МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
Факультет електроніки та інформаційних технологій

Кафедра комп'ютерних наук

Кваліфікаційна робота магістра

Інформаційна технологія проектування веб-ресурсу з підвищеними вимогами до шифруванням даних з використанням React.js

Здобувач освіти гр. ІН.м-01н Бурмака І.О.

Науковий керівник,
кандидат ф.-м. наук, доцент О.Б. Проценко

Завідувач кафедри
доктор технічних наук, професор. А.С. Довбиш

Суми 2022

Сумський державний університет
(назва вузу)

Факультет *ЕлІТ* Кафедра *Комп'ютерних наук*
Спеціальність *122 «Комп'ютерні науки»*

Затверджую:
зав.кафедрою _____
_____
"_____" _____ 20___р.

ЗАВДАННЯ
НА ДИПЛОМНИЙ ПРОЕКТ (РОБОТУ) СТУДЕНТОВІ

Бурмаці Іллі Олександровичу
(прізвище, ім'я, по батькові)

1. Тема проекту (роботи) *Інформаційна технологія проектування веб-ресурсу з підвищеними вимогами до шифруванням даних з використанням React.js*

затверджую наказом по інституту від "____" _____ 20 ___ р. № _____

2. Термін здачі студентом закінченого проекту (роботи) _____

3. Вхідні данні до проекту (роботи) _____
_____
_____
_____
_____

4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити) *1) Інформаційний огляд 2) Вибір програмних засобів 3) Практична реалізація*

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) _____
_____
_____
_____

6. Консультанти до проекту (роботи), із значенням розділів проекту, що стосується їх

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання _____

Керівник _____
(підпис)

Завдання прийняв до виконання _____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів дипломного проекту (роботи)	Термін виконання проекту (роботи)	Примітка
1.	*Інформаційний огляд*		
2.	*Вибір програмних засобів*		
3.	*Практична реалізація*		
4.	*Оформлення кваліфікаційної магістерської роботи*		

Студент – дипломник _____
(підпис)

Керівник проекту _____
(підпис)

РЕФЕРАТ

Записка: 65 стор., 27 рис., 1 табл., 16 додаток, 22 джерел.

Об’єкт дослідження — процес створення веб-додатку за допомогою React, Node.js.

Мета роботи — розробка та проектування веб-додатку для необанкінгу з шифруванням даних

Методи дослідження — методи шифрування даних

Результати — проаналізовано та розроблено веб-додаток для банку з використанням шифрування даних. В розробці використовується React, Node.js, а також модуль crypto для шифрування, із можливостями масштабування проекту, та створення додатку для Android та IOS. Розроблений веб-сайт реалізовано мовою програмування JavaScript.

ЗМІСТ

ВСТУП

Оскільки веб-програми продовжують отримувати доступ до великої кількості конфіденційних даних, які належать людям, організаціям і навіть урядам, загроза безпеці даних є найвищою. . Через свою доступність та простоту з точки зору безпеки вони часто стають ціллю для хакерів, тому рішення по захисту Web-додатків зараз є все більш актуальними. З перших днів програмування програмісти використовували криптографію та методи шифрування для захисту таких конфіденційних даних від зловмисників. Особливо після впровадження Інтернету методи криптографії відіграють вирішальну роль у гарантії безпеки даних

Результатом успішних атак зловмисників може стати витік або зміна конфіденційних даних компанії, несанкціонований доступ, зниження працездатності та недоступність публічних ресурсів, фінансові та репутаційні втрати.

У веб-розробці криптографія часто використовується для захисту даних, коли вони передаються по мережі або зберігаються в базах даних. Більшість криптографічних операцій виконується у веб-сервері

Традиційні засоби захисту часто не справляються з атаками на Web-ресурси. Інтернет-трафік легко проходить через периметр корпоративної мережі та отримує доступ до внутрішніх систем і серверів. Виявлення та усунення вразливостей в самому додатку, сайті або Web-ресурсі також часто не дає позитивних результатів - розробники можуть знаходити та виправляти тисячі вразливостей, але зловмиснику для проведення результативної атаки досить виявити всього одну. Професіонали в області захисту інформації, використовуючи у своїй роботі спеціалізовані програми і програмно-апаратні засоби, будуть успішно протистояти загрозам безпеки Web-додатків.

Метою даної роботи є розробка певного додатка з шифруванням даних, щоб злодій не мав можливості заволодіти ними також порівняння вже наявних методів для захисту Web-ресурсів Об’єктом дослідження є загрози інформаційній безпеці за допомогою несанкціонованого доступу та порушення політики безпеки. Предметом дослідження є програми для захисту Web-ресурсів. Мінімалістичний дизайн додатку виконаний в сучасному стилі з використанням Material Design, що дозволить сконцентрувати увагу саме на важливих для користувача елементах додатку.

1 ІНФОРМАЦІЙНИЙ ОГЛЯД

1.1 Аналіз предметної області

Зараз ми всі живимо в часи переходу від індустріальної епохи до інформаційної – коли стрімко зростає важливість накопичених та певним образом оброблених знань. З появою і стрімким розвитком комп'ютерних мереж велика кількість окремих людей і крупних організацій отримали ефективний та швидкий доступ до інформації. Однак, у локальних та глобальних комп'ютерних мереж, як і в інших способів передачі інформації, є недолік – загроза безпеці даних, а особливо, якщо не були організовані адекватні заходи для їх захисту від несанкціонованого доступу.

Інформаційні системи є одним із системотворчих факторів життя сучасного суспільства, і вплив інформаційної безпеки на всі сторони життя суспільства з часом буде лише зростати.

Саме тому у наш час інформаційного суспільства, одними з найголовніших інструментів вважаються засоби захисту інформації. Вони необхідні для забезпечення конфіденційності, секретності, довіри, здійснення авторизації, електронних платежів, корпоративної безпеки та безлічі інших надважливих атрибутів сучасності.

Безпечна діяльність повинна бути організована для всіх типів підприємств та установ, починаючи від державних організацій, банківських установ і закінчуючи дрібними приватними підприємствами. Захисту також підлягає і особиста інформація. Різниця полягатиме лише в тому, які засоби та методи та в якому обсязі знадобляться для забезпечення інформаційної безпеки у кожному конкретному випадку.

У зв'язку з цим все частіше для споживачі керуються під час вибору оптимального рішення наявністю вбудованих механізмів захисту інформації та ефективністю їх роботи в прикладних системах. Тому саме ці питання займають велику кількість уваги розробників програмних засобів.

Виходячи з історичної та міжнародної практики, об'єктами захисту, з урахуванням їх пріоритетів, є:

- Людина (особистість);
- Інформація;
- Матеріальні цінності.

Інформацію можна продати, купити, імпортувати, фальшувати, вкрасти і т. д., з цього випливає, що вона повинна якимось чином захищатися.

Завдання інформаційної безпеки – забезпечення захищеності інформації та інфраструктури, яка здійснює її підтримку, від різноманітних випадкових чи зловмисних впливів, в результаті яких може бути завданий збиток як самій інформації, і її власникам або підтримуючій інфраструктурі. Головними задачами інформаційної безпеки є мінімізація збитків, а також виконання прогнозів та запобіжних заходів таким впливам.

Напрямки інформаційних систем, які потребують захисту, можна розділити на категорії: забезпечення цілісності, доступності та конфіденційності інформаційних ресурсів.

Доступність слід розглядати як можливість отримання за кінцевий, бажано якнайкоротший, проміжок часу необхідної інформації або інформаційної послуги. Інформаційні системи створюються для одержання певних інформаційних послуг. Якщо отримання інформації з якихось причин стає неможливим, це завдає шкоди усім суб'єктам інформаційних відносин, і як наслідок – спричиняє загальний збиток організації.

Цілісність інформації визначає її актуальність та несуперечність, ступінь захищеності від руйнування та несанкціонованої зміни. Цілісність є основним аспектом інформаційної безпеки, оскільки порушення точності та правдивості інформації фактично можна розглядати як її втрату, а в більшості випадків неправдива спотворена інформація носить ще більш згубний характер, ніж її

відсутність. Наприклад, рецепти медичних ліків, поставлені діагнози, призначення препаратів та методів лікування.

Конфіденційність визначає захист від несанкціонованих доступів до інформації, її витік або передачу третім заінтересованим особам.

Існує кілька підходів до вирішення завдання забезпечення інформаційної безпеки. Спроба створення абсолютно надійного та недоступного іншим каналу зв’язку. Але досягнення такої мети досить складне, принаймні опираючись на існуючий рівень сучасного розвитку науки і техніки, який надає можливості не тільки здійснити передачі даних, а і отримати несанкціонований доступ до них;

Можна задіяти загальнодоступні канали зв'язку, але приховувати сам факт, що була здійснена передача будь-якої інформації. Даний підхід забезпечує наука стенографія. Але, на жаль, стенографія не має методів, які могли б надати гарантії високого рівня конфіденційності інформації;

Ще один підхід – використання загальнодоступного каналу зв'язку, але з передачею інформації в перетвореному виді таким чином, щоб її відновлення було можливе лише в адресата. Розробка методів перетворення даних, яке забезпечує їх шифрування, належить криптографії.

Необхідно зауважити, що робота криптосистеми відбувається за певними процедурами, які передбачають наявність:

- алгоритмів шифрування – одного або декількох, що можуть бути виражені математичними формулами;
- ключів, які будуть використовуватись цими алгоритмами шифрування;
- систем керування ключами;
- зашифрованого тексту (шифртексту).

Таким чином стає зрозуміло, що можливості криптографічних методів здатні забезпечити належний рівень захисту інформації.

1.2 Огляд існуючих програмних рішень

Криптографія - вивчає методології, які забезпечують конфіденційність та автентичність інформації.

Криптографія являє собою сукупність методів, завдяки яким можливо перетворювати дані таким чином, щоб вони виявилися повністю безкорисними для зловмисників. Завдяки таким перетворенням вирішуються дві головні проблеми безпеки інформації:

- захист конфіденційності;
- захист цілісності.

Проблеми захисту конфіденційності та цілісності інформації мають тісний зв’язок одна з одною, через що вирішувати їх часто можливо одними й тими ж методами.

Існує багато різних підходів до класифікації методів криптографічних перетворень інформації. Відштовхуючись від виду впливу на вхідні дані, можна виділити 4 групи методів криптографічного перетворення інформації:

- Шифрування;
- Стенографія;
- Кодування;
- Стиснення.

Процес шифрування відбувається за рахунок того, що здійснюються математичні, логічні, комбінаційні та інші перетворення вихідної інформації, а зашифрована в них інформація має вигляд набору букв, цифр, інших символів і двійкових кодів.

Щоб зашифрувати інформацію, необхідно задіяти алгоритм перетворення та ключ. Частіше всього, певний метод шифрування має незмінний алгоритм. Базування вихідних даних для алгоритму шифрування відбувається за рахунок

інформації, яка підлягає шифруванню, та ключі шифрування. У ключі міститься керуюча інформація, за допомогою якої відбувається вибір перетворення на певних етапах алгоритму та величин операндів, що використовуються у процесі алгоритму шифрування. Операнд - це константа, змінна, функція, вираз та інший об'єкт мови програмування, над яким проводяться операції. Під інформацією, що підлягає шифруванню, у загальному випадку можна розуміти текст.

Основний показник ефективності шифру – його криптостійкість. Її можна виміряти часом чи вартістю засобів, які необхідно задіяти криптоаналітикам для того, щоб отримати вихідну інформацію з шифротексту, при умові, що вони не знають ключ.

Збереження секретним алгоритму шифрування, який має широке використання, майже неможливе. Через це алгоритм не має містити прихованих слабких місць, які зможуть використати криптоаналітики. За умови виконання цього правила, визначення криптостійкості шифру буде визначатись довжиною ключа, оскільки єдиним шляхом для розкриття зашифрованої інформації буде перебір комбінацій ключа та виконання алгоритму розшифровки. Таким чином, довжина ключа та складність алгоритму шифрування має прямий вплив на час та засоби, необхідні для здійснення криптоаналізу.

Криптографія робить можливим перетворення інформації таким способом, щоб прочитати (відновити) дані було можливо лише із знанням ключа.

В якості інформації для шифрування і дешифрування можуть бути розглянуті тексти, які будуються на певному алфавіті. Ці терміни означають наступне:

- Алфавіт – представляє собою кінцеву множину знаків, що використовуються для кодування інформації;
- Текст - набір впорядкованих елементів алфавіту.

Прикладами алфавітів, якими користуються в сучасних інформаційних системах, можуть виступати:

- Алфавіт, що складається з 32-х літер алфавіту та пробілу;
- Алфавіт, що складається з символів, які належать до стандартних кодів ASCII та KOI-8;
- Бінарний алфавіт;
- Вісімковий та шістнадцятковий алфавіти.

Шифрування – це процес, під час якого вихідний текст (відкритий текст) перетворюється на шифрований текст.

Дешифрування – є зворотнім процесом до шифрування. Відбувається перетворення шифрованого тексту на вихідний на основі ключа.

Ключ являє собою інформацію, що необхідна для виконання шифрування та дешифрування текстів без перешкод. Процес шифрування та дешифрування представлений на рисунку 1.1.

Рисунок 1.1 - Процес шифрування та дешифрування інформації

Криптографічна система являє собою сімейство перетворень відкритого тексту, з параметром k – ключем, простором ключів K – набором значень, які може приймати ключ. Частіше за все ключем виступає послідовність (ряд) літер алфавіту, при чому ця послідовність обирається випадково.

Загальну схему роботи простої криптосистеми проілюстровано на рисунку 1.2

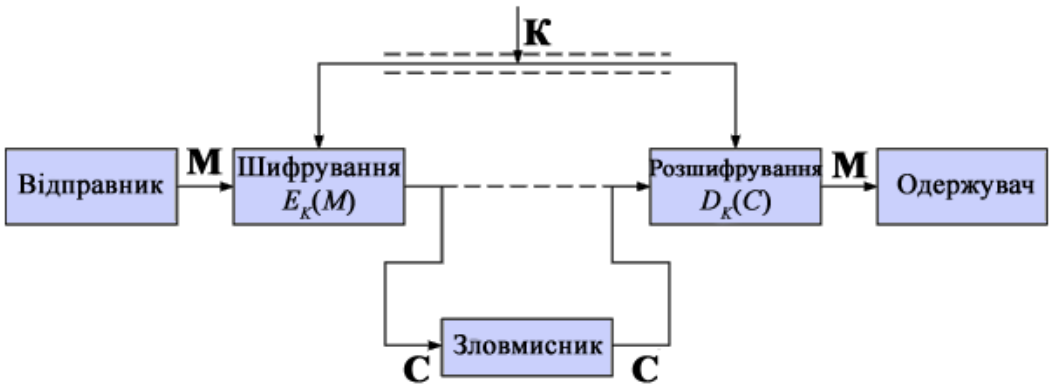

Рисунок 1.2 – Загальна схема роботи простої криптосистеми

Деталізація схеми криптосистеми вже матиме залежність від вибору методу, що впроваджується. Наприклад, у разі симетричного шифрування для процесів шифрування та дешифрування використовується той самий ключ (див. рисунок 1.3), у разі асиметричних алгоритмів використовується пара ключів (відкритий та закритий ключ).

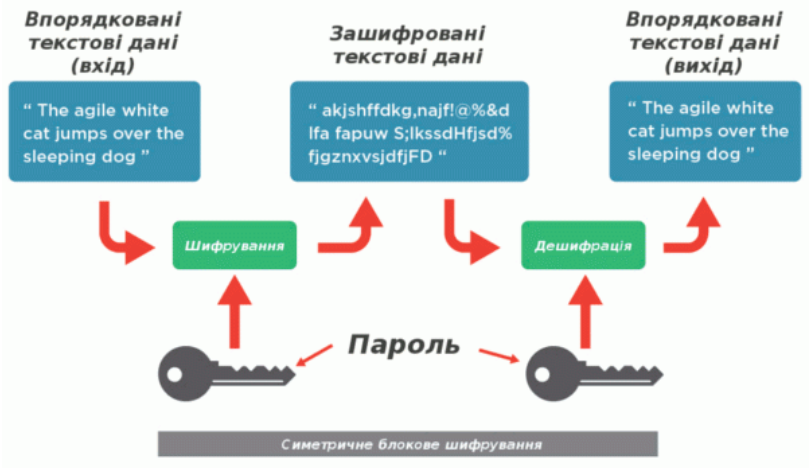

Рисунок 1.3 – Симетричне шифрування та дешифрування даних

Шифрування та дешифрування даних у симетричних криптосистемах виконується за допомогою того самого секретного ключа, обмін яким здійснюють завчасно по захищеним каналам сторони, що взаємодіють між собою.

Прикладами симетричних криптосистем є стандарти міжнародного рівня DES та AES.

AES – симетричний ітеративний блоковий алгоритм, зашифровує та розшифровує 128-бітові блоки даних. AES дозволяє використовувати три різні ключі довжиною 128, 192 або 256 біт. Шифрування складається з таких функцій перетворення, зображено на рисунку 1.4:

- Expand_key - Функція для обчислення всіх ключів;
- SubBytes - Функція для підстановки байтів, що використовує таблицю підстановок;
- ShiftRows - Функція, що забезпечує циклічний зсув у формі на різні величини;
- MixColumns - Функція, яка змішує дані всередині кожного стовпця форми;
- AddRound_key - Складання ключа раунду з формою.

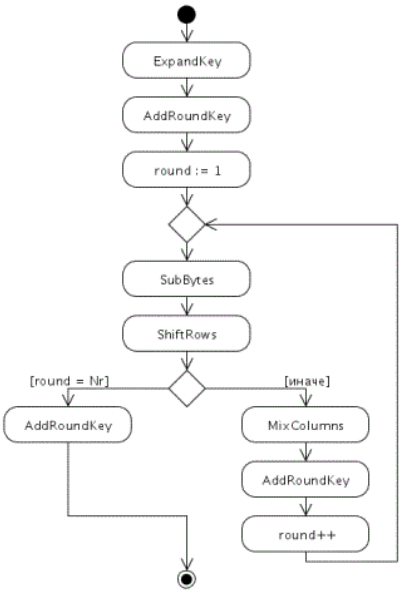

Рисунок 1.4 - Блок-схема роботи алгоритму AES

Розглядаючи симетричні шифри, варто виділити різновиди симетричних шифрів:

- Блокові шифри;
- Поточні шифри.

Блокові шифри працюють блоками, блок - сукупність біт фіксованої довжини. Таким чином, відбувається розбиття повідомлень блоки, потім ці блоки якимось методом шифруються. Спосіб шифрування вже визначається конкретним алгоритмом.

Наприклад, найпростіший і очевидний режим – ECB (Electronic Codebook): кожен блок шифрується, а далі всі блоки складаються (мається на увазі додавання рядків). Рисунок 1.5 показує цей режим роботи. Існують такі режими: CBC, PCBC, CFB, OFB, CTR.

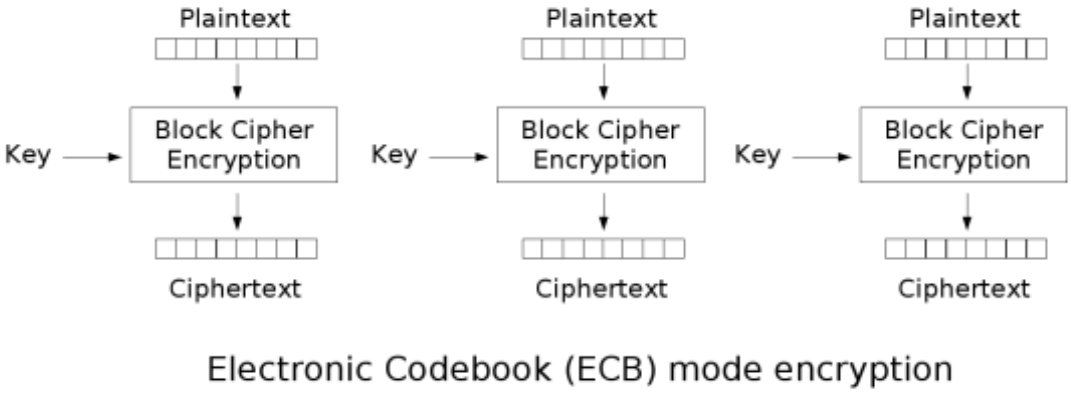

Рисунок 1.5 Режим шифрування ECB

Поточні шифри – такий вид шифру, в якому окремий символ відкритого повідомлення переходить у символ(и) шифротексту, причому результат шифру (мова йде про окремо взятий символ) залежить від ключа та розташування символу у відкритому повідомлення. Найпростіший випадок: відкритий текст складається з гамою (XOR) тощо. Гамма-випадкова послідовність символів (чисел). Поточне шифрування представлене на рисунок 1.6.

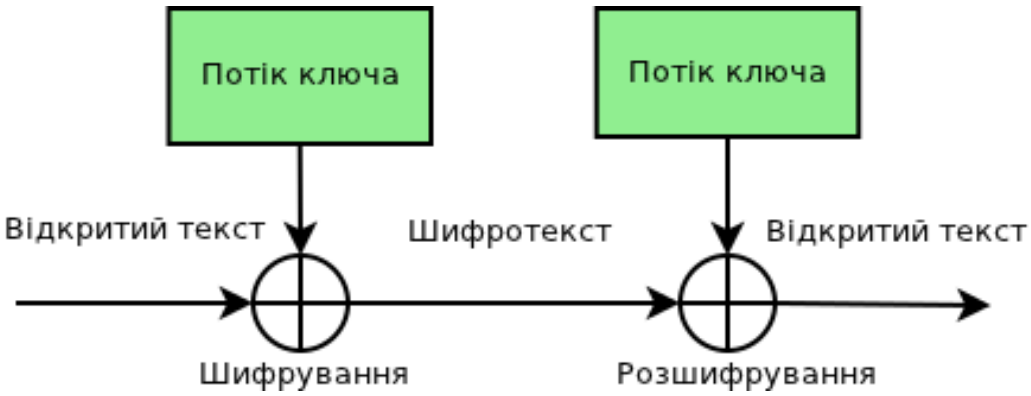

Рисунок 1.6 - Поточне шифрування

Необхідно відмітити, що за певних умов блокові шифри можна представити у вигляді потокового шифру.

В асиметричних криптосистемах для здійснення шифрування та розшифрування даних здійснюється використання різних ключей:

• Закритого (Private key) – відомого лише для свого власника. Одержувач являється єдиним власником закритого ключа, що необхідний для розшифрування, тому тільки він зможе виконати розшифрування призначених йому повідомлень;

• Відкритого (Public key) - може бути загальнодоступним, щоб будь-який користувач міг здійснити шифрування повідомлень для певного одержувача.

Головні відмінності ключової пари: закритий ключ дозволяє з легкістю обчислити відкритий ключ, при цьому – якщо в наявності є лише відкритий ключ, здійснити обчислення закритого ключа майже неможливо.

В цілому, завдяки використанню асиметричних алгоритмів знімається проблема поширення ключів під час використання симетричних криптосистем.

Симетричні алгоритми практично витіснені з комерційного та промислового споживання за рахунок досить високого рівня їхньої вразливості та низької криптостійкості порівняно з асиметричними. Хоча є приклади

симетричних алгоритмів, проти яких до сьогодні не розроблені універсальної ефективної атаки. Наприклад: модифікований AES – алгоритм RC6.

Сьогодні перспективними напрямками є застосування сімейств еліптичних кривих для шифрування даних, дані алгоритми теж відносяться до класу асиметричних. Загальна схема роботи алгоритму залишається такою ж, але внутрішня реалізація набагато складніша.

1.3 Постановка задачі

В даній роботі необхідно розробити зручний веб-додаток з шифруванням даних для фінансової установи, а саме з такими можливостями:

- адаптивний додаток(підтримка всіх форматів екрану),
- можливість спілкування в чаті,
- відкривання рахунку(картки),
- завантаження файлів;

Провівши аналіз – можливостей наявних алгоритмів, вирішено було створити на додаток на основі React та Node.js. Який має свої стандартні алгоритми для шифрування, та допоможе зробити власний шифратор даних та забезпечить цілісність. Результатом роботи має стати швидкий та безпечний веб-додаток з адаптивною підтримкою, та мати можливість подальшої інтеграції API, до мобільного додатку. Та саме головне бути «user friendly».

2 ВИБІР МЕТОДІВ ВИРІШЕННЯ ЗАДАЧІ

2.1 Вибір методів програмування

JavaScript став однією з найбільш затребуваних мов програмування за декілька останніх років. І, враховуючи тенденції, залишатиметься таким і у найближчому майбутньому.

В результаті проведеного на Stack Overflow в 2021 опитування, результати наведені на рисунку 2.1, 69.7% з 47 184 опитаних професійних розробників віддали перевагу JavaScript.

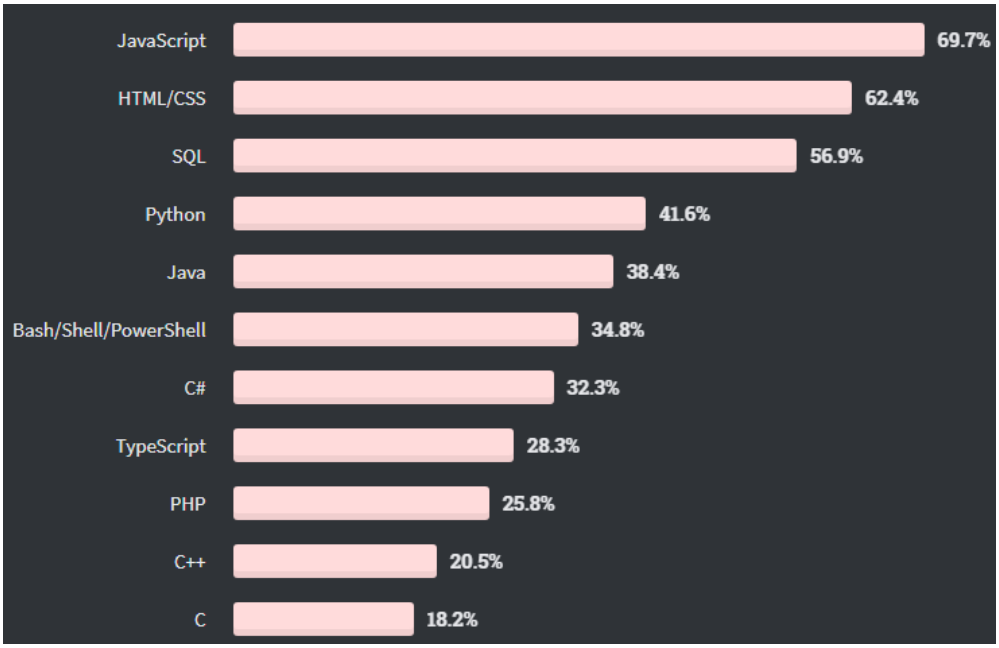

Рисунок 2.1 - Результати опитування, проведеного на Stack Overflow у 2021 році

Javascript популярний не випадково, а завдяки своїм безперечним перевагам:

- Незамінність для веб-розробки. Підтримка скриптів усіма популярними браузерами; повна інтеграція з верстkою сторінок (HTML+CSS) та серверною частиною (backend);

- Швидкість роботи та продуктивність. Javascript дає змогу частково обробляти веб-сторінки на комп'ютерах користувача без запитів до сервера. Це заощаджує час і трафік, знижує навантаження на сервер;
- Потужна інфраструктура (екосистема). Доступна велика кількість готових рішень у відкритому доступі, що полегшує роботу з Javascript та його фреймворками;
- Простота та раціональність застосування;
- Зручність інтерфейсів користувача. Заповнення форм, вибір дій, активація кнопок, перевірки введення, реагування на наведення/кліки миші тощо - це дає дуже високий рівень юзабіліті;
- Легкість освоєння.

Фреймворк – це абстракція, в якій програмне забезпечення, що надає функціональність додатку, може бути вибірково змінено додатковим користувацьким кодом, специфічне для програми, це універсальне програмне середовище багаторазового використання, яке надає особливу функціональність як частина великої програмної платформи для полегшення розробки програмних продуктів, продуктів і рішень. Фреймворки можуть включати в себе програми підтримки, компілятори, бібліотеки, набори інструментів і інтерфейси прикладного програмування (API), які об’єднують всі різні компоненти для розробки проекту або системи.

Для спрощення роботи з кодом корисно використовувати фреймворки. Фреймворки JS – це бібліотеки програмування JavaScript, в яких є попередньо написаний код для використання у стандартних функціях та завданнях програмування. Це основа для створення веб-сайтів або веб-програм.

JavaScript-фреймворки являють собою невід'ємну частину сучасної веб-розробки, надаючи розробникам перевірені та протестовані інструменти для створення масштабованих та інтерактивних веб-додатків.

Одна з найпопулярніших зв'язок технологій на ринку - Node.js + React.js. React — популярна фронтенд-бібліотека JavaScript для створення інтерактивних інтерфейсів користувача. Node.js — середовище бекенд-розробки JavaScript, яке дозволяє створювати масштабовані веб-програми з високою продуктивністю та низькою затримкою. Поєднання цих двох технологій дає величезні переваги при комплексній розробці веб-додатків.

Node.js – це середовище виконання JavaScript, яке створюється на віртуальних машинах Google Chrome, спеціально призначених для розробки швидких та масштабованих мережевих програм. Node.js використовує подієво-орієнтовану модель неблокуючого вводу-виводу, що робить цей інструмент легким та ефективним. Більш того, він дозволяє створювати веб-програми з використанням JavaScript на фронтенді та бекенді.

React дозволяє створювати повторно використовувані компоненти інтерфейсу користувача. Він дозволяє розробляти великі веб-програми для динамічного представлення керованих даних, а також забезпечує високу продуктивність та чуйність на різних пристроях.

Проста React-модель дає користувачам можливість легко розробляти масштабовані веб-програми на чистому Javascript без необхідності вивчати складні мови програмування.

Дані реального часу - Node.JS має подієво-орієнтовану архітектуру, що робить його ідеальним для додатків, що працюють з даними, що надходять у реальному часі. При використанні Node.JS з React можна легко створювати потужні веб-програми, які ефективно обробляють великі обсяги трафіку;

Прискорення процесу розробки – використання React та Node.js забезпечує високу окупність інвестицій (return on investment, ROI), а також економію часу та грошей. Ці дві технології добре поєднуються, забезпечуючи ефективну платформу для розробки швидких сайтів, які легко підтримувати протягом тривалого часу;

Масштабованість - React.js і Node.js дозволяють створювати веб-застосунки для динамічних представлень, керованих даними та реагуючих на різних пристроях. Це особливо корисно при роботі над великими проектами, які передбачають високу відвідуваність веб-додатка і потребують масштабованості для підтримання продуктивності сайту;

Одна мова для фронтенду та бекенда - використовуючи React з Node.js, розробникам не потрібно вивчати складні мови бекенда, такі як Ruby або Python. Можна використовувати одну мову для React-розробки інтерфейсу користувача та Node.js-розробки на стороні сервера. Це позбавить необхідності перемикатися між різними мовами програмування та фреймворками, заощадивши час, гроші та ресурси;

Висока продуктивність. Node.js – ідеальна платформа для високопродуктивних додатків. Вона використовує подієво-орієнтовану модель неблокуючого введення-виводу, що робить її легкою та ефективною. При використанні Node.js з React можна створювати потужні веб-програми, не турбуючись про повільний час відгуку або завантаження сторінок.

Односторінкові програми – дані два фреймворки забезпечують ідеальну платформу для створення потужних односторінкових програм. Ці програми більш ефективні, ніж традиційні, оскільки вони завантажуються лише один раз, а потім динамічно оновлюються в міру взаємодії з ними. Це може стати в нагоді при створенні складних веб-додатків, які повинні обробляти великий обсяг трафіку.

Чітко організований процес - фреймворки працюють разом, щоб забезпечити оптимізований процес розробки. Обидві технології розроблені для того, щоб бути швидкими, ефективними та масштабованими. Під час спільного використання вони допоможуть створити високопродуктивні корпоративні сайти.

Широке застосування JavaScript - об'єднання React.js з Node.js дозволяє використовувати всю потужність JavaScript для написання коду як для фронтенду, так і для бекенда. Це забезпечує велику гнучкість і свободу під час створення сайтів, надаючи розробнику можливість обходитися однією мовою під час виконання всіх завдань проекту. Додатково було прийнято рішення про використання бібліотеки crypto для Node.js, яка слугує для шифрування та дешифрування даних. А також TypeORM з Graphql, які дають потужні можливості при розробці додатку.

Для швидкого створення, тестування та розгортання програми можна використовувати програмну платформу Docker. Docker виконує упаковку програмного забезпечення в стандартизовані блоки, які мають назву контейнери. Кожен контейнер включає в себе все необхідне для роботи додатку: бібліотеки, системні інструменти, код і середовище виконання. Використовуючи Docker можна швидко розгортати та масштабувати програми в будь-якому середовищі та зберігати впевненість у тому, що код працюватиме.

Docker працює за рахунок використання стандартизованого способу виконання коду. Docker можна назвати операційною системою контейнерів. Контейнери відповідають за створення віртуального уявлення серверної операційної системи. Після того, як було здійснено встановлення на кожний сервер, Docker відкриває доступ до простих команд, за допомогою яких виконується збирання, запуск або зупинка контейнерів.

Контейнери загалом спрощують роботу як програмістам, так і адміністраторам, які розгортають ці додатки.

Docker вирішує проблеми залежностей та робочого оточення. Контейнери дозволяють упакувати в єдиний образ програму та всі її залежності: бібліотеки, системні утиліти та файли налаштування. Це спрощує перенесення додатка на іншу інфраструктуру;

Ізоляція та безпека. Контейнер являє собою сукупність процесів, які є ізольованими від основної операційної системи. Робота програм відбувається лише всередині контейнерів, вони не мають доступу до основної операційної системи.

Завдяки цьому підвищується безпека програм, оскільки в них немає можливості випадкового або навмисне завдати шкоди основній системі. Навіть якщо всередині контейнера відбудеться завершення роботи додатку з помилкою або він зависне – це ніяк не вплине на основну ОС;

Прискорення та автоматизація розгортання додатків та масштабованість. Контейнери спрощують розгортання програм. У класичному підході для встановлення програми може знадобитися виконати кілька дій: виконати скрипт, змінити файли налаштувань тощо. А завдяки контейнерам є можливість виконати автоматизацію цього процесу, оскільки до них включені всі необхідні залежності і порядок виконання дій.

Контейнери наближають до мікросервісної архітектури. Це підхід до розробки, при якому програма розбивається на невеликі компоненти, по можливості незалежні. Зазвичай протиставляється монолітній архітектурі, де всі частини системи пов'язані одна з одною. Це дозволяє розробляти нову функціональність швидше, адже у випадку з монолітною архітектурою зміна якоїсь частини може торкнутися всієї іншої системи.

Отже, мова JS динамічно розвивається, у відкритому доступі є велика інфосистема: бібліотеки, фреймворки, навчальні матеріали. Поєднання фреймворків React з Node.js — чудовий варіант для створення сучасних веб-застосунків, здатних обробляти великі обсяги даних. Поєднавши їх, можна отримати потужну комбінацію інструментів для розробки надійних сайтів.

Використання Docker дозволяє швидше і ефективніше доставляти або переміщувати код, стандартизувати операції, що виконуються додатками, і в цілому економить засоби, оптимізуючи використання ресурсів. Розглянувши та

порівнявши можливості створення власних додатків за допомогою різних інструментів, було прийнято рішення використовувати для розробки проектованого веб-додатку мову JavaScript, фреймворки React та Node.js, а також платформу Docker.

2.2 Середовище розробки

IDE (Integrated Development Environment) – це інтегроване, єдине середовище розробки, яке використовується розробниками для створення різного програмного забезпечення. IDE є комплексом з кількох інструментів, зокрема: текстового редактора, компілятора чи інтерпретатора, засобів автоматизації складання і наладчика. Крім цього, IDE може містити інструменти для інтеграції із системами керування версіями та інші корисні утиліти. Є IDE, які призначені для роботи лише з однією мовою програмування, проте більшість сучасних середовищ розробки дозволяє працювати одразу з декількома.

IDE є складнішим інструментом, ніж звичайний текстовий редактор. Незважаючи на те, що в текстових редакторах є багато корисних функцій на кшталт підсвічування синтаксису, єдине їхнє завдання - забезпечувати роботу з кодом.

Тобто для повноцінної розробки знадобиться ще хоча б компілятор та налагоджувач. IDE вже містить у собі всі ці та інші корисні компоненти. Під час вибору IDE необхідно звернути увагу на такі моменти:

- Наявність підтримки необхідної операційної системи. Якщо планується робота у команді, краще за все обирати кросплатформенні рішення;
- Наявність можливостей спільної розробки. Це важливо при командній роботі, коли планується вести роботу, використовуючи загальний репозиторій. Багато IDE платформ мають змогу інтегруватися з Git;
- Мови програмування, що підтримуються. Тут важливо подумати про довгострокові перспективи — можливо, через деякий час з'явиться необхідність

додати до проекту можливості, які реалізуються якоюсь іншою мовою. Тому краще обирати ті середовища, які підтримують декілька мов.

- Вартість. Є велика кількість безкоштовних середовищ розробки із відкритим вихідним кодом. Однак у платному програмному забезпеченні може бути доступна більша кількість корисних функцій.

Для реалізації практичної частини даного проекту, було прийнято рішення використовувати IDE WebStorm від JetBrains.

WebStorm - це інтегроване середовище для розробки на JavaScript та пов'язаних з ним технологіях. У IDE є все необхідне для роботи з JS, TS, React, Vue, Angular, Node.js, HTML і файлами стилів.

Зовнішній вигляд вікна середовища WebStorm представлений на рисунку 2.2.

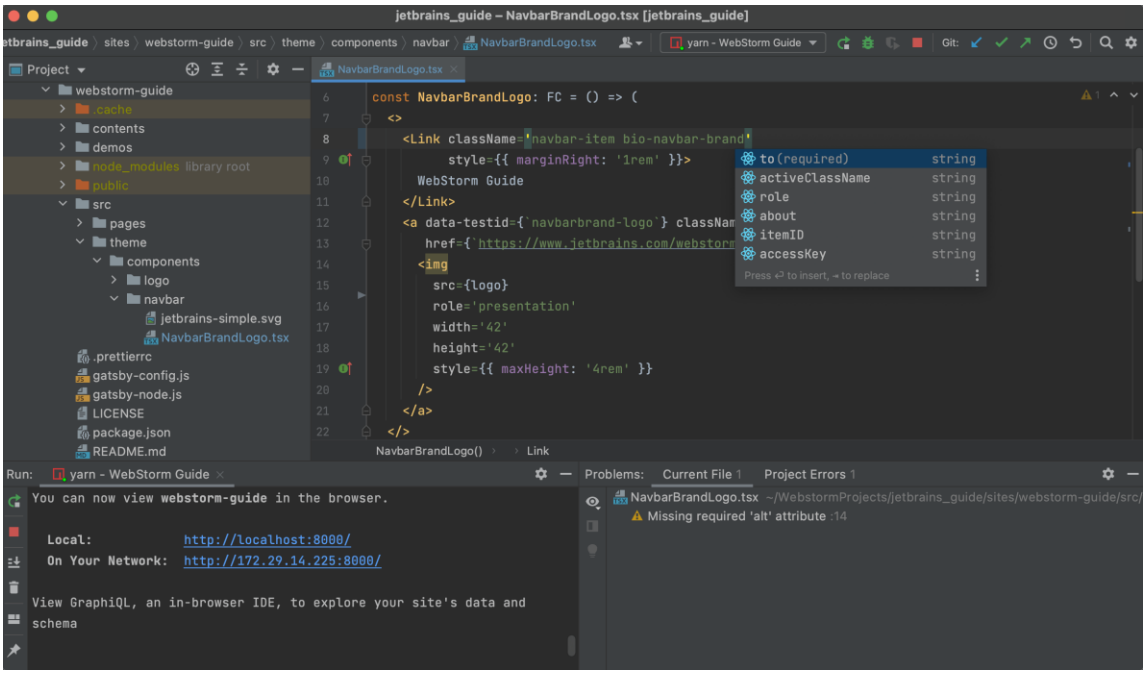

Рисунок 2.2 – Зовнішній вигляд IDE WebStorm

Головні можливості та переваги редактора WebStorm:

- Можливість писати надійний та підтримуваний код - WebStorm здійснює перевірку коду нальоту та допомагає знайти потенційні проблеми;

- IDE містить весь необхідний функціонал для розробки на JavaScript просто з коробки;
- У WebStorm є все необхідне для роботи з JS, TS, React, Vue, Angular, Node.js, HTML та файлами стилів;
- Розумний редактор коду добре розуміє структуру проектів та допомагає з будь-якими аспектами написання коду. Завжди доступні: автодоповнення коду, безпечний рефакторинг, постійний пошук потенційних проблем та підказки щодо їх виправлення та багато інших функцій;
- Вбудовані інструменти для розробників – середовище можна використовувати для налагодження та тестування клієнтського коду та програм на Node.js, а також для роботи з системою контролю версій. Доступні для використання інтегровані лінтери, інструменти складання, термінал та HTTP-клієнт;
- Швидка навігація та пошук дозволяє швидко переміщатися за кодом незалежно від розмірів проекту. Доступний пошук файлів, класів або символів, з можливістю перегляду всіх результатів в одному місці. Можна швидко та зручно здійснювати перехід до визначення функцій, методів, змінних, компонентів або класів;
- Ефективність командної роботи – доступна можливість програмувати разом із командою в реальному часі та спілкуватися з колегами прямо з IDE. Можна ділитися конфігурацією проекту, у тому числі параметрами стилю коду. Доступна інтеграція з Git та GitHub;
- Кастомізація – можливість налаштовувати інтерфейс на свій смак за допомогою різних тем і плагінів.

Також для розробки практичної частини проекту буде використовуватися DataGrip - інструмент від JetBrains для роботи з базами даних MySQL,

PostgreSQL, Oracle, SQL Server, Sybase, DB2, SQLite, HyperSQL, Apache Derby та H2.

Вікно середовища розробки DataGrip представлене на рисунок 2.3. Завдяки ньому можна аналізовати які дані були записані в базу даних та перевіряти коректність.

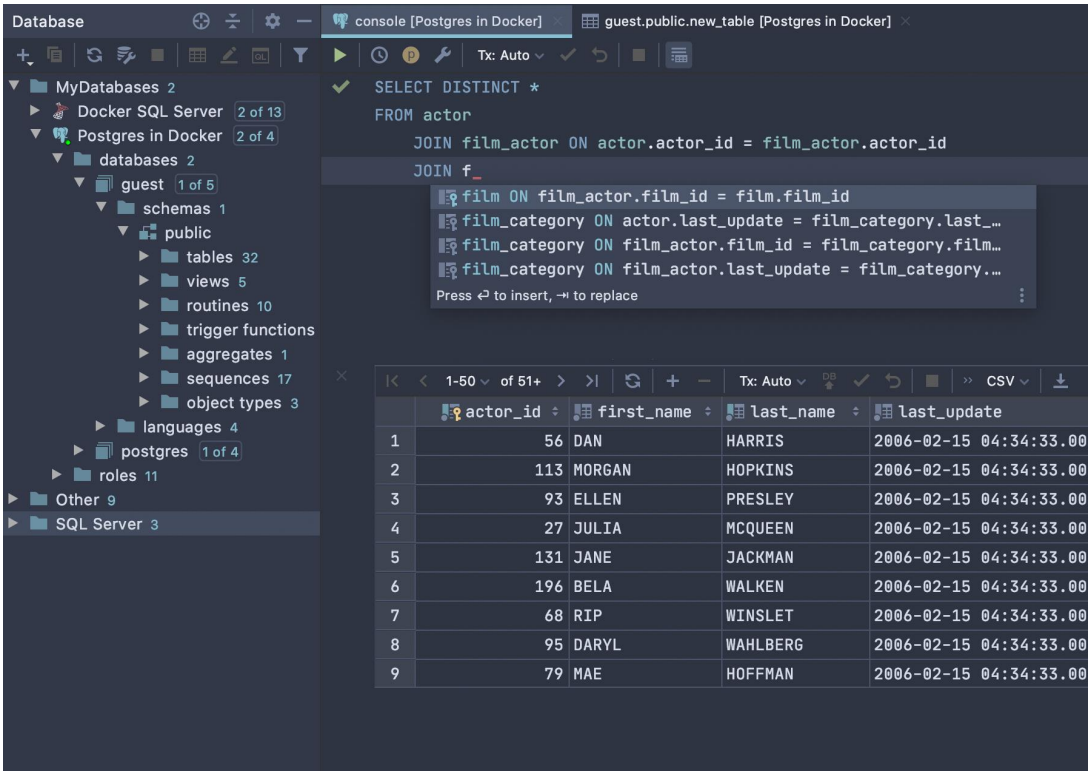

Рисунок 2.3 – Зовнішній вигляд вікна середовища розробки DataGrip

Головні можливості та переваги редактора DataGrip:

- Розумна консоль запитів – можливість писати та запускати запити у різних режимах: звичайному або «тільки для читання». Консоль зберігає історію запитів, клавіатурного введення та зберігається автоматично;
- Навігація по базі – можна знаходити будь-які об'єкти, набираючи ім'я у вікні пошуку, а також здійснювати перехід із SQL-скрипту до вихідного коду об'єкта або даних;
- Планувальник запитів – деревоподібний або графічний;

- Розумне автодоповнення коду – IDE розуміє контекст: пропонує ключові слова чи об'єкти залежно від того, що підходить у конкретному місці коду. Воно враховує зовнішні ключі, структуру об'єктів і навіть об'єкти, створені у тому самому скрипті;

- Аналіз коду та підказки - DataGrip виявляє можливі помилки в коді та пропонує найбільш підходящі варіанти виправлень на льоту;

- Рефакторинг – середовище розуміє, які об'єкти задіяні у скрипті. Якщо виконується перейменування псевдоніма чи змінної, вони будуть перейменовані у всіх місцях, де використовуються;

- Підтримка систем контролю версій – DataGrip підтримує роботу з Git, SVN, Mercurial та іншими системами контролю версій.

Отже, IDE WebStorm та DataGrip – зручні, сучасні та викопродуктивні середовища розробки. WebStorm підтримує популярні фреймворки для фронтенду (Angular, React, Vue.js) та бекенду (Node.js, Meteor). Має безліч важливих корисних функцій - розумне автодоповнення коду, вбудований наладчик, інструменти для тестування, інтеграцію з основними системами контролю версій. DataGrip незамінний для роботи з базами даних PostgreSQL, DataGrip включає потужний текстовий редактор з мультикурсорами, забезпечує синтаксичне виділення коду, підтримує інтеграцію із системами контролю версій. Саме тому ці інструменти були обрані для реалізації практичної частини даної роботи.

2.3 Вибір БД

У світі сучасних корпоративних технологій програмне забезпечення з відкритим вихідним кодом міцно зарекомендувало себе як одна з найбільших сил, з якою доводиться рахуватися. Зрештою, деякі з найбільших технологічних розробок з'явилися через рух відкритого вихідного коду.

Як і будь-яка інша категорія програмного забезпечення, функціональні можливості та можливості систем керування базами даних з відкритим вихідним кодом можуть суттєво відрізнятися. Не всі системи управління базами даних із відкритим вихідним кодом однакові. Вибираючи базу даних з відкритим вихідним кодом для свого проекту, важливо обрати зручну і здатну гнучко масштабуватися, а також з наявністю адекватних функцій безпеки.

Розглянемо три найбільш популярні СУБД – PostgreSQL, MariaDB, SQLite.

Оскільки PostgreSQL є надійною, безпечною та розширюваною базою даних, а також має в своєму розпорядженні велику екосистему доступних засобів, розробники використовують PostgreSQL у різноманітних сценаріях. Це програмне забезпечення сумісне з усіма основними операційними системами, включаючи Linux, Windows та Macintosh. Воно підтримує текст, зображення, звуки та відео. Завдяки цьому база даних користується популярністю у користувачів та компаній з різними потребами. PostgreSQL вважається однією з найбільш затребуваною технологією баз даних.

Переваги:

- Ефективність центрального алгоритму - що означає, що він перевершує багато баз даних, які рекламуються як більш просунуті. Це особливо корисно, коли ведеться робота з великою кількістю даних, для яких процеси введення-виведення можуть стати слабким місцем;
- Є однією з найгнучкіших баз даних з відкритим вихідним кодом. Доступна можливість писати функції у широкому спектрі серверних мов: Python, Perl, Java, Ruby, C і R;
- Підтримка спільноти PostgreSQL – одна з найкращих.

Недоліки:

- Нижча ефективність PostgreSQL для невеликих баз даних порівняно з високоефективною рабою з великими наборами даних;

- Недостатня розгорнутість базової документації;
- Для використання просунутих інструментів, таких як розпаралелювання та кластеризація, потрібні сторонні плагіни.

MariaDB – відкритий дистрибутив MySQL (випущений під GNU GPLv2). Він був створений після придбання Oracle MySQL. MariaDB була розроблена, щоб бути максимально сумісною з MySQL під час заміни кількох ключових компонентів. Використовує механізм зберігання Aria, який функціонує як транзакційний, так і нетранзакційний механізм.

Переваги;

- Часті поновлення систем безпеки;
- Відкритий вихідний код та сумісність з MySQL;
- Можливість встановити та запустити WordPress з MariaDB замість MySQL для кращої продуктивності та багатшого набору функцій.

Недоліки:

- Центральний файл журналу IDX має тенденцію ставати дуже великим після тривалого використання, що знижує продуктивність;
- Не достатньо швидкий процес кешування.

SQLite є широко використовуваним механізмом баз даних у світі завдяки його впровадженню у багатьох популярних веб-браузерах, операційних системах та мобільних телефонах. Він не є клієнт-серверним механізмом; швидше, його повне програмне забезпечення вбудоване у кожну реалізацію.

Це створює головну перевагу SQLite: у системах, що вбудовуються або розподілені, кожна машина несе повну реалізацію бази даних. Це може значно підвищити продуктивність баз даних, оскільки знижує потребу в міжсистемних викликах.

Переваги:

- Відмінно підходить для невеликих баз даних;
- Гарна швидкість роботи системи за рахунок невеликого розміру;
- Високо сумісна база даних. Це особливо важливо, якщо є необхідність інтегрувати систему зі смартфонами: система вбудована в iOS і буде служити в ній, доки існують сторонні програми.

Недоліки:

- Недостатність корисного функціоналу. Наприклад, відсутнє вбудоване шифрування даних, що стало стандартом для запобігання найбільш поширеним атакам хакерів в інтернеті;
- Широке використання та загальнодоступний код спрощують роботу з SQLite, але також збільшують площу його атаки;
- Хоча однофайловий підхід SQLite створює переваги у швидкості, немає простого способу реалізувати розраховане на багато користувачів середовище за допомогою цієї системи.

Виходячи з усіх перелічених особливостей різних СУБД, було прийнято рішення використовувати у роботі над проектом PostgreSQL.

PostgreSQL – це реляційна база даних з відкритим кодом, яка підтримується протягом 30 років розробки та є однією з найвідоміших серед усіх існуючих реляційних баз даних. Популярністю у розробників та адміністраторів база даних PostgreSQL зобов'язана своєю винятковою гнучкістю та цілісністю. Наприклад, база даних PostgreSQL підтримує як реляційні, так і нереляційні запити.

Основною характеристикою об’єктно-реляційної бази даних є підтримка визначених користувачем об’єктів та їх поведінки, включаючи типи даних, функції, оператори, домени та індекси. Це робить PostgreSQL надзвичайно

гнучким і надійним рішенням. Серед іншого, складні структури даних можна створювати, зберігати і читати. За результатами досліджень, нижче описані складні структури, які не підтримують стандартну СУБД.

PostgreSQL є найважливішим рішенням для баз даних у різних областях, таких як фінансові послуги, виробництво, роздрібна торгівля і логістика. Воно допомагає розробникам підтримувати цілісність даних, спрощує управління робочими навантаженнями будь-якого розміру та дозволяє виконувати масштабування за необхідності.

PostgreSQL відрізняється також значними перевагами для додатків з підтримкою геопросторових даних, а також додатків, що поєднують у собі часові ряди, JavaScript Object Notation Binding (JSONB) та реляційні дані. Крім того, адміністратори визнали високий рівень надійності PostgreSQL у забезпеченні збереження даних.

Ключові переваги використання PostgreSQL:

1. Доступ до потужних функцій - БД містить багато можливостей для користувачів. Наприклад, можна вибрати такі функції, як відновлення на час, попереджувальне ведення журналу, елементи деталізованого управління доступом, табличні простори, вкладені транзакції, оперативне резервне копіювання і багатоваріантне управління паралелізмом;

2. Надійність та відповідність вимогам - десятиліття розробки допомогли зробити базу даних PostgreSQL надзвичайно стійкою до відмови. Вона відповідає властивостям атомарності, узгодженості, ізольованості та довговічності (ACID) для транзакцій баз даних.

3. Крім того, PostgreSQL підтримує кілька мов для різних тригерів, атрибутів зовнішніх ключів, об'єднань та процедур, що зберігаються. PostgreSQL дозволяє працювати з найбільш поширеними типами даних, зокрема SQL. Крім того,

рішення підтримує Юнікод, міжнародні кодування та багатобайтове кодування символів;

4. Ліцензія на ПЗ з відкритим кодом - PostgreSQL надається за ліцензією на ПЗ з відкритим кодом, тому користувачі отримують більше гнучкості та можливостей для впровадження інновацій порівняно з комерційною системою баз даних;

5. Висока масштабованість - це програмне забезпечення може легко управляти великим обсягом даних. Масштабованість стосується і числа користувачів, що одночасно працюють у ній;

6. Різноманітні типи індексування та повнотекстовий пошук - БД пропонує користувачам різноманітні методи індексування, включаючи індексування на основі дерев B+, узагальнений інвертований індекс та узагальнене дерево пошуку, крім повнотекстового пошуку для пошуку за рядками та рядками векторних операцій;

7. Гнучкість - база даних сумісна з низкою найважливіших мов програмування і протоколів, включаючи C, C++, Go, Perl, Python, Java, .Net, Ruby, ODBC і Tcl. Це означає, що можна користуватись тією мовою, яку краще знаєш, без ризику виникнення системних конфліктів.

Розвинена екосистема підтримки – відкритий код PostgreSQL забезпечує користувачам підтримку профільної спільноти розробників, які постійно вдосконалюють систему, роблячи її безпечнішою та актуальнішою.

JSON - оскільки PostgreSQL підтримує реляційні та нереляційні запити, користувачі можуть отримати доступ до даних JSON за допомогою виразів шляху SQL і JSON.

Можливості розширення - PostgreSQL не просто зберігає дані - це програмне забезпечення дозволяє користувачам визначати функціональні мови та типи даних, включаючи типи, що настроюються або визначаються

користувачем. Крім того, розробникам доступні різноманітні розширення та надбудови, які допомагають налаштувати можливості PostgreSQL, включаючи PostGIS, Citus, pg_cron, HyperLogLog та t-digest.

Можна зробити висновок, що є безліч причин для вибору PostgreSQL як рішення для баз даних, такі як - підтримка складних структур і широкий спектр вбудованих і визначених користувачем типів даних, розширена ємність даних, гарне забезпечення цілісності даних, висока масштабованість та інші. Саме тому PostgreSQL було обрано реалізації практичної частини проектованої роботи.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1 Проектування web-додатку

Перш за все нам потрібно спроектувати backend частину додатку: а саме таблиці бази даних. Розпочнемо с сутностей. Для створення нео-банку необхідно мати такі таблиці:

- таблиця користувачів – user,
- таблиця файлів – asset,
- таблиця певним набором даних про людину – user_info
- таблиця рахунків – card,
- таблиця персоналу – worker,
- таблиця чатів – chat,
- таблиця повідомлень – message,
- таблиця транзакцій – transaction;

Наступне, це створення ERD-схеми для цих сутностей. На рисунку 3.1 можна побачити результат. Проаналізувавши можна зробити висновок, що нам не потрібно створювати додаткові таблиці, так як ми не маємо співвідношень N:N. Отже всього у нас буде 8 основних таблиць.

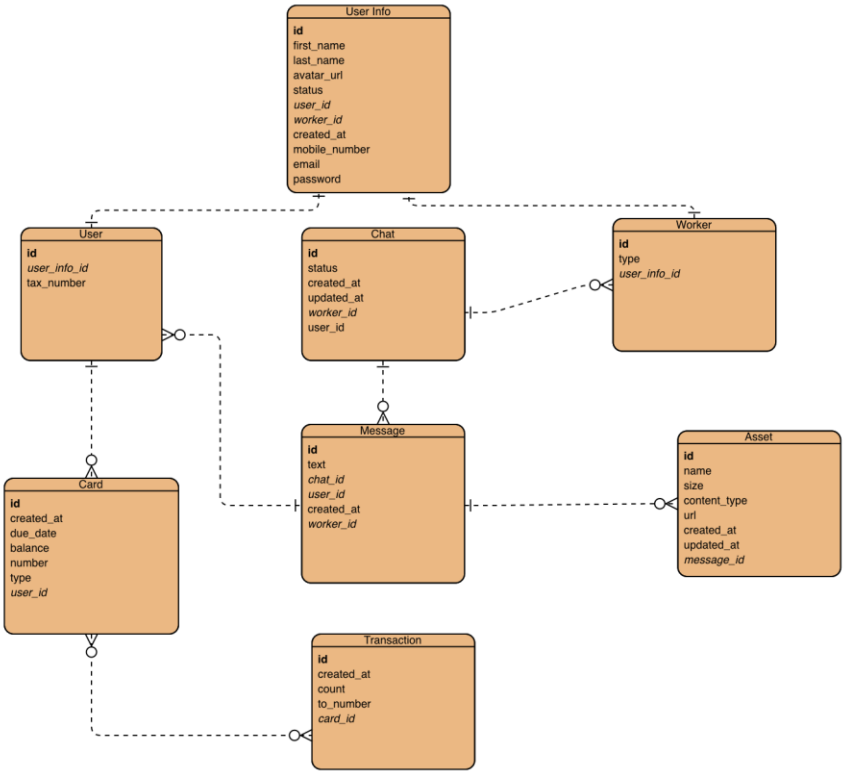

Рисунок 3.1 – ERD схема проектування відносин для БД

Далі потрібно проаналізувати можливі варіанти побудови групування файлів(для сутностей) та вибрати певний варіант. Всього є 2 гарні та сучасні варіанти побудови. Їх продемонстровано на рисунках 3.2 та 3.3.

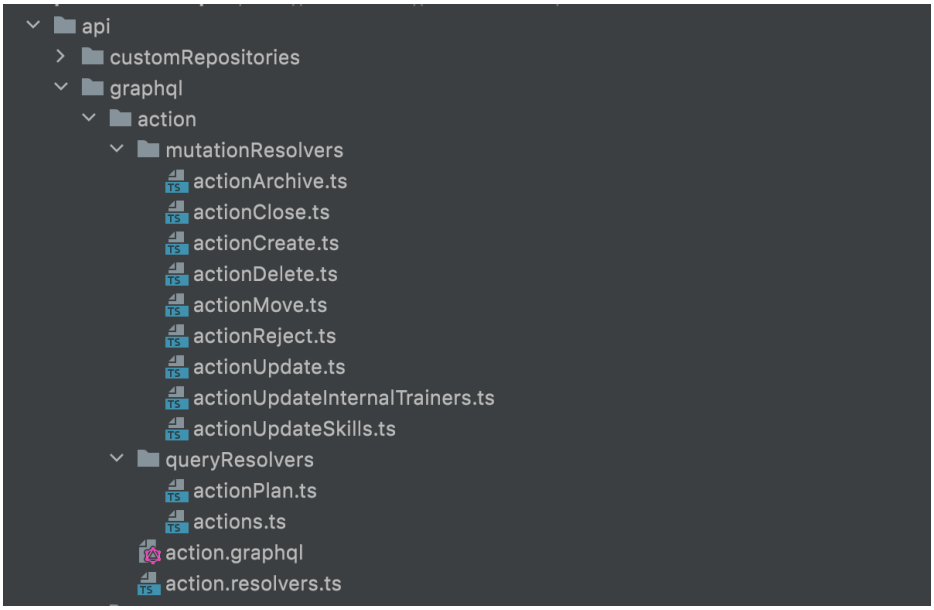

Рисунок 3.2 – Структура файлів, варіант групування в одну папку одна сутність

Рисунок 3.3 – Структура файлів, варіант групування всіх сутностей в одну папку для кожного блоку(модель, контролер, інші)

Проаналізувавши best-practices провідних розробників, розміщення файлів буде згруповано наступним чином, рисунок 3.4

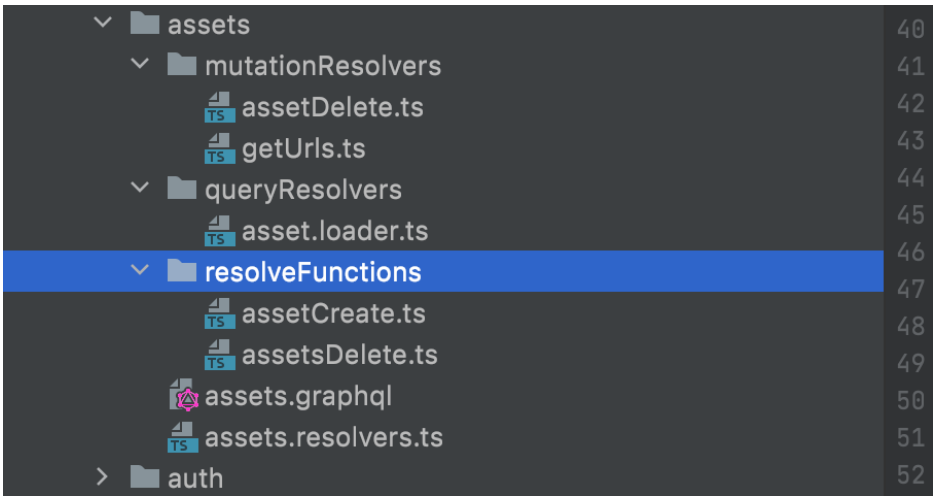

Рисунок 3.4 – Приклад побудови структури для сутностей web-додатку

3.2 Логічна реалізація бази даних

Проаналізувавши ERD схему(рисунок 3.1), потрібно описати поля для бази даних та їхнє значення. Для цього представимо імена необхідних таблиць, атрибутів, типів, їх призначення та обмеження - таблиця. 3.1.

Таблиця 3.1 Структура бази даних

Ім'я таблиці	Поле	Зміст	Тип	Ключ	Обмеження

Продовження Таблиці 3.1 – Структура бази даних

users	id	ідентифікатор користувача	uuid	PK	NOT NULL
	tax_number	ІПН людини	varchar	-	NOT NULL
	user_info_id	зовнішній ключ з таблиці user_info	uuid	FK	NOT NULL
user_infos	id	ідентифікатор	uuid	PK	NOT NULL
	first_name	ім’я	varchar	-	NOT NULL
	last_name	прізвище	varchar	-	NOT NULL
	avatar_url	посилання на аватарку	varchar	-	NULL
	status	статус акаунта	varchar	-	NOT NULL
	user_id	зовнішній ключ з таблиці user	uuid	FK	NULL
	worker_id	зовнішній ключ з таблиці worker	uuid	FK	NULL
	created_at	дата створення сутності	timestamp with time zone	-	NOT NULL
	mobile_number	мобільний номер телефону	varchar	-	NOT NULL
	password	пароль	varchar	-	NOT NULL
	email	почта	varchar	-	NOT NULL
workers	id	ідентифікатор	uuid	PK	NOT NULL
	type	тип працівника	varchar	-	NOT NULL
	user_info_id	зовнішній ключ з таблиці user_info	uuid	FK	NOT NULL
cards	id	ідентифікатор	uuid	PK	NOT NULL
	created_at	дата створення сутності	timestamp with time zone	-	NOT NULL
	due_date	кінцева дата активної картки	timestamp with time zone	-	NOT NULL
	balance	баланс рахунку	numeric	-	NOT NULL
	number	номер картки	varchar	-	NOT NULL
	type	тип картки	varchar	-	NOT NULL
	user_id	зовнішній ключ з таблиці user	uuid	FK	NOT NULL
transactions	id	ідентифікатор	uuid	PK	NOT NULL
	created_at	дата створення сутності	timestamp with time zone	-	NOT NULL
	count	сума переводу	numeric	-	NOT NULL
	to_number	номер картки на яку посилають	varchar	-	NOT NULL
	card_id	зовнішній ключ з таблиці card	uuid	FK	NOT NULL

Продовження Таблиці 3.1 – Структура бази даних

chats	id	ідентифікатор	uuid	PK	NOT NULL
	status	статус чату	varchar	-	NOT NULL
	created_at	дата створення чату	timestamp with time zone	-	NOT NULL
	updated_at	дата оновлення даних в чаті	timestamp with time zone	-	NOT NULL
	worker_id	зовнішній ключ з таблиці worker	uuid	FK	NOT NULL
	user_id	зовнішній ключ з таблиці user	uuid	FK	NOT NULL
messages	id	ідентифікатор	uuid	PK	NOT NULL
	text	повідомлення	varchar	-	NULL
	chat_id	зовнішній ключ з таблиці chat	uuid	FK	NOT NULL
	user_id	зовнішній ключ з таблиці user	uuid	FK	NOT NULL
	created_at	дата створення	timestamp with time zone	-	NOT NULL
	worker_id	зовнішній ключ з таблиці worker	uuid	FK	NOT NULL
assets	id	ідентифікатор	uuid	PK	NOT NULL
	name	назва файлу	varchar	-	NOT NULL
	size	розмір файлу	int	-	NOT NULL
	content_type	тип файлу	varchar	-	NULL
	url	посилання на файл()	varchar	-	NOT NULL
	created_at	дата створення	timestamp with time zone	-	NOT NULL
	updated_at	дата оновлення	timestamp with time zone	-	NOT NULL
	message_id	зовнішній ключ з таблиці message	uuid	FK	NOT NULL

Для створення таблиць, необхідно створити модель, та задати параметри, та виконати певні міграції. Розглянемо на прикладі таблиці chats, рисунок 3.5. Проект реалізовано за допомогою однієї з найпопулярніших фреймворка Nest.js та ORM(TypeORM), Graphql.

```
import { runForCompanySchemas } from '@utils/schemaHelpers'
import { MigrationInterface, QueryRunner, Table } from 'typeorm'

export class AddChatTable100000001 implements MigrationInterface {
  public async up(queryRunner: QueryRunner): Promise<any> {
    await runForCompanySchemas( callback: async (schema: string) => {
      await queryRunner.createTable(
        new Table( options: {
          name: `chats`,
          columns: [
            {
              name: 'id',
              type: 'uuid',
              isPrimary: true,
              default: 'public.uuid_generate_v4()',
              isNullable: false,
              isUnique: true
            },
            {
              name: 'updated_at',
              type: 'timestamp with time zone',
              default: 'now()',
              isNullable: false
            },
            {
              name: 'created_at',
              type: 'timestamp with time zone',
              default: 'now()',
              isNullable: false
            },
            {
              name: 'status',
              type: 'varchar',
              isNullable: false,
              default: "'ACTIVE'"
            },
            {
              name: 'worker_id',
              type: 'uuid',
              isNullable: false
            },
            {
              name: 'user_id',
              type: 'uuid',
              isNullable: false
            }
          ],
          foreignKeys: [
            {
              name: 'FK_chats_users_user_id_id',
              columnNames: ['user_id'],
              referencedColumnNames: ['id'],
              referencedTableName: `users`,
              onDelete: 'CASCADE'
            },
            {
              name: 'FK_chats_workers_worker_id_id',
              columnNames: ['worker_id'],
              referencedColumnNames: ['id'],
              referencedTableName: `workers`,
              onDelete: 'CASCADE'
            }
          ]
        })
      )
    })
  }

  public async down(queryRunner: QueryRunner): Promise<any> {
    console.log('THERE ARE NO ACTIONS NEED')
  }
}
```

Рисунок 3.5 – Міграція для створення таблиці chats

Наступним кроком буде створення моделі, та опис типів graphql. Для початку створюємо модель BaseEntity, рисунок 3.6. Вона допоможе уникнути

постійний копіпаст, та буде розташована в одному місці, що прискорить створення певних методів для всіх сутностей, та зменшить шанс допустити помилку.

```
export abstract class BaseEntity {
  @PrimaryGeneratedColumn( strategy: 'uuid')
  public id: string

  @CreateDateColumn( options: { name: 'created_at', type: 'timestamp with time zone' })
  public createdAt: Date

  @UpdateDateColumn( options: { name: 'updated_at', type: 'timestamp with time zone' })
  public updatedAt: Date

  /**
   * CONSTRUCTOR
   */

  protected constructor(params?: Partial<any>) {
    this.assignAttributes(params)
  }

  /**
   * METHODS
   */

  public assignAttributes<T>(attributes: Partial<T>): void {
    if (isUndefined(attributes)) {
      return
    }
    const entityAttributes = keys(getRepository(this.constructor.name).metadata.propertiesMap)

    entityAttributes.forEach((key: string) => {
      // @ts-ignore
      if (!isUndefined(attributes[key])) {
        // @ts-ignore
        this[key] = attributes[key]
      }
    })
  }

  public async validate(): Promise<boolean> {
    const errors = await validate(this)

    if (Boolean(errors.length)) {
      throw new ValidationError(getValidationErrorMessage(errors))
    }

    return true
  }
```

Рисунок 3.6 – Базова модель для всіх сутностей

Далі створюємо модель Chat, завдяки створеній раніше моделі(рисунок 3.6), ми уникаємо дуплікацій та створюємо модель швидше.

Описуємо всі поля, згідно нашої міграції з використанням BaseEntity, після чого додаємо відносини до вже створених моделей User та Worker, результат – рисунок 3.7.

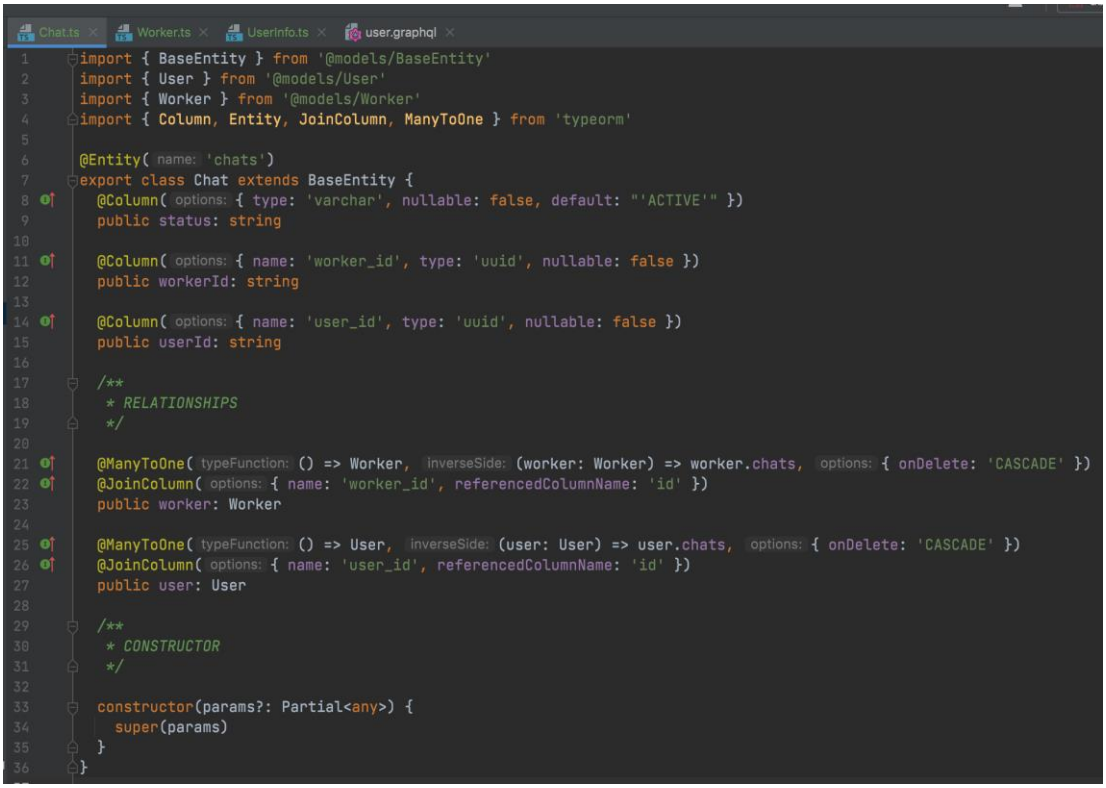

Рисунок 3.7 – Опис сутності Chat

3.3 Реалізація шифрування даних

Криптографія використовується для захисту даних, що зберігаються в базі даних або передаються через мережу додатку. Обробляючи, переміщуючи та зберігаючи дані, потрібно робити це безпечно та надійно. Шифрування та дешифрування спрямовані на підвищення безпеки.

Криптографія має вирішальне значення для розробки програмного забезпечення. Дані повинні бути захищені. Криптографія – це дослідження методів забезпечення безпеки даних. Він перетворює дані в секрет, перетворюючи відкритий текст у текст, який не можна читати, і навпаки. Тому лише відправник і одержувач цих даних можуть зрозуміти їх зміст.

Три основні компоненти криптосистеми включають відкритий текст, зашифрований текст і алгоритм. Щоб зробити інформацію зашифрованою, ми використовуємо шифр і алгоритм, який перетворює відкритий текст на шифрований. Перетворення даних у «незрозумілі символи» називається

шифруванням, а повернення їх у відкритий текст — дешифруванням. Криптографічні алгоритми використовують ключ для перетворення відкритого тексту в зашифрований. Перетворення зашифрованого тексту назад у відкритий текст можливе лише за наявності з собою потрібного ключа.

Щоб захистити дані в Node.js, потрібно зберігати хешовані паролі в базі даних. Таким чином, це не дасть перетворити дані в відкритий текст після їх хешування.Якщо зловмисники отримає доступ до бази даних, вони не читатимуть дані, оскільки вони зашифровані. Більше того, у них немає ключа, щоб допомогти їм це зробити.

Node.js має вбудований криптомодуль, який надає функції для виконання криптографічних операцій. Він включає в себе набір обгорток для функцій хешування OpenSSL, HMAC, шифрування, дешифрування, підпису та перевірки. Вся логіка буде реалізована за допомогою модуля crypto.

Під час шифрування даних важливо використовувати алгоритм. У цьому проекті використовується алгоритм aes-256-cbc.

Метод crypto.randomBytes() використовується для генерування криптографічно побудованих випадкових даних, згенерованих у написаному коді. InitVector (вектор ініціалізації) використовується тут для зберігання 16 байтів випадкових даних із методу randomBytes(), а ключ безпеки містить 32 байти випадкових даних. Та добавлено константу password, для того щоб показати як це працює.

Для шифрування даних використовується функція chiper. Функція шифрування нашого проекту створена за допомогою createCipheriv(), вектора ініціалізації з модуля crypto.

Перший аргумент функції це алгоритм, який ми використовуємо, другий аргумент це ключ безпеки і initVector -третій аргумент. Щоб зашифрувати повідомлення, потрібно використати метод update() у шифрі. Ця функція

приймає тест, який потрібно зашифрувати, utf-8 (тип кодування) і hex (кодування вихідних даних) як третій аргумент.

```
const crypto = require('crypto')

const algorithm = 'aes-256-cbc'

const initVector = crypto.randomBytes(16)

const password = 'IloveELIT1234'

const securitykey = crypto.randomBytes(32)

const cipher = crypto.createCipheriv(algorithm, securitykey, initVector)

let encryptedData = cipher.update(password, 'utf-8', 'hex')

encryptedData += cipher.final('hex')
```

Рисунок 3.8 – Стартові необхідні дані для реалізації шифрування

Отже, запускаємо наш сервер, і виконуємо тестовий запит, результат зображено на рисунку 3.9, повідомлення ‘IloveELIT1234’ – зашифроване в такий набір символів «a8a4cd82c64bb7b084a5e7f10d4d996b», а отже просто так уже його не розкодувати.

```
>
> console.log('Encrypted message: ', encryptedData)
Encrypted message:  a8a4cd82c64bb7b084a5e7f10d4d996b
undefined
```

Рисунок 3.9 – Зашифроване текстове повідомлення

Було використано алгоритм шифрування AES — це 256-бітний алгоритм Advanced Encryption Standard. Тут 256 біти вказують на довжину ключа.

```
const decipher = crypto.createDecipheriv(algorithm, securitykey, initVector)

let decryptedData = decipher.update(encryptedData, 'hex', 'utf-8')

decryptedData += decipher.final('utf8')

console.log('Decrypted message: ', decryptedData)
```

Рисунок 3.10 - Дешифратор

Рисунок 3.11 – Результат шифрування да дешифрування тексту

Для розшифровки даних ми використовуємо клас Decipher криптомодуля Node. Він реалізований подібно до того, як було реалізовано шифрування даних, рисунок 3.10 та 3.11 - результати. Щоб розшифрувати зашифрований текст, потрібно надіслати зашифроване повідомлення, передати вектор, який використовується для шифрування даних у функції шифрування та секретний ключ.

В додатках А-Е описано фронтенд частина для користувача, та оператора банка. А саме основні сторінки додатку.

Завдяки додатку Є, відбувається роутинг та відображення необхідної для користувача сторінки. В додатку Ж, розміщено допоміжні функції для калькуляції перерахувань та залишку коштів, пошуку користувачів.

В додатках З – Л наведені файли з бекенд частини. Додаток З описує в собі необхідні функції для шифрування та дешифрування даних. Всі інші додатки відносяться для завантаження файлу(файлів) на серверну частину, а також відображена їхня типизація.

3.4 Інтерфейс web-додатку

При створенні сайту банка, потрібно врахувати адаптивність сторінок, простоту для користувачів, та виявити зацікавленість.

Було створено дизайн за допомогою Figma. На рисунку 3.11 та 3.12 зображено основну частину додатку.

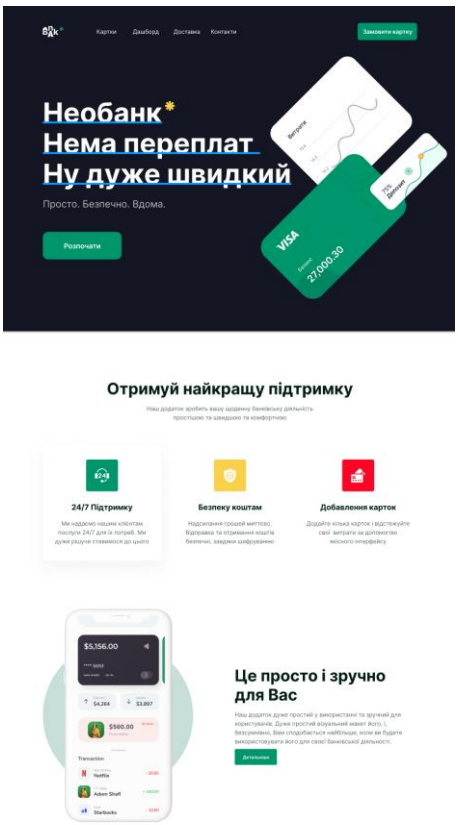

Рисунок 3.11 – Основна сторінка додатку, частина 1

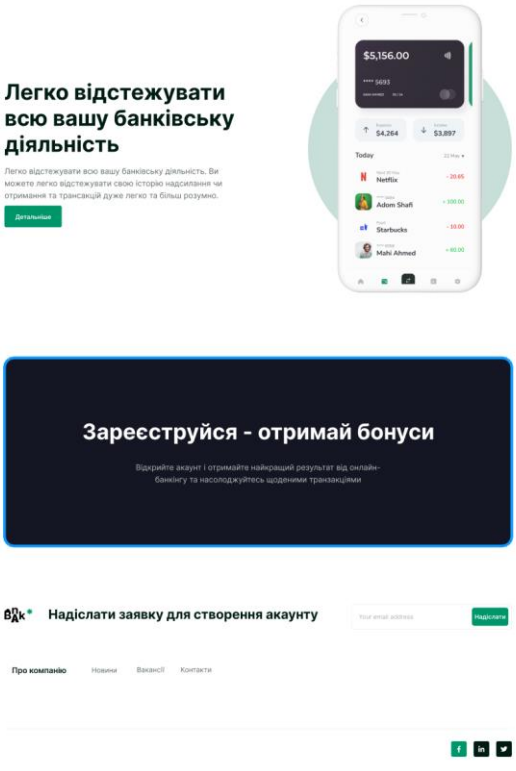

Рисунок 3.12 – Основна сторінка додатку, частина 2

Наступним було сплановано та виконано дизайн особистого кабінету для користувача та адміністратора. При вході на сторінку sign-in(рисунок 3.13), ми заповнюємо свої дані від акаунту, після чого переміщаємося на головну сторінку профіля - рисунок 3.14.

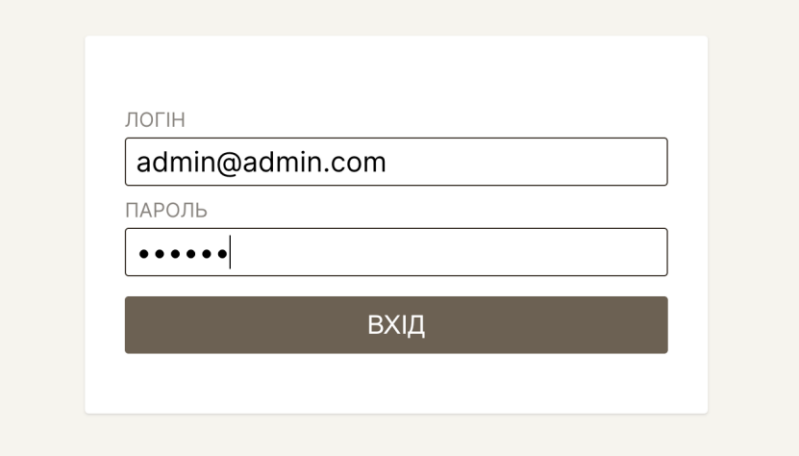

Рисунок 3.13 – Дизайн авторизації

Так виглядає основна сторінка, рисунок 3.14, для адміністратор банка, в ньому відображено залишки всіх коритсувачів. Також він зможе створювати нові облікові записи, виконувати переказ коштів як користувач банку, переводити на депозит, виводити кошти з рахунку, бачити історію своїх чатів.

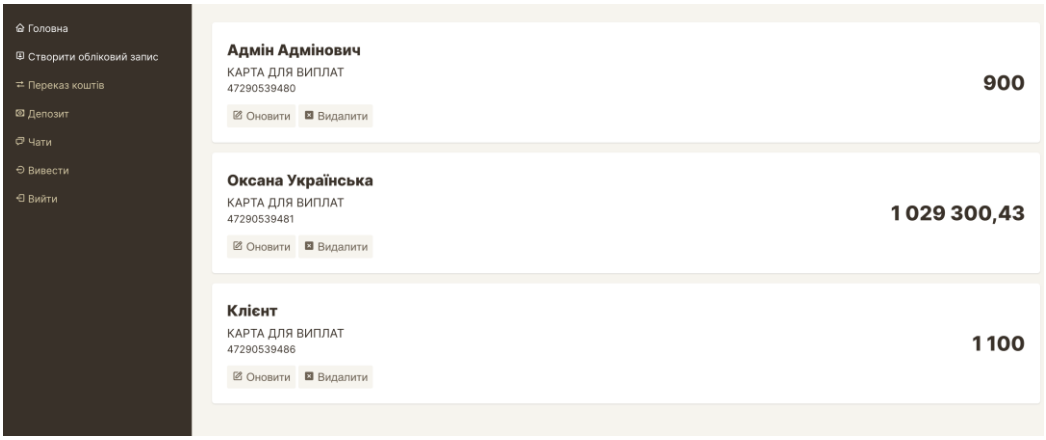

Рисунок 3.14 – Головна сторінка

Для створення облікового запису, рисунок 3.15, ми вказуємо такі поля, ПІП, номер карточки генерується автоматично, баланс, тип рахунку, пошта, ІПН та пароль.

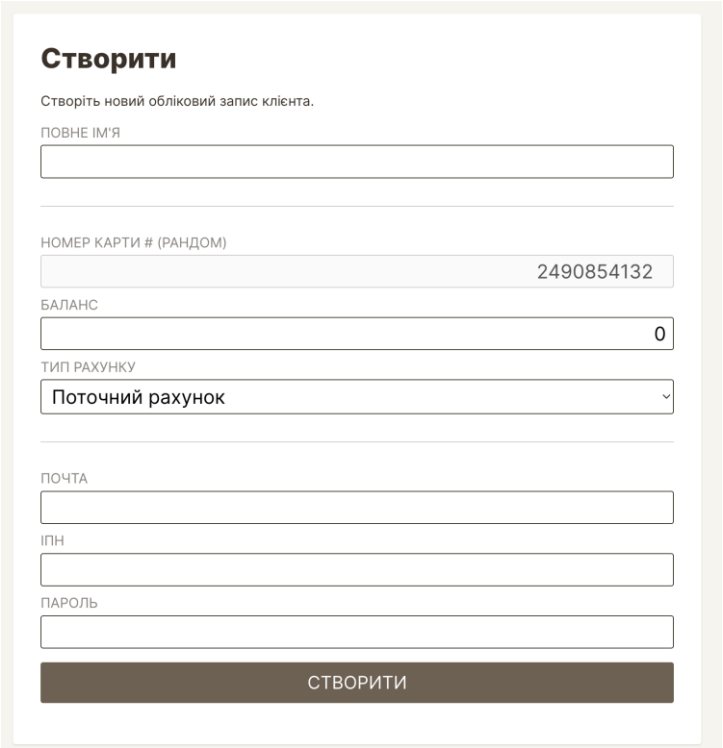

Рисунок 3.15 – Створення облікового запису

Переказ коштів матиме такі поля. Відправник, показує його баланс, він має змогу вказати суму яку хоче перевести на інший рахунок, та вказати отримувача, після чого натиснути кнопку «Надіслати».

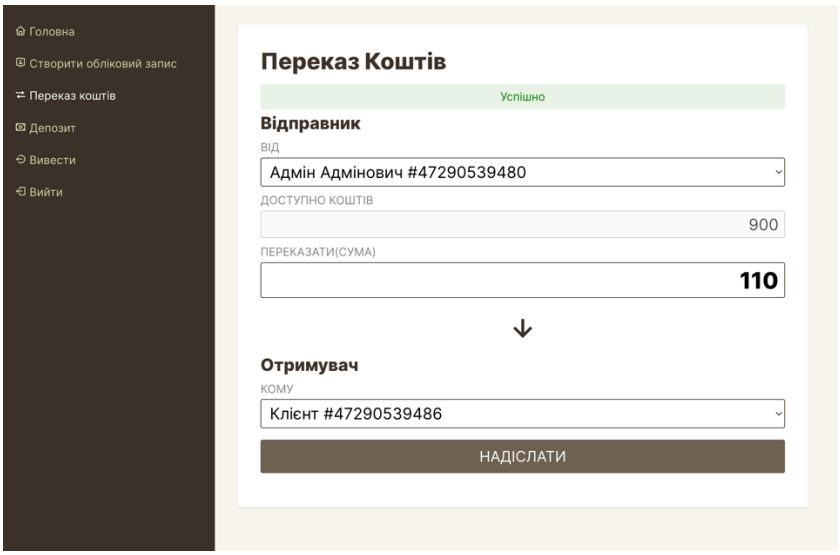

Рисунок 3.16 – Переказ коштів

На рисунку 3.17 відображено сторінку чатів, на ній зберігається історія листування між оператором та користувачами, ім’я(з ким листується оператор),

відображається останнє повідомлення, дата, статус чату, а також внизу можливість створити новий чат.

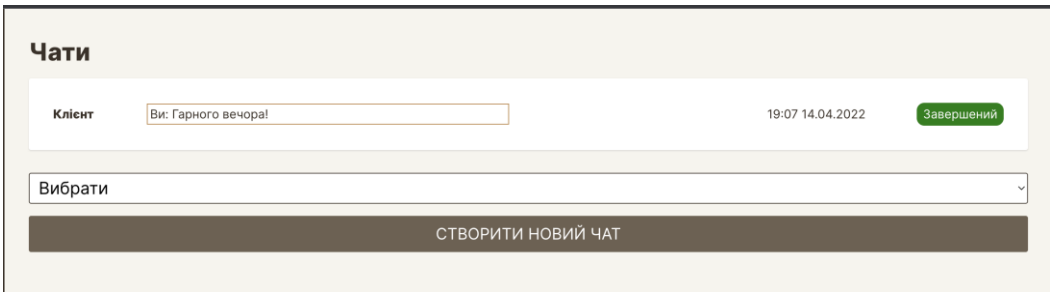

Рисунок 3.17 – Сторінка чатів

3.5 Розгортання web-додатку

Розгортання додатку проходить через український сервіс GitLab, з використанням Continuous integration (або скорочено CI), для того щоб бути впевненим, перед розгортанням, що проект не містить критичних помилок. GitLab — сайт та система керування репозиторіями програмного коду для Git, з додаткових можливостей - система відстеження помилок. Додаток розвернуто за допомогою контейнерів в Docker, онлайн ми це можемо виконати завдяки потужним AWS сервісам

Для збереження файлів локально використовується Minio. Він створює віртуальний сервер, в якому зберігаються всі файли з чатів, а також фото користувачів, працівників. Для збереження онлайн буде використано сервіс Digital Ocean.

ВИСНОВКИ

Успішно проаналізовано потребу в шифруванні даних, створено макет та реалізовано frontend частину. Розроблена база даних для простої банкової системи, з можливістю листування, та створення банку, фактично без відділень.

Було проаналізовано та вивчено асиметричне та симетричне шифрування, головна відмінність, це те що при симетричному шифруванні використовується один ключ, або один легко обчислюється з іншого та навпаки, на відміну від асиметричного, де ключ дешифрування важко обчислити. У ході розробки використовувались сучасні технології та методи проектування додатку. Завдяки сучасним технологіям, програмний додаток буде простіше підтримувати.

Розглянуто декілька підходів до розміщення файлів в проекті. Завдяки цьому ми маємо переваги в подальшій розробці, або модернізації проекту. Актуальність роботі заключається в розробці шифру на основі Node.js та crypto, для збереження даних та їхної приватності, тобто якщо шахраї заволодіють даними, вони не зможуть їх розшифрувати швидко.

СПИСОК ЛІТЕРАТУРИ

1. Regina O. PostgreSQL: Up and Running: A Practical Guide to the Advanced Open Source Database. - O'Reilly Media, Third edition, 2017. – 422с.
2. Hans-Jürgen Schönig. Mastering PostgreSQL 12: Advanced techniques to build and administer scalable and reliable PostgreSQL database applications - Packt Publishing, Third edition, 2019. – 726с.
3. Vallarapu Naga Avinash Kumar. PostgreSQL 13 Cookbook: Over 120 recipes to build high-performance and fault-tolerant PostgreSQL database solutions - Packt Publishing, First edition, 2021. – 344с.
4. Офіційний опис та документація PosgreSQL - https://www.postgresql.org/
5. Stefan Rosca. WebStorm Essentials - United Kingdom:Packt Publishing Limited, 2015 - 184с.
6. Офіційний опис та документація WebStorm - https://www.jetbrains.com/ru-ru/webstorm/
7. Офіційний опис та документація DataGrip - https://www.jetbrains.com/ru-ru/datagrip/
8. Девід Фленаган. JavaScript. Повне керівництво, 7-ме видання – Київ: Діалектика-Вільямс, 2021. – 722с.
9. Марейн Хавербеке. Виразний JavaScript - Київ: Діалектика-Вільямс, 3-тє видання, 2019. – 480с
10. Mario Casciaro, Luciano Mammino. Node.js Design Patterns: Design and implement production-grade Node.js applications using proven patterns and techniques. - Packt Publishin, 2020. – 660с.
11. Mark Tielens Thomas. React in Action. - Manning, 2018. – 360с.
12. Янг А., Мек Б., Кантелон М. Node.js у дії.- СПб.: Питер, 2018. – 432с.
13. Офіційний опис та документація Docker - https://www.docker.com/
14. Офіційний опис та документація React - https://ru.reactjs.org/
15. Офіційний опис та документація Node.js та модуль шифрування Crypto - https://nodejs.org/uk/

16. Senker C. Cybercrime and the Darknet: Revealing the hidden underworld of the internet. - Arcturus, 2016. – 194c.
17. Holden J. The Mathematics of Secrets: Cryptography from Caesar Ciphers to Digital Encryption. - Princeton University Press, 2017. – 392c.
18. Michael T., Hosmer C. Data Hiding: Exposing Concealed Data in Multimedia, Operating Systems, Mobile Devices and Network Protocols. - Syngress, 2012. – 350c.
19. Niels Ferguson, Bruce Schneier, Tadayoshi Kohno. Cryptography Engineering: Design Principles and Practical Applications. - Wiley, 2010. – 384c.
20. Шнайер Б. Прикладная криптография. Протоколы, алгоритмы. – М.:Диалектика, 2017. – 1040с.
21. Андресс Д. Захист даних. Від авторизації до аудиту.- СПб.: Питер, 2021. – 435с.
22. Серра Е. Кібербезпека: правила гри. Як керівники та співробітники впливають на культуру безпеки в компанії. Інтелектуальна література. А.:Паблишер, 2021. – 192с.

ДОДАТКИ

Додаток А

```
import React from "react";
import { ActionButtons } from "./ActionButtons";
import { formatNumber } from "./Utils";

export const Account = (props) => {

    const {type, accountNumber, balance, fullname, editingUser, setEditingUser, setDeleteUser,
index, isAdmin, setEditModal} = props;

    const action = isAdmin ? <ActionButtons index={index}
      editingUser={editingUser}
      setEditingUser={setEditingUser}
      setEditModal={setEditModal} setDeleteUser={setDeleteUser} /> : '';

    return (
      <div className="account">
          <div className="details">
              <AccountHolder fullname={fullname} />
              <AccountType type={type} />
              <AccountNumber accountNumber={accountNumber} />
              {action}
          </div>
          <AccountBalance balance={formatNumber(balance)} />
      </div>
    )
  }

export const AccountHolder = (props) => {
    return (
      <h1>{props.fullname}</h1>
    )
  }


export const AccountType = (props) => {
    return (
      <h3>{props.type}</h3>
    )
  }


export const AccountNumber = (props) => {
    return (
      <div>{props.accountNumber}</div>
    )
  }


export const AccountBalance = (props) => {
    const balance = props.balance;
    return (
      <div className="balance">{balance}</div>
    )
  }
```

Додаток Б

```
import React from "react";

export const ActionButtons = (props) => {
    const {editingUser, setEditingUser, index, setEditModal, setDeleteUser} = props;

    return (
      <div id="actions">
        <ActionButton
          icon="bx bx-edit"
          text="Оновити"
          index={index}
          actionType="edit"
```

```
          editingUser={editingUser}
          actionType='edit'
          setEditingUser={setEditingUser} setEditModal={setEditModal} />

        <ActionButton
          icon="bx bxs-x-square"
          index={index}
          actionType='delete'
          text="Видалити" editingUser={editingUser}
          setDeleteUser={setDeleteUser} />
      </div>
    )
  }

export const ActionButton = (props) => {
    const {icon, text, editingUser, actionType, setEditingUser, index, setEditModal, setDeleteUser}
= props;

    const click = (e, index) => {
      e.preventDefault();

      if(actionType === 'edit') {
        setEditingUser(index);
        setEditModal(true);
      }

      if(actionType === 'delete') {
        setDeleteUser(index);
      }
    }

    return (
      <button onClick={(e) => click(e, index)}><i className={icon} ></i> {text}</button>
    )
  }
```

Додаток В

```
import React, {useState} from 'react';
import { Sidebar } from './Sidebar';
import { MainClientContent } from './MainClientContent';
import { findAccount } from './Utils';
import { TransferPage } from './TransferPage';
import { BudgetApp } from './BudgetApp';
import { Chats } from './Chats';

export const ClientDashboard = (props) => {
    const { logout, client, setClient } = props;
    const [users, setUsers] = useState(props.users);
    const [ page, setPage ] = useState('home');


    const changePageHandler = (pageName) => {
      setPage(pageName);
      const currentUser = findAccount(client.number);
      setClient(currentUser);
    }

    if(page === 'home') {

      return (
        <main>
          <Sidebar changePage={changePageHandler} page={page} user={client}
logoutHandler={props.logout} />
          <MainClientContent user={client} />
        </main>
      )
    }

    if(page === 'budget') {
      return (
        <main>
          <Sidebar changePage={changePageHandler} page={page} user={client}
logoutHandler={props.logout} />
          <BudgetApp client={client} />
        </main>
```

```
      )
    }

    if(page === 'transfer') {
      return (
        <main>
          <Sidebar changePage={changePageHandler} page={page} user={client}
logoutHandler={props.logout} />
          <TransferPage isClient="true" client={client} setClient={setClient} users={users}
setUsers={setUsers}  />
        </main>
      )
    }

    if(page === 'chats') {
      return (
          <main>
              <Sidebar changePage={changePageHandler} page={page} user={client}
logoutHandler={props.logout} />
              <Chats user={client} />
          </main>
      )
    }
}
```

Додаток Г

```
import { useState } from "react";
import { Notif } from "./Notif";
import {formatNumber, trim} from './Utils';

export const CreateAccountPage = (props) => {
    const createRandomAccount = () => {
        return Math.floor(1000000000 + Math.random() * 9000000000);
    }

    const [notif, setNotif] = useState({message: 'Створіть новий обліковий запис клієнта.', style:
'left'});
    const [initialBalance, setInitialBalance] = useState(0);
    const [initialAccountNumber, setInitialAccountNumber] = useState(createRandomAccount());

    const createNewAccount = (user) => {

        const emptyInputs = Object.values(user).filter(input => {
            return input === ''
        });

        const localUsers = props.users;

        let alreadyExists = false;
        localUsers.forEach(row => {
            if(row.email === user.email) {
                alreadyExists = true;
            }
        });

        if(alreadyExists) {
            setNotif({message: 'Ця адреса електронної вже існує. Спробуйте ще раз.', style:
'danger'});
            return false;
        } else if(emptyInputs.length > 0) {
            setNotif({message: "Всі поля обов'язкові для заповнення.", style: 'danger'});
            return false;
        } else {
            setNotif('');
            localUsers.unshift(user);
            props.setUsers(localUsers);
            localStorage.setItem('users', JSON.stringify(localUsers));
            setNotif({message: 'Збережено', style: 'success'});
            return true;
        }
    }

    const handleCreateAccount = (event) => {
        event.preventDefault();
        const user = event.target.elements;
```

```
        const account = {
            email: user.email.value,
            password: user.password.value,
            fullname: user.fullname.value,
            type: user.accountType.value,
            number: user.accountNumber.value,
            isAdmin: false,
            balance: trim(user.initialBalance.value),
            transactions: []
        }

        const isSaved = createNewAccount(account);
        if(isSaved) {
            user.email.value = '';
            user.password.value = '';
            user.fullname.value = '';
            user.accountNumber.value = setInitialAccountNumber(createRandomAccount());
            user.initialBalance.value = setInitialBalance(0);
        }
    }

    const onInitialBalance = event => {
        const amount = trim(event.target.value) || 0;
        setInitialBalance(amount);
    }

    return (
        <section id="main-content">
            <form id="form" onSubmit={handleCreateAccount}>
                <h1>Створити</h1>
                <Notif message={notif.message} style={notif.style} />
                <label htmlFor="fullname">Повне ім'я</label>
                <input id="fullname" type="text" autoComplete="off" name="fullname" />
                <hr />
                <label htmlFor="account-number">Номер карти # (рандом)</label>
                <input id="account-number" name="accountNumber" className="right"
value={initialAccountNumber} type="number" disabled />

                <label htmlFor="balance">Баланс</label>
                <input id="balance" type="text" value={formatNumber(initialBalance)}
onChange={onInitialBalance} name="initialBalance" className="right" />

                <label htmlFor="account-type">Тип рахунку</label>
                <select name="accountType">
                    <option value="Checking Account">Поточний рахунок</option>
                    <option value="Savings Accounts">Депозит</option>
                </select>
                <hr />
                <label htmlFor="email">Почта</label>
                <input id="email" type="email" name="email" />
                <label htmlFor="email">ІПН</label>
                <input id="email" type="text" name="ipn" />
                <label htmlFor="password">Пароль</label>
                <input id="password" type="password" name="password" />
                <input value="Створити" className="btn" type="submit" />
            </form>
        </section>
    )
}
```

Додаток Ґ

```
import React, { useState } from 'react';
import { Notif } from './Notif';

export const LoginPage = (props) => {
    const [username, setUsername] = useState('');
    const [password, setPassword] = useState('');

    const onSubmitHandler = (event) => {
      event.preventDefault();
      props.loginHandler(username, password);
    }

    const onChangeUsername = (event) => {
```

```
      setUsername(event.target.value);
    }

    const onChangePassword = (event) => {
      setPassword(event.target.value);
    }

    return (
      <div id="login-page">
        <div id="login">
          <Notif message={props.notif.message} style={props.notif.style} />
          <form onSubmit={onSubmitHandler}>
            <label htmlFor="username">Логін</label>
            <input id="username" autoComplete="off" onChange={onChangeUsername}  value={username}
type="text" />
            <label htmlFor="password">Пароль</label>
            <input id="password" autoComplete="off" onChange={onChangePassword} value={password}
type="password" />
            <button type="submit" className="btn">Вхід</button>
          </form>
        </div>
      </div>
    )
}
```

Додаток Д

```
import { Account } from './Account';
import { formatNumber } from './Utils';

export const MainClientContent = props => {
    const {user} = props;

    const transactions = user.transactions.map((transaction, index) => {
      const className = index % 2 === 0 ? 'even' : 'odd'
      return <div className={`transaction-item ${className}`}>
        <div>{transaction.date}</div>
        <div>{transaction.title}</div>
        <div>{transaction.type === 'debit' ? formatNumber(transaction.amount * -1) :
formatNumber(transaction.amount)}</div>
      </div>
    });


    return (
      <section id="main-content">
        <h1 className="main">Мій аккаунт</h1>
        <Account type={user.type} accountNumber={user.number} balance={user.balance}
fullname={user.fullname} />
        <div id="transactions">
          <h2>Транзакції</h2>
          <div id="transaction-div">
          {transactions}
          </div>
        </div>
      </section>
    )
}
```

Додаток Е

```
import { Account } from "./Account";
import React, { useEffect, useState } from "react";

export const MainContent = (props) => {
    const users = props.users;
    const {editingUser, setEditingUser, setEditModal, setDeleteUser} = props;
    const [isCurrentUserAdmin, setIsCurrentUserAdmin] = useState(false);

    useEffect(() => {
      const localUser = JSON.parse(localStorage.getItem('currentUser'));
      setIsCurrentUserAdmin(localUser.isAdmin);
    }, [isCurrentUserAdmin]);
```

```
    const bankAccounts = users.map((user, index) => {
      return <Account key={index} index={index} fullname={user.fullname}
        type={user.type}
        isAdmin={isCurrentUserAdmin}
        accountNumber={user.number}
        balance={user.balance}
        editingUser={editingUser}
        setEditingUser={setEditingUser} setEditModal={setEditModal}
        setDeleteUser={setDeleteUser} />
    });

    return (
      <section id="main-content">
        {bankAccounts}
      </section>
    )

  }
```

Додаток Є

```
import React from 'react';

export const Sidebar = (props) => {
    const { user, logoutHandler, changePage, page } = props;
    let menu = null;

    if(!user) {
        menu = <SideMenu changePage={changePage} page={page} logoutHandler={logoutHandler} />;
    }

    if(user) {
        menu = <ClientMenu changePage={changePage} page={page} logoutHandler={logoutHandler} />
    }

    return(
        <section id="side-menu">
            {menu}
        </section>
    )
}

export const ClientMenu = (props) => {
    const {changePage, logoutHandler, page} = props;

    return (
        <ul>
            <SideLink onClickHandler={changePage} active={page} page="home" icon="bx bx-home"
text="Головна" />
            <SideLink onClickHandler={changePage} active={page} page="budget" icon="bx bx-money"
text="Бюджети" />
            <SideLink onClickHandler={changePage} active={page} page="chats" icon="bx bx-money"
text="Чати" />
            <SideLink onClickHandler={changePage} active={page} page="transfer" icon="bx bx-
transfer" text="Переказ коштів" />
            <SideLink onClickHandler={logoutHandler} active={page} icon="bx bx-log-out" text="Вийти"
/>

        </ul>
    )
}

export const SideMenu = (props) => {
    const {changePage, logoutHandler, page} = props;
    return (
        <ul>
            <SideLink onClickHandler={changePage} active={page} page="home" icon="bx bx-home"
text="Головна" />
            <SideLink onClickHandler={changePage} active={page} page="create-account" icon="bx bx-
user-pin" text="Створити обліковий запис" />
            <SideLink onClickHandler={changePage} active={page} page="transfer" icon="bx bx-
transfer" text="Переказ коштів" />
            <SideLink onClickHandler={changePage} active={page} page="deposit" icon="bx bx-money"
text="Депозит" />
            <SideLink onClickHandler={changePage} active={page} page="chats" icon="bx bx-chat"
```

```
text="Чати" />
            <SideLink onClickHandler={changePage} active={page} page="withdraw" icon="bx bx-log-out-
circle" text="Вивести" />
            <SideLink onClickHandler={logoutHandler} active={page} icon="bx bx-log-out" text="Вийти"
/>
        </ul>
    )
}

export const SideLink = (props) => {
    const {icon, text, page, active} = props;

    function clickLink(event) {
        if(page) {
            event.preventDefault();
            props.onClickHandler(page);
        } else {
            event.preventDefault();
            props.onClickHandler();
        }
    }

    return (
        <li><a onClick={clickLink} className={ active === page ? 'active' : '' } href="#"><i
className={icon} ></i> {text}</a></li>
    )
}
```

Додаток Ж

```
export function formatNumber(number)
{
    return number.toLocaleString(undefined, {maximumFractionDigits: 2});
}

export function trim(number) {
    return parseFloat(number.replace(/,/g, '')) || 0;
}

export function findAccount(number) {
    const users = JSON.parse(localStorage.getItem('users'));

    for(const user of users) {
        if(user.number === number) {
            return user;
        }
    }

    return false;
}

export function transact(number, amount, type, setUsers=null)
{
    let multiplier = 1;
    if(type === 'add' || type === 'credit') multiplier = 1;
    if(type === 'subtract' || type === 'debit') multiplier = -1;

    const users = JSON.parse(localStorage.getItem('users'));

    for(const user of users) {
        if(user.number === number) {
            user.balance += amount * multiplier;

            if(type === 'add' || type === 'credit') {
                user.transactions.unshift({
                    title: `Депозит`,
                    amount: amount,
                    type: "credit",
                    date: getDateToday()
                })
            }

            if(type === 'subtract' || type === 'debit') {
                user.transactions.unshift({
                    title: `Платіж`,
                    amount: amount,
```

```
                    type: "debit",
                    date: getDateToday()
                })
            }
        }
    }
    setUsers(users);
    localStorage.setItem('users', JSON.stringify(users));
}

export function capitalize(str)
{
    return str.charAt(0).toUpperCase() + str.slice(1);
}

export function saveBudgetToDB(accountNumber, newBudget)
{
    const user = findAccount(accountNumber);
    user.budget = newBudget;
    const filteredUsers = addUserToUsers(user);
    localStorage.setItem('users', JSON.stringify(filteredUsers));
}



function addUserToUsers(user) {
    const users = JSON.parse(localStorage.getItem('users'));

    const filteredUsers = users.filter(dbUser => {
        return dbUser.number !== user.number;
    });

    filteredUsers.push(user);
    return filteredUsers;
}

export function getDateToday() {
    const transDate = new Date();
    return `${transDate.toLocaleString("en-us", { month: "long" })} ${transDate.getDay()},
${transDate.getFullYear()}`;
}
```

Додаток З

```
const crypto = require('crypto');

const ENCRYPTION_KEY = process.env.ENCRYPTION_KEY;
const IV_LENGTH = 16;

function encrypt(text) {
    let iv = crypto.randomBytes(IV_LENGTH);
    let cipher = crypto.createCipheriv('aes-256-cbc', Buffer.from(ENCRYPTION_KEY), iv);
    let encrypted = cipher.update(text);

    encrypted = Buffer.concat([encrypted, cipher.final()]);

    return iv.toString('hex') + ':' + encrypted.toString('hex');
}

function decrypt(text) {
    let textParts = text.split(':');
    let iv = Buffer.from(textParts.shift(), 'hex');
    let encryptedText = Buffer.from(textParts.join(':'), 'hex');
    let decipher = crypto.createDecipheriv('aes-256-cbc', Buffer.from(ENCRYPTION_KEY), iv);
    let decrypted = decipher.update(encryptedText);

    decrypted = Buffer.concat([decrypted, decipher.final()]);

    return decrypted.toString();
}

module.exports = { decrypt, encrypt };
```

Додаток И

```
import { assetDelete } from '@graphql/assets/mutationResolvers/assetDelete'
import { getUrls } from '@graphql/assets/mutationResolvers/getUrls'
import { IResolvers } from 'graphql-tools'

export const resolvers: IResolvers = {
  Mutation: {
    getUrls,
    assetDelete
  }
}
```

Додаток І

```
type Asset {
  id: ID
  name: String
  size: Int
  contentType: String
  baseName: String
  url: String
  createdAt: DateString
  updatedAt: DateString
}

type AssetsResponse implements IResponse {
  ok: Boolean!
  message: String
  data: SignedUrls
}

type AssetsData {
  count: Int
  data: [Asset]
}

type SignedUrls {
  downloadUrl: String
  uploadUrl: String
  baseName: String
}

input FileInput {
  contentType: String!
  baseName: String
  assetName: String!
  isRemove: Boolean
}

input AssetCreateInput {
  name: String!
  size: Int!
  baseName: String
  contentType: String!
  url: String!
}

input AssetUpdateOrCreateInput {
  id: ID
  name: String!
  size: Int!
  baseName: String
  contentType: String!
  url: String!
}

extend type Mutation {
  getUrls(file: FileInput!): AssetsResponse!
  assetDelete(id: ID!): AssetsResponse!
}
```

Додаток Й

```
import { Asset } from '@models/Asset'
import {  IAssetCreateInput } from '@src/types/generated'
import { Connection, EntityManager } from 'typeorm'

export async function assetCreate(
  transaction: EntityManager | Connection,
  assetInput: IAssetCreateInput & { parent: any }
): Promise<Asset> {

  const asset = new Asset({assetInput })

  const assetRepository = transaction.getRepository(Asset)

  return assetRepository.save(asset)
}
```

Додаток Ї

```
export async function assets(args: any): Promise<any> {
  const ids = args.map((entity: any) => entity.parent.id)
  const [{ parent, ctx }] = args
  const model = parent.constructor.name

  const query = ctx.connection
    .getRepository(model)
    .createQueryBuilder(model)
    .leftJoinAndSelect(`${model}.assets`, 'attachment')
    .where(`${model}.id IN (:...ids)`, { ids })

  const [data, count] = await Promise.all([
    query
      .clone()
      .select([`${model}.id`, 'attachment'])
      .getMany(),
    query
      .clone()
      .select([`${model}.id AS id`, 'count(attachment.id)'])
      .groupBy(`${model}.id`)
      .getRawMany()
  ])

  return ids.map((id: string) => ({
    data: data.find((entity: any) => entity.id === id).attachments,
    count: count.find((entity: any) => entity.id === id).count
  }))
}
```

Додаток К

```
import { ContextType } from '@graphql/context'
import { StorageService } from '@services/StorageService'
import { IAssetsResponse, IGetUrlsMutationArgs } from '@src/types/generated'

export async function getUrls(_: any, args: IGetUrlsMutationArgs, { schema }: ContextType):
Promise<IAssetsResponse> {
  const storage = new StorageService({ schema, ...args.file })

  return {
    ok: true,
    data: {
      uploadUrl: await storage.getUploadSignedUrl(),
      downloadUrl: await storage.getDownloadSignedUrl(),
      baseName: await storage.getBaseName()
    }
  }
}
```

Додаток Л

```
import { ContextType } from '@graphql/context'
import { StorageService } from '@services/StorageService'
import { Asset } from '@models/Asset'

export async function assetDelete(
  _: any,
  args: any,
  { schema, connection }: ContextType
): Promise<any> {
  const { assetType, contentType, name, baseName } = await
connection.getRepository(Asset).findOneOrFail(args.id)

  const storage = new StorageService({ schema, assetType, contentType, assetName: name, baseName })

  await storage.removeFile()

  await connection.getRepository(Asset).delete(args.id)

  return { ok: true }
}
```