Вінницький національний технічний університет

Факультет інформаційних технологій та комп'ютерної інженерії

Кафедра програмного забезпечення

# Бакалаврська дипломна робота

на тему: Розробка веб сервісу для моніторингу дублікатів контактів

Виконала: студентка ___4___ курсу
групи ___2ПІ-18б_____
спеціальності
__121 – Інженерія програмного забезпечення_____
(шифр і назва напряму підготовки, спеціальності)
Найдюк В. І.
(прізвище та ініціали)
Керівник: к.т.н., доц. каф. ПЗ Войтко В. В.___
(прізвище та ініціали)
Рецензент:. к.т.н., доц. каф. КН Арсенюк І. Р.__
(прізвище та ініціали)

Допущено до захисту
Зав. кафедри Романюк О. Н.
«____» _____ 2022 р.

ВНТУ – 2022

Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра програмного забезпечення
Ступінь вищої освіти – бакалавр
Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри ПЗ
Романюк О. Н.
"25" березня 2022 року

**З А В Д А Н Н Я**
**НА БАКАЛАВРСЬКУ ДИПЛОМНУ РОБОТУ СТУДЕНТЦІ**

Найдюк Валерії Іванівні

1. Тема роботи – Розробка веб сервісу для моніторингу дублікатів контактів. Керівник роботи: Войтко Вікторія Володимирівна, к.т.н., доц. кафедри ПЗ, затверджені наказом вищого навчального закладу від "24" березня 2022 року №66

2. Строк подання студентом роботи 13 червня 2022 року

3. Вихідні дані до роботи: середовище розробки Visual Studio Code, мова розробки C#, JavaScript та Node.js, операційна система – Windows 10, Mac OS або Linux, система управління базою даних PostgreSQL.

4. Зміст розрахунково-пояснювальної записки: вступ; аналіз стану веб-сервісів для моніторингу дублікатів контактів; порівняльний аналіз аналогів; аналіз методів розв'язання задачі; постановка задач для веб-сервісу моніторингу дублікатів контактів; аналіз даних; розробка структури інтерфейсу веб-сервісу; розробка методу та моделі роботи веб-сервісу; розробка алгоритму пошуку дублікатів; варіантний аналіз і обґрунтування вибору засобів для реалізації програмного засобу; розробка бази даних; розробка api; розробка графічного інтерфейсу; розробка модуля інтеграції; тестування роботи веб-сервісу; висновки; список використаних джерел; додатки.

5. Перелік графічного матеріалу: назва роботи; мета, об’єкт і предмет дослідження; завдання бакалаврської дипломної роботи; аналоги; метод пошуку дублікатів контактів; алгоритм пошуку дублікатів контактів; модель роботи веб сервісу; модель роботи веб сервісу; тестування веб сервісу; реєстрація користувача; авторизація; початкова сторінка; створення аналізу; сторінка створеного аналізу; авторизація даних контактів; успішна авторизація; завантаження даних контактів; групування дублікатів; видалення дублікатів; видалення аналізу; вилогування користувача; висновки; апробації та публікації.

6. Консультанти розділів роботи

| Розділ | Прізвище, ініціали та посада консультанта | Підпис, дата | |
|---|---|---|---|
| | | завдання видав | завдання прийняв |
| 1-4 | Войтко В. В., к.т.н., доцент кафедри ПЗ | | |

7. Дата видачі завдання _25 березня 2022 року_____

КАЛЕНДАРНИЙ ПЛАН

| з/п | Назва етапів бакалаврської дипломної роботи | Строк виконання етапів роботи | Примітка |
|---|---|---|---|
| | Аналіз веб-сервісу для моніторингу дублікатів контактів | 26.03.2022 – 14.04.2022 | Вик. |
| | Розробка API та користувацького інтерфейсу | 14.04.2022 – 01.05.2022 | Вик. |
| | Розробка алгоритму визначення дублікатів контактів | 2.05.2022 – 30.06.2022 | Вик. |
| | Розробка модулів інтеграції із застосунками | 1.06.2022 – 10.06.2022 | Вик. |

Студентка _____ Найдюк В.І.
(підпис) (прізвище та ініціали)

Керівник бакалаврської дипломної роботи _____ Войтко В.В.
(підпис) (прізвище та ініціали)

# Анотація

Бакалаврська дипломна робота складається з 50 сторінок формату А4, на яких є 31 рисунок, 3 таблиці, список використаних джерел містить 15 найменувань.

У бакалаврській дипломній роботі проведено детальний аналіз процесу розробки веб-сервісу для моніторингу дублікатів контактів. Сформульовано мету досліджень – підвищення ефективності роботи із контактами шляхом розробки веб-сервісу для моніторингу, пошуку та видалення дублікатів контактів у широкого спектру застосунків для користувачів та бізнесу, що дозволить оптимізувати базу контактів.

Запропоновано метод пошуку дублікатів контактів шляхом розробки універсального розширеного веб-сервісу з підтримкою модулів інтеграції із зовнішніми базами даних контактів. Розроблено модель роботи веб-сервісу. Розроблено універсальний алгоритм пошуку та визначення дублікатів у базі даних контактів шляхом розгляду системи умов та перевірок на співпадіння даних.

Отримані в бакалаврській дипломній роботі результати можна використати для очистки контактів від дублікатів.

Для розробки веб сервісу було використано середовище розробки Visual Studio Code та технології .NET Core, Vue.js та NodeJS.

Ключові слова: веб-сервіс, контакти, дублікати.

## Abstract

The bechelor’s theses contains 50 pages of A4 format along with 31 pictures, 3 tables and 15 items of reference material.

During the bechelor’s theses analysis of contact duplicates monitoring web-service was carried out. The goal of research is formulated – improve user’s work with contacts by developing a duplicate monitoring web-service that supports a wide range of connector applications, this will optimize the contact base.

A method for search and detection of duplicate contacts is proposed by developing a universal and expandable service with modular support of third-part connector applications. A model of the web-service work is developed. An algorithm for determining contacts duplication through a system of checks and conditions is developed.

Results obtained in the bechelor’s thesis can be used to clean and deduplicate a contacts database.

The web service was developed using Visual Studio Code IDE and technologies .NET Core, Vue.js and NodeJS.


Keywords: web-service, contacts, duplicates.

ЗМІСТ

# ВСТУП

**Обґрунтування вибору теми дослідження**. У сучасному світі широкого попиту набувають хмарні технології – технології розподіленої обробки цифрових даних, за допомогою яких комп'ютерні ресурси надаються інтернет-користувачеві як онлайн-сервіс. Такі програми запускаються і видають результати роботи в вікні web-браузера на локальному ПК. При цьому всі необхідні для роботи програми та їх дані знаходяться на віддаленому інтернет-сервері і тимчасово кешуються на клієнтській стороні: на ПК та ін.

Перевага технології полягає в тому, що користувач має доступ до власних даних, але не повинен піклуватися про інфраструктуру, операційну систему та програмне забезпечення, з яким він працює. Слово «хмара» – це метафора, що уособлює складну інфраструктуру, що приховує за собою всі технічні деталі [1].

Більшість сучасних застосунків для споживачів та бізнесу створюються саме на хмарній основі і продаються як послуга (Software as a service, SaaS). Очевидними є переваги таких сервісів: легкість розгортання, гнучкість, доступність та глобальність.

Важливим при впровадженні хмарних додатків у роботу бізнесу чи організації є можливість їх інтеграції. Для уникнення повторного внесення тих самих даних часто використовують автоматизовані синхронізації даних. Це дозволяє уникнути непотрібної роботи та спростити роботу із сервісами, проте важливим є також те, аби забезпечити якість даних, уникнути випадкової втрати, або ж непотрібного створення дублікатів.

Тому актуальною є розробка веб-сервісу "Deduplicator" для вирішення питання, з яким стикаються користувачі хмарних веб-застосунків, що працюють з даними клієнтів. Сервіс дозволить знаходити, обробляти та видаляти дублікати контактів у популярних веб-застосунках, таких як iCloud, Microsoft Office 365, Google Contacts, численних CRM (customer relationship management) системах та ін. Окрім цього, сервіс буде універсальним та гнучким, адже до нього можна легко

додавати нові джерела даних, таким чином розширюючи аудиторію потенційних користувачів.

**Зв’язок роботи з науковими програмами, планами, темами.** Робота виконувалася згідно плану виконання наукових досліджень на кафедрі програмного забезпечення

**Мета та завдання дослідження.** Метою бакалаврської дипломної роботи є підвищення ефективності роботи із контактами шляхом розробки веб-сервісу для моніторингу, пошуку та видалення дублікатів контактів у широкого спектру застосунків для користувачів та бізнесу, що дозволить оптимізувати базу контактів.

Основними задачами роботи є:

– розробити базу даних відповідно до вимог веб-системи;
– розробити метод і алгоритм знаходження дублікатів контактів;
– розробити модель роботи веб-системи;
– розробити API;
– розробити модулі для з’єднання (інтеграції) зі сторонніми застосунками;
– розробити графічний інтерфейс для веб-середовища;
– налаштувати взаємодію між модулями з’єднання із застосунками, API та веб-інтерфейсом сервісу;
– провести тестування модулів та сервісу з використанням графічного інтерфейсу.

**Об’єкт дослідження** – процес розробки веб-сервісу для моніторингу дублікатів контактів.

**Предмет дослідження** – засоби реалізації веб-сервісу для моніторингу дублікатів контактів.

**Методи дослідження.** У процесі досліджень використовувались методи дослідження:

– методи теорії алгоритмів для побудови алгоритму пошуку дублікатів;
– методи побудови веб-систем для побудови стабільної веб-системи;

– методи передачі репрезентативного стану для передачі даних між клієнтом та сервером;
– методи зберігання секретів для захисту даних користувачів веб-системи;
– методи тестування для підтвердження працездатності веб-системи та її відповідності заданим вимогам.

**Наукова новизна отриманих результатів.**

- Подальшого розвитку отримав метод модульної інтеграції джерел даних контактів, який, на відміну від існуючих, орієнтований на підтримку великої кількості ітеграцій із зовнішніми користувацькими та бізнес застосунками, що дозволяє оптимізувати базу контактів і розширити аудиторію потенційних клієнтів сервісу.
- Подальшого розвитку отримала модель роботи веб-системи пошуку дублікатів, яка, на відміну від існуючих, орієнтована на модульну обробку джерел даних контактів з забезпеченням масштабної інтеграції із зовнішніми застосунками, що дозволяє оптимізувати базу контактів шляхом вилучення дублікатів.

**Практична цінність отриманих результатів.** Практична цінність полягає у кінцевій реалізації веб-серсвісу, що може застосовуватися клієнтами для моніторингу, пошуку та очищення бази даних контактів від дублікатів.

**Особистий внесок здобувача.** Усі наукові результати, викладені у бакалаврській дипломній роботі, отримані автором особисто. У науковій праці [2], опублікованій у співавторстві, автору належить розробка методу роботи системи пошуку дублікатів.

**Апробація матеріалів бакалаврської дипломної роботи.** Основні положення бакалаврської дипломної роботи доповідалися та обговорювалися на Всеукраїнській науково-практичній інтернет-конференції «Молодь в науці: дослідження, проблеми, перспективи – 2022».

**Публікації.** Основні результати дослідження опубліковані в науковій роботі [2] – в тезах доповіді на Всеукраїнській науково-практичній інтернет-конференції «Молодь в науці: дослідження, проблеми, перспективи - 2022».

**Аналіз.** У пояснювальній записці до бакалаврської дипломної роботи було розглянуто 4 розділи та було використане 15 літературних джерел.

# 1 ОБҐРУНТУВАННЯ ВИБОРУ МЕТОДУ РОЗРОБКИ ТА ПОСТАНОВКА ЗАДАЧ ДОСЛІДЖЕННЯ

## 1.1 Аналіз стану веб-сервісів для моніторингу дублікатів контактів

SaaS (Software as a Service) – це модель розгортання та реалізації програмного забезпечення, при якому постачальник (провайдер) розробляє додаток, ліцензує його, управляє ним, і надає споживачам (бізнес-клієнтам) доступ до ПЗ через Інтернет. SaaS – це сервіс, програмне забезпечення як послуга і на вимогу. Постачальник Software as a service надає клієнтові реалізацію бізнес-функцій, функціонала бізнес-додатків, вирішує питання інтеграції свого сервісу в IT-систему споживача, бере на себе всі функції з розвитку і підтримки рішень і забезпечення їх масштабування.

У процесі аналізу та розробки сервісу було розглянуто поточний стан веб-сервісів, які дозволяють користувачу аналізувати базу даних контактів та видаляти дублікати. Сучасний веб-сервіс для моніторингу дублікатів контактів повинен уміти виконувати такі основні функції:

- завантажити базу контактів із джерела;
- опрацювати контакти та знайти дублікати;
- можливість злити дані дублікатів у один;
- можливість видалити дублікати контактів.

Сервіс повинен максимально ефективно знаходити дублікати, бути гнучким у налаштуванні, коректно зливати дані та не допускати помилок або втрати інформації про контакти користувача.

Важливим аспектом такого веб-сервісу є доступність та універсальність. Сервіс повинен працювати у хмарі та бути доступним у будь-який час та із будь-якого пристрою через веб-переглядач.

Окрім цього, сервіс повинен надавати широкий спектр інтеграцій користувацьких та бізнес застосунків. Таким чином, користувач матиме впевненість, що сервіс зможе завантажити дані контактів зі потрібного джерела для подальшого їх опрацювання. Список доступних інтеграцій повинен постійно оновлятись, тим самим розширюючи аудиторію потенційних клієнтів.

Додатковою корисною функцією є можливість автоматизації процесу очищення бази контактів від дублікатів. Наприклад, користувач має мати можливість автоматичного запуску сервісу кожного дня.

### 1.2 Порівняльний аналіз аналогів

Серед відомих сервісів пошуку дублікатів контактів можемо навести наступні основні:

– Deduply;

– Contacts+;

– RingLead;

Deduply (рис. 1.1) – хмарний веб-сервіс, призначений для знаходження дублікатів контактів. Дозволяє видаляти дублікати по одному або усі відразу (bulk delete). Недоліком сервісу є підтримка – працює лише із трьома відомим CRM системами (HubSpot, Salesforce та Pipedrive), а також із Microsoft Excel. Перевагою сервісу є можливість автоматизації очищення контактів [3].

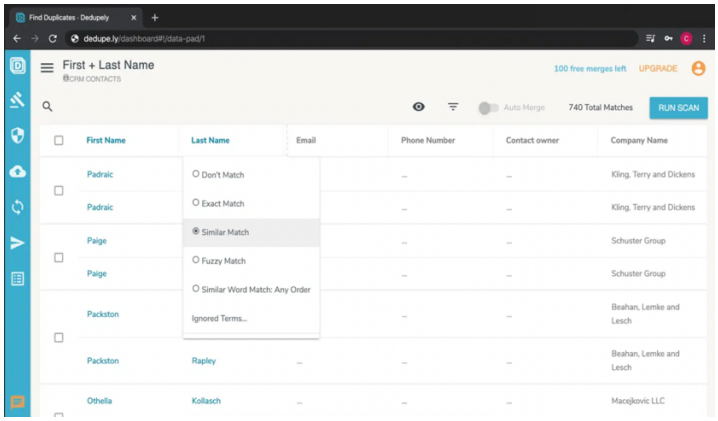


Рисунок 1.1 – Сервіс Deduply в дії

Contacts+ (рис. 1.2) – настільний та мобільний застосунок для роботи з контактами, який дозволяє очищати базу від дублікатів. Застосунок є доволі універсальним і пропонує інтеграцію із великою кількістю користувацьких та бізнес рішень, серед яких Google Contacts, Microsoft Office 365, iCloud Contacts, Salesforce, HubSpot, MailChimp та інші. Недоліком застосунку є відсутність автоматизації та хмарної версії [4].

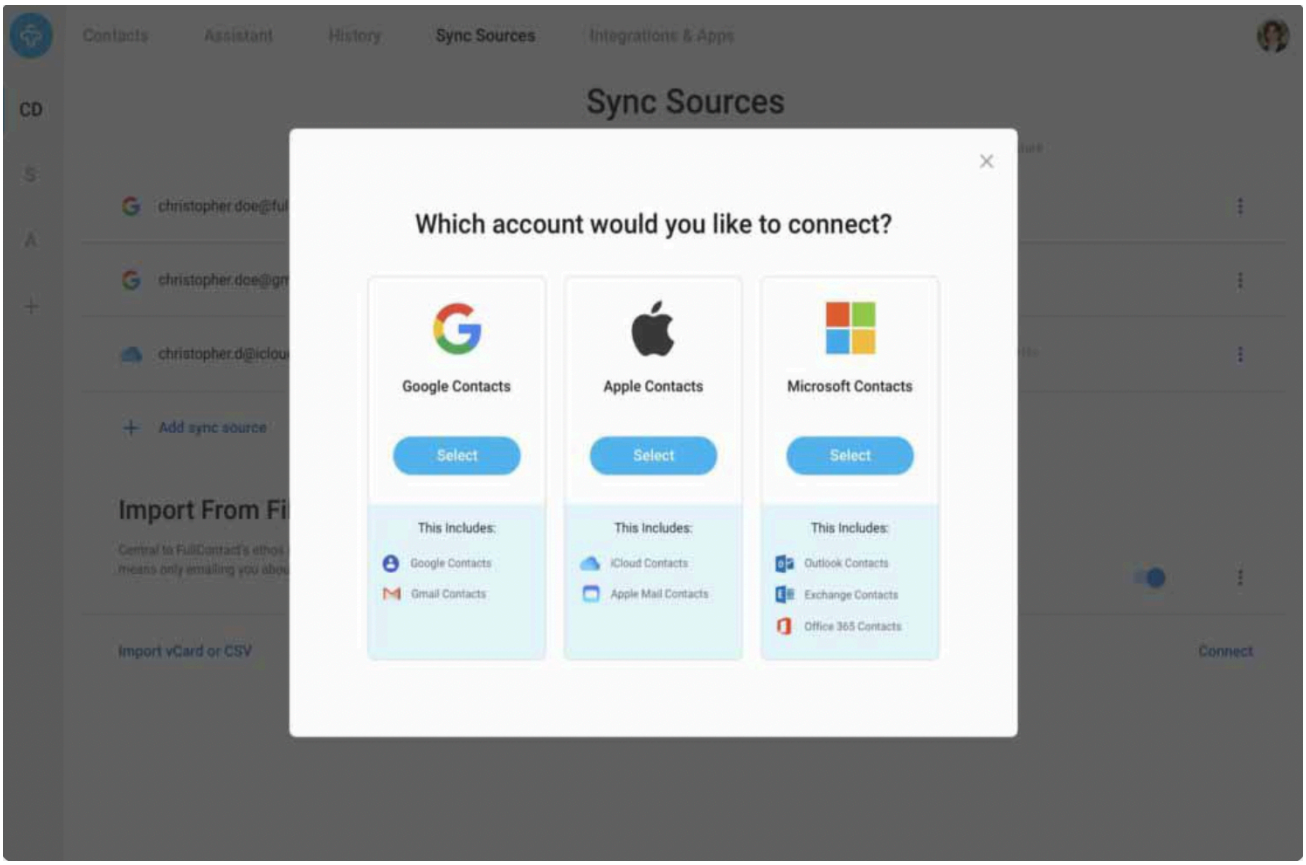


Рисунок 1.2 – Інтеграція Contacts+ із стороннім сервісом

RingLead (рис. 1.3) – ще один хмарний веб-сервіс для роботи з контактами, який пропонує широкий спектр функцій, серед яких збагачення даних контактів, сегментація, синхронізація та очищення від дублікатів. Перевагою додатку є його багата функціональність, серед недоліків: відсутність автоматизації, складність налаштування та використання, висока ціна [5].

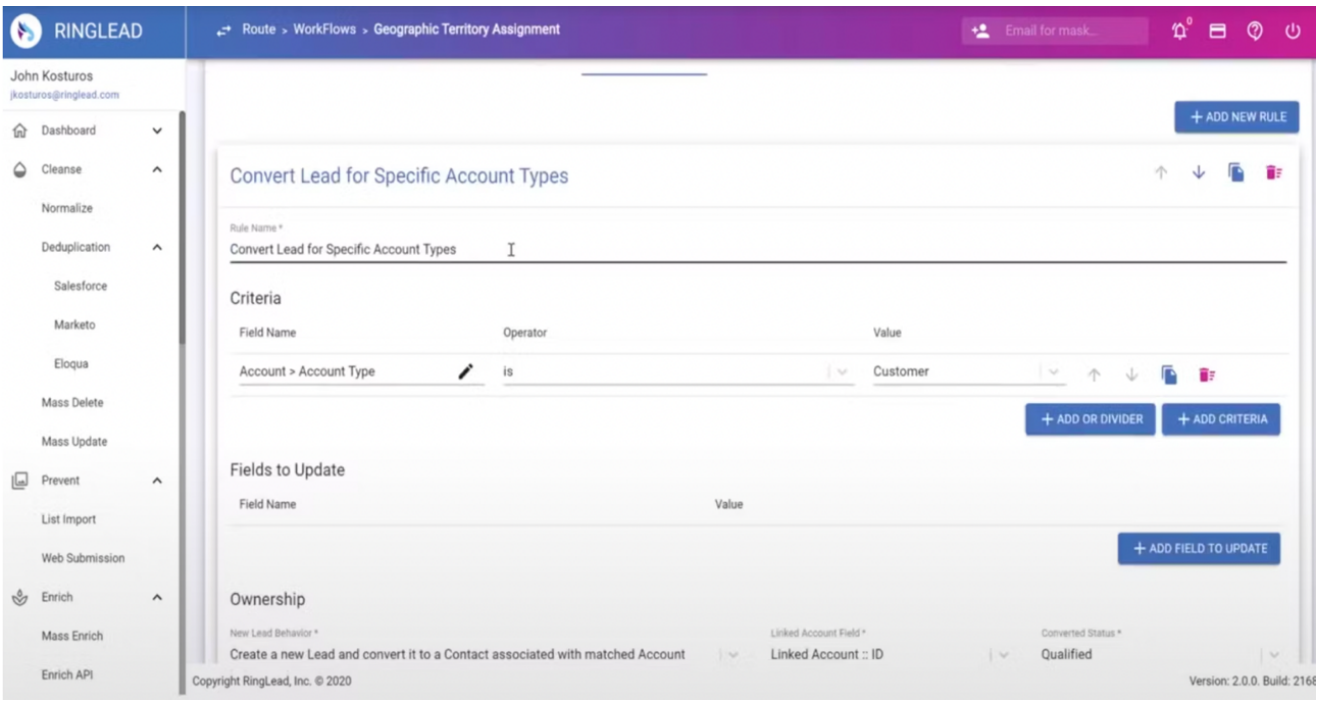


Рисунок 1.3 – Сервіс RingLead в ділі

Результати порівняння аналогів зведено в табл. 1.1.

Таблиця 1.1 – Порівняльні характеристики сервісів

| Критерії | Deduply | Contacts+ | RingLead | Власне рішення |
|---|---|---|---|---|
| Хмарне рішення | + | - | + | + |
| Можливість злиття дублікатів | + | + | + | + |
| Автоматизація | + | - | - | + |
| Універсальність | - | + | - | + |
| Збагачення даних контактів | - | - | + | - |
| Загальна оцінка | 60% | 40% | 60% | 80% |

Відповідно до таблиці порівняльних характеристик (табл. 1.1) розробка власного сервісу для пошуку та видалення дублікатів контактів є доцільною. Отриманий продукт зможе покрити недоліки існуючих рішень та забезпечити новий функціонал для спрощення роботи.

### 1.3 Аналіз методів розв'язання задачі

Деякі застосунки пропонують вбудовану функціональність для виявлення або ж видалення дублікатів контактів. Наприклад, сервіс Google Contacts автоматично визначає дублікати контактів та дозволяє злити їх в один. Однак його алгоритм визначення дублікатів не є ідеальним, а механізм злиття контактів не є настільки гнучким, наскільки хотілося б користувачам у певних випадках.

Проте далеко не кожен сервіс має вбудований механізм пошуку дублікатів. Навіть такий популярний сервіс як Apple iCloud Contacts не пропонує такої функції.

Можливим вирішенням цієї проблеми є створення окремого сервісу для кожного джерела даних контактів, який буде виконувати потрібну функцію. Для деяких застосунків існують сервіси, розроблені третьою стороною, які дозволяють очищати контакти, однак недоліком таких рішень є частий брак потрібних функцій, погана підтримка та відсутність централізованого інтерфейсу.

Найкращим ж рішенням є створення універсального веб-сервісу, який дозволяє завантажувати дані контактів з різних джерел даних та видаляти їх дублікати. Перевагами такого сервісу буде зручність уніфікованого інтерфейсу, а також легкість розширення та додавання нових джерел даних. Кожен користувацький або бізнес застосунок може бути з легкістю інтегрований із сервісом після розробки модуля, який, використовуючи API, зможе виконувати CRUD-операції (create, read, update, delete) із контактами.

Отже, було прийнято рішення розробити уніфікований веб-сервіс, який виконуватиме функцію моніторингу та видалення дублікатів контактів для широкого спектру користувацьких та бізнес застосунків.

### 1.4 Постановка задач для веб-сервісу моніторингу дублікатів контактів

Проаналізувавши переваги та недоліки існуючих сервісів моніторингу дублікатів контактів, було сформульовано задачі бакалаврської дипломної роботи:

- розробити базу даних відповідно до вимог веб-системи;
- розробити метод і алгоритм знаходження дублікатів контактів;
- розробити модель роботи веб-системи;
- розробити API;
- розробити модулі для з’єднання (інтеграції) зі сторонніми застосунками;
- розробити графічний інтерфейс для веб-середовища;
- налаштувати взаємодію між модулями з’єднання із застосунками, API та веб-інтерфейсом сервісу;
- провести тестування модулів та сервісу з використанням графічного інтерфейсу.

### 1.5 Висновки

У першому розділі було розглянуто стан веб-сервісів для моніторингу, пошуку, видалення та злиття дублікатів контактів. Було проведено порівняння існуючих веб-сервісів, таких як: Deduply, Contacts+, та RingLead. Проаналізувавши переваги та недоліки відомих веб-систем, було виявлено, що жоден з них не є достатньо універсальним та не забезпечує користувачів усіма функціями, необхідними для очистки бази контактів від дублікатів. Таким чином було доведено доцільність розробки власного програмного рішення. На основі отриманої інформації було сформовано перелік задач, які необхідно виконати для розробки власного веб-сервісу.

# 2 РОЗРОБКА МЕТОДУ ПОШУКУ ДУБЛІКАТІВ КОНТАКТІВ, МОДЕЛІ ТА АЛГОРИТМІВ РОБОТИ ВЕБ-СИСТЕМИ

## 2.1 Аналіз даних

Ефективна організація і швидкий обмін даними є ключовими аспектами будь-якого якісного веб-сервісу. Від того, як зберігаються і обробляються дані, залежить не тільки швидкодія додатку, а і його ефективність, гнучкість та стабільність роботи. Для збереження особистих даних користувача були використано реляційну базу даних, а для передачі даних між сервісом та користувацьким інтерфейсом було імплементовано REST API, яка виконує усі основні функції веб-сервісу та передає дані до користувача через безпечний протокол HTTPS.

Для ефективного аналізу бази даних контактів користувача потрібно завантажити усі основні поля, які важливі для знаходження дублікатів. Серед таких полів: ім’я, прізвище, список електронних адрес користувача та список номерів телефонів користувача. Цих даних контактів достатньо для того, аби змогти швидко та ефективно ідентифікувати дублікати у базі даних.

Необхідно налагодити канал зв’язку із сторонніми застосунками користувача, звідки будуть завантажуватись дані контактів. Завантаження цих даних відбувається за допомогою модулів інтеграцій. Завдання кожного модуля – отримати доступ до даних контактів та завантажити їх за допомогою публічного API цього застосунку. Окрім цього, варто зауважити, що дані контактів мають різний формат для кожного інтегрованого застосунку, отже окремим завданням модуля є зведення цих даних в уніфікований формат, з яким буде працювати сервіс моніторингу дублікатів.

Зауважимо, що швидкодія процесу завантаження контактів і пошуку дублікатів залежить від швидкодії API відповідного застосунку, а отже може займати більше часу ніж звичайний запит сервісу.

### 2.2 Розробка структури інтерфейсу веб-сервісу

Інтерфейс веб-сервісу складається із декількох сторінок. Головна сторінка містить основну інформацію про сервіс, а також список збережених аналізів дублікатів, які користувач виконував у минулому (рис 2.1).

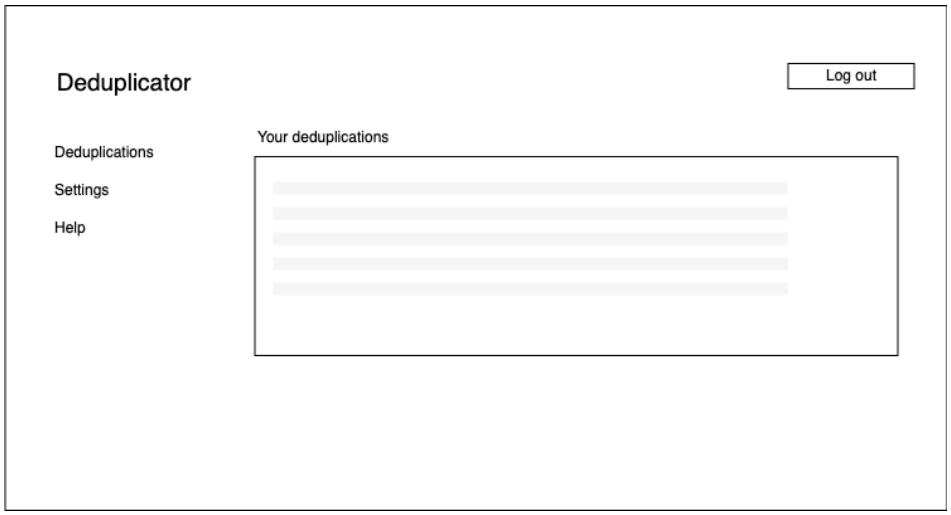


Рисунок 2.1 – Головна сторінка

Окрім цього, з головної сторінки можна перейти на сторінку створення нового аналізу контактів на дублікати. На цій сторінці користувачу необхідно вибрати користувацький або бізнес застосунок, для якого слід провести пошук дублікатів контактів (рис 2.2).

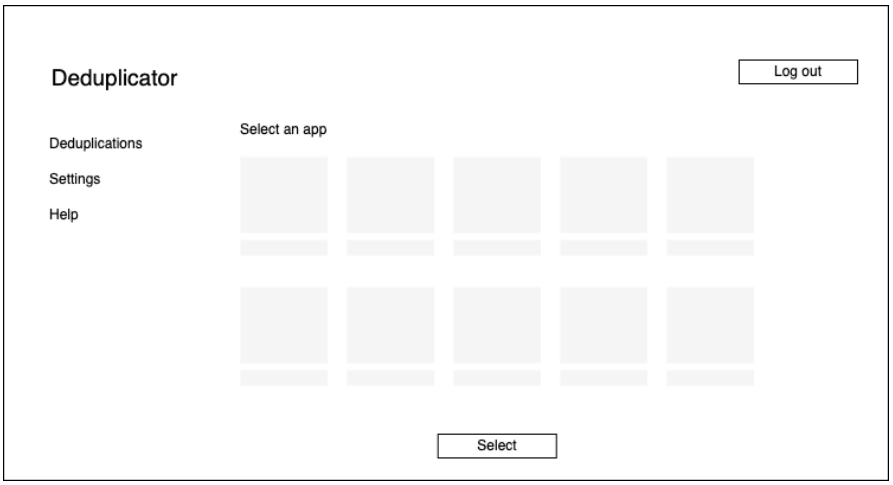


Рисунок 2.2 – Сторінка створення нового аналізу дублікатів

Найважливішою сторінкою є сторінка аналізу контактів на дублікати (рис. 2.3). Вона складається із двох основних частин. Спочатку користувач повинен надати доступ до даних контактів шляхом приєднання до зовнішнього сервісу. Після цього будуть завантажені дані контактів користувача та проведено аналіз контактів на дублікати. Користувачу буде запропоновано, які контакти слід видалити, і які поля злити. Після цього користувач може підтвердити та розпочати процес очищення даних від дублікатів.

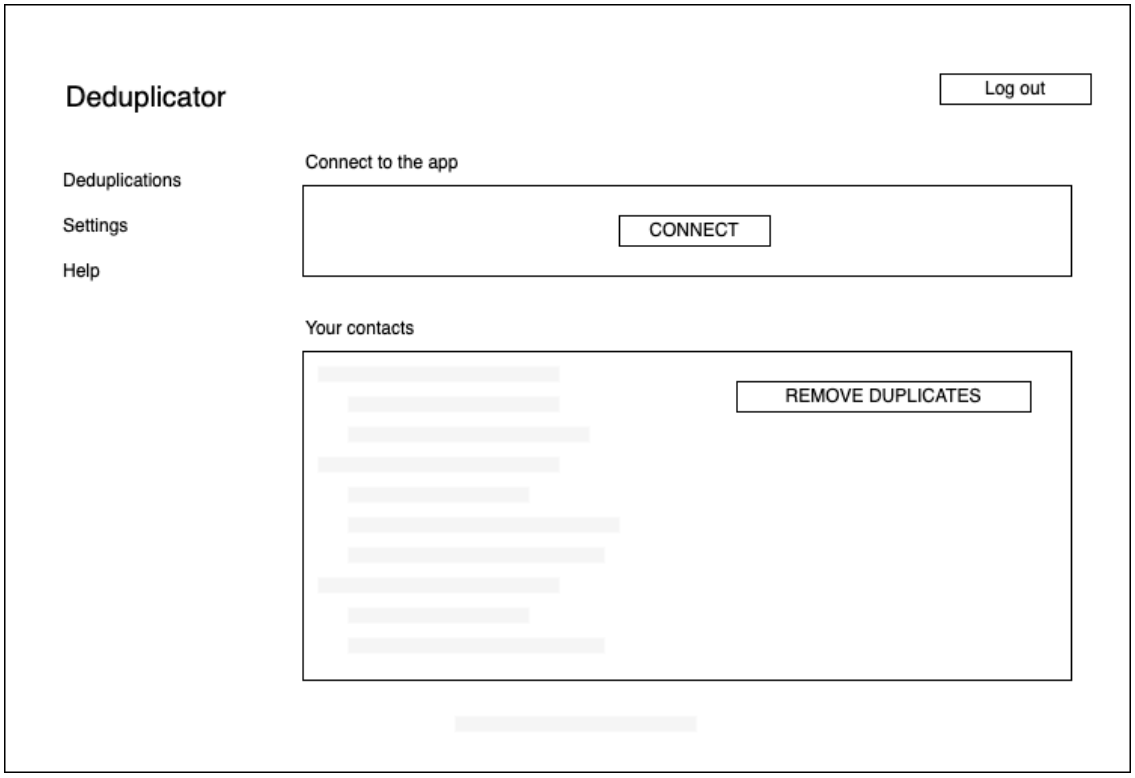


Рисунок 2.3 – Сторінка аналізу даних контактів

### 2.3 Розробка моделі роботи веб-сервісу

Для кращої візуалізації та розуміння роботи веб-сервісу було створено модель робити системи за допомогою діаграми діяльності (рис 2.4). Діаграма діяльності – це графічне зображення виконаного набору дій процедурної системи, яка описує дії та паралельні процеси системи на детальному рівні [6].

Згідно цієї діаграми, першим кроком користувач повинен залогуватись, або ж зареєструватись та увійти, якщо він ще не зареєстрований. Після цього користувачу слід створити новий аналіз контактів та вибрати, із якого зовнішнього застосунку слід завантажити дані контактів для аналізу. Після авторизації цього застосунку проводиться завантаження даних та пошук дублікатів. Користувачеві візуально відображаються усі контакти та знайдені дублікати, а також пропонується спосіб їх злиття. Користувач може підтвердити та розпочати процес видалення та злиття дублікатів, або ж не погодитись із запропонованим аналізом та внести свої зміни, після чого розпочати злиття. Разом із цим, сервіс зберігає виконану роботу для того, аби користувач міг в майбутньому повернутись та продовжити або повторити процес.

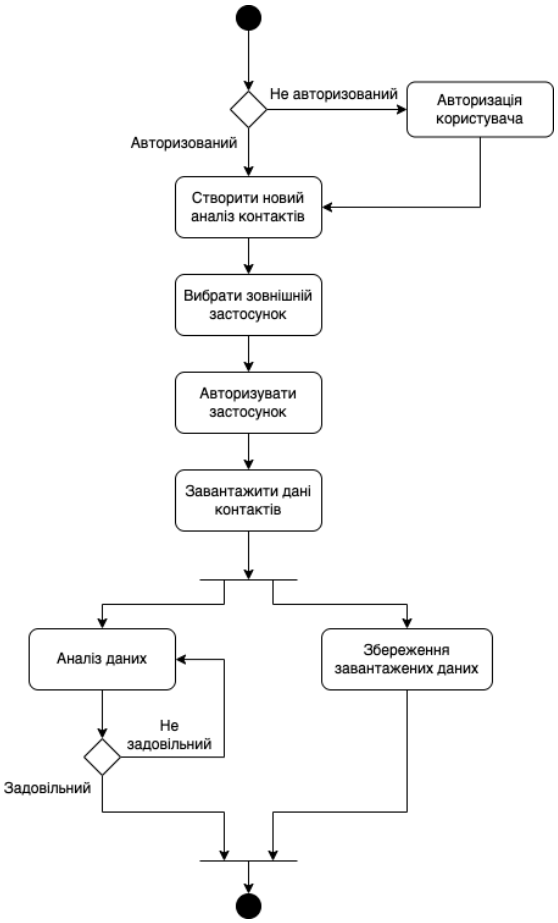


Рисунок 2.4 – Модель роботи веб-системи пошуку дублікатів

### 2.4 Розробка методу та алгоритму пошуку дублікатів

Для роботи веб сервісу було розроблено метод пошуку дублікатів – процес, згідно з якого відбувається завантаження, обробка та пошук дублікатів контактів у базі даних зовнішнього користувацького чи бізнес застосунку. Правильний вибір цього методу є ключовим аспектом ефективної роботи веб сервісу та максимізації його користі користувачу.

Після аналізу усіх необхідних кроків було сформовано наступний метод пошуку дублікатів контактів:

1. Відповідно до вибраного зовнішнього застосунку відбувається завантаження модуля доступу до даних.
2. Відбувається авторизація користувача цього застосунку.
3. У разі успішної авторизації завантажуються дані контактів користувача.
4. Дані контактів приводяться у спільний (унфікований формат), що дозволяє їх подальшу обробку.
5. Використовуючи алгоритм пошуку дублікатів контакти розбиваються на групи, в яких зібрано дублікати того ж самого контакту.
6. Для кожної групи контактів проводиться злиття даних кожного із основних полів (імена, номери телефонів та адреси електронних скриньок).
7. Для кожного групи вибирається один основний контакт.
8. Вибраний контакт оновлюється злитими даними.
9. Усі інші контатки кожної групи видаляються.

Такий метод дозволяє простим та зрозумілим для користувача чином відобразити усі знайдені дублікати, а також очистити базу даних контактів від них. Варто також зауважити, що в користувача є можливість змінити вибрані групи та контролювати, яким чином зливаються дані контактів.

Також важливою частиною веб-сервісу для моніторингу дублікатів є алгоритм, згідно з яким виконується пошук дублікатів, а саме яким чином два контакти перевіряються на однаковість. Саме завдяки цьому алгоритму контакти будуть групуватись і їхні дані зливатись в один. Для якомога більшої кількості пар контактів, які справді є дублікатами, алгоритм повинен повертати позитивний результат, і в цей самий час мінімізувати хибні позитиви, де контакти, які не є дублікатами, були визначені як дублікати.

Після детального аналізу проблеми було вирішено розробити систему умов, згідно з яких пара контактів буде перевірятись на однаковість. Це система умов була реалізована у вигляді алгоритм пошуку дублікатів контактів, зображеного блок-схемою на рисунку 2.5.

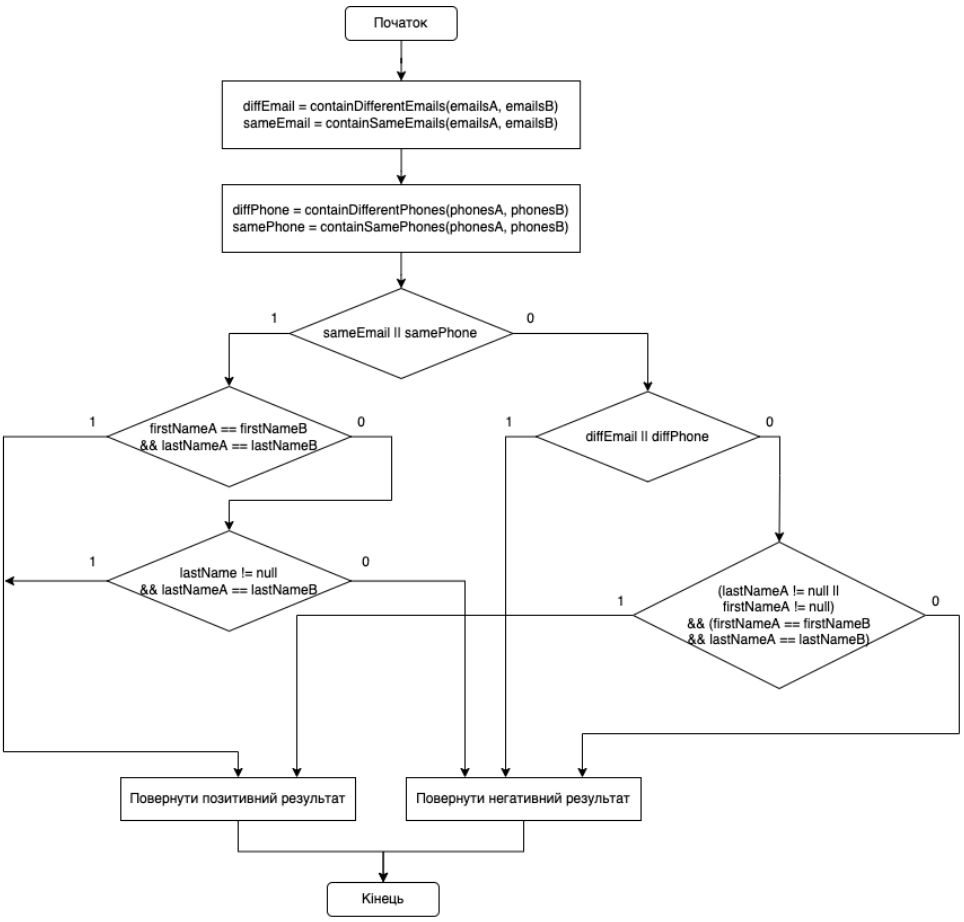


Рисунок 2.5 – Блок-схема алгоритм пошуку дублікатів контактів

## 2.5 Висновки

У другому розділі було розглянуто та проаналізовано дані, з якими працює веб-сервіс, а також визначено процеси передачі даних між сервісом, користувацьким інтерфейсом та зовнішніми інтегрованими застосунками. Було розроблено макет графічного інтерфейсу основних сторінок сервісу, розроблено модель робити веб-системи пошуку дублікатів, а також розроблено метод і алгоритм знаходження дублікатів контактів.

# 3 РОЗРОБКА ВЕБ-СЕРВІСУ МОНІТОРИНГУ ДУБЛІКАТІВ КОНТАКТІВ

## 3.1 Варіантний аналіз і обґрунтування вибору засобів для реалізації програмного засобу

Перед початком реалізації будь-якого інформаційного продукту потрібно правильно обрати технології розробки, адже від їх вибору часто залежить швидкість запуску продукту, зручність та гнучкість розробки та налаштування.

Для розробки серверної частини веб-сервісу було обрано C# – мова загального призначення, яка може використовуватися для розробки настільних додатків, ігор, веб-додатків та мобільних додатків [7]. C# передбачає використання платформ .NET Framework або .NET Core. Головною відмінністю даних платформ є кросплатформеність. Яскравим прикладом цього є особливість .NET Framework, яка передбачає використання системних бібліотек операційної системи Windows, тоді як .NET Core уникає необхідності їхнього використання та дає можливість розгортати додатки на різних операційних системах [8].

Для збереження та доступу до серверних даних було використано систему керування базами даних Postgres, яка дозволяє розгортати сервер на будь-якій операційній системі та забезпечує усі функції сучасної реляційної бази даних. Для доступу до даних бази було використано .NET Framework Core, яка дозволяє зручно та легко завантажувати та змінювати дані в коді C# та має реалізований модуль доступу до Postgres баз.

Графічний інтерфейс було реалізовано за допомогою Vue.js – сучасного фреймворку розробки функціональних HTML сторінок. Основною перевагою є можливість створення ізольованих модулів (компонент), які виконують окрему частину функціональності сайту та можуть бути перевикористаними.

Для реалізації модулів доступу до даних зовнішніх сервісів було використано модулі Node.JS. Основною перевагою є можливість використання численної бібліотеки готових рішень NPM, а також легкість роботи із JSON об’єктами, що дозволяє легко викликати методи публічних API та обробляти їх дані.

### 3.2 Розробка бази даних

Одним із найважливіших етапів розробки будь-якого веб-сервісу є проектування та реалізація реляційної бази даних. Сутності бази даних повинні максимально відповідати сутностям домену, бути нормалізованими та не містити ніяких зайвих зв’язків чи залежностей. Це забезпечить простоту робити із даними при реалізації сервісу, та дозволить легко та швидко додавати нові функції.

Для створення бази даних сервісу було використано .NET Entity Framework Core та модуль доступу до Postgres. Ці інструменти розробки та доступу до баз даних пропонують широкий набір гнучких функції, які дозволяють легко та швидко розробляти основну логіку програмного сервісу. Усі сутності були додані до бази даних і створені відповідні класи в C# коді. Рисунок 3.1 відображає діаграму сутностей.

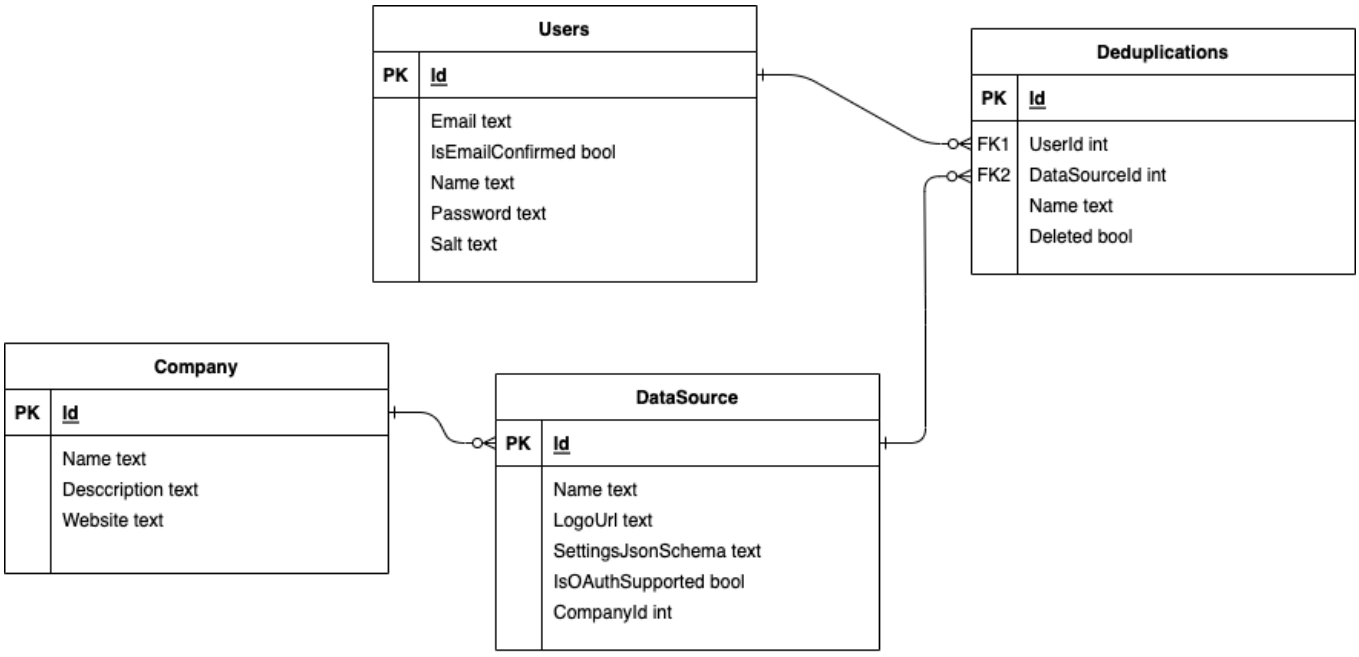


Рисунок 3.1 – ER-діаграма бази даних

На рисунку 3.2 наведено код оголошення бази даних інструментами Entity Framework Core мовою C#.

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata;
using Microsoft.EntityFrameworkCore.Metadata.Internal;
using Deduplicator.Core.Models;

namespace Deduplicator.Core
{
    public class DeduplicatorContext : DbContext
    {
        public DbSet<Deduplication> Deduplications { get; set; }

        public DbSet<DataSource> DataSources { get; set; }

        public DbSet<User> Users { get; set; }

        public DbSet<Company> Companies { get; set; }

        protected override void OnConfiguring(DbContextOptionsBuilder builder)
        {
            builder.UseNpgsql("User ID=postgres;Password=;Host=;Port=;Database=postgres;");
        }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            foreach (IMutableEntityType entityType in modelBuilder.Model.GetEntityTypes())
            {
                entityType.Relational().TableName = entityType.DisplayName();
            }
        }
    }
}
```

Рисунок 3.2 – Лістинг коду оголошення бази даних

### 3.3 Розробка API

API (Application Programming Interface, Прикладний програмний інтерфейс) – це набір визначень підпрограм, протоколів взаємодії та засобів для створення програмного забезпечення. Спрощено — це набір чітко визначених методів для взаємодії різних компонентів. API надає розробнику засоби для швидкої розробки програмного забезпечення.

API може бути для веббазованих систем, операційних систем, баз даних, апаратного забезпечення, програмних бібліотек. API є абстрактним поняттям – програмне забезпечення, що пропонує деякий API, часто називають реалізацією (англ. implementation) даного API. У багатьох випадках API є частиною набору розробки програмного забезпечення, водночас, набір розробки може включати як API, так і інші інструменти/апаратне забезпечення, отже ці два терміни не є взаємозамінювані [9].

Одним із підходів створення API для веб-сервісів є REST API – стиль архітектури API для розподілених систем. REST визначає, як дані надаються клієнту в зручному для нього форматі. Обмін даними відбувається у форматі JSON або XML (хоча сьогодні більш популярний формат JSON) [10]. Обмін даних в такому випадку найчастіше виконується за допомогою безпечного протоколу HTTPS.

Для реалізація веб-сервісу «Deduplicator» було використано інструменти .NET Core Web API – фреймворк розробки серверних веб аплікацій від компанії Microsoft, який працює у середовищі .NET. Вони дозволяють швидко та зручно створювати REST API за допомогою мови програмування C# [11].

За допомогою цього фреймворку було створено контролери, які відповідають сутностям бази даних і виконують усі команди, необхідні для обробки запитів графічного інтерфейсу та виконання необхідних функцій веб-сервісу. Для передачі даних між клієнтом та сервером було створено DTO (Data transfer object) – об’єкти C#, які відображають дані, які потрібно передати, і які автоматично конвертуються в та з формату JSON.

Всього реалізовано чотири контролери: DeduplicationController (для роботи із сутністю Deduplication), UserController (для роботи із користувачами), DataSourceController (для роботи із джерелами даних) та CompanyController (для роботи із компаніями, які розробляють джерело даних). На рисунку 3.3 нижче наведено код реалізація одного із контролерів веб-сервісу мовою програмування C#.

```
using System.Threading.Tasks;
using MediatR;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Deduplicator.Api.Infrastructure;
using Deduplicator.Core.Commands.Syncs;

namespace Deduplicator.Api.Controllers
{
    [Authorize]
    [Route("deduplications")]
    public class DeduplicationController : Controller
    {
        private readonly IMediator _mediator;

        public DeduplicationController(IMediator mediator)
        {
            _mediator = mediator;
        }

        [HttpGet]
        [Route("")]
        public async Task<IActionResult> GetDeduplications()
        {
            var request = new UserDeduplicationsRequest
            {
                Login = Request.HttpContext.User.GetLoginContext()
            };

            if (request.Login.UserId == null)
            {
                return Unauthorized();
            }

            return Ok(await _mediator.Send(request));
        }
    }
}
```

Рисунок 3.3 – Лістинг коду контролера

### 3.4 Розробка графічного інтерфейсу

Для розробки гнучкого графічного інтерфейсу веб-сервісу було використано Vue.js – JavaScript-фреймворк що використовує шаблон MVVM (Model-view-viewmodel) для створення інтерфейсів користувача на основі моделей даних, через реактивне зв'язування даних. Vue використовує синтаксис шаблонів на основі HTML, що дозволяє декларативно зв'язувати рендеринг DOM з основними екземплярами даних в Vue.

Одна із найвиразніших особливостей Vue – це ненав'язлива реактивна система. Моделі це просто плоскі JavaScript об'єкти. Це робить керування станами дуже простим та інтуїтивним. Vue надає оптимізований ре-рендеринг з коробки без потреби робити що-небудь додатково. Кожен компонент слідкує за своїми реактивними залежностями під час рендерингу, тому система знає точно коли має відбуватись ре-рендеринг і які компоненти потрібно ре-рендерити.

Варто зауважте, що при роботі із Vue.js не потрібно взаємодіяти напряму з HTML. Додаток на Vue прикріплює себе до DOM елементу (згідно із його класном або ідентифікатором) для повного контролю над ним. HTML є єдиною точкою входу, але все інше відбувається в межах новоствореного екземпляру Vue, що дозволяє чітко розмежувати DOM від загальної логіки інтерфейсу. Це дозволяє розробникам приділити увагу основніх механаці взаємодій елементів на бізнес рівні та не думати про те, як відобразити це відповідним чином на HTML сторінці у веб переглядачі [12].

Vue сам по собі не включає роутингу, та є vue-router пакет, який вирішує це питання. Він підтримує зв'язування вкладенних шляхів з вкладеними компонентами і пропонує деталізований контроль над переходами. Vue дозволяє створення додатків за допомогою компонентів. Якщо додати vue-router до цього, все що потрібно зробити це зв'язати ваші компоненти з роутами і дозвольте vue-router вирішувати де їх рендерити [13].

Використовуючи Vue.js було створено компоненти, які відповідають за кожну сторінку та окрему частину веб-сервісу. Окрім цього, було використано Vue.js Router для реалізація переходів між сторінками. Це необхідно, оскільки Vue.js створює SPA – Single page application, застосунок, який формально складається лише із одної сторінки. Router дозволяє емулювати багатосторінкову поведінку веб-сервісу та дозволяє зручно управляти сторінками. На рисунку 3.4 наведено приклад реалізації шаблону однієї із Vue-компонент, яка відображає список аналізів контактів, створених користувачем.

```
<template>
  <div>
    <el-row>
      <el-button-group>
        <el-button size="medium" icon="el-icon-plus" @click="onAdd">New deduplication</el-button>
      </el-button-group>
    </el-row>
    <el-row v-loading="isLoading">
      <el-table
        empty-text="No data"
        :data="deduplications">
        <el-table-column
          prop="id"
          label="ID"
          width="60">
        </el-table-column>
        <el-table-column
          width="56">
          <template slot-scope="scope">
            <img :src="scope.row.dataSource.logoUrl" :title="scope.row.dataSource.name"/>
          </template>
        </el-table-column>
        <el-table-column
          prop="name"
          min-width="150"
          label="Name">
        </el-table-column>
        <el-table-column
          width="150"
          label="Actions">
          <template slot-scope="scope">
            <router-link :to="{ name: 'deduplication', params: { deduplicationId: scope.row.id } }">
              <el-button size="mini">Open</el-button>
            </router-link>
            <el-button
              style="margin-left: 10px"
              icon="el-icon-delete"
              :disabled="isDeleting[scope.row.id]"
              :loading="isDeleting[scope.row.id]"
              size="mini"
              @click="onDelete(scope.row.id)"></el-button>
          </template>
        </el-table-column>
      </el-table>
    </el-row>
  </div>
</template>
```

Рисунок 3.4 – Лістинг коду шаблону Vue.js

На рисунку 3.5 наведено приклад реалізації функцій описаної вище компоненти Vue.js.

```
export default {
  components: {
  },
  data() {
    return {
      deduplications: [],
      isLoading: false,
      isDeleting: {},
    }
  },
  async created() {
    await this.load()
  },
  methods: {
    async load() {
      this.isLoading = true
      var response = await client.getDeduplications()
      this.isLoading = false

      this.deduplications = response.data
      console.log(this.deduplications)
    },
    onAdd() {
      this.$router.push({ name: 'deduplication-wizard' })
    },
    onDelete(id) {
      this.$confirm('Are you sure you want to delete this deduplication?', 'Warning', {
        confirmButtonText: 'Yes',
        cancelButtonText: 'Cancel',
        type: 'warning'
      }).then(() => {
        this.deleteDeduplication(id)
      });
    },
    async deleteDeduplication(id) {
      try {
        this.setIsDeleting(id, true)
        await client.deleteDeduplication(id)
        await this.load()
      } catch (e) {
        notify.responseError(e)
      } finally {
        this.setIsDeleting(id, false)
      }
    },
    setIsDeleting(id, value) {
      var newIsDeleting = {}
      Object.assign(newIsDeleting, this.isDeleting)
      newIsDeleting[id] = value
      this.isDeleting = newIsDeleting
    }
  }
}
```

Рисунок 3.5 – Лістинг коду компоненти Vue.js

Також було створено окремий JavaScript модуль для виклику API, описаного в попередньому підрозділі. Цей клієнт використовується в компонентах для виклику REST API методів та передачі даних до клієнта. На рисунку 3.6 наведено код реалізації цього клієнтського модуля.

```
const axios = require('axios');
import auth from './auth.js';
const baseUrl = process.env.VUE_APP_API;

export default {
  async getDeduplications() {
    return axios.get(`${baseUrl}/deduplications`)
  },
  async createDeduplication(value) {
    return axios.post(`${baseUrl}/deduplications/?userId=${await this.userId()}`, value)
  },
  async deleteDeduplication(id) {
    return axios.delete(`${baseUrl}/deduplications/${id}?userId=${await this.userId()}`)
  },
  async getDataSources() {
    return axios.get(`${baseUrl}/data-sources?userId=${await this.userId()}`)
  },
  async getCompanies() {
    return axios.get(`${baseUrl}/companies?userId=${await this.userId()}`)
  },
  async getDataSource(id) {
    return axios.get(`${baseUrl}/data-sources/${id}?userId=${await this.userId()}`)
  },
  async updateDataSource(id, value) {
    return axios.patch(`${baseUrl}/data-sources/${id}?userId=${await this.userId()}`, value)
  },
  async getUserSettings() {
    return axios.get(`${baseUrl}/settings?userId=${await this.userId()}`)
  },
  async updateUserSettings(value) {
    return axios.patch(`${baseUrl}/settings?userId=${await this.userId()}`, value)
  },
  async deleteUser() {
    return axios.post(`${baseUrl}/settings/delete-account?userId=${await this.userId()}`)
  },
  getDataSourceZipUploadUrl(dataSourceId, userId) {
    return `${baseUrl}/data-sources/${dataSourceId}/upload/zip?userId=${userId}`
  },
  async createDataSource(value) {
    return axios.post(`${baseUrl}/data-sources/?userId=${await this.userId()}`, value)
  },
  async userId() {
    var user = await auth.getUser();
    if (user) {
      return user.profile.sub
    }

    return null;
  }
}
```

Рисунок 3.6 – Лістинг коду клієнту

### 3.5 Розробка модуля інтеграції

Для доступу до даних сторонніх застосунків було розроблено інтерфейс модуля інтеграцій, який дозволяє стандартизувати доступ та обробку даних. Таким чином, веб-сервіс може завантажувати та опрацьовувати дані контактів із різних джерел. На першому етапі було створено три реалізації цього інтерфейсу для доступу до контактів сервісів Google Contacts, Microsoft Office 365 Contacts та iCloud Contacts.

Інтерфейс дозволяє виконувати основні CRUD операції над контактами:

– зчитати усі контакти;
– зчитати контакт по ID;
– створити контакт із заданими даними;
– змінити контакт за заданим ID та новими даними;
– видалити контакт за заданим ID;

Кожна із операцій викликає відповідний метод публічного API стороннього застосунку. Варто зауважити, що формат даних контактів відрізняється від сервісу до сервісу, тому дані уніфіковуються в фіксований формат, з яким працює веб-сервіс. Цей формат описаний стандартом JSONSchema та наведено на рисунку 3.7 на прикладі Google Contacts.

```
{
  "type": "object",
  "modelType": "GoogleContact",
  "modelBaseType": "GoogleContact",
  "title": "Google Contact",
  "properties": {
    "firstName": {
      "type": "string",
      "modelType": "GoogleContactFirstName",
      "modelBaseType": "ContactFirstName",
      "title": "First name",
      "fullTitle": "Google contact first name"
    },
    "lastName": {
      "type": "string",
      "modelType": "GoogleContactLastName",
      "modelBaseType": "ContactLastName",
      "title": "Last name",
      "fullTitle": "Google contact last name"
    },
    "emails": {
      "type": "array",
      "modelType": "GoogleContactEmailList",
      "modelBaseType": "ContactEmailList",
      "title": "Emails",
      "fullTitle": "Google contact email list",
    },
    "phones": {
      "type": "array",
      "modelType": "GoogleContactPhoneList",
      "modelBaseType": "ContactPhoneList",
      "title": "Phones",
      "fullTitle": "Google contact phone list",
    }
  }
}
```


Рисунок 3.7 – Лістинг коду JSONSchem

На рисунку 3.8 також наведено інтерфейс модуля інтеграцій із зовнішніми застосунками. Приклад повної реалізації цього інтерфейсу для сервісу Google Contacts наведено в додатку А.

```
module.exports = {
  // метод ініціалізації
  async init(account) {},

  // метод для завантаження конакту
  async get(id) {},

  // метод для видалення контакту
  async delete(id) {},

  // метод для створення контакту
  async insert(value) {},

  // метод для зміни контакту
  async patch(id, value) {},

  // метод для завантаження усіх контактів
  async getAll() {}
}
```

Рисунок 3.8 – Лістинг коду інтерфейсу модуля

### 3.6 Висновки

У третьому розділі було проаналізовано та обґрунтовано вибір мови програмування та технологій для розробки веб-сервісу. Для реалізації back-end було обрано мову програмування C# та фреймворк .NET Core Web API. Базу даних було реалізовано використовуючи систему менеджменту баз даних Postgres, а доступ до неї налаштовано через .NET Entity Framework Core. Для зручності розробки графічного дизайну було обрано JavaScript фреймворк Vue.js. Модулі доступу до даних контактів зовнішніх застосунків було реалізовано за допомогою Node.JS модулів.

# 4 ТЕСТУВАННЯ РОБОТИ ВЕБ-СЕРВІСУ

## 4.1 Тестування роботи веб-сервісу

Тестування (quality assurance, QA) будь-якого програмного продукту проводиться з ціллю перевірки правильності роботи усіх його функцій для різних можливих сценаріїв вхідних даних користувача. Під час тестування важливо перевірити не лише ті сценарії роботи із продуктом які очікуються від користувача, а й ті, які можуть бути несподіваним для системи. Будь-який такий сценарій має бути передбаченим і правильно обробленим під час розробки.

Під час тестування можуть оцінювати:

– відповідність вимогам, якими керувалися проектувальники та розробники;

– правильність відповіді для всіх можливих вхідних даних;

– виконання функцій за прийнятний час;

– практичність;

– сумісність із програмним забезпеченням та операційними системами;

– відповідність задачам замовника.

За об’єктом тестування виділяють наступні види тестування ПЗ:

– функціональне тестування;

– дослідницьке тестування;

– тестування продуктивності;

– навантажувальне тестування;

– тестування стабільності;

– тестування зручності використання;

– тестування інтерфейсу користувача;

– тестування безпеки.

У залежності від переслідуваних цілей, види тестування можна умовно розділити на наступні типи:

– функціональне тестування;
– нефункціональне тестування;
– тестування пов’язане зі змінами.

За знанням системи:

– тестування чорної скриньки;
– тестування білої скриньки;
– тестування сірої скриньки.

За ступенем автоматизації:

– ручне тестування;
– автоматизоване тестування;
– напівавтоматизоване тестування.

За часом проведення тестування:

– Альфа-тестування;
– тестування нової функціональності;
– Бета-тестування;
– регресійне тестування. [14]

Оскільки число можливих тестів навіть для нескладних програмних компонентів практично нескінченне, тому стратегія тестування полягає в тому, щоби провести всі можливі тести з урахуванням наявного часу та ресурсів. Як результат програмне забезпечення тестують стандартним виконанням програми з метою виявлення багів (помилок або інших дефектів) [15].

Для перевірки коректності роботи веб сервісу було прийнято рішення провести ручне тестування освновних його функцій. При початку роботи із веб-сервісом «Deduplicator» користувачу потрібно залогуватись (рис 4.1).

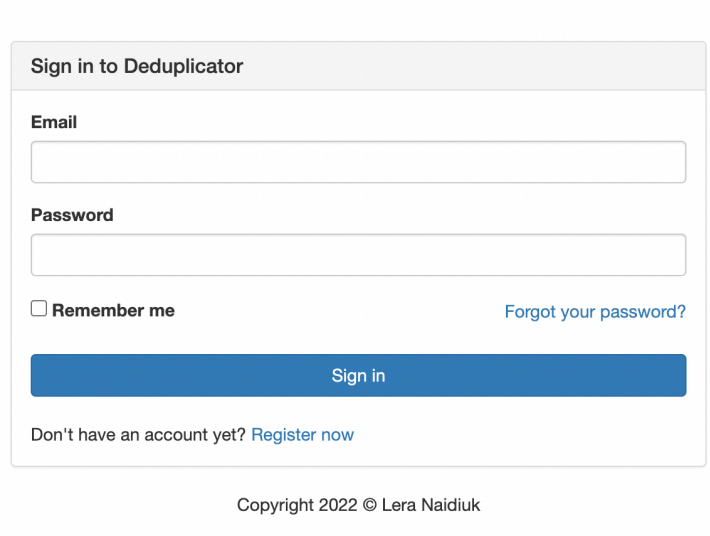


Рисунок 4.1 – Сторінка логування користувача

Якщо на момент входу в користувача немає акаунту, він має натиснути на «Register» та перейти на сторінку реєстрації нового користувача веб-сервісу (рисунок 4.2). Для реєстрації користувач повинен надати адресу електронної пошти, повне ім’я та пароль. Пароль потрібно також підтвердити. У разі некоректного заповнення будь-якого із полів, користувачу буде повідомлено про помилку.

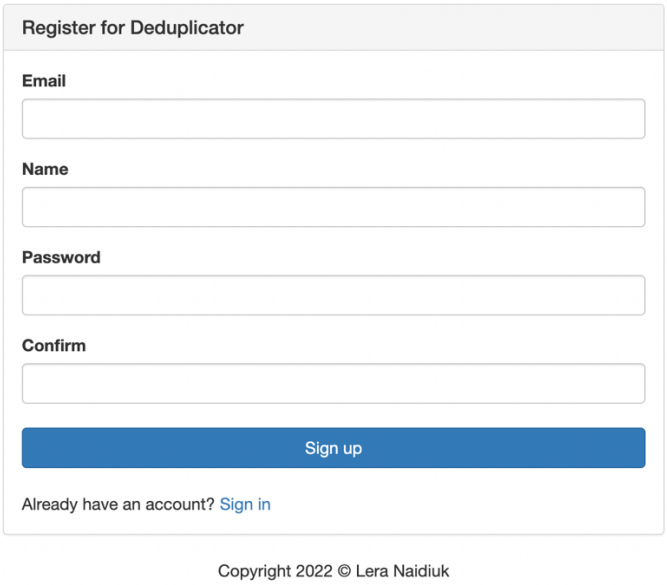


Рисунок 4.2 – Сторінка реєстрації нового користувача

Після реєстрації та входу користувач бачить головну сторінку веб-сервісу. У лівому меню користувач має доступ до трьох сторінок: список аналізів на дублікати (Deduplications), налаштування користувача (Settings) та допомога (Help). Список виконаних аналізів на дублікати початково є пустим (рис. 4.3)

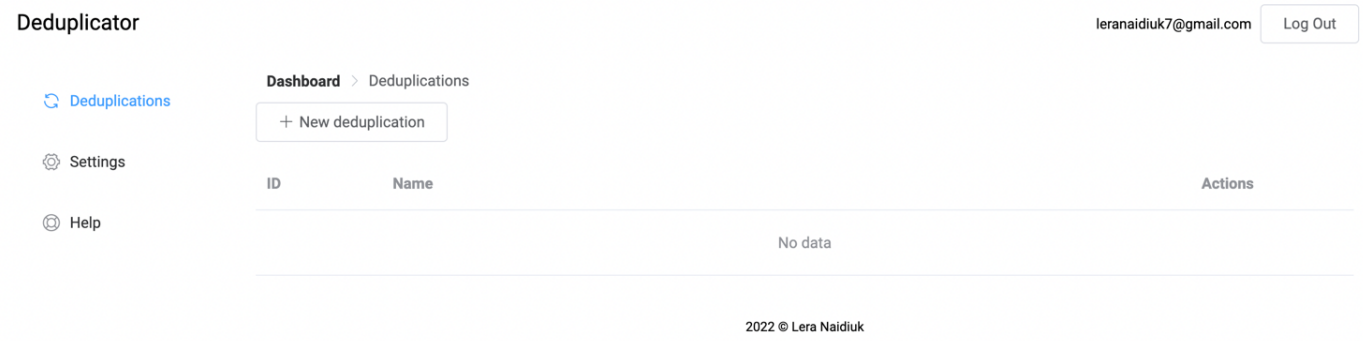


Рисунок 4.3 – Початкова сторінка нового користувача

Наступним етапом тестування веб-сервісу є створення нового аналізу на дублікати. Для цього користувачу потрібно натиснути на кнопку «New deduplication» (рис 4.4).

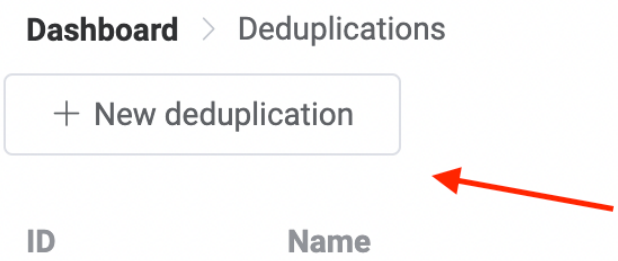


Рисунок 4.4 – Кнопка створення нового аналізу

Після натиснення кнопки користувач бачить сторінку створення нового аналізу на дублікати (рисунок 4.5). На ній перечислені зовнішні застосунки, з якими на цей момент є інтегрованим веб-сервіс. Вибравши застосунок буде створено новий аналіз.

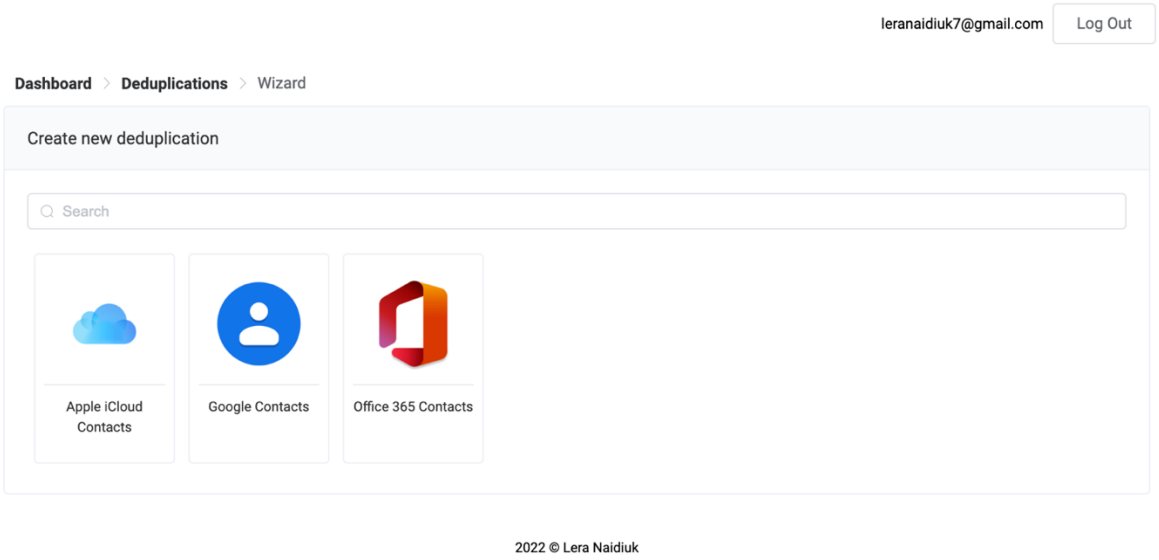


Рисунок 4.5 – Сторінка створення нового аналізу

Початковий вигляд сторінки аналізу контактів Google Contacts на дублікати показано на рисунку 4.6.

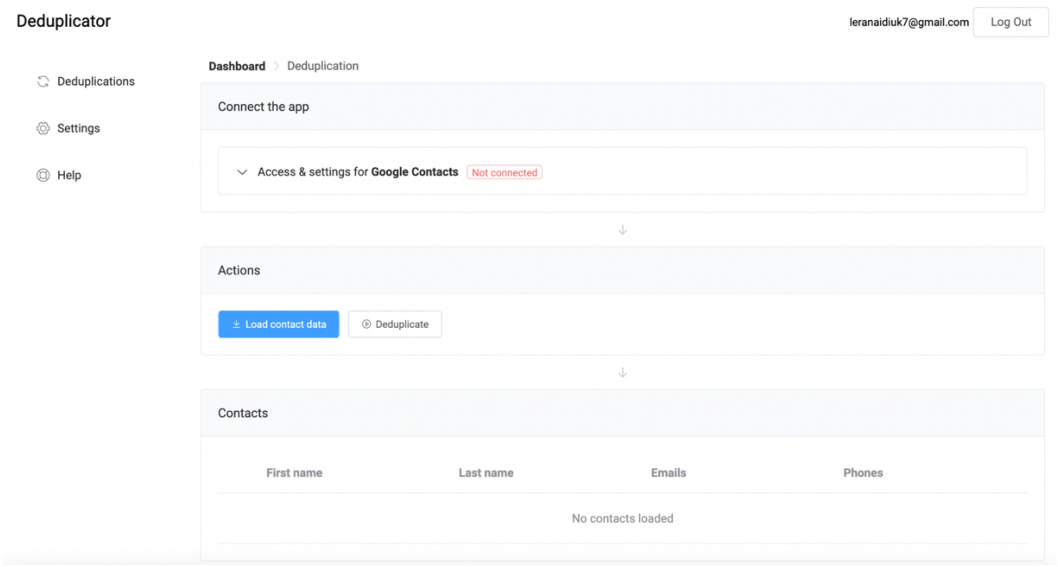


Рисунок 4.6 – Початковий вигляд сторінки аналізу контактів

Як бачимо, початково обліковий запис Google Contacts не є приєднаним, тож потрібно його приєднати, натиснувши кнопку «CONNECT» (рис. 4.7).

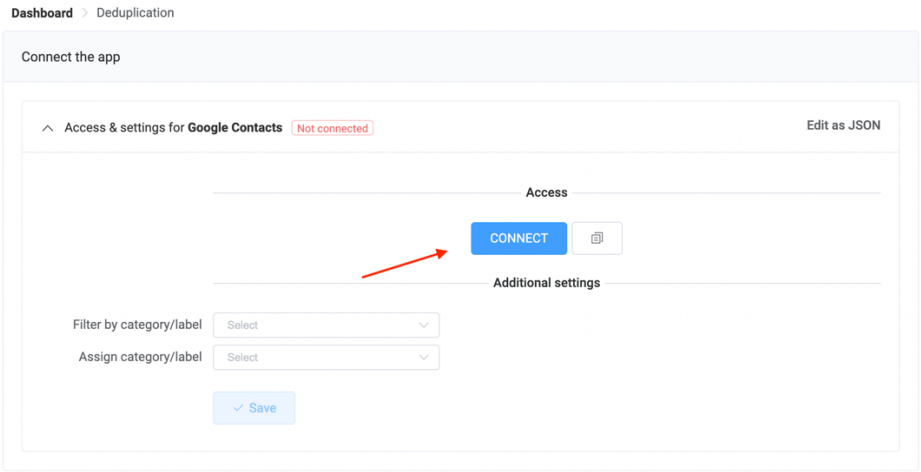


Рисунок 4.7 – Кнопка приєднання зовнішнього застосунку

Під час приєднання користувач буде переадресований на сторінку авторизації Google, де він має ввести свій логін та пароль та надати доступ до своїх контактів (рис. 4.8). Для авторизації користувача використовується протокол OAuth 2.0, який підтримує Google Contacts API.

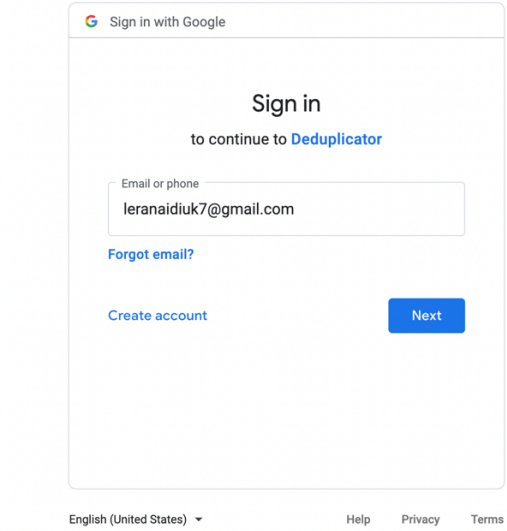


Рисунок 4.8 – Процес авторизації Google Contacts

Після успішного приєднання облікового запису Google, користувач побачить екран, показаний на рисунку 4.9. Зелений напис Connected означає, що Google профіль користувача був успішно приєднаним, і веб-сервіс має доступ до даних контактів.

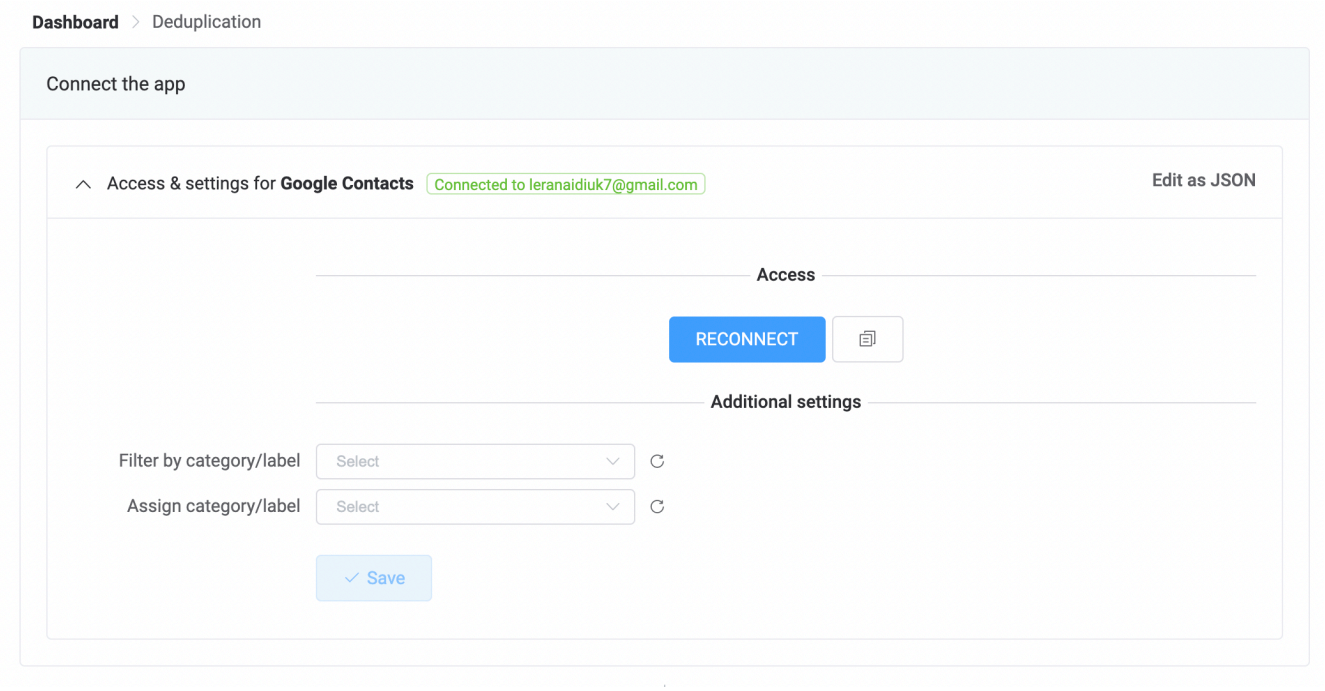


Рисунок 4.9 – Успішне приєднання до Google Contacts

Наступним етапом роботи із веб-сервісом є процес завантаження бази даних контактів користувача. Для цього потрібно натиснути кнопку «Load contact data» (рис. 4.10).

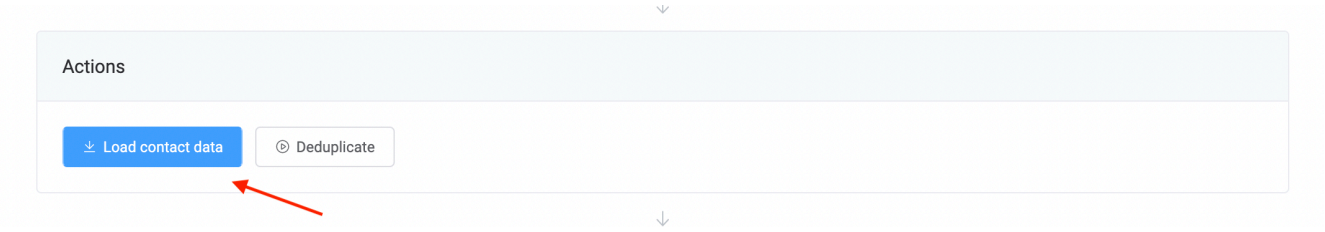


Рисунок 4.10 – Кнопка завантаження даних контактів

Після натискання цієї кнопки відбувається процес завантаження усіх даних контактів приєднаного користувача. Для заванитаження даних виконується запит до відповідного зовнішнього API, отримані дані якого обробляє модуль та приводить до спільного формату.

Завантажені контакти відображаються в таблиці контактів разом із відповідними іменами, прізвищами, електронними поштами та номерами телефонів (рис. 4.11). Зауважте, що процес завантаження може зайняти декілька секунд, в залежності від кількості контактів.

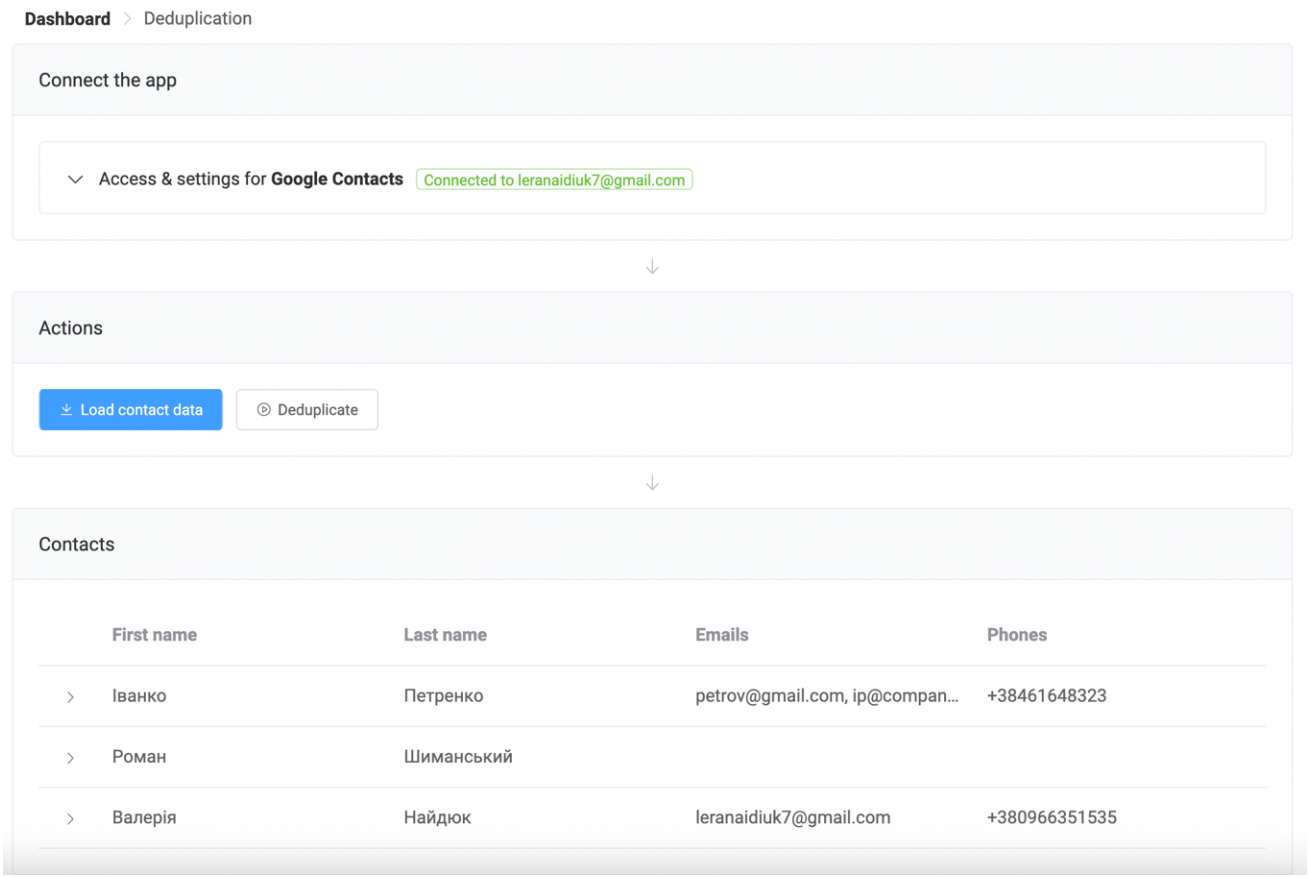


Рисунок 4.11 – Завантажені контакти

Одночасно із завантаженням контактів відбувається їх злиття та групування. Контакти, які було розпізнано як дублікати алгоритмом відображаються разом у одній групі (рис. 4.12).

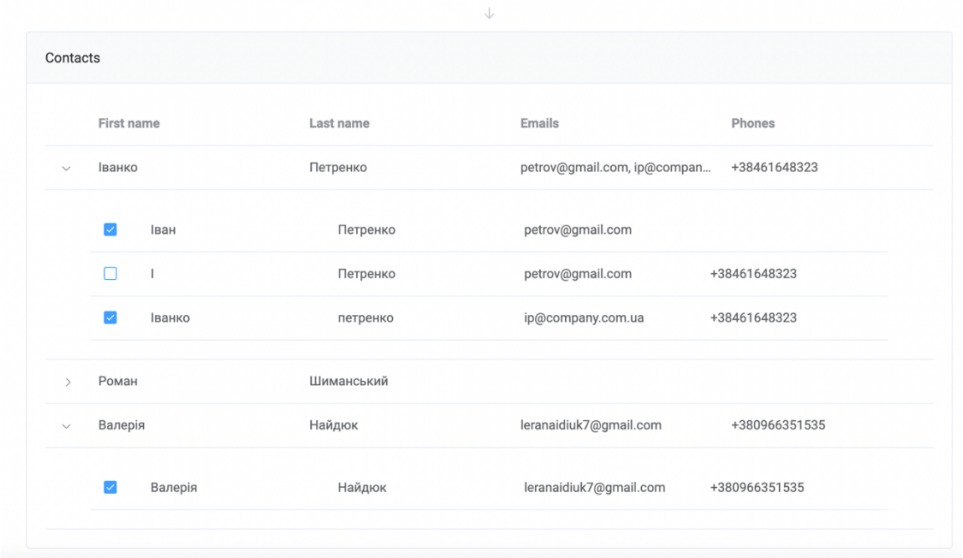


Рисунок 4.12 – Групування контактів

Користувач також може вибрати, які контакти входять в групу, яка буде злитою в один контакт. Номери телефонів та електронні пошти усіх контактів групи зливаються в один список.

Після завершення аналізу та групування користувач може натиснути кнопку «Deduplicate» (рис. 4.13) та розпочати процес видалення та злиття дублікатів.

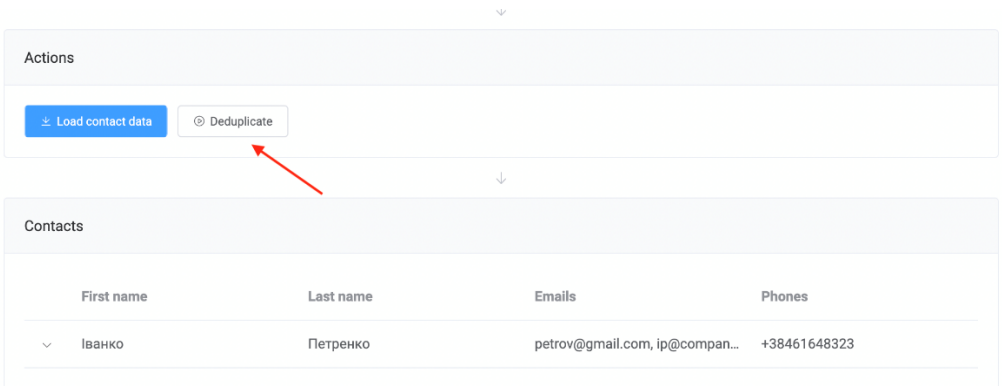


Рисунок 4.13 – Кнопка видалення дублікатів

Після завершення злиття контактів список буде перезавантаженим та не буде містити дублікатів. Як і з початковим завантаження, повторне завантаження контактів може зайняти певний час, в залежності від кількості контактів користувача.

Якщо користувач задоволений результатом і більше не потребує сервісу, він може повернутись до списку аналізів та видалити створений ним аналіз (рис 4.14). Користувач може також залишити створений аналіз та повернутись до нього у майбутньому для провторного видалення дублікатів.


Рисунок 4.14 – Видалення аналізу

Завершивши роботу користувач може вийти із профілю користувача, натиснувши кнопку «Log Out» (рис. 4.15). Після цього від буде автоматично вилогований із акануту і переадресований на сторінку входу користувача у веб сервіс.


Рисунок 4.15 – Кнопка виходу із профіля користувача

### 4.2 Розробка інструкції користувача

Інструкція користувача – це документ, який визначає усі технічі вимоги для запуску програмного продукту. Деталі щодо мінімальної та рекомендованої конфігурації серверного комп'ютера наведено в таблицях 4.1 та 4.2.

Таблиця 4.1 – Мінімальна конфігурація:

| | |
|---|---|
| Тип процесора | 64-розрядний процесор з тактовою частотою 1 ГГц |
| Об'єм оперативної пам'яті | 2 ГБ |
| Місце на жорсткому диску | 10 ГБ |
| Операційна система | Ubuntu 18.04 |
| Програмне забезпечення | .NET Core 2 або вище, Node JS 10 або вище |

Таблиця 4.2 – Рекомендована конфігурація:

| | |
|---|---|
| Тип процесора | 64-розрядний процесор з тактовою частотою 2 ГГц |
| Об'єм оперативної пам'яті | 4 ГБ |
| Розмір жорсткого диску | 50 ГБ |
| Операційна система | Ubuntu 20.04 |
| Програмне забезпечення | .NET Core 3, Node JS 16 |

Варто зауважити, що для запуску сервера слід спершу встановити середовище .NET Core Development Kit та Node JS, які використовуються для запуску серверної частини (API) та модулів доступу до контактів.

Окрім цього слід встановити базу даних Postgres та налаштування з'єднання з нею використовуючи Connection String. База даних повинна містити запис головного користувача (адміністратора системи). Для налаштування серверу у середовищі Ubuntu слід використати Nginx з кофігурацією відповідного сервісу, який оброблятиме запити та перенаправлятиме їх у відповідний процес.

### 4.3 Висновки

У четвертому розділі було проведено повноцінне ручне тестування програмного забезпечення, яке розглянуло основні сценарії роботи із продуктом та перевірило коректність виконання усіх основних функцій. Проведене тестування показало працездатність та ефективність використання веб сервісу на реальних даних. Окрім цього, було створено інструкцію користувача веб сервісу, в якій наведені мінімальні та стандартні параметри конфігурації сервера, необхідні для його коректної роботи.

# ВИСНОВКИ

Під час виконання бакалаврської дипломної роботи було розроблено веб-сервіс для моніторингу дублікатів контактів. Для розробки було використано середовище програмної розробки Microsoft Visual Studio Code.

Було проаналізовано стан питання на сьогоднішній день. Було розглянуто основні аналоги програмного продукту. Було визначено їхні недоліки та переваги порівняно з власним програмним продуктом. Базуючись на порівнянні, було сформульовано основні задачі бакалаврської дипломної роботи.

Також було проведено аналіз технологій розробки та обґрунтовано вибір мови програмування C# та технологій .NET Web API, Vue.js та Node.JS. Також було розглянуто переваги використання баз даних Postgres для збереження даних користувача. Було розроблено метод, модель і алгоритм роботи веб-системи пошуку дублікатів контактів.

Під час виконання бакалаврської дипломної роботи було зроблено:

– розроблено базу даних відповідно до вимог веб-системи;
– розроблено метод і алгоритм знаходження дублікатів контактів;
– розроблено модель роботи веб-системи;
– розроблено API;
– розроблено модулі для з’єднання (інтеграції) зі сторонніми застосунками;
– розроблено графічний інтерфейс для веб-середовища;
– налаштовано взаємодію між модулями з’єднання із застосунками, API та веб-інтерфейсом сервісу;
– проведено тестування модулів та сервісу з використанням графічного інтерфейсу.

Тестування веб-сервісу показало повну працездатність даного програмного продукту та відповідність поставленому завданню.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ


1. Що таке хмарні технології і навіщо вони потрібні [Електронний ресурс] – Режим доступу до ресурс: https://edin.ua/shho-take-xmarni-texnologi%D1%97-i-navishho-voni-potribni/
2. Бевз С. В. Особливості розробки веб сервісу для моніторингу дублікатів контактів / С. В. Бевз, С. М. Бурбело, В.В. Войтко, Л.М. Круподьорова, В.І. Найдюк // Матеріали Всеукраїнської науково-практичної інтернет-конференції "Молодь в науці: дослідження, проблеми, перспективи – 2022", Секція – Інформаційні технології та комп‘ютерна інженерія. [Електронний ресурс] – Режим доступу до ресурсу: https://conferences.vntu.edu.ua/index.php/mn/mn2022/paper/viewFile/16209/13646
3. Deduply [Електронний ресурс] – Режим доступу до ресурсу: https://dedupe.ly/
4. Contacts+ [Електронний ресурс] – Режим доступу до ресурсу: https://www.contactsplus.com/
5. RingLead [Електронний ресурс] – Режим доступу до ресурсу: https://www.ringlead.com/
6. Що таке діаграма діяльності? - визначення з техопедії - Розвиток - 2022 [Електронний ресурс] – Режим доступу до ресурсу: https://uk.theastrologypage.com/activity-diagram
7. Краткий обзор языка C# [Електронний ресурс] – Режим доступу до ресурсу: https://docs.microsoft.com/ru-ru/dotnet/csharp/tour-of-csharp/
8. Andrew Troelsen Pro C# 8 with .NET Core 3 : навч. посіб. / Andrew Troelsen, Phil Japikse – Apress, 2020. – 1881 ст.
9. Прикладний програмний інтерфейс [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Прикладний_програмний_інтерфейс

10. Розуміння основ роботи API і REST API – короткий вступ [Електронний ресурс] – Режим доступу до ресурсу: https://sebweo.com/rozuminnya-osnov-roboti-api-i-rest-api-korotkij-vstup/
11. Tutorial: Create a web API with ASP.NET Core [Електронний ресурс] – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-web-api?view=aspnetcore-6.0&tabs=visual-studio
12. Прогресивний JavaScript фреймворк [Електронний ресурс] – Режим доступу до ресурсу: https://vuejs.org.ua/v2/guide/index.html
13. Vue.js [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Vue.js
14. Види тестування та відмінності між ними. Шпаргалка з тестування [Електронний ресурс] – Режим доступу до ресурсу: https://qagroup.com.ua/publications/vydy-testuvannya-ta-vidminnosti-mizh-nymy/
15. Тестування програмного забезпечення [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Тестування_програмного_забезпечення

# ДОДАТКИ

**Додаток А – Технічне завдання**

Міністерство освіти і науки України
Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії

ЗАТВЕРДЖУЮ
д.т.н., проф. О. Н. Романюк
25 березня 2022 р.

**Технічне завдання**
**на бакалаврську дипломну роботу «Розробка веб сервісу для моніторингу дублікатів контактів»**
**за спеціальністю**
**121 – Інженерія програмного забезпечення**

Керівник бакалаврської дипломної роботи:
_____ доц. Войтко В. В.
"____"_____2022 р.
Виконав:
_____студентка гр. 2ПІ-18Б Найдюк В. І.
"____"_____2022 р.

Вінниця – 2022 року

**1. Найменування та галузь застосування**

Бакалаврська дипломна робота: «Розробка веб сервісу для моніторингу дублікатів контактів»

Галузь застосування – ПЗ для бізнесу.

**2. Підстава для розробки.**

Завдання на роботу, яке затверджене на засіданні кафедри програмного забезпечення – протокол № 66 від « 24 » березня 2022 р.

**3. Мета та призначення розробки.**

Метою бакалаврської дипломної роботи є підвищення ефективності роботи із контактами шляхом розробки веб-сервісу для моніторингу, пошуку та видалення дублікатів контактів у широкого спектру застосунків для користувачів та бізнесу, що дозволить оптимізувати базу контактів.

Призначення роботи – веб сервіс для моніторингу, пошуку, видалення та злиття дублікатів контактів.

**4. Вихідні дані для проведення НДР**

**1.** Andrew Troelsen Pro C# 8 with .NET Core 3 : навч. посіб. / Andrew Troelsen, Phil Japikse – Apress, 2020. – 1881 ст.

**2.** Designing for the Digital Age: How to Create Human-Centered Products and Services. / Kim Goodwin – Wiley, 2009. – 768 ст.

**3.** The Art of Software Testing, 3rd Edition / Glenford J. Myers, Corey Sandler, Tom Badgett, Wiley 2011. – 256 ст.

**5. Технічні вимоги**

*Склад* комплексу *технічних засобів*

Для розв’язку задачі буде використовуватися сучасний персональний комп’ютер.

*Вимоги до* складових *частин комплексу технічних засобів.*

Для запуску програми апаратне забезпечення повинне відповідати мінімальним вимогам:

- ПК із процесором Intel i3;
- 1 Гб оперативної пам’яті;
- дисплей;
- ОС Windows або Linux;

**6. Конструктивні вимоги.**

Конструкція пристрою повинна відповідати естетичним та ергономічним вимогам, повинна бути зручною в обслуговуванні та керуванні.

Графічна та текстова документація повинна відповідати діючим стандартам України.

**7. Перелік технічної документації, що пред’являється по закінченню робіт:**

– пояснювальна записка до бакалаврської дипломної роботи;

– технічне завдання;

– лістинги програми.

**8. Вимоги до рівня уніфікації та стандартизації**

При розробці програмних засобів слід дотримуватися уніфікації і ДСТУ.

## Додаток Б – Протокол перевірки на плагіат

ПРОТОКОЛ ПЕРЕВІРКИ БАКАЛАВРСЬКОЇ ДИПЛОМНОЇ РОБОТИ НА НАЯВНІСТЬ ТЕКСТОВИХ ЗАПОЗИЧЕНЬ

Назва роботи: Розробка веб-сервісу для моніторингу дублікатів контактів

Тип роботи: БДР

Підрозділ: кафедра програмного забезпечення, ФІТКІ

Науковий керівник: Войтко В. В.

| | |
|---|---|
| Оригінальність | |
| Схожість | |

### Аналіз звіту подібності

■ **Запозичення, виявлені у роботі, оформлені коректно і не містять ознак плагіату.**

□ Виявлені у роботі запозичення не мають ознак плагіату, але їх надмірна кількість викликає сумніви щодо цінності роботи і відсутності самостійності її автора. Роботу направити на доопрацювання.

□ Виявлені у роботі запозичення є недобросовісними і мають ознаки плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень.

Особа, відповідальна за перевірку _____ Черноволик Г. О.

Ознайомлені з повним звітом подібності, який був згенерований системою Unicheck

Автор роботи _____ Найдюк В. І.

Керівник роботи _____ Войтко В. В.

# Додаток В – Лістинг імплементації графічного інтерфейсу

## Deduplications.js

```
<template>
  <div>
    <el-row>
      <el-button-group>
        <el-button size="medium" icon="el-icon-plus" @click="onAdd">New deduplication</el-
button>
      </el-button-group>
    </el-row>
    <el-row v-loading="isLoading">
      <el-table
        empty-text="No data"
        :data="deduplications">
        <el-table-column
          prop="id"
          label="ID"
          width="60">
        </el-table-column>
        <el-table-column
          width="56">
          <template slot-scope="scope">
            <img :src="scope.row.dataSource.logoUrl" class="sync-img-small syncs-logo"
:title="scope.row.dataSource.name"/>
          </template>
        </el-table-column>
        <el-table-column
          prop="name"
          min-width="150"
          label="Name">
        </el-table-column>
        <el-table-column
          width="150"
          label="Actions">
          <template slot-scope="scope">
            <router-link :to="{ name: 'deduplication', params: { deduplicationId:
scope.row.id } }">
              <el-button size="mini">Open</el-button>
            </router-link>
            <el-button
              style="margin-left: 10px"
              icon="el-icon-delete"
              :disabled="isDeleting[scope.row.id]"
```

```
              :loading="isDeleting[scope.row.id]"
              size="mini"
              @click="onDelete(scope.row.id)"></el-button>
          </template>
        </el-table-column>
      </el-table>
    </el-row>
  </div>
</template>

<script>
import client from './../client.js'
import notify from './../notify.js'

export default {
  components: {
  },
  data() {
    return {
      deduplications: [],
      isLoading: false,
      isDeleting: {},
    }
  },
  async created() {
    await this.load()
  },
  methods: {
    async load() {
      this.isLoading = true
      var response = await client.getDeduplications()
      this.isLoading = false

      this.deduplications = response.data
      console.log(this.deduplications)
    },
    onAdd() {
      this.$router.push({ name: 'deduplication-wizard' })
    },
    onDelete(id) {
      this.$confirm('Are you sure you want to delete this deduplication?', 'Warning', {
        confirmButtonText: 'Yes',
        cancelButtonText: 'Cancel',
        type: 'warning'
      }).then(() => {
        this.deleteDeduplication(id)
```

```
      });
    },
    async deleteDeduplication(id) {
      try {
        this.setIsDeleting(id, true)
        await client.deleteDeduplication(id)
        await this.load()
      } catch (e) {
        notify.responseError(e)
      } finally {
        this.setIsDeleting(id, false)
      }
    },
    setIsDeleting(id, value) {
      var newIsDeleting = {}
      Object.assign(newIsDeleting, this.isDeleting)
      newIsDeleting[id] = value
      this.isDeleting = newIsDeleting
    }
  }
}
</script>

<style>
.sync-img-small {
  width: 23px;
  vertical-align: middle
}
.sync-char {
  vertical-align: middle
}
.syncs-logo {
  border: 1px solid #dcdfe6;
  border-radius: 50%;
  padding: 3px;
}
</style>
```

DeduplicationWizard.vue

```
<template>
  <div>
    <el-row v-loading="isLoading">
      <el-card shadow="never">
```

```
        <div slot="header">
          <span class="header">Create new deduplication</span>
        </div>
        <el-row>
          <div>
            <el-row>
              <el-input
                size="small"
                clearable
                placeholder="Search"
                prefix-icon="el-icon-search"
                v-model="searchTerm">
              </el-input>
            </el-row>
            <el-row>
              <DataSourceCard v-for="d in dataSourcesFiltered" :key="d.id" :data-source="d"
@click="createDeduplication(d)"></DataSourceCard>
            </el-row>
          </div>
        </el-row>
      </el-card>
    </el-row>

  </div>
</template>

<script>
import notify from './../notify.js'
import client from './../client.js'
import auth from './../auth.js'
import DataSourceCard from './DataSourceCard.vue'

export default {
  components: {
    DataSourceCard
  },
  data() {
    return {
      searchTerm: '',
      isLoading: false,
      isSaving: false,
      dataSources: [],
    }
  },
  async created() {
    var user = await auth.getUser();
```

```
    if (user != null) {
      this.isAuthenticated = true
    } else {
      this.isAuthenticated = false
    }

    await this.load();
  },
  watch: {
    $route: {
      immediate: true,
      handler() {
        if (this.$route.query != null && this.$route.query.search != null) {
          this.searchTerm = this.$route.query.search
        }
      }
    }
  },
  methods: {
    async load() {
      this.isLoading = true
      var response = await client.getDataSources()
      this.isLoading = false
      this.dataSources = response.data
    },
    async createDeduplication(dataSource) {
      console.log(dataSource)
      try {
        this.isSaving = true
        var response = await client.createDeduplication({
          dataSourceId: dataSource.id,
          name: 'Deduplication of ' + dataSource.name
        });

        var deduplicationId = response.data.id

        this.$router.push({ name: 'deduplication', params: { deduplicationId:
deduplicationId } })
      } catch (error) {
        notify.responseError(error)
      } finally {
        this.isSaving = false
      }
    }
  },
  computed: {
```

```
    dataSourcesFiltered() {
      return this.dataSources.filter(d => {
        return (d.id == 121 || d.id == 96 || d.id == 97) && d.category == 'Contacts' &&
d.customizable == true && (!this.searchTerm || this.searchTerm.length == 0 ||
d.name.toLowerCase().indexOf(this.searchTerm.toLowerCase()) >= 0 || (d.keywords != null &&
d.keywords.toLowerCase().indexOf(this.searchTerm.toLowerCase()) >= 0) ||
d.category.toLowerCase().indexOf(this.searchTerm.toLowerCase()) >= 0)
      })
    }
  }
}
</script>
<style scoped>
.wizard-info {
  font-size: 12px;
  margin-top: 20px;
  color: #6b6b6b;
}
.back-button-wizard {
  float: left;
  margin-right: 10px;
}
.el-steps--simple {
  padding: 10px 8%;
}
</style>
<style>
.app .el-step__title {
  white-space: nowrap;
  overflow: hidden;
  text-overflow: ellipsis;
  width: 125px;
}
.connector-button {
  width: 100px;
  height: 100px;
  font-size: 25px;
}
.connector-connection {
  display: inline-block;
  width: 100%;
  vertical-align: middle;
  border-color: #DCDFE6;
  border-style: dashed;
  border-width: 3px;
  box-sizing: border-box;
```

```
}
.wizard-part {
  text-align: center;
  margin: 25px
}
.wizard-header {
  color: #303133;
  font-size: 18px;
  font-weight: 500;
  margin-bottom: 10px;
}
.app-select-container {
  padding-top: 20px;
  padding-bottom: 20px;
}
.connector-name-container {
  margin-top: 15px;
  display: inline-block;
  margin-bottom: 0px;
  width: 100%;
}
.wizard-select {
  display: table;
  max-width: 600px;
  height: 180px;
  margin-left: auto;
  margin-right: auto;
  margin-bottom: 0px;
}
.connector-name {
  color: #606266;
  font-size: 14px;
  font-weight: 500;
}
.wizard-form {
  margin-top: 0px;
}
.connector-placeholder {
  display: inline-block;
  width: 100px;
  background-color: #eaeaea;
  height: 10px;
}
.connector-placeholder-small {
    width: 74px;
}
```

```
.connectors-not-found {
  font-style: italic;
  text-align: center;
}
.category-container {
  overflow: hidden;
  overflow-x: scroll;
  scrollbar-width: none;
}
.category-container::-webkit-scrollbar {
  display: none;
}
.data-source-category-divider {
  margin-bottom: 35px;
  margin-top: 35px;
}
.data-source-category-divider .el-divider__text {
  padding: 0 30px;
  width: -webkit-max-content;
  width: -moz-max-content;
  width: max-content;
}
</style>
```

# Додаток Г – Лістинг імплементації API

## DeduplicatorController.cs

```
using System.Threading.Tasks;
using MediatR;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Deduplicator.Api.Infrastructure;
using Deduplicator.Core.Commands.Syncs;

namespace Deduplicator.Api.Controllers
{
    [Authorize]
    [Route("deduplications")]
    public class DeduplicationController : Controller
    {
        private readonly IMediator _mediator;

        public DeduplicationController(IMediator mediator)
        {
            _mediator = mediator;
        }

        [HttpGet]
        [Route("")]
        public async Task<IActionResult> GetDeduplications()
        {
            var request = new UserDeduplicationsRequest
            {
                Login = Request.HttpContext.User.GetLoginContext()
            };

            if (request.Login.UserId == null)
            {
                return Unauthorized();
            }

            return Ok(await _mediator.Send(request));
        }

        [HttpGet]
        [Route("{id}")]
        public async Task<IActionResult> GetDeduplication(int id)
```

```
        {
            var request = new DeduplicationRequest
            {
                Login = Request.HttpContext.User.GetLoginContext(),
                SyncId = id
            };

            if (request.Login.UserId == null && request.Login.SyncId == null)
            {
                return Unauthorized();
            }

            return Ok(await _mediator.Send(request));
        }

        [HttpDelete]
        [Route("{id}")]
        public async Task<IActionResult> DeleteDeduplication(int id)
        {
            var request = new DeduplicationDeleteRequest
            {
                Login = Request.HttpContext.User.GetLoginContext(),
                DeduplicationId = id
            };

            if (request.Login.UserId == null)
            {
                return Unauthorized();
            }

            await _mediator.Send(request);
            return Ok();
        }

        [HttpPost]
        [Route("")]
        public async Task<IActionResult>
CreateDeduplication([FromBody]CreateDeduplicationCommand command)
        {
            command.Login = Request.HttpContext.User.GetLoginContext();

            if (command.Login.UserId == null)
            {
                return Unauthorized();
            }
```

```
            return Ok(await _mediator.Send(command));
        }

        [HttpPatch]
        [Route("{id}")]
        public async Task<IActionResult> UpdateDeduplication(int id,
[FromBody]UpdateDeduplicationCommand command)
        {
            command.Login = Request.HttpContext.User.GetLoginContext();
            command.SyncId = id;

            if (command.Login.UserId == null && command.Login.SyncId == null)
            {
                return Unauthorized();
            }

            return Ok(await _mediator.Send(command));
        }
    }
}
```

UserDeduplicatosRequest.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using MediatR;
using Microsoft.EntityFrameworkCore;
using Newtonsoft.Json;
using Deduplicator.Core.Contracts;
using Deduplicator.Core.Infrastructure;
using Deduplicator.Core.Models;
using Deduplicator.Core.Services;

namespace Deduplicator.Core.Commands.Deduplications
{
    public class UserDeduplicationsRequest : IRequest<List<DeduplicationInfo>>
    {
        public LoginContext Login { get; set; }
    }
```

```
    public class UserDeduplicationsRequestHandler :
IRequestHandler<UserDeduplicationsRequest, List<DeduplicationInfo>>
    {
        private readonly SyncContext _context;

        public UserDeduplicationsRequestHandler(SyncContext context)
        {
            _context = context;
        }

        async public Task<List<DeduplicationInfo>> Handle(UserDeduplicationsRequest
request, CancellationToken cancellationToken)
        {
            if (request.Login.UserId == null)
            {
                throw new Exception("Not allowed to read deduplications");
            }

            var dedups = await _context
                .Deduplications
                .Include(s => s.DataSource)
                .Where(s => s.UserId == request.Login.UserId && s.Deleted == false)
                .OrderBy(s => s.Id)
                .ToListAsync();

            return dedups
                .Select(s => new DeduplicationInfo(s))
                .ToList();
        }
    }
}
```

CreateDeduplicationCommand.cs

```
using System;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using MediatR;
using Microsoft.EntityFrameworkCore;
using Deduplicator.Core.Contracts;
using Deduplicator.Core.Infrastructure;
using Deduplicator.Core.Models;
using Deduplicator.Core.Services;
```

```
namespace Deduplicator.Core.Commands.Syncs
{
    public class CreateDeduplicationCommand : IRequest<DeduplicationInfo>
    {
        public LoginContext Login { get; set; }

        public int DataSourceId { get; set; }

        public string Name { get; set; }
    }

    public class CreateDeduplicationCommandHandler :
IRequestHandler<CreateDeduplicationCommand, DeduplicationInfo>
    {
        private readonly SyncContext _context;
        private readonly EncryptionService _encryption;

        public CreateDeduplicationCommandHandler(SyncContext context, EncryptionService
encryption)
        {
            _context = context;
            _encryption = encryption;
        }

        async public Task<DeduplicationInfo> Handle(CreateDeduplicationCommand command,
CancellationToken cancellationToken)
        {
            if (command.Login.UserId == null)
            {
                throw new Exception("Now allowed to create deduplication");
            }

            var dataSource = await _context
                .DataSources
                .FirstOrDefaultAsync(d => d.Id == command.DataSourceId);

            if (dataSource == null)
            {
                throw new Exception("Data source not found");
            }

            var dedup = new Deduplication
            {
                UserId = command.Login.UserId.Value,
                DataSourceId = command.DataSourceId,
                Name = command.Name,
```

```
                Data = "{}"
            };

            await _context.AddAsync(dedup);
            await _context.SaveChangesAsync();

            return new DeduplicationInfo(dedup);
        }
    }
}
```

# Додаток Ґ – Лістинг модуля інтеграції Google Contacts

## dataSourceApp.js

```
const moment = require('moment')
const helper = require('syncpeng-helper');
const SyncPenguinError = require('syncpeng-error')
const Access = require('./access');
const FieldsService = require('./fieldsService')

module.exports = {
  account: null,
  access: null,
  personFields: ['names', 'emailAddresses', 'addresses', 'urls', 'phoneNumbers', 'organizations', 'occupations', 'metadata', 'genders', 'birthdays', 'biographies', 'memberships'],

  async init(account) {
    this.account = account
    this.access = new Access(account)
  },

  async getModel() {
    var fieldsService = new FieldsService(this.account)

    return await fieldsService.getModelFields()
  },

  async connect(value) {
    await this.access.getTokens(value)

    return {
      patchAccountData: this.account
    }
  },

  async getAccount() {
    var res = {
      connectUrl: null,
      name: null,
      isEmpty: true,
      isConnected: false,
      isDisconnected: false,
      fieldValues: null,
      validationResults: null,
```

```
            patchAccountData: null
          }

          if (helper.hasStringContent(this.account.refreshToken)) {
            res.isEmpty = false
          }

          res.connectUrl = this.access.connectUrl

          try {
            await this.access.init()

            try {
              await this.requestWrapper(async () => { return await this.access.client.people.connections.list({
                personFields: this.personFields,
                resourceName: 'people/me',
                pageSize: 1
              }) })
              res.isConnected = true
            } catch (e) {
              if (this.is401(e)) {
                res.isDisconnected = true
                return res
              }
            }

            res.name = await this.getName()
            res.fieldValues = await this.getFieldValues()
            res.validationResults = await this.validate(res.fieldValues)
          } catch (e) { }

          res.patchAccountData = this.account

          return res
        },

        async getName() {
          try {
            var response = await this.access.authClient.userinfo.get()
            return response.data.email;
          } catch (e) {}

          return null
        },
```

```
        async getFieldValues() {
          var res = {
            labelId: [],
            assignLabelId: []
          }

          try {
            await this.access.init()

            var groups = await this.loadGroups()
            res.labelId = groups.map(g => {
              return {
                value: g.resourceName,
                label: g.formattedName
              }
            })

            res.assignLabelId = res.labelId
          } catch (e) { console.log(e) }

          return res
        },

        async loadGroups() {
          var groupsResponse = await this.requestWrapper(async () => { return await this.access.client.contactGroups.list({
            pageToken: null,
            pageSize: 1000
          }) })

          return groupsResponse.data.contactGroups
        },

        async validate(fieldValues) {
          try {
            await this.requestWrapper(async () => { return await this.access.client.people.connections.list({
              personFields: this.personFields,
              resourceName: 'people/me',
              pageSize: 1
            }) })
          } catch (e) {
            console.log(e)
            if (this.is401(e)) {
              return [{
                type: 'error',
```

```
            message: `Unable to connect. Please try to reconnect your account`
          }]
        }

        return [{
          type: 'error',
          message: `An error while verifying connection to the account`
        }]
      }

      if (helper.hasStringContent(this.account.assignLabelId) &&
!fieldValues.assignLabelId.some(a => a.value == this.account.assignLabelId)) {
        return [{
          type: 'error',
          field: 'assignLabelId',
          message: `Assigned Google label was not found`
        }]
      }

      if (helper.hasStringContent(this.account.labelId) && !fieldValues.labelId.some(a
=> a.value == this.account.labelId)) {
        return [{
          type: 'error',
          field: 'labelId',
          message: `Filtered Google label was not found`
        }]
      }

      return []
    },

    async get(id) {
      await this.access.init()

      try {
        var response = await this.requestWrapper(async () => { return await
this.access.client
          .people
          .get({
            resourceName: id,
            personFields: this.personFields
          }) })

        var labels = await this.loadLabels()

        return {
```

```
            lastUpdated:
moment.utc(response.data.metadata.sources[0].updateTime).toISOString(),
            value: this.buildModel(response.data, labels),
            patchAccountData: this.account
          }
        } catch (e) {
          throw new SyncPenguinError(`An unknown error occurred while retrieving Google
contact with ID '${id}'`, e)
        }
      },

      async delete(id) {
        await this.access.init()

        try {
          await this.requestWrapper(async () => { await this.access.client
            .people
            .deleteContact({
              resourceName: id
            }) })

          return {
            patchAccountData: this.account
          }
        } catch (e) {
          throw new SyncPenguinError(`An unknown error occurred while deleting Google
contact with ID '${id}'`, e)
        }
      },

      async insert(value) {
        await this.access.init()

        try {
          var googleContact = {}
          await this.setModel(googleContact, value)

          this.setGoogleLabel(googleContact)

          var response = await this.requestWrapper(async () => { return await
this.access.client
            .people
            .createContact({
              requestBody: googleContact
            }) })
```

```
            return {
              id: response.data.resourceName,
              patchAccountData: this.account
            }
          } catch (e) {
            throw new SyncPenguinError(`An unknown error occurred while creating a Google
contact`, e)
          }
        },

        async patch(id, value) {
          await this.access.init()

          try {
            var googleContact = {}
            await this.setModel(googleContact, value)

            this.setGoogleLabel(googleContact)

            var getResponse = await this.requestWrapper(async () => { return await
this.access.client
              .people
              .get({
                resourceName: id,
                personFields: this.personFields
              }) })

            var oldGoogleContact = getResponse.data;
            var newGoogleContact = Object.assign(oldGoogleContact, googleContact)

            await this.requestWrapper(async () => { await this.access.client
              .people
              .updateContact({
                resourceName: id,
                updatePersonFields: this.personFields.filter(f => f != 'metadata'),
                requestBody: newGoogleContact
              }) })

            return {
              patchAccountData: this.account
            }
          } catch (e) {
            throw new SyncPenguinError(`An unknown error occurred while updating Google
contact with ID '${id}'`, e)
          }
        },
```

```
        async getUpdates(lastSync) {
          return await this.getUpdatesBase(lastSync)
        },

        async getAll() {
          return await this.getUpdatesBase('0')
        },

        async getUpdatesBase(lastSync, retry) {
          await this.access.init()

          try {
            var connections = [];
            var nextSync = null

            var pageToken = null;
            var newConnections = [];

            try {
              do {
                var response = await this.requestWrapper(async () => { return await
this.access.client.people.connections.list({
                  personFields: this.personFields,
                  resourceName: 'people/me',
                  pageToken: pageToken,
                  requestSyncToken: true,
                  syncToken: lastSync == '0' ? null : lastSync,
                  pageSize: 500
                }) })

                nextSync = response.data.nextSyncToken

                newConnections = []

                if (response.data.connections != null) {
                  newConnections = response
                    .data
                    .connections
                }

                pageToken = response.data.nextPageToken
                connections = connections.concat(newConnections)
              } while (newConnections.length > 0 && pageToken != null)
            } catch (e) {
```

```
            if (!retry && e.errors != null && e.errors[0] != null && e.errors[0].message
== 'Sync token is expired. Clear local cache and retry call without the sync token.') {
              console.log('Sync token expired')
              return await this.getUpdatesBase(null, true)
            }

            throw e
          }

          if (lastSync == null) {
            connections = []
          }

          var deletedEntries = connections.filter(c => c.metadata.deleted == true).map(c
=> {
            return {
              id: c.resourceName,
              type: 'delete'
            }
          })

          var upsertEntries = []

          var labels = await this.loadLabels()

          for (var i = 0; i < connections.length; ++i) {
            var googleContact = connections[i]
            if (googleContact.metadata.deleted != true) {
              if            (helper.hasStringContent(this.account.labelId)            &&
(googleContact.memberships    ==    null    ||    !googleContact.memberships.some(m    =>
m.contactGroupMembership  !=  null  &&  m.contactGroupMembership.contactGroupResourceName  ==
`${this.account.labelId}`))) {
                continue
              }

              var model = this.buildModel(googleContact, labels)

              upsertEntries.push({
                id: googleContact.resourceName,
                type: 'upsert',
                lastUpdated:
moment.utc(googleContact.metadata.sources[0].updateTime).toISOString(),
                label: this.getLabel(model),
                value: model
              })
            }
```

```
            }

            return {
              nextSync: nextSync,
              entries: upsertEntries.concat(deletedEntries),
              patchAccountData: this.account
            }
          } catch (e) {
            throw new SyncPenguinError(`An unknown error occurred while retrieving Google
contacts`, e)
          }
        },

        setGoogleLabel(googleContact) {
          if (helper.hasStringContent(this.account.assignLabelId) &&
googleContact.memberships == null) {
            googleContact.memberships = [
              {
                contactGroupMembership: {
                  contactGroupId:
this.account.assignLabelId.substr('contactGroups/'.length),
                  contactGroupResourceName: this.account.assignLabelId
                }
              }
            ]
          }
        },

        getLabel(model) {
          if (!helper.hasStringContent(model.firstName) &&
!helper.hasStringContent(model.lastName)) {
            return (model.emails == null || model.emails.length == 0) ? null :
model.emails[0].email
          }

          var label = ''

          if (helper.hasStringContent(model.firstName)) label += model.firstName
          if (helper.hasStringContent(model.lastName)) {
            if (label != '') label += ' '
            label += model.lastName
          }

          if (model.emails != null && model.emails[0] != null &&
helper.hasStringContent(model.emails[0].email)) {
            label += ` (${model.emails[0].email})`
```

```
    }

    return label
  },


  buildModel(googleContact, labels) {
    return {
      firstName: this.getFirstName(googleContact),
      lastName: this.getLastName(googleContact),
      middleName: this.getMiddleName(googleContact),
      displayName: this.getDisplayName(googleContact),
      title: this.getTitle(googleContact),
      description: this.getDescription(googleContact),
      company: this.getCompany(googleContact),
      jobTitle: this.getJobTitle(googleContact),
      department: this.getDepartment(googleContact),
      website: this.getWebsite(googleContact),
      birthdate: this.getBirthdate(googleContact),
      emails: this.getEmails(googleContact),
      phones: this.getPhones(googleContact),
      addresses: this.getAddresses(googleContact),
      labels: this.getLabels(googleContact, labels)
    }
  },

  async setModel(googleContact, model) {
    this.setFirstName(googleContact, model.firstName)
    this.setLastName(googleContact, model.lastName)
    this.setMiddleName(googleContact, model.middleName)
    this.setTitle(googleContact, model.title)
    this.setDescription(googleContact, model.description)
    this.setCompany(googleContact, model.company)
    this.setJobTitle(googleContact, model.jobTitle)
    this.setDepartment(googleContact, model.department)
    this.setWebsite(googleContact, model.website)
    this.setBirthdate(googleContact, model.birthdate)
    this.setEmails(googleContact, model.emails)
    this.setPhones(googleContact, model.phones)
    this.setAddresses(googleContact, model.addresses)
    await this.setLabels(googleContact, model.labels)
  },

  getProp(googleContact, arrName, propName) {
    var item = googleContact[arrName] == null ? null : googleContact[arrName][0]
    if (item == null) return null
```

```
  return helper.hasStringContent(item[propName]) ? item[propName] : null
},

setProp(googleContact, modelValue, arrName, propName) {
  if (modelValue !== undefined) {
    if (googleContact[arrName] == null) {
      googleContact[arrName] = [{}]
    }

    googleContact[arrName][0][propName] = modelValue
  }
},

getFirstName(googleContact) {
  return this.getProp(googleContact, 'names', 'givenName')
},

setFirstName(googleContact, modelValue) {
  this.setProp(googleContact, modelValue, 'names', 'givenName')
},

getLastName(googleContact) {
  return this.getProp(googleContact, 'names', 'familyName')
},

setLastName(googleContact, modelValue) {
  this.setProp(googleContact, modelValue, 'names', 'familyName')
},

getMiddleName(googleContact) {
  return this.getProp(googleContact, 'names', 'middleName')
},

setMiddleName(googleContact, modelValue) {
  this.setProp(googleContact, modelValue, 'names', 'middleName')
},

getDisplayName(googleContact) {
  return this.getProp(googleContact, 'names', 'displayName')
},

getTitle(googleContact) {
  return this.getProp(googleContact, 'names', 'honorificPrefix')
},

setTitle(googleContact, modelValue) {
```

```
    this.setProp(googleContact, modelValue, 'names', 'honorificPrefix')
  },

  getDescription(googleContact) {
    return this.getProp(googleContact, 'biographies', 'value')
  },

  setDescription(googleContact, modelValue) {
    this.setProp(googleContact, modelValue, 'biographies', 'value')
  },

  getCompany(googleContact) {
    var value = this.getProp(googleContact, 'organizations', 'name')
    if (!helper.hasStringContent(value)) return null;
    return {
      name: value
    }
  },

  setCompany(googleContact, modelValue) {
    if (modelValue != null) {
      this.setProp(googleContact, modelValue.name, 'organizations', 'name')
    }
  },

  getJobTitle(googleContact) {
    return this.getProp(googleContact, 'organizations', 'title')
  },

  setJobTitle(googleContact, modelValue) {
    this.setProp(googleContact, modelValue, 'organizations', 'title')
  },

  getDepartment(googleContact) {
    return this.getProp(googleContact, 'organizations', 'department')
  },

  setDepartment(googleContact, modelValue) {
    this.setProp(googleContact, modelValue, 'organizations', 'department')
  },

  getWebsite(googleContact) {
    return this.getProp(googleContact, 'urls', 'value')
  },

  setWebsite(googleContact, modelValue) {
```

```
      this.setProp(googleContact, modelValue, 'urls', 'value')
    },

    getBirthdate(googleContact) {
      var value = this.getProp(googleContact, 'birthdays', 'date')
      try {
        return moment.utc(value.day + '-' + value.month + '-' + value.year, 'DD-MM-
YYYY').toISOString()
      } catch (e) {
        return null
      }
    },

    setBirthdate(googleContact, modelValue) {
      if (helper.hasStringContent(modelValue)) {
        var m = moment.utc(modelValue)
        this.setProp(googleContact, {
          year: m.format('YYYY'),
          month: m.format('MM'),
          day: m.format('DD')
        }, 'birthdays', 'date')
      }
    },

    getEmails(googleContact) {
      if (googleContact.emailAddresses == null) return []

      return                googleContact.emailAddresses.filter(e                =>
helper.hasStringContent(e.value)).map(e => {
        return {
          email: e.value,
          type: this.toBaseEmailAddressType(e.type)
        }
      })
    },

    setEmails(googleContact, modelValue) {
      if (modelValue == null) return;

      googleContact.emailAddresses = modelValue.map(e => {
        return {
          value: e.email,
          type: this.fromBaseEmailAddressType(e.type)
        }
      })
    },
```

```
        async loadLabels() {
          var response = await this.requestWrapper(async () => { return await
this.access.client.contactGroups.list({
            pageToken: null,
            pageSize: 1000
          }) })

          return response.data.contactGroups
        },

        getLabels(googleContact, labels) {
          var res = []

          if (googleContact.memberships != null) {
            googleContact.memberships.forEach(m => {
              if (m.contactGroupMembership != null) {
                var label = labels.filter(l => l.resourceName ==
m.contactGroupMembership.contactGroupResourceName)[0]
                if (label != null) {
                  res.push({
                    name: label.formattedName
                  })
                }
              }
            })
          }

          return res
        },

        async setLabels(googleContact, modelValue) {
          if (modelValue == null) return;

          var allLabels = await this.loadLabels()
          var memberships = []

          for (var i = 0; i < modelValue.length; ++i) {
            var label = modelValue[i]
            var existing = allLabels.filter(l => label.name != null && l.formattedName ==
label.name.trim())[0]

            if (existing == null) {
              var insertResponse = await this.requestWrapper(async () => { return await
this.access.client.contactGroups.create({
                requestBody: {
```

```
                contactGroup: {
                  name: label.name
                }
              }
            }) })

              existing = insertResponse.data
              allLabels.push(insertResponse.data)
            }

            memberships.push({
              contactGroupMembership: {
                contactGroupId: existing.resourceName.substr('contactGroups/'.length),
                contactGroupResourceName: existing.resourceName
              }
            })
          }

          if (memberships.length == 0) {
            memberships = [{
              contactGroupMembership: {
                contactGroupId: 'myContacts',
                contactGroupResourceName: 'contactGroups/myContacts'
              }
            }]
          }

          googleContact.memberships = memberships
        },

        getPhones(googleContact) {
          if (googleContact.phoneNumbers == null) return []

          return                    googleContact.phoneNumbers.filter(e                    =>
helper.hasStringContent(e.value)).map(e => {
            return {
              phone: e.value,
              type: this.toBasePhoneNumberType(e.type)
            }
          })
        },

        setPhones(googleContact, modelValue) {
          if (modelValue == null) return;

          googleContact.phoneNumbers = modelValue.map(e => {
```

```
            return {
              value: e.phone,
              type: this.fromBasePhoneNumberType(e.type)
            }
          })
        },

        getAddresses(googleContact) {
          if (googleContact.addresses == null) return []

          return googleContact.addresses.map(e => {
            return {
              country: helper.hasStringContent(e.country) ? e.country : null,
              postalCode: helper.hasStringContent(e.postalCode) ? e.postalCode : null,
              region: helper.hasStringContent(e.region) ? e.region : null,
              city: helper.hasStringContent(e.city) ? e.city : null,
              streetAddress: helper.hasStringContent(e.streetAddress) ? e.streetAddress :
null,
              extendedAddress: helper.hasStringContent(e.extendedAddress) ?
e.extendedAddress : null,
              type: this.toBaseAddressType(e.type),
            }
          })
        },

        setAddresses(googleContact, modelValue) {
          if (modelValue == null) return;

          googleContact.addresses = modelValue.map(e => {
            return {
              country: e.country,
              postalCode: e.postalCode,
              region: e.region,
              city: e.city,
              streetAddress: e.streetAddress,
              extendedAddress: e.extendedAddress,
              type: this.fromBaseAddressType(e.type)
            }
          })
        },

        toBaseEmailAddressType(type) {
          switch (type) {
            case 'home': return 'home'
            case 'work': return 'work'
          }
```

```
    return 'other'
  },

  fromBaseEmailAddressType(type) {
    switch (type) {
      case 'home': return 'home'
      case 'work': return 'work'
    }

    return 'other'
  },

  toBasePhoneNumberType(type) {
    switch (type) {
      case 'home': return 'home'
      case 'work': return 'work'
      case 'mobile': return 'mobile'
      case 'homeFax': return 'homeFax'
      case 'workFax': return 'workFax'
      case 'workMobile': return 'workMobile'
    }

    return 'other'
  },

  fromBasePhoneNumberType(type) {
    switch (type) {
      case 'home': return 'home'
      case 'work': return 'work'
      case 'mobile': return 'mobile'
      case 'homeFax': return 'homeFax'
      case 'workFax': return 'workFax'
      case 'workMobile': return 'workMobile'
    }

    return 'other'
  },

  toBaseAddressType(type) {
    switch (type) {
      case 'home': return 'home'
      case 'work': return 'work'
    }

    return 'other'
```

```
        },

        fromBaseAddressType(type) {
          switch (type) {
            case 'home': return 'home'
            case 'work': return 'work'
          }

          return 'other'
        },

        async requestWrapper(action, isRetry) {
          try {
            return await action()
          } catch (e) {
            if (this.is401(e) && !isRetry && helper.hasStringContent(this.account.refreshToken)) {
              try {
                await this.access.refreshToken()
              } catch (e) {}

              return await this.requestWrapper(action, true)
            }

            throw e
          }
        },

        is401(e) {
          return (e.response != null && e.response.status == 401);
        }
      }
```

access.js

```
const { google } = require('googleapis');
const queryString = require('query-string');
const axios = require('axios');

const client = {
  clientId: "",
  clientSecret: "",
  redirectUri: "",
```

```
  scope: "https://www.googleapis.com/auth/calendar
https://www.googleapis.com/auth/userinfo.email"
}

module.exports = function(account) {
  return {
    account: account,
    client: null,
    oauth: null,
    authClient: null,
    connectUrl: `https://accounts.google.com/o/oauth2/v2/auth?${queryString.stringify({
      scope: client.scope,
      access_type: "offline",
      redirect_uri: client.redirectUri,
      response_type: "code",
      prompt: "select_account consent",
      client_id: client.clientId
    })}`,

    async init() {
      this.oauth = new google.auth.OAuth2(
        client.clientId,
        client.clientSecret,
        client.redirectUrl)

      this.oauth.setCredentials({
        access_token: account.accessToken || "0"
      })

      this.client = google.calendar({
        version: 'v3',
        auth: this.oauth
      })

      this.authClient = google.oauth2({
        version: 'v2',
        auth: this.oauth
      })
    },

    async getTokens(connectValue) {
      var response = await axios
        .post(
          'https://oauth2.googleapis.com/token',
          queryString.stringify({
            grant_type: 'authorization_code',
```

```
            code: connectValue.code,
            client_id: client.clientId,
            client_secret: client.clientSecret,
            redirect_uri: client.redirectUri
          }),
          {
            timeout: 240000
          })

      this.account.accessToken = response.data.access_token
      this.account.refreshToken = response.data.refresh_token
    },

    async refreshToken() {
      console.log('Access token refresh')

      var response = await axios
        .post(
          'https://oauth2.googleapis.com/token',
          queryString.stringify({
            grant_type: 'refresh_token',
            refresh_token: this.account.refreshToken,
            client_id: client.clientId,
            client_secret: client.clientSecret,
            redirect_uri: client.redirectUri
          }),
          {
            timeout: 240000
          })

      this.account.accessToken = response.data.access_token
      if (helper.hasStringContent(response.data.refresh_token)) {
        this.account.refreshToken = response.data.refresh_token
      }

      this.oauth.setCredentials({
        access_token: account.accessToken
      })
    },
  }
}
```

**Додаток Д – Графічна частина**

РОЗРОБКА ВЕБ СЕРВІСУ ДЛЯ МОНІТОРИНГУ ДУБЛІКАТІВ КОНТАКТІВ


Рисунок Д.1 – Назва роботи


Рисунок Д.2 – Мета, об'єкт і предмет дослідження

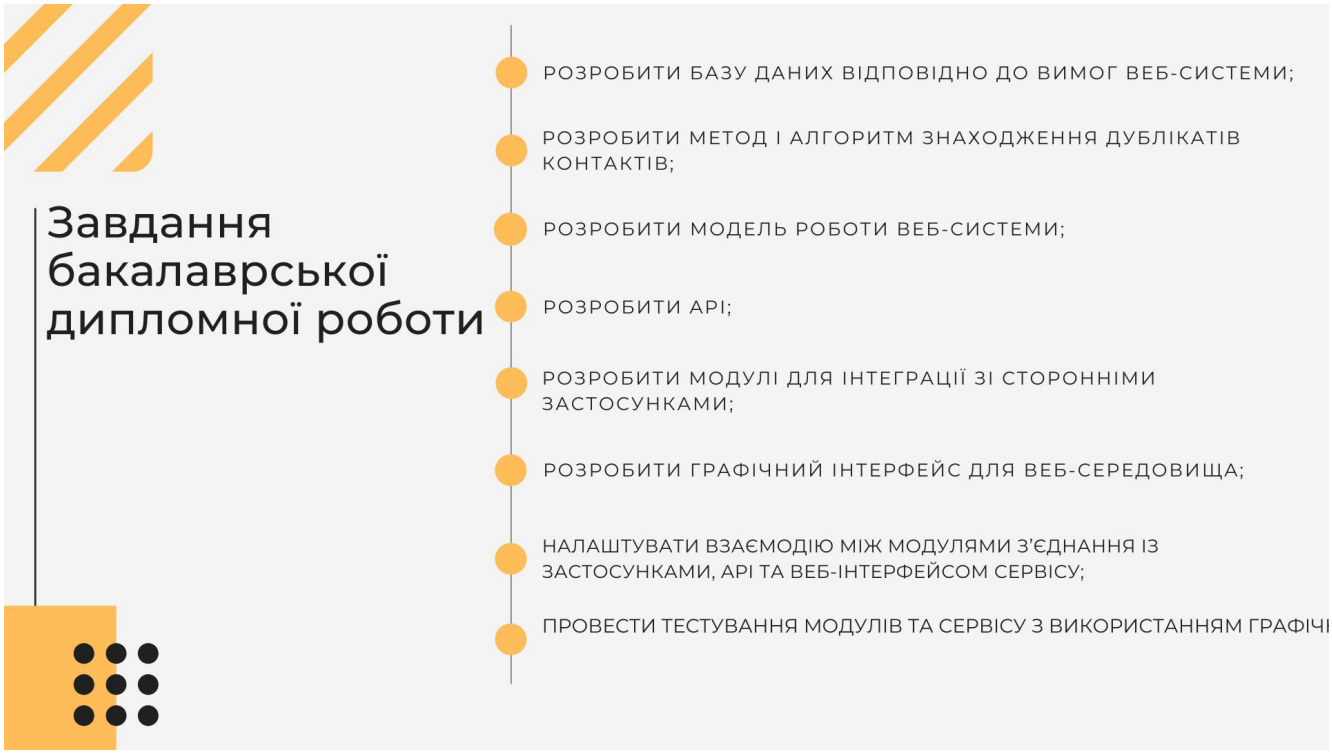


Рисунок Д.3 – Завдання бакалаврської дипломної роботи

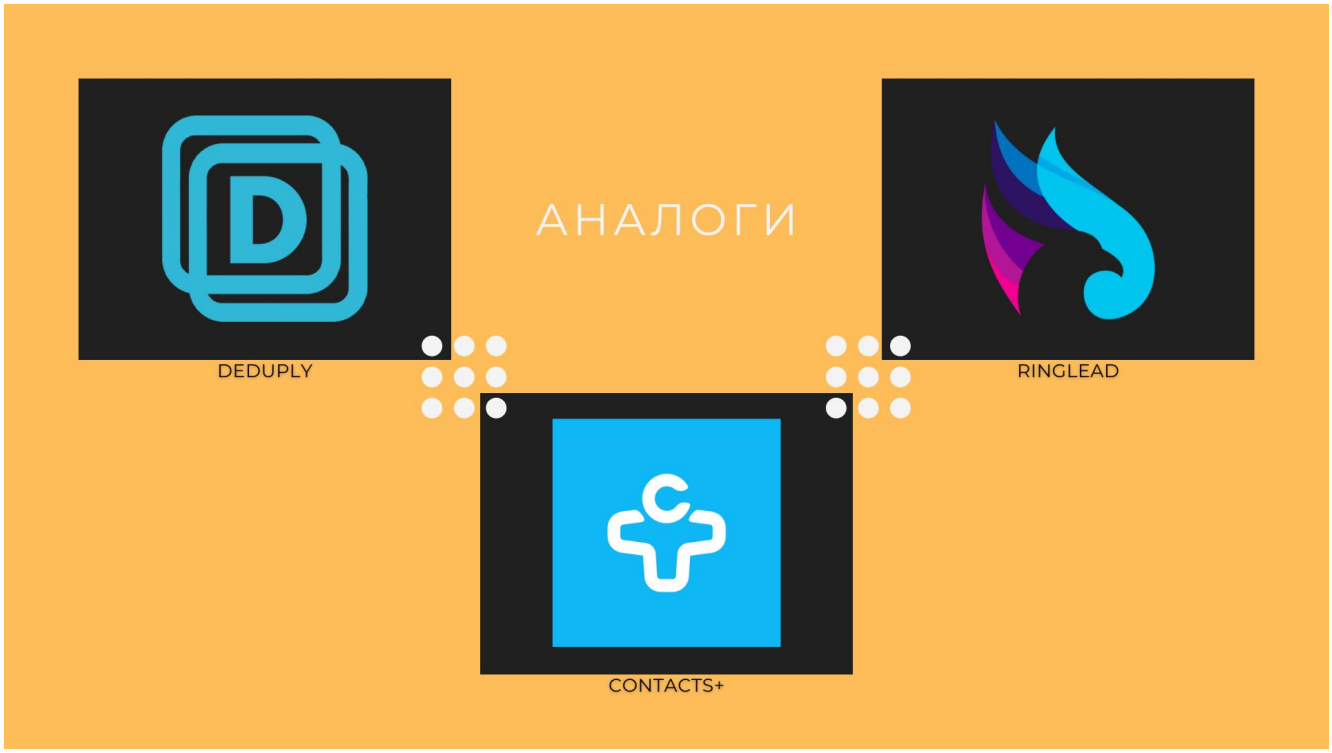


Рисунок Д.4 – Аналоги

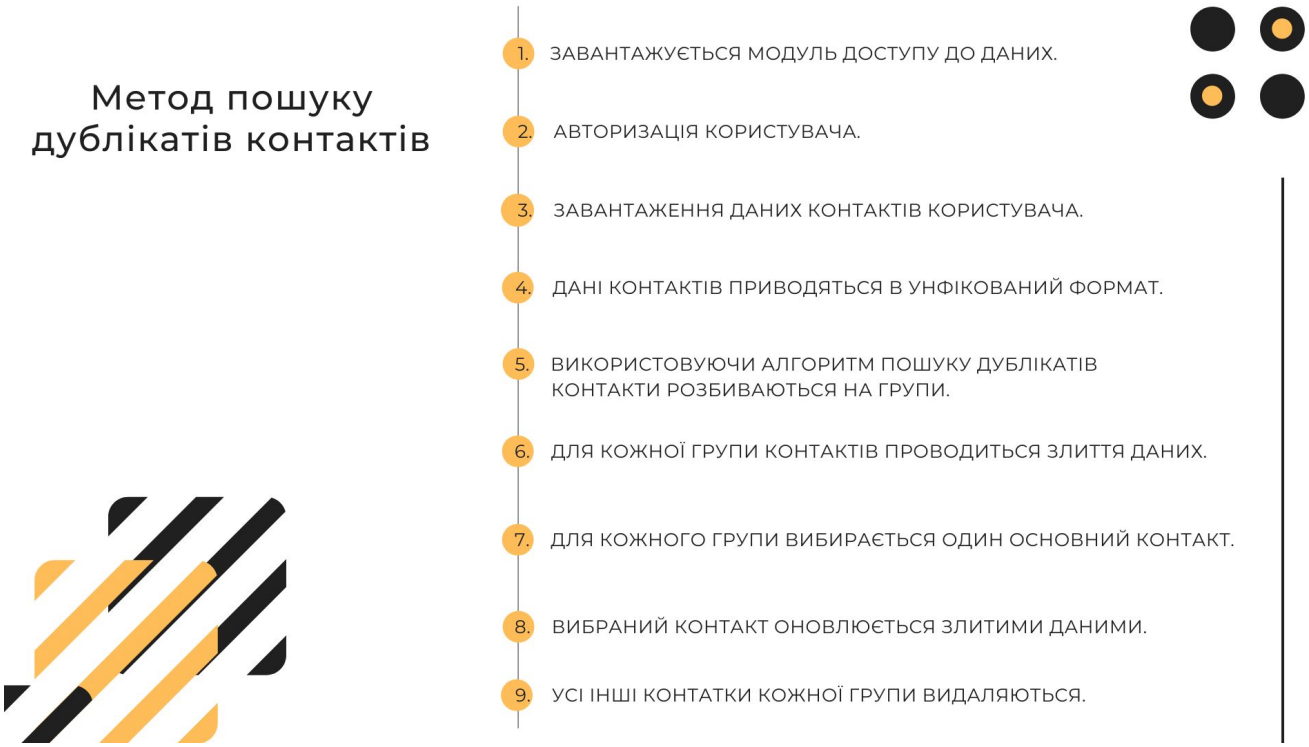


Рисунок Д.5 – Метод пошуку дублікатів контактів

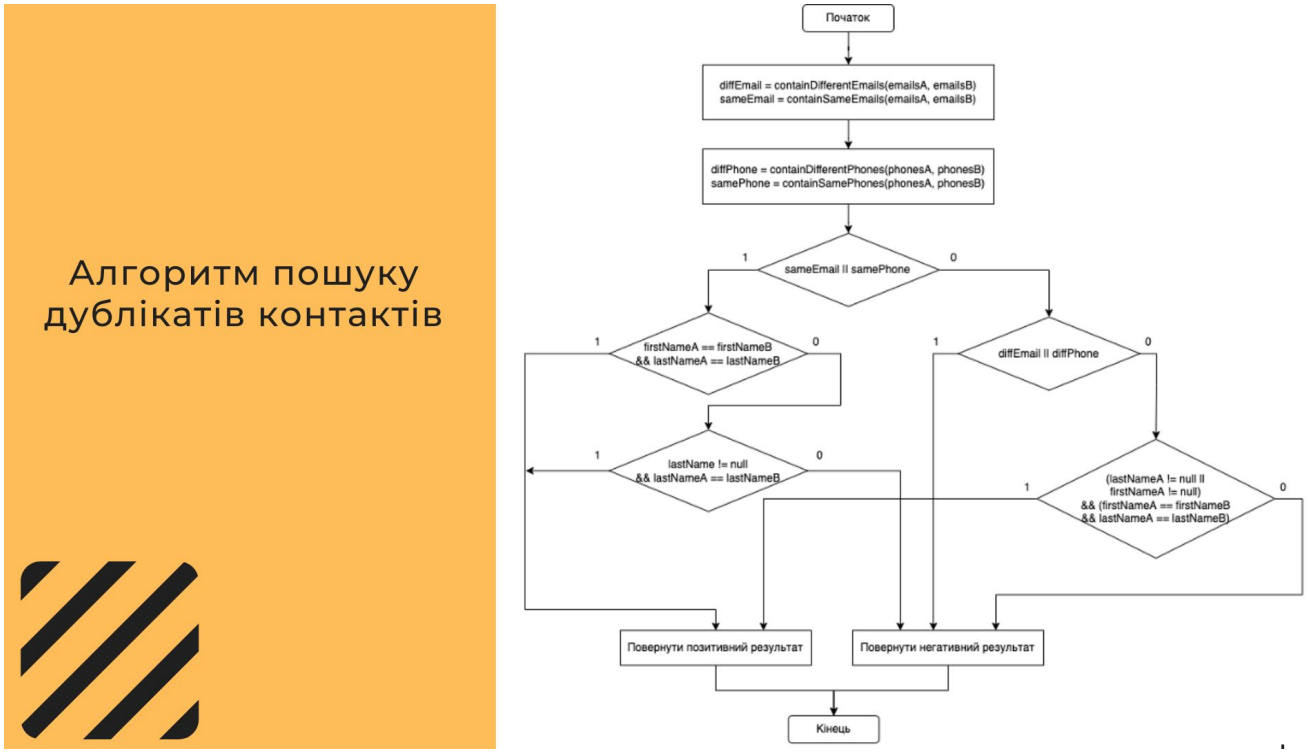


Рисунок Д.6 – Алгоритм пошуку дублікатів контактів

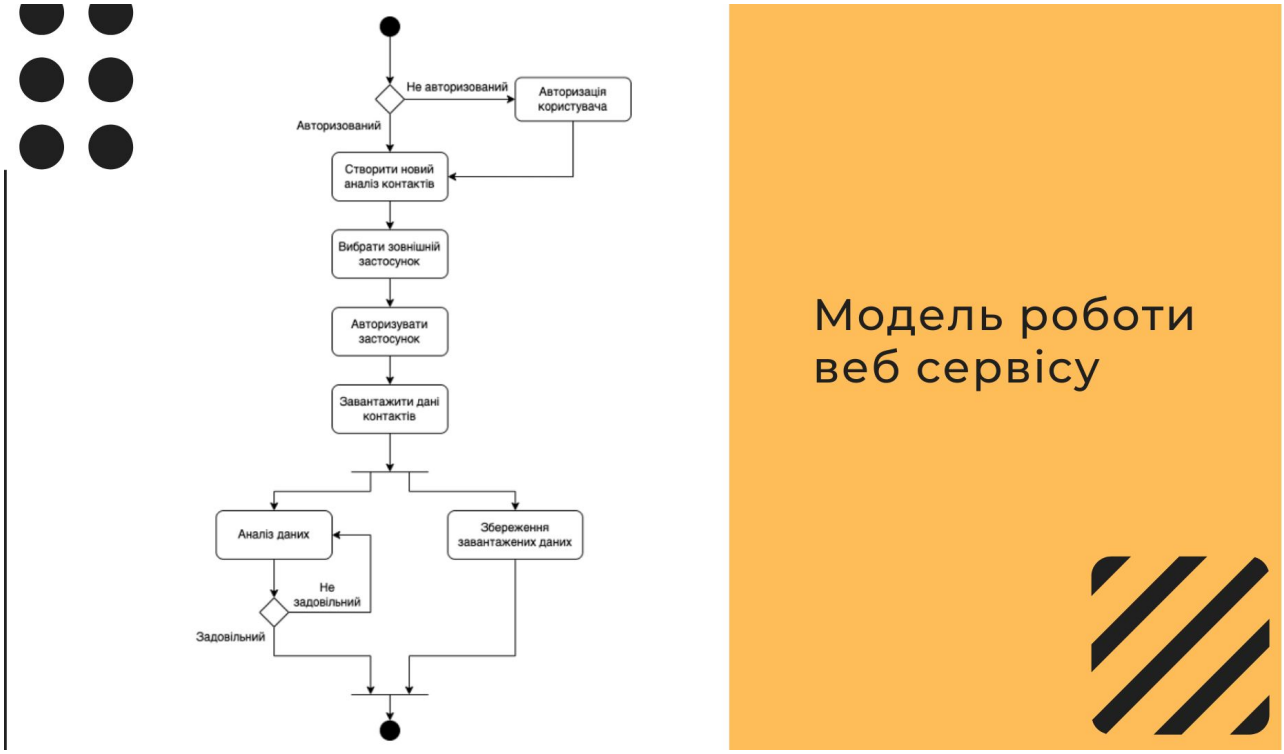


Рисунок Д.7 – Модель роботи веб сервісу


Рисунок Д.8 – Модель роботи веб сервісу

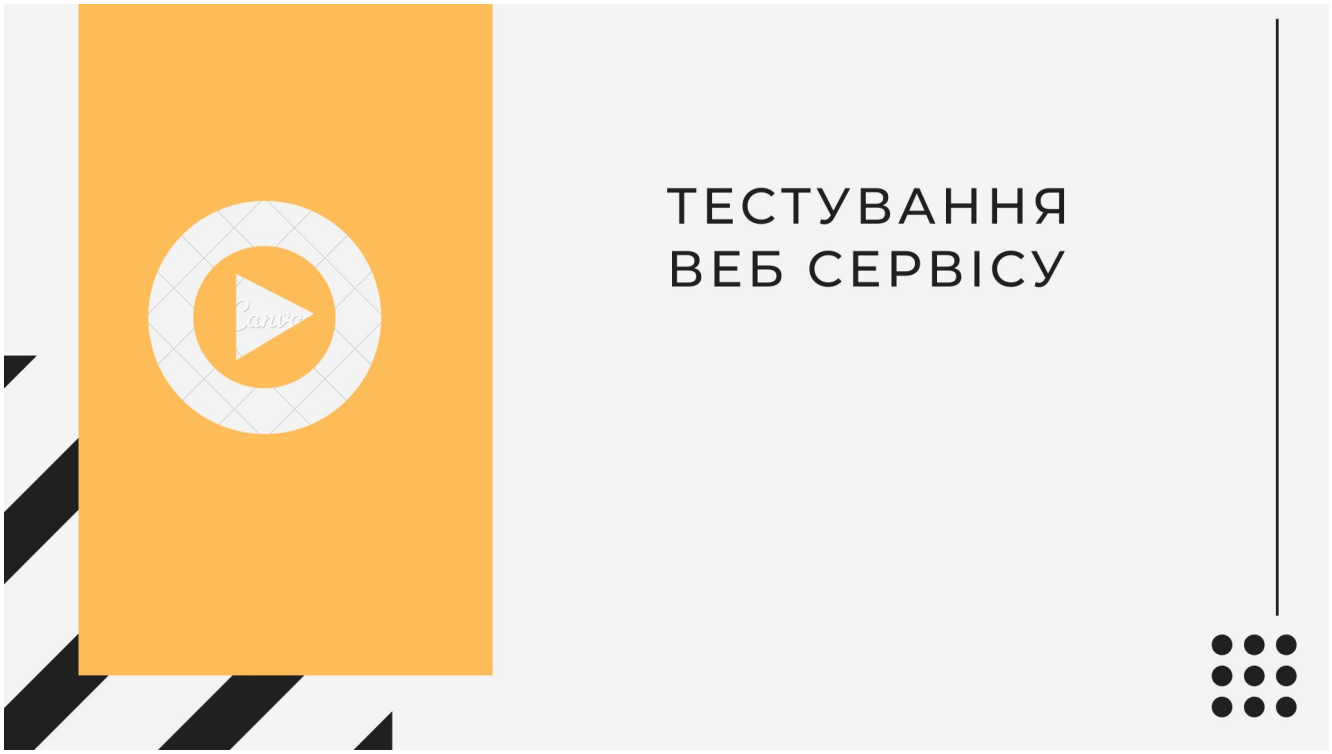


Рисунок Д.9 – Тестування веб сервісу

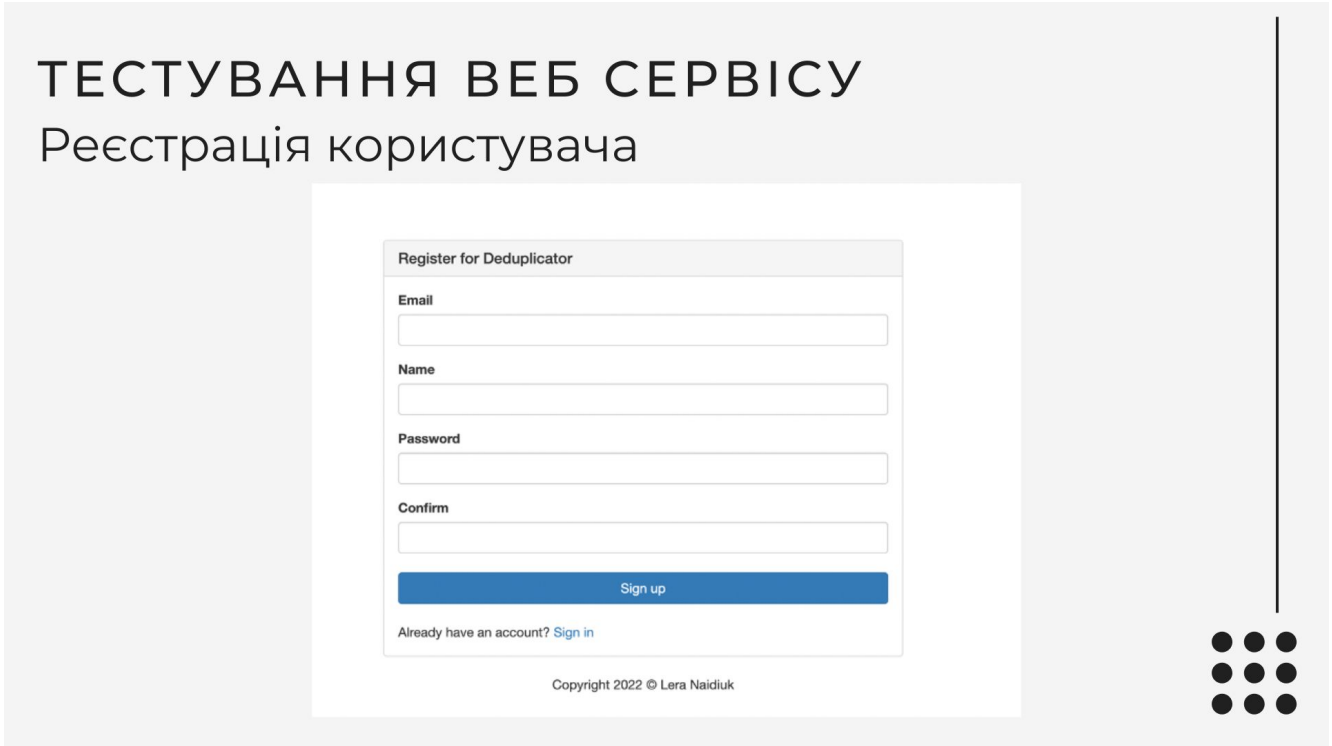


Рисунок Д.10 – Реєстрація користувача

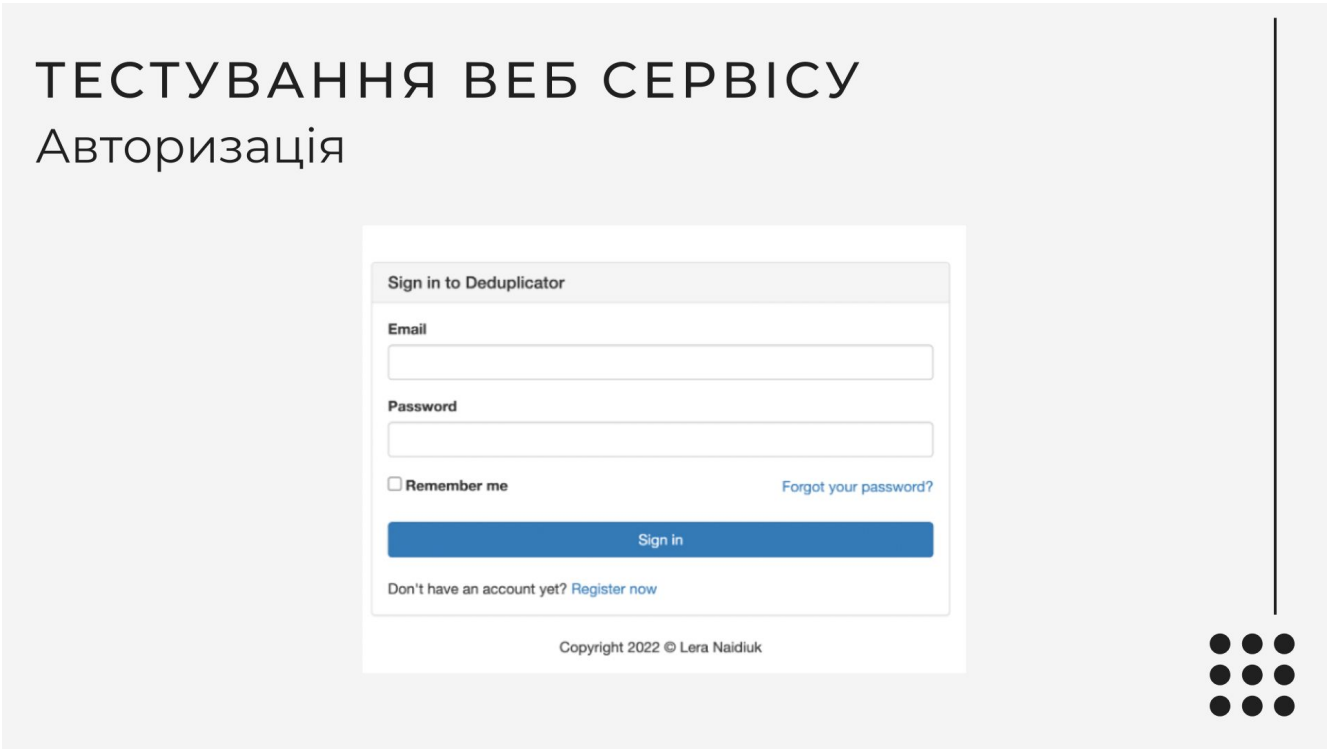


Рисунок Д.11 – Авторизація

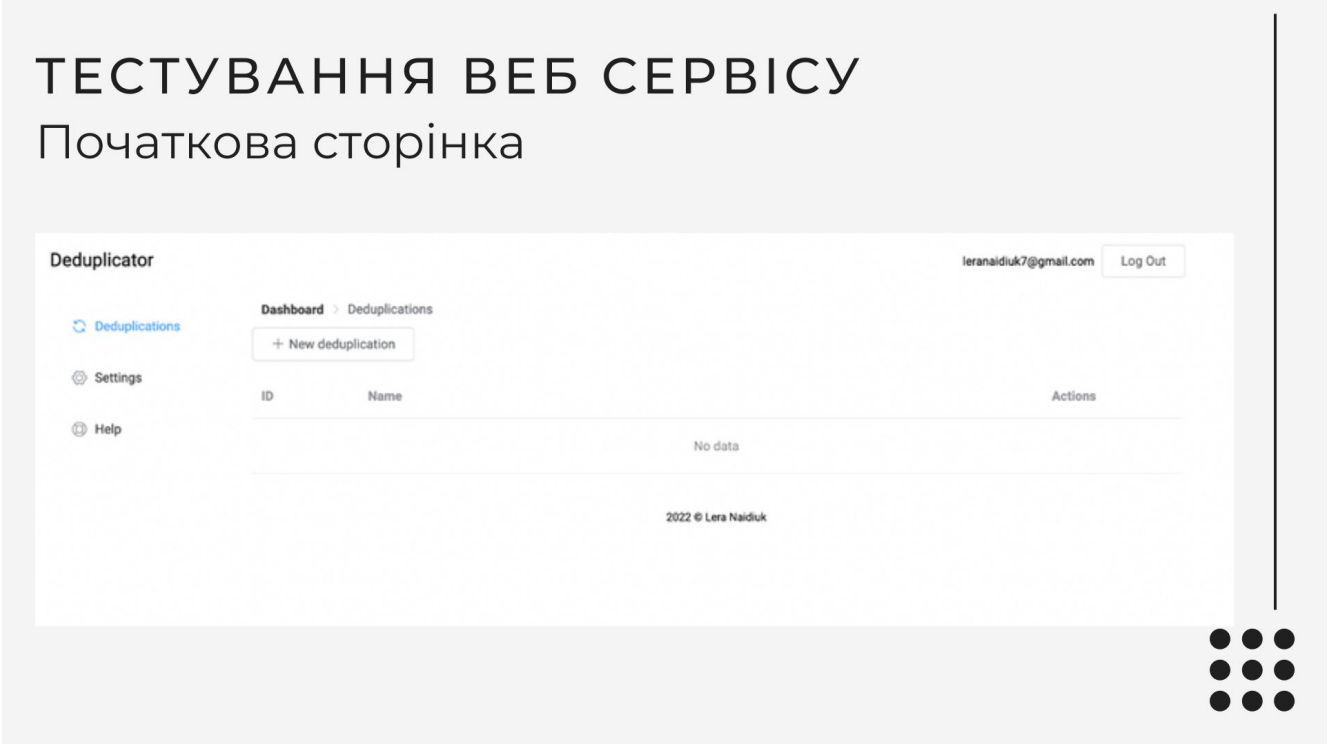


Рисунок Д.12 – Початкова сторінка

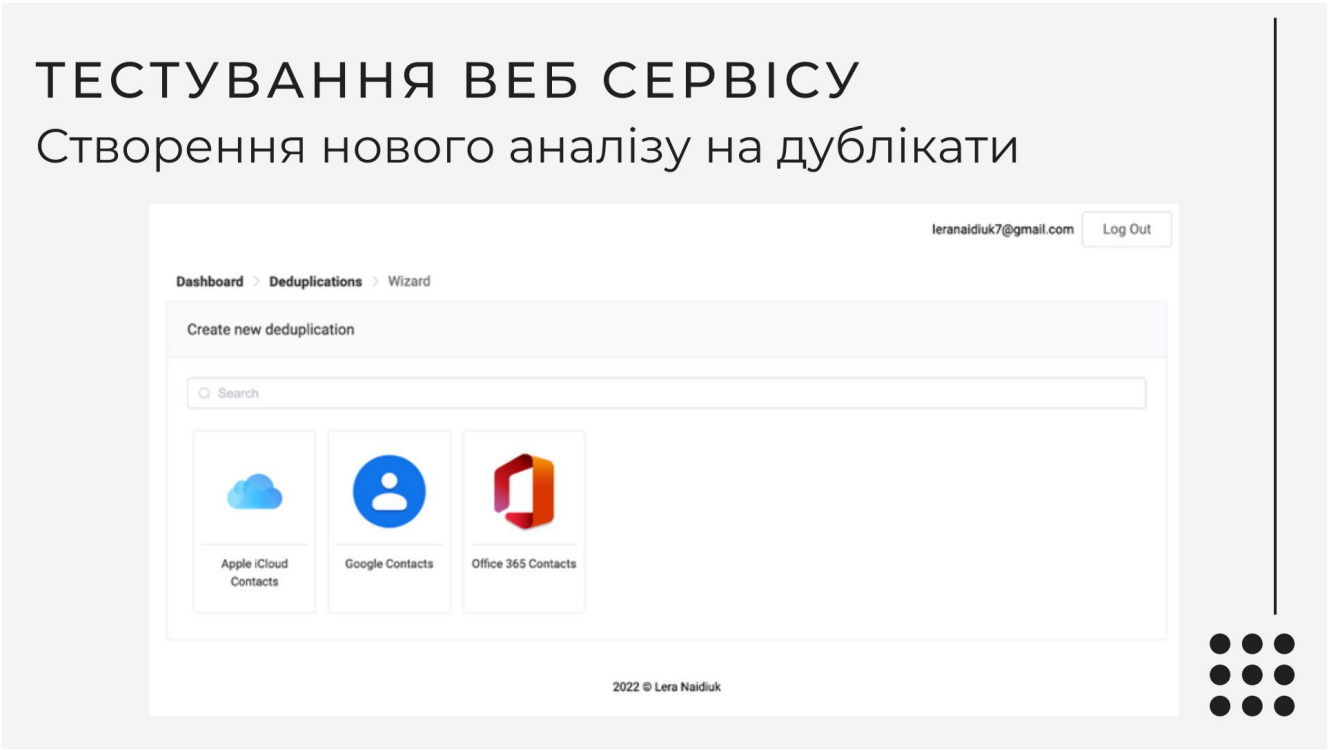


Рисунок Д.13 – Створення аналізу

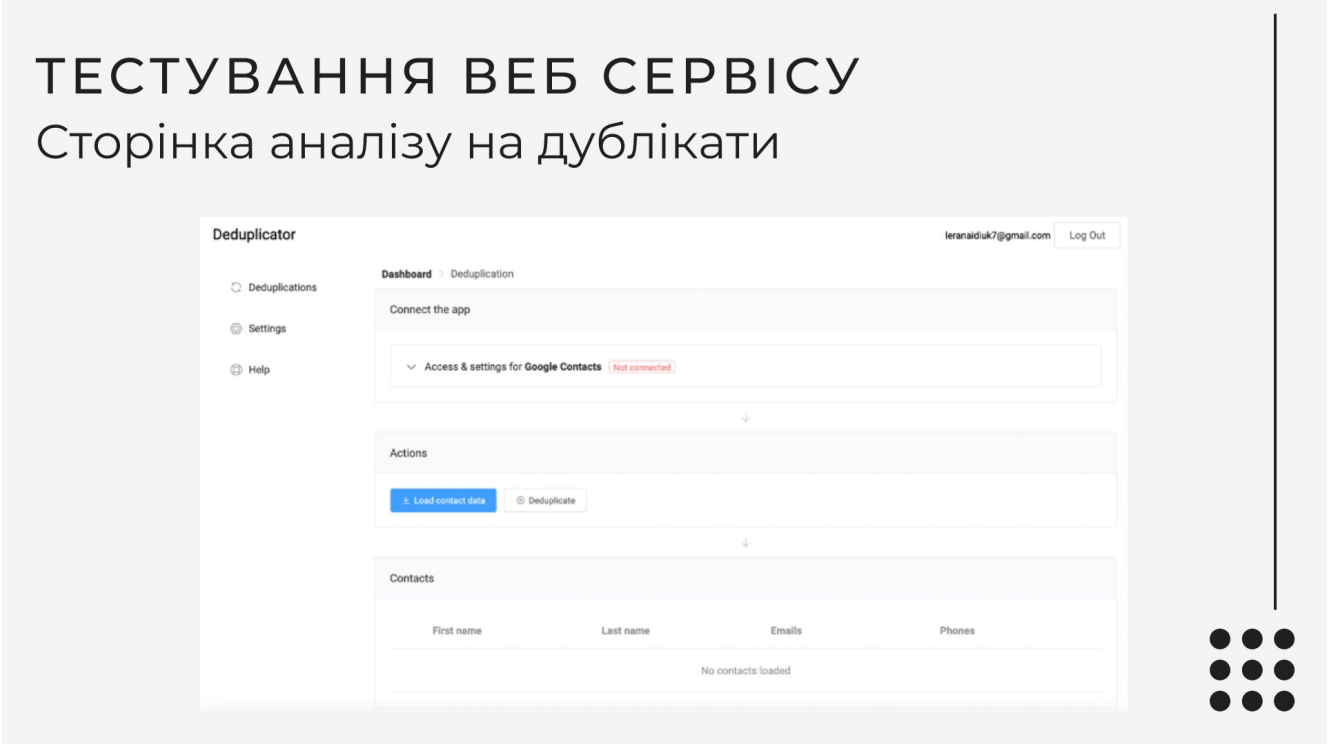


Рисунок Д.14 – Сторінка створеного аналізу

ТЕСТУВАННЯ ВЕБ СЕРВІСУ

Авторизація доступу до даних контактів

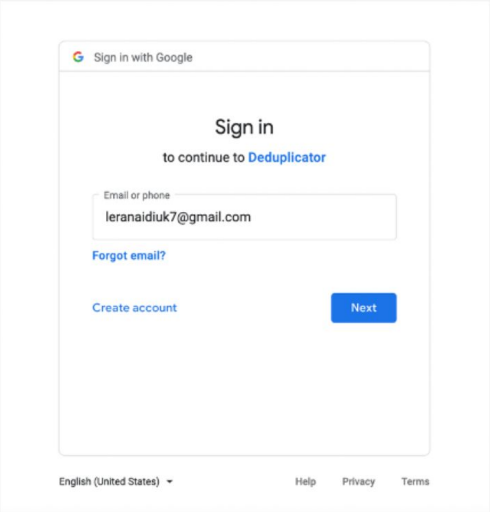


Рисунок Д.15 – Авторизація даних контактів

ТЕСТУВАННЯ ВЕБ СЕРВІСУ

Успішна авторизація

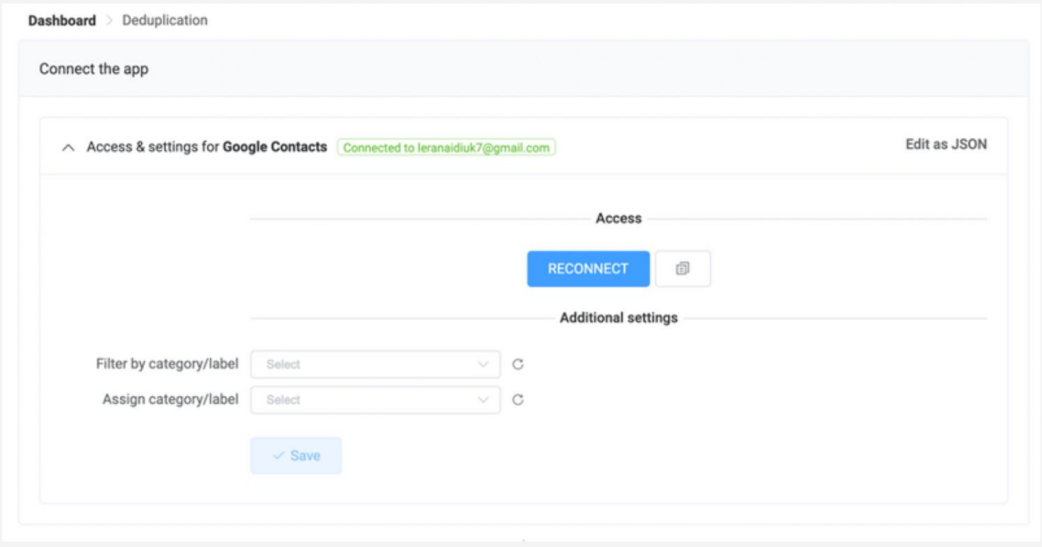


Рисунок Д.16 – Успішна авторизація

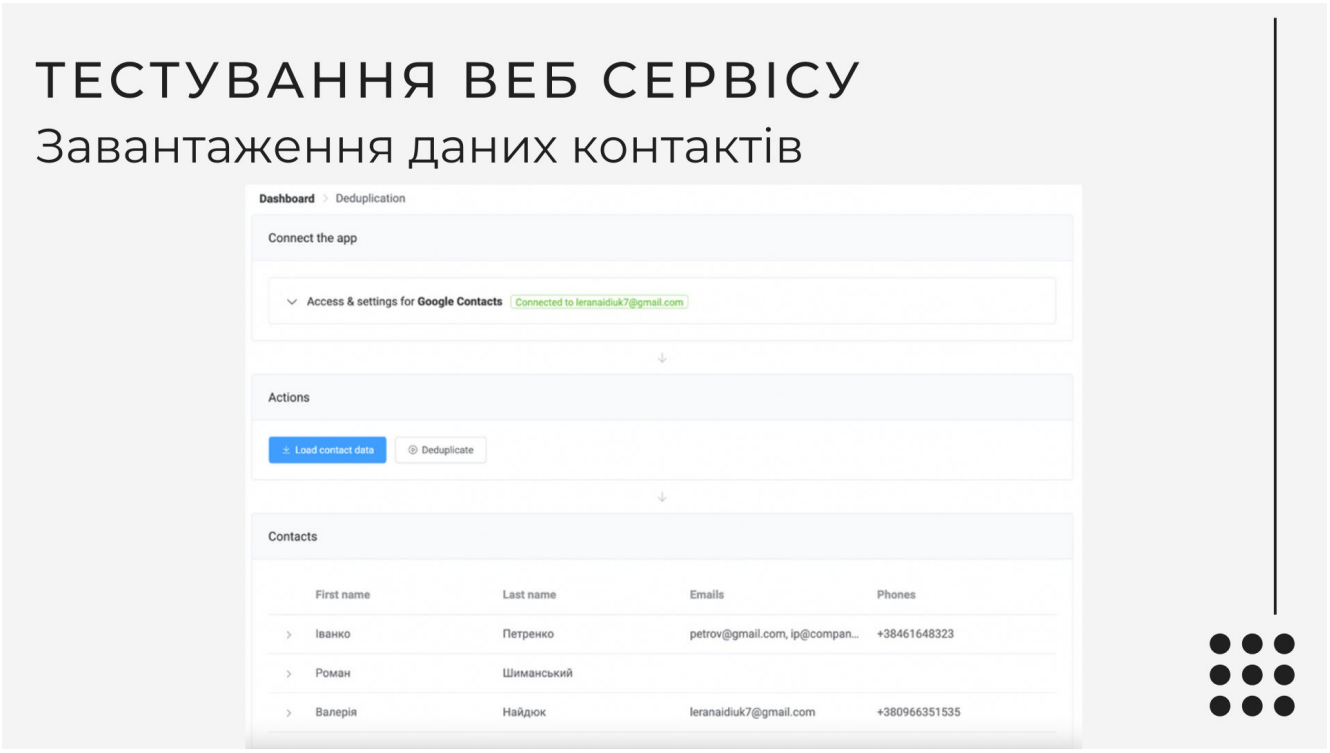


Рисунок Д.17 – Завантаження даних контактів

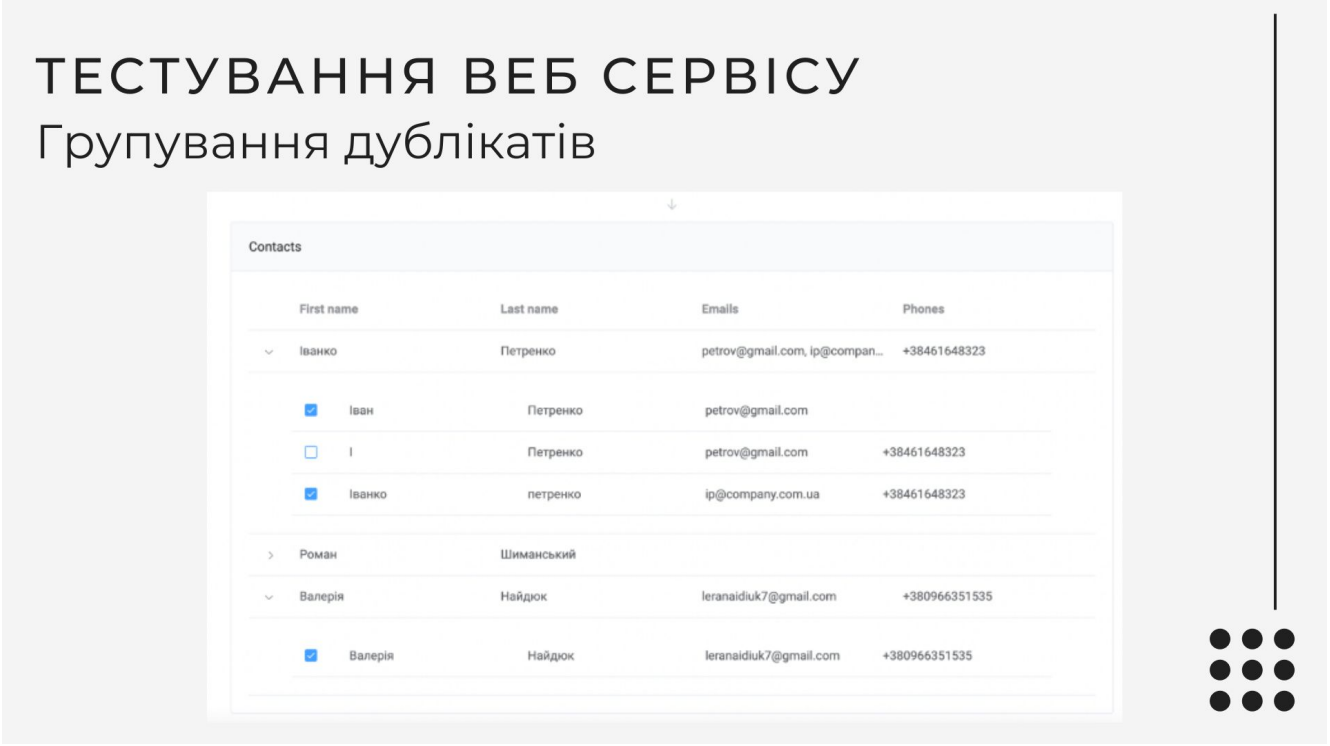


Рисунок Д.18 – Групування дублікатів

ТЕСТУВАННЯ ВЕБ СЕРВІСУ

Видалення дублікатів контактів

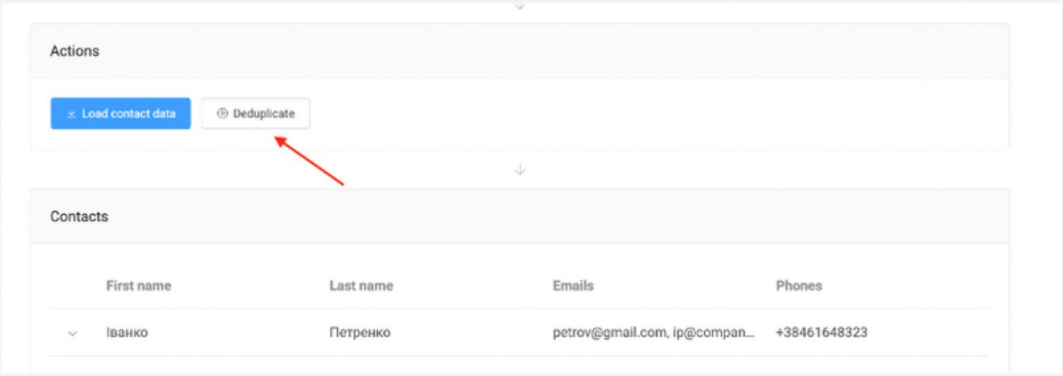


Рисунок Д.19 – Видалення дублікатів

ТЕСТУВАННЯ ВЕБ СЕРВІСУ

Видалення аналізу


Рисунок Д.20 – Видалення аналізу

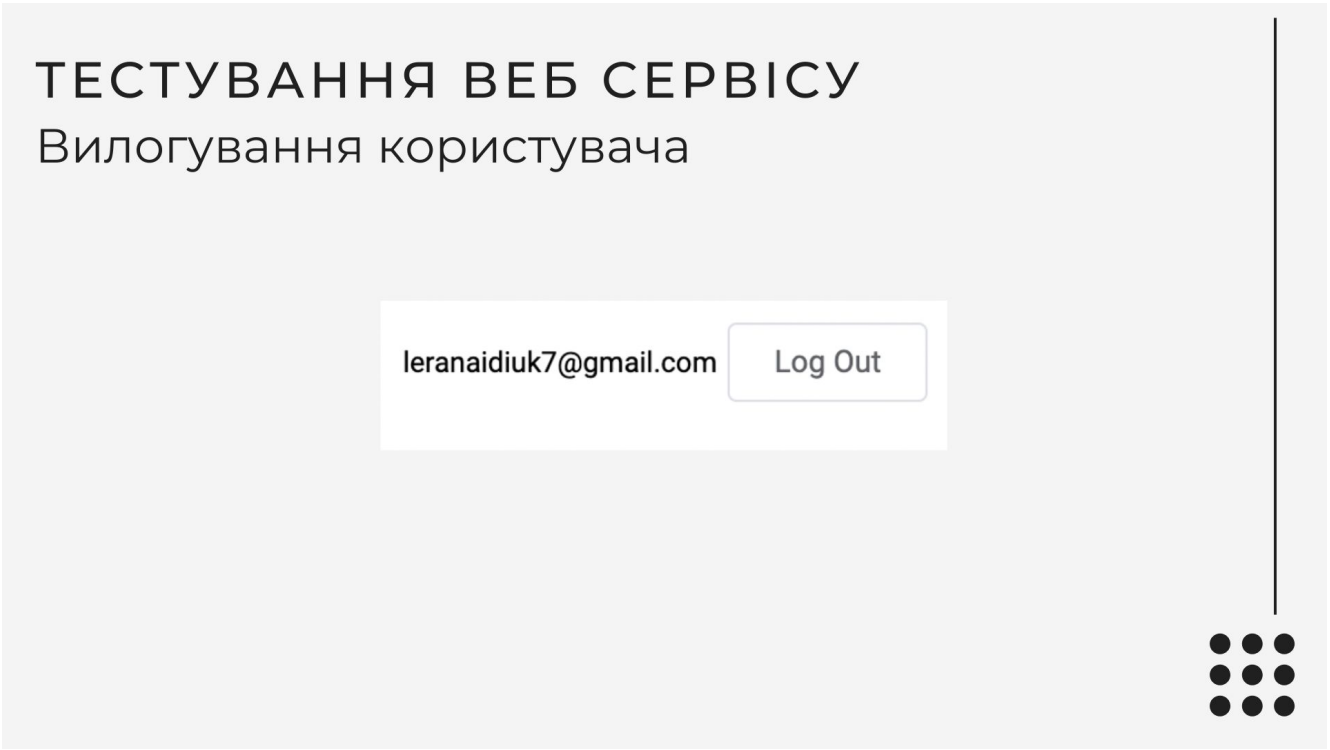


Рисунок Д.21 – Вилогування користувача


Рисунок Д.22 – Висновки

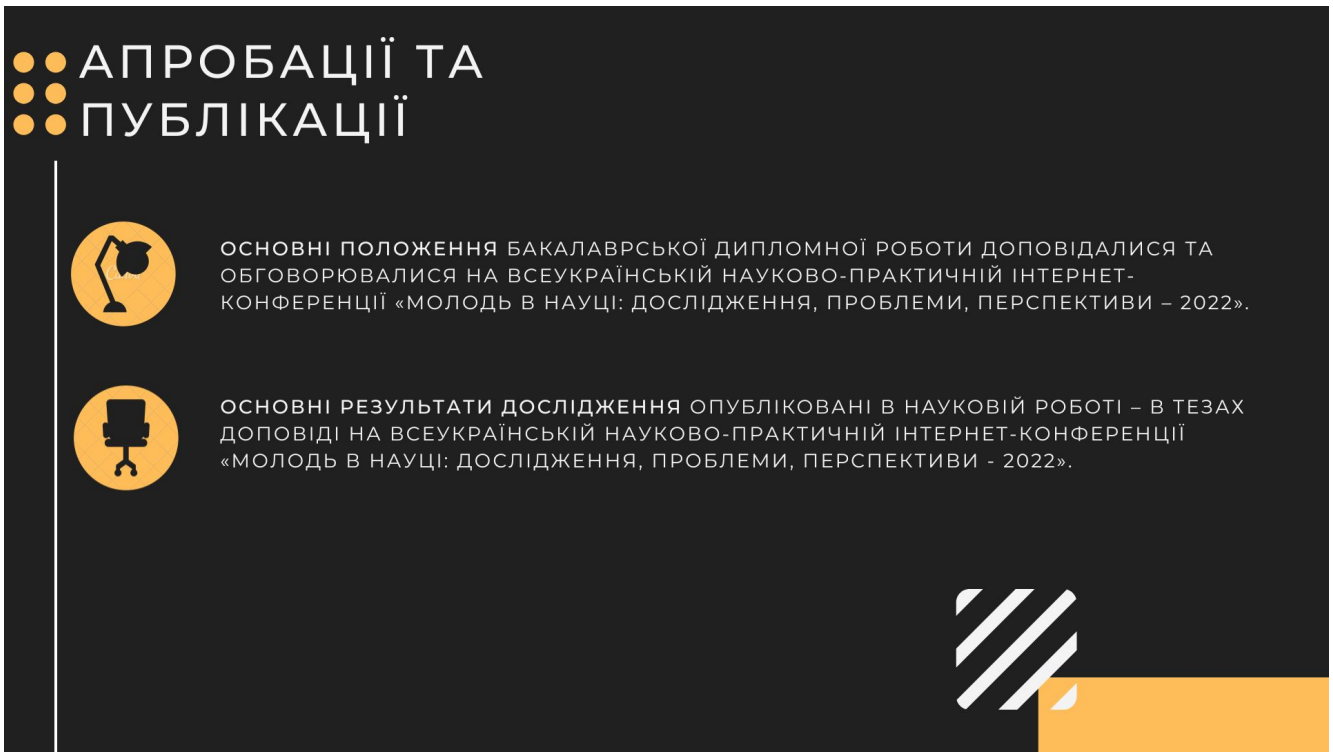


Рисунок Д.23 – Апробації та публікації