграційного тестування створення нового користувача

Виявлені помилки та усунення: Під час тестування були виявлені типові помилки щодо невірної обробки пустих значень у полях типу select, подвійного працювання автоматизації при зміні статусу, некоректного форматування JSON-умов при додаванні правил у ручному режимі, а також відсутності валідації на клієнтській частині для обов’язкових полів.

Після функціонального тестування система показала себе стабільно виходячи із базових сценаріїв, також вона повністю відповідає функціональним вимогам та готовність до подального розвитку. Тестування дозволило переконатися у надійності реалізованих функцій та готовності системи до використання у реальних умовах. Модульний підхід дозволив зробити систему зручною для перевірки та підтримки.

В ручному режимі проводилося тестування UI-елементів та всього додатку. Так, на рисунку 25 зображено вікно авторизації користувача в системі.

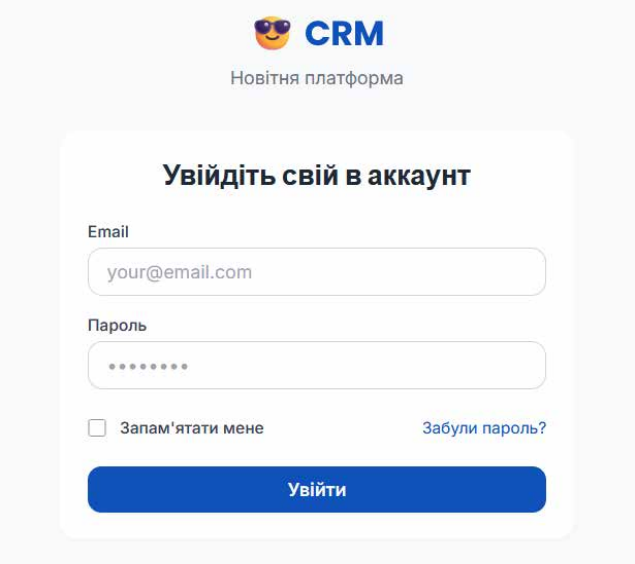

Рис. 27 Вікно авторизації користувача в CRM-системі

Далі адміністратор повинен створити певний тип сутності. Йому необхідно задати назву, кодову назву та вибрати зображення сутності. Разом з цим йому пропонується створити статуси, вказати до них кольори та назви. На рисунку 26 зображено весь процес.

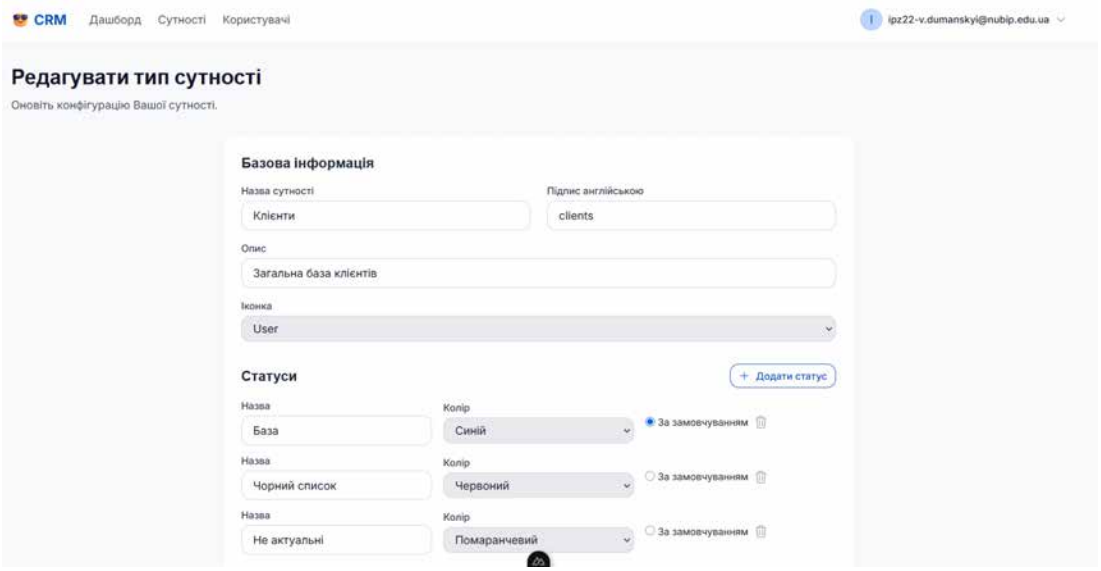

Рис. 28 Створення типу сутності та статусів

На рисунку 27 зображено перелік полів, які створюються користувачем. Він задає назву, тип, підпис та кодову назву поля. Також доступна опція обов’язковості цього поля. Кожне поле можна видалити по ходу додавання чи редагування типу сутності.

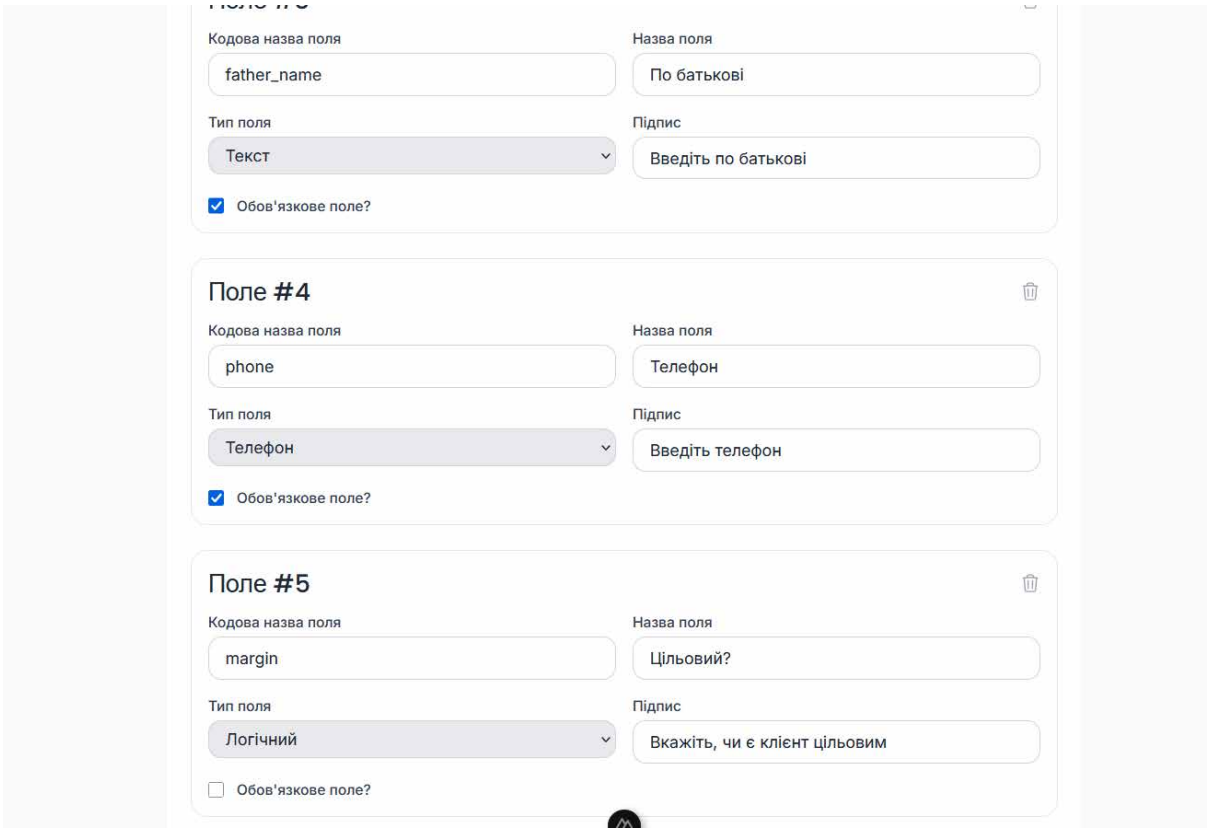

Рис. 29 Створення полів для типу сутності

Після того, як тип сутності створено, необхідно протестувати те, як створюється сам запис сутності. Для цього у ручному та автоматизованому режимах було створено понад 15 різних записів (клієнтів). На рисунку 28 зображено форму додавання запису (клієнту в даному випадку).

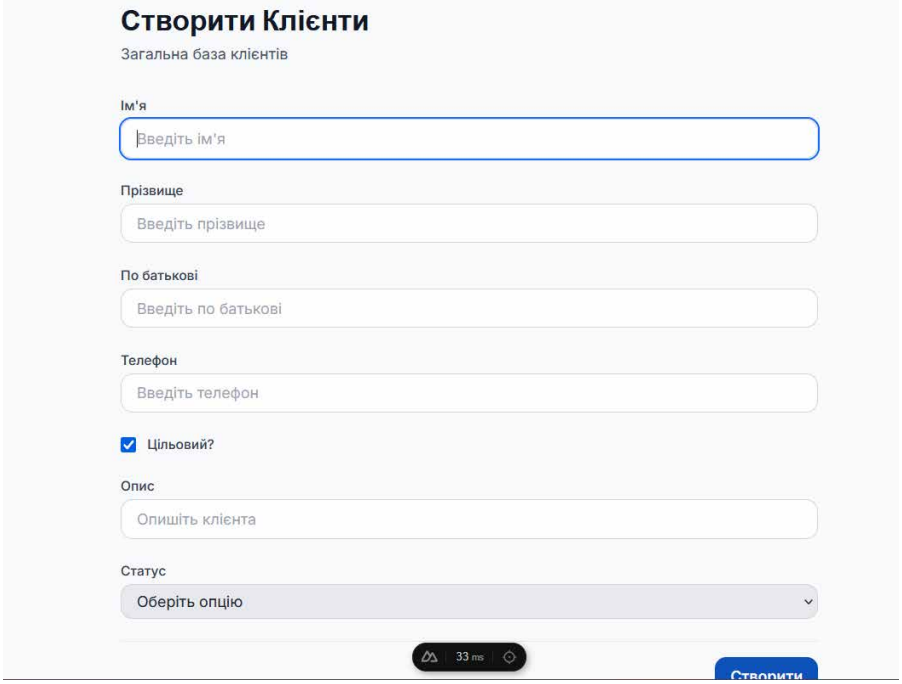

Рис. 30 Форма створення запису сутності

На сторінці певної сутності доступний канбан із списком записів. Кожен запис групується по статусам та має коротку інформацію про себе. Їх можна переміщати між статусами просто перетягуючи мишкою. На рисунку 29 зображено також просту аналітику, яка допомагає більш точно зрозуміти, скільки записів було створено за який період.

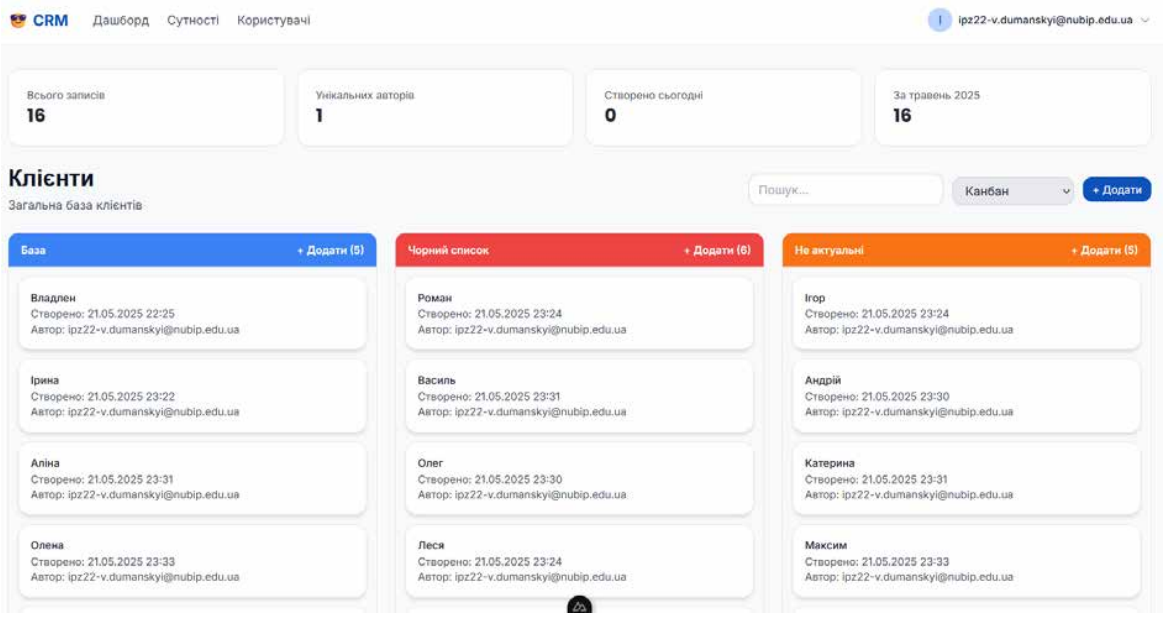

Рис. 31 Сторінка певної сутності у режимі “Канбан” та аналітика

Список записів має режим списку, окрім канбану. Також на сторінці доступний пошук по всім полям сутностей. Зараз пошук організовано як повнотекстовий.

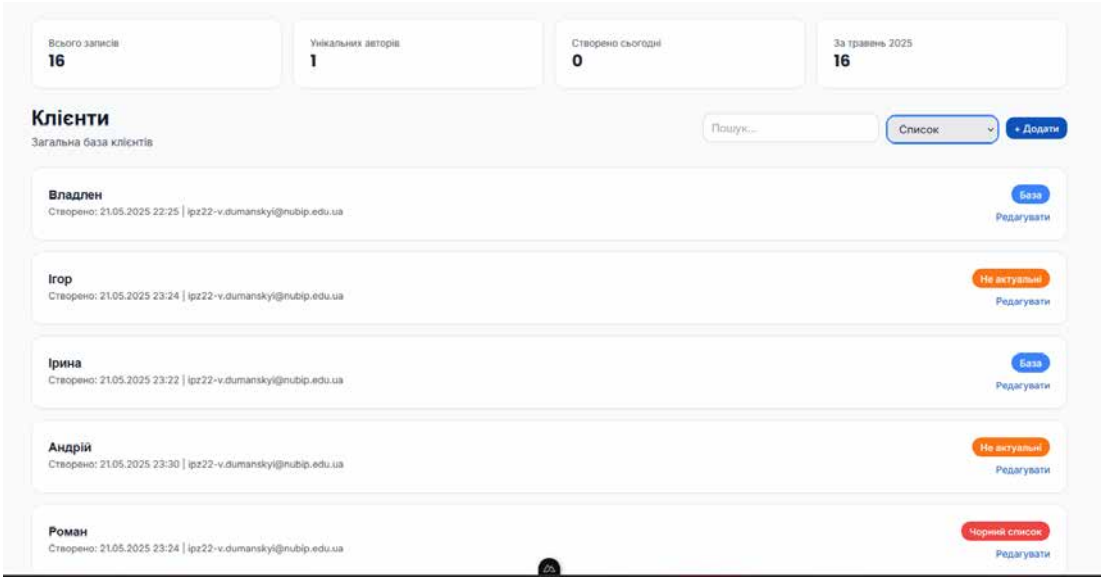

Рис. 32 Сторінка певної сутності у режимі “Список” та аналітика

У пошуку було введено ім’я “Роман”, система знайшла два записи: клієнт з ім’ям “Роман” та клієнтку з прізвищем “Романівна”.

Рис. 33 Демонстрація пошуку записів

Важливим етапом є редагування, видалення та детальний перегляд інформації про запис. У модальному вікні запису є поля з зазначеними даними, автор, дата створення, “Таймлайн” з історією змін полів та статусів, а також функціональні кнопки, які дозволяють опрацьовувати поточний запис. Всі функції були протестовані, а помилки були виправлені.

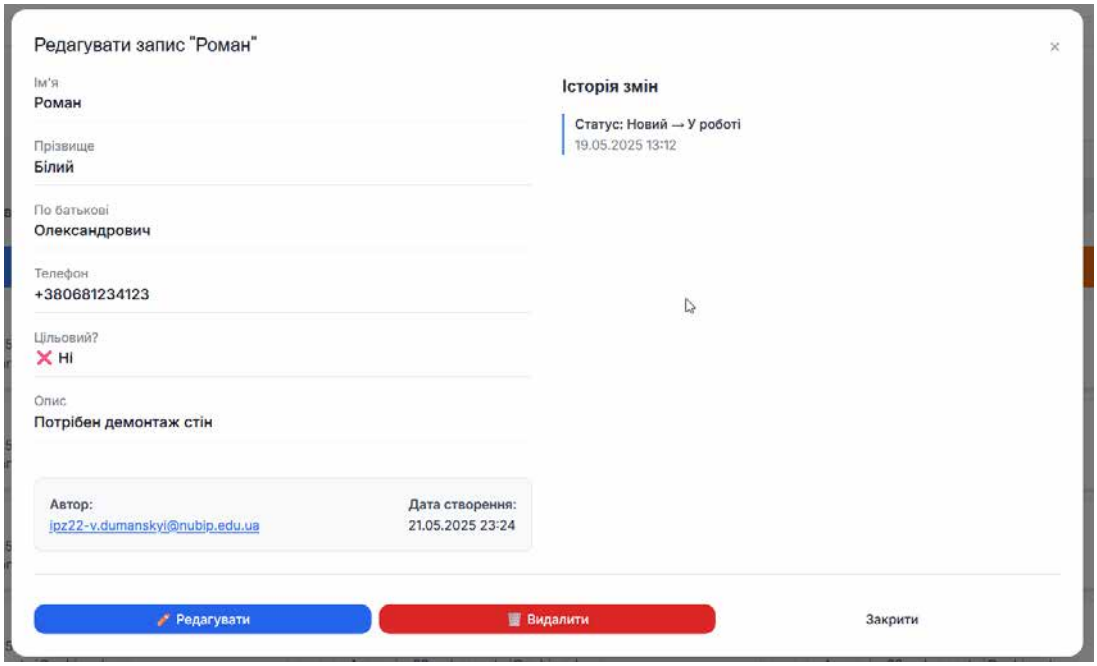

Рис. 34 Модальне вікно детальної інформації про запис

Також було протестовано редагування запису. Були змінені всі поля та статуси, валідація полів працює відмінно.

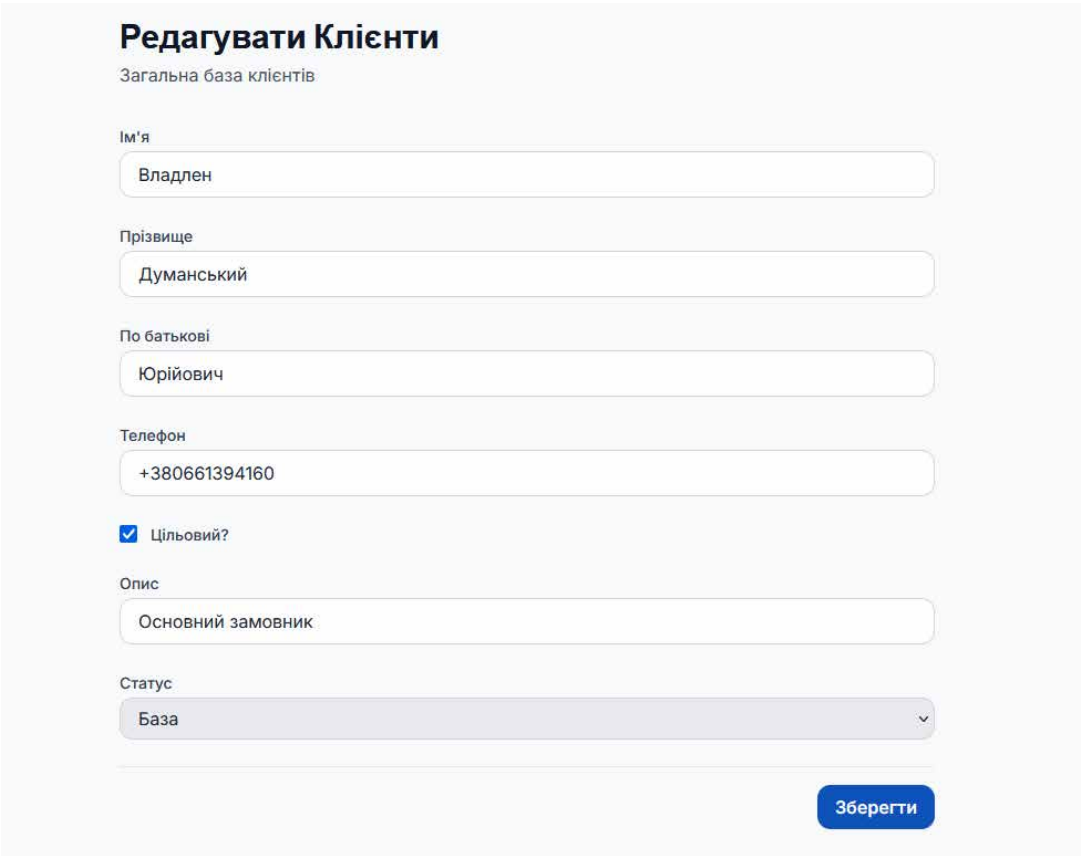

Рис. 35 Редагування запису сутності

Надзвичайно цікавим є розділ автоматизації. Була створена автоматизація для тестування. Її логіка полягає в тому, що якщо при створені клієнта його бюджет перевищує 100 000 гривень, то клієнт відразу попадає у статус “в роботі”, бо це клієнт є потенційно більш прибутковим та його обробку потрібно почати в тестовому режимі.

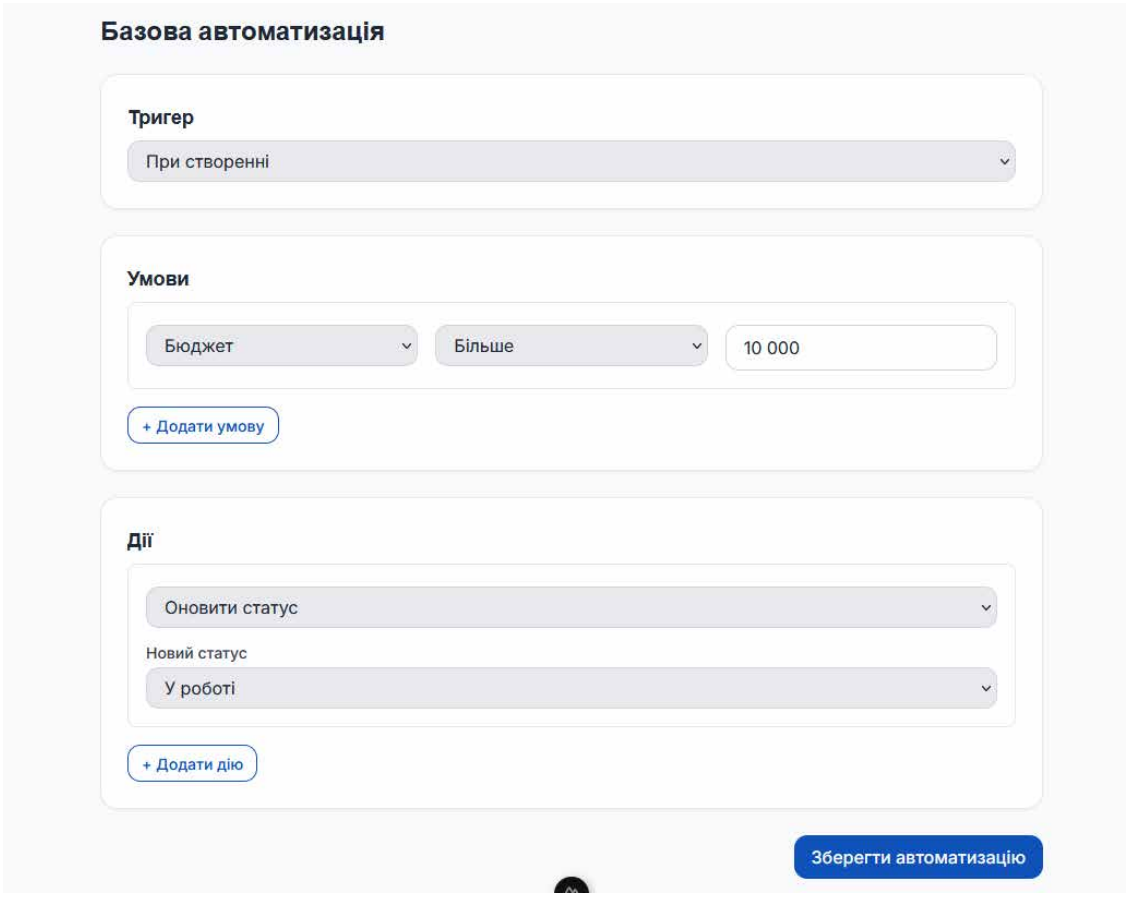

Рис. 36 Створення автоматизації

Були проведені тести із створеною автоматизацією, результати задовільні. Автоматизація спрацювала при створенні нового клієнта. На сторінці списку зображені створені для тестування автоматизації поточного типу сутності, що зображено на рисунку 32.

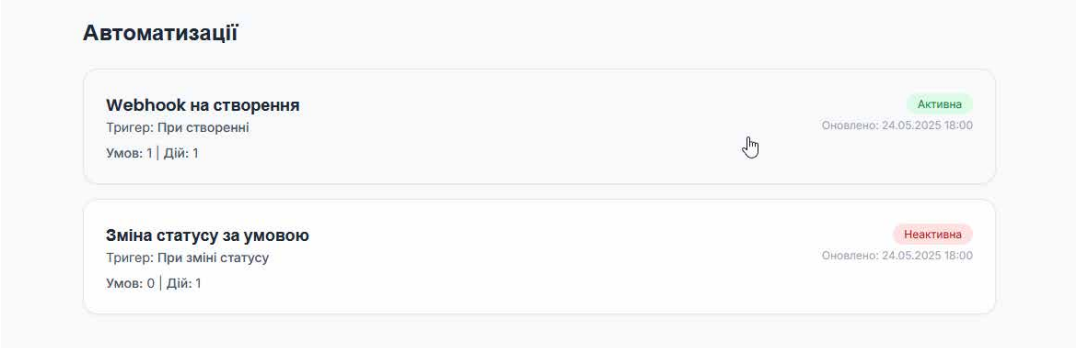

Рис. 37 Список автоматизацій

Також необхідно перевірити у ручному режимі, як додаються та видаляються користувачі із системи. Лише адміністратор має доступ до цієї сторінки. Список користувачів повністю відповідає створеним користувачам, видалення та авторизація під новими користувачами також працює.

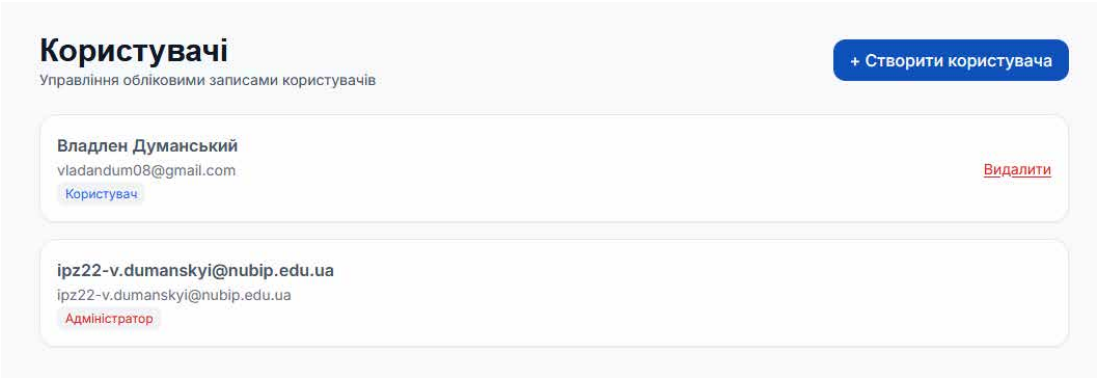

Рис. 38 Список користувачів

4.2. Вимоги до апаратного та програмного забезпечення

Для правильної роботи розробленої CRM-системи необхідно дотримуватися певних вимог до програмного та апаратного забезпечення. Система має клієнт-серверну архітектуру, використовує сучасні та поширені веб-технології. Вимоги поділяються на два блоки: вимоги до клієнтської сторони, тобто до користувача та вимоги до серверної сторони, тобто розгортання та обробка запитів.

4.2.1. Вимоги до клієнтського середовища:

Користувачам буде достатньо стандартного веб-браузера для доступу у мережу “Інтернет”. Весь інтерфейс працює у вигляді односторінкового веб-застосунку (SPA - Single Page Application).

Мінімальні вимоги:

1) Операційна система: Windows 7+, macOS 10+, Linux Ubuntu 16+, Android 8+, IOS 12+
2) Браузер: Google Chrome 95+, Mozilla Firefox 95+, Safari 12+, Microsoft Edge 95+
3) Процесор користувацького пристрою: будь-який x64/ARM.
4) Оперативна пам’ять: 2 ГБ і більше
5) Роздільна здатність дисплея: мінімум 1024x768 пікселів.
6) Інтернет-з’єднання: стабільне з пропускною здатністю не менше 1 Мбіт/с.

Програмні вимоги:

1) Увімкнена підтримка JavaScript та локального зберігання (localStorage)
2) Сумісність з HTTPS-протоколом

4.2.2. Вимоги до серверного середовища

Серверна частина розроблена на платформі Node.js з використанням фреймворку Fastify та СУБД PostgreSQL, тому для її роботи необхідно середовище, яке підтримує запуск Node-процесів, доступ до СУБД, а також, бажано, можливість розширення у майбутньому.

Мінімальні вимоги до серверу:

1) Операційна система: Linux (Ubuntu 20.04 LTS або новіше), можливе розгортання на Windows Server 2019+
2) Процесор: 1 ядро або вище (x64, ARM64), або не менше двух віртуальних ядер.
3) Оперативна пам’ять: від 1 ГБ.

4) Міце на диску: мінімум 5 ГБ з урахуванням бази, логів, та резервних копій.
5) Мережа: публічна IP-адреа, відкриті порти 80 / 443 / 3000.

Програмне забезпечення:

1) Node.js 18.x або новіше
2) npm у якості менеджера пакетів
3) PostgreSQL версіх 13 або новіше
4) Nginx як зворотній проксі та керування HTTPS
5) Docker для ізоляції та зручного розгортання

4.2.3. Вимоги для локального середовища розробника

Для тестування та опрацювання проєкту локально необхідно налаштувати середовище розробки з такими характеристиками:

1) ОС: Windows, macOS, Linux
2) Редактор коду: рекомендується Virtual Studio Code
3) Встановлені інструменти: Node.js , PostgreSQL, git, npm
4) Браузер: Chrome, Firefox для тестування інтерфейсу
5) Пам’ять: від 4 ГБ ОЗП

4.2.4. Масштабування та хостинг

Для продуктивного використання у робочому середовищі рекомендується розгортання на хмарних інфраструктурах з можливістю вертикального масштабування. До рекомендацій були винесені платформи: Fly.io, Heroku, DigitalOcean (гнучкіше, але складніше у налаштування). Для резервного копіювання бажано налаштувати на сервері PostgreSQL dump.

CRM-система розроблена з урахуванням кросплатформеності та мінімальних системних вимог, що дозволяє працювати з нею як на сучасних пристроях, так і на хмарних середнього рівня. Завдяки використанню веб-технологій, клієнтський інтерфейс доступний з будь-якого пристрою з браузером, а серверна частина вимагає лише стандартного середовища, яке було зазначено вище.

4.3 Склад інсталяційного пакету

Інсталяційний пакет ПЗ CRM-системи для малого і середнього бізнесу включає всі необхідні компоненти для розгортання та початку роботи як на локальному сервері, так і в хмарному середовищі. Пакет структуровано за принципом модульності з розділенням на фронтенд, бекенд, базу даних та документацію.

До складу інсталяційного пакету входять такі компоненти:

1) /backend - серверна частина, розроблена на Node.js із використанням фреймворку Fastify. Містить бізнес-логіку, маршрути API, обробку авторизації, роботу з БД тощо.
2) /frontend - клієнтська частина, побудована на Nuxt 3 (Vue). Забезпечує зручний інтерфейс для роботи з додатком.
3) /.env.example - шаблон конфігураційного файлу середовища.
4) /docs - коротка документація по запуску, опис основних REST-ендпоінтів, структура сутностей.
5) /docker - конфігураційні файли Docker та docker-compose.yaml для швидкого розгортання.
6) package.json / tsconfig.json / nuxt.config.ts - системні файли для встановлення залежностей і компіляції проєкту.

ВИСНОВКИ

У ході виконання бакалаврської кваліфікаційної роботи на тему “CRM-система для малого і середнього бізнесу” було реалізовано повний цикл проєктування та створення системи, яка орієнтована на автоматизацію обліку, керування сутностями та процесами в межах малого підприємства.

Було проведено системний аналіз управління бізнес-процесами малого та середнього бізнесу, що дало змогу виявити ключові потреби бізнесу та проблеми сучасних CRM-систем, а саме низький рівень інтуїтивності, обменежу гнучкість, надлишкову складність функціоналу та високу вартість підтримки. Сформульовано функціональні та нефункціональні вимоги до програмного забезпечення. На їх основі побудовано логічну ER-модель даних, описано взаємозв’язки між сутностями та реалізовано схему, що підтримує динамічні типи сутностей, статуси, поля та правила автоматизації.

Виконано моделювання архітектури системи. Була створена діаграма класів, а також діаграми пакетів, компонентів та кооперацій, які відображають логіку взаємодії користувачів, даних і функціональних модулів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1) CRM-системи. Що це та як працює? [Електронний ресурс] - Джерело: https://www.creatio.com/ua/crm/what-is-crm
2) Діаграми UML [Електронний ресурс] - Джерело: https://evergreens.com.ua/ua/articles/uml-diagrams.html
3) Найкращі CRM-системи для бізнесу. [Електронний ресурс] - Джерело: https://hostiq.ua/blog/ukr/best-crm-systems
4) PostgreSQL - що це і для чого використовується? [Електронний ресурс] - Джерело: https://foxminded.ua/postgresql-shcho-tse
5) Nuxt 3 - Офіційна документація [Електронний ресурс] - Джерело: https://nuxt.com/docs/guide
6) Модель діаграми зв’язків сутностей (ER) із прикладом СУБД [Електронний ресурс] - Джерело: https://www.guru99.com/uk/er-diagram-tutorial-dbms.html
7) TypeScript - плюси і мінуси використання в розробці [Електронний ресурс] - Джерело: https://foxminded.ua/ru/typescript
8) Що таке діаграма класів UML [Електронний ресурс] - Джерело: https://www.mindonmap.com/uk/blog/what-is-uml-class-diagram/

Додаток А

ОПУБЛІКОВАНІ ТЕЗИ НА VII ВСЕУКРАЇНСЬКІЙ НАУКОВО-ПРАКТИЧНІЙ ІНТЕРНЕТ КОНФЕРЕНЦІЇ СТУДЕНТІВ І АСПІРАНТІВ «ТЕОРЕТИЧНІ ТА ПРИКЛАДНІ АСПЕКТИ РОЗРОБКИ КОМП'ЮТЕРНИХ СИСТЕМ 2025» (24 КВІТНЯ 2025 РОКУ)

Сторінок – 2

2025

УДК 004.42

ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ CRM-СИСТЕМИ ДЛЯ МАЛОГО ТА СЕРЕДНЬОГО БІЗНЕСУ

Думанський В. Ю., науковий керівник: Василюк-Зайцева С. В.

У сучасному світі цифрових технологій малі та середні підприємства стикаються з необхідністю автоматизації процесів управління взаємодією з клієнтами. У зв’язку з високою конкуренцією та зростаючими вимогами до якості обслуговування виникає потреба у впровадженні CRM-систем (Customer Relationship Management) — програмного забезпечення для ефективного управління контактами, продажами, маркетингом та сервісом.

CRM-система дозволяє зберігати історію взаємодії з клієнтами, контролювати хід угод, планувати задачі, вести базу контактів, налаштовувати воронки продажів, інтегруватись з іншими сервісами, та загалом підвищує продуктивність бізнесу. Розробка власної CRM-системи передбачає створення базового функціоналу, який охоплює ключові потреби малого та середнього бізнесу:

Модуль	Опис
Управління клієнтами	Зберігання даних про клієнтів, історії взаємодій, угод, задач
Воронка продажів	Налаштування етапів продажу, контроль стадій, візуалізація процесів
Завдання та нагадування	Створення задач, дедлайнів, нагадування менеджерам
Комунікація	Інтеграція з email, телефоном, месенджерами (API)
Аналітика та звітність	Графіки продажів, ефективність співробітників, джерела заявок
Управління користувачами	Ролі, права доступу, логування активностей
Інтеграції	Підключення зовнішніх сервісів, можливість роботи з REST API

Табл. 1 - Основний функціонал системи

Система буде реалізована як веб-застосунок, що забезпечить доступ з будь-якого пристрою через браузер. Планується використання сучасного технологічного

стеку: **Nuxt.js** для фронтенду, **Node.js** (Fastify.js) для бекенду, **PostgreSQL** як БД, та REST API для взаємодії між модулями.

CRM-система	Платформа	Переваги	Недоліки
Zoho CRM	SaaS	Гнучка настройка, AI-аналітика, автоматизація процесів	Платна підписка, складний старт
Salesforce	SaaS	Потужна функціональність, масштабованість, велика екосистема	Висока вартість, складна інтеграція
HubSpot CRM	SaaS (є безкоштовна версія)	Простий інтерфейс, швидкий запуск, інтеграції	Обмежений функціонал у безкоштовній версії
Pipedrive	SaaS	Орієнтація на продажі, зручна воронка, візуалізація	Менше можливостей для кастомізації
Власна CRM	Web	Повна кастомізація, контроль над даними, безпека	Потрібен час на розробку, технічна підтримка

Табл. 2 - Основний функціонал системи

ВИСНОВКИ

Створення CRM-системи власного виробництва є доцільним для малого та середнього бізнесу з метою повного контролю над функціональністю, інтерфейсом і безпекою. Таке рішення дозволяє гнучко адаптуватися до потреб конкретного підприємства, уникнути зайвих витрат на ліцензії та обмеження сторонніх сервісів. Наразі система знаходиться на етапі розробки основного функціоналу, після чого буде створено прототип, проведено тестування на реальних сценаріях використання та подальша оптимізація згідно з фідбеком користувачів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ:

1. IBM – What is CRM? - https://www.ibm.com/topics/crm
2. Zoho CRM – Official - https://www.zoho.com/crm/
3. Salesforce – CRM Overview - https://www.salesforce.com/crm/
4. HubSpot – What is CRM? - https://www.hubspot.com/crm

Додаток Б

SQL-ЗАПИТИ

Сторінок – 2

2025

```
CREATE TABLE users (
        id SERIAL PRIMARY KEY,
        email TEXT NOT NULL UNIQUE,
        password_hash TEXT NOT NULL,
        name TEXT,
        last_name TEXT,
        role TEXT NOT NULL,
        is_active BOOLEAN DEFAULT TRUE,
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE entity_types (
        id SERIAL PRIMARY KEY,
        created_by INT REFERENCES users(id),
        code TEXT NOT NULL UNIQUE,
        name TEXT NOT NULL,
        description TEXT,
        icon TEXT,
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
CREATE TABLE entity_fields (
        id SERIAL PRIMARY KEY,
        field_key TEXT NOT NULL,
        entity_type_id INT REFERENCES entity_types(id),
        label TEXT NOT NULL,
        type TEXT NOT NULL,
        is_required BOOLEAN DEFAULT FALSE,
        options JSON,
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
CREATE TABLE entity_statuses (
        id SERIAL PRIMARY KEY,
        entity_type_id INT REFERENCES entity_types(id),
        label TEXT NOT NULL,
        color TEXT NOT NULL,
        is_default BOOLEAN DEFAULT FALSE
);
CREATE TABLE entity_records (
        id SERIAL PRIMARY KEY,
        created_by INT REFERENCES users(id),
        entity_type_id INT REFERENCES entity_types(id),
        status_id INT REFERENCES entity_statuses(id),
        data JSONB NOT NULL,
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
        updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
CREATE TABLE automation_rules (
        id SERIAL PRIMARY KEY,
        entity_type_id INT REFERENCES entity_types(id),
        trigger TEXT NOT NULL,
```

```
    condition JSON,
    action JSON,
    is_active BOOLEAN DEFAULT TRUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
CREATE TABLE activity_logs (
    id SERIAL PRIMARY KEY,
    entity_record_id INT REFERENCES entity_records(id),
    user_id INT REFERENCES users(id),
    action_type TEXT NOT NULL,
    payload JSON,
    timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
CREATE TABLE automation_logs (
    id SERIAL PRIMARY KEY,
    triggered_by INT REFERENCES users(id),
    entity_record_id INT REFERENCES entity_records(id),
    result TEXT NOT NULL, -- success / failed
    rule_id INT REFERENCES automation_rules(id),
    payload JSON,
    timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Додаток В

ТАБЛИЦІ

Сторінок – 1

2025

Таблиця 1

Порівняння існуючих CRM-систем

Параметр / Система	Zoho CRM	HubSpot CRM	ERP Next	Espo CRM	Пропонована система
Гнучкість структури даних	-	-	+	+ -	+
Простота інтерфейсу	-	+	-	+ -	+
Автоматизація без коду	+ -	-	-	+ -	+
Кастомізація без програміста	+ -	-	-	+ -	+
Наявність інтеграцій (API)	+	+ -	+ -	+ -	+
Орієнтація на МСБ	+ -	+ -	-	+ -	+
Безкоштовне використання	-	+ -	+	+	+

Додаток Г

КОД РЕАЛІЗАЦІЇ ОСНОВНИХ ФУНКЦІЙ

Сторінок – 7

2025

Реалізація механізму авторизації на серверній частині:

```
import { FastifyInstance } from "fastify";
import { compare } from "bcryptjs";
import { sign } from "jsonwebtoken";
import { db } from "../db";

export default async function (app: FastifyInstance) {
  app.post("/auth/login", async (req, reply) => {
        const { email, password } = req.body as { email: string; password: string };
        const user = await db.users.findFirst({
        where: { email, is_active: true },
        });
        if (!user || !(await compare(password, user.password_hash))) {
        return reply.status(401).send({ error: "Невірні дані" });
        }
        const token = sign(
        { id: user.id, email: user.email, role: user.role },
        process.env.JWT_SECRET!,
        { expiresIn: "7d" }
        );
        return {
        token,
        user: {
        id: user.id,
        name: user.name,
        last_name: user.last_name,
        email: user.email,
        role: user.role,
        },
        };
  });
}
```

Реалізація механізму створення типу сутності:

```
import { FastifyInstance } from "fastify"; import { db } from "../db";


export default async function (app: FastifyInstance) {
  app.post("/entity-types", async (req, reply) => {
        const body = req.body as {
        name: string; code: string; description?: string; icon?: string;
        fields?: { field_key: string; label: string; type: string; is_required?: boolean; options?:
any; }[];
        statuses?: { label: string; color: string; is_default?: boolean; }[];
        };
        const entity = await db.entity_types.create({
        data: { name: body.name, code: body.code, description: body.description, icon:
body.icon, created_by: req.user.id }
        });
        if (body.fields?.length) {
                await db.entity_fields.createMany({
                data: body.fields.map(f => ({
                entity_type_id: entity.id,
                field_key: f.field_key,
                label: f.label,
                type: f.type,
                is_required: f.is_required ?? false,
                options: f.options
        }))
        });
        }
        if (body.statuses?.length) {
                await db.entity_statuses.createMany({
                data: body.statuses.map(s => ({
                entity_type_id: entity.id,
                label: s.label,
```

```
                color: s.color,
                is_default: s.is_default ?? false
                }))
        });
        }
        return { id: entity.id };
  });
}
```

Реалізація механізму створення запису сутності:

```
import { FastifyInstance } from "fastify"; import { db } from "../db";

export default async function (app: FastifyInstance) {
  app.post("/entity-records", async (req, reply) => {
                const body = req.body as {
                entity_type_id: number;
                status_id?: number;
                data: any;
        };
        const record = await db.entity_records.create({
                data: {
                        created_by: req.user.id,
                        entity_type_id: body.entity_type_id,
                        status_id: body.status_id,
                        data: body.data
                }
        });
        await db.activity_logs.create({
                data: {
                        entity_record_id: record.id,
                        user_id: req.user.id,
                        action_type: "create",
                        payload: body.data
```

```
            }
        });
        return { id: record.id };
  });
}
```

Реалізація механізму запуску автоматизації:

```
const rules = await db.automation_rules.findMany({
  where: { entity_type_id: body.entity_type_id, trigger: "onCreate", is_active: true }
});
for (const rule of rules) {
  const pass = await evaluateCondition(rule.condition, body.data);
  if (!pass) continue;
  try {
        if (rule.action?.type === "webhook") {
        await sendWebhook(rule.action.url, rule.action.payload);
        } else if (rule.action?.type === "update") {
        const updated = { ...body.data, ...(rule.action.fields || {}) };
        await db.entity_records.update({ where: { id: record.id }, data: { data: updated,
updated_at: new Date() } });
        await db.activity_logs.create({ data: { entity_record_id: record.id, user_id: req.user.id,
action_type: "auto_update", payload: updated } });
        }
        await db.automation_logs.create({
        data: {
              entity_record_id: record.id,
              rule_id: rule.id,
              triggered_by: req.user.id,
              result: "success",
              payload: rule.action
        }
        });
  } catch (e) {
        await db.automation_logs.create({
```

```
        data: {
                entity_record_id: record.id,
                rule_id: rule.id,
                triggered_by: req.user.id,
                result: "failed",
                payload: { error: e.message }
        }
        });
  }
}
```

Реалізація механізму створення користувача:

```
import { FastifyInstance } from "fastify";
import { hash } from "bcryptjs";
import { db } from "../db";

export default async function (app: FastifyInstance) {
  app.post("/users", async (req, reply) => {
        const body = req.body as {
                email: string;
                password: string;
                name?: string;
                last_name?: string;
                role: string;
        };
        const exists = await db.users.findFirst({ where: { email: body.email } });
        if (exists) return reply.status(409).send({ error: "Користувач вже існує" });
        const user = await db.users.create({
        data: {
                email: body.email,
                password_hash: await hash(body.password, 10),
                name: body.name,
                last_name: body.last_name,
```

```
            role: body.role
        }
        });
        return { id: user.id };
  });
}
```

Реалізація сервісу Webhook:

```
import axios from "axios";
export type WebhookPayload = {
  event: string;
  data: any;
  metadata?: Record<string, any>;
};
export type WebhookConfig = {
  url: string;
  method?: "POST" | "PUT";
  headers?: Record<string, string>;
  timeout?: number;
};
export class WebhookService {
  static async send(config: WebhookConfig, payload: WebhookPayload) {
        const method = config.method || "POST";
        const timeout = config.timeout || 5000;
        const headers = {
        "Content-Type": "application/json",
        ...(config.headers || {})
        };
        try {
            const response = await axios.request({
                url: config.url,
                method,
                headers,
```

```
                data: payload,
                timeout
            });

            return { status: response.status, success: true };
        } catch (err: any) {
            console.error(`[Webhook] ${method} ${config.url} failed:`, err.message);
            return { status: err.response?.status || 500, success: false, error: err.message };
        }
  }
}
```

Реалізація механізму використання Webhook:

```
await WebhookService.send(
  { url: "https://example.com/hook", headers: { Authorization: "Bearer abc" } },
  { event: "entity_created", data: entityRecord, metadata: { source: "crm" } }
);
```

Додаток Ґ

СКРІНШОТИ ПРОГРАМИ

Сторінок – 6

2025

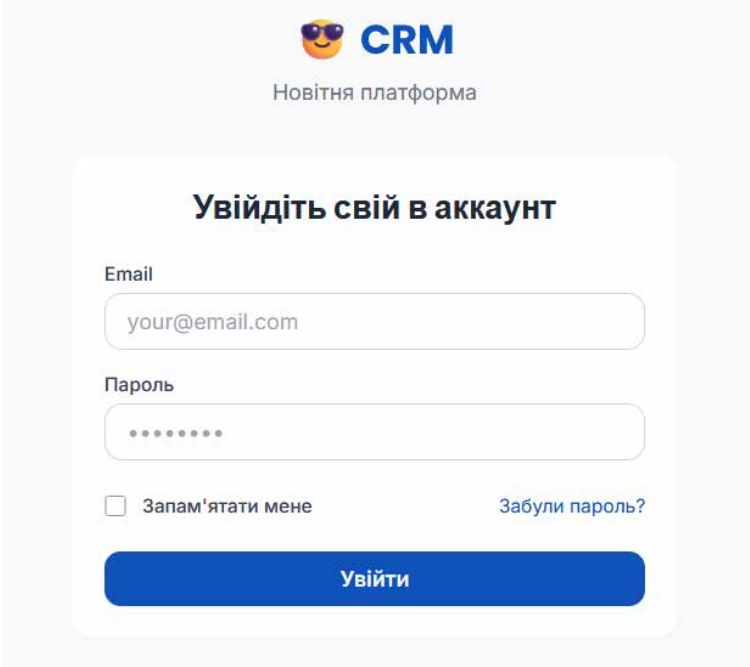

Рис. 27 Вікно авторизації користувача в CRM-системі

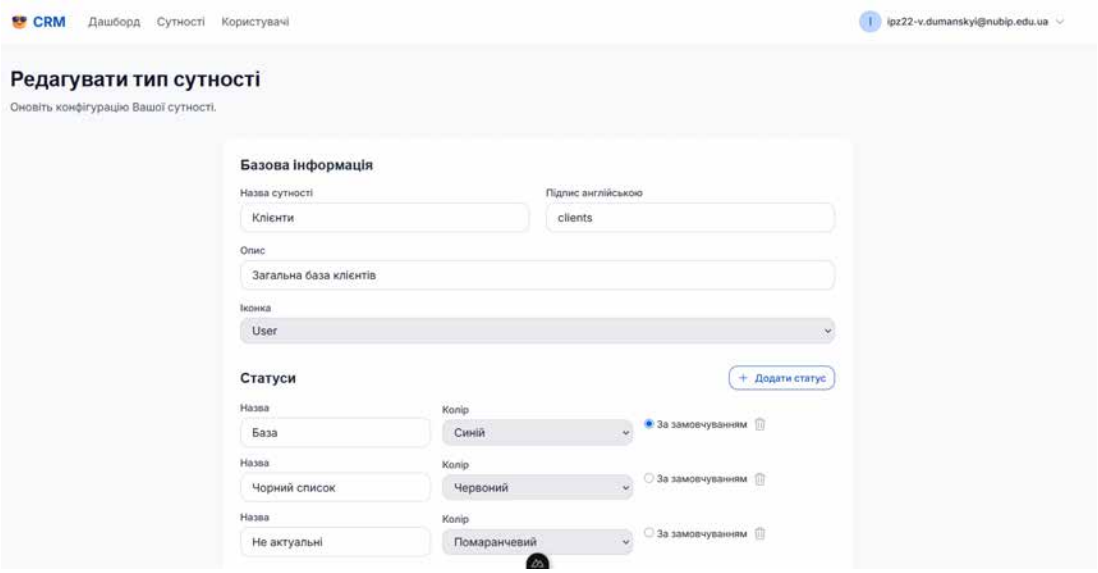

Рис. 28 Створення типу сутності та статусів

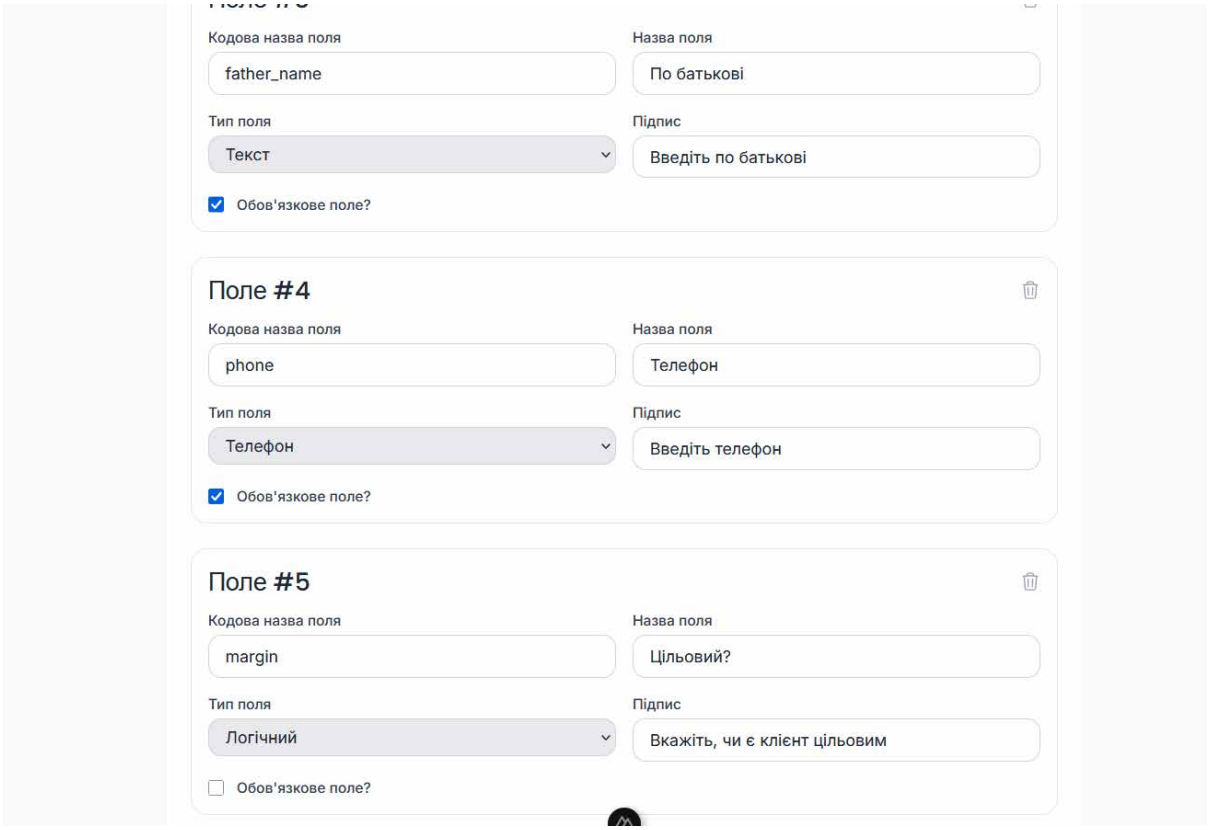

Рис. 29 Створення полів для типу сутності

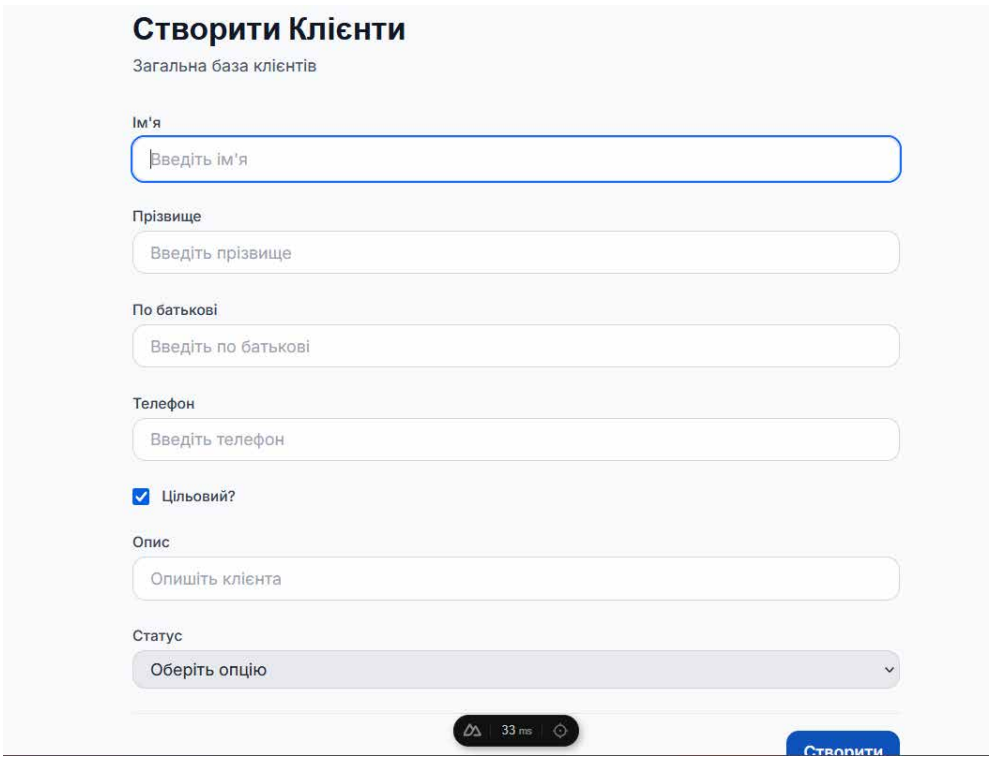

Рис. 30 Форма створення запису сутності

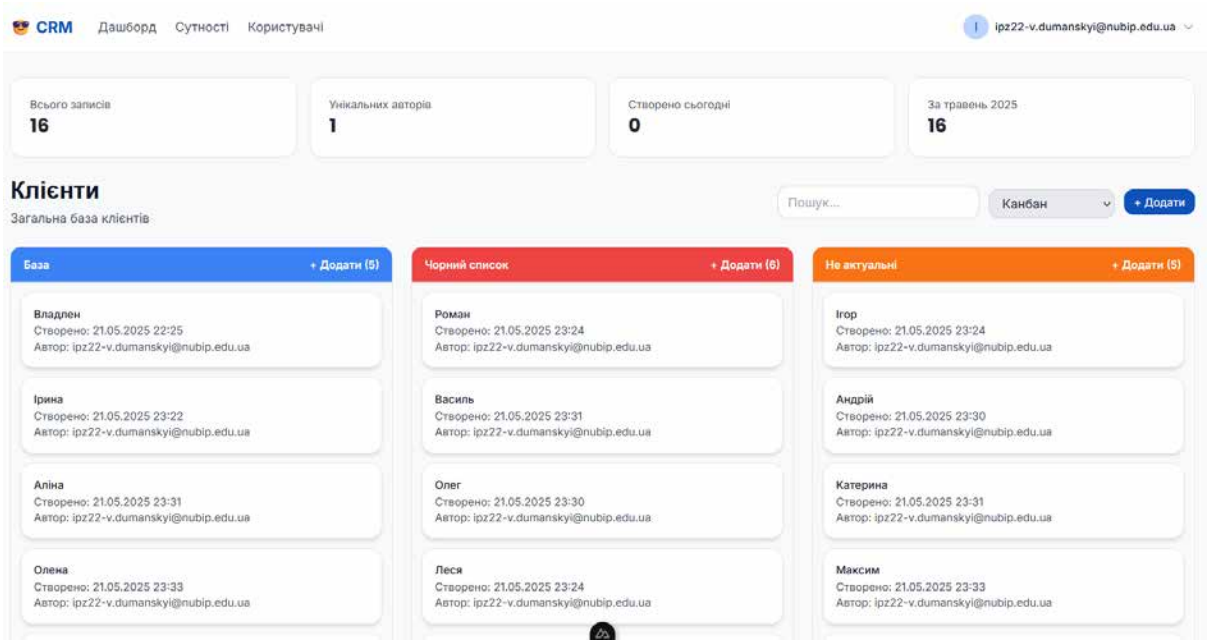

Рис. 31 Сторінка певної сутності у режимі “Канбан” та аналітика

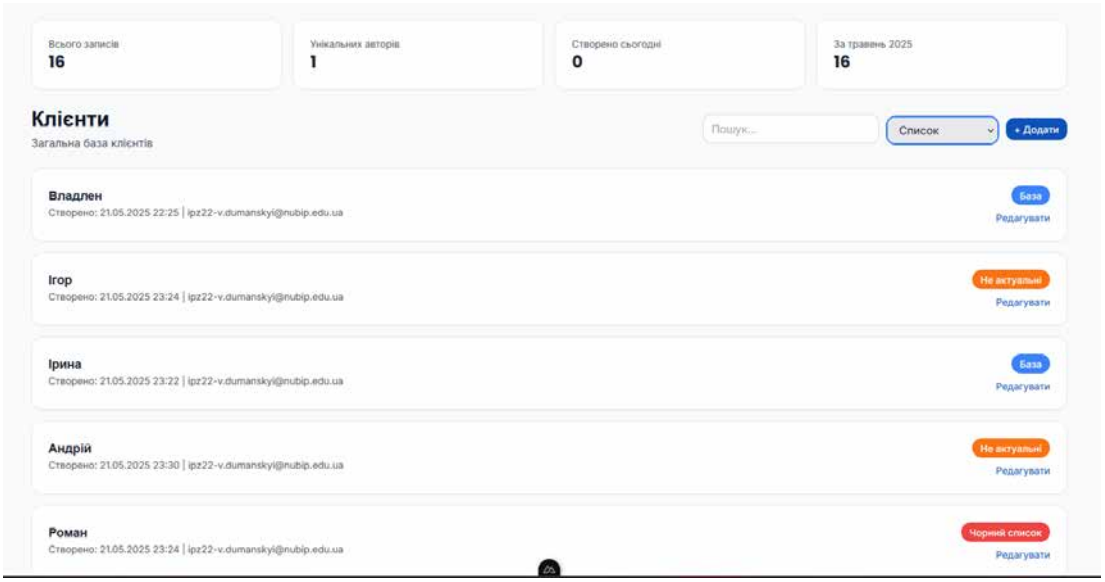

Рис. 32 Сторінка певної сутності у режимі “Список” та аналітика

Рис. 33 Демонстрація пошуку записів

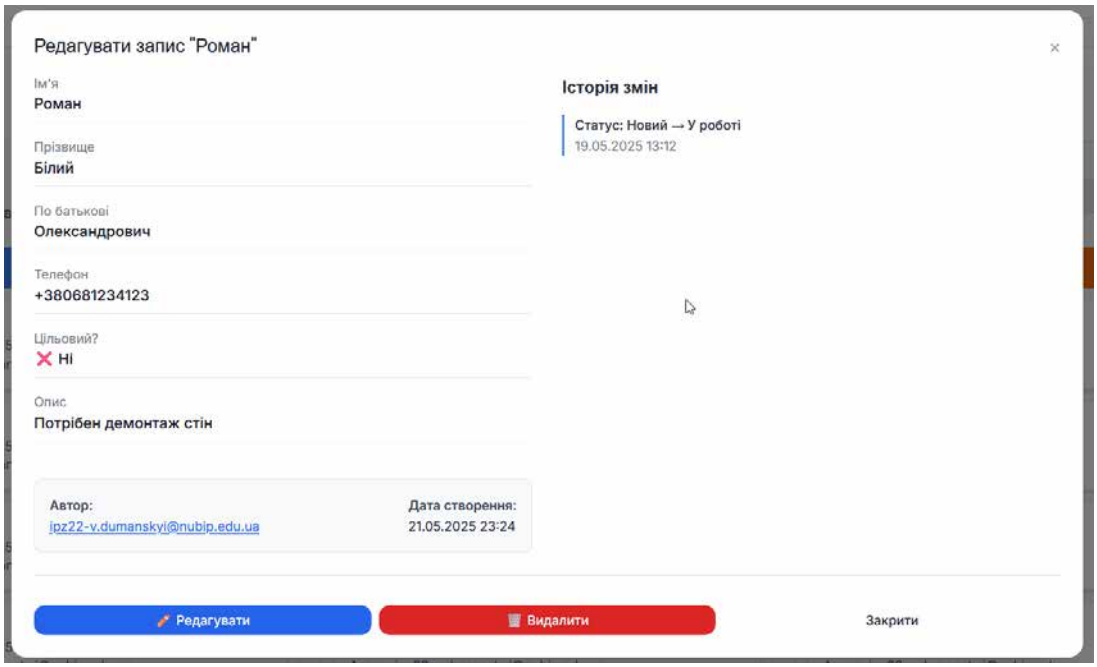

Рис. 34 Модальне вікно детальної інформації про запис

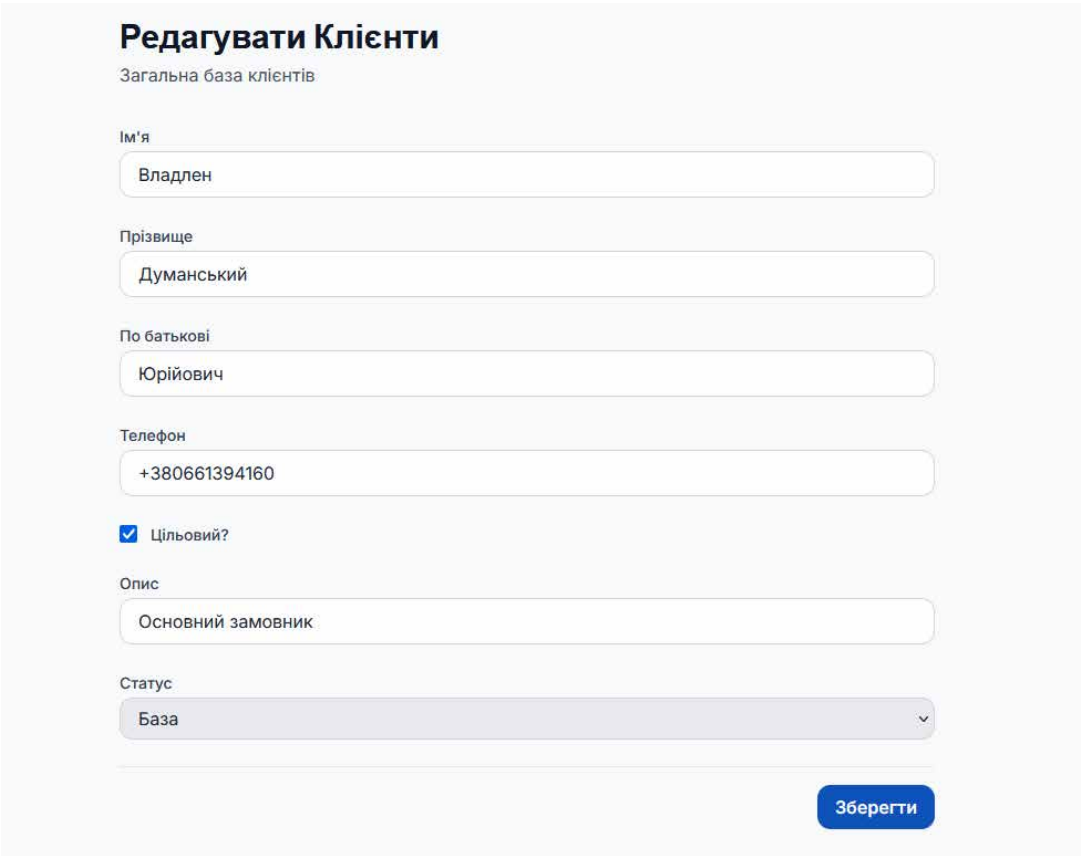

Рис. 35 Редагування запису сутності

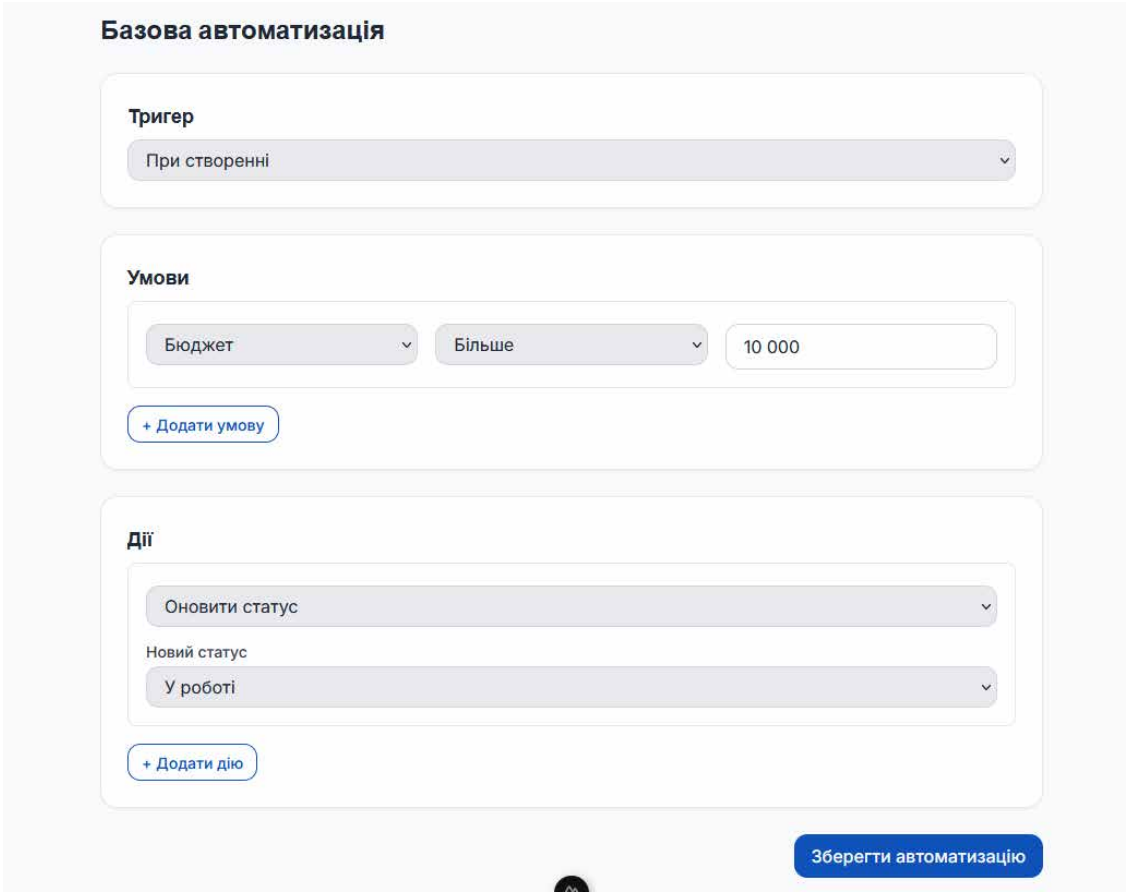

Рис. 36 Створення автоматизації

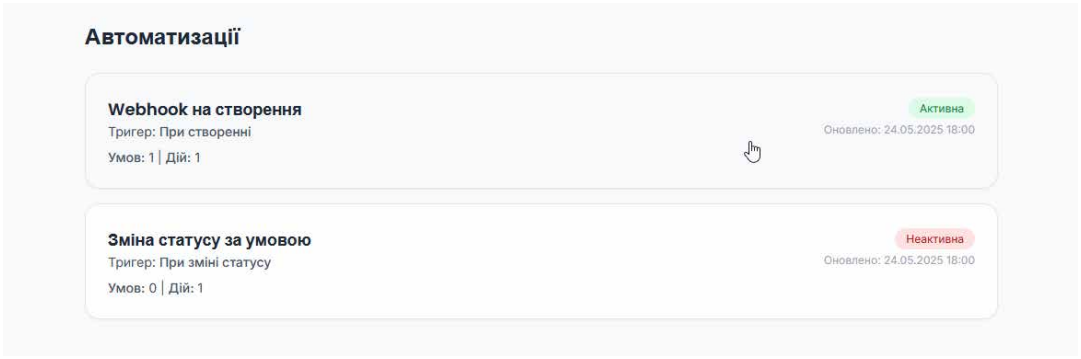

Рис. 37 Список автоматизацій

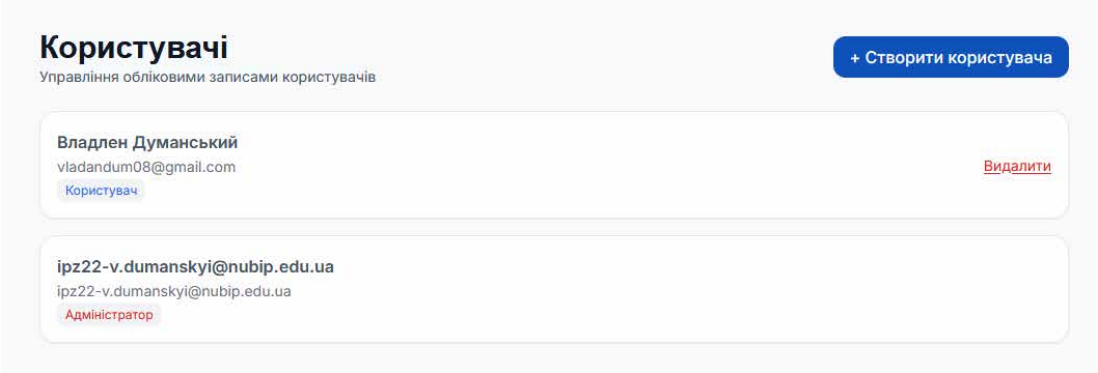

Рис. 38 Список користувачів