НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

«На правах рукопису»
УДК 004.75

До захисту допущено:
Завідувач кафедри
_____ Сергій СТІРЕНКО
«___»_____2024 р.

Магістерська дисертація

на здобуття ступеня магістра

за освітньо-професійною програмою «Комп'ютерні системи та мережі»

зі спеціальності 123 «Комп'ютерна інженерія»

на тему: «Підсистема захисту розподіленого сховища даних»

Виконав:
студент 2 курсу, групи ІО-21мп
Андрухович Микола Андрійович _____

Керівник:
доц., к.т.н., доц.,
Волокита Артем Миколайович _____

Консультант з нормоконтролю:
проф., д.т.н., проф.,
Кулаков Юрій Олексійович _____

Рецензент:
доц., к.т.н., доц.,
Шимкович Володимир Миколайович _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2024 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет *Інформатики та обчислювальної техніки*
(повна назва)

Кафедра *Обчислювальної техніки*
(повна назва)

Рівень вищої освіти – другий (магістерський)

Спеціальність *123. Комп'ютерна інженерія*
(код і назва)

Освітньо-професійна програма *Комп'ютерні системи та мережі*
(код і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
«_____» _____ 2024 р.

ЗАВДАННЯ
на магістерську дисертацію студенту
Андруховичу Миколі Андрійовичу
(прізвище, ім'я, по батькові)

1. Тема дисертації Підсистема захисту розподіленого сховища даних

Науковий керівник дисертації Волокита Артем Миколайович, доц. каф. ОТ
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «29» травня 2024 р. № ___

2. Строк подання студентом дисертації 05.06.2024

3. Об'єкт дослідження підсистема захисту розподіленого сховища даних

4. Предмет дослідження сукупність методів, підходів, технологій і засобів, які використовуються для захисту розподілених сховищ даних

5. Перелік завдань, які потрібно розробити: провести аналіз існуючих методів, підходів, технологій і засобів захисту розподілених сховищ даних, розробити нові методи, підходи, технології і засоби захисту розподілених сховищ даних, провести експериментальні дослідження нових методів, підходів, технологій і засобів захисту розподілених сховищ даних

6. Консультанти розділів дисертації:

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормконтроль	Кулаков Ю.О. д.т.н., професор		
1.	Кулаков Ю.О. д.т.н., професор		
2.	Кулаков Ю.О. д.т.н., професор		
3.	Кулаков Ю.О. д.т.н., професор		
4.	Кулаков Ю.О. д.т.н., професор		

7. Дата видачі завдання 1.09.2023

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Строк виконання етапів дисертації	Примітка
1.	Затвердження теми дослідження. Визначення предмету дослідження	1.09.23-7.09.23	
2.	Дослідження існуючих рішень та проблем захисту даних в досліджуваній області	8.09.23-21.09.23	
3.	Побудова та обґрунтування архітектурного рішення	22.09.23-28.09.23	
4.	Реалізація системи.	06.10.23-19.10.23	
5.	Впровадження модуля шифрування даних, авторизації, двофакторної автентифікації та SSO.	20.10.23-09.11.23	
6.	Тестування підсистеми захисту та аналіз її ефективності.	10.11.23-13.11.23	
7.	Розробка стартап-проекту	14.11.23-18.11.23	
8.	Оформлення магістерської дисертації.	19.11.23-24.11.23	

Студент Микола АНДРУХОВИЧ _____
(підпис)

Науковий керівник дисертації Артем ВОЛОКИТА _____
(підпис)

РЕФЕРАТ

на магістерську дисертацію

виконану на тему: Підсистема захисту розподіленого сховища даних

студентом: Андруховичем Миколою Андрійовичем

Робота складається із вступу та чотирьох розділів. Загальний обсяг роботи: 100 аркушів основного тексту, 27 ілюстрацій, 22 таблиці. При підготовці використовувалася література з 43 різних джерел.

Актуальність. Сучасні інформаційні системи генерують величезні обсяги даних, що вимагають ефективних та надійних методів їх зберігання та обробки. Розподілені сховища даних стають все більш популярними завдяки своїй здатності обробляти великі обсяги інформації, забезпечуючи масштабованість та високу доступність. Зі збільшенням обсягів даних зростає й кількість загроз, пов'язаних з їхньою безпекою. Несанкціонований доступ, витік даних та кібератаки можуть призвести до значних фінансових втрат, а також втрати репутації компаній. Впровадження надійної підсистеми захисту є критично важливим для забезпечення конфіденційності, цілісності та доступності даних.

Багато країн прийняли закони та регуляції, що вимагають від організацій забезпечення певного рівня захисту даних. Виконання цих вимог є обов'язковим для уникнення штрафів та юридичних санкцій. Розробка підсистеми захисту, яка відповідає цим стандартам, дозволяє організаціям бути у відповідності з законодавством. Сучасні технології, такі як шифрування даних, двофакторна автентифікація, SSO та використання хмарних сервісів (наприклад, AWS S3), надають нові можливості для покращення захисту даних. Інтеграція цих технологій у розподілені сховища даних забезпечує більш високий рівень безпеки та надійності. Бізнеси все більше покладаються на дані для прийняття рішень, аналітики та стратегічного планування. Забезпечення безпеки цих даних є важливим аспектом для стабільної та успішної роботи організацій. Підсистема захисту дозволяє бізнесам

зосередитися на своїх ключових завданнях, не турбуючись про безпеку своїх даних.

Мета і завдання дослідження. Метою даного дослідження є розробка та впровадження ефективної підсистеми захисту розподіленого сховища даних, яка забезпечуватиме високий рівень безпеки, конфіденційності, цілісності та доступності даних за допомогою сучасних технологій шифрування, автентифікації та хмарних рішень.

Для досягнення мети дослідження поставлено і вирішено такі завдання:

- Вивчити існуючі підходи та рішення для захисту даних у розподілених сховищах.
- Провести аналіз проблем та недоліків існуючих методів захисту даних.
- Розробити архітектуру підсистеми захисту, яка включає шифрування даних, двофакторну автентифікацію, авторизацію та єдиний вхід.
- Впровадити модуль шифрування даних для забезпечення конфіденційності збереженої інформації.
- Реалізувати механізми двофакторної автентифікації та єдиного входу для підвищення безпеки доступу до даних.
- Інтегрувати AWS S3 як основний сервіс для зберігання файлів у розподіленому середовищі.
- Провести тестування розробленої підсистеми захисту та оцінити її ефективність в реальних умовах експлуатації.
- Підготувати рекомендації щодо подальшого вдосконалення та масштабування підсистеми захисту розподіленого сховища даних.

Об’єкт дослідження – розподілене сховище даних, яке використовується для зберігання та обробки великих обсягів інформації в умовах високих вимог до безпеки, конфіденційності, цілісності та доступності даних.

Предмет дослідження – методи та технології забезпечення безпеки в розподілених сховищах даних, зокрема шифрування даних, автентифікація

(включаючи двофакторну автентифікацію), авторизація, SSO (Single Sign-On) та використання хмарних сервісів для зберігання файлів.

Методи досліджень. Для досягнення поставлених у магістерській роботі задач використано методи криптографії, методи роботи з великими даними для обробки та аналізу великих обсягів даних з метою ефективного зберігання та доступу, методи аутентифікації та авторизації, що передбачають впровадження двофакторної автентифікації та єдиного входу.

Наукова новизна Наукова новизна одержаних результатів роботи полягає у наступному:

- запропоновано спосіб забезпечення безпеки даних у розподілених сховищах, що дозволяє покращити рівень захисту та надійності зберігання даних за рахунок застосування методів криптографії та двофакторної автентифікації;
- розроблено крос-платформену модель підсистеми захисту, яка включає шифрування даних, авторизацію, двофакторну автентифікацію та єдиний вхід, що може бути інтегрована в існуючі архітектури розподілених сховищ даних та забезпечувати високий рівень безпеки і доступності інформації.

Проведене дослідження дає змогу використання розробленої підсистеми захисту в розподілених сховищах даних та її застосування для покращення безпеки, конфіденційності та доступності збереженої інформації.

Особистий внесок здобувача. Магістерське дослідження є самостійно виконаною роботою, в якій відображено особистий авторський підхід та особисто отримані теоретичні та прикладні результати, що стосуються вирішення задачі захисту даних у розподілених сховищах. Формулювання мети та завдань дослідження проводилось спільно з науковим керівником.

Практична цінність. Отримані результати можуть використовуватися у майбутніх дослідженнях за напрямками:

- вдосконалення методів захисту даних у розподілених сховищах;
- розробка нових методів шифрування та автентифікації;
- вирішення задачі забезпечення безпеки в хмарних середовищах;

- розробка аналітичних підходів до забезпечення конфіденційності та цілісності даних.

Ключові слова

Розподілене сховище даних, шифрування, авторизація, автентифікація, двофакторна автентифікація, єдиний вхід.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

MVC	(Model-View-Controller) Архітектура для розробки програмного забезпечення
CRUD	(Create, Read, Update, Delete) Основні операції з базами даних
OTP	(One-Time Password) Тимчасовий пароль для одного використання
2FA	(Two-Factor Authentication) Посилений процес безпеки, двофакторна автентифікація
SSO	(Single Sign-On) Одиночний вхід для декількох програм
AWS	(Amazon Web Services) Платформа хмарних служб
S3	(Simple Storage Service) Зберігання об'єктів AWS
SSL	(Secure Sockets Layer) Шифрування Інтернет трафіку
HTML	(Hypertext Markup Language) Мова структури веб сторінок
URL	(Uniform Resource Locator) Веб-адреса
IV	(Initialization Vector) Випадкове значення для шифрування
API	(Application Programming Interface) Набір функцій для взаємодії програмного забезпечення

ВСТУП

Ми щодня стикаємося з потоком мільйонів даних, представлених у різних формах. У сфері бізнесу, управління та повсякденного життя зберігання цих даних виявляється критично важливим. Великі та успішні компанії зазвичай мають добре організовані системи зберігання даних, які забезпечують безпеку та легкість доступу до потрібної інформації. Системи зберігання даних пропонують надійне сховище, захищають від втрат або крадіжки даних та дозволяють створювати резервні копії. Ефективне керування зберіганням даних також сприяє економії простору та є більш вигідним порівняно з паперовим зберіганням чи на комп'ютерах.

Кожна система зберігання включає сховище даних, що охоплює зовнішню та внутрішню пам'ять, інтерфейс для взаємодії з користувачами для введення запитів та отримання відповідей, а також процесор для обробки цих запитів і надання відповідей.

Системи зберігання поділяються на централізовані, децентралізовані та розподілені. Розподілене зберігання даних, де інформація розміщується у різних базах даних у різних місцях, є однією з поширених форм. Фахівець у галузі баз даних Крістофер Дейт визначає ключові особливості розподілених систем як обробку розподілених запитів і транзакцій, прозорість мережі, автономність вузлів, безперервний доступ до даних, незалежність від локальних збоїв, легкість масштабування системи, мульти-мережний доступ і розподіл навантаження.

Безпека є важливим аспектом у впровадженні розподілених систем зберігання. Захист конфіденційної інформації, особливо коли дані розподілені між різними локаціями, вимагає дотримання юридичних та організаційних стандартів. Ефективна система безпеки є рішенням для надійного захисту та конфіденційності даних у розподіленій системі.

Ця дипломна робота складається з чотирьох розділів, в яких розглядаються різні типи систем зберігання даних, аналізуються їхні

особливості, переваги та недоліки, вибираються інструменти для реалізації системи безпеки розподіленого зберігання та описується її впровадження.

РОЗДІЛ 1
ТЕОРЕТИЧНІ ОСНОВИ РОЗПОДІЛЕНОГО СХОВИЩА ДАНИХ

1.1. Загальна характеристика системи сховища даних

З огляду на неймовірний прогрес та інновації у сфері технологій зберігання даних, які ми спостерігаємо в останні роки, система сховища даних набула нових граней та можливостей. Це справжній технологічний прорив, що дозволяє ефективно обробляти та зберігати величезні обсяги інформації. Ці системи не просто обмежуються структурованими чи неструктурованими даними, вони також охоплюють широкий спектр різноманітних видів архівів, медіафайлів, копій даних та інших форм, які можуть бути необхідні для сучасного бізнесу та інших сфер життя.

Однією з найважливіших тенденцій у цій сфері є поступове переходження від традиційних жорстких дисків до більш сучасних та вдосконалених SSD- та HDD-систем, які пропонують значно більшу швидкість і надійність. Ці рішення, особливо ті, що базуються на технології NVMe, забезпечують високу продуктивність завдяки скороченню часу доступу до даних та підвищенню їхньої пропускної спроможності.

У міру того, як малі та середні компанії розширюють свою діяльність, вони все частіше вдаються до використання більш гнучких і масштабованих хмарних рішень. Ці рішення дозволяють їм комбінувати переваги локального та хмарного зберігання, тим самим оптимізуючи свої процеси та знижуючи витрати. Великі компанії, яким необхідно зберігати обсяги даних, що лічаться петабайтами, також виявляють великий інтерес до таких рішень, оскільки вони забезпечують не тільки швидкість і надійність, а й необхідну масштабованість[1].

Системи сховища даних сьогодні включають в себе не лише носії інформації та системи управління даними, але й складні мережеві структури для передачі даних. Ці мережі, інтегровані з передовими хмарними технологіями, забезпечують високу гнучкість та ефективність у роботі з даними, незалежно від їхнього розташування.

Крім того, сучасні системи зберігання даних застосовують різноманітні підходи, включаючи блокове, файлове та об’єктне зберігання, кожен з яких відповідає певним потребам. Блокове зберігання ідеально підходить для вимогливих застосувань, таких як бази даних, завдяки своїй швидкості та ефективності. Файлове зберігання, з іншого боку, є відмінним вибором для довгострокового зберігання «холодних» даних, тоді як об’єктне зберігання є чудовим рішенням для великих медіафайлів та аналітичних даних, особливо у контексті машинного навчання та хмарних додатків.

Ці системи стають все більш розумними, вбудовуючи елементи машинного навчання та штучного інтелекту для оптимізації та автоматизації процесів зберігання та аналізу даних, що є важливим кроком у напрямку майбутнього, де дані є новою валютою успіху.

У контексті розподіленого сховища даних, яке стає все більш поширеним та важливим, особливо актуальним є питання ефективної координації та управління даними, розподіленими між численними фізичними та географічними локаціями. Сучасні технології розподіленого зберігання даних використовують принципи розподіленої обробки для підвищення швидкості доступу до даних та їх надійності. Це забезпечує розподіл навантаження та оптимізує використання ресурсів, що є особливо важливим для великих організацій, які працюють з великими масивами даних.

Ключовим аспектом розподіленого зберігання даних є його гнучкість та масштабованість. Завдяки розподіленій архітектурі, системи зберігання можуть легко розширюватися, додаючи нові вузли та ресурси без необхідності перебудови всієї системи. Це дозволяє організаціям гнучко реагувати на зміни в потребах зберігання та обробки даних, а також забезпечує високу доступність даних, навіть у разі збоїв окремих компонентів системи.

Однією з найважливіших переваг розподіленого зберігання є його стійкість до збоїв та помилок. Дані автоматично реплікуються між різними вузлами, що забезпечує високий рівень довговічності та доступності. Це

особливо важливо для критичних систем, де довготривала втрата доступу до даних може мати серйозні наслідки.

Крім того, розподілене зберігання даних сприяє підвищенню ефективності обробки даних. Використання розподілених обчислень дозволяє обробляти запити та проводити аналітику біля самого джерела даних, мінімізуючи затримки та оптимізуючи використання мережевих ресурсів. Це важливо для застосувань, які потребують швидкої відповіді в реальному часі, наприклад, для аналізу потокових даних або великих обсягів транзакцій.

1.2. Види сховищ даних

Залежно від їх структури, сховища даних можуть бути класифіковані як розподілені, централізовані або децентралізовані (рис 1.1).

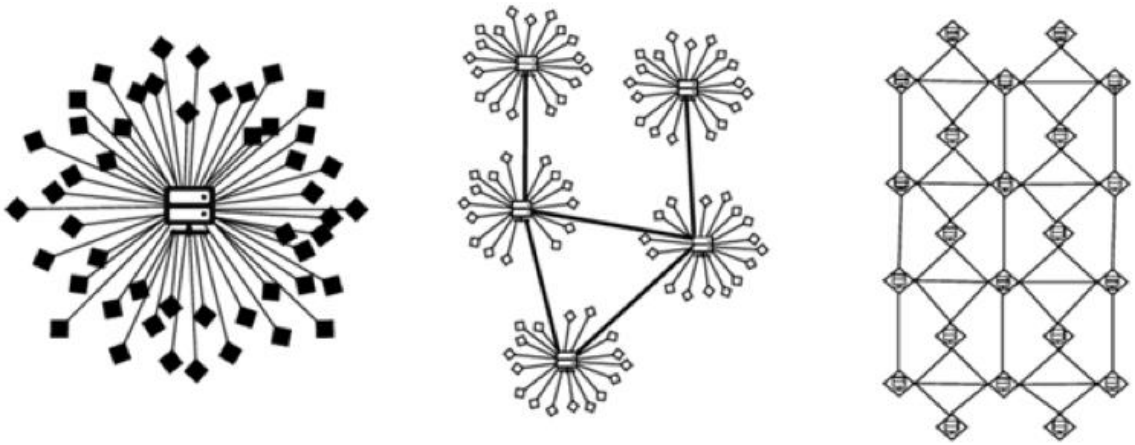

Рисунок 1.1 - Відображення розміщення вузлів у централізованому (а), децентралізованому (б) та розподіленому (в) сховищах даних

У розподілених сховищах даних інформація знаходиться на пристроях, розміщених у різних локаціях. Звернення до цих даних відбувається через глобальну мережу. Управління таким сховищем відбувається за допомогою центральної бази даних, хоча вона не пов'язана безпосередньо з центральним

процесором. Реплікація та дублювання даних у такій системі гарантують, що всі розподілені бази даних містять оновлену і однакову інформацію[2].

Централізовані сховища даних концентрують всю інформацію в єдиному місці, яким може бути, наприклад, сервер або мейнфрейм. Користувачі отримують доступ до даних через спеціалізовані додатки, підключені до глобальної мережі. Така концентрація даних спрощує процеси створення резервних копій і підтримання актуальності та цілісності інформації.

Децентралізовані сховища розподіляють навантаження між різними обчислювальними пристроями, які розташовані в різних місцях. Вони використовують складні алгоритми для оптимізації обробки вхідних та вихідних запитів, забезпечуючи оптимальний час відповіді. Цей тип системи є ідеальним, коли обсяги даних перевищують можливості одного обчислювального пристрою.

1.2.1. Централізоване сховище даних

Сховище даних із централізованою структурою базується на принципі єдиного місця для збору та управління всією інформацією. Всі зміни в даних та їх адміністрування виконуються через централізований механізм управління, який, як правило, є центральним комп'ютером. Централізоване сховище опрацьовує дані на рівні окремих блоків чи файлів[3].

Ключові особливості централізованого сховища даних включають:

• Єдиний центральний вузол – сервер, що об'єднує усі сторони, які взаємодіють з даними.

• Синхронізація часу через загальну часову вісь, що дозволяє усім компонентам системи працювати у координації з центральним годинником.

• Загальна залежність від стабільності центральної бази даних, що створює ризик сукупного збою: у разі виникнення технічних проблем з сервером, доступ до всіх компонентів системи може бути призупинений.

Сховище даних з централізованою конфігурацією пропонує цілий ряд переваг, які сприяють її широкому розповсюдженню для різноманітних застосувань. Ось деякі ключові плюси такого підходу до зберігання інформації:

• Висока цілісність даних: оскільки всі інформаційні ресурси сконцентровано в одному місці, це значно спрощує процеси управління ними. Інформація відзначається високою точністю та консистенцією, оскільки всі зміни та оновлення впроваджуються централізовано, що забезпечує єдиний і точний джерело даних для всіх запитів.

• Підвищена безпека: концентрація даних в єдиному локаційному вузлі дозволяє зосередити зусилля на захисті цієї критичної точки. Застосування комплексних механізмів шифрування та моніторингу стає більш простим і ефективним, знижуючи ризик несанкціонованого доступу або інших безпекових інцидентів.

• Зручність у доступі та управлінні: централізовані системи забезпечують легкість у координації та доступі до інформації, зокрема великих обсягів даних, які можуть бути легко керовані та аналізовані з єдиного місця.

• Економічність у обслуговуванні: з огляду на централізацію ресурсів, управління централізованим сховищем даних може бути більш економічно вигідним, адже необхідно підтримувати та обслуговувати меншу кількість обладнання. Також це веде до оптимізації енергоспоживання і зниження витрат на технічне обслуговування.

• Мінімізація надлишковості даних: в централізованому сховищі даних усі записи зберігаються в одному місці, що унеможливлює їх невиправдане розмноження та зайве зайняття простору. Це забезпечує кращий контроль над даними та їх ефективне використання.

• Простота міграції даних: з усіма даними, що знаходяться в одному місці, процес їх перенесення, будь то для резервного копіювання або відновлення, стає більш лінійним та менш складним.

• Швидкість передачі та отримання даних: порівняно з розподіленими системами, централізоване сховище часто дозволяє швидший доступ до інформації, оскільки всі дані розташовані в одному місці.

Незважаючи на перераховані переваги, централізоване сховище має і свої слабкі сторони. Через те, що всі дані розміщені в одній точці, процеси пошуку та доступу можуть бути більш часомісткими, особливо при великому обсязі одночасних запитів. Велике навантаження на сервер може призвести до зниження загальної продуктивності системи, а також до підвищення ризику, що в разі збою сервера, може бути втрачено всі дані.

Централізоване сховище даних надає змогу користувачам мати повний огляд доступної інформації, проте слід бути уважними до питань швидкості доступу та відновлення після можливих збоїв. У цілому, вибір між централізованим сховищем та іншими формами зберігання повинен базуватися на конкретних потребах компанії, обсягах даних, які необхідно зберігати, та вимогах до доступу до інформації. Незважаючи на очевидні вигоди централізації, важливо також враховувати потенційні ризики і витрати, пов'язані з підтримкою такої системи, та розробляти стратегії для ефективного управління та запобігання збоям, які можуть мати критичні наслідки для всієї організації.

1.2.2. Децентралізоване сховище даних

Сховище даних із децентралізованою архітектурою розподіляє обробку даних між численними вузлами, які забезпечують рівномірне розподілення навантаження. За допомогою розроблених алгоритмів оптимізації запитів, цей тип сховища спрямований на забезпечення ефективного часу реакції на запити користувачів.

Використання децентралізованого сховища даних має декілька ключових переваг, які роблять його привабливим вибором для збереження інформації:

• Можливість збільшення обсягів зберігання: децентралізоване сховище дозволяє розширювати простір для даних, перевищуючи обмеження одного фізичного пристрою.

• Ефективний розподіл робочого навантаження: завдяки розподіленню даних, система здатна підтримувати високу працездатність і швидкість реакції, уникаючи перевантажень.

• Резервне копіювання даних: це гарантує безпеку інформації, забезпечуючи її відновлення у випадку технічних збоїв.

• Забезпечення конфіденційності: розподілення даних по різних пристроях знижує ризик їх витоку чи несанкціонованого доступу.

• Підвищення продуктивності взаємодії між відділами та ланками виробництва: швидкий доступ до інформації з будь-якої точки децентралізованої системи сприяє оперативності виробничих процесів.

Вузли в децентралізованому сховищі даних функціонують без єдиного центрального сховища, тому інформація рівномірно розподіляється між ними[4]. Якщо на одному з комп'ютерів системи відбуваються зміни, вони автоматично синхронізуються з іншими, забезпечуючи актуальність даних по всій мережі. Ця модель також забезпечує високу стійкість системи до зовнішніх загроз та неавторизованих змін, роблячи її самодостатньою і саморегулюючою.

1.2.3. Розподілене сховище даних

Сховище даних у розподіленій системі представляє собою мережу взаємозалежних баз даних, які розміщені на різних вузлах цієї системи. У цьому випадку інформація розподілена між декількома базами даних, які знаходяться в різних фізичних локаціях, але залишаються зв'язаними одна з одною, дозволяючи здійснювати незалежне управління збереженими даними.

Програмне забезпечення для управління розподіленим сховищем даних дозволяє організувати та координувати роботу з розподіленою базою

даних, забезпечуючи при цьому прозорість взаємодії для користувачів. Однією з особливостей розподіленого сховища є відсутність єдиного глобального годинника, що означає, що кожен вузол системи працює згідно зі своїм власним часовим відліком.

Розподілене сховище даних являє собою комплекс багатьох баз даних, які логічно пов'язані між собою та розміщені на різних вузлах у межах розподіленої системи. Це такий тип сховища, де інформація розподіляється між декількома базами даних, розташованими в різних фізичних локаціях, але зв'язаними одна з одною, що дозволяє незалежно керувати збереженими даними[5].

Система управління розподіленим сховищем даних функціонує як спеціалізоване програмне забезпечення, яке дозволяє управляти розподіленими базами даних, роблячи використання системи простим та зручним для користувачів. Однією з особливостей розподіленого сховища є відсутність єдиного глобального годинника, оскільки кожен вузол системи використовує свій власний годинник.

Розподілене сховище даних має кілька переваг, включаючи можливість розширення обсягу зберігання даних, рівномірний розподіл робочого навантаження для підтримки стабільності системи та високу продуктивність завдяки розподілу навантаження між декількома вузлами. Ця система також відрізняється високою стійкістю до кібератак та забезпечує ефективне резервне копіювання інформації.

Однак, розподілене сховище даних також має свої недоліки, такі як складність обслуговування та висока вартість виправлення помилок, оскільки виявити проблемний вузол може бути складно, а їх виправлення може зайняти значний час.

У розподіленій обчислювальній системі працюють автономні процесорні елементи, які можуть відрізнятися за своїми характеристиками та способами зв'язку. Ці елементи не мають безпосереднього доступу до станів один одного, а обмінюються інформацією через повідомлення. Такий підхід

до управління даними в розподіленому сховищі вимагає особливої уваги та доповнюючого програмного забезпечення, яке дозволяє інтегрувати різні частини системи в єдину логічну структуру.

В розподіленому сховищі даних, відмінність полягає не лише в розміщенні файлів, але й у тому, що дані тісно взаємопов'язані, незалежно від їх фізичного розташування. Наприклад, у реляційних системах різні частини даних можуть зберігатися на різних сайтах, вимагаючи спеціальних операцій для з'єднання та об'єднання даних, які часто формулюються за допомогою SQL-запитів. У системах NoSQL, де взаємозв'язки між даними можуть бути менш жорсткими, наприклад у графових базах даних, вершини графа можуть зберігатися на різних сайтах, дозволяючи більш гнучку організацію даних.

Розподілені сховища даних можуть бути як географічно розподіленими, так і розташованими в одному місці. Георозподілені системи пов'язані між собою через глобальні мережі, що може призвести до більш тривалих затримок у відповідях і вищих ризиків помилок. Системи, розташовані в одному місці, зазвичай характеризуються коротшим часом обміну даними та нижчим рівнем помилок завдяки близькості процесорних елементів. Це дозволяє забезпечити більш ефективну та швидку обробку запитів.

Загалом, розподілене сховище даних пропонує унікальну комбінацію гнучкості, ефективності та безпеки. Його здатність до розширення, висока продуктивність та стійкість до кібератак роблять його ідеальним вибором для організацій, які потребують надійного та масштабованого рішення для зберігання великих обсягів даних. Водночас, потрібно враховувати складність управління та обслуговування розподіленої системи, особливо при виявленні та виправленні помилок у різних вузлах мережі.

1.3. Особливості розподіленого сховища даних

Розподілене сховище даних класифікується на основі виду використовуваного програмного забезпечення на однорідні та неоднорідні сховища.

В однорідних розподілених сховищах даних кожен вузол використовує однакові системи управління базами даних з ідентичними функціональними можливостями. Це також включає єдиний підхід до вибору техніки та програмного забезпечення на всіх вузлах[6].

У неоднорідних розподілених сховищах даних використовуються різні типи систем управління базами даних з відмінними функціями на кожному вузлі. Також можливе використання різного програмного забезпечення та обладнання в різних частинах системи.

Для ефективного функціонування розподілених сховищ даних застосовуються два основні механізми: фрагментація та реплікація. Фрагментація дозволяє розбити дані на окремі частини, що розподіляються між різними вузлами, в той час як реплікація забезпечує створення копій даних на різних вузлах для підвищення надійності та доступності інформації.

1.3.1. Фрагментація

Фрагментація у розподілених сховищах даних полягає у розділенні інформації на окремі частини, які потім розміщуються на різних вузлах мережі. Існують два основних види фрагментації: горизонтальна та вертикальна. Горизонтальна фрагментація використовує оператор вибору для розділення даних на основі рядків або кортежів, у той час як вертикальна фрагментація застосовує оператор проекту для розділення даних за колонками. Часто ці два види фрагментації можуть бути поєднані, утворюючи гібридну фрагментацію, яка включає елементи обох видів.

Горизонтальна фрагментація є особливо поширеною, особливо в паралельних сховищах даних[7]. Ця фрагментація дозволяє розподілити кортежі або рядки даних між різними вузлами, так що кожен фрагмент містить

підмножину рядків відношення. Такий підхід підтримує паралельну обробку запитів, що є ключовою перевагою для більшості сучасних платформ великих даних. Горизонтальна фрагментація ефективно використовується для забезпечення паралелізму у внутрішніх запитах, розбиваючи відношення по кортежах і розподіляючи їх між вузлами для оптимізації обробки даних.

Вертикальна фрагментація представляє собою більш складний процес порівняно з горизонтальною фрагментацією, особливо через велику кількість можливих варіантів поділу. Наприклад, у випадку горизонтальної фрагментації, якщо існує n простих предикатів, то кількість можливих мінтермних предикатів дорівнює 2^n. Проте, деякі з цих предикатів можуть бути суперечливими, що зменшує кількість потенційних фрагментів-кандидатів для поділу.

Існують два основні евристичні підходи до вертикальної фрагментації глобальних відносин:

1. Групування: цей підхід передбачає початкове присвоєння кожного атрибута до окремого фрагмента, а потім на кожному етапі відбувається об'єднання декількох фрагментів, поки не будуть досягнуті визначені критерії.
2. Розбиття: цей метод починається з повного відношення та базується на аналізі способів доступу програм до атрибутів, для визначення найбільш вигідних розділів.

Окрім цього, використовується також змішана фрагментація, що поєднує елементи вертикальної та горизонтальної фрагментацій. Методи систематичної фрагментації розроблені для забезпечення збереження семантики бази даних під час фрагментації, уникаючи втрати даних. У контексті горизонтальної фрагментації важливою властивістю є взаємна відокремленість фрагментів, що гарантує, що кожен фрагмент містить унікальну підмножину даних.

1.3.2. Реплікація

Реплікація - це механізм розподіленого зберігання даних, який передбачає зберігання копій даних на різних сховищах[8].

Реплікація має ряд переваг, зокрема:

1. Доступність: у разі відмови одного сховища дані будуть доступні з інших.
2. Продуктивність: дані можуть бути доступні з більш близьких до точки доступу сховищ, що скорочує час відповіді.
3. Масштабованість: реплікація дозволяє підтримувати зростання системи з прийнятним часом відповіді.
4. Вимоги до застосування: деякі програми можуть вимагати зберігання декількох копій даних.

Однак реплікація також має ряд проблем, зокрема:

1. Синхронізація: необхідно забезпечити, щоб всі копії даних були узгоджені.
2. Узгодженість: необхідно забезпечити, щоб значення даних були узгоджені між різними копіями.

Існує два основних види узгодженості реплікованої бази даних:

1. Взаємна узгодженість: значення даних на всіх копіях повинні бути однаковими.
2. Узгодженість транзакцій: значення даних на всіх копіях повинні бути узгодженими після виконання транзакції.

Взаємна узгодженість є більш сильною вимогою, ніж узгодженість транзакцій.

1.4. Переваги розподілених сховищ даних

Розподілена архітектура сховища даних володіє низкою значущих переваг. Чотири ключові аспекти цього підходу включають: прозорість у керуванні розподіленими та реплікованими даними, забезпечення надійного

доступу до даних через розподілені транзакції, поліпшення продуктивності та спрощення процесу розширення системи[9].

1. Прозорість у керуванні розподіленими та реплікованими даними. Прозорість тут означає відокремлення вищих рівнів системи від складностей, що існують на нижчих рівнях. Це означає, що кінцеві користувачі залишаються в невіданні щодо складностей реалізації системи. Така прозорість є надзвичайно корисною для створення складних додатків, оскільки вона підвищує рівень підтримки. Прозорість у розподілених сховищах даних розширює ідею незалежності даних, що існує у централізованих системах. Ця прозорість охоплює кілька аспектів:

• Незалежність даних: це стосується того, що зміни в структурі або організації даних не впливають на користувацькі програми. Ця незалежність поділяється на логічну та фізичну. Логічна незалежність стосується нечутливості програм до змін у логічній структурі (схемі) бази даних, тоді як фізична незалежність означає, що програми не відчувають змін у фізичній структурі сховища даних.

• Прозорість мережі: користувачі захищені від деталей мережевих операцій та структури, що зв'язує різні вузли. Цей аспект можна поділити на прозорість розташування (запити не залежать від місця зберігання даних) та прозорість іменування (унікальне іменування для кожного об'єкта в базі даних, незалежно від його розташування).

• Прозорість фрагментації: це стосується поділу даних бази на менші частини для оптимізації продуктивності, доступності та надійності. Фрагментація дозволяє кожному фрагменту функціонувати як незалежний об'єкт, що сприяє більш ефективній обробці.

• Прозорість реплікації: означає можливість розподіленого зберігання даних по мережі, при цьому система самостійно керує реплікаціями, дозволяючи користувачам взаємодіяти з даними, ніби вони існують в одному екземплярі.

2. Надійний доступ до даних через розподілені транзакції. Розподілені сховища даних забезпечують високу стійкість до збоїв, оскільки вони містять репліковані компоненти, що ефективно усувають окремі точки збою. У разі відмови одного вузла чи зв'язку, система продовжує функціонувати, забезпечуючи доступ до інших частин бази даних. Ефективна робота розподілених транзакцій потребує розширеного контролю паралельності та розподілених протоколів відновлення, які значно складніші порівняно з централізованими системами.

3. Покращена продуктивність. Розподілені сховища даних підвищують продуктивність за рахунок розподілу даних, що дозволяє розташувати їх ближче до точок використання (місцевість даних), мінімізуючи затримки у віддаленому доступі. Це забезпечує дві основні переваги:

• Зниження навантаження на центральні ресурси, оскільки кожен вузол обробляє лише частину даних, що зменшує конкуренцію за використання центрального процесора та систем введення-виведення.

• Зменшення затримок, що пов'язані з віддаленим доступом до даних, особливо в глобальних мережах, де великі відстані можуть значно збільшувати час відгуку.

4. Масштабованість системи. У контексті розподілених середовищ сховищ даних, адаптування до зростаючих обсягів даних та збільшення робочих навантажень є значно спрощеним завданням. Це досягається за рахунок здатності легко додавати до системи додаткові ресурси обробки та зберігання даних. Поки що забезпечення лінійного зростання продуктивності системи може бути ускладнене через витрати, пов'язані з розподілом ресурсів, проте можливість значного підвищення продуктивності залишається відкритою[10]. Важливість розподілених сховищ даних особливо велика у контексті розробки масштабованих архітектур, таких як кластерні та хмарні обчислення. Горизонтальне масштабування, що передбачає додавання додаткових серверів (іноді називаних "серверами масштабування") у систему зі слабкими зв'язками між ними, відкриває шлях до майже необмеженого

розширення можливостей сховища даних. Завдяки простоті інтеграції нових серверів у базу даних, розподілені системи сховищ даних можуть ефективно адаптуватися до мінливих потреб користувачів та обсягів даних.

Висновки до розділу 1

У першому розділі розглядалися основні характеристики та типи систем сховища даних.

Системи сховища даних – це комплексні рішення, які включають програмне та апаратне забезпечення для ефективної обробки та зберігання значних масивів інформації. У загальному випадку, зберігання даних можна класифікувати на три основні рівні: блокове, файлове та об'єктне зберігання. Залежно від структури та архітектурних рішень, системи сховища даних можна поділити на три основні типи: розподілені, централізовані та децентралізовані.

1. Децентралізовані системи сховища даних розподіляють обробку та зберігання даних між різними вузлами. У таких системах немає єдиного центрального місця зберігання, тому дані розподіляються по мережі.

2. Централізовані системи сховища даних концентрують усю інформацію в одному місці, що полегшує управління та відновлення даних, але може створювати затримки при доступі до інформації через великі обсяги даних, що обробляються.

3. Розподілені системи сховища даних зберігають інформацію у різних фізичних локаціях, забезпечуючи її взаємопов'язаність та незалежне управління збереженими даними. Ці системи вирізняються високою продуктивністю завдяки розподілу навантаження по різних вузлах і високій стійкості до кібератак.

У контексті розробки системи безпеки було вибрано розподілену систему сховища даних за її продуктивність та надійність доступу до даних через розподілені транзакції. Проте ця система має недоліки у плані захисту

даних, адже вона не передбачає використання захищеного доступу за допомогою ключів безпеки. Окрім того, користувачі не мають можливості вибирати специфічні вузли для зберігання своїх даних.

Таким чином, у даній дипломній роботі було вирішено розробити систему безпеки для розподіленого сховища даних, що має на увазі удосконалення існуючої структури з метою забезпечення більшої безпеки даних. Це може включати реалізацію додаткових заходів захисту, таких як шифрування даних, використання безпекових ключів та вдосконалення механізмів аутентифікації і авторизації для забезпечення контролю доступу до даних. Такий підхід дозволить підвищити загальний рівень безпеки в розподіленій системі сховища даних, зберігаючи при цьому її ключові переваги, такі як висока продуктивність та надійність.

РОЗДІЛ 2
МЕТОДОЛОГІЯ РОЗРОБКИ ПІДСИСТЕМИ БЕЗПЕКИ РОЗПОДІЛЕНОГО СХОВИЩА ДАНИХ

У даній роботі було вирішено розробили веб-додаток з розподіленою базою даних. Веб-додаток було вирішено зробити простою системою написання дописів та отримання логів, які можна розподілити між вузлами бази даних.

Метою розробки такого додатку є можливість керування даними через користувацький інтерфейс, а також імплементація додаткових систем захисту отриманих даних. Задля вирішення поставленої задачі потрібно проаналізувати наявні інструменти розробки та обрати ті, що дадауть змогу якнайкраще реалізувати потрібну систему.

2.1. Вибір сховища даних

Для впровадження розподіленого ссховища даних потрібно обрати систему баз даних, що матиме усі необхідні функції та буде найкращим варіантом для розробки. Для вибору найкращої опції потрібно проаналізувати існуючі системи

2.1.1. PostgreSQL

PostgreSQL, часто відомий як Postgres, — це потужна об’єктно-реляційна база даних із відкритим вихідним кодом, широко визнана своєю надійністю, надійними функціями та продуктивністю. Завдяки широким можливостям він дуже підходить для розподіленого зберігання даних.

PostgreSQL чудово справляється з розподіленим зберіганням даних завдяки підтримці властивостей ACID (Atomicity, Consistency, Isolation, Durability), забезпечуючи надійну обробку транзакцій і цілісність даних, які є важливими для розподілених систем[11].

Однією з ключових сильних сторін PostgreSQL є її масштабованість і продуктивність. Він пропонує горизонтальну масштабованість за допомогою

таких методів, як розділення, сегментування та реплікація. Крім того, PostgreSQL підтримує паралельне виконання запитів, що може значно підвищити продуктивність, особливо з великими наборами даних.

З точки зору реплікації та високої доступності PostgreSQL надає кілька варіантів. Потокова реплікація дозволяє безперервно реплікувати дані в реальному часі на резервні сервери, забезпечуючи високу доступність і ефективне аварійне відновлення. Логічна реплікація додає гнучкості, дозволяючи вибіркову реплікацію даних і синхронізацію між різними серверами.

Функція зовнішніх обгорток даних (FDW) PostgreSQL дозволяє інтегрувати його з іншими джерелами даних. FDW дозволяють PostgreSQL запитувати та керувати даними із зовнішніх баз даних і сховищ даних, як якщо б вони були локальними таблицями, полегшуючи уніфіковане керування розподіленими даними.

База даних підтримує широкий спектр типів даних, включаючи JSON, XML і масиви, що робить її універсальною для різних програм. PostgreSQL також пропонує розширені механізми індексування, такі як B-дерева, хеш-індекси, GiST, SP-GiST, GIN і BRIN, які покращують продуктивність запитів для складних і великомасштабних наборів даних.

Безпека є значною перевагою PostgreSQL. Він включає надійні функції безпеки, такі як SSL/TLS для зашифрованих з’єднань, безпеку на рівні рядків і різні методи автентифікації (наприклад, MD5, SCRAM-SHA-256, Kerberos і LDAP), забезпечуючи захист конфіденційних даних у розподілених середовищах[12].

Іншим важливим аспектом PostgreSQL є його розширюваність. Користувачі можуть створювати власні функції, типи даних і розширення, дозволяючи розробникам налаштовувати базу даних відповідно до конкретних потреб. Ця гнучкість особливо корисна в розподілених системах, де можуть виникнути унікальні вимоги.

PostgreSQL підтримується сильною та активною спільнотою, яка постійно робить внесок у його розвиток. Спільнота надає розширену документацію, навчальні посібники та підтримку, гарантуючи, що PostgreSQL залишається в курсі останніх досягнень у технології баз даних[13].

Переваги PostgreSQL:

- Надійність: PostgreSQL відомий своєю стабільністю та дотриманням властивостей ACID, що забезпечує надійну обробку транзакцій і цілісність даних.
- Масштабованість: підтримує горизонтальну масштабованість за допомогою розділення, сегментування та реплікації, що дозволяє ефективно обробляти великі набори даних.
- Висока доступність: пропонує надійні параметри реплікації, такі як потокова передача даних і логічна реплікація для реплікації даних у реальному часі та ефективного аварійного відновлення.
- Інтеграція даних: обгортки зовнішніх даних (FDW) забезпечують інтеграцію з різними зовнішніми джерелами даних, забезпечуючи уніфікований перегляд розподілених даних і керування ними.
- Розширене керування даними: підтримує широкий спектр типів даних і вдосконалених механізмів індексування, підвищуючи універсальність і продуктивність запитів.
- Безпека: містить надійні функції безпеки, такі як SSL/TLS, безпека на рівні рядків і кілька методів автентифікації для захисту конфіденційних даних.
- Розширюваність: висока розширюваність, що дозволяє створювати власні функції, типи даних і розширення, адаптовані до конкретних потреб.
- Активна спільнота: сильна та активна підтримка спільноти з обширною документацією та безперервним внеском у розробку, підтримуючи PostgreSQL в курсі останніх досягнень.

В свою чергу PostgreSQL має і недоліки:

- Складність: Налаштування та керування розподіленим середовищем PostgreSQL може бути складним, вимагаючи значного досвіду та розуміння функцій і конфігурацій бази даних.
- Накладні витрати на продуктивність: хоча PostgreSQL є високонадійним, деякі розширені функції, такі як реплікація та зовнішні оболонки даних, можуть призвести до накладних витрат на продуктивність, що може вплинути на загальну ефективність.
- Обмежений вбудований шардинг: PostgreSQL не має вбудованих можливостей шардингу, тому для реалізації ефективних стратегій шардингу потрібні інструменти сторонніх розробників або значні спеціальні розробки.
- Конфігурація та налаштування: досягнення оптимальної продуктивності часто потребує детальної конфігурації та налаштування, що може зайняти багато часу та потребує спеціальних знань.
- Ресурсовитратність: PostgreSQL може бути ресурсомістким, особливо у сценаріях високої доступності та високої продуктивності, що потребує потужного апаратного забезпечення та ретельного керування ресурсами.
- Крива навчання: широкий набір функцій і гнучкість PostgreSQL означають, що новим користувачам, особливо тим, хто не знайомий із розширеними концепціями та конфігураціями баз даних, доведеться швидко навчатися.
- Підтримка спільноти: Хоча PostgreSQL має сильну спільноту, їй бракує такого ж рівня комерційної підтримки та комплексних навчальних ресурсів, які пропонують деякі корпоративні бази даних.

2.1.2. MongoDB

MongoDB — популярна база даних NoSQL з відкритим кодом, відома своєю гнучкістю, масштабованістю та простотою використання. Він

призначений для обробки великих обсягів даних і особливо добре підходить для вимог розподіленого зберігання даних.

MongoDB зберігає дані в гнучких документах, схожих на JSON, що дозволяє створювати динамічні схеми. Це означає, що дані можна зберігати, не вимагаючи попередньо визначеної структури, що полегшує внесення змін до моделей даних, не порушуючи роботу програми.

Однією з основних сильних сторін MongoDB для розподіленого зберігання даних є його вбудована підтримка горизонтального масштабування за допомогою сегментування. Шардинг — це метод розподілу даних між декількома машинами або сегментами, щоб гарантувати, що база даних може обробляти великі набори даних і високу пропускну здатність. У MongoDB дані розділені між шардами на основі ключа шарду, що допомагає рівномірно розподілити навантаження та підвищити продуктивність[14].

MongoDB також підтримує набори реплік, які є кластерами серверів MongoDB, які забезпечують резервування та високу доступність. Набір реплік складається з основного вузла, який обробляє всі операції запису, і вторинних вузлів, які копіюють дані з основного. У разі збою основного вузла один із вторинних вузлів може бути автоматично підвищений до основного, забезпечуючи мінімальний час простою.

Гнучка модель даних бази даних і широкі можливості запитів роблять її ідеальною для різноманітних додатків, включаючи аналітику в реальному часі, керування вмістом і програми Інтернету речей (IoT). MongoDB підтримує широкий спектр запитів та параметрів індексування, включаючи геопросторові запити, текстовий пошук і структуру агрегації, що дозволяє обробляти та аналізувати складні дані.

Безпека є ще одним важливим аспектом MongoDB. Він пропонує надійні функції безпеки, такі як автентифікація, авторизація, шифрування та аудит. MongoDB підтримує різні методи автентифікації, включаючи сертифікати LDAP, Kerberos і x.509[15]. Він також забезпечує точне керування доступом із керуванням доступом на основі ролей (RBAC), що

дозволяє адміністраторам визначати ролі та дозволи для різних користувачів і програм.

Багата екосистема інструментів і сервісів MongoDB ще більше підвищує зручність використання для розподіленого зберігання даних. MongoDB Atlas, служба керованої бази даних, спрощує розгортання, масштабування та керування кластерами MongoDB у хмарі. Крім того, MongoDB Compass надає графічний інтерфейс для дослідження та маніпулювання даними, полегшуючи розробникам і адміністраторам роботу з базою даних[16].

Незважаючи на численні переваги, MongoDB має деякі обмеження щодо розподіленого зберігання даних. Гнучка схема, хоч і є перевагою, може призвести до непослідовних даних, якщо нею керувати не ретельно. Крім того, на продуктивність MongoDB можуть вплинути масштабні операції запису та складні транзакції, які можуть потребувати додаткового налаштування та оптимізації.

MongoDB має наступні переваги:

- Масштабованість: вбудоване сегментування для горизонтального масштабування, що дозволяє базі даних ефективно обробляти великі набори даних і високу пропускну здатність.
- Гнучкість: динамічна схема дозволяє легко адаптуватися до змін у моделях даних, не вимагаючи попередньо визначених структур.
- Висока доступність: набори реплік забезпечують резервування та автоматичне перемикання після відмови, забезпечуючи мінімальний час простою та безперервну роботу.
- Розширені можливості запитів: підтримує широкий спектр запитів, включаючи геопросторові запити, текстовий пошук і складні агрегації.
- Безпека: пропонує надійні функції безпеки, такі як автентифікація, авторизація, шифрування та контроль доступу на основі ролей (RBAC).

- Простота використання: багата екосистема інструментів і послуг, включаючи MongoDB Atlas для керованих хмарних служб і MongoDB Compass для графічного дослідження даних.
- Продуктивність: оптимізовано для високої пропускної здатності читання та запису, що робить його придатним для аналітики в реальному часі та інших програм, чутливих до продуктивності.
- Зручність для розробників: JSON-подібна модель документа добре узгоджується з сучасними методами розробки, спрощуючи обробку даних і маніпуляції.

Але і має наступні недоліки:

- Узгодженість даних: гнучка схема може призвести до неузгодженості даних, якщо не ретельно керувати нею, що вимагає суворої перевірки на рівні програми.
- Складні транзакції: хоча MongoDB підтримує багатодокументні транзакції, вони можуть бути складними та менш продуктивними порівняно з традиційними реляційними базами даних.
- Продуктивність запису: великомасштабні операції запису можуть вплинути на продуктивність, вимагаючи ретельного налаштування та оптимізації.
- Ресурсомісткі: шардинг і реплікація можуть бути ресурсомісткими, вимагаючи значної інфраструктури та управління.
- Крива навчання: розробники та адміністратори можуть зіткнутися з крутою кривою навчання, особливо коли мають справу з такими розширеними функціями, як шардинг і набори реплік.
- Обмеження індексування: широке використання індексів може призвести до збільшення вимог до пам’яті та потенційних проблем із продуктивністю.
- Операційні витрати: Управління розподіленим середовищем MongoDB може бути складним і вимагає ретельного планування та поточного обслуговування.

- Обмежена підтримка з’єднань: на відміну від реляційних баз даних, MongoDB має обмежену підтримку з’єднань, що може призвести до більш складних запитів і розширеної логіки програми.

2.1.3. MySQL

MySQL — це широко розповсюджена система керування реляційними базами даних із відкритим вихідним кодом, відома своєю надійністю, простотою використання та високою продуктивністю. Він використовує SQL для керування та обробки даних, забезпечуючи надійну обробку транзакцій і цілісність даних завдяки підтримці властивостей ACID. Це робить його придатним для розподілених систем зберігання даних.

MySQL пропонує різні методи реплікації, включаючи асинхронну, напівсинхронну та синхронну реплікацію. Вони дозволяють копіювати дані з головного сервера на один або кілька підлеглих серверів, забезпечуючи резервування та покращуючи продуктивність читання. Це налаштування покращує високу доступність і аварійне відновлення.

Хоча MySQL не має вбудованих можливостей шардингу, шардинг можна реалізувати вручну або за допомогою інструментів сторонніх розробників, таких як Vitess, ProxySQL або MySQL Fabric. Шардинг передбачає розподіл даних між кількома екземплярами бази даних для розподілу навантаження та покращення масштабованості.

MySQL Cluster надає рішення для високої доступності та масштабованості через архітектуру без спільного використання, де дані розподіляються між кількома вузлами без єдиної точки відмови. Це налаштування ідеально підходить для критично важливих додатків у режимі реального часу та забезпечує автоматичне перемикання після збоїв і самовідновлення[17].

Висока доступність у MySQL також досягається за допомогою реплікації, кластеризації та інструментів сторонніх виробників, таких як MHA

(Master High Availability Manager) і Orchestrator, що забезпечує доступність бази даних навіть під час апаратних або програмних збоїв.

Функції продуктивності та оптимізації в MySQL включають кешування запитів, індексування, розділення та систему зберігання InnoDB, яка розроблена для високої продуктивності та надійності. Правильна конфігурація та налаштування можуть значно підвищити продуктивність розподіленої установки MySQL.

MySQL містить надійні функції безпеки, такі як SSL/TLS для зашифрованих з'єднань, плагіни автентифікації та керування доступом на основі ролей. Він також підтримує шифрування даних у стані спокою та різні методи автентифікації, включаючи PAM і LDAP.

Інструменти резервного копіювання та відновлення в MySQL, такі як mysqldump, MySQL Enterprise Backup і сторонні рішення, такі як Percona XtraBackup, забезпечують безпеку даних і полегшують відновлення на певний момент часу, що є важливим для підтримки цілісності даних у розподілених середовищах[18].

MySQL має велику та активну спільноту, яка постійно сприяє її розвитку. Існує велика кількість документації, навчальних посібників і форумів для підтримки. Комерційна підтримка також доступна від Oracle, пропонуючи допомогу та послуги на рівні підприємства.

Переваги MySQL:

- Надійність: відома стабільністю та сильною підтримкою властивостей ACID, що забезпечує надійну обробку транзакцій і цілісність даних.
- Реплікація: різні методи реплікації (асинхронні, напівсинхронні, синхронні) підвищують надмірність даних і продуктивність читання.
- Висока доступність: такі рішення, як MySQL Cluster і інструменти сторонніх виробників (MHA, Orchestrator), забезпечують високу доступність і автоматичне відновлення після відмови.

- Оптимізація продуктивності: такі функції, як кешування запитів, індексація, розділення та система зберігання даних InnoDB покращують продуктивність і надійність.
- Безпека: надійні функції безпеки, включаючи SSL/TLS, плагіни автентифікації, контроль доступу на основі ролей і шифрування даних у стані спокою.
- Резервне копіювання та відновлення: комплексні інструменти для резервного копіювання та відновлення, що гарантує безпеку даних і відновлення в певний момент часу.
- Підтримка спільноти: велика активна спільнота з великою документацією, навчальними посібниками, форумами та комерційною підтримкою від Oracle.
- Масштабованість: можна горизонтально масштабувати за допомогою сегментування, що підтримується інструментами сторонніх розробників, такими як Vitess, ProxySQL і MySQL Fabric.

Недоліки MySQL:

- Складне сегментування: не вистачає вбудованих можливостей шардингу, що вимагає ручного впровадження або інструментів сторонніх розробників, що може бути складним і ресурсомістким.
- Операційні витрати: Управління розподіленим середовищем MySQL, включаючи конфігурації реплікації та високої доступності, може бути складним і вимагати значного досвіду.
- Вузькі місця продуктивності: високе робоче навантаження на запис і складні транзакції можуть вплинути на продуктивність, вимагаючи ретельного налаштування та оптимізації.
- Ресурсовитратність: реплікація та кластеризація можуть бути ресурсомісткими, вимагаючи надійного апаратного забезпечення та ретельного управління ресурсами.

- Складність конфігурації: досягнення оптимальної продуктивності та високої доступності часто вимагає детальної конфігурації та постійного обслуговування.
- Крива навчання: Крута крива навчання для впровадження розширених функцій, таких як шардинг, реплікація та налаштування продуктивності.
- Обмеження приєднання: хоча MySQL підтримує об’єднання, складні об’єднання в розподіленому середовищі можуть призвести до проблем з продуктивністю та можуть вимагати додаткової логіки програми.
- Складність резервного копіювання: керування резервним копіюванням і забезпечення узгодженості даних у розподілених системах може бути складним завданням і потребуватиме спеціальних інструментів і процесів.

2.2. Типи підсистем захисту даних

2.2.1. Автентифікація та авторизація

Автентифікація та авторизація є основними компонентами будь-якої підсистеми безпеки, особливо в середовищах розподіленого зберігання даних, де захист конфіденційної інформації є критично важливим. Ці процеси гарантують, що користувачі є тими, за кого себе видають, і мають необхідні дозволи для доступу до певних ресурсів або виконання певних дій, таким чином зберігаючи цілісність, конфіденційність і доступність системи.

Автентифікація – це процес підтвердження особи користувача або системи. Він виступає першою лінією захисту програми, запобігаючи несанкціонованому доступу. Забезпечення надійних механізмів автентифікації є важливим для будь-якої системи, яка має справу з конфіденційними або цінними даними. Найпоширенішою формою автентифікації є автентифікація на основі пароля. Користувачі вказують ім’я користувача та пароль для отримання доступу до системи. Цей метод відносно

простий у застосуванні та широко зрозумілий користувачам, що робить його популярним вибором. Однак його ефективність значною мірою залежить від надійності та унікальності паролів, створених користувачами, і здатності системи захищати ці паролі за допомогою розширених заходів безпеки, таких як хешування та соління[19].

Хешування перетворює паролі на рядок символів фіксованої довжини, який виглядає випадковим чином. Це перетворення є одностороннім, тобто повернути хеш до вихідного пароля з обчислювальної точки зору неможливо. Засіб передбачає додавання унікального значення до кожного пароля перед його хешуванням, гарантуючи, що навіть якщо два користувачі мають однаковий пароль, їхні хешовані значення будуть різними. Ці методи значно підвищують безпеку паролів, ускладнюючи зловмисникам зламати паролі за допомогою таких методів, як груба сила або атаки за словником.

Біометрична автентифікація — ще один розширений метод, який використовується для підтвердження особи користувача. Цей тип автентифікації базується на унікальних біологічних характеристиках, таких як відбитки пальців, розпізнавання обличчя або сканування райдужної оболонки ока. Біометрична автентифікація забезпечує високий рівень безпеки, оскільки ці характеристики надзвичайно важко відтворити. Крім того, біометричні системи часто зручні для користувача, забезпечуючи безперебійний і швидкий процес автентифікації. Однак впровадження біометричних систем вимагає спеціального обладнання та викликає занепокоєння щодо конфіденційності, оскільки біометричні дані є дуже чутливими.

Аутентифікація на основі маркерів передбачає використання маркерів для підтвердження особи користувача. Маркери, такі як веб-токени JSON (JWT) або маркери OAuth, генеруються після успішного входу та використовуються для наступних запитів[20]. Цей метод зменшує необхідність багаторазової передачі облікових даних користувача, тим самим знижуючи ризик перехоплення під час передачі. Маркери зазвичай містять закодовану інформацію про користувача та його дозволи, які сервер може

перевірити. Автентифікація на основі маркерів особливо корисна в розподілених системах і архітектурах мікросервісів, де різні компоненти системи повинні незалежно автентифікувати користувачів.

Авторизація, з іншого боку, визначає, що автентифікованому користувачеві дозволено робити в системі. Після автентифікації користувача механізми авторизації контролюють його доступ до ресурсів і дії на основі попередньо визначених правил і дозволів. Авторизація гарантує, що користувачі можуть виконувати лише дії та отримувати доступ до ресурсів, на які їм надано права, таким чином запобігаючи несанкціонованому доступу до конфіденційних даних або критичних функцій.

Контроль доступу на основі ролей (RBAC) є широко використовуваним підходом до авторизації. У RBAC користувачам призначаються ролі, і кожна роль має певні пов’язані з нею дозволи. Цей підхід спрощує керування дозволами користувачів, групуючи їх у ролі, а не призначаючи дозволи окремо кожному користувачеві. RBAC особливо ефективний у середовищах із чітко визначеними посадовими функціями, де ролі можна узгодити з організаційними обов’язками.

Контроль доступу на основі атрибутів (ABAC) пропонує більш гнучкий і детальний підхід до авторизації. У ABAC дозволи надаються на основі атрибутів користувача, ресурсу та середовища. Атрибути можуть включати ролі користувачів, відділ, час доступу та іншу контекстну інформацію. ABAC дозволяє приймати динамічні рішення щодо контролю доступу, адаптуючись до мінливих обставин і забезпечуючи вищий рівень безпеки та контролю.

Списки контролю доступу (ACL) надають ще один метод керування авторизацією. Списки керування доступом визначають окремі дозволи для кожного користувача або групи користувачів для певних ресурсів. Хоча списки керування доступом пропонують детальний контроль над дозволами доступу, вони можуть стати складними та складними в управлінні зі зростанням кількості користувачів і ресурсів. Належне управління та

організація ACL мають вирішальне значення для підтримки ефективної системи авторизації.

Контроль доступу на основі політик (PBAC) використовує політики для визначення правил для рішень щодо контролю доступу. Політики можуть враховувати різні фактори, включаючи атрибути користувача, атрибути ресурсів і контекстну інформацію, забезпечуючи гнучкий і динамічний підхід до авторизації. PBAC дозволяє організаціям впроваджувати комплексні та деталізовані стратегії контролю доступу, які можна адаптувати до конкретних потреб і сценаріїв.

Впровадження надійних механізмів автентифікації та авторизації має вирішальне значення для захисту конфіденційних даних у системах розподіленого зберігання. Автентифікація гарантує, що лише законні користувачі можуть отримати доступ до системи, запобігаючи неавторизованому доступу з самого початку. Тоді авторизація контролює, які дії можуть виконувати ці користувачі та до яких ресурсів вони можуть отримати доступ, гарантуючи, що навіть автентифіковані користувачі не зможуть переступити свої межі. Разом автентифікація та авторизація утворюють основу захищеної програми, запобігаючи несанкціонованому доступу та захищаючи важливу інформацію. Забезпечення надійності, масштабованості та адаптації цих механізмів до загроз, що розвиваються, має важливе значення для підтримки безпеки та цілісності будь-якої розподіленої системи зберігання даних.

Переваги автентифікації та авторизації:

- Покращена безпека: Впровадження надійних механізмів автентифікації та авторизації значно знижує ризик несанкціонованого доступу, захищаючи конфіденційні дані від потенційних порушень.
- Відповідальність користувача: Вимагаючи від користувачів автентифікації та надаючи їм певні дозволи, організації можуть

відстежувати дії користувачів у системі, забезпечуючи підзвітність і відстеження.

- Детальний контроль доступу: Авторизація дозволяє точно контролювати, хто може отримати доступ до певних ресурсів і виконувати певні дії, забезпечуючи детальний і індивідуальний підхід до безпеки.
- Відповідність нормам: Надійна автентифікація та авторизація допомагають організаціям дотримуватися галузевих правил і стандартів, таких як GDPR, HIPAA та PCI-DSS, які часто вимагають суворого контролю доступу.
- Покращена взаємодія з користувачем: Сучасні методи автентифікації, такі як біометрична автентифікація, пропонують бездоганний і зручний досвід, зменшуючи тертя, пов’язані з входом у систему та доступом до ресурсів.
- Захист від внутрішніх загроз: Обмежуючи доступ лише тим, хто його потребує, і відстежуючи дії користувачів, автентифікація та авторизація допомагають зменшити ризики, пов’язані з внутрішніми загрозами.
- Масштабованість: Розширені механізми авторизації, такі як контроль доступу на основі ролей (RBAC) і контроль доступу на основі атрибутів (ABAC), можуть ефективно масштабуватися разом із зростанням організації, керуючи доступом для все більшої кількості користувачів і ресурсів.
- Динамічна та контекстно-залежна безпека: Такі методи, як ABAC і контроль доступу на основі політики (PBAC), забезпечують динамічну безпеку з урахуванням контексту, адаптуючи елементи керування доступом на основі атрибутів і політик у реальному часі.
- Інтеграція з сучасними технологіями: Інфраструктури автентифікації та авторизації можна інтегрувати з різними сучасними технологіями

та протоколами, такими як OAuth і OpenID Connect, сприяючи безпечним і сумісним системам.

- Зниження адміністративних витрат: Впровадження централізованих механізмів контролю доступу на основі ролей спрощує керування дозволами користувачів, зменшуючи адміністративне навантаження, пов’язане з підтримкою контролю доступу.

Ці переваги підкреслюють важливість впровадження надійних механізмів автентифікації та авторизації в будь-якій підсистемі безпеки, забезпечуючи захист та ефективне управління конфіденційними даними та ресурсами в середовищах розподіленого зберігання даних.

Недоліки автентифікації та авторизації:

- Складність реалізації: Налаштування надійних систем автентифікації та авторизації може бути складним і трудомістким, вимагаючи ретельного планування, налаштування та тестування, щоб переконатися, що вони функціонують правильно.
- Незручності для користувачів: Хоча розширені методи автентифікації підвищують безпеку, вони також можуть створювати незручності для користувачів, особливо якщо від них вимагається запам’ятати кілька паролів або використовувати додаткові пристрої для автентифікації.
- Накладні витрати на технічне обслуговування: Регулярне оновлення та підтримка систем аутентифікації та авторизації має важливе значення для усунення вразливостей і адаптації до нових загроз безпеці. Це поточне технічне обслуговування може бути ресурсомістким.
- Проблеми масштабованості: У великих організаціях із великою кількістю користувачів і ресурсів керування списками контролю доступу (ACL) і дозволами може стати громіздким і схильним до помилок, що потенційно може призвести до прогалин у безпеці.

- Вплив на продуктивність: Впровадження детальних і тонких засобів контролю доступу може призвести до накладних витрат на продуктивність, особливо в середовищах із високим трафіком, де часто виконуються численні перевірки контролю доступу.
- Єдині точки відмови: Централізовані системи автентифікації, такі як єдиний вхід (SSO), можуть стати єдиною точкою збою. Якщо система автентифікації вийде з ладу, це може перешкодити користувачам отримати доступ до всіх залежних служб.
- Залежність від зовнішніх служб: Використання сторонніх постачальників або служб автентифікації (наприклад, постачальників OAuth) створює залежності, які можуть призвести до збоїв або порушень безпеки, якщо зовнішня служба скомпрометована або недоступна.
- Вразливі місця безпеки: Незважаючи на всі зусилля, системи автентифікації та авторизації все ще можуть мати вразливості, як-от неправильні налаштування, слабкі паролі або недоліки в реалізації протоколів безпеки, якими можуть скористатися зловмисники.
- Питання конфіденційності: Методи біометричної автентифікації викликають занепокоєння щодо конфіденційності, оскільки вони передбачають збір і зберігання конфіденційних персональних даних. Неправильне поводження з цими даними може призвести до порушення конфіденційності та втрати довіри користувачів.
- Складність керування користувачами: Управління ролями користувачів, дозволами й атрибутами в динамічних і великомасштабних середовищах може бути складним і вимагає складних інструментів і процесів для забезпечення точного й безпечного контролю доступу.

Ці недоліки підкреслюють проблеми, пов’язані з впровадженням ефективних систем автентифікації та авторизації. Хоча ці механізми мають вирішальне значення для захисту конфіденційних даних і ресурсів, вони

потребують ретельного розгляду та керування, щоб збалансувати потреби безпеки зі зручністю для користувача та продуктивністю системи.

2.2.2. Єдиний вхід (SSO)

Єдиний вхід (Single Sign-On) — це механізм автентифікації користувача, який дозволяє отримувати доступ до кількох програм і служб за допомогою єдиного набору облікових даних для входу, як правило, імені користувача та пароля. Ця система спрощує взаємодію з користувачем, усуваючи необхідність окремого входу для кожної програми, зменшуючи когнітивне навантаження, пов’язане із запам’ятовуванням кількох паролів і частотою підказок входу.

Процес SSO зазвичай працює наступним чином: користувач один раз входить до системи SSO, використовуючи свої облікові дані. Після успішної автентифікації система видає токен автентифікації або сеанс, який дозволяє отримати доступ до кількох пов’язаних програм, не вимагаючи від користувача повторного входу. Цей маркер зазвичай зберігається у файлі cookie у браузері користувача або керується центральною службою автентифікації.

Система єдиного входу підвищує безпеку шляхом централізації процесів автентифікації. Ця централізація дозволяє застосовувати більш сильні та узгоджені політики безпеки, такі як багатофакторна автентифікація (MFA) і вимоги до складності пароля. Оскільки користувачам потрібно запам’ятати лише один пароль, вони менш імовірно використовуватимуть слабкі або повторювані паролі, що зменшує ризик викрадення облікових даних.

Окрім зручності для користувачів, SSO забезпечує значні адміністративні переваги. IT-адміністратори можуть керувати доступом користувачів з єдиної точки, спрощуючи процес додавання та видалення користувачів і забезпечуючи послідовний контроль доступу для всіх інтегрованих програм. Це централізоване керування є особливо вигідним для

організацій зі складним IT-середовищем, оскільки воно зменшує адміністративні накладні витрати та ймовірність помилок.

Крім того, SSO сприяє кращій відповідності та можливостям аудиту. Централізоване реєстрування подій автентифікації полегшує моніторинг і перевірку дій користувачів, забезпечуючи дотримання нормативних вимог, таких як GDPR, HIPAA та SOX. Аудитори можуть легко відстежувати шаблони доступу та виявляти спроби несанкціонованого доступу.

Реалізації SSO часто використовують такі стандарти, як OAuth, OpenID Connect і SAML (Security Assertion Markup Language). Ці стандарти забезпечують безпечну передачу інформації автентифікації між браузером користувача, системою SSO і цільовими програмами. Наприклад, SAML широко використовується в корпоративних середовищах завдяки своїм надійним функціям безпеки, тоді як OAuth і OpenID Connect популярні для веб- і мобільних програм завдяки своїй простоті та гнучкості.

У контексті розподілених систем зберігання даних SSO відіграє вирішальну роль у забезпеченні безпечного та безперебійного доступу до даних на кількох вузлах зберігання. Оскільки дані часто розповсюджуються в різних місцях і доступ до них здійснюється через різні програми, система єдиного входу забезпечує постійну автентифікацію та авторизацію користувачів, зменшуючи ризик неавторизованого доступу до даних. Це особливо важливо в середовищах, де безпека даних має першочергове значення, наприклад у фінансових службах, охороні здоров’я та державному секторі.

Однак SSO також створює певні ризики. Основним ризиком є можливість виникнення єдиної точки збою: якщо систему SSO зламано, зловмисники можуть отримати доступ до кількох програм і конфіденційних даних. Щоб зменшити цей ризик, системи SSO мають бути захищені надійними механізмами автентифікації, регулярними перевірками безпеки та оновленими виправленнями безпеки. Крім того, інтеграція MFA з SSO може забезпечити додатковий рівень безпеки, гарантуючи, що навіть якщо один

набір облікових даних скомпрометовано, неавторизований доступ все одно буде запобігти.

Впровадження SSO вимагає ретельного планування та інтеграції з існуючою IT-інфраструктурою. Організаціям потрібно вибрати правильне рішення SSO, яке відповідає їхнім потребам, враховуючи такі фактори, як масштабованість, простота інтеграції, підтримка різних протоколів автентифікації та вартість. До популярних рішень SSO належать Okta, OneLogin, Microsoft Azure Active Directory і Google Identity Platform, кожне з яких пропонує низку функцій, які відповідають різним вимогам організації.

Таким чином, SSO — це потужний механізм автентифікації, який підвищує зручність і безпеку користувача, надаючи доступ до кількох програм за допомогою одного набору облікових даних. Він централізує автентифікацію, зменшує накладні витрати на адміністрування та покращує можливості відповідності та аудиту. Однак це також вимагає надійних заходів безпеки, щоб зменшити ризики, пов’язані з єдиною точкою відмови. Ретельно вибираючи та впроваджуючи рішення SSO, організації можуть значно підвищити рівень безпеки та досвід користувачів.

2.2.3. Двофакторна автентифікація

Двофакторна автентифікація (2FA) — це передовий механізм безпеки, який покращує традиційний процес автентифікації, вимагаючи від користувача двох різних форм ідентифікації. Цей метод значно посилює безпеку, гарантуючи, що навіть якщо один фактор скомпрометований, несанкціонований доступ все одно запобігає другий фактор. Двома факторами, які зазвичай беруть участь у 2FA, є те, що користувач знає (наприклад, пароль), і те, що користувач має (наприклад, смартфон або апаратний маркер).

На першому кроці 2FA користувач вводить своє ім’я користувача та пароль. Це перший фактор, про який користувач знає. Якщо пароль правильний, система запитує другий фактор, який має користувач. Це може

бути одноразовий пароль (OTP), згенерований програмою автентифікації на смартфоні користувача, код, надісланий через SMS, апаратний маркер або біометричний фактор, такий як відбиток пальця чи розпізнавання обличчя.

Одноразові паролі, створені програмами автентифікації або апаратними маркерами, зазвичай залежать від часу, тобто вони закінчуються через короткий період (зазвичай 30 секунд)[21]. Це гарантує, що навіть якщо зловмиснику вдасться викрасти OTP, йому потрібно буде використати його протягом дуже короткого періоду часу, що значно знижує ризик неправомірного використання. Апаратні маркери генерують OTP незалежно від Інтернету, забезпечуючи додатковий рівень безпеки, оскільки вони не схильні до онлайн-атак.

2FA на основі SMS надсилає код підтвердження на зареєстрований мобільний номер користувача. Хоча це зручно, цей метод може бути вразливим до атак із заміною SIM-карти, коли зловмисник отримує контроль над номером телефону жертви. Незважаючи на це, 2FA на основі SMS все ще широко використовується завдяки своїй простоті та легкості впровадження.

Біометричний 2FA використовує унікальні біологічні характеристики, такі як відбитки пальців, розпізнавання обличчя або сканування райдужної оболонки ока як другий фактор. Цей метод забезпечує високий рівень безпеки та зручності користувача, оскільки біометричні дані важко відтворити. Однак для цього потрібне спеціальне обладнання та виникає проблема конфіденційності щодо зберігання та використання біометричної інформації.

Інтеграція 2FA в підсистему безпеки має вирішальне значення для програм, які обробляють конфіденційні або критичні дані. Додаючи додатковий рівень безпеки, 2FA допомагає захистити від різних загроз, включаючи фішингові атаки, крадіжки паролів і несанкціонований доступ. Реалізація 2FA передбачає налаштування програми для запиту другого фактора після початкової перевірки пароля та забезпечення доступу користувачів до другого фактора, будь то програма автентифікації, апаратний маркер або біометричний сканер.

2FA є особливо корисним у розподілених середовищах зберігання даних, де безпека доступу до даних на кількох серверах має першочергове значення. Він забезпечує ефективний засіб перевірки ідентичності користувачів і захисту від несанкціонованого доступу, гарантуючи, що навіть якщо зловмисник зламав пароль користувача, йому все одно знадобиться другий фактор для отримання доступу.

Таким чином, двофакторна автентифікація є критично важливим заходом безпеки, який значно покращує захист систем і даних, вимагаючи двох окремих форм перевірки. Його реалізація передбачає використання того, що користувач знає, і того, що користувач має, забезпечуючи надійний захист від різних загроз безпеці.

Переваги двофакторної автентифікації (2FA):

- Покращена безпека: 2FA значно підвищує безпеку, додаючи другий рівень перевірки, що значно ускладнює неавторизований доступ для зловмисників, навіть якщо вони отримують пароль користувача.
- Захист від крадіжки пароля: Навіть якщо пароль користувача зламано через фішинг, клавіатурний журнал чи інші методи, зловмисник не зможе отримати доступ до облікового запису без другого фактора.
- Зменшення ризику несанкціонованого доступу: Вимагаючи двох форм ідентифікації, 2FA гарантує, що лише законний користувач може отримати доступ до свого облікового запису, забезпечуючи надійний захист від несанкціонованого доступу.
- Пом'якшення фішингових атак: 2FA допомагає зменшити ризик фішингових атак, вимагаючи додаткового етапу перевірки, який нелегко відтворити спроби фішингу.
- Зміцнення довіри та впевненості користувачів: Впровадження 2FA підвищує впевненість користувачів у безпеці системи, оскільки вони знають, що їхні облікові записи краще захищені.

- Дотримання правил безпеки: Багато нормативних стандартів і галузевих передових практик рекомендують або вимагають використання 2FA для захисту конфіденційних даних, допомагаючи організаціям дотримуватися вимог безпеки.
- Захист віддаленого доступу: 2FA особливо корисний для сценаріїв віддаленого доступу, гарантуючи, що навіть якщо облікові дані перехоплено, зловмисник не зможе отримати доступ без другого фактора.
- Універсальність реалізації: 2FA можна реалізувати за допомогою різних методів, таких як SMS, програми автентифікації, апаратні токени та біометрія, що дозволяє організаціям вибрати найкращий варіант для своїх потреб і вподобань користувачів.
- Стримування атак грубою силою: Додатковий рівень безпеки, який забезпечує 2FA, робить атаки грубою силою менш ефективними, оскільки зловмисникам потрібно буде зламати як пароль, так і другий фактор.
- Стратегія багаторівневої оборони: 2FA сприяє багаторівневій стратегії захисту, створюючи численні перешкоди для подолання зловмисників і підвищуючи загальну безпеку системи.

Ці переваги демонструють критичну роль 2FA у захисті облікових записів користувачів і конфіденційної інформації, що робить його важливим компонентом сучасних методів безпеки.

Недоліки двофакторної автентифікації (2FA):

- Незручності для користувачів: 2FA додає додатковий крок до процесу входу, який може бути незручним і трудомістким для користувачів, особливо якщо їм потрібно часто проходити автентифікацію.
- Залежність від другого фактора: Користувачі повинні мати доступ до свого другого фактора, такого як смартфон або апаратний

маркер. Якщо вони втрачені, пошкоджені або недоступні, користувачі можуть не мати доступу до своїх облікових записів.

- Можливість блокування: У сценаріях, коли другий фактор недоступний (наприклад, втрачений телефон), користувачі можуть бути заблоковані в своїх облікових записах, що потребує додаткової підтримки та процесів відновлення.
- Технічні питання: Технічні проблеми, такі як погане покриття мобільної мережі, проблеми з додатками для автентифікації або несправності апаратного маркера, можуть перешкодити користувачам завершити процес автентифікації.
- Уразливості SMS: 2FA на основі SMS вразливий до таких атак, як заміна SIM-карти та перехоплення, що може поставити під загрозу безпеку другого фактора.
- Розширена підтримка та обслуговування: Впровадження та підтримка систем 2FA вимагає додаткових ресурсів для підтримки користувачів, усунення несправностей і оновлення системи.
- Питання конфіденційності: Біометричні методи 2FA викликають занепокоєння щодо конфіденційності щодо збору, зберігання та можливого зловживання конфіденційними біометричними даними.
- Виклики зручності використання: Деякі користувачі, особливо ті, хто менш розбирається в техніці, можуть вважати 2FA заплутаним або складним у використанні, що призведе до розчарування та потенційних перешкод для впровадження.
- Помилкове відчуття безпеки: Хоча 2FA покращує безпеку, він не є надійним. Надмірна залежність від 2FA без усунення інших вразливостей безпеки може створити помилкове відчуття безпеки.

Ці недоліки підкреслюють проблеми, пов’язані з впровадженням і використанням 2FA. Незважаючи на те, що це значно підвищує безпеку, потрібно ретельно керувати його впровадженням і вирішувати потенційні

проблеми користувача та технічні проблеми, щоб забезпечити його ефективність.

2.2.4. Шифрування даних

Шифрування даних — це важливий захід безпеки, який передбачає перетворення простих текстових даних у нечитабельний формат, званий зашифрованим текстом, який може бути розшифрований назад у звичайний текст тільки тим, хто має правильний ключ розшифровки[22]. Цей процес гарантує, що навіть якщо неавторизовані сторони отримають доступ до даних, вони не зможуть прочитати або використовувати їх без ключа дешифрування. Оскільки було вирішено використовувати PostgreSQL, у контексті цієї системи, шифрування даних допомагає захистити конфіденційну інформацію, що зберігається в базі даних, від несанкціонованого доступу та потенційних порушень.

Типи шифрування даних:

- Симетричне шифрування – використовує один ключ як для шифрування, так і для дешифрування. Той самий ключ, який шифрує дані, використовується для їх дешифрування. Цей метод швидкий і ефективний для шифрування великих обсягів даних. Однак ключ має бути безпечно наданий і збережений, оскільки кожен, хто має доступ до ключа, може розшифрувати дані[23]. Поширені алгоритми симетричного шифрування включають AES (Advanced Encryption Standard) і DES (Data Encryption Standard).
- Асиметричне шифрування – використовує пару ключів: відкритий ключ для шифрування та закритий ключ для дешифрування. Відкритий ключ можна вільно ділитися, тоді як закритий ключ потрібно зберігати в секреті. Цей метод більш безпечний для розповсюдження ключів, але є повільнішим і потребує більше обчислень, ніж симетричне шифрування[24]. Поширені асиметричні

алгоритми шифрування включають RSA (Rivest-Shamir-Adleman) і ECC (криптографія еліптичної кривої).

- Гібридне шифрування – поєднує в собі симетричне й асиметричне шифрування, щоб використовувати переваги обох методів. Як правило, асиметричне шифрування використовується для безпечного обміну симетричним ключем, який потім використовується для фактичного шифрування даних. Такий підхід забезпечує як безпеку, так і ефективність[25].

PostgreSQL, в свою чергу, підтримує кілька методів шифрування даних, щоб забезпечити їх безпеку як у стані спокою, так і під час передачі.

- Шифрування в стані спокою – захищає дані, що зберігаються на диску, від несанкціонованого доступу. Цього можна досягти за допомогою шифрування на рівні файлової системи, шифрування на рівні бази даних або шифрування на рівні стовпців.
- Шифрування на рівні файлової системи – це передбачає шифрування всієї файлової системи, де зберігаються дані PostgreSQL. Такі інструменти, як LUKS (Linux Unified Key Setup) або BitLocker (у Windows), можна використовувати для шифрування розділів диска, що містять файли бази даних. Цей метод прозорий для PostgreSQL і не вимагає змін у конфігурації бази даних.
- Шифрування на рівні бази даних – PostgreSQL підтримує використання розширення pgcrypto, яке надає функції для шифрування та дешифрування даних у базі даних. Це дозволяє шифрувати певні стовпці або таблиці, гарантуючи, що шифруються лише конфіденційні дані, що мінімізує накладні витрати на продуктивність.
- Шифрування на рівні стовпців – за допомогою шифрування на рівні стовпців певні стовпці, що містять конфіденційні дані, шифруються за допомогою розширення pgcrypto. Цей підхід дозволяє точно

контролювати, які дані шифруються, і допомагає оптимізувати продуктивність шляхом шифрування лише необхідних даних.

- Шифрування під час передачі – гарантує, що дані, що передаються між клієнтом і сервером PostgreSQL, є безпечними та захищеними від перехоплення. PostgreSQL підтримує SSL/TLS (Secure Sockets Layer/Transport Layer Security) для шифрування каналу зв’язку.
- Налаштування SSL/TLS – щоб увімкнути SSL/TLS, у PostgreSQL потрібно налаштувати сертифікат сервера та закритий ключ. Клієнти також повинні бути налаштовані на довіру сертифікату сервера. Це налаштування гарантує, що всі дані, що передаються між клієнтом і сервером, зашифровані, запобігаючи прослуховування та атаки типу "людина посередині".
- Керування ключами – ефективне керування ключами має вирішальне значення для безпеки зашифрованих даних. Ключі шифрування мають бути безпечно створені, збережені, замінені та знищені, коли вони більше не потрібні. PostgreSQL може інтегруватися із зовнішніми службами керування ключами (KMS) для безпечного виконання завдань керування ключами.

Шифрування даних є життєво важливим компонентом комплексної стратегії безпеки для баз даних PostgreSQL. Шифруючи дані як у стані спокою, так і під час передачі, організації можуть захистити конфіденційну інформацію від несанкціонованого доступу та потенційних порушень. Впровадження шифрування передбачає вибір відповідних методів шифрування, налаштування PostgreSQL для використання шифрування та забезпечення надійних методів керування ключами[26]. Таким чином організації можуть гарантувати, що їхні дані залишаються в безпеці та відповідають нормативним вимогам.

Висновки до розділу 2

Під час перегляду потенційних баз даних для розподіленої системи зберігання даних було оцінено кілька відомих варіантів, включаючи PostgreSQL, MongoDB і MySQL. Кожна з цих баз даних пропонує відмінні переваги та можливості, які роблять їх придатними для розподілених середовищ. Після ретельного аналізу PostgreSQL виявився найкращим вибором завдяки своїй надійності, розширеним функціям і повній підтримці властивостей ACID (атомарність, узгодженість, ізоляція, довговічність). Ці якості необхідні для забезпечення надійної обробки транзакцій і підтримки цілісності даних у розподіленій системі.

PostgreSQL вирізняється масштабованістю завдяки підтримці розділення, шардингу та реплікації. Ці функції дозволяють ефективно розподіляти дані між кількома вузлами, підвищуючи продуктивність і доступність. Розширені можливості керування даними бази даних, включаючи підтримку широкого діапазону типів даних і складних механізмів індексування, забезпечують ефективну обробку складних запитів і великих наборів даних. Крім того, розширюваність PostgreSQL і активна підтримка спільноти забезпечують міцну основу для постійного вдосконалення та адаптації до нових потреб.

У поєднанні з вибором PostgreSQL як бази даних надзвичайно важливо впровадити надійні системи безпеки для захисту цілісності, конфіденційності та доступності даних. Структура безпеки для нашої розподіленої системи зберігання даних включатиме такі ключові компоненти:

- Шифрування даних: впровадження шифрування даних захистить конфіденційну інформацію як у стані спокою, так і під час передачі, забезпечуючи безпеку даних навіть у разі перехоплення або доступу неавторизованих сторін.
- Автентифікація та авторизація: надійні механізми автентифікації використовуватимуться для перевірки ідентичності користувачів, які отримують доступ до системи, тоді як детальні елементи керування

авторизацією гарантуватимуть, що користувачі матимуть доступ лише до даних і функцій, необхідних для виконання їхніх ролей. Це запобіжить несанкціонованому доступу та потенційному зловживанню даними.

- Двофакторна автентифікація (2FA): додаючи додатковий рівень безпеки, 2FA вимагатиме від користувачів надання другої форми перевірки, наприклад коду, надісланого на їхній мобільний пристрій, на додаток до пароля. Це значно знижує ризик несанкціонованого доступу, навіть якщо пароль користувача зламано.

Ці заходи безпеки працюватимуть у тандемі для створення безпечного середовища, яке захищає від широкого спектру загроз, забезпечуючи надійність і надійність нашої розподіленої системи зберігання даних.

Підсумовуючи, PostgreSQL буде використовуватися як база даних для нашої розподіленої системи зберігання даних завдяки її надійності, масштабованості та розширеним функціям. Цей вибір доповнюється комплексною системою безпеки, яка включає шифрування даних, надійні механізми автентифікації та авторизації, а також двофакторну автентифікацію. Разом ці компоненти створять надійну та безпечну основу для нашої програми, забезпечуючи цілісність, конфіденційність і доступність даних, що зберігаються в системі.

РОЗДІЛ 3
РЕАЛІЗАЦІЯ ПІДСИСТЕМИ ЗАХИСТУ РОЗПОДІЛЕНОГО СХОВИЩА ДАНИХ

Розробка підсистеми захисту розподіленого сховища даних полягає в здійсненні наступних кроків:

- реалізація розподіленого сховища даних;
- реалізація автентифікації та авторизації;
- реалізація двофакторної автентифікації;
- реалізація мсистеми шифрування даних;
- реалізація додаткової міри безпеки у вигляді капчі;
- реалізація інтерфейсу для візуалізації розроблених підсистем.

3.1. Огляд засобів реалізації

Для виконання моделювання програмного додатку способу конструювання трафіку в програмно-конфігурованих мереж було обрано мову програмування Python 3.7. Вибір заснований на широких можливостях

Для виконання розробки підсистеми захисту розподіленого сховища даних було обрано мову програмування Ruby та фреймворк для веб-додатку Ruby on Rails. Вибір базувався на можливостях цих інструментів щодо створення масштабованих і надійних веб-додатків. Зокрема, для реалізації функціоналу безпеки та автентифікації було використано наступні бібліотеки:

- Devise для авторизації;
- Pundit для автентифікації;
- ActiveModel::OTP для двофакторної автентифікації;
- reCAPTCHA для захисту від ботів;
- OpenSSL для шифрування та дешифрування даних.

Основною темою розробки була підсистема захисту розподіленого сховища даних. Для цього була впроваджена система, яка забезпечує високий рівень безпеки даних через використання сучасних технологій і бібліотек. Розробка програмного додатку виконувалась у інтегрованому середовищі розробки, що

забезпечує ефективну роботу та полегшує процес тестування й розгортання. Для забезпечення високої безпеки та інтеграції з різними сервісами було використано наступні технології:

- Devise для налаштування авторизації користувачів, що дозволяє легко додавати функції реєстрації, відновлення пароля та інші важливі компоненти безпеки;
- Pundit для визначення політик доступу до ресурсів на основі ролей користувачів;
- ActiveModel::OTP для впровадження двофакторної автентифікації, що забезпечує додатковий рівень безпеки для користувачів;
- reCAPTCHA для запобігання автоматизованому створенню акаунтів і захисту від спам-ботів;
- OpenSSL для шифрування та дешифрування даних, що гарантує безпеку зберігання та передачі конфіденційної інформації.

3.2. Структура програмного додатку

Розроблений додаток використовує архітектуру Model-View-Controller (MVC), яка забезпечує добре організовану модульну структуру, полегшуючи як розробку, так і підтримку[27]. Ця архітектура є основним шаблоном проектування в програмах Rails, що розділяє програму на три основні взаємопов’язані компоненти: моделі, представлення та контролери. Архітектура MVC наведена на рис. 3.1.

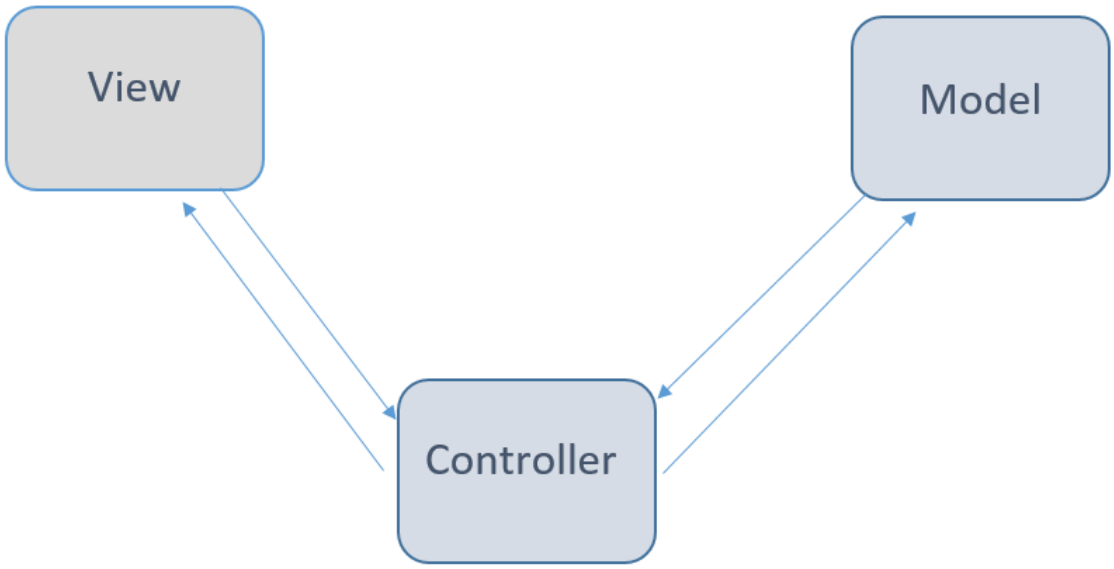

Рис. 3.1 – Архітектура додатку

Рівень моделі відповідає за бізнес-логіку та керування даними. Моделі безпосередньо взаємодіють з базою даних, інкапсулюючи дані та надаючи методи для маніпулювання ними. Наприклад, модель User обробляє інформацію користувача, таку як деталі автентифікації, профілі та зв'язки з іншими моделями, такими як Post і UserLog. Подібним чином модель публікації керує вмістом, створеним користувачами, гарантуючи, що кожна публікація правильно пов'язана з користувачем і містить такі атрибути, як заголовок, текст і позначки часу. Модель UserLog відстежує дії користувачів у програмі, записуючи такі дії, як спроби входу та створення публікацій. ActiveRecord, фреймворк Rails Object-Relational Mapping (ORM), використовується для керування цими взаємодіями, абстрагуючи операції бази даних у об'єктах Ruby, які легко керувати[28].

Рівень контролера діє як посередник між моделями та представленнями. Контролери відповідають за обробку вхідних HTTP-запитів, взаємодіють із рівнем моделі для отримання або оновлення даних і рендеринг відповідного представлення. Кожен контролер відповідає певній моделі або групі споріднених моделей[29]. Наприклад, UsersController керує діями, пов'язаними з обліковими записами користувачів, включаючи реєстрацію, вхід і оновлення профілю. PostsController виконує операції зі

створення, редагування, відображення та видалення публікацій, гарантуючи, що лише авторизовані користувачі можуть змінювати вміст. UserLogsController керує журналюванням і відображенням дій користувачів, надаючи цінну інформацію про поведінку користувачів і використання програми. Контролери використовують потужні параметри Rails для підвищення безпеки, запобігання небажаному масовому призначенню атрибутів і забезпечення обробки лише дозволених даних.

Рівень View відповідає за представлення даних користувачеві. Представлення створюють HTML, CSS і JavaScript, які формують інтерфейс користувача. У нашій програмі представлення відображаються за допомогою Slim, легкої мови шаблонів, яка спрощує створення HTML. Перегляди розроблені таким чином, щоб бути ефективними та швидко реагувати, забезпечуючи безперебійну роботу користувачів на різних пристроях. Загальні макети та частини використовуються для підтримки узгодженого дизайну в усій програмі[30]. Наприклад, спільний макет може містити елементи навігації, нижні колонтитули та загальні ресурси CSS/JavaScript. Спеціальні перегляди створюються для різних дій, таких як перелік публікацій, відображення окремих публікацій або показ профілів користувачів. Інтеграція Bootstrap забезпечує адаптивну структуру дизайну, покращуючи візуальну привабливість і доступність програми. Спеціальний JavaScript, керований через Webpacker, додає інтерактивність і покращує взаємодію з користувачами, вмикаючи такі функції, як динамічна перевірка форм і оновлення в реальному часі.

Обробка запитів і відповідей є фундаментальним аспектом будь-якої веб-програми. У розробленому додатку, створеному дотримуючись архітектури Model-View-Controller (MVC), цей процес є структурованим і ефективним[31]. Нижче наведено детальний робочий процес керування запитами та відповідями в нашій системі.

Коли користувач взаємодіє з програмою, наприклад, натискаючи посилання або надсилаючи форму, ініціюється HTTP-запит. Цей запит

проходить кілька рівнів, перш ніж досягти серверної логіки, яка його обробляє[32]. Ось покрокова розбивка життєвого циклу запиту:

1. Взаємодія з користувачем: користувач виконує дію у своєму веб-браузері, наприклад переходить на сторінку або надсилає форму.
2. HTTP-запит: браузер надсилає HTTP-запит на сервер. Цей запит містить такі деталі, як метод HTTP (GET, POST, PUT, DELETE), URL-адресу, заголовки та будь-які дані, надіслані разом із запитом (наприклад, дані форми).

Першою точкою контакту для вхідного запиту є маршрутизатор Rails, визначений у config/routes.rb[33]. Робота маршрутизатора полягає в тому, щоб відобразити запит на певну дію контролера на основі шаблону URL-адреси та методу HTTP.

Коли маршрутизатор визначає відповідний контролер і дію, він передає запит цьому контролеру. Контролер діє як посередник між запитом і бізнес-логікою програми. Він обробляє запит, взаємодіє з моделлю та готує відповідь[34].

Моделі інкапсулюють бізнес-логіку та керування даними програми. Коли контролер взаємодіє з моделлю, він зазвичай виконує такі операції, як запит до бази даних, перевірка даних і керування асоціаціями. ActiveRecord, ORM Rails, перетворює операції моделі в SQL-запити, абстрагуючись від складності взаємодії з базою даних.

Після того, як контролер обробив запит і взаємодіяв з моделлю, він візуалізує представлення для представлення даних користувачеві. Представлення відповідають за створення HTML, CSS і JavaScript[35]. Після відтворення представлення Rails пакує HTML-відповідь і надсилає його назад у браузер користувача. Потім браузер відображає відтворений HTML, завершуючи цикл запит-відповідь.

Amazon Web Services (AWS) є невід’ємною частиною інфраструктури розробленої програми, зокрема завдяки використанню Amazon S3 (Simple Storage Service). Amazon S3 пропонує безпечне, довговічне та добре

масштабоване сховище об’єктів, що є важливим для обробки завантажених користувачами файлів, таких як зображення профілю, документи та інші медіафайли. Ця служба гарантує, що файли зберігаються безпечно та можуть бути ефективно отримані.

Програма використовує S3 для зберігання завантажених користувачами файлів, використовуючи можливості S3 для забезпечення надійних і масштабованих рішень для зберігання. Процес завантаження файлів на хмарне сховище зображено на рисунках 3.2. та 3.3.

Рис. 3.2 – Завантаження файлу на хмарне сховище у додатку

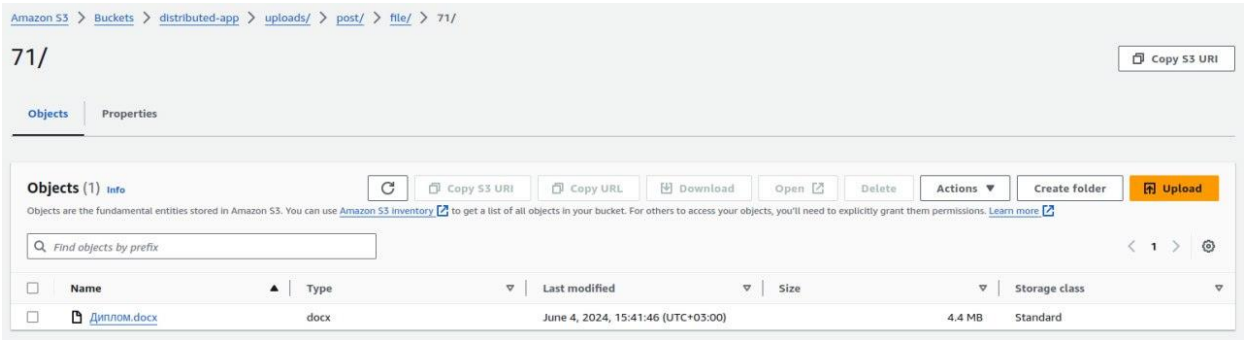

Рис. 3.3 – Відображення завантаженого файлу у S3

Процес інтеграції складається з кількох ключових етапів:

- Створення сегментів і керування ними: сегменти є основними контейнерами в S3, де зберігаються дані[36]. Кожен файл, завантажений користувачем, зберігається у визначеному сегменті, який можна налаштувати за допомогою спеціальних політик для контролю доступу та дозволів. Ці конфігурації забезпечують безпеку даних і доступ до них лише авторизованим користувачам.
- Контроль доступу та безпека: Amazon S3 забезпечує багаторівневий захист для захисту даних. Політики контролю доступу, визначені за допомогою AWS Identity and Access Management (IAM), гарантують, що лише авторизовані користувачі та програми можуть отримувати доступ або змінювати дані, що зберігаються в S3. Крім того, вбудоване шифрування S3 гарантує захист даних у стані спокою за допомогою галузевих стандартних алгоритмів шифрування.
- Завантаження та отримання даних: коли користувач завантажує файл, програма обробляє файл, а потім завантажує його в призначене відро S3. Цим процесом керують за допомогою AWS SDK для Ruby, який забезпечує безперебійний інтерфейс для взаємодії з S3. SDK обробляє деталі завантаження, гарантуючи, що файл правильно зберігається та може бути ефективно отриманий у разі потреби.
- Управління версіями та життєвим циклом: S3 пропонує такі функції, як керування версіями та життєвим циклом, які можна налаштувати для більш ефективного керування даними. Контроль версій дозволяє програмі зберігати кілька версій об’єкта, забезпечуючи спосіб відновлення після ненавмисного перезапису або видалення. Управління життєвим циклом забезпечує автоматичний перехід об’єктів до різних класів зберігання або

видалення об’єктів після визначеного періоду, оптимізуючи витрати на зберігання.

Підключенням до AWS S3 керують за допомогою змінних середовища та облікових даних Rails, що забезпечує безпечне керування ключами доступу та іншою конфіденційною інформацією. Конфігурація передбачає налаштування необхідних змінних середовища, таких як AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY та регіон. Ці конфігурації завантажуються в програму під час виконання, що дозволяє програмі автентифікуватися та безпечно взаємодіяти з S3[37].

AWS надає надійні функції безпеки, щоб забезпечити захист даних, що зберігаються в S3:

- Ролі та політики IAM: вони керують доступом до ресурсів AWS, гарантуючи, що лише авторизовані користувачі та програми можуть взаємодіяти з S3.
- Шифрування: дані шифруються як під час передачі, так і під час передачі. Шифрування на стороні сервера S3 використовує AES-256, а SSL/TLS забезпечує безпечну передачу даних.

Розроблена програма використовує Citus, розширення для PostgreSQL, для досягнення архітектури розподіленої бази даних. Це налаштування покращує масштабованість і продуктивність програми, розподіляючи дані та запити між кількома вузлами.

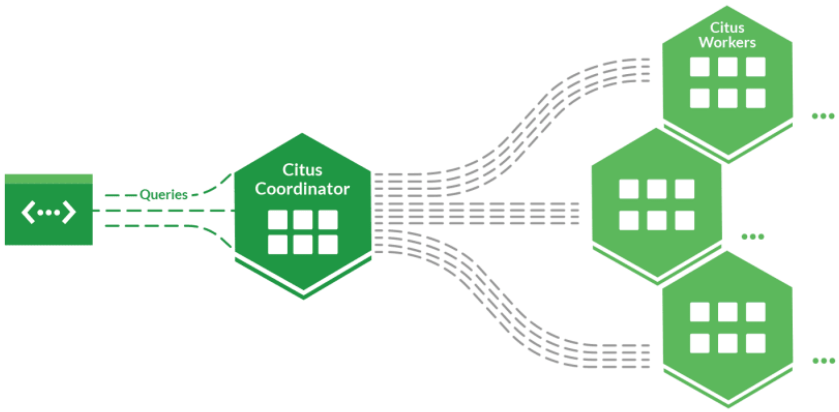

Рис. 3.4 – Діаграма архітектури розширення Citus

Citus перетворює стандартну базу даних PostgreSQL на розподілену систему, розділяючи дані на сегменти, які розподіляються між кількома серверами або вузлами[38]. Цей розподіл дозволяє паралельно обробляти запити, значно покращуючи продуктивність і масштабованість.

Програма використовує один екземпляр PostgreSQL як координатор і додаткові екземпляри як робочі вузли. Вузол-координатор керує метаданими про розподіл даних і планування виконання запиту. Це основна точка взаємодії програми під час виконання операцій з базою даних[39].

Вузол-координатор: вузол-координатор є основним вузлом, який обробляє вхідні запити SQL. Він керує метаданими про розподіл даних між робочими вузлами та створює плани виконання для запитів. Координатор не зберігає дані сам, а направляє запити до відповідних робочих вузлів.

Робочі вузли: робочі вузли зберігають фактичні сегменти даних і виконують запити, які розповсюджує координатор. Кожен робочий вузол обробляє підмножину даних, що забезпечує паралельну обробку запитів і ефективне керування даними[40].

Citus інстальовано як на координаційному, так і на робочому вузлах як розширення PostgreSQL. Екземпляри PostgreSQL налаштовані на розпізнавання та взаємодію один з одним, утворюючи цілісну розподілену систему.

Для розподілу таблиць використовується команда *create_distributed_table()*. Наприклад, розповсюдження таблиці *user_logs* на основі ключа розповсюдження *user_id* можна виконати наступним чином:

SELECT create_distributed_table('user_logs', 'user_id');

Ця команда розбиває таблицю *user_logs* на фрагменти, які потім розподіляються між робочими вузлами[41]. Ключ розподілу, в даному випадку *user_id*, визначає спосіб розподілу даних, забезпечуючи рівномірний розподіл даних у кластері

Коли дані вставляються в розподілену таблицю, вузол-координатор направляє дані до відповідних робочих вузлів на основі ключа розподілу. Кожен робочий вузол зберігає призначені сегменти, забезпечуючи рівномірний розподіл даних у кластері.

Для обробки запитів вузол-координатор отримує SQL-запити від програми, створює план виконання та розповсюджує частини запиту на відповідні робочі вузли для паралельної обробки. Робочі вузли виконують свої частини запиту та повертають результати координатору, який агрегує результати та надсилає кінцевий результат до програми.

3.3. Розробка підсистеми захисту сховища даних

3.3.1. Розробка автентифікації

Як базовий рівень захисту було розроблено процес автентифікації. Коли користувач намагається увійти, ініціюється складний багатоетапний процес автентифікації користувача. Суть цього процесу полягає в методі *authenticate!*, який викликається Warden, проміжне програмне забезпечення, що використовується для автентифікації.

Метод починається зі спроби знайти користувача в базі даних за допомогою наданої електронної пошти або іншого ключа автентифікації. Це

робиться за допомогою методу *find_for_database_authentication*, який створює запит для визначення місцезнаходження користувача.

```ruby
def find_for_database_authentication(warden_conditions)
    conditions = warden_conditions.dup
    if (login = conditions.delete(:login))
        where(conditions.to_h).where(["lower(email) = :value", { value: login.downcase }]).first
    else
        where(conditions.to_h).first
    end
end
```

Рис. 3.5 – Метод find_for_database_authentication

Цей метод гарантує, що пошук електронної пошти не враховує регістр за допомогою нижньої функції в SQL-запиті. Коли користувача знайдено, метод переходить до перевірки пароля за допомогою методу *valid_password?*.

Метод *valid_password?* відіграє вирішальну роль у забезпеченні відповідності наданого пароля збереженому зашифрованому паролю. Він використовує BCrypt для розшифровки збереженого пароля, а потім порівнює його з поданим паролем:

```ruby
def valid_password?(password)
    bcrypt = ::BCrypt::Password.new(encrypted_password)
    password = ::BCrypt::Engine.hash_secret(password, bcrypt.salt)
    Devise.secure_compare(password, encrypted_password)
end
```

Рис. 3.6 – Метод valid_password?

Цей метод спочатку розшифровує збережений пароль за допомогою BCrypt, потім хешує надісланий пароль тією самою сіллю (випадкові дані, що надходять як додатковий вхід до односторонньої функції, яка хешує дані, пароль або парольну фразу), яка використовувалася під час шифрування, і, нарешті, порівнює два хеші за допомогою *Devise.secure_compare*, щоб запобігти атакам за часом.

Якщо перевірка пароля пройшла успішно та якщо користувач вибрав опцію «Запам’ятати мене», викликається метод *remember_me*, щоб зберегти

сеанс під час перезапуску браузера. Метод *remember_me* генерує унікальний маркер і встановлює мітку часу:

```
def remember_me!
    self.remember_token = Devise.friendly_token
    self.remember_created_at = Time.now.utc
    save(validate: false)
end
```

Рис. 3.7 – Метод remember_me

Цей маркер зберігається в файлі cookie, що дозволяє запам'ятовувати користувача під час наступних сеансів. В кінці, якщо всі перевірки пройшли успішно, викликається метод *success!*, який позначає автентифікацію як успішну та встановлює сеанс користувача. Якщо будь-який крок завершується невдачею, наприклад користувач не знайдений або пароль неправильний, викликається *fail* із відповідним повідомленням про помилку:

```
def authenticate!
    resource = mapping.to.find_for_database_authentication(authentication_hash)
    return fail(:not_found_in_database) unless resource

    if validate(resource) { resource.valid_password?(password) }
      remember_me(resource)
      resource.after_database_authentication
      success!(resource)
    end
end
```

Рис. 3.8 – Метод authenticate!

Весь процес автентифікації забезпечує надійну перевірку облікових даних користувача та належне керування сеансами. Це включає в себе поєднання пошуку користувача, перевірки пароля, збереження сеансу та зворотних викликів після автентифікації, усе це організовано за допомогою чітко визначених методів Devise та проміжного програмного забезпечення Warden.

Двофакторна автентифікація додає додатковий рівень безпеки до процесу автентифікації, вимагаючи не лише пароля, але й коду, надісланого

електронною поштою. Цей метод підвищує безпеку, гарантуючи, що навіть якщо пароль зламано, обліковий запис залишається захищеним.

Коли обліковий запис користувача створюється, 2FA автоматично вмикається. Це передбачає створення унікального секретного ключа OTP (One Time Password) для користувача та збереження його в базі даних. Модель користувача налаштована на обробку функцій OTP, гарантуючи, що секретний ключ зашифровано для безпеки.

Модель користувача містить методи для створення та перевірки OTP. Секретний ключ зашифровано для безпеки. Після створення користувача метод *enable_two_factor_authentication* генерує унікальний секретний ключ і зберігає його в записі користувача:

```
def enable_two_factor_authentication
    self.otp_secret = User.generate_otp_secret
    self.otp_required_for_login = true
    save!
end
```

Рис. 3.9 – Метод enable_two_factor_authentication

Коли користувач із увімкненою 2FA намагається увійти, він починає з введення електронної пошти та пароля у формі входу. Дія створення в *SessionsController* обробляє цей запит. Якщо облікові дані правильні, програма перевіряє, чи ввімкнено 2FA для користувача. Якщо так, користувач перенаправляється на сторінку перевірки одноразового пароля, а на його електронну адресу надсилається код одноразового пароля.

```
if resource.otp_required_for_login
    session[:otp_user_id] = resource.id
    resource.send_two_factor_authentication_code
    sign_out(resource)
    redirect_to user_verify_otp_path and return
else
    sign_in_and_redirect resource, event: :authentication
    log_user_action(resource, 'login', 'User logged in') if resource.persisted?
end
```

Рис. 3.10 – Блок, що відповідає за перевірку двофакторної автентифікації

Після успішної перевірки пароля, ідентифікатор сеансу користувача зберігається в *session[:otp_user_id]*, а код OTP генерується та надсилається електронною поштою.

Для надсилання одноразового коду на пошту використовується поштова програма (*UserMailer*). *Mailer* — це компонент, який допомагає надсилати електронні листи з вашої програми. Він працює як контролер, але спеціально розроблений для обробки електронної пошти. Поштові програми використовуються для визначення методів, які відповідають різним шаблонам електронних листів, і можуть використовуватись для надсилання сповіщень, підтверджень та інших типів електронних листів користувачам.

```
class UserMailer < ApplicationMailer
    def send_otp(user)
      @otp_code = user.otp_code
      mail(to: user.email, subject: 'Your OTP Code')
    end
end
```

Рис. 3.11 – Поштова програма для надсилання одноразового коду

Метод *send_two_factor_authentication_code* у моделі користувача запускає цю поштову програму.

```
def send_two_factor_authentication_code
    UserMailer.send_otp_code(self).deliver_now
end
```

Рис. 3.12 – Метод запуску поштової програми

На сторінці підтвердження OTP користувач вводить код, отриманий електронною поштою. Дія *verify_otp* у *SessionsController* обробляє цей вхід, перевіряючи код за допомогою методу *authenticate_otp*.

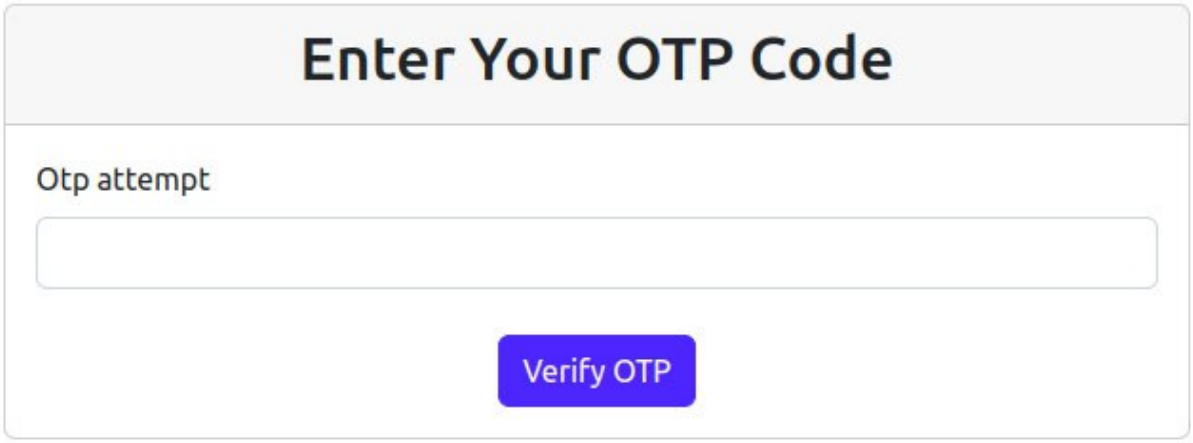

Рис. 3.13 – Сторінка перевірки OTP

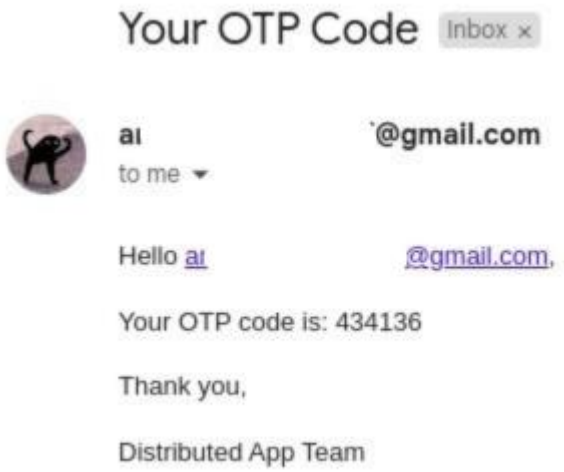

Рис. 3.14 – Лист на пошті з одноразовим кодом

```ruby
def verify_otp
    user = User.find(session[:otp_user_id])
    if user.authenticate_otp(params[:otp_attempt], drift: 60)
      session[:otp_user_id] = nil
      sign_in user
      redirect_to after_sign_in_path_for(user)
    else
      flash[:alert] = 'Invalid OTP'
      log_user_action(user, 'otp attempt', 'User entered wrong otp code')
      render 'verify_otp' and return
    end
end
```

Рис. 3.15 – Дія перевірки одноразового коду

Метод *authenticate_otp* перевіряє наданий код OTP на збережений секрет OTP користувача, дозволяючи невеликий дрейф у часі для врахування будь-яких розбіжностей між сервером і пристроєм користувача.

Також було вирішено імлементувати CAPTCHA, для додаткового захисту від можливих атак ботів. Інтеграція CAPTCHA підвищує безпеку, гарантуючи, що лише люди можуть надсилати форми, захищаючи від автоматизованих атак. У розробленому додатку використовується сервіс reCAPTCHA від Google, а його перевірка інтегрована в процес входу.

Для інтеграції reCAPTCHA камінь recaptcha додається до Gemfile і налаштовується за допомогою сайту та секретних ключів. Ці ключі отримані з консолі адміністратора Google reCAPTCHA. У форму входу reCAPTCHA додається за допомогою помічника recaptcha_tags, який генерує необхідні HTML і JavaScript для відображення віджета reCAPTCHA.

Рис. 3.16 – Відображення віджету reCAPTCHA

Метод *verify_recaptcha* викликається в *SessionsController*, щоб переконатися, що CAPTCHA було успішно завершено перед продовженням автентифікації. Він перевіряє відповідь CAPTCHA за допомогою служби Google reCAPTCHA, щоб перевірити її автентичність. Якщо перевірка CAPTCHA не вдається, відображається повідомлення про помилку, і користувач повертається на сторінку входу.

Останньою інтеграцією пов’язаною в автентифікацією ж імплементація SSO (Single Sign-On) в додаток. Інтеграція єдиного входу дозволяє користувачам входити за допомогою своїх облікових записів Google або Facebook, спрощуючи процес автентифікації та підвищуючи зручність для

користувачів. Для реалізації єдиного входу було використано проміжне програмне забезпечення OmniAuth.

Для початку потрібно налаштувати проміжне програмне забезпечення з ідентифікаторами клієнтів і секретними ключами для Google і Facebook, які зберігаються в облікових даних Rails.

```
Rails.application.config.middleware.use OmniAuth::Builder do
    provider :google_oauth2, Rails.application.credentials.dig(:google, :client_id), Rails.application.credentials.dig(:google, :client_secret), {
      scope: 'userinfo.email, userinfo.profile',
      prompt: 'select_account',
      redirect_uri: 'http://localhost:3000/users/auth/google_oauth2/callback'
    }

    provider :facebook, Rails.application.credentials.dig(:facebook, :app_id), Rails.application.credentials.dig(:facebook, :app_secret), {
      scope: 'email',
      info_fields: 'email,name',
      redirect_uri: 'http://localhost:3000/users/auth/facebook/callback'
    }
end

OmniAuth.config.allowed_request_methods = [:post, :get]
OmniAuth.config.silence_get_warning = true
```

Рис. 3.17 – Конфігурація бібліотеки OmniAuth

- *provider :google_oauth2, ...:* налаштовує Google як постачальника автентифікації.
- *provider facebook, ...:* налаштовує Facebook як постачальника автентифікації.
- *scope:* вказує дозволи, запитувані від користувача.
- *prompt:* визначає, як користувачеві буде запропоновано вибрати обліковий запис.
- *redirect_uri:* вказує, куди перенаправляти після успішної автентифікації.
- *OmniAuth.config.allowed_request_methods = [:post, :get]:* дозволяє OmniAuth обробляти запити POST і GET.
- *OmniAuth.config.silence_get_warning = true:* забороняє попередження, пов’язані із запитами GET.

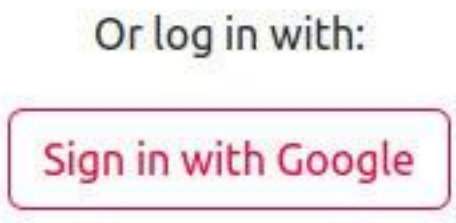

Рис. 3.18 – Відображення кнопки для єдиного входу

Для обробки зворотніх викликів був розроблений модуль *Users::OmniauthCallbacksController*, що виконує процес автентифікації, коли користувач повертається зі сторінки входу в Google або Facebook.

```ruby
class Users::OmniauthCallbacksController < Devise::OmniauthCallbacksController
  skip_before_action :verify_authenticity_token, only: [:google_oauth2]

  def google_oauth2
    handle_auth "Google"
  end

  def handle_auth(kind)
    @user = User.from_omniauth(request.env['omniauth.auth'])

    if @user.persisted?
      sign_in_and_redirect @user, event: :authentication
      set_flash_message(:notice, :success, kind: kind) if is_navigational_format?
    else
      session["devise.#{kind.downcase}_data"] = request.env['omniauth.auth'].except("extra")
      redirect_to new_user_registration_url, alert: @user.errors.full_messages.join("\n")
    end
  end
end
```

Рис. 3.19 – Модуль OmniauthCallbacksController

Коли користувач намагається ввійти через цих постачальників, він перенаправляється на URL-адресу зворотного виклику, яка обробляється методами *google_oauth2* і *facebook* у контролері. Обидва методи викликають *handle_auth* із рядком, що ідентифікує постачальника.

Метод *handle_auth* обробляє дані автентифікації з *request.env['omniauth.auth']*. Він використовує ці дані, щоб знайти або створити користувача через *User.from_omniauth*. Цей метод перевіряє, чи існує користувач із заданим провайдером та UID; якщо ні, він створює його,

використовуючи надану інформацію, включаючи електронну адресу, ім’я та зображення.

```ruby
def self.from_omniauth(auth)
  user = User.find_or_create_by(provider: auth.provider, uid: auth.uid) do |user|
    user.email = auth.info.email
    user.password = Devise.friendly_token[0, 20]
  end

  user.update(
    provider: auth.provider,
    uid: auth.uid,
  ) if user.persisted?

  user
end
```

Рис. 3.20 – Метод from_omniauth

Якщо користувача успішно знайдено або створено *(@user.persisted?* повертає *true*), викликається метод *sign_in_and_redirect*, щоб увійти користувача та перенаправити його на відповідну сторінку, установивши повідомлення про успіх. Якщо користувач не продовжує (тобто виникає помилка), дані автентифікації зберігаються в сеансі, а користувач перенаправляється на сторінку реєстрації з повідомленням про помилку з докладним описом проблем.

Метод моделі користувача *from_omniauth* має вирішальне значення, оскільки він обробляє логіку створення або пошуку користувача з даних OmniAuth. Він витягує необхідну інформацію з хешу автентифікації, забезпечуючи однозначну ідентифікацію кожного користувача за допомогою свого постачальника та UID.

3.3.2. Розробка авторизації

Авторизація в програмі керується за допомогою політик, які визначають дозволи користувача для різних дій на моделях. Ці політики використовуються для того, щоб лише авторизовані користувачі могли виконувати певні дії, наприклад створювати, оновлювати або видаляти

записи[42]. Політики створюються для кожної моделі, щоб визначити правила доступу до ресурсів і їх зміни.

```ruby
class PostPolicy
    attr_reader :user, :post

    def initialize(user, post)
      @user = user
      @post = post
    end

    def show?
      true
    end

    def create?
      user.present?
    end

    def update?
      user.admin? || post.user == user
    end

    def destroy?
      user.admin?
    end
end
```

Рис. 3.21 – Політики для моделі Post

Методи, що розроблені в цьому класі націлені на надання доступу користувачу до певної дії з даними. Метод *initialize(user, post):* ініціалізує політику з поточним користувачем і авторизованою публікацією. Він призначає користувача та публікацію змінним екземплярів для використання в інших методах. *show?:* визначає, чи може користувач переглядати певну публікацію. Це дозволяє кожному користувачу переглядати публікації. *create?:* визначає, чи може користувач створювати нову публікацію. Він дозволяє створення, лише якщо користувач увійшов у систему *(user.present?).*

update?: визначає, чи може користувач оновлювати певну публікацію. Він дозволяє оновлення, якщо користувач є адміністратором або власником публікації. *destroy?:* визначає, чи може користувач видалити певну публікацію. Він дозволяє видалення, лише якщо користувач є адміністратором. У контролерах політики застосовуються, щоб гарантувати, що лише авторизовані користувачі можуть виконувати певні дії.

Області використовуються для фільтрації записів на основі правил авторизації. Область політики визначає логіку для отримання записів, які користувач має право переглядати.

```ruby
class PostPolicy
  class Scope
    attr_reader :user, :scope

    def initialize(user, scope)
      @user = user
      @scope = scope
    end

    def resolve
      if user.admin?
        scope.all
      else
        scope.where(user: user)
      end
    end
  end
end
```

Рис. 3.22 – Області для політик моделі Post

Метод *resolve* в області дії політики визначає логіку для отримання записів. Наприклад, адміністратори можуть бачити всі дописи, а звичайні користувачі – лише власні.

3.3.3. Розробка модулю шифрування

Процес шифрування та дешифрування в програмі гарантує надійне зберігання та доступ до конфіденційних даних. Цим процесом керують модуль шифрування та менеджер ключів.

Модуль шифрування виконує шифрування та дешифрування даних за допомогою AES-256-GCM, симетричного алгоритму шифрування. Модуль містить методи як для шифрування, так і для дешифрування.

```ruby
ALGORITHM = 'aes-256-gcm'
IV_LENGTH = 12

def self.encrypt(plain_text)
  layers = Helpers::Encryption::KeyManagment.keys.size
  encrypted_data = plain_text
  ivs = []
  auth_tags = []
  digests = []

  layers.times do |i|
    cipher = OpenSSL::Cipher.new(ALGORITHM)
    cipher.encrypt
    iv = cipher.random_iv[0, IV_LENGTH]
    raise "IV must be 12 bytes" unless iv.bytesize == IV_LENGTH
    cipher.key = Helpers::Encryption::KeyManagment.key(i)
    encrypted = cipher.update(encrypted_data) + cipher.final
    auth_tag = cipher.auth_tag
    digest = OpenSSL::Digest::SHA256.digest(encrypted)

    ivs << Base64.strict_encode64(iv)
    auth_tags << Base64.strict_encode64(auth_tag)
    digests << Base64.strict_encode64(digest)
    encrypted_data = Base64.strict_encode64(encrypted)
  end

  {
    data: encrypted_data,
    ivs: ivs,
    auth_tags: auth_tags,
    digests: digests,
    key_version: layers - 1
  }
end
```

Рис. 3.23 – Метод шифрування

```ruby
def self.decrypt(encrypted_data)
  layers = encrypted_data[:key_version] + 1
  decrypted_data = Base64.strict_decode64(encrypted_data[:data])

  (layers - 1).downto(0) do |i|
    decipher = OpenSSL::Cipher.new(ALGORITHM)
    decipher.decrypt
    iv = Base64.strict_decode64(encrypted_data[:ivs][i])
    raise "IV must be 12 bytes" unless iv.bytesize == IV_LENGTH
    decipher.iv = iv
    decipher.key = Helpers::Encryption::KeyManagment.key(i)
    decipher.auth_tag = Base64.strict_decode64(encrypted_data[:auth_tags][i])

    decrypted = decipher.update(decrypted_data) + decipher.final
    digest = OpenSSL::Digest::SHA256.digest(decrypted)

    raise "Data integrity check failed" unless digest == Base64.strict_decode64(encrypted_data[:digests][i])

    decrypted_data = decrypted
  end

  decrypted_data
end
```

Рис. 3.24 – Метод дешифрування

В даному модулі розроблено два методи: *encrypt(plain_text)* – шифратор та *decrypt(encrypted_data)* – дешифратор:

- *encrypt(plain_text):* Шифрує заданий звичайний текст. Він використовує кілька рівнів шифрування, кожен з яких має унікальний вектор ініціалізації (IV), тег авторизації та дайджест. Остаточні зашифровані дані разом із IV, тегами авторизації, дайджестами та версією ключа повертаються як хеш.
- *decrypt(encrypted_data):* розшифровує дані зашифровані дані. Він ітеративно розшифровує дані шар за шаром, використовуючи відповідні ключі, IV, теги авторизації та дайджести.

Менеджер ключів забезпечує генерацію та керування ключами шифрування. Це гарантує надійне зберігання та доступ до ключів.

```ruby
module KeyManagment
  ALGORITHM = 'aes-256-gcm'

  def self.keys
    @keys ||= [
      Rails.application.credentials.dig(:encryption_keys, :primary),
      Rails.application.credentials.dig(:encryption_keys, :secondary),
      Rails.application.credentials.dig(:encryption_keys, :tertiary)
    ].compact
  end

  def self.key(index)
    [keys[index]].pack('H*')
  end
end
```

Рис. 3.25 – Метод менеджера ключів

Модуль має два методи, які працюють наступним чином:

- *keys*: отримує масив ключів шифрування з облікових даних Rails. Для безпеки ключі зберігаються у файлі облікових даних.
- *key(index)*: Повертає ключ за вказаним індексом, перетворюючи його з шістнадцяткового рядка на двійковий формат, придатний для використання OpenSSL.

Під час шифрування даних метод шифрування викликається з відкритим текстом, який потрібно зашифрувати. Метод виконує наступні кроки: дані шифруються на кількох рівнях, причому кожен рівень використовує окремий ключ і генерує унікальний IV та тег авторизації[43]. Для кожного рівня 12-байтовий IV генерується за допомогою методу *random_iv* шифру. Ключ для поточного рівня отримується за допомогою менеджера ключів. Потім дані шифруються за допомогою призначеного ключа та генерується IV, а отриманий тег авторизації зберігається. Дайджест SHA-256 зашифрованих даних обчислюється та зберігається. Нарешті, зашифровані дані, IV, теги автентифікації та дайджести кодуються за допомогою Base64, щоб гарантувати їх збереження у вигляді рядків.

Під час дешифрування даних метод дешифрування викликається із зашифрованими даними. Метод виконує наступні етапи: дані розшифровуються в порядку, зворотному шифруванню, починаючи з останнього рівня до першого. Зашифровані дані, IV, теги авторизації та дайджести декодуються з Base64 у вихідні двійкові формати. Для кожного рівня обчислюється дайджест SHA-256 розшифрованих даних і порівнюється зі збереженим дайджестом. Якщо вони не збігаються, виникає помилка перевірки цілісності. Потім дані розшифровуються за допомогою призначеного ключа та IV для поточного рівня.

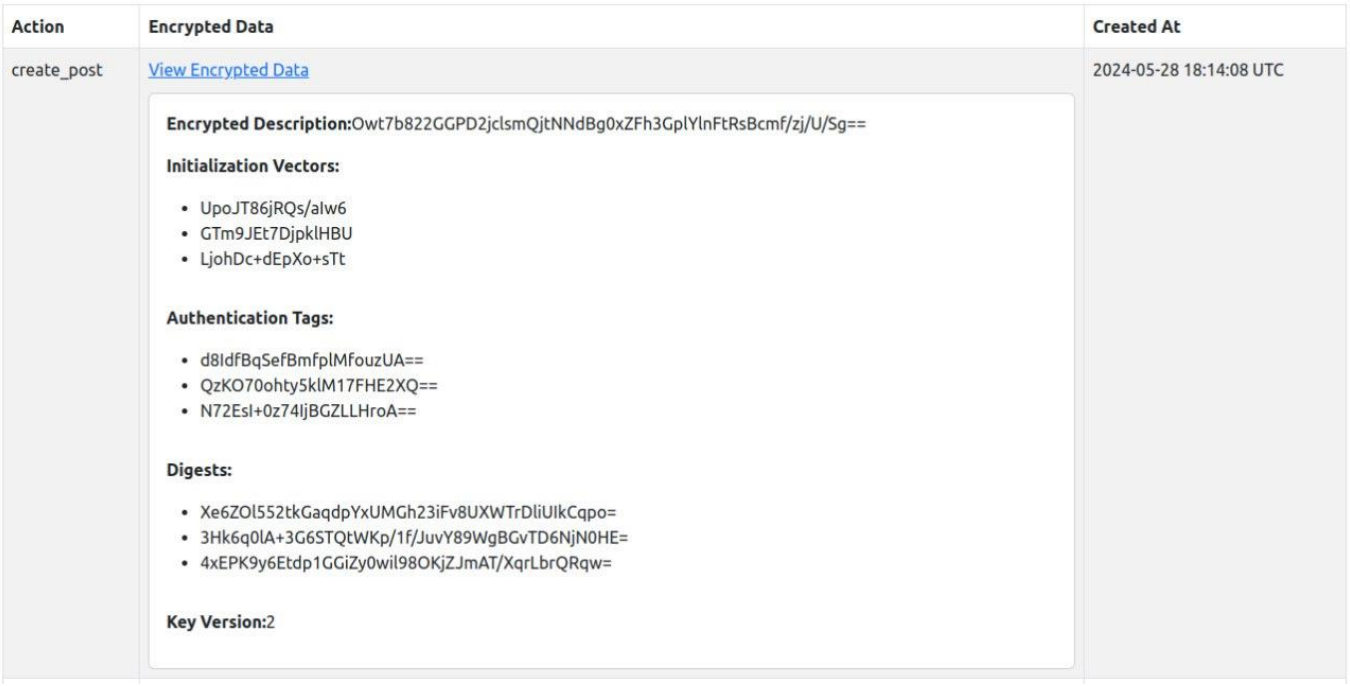

Рис. 3.26 – Вивід зашифрованих даних

Висновки до розділу 3

У цьому розділі виконано розробку програмного продукту для розподіленої системи зберігання даних, що включає застосування захищеної архітектури збору, шифрування та обробки даних. Зроблено огляд засобів реалізації програмного додатку з обґрунтуванням та переліком необхідних сервісів для роботи додатку. Уточнено мінімальні системні вимоги для коректної роботи розробленої системи, обґрунтовано кросплатформенність запропонованого рішення та визначено основні завдання, які має вирішувати програмне забезпечення.

Розглянуто структуру програмного додатку та описано основні функціональні рішення, використані при розробці. Подано короткі змістовні інструкції по роботі з системою. Визначено основні кроки розробленої системи, що формують та взаємопов’язують основні компоненти програмного рішення. Описано процеси генерації, шифрування, обробки, зберігання та резервного копіювання даних, а також взаємодії за допомогою веб-інтерфейсу.

Процеси шифрування та дешифрування разом із керуванням ключами для захисту даних було впроваджено та протестовано. Інтеграція механізмів автентифікації (включаючи двофакторну автентифікацію) і авторизації забезпечує безпечний доступ до системи. Тестування системи показало ефективну роботу описаних процесів, безпечну обробку даних і коректне функціонування розподіленої бази даних у Citus.

РОЗДІЛ 4
РОЗРОБКА СТАРТАП-ПРОЕКТУ

Захист розподілених сховищ даних відіграє ключову роль у сучасному світі інформаційних технологій. Розвиток цієї сфери відбувається з великою швидкістю, асимілюючи новітні технологічні досягнення та інновації. Забезпечення ефективного захисту розподілених сховищ даних є критично важливим для гарантування безпеки та цілісності важливих даних.

Хоча деякі методики та підходи до захисту цих сховищ уже були представлені раніше, слід зазначити, що цей перелік далеко не вичерпний. Основна інноваційна характеристика розроблюваної підсистеми захисту полягає у використанні передових технологій та стратегій для забезпечення максимальної безпеки розподілених сховищ, особливо тих, що характеризуються високим рівнем обробки та зберігання даних.

На цьому етапі надзвичайно важливо визначити ключові напрямки розвитку та стратегію дій для створення ефективної, надійної та конкурентоспроможної підсистеми захисту. В подальших розділах будуть детально розглянуті маркетингові, організаційні, фінансово-економічні та комерційні аспекти, які супроводжують розробку та впровадження такої підсистеми захисту.

4.1. Маркетинговий аналіз стартап-проекту

У рамках цього проекту основна ідея проекту: додаток прикладного рівня мережі SDN, що матиме підключення до рівня керування та реалізовуватиме збір, агрегацію, акумуляцію та аналіз інформації про стан мережі. На основі отриманої інформації виконується процес конструювання трафіка, що дозволяє підвищити надійність мережі. Додаток може бути сконфігурований для різних обсягів даних, апаратних можливостей та потреб кінцевого користувача.

Таблиця 4.1.

Опис ідеї стартап-проекту

Зміст ідеї	Напрямки застосування	Вигоди для користувача
Підсистема захисту розподіленого сховища даних з шифруванням, авторизацією, двофакторною автентифікацією, SSO та використанням AWS S3 для зберігання файлів	Захист даних у розподілених сховищах	Забезпечення високого рівня безпеки даних, захист від несанкціонованого доступу та підвищення надійності зберігання даних
	Шифрування даних	Захист конфіденційних даних шляхом шифрування, що запобігає несанкціонованому доступу
	Двофакторна автентифікація та SSO	Підвищення рівня безпеки доступу до даних шляхом використання двох факторів автентифікації та єдиного входу для зручності користувачів
	Використання AWS S3 для зберігання файлів	Надійне і масштабоване зберігання файлів з високим рівнем доступності та безпеки

Проведемо аналіз потенційних техніко-економічних переваг ідеї та визначимо, чим ідея відрізняється від існуючих аналогів та замінників. А саме:

- визначимо перелік техніко-економічних властивостей та характеристик ідеї;
- визначимо попередні кола проектів-конкурентів, що вже існують на ринку та проведемо збір інформації щодо значень техніко-економічних показників для ідеї власного проекту та проектів-конкурентів;

- проведемо порівняльний аналіз показників: для власної ідеї визначимо показники, що мають а) гірші значення (W, слабкі); б) аналогічні (N, нейтральні) значення; в) кращі значення (S, сильні).

Таблиця 4.2.

Визначення сильних, слабких та нейтральних характеристик ідеї проекту

№	Техніко-економічні характеристики ідеї	Проекти				W	N	S
		Мій	1	2	3			
1.	Шифрування даних	+	+	+	-			+
2.	Авторизація	+	+	-	+			+
3.	Звичайна автентифікація	+	+	-	+		+	
4.	Двофакторна автентифікація	+	-	-	+			+
5.	SSO (єдина точка входу)	+	-	+	-		+	
6.	Використання AWS S бакетів для зберіганн файлів	+		-	+		+	
7.	Використання PostgreSQL та Citus	+	-	-	-	+		
8.	Масштабованість	+	+	+	+		+	
9.	Високий рівень безпеки	+	+	+	+			+

Визначений перелік слабких, сильних та нейтральних характеристик та властивостей ідеї потенційного товару є підґрунтям для формування його конкурентоспроможності. Згідно з таблицею, серед основних переваг розробленого продукту є шифрування даних, авторизація, звичайна автентифікація, двофакторна автентифікація, SSO (єдина точка входу),

використання AWS S3 для зберігання файлів, використання PostgreSQL та Citus, а також високий рівень безпеки та масштабованість, на що і варто зробити акцент. Порівняно з існуючими рішеннями, головними недоліками системи є відсутність підтримки деяких сучасних технологій, що варто врахувати в майбутньому.

4.2. Технологічний аудит ідеї проекту

Таблиця 4.3.

Технологічна здійсненність ідеї проекту

№	Ідея проекту	Технології її реалізації	Наявність технологій	Доступність технологій
1.	Шифрування даних	AES, RSA	Є в наявності	Доступна безкоштовно
2.	Авторизація	OAuth 2.0, Pundit	Є в наявності	Доступна безкоштовно
3.	Звичайна автентифікація	Devise	Є в наявності	Доступна безкошнтовно
4.	Двофакторна автентифікація	Google Authenticator, Email OTP	Є в наявності	Доступна безкоштовно
5.	SSO (єдина точка входу)	OAuth 2.0	Є в наявності	Доступна безкоштовно
6.	Використання AWS S3 бакетів для зберігання файлів	Amazon S3 SDK	Є в наявності	Доступна безкоштовно
7.	Використання PostgreSQL та Citus	PostgreSQL, Citus	Є в наявності	Доступна безкоштовно
8.	Масштабованість	Citus	Є в наявності	Доступна безкоштовно

9.	Високий рівень безпеки	TLS, HTTPS, VPN	Є в наявності	Доступна безкоштовно

Згідно з описаними технологіями, необхідними для реалізації проекту, створення і розробка підсистеми захисту розподіленого сховища даних є можливим. Усі необхідні технології є в наявності та доступні безкоштовно.

4.3. Аналіз ринкових можливостей запуску стартап-проекту

Здійснимо визначення ринкових можливостей, які можна використати під час ринкового впровадження проекту, та ринкових загроз, які можуть перешкодити реалізації проекту. Цей етап дозволяє спланувати напрями розвитку проекту із урахуванням стану ринкового середовища, потреб потенційних клієнтів та пропозицій проектів-конкурентів. У таблиці 4.4 наведена попередня характеристика потенційного ринку для розроблюваного стартап-проекту.

Таблиця 4.4.

Попередня характеристика потенційного ринку стартап-проекту

№	Показники стану ринку (найменування)	Характеристика
1.	Кількість головних гравців	>3
2.	Загальний обсяг продаж, грн/ум.од.	780 тис.
3.	Динаміка ринку (якісна оцінка)	Зростає 35% (до 2026 р.)
4.	Наявність обмежень для входу (характер обмежень)	Фінансові затрати, наявність кваліфікованих спеціалістів
5.	Специфічні вимоги до стандартизації та сертифікації	Наявність патенту, стандартизація від організацій IETF та ONF
6.	Середня норма рентабельності в галузі або по ринку, %	40

За результатами аналізу таблиці ринок є привабливим для входження по причинах достатнього рівня рентабельності, невеликої кількості конурентів та умови зростання ринку по прогнозах до 2026 року.

Подальший аналіз ринкових можливостей потребує характеристики потенційних груп клієнтів. Характеристика та орієнтовний перелік вимог до продукту стартап-проекту представлені в таблиці 4.5.

Таблиця 4.5.

Характеристика потенційних клієнтів стартап-проекту

№	Потреба, що формує ринок	Цільова аудиторія	Відмінності у поведінці різних потенційних цільових груп клієнтів	Вимоги до споживачів продукту
1.	Підвищення безпеки розподілених даних	Корпоративні клієнти, державні установи, медичні установи	Високий рівень вимог до конфіденційності та захисту даних	Надійність; Безпека; Відповідність законодавчим нормам
2.	Шифрування даних	IT-компанії, фінансові установи, приватні підприємства	Різні стандарти шифрування, потреба у інтеграції з існуючими системами	Надійне шифрування; Легкість інтеграції
3.	Двофакторна втентифікація та SSO	Підприємства з високими вимогами до безпеки, банки, e-commerce	Вимога до простоти використання та високого рівня безпеки	Простота впровадження; Зручність користування; Висока безпека
4.	Використання AWS S3 для берігання файлів	Всі типи клієнтів, що потребують надійного	Потреба у масштабованому та надійному зберіганні даних	Масштабованість; Висока доступність; Надійність

		зберігання даних		
5.	Гнучкість у впровадженні	Підприємства різних галузей	Різні вимоги до налаштувань та конфігурації системи	Підтримка індивідуальних налаштувань; Надання воротного зв'язку
6.	Масштабо-ваність	Великі підприємства, корпорації	Вимога до можливості збільшення обсягів зберігання та обробки даних	Легкість масштабування; Надійність при великих обсягах даних
7.	Збір та обробка даних SDN	Інформаційні служби, приватні підприємства	Висока вимога до точності та швидкості обробки даних	Висока продуктивність; Відповідність стандартам якості обробки даних

Проведемо аналіз ринкового середовища, а саме: складемо таблиці факторів, що сприяють ринковому впровадженню проекту (таблиця 4.7.), та факторів, що йому перешкоджають (таблиця 4.6.).

Таблиця 4.6.

Фактори загроз

№	Фактор	Зміст загрози	Можлива реакція компанії
1.	Недостатній рівень попиту на міжнародному ринку	Усі потенційні конкуренти та більшість потенційних клієнтів розташовані закордоном	Проведення маркетингової компанії закордоном з урахуванням особливостей ринку
2.	Незацікавленість потенційних клієнтів	Недовіра до технологічної	Проведення демонстрацій, надання тестового

	у продукті	складової продукту та автоматизації процесів, що досі відбувалися вручну	доступу до продукту клієнтам на перших кроках
3.	Недостатній рівень конкуренто-спроможності	Рішення конкурентів задовольняють базові потреби клієнтів	Вдосконалення цінової політики за покращення якості надання послуг, проведення маркетингової компанії за ключовими перевагами продукту
4.	Неоднозначна економічна ситуація	Відсутність фінансування, недостатня кількість інвестицій	Пошук додаткових шляхів надходження коштів, заощадження витрат
5.	Вузький спектр функціоналу	Недостатній функціонал для забезпечення потреб клієнтів	Побудова ітеративних процесів розробки, введення чітких систем планування із врахуванням ринкових потреб
6.	Проблеми з отриманням патентів та стандартизації	Масштабний бюрократичний прошарок для отримання юридичного захисту продукту	Отримання кваліфікованої юридичної допомоги від спеціалістів

Таблиця 4.7.

Фактори можливостей

№	Фактор	Зміст можливості	Можлива реакція компанії
1.	Залучення інвестицій	Наявність чіткого бізнес-плану, результатів аналізу ринку та конкуруючих рішень приваблює інвесторів та дозволяє пришвидшити вихід на ринок	Збільшення штату співробітників, пришвидшення темпів розробки та прискорення виходу на самоокупність
2.	Розширення клієнтської бази	Детальне дослідження ринку та проведення маркетингової кампанії потенційним клієнтам	Використання коштів клієнтів на покращення продукту та використання позитивного клієнтського досвіду в подальших маркетингових рішеннях
3.	Розширення функціоналу	Розширення функціоналу продукту для охоплення більшого ринкового сегменту	Ітеративний аналіз ринку для пошуку нових можливостей та приваблення клієнтів
4.	Партнертсво з великими гравцями ринку послуг IaaS і PaaS	Надання послуг лідерами серед провайдерів послуг SDN	Збільшення клієнтської бази, використання позитивного клієнтського досвіду, додаткове залучення коштів,

			отримання інсайтів щодо ринкових трендів
5.	Вихід на міжнародний ринок	Надання доступнішого продукту для представників міжнародного ринку	Забезпечення лідерських позицій у ринковому сегменті, використання позитивного клієнтського досвіду, розширення функціоналу згідно з потребами міжнародного ринку

Далі проведемо аналіз пропозиції: визначемо риси конкуренції на ринку (таблиця 4.8.).

Таблиця 4.8.

Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)
Тип конкуренції: *олігополія*	На ринку присутні декілька компаній, що надають подібні рішення	Пропозиція вигідніших умов впровадження та користування, розробка ключового функціоналу, відсутнього у конкурентів, проведення маркетингових кампаній з акцентами на ключових відмінностях продукту, надання тестового періоду користування
За рівнем конкурентної боротьби: *глобальний*	Існуючі рішення можуть бути впроваджені по всьому світу	Одразу організувати вихід на міжнародну арену та запровадити стратегії відповідно до критеріїв

		світового ринку
За галузевою ознакою: *внутрішньо-галузева*	Конкуренція в галузі мереж SDN	Розробка продукту яка вирішує обмежену кількість задач, але краще, ніж інші рішення
Товарно-родова конкуренція	Продукти-замінники можуть виконувати подібні функції	Створення більш якісної та кращої продукції
Нецінова конкуренція	Конкуренція створюється за допомогою вдосконалення якості продукції	Продукт має постійно вдосконалюватись та покращувати свої характеристики
Не марочна інтенсивність	Споживачів цікавить більше характеристики продукту, ніж під яким брендом випущено продукт	Покращити якісні характеристики продукту порівняно з конкурентними

Проведемо більш детальний аналіз умов конкуренції в галузі за моделлю 5 сил М. Портера.

М. Портер вирізняє п’ять основних факторів, що впливають на привабливість вибору ринку з огляду на характер конкуренції. Це:

- конкуренти, що вже є в галузі;
- потенційні конкуренти;
- наявність товарів-замінників;
- постачальники, що конкурують за ринкову владу;
- споживачі (аналогічно).

Сильні позиції продукту за кожним з факторів означають її можливість забезпечити необхідні темпи обороту капіталу та її здатність впливати на інших агентів ринку, диктуючи їм власні умови співпраці. Характеристики факторів моделі відрізняються для різних галузей та змінюються з часом.

Таблиця 4.9.

Аналіз конкуренції в галузі за М. Портером

	Прямі конкуренти в галузі	Потенційні конкуренти в галузі	Постачальники	Клієнти	Продукти-замінники
Складові аналізу	HashiCorp Vault, Thales CipherTrust, IBM Guardium	Fortanix, HyTrust	Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform (GCP)	Фінансові установи, медичні заклади, державні організації, великі корпорації	Аппаратні безпекові модулі (HSM), традиційні засоби шифрування даних
Висновки	Присутня конкуренція на міжнародному ринку. Конкуренція висока, кожна з компаній зосереджується на різних аспектах захисту даних	Компанії, що розробляють новітні рішення для захисту даних та шифрування, можуть створити схожий продукт	Постачальниками є великі міжнародні хмарні провайдери, що забезпечують інфраструктуру для зберігання та обробки даних	Більшість клієнтів мають високі вимоги до безпеки даних і шукають надійні рішення для захисту конфіденційної інформації	Традиційні апаратні засоби захисту даних можуть бути використані як альтернативні рішення, але вони менш гнучкі та складніші в інтеграції

За результатами таблиці робимо висновок, що робота на ринку з огляду на ситуацію є можливою, конкуренція на даному етапі є розрідженою, постачальники не зацікавлені у наданні подібних рішень, а існуючі проекти покривають лише частину запропонованого функціоналу.

На основі аналізу конкуренції (таблиця 4.9.) а також із урахуванням характеристик ідеї проекту (таблиця 4.2.), вимог споживачів до товару (таблиця 4.5.) та факторів маркетингового середовища (таблиці 4.6.-4.7.) визначимо та обґрунтуємо перелік факторів конкурентоспроможності (таблиця 4.10.).

Таблиця 4.10.

Обґрунтування факторів конкурентоспроможності

№	Фактор конкурентоспроможності	Обґрунтування (наведення чинників, що роблять фактор для порівняння конкурентних проектів значущим)
1.	Унікальність функціоналу	Надання вичерпного переліку унікальних функціональних можливостей для вирішення основних задач захисту даних у розподілених сховищах, таких як шифрування, двофакторна автентифікація, авторизація та SSO.
2.	Кроссплатформеність	Незалежність від архітектурних характеристик системи; можливість швидко розгорнути підсистему захисту на будь-якій наявній інфраструктурі, враховуючи мінімальні системні вимоги.
3.	Надійність та масштабованість	Продукт є надійним та дозволяє масштабуватись при зростанні обсягів даних і кількості користувачів, що є особливо вигідним для клієнтів, які

		шукають стабільні та безпечні рішення для захисту своїх даних у розподілених сховищах.

За визначеними факторами конкурентоспроможності проведемо аналіз сильних та слабких сторін стартап-проекту.

Таблиця 4.11.

Порівняльний аналіз сильних та слабких сторін стартап-проекту

№	Фактор конкуретноспроможності	Бали 1-20	Рейтинг товарів-конкурентів у порівнянні з продуктом						
			-3	-2	-1	0	+1	+2	+3
1.	Унікальність функціоналу	17		✓					
2.	Кроссплатформеність	15				✓			
3.	Надійність та масштабованість	15					✓		

Фінальним етапом ринкового аналізу можливостей впровадження проекту є складання SWOT-аналізу (матриці аналізу сильних (Strength) та слабких (Weak) сторін, загроз (Troubles) та можливостей (Opportunities) (таблиця 4.12.) на основі виділених ринкових загроз та можливостей, а також сильних та слабких сторін.

Таблиця 4.12.

SWOT- аналіз стартап-проекту

Сильні сторони:	**Слабкі сторони:**
1. Унікальність функціоналу 2. Надійність та масштабованість 3. Сучасні технологічні підходи 4. Орієнтація на міжнародний	1. Наявність вже впроваджених конкуруючих рішень 2. Є потреба у часі для виходу на ринок

ринок 5. Вирішення чітко сформованої клієнтської потреби	3. Потреба у постійному надходженні коштів для підтримки та вдосконалення продукту
Можливості: 1. Партнерство зі світовими мережевими лідерами 2. Приваблення інвестицій 3. Вдосконалення продукту на основі клієнтського досвіду 4. Розширення клієнтської бази	**Загрози:** 1. Швидший або успішніший вихід на ринок нового конкуруючого рішення 2. Відсутність попиту на продукт

На основі SWOT-аналізу розробляються альтернативи ринкової поведінки (перелік заходів) для виведення стартап проекту на ринок та орієнтовний оптимальний час їх ринкової реалізації з огляду на потенційні проекти конкурентів, що можуть бути виведені на ринок (таблиця 4.13.)

Таблиця 4.13.

Альтернативи ринковоговпровадження стартап-проекту

№	Альтернатива ринкової поведінки	Ймовірність отримання ресурсів	Терміни реалізації
1.	Відкриття проекту та розповсюдження в рамках opensource ліцензії	Низька. Є можливість продажу компанії у випадку зацікавленості з боку великих компаній	Одразу після розробки
2.	Розповсюдження проекту за допомогою науково-публічної роботи: конференції, виступи і т.п.	Висока, оскільки публічними заходами можна поширити ідею проекту та завести знайомства з потенційними інвесторами та клієнтами	1 рік

3.	Участь у тендерах і укладення договорів з державними установами	Середня, так як на це впливають економічні, політичні, особисті мотиви і не можна гарантувати абсолютну прозорість та добросовісну конкуренцію за таких умов	0.5– 2 роки

4.4. Розробка ринкової стратегії стартап-проекту

Розроблення ринкової стратегії першим кроком передбачає визначення стратегії охоплення ринку: опис цільових груп потенційних споживачів (табл. 4.14).

Таблиця 4.14.

Вибір цільових груп потенційних споживачів

№	Опис профілю цільової групи потенційних клієнтів	Готовність споживачів сприйняти продукт	Орієнтовний попит в межах цільової групи	Інтенсивність конкуренції в сегменті	Простота входу в сегмент
1.	Провайдери інтернет-послуг	Не готові сприйняти продукт	Низький	Середня	Проста
2.	Приватні компанії	Готові сприйняти продукт	Середній	Висока	Проста
3.	Провайдери мобільних послуг	Не готові сприйняти продукт	Низький	Низька	Складна

4.	Дата-центри	Готові сприйняти продукт	Високий	Середня	Проста
5.	Інформцційні служби	Готові сприйняти продукт	Низька	Низька	Проста
Обрані цільові групи: дата-центри, приватні компанії. Має бути використана стратегія диференційованого маркетингу.					

Згідно з аналізом потенційних комерційних груп споживачів на запропонований продукт, було обрано стратегію диференційованого маркетингу, оскільки компанія працює водночас із кількома сегментами, з розробкою окремих програм для ринкового впливу.

Таблиця 4.15.

Визначення базової стратегії розвитку

№	Обрана альтернатива розвитку проекту	Стратегія охоплення ринку	Ключові конкурентоспроможі позиції відповідно до обраної альтернативи	Базова стратегія розвитку
1.	Стратегія зростання	Вибірковий розподіл	Нарощування функціоналу, внутрішнє розширення, виробництво нових продуктів	Стратегія диференційованого маркетингу

Таблиця 4.16.

Визначення базової стратегії конкурентної поведінки

№	Чи є проект «першопрохідцем» на ринку?	Чи буде компанія шукати нових споживачів, або забирати існуючих у конкурентів?	Чи буде компанія копіювати основні характеристики товару конкурента, і які?	Стратегія конкурентної поведінки
1.	Ні	Комібінований підхід: є ніші, які не використовують подібні продукти та завдяки наявним перевагам можна забирати існуючих клієнтів	Ні, компанія може використовувати наявний функціонал і вдосконалювати його згідно з клієнтським досвідом	Стратегія виклику лідера

На основі вимог споживачів з обраних сегментів до постачальника (стартап-компанії) та вимог до продукту (табл. 4.5.), а також в залежності від обраної базової стратегії розвитку (табл. 4.15) та стратегії конкурентної поведінки (табл. 4.16) розробляється стратегія позиціонування (табл. 4.17). що полягає у формуванні ринкової позиції (комплексу асоціацій), за яким споживачі мають ідентифікувати торгівельну марку/проект.

Таблиця 4.17.

Визначення стратегії позиціонування

№	Вимоги до товару цільової аудиторії	Базова стратегія розвитку	Ключові конкуренто-спроможні позиції стартап-проекту	Вибір асоціацій, які мають сформувати комплексну позицію власного проекту
1.	Вирішення проблеми конструювання трафіка, можливість прогнозування трафіка, надійність, стабільність, підтримка та зворотній зв'язок	Стратегія диференційова-ного маркетингу	Унікальність функціоналу, кросс платформеність, надійність, масштабованість	Унікальність Надійність Клієнто-орієнтованість

У результаті отримали узгоджену систему рішень щодо ринкової поведінки стартап-компанії, що визначатиме напрями роботи стартап-компанії на ринку.

4.5. Розробка маркетингової програми стартап-проекту

Першим кроком є формування маркетингової концепції продукту, який отримає споживач (таблиця 4.18.).

Таблиця 4.18.

Визначення ключових переваг концепції потенційного товару

№	Потреба	Вигода, яку пропонує товар	Ключові переваги перед конкурентами (існуючі або такі, що потрібно створити)
1.	Вирішення проблеми захисту даних у розподілених сховищах	Комплексне рішення для захисту даних з підтримкою шифрування, двофакторної автентифікації та SSO	Використання сучасних технологій та підходів, висока надійність, безпека і масштабованість
2.	Можливість інтеграції з існуючими системами безпеки	Можливість інтеграції підсистеми захисту з існуючими рішеннями	Інтеграція з існуючими системами безпеки, швидкість налаштування та розгортання
3.	Підтримка та зворотній зв’язок	Простота у використанні та налаштуванні, менші витрати часу на підтримку	Дружній інтерфейс, допомога у налаштуванні та підтримці сервісу

Далі розробляється трирівнева маркетингова модель товару: уточнюється ідея продукту та/або послуги, його фізичні складові, особливості процесу його надання.

Таблиця 4.19.

Опис трьох рівнів моделі товару

<table>
<tr><td>Рівні товару</td><td colspan="2">Сутність та складові</td></tr>
<tr><td>I. Товар за задумом</td><td colspan="2">Підсистема захисту розподіленого сховища даних з підтримкою шифрування, двофакторної автентифікації, SSO та можливістю інтеграції з існуючими системами безпеки.</td></tr>
<tr><td rowspan="8">II. Товар у реальному виконанні</td><td>Властивості/характеристики</td><td>Розмір</td></tr>
<tr><td>1. Сервіс шифрування даних</td><td>400MB</td></tr>
<tr><td>2. Сервіс автентифікації та авторизації</td><td>1GB</td></tr>
<tr><td>3. Інтеграція з AWS S3 для зберігання файлів</td><td>1GB</td></tr>
<tr><td>4. Розподілене сховище даних</td><td>1GB</td></tr>
<tr><td colspan="2">Якість: внутрішнє тестування, логування виконання, система підтримки</td></tr>
<tr><td colspan="2">Пакування: SaaS-концепти, створення API та дашбордів для клієнтів.</td></tr>
<tr><td colspan="2">Марка: назва організації-розробника + назва товару</td></tr>
<tr><td rowspan="2">III. Товар із підкріпленням</td><td colspan="2">До продажу: ліцензія на використання</td></tr>
<tr><td colspan="2">Після продажу: надання доступу до API</td></tr>
<tr><td colspan="3">За рахунок чого потенційний товар буде захищено від копіювання: Комплексне поєднання властивостей і характеристик, закладене на другому та третьому рівнях товару.</td></tr>
</table>

Таблиця 4.20.

Визначення меж встановлення ціни

№	Рівень цін на товари-замінники	Рівень цін на товари-аналоги	Рівень доходів цільової групи споживачів	Верхня та нижня межі становлення ціни на товар/послугу
1.	500$ - 1000$/міс.	500$ - 1000$/міс.	>500$/міс.	300-500/міс.

Таблиця 4.21.

Формування системи збуту

№	Специфіка закупівельної поведінки цільових клієнтів	Функції збуту, які має виконувати постачальник товару	Глибина каналу збуту	Оптимальна система збуту
1.	Покупка ліцензії на продукт або договір про надання послуг без передачі ПО до замовника	Надання доступу до API, розгортання системи на архітектурі клієнта	Канал нульового рівня	Інтернет, тендерні торги

Таблиця 4.22.

Концепція маркетингових комунікацій

№	Специфіка поведінки цільових клієнтів	Канали комунікацій, якими користуються цільові клієнти	Ключові позиції, обрані для позиціонування	Завдання рекламного повідомлення	Концепція рекламного звернення

1.	Купують товар на вимогу	Наукові кола, Інтернет, соціальні мережі, форуми	Інтернет-маркетинг	Презентація товару, унікальності і якості.	Пояснення функцій та можливо-стей продукту.

Висновки до розділу 4

Четвертий розділ присвячено розробці стартап-проекту на основі розробленої підсистеми захисту розподіленого сховища даних. Для дослідження в рамках розробки стартап-проекту було виконано наступні завдання:

1. Наведено загальний опис ідеї стартап-проекту, описано функціонал підсистеми захисту, визначено основних конкурентів та порівняно функціонал існуючих продуктів для визначення переваг розроблюваного продукту над конкурентами.
2. Проаналізовано сучасні ринкові можливості для запуску проекту та успішного виходу на ринок. Виявлено можливості та загрози стартап-проекту та сплановано дії щодо їх запобігання.
3. Порівняно перспективи запуску в порівнянні з конкурентами, встановлено план та ринкову стратегію розвитку продукту, визначено цільові групи та способи просування проекту.
4. Розроблено маркетингову програму для просування продукту на ринку, описано способи та основні канали збуту, визначено пріоритетні цільові групи та маркетингові повідомлення для розширення клієнтської бази.

В підсумку, отримано стартап-проект для запуску розробленої підсистеми захисту на базі дипломного проекту для виходу на ринок.

ВИСНОВКИ

Магістерська робота присвячена розробці підсистеми захисту розподіленого сховища даних. У роботі було виконано дослідження сучасного стану методів захисту даних у розподілених сховищах, визначено основні аспекти та вектори розвитку, а також виконано технологічний огляд сучасних підходів до шифрування та автентифікації. Розглянуто головні види загроз та відкриті задачі захисту даних у розподілених сховищах, виконано огляд існуючих рішень. Виявлено проблему, що особливо актуальна для високонавантажених систем: продукується значна кількість нових даних, що потребують надійного захисту, натомість традиційні рішення демонструють занизькі показники безпеки та є слабкомасштабованими. Запропоновано спосіб вирішення даної проблеми із застосуванням сучасних підходів до шифрування, двофакторної автентифікації та єдиного входу, а також розглянуто альтернативні підходи.

Сформовано основні завдання, які має вирішувати підсистема захисту. Сформульовано основні етапи створення та впровадження підсистеми захисту. Визначено процеси формування множин даних, які мають бути захищені, та процеси обробки даних для забезпечення конфіденційності та цілісності. Зазначено аналітичні підходи для опрацювання отриманої статистики та реалізації способу захисту даних. Окрім цього, описано основні етапи конвеєру обробки даних та даних про доступ до системи. Виокремлено та сформовано процеси автентифікації та авторизації користувачів. Розроблено спосіб обробки та зберігання даних, заснований на сучасних концепціях шифрування та розподіленого зберігання даних з використанням AWS S3.

За проведеною розробкою реалізовано модель підсистеми захисту розподіленого сховища даних. Виконано технологічний огляд засобів реалізації, обґрунтовано вибір мови програмування, архітектури та сервісів. Обрано технології, за допомогою яких реалізовано модель системи. Створено процес генерації даних для тестування роботи системи. Виконано опис

структури програмного додатку та реалізації його компонентів. Надано проміжні результати процесів шифрування даних, автентифікації користувачів та управління доступом до даних, описано призначення та методи задач, що реалізуються та виконуються на кожному етапі. Також надано опис візуального інтерфейсу системи обробки даних та показано результуючу модель захисту даних.

Завдяки технологічним та концептуальним особливостям дана підсистема може бути застосована до розподілених сховищ даних як з високою інтенсивністю доступу, так і до збалансованих. Запропоноване рішення є масштабованим, тому з практичної точки зору, підсистема є достатньо універсальною та стабільною для застосування у виробничому середовищі. Окрім цього, методи роботи з великими даними дозволяють забезпечувати високий рівень безпеки в режимі реального часу, що з практичної точки зору дозволяє вирішувати задачі захисту даних більш ефективно.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. H. Garcia-Molina, J. D. Ullman, and J. Widom, "Database Systems: The Complete Book," Prentice Hall, 2008.
2. A. S. Tanenbaum and M. van Steen, "Distributed Systems: Principles and Paradigms," Pearson, 2017.
3. C. Bell and B. W. Darnell, "MySQL High Availability: Tools for Building Robust Data Centers," O'Reilly Media, 2010.
4. M. T. Özsu and P. Valduriez, "Principles of Distributed Database Systems," Springer, 2011.
5. A. Silberschatz, H. F. Korth, and S. Sudarshan, "Database System Concepts," McGraw-Hill, 2010.
6. J. Gray and A. Reuter, "Transaction Processing: Concepts and Techniques," Morgan Kaufmann, 1993.
7. L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," Communications of the ACM, vol. 21, no. 7, 1978.
8. G. Coulouris, J. Dollimore, and T. Kindberg, "Distributed Systems: Concepts and Design," Pearson, 2005.
9. D. Kossmann, "The State of the Art in Distributed Query Processing," ACM Computing Surveys, vol. 32, no. 4, 2000.
10. S. Ceri and G. Pelagatti, "Distributed Databases: Principles and Systems," McGraw-Hill, 1984.
11. J. Hellerstein, "The MADlib Analytics Library: Big Data Machine Learning Made Simple," Proceedings of the VLDB Endowment, vol. 5, no. 12, 2012.
12. R. S. Ramakrishnan and J. Gehrke, "Database Management Systems," McGraw-Hill, 2003.
13. C. Curino et al., "Relational Cloud: A Database-as-a-Service for the Cloud," Proceedings of the 5th Biennial Conference on Innovative Data Systems Research (CIDR), 2011.
14. M. Brantner et al., "Building a Database on S3," Proceedings of the ACM SIGMOD International Conference on Management of Data, 2008.

15. J. F. Naughton et al., "The Niagara Internet Query System," IEEE Data Engineering Bulletin, vol. 24, no. 2, 2001.
16. P. Cudré-Mauroux et al., "A Demonstration of SciDB: A Science-Oriented DBMS," Proceedings of the VLDB Endowment, vol. 2, no. 2, 2009.
17. Y. Chen et al., "The U.S. Department of Energy's Scalable Data Management, Analysis, and Visualization (SDAV) Institute," Computing in Science & Engineering, vol. 15, no. 6, 2013.
18. M. Franklin et al., "Transactional Client-Server Cache Consistency: Alternatives and Performance," ACM Transactions on Database Systems, vol. 22, no. 3, 1997.
19. L. Lamport, "Password Authentication with Insecure Communication," Communications of the ACM, vol. 24, no. 11, 1981. J. F. Naughton et al., "The Niagara Internet Query System," IEEE Data Engineering Bulletin, vol. 24, no. 2, 2001.
20. H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication," RFC 2104, 1997. P. A. Bernstein, "Middleware: A Model for Distributed System Services," Communications of the ACM, vol. 39, no. 2, 1996.
21. D. Balfanz et al., "Two-factor authentication: a security analysis," Communications of the ACM, vol. 48, no. 5, 2005.
22. A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, "Handbook of Applied Cryptography," CRC Press, 1996.
23. W. Stallings, "Cryptography and Network Security: Principles and Practice," Pearson, 2016.
24. B. Schneier, "Applied Cryptography: Protocols, Algorithms, and Source Code in C," Wiley, 1996.
25. N. Ferguson, B. Schneier, and T. Kohno, "Cryptography Engineering: Design Principles and Practical Applications," Wiley, 2010.
26. D. Boneh and M. Franklin, "Identity-Based Encryption from the Weil Pairing," SIAM Journal on Computing, vol. 32, no. 3, 2003.

27. M. Hartl, "Ruby on Rails Tutorial: Learn Web Development with Rails," Addison-Wesley Professional, 2020.
28. D. B. Copeland, "Agile Web Development with Rails 7," Pragmatic Bookshelf, 2021.
29. S. Shapiro, "The Rails 7 Way," Addison-Wesley Professional, 2021.
30. B. Tate and C. Hibbs, "Ruby on Rails: Up and Running," O'Reilly Media, 2021.
31. S. Ruby, D. Thomas, and D. Heinemeier Hansson, "Agile Web Development with Rails 7," Pragmatic Bookshelf, 2021.
32. J. D. Eisenberg, "Ruby on Rails for Dummies," For Dummies, 2020.
33. B. Anderson, "Ruby Performance Optimization," Pragmatic Bookshelf, 2015.
34. S. Gupta, "Learning Rails 6: Rails from the Outside In," O'Reilly Media, 2016.
35. D. Thomas, C. Fowler, and A. Hunt, "Programming Ruby 1.9 & 2.0: The Pragmatic Programmers' Guide," Pragmatic Bookshelf, 2013.
36. D. Flanagan and Y. Matsumoto, "The Ruby Programming Language," O'Reilly Media, 2008.
37. H. Hasegawa, "Ruby on Rails Enterprise Application Development," Packt Publishing, 2013.
38. N. Sutterer, "Rails: Novice to Ninja," SitePoint, 2017.
39. M. Kehoe, "Ruby on Rails: The Complete Guide," CreateSpace Independent Publishing Platform, 2017.
40. A. Biggs, "Rails, Angular, Postgres, and Bootstrap: Powerful, Effective, and Efficient Full-Stack Web Development," Pragmatic Bookshelf, 2017.
41. R. Olsen, "Eloquent Ruby," Addison-Wesley Professional, 2011.
42. B. Ramsay, "Ruby on Rails in Action," Manning Publications, 2015.
43. A. Rajaraman and J. Ullman, "Mining of Massive Datasets," Cambridge University Press, 2011.

ДОДАТОК А

ЛІСТИНГ ПРОГРАМНГО КОДУ

Аркушів 15

Київ – 2024

```
class User < ApplicationRecord
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :validatable,
         :omniauthable, omniauth_providers: %i[google_oauth2
facebook]

  include ActiveModel::OneTimePassword

  has_many :posts, dependent: :destroy
  has_many :user_logs

  has_one_time_password column_name: :otp_secret_key

  before_save :set_otp_secret, :set_otp_required_for_login

  attr_accessor :otp_attempt

  enum role: { user: 0, admin: 1 }

  validates :email,
            uniqueness: true,
            presence: true,
            length: { maximum: 50 },
            format: { without: /http/, message: "urls are not
allowed"}

  def log_action(action)
    self.user_logs.create(log_message: action)
  end

  def self.from_omniauth(auth)
    user = User.find_or_create_by(provider: auth.provider, uid:
auth.uid) do |user|
      user.email = auth.info.email
      user.password = Devise.friendly_token[0, 20]
    end

    user.update(
      provider: auth.provider,
      uid: auth.uid,
    ) if user.persisted?

    user
  end

  def send_two_factor_authentication_code
    UserMailer.send_otp(self).deliver_now
  end

  private

  def set_otp_secret
    self.otp_secret_key ||= ROTP::Base32.random_base32
  end
```

```
  def set_otp_required_for_login
    self.otp_required_for_login = true if
self.otp_required_for_login.nil?
  end
end

class UserLog < ApplicationRecord
  belongs_to :user

  before_save :encrypt_description, :set_log_message
  after_find :decrypt_description

  validates :action, presence: true
  validates :log_message, presence: true

  def encrypted_data
    {
      description_encrypted: self.description_encrypted,
      ivs: self.ivs || [],
      auth_tags: self.auth_tags || [],
      digests: self.digests || [],
      key_version: self.key_version
    }
  end

  private

  def encrypt_description
    return if log_message.blank?
    encrypted =
Helpers::Encryption::Encryptor.encrypt(log_message)
    self.description_encrypted = encrypted[:data]
    self.ivs = encrypted[:ivs]
    self.auth_tags = encrypted[:auth_tags]
    self.digests = encrypted[:digests]
    self.key_version = encrypted[:key_version]
    self.log_message = nil
  end

  def decrypt_description
    return if description_encrypted.blank?
    encrypted_data = {
      data: description_encrypted,
      ivs: ivs,
      auth_tags: auth_tags,
      digests: digests,
      key_version: key_version
    }
    self.log_message =
Helpers::Encryption::Encryptor.decrypt(encrypted_data)
  end

  def set_log_message
```

```
    self.log_message ||= "Default log message" if
log_message.blank?
  end
end

class Users::OmniauthCallbacksController <
Devise::OmniauthCallbacksController
  skip_before_action :verify_authenticity_token, only:
[:google_oauth2]

  def google_oauth2
    handle_auth "Google"
  end

  def handle_auth(kind)
    @user = User.from_omniauth(request.env['omniauth.auth'])

    if @user.persisted?
      sign_in_and_redirect @user, event: :authentication
      set_flash_message(:notice, :success, kind: kind) if
is_navigational_format?
    else
      session["devise.#{kind.downcase}_data"] =
request.env['omniauth.auth'].except("extra")
      redirect_to new_user_registration_url, alert:
@user.errors.full_messages.join("\n")
    end
  end
end

class Users::OtpController < ApplicationController
  before_action :authenticate_user!

  def new
  end

  def create
    if
current_user.validate_and_consume_otp!(params[:otp_attempt])
      sign_in(current_user)
      redirect_to root_path, notice: 'Successfully
authenticated'
    else
      flash[:alert] = 'Invalid OTP code'
      render :new
    end
  end

  def send_otp
    current_user.update(otp_secret: User.generate_otp_secret)
    UserMailer.send_otp(current_user).deliver_now
    redirect_to users_otp_confirmation_path, notice: 'OTP code
sent to your email'
  end
```

```
end

class Users::SessionsController < Devise::SessionsController
  prepend_before_action :verify_recaptcha_method, only:
[:create]

  def create
    if @captcha_verified
      self.resource = warden.authenticate!(auth_options)
      if resource.otp_required_for_login
        session[:otp_user_id] = resource.id
        resource.send_two_factor_authentication_code
        sign_out(resource)
        redirect_to user_verify_otp_path and return
      else
        sign_in(resource_name, resource)
        log_user_action(resource, 'login', 'User logged in') if
resource.persisted?
        respond_with resource, location:
after_sign_in_path_for(resource)
      end
    else
      flash[:alert] = 'Invalid email, password, or reCAPTCHA.
Please try again.'
      redirect_to new_user_session_path and return
    end
  rescue ::ActiveRecord::RecordNotFound
    flash[:alert] = "Email is not registered"
    redirect_to new_user_session_path
  rescue ::Devise::Strategies::Authenticatable::Invalid
    flash[:alert] = 'Invalid email or password'
    redirect_to new_user_session_path and return
  end

  def verify_otp
    user = User.find(session[:otp_user_id])
    if user.authenticate_otp(params[:otp_attempt], drift: 60)
      session[:otp_user_id] = nil
      sign_in user
      redirect_to after_sign_in_path_for(user)
    else
      flash[:alert] = 'Invalid OTP'
      log_user_action(user, 'otp attempt', 'User entered wrong
otp code')
      render 'verify_otp'
    end
  end

  protected

  def after_sign_in_path_for(resource)
    super(resource)
  end
```

```
  private

  def verify_recaptcha_method
    @captcha_verified = verify_recaptcha
  end

  def auth_options
    { scope: resource_name, recall: "#{controller_path}#new" }
  end
end

require 'openssl'
require 'base64'

module Helpers
  module Encryption
    module Encryptor
      ALGORITHM = 'aes-256-gcm'
      IV_LENGTH = 12

      def self.encrypt(plain_text)
        layers = Helpers::Encryption::KeyManagment.keys.size
        encrypted_data = plain_text
        ivs = []
        auth_tags = []
        digests = []

        layers.times do |i|
          cipher = OpenSSL::Cipher.new(ALGORITHM)
          cipher.encrypt
          iv = cipher.random_iv[0, IV_LENGTH]
          raise "IV must be 12 bytes" unless iv.bytesize ==
IV_LENGTH
          cipher.key = Helpers::Encryption::KeyManagment.key(i)
          encrypted = cipher.update(encrypted_data) +
cipher.final
          auth_tag = cipher.auth_tag
          digest = OpenSSL::Digest::SHA256.digest(encrypted)

          ivs << Base64.strict_encode64(iv)
          auth_tags << Base64.strict_encode64(auth_tag)
          digests << Base64.strict_encode64(digest)
          encrypted_data = Base64.strict_encode64(encrypted)
        end

        {
          data: encrypted_data,
          ivs: ivs,
          auth_tags: auth_tags,
          digests: digests,
          key_version: layers - 1
        }
      end
```

```
      def self.decrypt(encrypted_data)
        layers = encrypted_data[:key_version] + 1
        decrypted_data =
Base64.strict_decode64(encrypted_data[:data])

        (layers - 1).downto(0) do |i|
          decipher = OpenSSL::Cipher.new(ALGORITHM)
          decipher.decrypt
          iv = Base64.strict_decode64(encrypted_data[:ivs][i])
          raise "IV must be 12 bytes" unless iv.bytesize ==
IV_LENGTH
          decipher.iv = iv
          decipher.key =
Helpers::Encryption::KeyManagment.key(i)
          decipher.auth_tag =
Base64.strict_decode64(encrypted_data[:auth_tags][i])

          decrypted = decipher.update(decrypted_data) +
decipher.final
          digest = OpenSSL::Digest::SHA256.digest(decrypted)

          raise "Data integrity check failed" unless digest ==
Base64.strict_decode64(encrypted_data[:digests][i])

          decrypted_data = decrypted
        end

        decrypted_data
      rescue => e
        puts "Decryption failed: #{e.message}"
      end
    end
  end
end

module Helpers
  module Encryption
    module KeyManagment
      ALGORITHM = 'aes-256-gcm'

      def self.keys
        @keys ||= [
          Rails.application.credentials.dig(:encryption_keys,
:primary),
          Rails.application.credentials.dig(:encryption_keys,
:secondary),
          Rails.application.credentials.dig(:encryption_keys,
:tertiary)
        ].compact
      end

      def self.key(index)
        [keys[index]].pack('H*')
      end
```

```
    end
  end
end

module UserOtp
  extend self

  def user_otp_token_form_params(params)
    params.dig(:user, :otp, :token).to_s.strip
  end

  def user_enabled?(user)
    user.otp_secret_key.present?
  end

  def otp_valid_regex?(otp_token)
    otp_token.present && otp_token.match?(/^[0-9]+$/)
  end

  def otp_valid?(user, otp_token)
    user.authenticate_otp(otp_token, drift: 30)
  end
end

class PostPolicy < ApplicationPolicy
  class Scope < Scope
    def resolve
      if user.admin?
        scope.all
      else
        scope.where(user: user)
      end
    end
  end

  def index?
    user.admin?
  end

  def show?
    user.admin?
  end

  def create?
    user.admin?
  end

  def update?
    user.admin?
  end

  def destroy?
    user.admin?
  end
```

```
end

<div class="container mt-5">
  <div class="row justify-content-center">
    <div class="col-md-4">
      <h2 class="text-center">Log in</h2>

      <%= form_for(resource, as: resource_name, url:
session_path(resource_name)) do |f| %>
        <div class="form-group">
          <%= f.label :email, class: 'form-label' %>
          <%= f.email_field :email, autofocus: true, id: 'email-
input', class: 'form-control' %>
        </div>

        <div class="form-group">
          <%= f.label :password, class: 'form-label' %>
          <%= f.password_field :password, autocomplete:
"current-password", class: 'form-control' %>
        </div>

        <% if devise_mapping.rememberable? %>
          <div class="form-group form-check">
            <%= f.check_box :remember_me, class: 'form-check-
input' %>
            <%= f.label :remember_me, class: 'form-check-label'
%>
          </div>
        <% end %>

        <div class="form-group mt-3">
          <%= recaptcha_tags %>
        </div>

        <div class="actions mt-3">
          <%= f.submit "Log in", class: 'btn btn-primary' %>
        </div>
      <% end %>

      <div class="text-center mt-4">
        <p>Or log in with:</p>
        <%= link_to "Sign in with Google",
user_google_oauth2_omniauth_authorize_path, method: :post,
class: "btn btn-outline-danger mb-2" %><br>
      </div>

      <%= render "devise/shared/links" %>
    </div>
  </div>
</div>

<div class="container">
  <div class="row justify-content-center">
    <div class="col-md-6">
```

```
      <h2>Profile</h2>

      <div class="row">
        <h3 class="mb-4">Create a New Post</h3>
        <%= form_with model: @post, url: posts_path, local: true
do |f| %>
          <div class="form-group mb-3">
            <%= f.label :title, class: 'form-label' %>
            <%= f.text_area :title, class: 'form-control' %>
          </div>
          <div class="form-group mb-3">
            <%= f.label :body, class: 'form-label' %>
            <%= f.text_area :body, class: 'form-control' %>
          </div>
          <div class="form-group mb-3">
            <%= f.label :file, class: 'form-label' %>
            <%= f.file_field :file, class: 'form-control' %>
          </div>
          <div class="actions">
            <%= f.submit 'Create Post', class: 'btn btn-primary
mt-3' %>
          </div>
        <% end %>
      </div>

      <div class="row">
        <h3>Your Posts</h3>
        <% @posts.each do |post| %>
          <div class="card mb-3">
            <div class="card-body">
              <p><%= post.body %></p>

              <% if post.file.present? %>
                <div class="download-link">
                  <%= link_to "Download file", post.file.url,
class: 'btn btn-primary mt-2' %>
                </div>
              <% else %>
                <div class="no-file-alert">
                  <p class="text-warning">No files attached</p>
                </div>
              <% end %>

              <div class="post-meta">
                <small class="text-muted">Created at: <%=
post.created_at %></small>
              </div>
            </div>
          </div>
        <% end %>
      </div>
    </div>
  </div>
</div>
```

```
<!DOCTYPE html>
<html>
  <head>
    <title>Distributed App</title>
    <meta name="viewport" content="width=device-width,initial-
scale=1">
    <%= csrf_meta_tags %>
    <%= csp_meta_tag %>

    <%= stylesheet_pack_tag 'application', media: 'all', 'data-
turbolinks-track': 'reload' %>
    <%= javascript_pack_tag 'application', 'data-turbolinks-
track': 'reload' %>
  </head>
  <body>
    <nav class="navbar navbar-expand-lg navbar-light bg-light
mb-4">
      <div class="container-fluid">
        <a class="navbar-brand" href="<%= root_path
%>">Distributed App</a>
        <button class="navbar-toggler" type="button" data-bs-
toggle="collapse" data-bs-target="#navbarNav" aria-
controls="navbarNav" aria-expanded="false" aria-label="Toggle
navigation">
          <span class="navbar-toggler-icon"></span>
        </button>
        <div class="collapse navbar-collapse" id="navbarNav">
          <ul class="navbar-nav me-auto"></ul>
          <ul class="navbar-nav ms-auto">
            <% if user_signed_in? %>
              <% if current_user.admin? %>
                <li class="nav-item">
                  <%= link_to 'Manage Users', admin_users_path,
class: 'nav-link' %>
                </li>
                <li class="nav-item">
                  <%= link_to 'Manage Posts', admin_posts_path,
class: 'nav-link' %>
                </li>
              <% end %>
              <li class="nav-item">
                <%= link_to 'Profile', profile_path, class:
'nav-link' %>
              </li>
              <li class="nav-item">
                <%= link_to 'Logout', destroy_user_session_path,
method: :delete, class: 'nav-link' %>
              </li>
            <% else %>
              <li class="nav-item">
                <%= link_to 'Login', new_user_session_path,
class: 'nav-link' %>
              </li>
              <li class="nav-item">
```

```
                <%= link_to 'Sign Up',
new_user_registration_path, class: 'nav-link' %>
              </li>
            <% end %>
          </ul>
        </div>
      </div>
    </nav>

    <% if flash[:notice] %>
      <div class="alert alert-success"><%= flash[:notice]
%></div>
    <% end %>
    <% if flash[:alert] %>
      <div class="alert alert-danger"><%= flash[:alert] %></div>
    <% end %>

    <div class="container">
      <%= yield %>
    </div>
  </body>
</html>

<div class="container mt-5">
  <div class="row justify-content-center">
    <div class="col-md-6">
      <div class="card">
        <div class="card-header text-center">
          <h2>Enter Your OTP Code</h2>
        </div>
        <div class="card-body">
          <%= form_with url: user_verify_otp_path, method:
:post, local: true do |form| %>
            <div class="form-group">
              <%= form.label :otp_attempt, class: 'form-label'
%>
              <%= form.text_field :otp_attempt, class: 'form-
control', autocomplete: "one-time-code" %>
            </div>
            <div class="text-center mt-4">
              <%= form.submit "Verify OTP", class: 'btn btn-
primary' %>
            </div>
          <% end %>
        </div>
      </div>
    </div>
  </div>
</div>

<h2>Edit <%= resource_name.to_s.humanize %></h2>
```

```
<%= form_for(resource, as: resource_name, url:
registration_path(resource_name), html: { method: :put }) do |f|
%>
  <%= render "devise/shared/error_messages", resource: resource
%>

  <div class="field">
    <%= f.label :email %><br />
    <%= f.email_field :email, autofocus: true, autocomplete:
"email" %>
  </div>

  <% if devise_mapping.confirmable? &&
resource.pending_reconfirmation? %>
    <div>Currently waiting confirmation for: <%=
resource.unconfirmed_email %></div>
  <% end %>

  <div class="field">
    <%= f.label :password %> <i>(leave blank if you don't want
to change it)</i><br />
    <%= f.password_field :password, autocomplete: "new-password"
%>
    <% if @minimum_password_length %>
      <br />
      <em><%= @minimum_password_length %> characters
minimum</em>
    <% end %>
  </div>

  <div class="field">
    <%= f.label :password_confirmation %><br />
    <%= f.password_field :password_confirmation, autocomplete:
"new-password" %>
  </div>

  <div class="field">
    <%= f.label :current_password %> <i>(we need your current
password to confirm your changes)</i><br />
    <%= f.password_field :current_password, autocomplete:
"current-password" %>
  </div>

  <div class="actions">
    <%= f.submit "Update" %>
  </div>
<% end %>

<h3>Cancel my account</h3>

<div>Unhappy? <%= button_to "Cancel my account",
registration_path(resource_name), data: { confirm: "Are you
sure?", turbo_confirm: "Are you sure?" }, method: :delete
%></div>
```

```
<%= link_to "Back", :back %>

<div class="container mt-5">
  <div class="row justify-content-center">
    <div class="col-md-4">
      <h2 class="text-center">Sign up</h2>

      <%= form_for(resource, as: resource_name, url:
registration_path(resource_name)) do |f| %>
        <%= render "devise/shared/error_messages", resource:
resource %>

        <div class="form-group">
          <%= f.label :email, class: 'form-label' %>
          <%= f.email_field :email, autofocus: true, class:
'form-control' %>
        </div>

        <div class="form-group">
          <%= f.label :password, class: 'form-label' %>
          <% if @minimum_password_length %>
          <em>(<%= @minimum_password_length %> characters
minimum)</em>
          <% end %><br />
          <%= f.password_field :password, autocomplete: "new-
password", class: 'form-control' %>
        </div>

        <div class="form-group">
          <%= f.label :password_confirmation, class: 'form-
label' %>
          <%= f.password_field :password_confirmation,
autocomplete: "new-password", class: 'form-control' %>
        </div>

        <div class="form-group mt-3">
          <%= recaptcha_tags %>
        </div>

        <div class="actions mt-3">
          <%= f.submit "Sign up", class: 'btn btn-primary' %>
        </div>

        <div class="text-center mt-4">
          <p>Or sign up with:</p>
          <%= link_to "Sign in with Google",
user_google_oauth2_omniauth_authorize_path, method: :post,
class: "btn btn-outline-danger mb-2" %><br>
        </div>
      <% end %>

      <%= render "devise/shared/links" %>
    </div>
```

```
  </div>
</div>

<div class="container mt-5">
  <div class="row justify-content-center">
    <div class="col-md-6">
      <div class="card">
        <div class="card-header text-center">
          <h2>Forgot your password?</h2>
        </div>
        <div class="card-body">
          <%= form_for(resource, as: resource_name, url:
password_path(resource_name), html: { method: :post }) do |f| %>
            <%= render "devise/shared/error_messages", resource:
resource %>

            <div class="form-group">
              <%= f.label :email, class: 'form-label' %>
              <%= f.email_field :email, autofocus: true,
autocomplete: "email", class: 'form-control' %>
            </div>

            <div class="form-group mt-3">
              <%= f.submit "Send me reset password
instructions", class: 'btn btn-primary btn-block' %>
            </div>
          <% end %>

          <%= render "devise/shared/links" %>
        </div>
      </div>
    </div>
  </div>
</div>

<div class="container mt-5">
  <div class="row justify-content-center">
    <div class="col-md-6">
      <div class="card">
        <div class="card-header text-center">
          <h2>Change your password</h2>
        </div>
        <div class="card-body">
          <%= form_for(resource, as: resource_name, url:
password_path(resource_name), html: { method: :put }) do |f| %>
            <%= render "devise/shared/error_messages", resource:
resource %>
            <%= f.hidden_field :reset_password_token %>

            <div class="form-group">
              <%= f.label :password, "New password", class:
'form-label' %>
              <% if @minimum_password_length %>
```

```
                <em>(<%= @minimum_password_length %> characters
minimum)</em><br />
              <% end %>
              <%= f.password_field :password, autofocus: true,
autocomplete: "new-password", class: 'form-control' %>
            </div>

            <div class="form-group">
              <%= f.label :password_confirmation, "Confirm new
password", class: 'form-label' %>
              <%= f.password_field :password_confirmation,
autocomplete: "new-password", class: 'form-control' %>
            </div>

            <div class="form-group mt-3">
              <%= f.submit "Change my password", class: 'btn
btn-primary btn-block' %>
            </div>
          <% end %>

          <%= render "devise/shared/links" %>
        </div>
      </div>
    </div>
  </div>
</div>
```