Міністерство освіти і науки України

Харківський національний університет імені В.Н. Каразіна

Факультет комп'ютерних наук

Спеціальність 125 «Кібербезпека»

Освітня програма «Безпека інформаційних та комунікаційних систем»

«Допущено до захисту»

В.о. зав. кафедрою БІСТ

Ольга МЕЛКОЗЬОРОВА

_____

«  »  2023 р.

**Пояснювальна записка**

до кваліфікаційної роботи магістра

на тему «Дослідження та розробка методів шифрування з можливістю пошуку у базах даних»

оцінка  «  »

Голова ЕК

Олександр ЛЕМЕШКО_____

Керівник д.т.н.  професор, Єсін В.І.

Рецензент д.т.н. професор, Доля Г.М.

Виконавець: студент групи КБ-61

Лілікович Сергій Олександрович

Харків – 2023

# РЕФЕРАТ

Пояснювальна записка містить 60 сторінок, 4 розділи, 33 рисунки, 4 таблиці, 1 додаток, 17 джерел.

Дослідження присвячене питанням пошуку даних за префіксом в зашифрованих базах даних.

Об’єктом дослідження є процес пошуку в криптографічно захищених базах даних.

Предметом дослідження є розробка методу пошуку за префіксом в криптографічно захищених базах даних.

Метою дослідження є розробка методу, що дозволить виконувати пошук в криптографічно захищених базах даних за префіксом, без надмірного розкриття інформації про зміст пошукового запиту та даних, що зберігаються.

Основними завданнями дослідження є аналіз існуючих підходів до пошуку в зашифрованих базах даних, розробка методу пошуку за префіксом, розробка прототипу програмного забезпечення, що реалізує запропонований метод.

Проблема пошуку за зашифрованими даними викликала інтерес у наукових колах і в індустрії. Попри велику різноманітність існуючих варіантів, не існує домінуючого рішення або набору методів. Користувачі повинні розуміти характеристики системи та компроміси для свого варіанта використання. Напрямок залишається плідною областю досліджень, розробки підходів та методів. Пропонований метод пошуку є одним із таких методів.

Ключові слова: БАЗА ДАНИХ, КОНФІДЕНЦІЙНІСТЬ, ШИФРУВАННЯ З МОЖЛИВІСТЮ ПОШУКУ, МЕТОД ПОШУКУ ЗА ПРЕФІКСОМ.

# ABSTRACT

The explanatory note contains 60 pages, 4 sections, 33 figures, 4 tables, 1 appendix, 17 references.

The research is dedicated to the issue of searching for data by prefix in encrypted databases.

The research object is the process of searching in cryptographically protected databases.

The research subject is the development of a method for prefix-based search in cryptographically protected databases.

The aim of the research is to develop a method that allows performing searches in cryptographically protected databases by prefix without excessive disclosure of information about the content of the search query and stored data.

The main tasks of the research include the analysis of existing approaches to searching in encrypted databases, the development of a prefix-based search method, and the creation of a prototype software implementing the proposed method.

The challenge of searching encrypted data has attracted interest in both scientific and industrial circles. Despite the wide variety of existing solutions, there is no dominant solution or set of methods. Users must understand the characteristics of the system and compromises for their specific use case. The direction remains a fruitful area of research, development of approaches, and methods. The proposed search method is one such approach.

Keywords: DATABASE, CONFIDENTIALITY, SEARCHABLE ENCRYPTION, PREFIX-BASED SEARCH METHOD.

ЗМІСТ

# ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ ТА СИМВОЛІВ

БД — База Даних,
ПЗ — Програмне Забезпечення,
ПС — Програмна Система,
СКБД — Система Керування Базами Даних,
API — Application Programming Interface,
DFD — Data Flow Diagram,
DO — Data Owner,
FHE — Fully Homomorphic Encryption,
GUI — Graphical User Interface,
HE — Homomorphic Encryption,
IDEF — Icam DEFinition for function modeling,
IND-CKA — Indistinguishability Against Adaptive Chosen Keyword Attack,
IND-CPA — Indistinguishability Against Chosen Plaintext Attack,
OPE — Order Preserving Encryption,
SDLC — Software Development LifeCycle,
SE — Searchable Encryption,
SQL — Structured Query Language,
SSE — Symmetric Searchable Encryption,
UML — Unified Modeling Language.

# ВСТУП

Сьогодні зі зростанням популярності аутсорсингу даних, багато користувачів так чи інакше розміщують свої дані на сторонніх серверах. Це створює проблеми, що пов’язані із забезпеченням безпеки даних, відомим розв’язанням яких є шифрування. Однак використання шифрування даних перешкоджає виконанню операції пошуку.

Через це в системах зазвичай передбачається компонент, який має доступ до ключової інформації, і здатен розшифрувати дані для виконання пошуку. В сучасних СКБД (Системах Керування Базами Даних) є вбудовані механізми, що працюють за таким принципом, проте їх використання має суттєвий недолік – розкриття захищеної інформації на стороні сервера баз даних.

Одним із підходів до розв’язання цієї проблеми є шифрування з можливістю пошуку (SE – Searchable Encryption), яке дозволяє виконувати пошук без необхідності розкриття інформації що зберігається. Ці рішення охоплюють широкий спектр криптографічних методів, наприклад: симетричне шифрування з можливістю пошуку (SSE – Symmetric Searchable Encryption), шифрування зі збереженням певних властивостей відкритого тексту в шифротексті (наприклад, зі збереженням порядку сортування – OPE), гомоморфне шифрування (HE – Homomorphic Encryption), та інших. Разом із тим, можна говорити про те, що домінуючого рішення досі не існує.

Найбільш широко використаним в застосунках підходом до забезпечення пошуку в зашифрованих базах даних є використання SSE. Цей метод реалізований у відкритому та комерційному програмному забезпеченні, втім відомим недоліком його використання є високі накладні витрати. Разом із тим відомо, що гомоморфне шифрування є ще більш ресурсомісткою операцією, і тому на практиці не використовується широко. Щодо

шифрування зі збереженням властивостей відмічається, що такий компроміс з боку безпеки часто може бути небажаним.

При цьому, коли говорять про методи SE, зазвичай розглядають пошук за ключовими словами за змістом відносно великих записів (які в термінах SE називають “документи”). На сьогодні не відомі методи SE, що були б спеціалізовані тільки на пошуці за префіксом документу – поширеній в сучасних програмних системах функціональній вимозі. Наприклад, такий пошук необхідний в ході предиктивного введення номерів телефонів або документів, фамілій тощо (тобто такого введення, при якій програмне забезпечення в процесі набору тексту пропонує варіанти закінчення слів, базуючись на змісті БД). В окремих варіантах застосування пошук за префіксом набуває фізичного сенсу – десяткові координати (де кожен наступний розряд звужує регіон пошуку), медичні показники (де пошук за префіксом може дозволити неявно обрати, наприклад, хворих осіб).

В роботі запропоновано новий метод пошуку по зашифрованим базам даних, показані алгоритми пошуку та додавання об’єкта до такого дерева, розкриті питання розробки програмного забезпечення, що використовує метод. Завдяки тому, що розглядається окремий випадок пошуку за префіксом, методом передбачено використання спеціалізованої структури даних – префіксного дерева [1, 2]. На основі отриманих результатів розроблено прототип програмного забезпечення методу що пропонується. Представляється, що цей підхід має перспективу практичного використання і заслуговує подальших досліджень його можливостей.

# 1 АНАЛІЗ СУЧАСНИХ МЕТОДІВ ПОШУКУ В ЗАШИФРОВАНИХ БАЗАХ ДАНИХ

## 1.1 Основні поняття та визначення

Сучасні широко використані СКБД, такі як PostgreSQL, MySQL, Oracle Database надають користувачам різні опції шифрування для захисту даних – шифрування рівня файлової системи, таблиць, колонок, криптографічні функції в запитах SQL (Structured Query Language) тощо. Такі власні засоби захисту даних на стороні СКБД мають суттєвий недолік – для виконання типових запитів, зокрема пошуку, СКБД повинна мати змогу розшифрувати дані, тобто мати доступ до ключової інформації. Таким чином сервер баз даних, а отже і адміністратор цього серверу завжди мають доступ до всіх даних що зберігаються. Типову діаграму розгортання застосунку з "тонким" клієнтом, що може використовувати власні механізми криптографічного захисту даних наведено на рисунку 1.1:

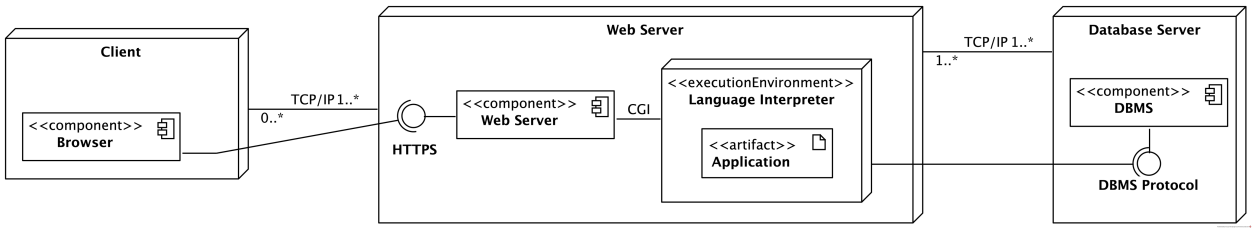


Рисунок 1.1 – Типова діаграма розгортання застосунку, що використовує криптографічний захист даних засобами СКБД

В цьому випадку захист інформації виконується вузлом "Database Server" за допомогою вбудованих в СКБД засобів. Для виконання пошуку СКБД повинна мати можливість розшифрувати дані які вона зберігає і виконати пошук за текстом запиту.

Для розв'язання цієї проблеми у 2000 році був запропонований метод симетричного шифрування з можливістю пошуку [3], який полягає в створенні окремого пошукового індексу, який забезпечує можливість виконання пошуку без розшифрування ані даних що зберігаються, ані самого індексу. В 2003 році

в роботі [4] автором була формально визначена модель захищеного індексу – структури даних, яка дозволяє користувачу за допомогою спеціального набору даних, так званої “лазівки” (“trapdoor”), для слова $x$ перевірити, чи міститься $x$ в індексі за час $O(1)$. При цьому індекс не розкриває жодної інформації про свій вміст без відповідних “лазівок”, які можуть бути згенеровані лише за допомогою секретного ключа. Ці роботи лежать в основі систем шифрування з можливістю пошуку – систем, за допомогою яких група користувачів може надавати дані віддаленим серверам для зберігання, обмінюватися та отримувати релевантні дані з них [5], не розкриваючи при цьому зміст документів що зберігаються, та не розкриваючи свої пошукові запити.

Також в роботі [4] і пізніше в [6] були формалізовані перші моделі безпеки для шифрування з можливістю пошуку:

– IND-CPA або «семантично захищений від атаки за вибраним відкритим текстом» (Indistinguishability Against Chosen Plaintext Attack) модель вимагає, щоб зашифровані дані не розкривали жодної інформації про відкриті тексти, навіть якщо зловмисник може обирати ключові слова [3]. Проте в шифруванні з можливістю пошуку витоки зазвичай відбуваються не з шифротекстів, а з пошукових запитів, тому така модель зараз вважається слабкою.

– IND1-CKA (Indistinguishability Against Adaptive Chosen Keyword Attack) або «семантично захищений від атаки за адаптивним вибраним ключовим словом» модель вимагає, щоб індекс не розкривав жодної інформації про документи, для яких він побудований, навіть якщо зловмисник може вибирати ключові слова для пошуку таким чином, щоб отримати максимум інформації [4]. Пізніше була запропонована адаптивна модель IND2-CKA, що розширює IND1-CKA в тому сенсі, що обмеження щодо безпеки документів розповсюджуються і на випадок, коли зловмисник може сам створити індекси та “лазівки” для довільно обраних документів і ключових слів, тобто має доступ до результатів пошуку відносно довільних запитів, що

потенційно дозволяє йому отримати знання про те, як індекс був побудований і як він реагує на ті чи інші запити.

– Іншою широко використаною моделлю є модель «захищений від атаки відомого шифротексту» [7], що вимагає, щоб зловмисник, отримавши доступ до шифротекстів пошукового індексу та сховища документів, не зміг отримати жодної інформації про документи.

Слід зазначити, що моделі безпеки формалізовані в контексті симетричного шифрування з можливістю пошуку, проте вони є справедливими і для методів, що використовують асиметричне шифрування, де додатково до секретного ключа з'являється відкритий ключ, який і використовується для генерування "лазівок".

В роботі [7] формалізовані метрики, що характеризують методи пошуку в зашифрованих БД. Найважливішими з них є:

– Ефективність пошуку. Цей критерій включає не тільки швидкість, а й час необхідний на побудову пошукового індексу та запиту, точність пошуку тощо.

– Ефективність ранжування результатів пошуку.

– Накладні витрати пам'яті на забезпечення пошуку, тобто на зберігання пошукового індексу.

– Розмір запиту. Метрика вимірює кількість бітів, що репрезентують пошуковий запит. Задля приховування довжини пошукового запиту деякі методи використовують фіксовану довжину запиту. Крім того, сюди враховуються можливі накладні витрати, що з'являються в ході перетворення запиту в "лазівку" навіть якщо довжина запиту не є фіксованою.

Окремим методом є гомоморфне шифрування – особливий тип шифрування, що дозволяє виконувати операції алгебри прямо над шифротекстом, без розшифрування [8]. Розрізняють повністю (FHE – Fully Homomorphic Encryption) та частково гомоморфні криптосистеми, відміна яких полягає в обсязі підтриманих операцій. Вважається, що FHE може вирішити проблему пошуку в зашифрованих даних, проте на практиці

повністю гомоморфні криптосистеми є вибагливими до обчислювальних ресурсів, і досі не було запропоновано жодної ефективної з цієї точки зору реалізації FHE.

Таким чином, в реальних застосунках найбільш доступними вважаються реалізації різних схем симетричного шифрування з можливістю пошуку, та схеми з відкритим ключем. Нижче узагальнено розглянуті ці два основних підходи.

## 1.2 Симетричне шифрування з можливістю пошуку

В схемах пошуку з використанням симетричного шифрування один секретний ключ використовується для захисту пошукового індексу і для генерування “лазівок”. Головні переваги такого підходу – простота реалізації, невибагливість до обчислювальних ресурсів. Процес пошуку з використанням SSE у вигляді IDEF0 (Icam DEFinition for function modeling) діаграми показаний на рисунку 1.2:

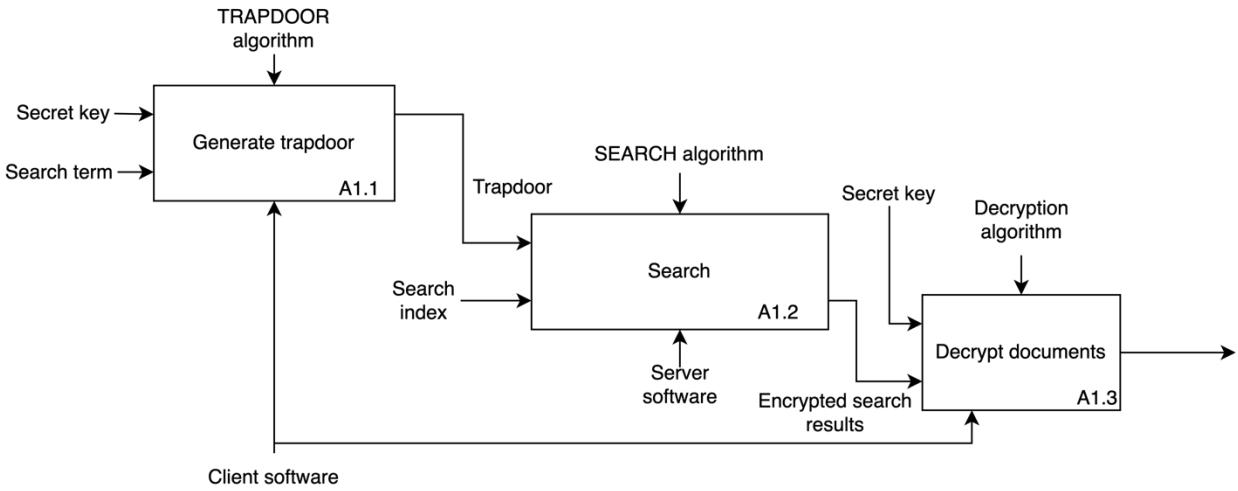


Рисунок 1.2 – Процес пошуку з використанням SSE

На рисунку 1.2 видно, що для виконання пошуку клієнтське програмне забезпечення створює “лазівку” на основі пошукового запиту (search term), секретного ключа (secret key) та алгоритму TRAPDOOR. Серверне програмне забезпечення виконує пошук без знання секретного ключа, і повертає

результати без розшифрування (encrypted search results). Останнє виконується на клієнтському боці.

Визначення алгоритмів TRAPDOOR та SEARCH узагальнено сформульовано в [7]. Перший використовується кінцевими користувачами для створення пошукового запиту що відповідає ключовим словам, на основі секретної інформації (секретного ключа). У деяких випадках запит може бути згенерований власником даних від імені кінцевих користувачів, уникаючи таким чином передачі секретного ключа користувачам.

Отримавши зашифрований пошуковий запит, сервер запускає алгоритм SEARCH, який порівнює зашифрований запит із записами в індексі, таким чином генеруючи список відповідних документів. Крім того, отримані документи можуть бути ранжовані, тобто організовані відповідно до релевантності пошукового запиту.

Алгоритм SEARCH може використовувати індекси у вигляді таблиць, як показано в таблиці 1.1:

Таблиця 1.1 – Приклад прямого індексу для алгоритму SEARCH

| Документ | Ключові слова |
|---|---|
| $D_1$ | $w_x$, …,$w_y$, $w_z$ |
| … | … |
| $D_n$ | $w_k$, …, $w_x$ |

В БД може бути побудований зворотний індекс, тобто такий, в якому кожному ключовому слову задається відповідна множина документів. Оскільки зазвичай в БД кількість записів значно більша ніж кількість ключових слів, це може прискорити процес пошуку.

## 1.3 Шифрування з відкритим ключем та можливістю пошуку

Схеми пошуку з відкритим ключем використовують пару ключів так, що публічний ключ використовується для шифрування індексу, тоді як закритий ключ використовується для створення “лазівки”. Ці схеми добре працюють у

багатокористувацьких середовищах, але є більш вибагливими до обчислювальних ресурсів. В роботі [9] представили концепцію шифрування з відкритим ключем та можливістю пошуку в хмарному середовищі. При цьому процес пошуку не відрізняється від такого, що показаний на діаграмі з рисунку 1.2.

## 1.4 Підтримка багатокористувальницького середовища

В схемах криптографічного захисту інформації з можливістю пошуку виділяють такі ролі [7]:

– Власник даних (DO – Data Owner). Особа або організація, відповідальна за створення пошукового індексу, за вибір ключових слів з документів, та за зашифрування самих документів. Користувачі з цією роллю завантажують ці дані на хмарний сервер.

– Користувач (data users) – це той хто бажає отримати доступ до розміщених даних. Користувач може. Бути власником даних (в схемах SSE), або третьою стороною, уповноваженою на доступ до даних (в схемах з асиметричним шифруванням).

– Хмарний сервер (cloud server). Ця сутність забезпечує збереження для Власника даних і засіб пошуку для Користувачів. Хмарний сервер виконує пошук потрібних документів, порівнюючи пошуковий запит із зашифрованим індексом або зашифрованими документами, і передає зашифровані результати кінцевому користувачеві. На основі отриманих результатів бажані документи завантажуються кінцевим користувачем і розшифровуються.

Деякі автори вводять необов’язковий четвертий об’єкт між Користувачем і Хмарним сервером, щоб забезпечити конфіденційність доступу або за її посередництвом виконувати автентифікацію Користувачів, генерувати пари секретних ключів. В роботі [10] запропонували використовувати окрему сторону, яка називається Файловим сервером, для зберігання документів, тоді як індекс документа зберігається на Хмарному сервері. Це забезпечує конфіденційність доступу для кінцевого користувача,

оскільки Хмарний сервер не може отримати інформацію про документи, які користувач завантажує з файлового сервера відповідно до пошукового запиту.

Таким чином, схематично застосування підходу криптографічного захисту інформації з можливістю пошуку можна зобразити як показано на рисунку 1.3:

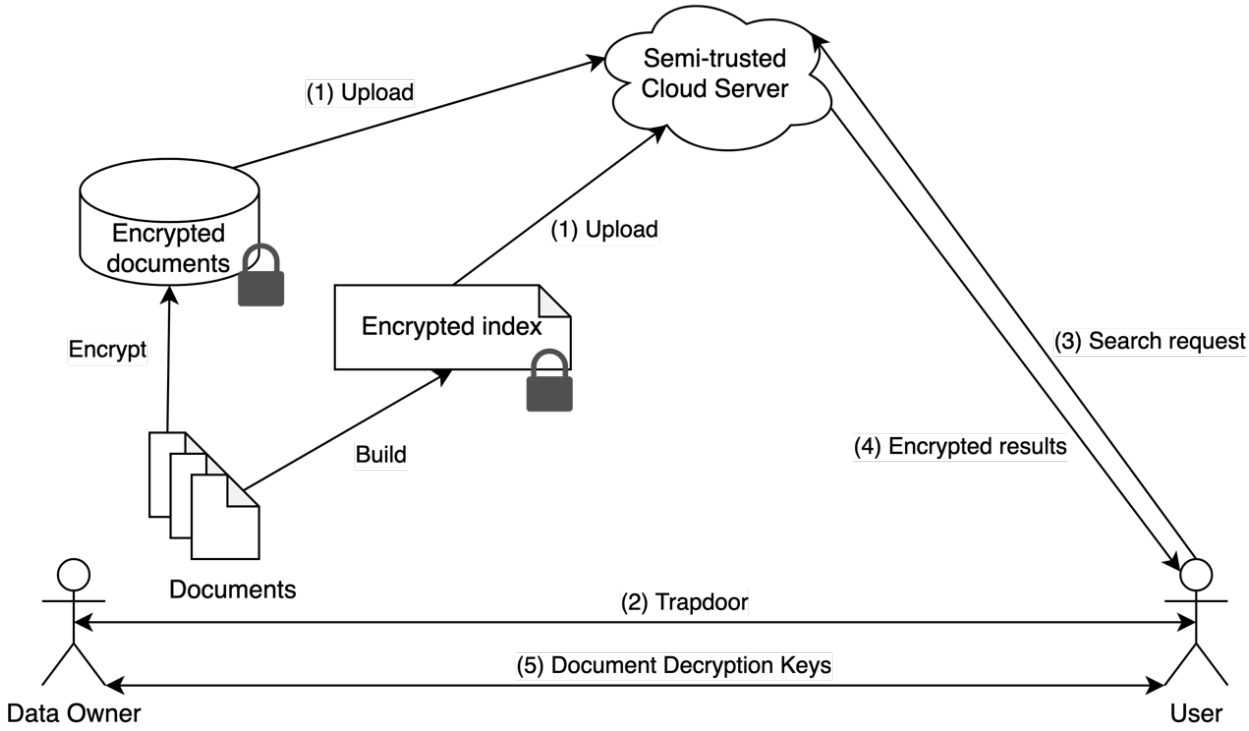


Рисунок 1.3 – Узагальнена схема застосування SE в багатокористувальницькому середовищі

На схемі з рисунку 1.3 показано, як зазначені вище типи користувачів Власник даних (data owner) та Користувач (user) взаємодіють в ході створення пошукового індексу (encrypted index) на основі документів (documents), як виконується пошуковий запит (search request) з отриманням зашифрованих результатів пошуку (encrypted results), та як виконується надання доступу на пошук за допомогою передачі “лазівки” (trapdoor) та ключів розшифрування документа (document decryption keys).

## 1.5 Висновки за розділом

В розділі розглянуті основні принципи використання шифрування з можливістю пошуку в сучасних базах даних. Показано, як застосування методів SE відрізняється від вбудованих в СКБД методів криптографічного

захисту інформації, а також показані два основних підходи до організації SE – з використанням симетричного та асиметричного шифрування. Сформульована узагальнена схема застосування методів криптографічного захисту інформації в умовах багатокористувальницького середовища.

# 2 РОЗРОБКА МЕТОДУ ПОШУКУ ЗА ПРЕФІКСОМ В ЗАШИФРОВАНИХ БАЗАХ ДАНИХ

## 2.1 Модель загроз

Відповідно до [11], процес моделювання загроз це процес активного пошуку можливих атак на ПС. Результатом процесу моделювання загроз є список ймовірних сценаріїв загроз. Підхід до моделювання загроз має розглядати систему в цілому, виконуватись починаючи з самих ранніх етапів проектування системи, і бути повторюваним. Автори виділяють три можливі підходи до моделювання загроз:

– Орієнтований на активи. Розглядає загрози, спираючись на перелік цінних даних та функцій системи, які можуть зазнавати різноманітних загроз.

– Орієнтований на зловмисника. Модель будують відштовхуючись від ймовірних цілей, засобів та можливостей, які може мати в своєму розпорядженні атакуючий.

– Орієнтований на застосунок. Модель будують спираючись на вичерпні знання про програмну систему, яку захищають.

Оскільки в даному випадку моделювання загроз виконується для системи що знаходиться на етапі проектування, доцільно використовувати саме орієнтований на застосунок підхід. Для цього підходу в [11] визначено такий процес:

– Декомпозувати застосунок, виявивши всі потоки даних, зокрема використовуючи моделювання за допомогою діаграм DFD (Data Flow Diagram).

– Перелічити загрози для кожного елемента з діаграм.

– Розставити пріоритети для загроз за допомогою певної моделі класифікації.

Таким чином, процес починається з побудови діаграм потоків даних, яка моделює зв’язки між бізнес-процесами, сховищами даних, та зовнішніми сутностями.

Зазвичай при конструюванні методів шифрування з можливістю пошуку розглядають схему, що складається з наступних вузлів:

– Сервер баз даних, який зберігає зашифровані дані користувачів. Цей сервер вважається чесним, але допитливим (honest but curious), тобто він слідує необхідним протоколам, проте в ході роботи буде намагатись отримати якомога більше відомостей про дані що зберігаються.

– Клієнтський застосунок, який може бути не тільки легітимним клієнтом, а і шкідливим (malicious), тобто такі вузли можуть активно вчиняти дії щодо отримання додаткової інформації та впливу на роботу системи.

Проте сучасне програмне забезпечення все частіше залучає мобільні пристрої, “тонкі” клієнти, тому при моделюванні загроз доцільно розглядати ще один вузол:

– Сервер застосунку, який також вважається чесним, але допитливим. Саме програмні компоненти цього вузла частіше за все потребують використання шифрування з можливістю пошуку, адже при використанні архітектурного патерну з “тонким” клієнтом вони імплементують бізнес-логіку застосунку, тоді як клієнтський застосунок використовується в якості інтерфейса користувача.

В залежності від того який вузол програмної системи буде імплементувати алгоритм TRAPDOOR, можна розглядати дві різні моделі загроз. Випадок, в якому схема шифрування з можливістю пошуку реалізована повністю на серверних вузлах показана на діаграмі DFD на рисунку 2.1:

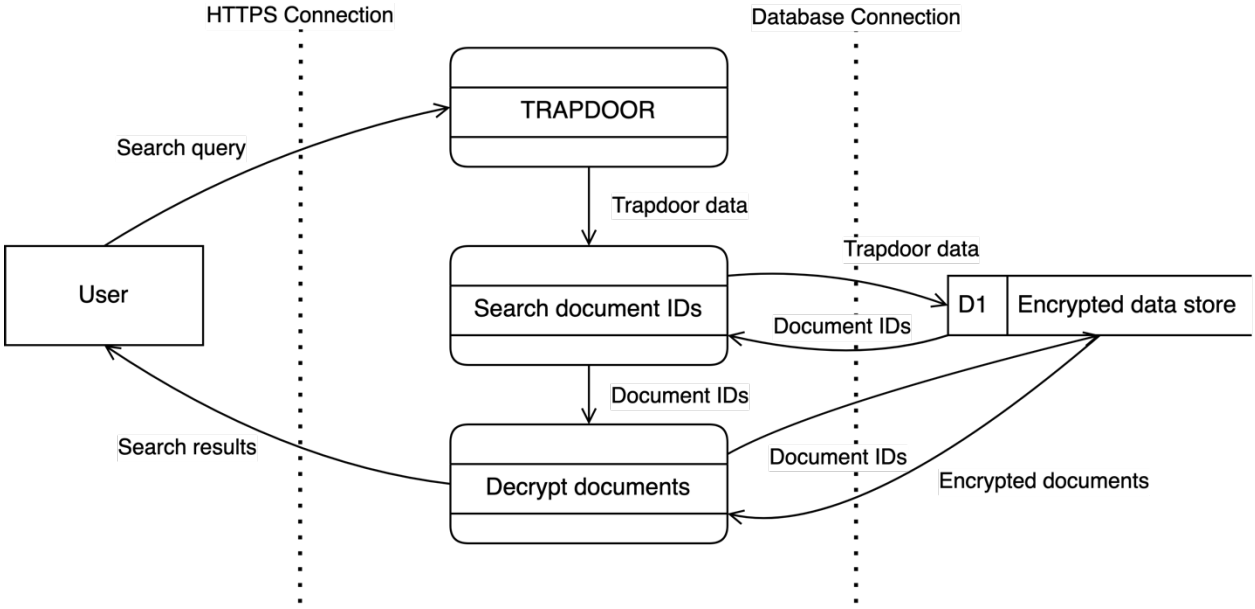


Рисунок 2.1 – Діаграма потоків даних для ПС з SE, що реалізовано на стороні серверних компонент

На рисунку 2.1 показано, як дані користувача (user) використовуються в процесі пошуку та передаються між компонентами ПС. Видно, що в такому варіанті алгоритм TRAPDOOR виконується в межах серверних компонентів, а отже секретний ключ повинен бути доступним їм. Проте клієнтське програмне забезпечення  не вимагає реалізації специфічних для SE функцій.

Очевидним недоліком такої реалізації пошуку є те, що захист даних за допомогою SE виконується лише для серверних компонент, при чому сервер застосунку має доступ до секретного ключа. Перевагою використання такої реалізації є невибагливість до обладнання та програмного забезпечення клієнта через відсутність необхідності реалізації схеми SE на його вузлі.

Проте для хмарного багатокористувальницького середовища схема SE повинна бути реалізована таким чином, що секретний ключ не може бути використаним третьою по відношенню власників та користувачів даних стороною. Для такого випадку діаграма потоків даних наведена на рисунку 2.2:

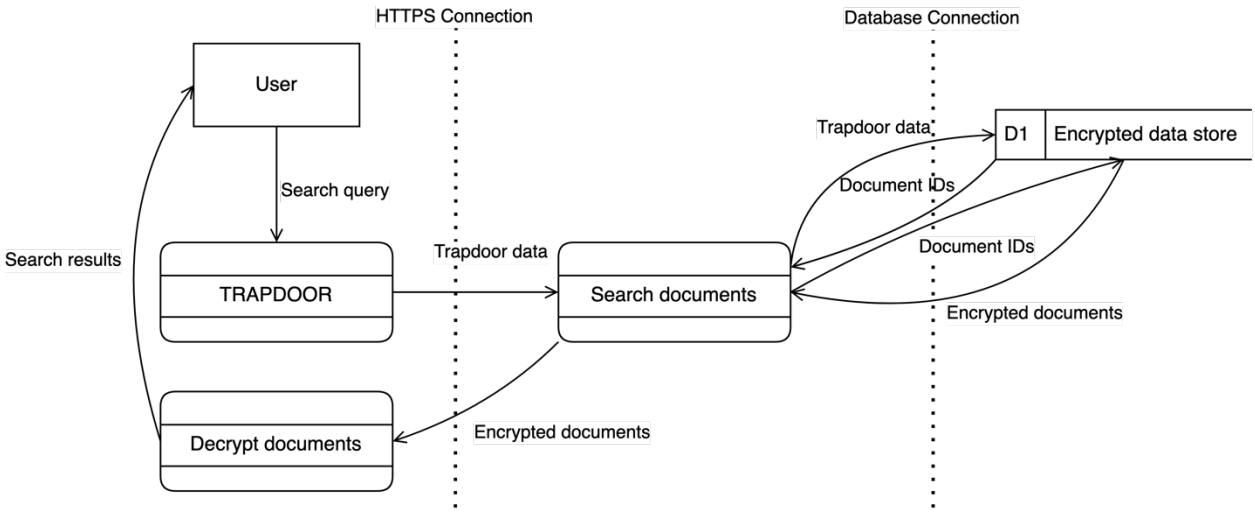


Рисунок 2.2 – Діаграма потоків даних для ПС з SE, що реалізовано на клієнтському вузлі

На рисунку 2.2 видно, що відкритий текст пошукового запиту (search query) не залишає меж клієнтських компонентів. Таким чином секретний ключ використовується лише на боці клієнта, проте алгоритми генерації “лазівки” та розшифрування документів повинні бути також реалізовані в межах клієнтської системи.

Виходячи з методології моделювання загроз та з представленої на рисунку 2.2 діаграми DFD, можна ідентифікувати такі загрози:

1) Атака “людина посередині” між вузлами системи, що мають мережеве з’єднання. Зловмисник зможе отримати відомості про “лазівки” (trapdoor data) та перехопити шифротексти документів (encrypted documents). Він також зможе перехопити ідентифікатори відповідних документів (document IDs).

2) Атака на алгоритм TRAPDOOR, що дозволить зловмиснику генерувати власні “лазівки” без знання ключової інформації.

3) Атака на вузол сховища даних “encrypted data storage” – атака “відомого шифротексту”, що дозволить зловмиснику отримати зашифровані дані без належної автентифікації, або відомості про зв’язки між документами тощо.

Вказані загрози передбачені описаними раніше моделями безпеки. Відповідно до першої та третьої загроз, що полягають в пасивному прослуховуванні та доступі до сховища даних протидіють методи, що відповідають моделі IND-CPA та IND1-CKA.

Виходячи з основних характеристик схем SE, та меж системи, визначеної на діаграмі з рисунку 2.2, а також характеристики основних вузлів ПС, наведеної в цьому підрозділі, можна сформулювати наступні безпекові вимоги до методу що розробляється:

– Метод повинен забезпечувати можливість пошук в зашифрованій базі даних за префіксом.

– Метод повинен відповідати моделі безпеки IND1-CKA та бути захищеним від атаки відомого шифротексту.

– Метод повинен використовувати криптографічно стійку реалізацію алгоритму TRAPDOOR.

– Метод повинен підтримувати можливість роботи на стороні клієнтського вузла, а отже бути невибагливим до обчислювального ресурсу клієнта.

Наведеним вимогам відповідають методи SSE. Для випадку пошуку за префіксом пошуковий індекс зручно зберігати у вигляді префіксного дерева. Детальніше розглянемо реалізацію такого методу.

## 2.2 Метод пошуку за індексом з використанням префіксного дерева

Вперше поняття префіксного дерева (trie, prefix tree) було сформульовано в 1960 році в роботі [1], де ця структура даних була запропонована для підвищення ефективності алгоритму пошуку. В [2] префіксне дерево визначається як $M$-ярусне дерево, вузлами якого є $M$-місні вектори, чиї компоненти відповідають цифрам або символам. Кожен вузол на ярусі $l$ представляє множину всіх записів, що починаються з визначеної послідовності з $l$ символів, яка називається її *префіксом*. Такий вузол визначає $(l+1)$-й символ.

В основу методу що пропонується покладено припущення, що пошук буде виконуватись за впорядкованою послідовністю символів (байт), які по відношенню до об’єкта пошуку є префіксом, тобто певною початковою частиною його байтів (символів). Тоді для пошуку між двох об’єктів “ABC…” і “ABD…” можна побудувати префіксне дерево, як показано на рисунку 2.3:

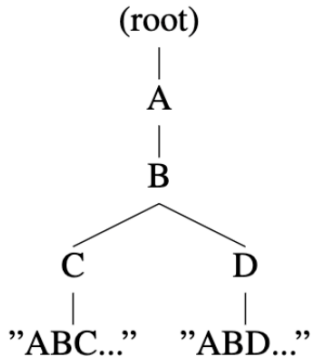


Рисунок 2.3 – Префіксне дерево для об’єктів “ABC” і “ABD”

На рисунку 2.3 видно, що  префікс AB є спільним для обох об’єктів, а префікс ABC дозволяє вибрати тільки перший об’єкт.

Пошук в такому дереві полягає в послідовному проходженні з кореня до певного кінцевого вузла, при чому для його виконання немає необхідності розкриття значення префіксу, а достатньо лише забезпечити єдине для всіх вершин перетворення $H$, за яким будь-який префікс $A$ однозначно перетворюється в значення $A'$, а будь-яке $K \neq A$ перетворюється на $H(K) = K' \neq A'$. Таким властивостям відповідають хеш-функції, а оскільки таке перетворення є незворотним, то зберігання пошукового префіксного дерева з перетвореними вершинами не розкриває даних, що зберігаються. Використання хеш-перетворення $H$ щодо вершин дерева і щодо префіксів пошукового запиту дозволяє виконувати всі дії з таким деревом так само як і зі звичайним префіксним деревом. В контексті шифрування з можливістю пошуку можна говорити, що функція $H$ є алгоритмом генерації “лазівки” TRAPDOOR.

Замість зберігання об’єктів пошуку у вершинах дерева, доцільно забезпечити лише зберігання посилань на об’єкти (документи) в зовнішньому

сховищі — це дозволяє використовувати різні відповідні типи сховищ для деревовидної структури пошукового індексу, і для статичних даних документів.

### 2.2.1 Пошук в префіксному дереві

Узагальнено алгоритм пошуку документа для Користувача даних можна представити у вигляді діаграми діяльності, як показано на рисунку 2.4:

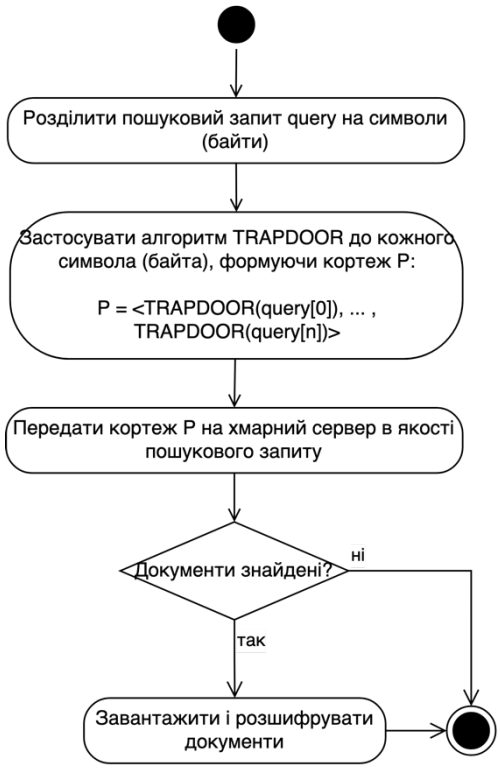


Рисунок 2.4 – Діаграма діяльності алгоритму пошуку документа для Користувача даних

Виконуючи наведений алгоритм, Користувач формує відповідний до його пошукового запиту кортеж префіксів, перетворює його на кортеж “лазівок” $P$, та отримує результати пошуку з Хмарного сервера.

Алгоритм пошуку документів за допомогою пошукового індексу на основі префіксного дерева за префіксом $P = \langle p_o, \dots, p_n \rangle$ для Хмарного сервера у вигляді псевдокоду показаний на рисунку 2.5:

**Algorithm** Hashed prefix tree search

**Require:** $query = \langle p_0, ..., p_i \rangle, i \geq 0$

```
function SEARCH(root, query, depth)
    D be a new array
    k ← 0
    if root.hash ≠ query[depth] then
        return []
    end if
    if |root.children| = 0 then
        return []
    end if
    if depth ≥ |query| then
        for all document ∈ root.documents do
            D[k] = document
            k ← k + 1
        end for
    end if
    for all p_i ∈ root.children do
        D.extend(SEARCH(p_i, query, depth + 1))
    end for
     return D
end function
```

Рисунок 2.5 – Алгоритм пошуку в префіксному дереві

На рисунку 2.5 видно, що алгоритм пошуку рекурсивно проходить по пошуковому дереву, розглядаючи черговий вузол $root$ у відповідності з пошуковим запитом $query$. У випадку, якщо поточна глибина перебору пошукового індексу $depth$ є більшою ніж довжина пошукового запиту $query$, то всі пов’язані документи $documents$ розглядаються як результат пошуку.

Результатом роботи алгоритму є множина документів $D$, які безпосередньо пов’язані з зазначеною кінцевою вершиною запиту $p_i$, або її дочірніми вершинами. На практиці необхідно забезпечити обмеження на глибину пошуку пов’язаних документів, щоб запобігти ненавмисній видачі надто великої кількості результатів пошуку за коротким пошуковим запитом.

Формалізація процесу пошуку у вигляді IDEF0 діаграми наведена на рисунку 2.6:

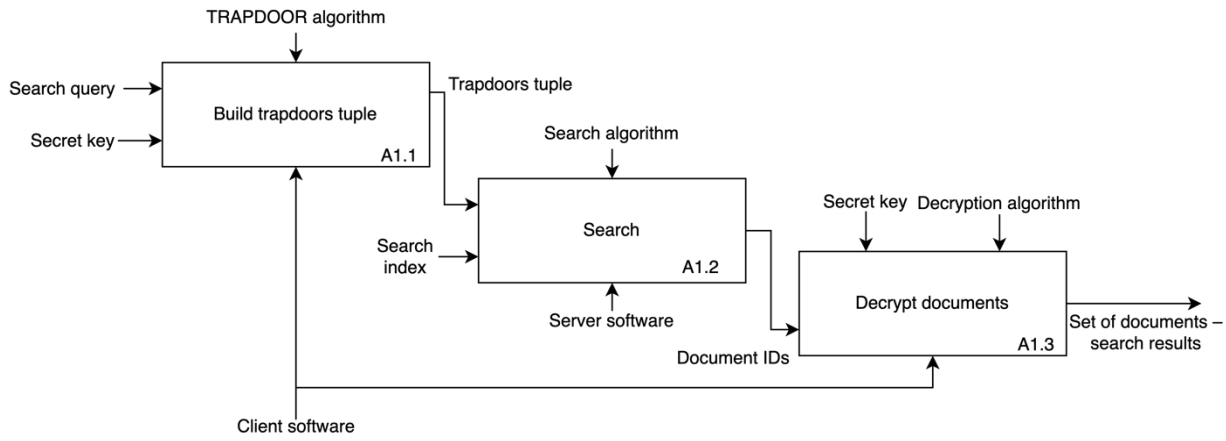


Рисунок 2.6 – IDEF0 діаграма процесу пошуку документа

На діаграмі з рисунку 2.6 видно, що пошуковий запит перетворюється на кортеж “лазівок” $P$ (на відміну від звичайних схем SE, де пошуковий запит має являти собою ціле ключове слово $w_i$), який надалі використовується пошуковим алгоритмом для перебору вершин пошукового дерева.

### 2.2.2 Вставлення в префіксне дерево

Узагальнено алгоритм додавання документа для Власника даних можна представити у вигляді діаграми діяльності, як показано на рисунку 2.7:


Рисунок 2.7 – Діаграма діяльності алгоритму додавання документа для Власника даних

Виконуючи наведений алгоритм, Власник даних формує кортеж префіксів довільної довжини $n$, перетворює його на кортеж “лазівок” $D$ для вставлення в префіксне дерево, та передає ці дані на бік Хмарного сервера.

Відповідний алгоритм вставлення посилання на документ $ref$ за префіксом $D$ в префіксне дерево $T$ у вигляді псевдокоду показаний на рисунку 2.8:

**Algorithm** Hashed prefix tree insert

**Require:** $D$ array of bytes
**Require:** $T$ prefix tree root
**Require:** $ref$ document ID
  **function** INSERT(D, T, ref)
    $H$ be a new array, $|H| = |D|$
    **for** $i = 0, \ldots, |D|$ **do**
      $H[i] = sha256(D_i)$
    **end for**
    $depth \leftarrow 0$
    **while** $depth < |D|$ **do**
      **if** not exist $T.children[H[depth]]$ **then**
        $T.children.extend(H[depth])$
      **end if**
      $T \leftarrow T.children[H[depth]]$
      $depth \leftarrow depth + 1$
    **end while**
    $T.documents.extend(ref)$
  **end function**

Рисунок 2.8 – Алгоритм пошуку в префіксному дереві

На рисунку 2.8 видно, що алгоритм спочатку перетворює елементи кортежу $D$ на кортеж “лазівок” $H_i = sha256(D_i)$ – в якості алгоритма TRAPDOOR в даному методі пропонується до використання криптографічна функція хешування SHA-256. Далі алгоритм перебирає вузли пошукового дерева $T$, перевіряючи наявність в ньому відповідних до $H$, та за потреби додаючи їх. До останнього $|H|$-го вузла додається посилання на документ $ref$. Результатом роботи є розширене новими вузлами (якщо поточна структура індексу цього потребує) та посиланням на документ $ref$ пошукове префіксне дерево $T$.

Формалізація процесу додавання документа в пошуковий індекс на основі префіксного дерева у вигляді IDEF0 діаграми показана на рисунку 2.9:

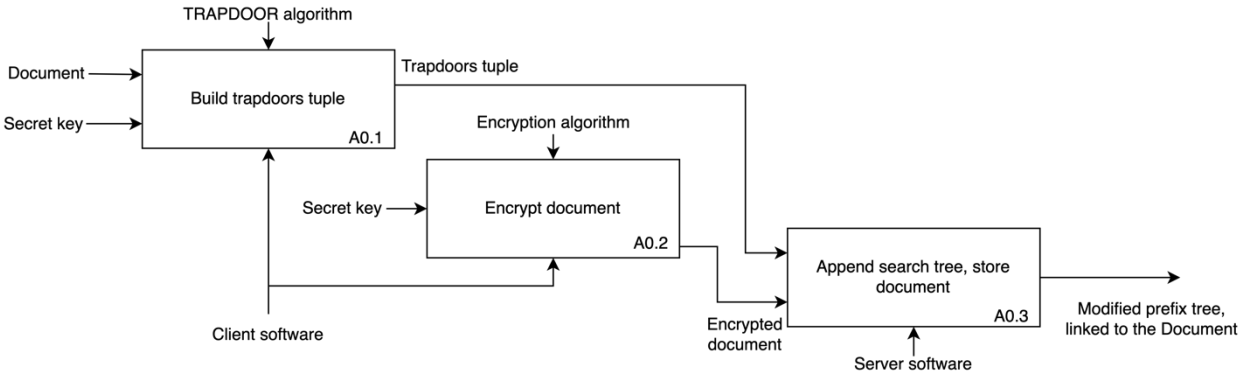


Рисунок 2.9 – IDEF0 діаграма процесу додавання документа в пошуковий індекс

На діаграмі з рисунку 2.9 видно, що на відміну від такого процесу для звичайної реалізації SE, для кожного документа замість ключових слів $w_i$ обирається кортеж префіксів, які після застосування до них алгоритму TRAPDOOR розміщуються в префіксному дереві – пошуковому індексі.

## 2.3 Висновки за розділом

1) В розділі розглянуто моделі загроз для різних варіантів використання методів SE.

2) На основі аналізу отриманої інформації були сформульовані вимоги до запропонованого методу.

3) Для запропонованого методу були формалізовані алгоритми пошуку та додавання в пошуковий індекс у вигляді псевдокоду та діаграм діяльності, відповідні процеси модельовані за допомогою IDEF0.

# 3 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ МЕТОДУ ПОШУКУ ЗА ПРЕФІКСОМ

## 3.1 Розробка специфікації системних вимог програмного забезпечення

Аналіз вимог полягає в визначенні потреб та умов які висуваються щодо нового, чи зміненого продукту, враховуючи можливо конфліктні вимоги різних замовників, таких як користувачі чи бенефіціари [12].

Аналіз вимог є критичним для успішної розробки проекту. Вимоги мають бути задокументованими, вимірними, тестовними, пов’язаними з бізнес-потребами, і описаними з рівнем деталізації достатнім для конструювання системи.

Подальша робота описує розробку прототипу програмного забезпечення, що використовує запропонований метод для прикладу пошуку за префіксами десяткових координат, оскільки в такому випадку пошук за префіксом набуває фізичного сенсу (кожний наступний префікс уточнює область пошуку об’єктів на карті) а отже наочніше демонструє роботу методу. В цьому випадку в якості документів виступають відомості про місцезнаходження деяких об’єктів на карті, разом з іншими відомостями про них.

### 3.1.1 Функціональні вимоги

Функціональні вимоги – це перелік функцій або сервісів, які повинна надавати система, а також обмежень на дані та поводження системи при їхньому виконанні. Специфікація функціональних вимог – опис функцій та їхніх властивостей, які не містять у собі протиріч і виключень.

Для системи, що розробляється, відповідно до визначеного в підрозділі 3.1 загального опису призначення програмного забезпечення, були сформульовані наступні функціональні вимоги:

а) Система повинна виконувати пошук за допомогою запропонованого методу SE в БД за довільним префіксом та відображати результати пошуку.

б) Система повинна надавати користувачу можливість додавати власні об’єкти в БД та пошуковий індекс.

### 3.1.2 Нефункціональні вимоги

Нефункціональні вимоги визначають умови виконання функцій (наприклад, захист інформації у БД, автентифікація доступу до ПС тощо) у середовищі, що безпосередньо не пов’язані з функціями, а відбивають потреби користувачів щодо їх виконання. Ці вимоги характеризують принципи взаємодії із середовищами або іншими системами, а також визначають показники часу роботи, захисту даних і досягнення якості з урахуванням рекомендацій деякого стандарту [12]. Вони можуть встановлюватися як числові значення (наприклад, час очікування на відповідь, кількість клієнтів, що обслуговуються тощо) у різних одиницях виміру, включаючи, наприклад, імовірність (значення імовірності безвідмовної роботи системи – показника її надійності).

Для системи що розробляється, відповідно до визначеного в підрозділі 3.1 загального опису призначення програмного забезпечення, були сформульовані такі нефункціональні вимоги, що також враховують вимоги висунуті в підрозділі 2.1 до методу в цілому:

а) Секретний ключ повинен використовуватись та зберігатись тільки на клієнтському вузлі.

б) Пошуковий індекс та документи, що зберігаються в БД, повинні бути криптографічно захищені, а їх зміст не повинен розкриватись на будь-якому вузлі системи, окрім клієнтського.

в) Достатньо висока швидкодія – час очікування результатів пошуку не більше 3 секунд, час розшифровування документу не більше 1 секунди.

г) Зручний користувальницький інтерфейс: результати пошуку повинні відображатися у вигляді таблиці та на мапі, поточна область пошуку також має бути позначена на мапі.

## 3.2 Вибір системної архітектури

Архітектура програмного забезпечення в [12, 13] визначається як модель фундаментальної структури та організації програмної системи. Її доцільно розглядати в декількох аспектах:

а) організації системи програмного забезпечення та його оточення, контексту в якому вона працює;

б) взаємодії системи з іншими системами, або взаємодії її внутрішніх компонентів;

в) організації системи як набору структур даних, які використовуються в системі;

г) поведінки системи, яка моделює динамічну поведінку системи і те, як вона реагує на події.

Архітектура програмного забезпечення стосується не тільки структури та поведінки, а й використання, функціональності, продуктивності, стійкості, повторного використання, зрозумілості, економічних і технологічних обмежень, компромісів і естетики ПЗ.

В застосунках де передбачається використання шифрування з можливістю пошуку доцільно використовувати трирівневий патерн (3-tier) [14] системної архітектури, тому що:

– Сховище даних є ізольованим від користувачів, та може бути фізично розподіленим на БД пошукового індексу та сховище зашифрованих документів.

– Сервер застосунку приховує подробиці взаємодії з пошуковим індексом.

– Такий патерн системної архітектури дозволяє перенести криптографічні функції на бік клієнтського застосунку.

## 3.3 Діаграма розгортання програмного забезпечення роботи

Діаграма розгортання – діаграма, на якій представлені вузли системи [15].

Діаграма розгортання застосовується для подання загальної структури та топології системи, і містить зображення розміщення компонентів по окремих вузлах системи. Крім того, діаграма розгортання показує наявність фізичних з'єднань – маршрутів передачі інформації між апаратними засобами, залученими в реалізації системи.

Типова діаграма розгортання застосунку, що використовує такий патерн системної архітектури наведений на рисунку 1.1. На рисунку 3.1 показана діаграма розгортання для системи, що розроблюється.

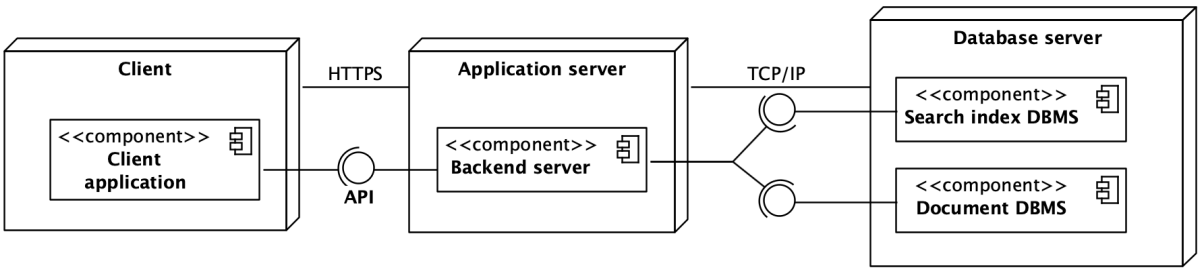


Рисунок 3.1 – Діаграма розгортання для ПС прототипу реалізації методу

На діаграмі з рисунку 3.1, яку можна порівняти з типовою схемою що представлена на рисунку 1.1, видно, що на фізичному рівні використання SE не призводить до значних змін в архітектурі застосунку, окрім можливого додавання нового компонента БД, що може зберігати пошуковий індекс.

## 3.4 Логічна модель бази даних

В результаті проведеного аналізу предметної області, що розглядається, була запропонована логічна модель бази даних в нотації UML як показано на рисунку 3.2:

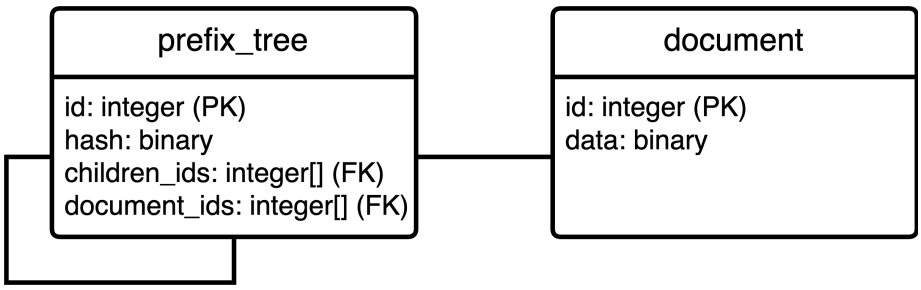


Рисунок 3.2 – Логічна модель бази даних

Виходячи з наведеної моделі БД можна зробити висновок про те, що для зберігання даних про структуру префіксного дерева (prefix_tree) та документів

(document) можна використовувати як реляційну СКБД, так і документ-орієнтовану, наприклад, MongoDB, яка має вбудовані можливості для зберігання деревовидних структур даних. В певних типах сховищ зберігання або виконання запитів до деревовидного пошукового індексу може бути пов’язано з накладними витратами, обумовленими особливостями тої чи іншої СКБД. Відмінності використання конкретної СКБД потребують додаткового дослідження та вимірювання на таких типових операціях як запити до пошукового дерева, вставлення та видалення піддерев тощо.

### 3.5 Вибір інструментальних засобів реалізації програмного забезпечення

На сьогодні ринок надає широкий вибір технологій реалізації програмного забезпечення роботи. В якості потенційних засобів розробки програмного забезпечення розглядалися наступні технології: Java, JavaScript, Python, ASP.NET, PHP. Список з цих технологій був складений виходячи з практичної можливості використання в роботі кожної з них, що дозволяє виключити з плану роботи витрати часу на вивчення нової технології.

При виборі інструментарію були сформульовані відповідні критерії:

– Багатоплатформність – один і той же програмний код повинен покрити якомога більше платформ.

– Доступність – розробка повинна бути максимально дешевою.

– Простота розробки – час розробки має бути мінімальним.

– Наявність зручного та сучасного GUI-фреймворку для швидкої розробки графічного інтерфейсу.

– Наявність простих інструментів статичного аналізу коду для підтримки його якості.

Детальніше розглянемо властивості зазначених засобів розробки.

#### 3.5.1 Мова програмування Java

Java – це об’єктно орієнтована мова програмування, яка була вперше представлена в 1995 році компанією Sun Microsystems як ключовий компонент платформи Java. З 2009 року розвитком та підтримкою мови займається

компанія Oracle, яка того року придбала Sun Microsystems. Офіційна реалізація програм на Java компілюється в байт-код, який потім виконується віртуальною машиною, специфічною для конкретної платформи.

Oracle надає компілятор Java та віртуальну машину Java, які відповідають специфікаціям Java Community Process, і це відбувається під ліцензією GNU General Public License.

Синтаксис Java схожий на C і C++, але з істотними відмінностями. Розробники технології врахували досвід розробки за допомогою цих двох мов, що призвело до усунення можливості виникнення деяких конфліктних ситуацій, які могли виникнути через помилки програміста, і спростило процес розробки об’єктно орієнтованих програм. Багато з тих дій, які вимагаються в C/C++, тепер виконує віртуальна машина. Головною метою Java завжди було створення мови, яка була б платформонезалежною, і це визначає її обмеження в роботі з апаратним забезпеченням порівняно з, наприклад, C++. Таким чином, швидкість роботи програми може бути меншою. Проте, Java надає можливість викликати підпрограми, написані іншими мовами програмування, коли це необхідно. Приклад програми, що написана мовою Java, в середовищі IntelliJ IDEA наведено на рисунку 3.3:

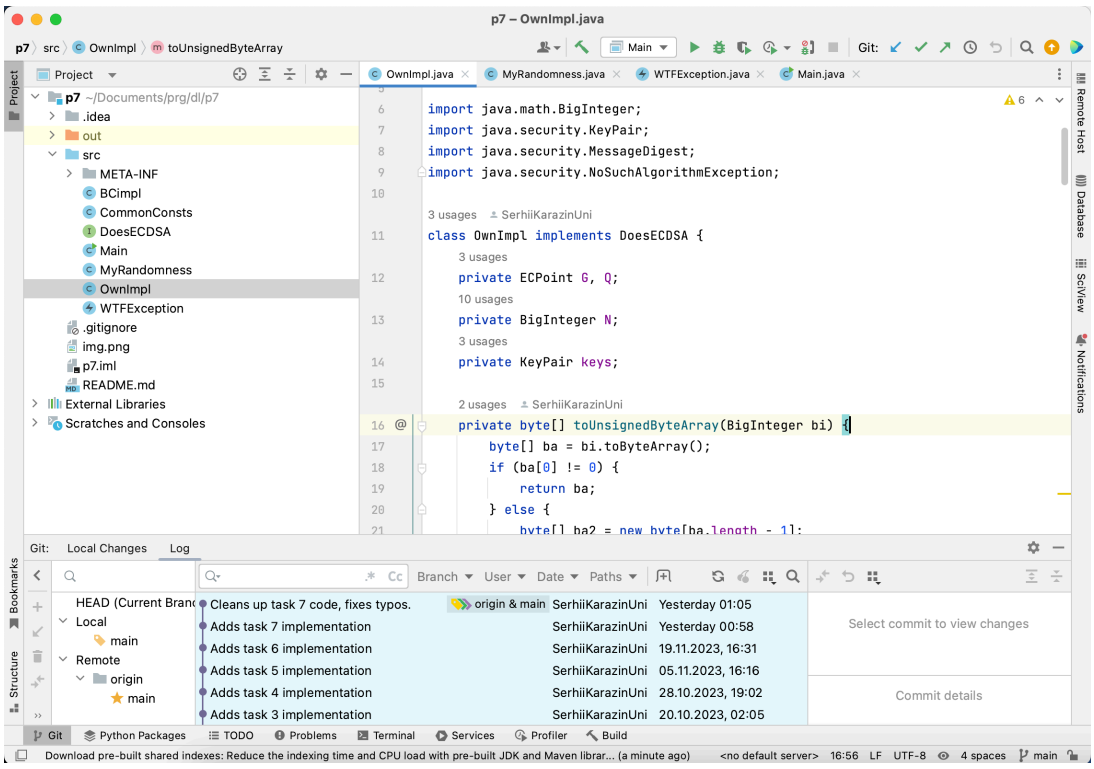


Рисунок 3.3 – Програма, що написана мовою Java в середовищі IntelliJ IDEA

Java мала значний вплив на розвиток мови програмування J++, яку розробляла компанія Microsoft. Роботу над J++ було припинено через судовий позов, поданий Sun Microsystems, оскільки ця мова програмування була модифікацією Java. Пізніше, при створенні нової платформи .NET компанією Microsoft, була представлена мова J#, яка спрощувала міграцію програмістів, які вже володіли J++ або Java, на нову платформу. З часом нова мова програмування C# стала основною мовою платформи .NET, позичивши багато ідей з Java. J# включали востаннє в версію Microsoft Visual Studio 2005.

Іншою технологією, що має широкі можливості як клієнтських, так і серверних застосунків є JavaScript.

### 3.5.2 Мова програмування JavaScript

JavaScript є однією з найпоширеніших мов програмування у світі. Вона підтримується всіма основними браузерами, включаючи Chrome, Firefox, Edge та Safari. JS також є основою для багатьох популярних фреймворків, таких як React, Angular та Vue.JS.

Серед основних переваг JavaScript можна назвати такі:

– JavaScript має простий синтаксис, подібний до C та C++.

– JavaScript є широко поширеною мовою, що робить її легко доступною для розробників.

– Java Script є стандартом де-факто для сучасних веб-застосунків, підтримується більшістю веб-браузерів.

– Деякі реалізації JavaScript є проєктами відкритого коду, що дозволяє розробникам робити свій внесок у розвиток мови, проводити незалежні аудити безпеки тощо.

Серед основних недоліків JavaScript виділяють:

– Програми на JavaScript інтерпретуються на стороні клієнта, а отже передаються туди у вигляді відкритого тексту. Це може впливати на безпеку застосунків, адже клієнт може модифікувати код.

– JavaScript є динамічною мовою без статичної типізації, що може призводити до помилок.

Приклад програми, що написана мовою JavaScript, в середовищі IntelliJ IDEA наведено на рисунку 3.4:

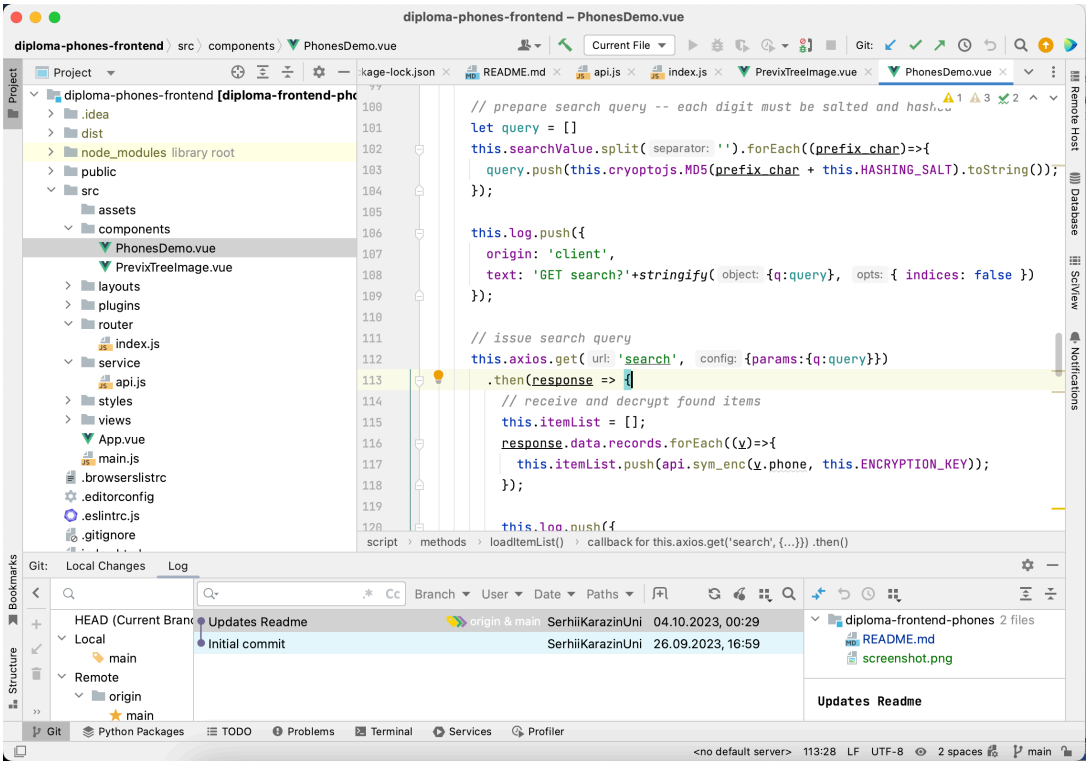


Рисунок 3.4 – Програма, що написана мовою JavaScript в IntelliJ IDEA

Для JavaScript існують фреймворки та численні інструменти підтримки якості коду, що можуть значно полегшити розробку програмного забезпечення роботи. Одним з них є Vue.JS – фреймворк для створення односторінкових веб застосунків, що прискорює розробку. Іншим фреймворком, що може бути використаним в роботі, є Vuetify – фреймворк для створення інтерфейсів користувача на основі Vue.JS. Vuetify надає широкий спектр готових компонентів, які можна використовувати для швидкого створення красивих і зручних у використанні інтерфейсів.

Альтернативою до використання JavaScript є технології на платформі .NET, зокрема ASP.NET.

### 3.5.3 Технологія ASP.NET

ASP.NET – це технологія створення веб застосунків і веб сервісів від компанії Microsoft, і вона є важливою частиною платформи Microsoft.NET. Ця технологія являє собою еволюцію попередньої версії – Microsoft ASP. На сьогодні останньою версією ASP.NET є ASP.NET 6.

ASP.NET має багато спільних рис зі своєю попередницею ASP, що дозволяє розробникам переходити на ASP.NET досить легко. Проте внутрішня структура ASP.NET суттєво відрізняється від ASP, оскільки ASP.NET базується на платформі .NET і використовує всі переваги цієї платформи. Приклад програми, що написана на платформі ASP.NET, в середовищі Microsoft Visual Studio наведено на рисунку 3.5:

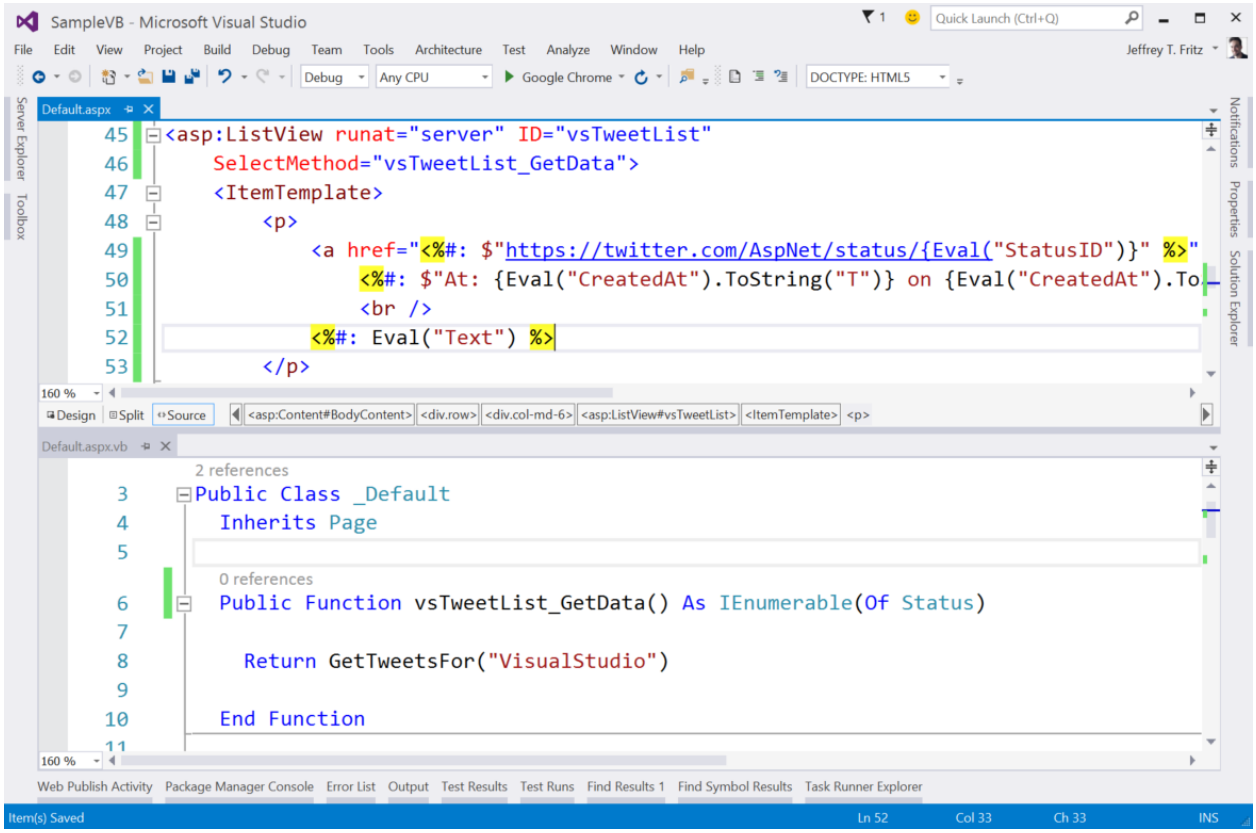


Рисунок 3.5 – Програма, що написана мовою ASP в середовищі Microsoft Visual Studio

Microsoft повністю переробила ASP.NET, використовуючи Common Language Runtime як основу, яка є фундаментальною для всіх застосунків Microsoft .NET. Один із головних плюсів ASP.NET полягає в тому, що розробники можуть писати код для ASP.NET, використовуючи практично будь-яку мову програмування, доступну в .NET Framework (такі як C#).

Основною перевагою ASP.NET є його продуктивність. Під час першого запиту код компілюється і зберігається в спеціальному кеші, що дозволяє його швидше виконувати в подальших запитах, і не потребує парсингу, оптимізації та інших проміжних операцій, що заощаджує час.

Іншою популярною технологією є Python.

### 3.5.4 Мова програмування Python

Python – це інтерпретована об’єктно-орієнтована мова програмування високого рівня зі строгою динамічною типізацією. Структури даних високого рівня разом із динамічною семантикою та динамічним зв’язуванням роблять її

привабливою для швидкої розробки програм, а також як засіб поєднання існуючих компонентів. Python підтримує модулі та пакети модулів, що сприяє модульності та повторному використанню коду. Інтерпретатор Python та стандартні бібліотеки доступні як у скомпільованій, так і у вихідній формі на всіх основних платформах.

В мові програмування Python підтримується декілька парадигм програмування, зокрема: об’єктно-орієнтована, процедурна, функціональна та аспектно-орієнтована.

Серед основних її переваг можна назвати такі:

– Чистий синтаксис.

– Переносність програм (що властиве більшості інтерпретованих мов).

– Зручний для розв’язання математичних задач (має засоби роботи з комплексними числами, може оперувати з цілими числами довільної величини).

– Відкритий код.

Виділяють такі недоліки Python:

– Низька швидкодія (у порівнянні з мовами більш низького рівня).

– Відсутність статичної типізації.

Для Python існують інструменти підтримки якості коду, та фреймворки, що можуть значно полегшити розробку програмного забезпечення прототипу. Одним з них є FastAPI – високопродуктивний веб-фреймворк для створення API. Приклад програми, що написана мовою Python з використанням FastAPI, в середовищі IntelliJ IDEA наведено на рисунку 3.6:

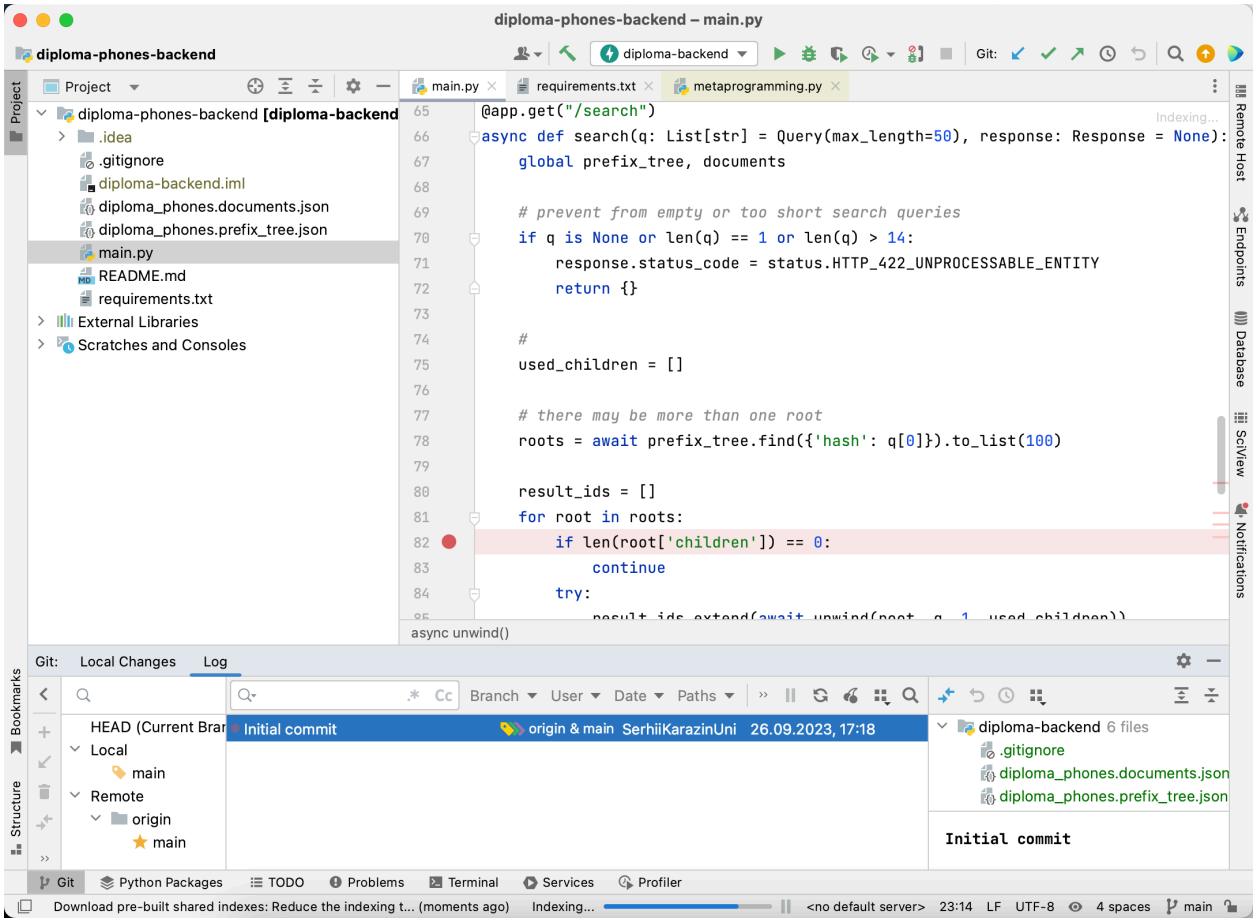


Рисунок 3.6 – Програма, що написана мовою Python з використанням FastAPI в середовищі IntelliJ IDEA

Важливі властивості FastAPI:

– Один з найшвидших веб фреймворків для Python. Він досягає високої продуктивності шляхом використання асинхронного запиту та відповіді, а також оптимізації коду.

– Має простий і інтуїтивно зрозумілий синтаксис. Він також має ряд вбудованих функцій, які спрощують створення API.

– Має вбудовані функції для документування API та тестування.

Втім, в розробці веб-застосунків є набагато популярніша технологія – PHP.

### 3.5.5 Мова програмування PHP

PHP – це скриптова мова програмування, створена для генерації HTML-сторінок на стороні веб сервера. PHP є однією з найпоширеніших мов, використовуваних у сфері веб розробки, разом з Java, .NET, Python та Ruby.

PHP була розроблена в 1994 році, а наразі є однією з найпоширеніших мов програмування для веб розробки. Велика кількість веб сайтів, включаючи популярні соціальні мережі, електронні комерційні платформи та блоги, побудовані з використанням PHP. Однією з переваг PHP є те, що вона підтримується більшістю хостинг-провайдерів, що робить її доступною для широкого кола веб розробників.

Завдяки вдосконаленням у внутрішній архітектурі та оптимізаціям, зараз PHP здатна конкурувати з іншими популярними мовами програмування, такими як Java і Python. Швидкодія стала особливо помітною у PHP 7, яка отримала значне прискорення завдяки оптимізаціям в роботі з пам’яттю та виконанню опкоду.

Основні переваги PHP включають:

– PHP – мова програмування з синтаксисом, схожим на мови з родини C.

– Спрямованість на веб розробку – PHP призначена для розв’язання задач, пов’язаних з веб розробкою, і не вимагає від програміста глибоких знань інших технологій.

– Висока популярність серед веб розробників і наявність великої спільноти користувачів.

– Ліцензія відкритого програмного забезпечення, що дозволяє вільне використання і розповсюдження.

Приклад програми, що написана мовою PHP, в середовищі IntelliJ IDEA наведено на рисунку 3.7:

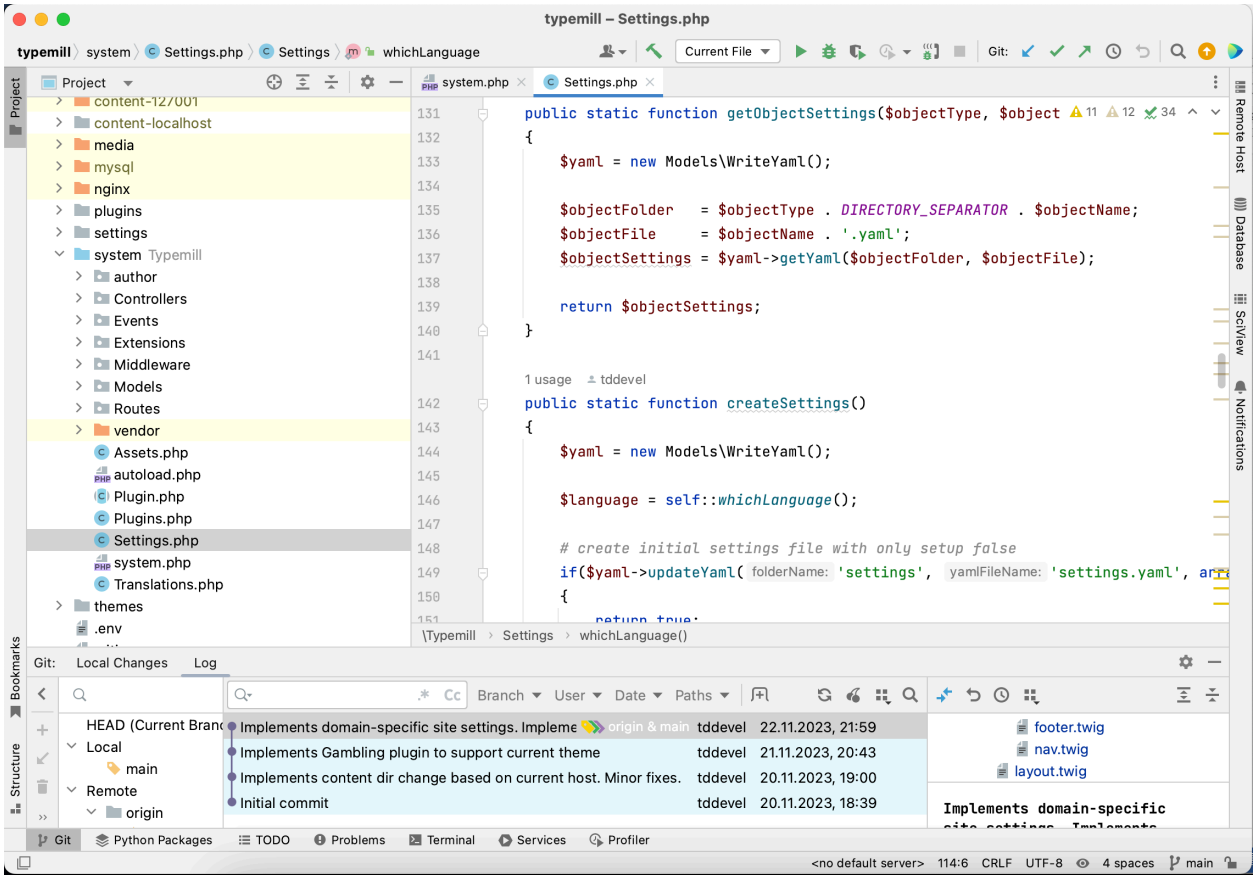


Рисунок 3.7 – Програма, що написана мовою PHP в середовищі IntelliJ IDEA

Серед недоліків виділяють:

– Безпека – PHP в минулому мав деякі серйозні проблеми з безпекою, такі як вразливості SQL-ін'єкції, вразливості з крос-сайт скриптінгом тощо. Відтоді в PHP були внесені покращення в безпеку, такі як впровадження підготованих виразів (prepared statements) для SQL та бібліотеки PDO, але питання безпеки є актуальним для PHP.

– Складні назви деяких поширених функцій, таких як htmlspecialchars, mysql_select_db, nl2br тощо. У PHP можуть існувати кілька функцій, які виконують схожі завдання, і це може призвести до плутанини та невизначеності у виборі правильної функції. Крім того, у різних функціях PHP можуть існувати різні стандарти назв функцій та порядку аргументів, що може призвести до плутанини та помилок.

Для PHP існують численні фреймворки та інструменти підтримки якості коду, проте в швидкодії Python зазвичай має перевагу над PHP.

### 3.5.6 Порівняння технологій

Для порівняння та вибору технологій можна прибігти до методу експертних оцінок за критеріями, які були сформульовані на початку розділу. Для цього технології які були розглянуті в даному підрозділі були оцінені за трьома критеріями – крос-платформність, доступність, та простота розробки. В таблиці 3.1 наведені аргументовані оцінки за кожним з критеріїв в діапазоні $[1; 5]$ для кожної з технологій:

Таблиця 3.1 – Порівняння інструментальних засобів реалізації ПЗ

| Критерій/ Засіб | Крос-платформність | Доступність | Простота розробки | Сума |
|---|---|---|---|---|
| Java | 5 – Виконується у віртуальній машині, що доступна для більшості платформ | 5 – Вільна ліцензія, багато безкоштовних інструментів | 3 – Висока складність для швидкої розробки | 13 |
| JavaScript | 5 – Скриптова мова програмування | 5 – Вільна ліцензія, відкритий код, багато безкоштовних інструментів | 4 – Можуть бути проблеми сумісності при виконанні в різних браузерах | 14 |
| ASP.NET | 3 – Виконується у віртуальній машині з обмеженою підтримкою | 4 – Вільна ліцензія, відкритий код | 3 – Висока складність для швидкої розробки | 10 |

Продовження таблиці 3.1 – Порівняння інструментальних засобів реалізації ПЗ

| | | | | |
|---|---|---|---|---|
| PHP | 5 – Скриптова мова програмування | 5 – Вільна ліцензія, відкритий код, багато безкоштовних інструментів | 5 – підтримує SDLC швидкої розробки | 15 |
| Python | 5 – Скриптова мова програмування | 5 – Вільна ліцензія, відкритий код, багато безкоштовних інструментів | 5 – підтримує SDLC швидкої розробки | 15 |

З урахуванням отриманих оцінок прийнято рішення про використання Python з фреймворком FastAPI для реалізації сервера застосунку та JavaScript з фреймворком Vuetify для клієнтського ПЗ. В якості СКБД може бути обрана будь-яка з підтриманих в Python, але з міркувань простоти реалізації роботи з деревовидними структурами даних була обрана MongoDB. Таким чином, діаграма розгортання застосунку набуває вигляду, який показано на рисунку 3.8:

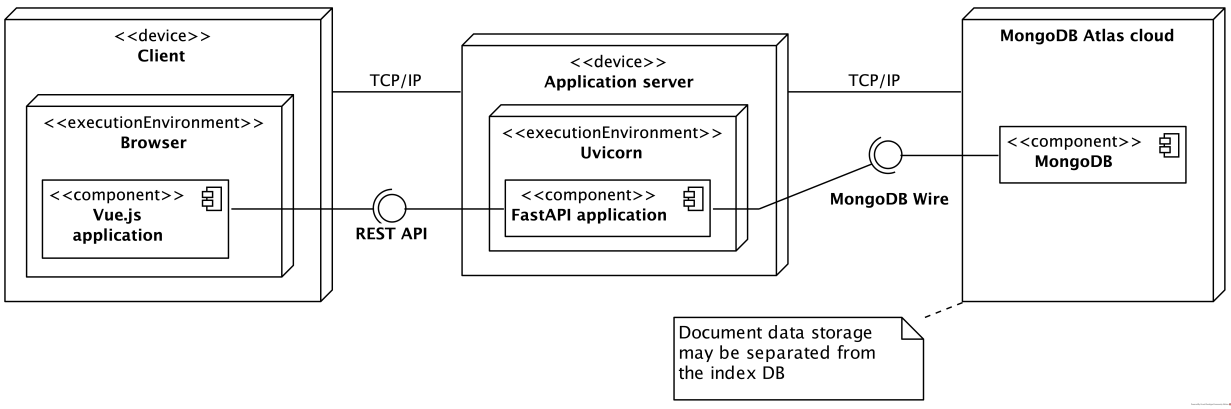


Рисунок 3.8 – Уточнена діаграма розгортання прототипу ПЗ

Як видно на діаграмі з рисунку 3.8, застосунок використовує єдину СКБД MongoDB для зберігання пошукового індексу та документів. Оскільки обсяг даних, що буде зберігатися, є невеликим, таке спрощення припустиме. В протилежному випадку, для забезпечення швидкодії пошукового індексу сховище документів повинне бути реалізовано окремо.

### 3.6 Вимоги до апаратного і програмного забезпечення

Оскільки прототип програмного забезпечення складається з декількох вузлів, вимоги до програмного і апаратного забезпечення висуваються окремо для кожного з них. Таким чином, для коректної роботи програмного забезпечення вузла «Client» в залежності від операційної системи висуваються такі вимоги як показано в таблиці 3.2:

Таблиця 3.2 – Вимоги вузла «Client» до апаратного і програмного забезпечення в залежності від операційної системи

| Операційна система | Вимоги |
|---|---|
| Windows | Windows 10 або новіша, 4 ГБ оперативної пам’яті (або більше), не менше 128 ГБ дискового простору.<br>Браузер Google Chrome або будь-який інший сумісний. |
| Mac | MacOS Catalina 10.15 або новіша, 4 ГБ оперативної пам’яті (або більше), не менше 128 ГБ дискового простору.<br>Браузер Google Chrome або будь-який інший сумісний. |

Продовження таблиці 3.2 – Вимоги вузла «Client» до апаратного і програмного забезпечення в залежності від операційної системи

| | |
|---|---|
| Linux | 64-бітна система Ubuntu 18.04, Debian 10, OpenSUSE 15.2, Fedora Linux 32 або новіші, 4 ГБ оперативної пам’яті (або більше), не менше 128 ГБ дискового простору, підтримка інструкцій SSE3.<br>Браузер Google Chrome або будь-який інший сумісний. |
| Android | Android 8.0 або новіша, 1 ГБ оперативної пам’яті (або більше), 32 ГБ дискового простору (або більше).<br>Браузер Google Chrome або будь-який інший сумісний. |

Програмне забезпечення вузла «Application server» використовує мову Python, яка дозволяє виконувати однаковий код на різних платформах, тому для цього вузла висуваються однакові вимоги для всіх підтриманих платформ (Windows, Linux, MacOS або будь-яка інша, що підтримує Python 3.11):

– 4 ГБ оперативної пам’яті (або більше), не менше 128 ГБ дискового простору.

– Інтерпретатор Python 3.11 (або сумісний).

– Бібліотеки залежностей: FastAPI 0.103.2 (або сумісна), PyMongo 4.5.0 (або сумісна), python-dotenv 1.0.0 (або сумісна).

Для вузла сервера баз даних, в якості якого в даній реалізації було обрано хмарне рішення MongoDB Atlas, висуваються такі вимоги:

– 4 ГБ оперативної пам’яті (або більше), не менше 10 ГБ дискового простору (в залежності від обсягу даних що зберігаються, ця вимога може бути більше).

– Процесор 64-бітної артіхектури.

– MongoDB 7.0.1 community (або будь-яка сумісна).

При цьому, хмарне рішення MongoDB Atlas надає декілька варіантів готових віртуальних машин для простого розгортання СКБД. Один з таких варіантів – «M0 Sandbox». Він має суттєві обмеження в ресурсах, проте задовольняє мінімальним вимогам до програмного та апаратного забезпечення прототипу.

## 3.7 Процес інсталяції програмного забезпечення

Для ознайомлення з ПЗ прототипу користувачам необхідно перейти за адресою https://se-uavs.lilikovych.name – окрім браузера Google Chrome (або сумісного) з боку користувача встановлювати будь-яке інше програмне забезпечення не потрібно.

Вихідний код програмного забезпечення прототипу доступний за адресами:

– https://github.com/SerhiiKarazinUni/diploma-uavs-frontend (інтерфейс користувача).

– https://github.com/SerhiiKarazinUni/diploma-uavs-backend (програмне забезпечення вузла сервера застосунку).

### 3.7.1 Інсталяція ПЗ сервера застосунку

Для інсталяції власної копії програмного забезпечення вузла “Application server” необхідно встановити інтерпретатор Python версії 3.11 (або новішої сумісної), який можна завантажити за адресою https://www.python.org. Знімок екрану під час інсталяції Python показаний на рисунку 3.9:

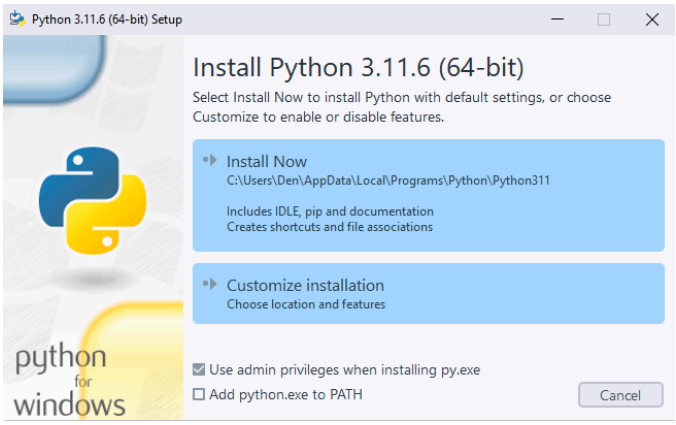


Рисунок 3.9 – Вікно інсталяції Python 3.11

Після встановлення Python, необхідно перейти в директорію “diploma-uavs-backend” та виконати команду встановлення залежностей “py -m pip install -r requirements.txt”, як показано на рисунку 3.10:

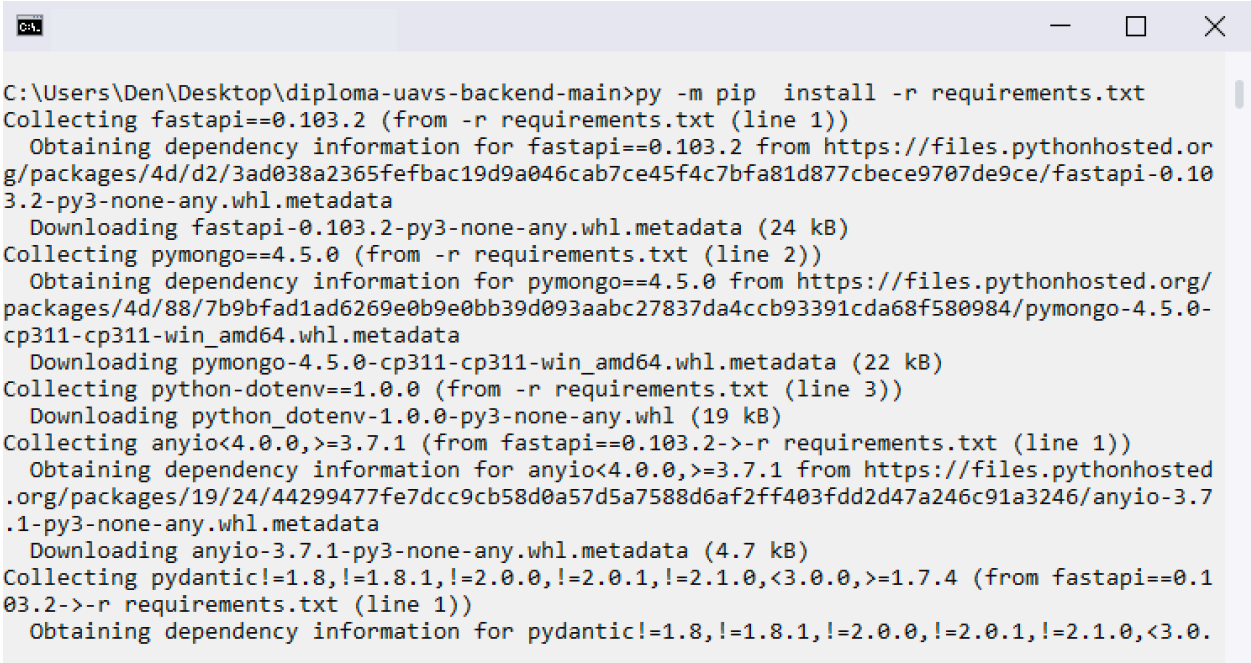


Рисунок 3.10 – Вікно інсталяції залежностей

Далі необхідно окремо встановити сервер Uvicorn за допомогою команди “py -m pip install uvicorn[standard]”.

На наступному кроці необхідно перейменувати файл “example.env” в “.env”, та відкрити його для редагування, як показано на рисунку 3.11:

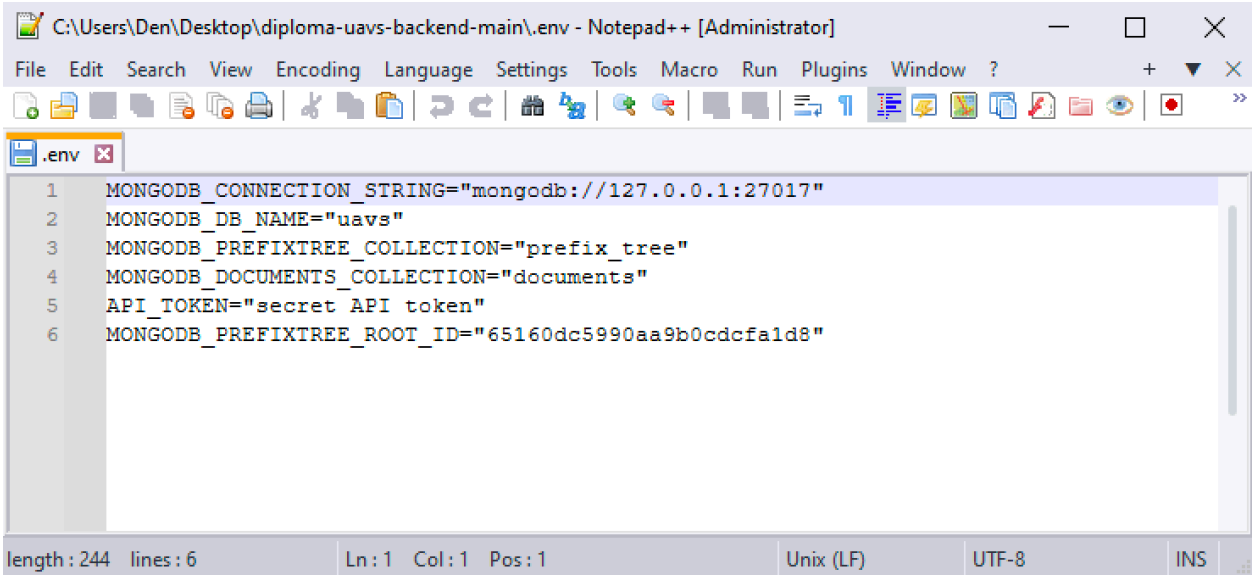


Рисунок 3.11 – Вікно редагування файлу конфігурації .env

Для налаштування необхідно встановити змінним такі значення:

– MONGODB_CONNECTION_STRING – строка, що встановлює налаштування з'єднання зі сховищем даних. У випадку використання MongoDB Atlas, її можна знайти в панелі налаштування сервера.

– MONGODB_DB_NAME – назва бази даних, що буде використовуватись.

– MONGODB_PREFIXTREE_COLLECTION – назва колекції, що буде використовуватись в якості сховища для збереження префіксного дерева.

– MONGODB_DOCUMENTS_COLLECTION – назва колекції, що буде використовуватись в якості сховища документів.

– API_TOKEN – токен авторизації API клієнтів, можна залишити без змін.

– MONGODB_PREFIXTREE_ROOT_ID – ідентифікатор документа з колекції MONGODB_PREFIXTREE_COLLECTION, який буде використовуватись в якості кореня префіксного дерева. Його необхідно створити вручну, як показано на рисунку 3.12.

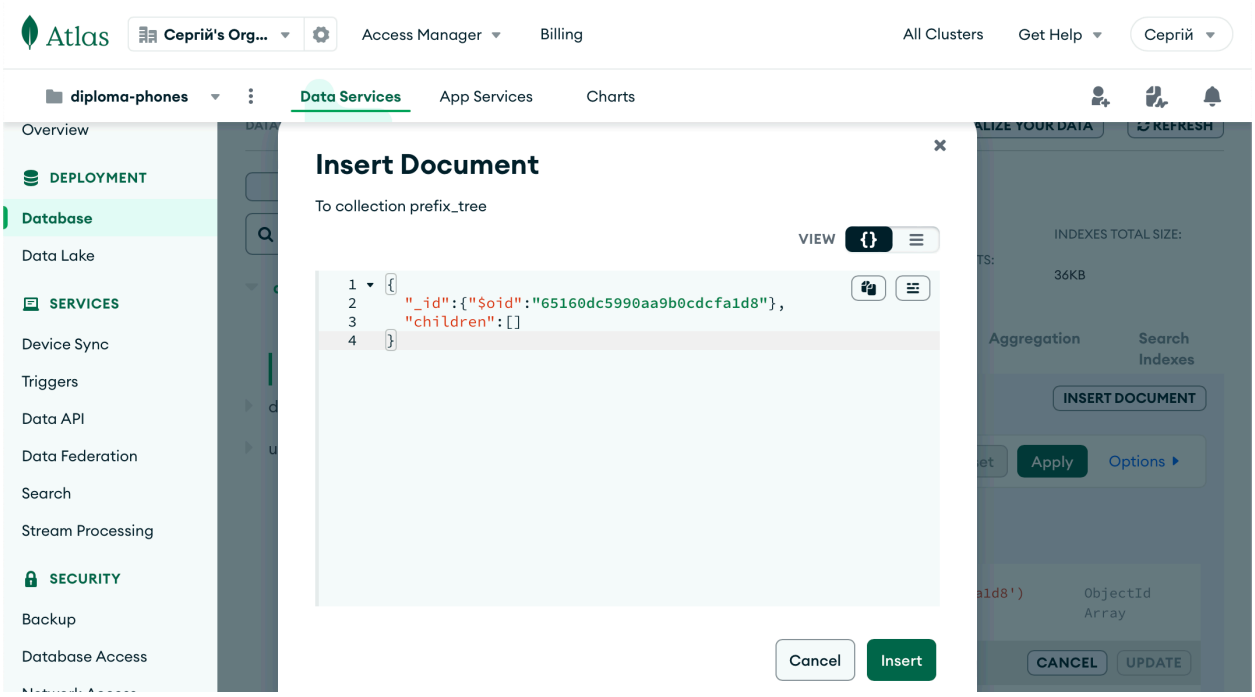


Рисунок 3.12 – Інтерфейс MongoDB Atlas під час вставлення нового документа в колекцію MONGODB_PREFIXTREE_COLLECTION

Після збереження файлу, сервер застосунку можна запустити командою “py -m uvicorn main:app --port 3000 --host 127.0.0.1”, як показано на рисунку 3.13:


Рисунок 3.13 – Вікно терміналу під час роботи ПЗ прототипу

На рисунку 3.13 видно, що до сервера застосунку за допомогою браузера було виконано HTTP-запит, на який застосунок відповів кодом 401 «Не авторизований» – це є передбачуваною поведінкою для запитів, що не містять токена API_TOKEN в заголовках, і свідоцтвом коректної роботи сервера застосунку.

### 3.7.2 Інсталяція ПЗ користувальницького інтерфейсу застосунку

Для встановлення ПЗ користувальницького інтерфейсу необхідно встановити Node.JS, дистрибутив якого можна завантажити за адресою https://nodejs.org/. Вікно встановлення Node.JS показано на рисунку 3.14:

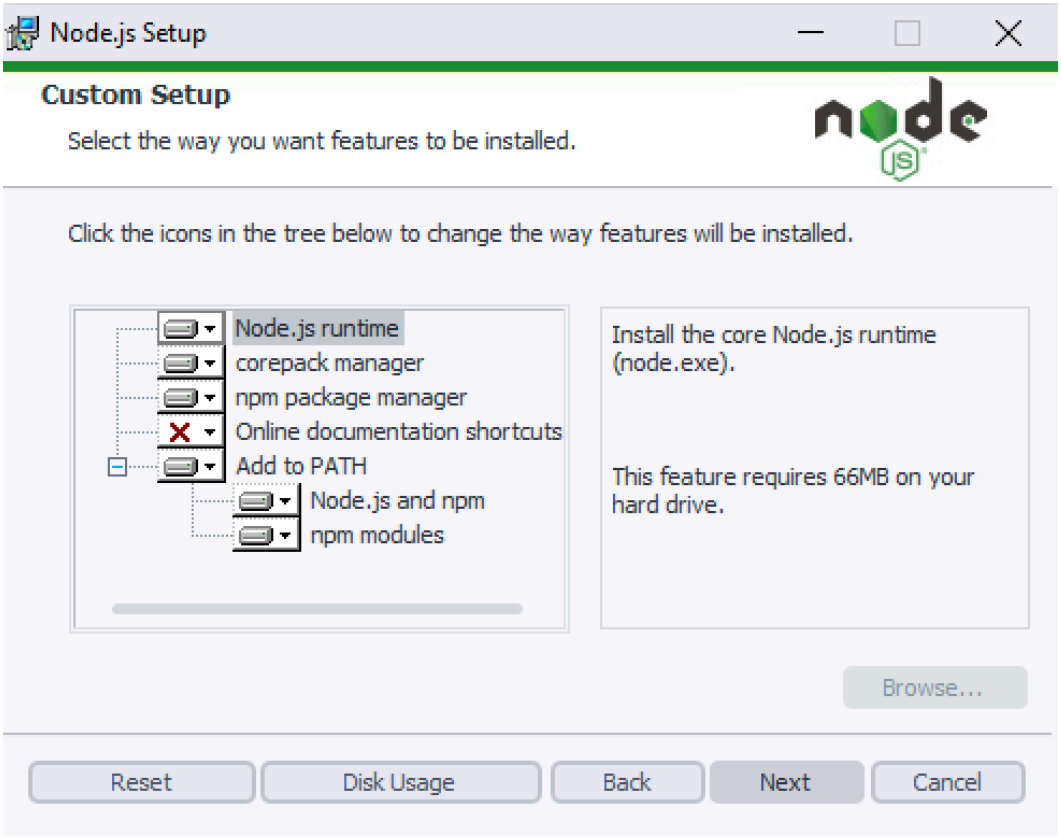


Рисунок 3.14 – Вікно інсталяції Node.JS

Після встановлення Node.JS необхідно виконати команду встановлення залежностей “npm install” з каталогу з вихідними кодами ПЗ користувальницького інтерфейсу, як показано на рисунку 3.15:

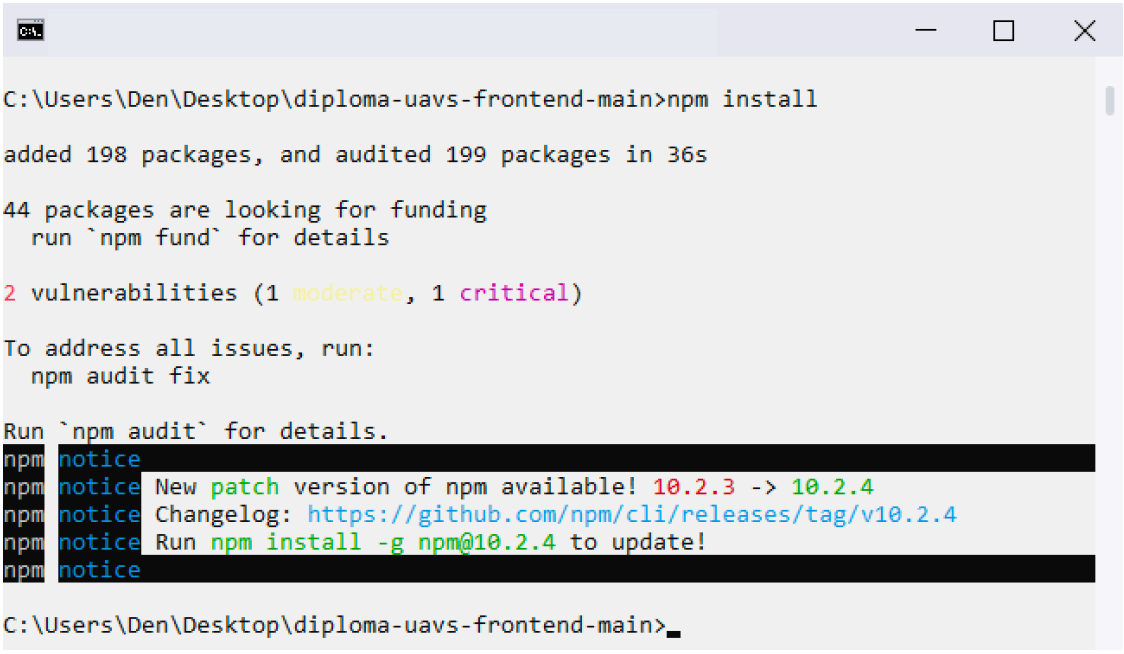


Рисунок 3.15 – Вікно інсталяції залежностей ПЗ користувальницького інтерфейсу

Після завершення процесу інсталяції залежностей, необхідно відкрити для редагування файл src/plugins/index.js, та в рядках 14 і 15 налаштувати відповідну адресу сервера застосунку. Також необхідно переконатися, що в рядку 16 встановлено актуальне значення токена клієнта, таке що відповідає API_TOKEN з налаштувань сервера. Приклад налаштувань наведено на рисунку 3.16:

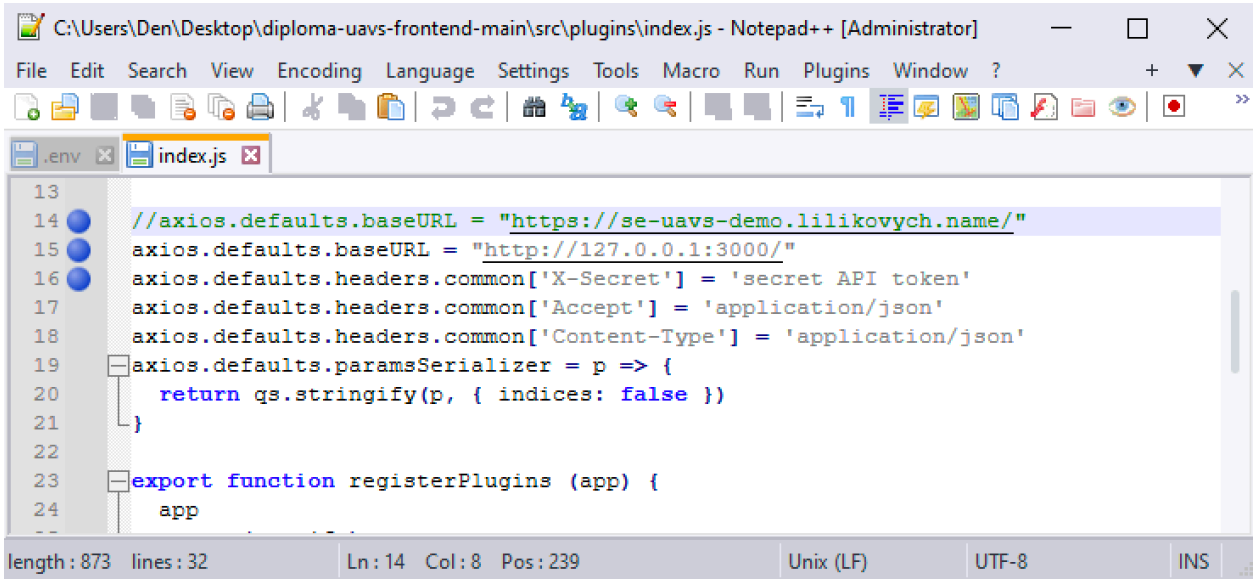


Рисунок 3.16 – Вікно редагування налаштувань ПЗ користувальницького інтерфейсу прототипу

Також в даному файлі, в рядках 27, 28, 29 можна налаштувати параметри криптографічних функцій, що використовуються в ПЗ – відповідно “сіль” хешування, та параметри секретного ключа і вектору ініціалізації для алгоритму AES, що використовується в даній реалізації для криптографічного захисту документів, як показано на рисунку 3.17:

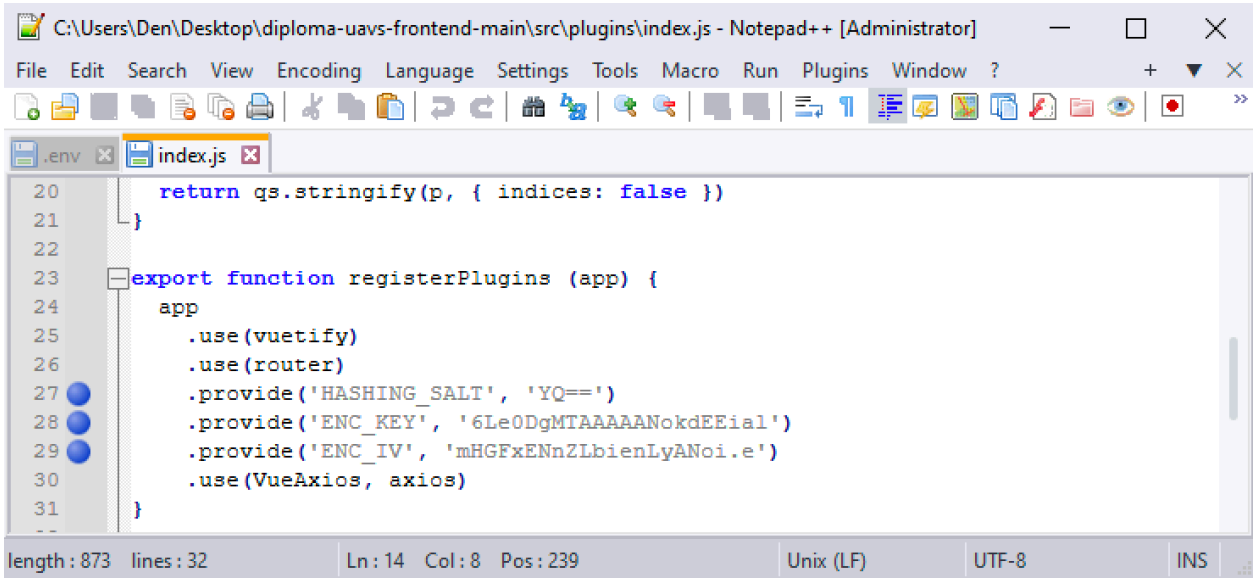


Рисунок 3.17 – Вікно редагування налаштувань криптографічних функцій ПЗ прототипу

Після збереження файлу налаштувань, можна виконати команду “npm run dev -- --port 80”, яка запускає сервер користувальницького інтерфейсу за адресою http://localhost:80, яку можна відкрити в браузері, як показано на рисунку 3.18:

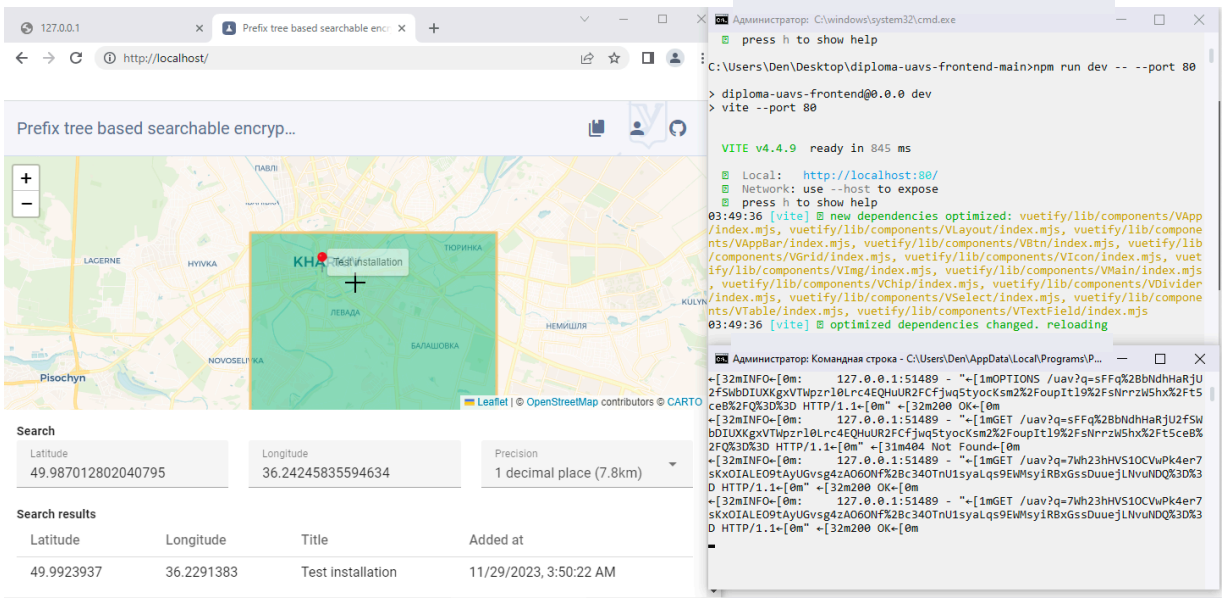


Рисунок 3.18 – Вікно браузера з користувальницьким інтерфейсом, та вікна терміналів, які відбивають нормальний статус роботи ПС

На рисунку 3.18 видно, що користувальницький інтерфейс коректно працює в браузерному вікні: додавання та пошук об’єктів на карті працює, а

вікно повідомлень сервера свідчить про відповіді з HTTP-статусом 200, що означає нормальну роботу ПС.

### 3.8 Розгортання тестового програмного прототипу для запропонованого методу

Програмне забезпечення прототипу було розгорнуте у відповідності з діаграмою розгортання, яка показана на рисунку 3.8. Фактичні характеристики вузлів системи є такими як наведено в таблиці 3.3:

Таблиця 3.3 – Специфікація апаратного забезпечення вузлів системи

| Назва вузла | Характеристики |
|---|---|
| Client | Виконується в середовищі користувача |
| MongoDB Atlas Cloud | M0 Sandbox – хмарне середовище без гарантії на доступність CPU та RAM |
| Application server | Віртуальний хостинг<br>2 ядра AMD Ryzen 9 5950X<br>2 GB RAM<br>Ресурси гарантовані<br>Python 3.9 |

Основні деталі програмної реалізації методу пошуку за префіксом в зашифрованій БД викладені в Додатку А. Після розгортання інтерфейс прототипу виглядає так, як показано на рисунку 3.9:

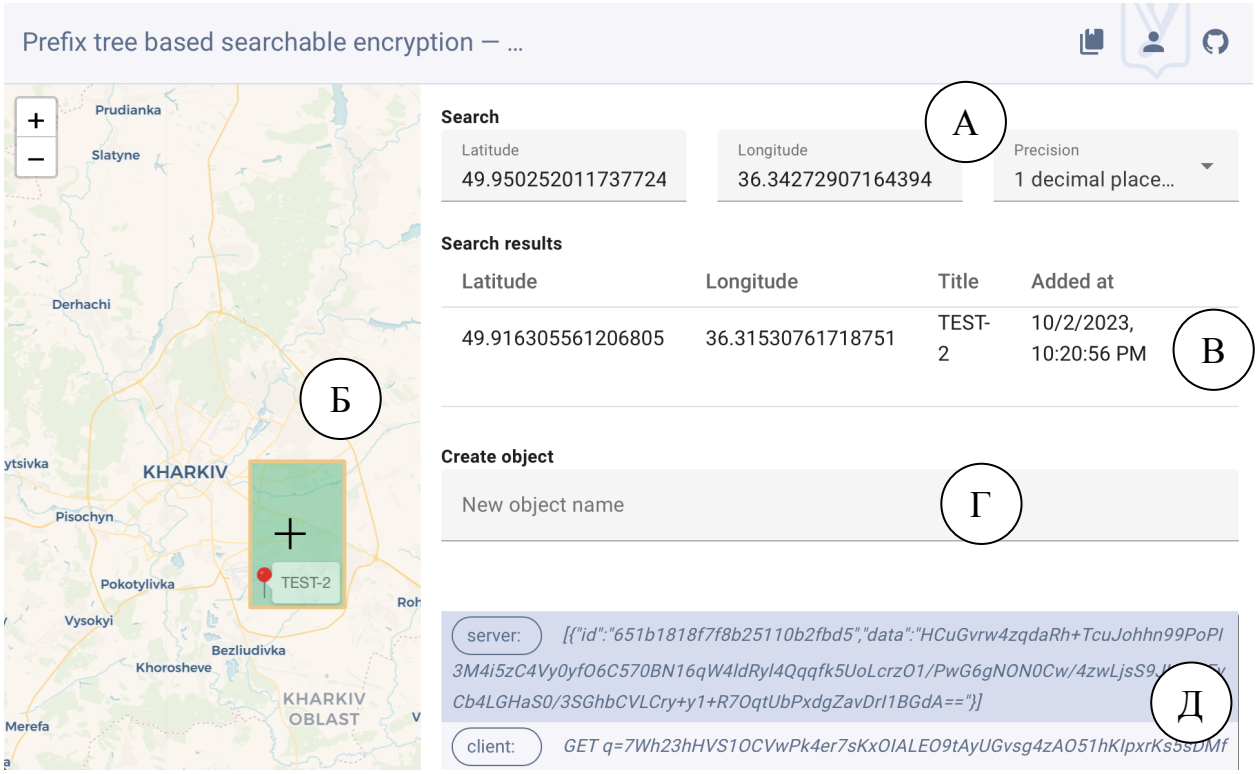


Рисунок 3.19 – Зовнішній вигляд інтерфейсу користувача прототипу ПЗ

На рисунку 3.19 показані:

а) Панель пошуку, що дозволяє задати координати центральної точки, яка визначає область пошуку, та обрати необхідну точність пошуку (остання визначається довжиною префіксу, яка буде використана для пошуку).

б) Карта, яка інтерактивно відображає поточні результати пошуку, зміни в умовах пошуку, а також дозволяє за допомогою подвійного кліку задати нову точку, навколо якої буде проводитись пошук.

в) Результати пошуку у вигляді таблиці.

г) Поле вводу для створення нового об’єкту в обраній точці на карті.

д) Вікно відомостей про сеанси обміну даними між клієнтом та сервером, яке в реальному часі відображає всі дані, що надсилає та отримує клієнт.

Для тестування швидкодії програмного забезпечення прототипу використано ПЗ Apache JMeter, яке було сконфігуровано як показано на рисунку 3.20.

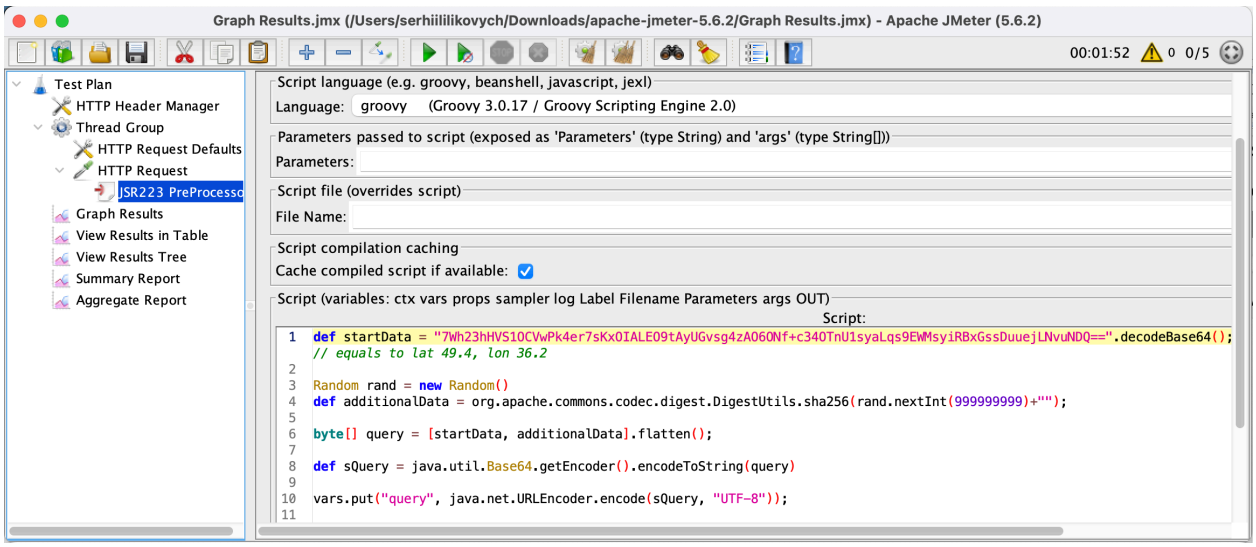


Рисунок 3.20 – Інтерфейс Apache JMeter в ході конфігурації тестування швидкодії

На рисунку 3.20 видно, що тестування складається з одної групи потоків (thread group), яка виконує HTTP-запити таким чином, що кожен наступний відрізняється від попередніх, за що відповідає спеціально створений об’єкт “JSR223 PreProcessor” – програма, яка генерує випадкові, але коректні “лазівки”. Група потоків сконфігурована таким чином, що імітує одночасне виконання пошукових запитів від імені п’яти користувачів. Кожен користувач виконує по 200 запитів, тобто сумарне навантаження досягає 1000 пошукових запитів. Графік пропускної здатності прототипу ПЗ показано на рисунку 3.21:

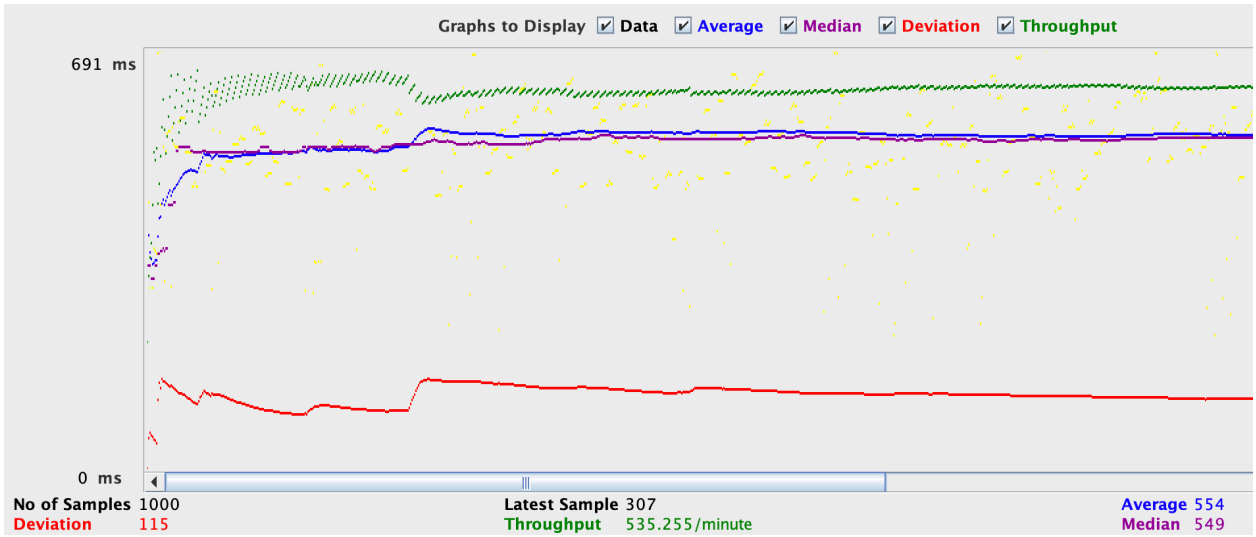


Рис. 3.21 – Графік пропускної здатності ПЗ прототипу, побудований за результатами тестування швидкодії Apache JMeter

Як можна бачити на рисунку 3.21, пропускна здатність ПЗ прототипу – 535.26 запити на хвилину, а отже приблизно 8.9 запитів на секунду, або 0.11 секунд на один запит, що задовольняє вимогам щодо швидкодії прототипу, висунутим в підрозділі 3.1.2.

## 3.9 Висновки за розділом

1) Сформульовані функціональні та нефункціональні вимоги до програмного забезпечення, з урахуванням висунутих вимог щодо безпеки даних в ПС.

2) Визначено архітектуру та інструментальні засоби для реалізації ПЗ.

3) Розроблено програмне забезпечення прототипу для запропонованого методу SE.

4) Показано процеси інсталяції та розгортання прототипу ПЗ.

5) Дано оцінку продуктивності запропонованого рішення. Середній час обробки запиту під час тестування швидкодії склав 0.11 с.

# 4 АНАЛІЗ РЕЗУЛЬТАТІВ РОБОТИ

## 4.1 Рекомендації щодо використання запропонованого методу

Запропонований метод пошуку в зашифрованих БД має такі характеристики:

– За типом використаного криптографічного перетворення: хеш-функція.

– За типом індексу: обернений хмарний індекс у вигляді префіксного дерева.

– За типом пошуку: точний пошук за одним або більше ключовим словом (префіксом) з використанням “лазівки”.

Можна виділити такі переваги використання запропонованого методу:

– Можливість пошуку за префіксом, а не ключовими словами, що дозволяє задовільнити потребу в реалізації такого методу SE для даного варіанту використання ПС, і при цьому застосувати ефективні структури даних пошукового індексу. Так, використана в даній роботі структура префіксного дерева дозволяє виконувати пошук за $O(dm)$, де $d$ – розмір алфавіту, а $m$ – довжина запиту, тоді як звичайний пошук за списком ключових слів має складність $O(n)$.

– Набуває фізичного сенсу, якщо шуканими префіксами є числові значення, які відбивають певні атрибути так, що прибирання одного або декількох розрядів або байтів такого значення має сенс в предметній області. Наприклад, це можуть бути десяткові координати (тоді прибирання молодших розрядів розширює область, в якій може знаходитись об’єкт), або результати медичних тестів (в такому випадку пошук за префіксом дозволяє обрати певну групу пацієнтів без необхідності явного формулювання визначення цієї групи в пошуковому запиті).

Запропонований метод має такі недоліки:

– Не повністю захищений від атаки «відкритого шифротексту» – отримавши доступ до пошукового індексу, зловмисник може встановити взаємозв’язок між документами. Однак при використанні запропонованого методу власник даних може на свій розсуд обирати глибину дерева, через яке його документ буде пов’язаним з префіксами та іншими документами в індексі, і таким чином регулювати обсяг можливого розкриття інформації.

– Високі накладні витрати на зберігання пошукового індексу – на кожен черговий байт префіксу створюється вершина в префіксному дереві з довжиною, рівною довжині блоку використаної криптографічної хеш-функції (в прототипі це 32 байти на кожен байт префіксу).

Таким чином, стає зрозуміло, що запропонований метод є компромісним з точки зору безпеки даних. Він розкриває певну частину метаданих документів, так само як це передбачається в методах PPE, проте запропонований метод дозволяє власнику даних самостійно регулювати обсяги розкриття метаданих.

З огляду на наведені вище характеристики методу, його використання може бути доцільним в ПС, де пошук за префіксом є функціональною вимогою, та компроміс щодо розкриття зв’язків між документами є припустимим, наприклад:

– В системах з предиктивним введенням (прізвищ, номерів телефонів, або документів тощо), де факт наявності спільного префіксу у записів є очевидним.

– В геоінформаційних системах, де розкриття наявності спільного префіксу у координат об’єктів розкриває лише факт їх присутності в певному квадраті, розмір якого визначає власник даних.

– В системах, що зберігають платіжні реквізити, наприклад номери карток, де потенційне розкриття приналежності записів до певної групи може розглядатися як припустимий компроміс.

## 4.2 Апробація результатів роботи

Протягом всього часу дослідження результати роботи були представлені в рамках VIII міжнародної науково-технічної конференції «Захист інформації і безпека інформаційних систем» [16], а також в рамках IX міжнародної науково-технічної конференції «Комп’ютерне моделювання у наукоємних технологіях» [17].

# ВИСНОВКИ

1) В ході роботи були проаналізовані методи пошуку в зашифрованих базах даних, принципи, що лежать в їх основі, та їх особливості. За допомогою отриманої інформації запропоновано метод пошуку даних за префіксом в зашифрованих базах даних, що використовує префіксне дерево в якості пошукового індексу.

2) Для розробленого методу виконано моделювання загроз за допомогою моделювання потоків даних. Формалізовані процеси пошуку та додавання в пошуковий індекс у вигляді псевдокоду та діаграм IDEF0.

3) Відповідно до розробленого методу сформульовані функціональні та нефункціональні вимоги до прототипу програмного забезпечення методу. Виходячи з них обрані інструментальні засоби реалізації, запропонована архітектура програмної системи, сформульована логічна модель даних.

4) Розроблено прототип програмного забезпечення методу для випадку, коли префікс набуває фізичного значення – уточнює місце певного об’єкта на карті.

5) Після реалізації прототипу програмного забезпечення, всі його компоненти були розгорнуті, його працездатність та відповідність раніше сформульованим вимогам емпірично протестовані. Середній час обробки запиту склав 0.11 с.

6) Для запропонованого методу були сформульовані рекомендації щодо його застосування – метод доцільно використовувати в системах, де пошук за префіксом є функціональною вимогою, а потенційне розкриття взаємозв’язку між документами через пошуковий індекс є припустимим.

7) Надалі необхідно забезпечити тестування швидкодії даної реалізації методу на великому обсязі даних, порівняння швидкодії з відомими схемами SE, реалізувати операції оновлення та видалення вузлів дерева пошукового індексу.

# ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ


1. Fredkin E. Trie memory. Communications of the ACM. 1960. Vol. 3, no. 9. P. 490–499. URL: https://doi.org/10.1145/367390.367400 (date of access: 24.11.2023).
2. Knuth D. Art of computer programming. Pearson Education, Limited. Vol. 3 : Sorting and searching.
3. Dawn Xiaoding Song, Wagner D., Perrig A. Practical techniques for searches on encrypted data. 2000 IEEE symposium on security and privacy, Berkeley, CA, USA. URL: https://doi.org/10.1109/secpri.2000.848445 (date of access: 24.11.2023).
4. Goh E.-J. Secure indexes. Cryptology ePrint Archive. URL: https://eprint.iacr.org/2003/216 (date of access: 24.11.2023).
5. Accelerated search over encrypted cloud data / F. Boucenna et al. 2017 IEEE international conference on big data and smart computing (bigcomp), Jeju Island, South Korea, 13–16 February 2017. 2017. URL: https://doi.org/10.1109/bigcomp.2017.7881734 (date of access: 24.11.2023).
6. Chang Y.C., Mitzenmacher M. Privacy preserving keyword searches on remote encrypted data. Applied cryptography and network security. Berlin, Heidelberg, 2005. P. 442–455. URL: https://doi.org/10.1007/11496137_30 (date of access: 24.11.2023).
7. Handa R., Krishna C. R., Aggarwal N. Searchable encryption: a survey on privacy-preserving search schemes on encrypted outsourced data. Concurrency and computation: practice and experience. 2019. P. e5201. URL: https://doi.org/10.1002/cpe.5201 (date of access: 24.11.2023).
8. Amorim I., Costa I. Leveraging searchable encryption through homomorphic encryption: a comprehensive analysis. Mathematics. 2023. Vol. 11, no. 13.

P. 2948. URL: https://doi.org/10.3390/math11132948 (date of access: 24.11.2023).

9. Public key encryption with keyword search / D. Boneh et al. Advances in cryptology - EUROCRYPT 2004. Berlin, Heidelberg, 2004. P. 506–522. URL: https://doi.org/10.1007/978-3-540-24676-3_30 (date of access: 24.11.2023).
10. Örencik C., Savaş E. An efficient privacy-preserving multi-keyword search over encrypted cloud data with ranking. Distributed and parallel databases. 2013. Vol. 32, no. 1. P. 119–160. URL: https://doi.org/10.1007/s10619-013-7123-9 (date of access: 24.11.2023).
11. OWASP DevSecOps Guideline | OWASP Foundation. OWASP Foundation, the Open Source Foundation for Application Security | OWASP Foundation. URL: https://owasp.org/www-project-devsecops-guideline/ (date of access: 24.11.2023).
12. Sommerville I. Software engineering. Pearson, 2015. 816 p.
13. Kruchten P. Rational unified process: an introduction. Pearson Education, 2001. 320 p.
14. Buschmann F. Pattern-oriented software architecture: a system of patterns. Chichester : Wiley, 1996. 457 p.
15. Booch G. The unified modeling language user guide. 2nd ed. Upper Saddle River, NJ : Addison-Wesley, 2005. 475 p.
16. Лілікович С. О., Єсін В. І. Підхід до шифрування з можливістю пошуку числових даних. Захист інформації і безпека інформаційних систем : Матеріали IX міжнар. науково-техн. конф., м. Львів, 25–26 трав. 2023 р. Львів, 2023. С. 77–78.
17. Лілікович С. О., Єсін В. І. Метод пошуку за префіксом в зашифрованих базах даних. Комп’ютерне моделювання в наукоємних технологіях : Зб. наук. пр. міжнар. науково-техн. конф., м. Харків, 25–27 жовт. 2023 р.

# Додаток А

## Програмна реалізація компонента, що реалізує алгоритми роботи з пошуковим індексом у вигляді префіксного дерева

*Лістинг А.1.*

```
from typing import List

from bson.objectid import ObjectId
from bson.typings import _DocumentType
from pymongo.collection import Collection


class PrefixTree:
    _prefix_tree: Collection

    def __init__(self, tree: Collection, dummy_root_id: ObjectId):
        self._prefix_tree = tree
        self._dummy_root = tree.find_one({'_id': dummy_root_id})

    def get_collection(self):
        return self._prefix_tree

    def get_dymmyroot(self):
        return self._dummy_root


    def search(self, root: _DocumentType, query: List[bytes], max_depth_overhead: int = 1, depth: int = 0):
        result = []

        # max_depth_overhead adjusts maximum possible search depth (after the query ends)
        if (depth - len(query)) > max_depth_overhead:
            return result

        # add current branch document IDs to result set
        if depth >= len(query) and 'documents' in root and len(root['documents']) > 0:
            for doc in root['documents']:
                result.append(doc)

        # check all the children
        for child in root['children']:
            child_obj = self._prefix_tree.find_one({'_id': child})
           if child_obj is None:
```

```
                    continue  # something went wrong, cannot find
this child object, skip
                if  depth  <  len(query)  and  query[depth]  !=
child_obj['hash']:
                    continue  # this branch is invalid, do not
look into it
                else:
                    # look into this branch
                    result.extend(self.search(child_obj,  query,
max_depth_overhead, depth + 1))

            return result

        def get_all_children(self, vertex: _DocumentType):
            return         self._prefix_tree.find({'_id':{'$in':
vertex['children']}})

        def   insert(self,   path:   List[bytes],   document_id:
ObjectId):
            inserted = []
            updated = []

            current_root = self._dummy_root
            current_depth = 0
            # 1. loop through each prefix tree item, look for
existing ones
            for depth in range(0, len(path)):
                found = False

                for                    child                    in
self.get_all_children(current_root):
                    if child['hash'] == path[depth]:
                        # found next corresponding child. Mark it
as current root and continue search
                        current_root = child
                        found = True
                        current_depth += 1
                        continue
                # if there is no corresponding child on the
current depth - interrupt the search
                if not found:
                    break
            try:
                # 2. create new prefix tree vertices if needed
                # 2.a need to append prefix tree with new vertices
                if current_depth < len(path):
                    for k in range(current_depth, len(path)):
                        # prepare and insert new child element
data, including target document ID on the last iteration

                        child_data          =          {'hash':
path[current_depth], 'children':[]}
```

```
                        if k+1 == len(path):
                            child_data['documents']             =
[document_id]
                        child                                   =
self._prefix_tree.insert_one(child_data)
                        inserted.append(child.inserted_id)

                        # add this child element to index
                        self._prefix_tree.update_one(
                            {'_id': current_root['_id']},
                            {'$push':               {'children':
child.inserted_id}},
                            upsert=False
                        )
                        updated.append({'_id':
current_root['_id'], 'child': child.inserted_id})
                        current_root                            =
self._prefix_tree.find_one({'_id': child.inserted_id})
                        current_depth += 1
                else:
                    # 2.b no need to append the prefix tree with
new vertices, just insert new data
                    self._prefix_tree.update_one(
                        {'_id': current_root['_id']},
                        {'$push': {'documents': document_id}},
                        upsert=False
                    )
                    updated.append(current_root['_id'])
            except Exception as e:
                # in case of error remove all created records and
rollback updates
                self._prefix_tree.update_many(
                    {   '_id':   {'$in':   [x['_id']   for   x   in
updated]}},
                    {'$pull': {'children': {'$in': [x['child']
for x in updated]}}}
                )
                self._prefix_tree.delete_many({'_id':    {'$in':
inserted}})
                raise e

            return [inserted, updated]
```