**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**
**ВОЛИНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ**
**ІМЕНІ ЛЕСІ УКРАЇНКИ**
**Кафедра комп'ютерних наук та кібербезпеки**

На правах рукопису

**ПАСТУШОК СТАНІСЛАВ ЮРІЙОВИЧ**

# АНАЛІЗ МЕТОДІВ ШИФРУВАННЯ НА ПРИКЛАДІ РОЗРОБКИ МЕНЕДЖЕРА ПАРОЛІВ

Спеціальність: 122 Комп'ютерні науки
Освітньо-професійна програма: Комп'ютерні науки та інформаційні технології
Кваліфікаційна робота на здобуття освітнього ступеня «магістр»

Науковий керівник:
МАМЧИЧ ТЕТЯНА ІВАНІВНА,
кандидат фізико-математичних наук, доцент
кафедри комп'ютерних наук та кібербезпеки

РЕКОМЕНДОВАНО ДО ЗАХИСТУ
Протокол №_____
засідання кафедри комп'ютерних наук
та кібербезпеки
від _____ 2024 р.
Завідувач кафедри

(_____) Гришанович Т. О.

ЛУЦЬК - 2024

# ЗМІСТ

# ВСТУП

**Актуальність теми.** В сучасному цифровому світі критичними питаннями все ще залишаються питання створення, зберігання та використання паролів. Кожного року обсяг персональних даних, котрі передаються через всесвітню мережу Інтернет, зростає в геометричній прогресії. Разом із цим, зростає кількість кібератак, які спрямовані на заволодіння конфіденційною інформацією. Паролі слугують першою лінією захисту, тому забезпечення їхньої безпеки має вирішальне значення для захисту приватних даних користувачів.

Великою проблемою є те, що багато пересічних користувачів використовують слабкі або повторювані паролі, що полегшує їх компрометацію. Тому надійні методи шифрування та хешування паролів відіграють неабияку роль у гарантуванні безпеки персональних даних.

Майже усі активні користувачі комп’ютером хоч раз забували свій пароль і розуміють що в деяких випадках відновлення власного паролю є неможливим завданням. Люди вигадали велику кількість шляхів вирішення проблем збереження і керування власними паролями і іншими конфіденційними даними, наприклад: файл Excel на робочому столі, або аркуш паперу з усіма паролями. Раніше це могло бути розумним рішенням, проте з розвитком технологій ваша цифрова гігієна також повинна розвиватися. Кіберзлочинці зараз більш небезпечні, ніж будь-коли. Найшвидший спосіб зламати вашу систему - це підібрати пароль, який легко вгадати.

**Мета роботи** полягає в розробці менеджера паролів.

Для досягнення вище вказаної мети, необхідно реалізувати наступні **завдання**:

- аналіз алгоритмів та програмних рішень для менеджерів паролів;
- розробка зручного та простого інтерфейсу для користувачів вебсервісу;
- реалізація back end частини;
- реалізація логіки на front end частині.

**Об’єктом дослідження** є програмні рішення для систем управління паролями.

**Предмет дослідження** – процес реалізації технологій керування паролями.

**Апробація:** Тези на тему: “Розробка менеджера паролів за допомогою ReactJs та NestJs” були опубліковані на “І Міжнародній науково-практичній конференції 2024 “Проблеми комп’ютерних наук, програмного моделювання та безпеки цифрових систем”” у червні 2024 року.

# РОЗДІЛ 1.
# ХАРАКТЕРИСТИКА ВЕБДОДАТКІВ ТА ДЕЯКИХ ТЕХНОЛОГІЙ ЇХ РОЗРОБКИ

## 1.1 Загальний огляд вебдодатків

### 1.1.1 Визначення вебдодатку

Вебдодаток, або, як його ще називають, вебпрограма – це інтерактивний програмний продукт, який не потребує інсталяції, а доступний через браузер користувача. Він зберігається на віддаленому сервері, а це означає, що користувачі можуть відкрити його з будь-якого пристрою.

Унікальність вебдодатків полягає в тому, що вони можуть бути різних форм. Залежно від потреб людей або організацій, вони можуть включати різні функції та можливості, щоб полегшити виконання певних завдань і надати користувачам кращий досвід. Незважаючи на це, кілька компонентів є спільними для всіх них:

- клієнт (або веббраузер) – це місце, де відбувається обмін даними між клієнтом і сервером;
- сервер – пристрій або програма, яка відповідає на запит клієнта, надаючи необхідні дані;
- протоколи – це набір стандартизованих правил, які визначають спосіб передачі даних від вебсервера до обчислювальних пристроїв. Два найважливіші протоколи – це Інтернет-протокол (IP) і протокол управління передачею (TCP) .

Принцип роботи вебдодатків полягає в наступному:

- користувач надсилає запит на сервер. Це можна зробити за допомогою веббраузерів або інших клієнтських програм, підключених до Інтернету;
- сервер отримує запит і дає вказівку серверу вебдодатків обробити його;
- сервер додатків робить те, що запитується, і надсилає назад результат;
- сервер відображає результат на екрані користувача.

### 1.1.2 Переваги вебдодатків

*Крос-платформенна сумісність.* Однією з найбільших переваг вебдодатків є те, що вони сумісні практично з будь-яким сучасним пристроєм. Стаціонарні комп'ютери та ноутбуки під управлінням Windows, macOS або Linux, мобільні телефони та планшети з iOS або Android на борту – для вебдодатків тип і операційна система пристрою не мають значення.

*Широка доступність.* Якщо ви розробляєте вебдодаток, вам не потрібно турбуватися про публікацію свого додатку в магазинах додатків і вживати додаткових заходів, щоб відповідати вимогам магазину додатків. Користувачам, у свою чергу, не потрібно встановлювати додаток, що може бути проблемою для пристроїв з обмеженим обсягом пам'яті, або регулярно оновлювати додаток, щоб він працював належним чином.

*Кастомізація та масштабованість.* Змінна природа вебдодатків дозволяє розробникам легко вносити необхідні зміни, не переробляючи всю архітектуру програми. Крім того, вебдодатки – ідеальний варіант, якщо ви плануєте масштабувати свій продукт у найближчому майбутньому і хочете зробити це найбільш економічно ефективним способом.

*Нижча вартість розробки.* Вартість розробки вебдодатку може бути вищою, ніж вартість розробки традиційного вебсайту, але її навіть не можна порівняти з типовим бюджетом на розробку нативного додатку. Крім того, нативний додаток потребує постійної підтримки та оновлень, щоб залишатися конкурентоспроможним, тоді як вебдодаток, як правило, довговічніший.

*Хороша безпека.* З традиційними десктопними або мобільними додатками, які зберігають більшу частину даних на пристрої користувача, занепокоєння щодо безпеки є цілком виправданим. Якщо пристрій загубиться або його вкрадуть, незнайомець може отримати доступ до конфіденційної інформації, не призначеної для публічного перегляду. Це не стосується вебдодатків, які надійно зберігають дані в хмарі.

*Швидша розробка та розгортання.* Фреймворки та інструменти, що використовуються для розробки вебдодатків, дозволяють скоротити цикл

розробки порівняно з нативними додатками. Це може бути конкурентною перевагою для компаній, які прагнуть швидше виводити свої продукти на ринок і швидко реагувати на ринкові зміни.

*Легка інтеграція зі сторонніми сервісами.* Вебдодатки можна легко інтегрувати з різноманітними сторонніми сервісами, API та інструментами, що є перевагою, яку розробники можуть використовувати для розширення функціональності своїх додатків, заощаджуючи при цьому час і зусилля на розробку.

*Хороший користувацький досвід.* І останнє, але, безумовно, не менш важливе, вебдодатки забезпечують позитивний досвід для користувачів і сприяють підвищенню впізнаваності бренду. Незалежно від того, яким пристроєм вони користуються, вони бачитимуть однаковий інтерфейс і функції, що дасть їм відчуття узгодженості та сприятиме лояльності.

### 1.1.3 Порівняння вебдодатків з іншими видами додатків

Вебдодатки і вебсайти дуже часто плутають, однак це не одне й те саме, хоча відмінності між ними можуть здаватися досить розмитими.

Вебсайт – це статична сторінка, яка відображає інформацію у вигляді тексту, фотографій, відео, аудіофайлів або інших типів медіаконтенту. Вебсайти не призначені для інтерактивних дій. Вміст на вебсайтах не оновлюється динамічно, і єдиний спосіб відобразити зміни на сторінці – це перезавантажити її.

З іншого боку, вебдодаток – це динамічний програмний продукт. Він може виглядати точно так само, як вебсайт, але завжди матиме принаймні один інтерактивний компонент. Це означає, що ви можете взаємодіяти з додатком і обмінюватися з ним інформацією без перезавантаження сторінки. Вебдодаток може бути створений для виконання одного конкретного завдання, як, наприклад, Gmail або Google Docs, або ж мати більш розширену функціональність, як Amazon або Netflix.

Вебдодатки, а саме їхні мобільні версії та мобільні додатки часто плутають, що не дивно, зважаючи на їхню схожість. Обидва вони призначені для надання користувачам легкого доступу до інформації та послуг, і обидва можуть використовуватися на мобільних пристроях. Тим не менш, існує ряд відмінностей, які роблять їх кардинально різними.

Вебдодаток призначений для доступу з браузера, і не має значення, яку платформу ви використовуєте. Хороший вебдодаток буде однаково добре працювати на пристроях iOS і Android, а також на стаціонарних комп'ютерах. Однак цього не можна очікувати від мобільного додатку. Мобільний додаток створюється спеціально для однієї платформи – нативний додаток для iOS можна використовувати лише на пристроях iOS. Крім того, мобільний додаток разом з даними, які він містить, прив'язаний до конкретного пристрою, а це означає, що для роботи з ним вам потрібно буде спочатку завантажити його, щоб мати можливість працювати з ним.

Вебдодатки проти гібридних та нативних додатків. Було би неправильно говорити про відмінності між різними типами додатків, без поглиблення у їх відмінності, зрештою кожен підхід до розробки додатків має свої переваги, які корисно знати.

Нативні та гібридні додатки - це два типи мобільних додатків. Нативні додатки мають назву, яка говорить сама за себе. Вони створені для однієї конкретної платформи, будь то iOS, Android або Windows, і не можуть бути використані на іншій платформі. Гібридні додатки, в свою чергу, поєднують в собі функції веб і нативних додатків. Це найкращий тип додатків для розробників, оскільки він є гнучким і може бути використаний для різних платформ і систем з мінімальними змінами в скрипті. Instagram - один з найкращих прикладів гібридних додатків.[1]

### 1.1.4 Класифікація вебдодатків

Існують різні підходи до класифікації вебдодатків. Можна класифікувати їх за призначенням, архітектурою та іншими характеристиками. Розглянемо декілька видів вебдодатків.

*Статичні вебдодатки* – це додатки, де користувачі отримують необхідні дані у своєму браузері без використання сервера. Можливо, статичні вебдодатки не дають розробникам стільки можливостей, як інші типи вебдодатків, але вони мають ряд переваг, які роблять їх важливими, наприклад, легка і швидка розробка, висока гнучкість, масштабованість без особливих зусиль і легка архітектура.

*Динамічні вебдодатки* – генерують дані в режимі реального часу на основі запиту кінцевого користувача і відповіді сервера. Інформація в динамічному вебдодатку постійно оновлюється, надаючи розробникам простий спосіб надати клієнтам те, що вони хочуть. Динамічний вебдодаток може потребувати більше часу та зусиль на розробку та підтримку порівняно зі статичним, але його універсальність робить його вартим того.

*Односторінкові додатки* – надають контент за допомогою лише однієї вебсторінки, яку не потрібно перезавантажувати для відображення нової інформації. Односторінкові додатки покладаються на браузер, а не на сервер для створення та відображення даних, що робить їх простими та швидкими для завантаження. Незважаючи на свою, здавалося б, базову архітектуру, SPA надають розробникам і власникам продуктів практично необмежені можливості і постійно вдосконалюються.

*Прогресивні вебдодатки.* Так само, як і односторінковий додаток, прогресивний вебдодаток працює з однієї сторінки без необхідності перезавантажувати його для отримання нової інформації. Це, а також той факт, що SPA і PWA часто створюються з використанням одних і тих же мов і фреймворків, змушує багатьох людей думати, що прогресивний вебдодаток – це просто більш загальна версія односторінкового додатка. Однак це не зовсім так,

насправді PWA більш функціональні, ніж SPA, і забезпечують кращий користувацький досвід, хоча їх розробка коштує дорожче. Крім того, прогресивний вебдодаток можна запускати без підключення до Інтернету, використовуючи кеш, чого не може зробити односторінковий додаток.

## 1.2 Протоколи та моделі зв'язку у вебдодатках

### 1.2.1 OSI модель

Модель взаємодії відкритих систем (OSI) – це теоретична, 7-рівнева еталонна модель, яка намагається концептуалізувати, як повинен працювати Інтернет, однак, простіша модель TCP/IP відображає те, як насправді працює сучасний Інтернет. Модель OSI представляє ідеальний стан роботи мереж, і багато інтернет-вендорів чинили опір, коли вона була запропонована разом з набором протоколів OSI, оскільки вважали її надто складною, ідеалістичною і складною для впровадження. Однак, незважаючи на свої недоліки, модель OSI є чудовою концептуальною моделлю для розуміння технологічних основ інтернету. [2]

Прикладний рівень (application layer) мережевої моделі OSI відповідає за надання послуг кінцевим користувачам, таких як електронна пошта, передача файлів і перегляд вебсторінок. Цей рівень безпосередньо взаємодіє з користувачем та його додатками і є найвищим рівнем моделі OSI. Його основна функція полягає в управлінні зв'язком між користувацьким додатком і нижчими рівнями моделі OSI. Цей рівень забезпечує різноманітні протоколи, які допомагають програмам взаємодіяти з іншими програмами, що працюють на різних машинах. Деякі з найпоширеніших протоколів цього рівня включають HTTP, FTP, SMTP і DNS. Загалом, прикладний рівень є критично важливим компонентом моделі OSI, оскільки він надає послуги кінцевим користувачам і гарантує, що їхні комунікаційні потреби будуть задоволені.

Рівень представлення (presentation layer) в мережевій моделі OSI відповідає за синтаксис і семантику даних, якими обмінюються мережеві додатки. Цей

рівень гарантує, що дані, якими обмінюються дві програми, мають формат, зрозумілий обом сторонам. Він відповідає за переклад, шифрування і стиснення даних, гарантуючи, що дані будуть представлені прикладному рівню в стандартизованому і легко зрозумілому форматі. Рівень представлення також відповідає за представлення даних, наприклад, кодування символів і перетворення даних з одного формату в інший. Цей рівень є критично важливим для забезпечення представлення даних у форматі, який може бути оброблений приймаючою програмою, і, таким чином, має вирішальне значення для успішної передачі даних через мережу.

Сеансовий рівень (session layer) мережевої моделі OSI керує сеансами зв'язку між двома мережевими додатками. Цей рівень встановлює, керує і завершує сеанси між двома додатками, дозволяючи їм обмінюватися даними організовано і контрольовано. Сеансовий рівень керує синхронізацією між додатками, гарантуючи, що дані передаються і приймаються впорядковано. Він також керує відновленням сеансу, якщо його перервано через мережеву помилку або збій. Цей рівень є важливим для управління потоком даних між додатками і гарантує, що дані передаються точно і ефективно. Сеансовий рівень відіграє важливу роль в моделі OSI, оскільки він дозволяє програмам взаємодіяти один з одним безперешкодно і надійно.

Транспортний рівень (transport layer) мережевої моделі OSI відповідає за забезпечення надійного зв'язку між додатками. Цей рівень встановлює наскрізні з'єднання між додатками, гарантуючи, що дані передаються правильно і в правильній послідовності. Він також керує управлінням потоком і контролем перевантаження, гарантуючи, що мережа не буде перевантажена даними. Транспортний рівень пропонує два типи протоколів: протокол керування передачею (TCP) і протокол користувацьких дейтаграм (UDP). TCP забезпечує надійний зв'язок, оскільки гарантує, що всі пакети доставляються і приймаються в правильному порядку. UDP, з іншого боку, забезпечує швидший зв'язок, але він не є надійним, оскільки не гарантує доставку пакетів. Транспортний рівень

має вирішальне значення для забезпечення точної та ефективної передачі даних, відіграючи життєво важливу роль в успіху мережевої комунікації.

Мережевий рівень (network layer) у мережевій моделі OSI відповідає за маршрутизацію пакетів даних між різними мережами. Цей рівень забезпечує передачу даних від джерела до одержувача, вибираючи найефективніший шлях через мережу. Мережевий рівень використовує логічні адреси, такі як IP-адреси, для ідентифікації пристроїв у мережі та гарантує, що пакети надсилаються за призначенням. Він також обробляє фрагментацію і повторне збирання пакетів, коли дані занадто великі, щоб бути переданими в одному пакеті. Мережевий рівень має вирішальне значення для забезпечення зв'язку між різними мережами і відіграє ключову роль у забезпеченні ефективної та надійної доставки даних.

Канальний рівень (data link layer) мережевої моделі OSI відповідає за забезпечення безпомилкової передачі кадрів даних на фізичному рівні. Він поділяється на два підрівні – рівень керування доступом до середовища (Media Access Control, MAC) і рівень керування логічним рівнем (Logical Link Control, LLC). Рівень MAC відповідає за адресацію кадрів на фізичному рівні, в той час як рівень LLC забезпечує управління потоком і механізми перевірки помилок при передачі даних. Канальний рівень також гарантує, що дані передаються надійно і без помилок, використовуючи такі методи, як циклічна перевірка надлишковості (CRC) і контрольні суми. Канальний рівень має вирішальне значення для забезпечення надійної та ефективної передачі даних через мережу.

Фізичний рівень (physical layer) мережевої моделі OSI відповідає за передачу бітів необроблених даних через фізичне середовище, наприклад, мідні дроти або оптоволоконні кабелі. Він має справу з електричними і механічними аспектами передачі даних, такими як рівні напруги, що використовуються для представлення 0 і 1, і фізичні характеристики кабелів, що використовуються для передачі сигналів. Фізичний рівень також визначає фізичні властивості мережевої інтерфейсної карти (NIC), яка використовується для передачі та отримання даних у мережі. Таким чином, фізичний рівень забезпечує фізичні засоби для передачі даних між пристроями в мережі.

### 1.2.2 TCP/IP модель

Модель TCP/IP – це набір мережевих протоколів, який використовується для встановлення зв'язку між пристроями в мережі. Він названий на честь двох основних протоколів: Transmission Control Protocol (TCP) та Internet Protocol (IP). Модель TCP/IP – це багаторівневий підхід до мережевої комунікації, де кожен рівень відповідає за певні завдання та обов'язки [3]. Концептуально модель складається з чотирьох рівнів:

- прикладний рівень (application layer);
- транспортний рівень (transport layer);
- інтернет рівень (internet layer);
- канальний рівень (data link layer).

Варто зазначити, що обидві моделі OSI та TCP/IP є важливими для розуміння мережевої комунікації, проте вони служать різним цілям. TCP/IP в першу чергу дозволяє хостам підключатися до Інтернету, використовуючи протокол управління передачею для передачі даних на рівні додатків. Він забезпечує гнучкість у дотриманні правил комунікації за умови дотримання загальних принципів. З іншого боку, модель OSI діє як шлюз між мережею та кінцевими користувачами, пропонуючи всеосяжну еталонну модель зі строгими протоколами та обмеженнями.

## 1.3 Технології створення вебдодатків

### 1.3.1 Контейнеризація за допомогою Docker

Docker – це платформа, розроблена для полегшення створення, розгортання та запуску додатків за допомогою контейнерів. Контейнери дозволяють розробникам упаковувати програму з усіма необхідними частинами, такими як бібліотеки та залежності, і відправляти все це одним пакетом. Це гарантує, що програма працюватиме на будь-якій іншій машині, де встановлено Docker, незалежно від будь-яких налаштувань, які може мати ця машина.[4]

Образи Docker є основою контейнерів Docker. Це шаблони, доступні лише для читання, які містять код програми, бібліотеки, залежності та конфігурацію, необхідні для запуску програми. Образи будуються з серії шарів, кожен з яких представляє інструкцію в Docker-файлі образу. Ці шари накладаються один на одного, і коли вони об'єднуються, то утворюють повний образ. Однією з ключових переваг образів Docker є їхня портативність. Після створення образу, ним можна поділитися і запустити на будь-якому комп'ютері, на якому встановлено Docker, незалежно від базової операційної системи або інфраструктури. Це забезпечує узгодженість у різних середовищах і усуває проблему «це працює на моїй машині». Образи Docker зберігаються в реєстрах образів, таких як Docker Hub, або приватних реєстрах всередині організації. Ці реєстри дозволяють розробникам легко ділитися, співпрацювати та розповсюджувати свої образи. Образи можна витягувати з реєстру, а також створювати контейнери на основі цих образів. Образи ідентифікуються за допомогою унікальної мітки, яка зазвичай містить назву та версію образу. Це дозволяє керувати різними версіями образу і гарантує, що при створенні контейнерів буде використано правильний образ. Коли контейнер створюється на основі образу, поверх шарів образу, призначених лише для читання, додається новий шар для запису. Цей шар для запису дозволяє контейнеру вносити зміни і зберігати дані, характерні для цього екземпляра контейнера. Однак, будь-які зміни, зроблені всередині контейнера, є ізольованими і не впливають на зображення, що лежить в основі.

Контейнери Docker – це виконувані екземпляри образів Docker. Вони інкапсулюють код програми, залежності та середовище виконання, забезпечуючи ізольовану та портативну одиницю програмного забезпечення. Контейнери легкі та швидкі, що дозволяє запускати декілька екземплярів програми на одному хості без конфліктів. Контейнерами керує середовище виконання Docker, яке займається створенням, виконанням і знищенням контейнерів. Кожен контейнер має власну ізольовану файлову систему, мережевий стек та простір процесів, що гарантує незалежність контейнерів один

від одного та від хост-системи. Однією з ключових переваг контейнерів Docker є їх портативність. Оскільки контейнери інкапсулюють всі необхідні компоненти, їх можна легко переміщати між різними середовищами, без додавання специфічних конфігурацій. Docker-файл – це текстовий файл, який містить набір інструкцій, що використовуються для автоматизації процесу створення образу Docker. Він слугує планом для створення образу, який включає всі необхідні компоненти, такі як код програми, залежності та середовище виконання. Docker-файли мають певний формат і використовують декларативний синтаксис для визначення кроків, необхідних для створення образу. Кожна інструкція в докер-файлі створює новий шар в образі, що дозволяє ефективно створювати образи та ділитися ними.

Контейнери та віртуальні машини (ВМ) забезпечують віртуалізацію, але по-різному. Віртуальні машини включають цілі операційні системи, що робить їх важчими та повільнішими у запуску. Контейнери, з іншого боку, використовують ядро ОС хост-системи та ізолюють лише додаток і його залежності. Це робить контейнери набагато легшими, ефективнішими у використанні ресурсів і швидшими у запуску, ніж віртуальні машини. Контейнери пропонують більш спрощений, підхід до віртуалізації порівняно з більш широкою системною емуляцією віртуальних машин.

### 1.3.2 Оркестрація контейнерів за допомогою Docker Compose

Docker Compose – це корисний помічник для організації та запуску складних додатків з декількома контейнерами. Даний інструмент спрощує керування багатоконтейнерними додатками, замість того, щоб керувати окремими конфігураціями контейнерів і залежностями, можна визначити всі сервіси, мережі та сховища за один раз. Таке централізоване налаштування спрощує управління, інкапсулюючи усі елементи додатку. [5]

Docker Compose значно підвищує ефективність розробки, дозволяючи розробникам визначати налаштування програми в одному YAML файлі. За допомогою лише однієї команди розробники можуть швидко розгорнути весь

додаток, оминаючи необхідність налаштовувати кожну частину окремо. Даний підхід зменшує затрати часу та енергії, дозволяючи розробникам зосередитися на написанні коду, а не на управлінні інфраструктурою. Крім того, Docker Compose гарантує, що всі розробники працюють з однаковими налаштуваннями в різних середовищах, сприяючи стандартизованому підходу до розробки. Це допомагає запобігти розбіжностям між налаштуваннями та мінімізує ймовірність виникнення помилок або проблем сумісності, що виникають через відмінності у середовищах.

У Docker Compose всі контейнери, вказані у файлі компіляції, підключаються до однієї внутрішньої мережі, що захищає їх від несанкціонованого доступу. Це не лише підвищує безпеку, але й спрощує керування мережею для багатоконтейнерних програм.

Ізоляція має велике значення в контейнерних середовищах і Docker Compose забезпечує певний рівень ізоляції, інкапсулюючи кожен компонент програми в його контейнер. Це запобігає конфліктам між залежностями і гарантує, що зміни, внесені в один компонент, не вплинуть на інші. [6]

### 1.3.3 Мова програмування Typescript

TypeScript – це строго типізована мова програмування, яка базується на JavaScript і розширює його можливості, дозволяючи розробникам визначати типи змінних, параметрів функцій та значень, що повертаються. Це означає, що код TypeScript містить явні анотації типів, що дозволяє компілятору виявляти помилки, пов’язані з типами, під час компіляції, а не під час виконання програми. TypeScript компілюється в JavaScript і може виконуватися в будь-якому середовищі виконання JavaScript, наприклад, в браузері, або на сервері Node.js. Також, TypeScript підтримує декілька парадигм програмування, таких як функціональне, та об’єктно-орієнтоване програмування. [7]

Спочатку мова програмування TypeScript була розроблена Андерсом Хейлсбергом у 2012 році, а зараз розробляється і підтримується корпорацією Майкрософт, як проект з відкритим вихідним кодом.

Наведемо основні особливості TypeScript.

*Статична типізація.* Можливість вказувати типи для змінних допомагає виявляти помилки під час розробки, а не під час виконання. Це може значно підвищити надійність вашого коду.

*Читабельність та зручність супроводу коду.* Завдяки типам код стає більш самодокументованим. Іншим розробникам легше зрозуміти, які типи даних очікуються і повертаються функціями, що призводить до кращої супроводжуваності.

*Підтримка та інструментарій IDE.* TypeScript без проблем працює з сучасними редакторами коду, надаючи такі функції, як автозавершення, перевірка типу та інтелектуальні підказки. Це додає зручності та прискорює розробку.

*Розширені можливості ООП.* TypeScript підтримує такі функції, як класи, інтерфейси та успадкування, що робить його більш придатним для створення великомасштабних додатків, використовуючи об’єктно-орієнтований підхід.

*Модульність коду.* TypeScript дозволяє використовувати модулі для організації коду. Це полегшує управління та масштабування кодової бази в міру зростання проекту.

*Широкі можливості конфігурації.* Файл `tsconfig.json` дозволяє налаштувати компілятор TypeScript відповідно до потреб проекту.

### 1.3.4 Front end технології React та Next.js

React – це JavaScript бібліотека, розроблена компанією Facebook, яка в основному використовується для створення користувацьких інтерфейсів. Дана бібліотека дозволяє розробникам створювати багаторазові компоненти декларативним методом, роблячи процес розробки більш ефективним та керованим. [8]

Зручність React полягає в наступних речах.

*Компонентна архітектура.* React заохочує модульний підхід до розробки, дозволяючи розробникам створювати інкапсульовані компоненти, які керують власним станом. Це сприяє багаторазовому використанню та підтримці.

*Virtual DOM.* Використовуючи віртуальне представлення DOM, React ефективно оновлює та рендерить лише ті компоненти, що змінилися, що призводить до підвищення продуктивності.

*Багата екосистема.* React має широку екосистему бібліотек та інструментів, включаючи Redux для управління станами та React Router для навігації. Це дозволяє легко знаходити рішення та інтегрувати сторонні інструменти.

*Підтримка спільноти.* Завдяки великій та активній спільноті розробники можуть легко знайти ресурси, навчальні посібники та підтримку для React, що забезпечує швидке навчання та саморозвиток.

React чудово підходить для сценаріїв, де динамічний контент та інтерактивність мають найбільше значення. Його підхід до рендерингу на стороні клієнта забезпечує швидке оновлення та безперебійну роботу додатку на стороні користувачів. Однак для рендерингу початкового контенту він покладається на JavaScript, що може вплинути на SEO та час, необхідний для першого повноцінного рендеру.

Зважаючи на вище сказане, можна виділити наступні області для застосування React, де ця бібліотека дасть найбільше результату.

Односторінкові додатки (SPA). React – чудовий вибір для SPA, які потребують високої інтерактивності та динамічного оновлення контенту.

Складні інтерфейси. React дуже добре підходить для додатків зі складним користувацьким інтерфейсом та частою зміною станів, оскільки React забезпечує гнучкість та продуктивність, необхідну для управління складними структурами проєктів.

Масштабні додатки. Завдяки компонентній архітектурі та розгалуженій екосистемі, React добре підходить для великомасштабних додатків, які вимагають модульності та супроводжуваності.

Для того щоб краще розуміти React та принципи його роботи, варто також поглибитись в найкращі практики використання React. [9]

Використання функціональних компонент замість класових компонентів дає дуже багато переваг, таких як читабельність, простота використання, повторне використання коду та гнучкість. Функції забезпечують чистіший код, усуваючи складнощі, пов’язані з керуванням ключовим словом ‘this’, конструктором і функціями життєвого циклу. Хуки покращують повторне використання коду, дозволяючи інкапсулювати і повторно використовувати логіку в декількох компонентах.

Використання стрілочних функцій не тільки зменшує кількість символів, порівняно зі звичайними функціями, але й покращують читабельність коду та зменшують його нагромадженість, особливо для невеликих функцій, таких як обробники подій або функції зворотного виклику. Їх спрощений синтаксис допомагає підтримувати лаконічність і покращує загальну читабельність кодової бази. Крім того, стрілочні функції не мають власного this, тобто не мають власного контексту, що призводить до більш чистого та інтуїтивно зрозумілого коду. Використовуючи стрілочні функції, можна покращити зручність супроводу коду та спростити процес розробки, що в підсумку сприятиме написанню ефективного та читабельного коду.

Підвищення ефективності рендерингу компонентів є важливим для підтримки оптимальної продуктивності React-додатків, особливо тих, що мають складні та розгалужені користувацькі інтерфейси. Можна підвищити продуктивність рендерингу за допомогою наступної стратегії – використання React.memo, компонента вищого порядку, який дозволяє запам’ятовувати результати рендерингу компонентів. Це запобігає непотрібному повторному рендерингу, коли пропси компонента залишаються незмінними.

Next.js, з іншого боку, це фреймворк на основі React, розроблений компанією Vercel. Він бере усе найкркаще від React, додаючи потужні функції, такі як рендеринг на стороні сервера (SSR), статична генерація сайтів (SSG) та

вбудований роутинг, за допомогою папок. Next.js надає комплексне рішення для легкого створення сучасних вебдодатків. [10]

Next.js є комплексним та потужним рішенням, його ключовими особливостями є:

- рендеринг на стороні сервера (SSR). Next.js підтримує SSR, який рендерить сторінки на сервері перед відправкою їх клієнту, що призводить до швидшого початкового завантаження сторінок та покращення SEO, оскільки пошукові системи можуть сканувати повністю відрендеризовані сторінки;

- статична генерація сайту (SSG). Next.js може генерувати статичні HTML-сторінки під час збірки, пропонуючи переваги статичних сайтів з динамічними можливостями React.js, що ідеально підходить для сайтів з великим вмістом, які потребують відмінної продуктивності та масштабованості;

- створення API. Next.js дозволяє розробникам створювати API в тій самій кодовій базі, що й фронтенд додаток, що спрощує процес розробки, консолідуючи логіку фронтенду та бекенду;

- вбудована підтримка css та Sass. Next.js має вбудовану підтримку CSS та Sass, що дозволяє розробникам стилізувати свої додатки без додаткових налаштувань;

- маршрутизація на основі файлів. Next.js використовує файлову систему маршрутизації, де маршрути визначаються файловою структурою проекту (цей інтуїтивно зрозумілий підхід спрощує створення складних шаблонів маршрутизації).

Щодо сфери застосування Next.js та коли його краще використовувати, можна виділити наступне:

- проекти, де важлива SEO оптимізація (можливості SSR та SSG роблять Next.js ідеальним інструментом для проектів, які значною мірою покладаються на SEO, таких як блоги, сайти новин та платформи електронної комерції);

- платформи з великою кількістю контенту (для сайтів зі великою кількістю статичного контенту, який не часто змінюється, SSG забезпечує блискавичне завантаження і відмінну продуктивність);

- гібридні додатки (Next.js ідеально підходить для додатків, які потребують поєднання статичного та динамічного контенту, дозволяючи розробникам використовувати SSR та SSG за потреби).

Окрім, перерахованих сфер застосування, Next.js, як і будь-який інструмент, можна використовувати будь-де, при цьому варто зважати на особливості інструменту, його підводні камені та які можуть бути проблеми при довгостроковому використанні.

Порівнюючи продуктивність Next.js та React.js, важливо враховувати конкретні вимоги проекту. React.js, завдяки своєму віртуальному DOM дереву, забезпечує чудову продуктивність рендерингу на стороні клієнта. Однак, Next.js робить ще один крок вперед завдяки рендерингу на стороні сервера і статичній генерації сайту, що може значно підвищити продуктивність і SEO оптимізацію додатку. [11]

### 1.3.5 Back end технології NestJS та MongoDB

NestJS – це прогресивний фреймворк для розробки вебдодатків на базі Node.js. Він використовує TypeScript і спирається на концепцію модульності, об’єктно-орієнтованого програмування та функціонального програмування для побудови масштабованих та ефективних додатків. [12]

Однією з головних переваг Nest.js є його архітектура, яка пропонує модульний підхід до розробки. За допомогою модулів, розробники можуть організовувати функціональність додатку на логічні блоки, що спрощує управління залежностями, перевикористання коду та розширення функціональності. Це дозволяє будувати додатки, які легко масштабувати та тестувати.

Nest.js підтримує парадигму реактивного програмування та використовує RxJS для роботи з асинхронними операціями. Це дозволяє легко виконувати операції на основі подій, такі як обробка запитів HTTP, маніпуляція даними в реальному часі та інші асинхронні операції. Реактивність дозволяє створювати

швидкі та потужні додатки, які можуть ефективно працювати з великою кількістю одночасних з’єднань.

Іншою перевагою Nest.js є його інтеграція з популярними бібліотеками тафреймворками. Він підтримує засоби для взаємодії з базами даних, які використовуються в Node.js, такі як TypeORM, Sequelize та Mongoose. Nest.js також надає спеціальні модулі для інтеграції з фреймворками вебсокетів, такими як Socket.io та GraphQL.

Незважаючи на свої переваги, Nest.js також має певні обмеження. Він може бути складним для новачків, оскільки вимагає знання TypeScript та розуміння принципів ООП. Деякі розробники можуть віддавати перевагу більш простим або легшим у використанні фреймворкам для розробки Node.js додатків.

У підсумку, Nest.js – це потужний фреймворк для розробки вебдодатків на базі Node.js. Він пропонує модульну архітектуру, підтримку реактивного програмування та інтеграцію з популярними бібліотеками. Nest.js є чудовим вибором для розробників, які шукають потужність та масштабованість у розробці серверних додатків на Node.js.

MongoDB – популярна база даних NoSQL, відома своєю гнучкістю, масштабованістю та простотою використання. На відміну від традиційних реляційних баз даних, MongoDB зберігає дані в JSON-подібних документах, що робить її чудовим вибором для сучасних вебдодатків, які потребують динамічних схем. [13]

NoSQL бази даних, такі як MongoDB, призначені для обробки та зберігання даних, з якими традиційні реляційні бази даних можуть не впоратися. Наведемо кілька сценаріїв, в яких бази даних NoSQL мають особливу перевагу.

*Вимоги до гнучкості схем.* Якщо програма вимагає гнучкої схеми, що постійно змінюється, бази даних NoSQL дозволять зберігати дані без попередньо визначеної схеми.

*Великий обсяг неструктурованих даних.* Для додатків, що мають справу з великими обсягами неструктурованих, або напівструктурованих даних, таких як

пости в соціальних мережах, журнали або мультимедійний контент, бази даних NoSQL можуть ефективно зберігати і управляти цими даними.

*Потреби в масштабованості.* Якщо додаток повинен обробляти великі обсяги даних і мати високу пропускну здатність, бази даних NoSQL забезпечують горизонтальну масштабованість, дозволяючи додавати більше серверів для обробки навантаження.

*Швидкі цикли розробки.* Гнучкість NoSQL баз даних може прискорити розробку, дозволяючи вносити зміни в модель даних без складних міграцій.

*Розподілені сховища даних.* Для додатків, які потребують розподіленого сховища даних, бази даних NoSQL призначені для роботи на кластерах машин, забезпечуючи високу доступність і надійність.

*Аналітика в режимі реального часу.* Якщо потрібна аналітика в реальному часі та швидкі операції читання/запису, бази даних NoSQL оптимізовані для швидкої роботи і можуть обробляти великі обсяги даних в реальному часі.

Отже, MongoDB – це потужна та гнучка база даних NoSQL, яка може задовольнити найрізноманітніші потреби у зберіганні даних. Її документно-орієнтований підхід дозволяє використовувати динамічні схеми, що робить її ідеальною для сучасних вебдодатків.

## 1.4 Особливості менеджерів паролів

У процесі створення таких додатків виникають численні виклики, пов’язані із забезпеченням безпеки чутливих даних, шифруванням, зберіганням, а також передачею інформації. Основні архітектурні особливості можна розділити на кілька категорій.

Шифрування та дешифрування даних – менеджери паролів використовують сильні алгоритми шифрування, щоб захистити збережені паролі. Шифрування, як правило, здійснюється з використанням симетричних алгоритмів, таких як AES-256, або його аналогів:

- ключі шифрування генеруються на основі головного пароля користувача за допомогою функцій похідного ключа, таких як PBKDF2 або Argon2, що дозволяє уникнути брутфорс-атак і захистити дані;

- дешифрування відбувається лише на клієнтському пристрої користувача, що забезпечує максимальну конфіденційність даних, адже нешифровані паролі ніколи не передаються на сервер.

Безпека зберігання даних:

- локальне зберігання – для менеджерів паролів, які зберігають дані на пристрої, використовується шифрування на стороні клієнта (доступ до даних можливий тільки з правильним головним паролем);

- хмарне зберігання – для синхронізації між пристроями дані можуть зберігатися у хмарному сховищі, але вони завжди зберігаються в зашифрованому вигляді (сервіс не має доступу до відкритих даних, завдяки принципу нульових знань).

Передача даних:

- протоколи безпеки для передачі – у разі передачі даних між пристроями або до хмарного сховища використовуються захищені протоколи, такі як HTTPS та TLS, що забезпечують безпечний канал передачі даних;

- шифрування на стороні клієнта – в ідеалі дані шифруються ще до передачі, що гарантує їх безпеку під час пересилання та недоступність для сервера.

## 1.5 Криптографічні технології та захист інформації в контексті розробки менеджерів паролів

### 1.5.1 Хешування

Хешування – це процес перетворення вхідних на рядок символів фіксованої довжини, який зазвичай є шістнадцятковим числом. Результат цього процесу називається “хеш-значення”, або “хеш-код”. Наведемо ключові особливості хешування.

*Детермінованість.* Хеш-функція завжди поверне один і той самий результат, при однакових вхідних даних. Ця властивість має вирішальне значення для перевірки цілісності даних та зберігання паролів.

*Фіксована довжина вихідних даних.* Незалежно від розміру або довжини вхідних даних, хеш-функція створює хеш-значення з фіксованою довжиною. Наприклад, хеш-функція SHA-256 завжди створює хеш-значення довжиною 256 біт (32 байти).

*Незворотність.* Хешування – це односторонній процес (рис. 1.1). Отримавши хеш-значення, його не можна змінити, щоб отримати вихідні дані. Це робить його придатним для захисту конфіденційної інформації, наприклад, паролів.

*Ефект лавини.* Невелика зміна вхідних даних повинна призвести до суттєвої зміни хеш-значення. Ця властивість гарантує, що схожі вхідні дані не призведуть до схожих хеш-значень.

*Ефективність* – алгоритми хешування розроблені таким чином, щоб бути швидкими та ефективними в обчисленні.

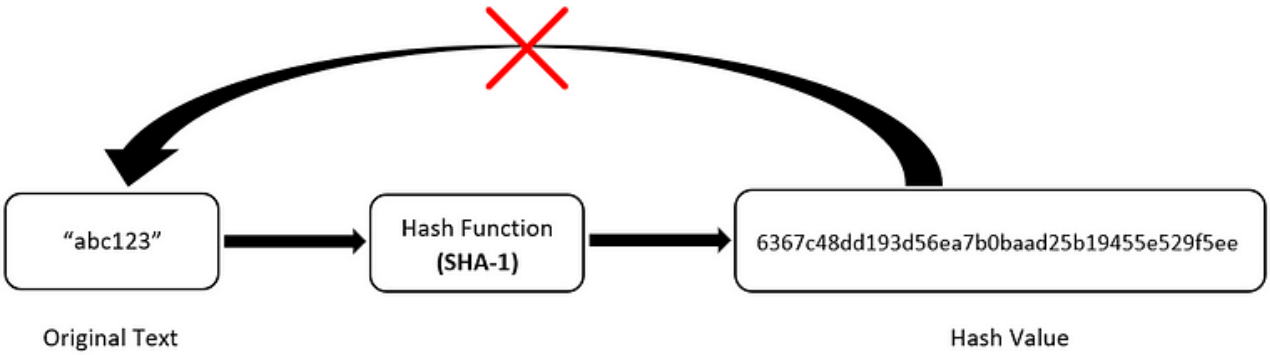


Рисунок 1.1 – Ілюстрація незворотності процесу хешування

Хешування є корисним і ефективним інструментом для багатьох задач в контексті безлічі додатків, наприклад. Хешування використовується для перевірки цілісності даних. Порівнюючи хеш-значення отриманих даних з очікуваним хеш-значенням, можна визначити, чи були дані підроблені під час передачі. Замість того, щоб зберігати паролі у відкритому вигляді, системи часто

зберігають хеш пароля. Коли користувач намагається увійти в систему, система хешує введений пароль і порівнює його зі збереженим хешем. Таким чином, навіть якщо база даних скомпрометована, зловмисники не відразу отримують доступ до паролів користувачів. Хеш-функції мають важливе значення в різних криптографічних протоколах і алгоритмах. Вони використовуються в цифрових підписах, центрах сертифікації тощо. Хешування використовується в структурах даних, таких як хеш-таблиці, для швидкого отримання даних на основі ключа. Це дозволяє ефективно шукати та отримувати дані. [14]

### 1.5.2 Шифрування

Шифрування – це фундаментальне поняття в інформатиці та кібербезпеці. Це процес перетворення відкритого тексту (даних, які можна прочитати) на зашифрований текст (дані, які не можна прочитати) за допомогою певного алгоритму та секретного ключа (рис 1.2). Основна мета шифрування – убезпечити дані та захистити їх від несанкціонованого доступу, забезпечуючи конфіденційність і приватність.

До ключових компонентів шифрування відносяться: вхідні дані, алгоритм шифрування, ключ шифрування, зашифровані дані, розшифрування.

Вхідні дані – це оригінальні, читабельні дані, які потрібно зашифрувати і захистити. Це може бути що завгодно: текстові повідомлення, файли, номери кредитних карток, особиста інформація, або будь-які інші конфіденційні дані.

Алгоритм шифрування – це набір математичних правил і операцій, які використовуються для перетворення відкритого тексту в зашифрований. Сучасні алгоритми шифрування розроблені таким чином, щоб бути надзвичайно складними, щоб несанкціоновані особи не могли змінити процес і розшифрувати дані без відповідного ключа. Прикладами алгоритмів шифрування є Advanced Encryption Standard (AES), RSA та DES.

Ключ шифрування – є найважливішим компонентом процесу шифрування. Це секретне значення або серія бітів, які використовуються як вхідні дані для алгоритму шифрування. Вибір ключа суттєво впливає на безпеку шифрування;

Зашифровані дані – це результат шифрування відкритого тексту за допомогою алгоритму шифрування та ключа. Він виглядає як зашифрований, нечитабельний набір символів або байтів. Навіть якщо хтось перехопить або отримає доступ до зашифрованого тексту, він не зможе розшифрувати його без правильного ключа розшифрування;

Розшифрування – це процес перетворення зашифрованого тексту назад у відкритий. Це зворотна операція шифрування і вимагає використання того ж алгоритму шифрування і правильного ключа розшифрування.

### 1.5.3 Симетричне шифрування

Шифрування симетричним ключем (рис. 1.2), також відоме як симетрична криптографія – це тип криптографії, який використовує один і той самий ключ для шифрування і розшифрування повідомлення. Це означає, що і відправник, і одержувач повідомлення повинні мати доступ до одного і того ж секретного ключа для його кодування і декодування. [15]

Симетричне шифрування є найпоширенішим типом шифрування і зазвичай використовується в таких додатках, як електронна пошта, обмін файлами та віртуальні приватні мережі (VPN). Advanced Encryption Standard (AES) – найпопулярніший симетричний алгоритм, а AES256 – найстійкіший з доступних симетричних алгоритмів. Уряд США використовує AES256 для захисту секретної інформації. Історично, Data Encryption Standard (DES) широко використовувався, але його коротка довжина ключа робила його вразливим до brute-force атак, що призвело до його заміни на більш безпечні алгоритми, такі як AES і Triple DES (3DES).

Серед переваг симетричного шифрування можна виділити: безпеку, швидкість, ефективність, простоту. Симетричне шифрування забезпечує надійний захист даних, оскільки для шифрування та розшифрування використовується один і той самий ключ. До поширених алгоритмів симетричного шифрування належать Advanced Encryption Standard (AES), Data Encryption Standard (DES) та International Data Encryption Algorithm (IDEA). Це

ускладнює доступ до даних для неавторизованих користувачів. Симетричне шифрування, як правило, швидше, ніж інші типи шифрування, оскільки для шифрування і розшифрування використовується один і той же ключ. Це робить його гарним вибором для програм, які потребують швидкого шифрування та розшифрування даних. Симетричне шифрування вимагає менше обчислювальної потужності і ресурсів для шифрування і розшифрування даних, що може заощадити час і гроші. Симетричне шифрування легко реалізувати і використовувати, оскільки воно вимагає лише одного ключа для шифрування і розшифрування. Це робить його популярним вибором для додатків, які потребують простого і зрозумілого шифрування.

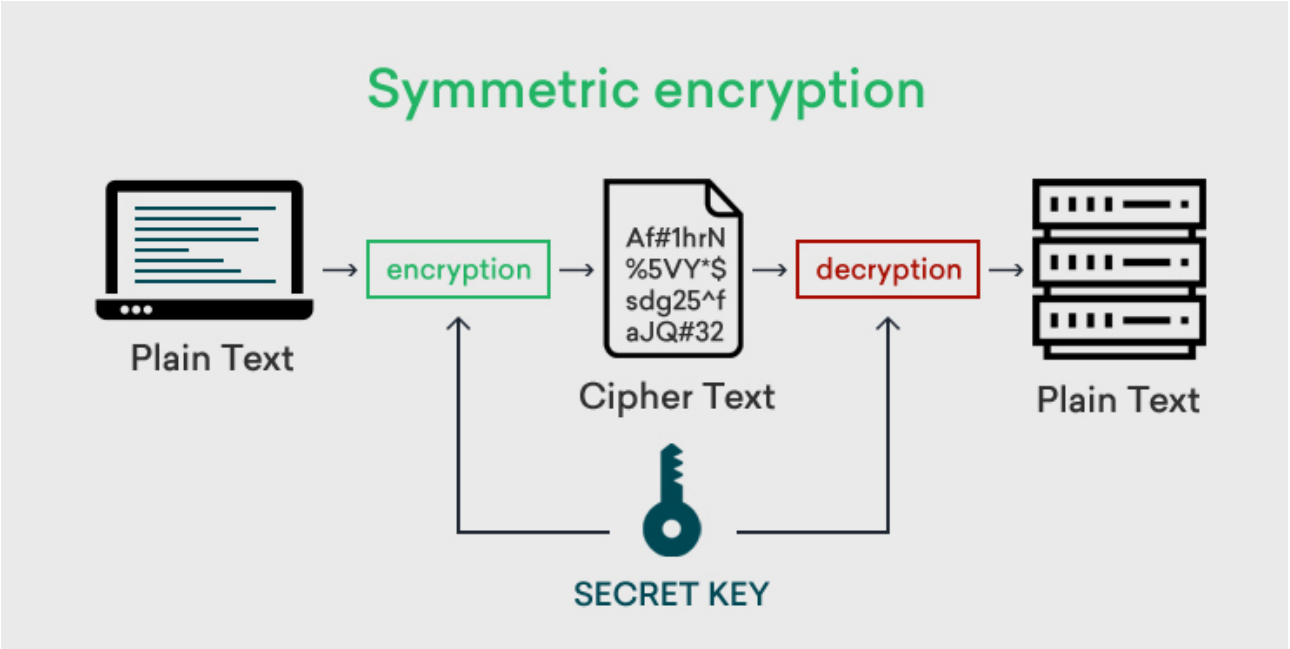


Рисунок 1.2 – Процес симетричного шифрування

### 1.5.4 Асиметричне шифрування

Асиметричне шифрування, також відоме як асиметрична криптографія, або шифрування з відкритим ключе – це тип криптографії, який використовує пару ключів – публічний ключ для шифрування і приватний для розшифрування даних (рис. 1.3) [16]. Цей метод шифрування є більш безпечним, ніж симетричне шифрування, оскільки він унеможливлює розшифрування даних особою, яка не має приватного ключа, навіть якщо вона має відкритий ключ. Асиметричне

шифрування має ряд переваг над іншими методами шифрування. Зокрема надійний захист даних, оскільки приватний ключ, який використовується для розшифрування, зберігається в таємниці і нікому не передається. Це ускладнює доступ неавторизованих користувачів до даних, зашифрованих асиметричними ключами. Асиметричне шифрування також може використовуватися для автентифікації, оскільки відкритий ключ може бути використаний для перевірки особи відправника повідомлення, а розшифрувати його можна лише за допомогою відповідного закритого ключа. Це допомагає запобігти шахрайству та захистити від зловмисних атак. Відсутність необхідності у захищеному каналі для розповсюдження ключів, оскільки кожен користувач має унікальну пару відкритий-закритий ключ, і тільки закритий ключ може розшифрувати повідомлення, зашифровані за допомогою відкритого ключа. Це спрощує розповсюдження ключів і керування доступом до зашифрованих даних. Асиметричне шифрування є гнучким і може використовуватися для широкого спектру завдань, включаючи безпечне спілкування електронною поштою, банківські операції в Інтернеті та безпечний доступ до мереж і систем, якщо зберігається таємниця приватного ключа.

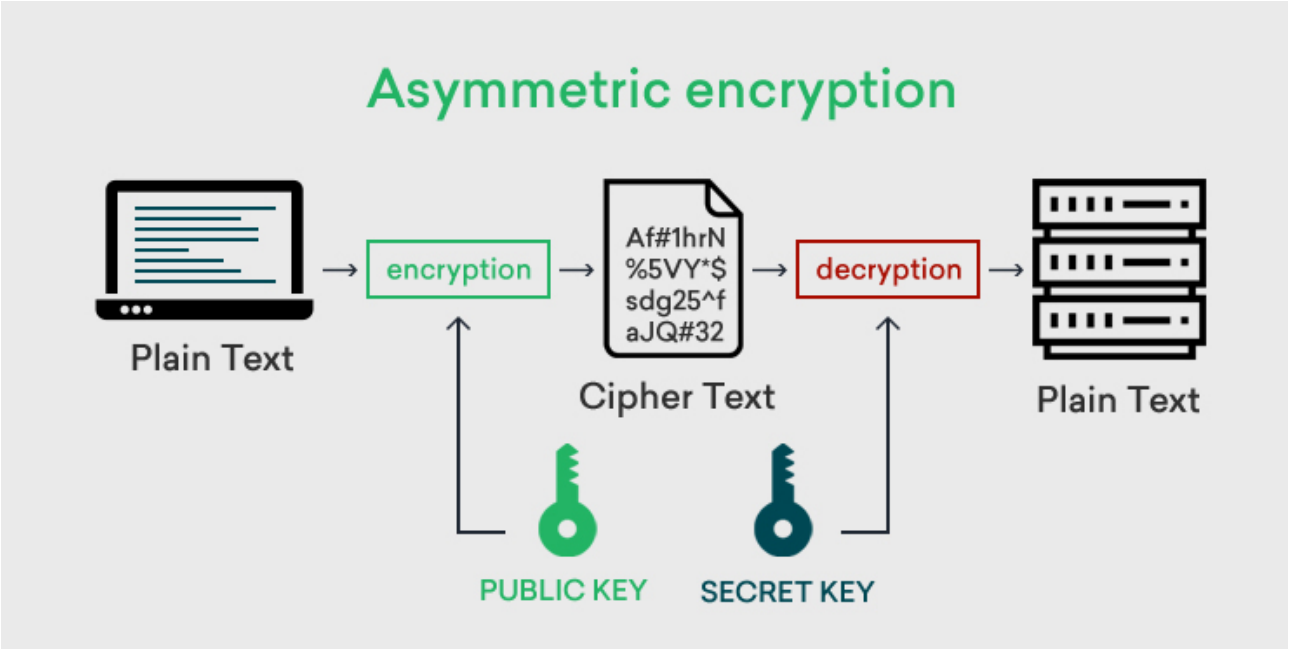


Рисунок 1.3 – Процес асиметричного шифрування

Загалом, асиметричне шифрування забезпечує надійний захист і автентифікацію даних, а також пропонує більшу гнучкість і можливості розподілу ключів, ніж симетричне шифрування, за умови збереження таємниці закритих ключів. Оскільки все більше програм використовують асиметричне шифрування, цілком ймовірно, що цей потужний інструмент стане ще більш поширеним.

Асиметричне шифрування менш поширене, ніж симетричне, але його популярність зростає. Воно використовується в додатках, де безпека є критично важливою, наприклад, в онлайн-банкінгу та криптовалюті. Найпопулярнішим асиметричним алгоритмом є алгоритм RSA, а RSA 2048 є найстійкішим асиметричним алгоритмом. Уряд США використовує RSA 2048 для захисту секретної інформації.

## 1.6 Дослідження аналогів додатку

Password (рис. 1.4) – це популярний менеджер паролів, що надає можливість зберігати та керувати паролями в захищеному сховищі, доступному через хмарний сервіс. Основні функціональні можливості включають:

- зберігання паролів, кредитних карток, нотаток та іншої конфіденційної інформації;
- генерацію надійних паролів для різних облікових записів;
- поширення паролів іншим користувачам;
- синхронізацію даних між пристроями через захищений канал.

LastPass (рис. 1.5) – це менеджер паролів, що працює на хмарній платформі та пропонує безкоштовні й платні плани для користувачів. Сервіс забезпечує високу безпеку паролів і легкий доступ до них з різних пристроїв. Основні функціональні можливості:

- автоматичне збереження і заповнення паролів на вебсайтах;
- створення надійних паролів та ключів;
- виявлення слабких паролів;

- підтримка двофакторної аутентифікації для додаткового захисту;
- можливість обміну паролями з іншими користувачами.

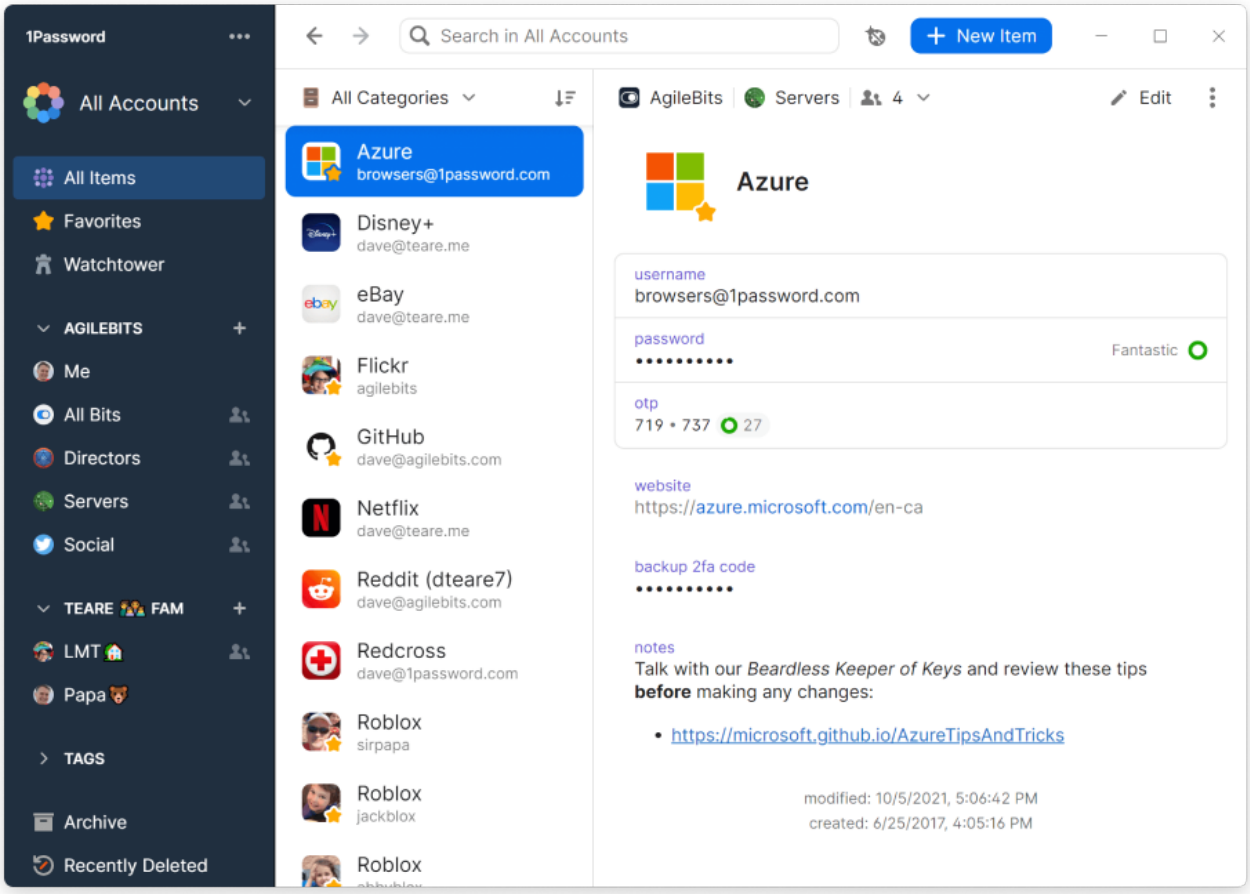


Рисунок 1.4 – 1Password

Bitwarden (рис. 1.6) – це відкритий і безкоштовний менеджер паролів, який пропонує широкий набір функцій для безпечного зберігання конфіденційних даних. Завдяки відкритому коду він є прозорим і надає впевненість у безпеці даних. Основні функціональні можливості:

- зберігання логінів, паролів, нотаток і приватних даних;
- підтримка спільного використання сховищ серед користувачів;
- захист даних за допомогою AES-256 бітного шифрування.

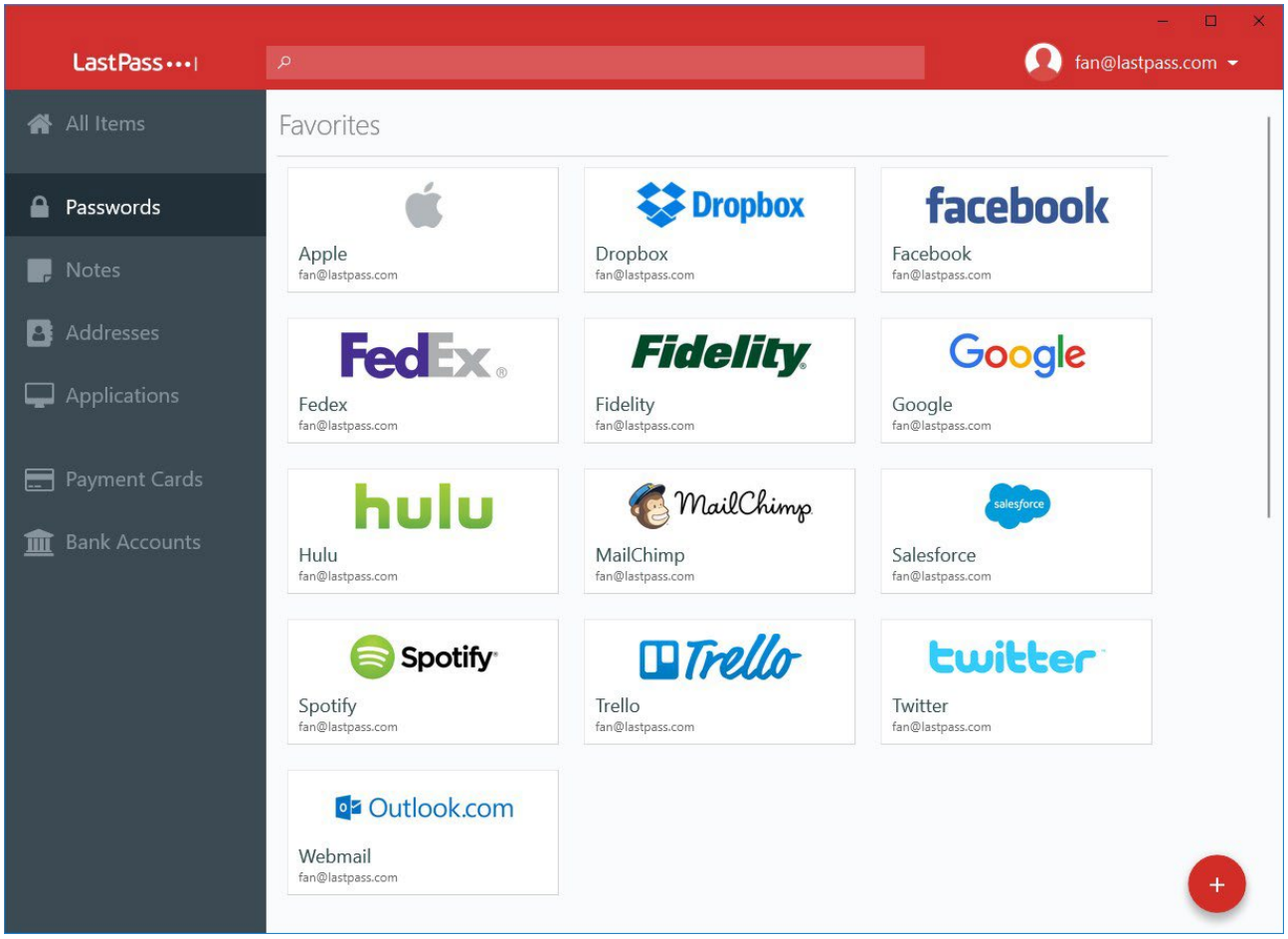


Рисунок 1.5 – LastPass

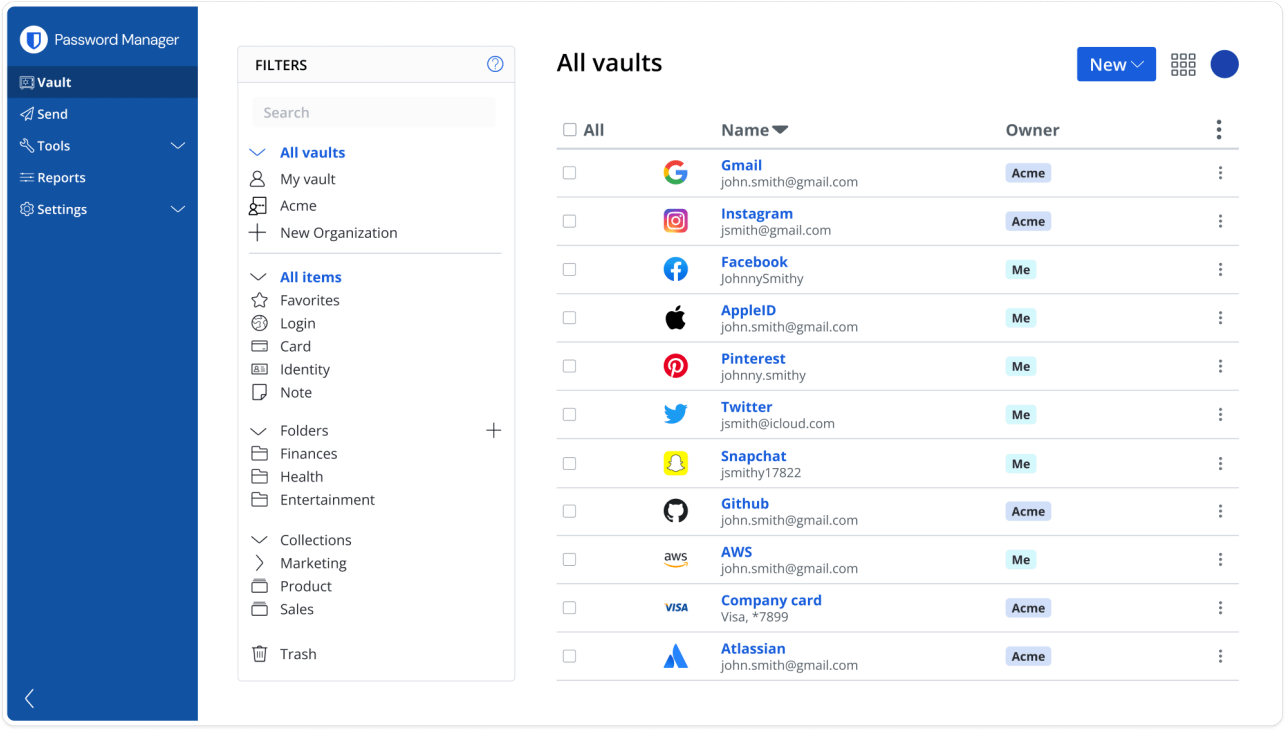


Рисунок 1.6 – Bitwarden

NordPass (рис. 1.7) від команди розробників NordVPN – це зручний, добре організований і простий у використанні сервіс для безпечного доступу до паролів через десктопні, мобільні додатки, або через веббраузер. Він отримав нагороду Editors' Choice за такі корисні функції, як повні та детальні звіти про витоки даних, маскування електронної пошти, звіт про стан пароля, безпечний спільний доступ та можливість успадкування пароля.

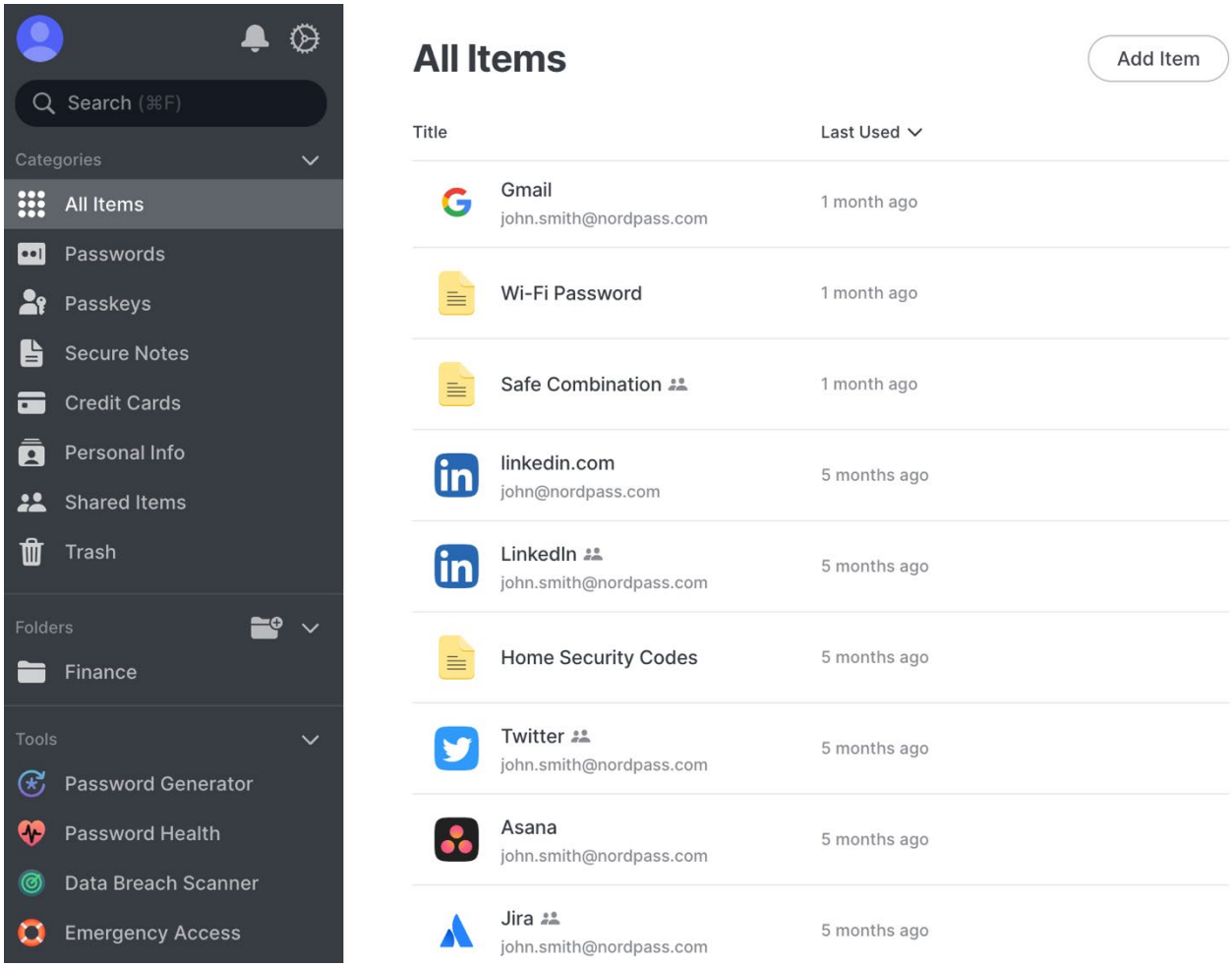


Рисунок 1.7 – NordPass

Зважаючи на функціонал існуючих аналогів, було прийнято рішення створити менеджер паролів із власним дизайном та адаптованим до потреб користувачів інтерфейсом. Розробка буде здійснюватися з використанням найкращих практик для забезпечення максимальної безпеки та зручності.

# РОЗДІЛ 2.
# РОЗРОБКА МЕНЕДЖЕРА ПАРОЛІВ

## 2.1 Постановка задачі та вимоги до менеджера паролів

Метою даної кваліфікаційної роботи є розробка менеджеру паролів, за допомогою сучасних методів розробки. Задля надання унікальності, було вирішено створити простий для розуміння і зручний дизайн, також було розроблено власний алгоритм шифрування, який використовує динамічні ключі. Даний менеджер паролів буде мати попит серед активних користувачів мережі Інтернет, які хочуть подбати про безпеку своїх облікових записів.

Головна проблема, котру має розв’язувати даний менеджер паролів – це можливість керування власними обліковими записами в одному місці, задля підвищення безпеки та зручності зберігання та використання облікових даних.

Розроблений менеджер паролів повинен підтримувати наступні функції:

- можливість створення, редагування та видалення нового сховища облікових записів;
- можливість створення, редагування та видалення облікового запису;
- можливість пошуку облікових записів по логіну, або ресурсу для якого призначений логін та пароль;
- можливість переглядати усі облікові записи до яких користувач має доступ.

Функціональні вимоги до вебдодатку:

- наявність форми реєстрації/входу;
- зберігання облікових записів;
- генерація паролів;
- шифрування та хешування чутливих даних;
- захист даних користувача за допомогою шифрування на основі мастер-пароля.

Нефункціональні вимоги:

- простота дизайну;
- зрозуміла навігація;
- безпека;
- масштабування;
- швидкодія.

Задля зайняття конкурентних позицій і забезпечення попиту серед користувачів, яким необхідні можливості надійного зберігання та управління паролями, необхідно розробити менеджер паролів із використанням сучасних технологій. Додаток повинен забезпечувати високий рівень захисту даних, простоту використання та функції, які відповідатимуть вимогам сучасної цифрової безпеки.

## 2.2 Опис проекту

Проектування і розробка менеджеру паролів проходитиме в наступні етапи:

- дослідження аналогів додатку;
- створення технічного дизайну додатку (рис. 2.1);
- проектування структури бази даних для надійного зберігання даних (рис. 2.2);
- розробка дизайну інтерфейсу для зручного використання менеджера паролів;
- розробка та впровадження власного алгоритму шифрування для підвищення безпеки збережених паролів;
- розробка серверної частини додатку та тестування ендпоінтів за допомогою Postman;
- розробка клієнтської частини додатку та перевірка зручності використання;
- комплексне тестування роботи менеджера паролів для забезпечення надійності та безпеки сервісу.

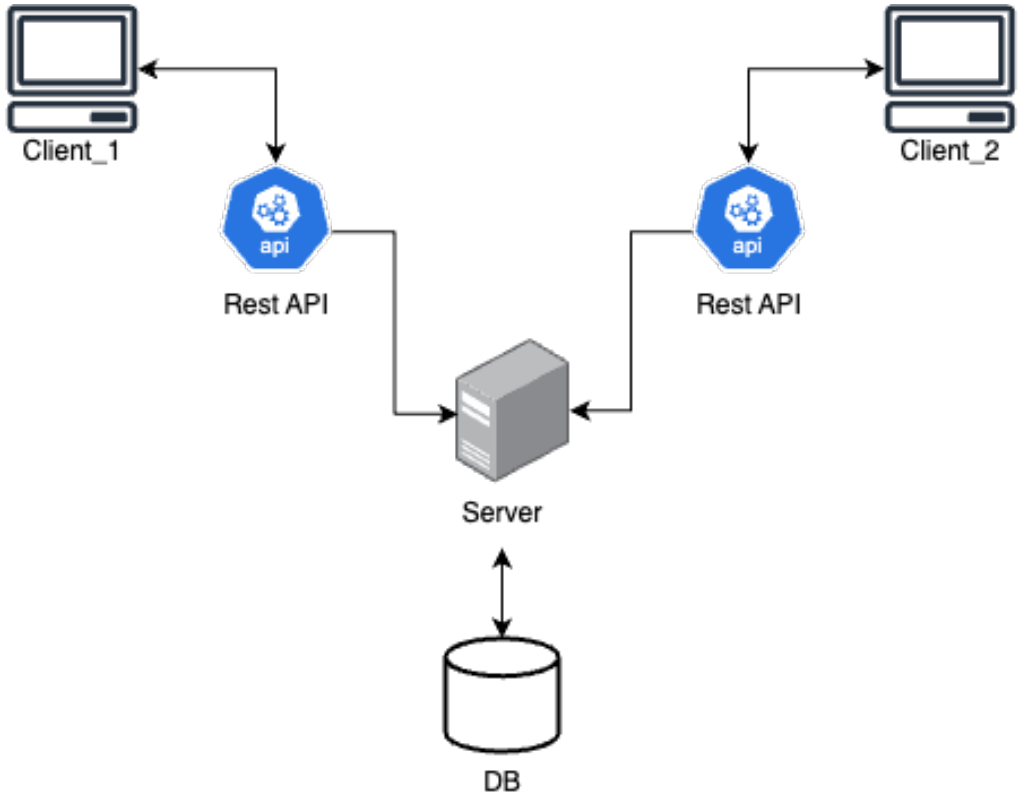


Рисунок 2.1 – Схема взаємодії клієнтської і серверної частин

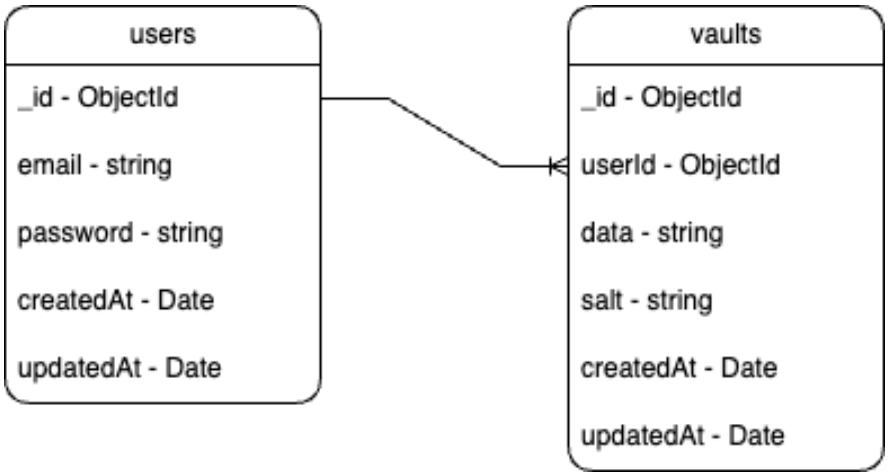


Рисунок 2.2 – Схема бази даних

## 2.3 Вибір моделі розробки

Найбільш доцільно для даного проекту було б використати ітеративну модель. Ітеративна модель (рис. 2.3) – це підхід до розробки програмного забезпечення, який передбачає розбиття проекту на невеликі, керовані ітерації.

Кожна ітерація є самодостатнім міні-проектом, результатом якого є робоча версія програмного забезпечення. Потім на основі зворотнього зв’язку проводять допрацювання та покращення продукту в наступних ітераціях. Цей процес триває доти, доки кінцевий продукт не відповідатиме бажаним вимогам. [17]

Переваги ітеративної моделі:

- гнучкість та адаптивність (ітеративна модель може легко адаптуватися до змін у вимогах під час виконання проекту, що робить її ідеальною для проектів з мінливими, або невизначеними вимогами);

- ранній зворотній зв’язок та валідація (зацікавлені сторони можуть надати цінний зворотній зв’язок на ранній стадії процесу розробки, гарантуючи, що кінцевий продукт відповідає їхнім потребам);

- зменшення ризиків (вирішуючи проблеми з високим ризиком або складні функції на ранній стадії процесу розробки, ітеративна модель допомагає знизити ризики проекту);

- простіше управління проектом (завдяки меншим, більш керованим ітераціям, управління проектами стає простішим);

- висока якість продукту (часте тестування та зворотній зв’язок дозволяють розробникам швидше виявляти та виправляти проблеми, що призводить до якіснішого кінцевого продукту).

Ітеративна модель містить наступні етапи.

*Збір та аналіз вимог.* На цьому етапі збираються вимоги від зацікавлених сторін;

*Проектування.* На етапі проектування команда представляє програму за допомогою різних діаграм, таких як діаграма потоків даних, діаграма діяльності, діаграма класів, діаграма сутностей, діаграма технічного дизайну;

*Реалізація.* На етапі реалізації усі вимоги і документацію перетворюють на робочу програму за допомогою коду;

*Тестування.* Після завершення етапу кодування починається тестування програмного забезпечення з використанням різних методів тестування. Існує

багато методів тестування, але найпоширенішими є методи білого, чорного та сірого ящиків;

*Розгортання.* Після завершення всіх етапів програмне забезпечення розгортається в робочому середовищі;

*Перевірка.* На цьому етапі, після розгортання продукту, перевіряють поведінку та валідність розробленого продукту. І якщо виявлено якусь помилку, то процес починається знову зі збору вимог;

*Обслуговування*. На етапі супроводу, після розгортання програмного забезпечення в робочому середовищі, можуть виникнути деякі помилки, деякі помилки або потрібні нові оновлення. Обслуговування включає в себе налагодження та додавання нових опцій.

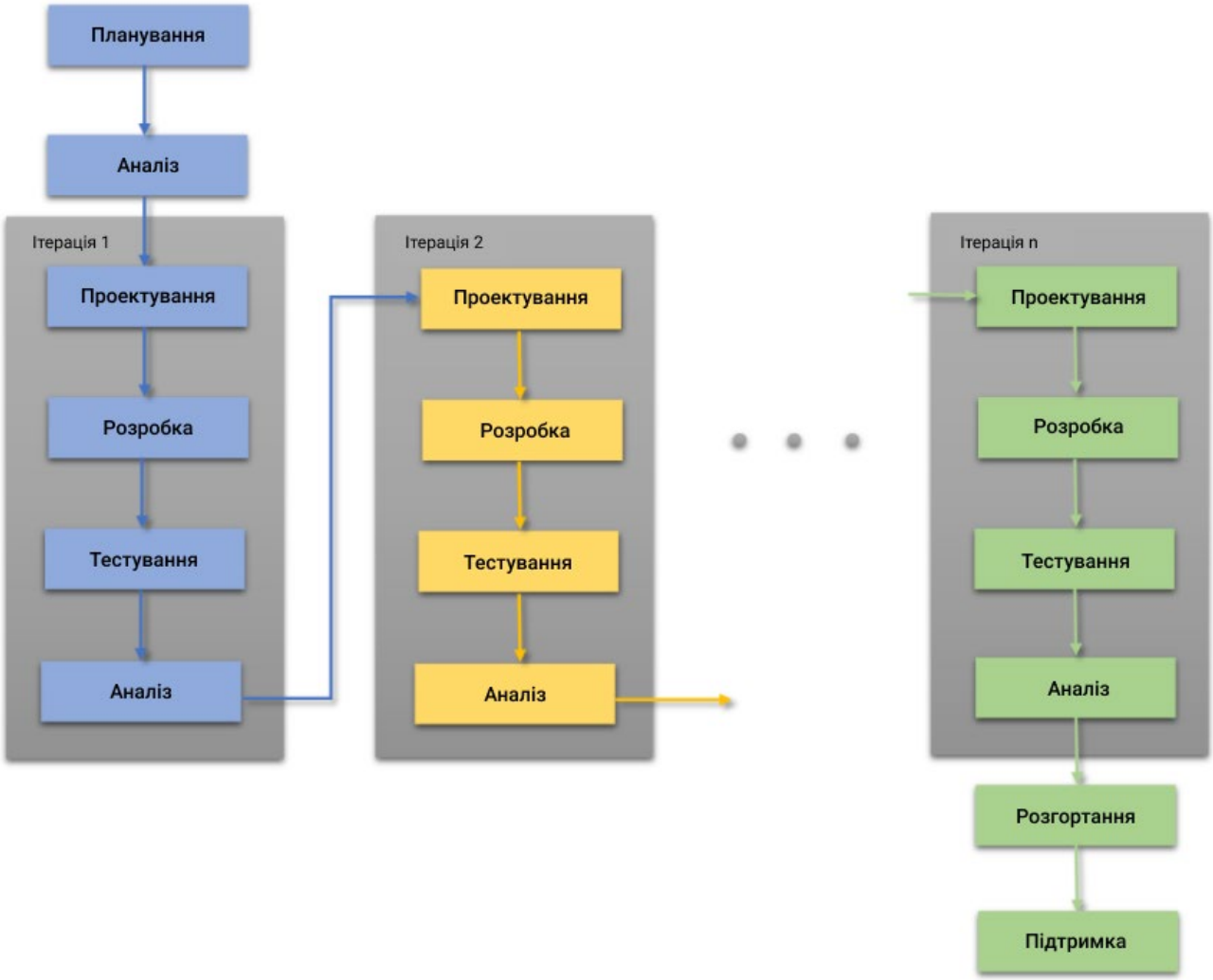


Рисунок 2.3 – Ітераційна модель розробки сайту

Отже, ітераційна модель є найбільш ефективним підходом у цьому випадку, оскільки вона забезпечує гнучкість і прозорість процесу розробки. Нові функції можуть додаватися на будь-якому етапі, а можливі недоліки додатка можна виявити на ранніх стадіях.

## 2.4 Обґрунтування вибору інструментальних засобів розробки

### 2.4.1 Редактор коду

Для розробки даного проекту в якості редактору коду я обрав WebStorm – це повнофункціональне середовище розробки з вбудованими інструментами налагодження, тестування та контролю версій. [18] Ці можливості стають надзвичайно корисними для великих і складних проектів.

Особливості та переваги WebStorm:

- розширені функції автокомпліту, рефакторингу та виявлення помилок, які можуть заощадити час розробників і підвищити продуктивність (Webstorm має функцію безпечного перейменування, де він шукає всі екземпляри файлу і самостійно рефакторить імпорт);
- присутні вбудовані засоби налагодження, рефакторингу, навігації за кодом, авто-доповнення та контекстні підказки;
- підтримка JavaScript, TypeScript, CSS і інших дотичних до веброзробки технолоігй;
- вбудований дебагер;
- вбудовані інструменти інтеграції з GitHub та GIT для швидкого тестування, збірки, упаковки та розвертання різних типів додатків;
- підтримка TypeScript, JavaScript, NodeJs, а також практично усіх мов програмування;
- відладчик всередині ide (підключення до відладчика chrome та firefox);
- розумне/контекстне автозавершення;
- консоль всередині ide;

- підтримка багатьох фреймворків та вебтехнологій, таких як React, Angular, Node.js та багато інших;
- чудова пошукова система;
- орієнтованість на клавіатуру, можливість пошуку у кожному меню, можливість зміни всіх комбінації клавіш, можливість додавання нових комбінацій клавіш;
- інтерфейс гнучко налаштовується і пропонує з коробки кілька чудових тем;
- має чудову підтримку git, svn та mercurial (управління кількома гілками, журнал, контриб'ютори, історія та багато іншого);
- підтримка Typescript з коробки;
- 15. менеджер плагінів з коробки;
- file watcher, який виконує команди, коли файли змінюються.
- загальна бібліотека шаблонів, готових фрагментів коду та сніппетів;
- можлива одночасна робота з декількома проектами, в окремих вікнах, також можливість розділити інтерфейс на дві панелі для зрівняння коду.

Інтерфейс WebStorm є зручним і зрозумілим, але щоб з ним розібратись потрібно більше часу. Оскільки, WebStorm – це повноцінне IDE, то воно забезпечує набагато більше можливостей в порівнянні з іншими текстовими редакторами. Щоб спростити роботу користувача, WebStorm поділено на п'ять основних регіонів:

- панель управління;
- бічна панель;
- редактор;
- панель консолі;
- статус бар.

### 2.4.2 Контроль версій

В якості системи контролю версій було обрано Git. Git – це безкоштовна система контролю версій з відкритим вихідним кодом, створена Лінусом

Торвальдсом у 2005 році. На відміну від старих централізованих систем контролю версій, таких як SVN та CVS, Git є розподіленою системою, тобто кожен розробник має повну історію свого сховища коду локально. Це робить початковий клон сховища повільнішим, але наступні операції, такі як commit, diff, merge та log журналу значно швидшими.

Git також має чудову підтримку розгалуження, злиття та перезапису історії сховища, що призвело до появи багатьох інноваційних та потужних робочих процесів та інструментів. Pull-запити – це один з таких популярних інструментів, який дозволяє командам співпрацювати над гілками Git'а та ефективно переглядати код один одного. Git є найпоширенішою системою контролю версій у світі і вважається сучасним стандартом для розробки програмного забезпечення. [19]

В якості вебсервісу для хостингу проєктів було обрано GitHub – це вебсервіс хостингу репозиторіїв Git, який дозволяє розробникам розміщувати та переглядати код, співпрацювати над проектами та відстежувати зміни. Належить Microsoft, він добре інтегрується з платформами Microsoft, такими як Azure, а також підтримує інтеграцію з AWS та іншими сервісами. [20]

GitHub має велику спільноту і пропонує безліч інструментів для підвищення ефективності роботи розробників:

- керування репозиторієм та хостингом;
- перегляд коду та управління кодом;
- трекер issues;
- інтегровані інструменти CI/CD та автоматизації, такі як Jenkins, Travis CI, CircleCI та інші;
- GitHub Copilot X – штучний інтелект, який допомагає при розробці.

### 2.4.3 Front end інструменти

В якості бібліотеки для написання інтерфейсу користувача було використано React. Це декларативна JavaScript-бібліотека, що забезпечує гнучкість та швидкість у розробці вебдодатків. Однією з ключових переваг React

є його компонентна архітектура, яка дозволяє створювати незалежні інкапсульовані компоненти, кожен з яких має власний стан та логіку. Це спрощує структуру проекту, оскільки зміни в одному компоненті не впливають на інші. Крім того, компоненти React можна використовувати повторно в різних частинах програми, що покращує модульність і спрощує підтримку та масштабування додатка. Використання JSX, HTML-подібного синтаксису, дозволяє легко інтегрувати компоненти з CSS-бібліотеками, налаштовуючи їх під індивідуальні потреби та забезпечуючи високий рівень інтерфейсу користувача.

В якості фреймворку для побудови проекту було використано Next.js. Він забезпечує додаткові можливості для організації додатка та оптимізації процесу розробки. Одна з переваг Next.js – проста система маршрутизації, яка базується на файловій структурі. Кожен файл у папці pages автоматично відповідає окремій сторінці, що робить маршрутизацію інтуїтивно зрозумілою та знижує складність структури проекту. Крім того, Next.js дозволяє створювати API-ендпоінти безпосередньо у фронтенд-проекті, що спрощує обробку даних та може бути корисним для невеликих сервісів чи тестових серверних функцій. Також Next.js має вбудовані інструменти для оптимізації зображень та статичних ресурсів, що знижує час завантаження сторінок і покращує продуктивність, позитивно впливаючи на користувацький досвід.

В якості бібліотеки для валідації форм було використано Zod. Це зручна бібліотека для декларативної валідації даних, що дозволяє створювати схеми для перевірки форматів та типів введених користувачем даних. Зручний API Zod спрощує налаштування валідації для форм і об’єктів, роблячи процес створення перевірок даних інтуїтивним та ефективним. Zod також інтегрується з TypeScript, що дозволяє використовувати типи, визначені в схемах Zod, для автодоповнення та перевірки типів під час розробки. Це знижує кількість помилок, пов’язаних із неправильними даними, ще до виконання програми та підвищує якість коду. [21]

В якості бібліотеки для управління формами було використано React Hook Form. Вона ефективно керує станом форм, мінімізуючи кількість повторних

ререндерів під час введення даних, що підвищує продуктивність додатка. React Hook Form інтегрується з бібліотеками валідації, такими як Zod, що дозволяє обробляти валідацію даних безпосередньо в формах. Це спрощує тестування та підтримку коду, особливо у великих проектах, де управління формами може бути складним. Зручний API React Hook Form також спрощує налаштування форм для обробки помилок та надсилання даних, роблячи процес роботи з формами швидшим та простішим. [22]

В якості бібліотеки для управління даними, що надходять із зовнішніх API, було використано React Query. Вона забезпечує простий та зручний спосіб роботи з запитами до API, включаючи автоматичне кешування, синхронізацію та повторні запити, що підвищує загальну продуктивність додатка. Завдяки автоматичному кешуванню React Query дозволяє уникнути надлишкових запитів до серверу, зменшуючи навантаження на нього та забезпечуючи швидкий доступ до вже завантажених даних. Крім того, React Query дозволяє налаштовувати поведінку запитів відповідно до потреб проекту, включаючи повторення запитів у випадку помилок, що підвищує надійність роботи з даними в додатку. [23]

### 2.4.4 Back end інструменти

В якості платформи для серверної частини було використано Node.js. Node.js – це серверне середовище, яке працює на основі JavaScript і побудоване на движку V8 від Google. Використання Node.js дозволяє розробляти швидкі й масштабовані серверні додатки завдяки подієво-орієнтованій архітектурі та неблокуючому введенню/виведенню (I/O). Ці особливості роблять Node.js ідеальним для роботи з реальним часом, де важлива швидкість обробки даних і висока продуктивність. Крім того, оскільки як фронтенд, так і бекенд написані на JavaScript, це спрощує спільну роботу над проектом і дозволяє використовувати один стек для обох частин додатка, що підвищує ефективність розробки та підтримки проекту.

В якості фреймворку для розробки серверної логіки було використано NestJS. Це TypeScript-фреймворк, який спрощує побудову модульних та

підтримуваних серверних додатків завдяки своїй архітектурі, яка базується на декораторах та концепції залежностей. NestJS підтримує ін’єкцію залежностей, що робить код більш організованим і легко розширюваним. Зручна структура та модульна архітектура NestJS дозволяють розбивати проект на окремі функціональні блоки, що спрощує тестування, налагодження та розширення функціональності. Крім того, фреймворк має потужну підтримку для роботи з REST API та GraphQL, що дозволяє легко інтегрувати його з будь-якими фронтенд-додатками або сторонніми сервісами, забезпечуючи високу сумісність та гнучкість проекту.

В якості бази даних було використано MongoDB. Це документоорієнтована база даних, яка працює на основі JSON-подібних документів, що робить її ідеальною для роботи з даними, що можуть змінювати свою структуру. Також MongoDB є ідеальним варіантом, коли потрібно абстрагуватись від складності реалізації відношень між таблицями, як от в реляційних базах дних, що було основним критерієм. MongoDB забезпечує високу швидкість зчитування та запису, що особливо важливо для додатків з великим обсягом даних і високою частотою запитів. Оскільки MongoDB не використовує традиційні SQL-таблиці, а працює з гнучкими документами, це дозволяє зберігати дані без необхідності жорсткого визначення схем, що спрощує масштабування та зміну структури даних в процесі розвитку додатка. Крім того, MongoDB підтримує масштабування через шардінг, що робить її оптимальним вибором для проєктів з високим навантаженням.

В якості ODM (Object-Document Mapping) бібліотеки для роботи з MongoDB було використано Mongoose. Mongoose дозволяє визначати схеми для документів MongoDB, що підвищує надійність та організованість даних, надаючи чіткі типи та структури для кожного документа. За допомогою Mongoose легко реалізувати валідацію даних та налаштування різних обмежень на рівні моделі, що забезпечує більшу відповідність вимогам додатка. Крім того, Mongoose надає зручний інтерфейс для обробки запитів до MongoDB, включаючи функціонал для пошуку, оновлення та видалення даних, що значно

спрощує роботу з базою даних та покращує організацію коду, особливо в проектах з великою кількістю складних запитів.

### 2.4.5 Криптографічні інструменти

В якості алгоритму хешування для захисту паролів на бекенді було використано Argon2. Argon2 – це сучасний та безпечний алгоритм хешування паролів, створений з урахуванням вимог до стійкості проти атак методом перебору (brute-force). Він дозволяє налаштувати витрати пам’яті та обчислень, що значно підвищує складність атаки. На відміну від таких алгоритмів, як MD5 чи SHA-256, які менш ресурсозатратні, Argon2 забезпечує захист від розподілених атак на апаратному рівні (зокрема, на графічних процесорах). Цей алгоритм забезпечує надійне хешування паролів перед збереженням їх у базі даних, тим самим захищаючи їх у разі компрометації даних.

В якості бібліотеки для роботи з аутентифікацією на основі токенів було обрано passport-jwt. Ця бібліотека дозволяє реалізувати аутентифікацію користувачів за допомогою JSON Web Token (JWT), що дає змогу створювати безпечний і масштабований механізм доступу до ресурсу. JWT-токени можуть зберігатися в браузері, а їх структура дозволяє закодувати інформацію про користувача та термін дії токена, забезпечуючи гнучкий контроль доступу. Крім того, використання токенів знижує навантаження на сервер, оскільки дозволяє зберігати сесії клієнтів у вигляді токенів замість створення сесій на сервері.

В якості методу для аутентифікації локальних користувачів було використано passport-local. Ця стратегія дозволяє реалізувати автентифікацію за допомогою звичайної пари "ім’я користувача – пароль", що особливо корисно для внутрішніх застосунків або проектів, де не потрібна зовнішня ідентифікація через соціальні мережі. Використання passport-local забезпечує зручний і надійний спосіб перевірки особистих даних користувача під час входу до системи, а також інтеграцію з іншими стратегічними обробниками passport, якщо потрібно розширити набір методів аутентифікації.

На фронтенді для додаткового захисту використовувався алгоритм SHA-256 із бібліотеки crypto-js. SHA-256 хешує пароль перед його відправкою на сервер, що допомагає уникнути передачі паролю у відкритому вигляді та забезпечує базовий рівень захисту під час передачі даних. Використання SHA-256 на фронтенді знижує ризик перехоплення паролю при передачі, навіть якщо зловмисник отримає доступ до мережевого трафіку, оскільки фактичний пароль залишається прихованим.

Для шифрування та дешифрування даних було створено власний алгоритм DynamicEncryptor. Він генерує ключі шифрування, які включають динамічний компонент, заснований на поточному часі, що ускладнює передбачення ключа. Цей алгоритм використовує XOR-операції для блочного шифрування, де кожен блок шифрується з урахуванням динамічного ключа, що підвищує безпеку. Однією з переваг цього алгоритму є гнучкість налаштувань блочного розміру та динамічного ключа, що знижує ймовірність успішного злому методом перебору.

Порівняно з AES, власний алгоритм шифрування надає більшу гнучкість в налаштуванні, що дозволяє оптимізувати його для специфічних потреб проекту. AES – це потужний і надійний алгоритм, але він має стандартну структуру, що може бути передбачуваною для деяких типів атак. Використання динамічного компоненту і специфічних блочних операцій надає цьому алгоритму більше варіантів для заплутування даних, що робить його менш вразливим до статистичних атак на ключ.

## 2.5 Реалізація вебсайту та його внутрішнє представлення

### 2.5.1 Розробка серверної частини

Структура back end частини виглядає наступним чином (рис. 2.5):

- main.ts – головний файл проекту;
- app.module.ts – головний модуль;
- ./common – папка для конфігураційних файлів та констант;
- ./modules – папка для модулів додатку;

- Dockerfile – файл з Docker конфігурацією;
- docker-compose.yaml – файл з docker-compose конфігурацією;
- .eslintrc.js – файл з налаштуваннями Eslint;

```
| - src/
    | - main.ts
    | - app.module.ts
    | - common/
    |   | - config/
    |   | - db/
    |   |   | - mongodb/
    |   |   |   | - configuration/
    |   |   |   |   | - mongoose.config.ts
    |   |   |   | - utils/
    |   |   |   |   | - {util-name}.util.ts
    |   | - decorators/
    |   |   | - {decorator-name}.decorator.ts
    |   | - pipes/
    |   |   | - {pipe-name}.pipe.ts
    |   | - utils/
    |   |   | - {util-name}.util.ts
    | - modules/
    |   | - {module-name}/
    |   |   | - {module-name}.module.ts
    |   |   | - {module-name}.controller.ts
    |   |   | - {module-name}.service.ts
    |   |   | - {module-name}.repository.ts
    |   |   | - dtos *
    |   |   | - schemas *
    |   |   | - strategies *
    |   |   | - guards *
    |   |   | - types *
    |   |   | - utils *
    |   |   | - config *
| - Dockerfile
| - docker-compose.yaml
| - .eslintrc.js
| - .prettierrc
| - package.json
| - pnpm-lock-yaml
| - tsconfig.json
| - README.md
| - .env
| - .env.sample
```

Рисунок 2.5 – Структура back end проєкту

- .prettierrc – файл з налаштуваннями prettier;
- package.json – файл в якому зберігаються назви залежностей, їхні версії і команди для запуску додатку, тестів та інші;
- pnpm-lock.yaml – файл створений менеджером пакетів pnpm. Він містить повний, розшифрований список усіх залежностей та їхніх версій, включно з усіма вкладеними залежностями;
- tsconfig.json – файл з налаштуваннями для компілятору TypeScript;
- README.md – файл з короткою документацією по проекту, де описується як запустити додаток та базові команди, які доступні;
- .env – файл зі змінними оточення. Даний файл не повинен потрапити на сервіси для хостингу коду, такі як GitHub, або GitLab;
- .env.sample – файл зі прикладами змінних оточення. Даний файл потрібний для того, щоб на його основі створити файл .env.

Головний файл серверної частини – main.ts. В даному файлі ми імпортуємо всі потрібні модулі, бібліотеки та конфігурації для роботи.

```
import { NestFactory } from '@nestjs/core';
import { ValidationPipe } from '@nestjs/common';
import * as cookieParser from 'cookie-parser';
import { AppModule } from './app.module';
import { corsOptions } from './common/config/cors.config';
```

Після чого ініціалізуємо bootstrap() функцію, яка є головною і запускає додаток.

```
async function bootstrap(): Promise<void> {
  const app = await NestFactory.create(AppModule);

  app.use(cookieParser());
  app.enableCors(corsOptions);
  app.useGlobalPipes(
    new ValidationPipe({ whitelist: false, forbidNonWhitelisted:
false }),
  );
  await app.listen(process.env.PORT || 3001);
}
bootstrap();
```

Наступним кроком потрібно налаштувати головний модуль додатку, підключити усі модулі і зробити потрібні налаштування бібліотек.

```
import { Module } from '@nestjs/common';
import { MongooseModule } from '@nestjs/mongoose';
import { ConfigModule } from '@nestjs/config';

import { VaultModule } from './modules/vault/vault.module';
import { UserModule } from './modules/user/user.module';
import { mongooseConfig } from './common/db/mongodb/configuration/
mongoose.config';
import { AuthModule } from './modules/auth/auth.module';
import { JwtConfig } from './modules/auth/config/jwt.config';

@Module({
  imports: [
    ConfigModule.forRoot({
      isGlobal: true,
      load: [JwtConfig],
    }),
    MongooseModule.forRoot(process.env.MONGODB_URL, mongooseConfig),
    AuthModule,
    VaultModule,
    UserModule,
  ],
})
export class AppModule {}
```

Тепер перейдемо до конфігурацій cors та cookies, які знаходяться в папці /common/config та конфігурації mongoose, яка знаходиться в /common/db/mongodb/configuration.

```
import { URL_ORIGIN } from './constants';

export const corsOptions = {
  origin: URL_ORIGIN,
  credentials: true,
};

export const COOKIE_DOMAIN = process.env.COOKIE_DOMAIN || 'localhost';

export const mongooseConfig = {
  dbName: process.env.MONGODB_DATABASE_NAME || 'password-manager',
  auth: {
    username: process.env.MONGODB_USER_NAME,
```

```
    password: process.env.MONGODB_USER_PASSWORD,
  },
};
```

В даному випадку авторизацію я реалізував за допомогою JWT токену, який генерується у відповідь на успішну реєстрацію, або всіх у систему. Для початку потрібно створити методи в контролері, які будуть обробляти запити на реєстарцію, вхід, валідацію користувача, отримання інформації для поточного користувачі на основі токену і вихід. Для того, щоб створити обробляти ендпоінти в контролері, нам просто потрібно передати базовий шлях в деоратор “@Controller”, або якщо нам потрібен унікальний шлях, ми можемо це зробити додавши його в декоратор HTTP методу, який надає Nest з коробки.

```
@Controller('auth')
export class AuthController {
  constructor(private readonly authService: AuthService) {}

  @Post('sign-up')
  async signUp(
    @Body() signUpDto: SignUpSignInDto,
    @Res({ passthrough: true }) response: Response,
  ): Promise<AuthResponseDto> {
    return this.authService.signUp(signUpDto, response);
  }

  @Post('sign-in')
  async signIn(
    @Body() signInDto: SignUpSignInDto,
    @Res({ passthrough: true }) response: Response,
  ): Promise<AuthResponseDto> {
    return this.authService.signIn(signInDto, response);
  }

  @Post('sign-out')
  async signOut(@Res({ passthrough: true }) response: Response):
Promise<void> {
    return this.authService.signOut(response);
  }

  @UseGuards(JwtGuard)
  @Get('check-auth')
  async checkAuth(): Promise<{ isAuth: true }> {
    return { isAuth: true };
  }
```

```
  @UseGuards(JwtGuard)
  @Post('current-user')
  async getCurrentUser(
    @Request() req: ExpressRequest & { user: AuthPayload },
  ): Promise<GetCurrentUserResponseDto> {
    return this.authService.getCurrentUser(req.user.email);
  }
}
```

Після чого необхідно створити аналогічні методи в сервісі авторизації. Для того щоб створити сервіс, потрібно використати "@Injectable" декоратор, який дозволяє пізніше використовувати принцип Dependency Injection (DI). Варто зазначити, що сервіси в контексті NestJs використовуються для інкапсуляції бізнес логіки.

```
@Injectable()
export class AuthService {
  constructor(
    private readonly userService: UserService,
    private readonly vaultService: VaultService,
    private readonly jwtService: JwtService,
  ) {}

  async signUp(
    signUpDto: SignUpSignInDto,
    response: Response,
  ): Promise<AuthResponseDto> {
    const existingUser = await
this.userService.findOneByEmail(signUpDto.email);

    if (existingUser) {
      throw new ConflictException('Email is already taken');
    }

    const { _id: userId, email } = await
this.userService.create(signUpDto);

    const salt = generateSalt();

    const vault = await this.vaultService.create({ userId, salt });

    const accessToken = await this.jwtService.signAsync({
      id: userId,
      email,
    });
```

```
    response.cookie('access-token', accessToken, {
      domain: COOKIE_DOMAIN,
      path: '/',
      secure: false,
      httpOnly: true,
      sameSite: false,
    });

    return {
      vault: vault.data,
      salt,
    };
  }

  async signIn(
    signInDto: SignUpSignInDto,
    response: Response,
  ): Promise<AuthResponseDto> {
    const user = await this.validateUser(signInDto.email,
signInDto.password);

    if (!user) {
      throw new UnauthorizedException();
    }

    const { _id: userId, email } = user;

    const vault = await this.vaultService.findOneByUserId(userId);

    const accessToken = await this.jwtService.signAsync({
      id: userId,
      email,
    });

    response.cookie('access-token', accessToken, {
      domain: COOKIE_DOMAIN,
      path: '/',
      secure: false,
      httpOnly: true,
      sameSite: false,
    });

    return {
      vault: vault?.data,
      salt: vault?.salt,
    };
  }

  async signOut(response: Response): Promise<void> {
    response.clearCookie('access-token', {
```

```
      domain: COOKIE_DOMAIN,
      path: '/',
      secure: false,
      httpOnly: true,
      sameSite: false,
    });
  }

  async validateUser(email: string, password: string): Promise<UserDocument> {
    const user = await this.userService.findOneByEmail(email);

    if (!user) {
      throw new NotFoundException('User with specified email does not exists');
    }

    const isValidPassword = await verify(user.password, password);

    if (!isValidPassword) {
      throw new BadRequestException('Incorrect password');
    }

    return user;
  }

  async getCurrentUser(userEmail: string): Promise<GetCurrentUserResponseDto> {
    const user = await this.userService.findOneByEmail(userEmail);

    if (!user) {
      throw new UnauthorizedException();
    }
    const { id, email } = user;

    return { id, email };
  }
}
```

Пізніше, для валідації користувача використовуються стратегія, яка являє собою наслідника класу PassportStrategy і після перевантаження методу validate дозволяє валідувати JWT токен так, як вимагає цього додаток. Також для того щоб використовувати динамічно дану стратегію в декораторі "@UseGuards", потрібно створити гуард (guard):

```
@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor(private readonly configService: ConfigService) {
    super({
      jwtFromRequest: accessTokenCookieExtractor,
      ignoreExpiration: false,
      secretOrKey: configService.get<string>('secret'),
    });
  }

  validate = async ({ id, email }: AuthPayload): Promise<AuthPayload> =>
{
    return { id, email };
  };
}

@Injectable()
export class JwtGuard extends AuthGuard('jwt') {}
```

Під час реєстрації і входу, також відбуваються виклики методів інших сервісів для створення сховища облікових записів, для створення користувача, для перевірки існування сховища облікових записів, чи користувача:

```
@Injectable()
export class VaultService {
  constructor(private readonly vaultRepository: VaultRepository) {}

  async create(createVaultDto: CreateVaultDto): Promise<VaultDocument>
{
    const createdVault = await
this.vaultRepository.create(createVaultDto);

    return createdVault;
  }

  async update(
    userId: ObjectId,
    updateVaultDto: UpdateVaultDto,
  ): Promise<VaultDocument> {
    const updatedVault = await this.vaultRepository.update(
      userId,
      updateVaultDto,
    );

    return updatedVault;
  }

  async findOneByUserId(userId: ObjectId): Promise<VaultDocument> {
```

```
    const foundVault = await
this.vaultRepository.findVaultByUserId(userId);

    return foundVault;
  }
}

@Injectable()
export class UserService {
  constructor(private readonly userRepository: UserRepository) {}

  async findOneByEmail(userEmail: string): Promise<UserDocument> {
    const foundUser = await
this.userRepository.findOneByEmail(userEmail);

    return foundUser;
  }

  async create(createUserDto: CreateUserDto): Promise<UserDocument> {
    const createdUser = await this.userRepository.create(createUserDto);

    return createdUser;
  }
}
```

Ми використовуємо MongoDB, як базу даних, для взаємодії з нею використовується ODM. Усю логіку взаємодії з БД містять репозиторії.

```
@Injectable()
export class VaultRepository {
  constructor(
    @InjectModel(Vault.name) private readonly vaultModel:
Model<VaultDocument>,
  ) {}

  async create(createVaultDto: CreateVaultDto): Promise<VaultDocument>
{
    const vault = new this.vaultModel(createVaultDto);
    const savedVault = await vault.save();

    return savedVault;
  }

  async update(
    userId: ObjectId,
    updateVaultDto: UpdateVaultDto,
  ): Promise<VaultDocument> {
    const updatedVault = await this.vaultModel.findOneAndUpdate(
      { userId },
```

```
      { data: updateVaultDto.encryptedVault },
      { new: true },
    );

    return updatedVault;
  }

  async findVaultByUserId(userId: ObjectId): Promise<VaultDocument> {
    const foundVault = await this.vaultModel.findOne({ userId });

    return foundVault;
  }
}

@Injectable()
export class UserRepository {
  constructor(
    @InjectModel(User.name)      private      readonly      userModel:
Model<UserDocument>,
  ) {}

  async create(createUserDto: CreateUserDto): Promise<UserDocument> {
    const user = new this.userModel(createUserDto);
    const savedUser = await user.save();

    return savedUser;
  }

  async findOneByEmail(userEmail: string): Promise<UserDocument> {
    const [user] = await this.userModel.find({
      email: userEmail,
    });

    return user;
  }
}
```

Оскільки ми використовуємо MongoDB в парі з Mongoose, то для того щоб створити схему документів, які будуть зберігатись в базі даних, ми можемо використати декоратор ”Schema” з бібліотеки nestjs/mongoose. В клас, де ми створюємо схему, ми також можемо виконувати й інші дії, як от хешування паролю перед записом в базу даних.

```
export type UserDocument = User & Document;

@Schema({ timestamps: true })
```

```
export class User {
  @Prop({ unique: true, required: true })
  email: string;

  @Prop({ required: true })
  password: string;
}

const UserSchema = SchemaFactory.createForClass(User);

UserSchema.index({});

UserSchema.pre<UserDocument>('save', async function (next) {
  if (this.isModified('password') || this.isNew) {
    const hashedPassword = await hash(this.password);
    this.password = hashedPassword;
  }
  next();
});

export { UserSchema };

export type VaultDocument = Vault & Document;

@Schema({ timestamps: true })
export class Vault {
  @Prop({ type: mongoose.Schema.Types.ObjectId, ref: 'User' })
  userId: User;

  @Prop({ default: '' })
  data: string;

  @Prop({ required: true })
  salt: string;
}

const VaultSchema = SchemaFactory.createForClass(Vault);

VaultSchema.index({});

export { VaultSchema };
```

### 2.5.2 Розробка клієнтської частини

Структура front end частини виглядає наступним чином (рис. 2.6):

- src/app/page.tsx – головний файл проекту;
- src/api/index.tsx – файл для взаємодії з API;

- src/app – папка для створення роутів всередині додатку використовуючи патерни, які пропоную документація Next.js;
- src./components – папка для перевикористовуваних компонентів;
- src/cdk – папка для головних елементів додатку, таких як утиліти, константи, типи, хуки, провайдери;
- src/cdk/constants – папка для глобальних констант;
- src/cdk/hooks – папка для кастомних хуків;
- src/cdk/libs – папка для екземплярів бібліотек;
- src/cdk/providers – папка для провайдерів, тобто для компонентів вищого порядку, які будуть огортати інші компоненти власною логікою;
- src/cdk/types – папка для глобальних типів;
- src/cdk/utils – папка для глобальних утиліт;
- src/middleware – файл для проміжного обробника на стороні серверу;
- .eslintrc.js – файл з налаштуваннями Eslint;
- .prettierrc – файл з налаштуваннями prettier;
- package.json – файл в якому зберігаються назви залежностей, їхні версії і команди для запуску додатку, тестів та інші;
- pnpm-lock.yaml – файл створений менеджером пакетів pnpm. Він містить повний, розшифрований список усіх залежностей та їхніх версій, включно з усіма вкладеними залежностями;
- tsconfig.json – файл з налаштуваннями для компілятору TypeScript;
- README.md – файл з короткою документацією по проекту, де описується як запустити додаток та базові команди, які доступні;
- .env – файл зі змінними оточення. Даний файл не повинен потрапити на сервіси для хостингу коду, такі як GitHub, або GitLab;
- .env.sample – файл зі прикладами змінних оточення. Даний файл потрібний для того, щоб на його основі створити файл .env.

Головна сторінка клієнтської частини – src/app/page.tsx. В даному файлі спочатку імпортуються основні бібліотеки та компонент VaultDashboard, який представляє інтерфейс основного функціоналу додатка. Зокрема,

використовується NextPage для визначення типу сторінки, Head для додавання мета-даних до HTML-документа, а також cookies з next/headers для роботи з cookies на стороні сервера.

```
| - src/
    | - api/
    |   | - index.ts
    | - app/
    |   | - (auth)/
    |   |   | - components/
    |   |   |   | - {component-name}/
    |   |   |   |   | - {component-name}.tsx
    |   |   |   | - {route-name}/
    |   |   |   |   | - components/
    |   |   |   |   | - types/
    |   |   |   |   | - page.tsx
    |   | - page.tsx
    |   | - layout.tsx
    |   | - global.css
    | - cdk/
    |   | - constants/
    |   | - hooks/
    |   | - lib/
    |   | - providers/
    |   | - types/
    |   | - utils/
    | - components/
    |   | - shared/
    |   | - ui/
    | - middleware.ts
| - .eslintrc.js
| - .prettierrc
| - package.json
| - pnpm-lock-yaml
| - tsconfig.json
| - README.md
| - .env
| - .env.sample
```

Рисунок 2.6 – Структура front end проекту

У функціональному компоненті Home за допомогою cookies().get(...) отримуються дані cookies, які зберігають налаштування інтерфейсу — layout для

розташування панелей і collapsed для стану згорнутого/розгорнутого вигляду панелей. Значення цих cookies зберігаються у вигляді JSON, тому їх необхідно розпарсити за допомогою JSON.parse(...). Якщо даних немає, змінні defaultLayout і defaultCollapsed залишаються невизначеними (undefined).

Далі повертається JSX-код для сторінки. У компоненті Head встановлюються мета-дані для HTML-документа. В основному контейнері <main> відображається компонент VaultDashboard, якому передаються отримані значення defaultLayout і defaultCollapsed для відновлення попередніх налаштувань інтерфейсу.

```
import { NextPage } from 'next';
import Head from 'next/head';
import { cookies } from 'next/headers';
import VaultDashboard from '@/app/components/vault-dashboard/vault-dashboard';

const Home: NextPage = () => {
  const layout = cookies().get('react-resizable-panels:layout:mail');
  const collapsed = cookies().get('react-resizable-panels:collapsed');
  const defaultLayout = layout ? JSON.parse(layout.value) : undefined;
  const defaultCollapsed = collapsed ? JSON.parse(collapsed.value) : undefined;
  return (
    <div>
      <Head>
        <title>Password Manager</title>
        <meta name='description' content='Password Manager' />
        <link rel='icon' href='/favicon.ico' />
      </Head>
      <main>
        <VaultDashboard defaultLayout={defaultLayout} defaultCollapsed={defaultCollapsed} navCollapsedSize={4} />
      </main>
    </div>
  );
};

export default Home;
```

В якості HTTP клієнта ми використовуємо бібліотеку Axios. В файлі axios-instance.lib.ts ми створюємо конфігурацію для Axios. Основна мета цього файлу – забезпечити централізовану конфігурацію для всіх запитів у додатку,

включаючи встановлення базового URL, таймауту, а також передачу токену авторизації, якщо він є в cookies.

```
‘use client’;

import axios from ‘axios’;

const baseUrl = process.env.NEXT_PUBLIC_API_ENDPOINT || ‘http://localhost:3001’;

const axiosInstance = axios.create({
  baseURL: baseUrl,
  withCredentials: true,
  timeout: 1000,
});

axiosInstance.interceptors.request.use(
  async function (config) {
    const isServer = typeof window === ‘undefined’;

    if (isServer) {
      const { cookies } = await import(‘next/headers’);

      const token = cookies().get(‘access-token’)?.value;

      if (token) {
        // eslint-disable-next-line no-param-reassign
        config.headers.Authorization = token;
      }
    }

    return config;
  },

  function (error) {
    return Promise.reject(error);
  }
);

export default axiosInstance;
```

Тепер axiosInstance ми модемо зручно використовувати при створенні API запитів.

```
import axiosInstance from ‘@/cdk/lib/axios-instance.lib’;

export const signUp = async (payload: {
```

```
  password: string;
  email: string;
}): Promise<{ salt: string; vault: string; accessToken: string }> => {
  return await axiosInstance
    .post<{     salt:     string;     vault:     string;     accessToken:
string }>('auth/sign-up', payload)
    .then((res) => res.data);
};

export const signIn = async (payload: {
  email: string;
  password: string;
}): Promise<{ salt: string; vault: string; accessToken: string }> => {
  return axiosInstance
    .post<{     salt:     string;     vault:     string;     accessToken:
string }>('auth/sign-in', payload)
    .then((res) => res.data);
};

export const signOut = async (): Promise<void> => {
  return axiosInstance.post('auth/sign-out').then(() => {
    sessionStorage.removeItem('vault');
    sessionStorage.removeItem('vault-key');
  });
};

export const checkAuth = async (): Promise<{ isAuth: true }> => {
  return axiosInstance.get('auth/check-auth').then((res) => res.data);
};

export const getCurrentUser = async (): Promise<{ id: string; email:
string }> => {
  return axiosInstance.post('auth/current-user').then((res) => res.data);
};

export const saveVault = async ({ encryptedVault }: { encryptedVault:
string }): Promise<void> => {
  return axiosInstance.put('vault', { encryptedVault }).then((res) =>
res.data);
};
```

В даному випадку, було використано React Context для глобального сховища стану. Контекст VaultContext використовується для збереження та оновлення даних сховища (vault) і його ключа (vaultKey) у додатку. Компонент VaultProvider перевіряє, чи знаходиться користувач на захищеній сторінці, і якщо немає валідного ключа сховища, здійснює вихід і перенаправляє його на сторінку входу. Функція refresh зчитує vault і vaultKey із sessionStorage. Якщо даних

немає, а користувач на захищеній сторінці, виконується вихід. Поки vault або vaultKey оновлюються, відображається значок завантаження. Контекст VaultContext робить дані сховища доступними для всіх дочірніх компонентів.

```
‘use client’;
import { usePathname, useRouter } from ‘next/navigation’;
import React, { createContext, useCallback, useEffect, useState } from
‘react’;
import { signOut } from ‘@/api’;
import { Vault } from ‘@/cdk/types/vault.type’;
import { Icons } from ‘@/components/ui/icons’;
export type VaultContextType = {
  vault: Vault;
  vaultKey: string;
  refresh: () => Promise<void>;
};

export const VaultContext = createContext<VaultContextType |
undefined>(undefined);

const protectedRoutes = [‘/’];
const publicRoutes = [‘/sign-in’, ‘sign-up’];

export const VaultProvider: React.FC<{ children: React.ReactNode }> =
({ children }) => {
  const pathname = usePathname();
  const router = useRouter();
  const isProtectedRoute = protectedRoutes.includes(pathname);
  const isPublicRoute = publicRoutes.includes(pathname);

  const [vault, setVault] = useState<Vault>({});
  const [vaultKey, setVaultKey] = useState<string>(‘‘);
  const [isVaultUpdating, setIsVaultUpdating] = useState(true);
  const [isSigningOut, setIsSigningOut] = useState(false);

  const memoizedRefresh = useCallback(refresh, [isProtectedRoute,
isPublicRoute, router]);

  useEffect(() => {
    void memoizedRefresh();
  }, [memoizedRefresh]);

  async function refresh(): Promise<void> {
    setIsVaultUpdating(true);
    const storedVault = window.sessionStorage.getItem(‘vault’);
    const storedVaultKey = window.sessionStorage.getItem(‘vault-key’);

    if (storedVault) {
```

```
      setVault(JSON.parse(storedVault));
    }
    if (storedVaultKey) {
      setVaultKey(storedVaultKey);
    }
    if (!storedVaultKey) {
      console.error('Error updating vault: No vault or vault key were
found');
      if (isProtectedRoute) {
        setIsSigningOut(true);

        await signOut().then(() => {
          router.push('/sign-in');
        });
      } else if (isPublicRoute) {
        setIsSigningOut(false);
      }
    }
    setIsVaultUpdating(false);
  }

  if (isVaultUpdating || isSigningOut) {
    return (
      <div className='h-screen flex items-center justify-center'>
        <Icons.spinner className='animate-spin' />
      </div>
    );
  }

  return (
    <VaultContext.Provider    value={{    vault,    vaultKey,    refresh:
memoizedRefresh }}>{children}</VaultContext.Provider>
  );
};
```

Однією із головних функцій, яку повинен надавати менеджер паролів це генерування складних паролів. Пароль має генеруватись випадковим чином на основі заданих опцій. Спочатку створюються масиви символів: малі та великі літери, цифри і спеціальні символи. Функція generatePassword приймає об'єкт параметрів, який визначає довжину пароля і чи включати в нього цифри та символи. Для кожного типу символів, що має бути включений, функція додає один випадковий символ цього типу, забезпечуючи мінімальні вимоги. Далі пароль доповнюється випадковими символами до досягнення потрібної

довжини. Після цього символи перемішуються для уникнення передбачуваної послідовності.

```
const lowerCasedAlphabets = [...’abcdefghijklmnopqrstuvwxyz’.split(‘‘)];
const upperCasedAlphabets = lowerCasedAlphabets.map((alphabet) =>
alphabet.toUpperCase());
const numbers = [...’1234567890’.split(‘‘)];
const symbols = [...’!"#$%&\’()*+,-./:;<=>?@[\\]^_`{|}~’.split(‘‘)];

const getRandomNumber = (max: number): number => Math.floor(Math.random()
* max);

interface GeneratePasswordOptions {
  passwordLength: number;
  isNumbersIncluded?: boolean;
  isSymbolsIncluded?: boolean;
}

export const generatePassword = (options: GeneratePasswordOptions): string
=> {
  const { passwordLength, isNumbersIncluded, isSymbolsIncluded } = options;
  const allCharacters = [...lowerCasedAlphabets, ...upperCasedAlphabets];

  if (isNumbersIncluded) {
    allCharacters.push(...numbers);
  }
  if (isSymbolsIncluded) {
    allCharacters.push(...symbols);
  }

  // Ensure password has at least one of each required type of character
  const generatedPassword = [];

generatedPassword.push(upperCasedAlphabets[getRandomNumber(upperCasedAlph
abets.length)]);

generatedPassword.push(lowerCasedAlphabets[getRandomNumber(lowerCasedAlph
abets.length)]);

  if (isNumbersIncluded) {
    generatedPassword.push(numbers[getRandomNumber(numbers.length)]);
  }

  if (isSymbolsIncluded) {
    generatedPassword.push(symbols[getRandomNumber(symbols.length)]);
  }

  // Fill the remaining part of the password with random characters
  for (let i = generatedPassword.length; i < passwordLength; i++) {
```

```
generatedPassword.push(allCharacters[getRandomNumber(allCharacters.length
)]);
  }

  // Shuffle the password to avoid a predictable pattern
  return generatedPassword.sort(() => Math.random() - 0.5).join(‘‘);
};
```

Шифрування та розшифрування є найважливішими аспектами в додатках такого типу, тому було розроблено власний клас DynamicEncryptor, який забезпечує простий механізм шифрування та дешифрування текстових даних, використовуючи ключ, який формується на основі заданого vaultKey та динамічного компонента, пов’язаного з часом. Клас містить методи для роботи з масивами байтів, блочне шифрування з XOR-операцією, генерування ініціалізаційного вектора (IV) і керування підкладкою для блочного розміру.

Опис методів:

- generateKey: генерує ключ на основі вхідного vaultKey та динамічного компонента, який змінюється з кожним запуском. Ключ формується з першого символу vaultKey, доповненого числовим значенням поточного часу (в мілісекундах) за модулем 1000. Ключ перетворюється на масив байтів за допомогою методу stringToByteArray;
- stringToByteArray: перетворює рядок на масив байтів, де кожен символ конвертується в ASCII-значення та додається до масиву. Це дозволяє працювати з текстом як з масивом чисел;
- byteArrayToString: зворотний метод stringToByteArray, що перетворює масив байтів назад у рядок;
- encryptBlock: шифрує блок даних XOR-операцією. Для кожного елемента блоку застосовується XOR з відповідним елементом ключа, повторюючи ключ при необхідності. Це забезпечує базове шифрування;
- decryptBlock: дешифрує блок, повторюючи XOR-операцію з тим самим ключем, оскільки для XOR операції шифрування і дешифрування ідентичні;

- generateIV: генерує ініціалізаційний вектор (IV) потрібної довжини (16 байтів), де кожен байт є випадковим числом від 0 до 255. IV забезпечує випадковість у результатах шифрування;

- padData: додає підкладку до даних, щоб забезпечити їх відповідність блочному розміру. Підкладка доповнюється числом, яке вказує кількість доданих байтів;

- unpadData: видаляє підкладку, повертаючи оригінальні дані. Зчитує останній байт як кількість байтів підкладки та видаляє їх з кінця масиву.

Шифрування та дешифрування

- encrypt: метод шифрування, який приймає текстові дані та vaultKey. Спочатку генерує ключ і ініціалізаційний вектор (IV), потім перетворює текст на байти і додає підкладку для відповідності блочному розміру. Після цього кожен блок обробляється методом encryptBlock, результати зберігаються як зашифровані байти. IV додається на початок зашифрованого масиву, який перетворюється у Base64 для зручного зберігання;

- decrypt: метод дешифрування, який приймає Base64-кодовані зашифровані дані та vaultKey. Після декодування Base64 метод виділяє IV з початку масиву, зчитує зашифровані дані та дешифрує кожен блок за допомогою decryptBlock. Видаляє підкладку та повертає розшифрований текст.

```
class DynamicEncryptor {
  private ivLength: number = 16;
  private blockSize: number = 16;

  private generateKey(vaultKey: string): number[] {
    const dynamicComponent = (new Date().getTime() % 1000).toString();
    const key = (vaultKey + dynamicComponent).slice(0, 16);
    return this.stringToByteArray(key);
  }

  private stringToByteArray(str: string): number[] {
    const byteArray = [];
    for (let i = 0; i < str.length; i++) {
      byteArray.push(str.charCodeAt(i) & 0xff);
    }
    return byteArray;
  }
```

```
private byteArrayToString(byteArray: number[]): string {
  return String.fromCharCode.apply(null, byteArray);
}

private encryptBlock(block: number[], key: number[]): number[] {
  const encryptedBlock = [];
  for (let i = 0; i < block.length; i++) {
    encryptedBlock[i] = block[i] ^ key[i % key.length];
  }
  return encryptedBlock;
}

private decryptBlock(block: number[], key: number[]): number[] {
  return this.encryptBlock(block, key);
}

private generateIV(): number[] {
  const iv = [];
  for (let i = 0; i < this.ivLength; i++) {
    iv.push(Math.floor(Math.random() * 256));
  }
  return iv;
}

private padData(data: number[]): number[] {
  const paddingLength = this.blockSize - (data.length % this.blockSize);
  const paddedData = data.slice();
  for (let i = 0; i < paddingLength; i++) {
    paddedData.push(paddingLength);
  }
  return paddedData;
}

private unpadData(data: number[]): number[] {
  const paddingLength = data[data.length - 1];
  return data.slice(0, data.length - paddingLength);
}

public encrypt(data: string, vaultKey: string): string {
  const key = this.generateKey(vaultKey);
  const iv = this.generateIV();
  const byteData = this.stringToByteArray(data);
  const paddedData = this.padData(byteData);

  const encryptedData: number[] = [];
  for (let i = 0; i < paddedData.length; i += this.blockSize) {
    const block = paddedData.slice(i, i + this.blockSize);
    const encryptedBlock = this.encryptBlock(block, key);
    encryptedData.push(...encryptedBlock);
  }
```

```
    const result = [...iv, ...encryptedData];
    return btoa(this.byteArrayToString(result)); // Кодування в Base64
  }

  public decrypt(data: string, vaultKey: string): string {
    const decodedData = this.stringToByteArray(atob(data)); // Декодування
з Base64
    const key = this.generateKey(vaultKey);

    const encryptedData = decodedData.slice(this.ivLength);

    const decryptedData: number[] = [];
    for (let i = 0; i < encryptedData.length; i += this.blockSize) {
      const block = encryptedData.slice(i, i + this.blockSize);
      const decryptedBlock = this.decryptBlock(block, key);
      decryptedData.push(...decryptedBlock);
    }

    const unpaddedData = this.unpadData(decryptedData);
    return this.byteArrayToString(unpaddedData);
  }
}

export const dynamicEncryptor = new DynamicEncryptor();
```

Пізніше використовуємо dynamicEncryptor в утлітах для шифрування та розшифрування vault. Також, в тому ж самому файлі створюємо утиліти для генерації vaultKey і хешування паролю.

```
import { SHA256 } from 'crypto-js';
import pbkdf2 from 'crypto-js/pbkdf2';

import { dynamicEncryptor } from '@/cdk/utils/dynamic-encryptor.util';

interface VaultWithVaultKey {
  vaultKey: string;
  vault: string;
}

interface MetaDataForVaultKey {
  email: string;
  hashedPassword: string;
  salt: string;
}

export const hashPassword = (password: string): string => {
```

```
  return SHA256(password).toString();
};

export const generateVaultKey = ({ email, hashedPassword, salt }:
MetaDataForVaultKey): string => {
  return pbkdf2(`${email}:${hashedPassword}`, salt, {
    keySize: 32,
    iterations: 10000,
  }).toString();
};

export const decryptVault = ({ vaultKey, vault }: VaultWithVaultKey):
string | null => {
  const decryptedVault = dynamicEncryptor.decrypt(vault, vaultKey);

  try {
    return JSON.parse(decryptedVault);
  } catch (e) {
    return null;
  }
};
export const encryptVault = ({ vaultKey, vault }: VaultWithVaultKey):
string => {
  return dynamicEncryptor.encrypt(JSON.stringify(vault), vaultKey);
};
```

## 2.6 Тестування та налагодженя додатку

Тестування – це невід’ємна частина процесу розробки програмного забезпечення, коли відбувається перевірка продукту на коректність роботи та наявність дефектів. Метою тестування є впевненість, що додаток функціонує належним чином у різних умовах і ситуаціях. Для цього перевірка має здійснюватися на різноманітних наборах даних. Тестування буває кількох типів:

- ручне тестування;
- напівавтоматизоване;
- автоматизоване.

Основний акцент під час тестування був зроблений на ручному тестуванні, що дало змогу перевірити коректність роботи додатку в різних сценаріях. В процесі розробки серйозних помилок не виникало, однак деякі труднощі були пов’язані зі створенням власного алгоритму шифрування та розшифрування. Додатковий час зайняло також дослідження принципів роботи менеджерів

паролів, щоб забезпечити надійне зберігання даних. Крім того, особлива увага була приділена типізації та структуризації коду, що потребувало більше часу, ніж очікувалося, але сприяло підвищенню його якості та зручності підтримки в майбутньому.

Додатково виконувалось тестування сумісності, яке показало відсутність проблем. Додаток працював у різних браузерах, таких як Google Chrome, Microsoft Edge та Mozilla Firefox.

cОскільки додаток запускається в локальному середовищі, перед запуском потрібно налаштувати його back end частину. Для того щоб налаштувати в локальному середовищі, потрібно спочатку встановити залежності (рис. 2.7), дана команда виконується з головної папки проекту.

```
pnpm install
```

Рисунок 2.7 – Команда для встановлення залежностей

Для конфігурації проекту, потрібно скопіювати файл .env.sample і на основі нього створити .env файл (рис. 2.8).

```
# Using for DB connection configuration
MONGODB_USER_NAME = 'your username'
MONGODB_USER_PASSWORD = 'your user password'
MONGODB_DATABASE_NAME = 'password-manager' // MongoDB database name
MONGODB_URL = 'mongodb://mongodb:27017' // Default MongoDB URL

# Using for better debugging
IS_VERBOSE_MODE = true

# Using for JWT token configuration
JWT_SECRET = 'your jwt secret'
JWT_EXPIRE_IN = '1h'

# Using for cookies configuration
COOKIE_DOMAIN = 'localhost'
```

Рисунок 2.8 – Приклад конфігураційного файлу .env

Після усіх налаштувань можна запустити проект за допомогою Docker (рис. 2.9).

```
# Running in dev mode
docker compose up
```

Рисунок 2.9 – Команда для запуску проекту з використанням Docker

Для того щоб запустити front end частину потрібно встановити залежності (рис. 2.10), створити конфігураційний файл .env (рис. 2.11), після чого ми можемо запустити проект (рис. 2.12).

```
pnpm install
```

Рисунок 2.10 – Команда для встановлення залежностей

```
# Base url for axios
NEXT_PUBLIC_API_ENDPOINT = 'http://localhost:3001'
```

Рисунок 2.11 – Приклад конфігураційного файлу .env

```
pnpm dev
```

Рисунок 2.12 – Команда для запуску проекту

Після усіх налаштувань і запуску, додаток готовий до використання.

# ВИСНОВКИ

У результаті виконання магістерської роботи були застосовані сучасні технології для розробки менеджера паролів, зокрема стек, що включає React, Next.js, NestJS та MongoDB. Для досягнення поставленої мети реалізовано як клієнтську, так і серверну частини вебсервісу, зосередившись на безпеці та зручності користування. Було розроблено власний алгоритм шифрування для захисту даних користувачів, що значно підвищило рівень безпеки завдяки шифруванню за допомогою динамічного ключа.

Практична реалізація проєкту базувалася на дотриманні найкращих практик структуризації та типізації коду, що забезпечило надійність, стабільність і легкість подальшого супроводу та розвитку системи.

В майбутньому можливе вдосконалення функціональних можливостей вебсервісу, додавання нових алгоритмів шифрування, а також оптимізація для підвищення продуктивності та розширення можливостей зберігання й обробки даних.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ


1. Sahu R. An Extensive Overview of Web Applications for Beginners. Medium. URL: https://medium.com/@riteshs4hu/an-extensive-overview-of-web-applications-for-beginners-1d260dbc3675 (дата звернення: 01.10.2024).
2. What is OSI Model | 7 Layers Explained. GeeksforGeeks. URL: https://www.geeksforgeeks.org/open-systems-interconnection-model-osi/ (дата звернення: 02.10.2024).
3. CAROLINE. Understanding the TCP/IP Model: The Backbone of Internet Communication. Medium. URL: https://medium.com/@gwenilorac/understanding-the-tcp-ip-model-the-backbone-of-internet-communication-aaa69b5b6595 (дата звернення: 03.10.2024).
4. Moraneus. A Comprehensive Guide to Docker. Medium. URL: https://medium.com/@moraneus/a-comprehensive-guide-to-docker-286d6f3ad122 (дата звернення: 04.10.2024).
5. LLC K. T. How to Set up Docker Compose: Step-By-Step Tutorial. Medium. URL: https://medium.com/@kaaiot/how-to-set-up-docker-compose-step-by-step-tutorial-9c339df67a2d (дата звернення: 04.10.2024).
6. Balian's techologies and innovation lab. Docker vs. Docker Compose: Simple and Fun Explanation. Medium. URL: https://medium.com/@ShantKhayalian/docker-vs-docker-compose-simple-and-fun-explanation-4811582127f7 (дата звернення: 04.10.2024).
7. Документація TypeScript. URL: https://www.typescriptlang.org/ (дата звернення: 07.10.2024).
8. Документація React. URL: https://react.dev/ (дата звернення: 07.10.2024).
9. React O. React Best Practices to Improve Your Code. Medium. URL: https://medium.com/@onix_react/react-best-practices-to-improve-your-code-a4c68962d5dd (дата звернення: 08.10.2024).

10. Документація Next.js. URL: https://nextjs.org/docs (дата звернення: 08.10.2024).
11. Truths T. React vs Next.js–In the Ring for Modern Web Development. Medium. URL: https://techtruths110.medium.com/react-vs-next-js-in-the-ring-for-modern-web-development-9d0871266f5c (дата звернення: 08.10.2024).
12. Документація NestJS. URL: https://docs.nestjs.com/ (дата звернення: 09.10.2024).
13. Vishnurathan A. MongoDB. Medium. URL: https://medium.com/@ananthvishnu/mongodb-b95e8364ceed (дата звернення: 09.10.2024).
14. isuru jayathilake. Introduction to Hashing. Medium. URL: https://medium.com/@isuruj/introduction-to-hashing-5b4daf343889 (дата звернення: 10.10.2024).
15. Johnson C. Introducing CipherX-256: A Robust Symmetric-Key Block Cipher. Medium. URL: https://medium.com/@christianjohnson_12732/introducing-cipherx-256-a-robust-symmetric-key-block-cipher-a728e1774ab4 (дата звернення: 10.10.2024).
16. Symmetric vs Asymmetric Encryption – What Are the Difference?. ClickSSL. URL: https://www.clickssl.net/blog/symmetric-encryption-vs-asymmetric-encryption (дата звернення: 10.10.2024).
17. Iterative Process Guide. Atlassian. URL: https://www.atlassian.com/work-management/project-management/iterative-process (дата звернення: 15.10.2024).
18. Документація WebStorm. URL: https://www.jetbrains.com/webstorm/ (дата звернення: 16.10.2024).
19. What is Git | Atlassian Git Tutorial. Atlassian. URL: https://www.atlassian.com/git/tutorials/what-is-git (дата звернення: 17.10.2024).
20. Документація GitHub. URL: https://github.com/ (дата звернення: 17.10.2024).

21. Документація ZOD. URL: https://zod.dev/ (дата звернення: 18.10.2024).
22. Документація React Hook Form. URL: https://react-hook-form.com/ (дата звернення: 18.10.2024).
23. Документація TanStack React Query. URL: https://tanstack.com/query/v3/docs/framework/react/overview (дата звернення: 18.10.2024).

# ДОДАТКИ

## ДОДАТОК А

## Технічне завдання

**1. Призначення**

Розробка додатку для управління паролями.

**2. Мета додатку**

Продукт спрямований на зберігання, організацію та захист паролів користувачів у безпечному середовищі. Додаток дозволяє зберігати конфіденційні дані, створювати нові паролі з відповідними характеристиками та надає зручний доступ до них.

**3. Цільова аудиторія**

Цільовими користувачами є всі, хто регулярно використовує велику кількість паролів для облікових записів та сервісів.

**4. Можливості додатку:**

- можливість створення, редагування та видалення нового сховища облікових записів;
- можливість створення, редагування та видалення облікового запису;
- можливість пошуку облікових записів по логіну, або ресурсу для якого призначений логін та пароль;
- можливість переглядати усі облікові записи до яких користувач має доступ.

**5. Функціональні вимоги:**

- наявність форми реєстрації/входу;
- зберігання облікових записів;
- генерація паролів;
- шифрування та хешування чутливих даних;
- захист даних користувача за допомогою шифрування на основі майстер-пароля.

**6. Нефункціональні вимоги:**

- простота дизайну;
- зрозуміла навігація;
- безпека;
- масштабування;
- швидкодія.

ДОДАТОК Б

# Інструкція користувачу

## 1. Загальні відомості

Менеджер паролів.

## 2. Функціональне призначення

Дана розробка призначена для створення та зберігання паролів користувачів в одному місці, для забезпечення безпеки облікових записів.

## 3. Умови застосування програми

Для виконання даного додатку необхідно забезпечити себе стаціонарним персональним комп’ютером, або ж ноутбуком, з будь-якою операційною системою, яка пітримує Nodejs 19.6 версії та Docker. Мінімальний обсяг оперативної пам’яті – 4 гб. Програмне забезпечення, яке нам потрібне – це будь-який браузер, редактор коду, Nodejs, Docker та термінал.

## 4. Опис роботи програми

Після того як ми відкрили додаток, ми потрапляємо на сторінку логіну (рис. Б1).

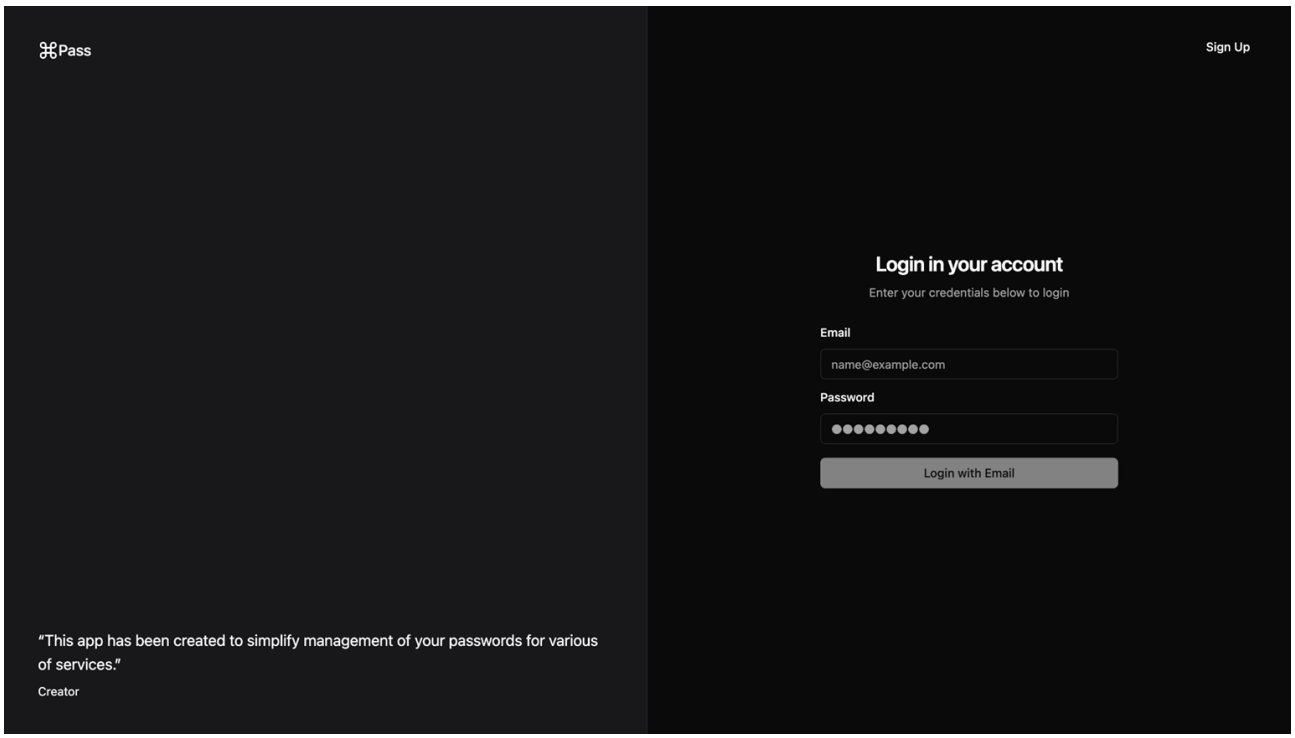


Рисунок Б1 – Сторінка логіну

На сторінці логіну ми можемо перейти на сторінку з реєстрацією (рис. Б2).

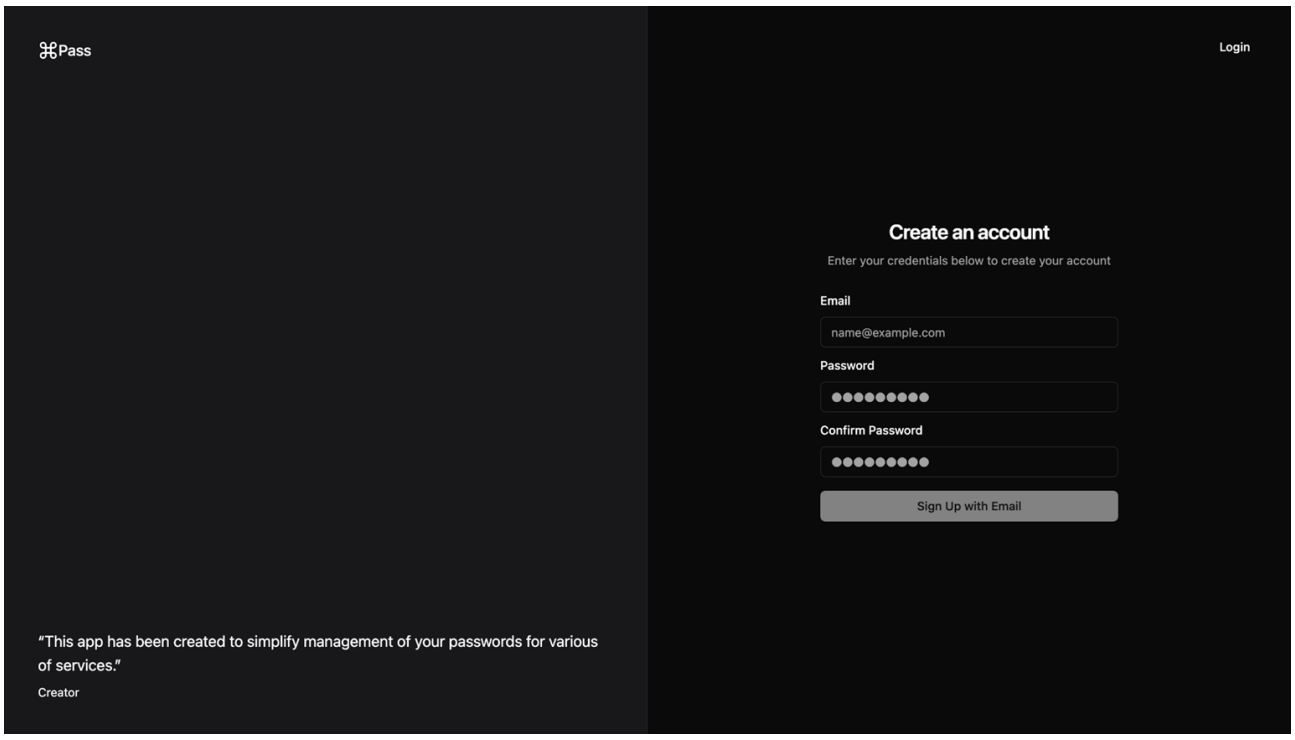


Рисунок Б2 – Сторінка реєстрації

Після того як користувач залогінився, він потрапляє на головну сторінку, де йому потрібно створити сховище облікових записів, якщо такого ще немає (рис Б3).

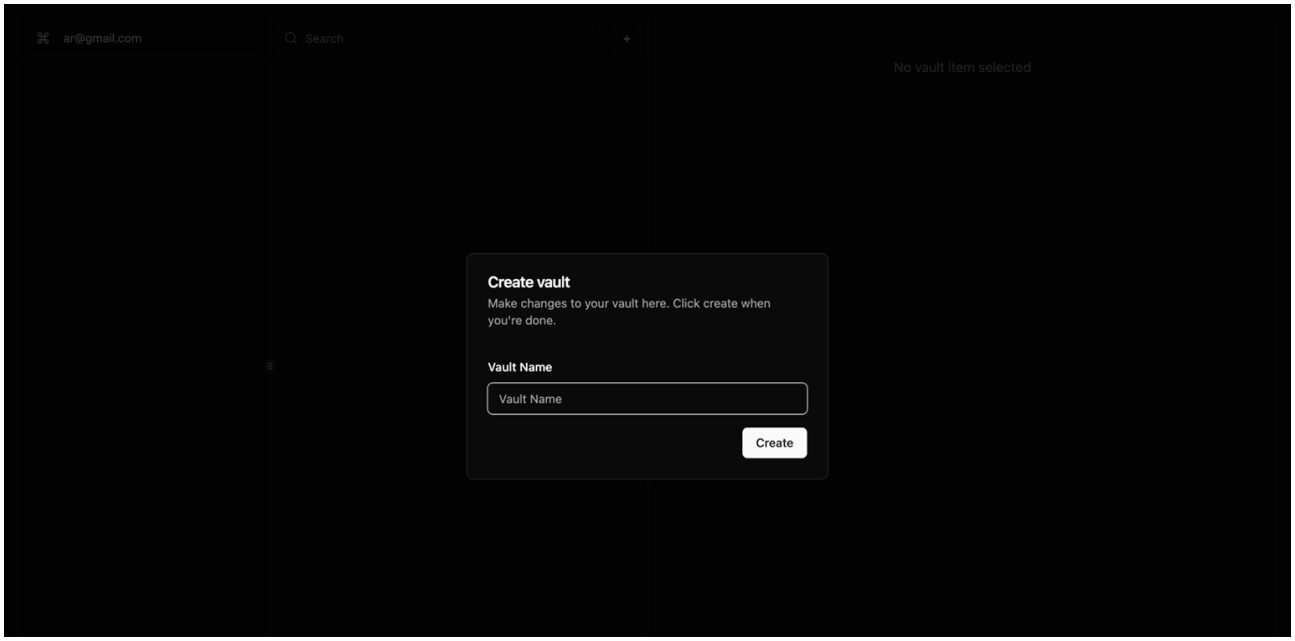


Рисунок Б3 – Головна сторінка

Після цього користувач має можливість додати обліковий запис, натиснувши плюс (рис. Б4). При цьому користувач має змогу налаштувати і згенерувати складний пароль (рис. Б5).

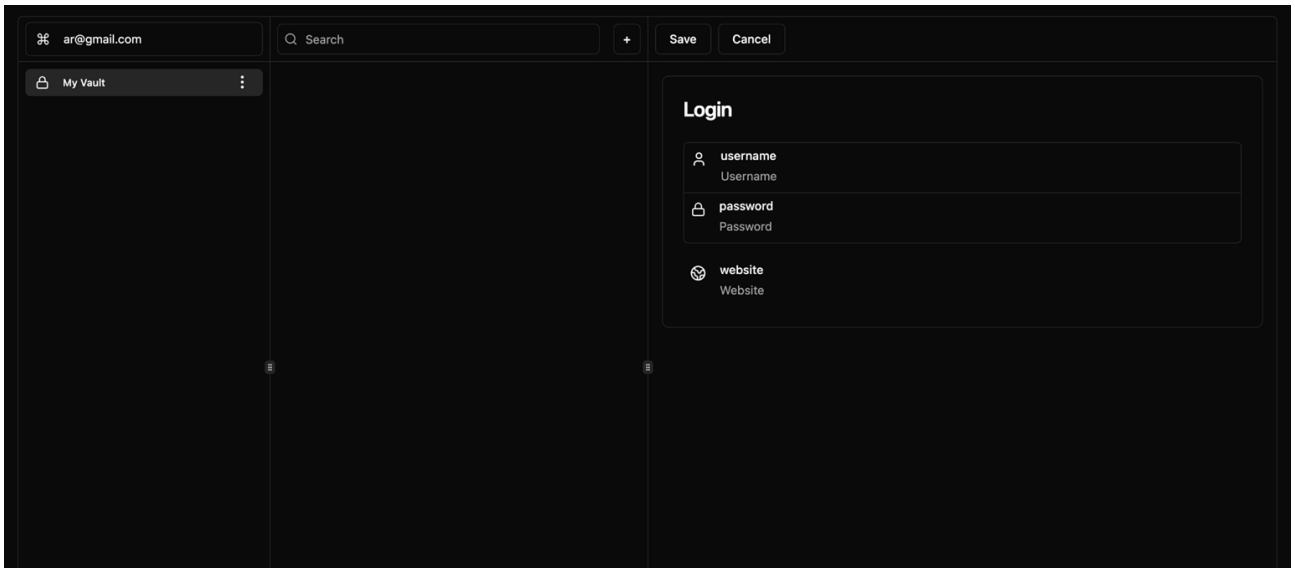


Рисунок Б4 – Додавання облікового запису

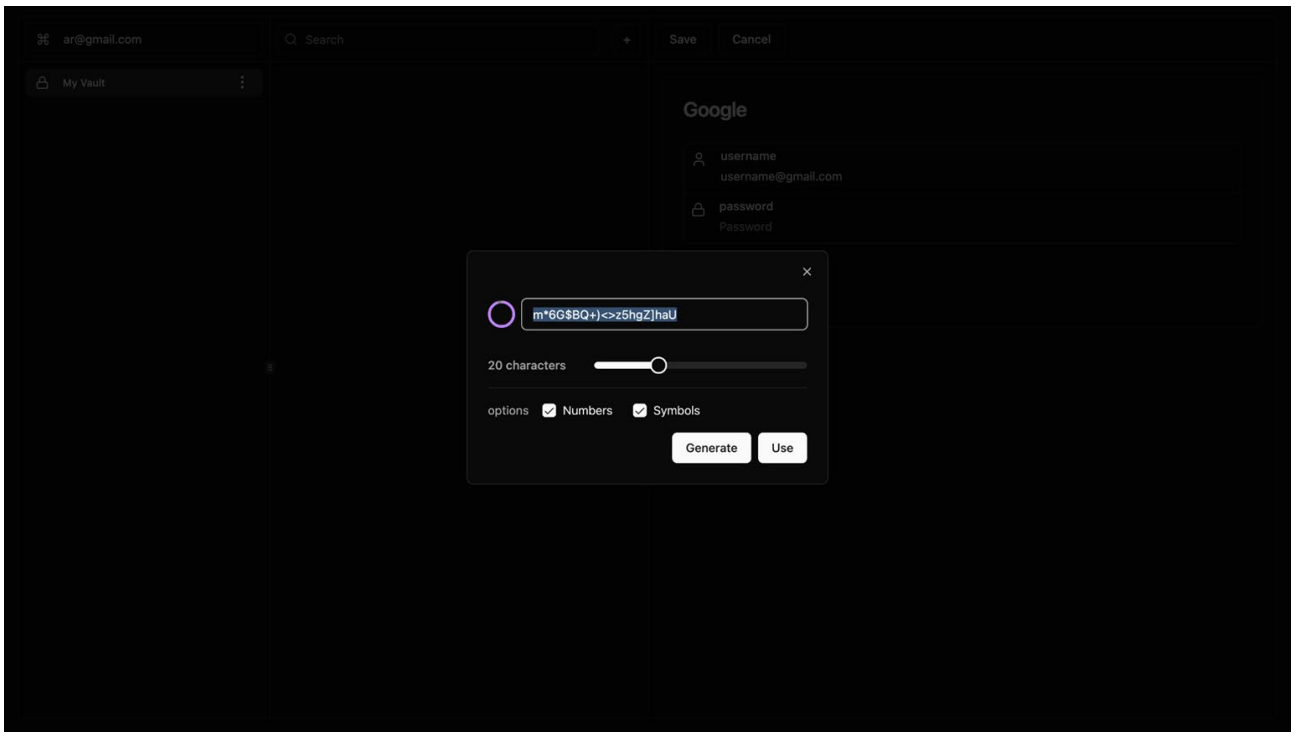


Рисунок Б5 – Налаштування і генерація складного паролю

Після додавання облікового запису, він відобразиться на головній сторінці в поточному сховищі облікових записів(рис. Б6).

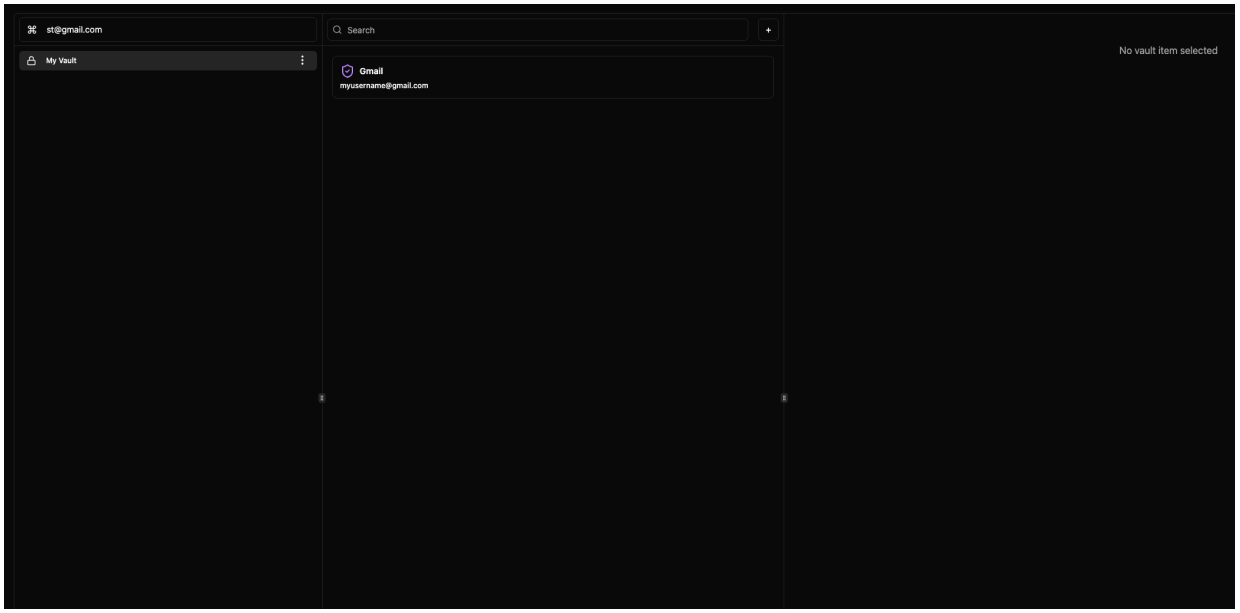


Рисунок Б6 – Головна сторінка із доданими обліковими записами в поточному сховищі

Користувач, також має можливість редагування (рис. Б7) та видалення (рис. Б8) облікового запису.

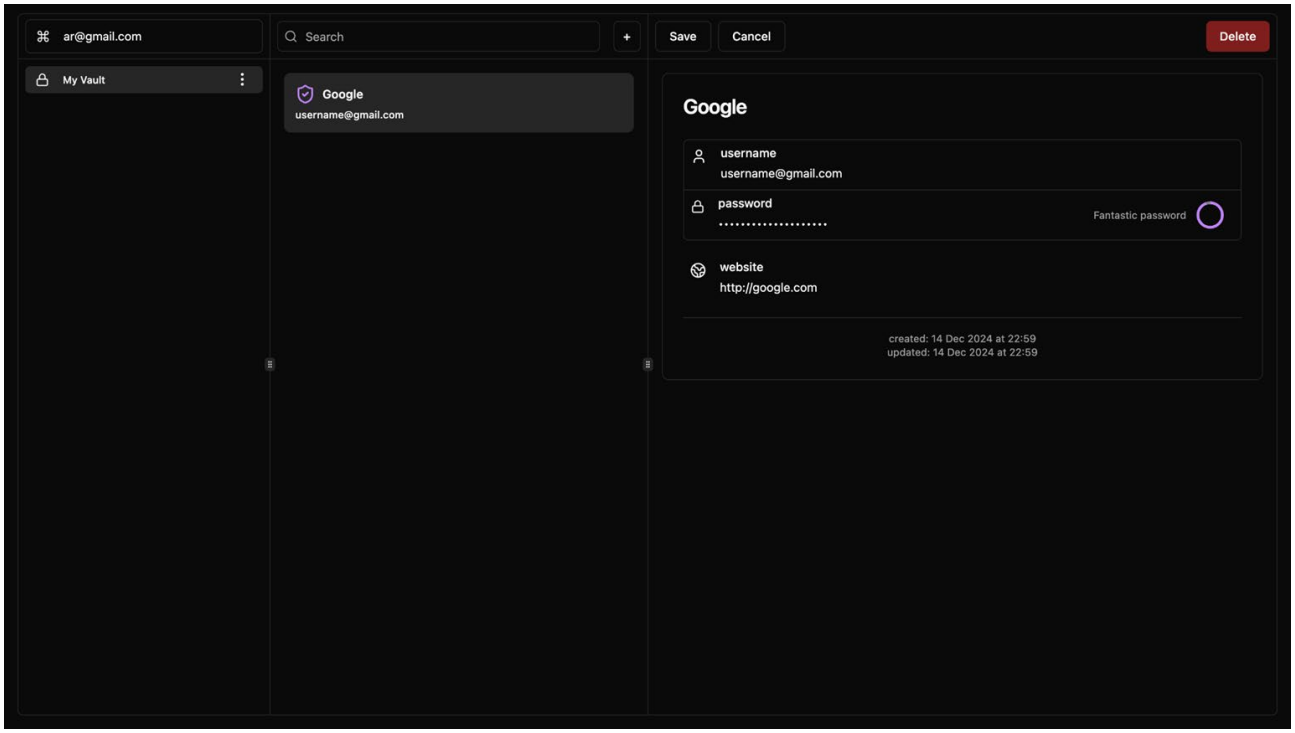


Рисунок Б7 – Редагування облікового запису

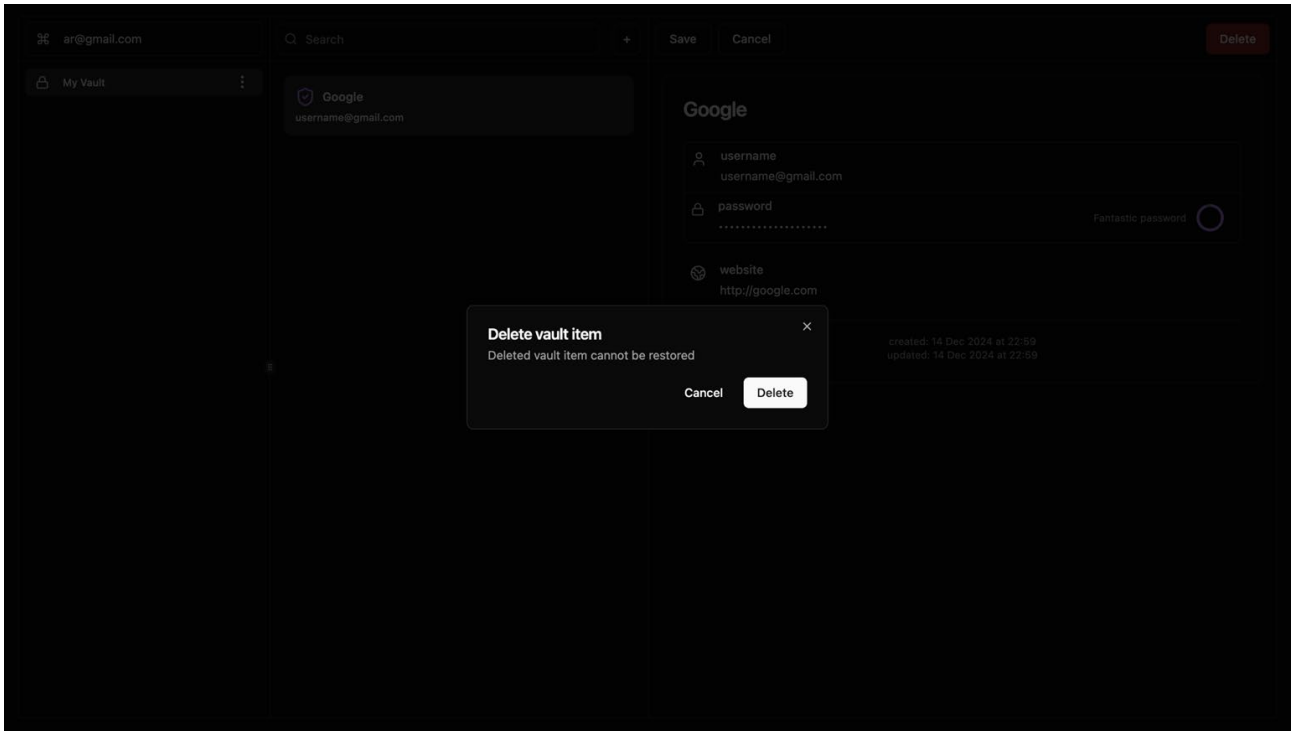


Рисунок Б8 – Модальне вікно для видалення облікового запису

В залежності від того на скільки хороший пароль, користувач буде бачити іконку різного кольору біля кожного облікового запису, також можна побачити таку індикацію в полі з паролем (рис. Б9).

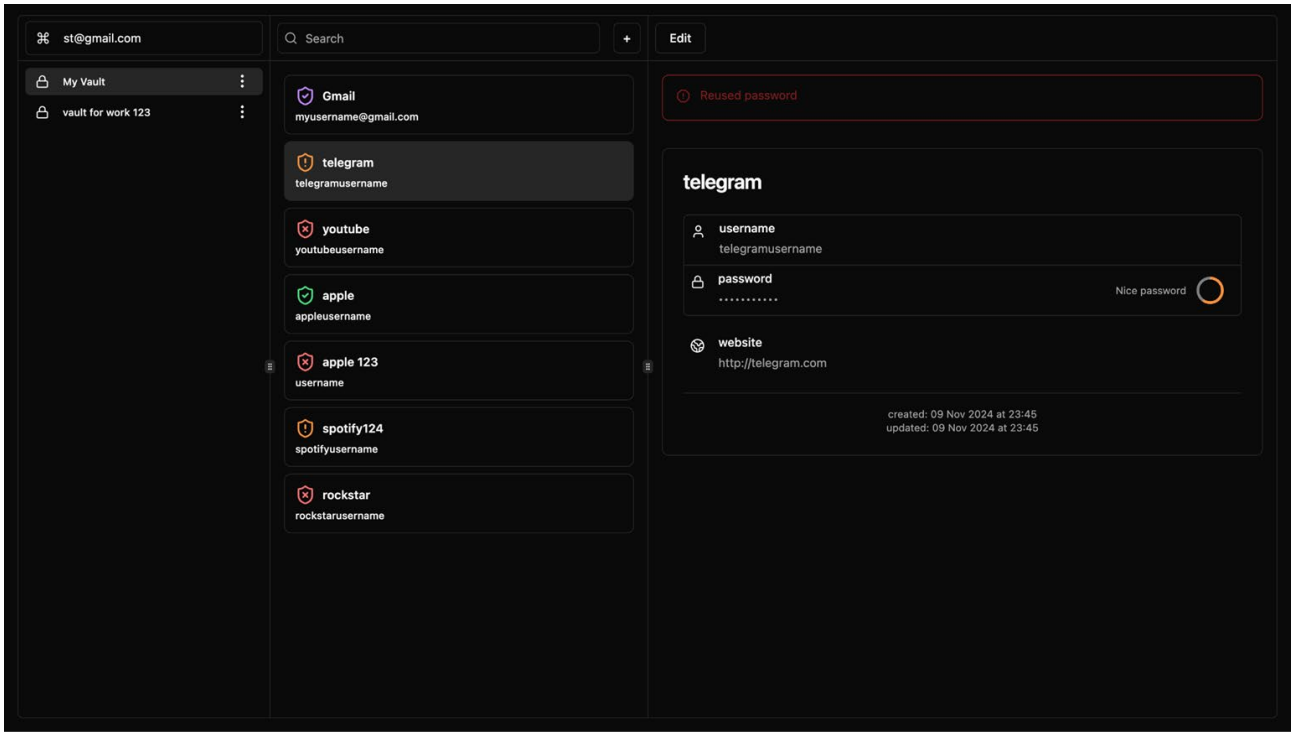


Рисунок Б9 – Індикація на скільки хороший пароль

У випадку якщо користувач використає пароль, який уже є у використанні, з’явиться попередження про повторне використання паролю (рис. 10).

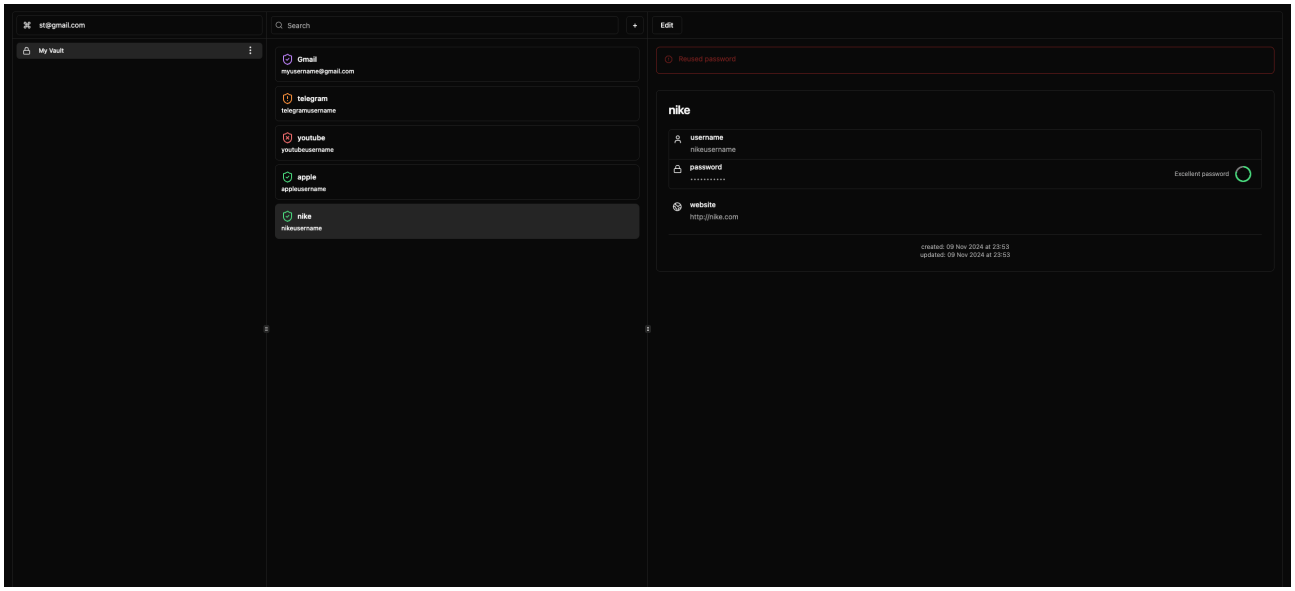


Рисунок Б10 – Попередження про повторне використання паролю

Також кожен користувач має змогу здійснювати пошук облікових записів (рис. Б11).

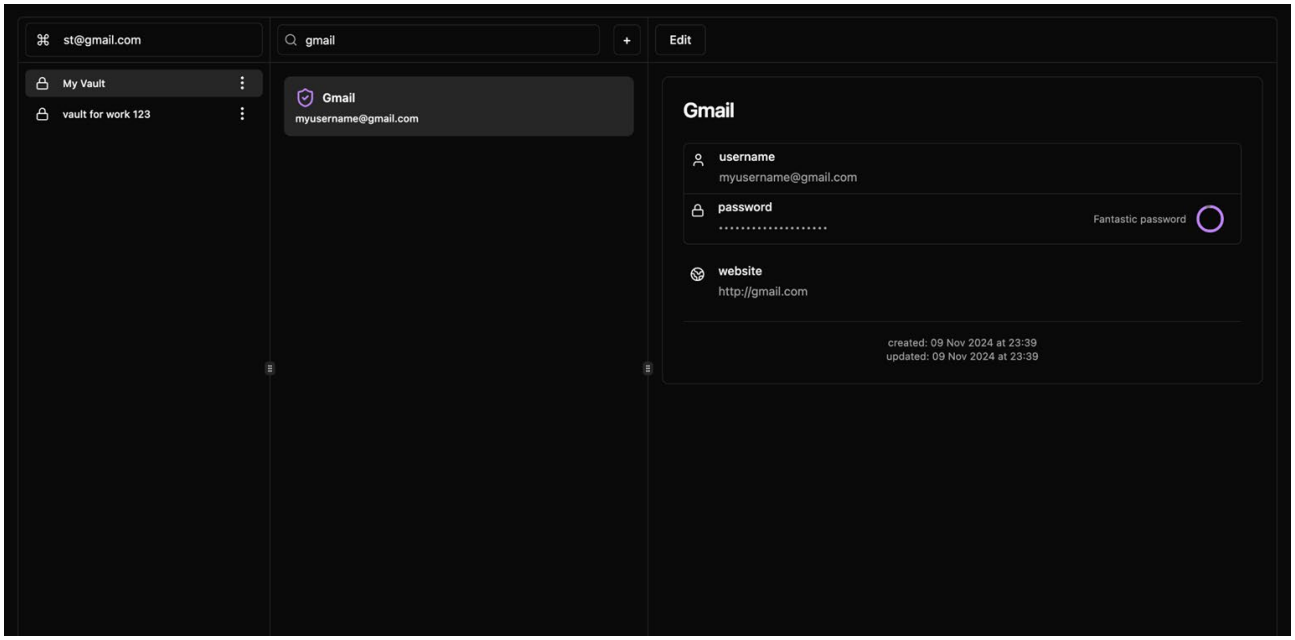


Рисунок Б11 –Пошук облікових записів

До усього вище перерахованого, користувачі мають змогу створювати нові сховища облікових записів (рис. Б12), змінювати їхні назви (рис. Б13) та видаляти їх (рис. Б14).

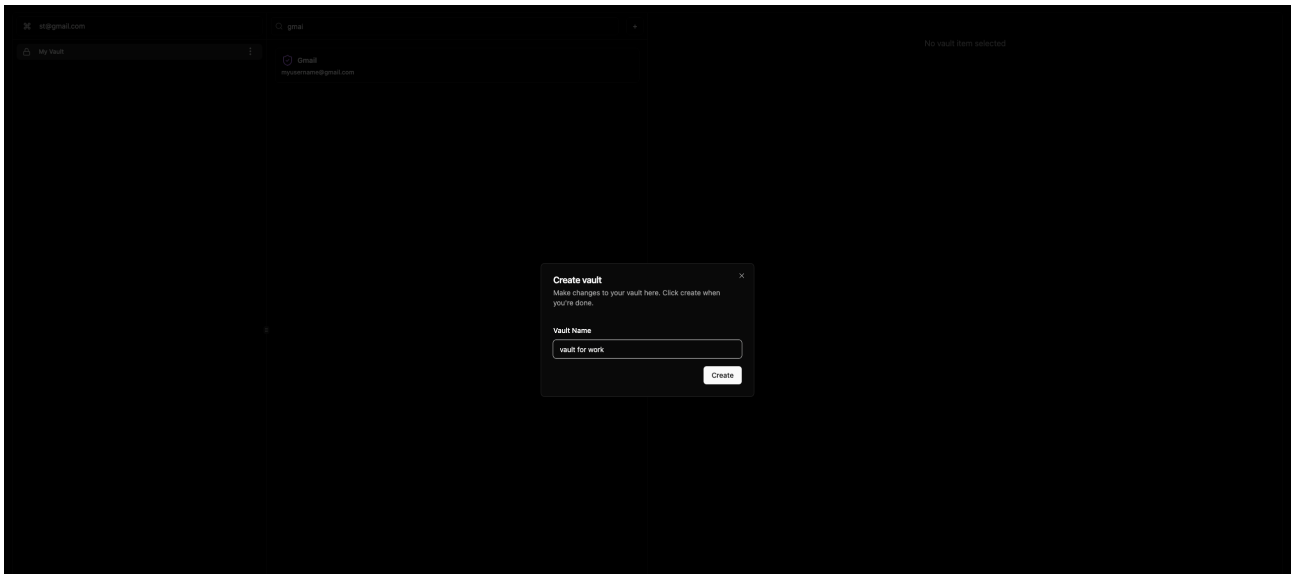


Рисунок Б12 – Створення нового сховища облікових записів

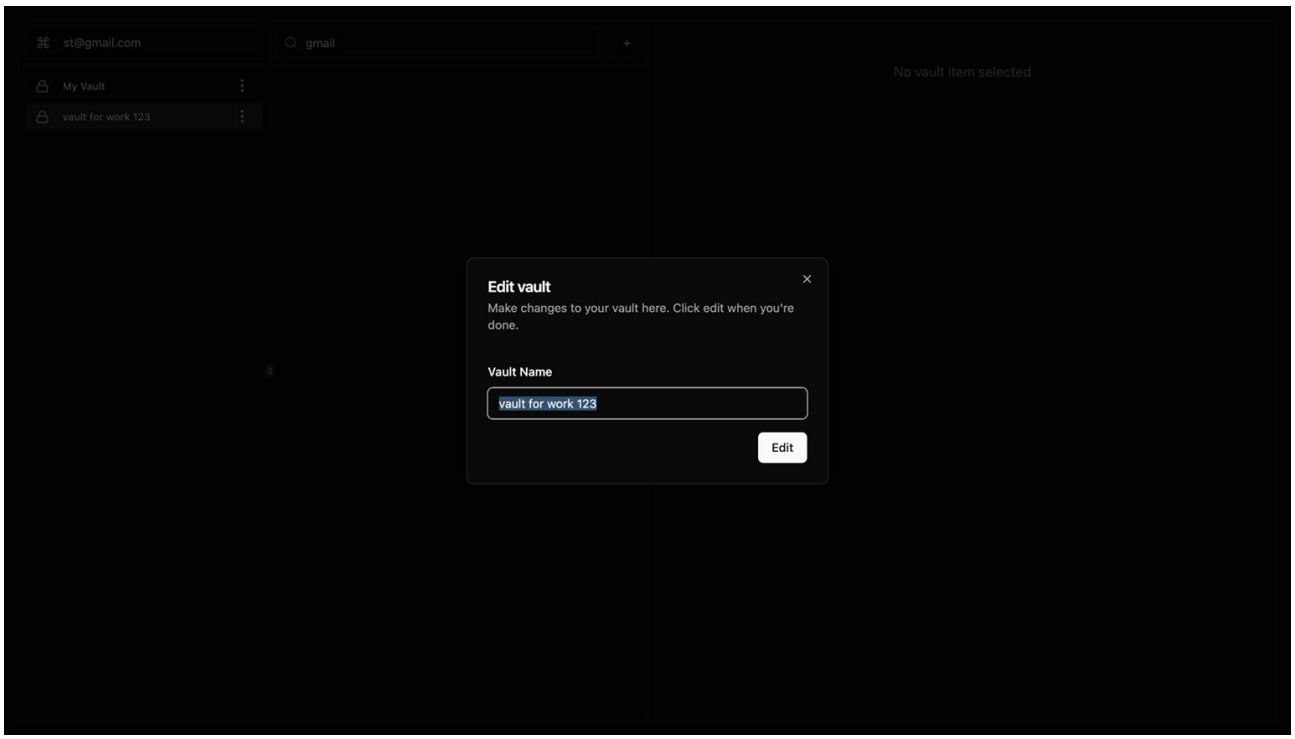


Рисунок Б13 – Зміна назви сховища облікових записів

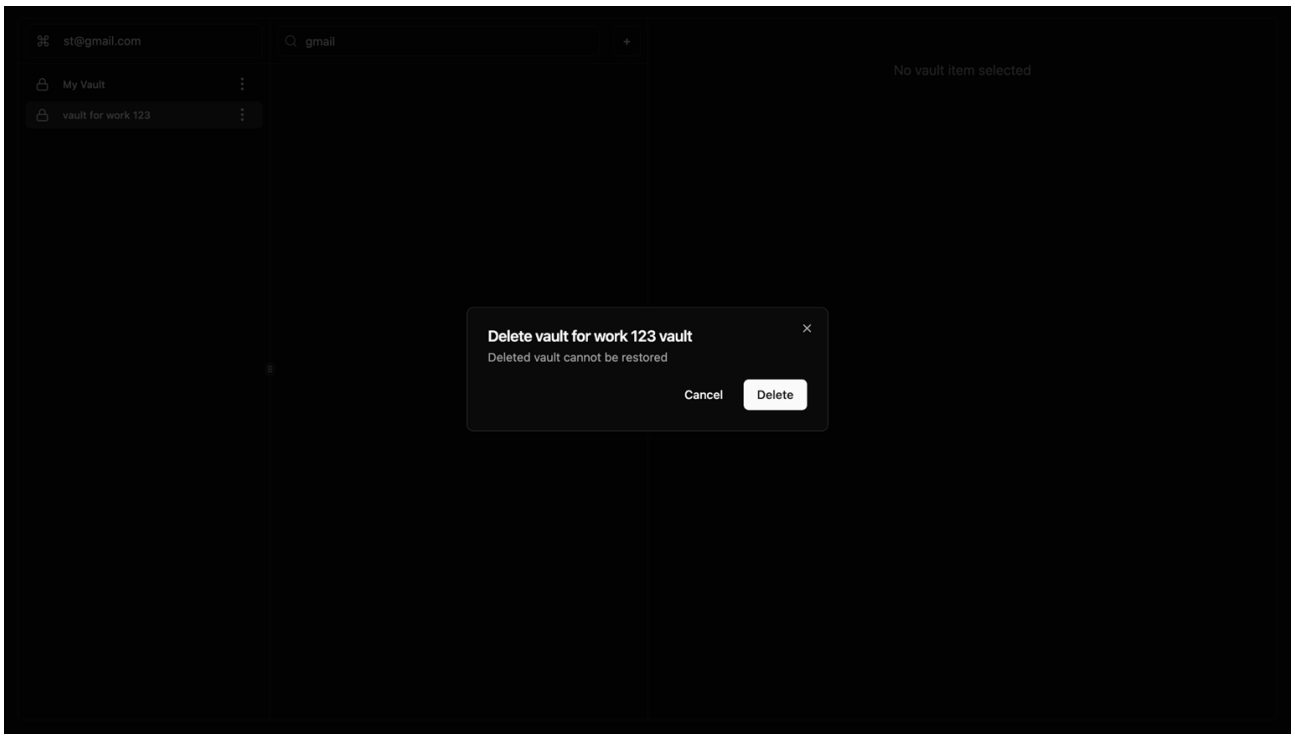


Рисунок Б14 – Видалення сховища облікових записів

# АНОТАЦІЯ

**Пастушок С.Ю. Аналіз методів шифрування на прикладі розробки менеджера паролів.** Рукопис.

Кваліфікаційна робота на здобуття освітнього ступеня «магістр» за спеціальністю 122 Комп'ютерні науки. Волинський національний університет імені Лесі Українки, Луцьк, 2024 р.

Актуальність роботи обумовлена тим, що в умовах сучасного цифрового світу питання безпеки паролів стають надзвичайно важливими. Зростаючий обсяг персональних даних та кібератак вимагають ефективних методів захисту, і надійне зберігання паролів є першочерговим завданням для гарантування конфіденційності користувачів. Водночас, багато користувачів використовують слабкі або повторювані паролі, що збільшує ризик їх компрометації. Тому розробка надійних методів шифрування та хешування паролів стає критично важливою для захисту особистої інформації.

Основною метою роботи є розробка менеджера паролів, який забезпечить зручне і безпечне зберігання та управління паролями користувачів. У роботі буде реалізовано інтерфейс для вебсервісу, створено бекенд та логіку на фронтенді. Задля досягнення цієї мети були визначені наступні завдання: розробка простого та інтуїтивно зрозумілого інтерфейсу, реалізація безпечного зберігання паролів з використанням шифрування та хешування, інтеграція з серверною частиною додатку для взаємодії з базою даних.

Об'єктом дослідження є розробка програмних рішень для менеджера паролів. Предметом дослідження є технології, що використовуються для створення безпечних і зручних менеджерів паролів.

Комплексний програмний продукт складається з трьох основних складових: бази даних, клієнтської частини та серверної частини. В роботі надано детальний опис процесу розробки програмного продукту, з метою показати етапи та методики, які використовувалися для створення розробленого менеджеру паролів. Були розглянуті вимоги до менеджеру паролів та наведено опис його архітектури. Також надано рекомендації щодо подальшого розвитку

та вдосконалення продукту. У роботі наведена покрокова інструкція для запуску додатку, що дозволяє користувачам швидко розпочати використання додатку.

Для розробки були використані такі технології та інструменти: TypeScript для реалізації типізації коду, React та Next.js для реалізації клієнтської частини, NestJs для створення серверної частини, MongoDB для зберігання даних та Git для контролю версій.

Ключові слова: менеджер паролів, безпека паролів, шифрування, хешування, вебсервіс, бібліотека ReactJs, фреймворк Next.js, фреймворк NestJs, цифрова гігієна, захист даних.

# ANNOTATION

**Pastushok Stanislav. Analysis of encryption methods on the example of a password manager.** Manuscript.

Master's Thesis for the Degree of "Master" in Computer Science, Specialty 122. Lesya Ukrainka Volyn National University, Lutsk, 2024.

The study addresses the growing importance of password security in the modern digital landscape. The increasing volume of personal data and cyberattacks demands effective protection methods, with secure password storage being critical for ensuring user confidentiality. Many users rely on weak or reused passwords, heightening the risk of compromise. Thus, developing robust encryption and hashing techniques is essential for safeguarding personal information.

The primary goal of this thesis is to develop a password manager offering convenient and secure storage and management of user passwords. The project includes designing a web service interface, implementing backend functionality, and developing frontend logic. Key objectives include creating an intuitive user interface, implementing secure password storage with encryption and hashing, and integrating the application with a database via the server-side architecture.

The research object is software solutions for password management systems, while the subject is the implementation of password management technologies.

The developed software product consists of three main components: a database, client-side interface, and server-side logic. The thesis provides a detailed description of the development process, outlining the stages and methodologies used. It discusses the requirements and architecture of the password manager and offers recommendations for future enhancements. Additionally, it includes a step-by-step guide for the application, enabling users to quickly get started.

The following technologies and tools were used: TypeScript for type-safe coding, React and Next.js for the frontend, NestJS for the backend, MongoDB for data storage and Git for version control.

Keywords: password manager, password security, encryption, hashing, web service, ReactJS, Next.js, NestJS, digital hygiene, data protection.