МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Сумський державний університет

Факультет електроніки та інформаційних технологій

Кафедра комп'ютерних наук

«До захисту допущено»

В.о. завідувача кафедри

Оксана ШОВКОПЛЯС

(підпис)

04 грудня 2024 р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня магістр

зі спеціальності 122 «Комп'ютерні науки»

освітньо-професійної програми «Інформатика»

на тему: «Інформаційна технологія проєктування хмарної платформи для забезпечення захищеного зберігання та доступу до чутливої інформації у моделі програмного забезпечення як послуга»

здобувача групи ІН.м-32 Безверхого Миколи Івановича

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

Микола БЕЗВЕРХИЙ

(підпис)

Керівник,

асистент кафедри комп'ютерних наук,

кандидат фізико-математичних наук Олександр ВЛАСЕНКО

(підпис)

Суми – 2024

Сумський державний університет
Факультет електроніки та інформаційних технологій
Кафедра комп'ютерних наук

«Затверджую»
В.о. завідувача кафедри
_____ Оксана ШОВКОПЛЯС
(підпис)

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня магістра

зі спеціальності 122 «Комп'ютерні науки», освітньо-професійної програми «Інформатика»
здобувача групи ІН.м-32 Безверхого Миколи Івановича

1. Тема роботи: «Інформаційна технологія проєктування хмарної платформи для забезпечення захищеного зберігання та доступу до чутливої інформації у моделі програмного забезпечення як послуга» затверджена наказом по СумДУ від *«03» грудня 2024 року № 1257-VI*
2. Термін здачі здобувачем кваліфікаційної роботи *до 06 грудня 2024 року*
3. Вхідні дані до кваліфікаційної роботи _____
4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити) *1) Аналіз проблеми предметної області, постановка й формування завдань дослідження. 2) Огляд сучасних технологій та підходів, що використовуються для проєктування веб-орієнтованих інформаційних систем. 3) Програмна реалізація хмарної платформи для зберігання чутливої інформації. 4) Аналіз результатів.*
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) _____
6. Консультанти до проєкту (роботи), із значенням розділів проєкту, що стосується їх

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання «18» серпня 2024 р.

Завдання прийняв до виконання _____ (підпис) Керівник _____ (підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів кваліфікаційної роботи	Термін виконання	Примітка
1	*Аналіз проблеми предметної області, постановка й формування завдань дослідження*		
2	*Огляд сучасних технологій та підходів, що використовуються для проєктування веб-орієнтованих інформаційних систем.*		
3	*Програмна реалізація хмарної платформи для зберігання чутливої інформації.*		
4	*Аналіз отриманих результатів*		
5	*Оформлення пояснювальної записки до кваліфікаційної роботи*		

Здобувач вищої освіти _____ (підпис) Керівник _____ (підпис)

АНОТАЦІЯ

Записка: 81 стор., 19 рис., 3 табл., 1 додаток, 24 використаних джерел.

Обґрунтування актуальності теми роботи – Тема кваліфікаційної роботи є актуальною, оскільки у сучасну епоху стрімкого розвитку цифрових технологій питання безпечного зберігання та доступу до конфіденційної інформації стає дедалі більше важливим. Використання хмарних сервісів у моделі програмного забезпечення як послуги відкриває широкі можливості для створення надійної та масштабованої системи.

Об’єкт дослідження –хмарна платформа для забезпечення захищеного зберігання та доступу до чутливої інформації у моделі програмного забезпечення як послуга.

Мета роботи – проєктування та розробка хмарної платформи для забезпечення захищеного зберігання та доступу до чутливої інформації.

Методи дослідження – інструменти для розробки хмарних веб-платформ, інструменти для побудови веб-клієнтів, інструменти для безпечного керування та зберігання даних.

Результати – спроєктовано та реалізовано хмарну платформу для забезпечення захищеного зберігання та доступу до чутливої інформації. Платформа надає можливість безпечно зберігати чутливу інформацію, керувати клієнтами, керувати користувачами, керувати ключами авторизації. Також платформа відкрита до інтеграції зі сторонніми сервісами.

SECRETS MANAGER, SAAS, SECURITY, NODE JS, EXPRESS, POSTGRE SQL, REACT, SPA, CLOUD PLATFORM

ЗМІСТ

ВСТУП 5

1. ІНФОРМАЦІЙНИЙ ОГЛЯД 7

1.1. Загальний огляд моделі «Програма як послуга» 7

1.2. Поширені проблеми зберігання та доступу до конфіденційної інформації в SaaS 8

1.3. Ключові компоненти безпеки хмарної платформи для SaaS 9

1.4. Забезпечення масштабованості та високої продуктивності 11

1.5. Основні принципи будови веб-застосунку 12

1.5.1. Архітектурні рішення для серверної частини 13

1.5.2. Архітектурні рішення для клієнтської частини 15

1.6. Постановка задачі 16

2. ВИБІР МЕТОДІВ РОЗВ'ЯЗАННЯ ПОСТАВЛЕНОЇ ЗАДАЧІ 18

2.1. Серверна частина 18

2.1.1. Фреймворк для серверної частини 19

2.2. Клієнтська частина 21

2.2.1. Фреймворк для клієнтської частини 22

2.3. Система керування даними 24

3. ПРОГРАМНА РЕАЛІЗАЦІЯ 27

3.1. Проєктування системи 27

3.2. Серверна складова 30

3.3. Фронтенд складова 33

3.4. Тестування системи 36

ВИСНОВКИ 41

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 42

ДОДАТОК А 45

ВСТУП

Обґрунтування вибору теми роботи. У сучасну епоху стрімкого розвитку цифрових технологій питання безпечного зберігання та доступу до конфіденційної інформації стає дедалі більше важливим. Використання хмарних сервісів у моделі програмного забезпечення як послуги відкриває широкі можливості для створення надійної та масштабованої системи для вирішення цього завдання.

Актуальність. У сучасному цифровому світі, де обсяги конфіденційної інформації, яка потребує безпечного зберігання та обробки, швидко зростають, попит на надійний захист даних продовжує збільшуватися. Хмарні технології, особливо в моделі програмного забезпечення як послуги (SaaS), пропонують значні можливості для ефективного управління даними, проте безпека залишається критичним викликом. Запропонована система спрямована на забезпечення безпечного зберігання секретних даних із можливістю їх контрольованого доступу та передачі. Завдяки інтеграції сучасних методів шифрування та багаторівневій автентифікації користувачі зможуть зберігати конфіденційну інформацію без ризику витоку. Система буде пристосованою до вимог безпеки, що дозволить використовувати її для передавання важливих даних, забезпечуючи стабільний доступ без втрати цілісності. Розробка такого рішення є актуальною, оскільки воно сприяє підвищенню ефективності та надійності роботи сучасних програмних платформ, відповідаючи на зростаючі виклики кібербезпеки та забезпечуючи комфортну інтеграцію у різноманітні середовища.

Об’єкт дослідження. Процес проєктування хмарної платформи для забезпечення захищеного зберігання та доступу до чутливої інформації.

Предмет дослідження. Технології проєктування хмарних платформ, системи управління доступом до чутливої інформації.

Новизна. Відмінною рисою розробленої хмарної платформи є інтеграція сучасних методів шифрування та багаторівневого контролю доступу, які

забезпечують високий рівень захищеності чутливої інформації. Особливістю системи є адаптивний підхід до управління даними, що дозволяє динамічно налаштовувати доступ залежно від типу користувачів та контексту використання. Завдяки цьому платформа забезпечує безпечну передачу чутливих даних між програмами, зберігаючи цілісність і конфіденційність інформації.

Структура. Дана робота складається зі вступу, інформаційного огляду, постановки задачі, вибору методів для розв’язання поставленої задачі, програмної реалізації, висновків, списку використаних джерел та додатків.

1. ІНФОРМАЦІЙНИЙ ОГЛЯД

1.1. Загальний огляд моделі «Програма як послуга»

Модель «Програма як послуга» (Software as a Service, SaaS) є сучасним підходом до надання послуг програм через інтернет. На відміну від традиційних методів, коли користувачі купують ліцензії та встановлюють програми на свої пристрої, SaaS дозволяє отримувати доступ до програм через веб-браузер без необхідності встановлення чи обслуговування. Такий підхід є вкрай привабливим, оскільки він звільняє бізнеси та користувачів від витрат на дороге обладнання чи управління складною IT-інфраструктурою. Натомість за усі аспекти, пов’язані з хостингом, оновленнями та безпекою, відповідає постачальник SaaS.

Головною особливістю SaaS є використання хмарних технологій. Рішення SaaS розміщуються на розподілених серверах, що забезпечує високу доступність і масштабованість. Така архітектура дозволяє бізнесу динамічно регулювати обсяги використання, збільшуючи чи зменшуючи ресурси в залежності від потреб, без значних капіталовкладень. Наприклад, малий бізнес може почати з базового тарифного плану і поступово розширювати використання мірою підвищення потреб, сплачуючи лише за необхідні послуги. Така гнучкість робить SaaS привабливим вибором для компаній будь-якого розміру.

Ще однією перевагою SaaS є доступність. Користувачі можуть працювати із застосунками SaaS з будь-якого місця, де є доступ до інтернету, та з будь-якого пристрою з браузером. Це є особливо цінним у сучасному світі, де багато хто працює віддалено. SaaS також підтримує оновлення та спільну роботу в реальному часі, що дозволяє командам ефективніше співпрацювати і швидко ділитися інформацією.

Вкрай важливими аспектами моделі SaaS є безпека та обслуговування. Завдяки централізованому хостингу постачальники можуть забезпечувати

високий рівень захисту, наприклад, через шифрування, двофакторну автентифікацію та регулярні перевірки безпеки. Вони також відповідають за своєчасне оновлення програм, що зменшує ризики, пов’язані з використанням застарілих версій. Це особливо важливо для компаній, які працюють з чутливими даними, такими як фінансова інформація чи записи клієнтів.

1.2.Поширені проблеми зберігання та доступу до конфіденційної інформації в SaaS

Зберігання та доступ до конфіденційної інформації в межах платформ SaaS включають ряд аспектів, які організації повинні враховувати для забезпечення безпеки.

Однією з головних проблем є ризик витоку даних. Платформи SaaS є привабливими цілями для кіберзлочинців, оскільки вони зберігають великі обсяги чутливої інформації. Витоки можуть статися через слабке шифрування, неправильно налаштовані параметри безпеки або вразливості в інфраструктурі платформи. Якщо зловмисники отримають несанкціонований доступ, вони можуть використати дані для крадіжки особистої інформації, фінансових махінацій або корпоративного шпигунства, що завдасть серйозної шкоди як бізнесу, так і його клієнтам.

Ще однією поширеною проблемою є втрата даних, яка може статися через випадкове видалення, збої системи або зловмисні атаки, наприклад, із використанням програм-вимагачів. Оскільки платформи SaaS зазвичай працюють у спільному середовищі, збій в одній частині системи може вплинути на доступ до важливих даних для багатьох клієнтів. Організації, які використовують рішення SaaS, не завжди можуть мати надійні системи резервного копіювання, що залишає їх вразливими в разі відсутності у провайдера належного плану відновлення після збоїв.

Правові вимоги щодо захисту даних – наступний вкрай важливий аспект, особливо для організацій, які працюють у різних регіонах з різними

юридичними вимогами. Платформи SaaS часто зберігають дані на хмарних серверах, розташованих у різних країнах. Це може викликати проблеми з дотриманням законодавства, якщо платформа не відповідає законам про захист даних у відповідних країнах. Наприклад, такі регламенти, як Загальний регламент із захисту даних (GDPR) в Європейському Союзі, висувають суворі вимоги до збору, зберігання та передачі персональних даних. Недотримання цих стандартів може призвести до значних штрафів і юридичних наслідків.

Ще однією складністю є обмежений контроль організацій над своїми даними в середовищі SaaS. Передаючи управління інформацією сторонньому постачальнику, компанії змушені покладатися на його практики безпеки, які можуть не завжди відповідати їхнім власним політикам. Такий брак прямого контролю створює невизначеність щодо того, як дані обробляються, зберігаються та хто має до них доступ. До того ж перебої в роботі сервісу, такі як простої чи збої, можуть тимчасово заблокувати доступ до важливої інформації, що може порушити роботу бізнесу.

Останнім аспектом є внутрішні загрози та помилки користувачів, які також створюють значні ризики для безпеки конфіденційної інформації в SaaS. Співробітники або підрядники, які мають доступ до чутливих даних, можуть зловживати ними або випадково розкрити через фішингові атаки чи нехтування кібербезпекою. Навіть з використанням передових технологій людський фактор залишається однією з найпоширеніших причин інцидентів, пов’язаних із безпекою даних.

1.3.Ключові компоненти безпеки хмарної платформи для SaaS

Безпечна хмарна платформа для SaaS побудована на комбінації ключових компонентів, які працюють разом для захисту чутливої інформації та забезпечення надійного сервісу. Основа цієї платформи – це архітектура, яка визначає рівень безпеки. Типова хмарна архітектура SaaS розроблена так, щоб підтримувати багатокористувацькість, дозволяючи кільком клієнтам

ділити одну інфраструктуру, при цьому забезпечуючи ізоляцію їхніх даних. Ця ізоляція досягається за допомогою різних технік, таких як використання окремих віртуальних машин, контейнерів або виділених баз даних для кожного клієнта. Завдяки такому підходу платформа запобігає несанкціонованому доступу до даних між користувачами, навіть якщо інфраструктура є спільною.

На цій архітектурній основі шифрування відіграє важливу роль у захисті даних протягом усього їх життєвого циклу. Для даних, що зберігаються на платформі, та для даних, що передаються між користувачами та хмарою, шифрування гарантує, що інформація залишатиметься в безпеці. Дані перетворюються в незрозумілий формат за допомогою шифрування, який можна повернути до оригінального вигляду тільки за допомогою правильного ключа для розшифрування. Це означає, що навіть якщо дані будуть перехоплені під час передачі і стануть доступними несанкціонованим користувачам, вони не зможуть бути використаними зловмисниками, тим самим захищаючи цілісність інформації.

В доповнення механізмів шифрування та ізоляції даних, управління доступом та ідентифікацією (IAM) є критичним для того, щоб регулювати, хто може отримати доступ до платформи і що саме вони можуть робити після входу. Системи IAM працюють шляхом перевірки автентифікації, підтвердження особи користувачів та застосування авторизації, яка визначає дії та ресурси, до яких користувачі можуть мати доступ. За допомогою контролю доступу на основі ролей (RBAC) організації можуть призначати різні рівні дозволів в залежності від ролі користувача, забезпечуючи, щоб доступ до чутливих даних чи важливих функцій мали тільки ті, кому насправді потрібен цей доступ. Такий підхід до управління користувачами мінімізує ризик надмірних прав доступу, що могло б призвести до витоку інформації.

Важливим аспектом є мережна безпека. Вона додає ще один рівень захисту платформи. Брандмауери блокують несанкціонований трафік ще до того, як він досягне інфраструктури платформи. Крім того, віртуальні приватні

мережі (VPN) створюють безпечні, зашифровані з’єднання для віддалених користувачів, гарантуючи, що всі комунікації між користувачами та платформою залишаються приватними. Безпечні API також є необхідними, оскільки вони дозволяють різним додаткам взаємодіяти з платформою. Ці API повинні бути гарно захищеними, щоб запобігти несанкціонованому доступу та забезпечити, щоб лише затверджені додатки могли робити запити до системи, що додає ще один рівень захисту від вразливостей.

Зрештою, жодна платформа не може вважатися повністю безпечною без надійних стратегій резервного копіювання та відновлення. Ці механізми гарантують, що в разі збою системи, кібернападу чи будь-якої іншої проблеми, важливі дані залишаться доступними та відновлюваними. Регулярне резервне копіювання всіх важливих даних є необхідним для зменшення ризику втрати даних. Автоматизація процесів відновлення та збереження доступності даних дозволяє організаціям забезпечити безперервність бізнесу та захистити себе від непередбачених збоїв.

1.4.Забезпечення масштабованості та високої продуктивності

Забезпечення масштабованості та високої продуктивності хмарної платформи без шкоди для безпеки є важливим завданням для постачальників SaaS. Ключовим фактором для досягнення цього балансу є використання масштабованих рішень для зберігання даних, таких як об'єктне зберігання та шардінг (Sharding) баз даних. Об'єктне зберігання забезпечує гнучкий спосіб зберігання великих обсягів даних, даючи змогу платформі рости за потреби, зберігаючи при цьому безпеку. Дані, розподілені між різними вузлами, залишаються легко доступними та керованими. В свою чергу, шардінг баз даних розбиває великі бази даних на менші, більш керовані частини, розподіляючи дані між кількома серверами. Це не тільки покращує продуктивність, знижуючи навантаження на сервери, а й забезпечує, щоб

безпекові протоколи, такі як шифрування, застосовувалися послідовно на всіх шардових частинах, зберігаючи чутливу інформацію захищеною.

Якщо платформа масштабується, балансування навантаження стає необхідним для обробки зростаючих вимог користувачів. Балансування навантаження рівномірно розподіляє трафік користувачів між кількома серверами, запобігаючи перевантаженню одного сервера. Це забезпечує швидкий доступ користувачів до платформи навіть під час пікових навантажень. Однак балансування трафіку також створює потенційні безпекові ризики, наприклад, можливість розкриття чутливих даних під час розподілу трафіку. Для зменшення цих ризиків важливо впровадити заходи безпеки, такі як зашифровані канали зв'язку та безпечні протоколи, щоб дані користувачів залишалися захищеними, забезпечуючи швидкий і надійний доступ.

Щоб додатково підтримувати як продуктивність, так і безпеку, моніторинг і аналітика відіграють важливу роль. Постійний моніторинг інфраструктури платформи та аналітика в реальному часі допомагають виявляти незвичні шаблони трафіку або потенційні загрози безпеці. Завдяки активному виявленню та усуненню вразливостей постачальники SaaS можуть нейтралізувати загрози до того, як вони вплинуть на продуктивність. Регулярний аналіз стану системи та даних трафіку дозволяє організаціям підтримувати безперебійну роботу, водночас швидко реагуючи на проблеми з безпекою без порушення здатності платформи обробляти великі обсяги користувачів.

1.5. Основні принципи будови веб-застосунку

Веб-застосунок побудований на багатошаровій архітектурі, де серверна частина та клієнтська частина працюють разом, щоб забезпечити ефективну та швидку взаємодію користувача з застосунком. Основою цієї структури є взаємодія між цими двома компонентами, підтримувана базою даних, яка

зберігає та керує даними застосунку. Кожна частина виконує свою роль, щоб застосунок працював ефективно.

Клієнтська частина веб-застосунку відповідає за інтерфейс користувача. Вона працює у веб-браузері користувача і відповідає за відображення контенту та отримання введених користувачем даних. Коли користувач взаємодіє із застосунком, наприклад, заповнює форму, натискає кнопку, чи переходить між сторінками – клієнтська частина відправляє запити на сервер для обробки. Сервер зазвичай знаходиться на віддаленій машині й відповідає за обробку бізнес-логіки застосунку, обробку запитів користувачів та взаємодію з базою даних. Потім сервер генерує відповідь на запит і надсилає її назад клієнту для відображення.

База даних є основним компонентом, що зберігає всі необхідні дані для підтримки взаємодії з користувачем. Вона зберігає таку інформацію, як дані користувачів, налаштування застосунку та контент, і взаємодіє із сервером для отримання чи оновлення цих даних, коли це потрібно. Сервер обробляє дані, виконує необхідні обчислення чи перетворення і надсилає їх назад клієнту у вигляді динамічного контенту, з яким користувач може взаємодіяти.

Ці компоненти працюють разом, щоб забезпечити функціональність і продуктивність веб-застосунку. Клієнтська частина взаємодіє з серверною, відправляючи запити та отримуючи відповіді, тоді як серверна частина обробляє бізнес-логіку і отримує необхідні дані для обробки запитів. Така організація гарантує ефективне зберігання та доступ до інформації.

1.5.1. Архітектурні рішення для серверної частини

Існують різні архітектурні рішення для організації серверної частини веб-застосунку, кожне з яких має свої переваги та недоліки. Три основні серверні архітектури – це монолітна архітектура, архітектура мікросервісів і безсерверна архітектура. Кожне з цих рішень визначає, як організована серверна частина веб-застосунку та як вона взаємодіє з іншими компонентами.

1. Монолітна архітектура

Монолітна архітектура – це традиційне рішення, коли весь застосунок, включаючи інтерфейс користувача, бізнес-логіку та взаємодію з базою даних, будують як єдину одиницю. У монолітному застосунку всі серверні компоненти тісно пов’язані між собою і працюють разом. Це означає, що будь-які зміни чи оновлення застосунку зазвичай потребують перевстановлення всієї системи. Хоча така архітектура проста в налаштуванні і може бути ефективною для невеликих застосунків, з часом вона стає складною в масштабуванні та підтримці, оскільки зміни в одній частині системи можуть вплинути на інші частини.

2. Мікросервісна архітектура

Мікросервісна архітектура – це більш сучасне рішення, коли застосунок ділиться на менші, незалежні сервіси. Кожен мікросервіс відповідає за конкретну функцію або можливість застосунку, наприклад, аутентифікацію користувачів, обробку платежів чи управління інвентарем. Ці сервіси взаємодіють між собою через мережу, що дозволяє розробляти, розгортати та масштабувати їх незалежно один від одного. Така архітектура забезпечує більшу гнучкість і масштабованість, оскільки кожен сервіс можна масштабувати в залежності від попиту без впливу на весь застосунок. Однак вона також додає складності, оскільки управління кількома сервісами і забезпечення їх безперебійної роботи може бути складним.

3. Безсерверна архітектура

Безсерверна архітектура – це архітектурне рішення, коли розробники створюють і розгортають окремі функції або невеликі частини коду, які виконуються у відповідь на певні події, наприклад, запит користувача чи зміну в базі даних. У безсерверному середовищі немає традиційних серверів, якими треба керувати. Натомість хмарні провайдери автоматично виділяють ресурси та масштабують застосунок за потребою. Це дозволяє розробникам зосередитись на написанні коду, не турбуючись про управління серверами.

Безсерверна архітектура є доволі економною, оскільки доводиться платити тільки за використані ресурси.

1.5.2. Архітектурні рішення для клієнтської частини

Архітектура клієнтської частини також відіграє важливу роль у функціонуванні застосунку. Архітектура клієнтської частини визначає, як структуровані різні компоненти інтерфейсу користувача (UI) і як вони взаємодіють з сервером для отримання або надсилання даних. Існують кілька архітектурних рішень, які використовуються в клієнтській розробці і допомагають покращити продуктивність, досвід користувача та підтримку.

1. Модель односторінкового застосунку (Single-page application, SPA)

У SPA весь застосунок завантажується як одна веб-сторінка, і тільки контент, який змінюється, динамічно оновлюється. Таке архітектурне рішення забезпечує більш плавний досвід для користувача, оскільки сторінка не потребує перезавантаження кожного разу, коли користувач взаємодіє з нею. Односторінкові застосунки зазвичай будуються з використанням JavaScript фреймворків, таких як React, Angular або Vue.js, які займаються динамічним рендерингом контенту та управлінням станом застосунку. SPAs дозволяють швидше переміщатися та взаємодіяти, але вони також потребують особливої уваги до SEO (оптимізація для пошукових систем) і клієнтської маршрутизації.

2. Модель багатосторінкового застосунку (Multi-Page Application, MPA)

MPA – це рішення, де кожна сторінка перезавантажується з сервера, коли користувач переходить до нової частини застосунку. У цій архітектурі клієнт робить окремі запити до сервера для кожної сторінки чи секції. Хоча MPA може бути простіше розробляти і така модель нерідко краще працює для SEO, для користувачів вона часто буде повільнішою, оскільки кожен раз потрібно перезавантажувати всю сторінку.

3. Рендеринг на стороні сервера (Server-Side Rendering, SSR)

У SSR більша частина рендерингу веб-сторінки (HTML, CSS, JavaScript) відбувається на сервері, і клієнт отримує вже повністю сформовану HTML-сторінку. Таке рішення підходить для SEO, оскільки пошукові системи можуть зчитувати вміст сторінки без необхідності виконувати додаткові клієнтські процеси. Застосунки, побудовані за цією архітектурою, можуть комбінувати SSR із клієнтськими фреймворками, такими як React або Vue.js, для динамічного оновлення окремих частин сторінки без її повного перезавантаження. Це дозволяє зберегти переваги швидкого рендерингу та оптимізації для пошукових систем, одночасно забезпечуючи динамічну взаємодію з користувачем.

1.6. Постановка задачі

Метою роботи є розробка хмарної платформи для забезпечення захищеного зберігання та доступу до чутливої інформації у моделі програмного забезпечення як послуга.

Функціональні вимоги до системи:

- Адміністратор може переглядати, створювати, редагувати та видаляти клієнтів.
- Адміністратор може переглядати, створювати та видаляти користувачів для клієнтів.
- Клієнтський користувач може входити у систему.
- Клієнтський користувач може фільтрувати, створювати та видаляти секрети, які належеть цьому клієнту.
- Клієнткий користувач може переглядати, створювати та видаляти ключи авторизації, які використовуються цьому клієнтом.
- Користувач, використовуючи створений клієнтський ключ, може фільтрувати та переглядати секрети цього клієнта.
- Існуючий клієнтський ключ після видалення повинен ставати неактивним.

В рамках проходження переддипломної практики потрібно реалізувати серверну частину веб-додатку.

Для цього необхідно виконати наступні кроки:

1. Спроєктувати та реалізувати базу даних для зберегіння клієнтів, їх даних та службових сутностей.
2. Реалізувати систему автентифікації та авторизації.
3. Створити функціонал для перегляду, створення, редагування та видалення клієнтів.
4. Створити функціонал для перегляду, створення та видалення користувачів, які прив’язані до клієнтів.
5. Додати можливість створення, перегляду, фільтрації та видалення секретів, які прив’язані до клієнтів.
6. Створити функціонал для генерації, перегляду та видалення ключів авторизації клієнтів.

2. ВИБІР МЕТОДІВ РОЗВ'ЯЗАННЯ ПОСТАВЛЕНОЇ ЗАДАЧІ

Вибір методів розв'язання поставленої задачі відіграє ключову роль у процесі розробки веб-застосунку. Правильно підібрані інструменти, технології та підходи дозволяють досягти оптимального співвідношення між швидкістю виконання, якістю програмного забезпечення та його функціональністю. Це також сприяє підвищенню стабільності роботи застосунку, забезпеченню безпеки даних та спрощенню процесів подальшого оновлення чи інтеграції з іншими системами.

2.1. Серверна частина

При розробці хмарної платформи для забезпечення захищеного зберігання та доступу до чутливої інформації у моделі SaaS буде обрано монолітну архітектуру. Це рішення зумовлене необхідністю забезпечити швидкий старт проєкту, ефективне керування ресурсами та централізований контроль безпеки. Варто зазначити, що в даному випадку є певна відмінність від монолітної архітектури, а саме те, що клієнтська частина буде знаходитись окремо від сервера.

Монолітна архітектура дозволить розробити систему як єдиний цілісний застосунок, що спрощує інтеграцію всіх компонентів. Це особливо важливо для платформи, яка працює з чутливою інформацією, оскільки єдина кодова база забезпечує цілісність і централізований захист даних. Реалізація механізмів автентифікації та авторизації, таких як JWT (JSON Web Token), у межах моноліту значно полегшує контроль доступу та знижує ризик помилок, які можуть виникати при взаємодії розподілених систем.

JSON Web Token (JWT) – це стандарт відкритого формату для безпечної передачі інформації між сторонами у вигляді JSON-об'єкта. Він дозволяє передавати дані, підписані за допомогою секретного ключа або пари публічного/приватного ключів, що гарантує їх цілісність і достовірність.

У контексті обраної архітектури JWT використовується для автентифікації користувачів і забезпечення безпеки передачі даних між клієнтською та серверною частинами. Під час входу користувача система генерує токен, який містить необхідну інформацію про ідентифікацію та права доступу. Завдяки підпису сервер може перевірити справжність токена без необхідності звертатися до бази даних, що покращує продуктивність і знижує навантаження.

Цей підхід робить систему більш гнучкою, дозволяючи легко розширювати її функціональність, наприклад, інтегрувати додаткові сервіси або модулі в майбутньому. При цьому використання JWT забезпечує збереження централізованого контролю безпеки навіть у разі подальшої трансформації архітектури.

Крім того, монолітна архітектура забезпечує простоту розгортання та оновлення. Усі зміни в системі впроваджуються одночасно, що мінімізує ризики несумісності компонентів. Це особливо важливо для SaaS-рішення, де стабільність і безперервність роботи платформи є ключовими вимогами.

Незважаючи на те, що мікросервісна архітектура може забезпечити вищу масштабованість у майбутньому, вона вимагає більш складної інфраструктури та значних зусиль на етапі впровадження. На поточному етапі розвитку платформи монолітний підхід дозволяє сконцентруватися на основній функціональності, одночасно зберігаючи можливість поступового переходу до мікросервісної архітектури в разі зростання вимог до масштабованості.

2.1.1. Фреймворк для серверної частини

Таблиця 2.1 – Порівняння бекенд фреймворків.

Характеристика	Node.js	Django	Laravel	.NET Core	Spring
Мова програмування	JavaScript	Python	PHP	C#	Java

Продуктивність	Висока	Середня	Середня	Висока	Висока
Найкраще підходить	Сервісні додатки, RESTful API, SPA	Сайти з великим обсягом контенту	Електронна комерція	Корпоративні застосунки	Корпоративні системи, великі застосунки
Масштабованість	Дуже висока	Висока	Середня	Висока	Дуже висока
Безпека	Залежить від бібліотеки та практик розробки	Висока	Середня, залежить від бібліотек	Висока	Висока

Для розробки серверної частини хмарної платформи буде обрано Node.js з фреймворком Express.js, що забезпечує високу ефективність і масштабованість. Node.js є асинхронною платформою, що дозволяє обробляти численні запити одночасно без блокування процесу, що критично важливо для SaaS-платформи, яка повинна безперебійно обслуговувати велику кількість користувачів. Завдяки цьому підходу платформа може обробляти запити в реальному часі, зберігаючи стабільність і продуктивність навіть при високих навантаженнях.

Використання Express.js в поєднанні з Node.js забезпечує зручність у розробці та масштабуванні серверної частини. Express.js спрощує маршрутизацію запитів, обробку даних і налаштування API, що дозволяє значно скоротити час розробки та підвищити ефективність взаємодії з клієнтською частиною платформи. Крім того, Express.js надає потужні можливості для налаштування безпеки, що є важливим аспектом для платформи, яка працює з чутливою інформацією.

Node.js також є доволі гнучким фреймворком завдяки широкій підтримці для інтеграції з різними механізмами безпеки, такими як JSON Web Token, наприклад. Завдяки цьому розробка серверної частини платформи є швидкою, й безпечною, що є основною вимогою для будь-якої системи, що працює з конфіденційною інформацією.

Ще однією перевагою є те, що Node.js є кросплатформенним, що дозволяє зручно розгортати застосунок на різних операційних системах, таких як Windows, macOS та Linux, а також в хмарних середовищах. Це дає додаткову гнучкість у виборі інфраструктури та дозволяє розробникам адаптувати систему під конкретні потреби.

2.2. Клієнтська частина

Для клієнтської частини платформи для зберігання чутливої інформації обрана односторінкова архітектура (SPA), що дозволяє забезпечити високий рівень зручності, швидкості та ефективності для користувачів. Однією з основних переваг SPA є здатність завантажити весь застосунок за один раз, після чого всі подальші запити до сервера здійснюються асинхронно, без необхідності перезавантаження сторінки. Це дозволяє значно скоротити час відгуку системи та створити безперервний інтерфейс, що значно покращує користувацький досвід.

Такий підхід дуже важливий для роботи з чутливою інформацією, оскільки кожна операція повинна виконуватися швидко та без затримок. З SPA-клієнтом кожен запит до сервера не вимагає повного перезавантаження сторінки, що забезпечує швидку реакцію на дії користувача.

Ще однією важливою перевагою є зменшене навантаження на сервер. Оскільки сервер відповідає лише на API-запити, а не на повний рендер HTML-сторінок, це дозволяє оптимізувати використання серверних ресурсів. Сервер зосереджується на обробці бізнес-логіки та управлінні безпекою, що критично важливо для забезпечення захищеного зберігання та доступу до чутливої

інформації. Це дозволяє підвищити рівень безпеки та забезпечити ефективний контроль доступу, а також зменшити час затримки між запитом користувача та його обробкою.

Також, варто додати, що SPA-архітектура забезпечує зручність у масштабуванні та адаптації платформи до різних умов. Завдяки цьому підходу користувачі можуть працювати з платформою на різних пристроях та з різних локацій, що робить платформу більш доступною та універсальною. Кожен новий елемент функціоналу можна додавати без порушення існуючого процесу, що значно полегшує підтримку та розвиток системи в майбутньому.

2.2.1. Фреймворк для клієнтської частини

Таблиця 2.2 – Порівняння фронтенд фреймворків.

Характеристика	React	Angular	Vue
Мова програмування	JavaScript, TypeScript	TypeScript	JavaScript, TypeScript
Найкраще підходить	SPA, складні інтерфейси	Великі застосунки, SPA	SPA, PWA
Можливість розширення	Дуже висока	Висока, багато вбудованих модулів	Висока, гарна підтримка плагінів та компонентів
Інтеграція з іншими бібліотеками	Легко інтегрується	Обмежений своєю екосистемою	Легко інтегрується
Гнучкість налаштування	Дуже гнучкий	Обмежений своєю екосистемою	Дуже гнучкий
Безпека	Залежить від використовуваних бібліотек	Висока, має вбудовані механізми бепеки	Залежить від використовуваних бібліотек

Для розробки клієнтської частини платформи для зберігання чутливої інформації обрано React.js – одну з найпопулярніших бібліотек для створення інтерфейсів користувача. Вона ідеально підходить для створення ефективних, динамічних та масштабованих односторінкових застосунків (SPA), що є основою нашої платформи. React дозволяє створювати сучасні та інтерактивні інтерфейси, які забезпечують високу швидкість роботи і зручність користування, що критично важливо для платформи, яка обробляє чутливу інформацію.

Однією з основних переваг React є його компонентна архітектура, яка дозволяє розробляти додатки з окремих повторно використовуваних блоків. Кожен компонент є незалежною частиною інтерфейсу, що спрощує його тестування, підтримку та розширення. Така структура дозволяє швидко реагувати на зміни вимог до продукту та забезпечує високу масштабованість, що важливо для подальшого розвитку платформи.

React працює з віртуальним DOM, що значно підвищує продуктивність застосунка. Замість того, щоб оновлювати весь інтерфейс при кожній зміні, React оновлює лише ті частини сторінки, які потребують змін. Це дозволяє значно зменшити час відгуку і підвищити загальну швидкість роботи, що є важливим для платформи, де збереження часу користувача та швидкий доступ до даних є ключовими факторами.

Ще однією важливою перевагою React є його велика екосистема та підтримка сторонніх бібліотек, що дозволяє розширювати можливості фреймворка без необхідності писати все з нуля. Завдяки таким інструментам, як React Router для маршрутизації та Redux для управління станом, можна легко організувати складну логіку застосунку, зберігаючи код чистим і зрозумілим. Це дозволяє ефективно організувати взаємодію з сервером для отримання чутливих даних, а також створювати зручний і безпечний інтерфейс.

Тому React.js є оптимальним вибором для розробки клієнтської частини нашої хмарної платформи для забезпечення захищеного зберігання та доступу

до чутливої інформації, оскільки він забезпечує необхідну гнучкість, високу продуктивність та масштабованість для створення сучасного, безпечного і ефективного інтерфейсу користувача.

2.3. Система керування даними

Таблиця 2.3 – Порівняння СУБД.

Характеристика	PostgreSQL	MS SQL Server	SQLite	MariaDB
Ліцензія	Відкритий код (MIT License)	Програмне забезпечення (Microsoft)	Публічне доменне (Public Domain)	Відкритий код (GPL)
Тип	Об'єктно-реляційна СУБД	Реляційна СУБД	Вбудована SQL база даних	Реляційна СУБД
Модель даних	Реляційна, NoSQL частково (JSONB)	Реляційна	Реляційна	Реляційна
Підтримка платформ	Крос-платформенна	Windows, Linux, macOS	Крос-платформенна	Крос-платформенна
Паралельність	Відповідність ACID, MVCC	Відповідність ACID, блокування	Відповідність ACID	Відповідність ACID
Продуктивність	Висока (складні запити)	Висока (підприємницький рівень)	Висока (малі та середні додатки)	Висока (оптимізовано для MySQL)

Масштабованість	Висока (великий масштаб, розподілені)	Висока (великий бізнес)	Низька (вбудовані додатки)	Висока (сумісність з MySQL)
Найкраще підходить	Великі додатки, цілісність даних	Підприємницькі системи, бізнес-додатки	Мобільні, вбудовані, малі додатки	Веб-додатки, проекти з відкритим кодом

Для зберігання та ефективного керування даними в рамках платформи для зберігання чутливої інформації буде використано PostgreSQL, потужну реляційну систему управління базами даних з відкритим кодом. PostgreSQL забезпечує високу надійність і продуктивність при роботі з великими обсягами даних, що є критично важливим для платформи, яка оперує конфіденційною інформацією. Вона підтримує стандарт ANSI SQL і пропонує розширені можливості, такі як робота з JSON, XML і масивами, що дозволяє ефективно обробляти структуровані та напівструктуровані дані.

Однією з ключових переваг PostgreSQL є її розширюваність та підтримка потужних інструментів для оптимізації роботи з даними, включно з підтримкою складних запитів, створенням індексів різних типів, а також вбудованими механізмами транзакцій із дотриманням ACID-принципів. Це робить PostgreSQL ідеальним вибором для платформ, які потребують високої продуктивності та безпеки.

Для адміністрування та роботи з базою даних доступний зручний графічний інтерфейс, такий як pgAdmin, який дозволяє розробникам і адміністраторам легко виконувати операції, оптимізувати запити та управляти схемами бази даних. Додатково PostgreSQL підтримує вбудовані функції та тригери, що дозволяє розширювати її функціональність і автоматизувати ключові процеси. Усе це забезпечує стабільність, безпеку та гнучкість управління базою даних для критично важливих завдань. Для взаємодії з базою даних на сервері застосовується Knex.js, гнучка бібліотека для роботи з

SQL базами даних на платформі Node.js. Knex.js забезпечує високий рівень абстракції при виконанні SQL-запитів, що дозволяє зручно працювати з різними базами даних. Завдяки цьому підходу ми зменшуємо складність коду та робимо систему більш гнучкою та масштабованою.

Knex.js також надає можливості для роботи з міграціями, що автоматизують процес змін у схемі бази даних. Це дозволяє ефективно керувати версіями бази даних, спрощуючи процес оновлення та масштабування системи, особливо коли мова йде про інтеграцію нових функціональностей або зміну структури даних. Окрім того, Knex.js підтримує побудову запитів, що дає можливість динамічно формувати SQL-запити без необхідності ручного написання складних запитів, що значно пришвидшує розробку.

3. ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1. Проєктування системи

Проєктування системи є одним з ключових етапів розробки програмного забезпечення, оскільки це визначає архітектуру, функціональність та взаємодію системи з користувачами та зовнішніми компонентами. Для створення ефективної системи необхідно створити чіткий опис її функціональності з урахуванням потреб користувачів та специфіки бізнес-процесів. Також на цьому етапі важливим завданням є визначення основних варіантів використання системи. Це допомагає деталізувати функції, які повинна виконувати система, як з нею взаємодіють зовнішні користувачі, а також окреслити межі відповідальності системи.

Для візуального представлення функціональності системи було розроблено діаграму варіантів використання, яка ілюструє основні сценарії взаємодії між акторами (користувачами або іншими системами) та запропонованими функціями. Діаграма забезпечує візуалізацію всіх основних процесів всередині системи, дозволяючи визначити ключові взаємодії та залежності.

На рис 3.1 наведено діаграму варіантів використання, яка ілюструє структуру та основні сценарії роботи системи.

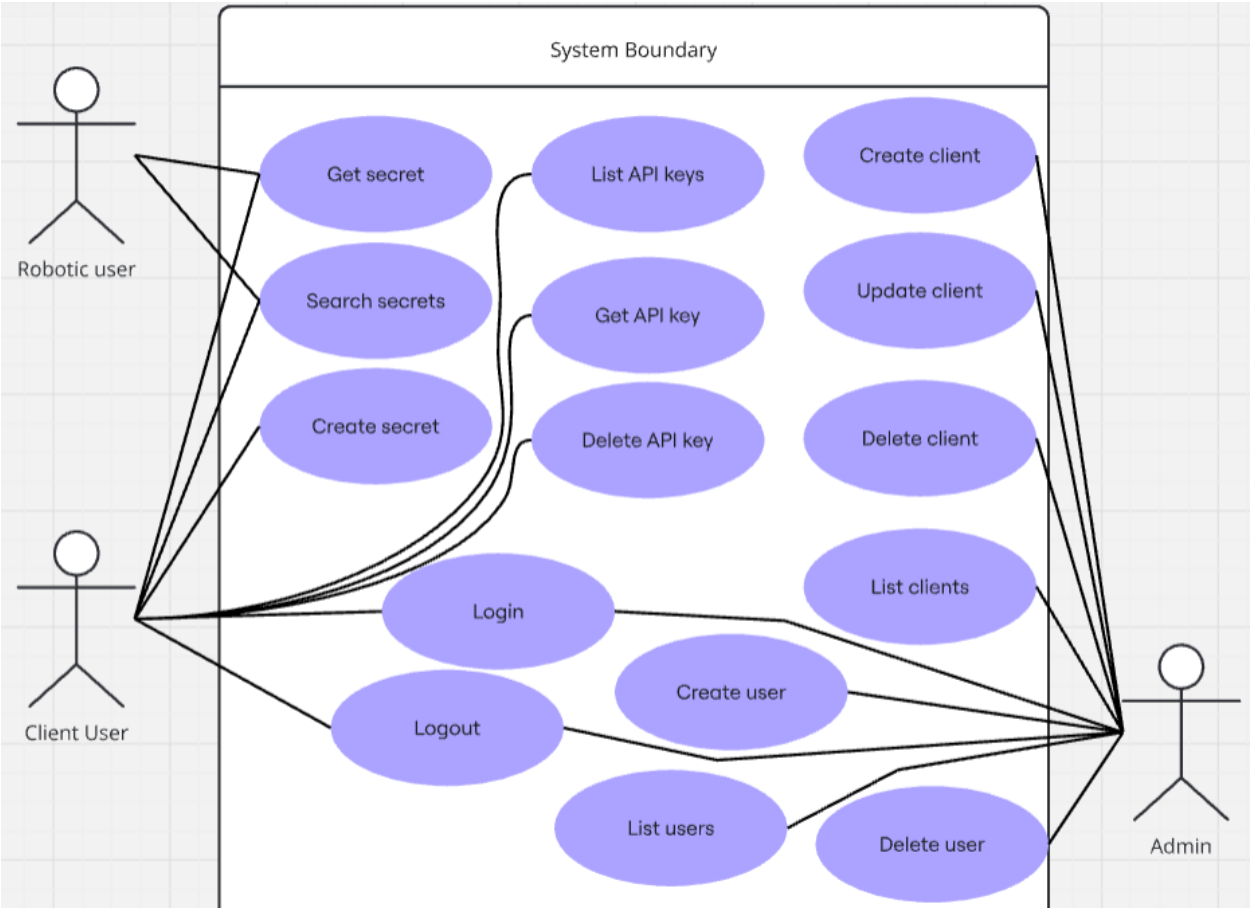

Рисунок 3.1 Діаграма варіантів використання

Дана система включає таких акторів:

1. Адміністратор – внутрішній користувач, який створює необхідну інфраструктуру для клієнтів.
2. Клієнтський користувач – адміністратор рівня клієнта, керує секретами та ключами, які надають доступ до секретів клієнта.
3. Програмний користувач – умовний користувач, який має доступ для читання секретів клієнта, якому він належить.

Розберемо більш детально функціонал застосунку, створивши діаграму послідовності. Для початку розглянемо процес авторизації.

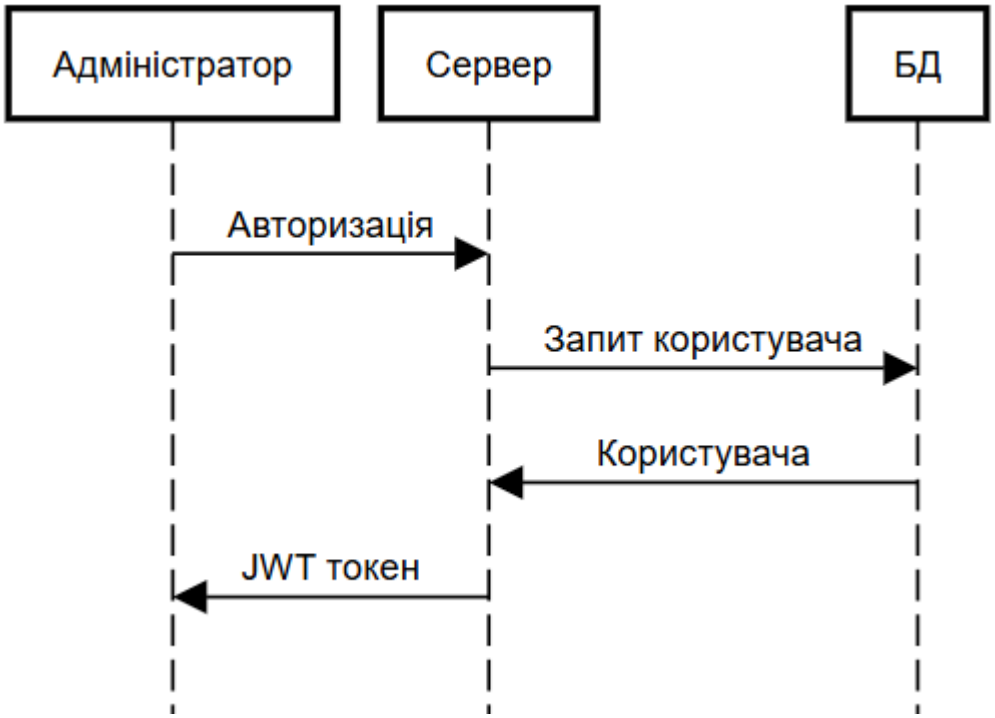

Рисунок 3.2 Авторизація

Авторизовані користувачі мають доступ до керування чутливою інформацією.

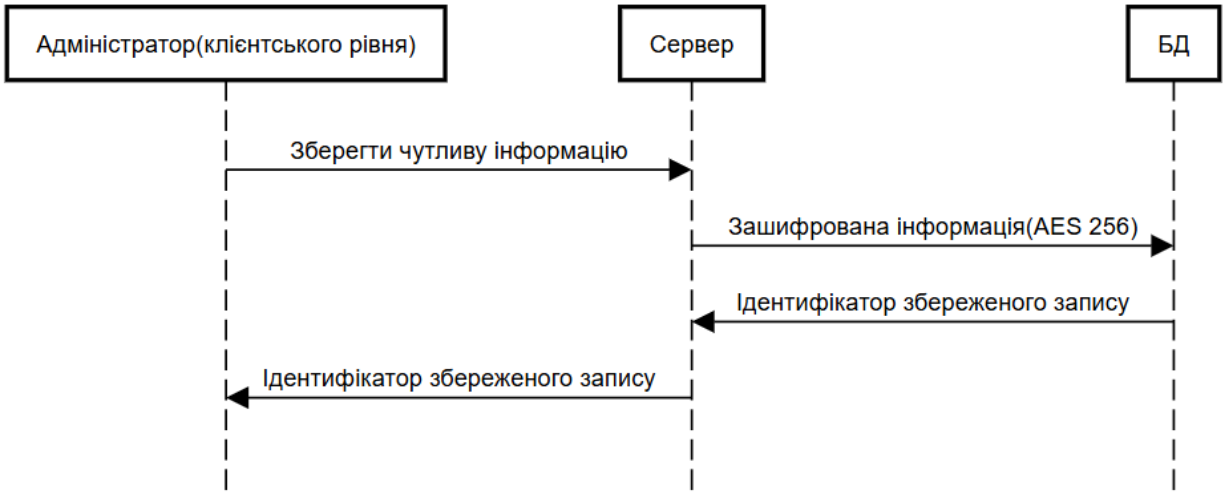

Рисунок 3.3 Збереження чутливої інформації

Спираючись на дані вимоги і варіанти використання, спроєктуємо базу даних цієї платформи. Для моделювання використаємо діаграму зв’язку сутностей.

Діаграма зв’язку сутностей – це діаграма, яка відображає зв’язок сутностей, які є в базі даних, також відома як ERD. Ця діаграма допомагає краще та більш наочно зрозуміти структуру бази даних.

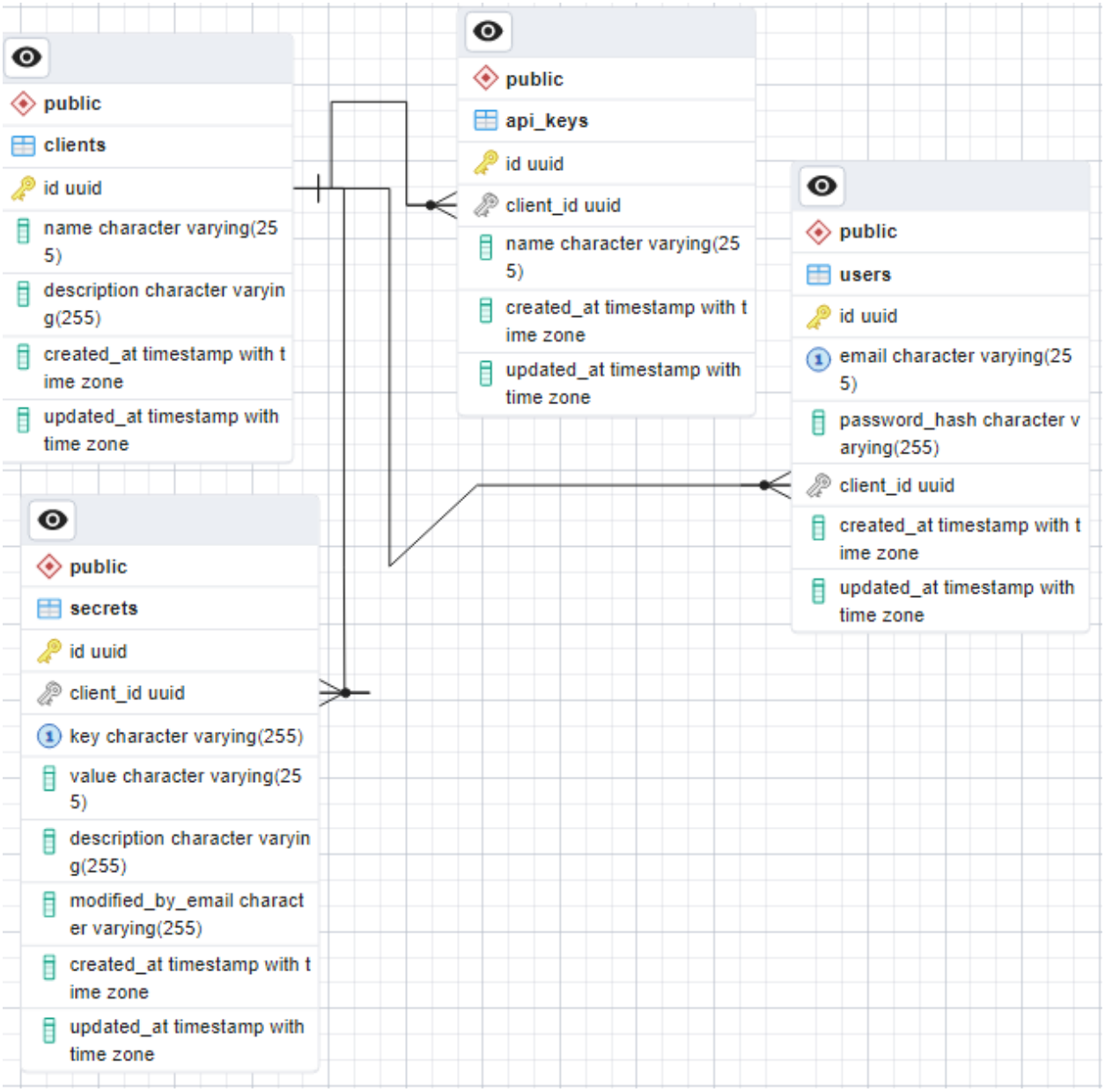

Рисунок 3.4 ERD системи

3.2. Серверна складова

Система автентифікації токенів є частиною забезпечення безпеки розроблюваної платформи. Для авторизації реалізуємо middleware-функцію «authenticateToken», яка буде використовуватися роутерами у Express.js екосистемі. Ця функція буде фабрикою middleware-функцій. Вона буде приймати параметри, які визначатимуть чи буде блокувати, чи пропускати певних користувачів до кінцевого місця призначення.

```
const defaultParams: AuthenticateTokenParams = {
  allowRoboticUsers: false,
  allowAdminUsers: false,
  allowClientUsers: false,
};

export const authenticateToken =
  (params: AuthenticateTokenParams = defaultParams) =>
  async (req: Request, res: Response, next: NextFunction) => {
    const authHeader = req.headers["authorization"];
    const token = authHeader?.split(" ")[1];

    if (!token) {
      return res.status(401).json({ message: "Authentication token required" });
    }

    try {
      const user = jwt.verify(token, config.jwt.secret) as TokenPayload;
      req.user = user;
```

Рисунок 3.5 Реалізація authenticateToken

Для роботи з користувачами створимо UserService.

Даний сервіс має наступні методи:

- findByClientId – повертає користувачів, які належать специфічному клієнту.
- findAll – повертає всіх користувачів.
- deleteUser – видалення користувача за ідентифікатором.

Для підтримки реєстрації та логіну користувачами створимо AuthService.

Даний сервіс має наступні методи:

- login – отримання JWT-токену з поштою та паролем.
- register – створення нового користувача.

Для роботи з клієнтами створимо ClientService.

Даний сервіс має наступні методи:

- findAll – отримати список всіх клієнтів.
- findById – отримати клієнта за його ідентифікатором.
- create – створити нового клієнта.
- update – оновити існуючого клієнта за його ідентифікатором.

- delete – видалити клієнта за його ідентифікатором.

Для керування чутливою інформацією створимо SecretService.

Даний сервіс має наступні методи:

- list – отримати список існуючих секретів. Приймає текст пошуку як аргумент.
- getByKey – отримати секрет і його значення за ключем.
- upsert – створити або оновити наявний секрет.
- delete – видалити секрет за його ідентифікатором.

Для роботи з ключами авторизації створимо ApiKeyService.

Даний сервіс має методи:

- create – створення нового ключа, і генерація JWT-токена
- list – отримання списку існуючих ключів на рівні клієнта.
- getById – пошук ключа за ідентифікатором.
- delete – видалення ключа за ідентифікатором.

Пошук ключа за ідентифікатором стає в нагоді, коли користувач авторизується з використанням такого токену. Система розпізнає даний тип токенів і перевіряє чи не видалений даний токен. При генерації API-токену зберігається також його тип «api-key» та ідентифікатор ключа в тілі токена. Таким чином система має змогу відслідковувати та блокувати недійсні токени. EncryptionService призначений для роботи з шифруванням і розшифруванням даних. Його основна мета – забезпечити шифрування чутливої інформації перед збереженням до бази даних та після отримання – розшифрувати, повернувши попередній вигляд. Для шифрування використаний алгоритм AES-256.

AES (Advanced Encryption Standard) — це алгоритм симетричного шифрування, прийнятий як міжнародний стандарт для захисту даних. В цьому алгоритмі один ключ використовується для шифрування та розшифрування даних.

Маючи сервіси, створимо відповідні контролери та додамо їх до роутингу. Загалом має вийти наступна файлова структура серверної частини:

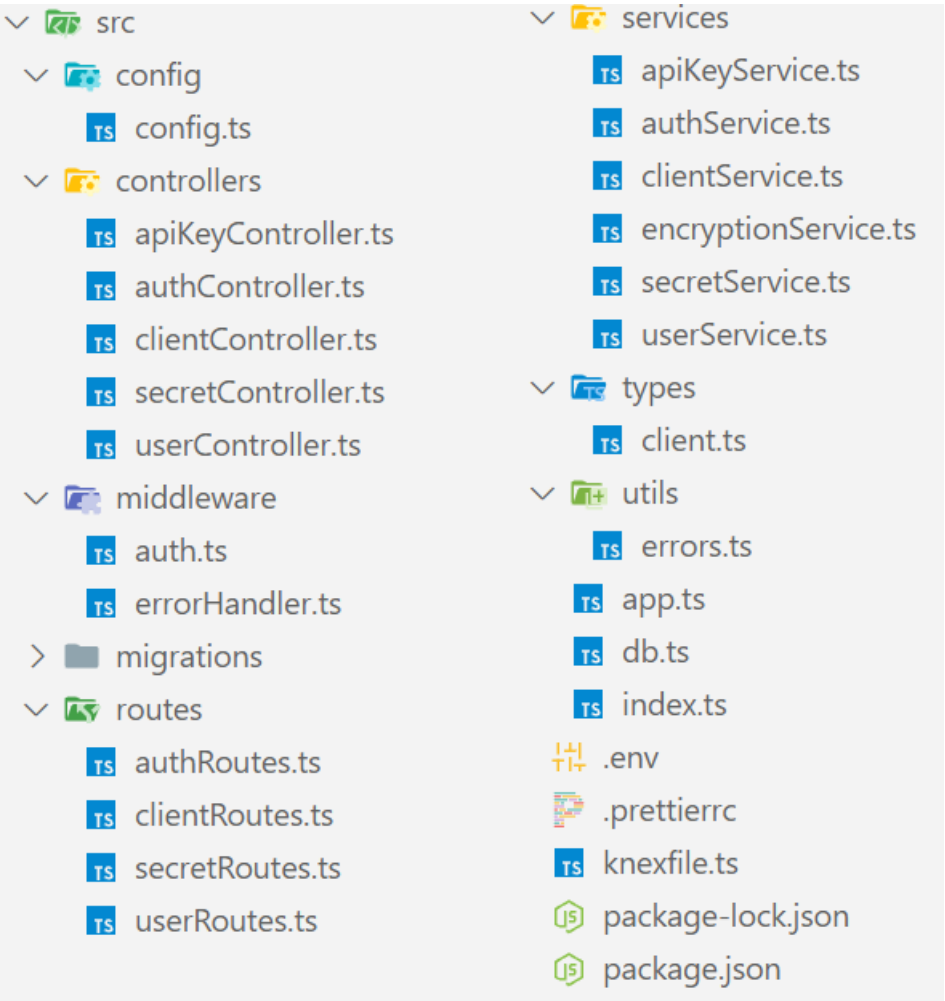

Рисунок 3.6 Серверна файлова структура

3.3. Фронтенд складова

Клієнтська частина написана з використанням React. При побудові користувацького інтерфейсу буде використане розподілення між сторінками. Сторінки будуть окремими наборами компонентів, які малопов’язані одна з одною. Також окремо будуть створені компоненти, які будуть перевикористовуватися у різних місцях застосунку.

Створимо сторінку авторизації.

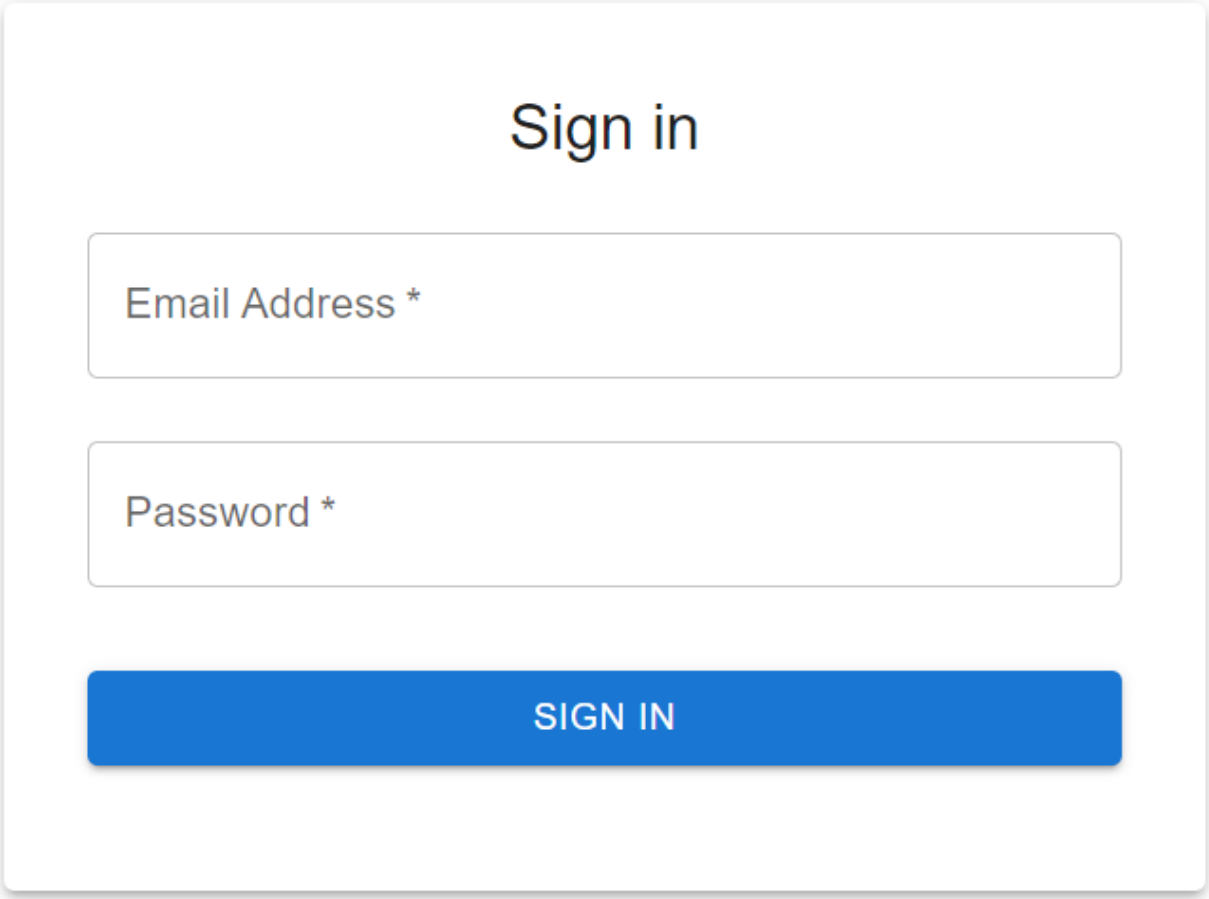

Рисунок 3.7 Зовнішній вигляд сторінки для авторизації

Домашніх сторінок буде дві. Адміністратор повинен бути переадресованим на сторінку керування клієнтами. Клієнтський користувач повинен бути переадресований на сторінку керування свого клієнта. Клієнтський користувач повинен мати змогу керувати секретами клієнта та його API ключами.

Створимо сторінку керування клієнтами.

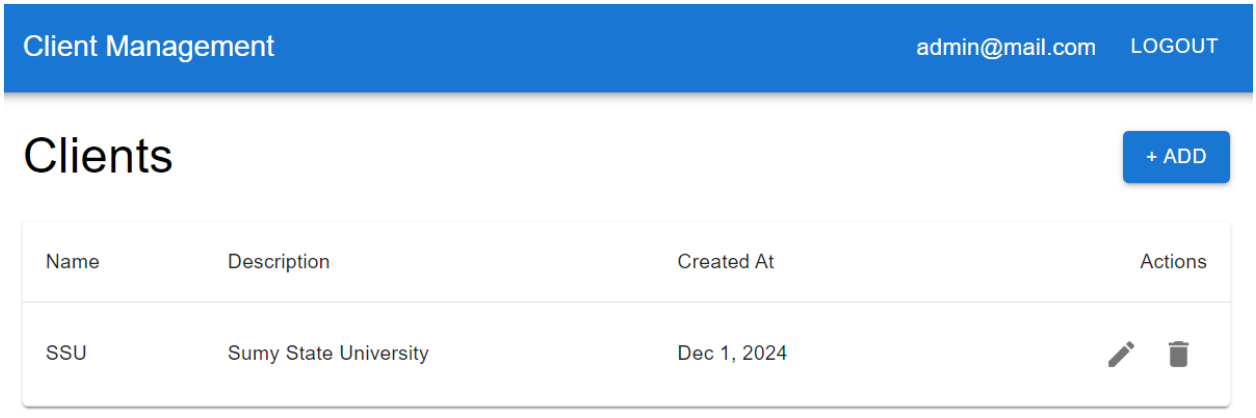

Рисунок 3.8 Зовнішній вигляд сторінки керування клієнтами

При відкритті певного клієнта, адміністратор має доступ до редагування та створення користувачів для клієнта. Створимо сторінку з деталями клієнта.

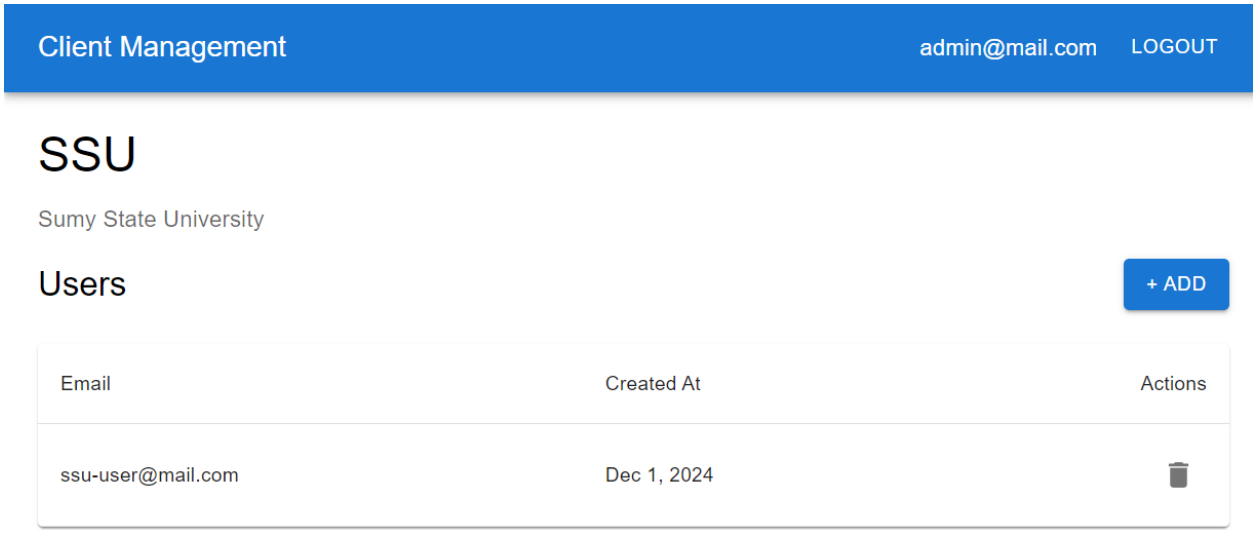

Рисунок 3.9 Зовнішній вигляд сторінки керування деталями клієнта

Тепер необхідно додати сторінку керування клієнтських даних. На таку сторінку буде переадресований клієнтський користувач після авторизації.

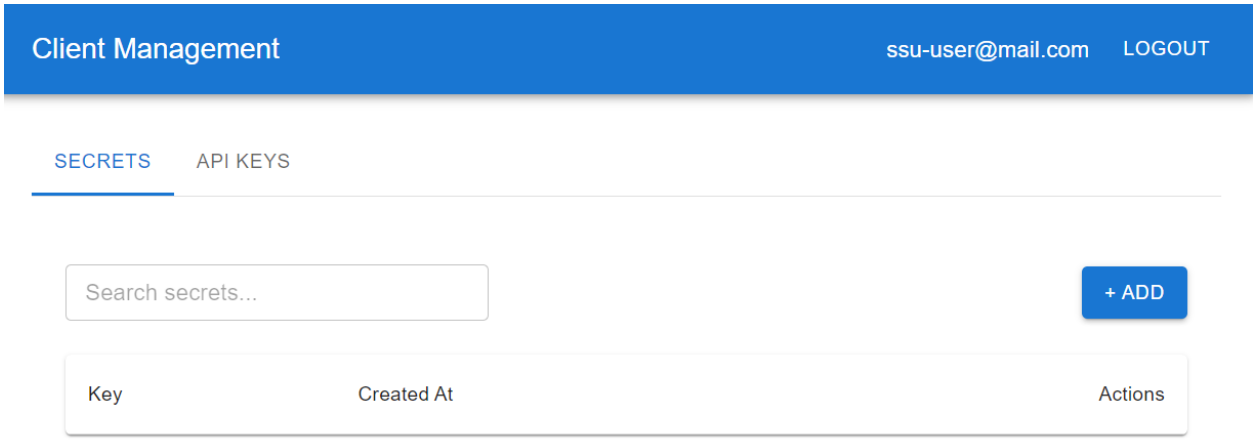

Рисунок 3.10 Зовнішній вигляд сторінки керування клієнтських даних

Загалом має вийти наступна файлова структура клієнтської частини:

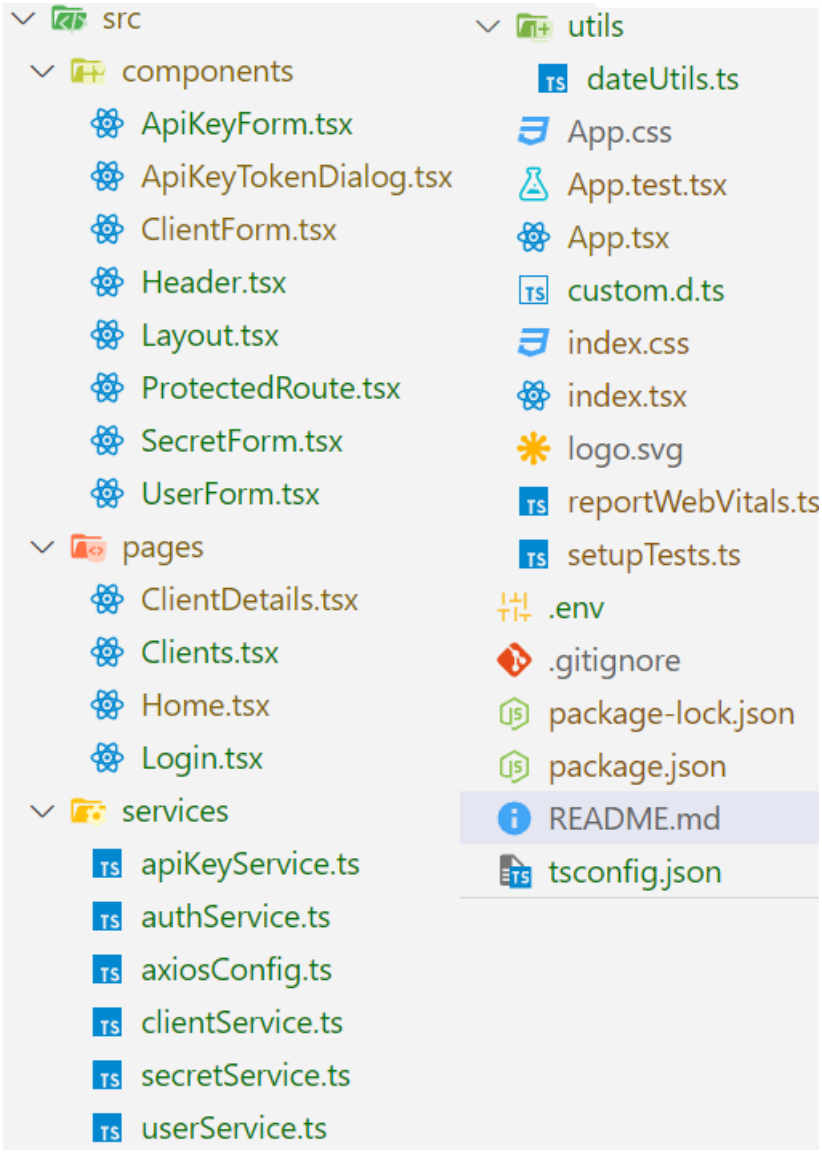

Рисунок 3.11 Клієнтська файлова структура

3.4. Тестування системи

В рамках тестування виконаємо увесь флов від початку до кінця.

Створимо нового клієнта. Для цього заповнимо його ім’я і натиснемо кнопку створення «Create».

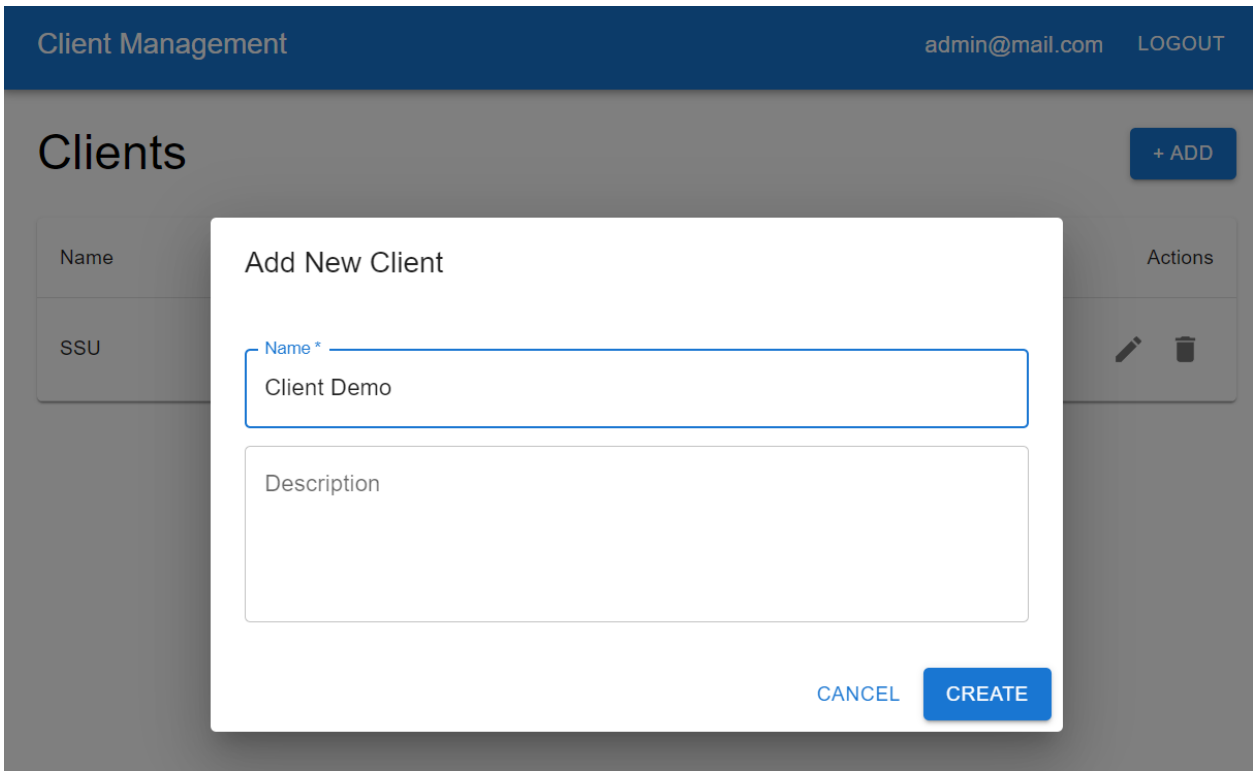

Рисунок 3.12 Створення клієнта

Після створення клієнт буде доданий до загального списку.

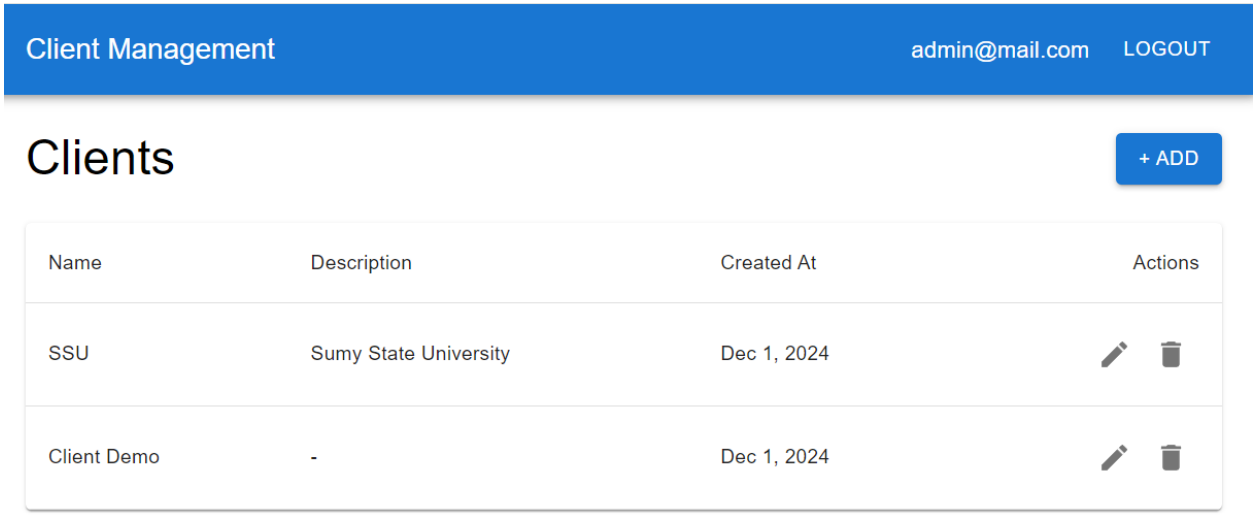

Рисунок 3.13 Список клієнтів

Натиснувши на рядку клієнта ми відкриємо сторінку керування користувачами клієнта. Перейшовши на ту сторінку додамо користувача, заповнивши логін та пороль.

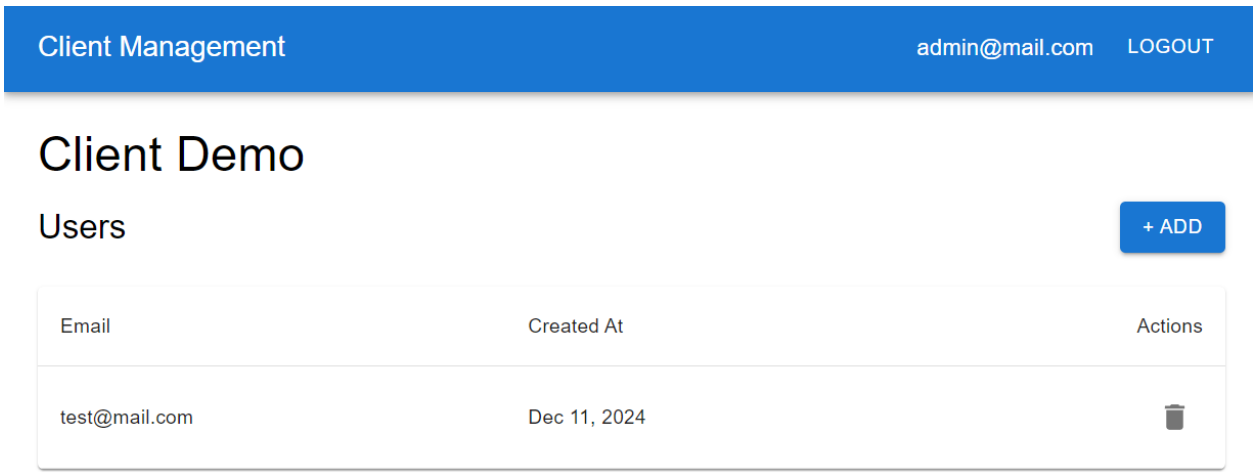

Рисунок 3.14 Сторінка зі створеним користувачем

На цьому етапі підготовлення завершені і новосторений користувач готовий до самостійної роботи і інтеграції зі своїми API. Залогінимося цим користувачем та додамо новий секрет для клієнта.

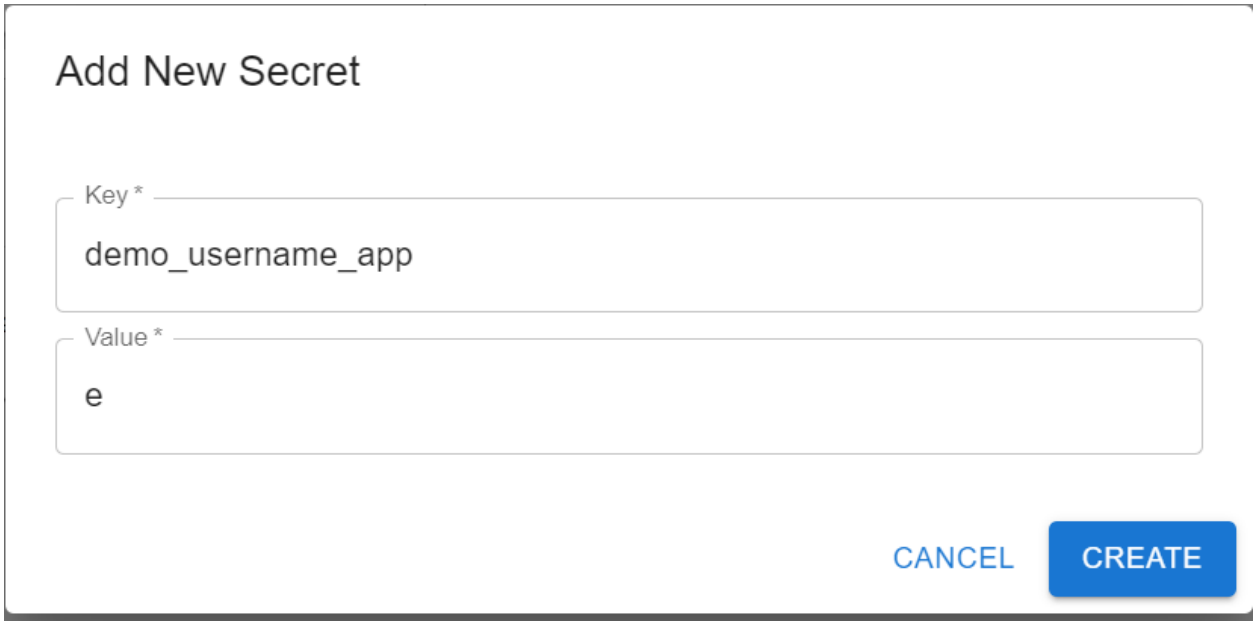

Рисунок 3.15 Додавання нового секрету

Додамо ще кілька значень для збільшення вибірки.

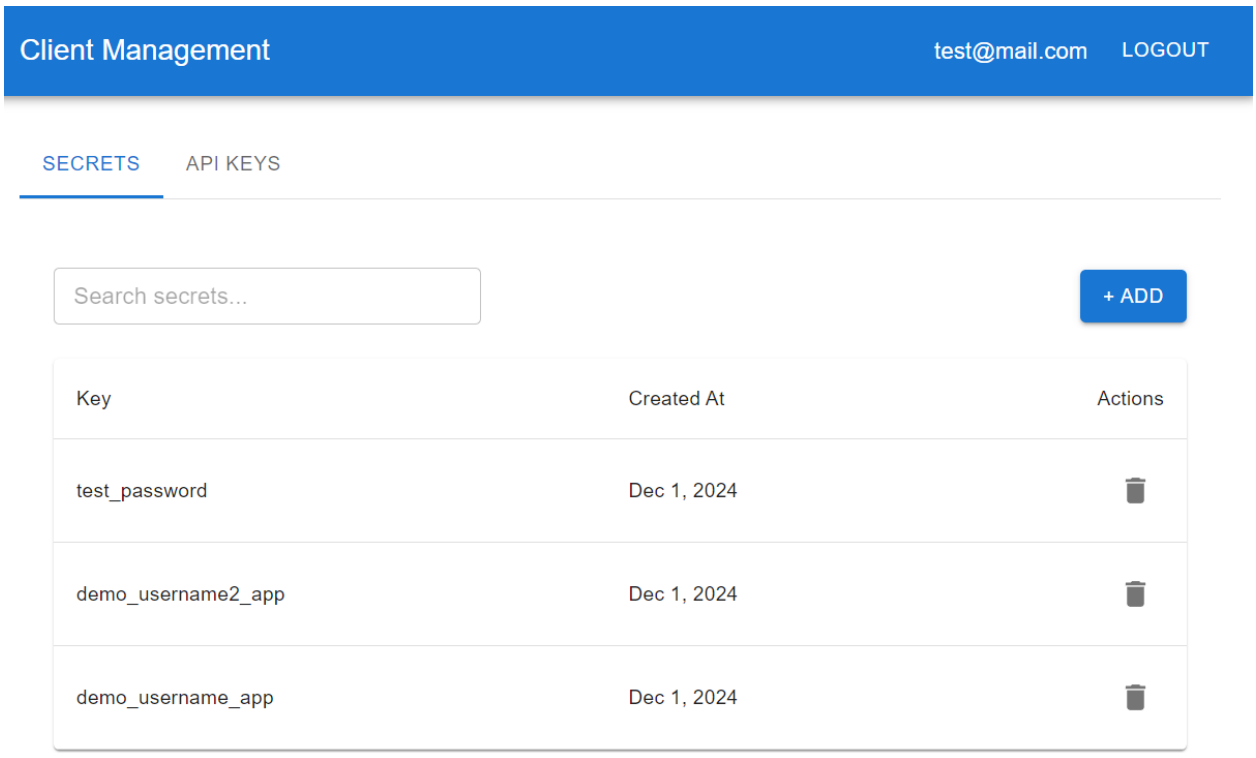

Рисунок 3.16 Створені секрети

Перейшовши на вкладку «API KEYS» ми побачимо сторінку з керуванням ключів доступу для сторонніх учасників. Користувач може додати новий ключ чи видалити існуючий.

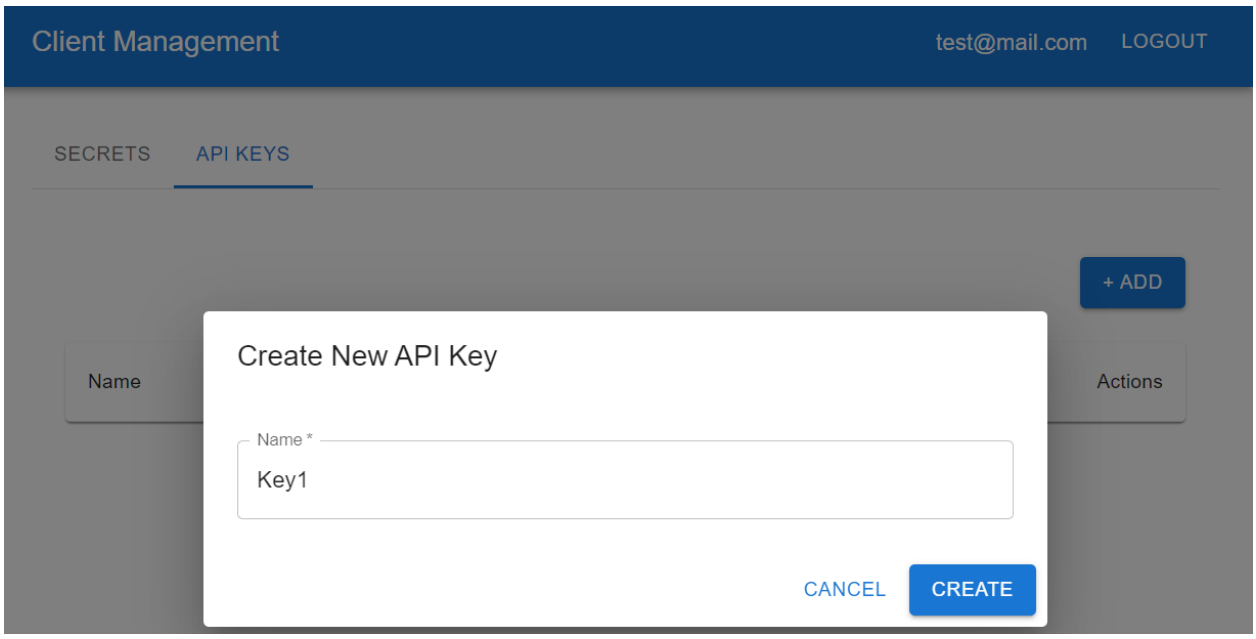

Рисунок 3.17 Створення нового ключа доступу

Після успішного створення користувачу буде відображено новостворений токен доступу. Цей токен буде показаний лише раз при створенні.

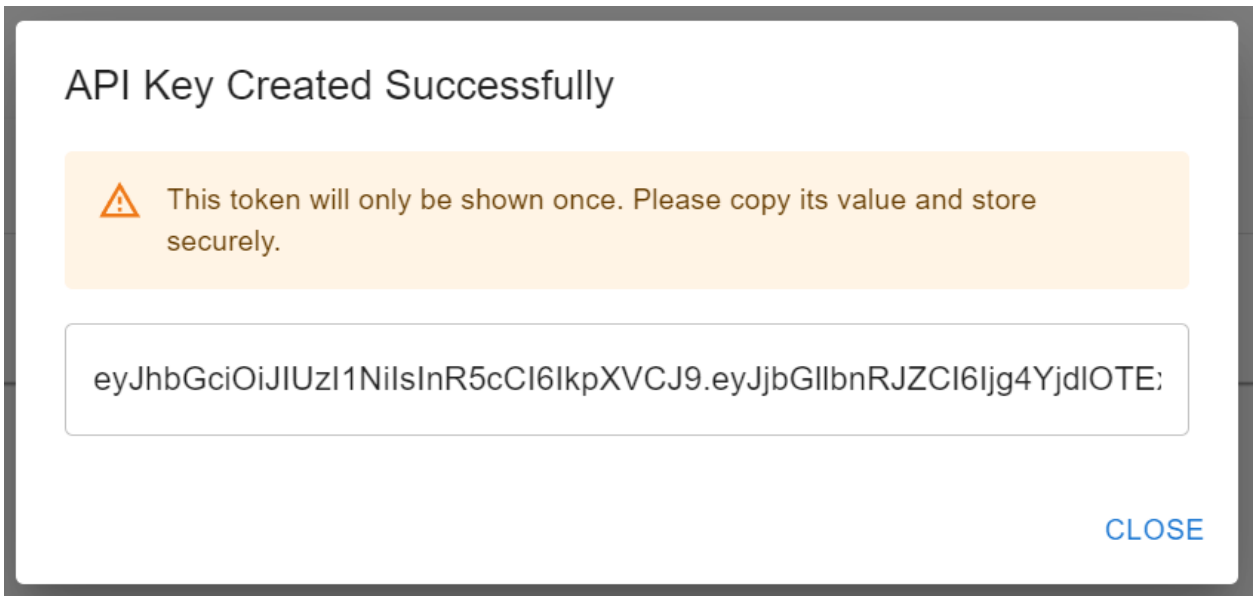

Рисунок 3.18 Щойно створений ключ доступу

Використовуючи даний токен для авторизації можна отримати значення секретів клієнта.

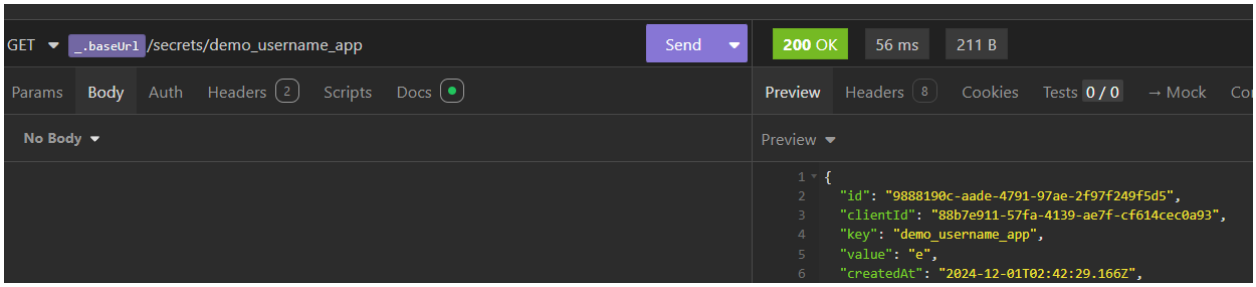

Рисунок 3.19 Демонстрація отримання доступу з ключем доступом

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було проведено аналіз сучасних підходів до побудови хмарних застосунків. Завдяки цьому аналізу вдалося підготуватися до подальшого проєктування та сформувати мету дослідження.

За результатами виконаного аналізу були зібрані вимоги до хмарної платформи для забезпечення захищеного зберігання та доступу до чутливої інформації у моделі програмного забезпечення як послуга. Ці вимоги включають підтримку різних користувачів, управління секретами, створення та використання ключів авторизації, а також забезпечення високого рівня захисту даних.

Було спроєктовано і розроблено серверну частину системи, що забезпечує реалізацію основного функціоналу. З метою візуалізації логіки роботи платформи та демонстрації її ключових можливостей створено діаграму варіантів використання (Use Case Diagram), яка відображає акторів системи та основні функції застосунку.

Також було розроблено клієнтську частину, завдяки якій можна зручно взаємодіяти із застосунком та користуватись розробленим функціоналом.

Наостанок було проведено тестування розробленої платформи для забезпечення захищеного зберігання та доступу до чутливої інформації. В результаті цього тестування було перевірено функціонал системи на коректність роботи.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Mather, T., Kumaraswamy, S., & Latif, S. (2009). *Cloud security and privacy: An Enterprise Perspective on Risks and Compliance*. O'Reilly Media.
2. Jamsa, K. (2013). *Cloud computing: SaaS, PaaS, IaaS, Virtualization, Business Models, Mobile, Security and More*. Jones & Bartlett Publishers.
3. Ko, R., & Choo, R. (2015). *The cloud Security Ecosystem: Technical, Legal, Business and Management Issues*. Syngress Media Incorporated.
4. Rittinghouse, J. W., & Ransome, J. F. (2009). *Cloud computing: Implementation, Management, and Security*. CRC Press.
5. Kleppmann, M. (2017). *Designing data-intensive applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. Oreilly & Associates Incorporated.
6. Fowler, M. (2003). *Patterns of enterprise Application Architecture*. Addison-Wesley Professional.
7. Newman, S. (2019). *Monolith to microservices: Sustaining Productivity While Detangling the System*. O'Reilly Media.
8. Osmani, A. (2012). *Learning JavaScript design patterns*. "O'Reilly Media, Inc."
9. Casciaro, M., & Mammino, L. (2020). *Node. JS Design Patterns: Design and Implement Production-Grade Node. Js Applications Using Proven Patterns and Techniques, 3rd Edition*.
10. Herron, D. (2020b). *Node.js Web Development - Fifth Edition: Server-side Web Development Made Easy with Node 14 Using Practical Examples*.
11. Martin, R. C. (2018). *Clean architecture: A Craftsman's Guide to Software Structure and Design*. Pearson Professional.
12. Herron, D. (2020). *Node.js Web Development - Fifth Edition: Server-side Web Development Made Easy with Node 14 Using Practical Examples*.

13. Mikowski, M., & Powell, J. C. (2013). *Single page web applications: JavaScript End-to-end*. Manning Publications.
14. Wieruch, R. (2018). *The road to react: With React 18 and React Hooks : Required Knowledge: JavaScript*.
15. Ciolli, G., Angelakos, J., & Mejías, B. (2023). PostgreSQL 16 Administration Cookbook: Solve Real-world Database Administration Challenges with 180+ Practical Recipes and Best Practices.
16. *Index | Node.js v23.3.0 Documentation*. (n.d.). https://nodejs.org/docs/latest/api/
17. auth0.com. (n.d.). *JWT.IO - JSON Web Tokens Introduction*. JSON Web Tokens - jwt.io. https://jwt.io/introduction
18. *Express/Node introduction - Learn web development | MDN*. (2024, November 22). MDN Web Docs. https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction
19. *Knex.js*. (n.d.). AppSignal Documentation. https://docs.appsignal.com/nodejs/3.x/integrations/knexjs.html
20. *Getting started with React - Learn web development | MDN*. (2024, November 22). MDN Web Docs. https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/React_getting_started
21. PostgreSQL: documentation. (n.d.). The PostgreSQL Global Development Group. https://www.postgresql.org/docs/
22. Вікімедіа, У. П. (n.d.). *Advanced Encryption Standard*. https://uk.wikipedia.org/wiki/Advanced_Encryption_Standard
23. *Crypto | Node.js v23.4.0 Documentation*. (n.d.). https://nodejs.org/api/crypto.html

24. Kananda, V. (2024, November 13). Why you should use AES 256 encryption to secure your data. *Progress Blogs*. https://www.progress.com/blogs/use-aes-256-encryption-secure-data

ДОДАТОК А

```
import { Request, Response, NextFunction } from "express";
import { ClientService } from "../services/clientService";
import { CreateClientDto, UpdateClientDto } from "../types/client";
import { NotFoundError } from "../utils/errors";

export class ClientController {
  constructor(private readonly clientService: ClientService) {}

  createClient = async (req: Request, res: Response, next:
NextFunction): Promise<void> => {
    const data: CreateClientDto = req.body;
    const client = await this.clientService.create(data);
    res.status(201).json(client);
  };

  getAllClients = async (req: Request, res: Response, next:
NextFunction): Promise<void> => {
    const clients = await this.clientService.findAll();
    res.json(clients);
  };

  getClientById = async (req: Request, res: Response, next:
NextFunction): Promise<void> => {
    const client = await this.clientService.findById(req.params.id);
    if (!client) {
      throw new NotFoundError("Client not found");
    }
    res.json(client);
  };

  updateClient = async (req: Request, res: Response, next:
NextFunction): Promise<void> => {
    const data: UpdateClientDto = req.body;
    const client = await this.clientService.update(req.params.id,
data);
    if (!client) {
      throw new NotFoundError("Client not found");
    }
    res.json(client);
  };
```

```
  deleteClient = async (req: Request, res: Response, next:
NextFunction): Promise<void> => {
    const deleted = await this.clientService.delete(req.params.id);
    if (!deleted) {
      throw new NotFoundError("Client not found");
    }
    res.status(204).send();
  };
}


import { Request, Response } from "express";
import { UserService } from "../services/userService";

export class UserController {
  constructor(private readonly userService: UserService) {}

  listUsers = async (req: Request, res: Response): Promise<void> => {
    const { clientId } = req.query;

    if (clientId) {
      const users = await this.userService.findByClientId(clientId as
string);
      res.json(users);
      return;
    }

    const users = await this.userService.findAll();
    res.json(users);
  };

  deleteUser = async (req: Request, res: Response): Promise<void> => {
    await this.userService.delete(req.params.id);
    res.status(204).send();
  };
}



import { Request, Response } from "express";
import { AuthService } from "../services/authService";
import { BadRequestError } from "../utils/errors";

export class AuthController {
```

```
  constructor(private readonly authService: AuthService) {}

  login = async (req: Request, res: Response): Promise<void> => {
    const { email, password } = req.body;

    if (!email || !password) {
      throw new BadRequestError("Email and password are required");
    }

    const token = await this.authService.login(email, password);
    res.json({ token });
  };

  register = async (req: Request, res: Response): Promise<void> => {
    const { email, password, clientId } = req.body;

    if (!email || !password) {
      throw new BadRequestError("Email and password are required");
    }

    const token = await this.authService.register({
      email,
      password,
      clientId: clientId || null,
    });

    res.status(201).json({ token });
  };
}



import { Request, Response } from "express";
import { SecretService } from "../services/secretService";

export class SecretController {
  constructor(private readonly secretService: SecretService) {}

  async list(req: Request, res: Response) {
    const clientId = this.getClientId(req);
    const { key } = req.query;

    const filters = {
      clientId,
```

```
      ...(typeof key === "string" && { key }),
    };

    const secrets = await this.secretService.list(filters);
    res.json(secrets);
  }

  async upsert(req: Request, res: Response) {
    const clientId = this.getClientId(req);
    const { id, key, value } = req.body;

    const secret = await this.secretService.upsert(clientId, {
      id,
      key,
      value,
    });

    res.status(id ? 200 : 201).json(secret);
  }

  async getByKey(req: Request, res: Response) {
    const clientId = this.getClientId(req);
    const { key } = req.params;
    const secret = await this.secretService.getByKey(key, clientId);
    res.json(secret);
  }

  async delete(req: Request, res: Response) {
    const clientId = this.getClientId(req);
    const { id } = req.params;
    await this.secretService.delete(id, clientId);
    res.status(204).send();
  }

  private getClientId(req: Request): string {
    if (!req.user) {
      throw new Error("Unauthorized: User not authenticated");
    }

    const { clientId } = req.user;
    if (!clientId) {
      throw new Error("Forbidden: User is not associated with a
client");
    }
```

```
    return clientId;
  }
}



import { Request, Response } from "express";
import { ApiKeyService } from "../services/apiKeyService";

export class ApiKeyController {
  constructor(private readonly apiKeyService: ApiKeyService) {}

  async create(req: Request, res: Response) {
    const clientId = this.getClientId(req);
    const { name } = req.body;

    const result = await this.apiKeyService.create({
      clientId,
      name,
    });

    res.status(201).json(result);
  }

  async list(req: Request, res: Response) {
    const clientId = this.getClientId(req);
    const apiKeys = await this.apiKeyService.list(clientId);
    res.json(apiKeys);
  }

  async delete(req: Request, res: Response) {
    const clientId = this.getClientId(req);
    const { id } = req.params;
    await this.apiKeyService.delete(id, clientId);
    res.status(204).send();
  }

  private getClientId(req: Request): string {
    if (!req.user) {
      throw new Error("Unauthorized: User not authenticated");
    }

    const { clientId } = req.user;
```

```
    if (!clientId) {
      throw new Error("Forbidden: User is not associated with a
client");
    }

    return clientId;
  }
}



import { Request, Response, NextFunction } from "express";
import jwt from "jsonwebtoken";
import config from "../config/config";
import { ApiKeyService } from "../services/apiKeyService";
import { db } from "../db";

interface TokenPayload {
  userId: string;
  clientId: string | null;
  keyId?: string;
  type?: string;
  email: string;
}

declare global {
  namespace Express {
    interface Request {
      user?: TokenPayload;
    }
  }
}

interface AuthenticateTokenParams {
  allowRoboticUsers?: boolean;
  allowAdminUsers?: boolean;
  allowClientUsers?: boolean;
}

const defaultParams: AuthenticateTokenParams = {
  allowRoboticUsers: false,
  allowAdminUsers: false,
  allowClientUsers: false,
};
```

```
const isRoboticUser = (user: TokenPayload) => {
  return user.type === "api_key";
};

export const authenticateToken =
  (params: AuthenticateTokenParams = defaultParams) =>
  async (req: Request, res: Response, next: NextFunction) => {
    const authHeader = req.headers["authorization"];
    const token = authHeader?.split(" ")[1];

    if (!token) {
      return res.status(401).json({ message: "Authentication token
required" });
    }

    try {
      const user = jwt.verify(token, config.jwt.secret) as
TokenPayload;
      req.user = user;

      if (
        (params.allowAdminUsers && user.clientId === null) ||
        (params.allowClientUsers && !isRoboticUser(user) &&
user.clientId !== null && user.clientId !== "")
      ) {
        return next();
      }

      if (params.allowRoboticUsers && isRoboticUser(user)) {
        const apiKeyService = new ApiKeyService(db);
        const apiKey = await apiKeyService.getById(user.keyId!);

        if (!apiKey) {
          throw new Error("Invalid API key");
        }

        return next();
      }

      throw new Error("Invalid token");
    } catch (error) {
      return res.status(403).json({ message: "Invalid or expired
token. Please login again." });
```

```
    }
  };
```

```
import { ErrorRequestHandler } from "express";
import { AppError } from "../utils/errors";

export const errorHandler: ErrorRequestHandler = (err, req, res, next)
=> {
  if (err instanceof AppError) {
    res.status(err.statusCode).json({
      status: "error",
      message: err.message,
    });
    return;
  }

  // Log unexpected errors
  console.error("Unexpected error:", err);

  res.status(500).json({
    status: "error",
    message: "Internal server error",
  });
};
```

```
import { Router } from "express";
import { AuthController } from "../controllers/authController";
import { authenticateToken } from "../middleware/auth";

export const createAuthRouter = (authController: AuthController):
Router => {
  const router = Router();

  router.post("/login", authController.login);

  // Only authenticated admins can register new users
  router.post("/register", authenticateToken({ allowAdminUsers: true
}), authController.register.bind(authController));

  return router;
```

```
};
```

```
import { Router } from "express";
import { ApiKeyController } from "../controllers/apiKeyController";
import { ClientController } from "../controllers/clientController";
import { authenticateToken } from "../middleware/auth";

export const createClientRouter = (clientController: ClientController,
apiKeyController: ApiKeyController): Router => {
  const router = Router();

  router.get("/", authenticateToken({ allowAdminUsers: true }),
clientController.getAllClients.bind(clientController));
  router.get("/:id", authenticateToken({ allowAdminUsers: true }),
clientController.getClientById.bind(clientController));
  router.post("/", authenticateToken({ allowAdminUsers: true }),
clientController.createClient.bind(clientController));
  router.put("/:id", authenticateToken({ allowAdminUsers: true }),
clientController.updateClient.bind(clientController));
  router.delete("/:id", authenticateToken({ allowAdminUsers: true }),
clientController.deleteClient.bind(clientController));

  // API key management routes (for client users)
  router.get("/:clientId/api-keys", authenticateToken({
allowClientUsers: true }),
apiKeyController.list.bind(apiKeyController));
  router.post("/:clientId/api-keys", authenticateToken({
allowClientUsers: true }),
apiKeyController.create.bind(apiKeyController));
  router.delete("/:clientId/api-keys/:id", authenticateToken({
allowClientUsers: true }),
apiKeyController.delete.bind(apiKeyController));

  return router;
};
```

```
import { Router } from "express";
import { SecretController } from "../controllers/secretController";
import { authenticateToken } from "../middleware/auth";
```

```
export const createSecretRouter = (secretController:
SecretController): Router => {
  const router = Router();

  router.get("/", authenticateToken({ allowClientUsers: true,
allowRoboticUsers: true }),
secretController.list.bind(secretController));
  router.get("/:key", authenticateToken({ allowClientUsers: true,
allowRoboticUsers: true }),
secretController.getByKey.bind(secretController));
  router.post("/", authenticateToken({ allowClientUsers: true }),
secretController.upsert.bind(secretController));
  router.delete("/:id", authenticateToken({ allowClientUsers: true }),
secretController.delete.bind(secretController));

  return router;
};



import { Router } from "express";
import { UserController } from "../controllers/userController";
import { authenticateToken } from "../middleware/auth";

export const createUserRouter = (userController: UserController):
Router => {
  const router = Router();

  router.use(authenticateToken({ allowAdminUsers: true }));

  router.get("/", userController.listUsers.bind(userController));
  router.delete("/:id",
userController.deleteUser.bind(userController));

  return router;
};



import { Knex } from "knex";
import jwt from "jsonwebtoken";
import config from "../config/config";

export interface ApiKey {
  id: string;
```

```
  clientId: string;
  name: string;
  createdAt: Date;
  updatedAt: Date;
}

export interface CreateApiKeyDTO {
  clientId: string;
  name: string;
}

export class ApiKeyService {
  constructor(private readonly db: Knex) {}

  async create(data: CreateApiKeyDTO): Promise<{ apiKey: ApiKey;
token: string }> {
    const [apiKey] = await this.db("api_keys")
      .insert({
        client_id: data.clientId,
        name: data.name,
      })
      .returning("*");

    const mappedApiKey = this.mapToApiKey(apiKey);
    const token = this.generateToken(data.clientId, mappedApiKey.id);

    return {
      apiKey: mappedApiKey,
      token,
    };
  }

  async list(clientId: string): Promise<ApiKey[]> {
    const apiKeys = await this.db("api_keys").where({ client_id:
clientId }).orderBy("created_at", "desc");

    return apiKeys.map(this.mapToApiKey);
  }

  async getById(id: string): Promise<ApiKey | null> {
    const apiKey = await this.db<ApiKey>("api_keys").where({ id
}).first();

    return apiKey ? this.mapToApiKey(apiKey) : null;
```

```
  }

  async delete(id: string, clientId: string): Promise<void> {
    const deleted = await this.db("api_keys").where({ id, client_id:
clientId }).delete();

    if (!deleted) {
      throw new Error("API key not found");
    }
  }

  private mapToApiKey(dbApiKey: any): ApiKey {
    return {
      id: dbApiKey.id,
      clientId: dbApiKey.client_id,
      name: dbApiKey.name,
      createdAt: dbApiKey.created_at,
      updatedAt: dbApiKey.updated_at,
    };
  }

  private generateToken(clientId: string, keyId: string): string {
    return jwt.sign(
      {
        clientId,
        keyId,
        type: "api_key",
      },
      config.jwt.secret,
      { expiresIn: config.apiKey.expiresIn },
    );
  }
}




import { Knex } from "knex";
import jwt from "jsonwebtoken";
import bcrypt from "bcrypt";
import config from "../config/config";
import { BadRequestError } from "../utils/errors";

interface RegisterUserDto {
```

```
  email: string;
  password: string;
  clientId?: string | null; // if null, user will be admin
}

export class AuthService {
  constructor(private readonly db: Knex) {}

  async login(email: string, password: string): Promise<string> {
    const user = await this.db("users").where({ email }).first();

    if (!user) {
      throw new BadRequestError("Invalid email or password");
    }

    const isPasswordValid = await bcrypt.compare(password,
user.password_hash);
    if (!isPasswordValid) {
      throw new BadRequestError("Invalid email or password");
    }

    const token = jwt.sign(
      {
        userId: user.id,
        email: user.email,
        clientId: user.client_id,
      },
      config.jwt.secret,
      {
        expiresIn: config.jwt.expiresIn,
      },
    );

    return token;
  }

  async register(data: RegisterUserDto): Promise<string> {
    // Check if user already exists
    const existingUser = await this.db("users").where({ email:
data.email }).first();

    if (existingUser) {
      throw new BadRequestError("Email already registered");
    }
```

```
    // If clientId is provided, verify it exists
    if (data.clientId) {
      const clientExists = await this.db("clients").where({ id:
data.clientId }).first();

      if (!clientExists) {
        throw new BadRequestError("Invalid client ID");
      }
    }

    const passwordHash = await bcrypt.hash(data.password, 10);

    const [user] = await this.db("users")
      .insert({
        email: data.email,
        password_hash: passwordHash,
        client_id: data.clientId, // null for admin, client_id for
client user
      })
      .returning("*");

    const token = jwt.sign(
      {
        userId: user.id,
        email: user.email,
        clientId: user.client_id,
      },
      config.jwt.secret,
      {
        expiresIn: config.jwt.expiresIn,
      },
    );

    return token;
  }
}



import { Knex } from "knex";
import { Client, CreateClientDto, UpdateClientDto } from
"../types/client";
import { BadRequestError } from "../utils/errors";
```

```
export class ClientService {
  constructor(private readonly db: Knex) {}

  async create(data: CreateClientDto): Promise<Client> {
    if (!data.name) {
      throw new BadRequestError("Client name is required");
    }

    const [client] = await this.db<Client>("clients")
      .insert({
        name: data.name,
        description: data.description,
      })
      .returning("*");

    return client;
  }

  async findAll(): Promise<Client[]> {
    return await this.db<Client>("clients").select("*");
  }

  async findById(id: string): Promise<Client | null> {
    const client = await this.db<Client>("clients").where({ id
}).first();

    return client || null;
  }

  async update(id: string, data: UpdateClientDto): Promise<Client |
null> {
    const [client] = await this.db<Client>("clients")
      .where({ id })
      .update({
        name: data.name,
        description: data.description,
        updated_at: this.db.fn.now(),
      })
      .returning("*");

    return client || null;
  }
```

```
  async delete(id: string): Promise<boolean> {
    const count = await this.db<Client>("clients").where({ id
}).delete();

    return count > 0;
  }
}



import { Knex } from "knex";
import { encryptionService } from "./encryptionService";
import { NotFoundError } from "../utils/errors";

export interface Secret {
  id: string;
  clientId: string;
  key: string;
  value: string | null;
  createdAt: Date;
  updatedAt: Date;
}

export interface CreateSecretDTO {
  clientId: string;
  key: string;
  value: string;
}

export interface SecretFilters {
  clientId: string;
  key?: string;
}

export interface UpsertSecretDTO {
  id?: string;
  key: string;
  value: string;
}

export class SecretService {
  constructor(private readonly db: Knex) {}

  async list(filters: SecretFilters): Promise<Secret[]> {
```

```
    const query = this.db("secrets").where("client_id",
filters.clientId).orderBy("created_at", "desc");

    if (filters.key) {
      query.whereILike("key", `%${filters.key}%`);
    }

    const secrets = await query;
    return secrets.map((secret) => this.mapToSecret(secret));
  }

  async delete(id: string, clientId: string): Promise<void> {
    const deleted = await this.db("secrets").where({ id, client_id:
clientId }).delete();

    if (!deleted) {
      throw new Error("Secret not found");
    }
  }

  async upsert(clientId: string, data: UpsertSecretDTO):
Promise<Secret> {
    const encryptedValue = encryptionService.encrypt(data.value);

    if (data.id) {
      const [updated] = await this.db("secrets")
        .where({
          id: data.id,
          client_id: clientId,
        })
        .update({
          key: data.key,
          value: encryptedValue,
          updated_at: new Date(),
        })
        .returning("*");

      if (updated) {
        return this.mapToSecret(updated);
      }
    }

    // If no id provided or update failed (record not found), create
new
```

```
    const [created] = await this.db("secrets")
      .insert({
        client_id: clientId,
        key: data.key,
        value: encryptedValue,
      })
      .returning("*");

    return this.mapToSecret(created);
  }

  async getByKey(key: string, clientId: string): Promise<Secret> {
    const secret = await this.db("secrets")
      .where({
        key,
        client_id: clientId,
      })
      .first();

    if (!secret) {
      throw new NotFoundError("Secret not found");
    }

    return this.mapToSecret(secret, false);
  }

  private mapToSecret(dbSecret: any, omitSensitiveData = true): Secret
{
    return {
      id: dbSecret.id,
      clientId: dbSecret.client_id,
      key: dbSecret.key,
      value: omitSensitiveData ? null :
encryptionService.decrypt(dbSecret.value),
      createdAt: dbSecret.created_at,
      updatedAt: dbSecret.updated_at,
    };
  }
}



import { Knex } from "knex";
import { BadRequestError, NotFoundError } from "../utils/errors";
```

```
export interface User {
  id: string;
  email: string;
  client_id: string | null;
  created_at: Date;
  updated_at: Date;
}

export class UserService {
  constructor(private readonly db: Knex) {}

  async findByClientId(clientId: string | null): Promise<User[]> {
    return await this.db<User>("users").select("id", "email",
"client_id", "created_at", "updated_at").where({ client_id: clientId
}).orderBy("email");
  }

  async findAll(): Promise<User[]> {
    return await this.db<User>("users").select("id", "email",
"client_id", "created_at", "updated_at").orderBy("email");
  }

  async delete(userId: string): Promise<void> {
    // Check if user exists
    const user = await this.db<User>("users").where({ id: userId
}).first();

    if (!user) {
      throw new NotFoundError("User not found");
    }

    // Don't allow deleting the last admin
    if (!user.client_id) {
      const adminCount = await this.db<User>("users").where({
client_id: null }).count("id as count").first();

      // if (adminCount && Number(adminCount.count) <= 1) {
      //   throw new BadRequestError("Cannot delete the last admin
user");
      // }
    }

    await this.db("users").where({ id: userId }).delete();
```

```
  }
}

import cors from "cors";
import express from "express";
import { ApiKeyController } from "./controllers/apiKeyController";
import { AuthController } from "./controllers/authController";
import { ClientController } from "./controllers/clientController";
import { SecretController } from "./controllers/secretController";
import { UserController } from "./controllers/userController";
import { db } from "./db";
import { errorHandler } from "./middleware/errorHandler";
import { createAuthRouter } from "./routes/authRoutes";
import { createClientRouter } from "./routes/clientRoutes";
import { createSecretRouter } from "./routes/secretRoutes";
import { createUserRouter } from "./routes/userRoutes";
import { ApiKeyService } from "./services/apiKeyService";
import { AuthService } from "./services/authService";
import { ClientService } from "./services/clientService";
import { SecretService } from "./services/secretService";
import { UserService } from "./services/userService";

const app = express();

app.use(cors());
app.use(express.json());

// Initialize services and controllers
const clientService = new ClientService(db);
const clientController = new ClientController(clientService);
const authService = new AuthService(db);
const authController = new AuthController(authService);
const userService = new UserService(db);
const userController = new UserController(userService);
const secretService = new SecretService(db);
const secretController = new SecretController(secretService);
const apiKeyService = new ApiKeyService(db);
const apiKeyController = new ApiKeyController(apiKeyService);

// Register routes
app.use("/auth", createAuthRouter(authController));
app.use("/clients", createClientRouter(clientController,
apiKeyController));
app.use("/users", createUserRouter(userController));
```

```
app.use("/secrets", createSecretRouter(secretController));

app.use(errorHandler);

export default app;


import app from "./app";

const PORT = process.env.PORT || 3000;

app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});


import { useState, useEffect } from "react";
import { useParams } from "react-router-dom";
import {
  Container,
  Typography,
  Box,
  Paper,
  Table,
  TableBody,
  TableCell,
  TableContainer,
  TableHead,
  TableRow,
  Button,
  IconButton,
} from "@mui/material";
import DeleteIcon from "@mui/icons-material/Delete";
import { formatDate } from "../utils/dateUtils";
import { Client, clientService } from "../services/clientService";
import { User, userService, UserCreate } from
"../services/userService";
import { UserForm } from "../components/UserForm";

export const ClientDetails = () => {
  const { clientId } = useParams<{ clientId: string }>();
  const [client, setClient] = useState<Client | null>(null);
  const [users, setUsers] = useState<User[]>([]);
  const [openForm, setOpenForm] = useState(false);
```

```
  const [loading, setLoading] = useState(true);

  const loadClient = async () => {
    if (!clientId) return;
    try {
      const data = await clientService.getClient(clientId);
      setClient(data);
    } catch (error) {
      console.error("Failed to load client:", error);
    }
  };

  const loadUsers = async () => {
    if (!clientId) return;
    try {
      const data = await userService.getUsersByClient(clientId);
      setUsers(data);
    } catch (error) {
      console.error("Failed to load users:", error);
    } finally {
      setLoading(false);
    }
  };

  useEffect(() => {
    loadClient();
    loadUsers();
  }, [clientId]);

  const handleCreateUser = async (userData: UserCreate) => {
    try {
      await userService.createUser(userData);
      setOpenForm(false);
      loadUsers();
    } catch (error) {
      console.error("Failed to create user:", error);
    }
  };

  const handleDeleteUser = async (userId: string) => {
    if (window.confirm("Are you sure you want to delete this user?"))
{
      try {
        await userService.deleteUser(userId);
```

```
      loadUsers();
    } catch (error) {
      console.error("Failed to delete user:", error);
    }
  }
};

if (!client) {
  return null;
}

return (
  <Container maxWidth="lg">
    <Typography variant="h4" gutterBottom>
      {client.name}
    </Typography>
    {client.description && (
      <Typography variant="body1" color="text.secondary" paragraph>
        {client.description}
      </Typography>
    )}

    <Box
      display="flex"
      justifyContent="space-between"
      alignItems="center"
      mb={3}
    >
      <Typography variant="h5">Users</Typography>
      <Button variant="contained" onClick={() => setOpenForm(true)}>
        + Add
      </Button>
    </Box>

    <TableContainer component={Paper}>
      <Table>
        <TableHead>
          <TableRow>
            <TableCell>Email</TableCell>
            <TableCell>Created At</TableCell>
            <TableCell align="right">Actions</TableCell>
          </TableRow>
        </TableHead>
        <TableBody>
```

```
            {users.map((user) => (
              <TableRow key={user.id}>
                <TableCell>{user.email}</TableCell>
                <TableCell>{formatDate(user.created_at)}</TableCell>
                <TableCell align="right">
                  <IconButton onClick={() =>
handleDeleteUser(user.id)}>
                    <DeleteIcon />
                  </IconButton>
                </TableCell>
              </TableRow>
            ))}
          </TableBody>
        </Table>
      </TableContainer>

      {clientId && (
        <UserForm
          open={openForm}
          onClose={() => setOpenForm(false)}
          onSubmit={handleCreateUser}
          clientId={clientId}
        />
      )}
    </Container>
  );
};
```

```
import { useState, useEffect } from "react";
import {
  Container,
  Paper,
  Table,
  TableBody,
  TableCell,
  TableContainer,
  TableHead,
  TableRow,
  Button,
```

```
  IconButton,
  Typography,
  Box,
} from "@mui/material";
import EditIcon from "@mui/icons-material/Edit";
import DeleteIcon from "@mui/icons-material/Delete";
import {
  Client,
  clientService,
  ClientCreateUpdate,
} from "../services/clientService";
import { ClientForm } from "../components/ClientForm";
import { formatDate } from "../utils/dateUtils";
import { useNavigate } from "react-router-dom";

export const Clients = () => {
  const [clients, setClients] = useState<Client[]>([]);
  const [openForm, setOpenForm] = useState(false);
  const [selectedClient, setSelectedClient] = useState<Client |
undefined>();
  const [loading, setLoading] = useState(true);
  const navigate = useNavigate();

  const loadClients = async () => {
    try {
      const data = await clientService.getClients();
      setClients(data);
    } catch (error) {
      console.error("Failed to load clients:", error);
    } finally {
      setLoading(false);
    }
  };

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

  const handleCreate = async (clientData: ClientCreateUpdate) => {
    try {
      await clientService.createClient(clientData);
      setOpenForm(false);
      loadClients();
    } catch (error) {
```

```
      console.error("Failed to create client:", error);
    }
  };

  const handleUpdate = async (clientData: ClientCreateUpdate) => {
    if (!selectedClient) return;
    try {
      await clientService.updateClient(selectedClient.id, clientData);
      setOpenForm(false);
      setSelectedClient(undefined);
      loadClients();
    } catch (error) {
      console.error("Failed to update client:", error);
    }
  };

  const handleDelete = async (id: string) => {
    if (window.confirm("Are you sure you want to delete this
client?")) {
      try {
        await clientService.deleteClient(id);
        loadClients();
      } catch (error) {
        console.error("Failed to delete client:", error);
      }
    }
  };

  const handleEdit = (client: Client) => {
    setSelectedClient(client);
    setOpenForm(true);
  };

  const handleRowClick = (clientId: string) => {
    navigate(`/clients/${clientId}`);
  };

  return (
    <Container maxWidth="lg">
      <Box
        display="flex"
        justifyContent="space-between"
        alignItems="center"
        mb={3}
```

```
      >
        <Typography variant="h4">Clients</Typography>
        <Button
          variant="contained"
          onClick={() => {
            setSelectedClient(undefined);
            setOpenForm(true);
          }}
        >
          + Add
        </Button>
      </Box>

      <TableContainer component={Paper}>
        <Table>
          <TableHead>
            <TableRow>
              <TableCell>Name</TableCell>
              <TableCell>Description</TableCell>
              <TableCell>Created At</TableCell>
              <TableCell align="right">Actions</TableCell>
            </TableRow>
          </TableHead>
          <TableBody>
            {clients.map((client) => (
              <TableRow
                key={client.id}
                onClick={() => handleRowClick(client.id)}
                sx={{
                  cursor: "pointer",
                  "&:hover": { backgroundColor: "rgba(0, 0, 0, 0.04)"
},
                }}
              >
                <TableCell>{client.name}</TableCell>
                <TableCell>{client.description || "-"}</TableCell>
                <TableCell>{formatDate(client.created_at)}</TableCell>
                <TableCell align="right">
                  <IconButton
                    onClick={(e) => {
                      e.stopPropagation();
                      handleEdit(client);
                    }}
                  >
```

```
                    <EditIcon />
                  </IconButton>
                  <IconButton
                    onClick={(e) => {
                      e.stopPropagation();
                      handleDelete(client.id);
                    }}
                  >
                    <DeleteIcon />
                  </IconButton>
                </TableCell>
              </TableRow>
            ))}
          </TableBody>
        </Table>
      </TableContainer>

      <ClientForm
        open={openForm}
        onClose={() => {
          setOpenForm(false);
          setSelectedClient(undefined);
        }}
        onSubmit={selectedClient ? handleUpdate : handleCreate}
        initialData={selectedClient}
        title={selectedClient ? "Edit Client" : "Add New Client"}
      />
    </Container>
  );
};
```

```
import { useState, useEffect, useCallback } from "react";
import {
  Container,
  Paper,
  Table,
  TableBody,
  TableCell,
  TableContainer,
```

```
  TableHead,
  TableRow,
  Button,
  IconButton,
  Typography,
  Box,
  TextField,
  Tab,
  Tabs,
} from "@mui/material";
import DeleteIcon from "@mui/icons-material/Delete";
import { format } from "date-fns";
import debounce from "lodash/debounce";
import { Secret, secretService, SecretCreate } from
"../services/secretService";
import { SecretForm } from "../components/SecretForm";
import { formatDate } from "../utils/dateUtils";
import { ApiKey, apiKeyService, ApiKeyCreate } from
"../services/apiKeyService";
import { ApiKeyForm } from "../components/ApiKeyForm";
import { ApiKeyTokenDialog } from "../components/ApiKeyTokenDialog";
import { authService } from "../services/authService";

interface TabPanelProps {
  children?: React.ReactNode;
  index: number;
  value: number;
}

const TabPanel = (props: TabPanelProps) => {
  const { children, value, index, ...other } = props;

  return (
    <div
      role="tabpanel"
      hidden={value !== index}
      id={`simple-tabpanel-${index}`}
      {...other}
    >
      {value === index && <Box sx={{ p: 3 }}>{children}</Box>}
    </div>
  );
};
```

```
export const Home = () => {
  const [tabValue, setTabValue] = useState(0);
  const [secrets, setSecrets] = useState<Secret[]>([]);
  const [searchKey, setSearchKey] = useState("");
  const [openForm, setOpenForm] = useState(false);
  const [loading, setLoading] = useState(true);
  const [apiKeys, setApiKeys] = useState<ApiKey[]>([]);
  const [openApiKeyForm, setOpenApiKeyForm] = useState(false);
  const [newApiKeyToken, setNewApiKeyToken] = useState<string |
null>(null);

  const loadSecrets = async (key?: string) => {
    try {
      const data = await secretService.getSecrets(key);
      setSecrets(data);
    } catch (error) {
      console.error("Failed to load secrets:", error);
    } finally {
      setLoading(false);
    }
  };

  // Debounced search function
  const debouncedSearch = useCallback(
    debounce((searchText: string) => {
      loadSecrets(searchText || undefined);
    }, 200),
    [],
  );

  useEffect(() => {
    debouncedSearch(searchKey);
    return () => {
      debouncedSearch.cancel();
    };
  }, [searchKey, debouncedSearch]);

  const handleCreateSecret = async (secretData: SecretCreate) => {
    try {
      await secretService.createSecret(secretData);
      setOpenForm(false);
      loadSecrets(searchKey || undefined);
    } catch (error) {
      console.error("Failed to create secret:", error);
```

```
    }
  };

  const handleDeleteSecret = async (id: string) => {
    if (window.confirm("Are you sure you want to delete this
secret?")) {
      try {
        await secretService.deleteSecret(id);
        loadSecrets(searchKey || undefined);
      } catch (error) {
        console.error("Failed to delete secret:", error);
      }
    }
  };

  const loadApiKeys = async () => {
    try {
      const clientId = authService.getClientId();
      if (!clientId) return;
      const data = await apiKeyService.getApiKeys(clientId);
      setApiKeys(data);
    } catch (error) {
      console.error("Failed to load API keys:", error);
    }
  };

  useEffect(() => {
    if (tabValue === 1) {
      loadApiKeys();
    }
  }, [tabValue]);

  const handleCreateApiKey = async (data: ApiKeyCreate) => {
    try {
      const clientId = authService.getClientId();
      if (!clientId) return;
      const response = await apiKeyService.createApiKey(clientId,
data);
      setOpenApiKeyForm(false);
      setNewApiKeyToken(response.token);
      loadApiKeys();
    } catch (error) {
      console.error("Failed to create API key:", error);
    }
```

```
  };

  const handleDeleteApiKey = async (id: string) => {
    if (window.confirm("Are you sure you want to delete this API
key?")) {
      try {
        const clientId = authService.getClientId();
        if (!clientId) return;
        await apiKeyService.deleteApiKey(clientId, id);
        loadApiKeys();
      } catch (error) {
        console.error("Failed to delete API key:", error);
      }
    }
  };

  return (
    <Container maxWidth="lg">
      <Box sx={{ borderBottom: 1, borderColor: "divider", mb: 3 }}>
        <Tabs
          value={tabValue}
          onChange={(_, newValue) => setTabValue(newValue)}
        >
          <Tab label="Secrets" />
          <Tab label="API Keys" />
        </Tabs>
      </Box>

      <TabPanel value={tabValue} index={0}>
        <Box
          display="flex"
          justifyContent="space-between"
          alignItems="center"
          mb={3}
        >
          <TextField
            placeholder="Search secrets..."
            value={searchKey}
            onChange={(e) => setSearchKey(e.target.value)}
            size="small"
            sx={{ width: 300 }}
          />
          <Button variant="contained" onClick={() =>
setOpenForm(true)}>
```

```
            + Add
          </Button>
        </Box>

        <TableContainer component={Paper}>
          <Table>
            <TableHead>
              <TableRow>
                <TableCell>Key</TableCell>
                <TableCell>Created At</TableCell>
                <TableCell align="right">Actions</TableCell>
              </TableRow>
            </TableHead>
            <TableBody>
              {secrets.map((secret) => (
                <TableRow key={secret.id}>
                  <TableCell>{secret.key}</TableCell>
                  <TableCell>{formatDate(secret.createdAt)}</TableCell
>
                  <TableCell align="right">
                    <IconButton onClick={() =>
handleDeleteSecret(secret.id)}>
                      <DeleteIcon />
                    </IconButton>
                  </TableCell>
                </TableRow>
              ))}
            </TableBody>
          </Table>
        </TableContainer>

        <SecretForm
          open={openForm}
          onClose={() => setOpenForm(false)}
          onSubmit={handleCreateSecret}
        />
      </TabPanel>

      <TabPanel value={tabValue} index={1}>
        <Box
          display="flex"
          justifyContent="space-between"
          alignItems="center"
          mb={3}
```

```
        >
          <Typography variant="h5"></Typography>
          <Button variant="contained" onClick={() =>
setOpenApiKeyForm(true)}>
            + Add
          </Button>
        </Box>

        <TableContainer component={Paper}>
          <Table>
            <TableHead>
              <TableRow>
                <TableCell>Name</TableCell>
                <TableCell>Created At</TableCell>
                <TableCell align="right">Actions</TableCell>
              </TableRow>
            </TableHead>
            <TableBody>
              {apiKeys.map((apiKey) => (
                <TableRow key={apiKey.id}>
                  <TableCell>{apiKey.name}</TableCell>
                  <TableCell>{formatDate(apiKey.createdAt)}</TableCell
>
                  <TableCell align="right">
                    <IconButton onClick={() =>
handleDeleteApiKey(apiKey.id)}>
                      <DeleteIcon />
                    </IconButton>
                  </TableCell>
                </TableRow>
              ))}
            </TableBody>
          </Table>
        </TableContainer>

        <ApiKeyForm
          open={openApiKeyForm}
          onClose={() => setOpenApiKeyForm(false)}
          onSubmit={handleCreateApiKey}
        />

        <ApiKeyTokenDialog
          open={!!newApiKeyToken}
          onClose={() => setNewApiKeyToken(null)}
```

```
          token={newApiKeyToken || ""}
        />
      </TabPanel>
    </Container>
  );
};
```

```
import { useState } from "react";
import { useNavigate, useLocation } from "react-router-dom";
import {
  Container,
  Paper,
  TextField,
  Button,
  Typography,
  Box,
  Alert,
} from "@mui/material";
import { authService } from "../services/authService";

export const Login = () => {
  const navigate = useNavigate();
  const location = useLocation();
  const [email, setEmail] = useState("");
  const [password, setPassword] = useState("");
  const [error, setError] = useState("");

  const from = location.state?.from?.pathname || "/";

  const handleSubmit = async (e: React.FormEvent) => {
    e.preventDefault();
    try {
      await authService.login({ email, password });
      navigate(from, { replace: true });
    } catch (err) {
      setError("Invalid email or password");
    }
  };

  return (
    <Container component="main" maxWidth="xs">
```

```
        <Box
          sx={{
            marginTop: 8,
            display: "flex",
            flexDirection: "column",
            alignItems: "center",
          }}
        >
          <Paper elevation={3} sx={{ padding: 4, width: "100%" }}>
            <Typography component="h1" variant="h5" align="center">
              Sign in
            </Typography>
            {error && (
              <Alert severity="error" sx={{ mt: 2 }}>
                {error}
              </Alert>
            )}
            <Box component="form" onSubmit={handleSubmit} sx={{ mt: 1
}}>
              <TextField
                margin="normal"
                required
                fullWidth
                id="email"
                label="Email Address"
                name="email"
                autoComplete="email"
                autoFocus
                value={email}
                onChange={(e) => setEmail(e.target.value)}
              />
              <TextField
                margin="normal"
                required
                fullWidth
                name="password"
                label="Password"
                type="password"
                id="password"
                autoComplete="current-password"
                value={password}
                onChange={(e) => setPassword(e.target.value)}
              />
              <Button
```

```
              type="submit"
              fullWidth
              variant="contained"
              sx={{ mt: 3, mb: 2 }}
            >
              Sign In
            </Button>
          </Box>
        </Paper>
      </Box>
    </Container>
  );
};
```