МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ОДЕСЬКИЙ ДЕРЖАВНИЙ ЕКОЛОГІЧНИЙ УНІВЕРСИТЕТ

Факультет комп’ютерних наук,
управління та адміністрування
Кафедра інформаційних
технологій

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему: Сервіс сховища файлів та безпечного обміну ними в мережі Інтернет

Виконав студент 2 курсу групи МІС-22
спеціальності 122 Комп’ютерні науки
Міхов Ілля Сергійович

Керівник к.ф.-м.н.,
Ткач Тетяна Борисівна

Рецензент к.т.н., доцент
Перелигін Борис Вікторович

Одеса 2023

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ 5
ВСТУП 7
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ, ІСНУЮЧИХ ПРОГРАМНИХ СИСТЕМ ТА ПОСТАНОВКА ЗАДАЧІ 10
1.1 Характеристика хмарних технологій 10
1.2 Характеристика хмарних сховищ даних 12
1.3 Аналіз існуючих рішень 15
1.4 Постановка задачі 17
2 СИСТЕМА БЕЗПЕКИ СЕРВІСУ 19
2.1 Джерела небезпеки 19
2.2 Вимоги та методи безпеки сервісів 22
2.3 Криптографічні методи захисту даних 29
3 ПРОЄКТУВАННЯ СЕРВІСУ СХОВИЩА ДАНИХ 32
3.1 Використані інструменти 32
3.1 Технічне завдання 33
3.2 Діаграма варіантів використання системи 34
3.3 Діаграма послідовностей 36
4 РЕАЛІЗАЦІЯ СЕРВІСУ СХОВИЩА ФАЙЛІВ 39
4.1 Обґрунтування вибору платформи 39
4.2 Вибір технологій 43
4.3 Спосіб доступу до бази даних 45
4.4 Автентифікація та авторизація 49
4.5 Валідація вхідних даних 51
4.6 Черги завдань 53
4.7 Платіжний сервіс 55
4.8 Вибір та створення архітектури 57
4.9 Тестування роботи сервісу 62
4.10 Реалізація шифрування користувацьких даних 68
ВИСНОВКИ 74
ПЕРЕЛІК ДЖЕРЕЛ І ПОСИЛАННЯ 76

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

БД – база даних.

ОС – операційна система.

ПЗ – програмне забезпечення.

API – Application Programming Interface – програмний інтерфейс програми.

REST – Representational State Transfer – підхід до архітектури мережевих протоколів, які надають доступ до інформаційних ресурсів.

IDE – Integrated Development Environment – інтегроване середовище розробки.

AWS – Amazon Web Services – один з перших сервісів хмарних технологій.

UML – Unified Modeling Language – мова моделювання, яка використовується для візуалізації структури програмних систем.

ORM – Object-Relational Mapping – техніка, що дозволяє зв'язувати об'єктно-орієнтовану модель програми з реляційною базою даних.

SQL – Structured Query Language – мова запитів для роботи з базами даних.

HTTP – Hypertext Transfer Protocol – протокол передачі гіпертексту, використовується для передачі даних на Всесвітній павутині.

TCP – Transmission Control Protocol – протокол керування передачею, який забезпечує надійний обмін даними між комп'ютерами.

RSA – Rivest-Shamir-Adleman – алгоритм шифрування з використанням асиметричних ключів.

DES – Data Encryption Standard – стандарт шифрування даних, що застосовувався раніше, але вважається застарілим з точки зору безпеки.

ECC – Elliptic Curve Cryptography – алгоритм шифрування, що використовує еліптичні криві для створення криптографічних ключів.

XSS – Cross-Site Scripting – вид кібератаки, коли зловмисник впроваджує скрипти на веб-сторінку, які впливають на користувачів.

CSRF – Cross-Site Request Forgery – вид атаки, коли зловмисник використовує авторизацію користувача для виконання небажаних дій.

OWASP – Open Web Application Security Project – організація, що спеціалізується на питаннях безпеки веб-додатків.

TDD – Test-Driven Development – метод розробки, де тестування пишеться перед написанням коду функціональності.

DTO – Data Transfer Objects - Об'єкти передачі даних. Це об'єкти, які використовуються для передачі даних між підсистемами в програмному забезпеченні.

CDN – Content Delivery Network - Мережа доставки контенту. Це група серверів, розташованих у різних частинах світу, призначених для ефективної та швидкої доставки контенту до кінцевого користувача.

SDK – Software Development Kit – Набір розробника програмного забезпечення. Це набір інструментів, бібліотек і документації, які допомагають розробникам створювати програмне забезпечення для певної платформи, мови програмування чи середовища.

ВСТУП

У світі, насиченому інформацією, потреба в ефективному, безпечному та зручному зберіганні та обміні файлами виростає експоненційно. Завдяки стрімкому розвитку технологій та зростанню кількості цифрових даних, виникає неабияка потреба у засобах, які забезпечують не лише зручність, але й надійність у зберіганні та обміні файлами. Сховища в Інтернеті, подібні до Google Drive, стали невід'ємною частиною щоденного життя, проте разом з ростом популярності з'явилися й питання про безпеку та конфіденційність даних. Потенційні загрози безпеці, включаючи можливість несанкціонованого доступу, втрату чи пошкодження файлів, та проблеми з конфіденційністю виникають у зв'язку зі збільшенням обсягів цифрової інформації, що зберігається в хмарних сервісах. Розробка надійного веб-додатка для зберігання та обміну файлами стає критично важливою для забезпечення не лише зручності, але й високого рівня захисту даних. Вирішення проблем безпеки та конфіденційності у подібних сервісах вимагає комплексного підходу та впровадження передових технологій шифрування, аутентифікації та систем контролю доступу.

Спостерігається тенденція до уникання завантаження додатків на пристрої, особливо серед користувачів, які шукають простоту, ефективність та економію місця на пристроях. Багато людей віддають перевагу використанню продуктів через веб-браузер замість завантаження та установки окремих додатків на конкретні платформи. Сучасні веб-технології дозволяють створювати потужні та функціональні веб-додатки, які можуть замінити функціонал та можливості традиційних мобільних додатків. Вони стають доступними через будь-який веб-браузер на різних пристроях без необхідності установки додатків на кожну платформу окремо. Це особливо актуально в умовах, коли користувачі хочуть мати доступ до функціоналу без зайвого обмеження часу та простору на пристроях. Веб-додатки дозволяють негайний доступ до сервісу або продукту без необхідності завантажувати та

встановлювати окремі версії на кожному пристрої, що спрощує користування та забезпечує єдність досвіду для всіх користувачів. Така тенденція до використання веб-платформ для надання послуг та функціоналу може виявитися дуже привабливою для бізнесу, оскільки дозволяє зменшити витрати на розробку та підтримку додатків для різних платформ, а також забезпечує швидкий доступ для користувачів без необхідності завантаження та оновлення програм.

Веб-додаток має ряд переваг порівняно з десктоп-програмами та мобільними додатками для Android та iOS, особливо коли йдеться про універсальність та доступність на різних платформах. По-перше, веб-додаток не обмежений операційною системою чи пристроєм. Це означає, що користувач може отримати доступ до нього через будь-який веб-браузер, будь то на Windows, macOS, Android, чи iOS. Такий підхід спрощує розгортання та оновлення, оскільки не потрібно створювати окремі версії для кожної платформи. По-друге, веб-додатки зазвичай не потребують установки і займають менше місця на пристрої. Це зручно для користувачів з обмеженим обсягом пам'яті на пристрої або для тих, хто уникає завантаження великої кількості додатків. Крім того, веб-додатки можуть мати однаковий інтерфейс та функціонал на різних пристроях, що полегшує користування та навчання нових користувачів. Це дозволяє підтримувати єдиний стиль та функціонал незалежно від пристрою, що сприяє єдності користувацького досвіду. Немає необхідності підтримувати окремі версії для кожної платформи, що дозволяє економити час і ресурси, оскільки зміни вносяться в одному місці і автоматично стають доступними для всіх користувачів незалежно від пристрою. Отже, розробка універсального веб-додатку може бути вигідною для компанії, оскільки спрощує процес розгортання, підтримки та оновлення, забезпечуючи єдність користувацького досвіду на різних платформах.

Метою кваліфікаційної роботи є розробка веб-сервісу для зручного та швидкого отримання доступу до файлів у хмарі. Для досягнення поставленої мети були сформульовані наступні завдання:

- провести аналіз предметної області
- провести порівняльний аналіз існуючих додатків-аналогів;
- обґрунтувати вибір програмних засобів розробки та технологій;
- провести проектування системи з використанням мови UML;
- виконати реалізацію застосунку;
- підготувати інструкцію користувача;
- виконати тестування застосунку.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ, ІСНУЮЧИХ ПРОГРАМНИХ СИСТЕМ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Характеристика хмарних технологій

Історія хмарних технологій сягає початку 2000-х років, коли розвиток Інтернету та зростання обсягів цифрових даних поставили перед людством завдання ефективного зберігання та доступу до інформації з будь-якого місця.

Одним із перших прикладів хмарних технологій був Amazon Web Services (AWS), запущений у 2006 році. AWS надавав інфраструктуру для зберігання даних та роботи з ними через Інтернет, спрощуючи роботу компаній та дозволяючи їм зосередитися на розвитку продуктів, а не на власній інфраструктурі.

Перші хмарні сервіси вирішували проблему обмеженості простору на комп'ютерах та необхідності постійного фізичного зберігання даних. Вони дозволяли користувачам зберігати файли в онлайн-серверах, забезпечуючи доступ до цих даних з будь-якого пристрою, підключеного до Інтернету. Хмарні технології зробили зберігання та обробку даних більш доступними та зручними. Вони надали можливість ефективно працювати з великими обсягами інформації, спростивши спільну роботу над файлами, дозволяючи співпрацювати та вносити зміни в документи одночасно. Хмарні технології загалом характеризуються гнучкістю, високою доступністю, захищеністю даних та можливістю масштабування. Вони стали не тільки інструментом для зберігання даних, а й платформою для розвитку нових програм та сервісів, що полегшує життя користувачів та бізнесу.

Хмарні технології надають користувачам можливість працювати з даними без необхідності володіти великими обсягами власної інфраструктури чи високими обчислювальними потужностями. Основна характеристика хмарних технологій полягає в тому, що користувачі можуть

зберігати, обробляти та отримувати доступ до своїх даних через Інтернет, використовуючи ресурси, що належать постачальникам хмарних послуг.

Замість того, щоб купувати, обслуговувати та оновлювати власне обладнання, користувачі можуть орендувати потужності обчислення, зберігання даних та інші послуги у хмарних центрах, що належать провайдерам хмарних послуг. Це дозволяє їм спростити управління інфраструктурою та скористатися вигодами масштабування, оптимізації витрат та відсутності необхідності великих початкових витрат. Користувачам не потрібно відокремлено закуповувати та встановлювати обладнання чи програмне забезпечення. Замість цього вони отримують доступ до послуг через Інтернет за певну плату, що дає можливість оптимізувати витрати та використовувати ресурси при необхідності, але без обов'язку власної підтримки інфраструктури. Такий підхід використання хмарних технологій дозволяє користувачам ефективно працювати з даними без великих витрат на обладнання та ресурси, забезпечуючи гнучкість, доступність та зручність використання.

Для багатьох нових компаній та бізнесів використання хмарних технологій може мати ряд переваг порівняно з власним розгортанням дата-центрів та інфраструктури, це дозволяє ефективно використовувати ресурси, зменшує початкові витрати та спрощує процеси запуску та масштабування бізнесу.

Зменшення початкових витрат: Замість великих початкових інвестицій у придбання обладнання та налаштування дата-центрів, компанії можуть оплатити лише те, що вони використовують через хмарні послуги, що спрощує фінансове планування та дозволяє економити кошти на початковому етапі.

Гнучкість та масштабованість: Хмарні технології надають можливість миттєво масштабувати обсяги ресурсів в залежності від потреб бізнесу. Це дозволяє швидше реагувати на зростання обсягів роботи та масштабувати інфраструктуру без великих зусиль.

Швидкість запуску та доступу: Використання хмарних технологій дозволяє новим компаніям швидше запустити свої продукти чи послуги на ринку, оскільки вони можуть скористатися готовими рішеннями та послугами без необхідності власної розробки та конфігурації інфраструктури.

Забезпечення безпеки та підтримки: Великі постачальники хмарних послуг мають розгорнуті системи безпеки та підтримки, що може бути важливим для нових компаній, які можуть не мати достатньої експертизи чи ресурсів для власної безпеки даних та підтримки.

Актуалізація та інновації: Хмарні постачальники постійно вдосконалюють свої послуги та пропонують нові інноваційні можливості, що дозволяє компаніям використовувати оновлення та нові функції без зусиль зі свого боку.

1.2 Характеристика хмарних сховищ даних

Хмарні сховища даних – це онлайн-платформи, які дозволяють користувачам зберігати, керувати та доступатися до своїх даних через Інтернет. Вони функціонують на великих серверах, розташованих у різних частинах світу, і забезпечують можливість зберігати інформацію та файли безпечно та доступно. Хмарні сховища виникли як відповідь на ріст обсягів даних, що зберігаються на пристроях. Користувачі шукали способи ефективного зберігання і доступу до цих даних з будь-якого місця та пристрою.

Такі сховища дозволяють зберігати різноманітні дані: від текстових документів, фотографій, відео до аудіо-файлів, архівів та програмних кодів. Користувачі зазвичай зберігають на своїх комп'ютерах особисті дані, які можуть бути важливими та чутливими: від особистих заміток та фотографій до робочих документів і фінансових інформацій, але з більшим інтегруванням Інтернету у повсякденні робочі задачі, навіть такі важливі дані

наразі зберігають у хмарних сховищах. Тому з кожним роком вимоги щодо безпеки та шифрування даних у хмарних сховищах росте. Згідно зі статистикою за 2023 рік - 48% користувачів хмарних сховищ використовують їх для зберігання своїх конфіденційних та приватних даних, таких як: паролі, документи, комерційна статистика, бази даних тощо. І цей відсоток в перспективі буде рости разом з відсотком загальної кількості користувачів хмарних сховищ. 65,28% людей використовують хмарні сховища даних як свій основний спосіб зберігання даних.

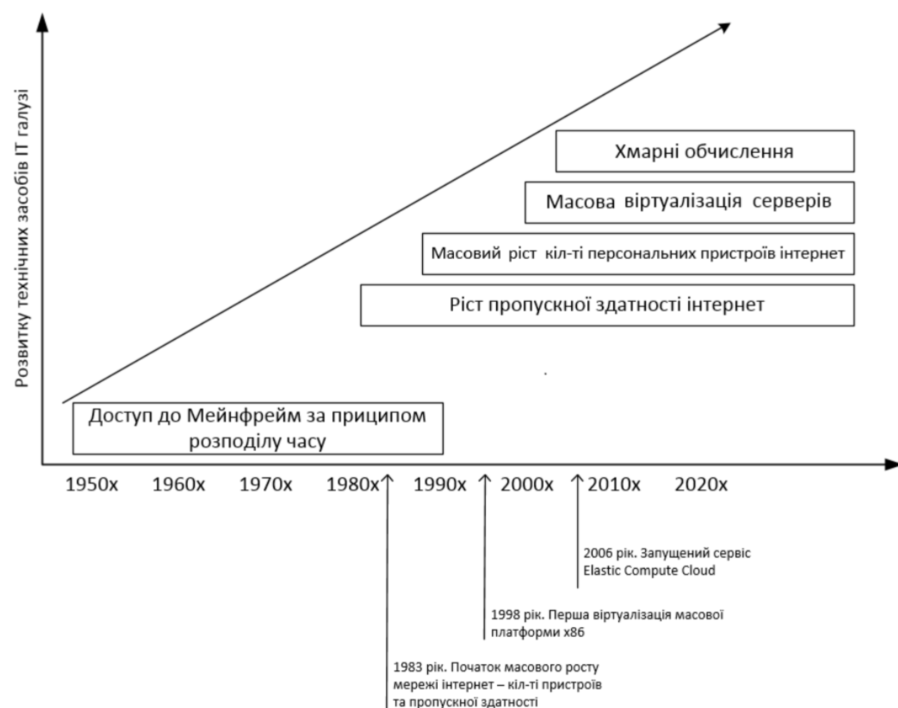

Рисунок 1.1 – Тенденція масового попиту на використання хмарних технологій

Хмарні сховища стали популярними через зручність доступу до цих даних з будь-якого місця та пристрою, роблячи їх доступними для синхронізації між пристроями. Сучасні хмарні сховища вирішують низку проблем з якими стикаються користувачі при використанні фізичних носіїв інформації (HDD/SSD, USB, CD/DVD диски та ін.) По-перше, вони надають

можливість зберігання даних безпечно, оскільки вони мають рівень захисту, який зазвичай включає шифрування та системи захисту доступу. По-друге, вони розв'язують проблему обмеженого місця на пристроях, оскільки користувач може зберігати дані в онлайн-сховищі без необхідності підвищення обсягу пам'яті на своєму пристрої. Також, вони забезпечують можливість спільної роботи над документами та обміну файлами між користувачами, що робить їх популярними для командної роботи та спільного використання ресурсів. Саме ці характерні для кооперативної роботи переваги хмарних сховищ рухають уперед цю індустрію. Більшість бізнесів які надають сховища даних як основну послугу, мають зручні варіанти надання послуг для інших компаній, наприклад корпоративні підписки. Google запровадили цілий набір функціоналу для бізнесів назвавши його Google Workspace.

Google Workspace – це платформа, яка об'єднує різноманітні інструменти та сервіси для роботи, спілкування та спільного вирішення завдань у бізнесі чи навчанні. Раніше відомий як GSuite, Google Workspace пропонує різноманітні засоби спілкування, колаборації, створення та обміну документами, що робить його популярним серед багатьох організацій та користувачів. Основні компоненти Google Workspace включають Gmail для електронної пошти, Google Drive для зберігання та спільного доступу до файлів, Google Calendar для планування подій та зустрічей, Google Meet для відеоконференцій та спілкування у реальному часі, Google Docs, Sheets та Slides для створення та спільного редагування документів, таблиць та презентацій відповідно. Один з головних принципів Google Workspace - це можливість спільної роботи над документами у реальному часі. Користувачі можуть спільно редагувати файли, спілкуватися під час роботи та спостерігати за змінами, що робить цю платформу корисною для командної роботи та колаборації. Google Workspace також надає розширені можливості для адміністраторів, щоб керувати користувачами, захистом даних, налаштуваннями безпеки та іншими аспектами віддаленого робочого

процесу. Завдяки своїм різноманітним інструментам та можливостям, Google Workspace став популярним інструментом для організацій будь-якого розміру, сприяючи спрощенню спілкування, покращенню продуктивності та полегшенню спільної роботи.

1.3 Аналіз існуючих рішень

Ідея хмарних сховищ даних на теперішній час вже не є новинкою згідно статистики. Тому конкуренція грає велику роль у розвитку цих сервісів. Кожний представник подібної платформи намагається вибрати свою основну авдиторію та заглибити його у свої додаткові сервіси тим самим заглибити його у свою екосистему, де всі інструменти доступні та під рукою. Одним з таких великих представників є Google Drive.

Наразі Google Drive забезпечує найбільший розмір сховища для безкоштовного використання. Але треба пам’ятати про те що кожний сервіс від Google використовує один і той самий резерв сховища розмір якого 15 ГБ. Тобто: Google Mail, Google Docs, Google Slides, Google Photos та інші – всі використовують Google Drive для зберігання користувацького контенту. Це допомагає розвивати концепт екосистеми – всі сервіси зв’язані між собою та мають доступ один до одного. Чим більше користувач поглинається у додаткові сервіси від Google тим більше у нього з’являється натхнення перейти на платний рівень використання їх сервісів. Google Drive був запущений у 2012 році, та став лідером у сфері хмарного зберігання та дистрибуції даних, має більше мільярда активних користувачів по всьому світу.

Google Drive використовується як для особистого, так і для бізнес-використання. Компанія яка співпрацює з Google має можливість створити дописи Google для кожного свого співпрацівника, додати його їх до облікового запису Google Workspace, налаштувати доступ до специфічних документів та файлів для кожного унікального працівника. Це дозволяє

маючи один і той самий обліковий запис компанії, маніпулювати доступом для кожного працівника. Наприклад, кожний відділ може мати доступ до своєї групи файлів та документів. Або один відділ може тільки переглядати окремі документи, але не має можливості їх редагувати на відміну від іншого відділу.

Подібні спроможності платформи Google Workspace зробили його одним з найпоширеніших виборів між бізнесами для корпоративного доступу до даних.

На мою думку, одним з найкращих рішень яке зробили Google під час створення Google Drive стала імплементація доступу до сервісу через веб-браузер. Користувачу не було потрібно скачувати та встановлювати клієнт доступу на кожний пристрій, а має доступ до своїх даних з будь-якого місця та пристрою який має доступ до інтернету та веб-браузер.

Dropbox на відміну від Google Drive після випуску мав клієнти для кожної платформи, але не мав можливості отримати доступ до даних через веб-браузер, тому кожний користувач повинен був встановлювати додаток через який міг мати доступ до даних. Але Dropbox створили раніше Google Drive – та веб-версія в решті решт була виконана. Перша версія Dropbox була випущена у 2008 році. Ідея продукту виникла через те, що автор проєкту дуже часто забував на роботі чи вдома USB носій з даними які йому були необхідні. Така банальна проблема призвела до появи сервісів які ми використовуємо зараз дуже часто.

Dropbox на відміну від Google Drive має лише 2GB безкоштовного простору пам'яті для безкоштовного використання. Щодо бізнес-функціоналу, Dropbox надає схожі з Google Drive речі: спільна робота, тобто створення ієрархії папок з різними рівнями доступу над файлами між співробітниками, і навіть зовнішніми бізнес-партнерами; зручний доступ до файлів - можливість отримати доступ до файлів з будь-якого пристрою та будь-якого місця, що полегшує роботу з документами в рухливому режимі; інтеграція з іншими сервісами, наприклад з Google Docs.

Для бізнес-клієнтів Dropbox пропонує декілька варіантів платних послуг та підписок:

- Dropbox Business Standard: Це стандартний план для невеликих та середніх компаній, який надає 5 TB простору для кожного користувача, розширені функції адміністрування та підтримку.
- Dropbox Business Advanced: Цей план пропонує до 5 TB простору на користувача, додаткові засоби безпеки, підтримку клієнтів та додаткові функції адміністрування.
- Dropbox Enterprise: Для великих компаній і корпорацій, цей план має індивідуальний підхід до вирішення потреб компанії, включаючи розширені можливості безпеки, аналітику та підтримку.

Кожен з цих планів має свої особливості, спрямовані на вирішення різних потреб та масштабів бізнесу, що робить Dropbox привабливим вибором для різних типів підприємств.

1.4 Постановка задачі

Перед початком розробки веб-додатка потрібно визначити головні цілі та завдання для вирішуваної проблеми, після цього можна буде розпочинати проектування. Насамперед сформулюємо короткий опис поставленого завдання.

Найменування завдання – це розробка сервісу сховища файлів та безпечного обміну ними в мережі Інтернет. Метою сервісу є надання віддаленого доступу до даних користувача, можливість завантажити дані до хмарного сервісу, додаткові можливості маніпулювання даними з будь-якого пристрою.

Зважаючи на сучасний ринок сервісів зберігання даних у хмарі, для того щоб конкурувати з ними подібний веб-додаток потребує наступних рис:

- Сильне шифрування: забезпечення захисту даних шляхом потужного шифрування на рівні транспорту та зберігання даних.

- Контроль Доступу: Можливість точно налаштовувати, хто має доступ до яких даних та в який час.
- Швидкість та Надійність: Забезпечення швидкого доступу до даних та високої доступності сервісу.
- Простота й Інтуїтивність: Легкість використання та інтуїтивний інтерфейс для користувачів будь-якого рівня.
- Інтеграція з іншими Сервісами: Забезпечення сумісності та легкої інтеграції з іншими популярними сервісами.
- Масштабованість: Здатність пристосовуватися до зростання обсягів даних та потреб користувачів.
- Гнучкість: Наявність різних планів та можливостей, щоб користувачі могли вибирати, що підходить їм найкраще.

За результатами проведеного аналізу були розроблені вимоги до веб-додатку та розроблено додаток «Cloud Drive». Основними функціями додатку є зберігання та передача даних більшості форматів. Гарний сервіс та приваблива ціна використання сервісу буде сприяти поповненню клієнтської аудиторії і, як наслідок, збільшенню прибутку, що є головною метою бізнесу. Дана розробка може бути перспективною та корисною з комерційної та споживчої точки зору. Найбільше уваги під час розробки та планування проекту потрібно приділити швидкості та безпеці сервісу тобто:

- Використання кешування у місцях де це технічно можливо, використання валідації кешу, тестування актуальності кешу, зменшити кількість запитів у базу даних за однотипними даними.
- Використання сучаних та ефективних алгоритмів шифрування користувацьких даних на серверах для гарантії конфіденційності для користувачів.
- Використання потужних серверів для розміщення сервісу хмарного зберігання даних, мати велику та розширувану кількість фізичного місця для даних клієнтів.

2 СИСТЕМА БЕЗПЕКИ СЕРВІСУ

2.1 Джерела небезпеки

Технології показують стрімкий розвиток у різних сферах життя. Яркий показник цього – вплив технологій які використовують штучний інтелект на повторювані та навіть креативні задачі за останні декілька років. Майже кожного тижня великі видання лякають людей тим що пройде деякий час, і штучний інтелект замінить значну кількість робочих місць собою. Але у недавньому інтерв'ю, колишній директор Microsoft Білл Гейтс зазначив, що він бачить майбутнє таким, коли люди працюють 4-6 години на день, бо основні задачі будуть вирішувати машини з дуже розвиненим використанням штучного інтелекту. Тим самим люди зможуть більше зосереджуватися на саморозвитку і креативних задачах, що покращить життя мільйонів людей по всьому світу.

Штучний інтелект дозволяє системам вчитися, адаптуватися та удосконалюватися з часом, набуваючи нових функцій та здібностей. Цей розвиток відкриває безліч можливостей у багатьох галузях, від розваг та освіти до бізнесу та медицини. Проте, із таким стрімким ростом технологій також постає загроза безпеці. Розвиток штучного інтелекту може стати об'єктом для кіберзлочинців, які шукають способи використовувати ці технології для атак, злому систем, або для поширення шкідливих програм. Це робить безпеку сервісів та систем унікально важливою в умовах стрімкого росту технологій. Захист основних даних, використання надійних методів шифрування, а також застосування сучасних методів виявлення та захисту від кіберзагроз стають невід'ємною частиною розвитку технологій. Збільшення швидкості та масштабу розвитку технологій змушує компанії та розробників звертати особливу увагу на забезпечення безпеки та конфіденційності даних. Це дозволяє залишатися впевненими в тому, що технології розвиваються в безпечний спосіб, зберігаючи конфіденційність та захищаючи користувачів від потенційних загроз.

Забезпечення безпеки в програмному забезпеченні важливо для забезпечення захисту користувачів, попередження втрати даних та запобігання негативним наслідкам для бізнесу. Автори ПЗ повинні вживати заходів щодо аудиту безпеки, регулярного вдосконалення безпекових протоколів та постійного моніторингу заходів безпеки для захисту від подібних загроз. Джерела небезпеки для продуктів ПЗ можуть бути різноманітні. Розглянемо їх.

- Кіберзлочинці: хакери та зловмисники можуть намагатися використати вразливості у програмному забезпеченні для отримання доступу до системи, крадіжки конфіденційної інформації, розповсюдження шкідливих програм або вимагання викупу (Ransomeware).
- Внутрішні загрози: іноді загроза може прийти зсередини самої компанії, коли працівники або колишні співробітники мають доступ до конфіденційної інформації та можуть використовувати цю інформацію неправомірно.
- Віруси та шкідливі програми: віруси, троянські програми та інші шкідливі програми можуть бути внесені в програмне забезпечення та поширюватися через неохайний код або завдяки вразливостям у системі.
- Вразливості безпеки: наявність вразливостей у програмному забезпеченні може спричинити можливість зламу системи чи неправомірного доступу до даних.
- Неадекватне керування доступом: недостатній контроль над правами доступу до системи може призвести до неправомірного використання даних.
- Недостатнє тестування та якість програмного забезпечення: програмне забезпечення, яке не пройшло відповідне тестування, може мати внутрішні помилки, які стануть джерелом проблем під час експлуатації.

– Фізичні загрози: наприклад, крадіжка комп'ютерів або серверів, може викликати втрату конфіденційної інформації.

Найбільш критичними ситуаціями для власників продукту системи хмарного сховища даних є втрата приватних користувацьких даних загалом, чи доступність цих даних для третіх осіб. Якщо дані користувачів стають недоступними через випадкове видалення, технічні збої або кібератаки, це може призвести до серйозних втрат для користувачів і для компанії взагалом. Небезпека порушення конфіденційності та витоку даних завжди присутня. Необхідно забезпечити надійне шифрування даних користувачів під час передачі та зберігання на серверах. Нестача або неправильна настройка систем контролю доступу може призвести до несанкціонованого доступу до даних. У випадку втрати даних важливо мати ефективну систему резервного копіювання та відновлення, щоб можна було швидко відновити інформацію. Для подібної системи має проводитися частий аудит, який перевіряє якість роботи механізму відновлення даних при критичній ситуації.

У 2018 році компанія GitLab зіткнулася з проблемою відновлення користувацьких даних при критичній ситуації. GitLab – це сервіс, який надає хостинг репозиторіїв для IT проєктів. Працівник ненавмисно видалив одну з баз даних яка мала у собі переважну частину користувацьких даних за останні 24 години. Через недостатню увагу команди до системи резервного копіювання та відновлення даних, сервіс з багатотисячною базою користувачів перестав працювати майже на 9 годин, що для сервісу такого масштабу призвело до великих репутаційних та коштовних збитків.

Система відновлення даних деякий час не була протестована, тобто, після великої кількості змін вихідного коду основного продукту не було проведене регресійне тестування цієї системи, яка була розроблена для того щоб швидко завантажити дані користувачів із резерву та продовжити роботу. Інженерам GitLab прийшлося працювати вночі та хутко вирішувати проблему мануально, імпортуючи дані користувачів з резервної бази даних до основної. Нажаль, деякі дані так і не вдалося поновити через те що

остання копія даних основної бази даних була не актуальною до стану основної під час виникнення проблеми.

Загублення або пошкодження даних користувачів у хмарному сховищі є критичним сценарієм. Надійна система резервного копіювання є життєво важливою для забезпечення можливості відновлення даних в разі потреби. Також, шифрування даних користувачів на серверах та базах даних є обов'язковим для захисту конфіденційності даних від несанкціонованого доступу чи можливих загроз. Ці заходи можуть допомогти у попередженні серйозних проблем та мінімізації втрат для користувачів у випадку непередбачуваних подій чи атак.

2.2 Вимоги та методи безпеки сервісів

Згідно з дослідженням корпоративних ризиків від Allianz у 2016 році, що базувалося на опитуванні понад 820 експертів з ризик-менеджменту та страхування з 44 країн, кіберзлочини вперше увійшли до трійки найбільш значущих загроз для підприємств. Цей вид ризику був визначений як один з найбільш серйозних для компаній у наступних 10 роках. За даними Cybersecurity Ventures, до 2021 року збитки від кіберзлочинності подвояться в порівнянні з 2015 роком, досягнувши $6 трлн. Основна ціль кіберзлочинців традиційно спрямована на різні фінансові установи, особливо банки, а також платформи електронної торгівлі. За даними звіту компанії Trend Micro Incorporated за перше півріччя 2016 року, однією з найбільш важливих загроз у фінансовому секторі залишаються банківські трояни.

Коли мова іде про безпеку у веб-просторі неможливо не сказати про організацію OWASP (Open Web Application Security Project), яка займається питаннями безпеки веб-додатків. Вони створюють рекомендації, стандарти та інструменти для захисту веб-додатків від різноманітних кіберзагроз.

OWASP була заснована у 2001 році та стала однією з ключових організацій, що надає універсальні стандарти безпеки для веб-додатків.

Організація складається з добровольців, які спільно працюють над розробкою рекомендацій та рішень для захисту веб-додатків від різних загроз. Товариство OWASP включає в себе корпорації і научні заклади багатьох країн. OWASP працює над створенням статей, навчальних посібників, рекомендацій, документацій, інструментів і технологій, які зберігаються у відкритому доступі.

OWASP публікує різноманітні матеріали, такі як:

- OWASP Top 10: це список найбільш поширених уразливостей веб-додатків, який публікується періодично.
- Документація та рекомендації: статті та інструкції, які надають розробникам та інженерам безпеки важливу інформацію про те, як захищати веб-додатки від різних загроз.
- Інструменти та проекти: OWASP також розробляє власні інструменти для тестування та аналізу безпеки веб-додатків.

Організація використовує відкритий підхід до розробки та надає широкому загалу користувачів можливість внести свій внесок у вдосконалення стандартів безпеки для веб-додатків. Їхні рекомендації стали стандартом в індустрії програмного забезпечення та веб-розробки, допомагаючи підвищити рівень безпеки для веб-додатків у всьому світі. Завдяки подібним організаціям та їх праці, розробники по всьому світу мають можливість ознайомитися з поточними загрозами для додатків.

Розробка безпечного у використанні веб-сервісу включає в себе кілька ключових етапів та вимог.

- Захист від кіберзагроз: розробники повинні усвідомлювати потенційні кіберзагрози та вміти захищати свої системи від відомих типів атак. Це включає захист від SQL-ін'єкцій, XSS (міжсайтовий скриптінг), CSRF (підробка міжсайтових запитів), а також застосування принципів найменших привілеїв.
- Використання сучасних алгоритмів шифрування – це критичний аспект розробки безпечних веб-сервісів, оскільки шифрування дозволяє

захистити дані під час їхньої передачі та зберігання від несанкціонованого доступу. Це особливо важливо для конфіденційної інформації, такої як паролі, особисті дані користувачів, банківська інформація тощо.

- Аутентифікація та авторизація: правильна ідентифікація та авторизація користувачів – це ключовий аспект безпеки. Важливо використовувати сильні паролі, двофакторну аутентифікацію, правильно налаштовувати права доступу користувачів.
- Регулярні оновлення: постійне оновлення та виправлення вразливостей у програмному забезпеченні, бібліотеках та фреймворках, що використовуються.
- Безпечність даних: захист конфіденційної інформації, шифрування даних під час передачі та зберігання на серверах, а також коректна обробка чутливих даних.
- Тестування безпеки: проведення регулярних тестів на безпеку, таких як тестування на проникнення, аудит коду, тестування на вразливості, що дозволяє виявляти та виправляти проблеми безпеки перед випуском продукту.
- Документація та навчання персоналу: важливо навчати персонал з питань безпеки, а також створювати та підтримувати документацію з правилами безпеки та процедурами реагування на інциденти.
- Забезпечення резервного копіювання: регулярне створення резервних копій даних для відновлення інформації в разі потреби.

Ці практики допомагають розробникам забезпечувати високий рівень безпеки у веб-сервісах та додатках, зменшуючи ризики втрати даних та кібератак.

Деякі алгоритми шифрування, які раніше були вважалися надійними, тепер вважаються застарілими або вразливими через появу нових методів атак та зростання обчислювальної потужності, тому вони не відповідають сучасним стандартам і вимогам безпеки веб-сервісів. Наприклад, алгоритми

DES (Data Encryption Standard) та MD5, які колись були популярними, тепер вважаються недостатньо надійними для захисту конфіденційних даних.

DES був розроблений у 1970-х роках і є одним з перших широко використовуваних алгоритмів шифрування даних. Його популярність була пов'язана з тим, що він був першим стандартизованим алгоритмом шифрування, рекомендованим для застосування у федеральних установах США. Однак з часом було виявлено, що DES має деякі вади, особливо пов'язані з його коротким ключем. DES використовує ключ довжиною лише 56 біт, що за сучасними стандартами шифрування вважається недостатньою довжиною для надійного захисту даних. З розвитком обчислювальної техніки та методів криптоаналізу, стало відомо, що DES може бути вразливим до підбору ключа шляхом перебору. У 1999 році Electronic Frontier Foundation (EFF) за допомогою спеціального обладнання вивело доведення, що можна розгадати ключ DES за прийнятний час, який різко підірвав його стійкість. Цей експеримент підкреслив важливість використання більш довгих ключів для захисту даних від сучасних обчислювальних потужностей та методів атак. В результаті цього DES вважається застарілим алгоритмом шифрування і не рекомендується для використання у сучасних системах.

MD5 теж не вважається безпечним для захисту конфіденційної інформації. Цей алгоритм був розроблений більше 30 років тому і зазнав численних криптографічних атак, що показали його вразливість.

Однією з основних проблем MD5 є можливість колізій, коли два різних екземпляра даних можуть виробити однаковий хеш. Це означає, що атаки на MD5 можуть бути успішними через злам хеш-функції, особливо в обчислювально-інтенсивних сценаріях, таких як атаки на паролі, де зловмисники можуть використовувати велику обчислювальну потужність для створення колізій.

Тому рекомендація з криптографічної безпеки полягає в уникканні використання MD5 для захисту конфіденційної інформації. Замість цього

рекомендується використовувати більш безпечні алгоритми хешування, такі як SHA-256 або SHA-3, які мають більшу стійкість до криптографічних атак.

Сучасні безпечні алгоритми шифрування включають:

AES (Advanced Encryption Standard): Це один з найбільш популярних та надійних алгоритмів. Використовується з ключами різної довжини (наприклад, AES-256), що робить його відмінним варіантом для захисту даних.

RSA (Rivest-Shamir-Adleman): Часто використовується для шифрування та підпису даних, особливо в системах з асиметричним шифруванням.

ECC (Elliptic Curve Cryptography): Це ще один асиметричний алгоритм, який володіє великою стійкістю та використовує менше ресурсів порівняно з RSA.

Важливо відзначити, що безпечність шифрування може залежати не лише від самого алгоритму, але й від правильного використання його реалізації в програмному коді. Навіть найсильніший алгоритм може стати слабким при неправильному використанні або застосуванні його зі старими версіями протоколів. Тому важливо постійно оновлювати та вдосконалювати методи шифрування відповідно до сучасних стандартів та рекомендацій.

Під час розробки веб-додатку важливо мати на увазі досвід минулих зламів та технік для отримання несанкціонованого доступу до ресурсу зловмисниками. Розробники веб-фреймворків, бібліотек для взаємодії з базами даних та інших інструментів для інших розробників намагаються імплементувати захист від подібних широковідомих проблем, для того щоб починаючий розробник не зіткнувся з подібними проблемами зараз. Але не завжди у таких розробників в руках є всі інструменти котрими можна захистити кінцевого користувача. Тому одною з вимог для безпечного веб-додатка – знання розробником ключових уразливостей з якими він може стикнутися навіть цього не підозрюючи. Такими уразливостями можуть бути:

SQL-ін'єкція - це вид атаки, коли зловмисник використовує вразливість в системі бази даних для виконання шкідливого SQL-коду через веб-

застосунок. Зазвичай це стає можливим через погану обробку введених даних, коли користувацький ввід не фільтрується чи не екранується належним чином перед виконанням запиту до бази даних. Сучасні ORM (Object Relational Mapping) бібліотеки чи SQL-білдери враховують подібні атаки, та не створюють такі вихідні SQL-запити які могли би містити SQL-ін'єкцію.

Основна ідея полягає в тому, що зловмисник може вставити SQL-код у поля введення, який пізніше виконається в базі даних. Наприклад, якщо в рядку пошуку URL-адреси відбувається вставка SQL-коду, то це може призвести до виконання SQL-запиту до бази даних.

Щоб уникнути SQL-ін'єкцій, розробники повинні виконувати наступні кроки:

- Використання параметризованих запитів: Використання параметризованих запитів або підготовлених виразів дозволяє виділити користувацький ввід як дані, а не як частину SQL-запиту. Це допомагає уникнути можливості виконання SQL-коду, вставленого через введення користувача.
- Екранування введених даних: Перед використанням даних, що вводяться користувачем, вони повинні бути відфільтровані та екрановані для забезпечення того, щоб вони не виконувалися як частина SQL-запиту.
- Обмеження прав доступу: Мінімізація прав доступу користувача до бази даних, щоб вони мали доступ тільки до необхідних таблиць та функцій.
- Регулярні аудити та оновлення: Проведення регулярних аудитів коду для виявлення потенційних вразливостей та вчасне оновлення заходів безпеки.

Враховуючи ці практики та вдосконалення методів фільтрації та обробки користувацького введення, розробники можуть запобігти SQL-ін'єкціям та зберегти безпеку веб-додатків, що працюють з базами даних.

Ще один розповсюджений тип атаки який потрібно мати на увазі - CSRF (Cross-Site Request Forgery). Це атака, при якій зловмисник отримує доступ до веб-додатку через авторизованого користувача та виконує дії в цьому додатку від його імені, використовуючи підмінені запити. Основна ідея полягає в тому, що зловмисник може виконати певні запити (наприклад, зміну паролю, відправку грошей тощо) через веб-додаток від імені авторизованого користувача, якщо у цього користувача відкрита сесія в додатку. Щоб уникнути атак CSRF під час розробки веб-застосунку, слід вжити наступні заходи:

- Використання токенів CSRF (CSRF tokens): генерація унікальних токенів для кожного запиту від користувача та їх включення до форм або запитів. При отриманні запиту сервер перевіряє наявність токену, що допомагає визначити легітимність запиту.
- Підтвердження дій: перевірка на стороні сервера, що дії, які виконуються, відповідають інтенціям користувача. Наприклад, перед видаленням об'єкта чи виконанням важливої операції, запит може вимагати підтвердження від користувача.
- Використання SameSite cookies: встановлення атрибута SameSite для cookies, який може обмежувати відправку cookies зовнішніми запитами, що також допомагає уникнути CSRF.
- Додаткова перевірка авторизації: перевірка додаткових факторів аутентифікації, таких як пароль або фінгерпринт, перед виконанням сенситивних дій.

XSS (Cross-Site Scripting) – це тип атаки, коли зловмисник вставляє веб-скрипти (зазвичай JavaScript) у веб-сторінки або введення користувача, які потім виконуються на боці користувача в його браузері. Ця атака дозволяє зловмисникові отримати доступ до куків, сесій або іншої конфіденційної інформації, а також виконувати різні дії від імені користувача.

Щоб уникнути атак XSS під час розробки веб-застосунку, слід вжити наступні заходи безпеки:

- Екранування даних: екранування (escaping) усіх введених даних перед виведенням їх на сторінку. Це означає, що будь-який користувацький ввід повинен бути очищений від потенційно небезпечних тегів або символів.
- Використання безпечних API: використання безпечних методів внесення даних у HTML, таких як методи вставки тексту (text insertion) замість внесення HTML-коду напряму.
- Коректна обробка вхідних даних: перевірка та фільтрація введених даних з боку користувача. Всі дані, що вводяться користувачем, повинні бути перевірені на наявність потенційно небезпечних символів та відфільтровані.
- HTTP заголовки безпеки: встановлення заголовків безпеки, таких як Content Security Policy (CSP) та HTTP Strict Transport Security (HSTS), які допоможуть у запобіганні XSS-атак.
- Регулярні аудити: проведення регулярних аудитів веб-додатку для виявлення потенційних XSS-вразливостей та виправлення їх негайно після виявлення.

Ці заходи допоможуть підвищити безпеку веб-додатку та запобігти можливості атак, зменшуючи ризик впливу зловмисника на користувачів веб-додатку.

2.3 Криптографічні методи захисту даних

Криптографія – це наука про захист інформації за допомогою математичних методів та алгоритмів. Вона має давню історію, починаючи з часів стародавнього світу, коли люди використовували різні шифри та коди для захисту важливих повідомлень. Сучасна криптографія розвинулася до

складних математичних алгоритмів, які забезпечують високий рівень безпеки.

У сучасному світі криптографія використовується практично в усіх сферах життя, де потрібно захистити конфіденційні дані. Вона використовується в банківській справі для шифрування фінансових транзакцій, в онлайн-торгівлі для захисту особистих даних користувачів, в мережах зв'язку для забезпечення конфіденційності даних, і багато в чому іншому.

При розробці функціональностей для веб-застосунків криптографія використовується для шифрування даних, забезпечення безпеки сесій, створення безпечних з'єднань та перевірки ідентичності користувачів. Криптографічні методи захисту даних включають у себе алгоритми шифрування, хеш-функції, цифрові підписи, аутентифікацію та авторизацію. Існує багато криптографічних методів захисту даних, які використовуються для забезпечення конфіденційності, цілісності та доступності інформації.

- Шифрування даних: використовується для захисту конфіденційності даних. Цей метод перетворює звичайний текст у незрозумілий шифрований вигляд за допомогою ключа. AES, RSA, та ECC - це деякі з популярних алгоритмів шифрування.
- Хешування: використовується для перетворення даних у фіксований рядок фіксованої довжини (хеш), який слід представляти оригінальні дані. MD5 та SHA-2 (включаючи SHA-256, SHA-384, і SHA-512) - це деякі з алгоритмів хешування.
- Електронний цифровий підпис (Electronic Digital Signature): використовується для перевірки цілісності та автентифікації даних. Асиметрична криптографія використовується для створення та перевірки підпису.
- Аутентифікація: включає методи, такі як аутентифікація двофакторна або багатофакторна, біометрична аутентифікація та інші способи перевірки ідентичності користувача.

- Ключі та сертифікати: використовуються для керування доступом, підтвердження ідентичності та безпечного обміну даними.

Ці методи криптографії часто використовуються в поєднанні для забезпечення максимального рівня захисту даних. Кожен з них має свої унікальні властивості та застосується в залежності від потреб конкретної системи чи ситуації.

Криптографічні технології повинні постійно розвиватися через швидкий розвиток технологій, які вимагають нові методи захисту від сучасних загроз. Що сьогодні вважається безпечним, може стати вразливим завтра через нові методи атак. Тому криптографічні алгоритми постійно еволюціонують, щоб залишатися надійними перед сучасними атаками. Із зростанням обчислювальної потужності нові методи стають доступними для взлому. Криптографія повинна вдосконалюватися, щоб утримувати крок з розвитком обчислювальних технологій та залишатися надійною перед потенційними загрозами. Різні технології використовують криптографію (наприклад, IoT, хмарні технології, медицина). Стандартизація криптографічних методів допомагає забезпечити їх сумісність із різними системами. З появою нових технологій з'являються і нові загрози. Наприклад, розвиток квантових обчислень може вплинути на поточні криптографічні методи, оскільки квантові комп'ютери можуть ламати певні сучасні шифри.

Розвиток криптографічних технологій є ключовим для забезпечення приватності та безпеки у сучасному цифровому світі. Нові відкриття і методи дозволяють зберігати дані в безпеці від сучасних і майбутніх загроз.

3 ПРОЄКТУВАННЯ СЕРВІСУ СХОВИЩА ДАНИХ

3.1 Використані інструменти

Проектування інформаційної системи є наступним важливим етапом після аналізу предметної області та аналогів програмних продуктів. Перед тим як розпочати розробку, потрібно проаналізувати який продукт ми хочемо бачити у своїй MVP (minimum viable product) версії. Для цього потрібно визначити найголовніші бізнес-функціональності без яких продукт не буде виконувати свою основну функцію. Після їх нотування можна переходити до менш пріоритетних задач. Таким чином – способом встановлення пріоритетів можна отримати базовий backlog (загальний список задач на виконання), чи список «майлстоунів» (великих задач які включають у себе менші за об’ємом задачі). Під час проектування проєкту Cloud Drive, я використовував текстовий редактор Notion для створення технічної документації, плануванням та слідкуванням за процесом розробки. За допомогою Notion можна створити Kanban-дошку подібного формату для слідкуванням за розвитком проєкту.

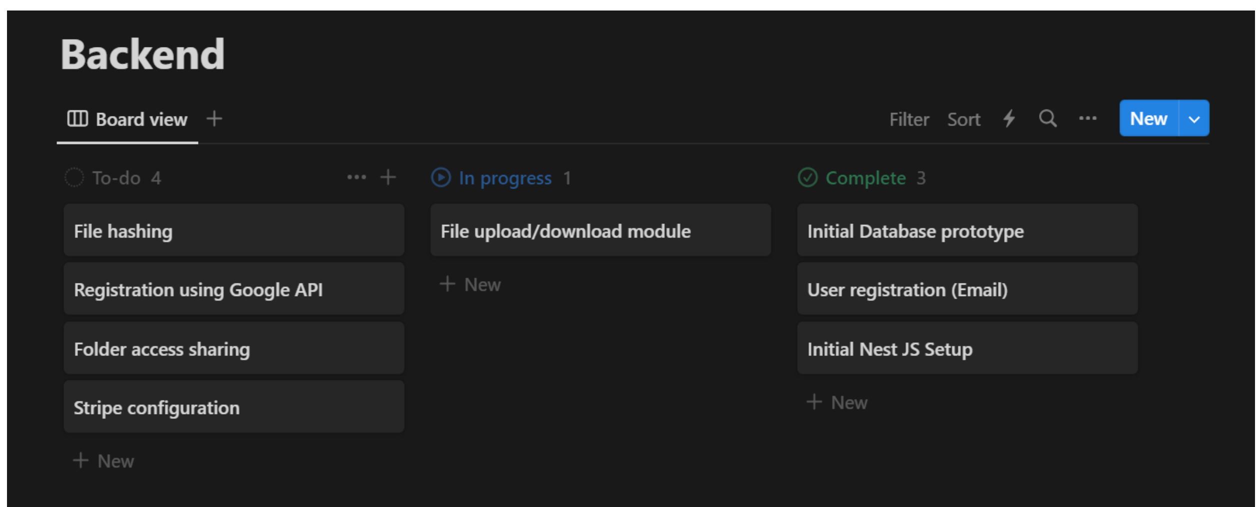

Рисунок 3.1 – Приклад Kanban-дошки

UML-діаграми варіантів використання, активності й послідовності, а також розробка макетів інтерфейсу додатка створюються на даному етапі.

3.1 Технічне завдання

Основна бізнес-функціональність хмарного сховища даних полягає в забезпеченні зручного та постійного доступу користувача до своїх даних, які зберігаються на віддаленому сервері. Головною метою є забезпечення безпеки цих даних через їх шифрування та захист від несанкціонованого доступу. Користувач може отримати доступ до своїх файлів з будь-якого місця та пристрою, забезпечуючи конфіденційність та надійність зберігання інформації. Клієнт має мати можливість управління доступом до файлів для безпеки даних. Також плюсом може бути застосування декількох способів авторизації для забезпечення сучасного та зручного способу дістатися до своїх файлів. Файли повинні синхронізуватися між усіма пристроями, та мати свої резервні копії на випадок критичних ситуацій. Часто, користувачі видаляють свої файли, і потім в них знову з'являється в них потреба – функціонал «Корзина» також вважається стандартом для подібного роду сервісів.

В той самий час користувачу не потрібно думати про технічну частину реалізації подібного функціоналу, такого як використання функцій шифрування на різних рівнях, з якою періодичністю створюються резервні копії та як вони архівуються, як працює кешування відображених даних, за яким алгоритмом працює завантаження файлів користувачів на сервера і так далі.

Ці функції спрямовані на забезпечення зручного, безпечного та ефективного управління даними у хмарі, що відповідає сучасним потребам користувачів та бізнесу. Зберігання балансу між ефективністю, складністю всередині та простоти використання клієнтом ззовні є найціннішим та найважливішим у розробці сучасного програмного забезпечення.

3.2 Діаграма варіантів використання системи

Інвестиція в планування розробки будь-якої системи чи додатку завжди доводить те, що є ситуації та функціональності про які спочатку не згадуєш. Завжди зручно працювати з готовим продуманим з усіх сторін планом системи аніж продумувати реалізацію «по-шляху». Недостатня увага на планування може призвести до:

- Недооцінення обсягу робіт: якщо не прорахувати всі можливі сценарії та функціональні вимоги, це може призвести до недооцінення ресурсів (часу, коштів, людських ресурсів), необхідних для втілення плану.
- Несумісності та проблеми інтеграції: відсутність докладного аналізу може призвести до несподіваних проблем під час інтеграції з іншими системами або компонентами, що затримає роботу проекту.
- Невідповідності вимогам користувачів: якщо вимоги користувачів не були достатньо проаналізовані або враховані, це може призвести до створення продукту, який не відповідає реальним потребам або очікуванням клієнтів.
- Затримки в реалізації та збільшення вартості: несподівані труднощі під час розробки через відсутність плану можуть призвести до затримок у впровадженні та збільшення витрат.
- Нестабільна архітектура системи: недостатнє планування може призвести до непридатної архітектури, яка може стати причиною недоліків у майбутньому та потреби в доробках.

Одним з прикладів документування роботи системи є розробка UML-діаграми варіантів системи (або діаграми прецедентів). Діаграма варіантів використання є інструментом, що надає візуальне представлення функціональності проєктуваної програмної системи та взаємодії різних користувачів з нею. Основна мета цієї діаграми – описати, як різні ролі користувачів будуть використовувати систему для вирішення своїх завдань. Актори, які зображені у вигляді стилізованих символів, виступають у ролях

користувачів, пристроїв або зовнішніх систем, що взаємодіють з основною програмою. Вони представляють собою ключові сутності, які мають доступ до системи та виконують визначені завдання. Ці актори ініціюють та управляють варіантами використання, що є конкретними сценаріями взаємодії з системою. Діаграма варіантів використання визначає, як саме користувачі або зовнішні сутності взаємодіють з системою, які дії вони здійснюють та які функції системи вони використовують для досягнення своїх цілей. Це дає можливість детально описати усі можливі сценарії використання системи та допомагає розробникам краще розуміти потреби користувачів.

Для розробки діаграми проєкту Cloud Drive був використаний веб-сервіс Draw Io. За результатами сформованих варіантів використання та акторів системи було розроблено діаграму варіантів використання, яку наведено на рис. 3.2.

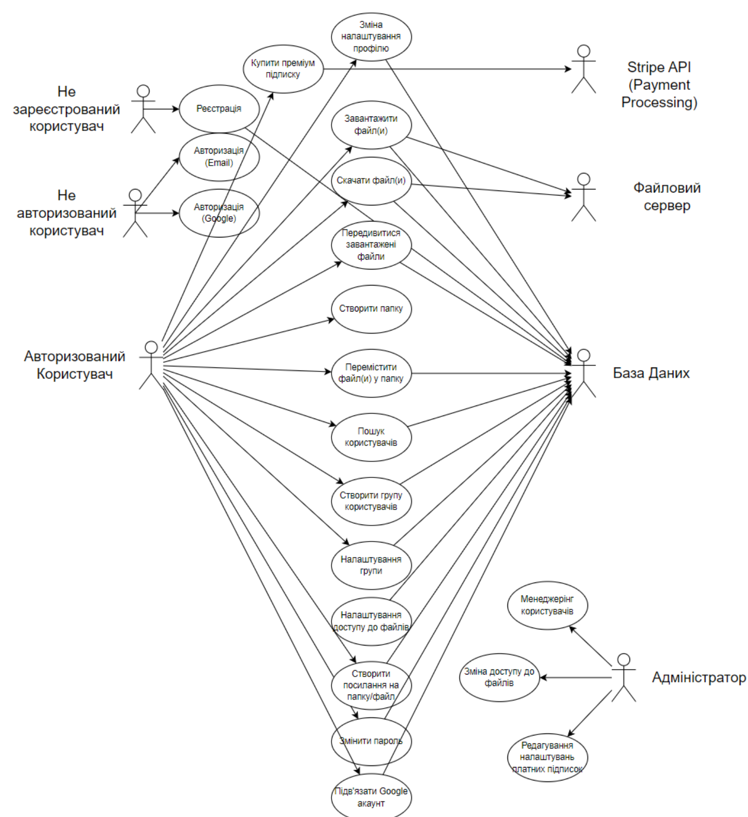

Рисунок 3.2 – Часткова діаграма варіантів використання

3.3 Діаграма послідовностей

Ще один спосіб детально зобразити те як повинна працювати система, чи окрема її частина – використання діаграми послідовностей. Завдяки їй можна зобразити взаємодію окремих модулів системи під час виконання процесу.

Діаграма послідовностей – це візуальне зображення послідовності взаємодії між об'єктами у системі у конкретному сценарії. Це може бути використано для уточнення деталей та послідовності дій, що відбуваються під час виконання конкретної операції чи функції. Такі діаграми використовуються для уточнення діаграм прецедентів і більш детального опису логіки сценаріїв використання.

Один з сценаріїв послідовностей у проєкті є процес покупки преміум підписки, який зображений на діаграмі на рис. 3.2: користувач отримує список доступних йому підписок для покупки, вибирає потрібну підписку та оплачує її ціну за допомогою платіжного сервісу Stripe, який ізолює в собі всю логіку розрахункових операцій та зберігання приватних даних платіжних способів користувачів. На початку транзакції сервер створює запис про початок операції оплати, а далі Stripe повертає до серверу відповідь зі статусом платежу, який потім обробляє и повертає в сформованому форматі користувачу, оновлюючи стан розрахункової транзакції в базі даних. Завдяки тому що база даних Cloud Drive має ідентифікатор користувача у Stripe, ми можемо розробити систему таким чином, щоб кожного разу коли новий користувач реєструється у Cloud Drive у Stripe створювався відповідний Stripe Customer (клієнт). Таким чином дуже зручно ідентифікувати клієнта який проводить транзакцію. Саме тому у моделі користувача в базі даних є два ідентифікатора: userId - ідентифікатор користувача у системі Cloud Drive, customerId – ідентифікатор цього же користувача у базі даних Stripe.

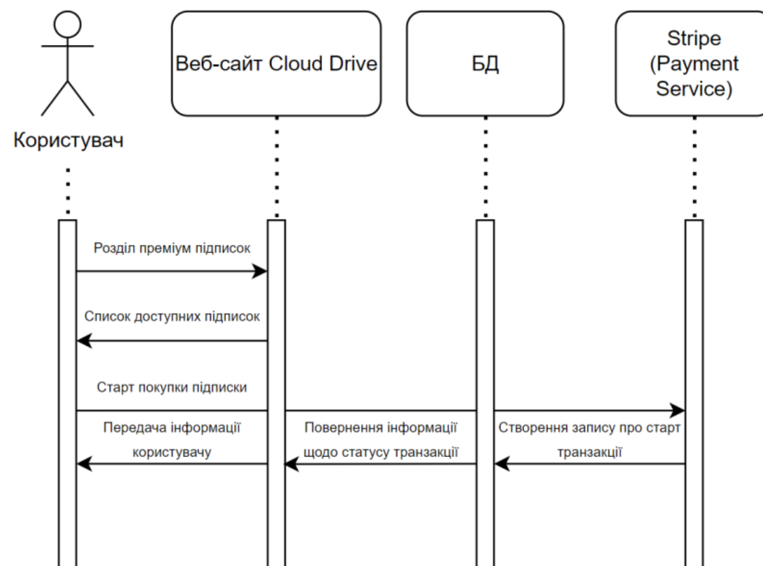

Рисунок 3.3 – Процес покупки зображений на діаграмі послідовностей

Під час обробки транзакцій на стороні Stripe виконуються процеси валідації та логіка обробки банківських платежів від яких цей сервіс нас абстрагує. Транзакція у Stripe це об'єкт який має всю наявну інформацію та метадані які були згенеровані на бек-енді сервісом або у Stripe під час обробки платежу. Ці дані можуть бути: будь-які дані які сервіс передав у метадані, наприклад вказання який тип підписки на вашій платформі покупається, на яку суму – це може допомогти у подальшій обробці платежу на стороні серверу за допомогою веб-хуків Stripe, дані про клієнта який створив платіж, статус платежу, спосіб оплати (картка, банківський рахунок, тощо)

Вебхук – це механізм веб-розробки, що дозволяє веб-додаткам надсилати автоматичні повідомлення або дії іншим програмам. Це спосіб сповіщення системи про події, які відбуваються в інших програмах або сервісах. Коли певна подія стає активною в системі, сервер може відправити HTTP-запит до URL-адреси, вказаної у вебхуці, для сповіщення про цю подію.

Вебхуки широко використовуються платіжними системами. Якщо платіж успішно опрацьовується, сервіс може надсилати вебхук до вказаної URL-адреси, що сповіщає про успішну транзакцію. Це дозволяє автоматично оновлювати статуси оплати на веб-сайтах або в інших системах без необхідності постійно перевіряти цю інформацію.

Вебхуки, це потужний інструмент, який дозволяє системам взаємодіяти із зовнішніми сервісами в реальному часі, сповіщаючи про події та виконуючи відповідні дії після їх виникнення. Stripe наполягає у використанні своїх вебхуків для подальшої обробки платежів на сервері. Кожна зміна яка відбувається на серверах Stripe з платежами вашего акаунту ініціює запит від Stripe до серверу, з інформацією про деякі зміни, наприклад платіж пройшов успішно, створення нового клієнта Stripe, відміна платежу і так далі. Сервіс наполягає у використанні лише тих вебхуків які потрібні системі для виконання своїх функцій. У налаштуваннях Stripe можна визначити на які події реагувати та створювати запити до серверу. Таким чином ми розвантажуємо сервер Stripe та засвідчуємося у тому що обробляємо мінімально потрібну кількість запитів.

У моїй кваліфікаційній роботі прописана обробка вебхук запитів Stripe на події оплат, та на платежі які потребують додаткових дій від користувача. Наприклад, якщо банк користувача потребує додаткових дій в аутентифікації транзакції, чи рахунок клієнта не має достатньо грошей:

```
  async handleEvent(event: Stripe.Event) {
    switch (event.type) {
      case 'payment_intent.succeeded':
        return await this.handlePaymentIntentSucceeded(event);
      case 'setup_intent.succeeded':
        return await this.handleSetupIntentSucceeded(event);
      case 'setup_intent.requires_action':
        return await
this.handleSetupIntentRequiresAction(event);
    }
  }
```

4 РЕАЛІЗАЦІЯ СЕРВІСУ СХОВИЩА ФАЙЛІВ

4.1 Обґрунтування вибору платформи

Стартапи, які стикаються з обмеженими ресурсами, часто розглядають стратегії ефективного використання доступних коштів та часу для максимальної користі від свого продукту. У світі, де використання мобільних технологій постійно зростає, важливо зрозуміти, що розвиток нативних додатків для кожної платформи може вимагати значних зусиль і фінансових вкладень. Розробка веб-додатків може бути привабливою альтернативою. Вона дозволяє стартапам створювати програмні продукти, які можуть працювати на різних платформах, у тому числі на десктопах та мобільних пристроях, мінімізуючи зусилля, необхідні для розробки і підтримки окремих нативних додатків.

Одним з ключових переваг веб-додатків є їх універсальність. Вони можуть бути запущені на будь-якому пристрої з веб-браузером, що робить їх доступними для широкого кола користувачів. Такий підхід спрощує розгортання та оновлення додатків, а також дозволяє ефективно керувати їхніми версіями. Однак, варто зазначити, що веб-додатки можуть мати свої обмеження у функціональності та доступі до апаратних можливостей пристроїв порівняно з нативними додатками. Незважаючи на це, для багатьох стартапів веб-додатки можуть стати ефективним рішенням, що дозволяє зосередитися на основних можливостях продукту та швидко виводити його на ринок. Однією з ключових характеристик веб-додатків є можливість оновлювати їх функціонал без необхідності встановлювати оновлення на пристроях користувачів. Зміни в системі відбуваються централізовано на сервері, тому користувачі автоматично отримують доступ до оновлень, коли вони відкривають додаток у своєму браузері. Це робить процес оновлення більш безболісним та зручним для користувачів і більш зручним для розробника, бо йому не треба думати про підтримку користувачів на кожній

попередній версії додатку, можна простіше та швидше без додаткових раундів тестування проводити оновлення системи.

Таким чином користувачі завжди використовують останню версію додатку, оскільки веб-додатки не вимагають встановлення окремих оновлень на пристроях користувачів. Це робить роботу з додатком більш безпечною і зручною для всіх зацікавлених сторін. Розробники можуть впроваджувати нові функції, виправляти помилки чи додавати поліпшення, і користувачі миттєво отримують доступ до цих змін без необхідності проводити окреме оновлення.

Проблему використання нативних для платформи функціональностей, такі як: камера, датчики, гіроскоп, обробка жестів на сенсорному екрані поступово вирішують розробники браузерів, доповнюючи клієнтський API доступами до даних подібних джерел. При написані веб-додатку у розробника відкривається доступ до Web-API щоб отримати контекст про пристрій на якому відкритий застосунок. Багато подібних API ще знаходяться у експериментальному стані, та не готові до застосування у комерційних проектах. Сучасні зміни у Web API розширюють можливості браузера, дозволяючи йому отримувати більше інформації про пристрій та навколишній середовище. Однією з цих API є Ambient Light Sensor API, який дозволяє отримувати дані про рівень освітлення навколишнього середовища. Geolocation API надає доступ до геолокаційних даних пристрою, що дозволяє визначати місцезнаходження користувача. MediaStream API дозволяє отримати доступ до камери та мікрофону пристрою. Battery Status API, надає інформацію про рівень заряду батареї, стан батареї та час, який залишився до розряду. Інші API, як Web Bluetooth API (для взаємодії з Bluetooth-пристроями), Web USB API (для роботи з USB-пристроями), а також Web NFC API (для роботи з NFC-модулями) також надають доступ до функцій пристрою через веб-браузер.

Ці зміни відкривають нові можливості для веб-додатків, дозволяючи їм взаємодіяти з різноманітними функціями пристрою безпосередньо через браузер. Це може сприяти створенню універсальних додатків, які працюватимуть на будь-якій платформі без необхідності розробки окремих нативних версій. Це може зробити веб-технології більш привабливими для розробників, оскільки вони матимуть доступ до широкого спектру функціональностей, що раніше були доступні лише через нативні додатки:

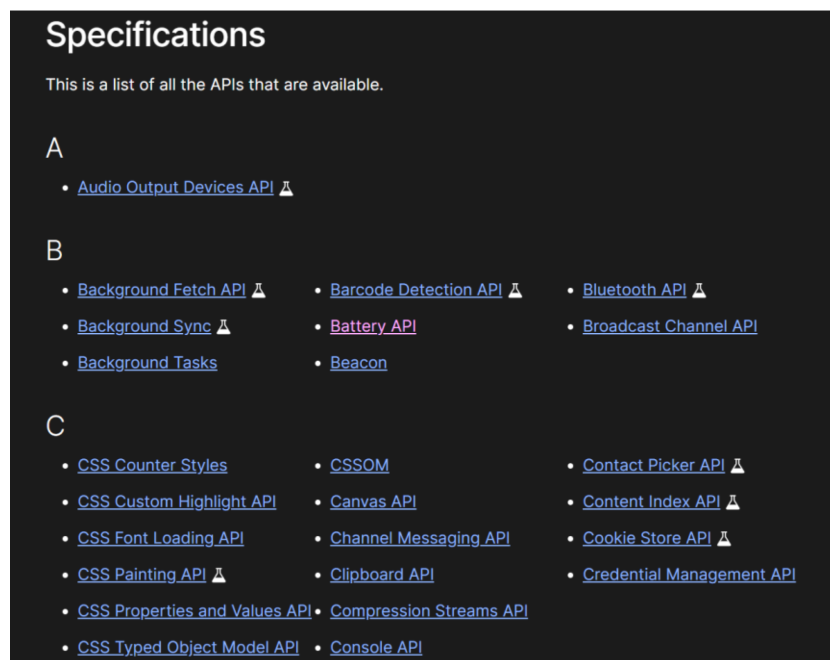

Рисунок 4.1 – Доступні API для використання (експериментальні помічені колбою)

Ще 5-10 років тому, у браузерів зовсім не було прав для отримання настільки глибоких даних. Наприклад, за допомогою Battery Status API через браузер можна дізнатися стан батареї пристрою. Цей API може бути цікавим у використанні в проектах з елементами розумного дому, чи розробці нативних програм за допомогою JavaScript, наприклад за допомогою технології Electron.

```
< ▸ Promise {<pending>}
Battery charging? No
Battery level: 56.99999999999999%
Battery charging time: Infinity seconds
Battery discharging time: 8181 seconds
```

Рисунок 4.2 – Стан батареї отриманий через Web API

Подібні зміни у напрямку розвитку браузерів як єдиною універсальною системою взаємодії з пристроєм свідчать про те що на сьогодень розробка нового продукту у вигляді веб-додатку є доволі перспективним і менш затратним ніж впровадження команд для розробки версій ПЗ для кожної доступної платформи.

Статистика говорить про те що більше половини користувачів смартфонів завантажує нуль додатків на свій мобільний пристрій у місяць. Ця статистика вказує на зміни у способі, яким користувачі сприймають мобільні додатки. Для розробників стартапів це важливе відкриття, оскільки воно вказує на необхідність ретельного аналізу ринку та пріоритетів при розробці продукту.

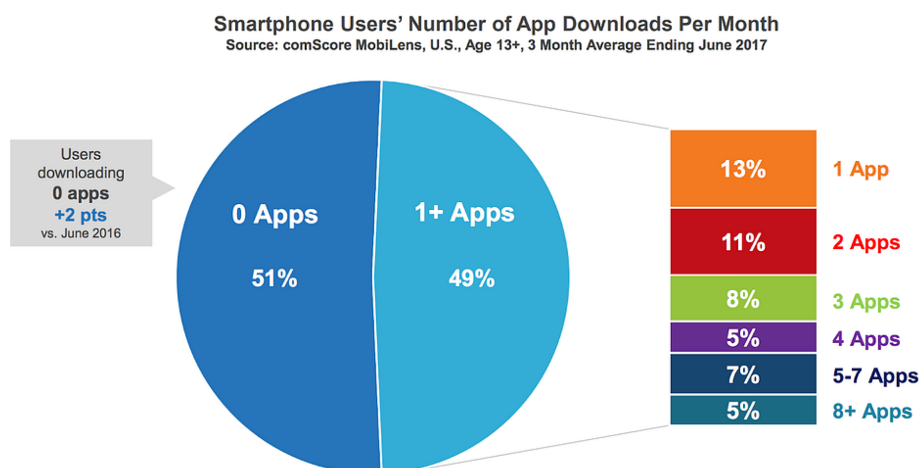

Рисунок 4.3 – Тенденція завантажування нативних застосунків на мобільний пристрій

Це означає, що для малих стартапів може бути розумним стратегічним кроком розробляти для веб-платформи. Ось кілька причин, чому це може бути вигідно:

- Доступність: веб-додатки доступні через браузер на будь-якому пристрої, що робить їх більш універсальними та доступними для ширшої аудиторії.
- Зручність: відсутність необхідності завантажувати та оновлювати додатки може зробити веб-застосунки більш зручними для користувачів.
- Витрати: розробка веб-додатків може бути витратнішою порівняно з розробкою нативних додатків для кількох платформ.
- Пошукова оптимізація: веб-додатки можуть бути краще оптимізовані для пошукових систем, що робить їх більш видимими для потенційних користувачів.

Для малих стартапів, які мають обмежені ресурси та хочуть привернути більше уваги до свого продукту, розробка для веб-платформ може бути стратегічним кроком, який дозволить досягти більшого охоплення аудиторії та збільшити свій вплив.

4.2 Вибір технологій

Для розробки сервісу хмарного сховища файлів, враховуючи попередньо проведений аналіз була вибрана платформа веб – найвигідніша платформа для розробки стартапів на сьогоднішній час, в тому числі у сфері хмарних сервісів. У якості бек-енд фреймворку для створення серверу для системи хмарного сховища даних був вибраний Nest.js.

Nest.js – це фреймворк для розробки масштабованих та ефективних веб-додатків на Node.js, який дозволяє легко створювати мікросервіси. Він надає структуровану архітектуру для побудови серверних додатків,

засновану на концепції модульності, яка ідеально підходить для мікросервісної архітектури.

Найчастіше для взаємодії між мікросервісами використовуються протоколи HTTP чи TCP.

Nest.js використовує модульну структуру і вбудовані засоби для створення масштабованих веб-додатків та API. Фреймворк базується на TypeScript, що дозволяє розробникам писати більш безпечний код за рахунок статичної типізації, що в свою чергу спрощує рефакторинг та робить код більш зрозумілим. Фреймворк пропонує модульну архітектуру, яка сприяє створенню масштабованих та легко супроводжуваних додатків. Це дозволяє розбивати функціонал на окремі модулі, що полегшує тестування та зберігання коду. Nest.js надає багато вбудованих функцій для автентифікації, авторизації, реєстрації, валідації даних та інших аспектів розробки, що дозволяє швидше створювати повноцінні додатки. Заснований на модульності, Nest.js має великий екосистему розширень та плагінів, які допомагають розробникам у вирішенні різних завдань, що сприяє швидкому розвитку проектів. Nest.js пропонує підтримку ООП, що полегшує створення та управління класами, що дозволяє розробникам створювати більш структуровані та організовані додатки.

Під час вибору фреймворку у якості одного з варіантів розглядався варіант використання ASP.NET – веб-фреймворку від компанії Microsoft. Його перевага в тому що більшість вбудованих пакетів протестована часом та підтримується Microsoft, що може дати більше впевненості в коректності роботи додаткових бібліотек. Більшість пакетів для Nest.js це розробки спільноти, тому не завжди можна на сто відсотків бути впевненими в якості їх роботи в великих продуктах. Команда Nest.js дуже пильно слідкує за проектами фанатів їх розробки, та іноді підключає свою команду для допомоги і інтеграції користувацьких бібліотек в основу Nest, тому можна бути впевненими в тому що з часом він стане еталоном.

Хоча Nest.js та ASP.NET - це фреймворки для розробки веб-додатків, вони мають кілька спільних рис, але також істотні відмінності:

- Nest.js базується на JavaScript/TypeScript, тоді як ASP.NET підтримує кілька мов, включаючи C#, F#, і VB.NET.
- ASP.NET є фреймворком, розробленим Microsoft, і має широку підтримку від спільноти та корпорації. Nest.js, хоча й має зростаючу спільноту, але ще не має такої великої корпоративної підтримки.
- Обидва фреймворки підтримують модульну структуру. Nest.js зазвичай використовується для створення REST API та мікросервісів, тоді як ASP.NET може працювати з різними типами проектів: від веб-додатків до масштабних корпоративних рішень.
- Nest.js спирається на популярний фронтенд фреймворк - Angular, тому його структура та підходи можуть бути ближчими до розробки на Angular. ASP.NET здебільшого використовується в .NET-екосистемі та інтегрується з Visual Studio.
- Nest.js є більш відкритим та платформенним, оскільки базується на JavaScript, що дозволяє використовувати його на різних платформах. ASP.NET, з іншого боку, досить тісно пов'язаний з екосистемою Microsoft та зазвичай використовується на платформах Windows.

Обидва фреймворки володіють своїми перевагами та властивостями, і вибір між ними зазвичай залежить від конкретних потреб та вимог проекту, а також від власних вподобань розробника чи команди. Я вже доволі давно слідкую за розвитком Nest.js та використовую його у своїх проектах, тому вирішив взяти цю технологію як основу свого дипломного проекту.

4.3 Спосіб доступу до бази даних

На сьогоднішній день, є три основних способи доступу до бази даних:

- Query Builder (Конструктор запитів);
- SQL код;

– ORM (Object-Relational Mapping, об'єктно-реляційне відображення).

В спільноті дуже давно і жваво критикуються та обговорюються всі доступні способи взаємодії з даними. У кожного з них є свої об’єктивні і суб’єктивні плюси та мінуси. Нажаль, поки немає універсального способу доступу до бази даних який би у всіх параметрах ідеального способу доступу до даних був би ідеальним: швидкість відповіді на запити, складність вивчення технології, швидкість імплементації під час роботи. Кожний з цих параметрів різниться для кожного способу роботи з базою даних, наприклад: ORM може бути доволі простою у використанні, та процес пізнання способу її роботи та синтаксису може бути також доволі прямолінійне, але дуже часто за це інженер платить тим що вихідні SQL запити які створює ORM є дуже не оптимізованими, через що страждає клієнт який очікує відповідь від серверу. Це не завжди так, сучасні ORM доволі добре створюють запити, але це, нажаль, все же не результат один до одного по швидкості при порівнянні з сирим SQL запитом. Ще одною проблемою ORM може бути недостатня підтримка всіх наявних типів баз даних, та їх унікальних функцій (наприклад використання типу даних enum у PostgreSQL), через це користувачу ORM все одно доведеться використовувати SQL, як і в випадку коли ORM не підтримує можливість робити унікальні оптимізовані та складні запити.

Конструктори запитів є чимось на межі ORM та сирого SQL коду, але вони не такі прості у використанні як ORM, і також можуть створювати не оптимізований код.

Незважаючи на це, індустрія намагається підтримувати розробку ORM і намагатися зробити їх краще, одною з таких молодих ORM-проектів є Prisma.

Prisma – це сучасна ORM для баз даних, яка пропонує зручний спосіб взаємодії з реляційними базами даних з використанням звичайних мов програмування та міграції даних. Інтеграція Prisma з Nest.js відбувається досить легко, адже Nest.js - це фреймворк для створення веб-додатків на TypeScript або JavaScript. Використання Prisma у проекті Nest.js дозволяє

зручно працювати з базою даних та використовувати сучасний підхід до роботи з нею.

Щоб інтегрувати Prisma у Nest.js, її потрібно встановити та налаштувати його для конкретного типу бази даних (наприклад, PostgreSQL, MySQL тощо) у Prisma Schema файлі.

Рисунок 4.4 – Налаштування Prisma для використання з PostgreSQL

Опис схеми бази даних у файлі (schema.prisma). Це визначає моделі даних та їхні взаємозв'язки. Prisma надає CLI-інструменти для генерації коду на основі схеми даних. Отримані моделі можна використовувати в сервісах та контролерах Nest.js для взаємодії з базою даних. Наприклад, ви можете виконувати запити до бази даних, створювати, зчитувати, оновлювати або видаляти дані, використовуючи згенеровані методи Prisma. Prisma також надає можливість виконувати міграції баз даних, що спрощує процес розвитку додатку та зберігання цілісності даних.

Міграція в контексті баз даних - це процес зміни схеми бази даних з метою збереження та оновлення даних у відповідності до нової структури.

Навіщо потрібна міграція:

– Структурні зміни: Додавання нових таблиць, зміна типів даних, видалення або модифікація колонок і зв'язків.

- Оновлення даних: Зміна формату або типу даних потребує оновлення існуючих записів, щоб вони відповідали новій структурі.

У Prisma міграції зазвичай виконуються через CLI-інструменти. Присутній інструмент командного рядка, який дозволяє автоматично створювати файли міграцій на основі змін у схемі даних. Ці файли описують, як база даних повинна змінитися для відповідності новій схемі. Файли міграцій застосовуються до бази даних за допомогою команди в CLI. Це виконує необхідні SQL-запити для зміни структури бази даних та оновлення даних. Prisma зберігає історію міграцій, дозволяючи розробникам відслідковувати та керувати змінами структури бази даних. Міграції можуть бути легко передані між розробниками для спільної роботи над структурними змінами.

Міграція в Prisma допомагає зберігати цілісність даних під час змін у схемі бази даних, забезпечуючи безпечну та ефективну роботу з даними в процесі розвитку проектів.

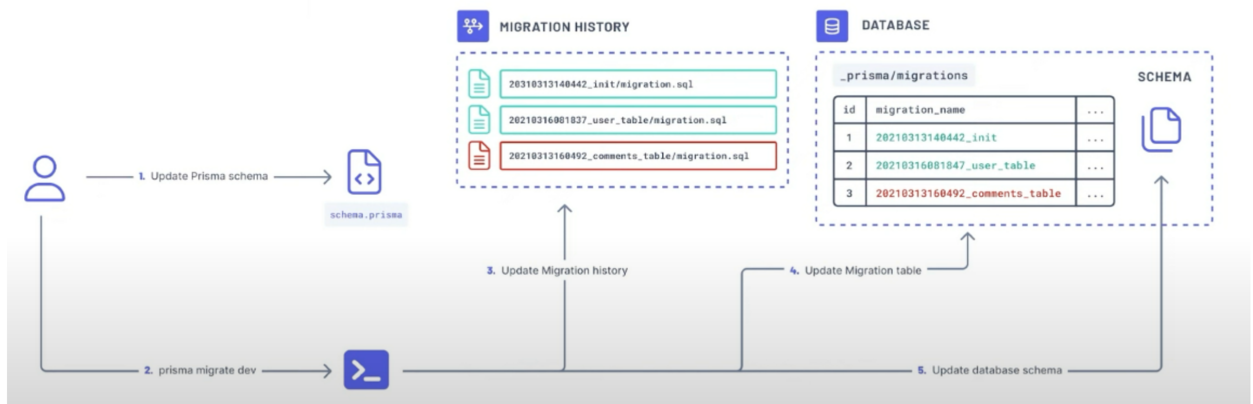

Рисунок 4.5 – Процес міграції у Prisma

У кожного розробника є свої улюблені технології, але якщо з мого боу суб’єктивно порівняти сучасні ORM то можна прийти до такого результату, що представлений у таблиці 4.1:

Таблиця 4.1 – Порівняння сучаних ORM

ORM	Швидкість	Функціонал	Документація	Крива навчання	Традиційність
Prisma	+	+++	+++	++++	+
TypeORM	++	++	++	++	+++
Sequelize	++	++	+	++	++
Hibernate	+++	++++	++++	+	++++

Prisma має найкращу на мій погляд документації серед інших варіантів таких як TypeORM чи Sequelize, але дуже не схожа на інші системи. Бо не використовує у собі ООП принципи як це роблять інші системи, тому навіть називати Prisma ORM на мою думку не досить вірно, тому что на виході Prisma повертає динамічний об’єкт а не клас. Hibernate дуже потужна ORM але доволі важка в роботі для починаючих розробників. Ця система перевірена роками ще з часів Java, тому їй можна довіряти, проект Prisma для порівняння почав використовуватись в проектах лише 3 роки тому, для програмного забезпечення цього недостатньо для того щоб бути на сто відсотків впевненим в тому що проект не припине існування, чи просто він якісний.

4.4 Автентифікація та авторизація

Автентифікація користувача грає важливу роль у безпеці та якості досвіду клієнта під час використання веб-сайту. Невдала реалізація цієї частини програми може радикально вплинути на захищеність користувацьких даних та системи в цілому, тому варто заздалегідь проаналізувати всі варіанти реалізації алгоритмів та систем безпечної автентифікації.

JWT (JSON Web Tokens) та сесії (Sessions) – це два різних методи автентифікації, кожен з яких має свої переваги та особливості.

Принцип роботи JWT – це компактний токен у форматі JSON, який містить інформацію про користувача та підписаний цифровим ключем. Він генерується на сервері після успішної аутентифікації та передається клієнту. Клієнт передає токен з кожним запитом, що дозволяє серверу перевіряти ідентичність та дозволи користувача без збереження стану на сервері. JWT є легким, компактним, переносним та не залежить від серверного стану. Він дозволяє розширену автентифікацію і авторизацію. Проблемою JWT є те що він не може бути відкликаний та має обмежену можливість в контролі за безпекою при зберіганні на клієнті, оскільки дані в ньому можуть бути прочитані (хоча й зашифровані).

Сесії – це механізм, за яким сервер створює унікальний ідентифікатор сесії (часто у вигляді токену або ID), який передається клієнту як cookie або як параметр у URL-адресі. Після цього, клієнт використовує цей ідентифікатор для доступу до даних сесії, які зберігаються на сервері. Сесії дозволяють більше контролю за безпекою, оскільки дані зберігаються на сервері, а не на клієнті. Їх легше відкликати або оновлювати. Вони можуть призводити до більшого навантаження на сервер через необхідність зберігання стану для кожної сесії.

Сучасні великі сервіси створюють навколо себе мережу пов’язаних сервісів для роботи з якими потрібен лише один аккаунт. Google має велику кількість таких сервісів, і використовує один обліковий запис для роботи з ними. Ця велика корпорація може собі дозволити поширювати свої облікові записи за даними користувачів для інших сервісів. API Google аутентифікації відкритий для всіх хто має намір використовувати обліковий запис Google як один з способів аутентифікації користувачів на третіх платформах.

Google OAuth2 – це протокол автентифікації та авторизації, який дозволяє користувачам надавати доступ до своїх облікових записів Google іншим службам або додаткам без передачі свого пароля. Він використовується для отримання доступу до ресурсів, представлених

користувачем на платформі Google, наприклад, до їхніх електронних листів, календаря, фотографій, документів тощо.

Використання Google OAuth2 може бути обмежене наступними способами:

- Google може застосовувати обмеження на кількість запитів чи доступ до певних ресурсів через OAuth2. Це може включати обмеження на кількість запитів на авторизацію за певний період часу.
- Інколи, в залежності від налаштувань проекту або API, можуть вимагати додаткової аутентифікації з боку користувача під час процесу авторизації через Google OAuth2.
- Деякі API Google можуть накладати обмеження на доступ до певних типів даних або дозволяти доступ лише певним видам ресурсів. Це може вплинути на рівень доступу, який дозволяє OAuth2 авторизація.

Для роботи потрібно лише мати підтверджений Google аккаунт, та перейти у консоль розробника Google, де створити свій додаток. Google згенерує ключі доступу завдяки яким буде проводитися автентифікація через сервера Google.

Для підтримки декількох способів аутентифікації було використано пакет - passport.js та його доробку для архітектури Nest.js - @nest/passport. Цей пакет абстрагує аутентифікаційні методи, чим дозволяє використовувати один інтерфейс для великої кількості різних видів аутентифікації. Для коректної роботи Google авторизації потрібно налаштувати так звану - passport-стратегію Google авторизації, на бек-енді, а в Google Cloud Console вказати перенапрямок для вдалої та невдалої авторизації. Далі під час вдалої авторизації - Google надішле публічні дані користувача, які Cloud Drive може використовувати у своїх цілях.

4.5 Валідація вхідних даних

Дуже важливо перевіряти які дані надходять до серверу. Некоректні дані можуть вразливо вплинути на систему в цілому:

- зміна значень даних у системі на нелогічні;
- доступ до приватних даних;
- знищення даних;
- SQL-ін’єкції;
- надмінне завантаження сервера;
- доступ до приватного функціоналу тощо.

Для того щоб зменшити вірогідність таких сценаріїв було використано модуль “class-validator” який займається валідацією даних які приходять до серверу з клієнтів. Як це працює: наприклад наш сервер має можливість реєстрації нових користувачів. Ми знаємо що користувач повинен надіслати адрес електронної пошти, пароль, та за бажанням - ім’я користувача. Наша задача - зробити класс, який буде являти собою той самий об’єкт запиту користувачем з зазначеними зверху даними. Це буде виглядати наступним чином:

```
export class RegistrationDto{
     @IsEmail()
     @IsNotEmpty()
     public email: string

     @IsNotEmpty()
     @IsString()
     @MinLength(6)
     @MaxLength(64)
     public password: string

     @IsOptional()
     @IsString()
     @MinLength(3)
     @MaxLength(64)
     public username: string
}
```

Як можна побачити, бібліотека class-validator також має можливість перевіряти дані які надходять на різні кількісні та якісні параметри, такі як чи є параметр строкою, чи його довжина більша за 3 та інші. Також можна помічати дані які є опціональними, чи навпаки не мають бути пустими. Команда Nest інтегрувала цю бібліотеку у свій фреймворк, тому по стандарту

при помилках валідації Nest відповідає на такий запит помилкою з кодом Bad Request 400 (поганий запит):

```
{
    "statusCode": 400,
    "error": "Bad Request",
    "message": [email must be an email]
}
```

Також є можливість кастомізувати поведінку ПЗ при фіксації некоректних вхідних даних, наприклад змінити формат відповіді, чи зробити власний декоратор для перевірки вхідних даних. У назві класу зазначена абревіатура DTO (Data Transfer Object) – один із шаблонів проектування, який використовують для передачі даних між підсистемами програми. Саме такі об’єкти перевіряються бібліотекою найчастіше.

4.6 Черги завдань

Деякі задачі які виконує сервер, не завжди працюють по алгоритму: запит від клієнта, обробка запиту, відповідь клієнту. Іноді клієнт повинен відразу отримати відповідь від API поки сервер почне роботу над часозатратним завданням. Використання черг є корисним для вирішення ситуацій, коли сервер має виконати завдання, що може зайняти тривалий час, та при цьому не блокувати інші запити від клієнтів. У фреймворку Nest.js можна використовувати пакет Bull, наприклад для розробки черги надсилання email листів. Формування email листу може бути довгою операцією, в залежності від реалізації та від складності шаблону, тому гарною практикою є мати окремий мікросервіс для цієї задачі, чи використовувати чергу операцій.

Що роблять черги:

- Асинхронні завдання: завдання, які можуть бути виконані поза основним процесом сервера, дозволяють зберігати і виконувати їх по черзі.
- Зменшення часу відповіді: коли клієнт подає запит, який не потребує негайної відповіді, черги дозволяють надіслати підтвердження про прийняття запиту швидко, а потім обробити його пізніше, коли сервер буде вільним.
- Контроль навантаження: черги дозволяють регулювати обробку завдань, запобігаючи перевантаженню сервера під час одночасної обробки багато запитів.

Щодо альтернатив, є інші сервіси, які пропонують різні підходи до керування чергами, такі як RabbitMQ, Kafka, Redis, ZeroMQ тощо. Кожен з них має свої переваги та недоліки в залежності від конкретних потреб проєкту, складності і пропускної здатності. Загалом, черги - це потужний інструмент, який дозволяє оптимізувати роботу сервера, розподіляючи завдання та забезпечуючи ефективну обробку запитів в асинхронному режимі. Bull виграє у інших сервісів своєю швидкістю інтеграції в систему – почати використовувати цей пакет доволі просто, треба лише встановити пакет до серверу, та мати працюючий Redis сервер, який Bull використовує для обробки та зберігання своїх майбутніх завдань.

RabbitMQ – це популярний брокер повідомлень (message broker), який використовується для обміну повідомленнями між різними частинами програмної системи. Навіть той факт, що він написаний на Java, не перешкоджає його використанню в багатьох інших мовах програмування, включаючи JavaScript, яка використовується в середовищі Node.js для розробки.

Основними перевагами RabbitMQ є: нучкість та масштабованість, RabbitMQ забезпечує широкі можливості налаштування для різних сценаріїв використання та гнучкість у роботі з повідомленнями. Він гарантує доставку повідомлень, навіть при виникненні помилок або відмов системи. RabbitMQ

підтримує різні протоколи комунікації, такі як AMQP, MQTT, інші. Він має багато бібліотек для різних мов програмування, що полегшує інтеграцію в різноманітні системи. Одним з недоліків може бути складність конфігурації та відсутність простоти у порівнянні з іншими брокерами, такими як Bull. Але саме ці багатофункціональність і гнучкість дозволяють вирішувати складні завдання комунікації між різними частинами системи.

На сайті документації Nest.js можна знайти підтримку для обох сервісів. Кожен з них має свої плюси та відмінності в конфігурації та використанні, тому вибір може залежати від конкретних потреб проєкту та його архітектурних особливостей.

4.7 Платіжний сервіс

Кожний бізнес, в першу чергу має на меті отримання прибутку. Сервіс який дозволяє зберігати величезну кількість користувацьких файлів повинен також мати непогану бізнес план, для того щоб в один момент не дійти до стану, коли грошей недостатньо для утримання, підтримки та обслуговування програмного забезпечення чи фізичних серверів та баз даних. Для цього потрібно розробити систему яка дозволить сервісу не тільки витрачати кошти та ресурси для користувачів, чим можна заробити впізнаваність у спільноті, але і мати спосіб отримувати прибуток від використання спільнотою ваших ресурсів у свої цілях.

Під час розробки плану проекту, було вирішено інтегрувати у систему платіжний сервіс Stripe.

Stripe – це платіжна система, яка надає інструменти для прийому онлайн-платежів. Для розробників це важливий інструмент, оскільки дозволяє легко та безпечно інтегрувати функціонал оплати на веб-сайт.

Stripe дозволяє приймати платежі в різних форматах – кредитні та дебетові картки, платіжні системи Apple Pay, Google Pay, а також інші методи. Stripe надає інструменти для безпечної обробки платежів та

відстеження їх статусу. Вони забезпечують захист від шахраїв та мають високий рівень безпеки. Можливість створення систем підписок, рекурентних та автоматичних платежів для продуктів чи послуг. Набір інструментів для аналізу платежів, створення звітів та статистики про операції.

Інтеграція Stripe у веб-сайт не така складна, як може здатися на перший погляд. Stripe надає документацію, SDK (Software Development Kit) та навіть готові бібліотеки для різних мов програмування, що спрощує процес інтеграції. Розробники можуть використовувати готові плагіни для популярних CMS, таких як WordPress або Shopify, або ж інтегрувати Stripe власноручно, якщо потрібно більше контролю. Однак, процес інтеграції може бути трошки технічно складним для початківців, особливо якщо це перша робота з оплатою через Інтернет. Однак з документацією та підтримкою Stripe це можливо зробити ефективно та швидко.

Інтеграція Stripe у веб-додаток з використанням React на клієнтській стороні та Nest.js на сервері відбувається у двох частинах: на клієнтській стороні використовується Stripe Elements для створення форм платежів, а на серверній – Stripe API для обробки платежів та отримання необхідної інформації від Stripe.

Stripe Elements – це набір готових компонентів, які дозволяють безпечно вводити дані карти. Ви можете імпортувати ці компоненти у ваш проект React та використовувати для створення форми оплати. Вони забезпечують можливість вводу карти та зберігання даних про платіж безпосередньо на сервер Stripe. Stripe.js – це JavaScript-бібліотека Stripe, яка дозволяє взаємодіяти з Stripe API, отримувати токени оплати та відправляти їх на сервер для подальшої обробки.

Stripe Node.js бібліотека використовується для взаємодії з Stripe API на сервері. Ця бібліотека дозволяє виконувати різноманітні операції, такі як створення та обробка платежів, створення підписок і т.д.

Загальна структура інтеграції виглядає наступним чином:

- Користувач заповнює дані карти через Stripe Elements на клієнтській стороні.
- За допомогою Stripe.js збирається інформація та створюється токен оплати.
- Токен оплати відправляється на сервер Nest.js.
- На сервері використовується Stripe Node.js бібліотека для подальшої обробки цього токену, проведення операції оплати через Stripe API.
- Створення та обробка платежів через Stripe API вимагає певних безпекових заходів, тому обов'язково слід дотримуватись належних практик розробки для забезпечення безпеки та конфіденційності платіжних даних користувачів.

Завдяки тому що Stripe має вбудовану архітектуру для різних сценаріїв та способів оплати, було вирішено використовувати систему платних підписок для користувачів сервісу Cloud Drive. Безкоштовний ліміт на використання сервісу (без платної підписки) – 10 гігабайт. Користувачі люблять прозоре та просте розуміння того за що вони платять, тому було вирішено зробити єдиний вид платної підписки – 15$ у місяць за 500 гігабайт користувацького простору.

4.8 Вибір та створення архітектури

Гарна система має бути готова для будь-якого навантаження, і завжди виконувати на сто відсотків запити кожного користувача з найменшим часом очікування, особливо важливим є метрика TTI (time to interactive), яка вказує на час який пройшов з часу початку завантаження сторінки до того моменту коли користувач може інтерактувати з нею.

TTI (Time to Interactive) – це метрика продуктивності веб-сторінок, яка вимірює час, який потрібен браузеру для завантаження сторінки та забезпечення інтерактивності для користувача. Основна ідея полягає у тому,

щоб користувач міг взаємодіяти з вмістом сторінки якнайшвидше після переходу на неї.

Розробники веб-сервісів можуть вплинути на TTI, оптимізуючи різні аспекти розробки:

- Оптимізація завантаження ресурсів: мінімізація та компресія файлів, асинхронне завантаження скриптів та стилів, використання кешування для прискорення завантаження.
- Використання CDN: використання Content Delivery Network (CDN) для швидкого доставлення вмісту користувачам з різних частин світу.
- Оптимізація JavaScript: використання легких бібліотек, відкладене завантаження скриптів, уникання блокуючого JavaScript, яке затримує завантаження сторінки.
- Кешування: використання кешування для збереження даних на клієнтській стороні та прискорення завантаження наступних сторінок.

Незважаючи на можливість впливу розробників, є обмеження, на які вони не мають прямого впливу, такі як швидкість інтернет-з'єднання користувача, обчислювальна потужність його пристрою та віддаленість до серверів, що також впливає на час завантаження сторінок. TTI важлива для стартапів та веб-сайтів, оскільки короткий час до досягнення інтерактивності сприяє збільшенню задоволення користувачів, зниженню відскоку та покращує загальний досвід використання веб-сервісу. Швидке реагування сторінки також сприяє підвищенню її позицій у пошукових системах.

Для підвищення швидкості роботи системи треба зазначити способи як це можна зробити. З збільшенням навантаження на систему, наприклад кількістю запитів на сервер в хвилину, є шанс збільшення часу на обробку окремих запитів, що в може вплинути на середній показник швидкості відповіді від веб-сервісу. У такий ситуації необхідно мати систему яка зручна для масштабування.

Горизонтальне та вертикальне масштабування - це два підходи до збільшення пропускної здатності та ефективності бекенд-систем.

Вертикальне масштабування – цей підхід полягає у збільшенні потужності сервера чи ресурсів окремого компонента системи. Наприклад, збільшення обсягу пам'яті, швидкості процесора чи додавання нових дисків. Це може бути як апгрейд апаратного забезпечення, так і оптимізація ПЗ. Подібний спосіб масштабування легше впроваджувати та керувати, оскільки потребує змін на одному сервері чи у межах одного компонента. Дієвий для систем, де важлива єдність даних або необхідно обробляти великі обсяги інформації. З мінусів подібного підходу може буди виникнення обмежень фізичних можливостей апаратних ресурсів. При підвищенні обсягів може збільшити загальні витрати на підтримку.

Горизонтальне масштабування – цей підхід полягає в додаванні нових екземплярів серверів чи компонентів для розподілу навантаження. Це може бути реалізовано через кластеризацію серверів чи розподілення додатків на кілька серверів. Цей спосіб є більш гнучким, оскільки може дозволити збільшення обсягів без значного збільшення навантаження на окремі компоненти. Також він підвищує надійність та доступність, оскільки якщо один сервер вийде з ладу, інші можуть продовжувати працювати. З мінусів можна зазначити складніше впровадження та керування, оскільки потребує координації та синхронізації між різними серверами. Не завжди ефективний для задач, де необхідно одночасно використовувати єдину базу даних або ресурси.

Обидва підходи мають свої сценарії застосування. У більшості випадків, оптимальний вибір полягає в комбінації обох методів - вертикальне масштабування для підвищення продуктивності окремих компонентів та горизонтальне масштабування для розподілу навантаження на кілька серверів для забезпечення ефективності та надійності системи.

Масштабування, будь то горизонтальне чи вертикальне, може впливати на метрику TTI у різних ситуаціях. Горизонтальне масштабування передбачає збільшення кількості серверів чи примірників програмного забезпечення. Це може поліпшити TTI через розподілення трафіку і

оптимізацію обробки запитів. Коли навантаження на сервер зростає, горизонтальне масштабування дозволяє розподілити навантаження між більшою кількістю серверів, що може покращити швидкість обробки запитів і відповідно TTI. Ці сервери виступають свого роду як Load Balancer (балансир навантаження).

З іншого боку, вертикальне масштабування передбачає збільшення ресурсів (потужності) окремого сервера чи пристрою. Це може також позитивно вплинути на TTI, оскільки більш потужний сервер може більш ефективно обробляти запити і відповідати на них швидше.

Обидва типи масштабування можуть мати вплив на TTI, але в конкретному випадку успішність може залежати від багатьох факторів, таких як тип додатку, його архітектура, поточне навантаження та рівень оптимізації коду. Найкращий спосіб вибору масштабування – експериментувати та проводити тестування, щоб знайти оптимальний підхід конкретного веб-сервісу.

Тому при розробці хмарного сховища файлів треба продумати потенційну можливість масштабування сервісу використовуючи обидва типи масштабування.

Популярним способом децентралізації залежності і відповідальності в великих та навантажених системах на зараз є підхід використання мікросервісної архітектури.

Мікросервісна архітектура — це підхід до розробки програмного забезпечення, при якому програма складається з невеликих, незалежних компонентів (мікросервісів), кожен з яких відповідає за виконання конкретної функції або послуги. Основна ідея полягає у розбитті складної системи на більш прості, самодостатні компоненти, які можуть працювати разом через мережу. Ця архітектура вирішує проблеми монолітних додатків, де всі функції об'єднані в одному блоку. Мікросервіси полегшують розвиток, оновлення та масштабування, оскільки кожен сервіс може бути розгорнутий, масштабований та оновлений незалежно від інших. Один з головних переваг

мікросервісної архітектури полягає у зручності масштабування. Кожен мікросервіс може бути масштабований окремо, щоб відповідати певним потребам навантаження, що робить цей підхід більш гнучким і ефективним у використанні ресурсів.

Сьогодні мікросервісна архітектура є досить актуальною, особливо у великих або швидко зростаючих проектах. Вона дозволяє розробникам швидше впроваджувати нові функції, покращувати окремі частини додатку без впливу на інші, та спрощує процес супроводу та розвитку програмного забезпечення. У порівнянні з монолітною архітектурою, мікросервіси виграють у більшій гнучкості, розділенні відповідальності та зручності масштабування. Однак вони можуть вимагати більшої уваги до керування мережевою взаємодією, а також збільшувати складність тестування та конфігурації.

Саме тому під час проектування системи хмарного сховища даних було прийняте рішення використання мікросервісної архітектури. Було прийняте рішення винести функціонал обробки файлів завантажених користувачами в окремий мікросервіс. Таким чином будь-яка фонова робота з обробки файлів не буде впливати на роботу основного REST-API серверу чим не знизить швидкість відгуку системи на запити користувачів.

У контексті створення мікросервісу для обробки файлів, Nest.js дозволяє розділити цю функціональність на окремий мікросервіс. Наприклад, можна створити сервіс, який приймає файли від користувачів, зберігає їх та виконує обробку. Цей мікросервіс може взаємодіяти з іншими частинами системи через HTTP або інші протоколи, обробляючи завантажені файли та повертаючи результати обробки.

Nest.js дає можливість побудувати модульну структуру, яка легко масштабується за рахунок розбиття на окремі сервіси. Це спрощує розробку, тестування та супровід мікросервісів, забезпечуючи велику гнучкість та швидкість розвитку програмного забезпечення.

4.9 Тестування роботи сервісу

Тестування є не менш важливою частиною розробки програмного забезпечення аніж імплементація нових функцій в систему. Дуже часто буває так що при додані нового функціоналу, який співпрацює з вже написаними модулями системи, виявляється що якийсь з модулів працює некоректно. Добре коли є система, яка може виявити проблему перед тим як оновлення з помилкою буде завантажене для використання користувачами. Коли кажуть про тестування програмного забезпечення, часто згадують інтеграційне та модульне тестування. Інтеграційне та модульне тестування - це дві важливі складові процесу тестування програмного забезпечення, але вони спрямовані на різні аспекти тестування.

Модульне тестування – це процес тестування окремих модулів, функцій або класів програмного забезпечення ізольовано від інших частин системи. Основна мета полягає в перевірці правильності роботи кожного окремого компонента програми. Основні аспекти модульного тестування: тестувальник тестує конкретний модуль без залучення інших частин програми. Це дозволяє швидко виявляти помилки в окремих частинах системи та робити зміни без страху, що це вплине на інші частини. Тести часто автоматизуються для ефективності. Це означає, що після написання тестів вони можуть бути запущені автоматично під час будь-яких змін в коді. Для писання модульних тестів часто використовують спеціальні фреймворки, такі як Jest для JavaScript або JUnit для Java. Nest.js підтримує тестування за допомогою Jest.

Модульні тести перевіряють правильність роботи конкретного модуля для впевненості, що він працює так, як очікується. Це дозволяє виявляти та виправляти проблеми на ранніх етапах розробки, що робить процес створення програмного забезпечення більш ефективним та економічним.

Модульне тестування забезпечує впевненість у коректності окремих частин програми, сприяє швидкому виявленню помилок та полегшує процес розробки, допомагаючи відокремити та перевірити кожну частину програми окремо.

Інтеграційне тестування – це процес перевірки взаємодії між різними частинами програми або системи, коли вони об'єднуються в єдину структуру. Основна мета полягає в перевірці правильності спільної роботи різних модулів, компонентів або сервісів як окремо, так і взаємодії між ними.

Основні аспекти інтеграційного тестування:

- Це тестування виконується для виявлення проблем у взаємодії різних частин системи.
- Пом'якшення залежностей: Часто різні компоненти системи мають залежності один від одного. Тестування допомагає виявляти помилки в цих залежностях.
- Тестування інтерфейсів: Це може включати тестування API, інтерфейсів користувача, взаємодії з базами даних тощо.
- Автоматизація інтеграційних тестів: Тестові сценарії можуть бути автоматизовані для ефективності. Це допомагає запускати тестування після будь-яких змін в коді автоматично.
- Складність та зручність тестів: Інтеграційні тести можуть бути складнішими, оскільки вони включають в себе більше частин системи, але вони дозволяють виявляти проблеми, які не вдається виявити на рівні окремих компонентів.

Інтеграційне тестування допомагає впевнитися в правильності взаємодії різних частин системи, забезпечує коректну роботу системи як цілісної одиниці та дозволяє виявляти проблеми на ранніх етапах розробки.

Системи у великих компаніях мають бути протестовані ідеально, тому вони здійснюють всі можливі способи тестування коду. На зараз є багато обговорень на тему необхідності модульного тестування, чи повинен кожний рядок коду та кожна функція мати набір тестів для фіксації помилок, чи це не

є важливим і краще фокусувати увагу на інтеграційне тестування, де завжди симулюється сумісна робота всіх модулів разом, і де можна створити багато різних сценаріїв контексту програми для перевірки її роботи. Тому кожна компанія і розробник сам вибирає який шлях йому обрати.

Підходи до розробки з додаванням тестування в завдання розробки бувають досить цікавими, наприклад деякі розробники є прихильниками методології розробки TDD.

Test-Driven Development (TDD) - це методика розробки програмного забезпечення, де спочатку пишуться тести для функціональності, яка має бути реалізована, а потім вже сама функціональність. Процес TDD складається з трьох основних кроків: "Red-Green-Refactor".

- Red (червоний): Спочатку пишеться тест, який описує очікувану поведінку функціоналу. Цей тест має провалитися (тобто мати стан "червоний"), оскільки функціонал ще не реалізований.
- Green (зелений): Після написання тесту, розробник реалізує мінімальний функціонал, необхідний для того, щоб тести пройшли успішно (тобто перейшли зі стану "червоний" в стан "зелений"). На цьому етапі функціонал може бути мінімальним та простим.

Після того, як тести пройшли успішно, розробник оптимізує код, покращує його архітектуру та чистоту, не змінюючи функціональність, щоб зробити код більш зрозумілим, ефективним та підтримуваним.

Основними перевагами TDD є більш висока якість кінцевого коду – тестове покриття допомагає виявляти баги та проблеми на ранніх етапах розробки, що призводить до стабільнішого та менш помилкового коду. TDD спонукає до створення коду, який краще відповідає вимогам, оскільки розробник спочатку думає про те, як буде тестувати функціонал. Відлагодження проблем стає простішим, оскільки при кожній зміні функціоналу виконуються всі тести, що дозволяє швидко виявити, чи не порушена попередня функціональність.

Існує кілька причин, чому деякі розробники вважають TDD неідеальним або непрактичним підходом: написання тестів перед реалізацією функціоналу може здаватися часовими витратами. Деякі розробники вважають, що спочатку розробляти тести, а потім сам функціонал, відкладає вирішення завдання, що може бути швидше виконане без TDD. Наочність і належна настройка тестового середовища для покриття широкого спектру вимог може бути важливою завданням, особливо для початківців. Для TDD потрібно писати більше коду в цілому, включаючи тести. Деякі вважають це надлишковим навантаженням. В проектах, які вже мають значний обсяг коду, додавання тестів пізніше може бути важким завданням і вимагати значних змін у вже наявній структурі коду. TDD вимагає певного рівня професійної експертизи, а деякі розробники можуть вважати, що для ефективного використання TDD потрібно багато навчання. Хоча TDD має свої переваги, не завжди він є підходом, який відповідає всім проектам та розробникам. У деяких випадках, зокрема, у деяких тісно зв'язаних з дедлайнами проекта або у проектах із значним обсягом вже наявного коду, інші підходи до тестування можуть бути більш практичними або швидкими. Зважаючи на ці негативні деталі, я вирішив не використовувати принцип TDD під час виконання дипломного проекту.

Я вирішив використовувати інтеграційне тестування для свого проекту. За допомогою цього я зміг описати сценарії більшостей функціональностей системи. Описані тести були як позитивні так і негативні, для того щоб збільшити обізнаність поведінки системи в неоднозначних ситуаціях. Гарна система, має передбачати обробку будь-яких сценаріїв, вхідних даних користувача – це підвищує її регідність. Якість інтеграційних тестів має важливе значення для остаточної якості продукту. Написання ефективних та комплексних інтеграційних тестів допомагає виявляти проблеми, що можуть виникнути при взаємодії між різними частинами програми, забезпечуючи таким чином більшу стабільність та надійність системи в цілому. Крім того, вони сприяють вчасному виявленню помилок та забезпечують швидше їх

виправлення, що зменшує витрати на виправлення проблем у майбутньому. Тому тести потрібно створювати добре розуміючись на нюансах та реалізації окремих модулів системи, тоді тести будуть дійсно допомагати в розробці.

Коли бюджет чи час не дозволяють проводити заходи щодо створення тестів, розробникам стає важче додавати нові модулі до системи. З розширенням функціоналу відстежувати всі зміни в коді, бути впевненим в тому що старі підсистеми працюють так само як і нові, що вони можуть дійсно добре взаємодіяти один з одним стає дуже тяжким завданням. Тому варто не нехтувати інтегруванням тестів у свої розробки. Альтернативою написанням тестів є використання мануального тестування.

Мануальне тестування та написання інтеграційних тестів – це дві різні стратегії, кожна з яких має свої переваги та обмеження.

Мануальне тестування – це процес, коли людина вручну перевіряє різні функції та можливості веб-додатку, спираючись на свої знання та досвід. Воно дає можливість оцінити веб-додаток з точки зору кінцевого користувача, виявити певні аспекти, які можуть бути пропущені під час автоматизованого тестування, і надати цінні враження про користувацький досвід.

Обидва підходи до тестування мають свої переваги та обмеження. Мануальне тестування може дозволити виявити аспекти, які важко автоматизувати, а також оцінити користувацький досвід в реальних умовах. Проте це часомірне та ресурсомістке завдання. Інтеграційні тести можуть швидше виявляти проблеми, є більш точними та менш витратними у довгостроковій перспективі, але не завжди відображають умови реального використання додатку.

У більшості випадків, оптимальна стратегія – це комбінація обох методів: автоматизоване тестування для швидкої перевірки коду та мануальне тестування для перевірки користувацького досвіду та нюансів, що не піддаються автоматизації.

Для додання інтеграціонних тестів у свій проект Nest.js я вирішив використовувати підхід завантаження окремого процесу серверу, у якому можлива підміна конфігурації. Тобто, під час запуску процесу тестування запускається сервер який підключається до заздалегідь створеної бази даних з даними які потрібні для тестування окремих модулів. Процес заповнення бази даних стартовими даними називається посівом (seeding). Посів відтворюється завдяки прописаними в коді сценаріями генерації сутностей, це можуть бути SQL команди які відтворюються у базі даних по команді початку тестів, чи виклики запитів до БД через ORM. У моєму випадку я використував створення фікчур (fixtures) – описані сутності у коді, які зберігаються у базу даних, та в подальшому їх можна використати для створення тестів.Процес тестування стартує завдяки бібліотекам Jest та Supertest.

Бібліотека Jest - це популярний фреймворк для тестування JavaScript, який дозволяє писати, запускати та організовувати різні види тестів (модульні, інтеграційні, функціональні тощо) для програмних проектів. Supertest - це бібліотека, яка надає можливість ефективно тестувати HTTP-запити до вашого веб-сервера, включаючи API.

Коли вони використовуються разом для тестування веб-додатку, Jest виступає як основа для організації й виконання тестів, а Supertest допомагає тестувати HTTP-запити, які надходять до сервера.

Таке поєднання дозволяє створювати тестові сценарії для перевірки роботи веб-сервера та його API. Jest надає інструменти для організації й запуску тестів, а Supertest дає можливість виконувати HTTP-запити та перевіряти їхні відповіді, переконуючись, що ваш веб-сервер поводиться коректно. Таке тестування допомагає виявити можливі проблеми в роботі сервера та API ще до того, як програмний код потрапить до реальних користувачів або в продакшен.

За допомогою Jest перед початком тестування можна виконати більшість підготовчих заходів, наприклад запустити базу даних для

тестування та провести її посів, для приведення даних до єдиного стану. Після проходу тестових сценаріїв, Jest видає результат зі статистикою щодо успішності проходження тестів:

```
PS npm test

> learning-jest@1.0.0 test
> jest

 PASS  ./index.test.js
  FizzBuzz
    √ [3] should result in "fizz" (2 ms)
    √ [5] should result in "buzz" (1 ms)
    √ [15] should result in "fizzbuzz" (1 ms)
    √ [1,2,3] should result in "1, 2, fizz" (1 ms)

Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        0.586 s, estimated 1 s
Ran all test suites.
```

Рисунок 4.6 – Результат проведення тестування за допомогою Jest

4.10 Реалізація шифрування користувацьких даних

Зважаючи на важливість відповідальність яку бере на себе система для зберігання користувацьких даних, дуже важливим є використання всіх можливих способів ефективного їх хешування та зберігання цього хешу у системі. По-перше, для ефективної роботи системи, потрібно зберігати файли у такому форматі, який буде складно декодувати зловмисникам якщо вони неправомірним чином отримають доступ до цих даних. По-друге, ці дані потрібно архівувати за допомогою сучасних алгоритмів щоб в перспективі зменшити вартість зберігання користувацьких даних.

Ефективне архівування та хешування даних є важливими компонентами забезпечення безпеки даних. Щоб забезпечити безпеку файлів, їх можна архівувати та хешувати відповідно до кращих практик безпеки даних.

Перш ніж зберігати дані, вони можуть бути зашифровані сучасними алгоритмами шифрування (AES, RSA тощо). Шифрування забезпечує

безпеку навіть у випадку, якщо зловмисники отримають доступ до архівованих даних.

Важливо зберігати метадані (наприклад, час створення файлу, тип файлу) окремо від вмісту файлу, щоб уникнути втрати цих важливих даних при архівуванні.

Хешування користувацьких файлів: використовують алгоритми хешування, такі як SHA-256 або SHA-512, щоб створити унікальний хеш (хеш-суму) для кожного файлу. Цей хеш може служити як "відбиток пальця" файлу і використовується для перевірки цілісності файлу. Додавання "солі" (випадкової додаткової інформації) до даних перед хешуванням може ускладнити спроби перебору паролів. Для максимальної безпеки рекомендується періодично перехешувати файли, особливо якщо це конфіденційна і важлива інформація.

Недостатньо лише архівувати дані користувачів для зменшення вартості їх зберігання, важливо привести цей архів у хеш формат який можна буди декодувати лише знаючи спеціальні секретні ключі. Такі ключі можна зберігати глобально для всієї системи, чи зробити алгоритм схожий на те що використовує сервіс хмарного зберігання файлів – MEGA.

MEGA – це хмарний сервіс зберігання файлів, який використовує шифрування на клієнтському боці, що означає, що дані шифруються перед відправкою на сервери MEGA. Однією з особливостей MEGA є те, що вони використовують пароль користувача як частину ключа для дешифрування файлів.

Коли користувач створює обліковий запис на MEGA, його пароль використовується для генерації ключа шифрування/дешифрування. Цей ключ не зберігається на серверах MEGA. Замість цього, він обчислюється на стороні клієнта на основі введеного пароля. Таким чином, саме пароль користувача виступає як ключ, необхідний для розшифрування його файлів. Цей підхід забезпечує додатковий рівень конфіденційності, оскільки навіть команда MEGA не має доступу до пароля користувача або ключа

шифрування. Проте, важливо зберігати свій пароль безпечним і не втрачати його, оскільки без нього доступ до даних буде втрачено назавжди. Цей підхід дозволяє MEGA забезпечити високий рівень приватності та безпеки, оскільки навіть у випадку порушення безпеки та доступу до серверів MEGA, шифрована інформація залишатиметься зашифрованою і недоступною для зловмисників без пароля користувача.

Зважаючи на те що в Cloud Drive присутня можливість переглядати та завантажувати файли інших користувачів, тобто не завжди файл бачить його автор, було прийняти рішення не копіювати реалізацію дешифрування від MEGA.

Алгоритм зберігання даних у Cloud Drive схематично можна представити наступним чином:

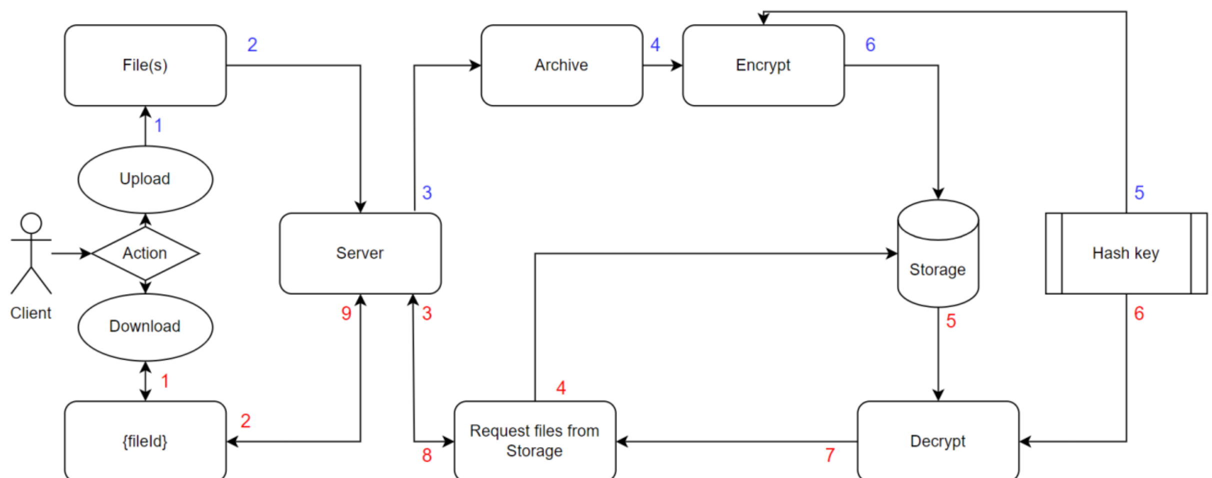

Рисунок 4.7 – Обробка користувацьких файлів у системі

Кожний сервіс хмарного зберігання файлів має виконувати 2 основні функції: завантажування та отримання файлів з серверу. Це основна робота сервісу, тому на неї потрібно спрямувати якнайбільше уваги. Схема на рисунку 4.7 зображає сценарій по якому працює отримання та завантаження користувацьких файлів у Cloud Drive. Під час завантаження клієнтом файлу

(чи файлів) на сервер, починається процес обробки «шматків» (chunks) кожного окремого файлу. Шматок файлу це довільна кількість байт-репрезентації цього файлу. Хорошою практикою є обробка завантаження файлів «по шматкам», це дозволяє уникнути помилки очікування запиту при завантаженні дуже великих файлів. Багато сервісів намагаються уникнути обробки файлів такого розміру, та роблять обмеження на розмір файлу який можна загрузити.

Якщо ваш сайт захищений таким сервісом як Cloudflare, стандартна його поведінка не дозволить клієнту сервісу робити запит який проходить більше трьохсот секунд, бо вважатиме його за шкідливий для безпеки.

Cloudflare – це компанія, яка пропонує різноманітні послуги для покращення безпеки, продуктивності та доступності веб-сайтів. Вони працюють як посередник між користувачами та серверами веб-сайтів, надаючи цілий набір функцій:

- CDN (Content Delivery Network): Cloudflare надає глобальну мережу серверів, які допомагають прискорити завантаження контенту на веб-сайтах. Вони розподіляють контент по всьому світу, дозволяючи користувачам отримувати доступ до сайту швидше.
- Захист від DDoS атак: системи Cloudflare можуть виявляти та блокувати DDoS-атаки, що забезпечує нормальне функціонування сайту навіть під навантаженням.
- SSL/TLS шифрування: Cloudflare дозволяє безкоштовно встановлювати SSL-сертифікати, які зашифровують трафік між користувачем та сервером, забезпечуючи більшу безпеку.
- Firewall та інші захисні функції: вони надають інструменти для налаштування правил безпеки, наприклад, блокування певних типів запитів або встановлення прав доступу.
- Аналітика трафіку: Cloudflare забезпечує засоби для аналізу трафіку, які допомагають розуміти, як користувачі взаємодіють з веб-сайтом.

Основна мета Cloudflare – покращити безпеку, швидкість та доступність веб-сайтів, зменшити навантаження на сервери та запобігти небажаним атакам і втраті даних.

Ще одна перевага використання завантаження «по шматкам» є постійна взаємодія серверу та браузеру у відображенні прогресу. Кожний успішно завантажений шматок є по суті HTTP запитом, який може повертати відповідь. При успішному завантажені шматка, сервер повертає на клієнт інформацію про це, включаючи додаткову мета-дату, наприклад відсоток завершеності загального завантаження.

Файли які часто скачуються з сервісу повинні зберігатися у кеші, для того щоб покращити користувацький досвід більш швидким відгуком на запит до файлу, а також розвантажити мікросервіс який займається розархівуванням архівів. Якщо файл, деякий час не був завантажений, його кеш можна видалити. Кешування у Node.js використовується для зберігання результатів попередніх запитів, щоб уникнути повторного обчислення або обробки. Основна мета кешування - поліпшити швидкодію сервера та зменшити час відповіді на запити. У Node.js кешування може бути реалізоване за допомогою різних бібліотек, таких як:

- node-cache: Ця бібліотека дозволяє зберігати дані в оперативній пам'яті, що дає швидкий доступ до них. Вона підтримує різні стратегії кешування, такі як TTL (час життя ключа), і дозволяє зберігати об'єкти, що зручно для багатьох типів даних.
- Redis: Redis – це інша популярна база даних ключ-значення, яка широко використовується для кешування. Її можна використовувати для зберігання кешу у пам'яті або на диску, що дозволяє швидко отримувати доступ до даних.
- Memcached: Ця технологія також використовується для кешування даних. Memcached працює подібно до Redis, зберігаючи ключі та значення у пам'яті.

Частота використання кешування у нових серверах написаних на Node.js залежить від конкретного випадку використання. Для швидких серверів, які мають велику кількість запитів, кешування може бути критичним для забезпечення оптимальної швидкодії та відповіді на запити. Однак, у деяких випадках, де дані рідко змінюються або мають низьку ймовірність повторного використання, кешування може бути менш важливим.

Кешування у серверах Node.js - це важливий механізм оптимізації, який дозволяє зберігати певні результати запитів у пам'яті сервера на певний час. Якщо дані запиту, що повертається клієнту, статичні або змінюються нечасто, використання кешування може значно поліпшити продуктивність системи. Це особливо корисно для великих обчислювальних або запит-інтенсивних операцій, таких як обробка складних запитів до баз даних або зовнішніх API.

Кешування дозволяє зберігати результати попередніх запитів і використовувати їх для майбутніх запитів, уникаючи повторних важких операцій. Продумана реалізація кешування може суттєво зменшити час відповіді на запити, зменшити навантаження на сервер та покращити загальну продуктивність системи. Однак важливо пам'ятати, що кешування повинно бути використане обережно, оскільки неправильне його застосування може призвести до невірних або застарілих даних, що може негативно вплинути на роботу системи. Також потрібно спостерігати за розміром даних які кешуються у оперативній пам'яті. Якщо об'єм кешованих файлів буде надо завеликим і оперативна пам'ять буде заповнена, можуть виникнути серйозні проблеми з роботою серверу, можливе навіть його відключення.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи магістра був створений сучасний веб-сервіс хмарного сховища користувацьких файлів. Для цього був проведений аналіз предметної області та порівняльний аналіз існуючих аналогів. Завдяки аналітичній роботі було запроваджено низку змін до логіки та реалізації деяких частин проекту.

Середовищем розробки веб-сервісу став редактор коду Visual Studio Code, у якості сервера використовується JavaScript виконувач Node.js з фреймворком Nest.js. Під час розробки серверної частини кваліфікаційної роботи було використано наступні підходи так технології: кешування, хешування, архівування, автентифікація та авторизація за допомогою пошти та сторонніх API (Google OAuth2), інтеграційне тестування, пагінація, мікросервісна взаємодія, валідація вхідних даних, сервіс для проведення платежів та інші. Основним способом взаємодії з базою даних стала сучасна ORM – Prisma. Основною бібліотекою для розробки візуальної частини сервісу використовувався React.

Бібліотеки в Node.js дуже корисні, оскільки виконують у собі різні трудомісткі завдання, звільняючи розробника від реалізації у своєму додатку. У роботі була використана низка бібліотек з відкритим вихідним кодом:

- passport – для налаштування аутентифікації за допомогою Google OAuth2. Passport також підтримує інтеграцію безлічі інших способів входу у систему, такі як: Meta, Github, Microsoft, Twitter та інші;
- bull – бібліотека для налаштування черг завдань у системі;
- mailer – бібліотека для створення і відправлення електронних листів;
- nest.js – фреймворк для обробки запитів та додавання інших сервісів у систему;
- JWT – бібліотека для створення та валідації JWT-токенів;

- class-validator – бібліотека для валідування будь-яких даних по заданому розробником формату. У Nest.js інтегрується за допомогою декораторів для перевірки всіх вхідних даних до визначених запитів;
- stripe – бібліотека для зручної взаємодії з сервісом прийому платежів Stripe;
- prisma – ORM для взаємодії з базою даних;
- та інші.

Запити до сервісу були покриті інтеграційними тестами, що підвищило його стабільність та ригідність.

Під час розробки кваліфікаційної роботи, було отримано велику кількість досвіду стосовно використання окремих технологій, таких як Prisma чи Nest.js, цей досвід можна буде легко перенести на наступні мої проекти. Також завдяки дослідженню предметної області сервісів для хмарного зберігання користувацьких даних, було зроблено багато відкриттів та висновків про роботу подібних сервісів, що допомогло мені при розробці особистого сучасного аналогу.

На етапі проектування були розроблені UML-діаграми варіантів використання (прецедентів), послідовності та блок-схеми роботи окремих функціональностей.

Досвід з аналізу предметної області сервісів хмарного зберігання даних доповідався на Студентській науковій конференції ОДЕКУ 30 травня 2023.

ПЕРЕЛІК ДЖЕРЕЛ І ПОСИЛАННЯ

1. Офіційний сайт Організації OWASP [Електронний ресурс] URL: https://www.owasp.org/
2. Іванов П.В. Криптографія для розробників, 2015, "Пітер", 2015, с. 153-171.
3. Douglas Crockford, JavaScript: The Good Parts, 2008, O'Reilly Media. с 31-55.
4. Стефанов С., Метц Д., Розробка веб-додатків з Node.js. БХВ-Петербург. 2017. с. 88-110.
5. Leonard Richardson, Mike Amundsen, Sam Ruby, RESTful Web APIs, O'Reilly Media, 2016. с. 1-44.
6. Офіційна документація Mozilla Developer Network [Електронний ресурс] URL: https://developer.mozilla.org/.
7. Kirupa Chinnathambi, Learning React: A Hands-On Guide to Building Web Applications Using React and Redux, Addison-Wesley Professional, 2018, с: 18-98.
8. Бен Форт, Рей Konцiлio, SQL за 10 хвилин, Вильямс, 2019, с. 42-74.
9. David Gourley, Brian Totty, Marjorie Sayer, Anshu Aggarwal, Sailu Reddy, HTTP: The Definitive Guide, O'Reilly Media, 2002, с. 300-320.
10. Мікеаль Касіс, Node.js в дії. Кибернетика, 2016, с. 1-67.
11. Prakhar Prasad, Mastering Modern Web Penetration Testing, Packt Publishing, 2016, с. 230-250.
12. Pedro Teixeira, Testing Node.js: Applying Unit, Integration, and Functional Testing with Examples in JavaScript, O'Reilly Media, 2015, с. 120-140.
13. David Flanagan, JavaScript: The Definitive Guide, O'Reilly Media, 2011, с. 155-198.
14. Микола Васильєв, Мережі для самостійних керівників. Технології VPN, IPsec, MPLS, VoIP, P2P, Кибернетика, 2017, с. 66-74.

15. Мікеаль Момо, NestJS: Від початків до експерта, А-ба-ба-га-ла-ма-га, 2020, с. 45-88.
16. Sandro Pasquali, Mastering Node.js, Packt Publishing, 2014, с. 55-95.
17. Михайло Малишевський, Відкритий урок мереж, Кибернетика, 2019, с. 110-130.
18. Akmal Chaudhri, Microservices Architecture: Make the architecture of a software as simple as possible, Packt Publishing, 2017, сторінки: 55-122.
19. Дуглас Крокфорд, JavaScript: The Good Parts, Вильямс, 2008, с. 30-50.
20. Martin Kleppmann, Designing Data-Intensive Applications, O'Reilly Media, 2017, с. 2-45.