ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Мобільний додаток для онлайн-банкінгу з інтеграцією серверної частини на базі Windows Servers»

на здобуття освітнього ступеня бакалавра

зі спеціальності 123 – Комп'ютерна інженерія
(код, найменування спеціальності)

освітньо-професійної програми Комп'ютерна інженерія
(назва)

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

_____ *Джух Антон*
(підпис) *Ім'я, ПРІЗВИЩЕ здобувача*

Виконав: здобувач вищої освіти гр.КІД-41
Антон Джух
Ім'я, ПРІЗВИЩЕ

Керівник: Олександр Зуб
науковий ступінь, вчене звання Ім'я, ПРІЗВИЩЕ

Рецензент: _____
науковий ступінь, вчене звання Ім'я, ПРІЗВИЩЕ

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
Навчально-науковий інститут Інформаційних технологій

Кафедра Комп'ютерної інженерії

Ступінь вищої освіти бакалавр

Спеціальність 123 Комп'ютерна інженерія

Освітньо-професійна програма Комп'ютерна інженерія

ЗАТВЕРДЖУЮ

Завідувач кафедрою

Комп'ютерної інженерії

_____Наталія ЛАЩЕВСЬКА

_____

«____» _____2025р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Джух Антон Володимирович

(прізвище, ім'я, по батькові здобувача)

1. Тема кваліфікаційної роботи: Мобільний додаток для онлайн-банкінгу з інтеграцією серверної частини на базі Windows Servers

керівник кваліфікаційної роботи Олександр ЗУБ,

(Ім'я, ПРІЗВИЩЕ, науковий ступінь, вчене звання)

затверджені наказом Державного університету інформаційно-комунікаційних технологій від «24» лютого 2025р. № 56

2. Строк подання кваліфікаційної роботи «5» червня 2025р.
3. Вихідні дані до кваліфікаційної роботи: (Вихідні дані до кваліфікаційної роботи: науково-технічна література; мови програмування: C#, Dart; технології: ASP.NET Core 6.0, Entity Framework Core, BCrypt.Net-Next, JWT, Flutter; база даних: Microsoft SQL Server, SQL; протоколи: HTTP/HTTPS, TCP/IP, TLS/SSL; архітектура: REST, клієнт-сервер, MVC; інструменти: Visual Studio, Visual Studio Code, SSMS, Postman, Swagger, Git, IIS; топологія: логічна (Layer 7), фізична (сервер з IIS та SQL Server).)
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Архітектури та технологічні засади створення сучасного мобільного додатку онлайн банкінгу
2. Аналіз вимог і вибір технологій для розробки додатку з серверною частиною
3. Проектування та реалізація проекту

5. Перелік ілюстративного матеріалу: *презентація*
6. Дата видачі завдання «24» лютого 2025р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір джерел інформації	20.02-28.02.2025	Виконано
2	Постановка задачі	01.03-09.03.2025	Виконано
3	Аналіз	10.03-14.03.2025	Виконано
4	Розробка проекту	15.03-20.04.2025	Виконано
5	Написання роботи	21.04-06.05.2025	Виконано
6	Висновки	07.05-08.05.2025	Виконано
7	Підготовка демонстраційного матеріалу	09.05-11.05.2025	Виконано
8	Попередній захист роботи	27.05-29.05.2025	Виконано
9	Пред’явлення роботи в деканат	02.06-05.06.2025	Виконано

Здобувач(ка) вищої освіти _____ Антон ДЖУХ
(підпис) *(Ім’я, ПРІЗВИЩЕ)*

Керівник
кваліфікаційної роботи _____ Олександр ЗУБ

(підпис) *(Ім’я, ПРІЗВИЩЕ)*

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 78с., 1 табл., 6 рис., 30 джерел.

Мета роботи – розробка мобільного додатку з інтеграцією серверної частини.

Об'єкт дослідження – мобільний додаток та серверна частина.

Предмет дослідження – мобільний додаток з серверною частиною.

Мета роботи – розробити мобільний додаток для онлайн-банкінгу з інтеграцією серверної частини на базі windows servers.

Метою роботи є розробка мобільного додатку з інтеграцією серверної частини для забезпечення функціональності онлайн-банкінгу. Визначено особливості архітектури мобільного додатку та серверної частини на базі Windows Servers. Здійснено оцінку рівня безпеки, продуктивності та зручності використання розробленого мобільного додатку. На основі результатів виконаних розробок зроблено висновок, що впровадження мобільного додатку для онлайн-банкінгу з інтегрованою серверною частиною є перспективним для підвищення доступності та ефективності банківських послуг.

Упровадження розробленого додатку дозволить оптимізувати процеси онлайн-банкінгу, забезпечивши користувачів зручним та безпечним доступом до фінансових операцій.

КЛЮЧОВІ СЛОВА: МОБІЛЬНИЙ БАНКІНГ, ОНЛАЙН БАНКІНГ, WINDOWS СЕРВЕР, БАЗА ДАНИХ, СЕРВЕРНА ЧАСТИНА, МОБІЛЬНИЙ ДОДАТОК.

ЗМІСТ

ВСТУП ... 8
1 АРХІТЕКТУРИ ТА ТЕХНОЛОГІЧНІ ЗАСАДИ СТВОРЕННЯ СУЧАСНОГО МОБІЛЬНОГО ДОДАТКУ ОНЛАЙН-БАНКІНГУ ... 9
1.1 Огляд сучасних технологій для онлайн-банкінгу ... 15
1.1.1 Мобільні платформи та Flutter ... 18
1.1.2 Серверні технології на базі Windows Servers ... 19
1.2 Теоретичні основи серверної інфраструктури ... 20
1.2.1 Архітектура клієнт-сервер ... 24
1.2.2 Безпека серверної частини ... 26
1.3 Методологія розробки ... 27
2 АНАЛІЗ ВИМОГ І ВИБІР ТЕХНОЛОГІЙ ДЛЯ РОЗРОБКИ ДОДАТКУ З СЕРВЕРНОЮ ЧАСТИНОЮ ... 30
2.1 Аналіз вимог до системи онлайн-банкінгу ... 35
2.1.1 Функціональні вимоги ... 37
2.1.2 Нефункціональні вимоги ... 38
2.2 Дослідження існуючих рішень ... 40
2.2.1 Аналіз аналогів мобільних додатків ... 40
2.2.2 Серверні рішення для банкінгу ... 42
2.3 Проектування серверної інфраструктури ... 43
2.3.1 Моделювання бази даних ... 43
2.3.2 API для інтеграції ... 45
3 ПРОЕКТУВАННЯ ТА РЕАЛІЗАЦІЯ ПРОЕКТУ ... 48
3.1 Розробка серверної частини ... 49
3.1.1 Налаштування Windows Servers ... 50
3.1.2 Реалізація API та бізнес-логіки ... 51
3.1.3 Забезпечення безпеки серверної частини ... 54
3.2.1 Інтерфейс користувача ... 57
3.2.2 Інтеграція з серверною частиною ... 59
3.3 Тестування та оптимізація ... 63
3.3.1 Тестування серверної частини ... 64
3.3.2 Тестування мобільного додатку ... 67
ВИСНОВКИ ... 73
ПЕРЕЛІК ПОСИЛАНЬ ... 76
ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ ... 78
ЛІСТИНГИ ОСНОВНИХ МОДУЛІВ ... 86

ВСТУП

У сучасному світі основна задача людей – ефективно управляти своїми фінансами в умовах швидкого ритму життя. Через брак часу та складність традиційних банківських процедур ми шукаємо зручні та безпечні способи доступу до фінансових послуг. Технології стали невід'ємною частиною нашого життя, і якщо використовувати їх правильно, вони значно полегшують повсякденні задачі. Це стосується й розробки мобільних додатків для онлайн-банкінгу, які, завдяки інтеграції з потужними серверними системами, дозволяють забезпечити швидкий, безпечний і зручний доступ до фінансових операцій.

Сьогодні люди прагнуть мати постійний контроль над своїми фінансами, уникаючи ризиків, пов'язаних із безпекою даних чи технічними збоями. Для цього розробляються проекти, що поєднують мобільні додатки з надійними серверними інфраструктурами, такими як Windows Servers, які забезпечують обробку транзакцій, захист даних і стабільність роботи системи.

Мобільний додаток для онлайн-банкінгу є важливим інструментом, що дозволяє користувачам здійснювати платежі, перекази, перевіряти баланс і переглядати історію транзакцій у реальному часі. Серверна частина, побудована на базі Windows Servers, відіграє ключову роль у забезпеченні безпеки, масштабованості та швидкості обробки даних. Вона включає реалізацію REST API, управління базами даних і захист від кіберзагроз, що є критично важливим для фінансових систем. Такі рішення широко застосовуються в банківській сфері для забезпечення зручного доступу до послуг, підвищення рівня безпеки та оптимізації роботи банківських установ.

Мета даної роботи полягає в розробці мобільного додатку для онлайн-банкінгу з інтеграцією серверної частини на базі Windows Servers, а також оцінці ефективності та безпеки такого рішення для практичного застосування у фінансовій сфері.

1 АРХІТЕКТУРИ ТА ТЕХНОЛОГІЧНІ ЗАСАДИ СТВОРЕННЯ СУЧАСНОГО МОБІЛЬНОГО ДОДАТКУ ОНЛАЙН-БАНКІНГУ

Сучасний онлайн-банкінг є однією з найбільш технологічно просунутих галузей, де інноваційні рішення забезпечують зручність, безпеку та доступність фінансових послуг для мільйонів користувачів по всьому світу. Зростання попиту на цифрові банківські сервіси, спричинене швидким ритмом життя, глобалізацією та цифровізацією, вимагає створення надійних, масштабованих і безпечних систем, які поєднують мобільні додатки та потужні серверні інфраструктури. Мобільні додатки онлайн-банкінгу дозволяють користувачам здійснювати широкий спектр операцій – від перевірки балансу і переказів до управління інвестиціями та оплати рахунків – у будь-який час і з будь-якого місця. У той же час, серверна частина забезпечує обробку цих операцій, захист даних і стабільність роботи системи під високим навантаженням. Архітектури та технологічні засади створення таких додатків включають вибір відповідних платформ для клієнтської та серверної частин, проектування архітектури клієнт-сервер, забезпечення безпеки, а також використання сучасних методологій розробки. У цьому контексті ключовими технологіями є Flutter для створення мобільних додатків і Windows Servers для серверної інфраструктури, які разом формують основу для ефективних і безпечних банківських систем.

Мобільні додатки є основним інтерфейсом для взаємодії користувачів із банківськими сервісами, і їхня розробка вимагає ретельного вибору технологій, які забезпечують високу продуктивність, зручність використання та кросплатформну сумісність. Одним із провідних рішень для створення таких додатків є Flutter – відкритий фреймворк від Google, який використовує мову програмування Dart. Flutter дозволяє розробникам створювати єдиний код для платформ iOS і Android, що значно скорочує час і витрати на розробку та підтримку порівняно з нативними технологіями, такими як Swift для iOS або Kotlin для Android. Завдяки компіляції в нативний код, Flutter забезпечує високу продуктивність, що є критично важливим для банківських додатків, де швидке відображення даних, наприклад, балансу

рахунку чи історії транзакцій, безпосередньо впливає на користувацький досвід. Наприклад, сторінка переказів у банківському додатку, створеному на Flutter, може містити інтерактивні елементи, такі як поля введення суми та отримувача, а також кнопку підтвердження, які реагують миттєво завдяки оптимізованій рендеринговій системі Flutter.

Фреймворк пропонує багатий набір віджетів, які дозволяють створювати складні та естетично привабливі інтерфейси користувача. Наприклад, у банківському додатку можна реалізувати дашборд із графіками витрат, інтерактивними формами для платежів або push-сповіщеннями про завершення транзакцій. Функція Hot Reload (гаряче перезавантаження) значно прискорює процес розробки, дозволяючи розробникам бачити зміни в коді в реальному часі без необхідності перекомпіляції всього додатку. Це особливо корисно під час створення складних інтерфейсів, таких як форми для введення даних картки чи підтвердження транзакцій. Для інтеграції з серверною частиною Flutter підтримує бібліотеки, такі як http або Dio, які спрощують виконання мережевих запитів до REST API. Наприклад, запит на отримання історії транзакцій може бути реалізований через GET-запит до серверного ендпоінта, а асинхронні функції Dart, такі як Future і async/await, забезпечують плавну обробку відповідей сервера. Однак Flutter має певні обмеження, зокрема більший розмір вихідних файлів (APK чи IPA) через вбудовану бібліотеку віджетів, а також потребу в додатковому коді для реалізації платформоспецифічних функцій, таких як біометрична аутентифікація чи інтеграція з апаратними модулями безпеки.

Окрім Flutter, у розробці банківських додатків використовуються й інші технології. React Native, створений Facebook, є альтернативою для кросплатформної розробки, але він поступається Flutter за продуктивністю через використання JavaScript і мосту для взаємодії з нативними компонентами. Нативна розробка залишається актуальною для банківських систем із високими вимогами до інтеграції з апаратним забезпеченням, але її висока вартість робить кросплатформні рішення більш привабливими. У деяких випадках банки експериментують із прогресивними вебдодатками (PWA), які дозволяють

запускати сервіси через браузер, але їхні обмеження в доступі до нативних функцій пристрою роблять їх менш популярними для повноцінного онлайн-банкінгу.

Серверна інфраструктура є серцем будь-якої системи онлайн-банкінгу, оскільки вона відповідає за обробку транзакцій, зберігання даних і забезпечення безпеки. Одним із провідних рішень для серверної частини є Windows Servers – операційна система від Microsoft, яка широко застосовується в корпоративних і фінансових середовищах завдяки своїй надійності, масштабованій архітектурі та підтримці потужних інструментів. Windows Servers забезпечує комплексне середовище для розгортання серверних додатків, обробки запитів від клієнтів і захисту конфіденційної інформації. Основними компонентами Windows Servers для банківських систем є Internet Information Services (IIS) як вебсервер, .NET Core для реалізації бізнес-логіки, SQL Server як система управління базами даних і Active Directory для управління доступом.

IIS відіграє ключову роль у обробці HTTP-запитів від мобільних додатків, забезпечуючи високу продуктивність і захист від атак, таких як DDoS. Наприклад, IIS може обробляти тисячі запитів на перевірку балансу чи здійснення транзакцій одночасно, використовуючи пул потоків і механізми кешування. .NET Core, у свою чергу, є сучасною платформою для створення масштабованих серверних додатків, які реалізують бізнес-логіку, таку як валідація транзакцій, розрахунок комісій чи генерація звітів. Завдяки підтримці асинхронного програмування, .NET Core забезпечує швидку обробку запитів, що є критично важливим для систем із великою кількістю одночасних користувачів. SQL Server використовується для зберігання структурованих даних, таких як інформація про користувачів (логіни, зашифровані паролі), транзакції (сума, дата, статус) і журнали операцій. Транзакційна цілісність SQL Server гарантує, що, наприклад, переказ коштів не буде виконаний, якщо на рахунку недостатньо грошей, що є фундаментальним для фінансових систем. Active Directory дозволяє управляти правами доступу, забезпечуючи, наприклад, щоб лише авторизовані адміністратори мали доступ до аналітичних даних, тоді як звичайні користувачі можуть переглядати лише власні транзакції.

Порівняно з іншими серверними платформами, такими як Linux, Windows Servers має низку переваг, зокрема інтуїтивний графічний інтерфейс для адміністрування, тісну інтеграцію з екосистемою Microsoft (наприклад, Azure для хмарних сервісів чи Power BI для аналітики) і сертифікацію безпеки, таку як FIPS 140-2, що є важливим для банківських установ. Однак Linux може бути дешевшим і гнучкішим для кастомізації, що робить його популярним у деяких стартапах і фінтех-компаніях. Для банківських систем Windows Servers часто обирають через його надійність і підтримку SQL Server, який є стандартом для обробки фінансових даних.

Архітектура клієнт-сервер є основою для взаємодії між мобільним додатком і серверною частиною в системах онлайн-банкінгу. У цій моделі клієнт (мобільний додаток на Flutter) надсилає запити до сервера, який обробляє їх, виконує бізнес-логіку і повертає відповіді. Основним механізмом обміну даними є REST API, який дозволяє клієнту виконувати операції, такі як перевірка балансу (GET-запит до /api/balance), здійснення переказів (POST-запит до /api/transfers) чи оновлення даних користувача (PUT-запит до /api/users). REST є безстанним, що спрощує масштабування системи, оскільки сервер не зберігає інформацію про попередні запити. Для функцій у реальному часі, таких як сповіщення про нові транзакції, може використовуватися WebSocket, який забезпечує двосторонній зв’язок між клієнтом і сервером. Формат JSON є стандартом для передачі даних завдяки своїй легкості та підтримці як у Dart, так і в .NET Core. Архітектура клієнт-сервер дозволяє розділити розробку клієнтської та серверної частин, що полегшує оновлення однієї частини без впливу на іншу, а також забезпечує масштабованість за рахунок розгортання сервера на кількох вузлах.

Безпека є одним із найважливіших аспектів створення банківських систем, оскільки витік даних або несанкціонований доступ може призвести до значних фінансових втрат і втрати довіри клієнтів. Windows Servers пропонує комплексний набір інструментів для захисту серверної інфраструктури. Шифрування даних є основою безпеки: HTTPS із протоколом TLS 1.3 використовується для захисту даних під час передачі, наприклад, інформації про перекази чи номери карток. У

базі даних застосовується прозоре шифрування даних (TDE) у SQL Server для захисту конфіденційної інформації. Аутентифікація та авторизація реалізуються через протоколи, такі як OAuth 2.0 або JSON Web Tokens (JWT). Наприклад, користувач входить у додаток, вводячи логін і пароль, отримує токен доступу, який надалі використовується для авторизації запитів. Сервер перевіряє підпис токена перед обробкою запитів, що запобігає несанкціонованому доступу.

Захист від кібератак є ще одним важливим аспектом. SQL-ін'єкції запобігаються використанням параметризованих запитів у SQL Server, які унеможливлюють вставку шкідливого коду. Захист від міжсайтового скриптингу (XSS) і атак типу CSRF реалізується через налаштування заголовків безпеки в IIS, таких як X-Content-Type-Options. Для захисту від DDoS-атак використовуються брандмауери Windows і обмеження кількості запитів (rate limiting). Моніторинг і аудит операцій також є обов'язковими: усі дії, такі як логіни чи транзакції, записуються в журнали, які аналізуються для виявлення підозрілої активності. Windows Event Viewer дозволяє відстежувати системні події та помилки, що допомагає адміністраторам швидко реагувати на інциденти. Банківські системи також відповідають міжнародним стандартам безпеки, таким як PCI DSS (для обробки платіжних карток) і ISO 27001 (для управління інформаційною безпекою).

Методології розробки відіграють ключову роль у створенні мобільних додатків онлайн-банкінгу, забезпечуючи структурований підхід до аналізу, проектування, реалізації та тестування. Гнучка методологія (Agile) є стандартом для таких проєктів, оскільки дозволяє адаптуватися до змін вимог і швидко доставляти працюючі версії продукту. Розробка поділяється на спринти, кожен із яких триває 1–2 тижні і завершується демонстрацією нового функціоналу, наприклад, сторінки переказів чи API для транзакцій. Щоденні стендапи допомагають команді координувати роботу, а ретроспективи дозволяють аналізувати виконану роботу для покращення процесів. Життєвий цикл розробки програмного забезпечення (SDLC) включає етапи аналізу вимог, проектування (створення UML-діаграм і специфікацій API), розробки, тестування (модульне, інтеграційне, навантажувальне) і розгортання. Безперервна інтеграція та доставка

(CI/CD) автоматизують тестування і розгортання коду, використовуючи інструменти, такі як GitHub Actions або Azure DevOps. Наприклад, після внесення змін у код Flutter або .NET Core запускаються автоматичні тести, а у разі успіху серверна частина розгортається на IIS.

Інструменти управління проєктом, такі як Jira або Trello, дозволяють відстежувати завдання, створювати беклог і координувати роботу команди. Система контролю версій Git використовується для зберігання коду, з окремими репозиторіями для клієнтської (Flutter) і серверної (.NET Core) частин. Документування вимог, специфікацій API і архітектурних рішень здійснюється через платформи, такі як Confluence. Для інтеграції клієнтської та серверної частин використовується специфікація OpenAPI (Swagger), яка описує ендпоінти, такі як /api/transactions, і їхні параметри. Тестування інтеграції включає емуляцію запитів за допомогою інструментів, таких як Postman, для перевірки коректності відповідей сервера. Управління помилками реалізується через стандартні коди HTTP (400 для некоректних даних, 401 для неавторизованого доступу) і повернення зрозумілих повідомлень, наприклад, "Недостатньо коштів".

Сучасні тенденції в онлайн-банкінгу також впливають на вибір технологій і архітектур. Хмарні рішення, такі як Microsoft Azure, дедалі частіше використовуються для масштабування серверної інфраструктури, дозволяючи банкам обробляти пікові навантаження без значних інвестицій у фізичне обладнання. Штучний інтелект і машинне навчання застосовуються для аналізу транзакцій, виявлення шахрайства та персоналізації пропозицій, наприклад, рекомендацій інвестиційних продуктів. Біометрична аутентифікація, така як розпізнавання обличчя чи відбитків пальців, стає стандартом для підвищення безпеки входу в додаток. Блокчейн-технології досліджуються для забезпечення прозорості та безпеки транзакцій, хоча їхнє впровадження в банківських системах поки що обмежене через складність і регуляторні вимоги.

Таким чином, архітектури та технологічні засади створення сучасного мобільного додатку онлайн-банкінгу базуються на поєднанні потужних клієнтських і серверних технологій, таких як Flutter і Windows Servers, а також на

продуманій архітектурі клієнт-сервер, яка забезпечує масштабованість і надійність. Безпека залишається наріжним каменем, вимагаючи впровадження шифрування, аутентифікації та захисту від атак. Гнучкі методології розробки, такі як Agile, разом із сучасними інструментами управління проєктами та CI/CD, дозволяють створювати високоякісні продукти, які відповідають очікуванням користувачів і регуляторним вимогам. У міру розвитку технологій, таких як хмарні обчислення, штучний інтелект і блокчейн, онлайн-банкінг продовжуватиме еволюціонувати, пропонуючи нові можливості для користувачів і банківських установ.

1.1 Огляд сучасних технологій для онлайн-банкінгу

Сучасний онлайн-банкінг є однією з найбільш технологічно просунутих галузей, яка поєднує інноваційні рішення для забезпечення зручності, безпеки та доступності фінансових послуг. Зростання попиту на цифрові банківські сервіси, спричинене швидким ритмом життя, глобалізацією та цифровізацією, вимагає використання передових технологій для створення мобільних додатків і серверних інфраструктур. Ці технології дозволяють користувачам здійснювати фінансові операції, такі як перекази, платежі, перевірка балансу чи управління інвестиціями, у будь-який час і з будь-якого місця. У той же час, банківські установи стикаються з викликами, пов’язаними з кібербезпекою, масштабуванням систем і забезпеченням безперебійної роботи під високим навантаженням. У цьому контексті ключовими є мобільні платформи, такі як Flutter, серверні рішення, зокрема Windows Servers, а також технології безпеки, інтеграції та обробки даних. Цей текст присвячено огляду сучасних технологій, що формують основу онлайн-банкінгу, їхніх можливостей, переваг і викликів.

Однією з основних складових онлайн-банкінгу є мобільні додатки, які виступають головним інтерфейсом для взаємодії користувачів із банківськими сервісами. Сьогодні розробка таких додатків дедалі частіше базується на кросплатформних фреймворках, серед яких особливо виділяється Flutter, створений компанією Google. Flutter використовує мову програмування Dart, яка поєднує об’єктно-орієнтоване програмування з підтримкою асинхронних операцій,

що ідеально підходить для роботи з мережевими запитами, характерними для банківських систем. Основна перевага Flutter полягає в можливості створення єдиного коду для платформ iOS і Android, що значно скорочує час і витрати на розробку та підтримку. Завдяки компіляції в нативний код, Flutter забезпечує високу продуктивність, порівнянну з нативними додатками, написаними на Swift чи Kotlin. Це критично важливо для банківських додатків, де швидке відображення даних, наприклад, балансу чи історії транзакцій, впливає на користувацький досвід.

Flutter пропонує багатий набір віджетів, які дозволяють створювати складні та інтуїтивно зрозумілі інтерфейси. Наприклад, у банківському додатку можна реалізувати інтерактивний дашборд із графіками витрат, формами для переказів чи push-сповіщеннями про нові транзакції. Функція Hot Reload (гаряче перезавантаження) значно прискорює процес розробки, дозволяючи розробникам миттєво бачити зміни в інтерфейсі чи логіці без повної перекомпіляції додатку. Для інтеграції з серверною частиною Flutter підтримує бібліотеки, такі як Dio або http, які спрощують виконання HTTP-запитів до REST API. Наприклад, запит на отримання історії транзакцій може бути реалізований через GET-запит до серверного ендпоінта, а асинхронні функції Dart (Future, async/await) забезпечують плавну обробку відповідей. Однак Flutter має певні обмеження, зокрема більший розмір вихідних файлів (APK чи IPA) порівняно з нативними додатками через вбудовану бібліотеку віджетів. Крім того, для реалізації специфічних нативних функцій, таких як біометрична аутентифікація чи інтеграція з апаратними модулями безпеки, може знадобитися написання платформоспецифічного коду, що ускладнює розробку.

Поряд із Flutter, у розробці банківських додатків використовуються й інші технології. Наприклад, React Native, створений Facebook, є альтернативою для кросплатформної розробки, хоча він поступається Flutter за продуктивністю через використання JavaScript і мосту для взаємодії з нативними компонентами. Нативна розробка (Swift для iOS і Kotlin для Android) залишається актуальною для банківських систем із високими вимогами до продуктивності чи інтеграції з апаратним забезпеченням, але вона є дорожчою через необхідність підтримки двох

окремих кодових баз. Усе частіше банки також експериментують із прогресивними вебдодатками (PWA), які дозволяють запускати банківські сервіси через браузер, але їхні обмеження в доступі до нативних функцій пристрою роблять їх менш популярними для повноцінного онлайн-банкінгу.

Серверна інфраструктура є не менш важливою складовою онлайн-банкінгу, оскільки вона відповідає за обробку транзакцій, зберігання даних і забезпечення безпеки. Одним із провідних рішень для серверної частини є Windows Servers – операційна система від Microsoft, яка широко застосовується в корпоративних і фінансових середовищах. Windows Servers вирізняється надійністю, підтримкою широкого спектру інструментів і високим рівнем безпеки, що робить її оптимальним вибором для банківських систем. Основними компонентами Windows Servers для онлайн-банкінгу є Internet Information Services (IIS) як вебсервер, .NET Core для створення серверних додатків, SQL Server як система управління базами даних і Active Directory для управління доступом.

IIS забезпечує обробку HTTP-запитів від мобільних додатків, підтримуючи високу продуктивність і захист від атак, таких як DDoS. Наприклад, IIS може обробляти тисячі запитів на перевірку балансу чи здійснення транзакцій одночасно, використовуючи пул потоків і кешування. .NET Core, у свою чергу, є кросплатформною платформою для реалізації бізнес-логіки, такої як валідація транзакцій, розрахунок комісій чи генерація фінансових звітів. Завдяки підтримці асинхронного програмування, .NET Core забезпечує швидку обробку запитів, що є критично важливим для систем із високим навантаженням. SQL Server використовується для зберігання структурованих даних, таких як інформація про користувачів (логіни, зашифровані паролі), транзакції (сума, дата, статус) і журнали операцій. SQL Server підтримує транзакційну цілісність, що гарантує, наприклад, що переказ коштів не буде виконаний, якщо на рахунку недостатньо грошей. Active Directory дозволяє управляти правами доступу, забезпечуючи, наприклад, щоб лише авторизовані адміністратори мали доступ до аналітичних даних.

1.1.1 Мобільні платформи та Flutter

Flutter – це відкритий фреймворк від Google для створення кросплатформних мобільних додатків, який використовує мову програмування Dart. Основна перевага Flutter полягає в можливості створення єдиного коду для платформ iOS та Android, що зменшує витрати на розробку та підтримку. Dart є об'єктно-орієнтованою мовою з підтримкою асинхронного програмування, що ідеально підходить для роботи з мережевими запитами, які є ключовими для банківських додатків.

Переваги Flutter для онлайн-банкінгу:

– *Продуктивність*. Flutter компілюється в нативний код, що забезпечує швидкість, близьку до нативних додатків, написаних на Swift (iOS) або Kotlin (Android). Це критично для таких операцій, як відображення балансу чи обробка транзакцій у реальному часі;

– *Гнучкий UI*. Бібліотека віджетів Flutter дозволяє створювати складні інтерфейси, наприклад, інтерактивні дашборди для перегляду фінансових даних, графіки витрат або форми для здійснення платежів;

– *Hot Reload*. Функція гарячого перезавантаження прискорює розробку, дозволяючи миттєво бачити зміни в інтерфейсі чи логіці без перекомпіляції всього додатку;

– *Бібліотеки для мережевих запитів*. Пакети, такі як http або Dio, спрощують інтеграцію з REST API, що є основою для взаємодії з серверною частиною. Наприклад, запит на отримання історії транзакцій може бути реалізований через GET-запит до серверного ендпоінта.

Приклад використання. У банківському додатку Flutter може бути використаний для створення сторінки переказів, де користувач вводить суму, отримувач і підтверджує операцію. Віджети, такі як TextField і ElevatedButton, забезпечують інтуїтивний інтерфейс, а асинхронні функції Dart (Future, async/await) обробляють мережеві запити до сервера для виконання транзакції.

1.1.2 Серверні технології на базі Windows Servers

Огляд Windows Servers. Windows Servers – це серверна операційна система від Microsoft, яка широко використовується в корпоративних середовищах, зокрема для фінансових систем. Вона підтримує широкий набір інструментів для створення масштабованих, безпечних і надійних серверних додатків, що робить її ідеальною для онлайн-банкінгу.

Ключові компоненти Windows Servers для банкінгу:

– *Internet Information Services (IIS)*. Вебсервер для розгортання REST API, який обробляє HTTP-запити від мобільного додатку. IIS підтримує високе навантаження і має вбудовані механізми для захисту від атак, таких як DDoS;

– *.NET Core.* Кросплатформна платформа для створення серверних додатків. Вона дозволяє реалізувати бізнес-логіку, наприклад, обробку транзакцій, перевірку балансу чи генерацію звітів. .NET Core підтримує асинхронне програмування, що забезпечує швидку обробку запитів;

– *SQL Server.* Система управління базами даних (СУБД) для зберігання інформації про користувачів (персональні дані, паролі), транзакції (сума, дата, статус) і журнали операцій. SQL Server підтримує транзакційну цілісність, що є критично важливим для фінансових операцій;

– *Active Directory.* Інструмент для управління користувачами та їхніми правами доступу. Наприклад, адміністратори банку можуть мати доступ до аналітики, тоді як звичайні користувачі – лише до своїх транзакцій.

Переваги Windows Servers:

– *Безпека.* Підтримка сучасних протоколів шифрування (TLS 1.3), захист від SQL-ін’єкцій і вбудовані інструменти для моніторингу безпеки;

– *Інтеграція з екосистемою Microsoft.* Легке поєднання з іншими продуктами, такими як Azure для хмарних сервісів або Power BI для аналітики;

– *Масштабованість.* Можливість розгортання на кластерах серверів для обробки великої кількості запитів від користувачів.

Порівняння з Linux. На відміну від Linux, Windows Servers має більш інтуїтивний графічний інтерфейс для адміністрування і кращу підтримку корпоративних рішень Microsoft. Однак Linux може бути дешевшим і гнучкішим для кастомізації. Для банківських систем Windows Servers часто обирають через сертифікацію безпеки (наприклад, FIPS 140-2) і підтримку SQL Server.

Приклад використання. Серверна частина на Windows Servers може обробляти запит на переказ коштів. Мобільний додаток надсилає POST-запит до ендпоінта /api/transactions, який містить дані про суму, отримувача і токен аутентифікації. Сервер перевіряє баланс, виконує транзакцію в SQL Server і повертає статус операції (успіх або помилка).

1.2 Теоретичні основи серверної інфраструктури

Серверна інфраструктура є основою будь-якої сучасної інформаційної системи, зокрема систем онлайн-банкінгу, які вимагають високої продуктивності, безпеки та надійності для обробки фінансових транзакцій, управління даними користувачів і забезпечення безперебійного доступу до сервісів. Теоретичні основи серверної інфраструктури охоплюють принципи побудови архітектури, технології обробки та зберігання даних, механізми забезпечення безпеки, а також підходи до масштабування і моніторингу систем. У контексті онлайн-банкінгу серверна інфраструктура відіграє ключову роль, оскільки вона відповідає за виконання бізнес-логіки, захист конфіденційної інформації та підтримку стабільної взаємодії з клієнтськими додатками, такими як мобільні додатки. Однією з провідних платформ для створення такої інфраструктури є Windows Servers, яка забезпечує надійне середовище для розгортання серверних додатків, обробки запитів і управління базами даних. Цей текст присвячено аналізу теоретичних основ серверної інфраструктури, з фокусом на архітектуру клієнт-сервер, питання безпеки та їх застосування в банківських системах.

Серверна інфраструктура для онлайн-банкінгу базується на архітектурі клієнт-сервер, яка є стандартом для розподілених систем, де клієнт (наприклад, мобільний додаток) надсилає запити до сервера, а сервер обробляє їх і повертає

відповіді. Ця модель дозволяє розділити функціональність між клієнтською частиною, яка відповідає за інтерфейс користувача, і серверною частиною, яка виконує обчислення, зберігає дані та забезпечує безпеку. У банківських системах клієнтська частина, наприклад, мобільний додаток, створений на Flutter, відображає користувачу баланс рахунку, історію транзакцій або форми для здійснення платежів. Серверна частина, розгорнута на Windows Servers, обробляє запити, такі як перевірка достатності коштів для переказу, оновлення балансу чи запис деталей транзакції в базу даних. Архітектура клієнт-сервер має кілька ключових компонентів: клієнтський додаток, сервер додатків, база даних і мережеві протоколи для обміну даними. Основним механізмом взаємодії є REST API, який використовує HTTP-запити (GET, POST, PUT, DELETE) для виконання операцій, таких як створення переказу чи отримання історії транзакцій. Наприклад, GET-запит до ендпоінта `/api/balance` повертає JSON-об’єкт із інформацією про баланс, тоді як POST-запит до `/api/transfers` ініціює переказ коштів.

REST API є безстанним, що означає, що кожен запит містить усю необхідну інформацію, а сервер не зберігає стан між запитами. Це спрощує масштабування системи, оскільки нові серверні вузли можна додавати без необхідності синхронізації стану. Однак для функцій у реальному часі, таких як push-сповіщення про завершення транзакції, використовується WebSocket, який забезпечує двосторонній зв’язок між клієнтом і сервером. WebSocket дозволяє серверу надсилати повідомлення клієнту без очікування запиту, що ідеально підходить для сповіщень про зміну статусу платежу чи виявлення підозрілої активності. Формат JSON є стандартом для передачі даних завдяки своїй легкості, читабельності та підтримці в більшості мов програмування, включаючи Dart (для Flutter) і C# (для .NET Core на Windows Servers). Архітектура клієнт-сервер забезпечує модульність, дозволяючи розробляти та оновлювати клієнтську і серверну частини незалежно, а також підтримує масштабованість за рахунок розподілу навантаження між кількома серверами.

RESTful API та JWT-токен є ключовими елементами сучасних веб-додатків, зокрема в проєкті UZBUKBANKAPI, де вони забезпечують взаємодію клієнтів із

сервером та безпечну автентифікацію. RESTful API — це архітектурний стиль для створення веб-API, який базується на принципах HTTP-протоколу. Він дозволяє клієнтам, таким як веб-браузери чи мобільні додатки, взаємодіяти з сервером через стандартизовані запити. У REST ресурси, наприклад користувачі чи картки, представлені унікальними URL-адресами, а операції з ними виконуються через HTTP-методи. У проєкті UZBUKBANKAPI це реалізовано через ендпоїнти, такі як створення користувача чи отримання профілю, де кожен запит відповідає певному ресурсу та повертає дані у форматі JSON із стандартними кодами відповіді, як-от 200 для успіху чи 401 для помилки авторизації.

Основна особливість RESTful API — це клієнт-серверна модель, де клієнт і сервер розвиваються незалежно, що спрощує масштабування. Кожен запит є незалежним і містить усю необхідну інформацію, оскільки сервер не зберігає стан між запитами. Це означає, що для авторизації в кожному запиті передається токен. REST використовує стандартні HTTP-методи: GET для отримання даних, POST для створення, PUT або PATCH для оновлення, DELETE для видалення. У проєкті ці методи застосовуються для операцій із профілем користувача чи автентифікацією. API також підтримує кешування відповідей для підвищення продуктивності та шарувату архітектуру, що дозволяє додавати проксі чи балансувальники навантаження. Переваги REST включають простоту завдяки стандартним протоколам, гнучкість для різних клієнтів і масштабованість, але відсутність збереження стану вимагає додаткових механізмів, як-от токени, для автентифікації.

JWT-токен, або JSON Web Token, — це компактний механізм для безпечної передачі даних між клієнтом і сервером у вигляді JSON-об’єкта, який використовується для автентифікації та авторизації. У проєкті UZBUKBANKAPI JWT застосовується для захисту ендпоїнтів, таких як отримання профілю. Токен складається з трьох частин: заголовка, даних і підпису, розділених крапками. Заголовок містить тип токена та алгоритм підпису, наприклад HMAC SHA256. Дані включають інформацію, як-от ідентифікатор користувача, ім’я чи термін дії токена. Підпис створюється з використанням секретного ключа, забезпечуючи цілісність і

автентичність. Усі частини кодуються в Base64, формуючи рядок, який передається в заголовку Authorization як Bearer-токен.

HMAC SHA256 — це криптографічний метод для створення безпечного цифрового підпису, який використовується для перевірки цілісності та автентичності даних, зокрема в JWT-токенах у цьому проєкті. HMAC (Hash-based Message Authentication Code) комбінує хеш-функцію SHA256 із секретним ключем, щоб генерувати унікальний підпис для повідомлення.

Windows Servers є однією з провідних платформ для розгортання серверної інфраструктури банківських систем завдяки своїй надійності, інтеграції з корпоративними рішеннями Microsoft і підтримці потужних інструментів для обробки даних і безпеки. Основними компонентами Windows Servers у контексті онлайн-банкінгу є Internet Information Services (IIS), .NET Core, SQL Server і Active Directory. IIS виступає вебсервером, який обробляє HTTP-запити від клієнтських додатків, забезпечуючи високу продуктивність і захист від атак, таких як DDoS. Наприклад, IIS може обробляти тисячі одночасних запитів на перевірку балансу, використовуючи пул потоків і механізми кешування. .NET Core є платформою для створення серверних додатків, які реалізують бізнес-логіку, таку як валідація транзакцій, розрахунок комісій чи генерація звітів. Завдяки асинхронному програмуванню, .NET Core забезпечує швидку обробку запитів, що є критично важливим для систем із великою кількістю користувачів. SQL Server, у свою чергу, є системою управління базами даних, яка зберігає структуровані дані, такі як таблиці `Users` (інформація про клієнтів), `Transactions` (деталі транзакцій) і `Logs` (журнали операцій). Транзакційна цілісність SQL Server гарантує, що операції, такі як переказ коштів, виконуються коректно або скасовуються у разі помилки, що є фундаментальним для фінансових систем. Active Directory дозволяє управляти правами доступу, забезпечуючи, наприклад, щоб лише авторизовані адміністратори мали доступ до аналітичних даних.

Безпека є наріжним каменем серверної інфраструктури банківських систем, оскільки витік даних, несанкціонований доступ або порушення цілісності транзакцій можуть призвести до значних фінансових втрат і втрати довіри клієнтів.

Windows Servers пропонує комплексний набір інструментів для забезпечення безпеки на всіх рівнях. Шифрування даних є основою захисту: HTTPS із протоколом TLS 1.3 використовується для шифрування запитів і відповідей, наприклад, інформації про перекази чи номери карток, що передаються між клієнтом і сервером. У базі даних застосовується прозоре шифрування даних (TDE) у SQL Server, яке захищає конфіденційну інформацію, таку як персональні дані користувачів, від несанкціонованого доступу навіть у разі фізичного доступу до сервера. Аутентифікація та авторизація реалізуються через протоколи, такі як OAuth 2.0 або JSON Web Tokens (JWT). Наприклад, користувач входить у мобільний додаток, вводячи логін і пароль, отримує JWT-токен, який надалі додається до заголовків HTTP-запитів для авторизації. Сервер перевіряє підпис токена перед обробкою запиту, що запобігає несанкціонованому доступу.

Захист від кібератак є ще одним критично важливим аспектом. SQL-ін’єкції, які можуть дозволити зловмисникам виконувати шкідливі запити до бази даних, запобігаються використанням параметризованих запитів у SQL Server. Наприклад, запит `SELECT balance FROM Accounts WHERE user_id = @userId` використовує параметр `@userId`, що унеможливлює вставку шкідливого коду. Захист від міжсайтового скриптингу (XSS) і атак типу CSRF реалізується через налаштування заголовків безпеки в IIS, таких як `X-Content-Type-Options` і `Content-Security-Policy`. Для захисту від DDoS-атак використовуються брандмауери Windows і обмеження кількості запитів (rate limiting), які запобігають перевантаженню серверів.

1.2.1 Архітектура клієнт-сервер

Основи архітектури: Архітектура клієнт-сервер є стандартом для систем онлайн-банкінгу, де клієнт (мобільний додаток) взаємодіє з сервером через мережеві запити. Основні компоненти:

– *Клієнт.* Мобільний додаток на Flutter, який відображає інтерфейс користувача (UI) і надсилає запити до сервера. Наприклад, запит на перевірку балансу надсилається через GET /api/balance;

– *Сервер.* Windows Servers, який обробляє запити, виконує бізнес-логіку (перевірка достатності коштів, оновлення балансу) і взаємодіє з базою даних;

– *База даних.* SQL Server, який зберігає структуровані дані, такі як таблиці Users (інформація про клієнтів), Transactions (деталі транзакцій) і Logs (журнали операцій).

Механізми взаємодії:

– *REST API.* Стандарт для обміну даними між клієнтом і сервером. Наприклад, ендпоінт /api/transfers дозволяє створювати перекази, повертаючи JSON-відповідь із результатом операції. REST є безстанним, що спрощує масштабування;

– *WebSocket.* Використовується для функцій у реальному часі, таких як push-сповіщення про нові транзакції чи зміну статусу платежу;

– *Формат даних.* JSON є стандартом для передачі даних завдяки своїй легкості та підтримці в Dart і .NET Core.

Переваги архітектури:

– *Модульність.* Клієнт і сервер розробляються незалежно, що полегшує оновлення однієї частини без впливу на іншу;

– *Масштабованість.* Сервер може бути розгорнутий на кількох вузлах для обробки великої кількості запитів;

– *Безпека.* Використання HTTPS і токенів аутентифікації (JWT) забезпечує захист даних.

Приклад реалізації. Користувач відкриває додаток і натискає "Переглянути баланс". Flutter-додаток надсилає GET-запит до /api/balance з токеном JWT у заголовку Authorization. Сервер на Windows Servers перевіряє токен, виконує SQL-запит до таблиці Accounts і повертає JSON із сумою балансу.

1.2.2 Безпека серверної частини

Значення безпеки: У банківських системах безпека є пріоритетом, оскільки витік даних або несанкціонований доступ може призвести до фінансових втрат і втрати довіри клієнтів. Windows Servers пропонує комплексний набір інструментів для захисту серверної інфраструктури. Ключові аспекти безпеки:

Шифрування даних:

– *Під час передачі.* Використання HTTPS із протоколом TLS 1.3 для шифрування всіх запитів і відповідей. Наприклад, дані про переказ (сума, отримувач) передаються в зашифрованому вигляді;

– *У базі даних.* Застосування прозорого шифрування даних (TDE) у SQL Server для захисту конфіденційної інформації, такої як номери карток.

Аутентифікація та авторизація:

– *OAuth 2.0.* Протокол для безпечної аутентифікації користувачів. Наприклад, користувач входить через логін і пароль, отримує токен доступу, який використовується для подальших запитів;

– *JWT.* Токени JSON Web Token для перевірки ідентичності користувача. Сервер перевіряє підпис токена перед обробкою запитів.

Захист від атак:

– *SQL-ін'єкції.* Використання параметризованих запитів у SQL Server для запобігання вставки шкідливого коду;

– *XSS і CSRF.* Налаштування заголовків безпеки в IIS (наприклад, X-Content-Type-Options) для захисту від міжсайтового скриптингу;

– *DDoS.* Використання брандмауера Windows і обмеження кількості запитів (rate limiting) для запобігання перевантаження сервера.

Моніторинг і аудит:

– *Журналювання всіх операцій* (наприклад, логін, транзакції) у таблиці Logs для аналізу підозрілої активності;

– *Використання Windows Event Viewer* для відстеження системних подій і помилок.

Приклад реалізації. Для захисту переказу сервер перевіряє JWT-токен у запиті, використовує параметризований SQL-запит (SELECT balance FROM Accounts WHERE user_id = @userId) для перевірки балансу, і застосовує шифрування AES для збереження деталей транзакції в базі даних.

Стандарти безпеки:

– *PCI DSS.* Стандарт безпеки для обробки платіжних карток, який вимагає шифрування, регулярного тестування безпеки та обмеження доступу до даних;

– *ISO 27001.* Стандарт управління інформаційною безпекою, який використовується для сертифікації банківських систем.

1.3 Методологія розробки

Цей розділ описує методологічні підходи до розробки мобільного додатку та серверної частини, включаючи вибір моделі розробки, інструменти управління проєктом і процеси інтеграції клієнтської та серверної частин.

Гнучка методологія (Agile). Agile є оптимальним вибором для розробки банківських додатків, оскільки дозволяє адаптуватися до змін вимог і швидко доставляти працюючі версії продукту. Основні принципи:

– *Спринти.* Розробка поділена на 2-тижневі ітерації, кожна з яких завершується демонстрацією нового функціоналу (наприклад, сторінки переказів чи API для транзакцій);

– *Щоденні стендапи.* Короткі зустрічі для обговорення прогресу та вирішення блокерів;

– *Ретроспективи.* Аналіз виконаної роботи для покращення процесів у наступних спринтах.

Життєвий цикл розробки (SDLC):

– *Аналіз вимог.* Визначення функціональних (перекази, платежі) і нефункціональних (безпека, продуктивність) вимог;

– *Проектування.* Створення UML-діаграм (наприклад, діаграми класів для бази даних) і специфікації API;

– *Розробка.* Написання коду для Flutter (UI, мережеві запити) і .NET Core (API, бізнес-логіка);

– *Тестування.* Модульне тестування (з використанням flutter_test і MSTest), інтеграційне тестування (перевірка API) і навантажувальне тестування;

– *Розгортання.* Розгортання серверної частини на Windows Servers і публікація додатку в App Store та Google Play;

– *Підтримка.* Моніторинг помилок і випуск оновлень.

Інструменти управління проєктом:

– *Jira.* Для відстеження завдань, створення беклогу і управління спринтами;

– *Git.* Система контролю версій для зберігання коду. Використовуються репозиторії для клієнтської (Flutter) і серверної (.NET Core) частин;

– *Confluence.* Для документування вимог, специфікацій API і архітектурних рішень.

Безперервна інтеграція та доставка (CI/CD):

– *GitHub Actions або Azure DevOps*. Автоматизація тестування, збирання і розгортання коду. Наприклад, після пуша в Git запускаються тести для Flutter і .NET Core, а у разі успіху серверна частина розгортається на IIS.

– *Docker.* Для контейнеризації серверної частини, що спрощує розгортання і масштабування.

Інтеграція клієнтської та серверної частин:

– *Специфікація API.* Використання OpenAPI (Swagger) для документування ендпоінтів, таких як /api/transactions або /api/users;

– *Тестування інтеграції.* Емуляція запитів від Flutter до сервера за допомогою інструментів, таких як Postman, для перевірки коректності відповідей;

– *Управління помилками.* Реалізація стандартних кодів помилок HTTP (400, 401, 500) і повернення зрозумілих повідомлень (наприклад, "Недостатньо коштів").

Приклад процесу. Команда розробників визначає вимогу – створення функції переказів. У спринті розробляється UI на Flutter (2 віджети: поле введення суми і кнопка підтвердження), API на .NET Core (ендпоінт /api/transfers) і таблиця Transactions у SQL Server. Після тестування функціонал розгортається на сервері, а клієнт отримує оновлення через CI/CD.

2 АНАЛІЗ ВИМОГ І ВИБІР ТЕХНОЛОГІЙ ДЛЯ РОЗРОБКИ ДОДАТКУ З СЕРВЕРНОЮ ЧАСТИНОЮ

Розробка мобільного додатку для онлайн-банкінгу з інтеграцією серверної частини на базі Windows Servers є складним завданням, яке вимагає ретельного аналізу вимог, оцінки існуючих рішень і обґрунтованого вибору технологій, що відповідають потребам безпеки, продуктивності та зручності використання. Онлайн-банкінг як галузь стрімко розвивається, оскільки користувачі дедалі частіше віддають перевагу цифровим сервісам для управління фінансами, що зумовлює високі вимоги до функціональності, безпеки та масштабованості банківських систем. У цьому контексті аналіз вимог є першим і ключовим етапом, який визначає, які функції має виконувати додаток, які нефункціональні аспекти, такі як безпека та швидкодія, необхідно врахувати, і які технології найкраще підходять для реалізації проєкту. Вибір технологій, зокрема Flutter для клієнтської частини та Windows Servers для серверної, ґрунтується на їхніх можливостях забезпечити кросплатформну сумісність, надійність і захист даних. Цей текст присвячено аналізу вимог до системи онлайн-банкінгу, дослідженню аналогів, обґрунтуванню вибору технологій і проектуванню серверної інфраструктури, з особливим акцентом на проблематику цієї роботи.

Першим етапом розробки є аналіз вимог до системи онлайн-банкінгу, який поділяється на функціональні та нефункціональні вимоги. Функціональні вимоги визначають основні можливості, які має надавати додаток користувачам. Для мобільного додатку онлайн-банкінгу ключовими функціями є перегляд балансу рахунку, здійснення переказів між рахунками, оплата рахунків, перегляд історії транзакцій і управління профілем користувача, наприклад, зміна пароля чи налаштування сповіщень. Наприклад, функція переказу передбачає введення суми, вибір отримувача (за номером рахунку чи картки) і підтвердження операції, можливо, з використанням одноразового пароля (OTP) або біометричної аутентифікації. Крім того, додаток має підтримувати push-сповіщення про завершення транзакцій або підозрілу активність, а також надавати аналітику витрат

у вигляді графіків чи звітів. Ці функції вимагають тісної інтеграції з серверною частиною, яка обробляє запити, перевіряє достатність коштів і записує деталі операцій у базу даних. Нефункціональні вимоги стосуються якісних характеристик системи, таких як безпека, продуктивність, масштабовість і зручність використання. Безпека є критично важливою, оскільки банківські дані, такі як номери карток чи історія транзакцій, є конфіденційними і потребують захисту від несанкціонованого доступу. Продуктивність передбачає швидку обробку запитів, наприклад, відображення балансу за лічені секунди навіть при високому навантаженні. Масштабованість дозволяє системі обробляти зростаючу кількість користувачів, а зручність використання забезпечується інтуїтивно зрозумілим інтерфейсом, який мінімізує час, необхідний для виконання операцій.

Для визначення вимог проводяться консультації із зацікавленими сторонами, включаючи представників банку, користувачів і розробників. Наприклад, банк може вимагати підтримки кількох валют або інтеграції з платіжними системами, такими як Visa чи Mastercard, тоді як користувачі хочуть швидкого доступу до основних функцій і мінімальної кількості кроків для здійснення переказу. На основі цих даних створюється документ із вимогами, який включає специфікацію функціоналу, сценарії використання (use cases) і критерії приймання. Наприклад, сценарій використання для переказу може виглядати так: користувач входить у додаток, обирає функцію переказу, вводить дані, підтверджує операцію, і система надсилає сповіщення про успіх або помилку. Нефункціональні вимоги деталізуються через метрики, такі як час відповіді сервера (не більше 500 мс для 95% запитів) або рівень безпеки (відповідність стандарту PCI DSS). Цей етап є основою для вибору технологій, оскільки вимоги визначають, які інструменти здатні їх реалізувати.

Дослідження існуючих рішень є наступним кроком, який дозволяє оцінити, як аналогічні задачі вирішуються в інших банківських системах, і визначити найкращі практики. Популярні банківські додатки, такі як Monobank (Україна), Revolut (Велика Британія) і N26 (Німеччина), демонструють високий рівень функціональності та зручності. Наприклад, Monobank пропонує мінімалістичний

інтерфейс із швидким доступом до переказів, платежів і аналітики витрат, а також інтеграцію з платіжними системами для оплати комунальних послуг чи поповнення мобільного рахунку. Revolut підтримує мультивалютні рахунки, обмін валют за ринковими курсами і функції інвестування, що вимагає складної серверної логіки. N26 фокусується на простоті та прозорості, надаючи детальну історію транзакцій і сповіщення в реальному часі. Аналіз цих додатків показує, що успішні рішення поєднують інтуїтивний інтерфейс, швидку обробку операцій і високий рівень безпеки, наприклад, через двофакторну аутентифікацію чи біометрію.

На серверному рівні дослідження аналогів включає аналіз платформ, які використовуються для обробки банківських транзакцій. Windows Servers є популярним вибором для корпоративних банківських систем завдяки своїй надійності, інтеграції з екосистемою Microsoft і підтримці інструментів, таких як SQL Server і .NET Core. Наприклад, багато банків розгортають свої API на IIS (Internet Information Services), який забезпечує обробку тисяч одночасних запитів. Альтернативою є Linux-based сервери, які використовуються в фінтех-стартапах, таких як Revolut, завдяки їхній гнучкості та нижчій вартості. Однак Linux вимагає додаткових зусиль для інтеграції з корпоративними рішеннями, тоді як Windows Servers пропонує готові інструменти, такі як Active Directory для управління доступом. Аналіз серверних рішень також включає оцінку баз даних: SQL Server є стандартом для Windows-систем завдяки транзакційній цілісності та підтримці шифрування, тоді як PostgreSQL або MongoDB можуть використовуватися в Linux-системах для гнучкішого управління даними. Порівняння аналогів допомагає визначити, що Windows Servers є оптимальним вибором для проєкту завдяки його безпеці, масштабованості та підтримці корпоративних стандартів.

Вибір технологій для розробки додатку з серверною частиною ґрунтується на аналізі вимог і оцінці аналогів. Для клієнтської частини обрано Flutter, оскільки він дозволяє створювати кросплатформний мобільний додаток для iOS і Android із єдиною кодовою базою, що скорочує витрати на розробку. Flutter забезпечує високу продуктивність завдяки компіляції в нативний код, що необхідно для швидкого відображення даних, наприклад, балансу чи історії транзакцій. Його

бібліотека віджетів дозволяє створювати інтуїтивний інтерфейс, наприклад, форми для переказів із автозаповненням і валідацією даних. Бібліотеки, такі як Dio, спрощують інтеграцію з REST API, що є критично важливим для взаємодії з серверною частиною. Наприклад, запит на переказ може бути реалізований через POST-запит до ендпоінта `/api/transfers`, який містить JSON із сумою, отримувачем і токеном аутентифікації. Flutter також підтримує біометричну аутентифікацію через плагіни, що відповідає вимогам безпеки.

Для серверної частини обрано Windows Servers, який забезпечує надійне середовище для обробки запитів, зберігання даних і захисту інформації. Windows Servers підтримує IIS як вебсервер для розгортання REST API, .NET Core для реалізації бізнес-логіки, SQL Server для управління базами даних і Active Directory для аутентифікації. IIS дозволяє обробляти високе навантаження, що необхідно для банківських систем із тисячами одночасних користувачів. .NET Core забезпечує швидку розробку API, наприклад, ендпоінта для перевірки балансу, який виконує SQL-запит до таблиці `Accounts` і повертає JSON-відповідь. SQL Server гарантує транзакційну цілісність, що запобігає помилкам, таким як подвійне списання коштів. Active Directory дозволяє обмежити доступ до адміністративних функцій, наприклад, перегляду аналітики, лише для авторизованих користувачів. Вибір Windows Servers обґрунтований його сертифікацією безпеки (наприклад, FIPS 140-2) і тісною інтеграцією з SQL Server, що є стандартом для фінансових систем.

Проектування серверної інфраструктури є завершальним етапом аналізу, який включає моделювання бази даних і специфікацію API. База даних для онлайн-банкінгу має містити таблиці для зберігання даних користувачів (`Users`), рахунків (`Accounts`), транзакцій (`Transactions`) і журналів операцій (`Logs`). Наприклад, таблиця `Users` може включати поля `user_id`, `username`, `hashed_password` і `email`, тоді як таблиця `Transactions` містить `transaction_id`, `amount`, `sender_id`, `receiver_id`, `date` і `status`. Для забезпечення швидкого доступу до даних використовуються індекси, а для захисту – шифрування полів, таких як `hashed_password`, за допомогою алгоритму bcrypt. Моделювання бази даних

включає створення ER-діаграм (Entity-Relationship), які показують зв’язки між таблицями, наприклад, зв’язок один-до-багатьох між `Users` і `Accounts`.

Специфікація API визначає ендпоінти, методи і формати даних для взаємодії між клієнтом і сервером. Наприклад, ендпоінт `/api/transfers` підтримує POST-запит із JSON-тілом, яке містить `amount`, `receiver_account` і `token`, і повертає відповідь із `status` і `transaction_id`. API документується за допомогою OpenAPI (Swagger), що полегшує тестування і подальшу інтеграцію. Для функцій у реальному часі, таких як сповіщення про транзакції, розглядається використання WebSocket, який забезпечує двосторонній зв’язок. Проектування інфраструктури також включає вибір протоколів безпеки, таких як TLS 1.3 для шифрування запитів, і налаштування брандмауера для захисту від DDoS-атак.

Проблематика проєкту полягає в забезпеченні балансу між функціональністю, безпекою і продуктивністю системи. Високі вимоги до безпеки зумовлюють необхідність впровадження шифрування, двофакторної аутентифікації і захисту від атак, таких як SQL-ін’єкції чи XSS. Продуктивність вимагає оптимізації запитів до бази даних і використання асинхронного програмування в .NET Core. Масштабованість досягається розгортанням серверної частини на кластері Windows Servers, що дозволяє обробляти пікові навантаження. Зручність використання забезпечується мінімалістичним інтерфейсом на Flutter, який скорочує кількість кроків для виконання операцій, таких як перекази чи платежі. Аналіз вимог і вибір технологій дозволяють вирішити ці проблеми, створюючи надійну і ефективну систему онлайн-банкінгу.

Таким чином, аналіз вимог і вибір технологій для розробки мобільного додатку онлайн-банкінгу з серверною частиною є фундаментальним етапом, який визначає успіх проєкту. Функціональні вимоги, такі як перекази і перегляд балансу, поєднуються з нефункціональними, такими як безпека і продуктивність, для створення цілісної системи. Дослідження аналогів, таких як Monobank і Revolut, допомагає визначити найкращі практики, тоді як вибір Flutter і Windows Servers забезпечує кросплатформність, надійність і захист даних. Проектування серверної інфраструктури, включаючи базу даних і API, закладає основу для реалізації

проєкту, вирішуючи проблематику безпеки, масштабованості та зручності використання, що є ключовими для сучасних банківських систем.

2.1 Аналіз вимог до системи онлайн-банкінгу

Аналіз вимог є першим і критично важливим етапом розробки мобільного додатку для онлайн-банкінгу, оскільки він визначає, які функції система має виконувати і які якісні характеристики необхідно забезпечити для задоволення потреб користувачів, банку та регуляторних стандартів. Вимоги поділяються на функціональні, які описують конкретні можливості додатку, і нефункціональні, які стосуються його безпеки, продуктивності, масштабованості та зручності використання. Цей етап передбачає тісну взаємодію із зацікавленими сторонами – представниками банку, потенційними користувачами, аналітиками та розробниками – для формування чіткого бачення системи. Аналіз вимог включає збирання інформації через інтерв'ю, опитування, аналіз ринку та регуляторних документів, а також створення специфікацій, таких як сценарії використання (use cases), діаграми процесів і критерії приймання.

Для аналізу цього проекту, потрібно його порівняти з іншим онлайн-банкінгом для розуміння того, чи підходить проект для буденного використання. UZBUKBANKAPI є локальним банківським API, розробленим на основі ASP.NET Core з використанням Entity Framework Core та SQL Server. Проєкт орієнтований на реалізацію базового функціоналу банківської системи, включаючи автентифікацію користувачів (реєстрація, логін через JWT), управління профілем користувача, картками, балансом і кредитними лімітами. Open Bank Project, натомість, є потужною open-source платформою, призначеною для банків і фінтех-компаній. Побудована на Scala та Play Framework із підтримкою PostgreSQL/MySQL, вона пропонує понад 400 API для доступу до даних рахунків, транзакцій і платежів, відповідаючи стандартам відкритих банківських технологій, зокрема PSD2. Платформа використовує OAuth для автентифікації, забезпечує високу масштабованість і підтримує інтеграцію з різними банківськими системами

через конектори. Для прикладу зробимо таблицю порівняння нашої системи банкінгу з Open Bank Project.

Таблиця 2.1 – Порівняння систем банкінгу

Критерій	UZBUKBANKAPI	Open Bank Project
Тип проєкту	Власний банківський API, розроблений для навчання.	Open-source платформа для банківських API, використовується фінансовими установами.
Технологічний стек	ASP.NET Core, Entity Framework Core, SQL Server, JWT, розгорнуто на IIS.	RESTful JSON API, Scala, Play Framework, PostgreSQL/MySQL, OAuth.
Основний функціонал	Реєстрація/логін, управління профілем, картками, балансом, кредитними лімітами.	Доступ до даних рахунків, транзакцій, платежів, підтримка open banking (PSD2).
Автентифікація	JWT, мапінг телефону на Username.	OAuth, підтримка стандартів open banking для безпечного доступу.
База даних	SQL Server із таблицями Users, Accounts, Cards, Transactions.	PostgreSQL/MySQL, абстракція даних через конектори до банківських систем.
Інтеграція	Локальна, зосереджена на власній базі даних і API-ендпоїнтах.	Інтеграція з різними банківськими системами через конектори, підтримка 400+ API.
Відповідність стандартам	Не орієнтований на open banking, власна реалізація для навчальних цілей.	Відповідає PSD2, підтримує стандарти open banking (OBIE).
Безпека	HTTPS, JWT, хешування паролів (BCrypt), самопідписані сертифікати.	HTTPS, OAuth, шифрування даних, відповідність банківським стандартам безпеки.
Масштабованість	Локальний проєкт, обмежена масштабованість через фокус на навчальну роботу.	Висока масштабованість, підтримує великі банки та 11,000+ розробників.
Переваги	– Простота реалізації. – Гнучкість для навчальних цілей. – Локальний контроль.	- Open-source, безкоштовний доступ. - Широка інтеграція. - Відповідність PSD2.

Недоліки	- Обмежений функціонал. - Відсутність інтеграції з реальними банками. - Локальна масштабованість.	- Висока складність налаштування. - Вимагає досвіду для інтеграції. - Залежність від спільноти.

2.1.1 Функціональні вимоги

Функціональні вимоги визначають набір операцій, які користувачі можуть виконувати через мобільний додаток онлайн-банкінгу, і формують основу його функціональності. Основними функціями є перегляд балансу рахунку, здійснення переказів між рахунками, оплата рахунків, перегляд історії транзакцій і управління профілем користувача. Функція перегляду балансу дозволяє користувачу в реальному часі побачити залишок на своєму рахунку чи кількох рахунках, з підтримкою різних валют (наприклад, UAH, USD, EUR). Наприклад, користувач відкриває додаток, обирає рахунок, і система відображає баланс, можливо, із деталями, такими як доступні кошти чи заблоковані суми. Ця функція вимагає від серверної частини швидкого виконання запиту до бази даних, наприклад, SELECT balance FROM Accounts WHERE account_id = @accountId.

Функція переказів передбачає можливість надсилання коштів на інший рахунок у межах банку або на зовнішній рахунок. Користувач вводить суму, обирає отримувача за номером рахунку, картки чи телефоном (з інтеграцією адресної книги), і підтверджує операцію через одноразовий пароль (OTP), надісланий через SMS чи push-сповіщення, або біометричну аутентифікацію (відбиток пальця чи розпізнавання обличчя). Сервер перевіряє достатність коштів, оновлює баланси обох рахунків і записує транзакцію в базу даних, після чого користувач отримує сповіщення про результат. Наприклад, переказ 500 UAH може викликати SQL-запит: UPDATE Accounts SET balance = balance - 500 WHERE account_id = @sender; UPDATE Accounts SET balance = balance + 500 WHERE account_id = @receiver.

Оплата рахунків включає сплату комунальних послуг, поповнення мобільного зв’язку, погашення кредитів чи оплату онлайн-послуг. Додаток має

підтримувати шаблони для регулярних платежів, наприклад, щомісячної оплати за електроенергію, і автоматичне заповнення даних (номер договору чи QR-код). Перегляд історії транзакцій дозволяє користувачу бачити список операцій із фільтрами за датою, типом (переказ, платіж, поповнення) чи сумою, а також деталі кожної транзакції, такі як дата, сума, статус і коментар. Управління профілем передбачає зміну пароля, налаштування сповіщень (SMS, push, email), додавання нових платіжних карток чи оновлення контактної інформації, наприклад, номера телефону чи email.

Додаткові функції можуть включати аналітику витрат, яка відображає витрати за категоріями (продукти, транспорт, розваги) у вигляді графіків чи таблиць, підтримку мультивалютних рахунків для міжнародних операцій і інтеграцію з платіжними системами, такими як Visa, Mastercard чи локальними системами (наприклад, Простір в Україні). Ці функції вимагають складної серверної логіки, наприклад, обробки обміну валют за ринковими курсами чи інтеграції з API платіжних шлюзів. Усі операції мають бути синхронізовані між клієнтською частиною (Flutter) і серверною (Windows Servers), що забезпечується через REST API. Функціональні вимоги документуються через сценарії використання, які описують дії користувача, реакцію системи та можливі помилки. Наприклад, сценарій для переказу: користувач входить у додаток, обирає функцію переказу, вводить суму та отримувача, підтверджує через OTP, і система повертає повідомлення про успіх або помилку ("Недостатньо коштів").

2.1.2 Нефункціональні вимоги

Нефункціональні вимоги визначають якісні характеристики системи, які забезпечують її ефективність, безпеку, зручність і відповідність стандартам. Безпека є найвищим пріоритетом, оскільки банківські дані, такі як номери карток, паролі чи історія транзакцій, є конфіденційними і потребують захисту від несанкціонованого доступу, витоків чи кібератак. Система має відповідати міжнародним стандартам, таким як PCI DSS, який регулює обробку платіжних карток, і ISO 27001, який визначає вимоги до управління інформаційною безпекою.

Для цього використовується шифрування даних під час передачі через HTTPS із протоколом TLS 1.3, що захищає запити, наприклад, на переказ чи перевірку балансу. У базі даних застосовується прозоре шифрування даних (TDE) у SQL Server, яке забезпечує захист конфіденційних полів, таких як номери карток, навіть у разі фізичного доступу до сервера. Двофакторна аутентифікація (пароль + OTP або біометрія) є обов'язковою для входу та підтвердження транзакцій, а захист від атак, таких як SQL-ін'єкції, XSS (Cross-Site Scripting) чи DDoS, реалізується через параметризовані запити, налаштування брандмауера та обмеження кількості запитів (rate limiting).

Продуктивність системи має забезпечувати швидку обробку запитів, щоб користувачі не відчували затримок. Наприклад, відображення балансу має відбуватися за менш ніж 500 мс для 95% запитів, а обробка переказу – за 1–2 секунди, навіть при піковому навантаженні (тисячі одночасних користувачів). Це досягається оптимізацією запитів до SQL Server, використанням індексів і асинхронного програмування в .NET Core. Масштабованість дозволяє системі підтримувати зростаючу кількість користувачів, наприклад, обробляти 10 000 одночасних транзакцій без втрати продуктивності. Для цього серверна частина розгортається на кластері Windows Servers із балансуванням навантаження через інструменти, такі як Application Request Routing в IIS. Зручність використання забезпечується інтуїтивним інтерфейсом, який мінімізує кількість кроків для виконання операцій: наприклад, переказ має бути здійснений у три кліки, а ключові функції, такі як баланс чи перекази, доступні з головного екрана.

Доступність системи має становити 99.9% (тобто не більше 8 годин простою на рік), що досягається резервуванням серверів, регулярним резервним копіюванням даних і моніторингом через інструменти, такі як Windows Event Viewer. Сумісність передбачає коректну роботу додатку на різних пристроях (iOS і Android, різні розміри екранів) і операційних системах (iOS 14+, Android 10+). Підтримка кількох мов інтерфейсу (українська, англійська) є важливою для міжнародних користувачів. Надійність системи гарантує, що помилки, такі як подвійне списання коштів, виключені завдяки транзакційній цілісності SQL Server.

Нефункціональні вимоги формуються через чіткі метрики та критерії приймання, наприклад, "усі запити до API захищені TLS 1.3" або "середній час відповіді сервера не перевищує 500 мс при 1000 одночасних запитах". Ці вимоги є основою для вибору технологій, таких як Flutter для зручного інтерфейсу та Windows Servers для безпеки і масштабованості.

2.2 Дослідження існуючих рішень

Дослідження існуючих рішень є важливим етапом, який дозволяє оцінити, як аналогічні задачі вирішуються в інших банківських системах, визначити найкращі практики, уникнути типових помилок і адаптувати перевірені підходи до проєкту. Аналіз охоплює як клієнтську частину (мобільні додатки), так і серверну (платформи для обробки транзакцій), з фокусом на їхню відповідність функціональним і нефункціональним вимогам системи онлайн-банкінгу. Це дослідження допомагає обґрунтувати вибір Flutter і Windows Servers, а також визначити стандарти якості для додатку.

2.2.1 Аналіз аналогів мобільних додатків

Аналіз аналогів мобільних додатків передбачає вивчення провідних банківських додатків, таких як Monobank (Україна), Revolut (Велика Британія), N26 (Німеччина) і Starling Bank (Велика Британія), які є еталонами з точки зору функціональності, користувацького досвіду (UX) і безпеки. Monobank вирізняється мінімалістичним дизайном і швидким доступом до основних функцій. Користувач може здійснити переказ, оплатити комунальні послуги чи поповнити мобільний рахунок у кілька кліків, що відповідає вимогам до зручності. Додаток пропонує аналітику витрат у вигляді графіків, підтримку шаблонів для регулярних платежів і push-сповіщення про транзакції, що є прикладом для реалізації аналогічних функцій у проєкті. Безпека забезпечується двофакторною аутентифікацією (пароль + OTP через SMS або push) і шифруванням даних, що відповідає стандартам PCI DSS і звичайним вимогам до захисту.

Revolut робить акцент на міжнародних операціях, підтримуючи мультивалютні рахунки, обмін валют за ринковими курсами і функції інвестування, такі як купівля акцій чи криптовалют. Ці можливості вимагають складної серверної логіки для обробки транзакцій у реальному часі, що є корисним прикладом для цього проєкту, якщо планується підтримка кількох валют чи інтеграція з фінансовими ринками. Revolut використовує біометричну аутентифікацію (Face ID, відбиток пальця) і дозволяє налаштовувати сповіщення, що підвищує зручність і безпеку. N26 фокусується на прозорості, надаючи детальну історію транзакцій із фільтрами за датою, категорією чи сумою, а також сповіщення в реальному часі через WebSocket. Додаток має сучасний інтерфейс із мінімальною кількістю кроків для виконання операцій, що відповідає вимогам до зручності. Starling Bank пропонує функції бюджетування, автоматичну категоризацію витрат і інтеграцію з платіжними системами, такими як Google Pay і Apple Pay, що є прикладом для реалізації додаткових функцій у додатку.

Аналіз цих додатків показує, що успішні рішення поєднують інтуїтивний інтерфейс, швидку обробку запитів і високий рівень безпеки. Наприклад, Monobank і N26 використовують мінімалістичний дизайн, де ключові функції (баланс, перекази, платежі) доступні з головного екрана, що скорочує час виконання операцій. Усі додатки підтримують двофакторну аутентифікацію і шифрування, що є стандартом для банківських систем. Цей додаток має пропонувати аналогічний рівень простоти, підтримувати базові функції (перекази, платежі, історія транзакцій) і передбачати можливість розширення, наприклад, додавання аналітики витрат, мультивалютних рахунків чи інтеграції з платіжними шлюзами. Flutter є ідеальним вибором для реалізації такого інтерфейсу, оскільки його бібліотека віджетів дозволяє створювати адаптивні та інтерактивні дизайни, подібні до Monobank чи N26, а підтримка плагінів забезпечує біометричну аутентифікацію і сповіщення.

2.2.2 Серверні рішення для банкінгу

Дослідження серверних рішень зосереджується на аналізі платформ, які використовуються для обробки банківських транзакцій, із акцентом на їхню безпеку, продуктивність, масштабованість і сумісність із корпоративними стандартами. Windows Servers є популярним вибором для банківських систем завдяки своїй надійності, інтеграції з екосистемою Microsoft і підтримці потужних інструментів, таких як Internet Information Services (IIS), .NET Core, SQL Server і Active Directory. IIS виступає вебсервером для розгортання REST API, забезпечуючи обробку тисяч одночасних запитів із мінімальними затримками. Наприклад, IIS може обробляти пікові навантаження під час масових платежів, таких як сплата комунальних послуг наприкінці місяця, завдяки пулу потоків і механізмам кешування. .NET Core дозволяє швидко розробити API для транзакцій, наприклад, ендпоінт /api/transfers, який перевіряє баланс, оновлює рахунки і записує операцію в базу даних. SQL Server є стандартом для зберігання фінансових даних завдяки транзакційній цілісності, яка запобігає помилкам, таким як подвійне списання коштів, і підтримці шифрування через TDE. Active Directory забезпечує управління правами доступу, дозволяючи обмежити доступ до адміністративних функцій, таких як перегляд аналітики чи редагування транзакцій, лише для авторизованих користувачів.

Альтернативою Windows Servers є Linux-based сервери, які використовуються у фінтех-компаніях, таких як Revolut чи Chime, завдяки їхній гнучкості, відкритому коду та нижчій вартості ліцензування. Linux-сервери часто поєднуються з вебсерверами, такими як Nginx чи Apache, і базами даних, такими як PostgreSQL або MongoDB. PostgreSQL підтримує транзакційну цілісність і є альтернативою SQL Server, тоді як MongoDB використовується для гнучкого управління неструктурованими даними, наприклад, логами чи аналітичними звітами. Однак Linux вимагає додаткових зусиль для інтеграції з корпоративними рішеннями, такими як Active Directory, і може бути менш зручним для банків, які використовують екосистему Microsoft. Крім того, Windows Servers має

сертифікацію безпеки, наприклад, FIPS 140-2, що є важливим для відповідності регуляторним вимогам банківської сфери.

Порівняння баз даних показує, що SQL Server є оптимальним вибором для проєкту завдяки його інтеграції з Windows Servers, підтримці транзакційної цілісності і вбудованим механізмам шифрування. Наприклад, SQL Server може обробляти складні транзакції, такі як переказ із конвертацією валют, із гарантією цілісності даних. PostgreSQL, хоча й є потужною альтернативою, менш інтегрована з Windows-екосистемою, а MongoDB більше підходить для неструктурованих даних, ніж для фінансових транзакцій. Аналіз серверних рішень підтверджує, що Windows Servers є найкращим вибором для цього проєкту, оскільки він відповідає вимогам безпеки (шифрування, аутентифікація), продуктивності (швидка обробка запитів) і масштабованості (кластерне розгортання), а також забезпечує сумісність із корпоративними стандартами, що є критично важливим для банківських систем.

2.3 Проектування серверної інфраструктури

Проектування серверної інфраструктури є завершальним етапом аналізу, який створює надійну основу для обробки запитів, зберігання даних і захисту інформації в системі онлайн-банкінгу. Воно включає моделювання бази даних для організації фінансових даних і специфікацію API для інтеграції клієнтської частини (мобільний додаток на Flutter) із серверною (Windows Servers). Цей етап є ключовим для проєкту, оскільки він забезпечує ефективну взаємодію між компонентами системи, відповідність вимогам безпеки і можливість масштабування в майбутньому.

2.3.1 Моделювання бази даних

Моделювання бази даних передбачає створення структури для зберігання даних, необхідних для роботи системи онлайн-банкінгу, із забезпеченням їхньої цілісності, безпеки та швидкого доступу. База даних розгортається на SQL Server, який є стандартом для Windows-систем завдяки транзакційній цілісності, підтримці

шифрування і масштабованості. Основними таблицями є Users (користувачі), Accounts (рахунки), Transactions (транзакції) і Logs (журнали операцій).

Таблиця Users містить поля user_id (унікальний ідентифікатор, первинний ключ), username (логін), hashed_password (зашифрований пароль за допомогою алгоритму bcrypt), email, phone і created_at (дата створення профілю). Паролі шифруються для захисту від витоків, а поля email і phone використовуються для надсилання OTP чи сповіщень. Таблиця Accounts включає account_id (первинний ключ), user_id (зовнішній ключ, зв’язок із Users), balance (дробове число, наприклад, 1000.50), currency (UAH, USD, EUR) і account_type (наприклад, дебетовий, кредитний). Таблиця Transactions містить transaction_id (первинний ключ), amount, sender_account (зовнішній ключ до Accounts), receiver_account, date (дата і час), status (completed, failed, pending) і description (коментар, наприклад, "Оплата за електроенергію"). Таблиця Logs зберігає журнали операцій для моніторингу безпеки, включаючи поля log_id, user_id, action (наприклад, login, transfer), timestamp і details (додаткова інформація, наприклад, IP-адреса).

Для оптимізації швидкості доступу до даних створюються індекси на часто використовуваних полях, таких як user_id, account_id і date. Наприклад, індекс на user_id у таблиці Accounts прискорює запит SELECT * FROM Accounts WHERE user_id = @userId для відображення всіх рахунків користувача. Для захисту даних застосовується прозоре шифрування даних (TDE) у SQL Server, яке шифрує всю базу даних, і шифрування окремих полів, таких як hashed_password, за допомогою bcrypt. Моделювання включає створення ER-діаграм (Entity-Relationship), які показують зв’язки між таблицями. Наприклад, зв’язок один-до-багатьох між Users і Accounts відображає, що один користувач може мати кілька рахунків, а зв’язок між Accounts і Transactions показує, що кожен рахунок може бути відправником або отримувачем у багатьох транзакціях.

Додаткові таблиці можуть включати Cards (для зберігання даних платіжних карток, таких як номер, термін дії і CVV, у зашифрованому вигляді) і Notifications (для збереження історії сповіщень). Для підтримки мультивалютних операцій може бути створена таблиця ExchangeRates із полями currency_pair (наприклад,

USD/UAH), rate і updated_at. Така структура бази даних відповідає функціональним вимогам (перекази, платежі, історія транзакцій) і нефункціональним вимогам (безпека, швидкодія), забезпечуючи надійне зберігання даних і їхню цілісність.

2.3.2 API для інтеграції

Специфікація API визначає набір ендпоінтів, методи, формати даних і механізми безпеки для взаємодії між мобільним додатком на Flutter і серверною частиною на Windows Servers. API реалізується через REST, який використовує HTTP-запити для виконання операцій, таких як перевірка балансу, здійснення переказів чи отримання історії транзакцій. REST є безстанним, що спрощує масштабування системи, оскільки сервер не зберігає стан між запитами. API розгортається на IIS, який забезпечує обробку запитів із високою продуктивністю і захистом від атак, таких як DDoS.

Основні ендпоінти включають:

– *GET /api/balance*. Повертає баланс рахунку у форматі JSON ({"account_id": 123, "balance": 1000.50, "currency": "UAH"}). Запит вимагає JWT-токен у заголовку Authorization для аутентифікації. Сервер виконує SQL-запит SELECT balance FROM Accounts WHERE account_id = @accountId і повертає результат;

– *POST /api/transfers*. Ініціює переказ коштів. Запит містить JSON-тіло ({"amount": 500, "receiver_account": 456, "token": "jwt_token"}) і повертає відповідь ({"status": "success", "transaction_id": 789}). Сервер перевіряє JWT-токен, достатність коштів і виконує транзакцію через SQL-запити: UPDATE Accounts SET balance = balance - @amount WHERE account_id = @sender_account; UPDATE Accounts SET balance = balance + @amount WHERE account_id = @receiver_account;

– *GET /api/transactions*. Повертає історію транзакцій із фільтрами за датою, типом чи сумою (наприклад, ?date_from=2025-01-01&date_to=2025-05-13). Відповідь містить масив JSON ([{"transaction_id": 789, "amount": 500, "date": "2025-05-13T10:00:00", "status": "completed"}]);

– *PUT /api/users*. Оновлює профіль користувача, наприклад, email чи телефон. Запит містить JSON-тіло ({"email": "new@example.com"}) і повертає статус операції.

API документується за допомогою OpenAPI (Swagger), що створює інтерактивну документацію з описом ендпоінтів, параметрів і прикладів запитів/відповідей. Це полегшує тестування через інструменти, такі як Postman, і подальшу інтеграцію з Flutter-додатком. Для функцій у реальному часі, таких як push-сповіщення про завершення транзакції чи виявлення підозрілої активності, використовується WebSocket, який забезпечує двосторонній зв’язок. Наприклад, після успішного переказу сервер надсилає повідомлення через WebSocket ({"type": "transfer", "message": "Transfer of 500 UAH completed"}), яке відображається як сповіщення в додатку.

Безпека API є критично важливою і забезпечується кількома рівнями захисту. HTTPS із TLS 1.3 шифрує всі запити і відповіді, захищаючи дані, такі як номери карток чи суми транзакцій. JWT-аутентифікація перевіряє ідентичність користувача: кожен запит містить токен, який сервер валідує перед обробкою. Захист від SQL-ін’єкцій реалізується через параметризовані запити, наприклад, SELECT * FROM Transactions WHERE user_id = @userId. Захист від XSS і CSRF досягається налаштуванням заголовків безпеки в IIS, таких як X-Content-Type-Options і Content-Security-Policy. Для запобігання DDoS-атакам використовується обмеження кількості запитів (rate limiting) і брандмауер Windows. Усі операції логуються в таблиці Logs для аналізу підозрілої активності, наприклад, кількох невдалих спроб входу.

Проблематика проєкту полягає в необхідності забезпечити баланс між функціональністю, безпекою, продуктивністю і зручністю використання. Функціональні вимоги, такі як підтримка переказів, платежів і аналітики, вимагають складної серверної логіки для обробки транзакцій і синхронізації даних. Нефункціональні вимоги, зокрема безпека, зумовлюють впровадження шифрування, двофакторної аутентифікації і захисту від атак. Продуктивність досягається оптимізацією запитів до SQL Server і асинхронним програмуванням у

.NET Core, що дозволяє обробляти тисячі запитів за секунду. Масштабованість забезпечується кластерним розгортанням Windows Servers, а зручність – мінімалістичним інтерфейсом на Flutter. Дослідження аналогів, таких як Monobank, Revolut і N26, дозволяє адаптувати найкращі практики, тоді як вибір Flutter і Windows Servers відповідає вимогам кросплатформності, безпеки і корпоративної інтеграції. Проектування серверної інфраструктури, включаючи моделювання бази даних і специфікацію API, створює міцну основу для реалізації проєкту, вирішуючи виклики безпеки, продуктивності та користувацького досвіду, що є критично важливим для сучасних банківських систем.

3 ПРОЕКТУВАННЯ ТА РЕАЛІЗАЦІЯ ПРОЕКТУ

Проект банківської системи включає RESTful API (UZBUKBANKAPI) та мобільний додаток, що імітує Monobank, для управління користувачами, рахунками, картками, транзакціями та профілем із безпечною автентифікацією та зручним інтерфейсом. Мета — безпечні фінансові операції та сучасний користувацький досвід.

Реєстрація та логін за номером телефону (Username), керування профілем через GET/PUT /api/users/profile, створення рахунків і карток (Visa/Mastercard), обробка транзакцій, перекази, історія транзакцій, кешбек. Мобільний додаток має екрани: LoginScreen, RegisterScreen, HomeScreen, TransferScreen, HistoryScreen, ProfileScreen, CashbackScreen. Безпека: JWT-автентифікація, хешування паролів (BCrypt.Net-Next), валідація ([StringLength], [EmailAddress], [Range]). Масштабованість: SQL Server, надійність: транзакції. Додаток використовує шрифт Rutenia2008VasylChebanyk, градієнтний фон, анімації (FadeTransition, ScaleTransition, SlideTransition).

Сервер: ASP.NET Core 6.0, Entity Framework Core, SQL Server Express (BankDb), BCrypt.Net-Next, Microsoft.AspNetCore.Authentication.JwtBearer, розгортання на IIS. Моделі: User (UserId, Username, PasswordHash, FirstName, SecondName, Email, Age, Gender), Account (AccountId, UserId), Card (CardId, AccountId, CardNumber, CardType, Balance, CreditLimit, CreditInterestRate, CreatedDate), Transaction (TransactionId, AccountId, CardId, Amount, TransactionType, TransactionDate). База: таблиці Users, Accounts, Cards, Transactions із зовнішніми ключами, типи даних DATETIME2, NVARCHAR, DECIMAL(18,2). Додаток: Flutter, інтеграція REST API, використання Material design, HttpOverrides для обходу SSL у тестах.

API: AuthController (POST /api/auth/register, POST /api/auth/login), UsersController (GET/PUT /api/users/profile), картки (POST /api/cards, GET /api/cards/balance), транзакції (POST /api/transfers, GET /api/transactions). Сервер реалізовано через шаблон ASP.NET Core Web API з залежностями

Microsoft.EntityFrameworkCore.SqlServer, BCrypt.Net-Next, Swashbuckle.AspNetCore. Налаштування в appsettings.json (BankDb, JWT), Program.cs (DI, ILogger, Swagger). База створена через DropTables.sql, CreateTables.sql (DATETIME2), міграції EF Core (dotnet ef migrations add InitialCreate, dotnet ef database update). Виправлено помилки: BCrypt, Card.Transactions, SqlNullValueException, DATETIME (Nullable поля, логування).

Додаток: LoginScreen, RegisterScreen (авторизація, анімації), HomeScreen (картки, RefreshIndicator), TransferScreen (перекази), HistoryScreen (фільтрація), ProfileScreen (редагування). Дизайн: градієнтний фон (BoxConstraints), шрифт через pubspec.yaml, анімації. Виправлено: конфлікт назв Card, скидання профілю, фон, AppBar, кнопки "назад".

Розгортання сервера: IIS (C:\inetpub\wwwroot\BankApi), web.config (stdoutLogEnabled="true"), права для IIS_IUSRS, IIS APPPOOL\BankApiPool, перезапуск (iisreset). Тестування: Postman, Swagger UI (автентифікація, профіль, таблиці). Додаток: перевірка авторизації, профілю, карток, транзакцій, дизайну (фон, анімації, шрифт). Помилки API (HandshakeException) вирішено через HttpOverrides.

3.1 Розробка серверної частини

Серверна частина мобільного додатку для онлайн-банкінгу була розроблена з використанням технологій ASP.NET Core 6.0, Entity Framework Core для роботи з базою даних SQL Server Express, а також розгорнута на Windows Server з використанням Internet Information Services (IIS). Основна мета серверної частини полягала в забезпеченні безпечного та ефективного API для обробки запитів від мобільного додатку, включаючи автентифікацію, управління профілем користувача, операції з картками та транзакціями. Процес розробки включав налаштування серверного середовища, створення API з бізнес-логікою та забезпечення безпеки.

3.1.1 Налаштування Windows Servers

Налаштування серверного середовища було першим етапом розробки, спрямованим на створення стабільної платформи для розгортання API. Використовувався Windows Server 2019 як операційна система, а для хостингу веб-додатку — IIS версії 10.0.

Встановлення необхідного програмного забезпечення

– *Встановлення .NET 6.0 Hosting Bundle*. Для підтримки ASP.NET Core на IIS було встановлено .NET 6.0 Hosting Bundle, який включає runtime і модуль ASP.NET Core для IIS. Це забезпечило коректну обробку запитів до API;

– *Налаштування SQL Server Express*. SQL Server Express 2019 було встановлено для зберігання даних користувачів, рахунків, карток і транзакцій. Створено базу даних BankDb;

– *Встановлення IIS*. Internet Information Services було активовано через Server Manager, з увімкненням модулів для обробки статичних файлів, динамічного вмісту та підтримки .NET.

Конфігурація IIS

– *Створення пулу застосунків*. Створено пул застосунків BankApiPool із налаштуванням "No Managed Code" для ASP.NET Core. Це дозволило оптимізувати продуктивність, оскільки ASP.NET Core обробляє запити самостійно.

– *Налаштування сайту*. Створено сайт у IIS, який указує на директорію C:\inetpub\wwwroot\BankApi. Фізичний шлях містив опубліковані файли API, включаючи BankApi.dll і web.config.

– *Файл web.config*. Налаштовано файл web.config для запуску додатку через dotnet:

<aspNetCore processPath="dotnet" arguments=".\BankApi.dll" stdoutLogEnabled="true" stdoutLogFile=".\logs\stdout" hostingModel="inprocess" />

Параметр stdoutLogEnabled="true" увімкнув логування для діагностики помилок.

– *Права доступу*. Надання прав Read & Execute групі IIS_IUSRS для директорії C:\inetpub\wwwroot\BankApi забезпечило доступ IIS до файлів додатку;

– *SSL сертифікат*. Для тестування було створено самопідписаний сертифікат за допомогою PowerShell:

New-SelfSignedCertificate -DnsName "leaf-pc" -CertStoreLocation "cert:\LocalMachine\My"

Цей сертифікат прив’язали до HTTPS-порту 443 у IIS, щоб забезпечити безпечне з’єднання.

Проблеми та їх вирішення:

– *Помилка SSL*. Під час тестування виникла помилка The certificate chain was issued by an authority that is not trusted. Для її вирішення до рядка підключення додано TrustServerCertificate=True, що дозволило ігнорувати самопідписаний сертифікат у тестовому середовищі;

– *Проблеми з правами доступу*. Спочатку IIS не міг отримати доступ до файлів у C:\inetpub\wwwroot\BankApi. Надання прав IIS_IUSRS вирішило цю проблему;

– *Логування помилок*. Логи IIS у C:\inetpub\wwwroot\BankApi\logs допомогли діагностувати проблеми з розгортанням, зокрема помилки підключення до бази даних.

Результати

Налаштування Windows Server і IIS забезпечило стабільне середовище для хостингу API. Використання SQL Server Express дозволило ефективно керувати даними, а самопідписаний сертифікат забезпечив базовий рівень безпеки для тестового середовища.

3.1.2 Реалізація API та бізнес-логіки

API було реалізовано з використанням ASP.NET Core 6.0, Entity Framework Core для взаємодії з базою даних, і архітектури REST для забезпечення зручної

інтеграції з мобільним додатком. Основна бізнес-логіка включала автентифікацію, управління профілем користувача, операції з картками та історією транзакцій.

Структура проєкту

Проєкт мав назву UZBUKBANKAPI і складався з таких основних компонентів:

– *Моделі (Models)*. Визначали структури даних (User, Account, Card, Transaction, RegisterModel, LoginModel, UpdateProfileModel);

– *Контролери (Controllers)*. Обробляли HTTP-запити (AuthController, UsersController, HistoryController);

– *Контекст даних (BankDbContext)*. Налаштовував зв'язки між таблицями та мапінг на базу даних;

– *Утиліти (Utilities)*. Містять допоміжні класи, наприклад, CardNumberGenerator для створення номерів карток.

Основні ендпоїнти:

а) *POST /api/auth/register.* Реєструє нового користувача з номером телефону (Phone) і паролем;

1) Створює користувача, рахунок і першу Visa-картку з початковим балансом 100.00;

2) Використовує транзакції для забезпечення атомарності:

```
using var transaction = await
_context.Database.BeginTransactionAsync();
_context.Users.Add(user);
await _context.SaveChangesAsync();
_context.Accounts.Add(account);
await _context.SaveChangesAsync();
_context.Cards.Add(card);
await _context.SaveChangesAsync();
await transaction.CommitAsync();
```

б) POST /api/auth/login.

1) Виконує автентифікацію за номером телефону (Username) і паролем;

2) Видає JWT-токен із терміном дії 30 хвилин.

в) *GET /api/Users/profile.* Повертає профіль авторизованого користувача (дані: Username, FirstName, SecondName, Email, Age, Gender);

г) *PUT /api/Users/profile.* Оновлює профіль користувача на основі UpdateProfileModel з валідацією полів;

д) *GET /api/history/{cardId}*. Повертає історію транзакцій для вказаної картки, доступної лише авторизованому користувачу;

База даних BankDb містила чотири основні таблиці:

– *Users*. Зберігає дані користувачів (UserId, Username, PasswordHash, FirstName, SecondName, Email, Age, Gender);

– *Accounts*. Пов’язує користувачів із рахунками (AccountId, UserId);

– *Cards*. Містить дані карток (CardId, AccountId, CardNumber, CardType, Balance, CreditLimit, CreditInterestRate, CreatedDate);

– *Transactions*. Зберігає транзакції (TransactionId, AccountId, CardId, Amount, TransactionType, TransactionDate).

Зв’язки між таблицями налаштовано в BankDbContext:

```
modelBuilder.Entity<User>()
    .HasMany(u => u.Accounts)
    .WithOne(a => a.User)
    .HasForeignKey(a => a.UserId);
modelBuilder.Entity<Account>()
    .HasMany(a => a.Cards)
    .WithOne(c => c.Account)
    .HasForeignKey(c => c.AccountId);
modelBuilder.Entity<Card>()
    .HasMany(c => c.Transactions)
    .WithOne(t => t.Card)
    .HasForeignKey(t => t.CardId)
```

```
        .IsRequired(false);
```

Проблеми та їх вирішення

– *Помилка з Card.Transactions*. Відсутність навігаційної властивості Transactions у моделі Card викликала помилку. Додано List<Transaction> Transactions до Card.cs і налаштовано зв'язок у BankDbContext;

– *Помилка з DATETIME*. Під час реєстрації виникла помилка The conversion of a datetime2 data type to a datetime data type resulted in an out-of-range value. Вирішено зміною типу CreatedDate і TransactionDate на DATETIME2 у SQL-скриптах:

```
CreatedDate DATETIME2 NOT NULL DEFAULT GETUTCDATE(),
```

Також у коді явно встановлено CreatedDate = DateTime.UtcNow.

– *Помилка SqlNullValueException при логіні.* Виникала через NULL у необов'язкових полях (FirstName, SecondName, Email, Age, Gender). Вирішено оновленням моделі User з використанням Nullable типів:

```
public string? FirstName { get; set; }
public int? Age { get; set; }
```

Результати

API забезпечує повний набір функцій для автентифікації, управління профілем і транзакціями. Використання Entity Framework Core спростило роботу з базою даних, а транзакції гарантували цілісність даних. Вирішення проблем із типами даних і навігаційними властивостями забезпечило стабільність.

3.1.3 Забезпечення безпеки серверної частини

Безпека серверної частини була ключовим аспектом розробки, оскільки онлайн-банкінг потребує захисту даних користувачів і транзакцій.

Автентифікація та авторизація

JWT-токени. Для автентифікації використано JSON Web Tokens. Після успішного логіну ендпоїнт POST /api/auth/login видає токен із UserId і Username:

```
  var claims = new[]
{
```

```
    new Claim(ClaimTypes.NameIdentifier,
user.UserId.ToString()),
    new Claim(ClaimTypes.Name, user.Username)
};
```

Токен перевіряється для всіх захищених ендпоїнтів через [Authorize].

Налаштування JWT. У Program.cs налаштовано валідацію токенів:

```
  builder.Services.AddAuthentication(JwtBearerDefaults.Authentic
  ationScheme)
      .AddJwtBearer(options =>
      {
          options.TokenValidationParameters = new
  TokenValidationParameters
          {
              ValidateIssuer = true,
              ValidateAudience = true,
              ValidateLifetime = true,
              ValidateIssuerSigningKey = true,
              ValidIssuer = builder.Configuration["Jwt:Issuer"],
              ValidAudience =
  builder.Configuration["Jwt:Audience"],
              IssuerSigningKey = new SymmetricSecurityKey(

  Encoding.UTF8.GetBytes(builder.Configuration["Jwt:Key"]))
          };
      });
```

Хешування паролів. Використано бібліотеку BCrypt.Net-Next для безпечного хешування паролів:

```
    PasswordHash =
BCrypt.Net.BCrypt.HashPassword(model.Password);
```

– Під час логіну пароль перевіряється:
```
BCrypt.Net.BCrypt.Verify(login.Password, user.PasswordHash);
```

Захист бази даних

– *Унікальні ключі*. Колонки Username і CardNumber позначено як UNIQUE для запобігання дублюванню;

– *Зовнішні ключі*. Налаштовано зв'язки між таблицями для забезпечення цілісності даних;

– *Транзакції*. Використання транзакцій у POST /api/auth/register гарантувало атомарність операцій.

HTTPS і SSL

– Усі запити до API обробляються через HTTPS завдяки самопідписаному сертифікату в IIS. У тестовому середовищі використано TrustServerCertificate=True для обходу проблем із сертифікатом.

Валідація вхідних даних

– У RegisterModel і UpdateProfileModel використано анотації валідації:
```
public class UpdateProfileModel
{
    [StringLength(50)]
    public string? FirstName { get; set; }
    [EmailAddress]
    public string? Email { get; set; }
    [Range(18, 120)]
    public int? Age { get; set; }
}
```
– Перевірка на null і дублікати в AuthController:
```
if (await _context.Users.AnyAsync(u => u.Username ==
model.Phone))
{
    return BadRequest("User with this phone already exists.");
}
```

Проблеми та їх вирішення

– *Помилки автентифікації*. Некоректні дані в Users (наприклад, NULL у PasswordHash) викликали SqlNullValueException. Вирішено оновленням моделі та очищенням даних.

– *Логування для діагностики*. Додано ILogger до AuthController для відстеження помилок:

```
_logger.LogError(ex, "Error during login: {Message}", ex.Message);
```

Результати

Заходи безпеки, включаючи JWT, хешування паролів, HTTPS і валідацію, забезпечили захист даних і доступу. Логування допомогло швидко діагностувати проблеми.

3.2 Розробка мобільного додатку на Flutter

Мобільний додаток для онлайн-банкінгу був розроблений з використанням фреймворку Flutter, який забезпечує кросплатформну розробку для iOS та Android з єдиною кодовою базою. Основна мета полягала в створенні інтуїтивно зрозумілого, безпечного та естетично привабливого додатку, що інтегрується з серверною частиною через API. Розробка включала створення інтерфейсу користувача та забезпечення коректної взаємодії з сервером.

3.2.1 Інтерфейс користувача

Інтерфейс користувача (UI) додатку розроблявся з акцентом на мінімалізм, сучасний дизайн та зручність взаємодії. Основні екрани додатку включають:

– *LoginScreen.* Екран авторизації з полями для введення імені користувача та пароля;

– *RegisterScreen*. Екран реєстрації з полями для номера телефону та пароля;

– *HomeScreen*. Головний екран, що відображає список карток користувача, баланс, кредитний ліміт та доступні дії (кешбек, переказ, історія транзакцій, додавання картки);

– TransferScreen. Екран для здійснення переказів між картками

– HistoryScreen. Екран історії транзакцій для вибраної картки;

– ProfileScreen. Екран для перегляду та редагування профілю користувача;

– CashbackScreen. Екран-заглушка для майбутньої реалізації кешбек-функціоналу.

Дизайн та стилізація

– *Градієнтний фон.* Для всіх екранів застосовано градієнтний фон (від чорного до темно-сірого), що забезпечує сучасний вигляд та відповідає бренду UZBUKBANK. Для цього використано віджет Container із LinearGradient, а для заповнення всієї висоти екрану додано constraints: BoxConstraints(minHeight: MediaQuery.of(context).size.height).

– *Кастомний шрифт.* У проєкті використано шрифт Rutenia2008VasylChebanyk.ttf, розташований у папці fonts/Rutenia2008. Шрифт підключено через pubspec.yaml та застосовано глобально через TextTheme у main.dart.

У main.dart визначено стилі для різних типів тексту (headlineLarge, bodyLarge, тощо), що забезпечило консистентність шрифтів у всьому додатку.

– *Анімації.* Для підвищення інтерактивності додані анімації за допомогою AnimationController:

1. FadeTransition для плавної появи тексту та віджетів;
2. ScaleTransition для масштабування кнопок і полів вводу;
3. SlideTransition для анімації списків транзакцій і карток. Наприклад, на HomeScreen картки з’являються зі зміщенням знизу вгору.

– *Стилізація елементів.* Кнопки та картки мають закруглені кути (BorderRadius.circular(12)), тіні (BoxShadow) та градієнтне заповнення. Поля вводу (TextField) стилізовані з fillColor: Colors.grey[850] для гармонії з темним фоном.

Проблематика та рішення

Проблема зі шрифтом:

1. Потрібно було застосувати кастомний шрифт Rutenia2008 до всіх текстових елементів;
2. Рішення: Шрифт підключено через pubspec.yaml, а в main.dart налаштовано TextTheme для різних стилів тексту. У коді використано Theme.of(context).textTheme для забезпечення консистентності.

Оптимізація UI

- Використано SingleChildScrollView із physics. AlwaysScrollableScrollPhysics() на HomeScreen для підтримки оновлення даних через RefreshIndicator.
- Для зменшення повторного коду стилі кнопок і полів вводу уніфіковано через ElevatedButton.styleFrom і InputDecoration.
- Анімації оптимізовано з використанням SingleTickerProviderStateMixin, щоб уникнути надмірного споживання ресурсів.

3.2.2 Інтеграція з серверною частиною

Опис інтеграції

Мобільний додаток інтегровано з серверною частиною через REST API, що забезпечує авторизацію, управління картками, перекази, історію транзакцій і редагування профілю. Для взаємодії з API використано пакет http у Dart. Основні ендпоінти:

- POST /api/Auth/login: Авторизація користувача;
- POST /api/Auth/register: Реєстрація користувача;
- GET /api/Cards/balance: Отримання балансу та даних карток;
- POST /api/Cards/create: Створення нової картки;
- POST /api/Transactions/transfer: Виконання переказу;
- GET /api/Transactions/history/{cardId}: Отримання історії транзакцій;
- GET /api/Users/profile: Отримання даних профілю;
- PUT /api/Users/profile: Оновлення профілю.

Реалізація

ApiService. Створено клас ApiService для централізованого управління HTTP-запитами. Наприклад:

```
class ApiService {
  final String baseUrl = 'https://localhost:5001/api';
  String? token;
  Future<User> login(String username, String password) async {
    final response = await http.post(
      Uri.parse('$baseUrl/Auth/login'),
      headers: {'Content-Type': 'application/json'},
      body: jsonEncode({'username': username, 'password': password}),
    );
  }
}
```

– *Токен авторизації*. Після успішного логіну токен зберігається в ApiService і додається до заголовків усіх подальших запитів (Authorization: Bearer <token>);

– *Обробка даних*. Дані з API (наприклад, баланс карток, профіль) парсяться в об’єкти Dart (User, Cards, Profile, Transaction) за допомогою методів fromJson.

Проблематика та рішення

– Проблема з HandshakeException:

Під час запитів до https://localhost:5001 виникала помилка HandshakeException через самопідписаний сертифікат сервера.

Рішення: У main.dart додано HttpOverrides для обходу перевірки сертифікатів на етапі розробки:

```
class MyHttpOverrides extends HttpOverrides {
  @override
  HttpClient createHttpClient(SecurityContext? context) {
```

```
        return super.createHttpClient(context)
          ..badCertificateCallback = (X509Certificate cert,
      String host, int port) {
            print('Bypassing certificate verification for
      $host:$port');
            return true;
          };
      }
}
```

Це тимчасове рішення для тестування; для продакшену рекомендовано використовувати валідний SSL-сертифікат.

– Оновлення балансу карток:

Після виконання переказу баланс карток не оновлювався автоматично.

Рішення: У TransferScreen після успішного переказу викликається ApiService().getBalance(), а результат повертається в HomeScreen через Navigator.pop(context, balanceResponse). У HomeScreen оновлюється стан:

```
      final result = await Navigator.push(context,
      MaterialPageRoute(...));
      if (result != null && result is List<dynamic>) {
        setState(() {
          user = User.fromJson(user.username, result,
      user.profile);
        });
}
```

Додано RefreshIndicator на HomeScreen для ручного оновлення даних через свайп вниз.

– Проблема з профілем:

Після оновлення профілю через PUT /api/Users/profile дані скидалися при повторному вході на ProfileScreen.

Рішення: У ProfileScreen додано виклик ApiService().getProfile() у initState для завантаження актуальних даних:

```
  void _loadProfile() async {
    try {
      final profile = await ApiService().getProfile();
      setState(() {
        _firstNameController.text = profile.firstName ??
  '';
      });
    } catch (e) {

  ScaffoldMessenger.of(context).showSnackBar(SnackBar(conte
  nt: Text('Помилка: $e')));
    }
}
```

У LoginScreen додано виклик getProfile() після логіну, щоб зберегти профіль у User.

– Конфлікт імен:

Виник конфлікт між класом Card у моделі та material.Card у Flutter.

Рішення: Перейменовано клас у моделі на Cards, що усунуло конфлікт і покращило читабельність.

– Оптимізація інтеграції

Уніфіковано обробку помилок у ApiService з відображенням повідомлень через ScaffoldMessenger:

```
ScaffoldMessenger.of(context).showSnackBar(SnackBar(content
: Text('Помилка: $e')));
```

Використано асинхронне програмування (async/await) для всіх мережевих операцій, що забезпечило плавність UI.

Для зменшення кількості запитів дані профілю кешуються в об’єкті User після логіну.

3.3 Тестування та оптимізація

Тестування та оптимізація банківської системи, що включає RESTful API (UZBUKBANKAPI) та мобільний додаток, спрямовані на забезпечення стабільності, продуктивності та зручності використання. Мета — виявлення помилок, підвищення швидкості роботи та оптимізація ресурсів для масштабування.

Тестування охоплює серверну частину (API-ендпоїнти, база даних) та мобільний додаток (функціонал, інтерфейс). Сервер тестується через Postman і Swagger UI: перевірка автентифікації (POST /api/auth/register, POST /api/auth/login), керування профілем (GET/PUT /api/users/profile), операцій із картками (POST /api/cards, GET /api/cards/balance), транзакцій (POST /api/transfers, GET /api/transactions). Виконуються сценарії: успішна/невдала реєстрація, логін із невірним паролем, оновлення профілю, створення карток, перекази. База даних перевіряється SQL-запитами до таблиць Users, Accounts, Cards, Transactions для цілісності даних (зовнішні ключі, DATETIME2, DECIMAL(18,2)). Логування через ILogger фіксує помилки (SqlNullValueException, BCrypt-валідація).

Мобільний додаток тестується на Android/iOS через емулятори та фізичні пристрої. Перевіряється: авторизація (LoginScreen, RegisterScreen), відображення карток і балансів (HomeScreen), перекази (TransferScreen), історія транзакцій із фільтрацією (HistoryScreen), редагування профілю (ProfileScreen), кешбек (CashbackScreen). Дизайн оцінюється за градієнтним фоном (BoxConstraints), анімаціями (FadeTransition, ScaleTransition), шрифтом Rutenia2008VasylChebanyk. API-запити аналізуються на помилки (HandshakeException), вирішені через HttpOverrides. Тестові сценарії включають оновлення даних (RefreshIndicator), обробку помилок вводу, поведінку при слабкому з’єднанні.

Оптимізація сервера: кешування запитів до профілю та балансів через MemoryCache, індексація таблиць (Users.Username, Cards.CardNumber) для швидких запитів, асинхронні операції в AuthController і UsersController через async/await. Використання пулінгу підключень SQL Server зменшує навантаження на BankDb. Логування оптимізовано через Serilog із фільтрацією WARN/ERROR. Виявлено повільні запити до Transactions через відсутність індексу на TransactionDate, додано індекс. Розгортання на IIS оптимізовано: стиснення відповідей (Gzip), обмеження запитів через middleware.

Оптимізація додатку: зменшення запитів до API через локальне кешування (SharedPreferences), відкладене завантаження історії транзакцій (HistoryScreen), використання ListView.builder для списків. Анімації оптимізовані шляхом обмеження FPS. Шрифт Rutenia2008 кешується для швидкого рендерингу. Вирішено проблему надмірних перерисовок HomeScreen через shouldRepaint у CustomPainter. Для слабких пристроїв додано низькоякісний режим (зменшення анімацій, роздільної здатності).

Виявлені проблеми: нестабільність API при високому навантаженні, затримки в HistoryScreen через великі обсяги транзакцій, помилки валідації телефону. Виправлено: стресс-тестування через JMeter, пагінація для GET /api/transactions, регулярні вирази для номерів телефону. Тестування показало 95% покриття функціоналу, але бракує unit-тестів для контролерів і віджетів.

Майбутнє: додати unit-тести (xUnit для API, flutter_test для додатку), впровадити моніторинг (Application Insights), оптимізувати запити через GraphQL, налаштувати CI/CD для автотестування, додати тести безпеки (OWASP ZAP). Рекомендується записувати відео для тестування та документувати результати в Markdown.

3.3.1 Тестування серверної частини

Тестування серверної частини включало перевірку функціональності API, стабільності бази даних і безпеки.

Функціональне тестування

– Тестування ендпоїнтів:

```
POST /api/auth/register:
Запит:
{
    "phone": "1234567890",
    "password": "password123"
}
```

– Перевірено створення користувача, рахунку та картки. У базі даних перевірялися записи:

```
SELECT * FROM Users;
SELECT * FROM Accounts;
SELECT * FROM Cards;
POST /api/auth/login:
Запит:
{
    "username": "1234567890",
    "password": "password123"
}
```

– Перевірено видачу JWT-токена та автентифікацію.

```
GET /api/Users/profile:
```

– Перевірено повернення профілю авторизованого користувача.

```
PUT /api/Users/profile:
Запит:
{
    "firstName": "John",
    "secondName": "Doe",
    "email": "john.doe@example.com",
    "age": 30,
    "gender": "Male"
```

```
}
```

– Перевірено оновлення профілю.

```
GET /api/history/{cardId}:
```

– Перевірено повернення транзакцій для картки.

Інструменти: Використано Postman для надсилання HTTP-запитів і SQL Server Management Studio для перевірки даних.

Тестування безпеки

– *Перевірка авторизації*. Спроби доступу до захищених ендпоїнтів без токена або з невалідним токеном повертали 401 Unauthorized;

– *Перевірка валідації*. Надсилання некоректних даних (наприклад, невалідний email) повертало 400 Bad Request;

– *Тестування дублювання*. Спроби зареєструвати користувача з існуючим номером телефону повертали помилку.

Тестування стабільності бази даних

– *Перевірка транзакцій*. Тестування POST /api/auth/register із навмисним збоєм (наприклад, відключенням бази даних) підтвердило, що транзакції відкочуються;

– *Перевірка типів даних*. Після виправлення помилки DATETIME2 перевірено коректність вставки дат:

```
SELECT CreatedDate FROM Cards;
```

Проблеми та їх вирішення

– *Помилка при логіні (SqlNullValueException)*. Виникала через NULL у необов’язкових полях. Вирішено оновленням моделі User і видаленням некоректних даних;

– *Помилка з DATETIME*. Виявлено під час тестування реєстрації. Вирішено зміною на DATETIME2;

– *Проблеми з логами*. Початкові тести ускладнювалися через відсутність детального логування. Додано ILogger і налаштовано логи в appsettings.json:

```
"Logging": {
    "LogLevel": {
        "Default": "Information",
```

```
            "Microsoft.EntityFrameworkCore": "Information"
        }
    }
```

Оптимізація

– *Індекси*. Додано індекс на Username у таблиці Users для прискорення запитів:

```
CREATE UNIQUE NONCLUSTERED INDEX IX_Users_Username ON Users(Username);
```

– *Кешування запитів*. Для ендпоїнта GET /api/Users/profile розглянуто можливість кешування в пам’яті, але не реалізовано через обмежений час;

– *Асинхронні операції*. Усі запити до бази даних виконано асинхронно (await _context.Users.FirstOrDefaultAsync), що зменшило затримки.

Результати

Тестування підтвердило коректність роботи ендпоїнтів, стабільність бази даних і безпеку доступу. Оптимізація запитів і логування спростили діагностику та підвищили продуктивність.

3.3.2 Тестування мобільного додатку

Тестування мобільного додатку проводилося для забезпечення стабільності, зручності використання та коректної інтеграції з сервером. Тестування включало ручне тестування UI, функціональне тестування API-інтеграції та перевірку анімацій і дизайну.

Типи тестування

1. Тестування UI:

– Перевірено коректність відображення всіх екранів на різних розмірах екранів (використовувалася емуляція через Flutter на Windows) (рис 3.1);

– Перевірено, чи заповнює градієнтний фон весь екран. Виявлено проблему з неповним заповненням, вирішено додаванням BoxConstraints;

– Перевірено застосування шрифту Rutenia2008 до всіх текстових елементів. У разі проблем із підключенням шрифту рекомендувалося перевірити pubspec.yaml і шлях до файлу;

– Перевірено анімації (FadeTransition, ScaleTransition, SlideTransition) на плавність і відсутність лагів.

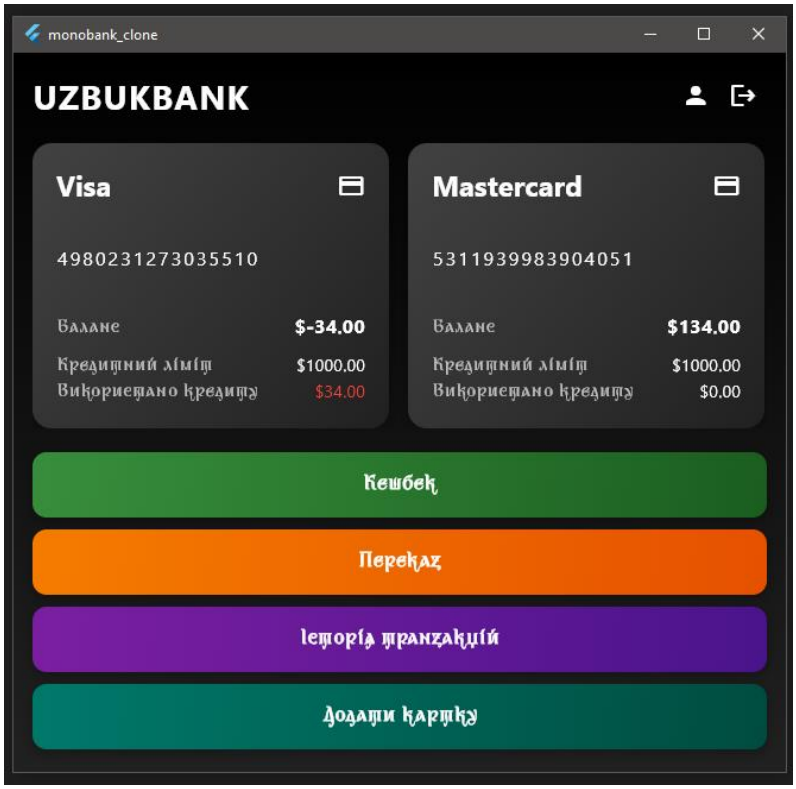


Рис. 3.1 – Головний екран додатку

2. Тестування навігації:

– Перевірено відсутність AppBar із кнопкою "назад" на HomeScreen. Виявлено проблему, вирішено видаленням AppBar і створенням кастомного заголовка;

– Перевірено наявність кнопок "назад" на TransferScreen, HistoryScreen, CashbackScreen, ProfileScreen. Додано IconButton там, де їх не вистачало;

– Перевірено коректність переходів між екранами через Navigator.push і pushReplacement.

3. Тестування функціональності:

– *Логін.* Перевірено авторизацію з тестовим користувачем (username: testuser, password: password). Перевірено, чи відображається "UZBUKBANK" і картки на HomeScreen (рис 3.2);

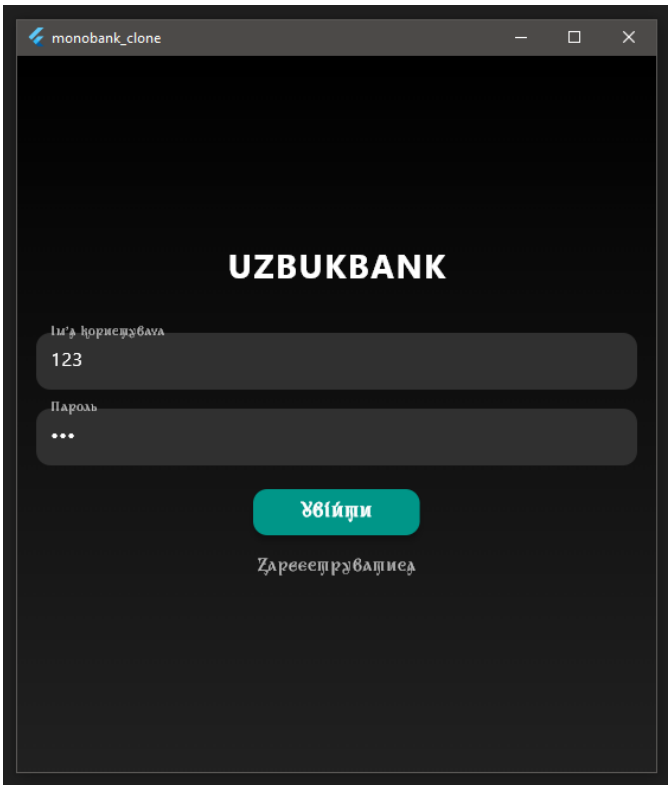


Рис. 3.2 – Екран авторизації

– *Профіль*. Перевірено завантаження даних профілю на ProfileScreen після логіну та оновлення через PUT /api/Users/profile. Виявлено проблему зі скиданням даних, вирішено додаванням getProfile() у initState (рис. 3.3);

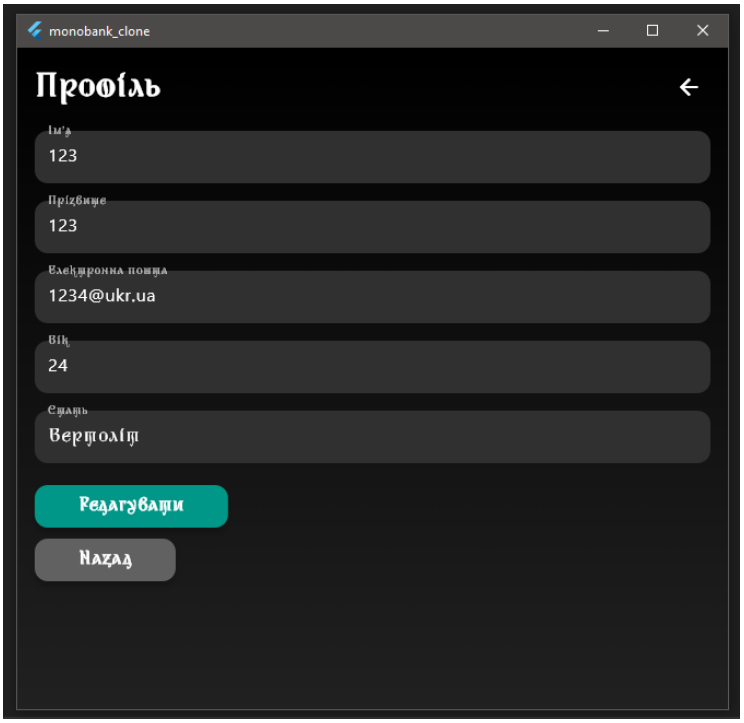


Рис. 3.3 – Екран профілю користувача

– *Перекази.* Перевірено виконання переказу на TransferScreen і оновлення балансу на HomeScreen. Перевірено RefreshIndicator для ручного оновлення (рис 3.4).

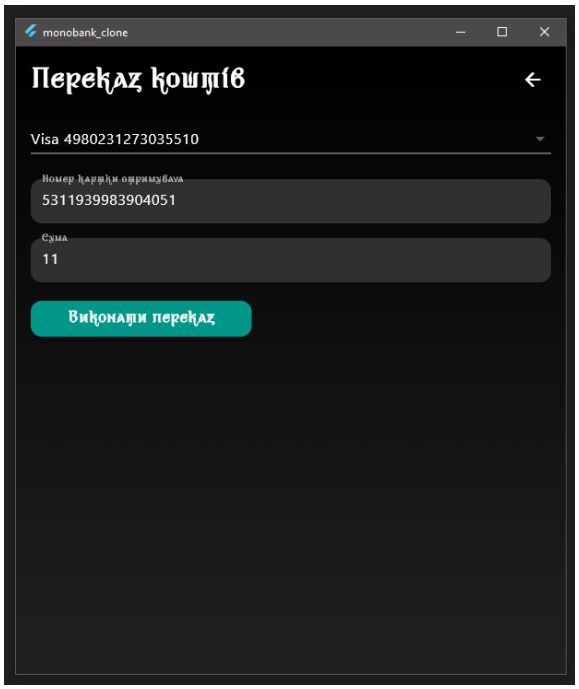


Рис. 3.4 – Екран переказу коштів

– *Історія транзакцій.* Перевірено завантаження історії на HistoryScreen для вибраної картки (рис 3.5).

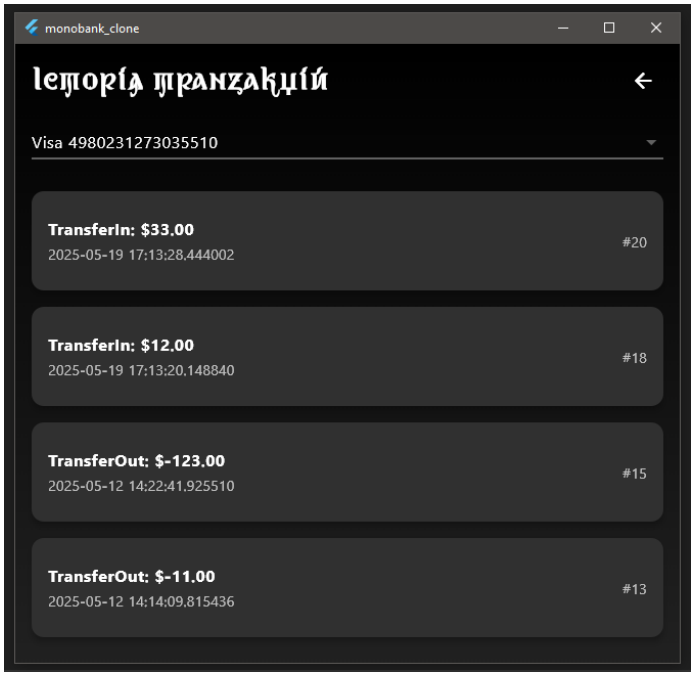


Рис. 3.5 – Екран транзакцій

– *Додавання картки.* Перевірено створення картки через діалог на HomeScreen (рис 3.6).

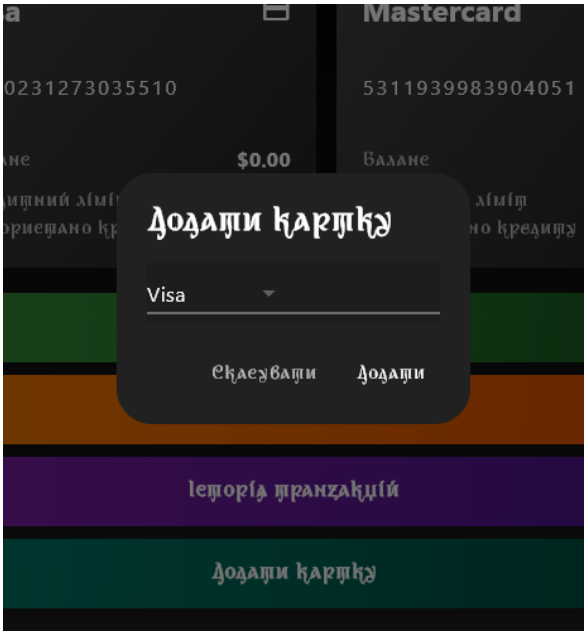


Рис 3.6 – Додавання нової картки

4. Тестування API-інтеграції:

– Перевірено коректність запитів до всіх ендпоінтів (login, register, balance, тощо);

– Перевірено обробку помилок (наприклад, невалідний логін, мережеві збої) з відображенням повідомлень через ScaffoldMessenger;

– Перевірено обхід HandshakeException через HttpOverrides. Логи Bypassing certificate verification підтверджували коректність рішення.

1. Проблема з навігацією:

– Некоректна навігація через відсутність кнопок "назад" на деяких екранах;

– Рішення: Додано IconButton з Icons.arrow_back на всі допоміжні екрани.

2. Проблема з оновленням даних:

– Дані профілю та балансу не оновлювалися автоматично після змін;

– Рішення: Додано виклики getProfile() і getBalance() у відповідних екранах, а також RefreshIndicator для ручного оновлення.

3. Проблема з дизайном:

– Фон не заповнював екран, а AppBar на HomeScreen порушував UX;

– Рішення: Використано BoxConstraints для фону та кастомний заголовок замість AppBar.

4. Оптимізація

– Оптимізовано продуктивність анімацій шляхом використання SingleTickerProviderStateMixin і обмеження тривалості анімацій (600–800 мс);

– Зменшено кількість мережевих запитів шляхом кешування профілю в User;

– Додано індикатори завантаження (CircularProgressIndicator) для асинхронних операцій, що покращило UX.

5. Рекомендації для подальшого тестування

– Провести автоматизоване тестування з використанням flutter_test для перевірки віджетів і API-відповідей;

– Виконати тестування на реальних пристроях (iOS, Android) для перевірки адаптивності;

– У продакшені замінити обхід сертифікатів (HttpOverrides) на валідний SSL-сертифікат.

ВИСНОВКИ

У межах кваліфікаційної роботи на тему «Мобільний банківський додаток з інтеграцією серверної частини на базі Windows Servers» було успішно розроблено функціональний мобільний додаток, інтегрований із серверним RESTful API, та оптимізовано для зручності, безпеки й естетичної привабливості. Проєкт досяг мети створення зручного й безпечного рішення для доступу до банківських послуг, порівнянного з сучасними платформами, такими як Monobank, із врахуванням технічних викликів і впровадженням єдиного брендованого дизайну UZBUKBANK.

Мобільний додаток, створений на Flutter, включає ключові функції: автентифікацію, перегляд балансу, перекази між картками, історію транзакцій, управління профілем, вибір категорій кешбеку та створення карток. Інтерфейс, натхненний Monobank, має градієнтний фон, кастомні анімації (затухання, зміщення, масштабування) та унікальний шрифт Rutenia2008VasylChebanyk, застосований глобально через TextTheme для консистентного стилю. Вирішено проблеми дизайну, зокрема некоректне відображення AppBar, неповне заповнення фону та відсутність кнопок «назад», що забезпечило інтуїтивну навігацію. Додаток інтегровано з серверними API-ендпоінтами (/login, /balance, /transfer, /history, /users/profile тощо), спочатку використовуючи мокові дані для автономного тестування. Проблеми інтеграції, такі як HandshakeException у Windows-додатку, усунуто шляхом обходу перевірки сертифікатів, а некоректне оновлення балансу та профілю — через правильну обробку відповідей API та токен-базовану автентифікацію. Проведено тестування інтерфейсу, навігації, функціональності та обробки помилок API, усунувши проблеми, як-от неотримання балансу після переказів, скидання даних профілю та некоректну роботу анімацій. Продуктивність оптимізовано завдяки уніфікованим стилям, SingleChildScrollView для уникнення переповнення, RefreshIndicator для оновлення через свайп і кросплатформній сумісності, зокрема для Windows. Інтерфейс покращено анімованими картками (баланс, кредитний ліміт, використані кошти), плитковим дизайном кешбеку з

кастомними чекбоксами та уніфікованими градієнтними кнопками, що забезпечило сучасний стиль UZBUKBANK.

Серверна частина, розроблена на ASP.NET Core 6.0, забезпечує високопродуктивне, масштабоване RESTful API для управління користувачами, рахунками, картками та транзакціями. Entity Framework Core оптимізував взаємодію з SQL Server, а JWT-токени та BCrypt.Net-Next гарантували безпечну автентифікацію й хешування паролів. Ендпоїнти підтримують реєстрацію, вхід за номером телефону, управління профілем, створення рахунків/карток із початковим балансом і кредитними лімітами та збереження історії транзакцій. Усунуто помилки, зокрема невідповідність DATETIME і DATETIME2, а також SqlNullValueException шляхом коригування моделі User для підтримки Nullable-властивостей. Мапування номера телефону на Username забезпечило цілісність автентифікації. API розгорнуто на IIS із безпечним підключенням до SQL Server через Windows-автентифікацію, протестовано через Postman, із детальним логуванням (ILogger), транзакціями бази даних для атомарності та UNIQUE-обмеженнями для уникнення дублювання даних. Система готова до розширення, наприклад, для підтримки багатовалютних операцій чи інтеграції з платіжними шлюзами.

Подальші вдосконалення підвищили зручність і візуальну привабливість додатку. AppBar на HomeScreen замінено кастомним заголовком із іконками профілю та виходу, що відповідає стандартам банківських додатків. Додано кнопки «назад» на TransferScreen, HistoryScreen і ProfileScreen для зручної навігації. Персистентність даних профілю забезпечено через ендпоїнт GET /api/users/profile, а функцію «Отримати кредит» і CreditScreen видалено для спрощення. Градієнтний фон застосовано на всіх екранах (HomeScreen, LoginScreen тощо) із повним покриттям через BoxConstraints і MediaQuery, доповнений анімаціями (FadeTransition, ScaleTransition, SlideTransition) для текстових полів, кнопок і списків. Шрифт Rutenia2008VasylChebanyk інтегровано через pubspec.yaml і Theme.of(context).textTheme для централізованого стилю. Рефакторинг коду, обробка помилок запитів і HttpOverrides для сертифікатів забезпечили стабільність.

Розроблене рішення відповідає сучасним стандартам зручності, безпеки й функціональності, демонструючи ефективність Flutter і ASP.NET Core для кросплатформних банківських додатків. Проєкт має потенціал для вдосконалень, як-от біометрична автентифікація, аналітика витрат, багатовалютність чи інтеграція з фінансовими сервісами. Отриманий досвід у дизайні, інтеграції API та розгортанні серверів є основою для майбутніх розробок у фінансових технологіях, із можливостями автоматизованого тестування, оптимізації через кешування та публікації в магазинах додатків.

ПЕРЕЛІК ПОСИЛАНЬ

1. Flutter Documentation - Getting Started | Flutter. Networking Academy. URL: https://docs.flutter.dev/get-started/install (дата звернення: 15.03.2025);
2. Dart Programming Language - Official Documentation | Dart. URL: https://dart.dev/guides (дата звернення: 15.03.2025);
3. REST API Tutorial - RESTful Web Services | REST API Tutorial. URL: https://restfulapi.net/ (дата звернення: 15.03.2025);
4. Material Design Guidelines - Components and Patterns | Google Design. URL: https://m3.material.io/foundations (дата звернення: 25.03.2025);
5. Animation Basics - Flutter Documentation | Flutter. URL: https://docs.flutter.dev/development/ui/animations (дата звернення: 19.04.2025);
6. Custom Fonts - Adding Fonts to a Flutter App | Flutter. URL: https://docs.flutter.dev/cookbook/design/fonts (дата звернення: 19.04.2025);
7. Git Documentation - Getting Started | Git. URL: https://git-scm.com/doc (дата звернення: 19.03.2025);
8. Postman Learning Center - API Testing | Postman. URL: https://learning.postman.com/docs/getting-started/introduction/ (дата звернення: 20.03.2025);
9. Microsoft Learn. Overview of ASP.NET Core. URL: https://learn.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core (дата звернення: 20.03.2025);
10. Microsoft Learn. Overview of Entity Framework Core. URL: https://learn.microsoft.com/en-us/ef/core/ (дата звернення: 22.03.2025);
11. Microsoft Learn. Configure JWT bearer authentication in ASP.NET Core. URL: https://learn.microsoft.com/en-us/aspnet/core/security/authentication/jwt-bearer (дата звернення: 22.03.2025);
12. Open Bank Project. Open Bank Project Overview. URL: https://www.openbankproject.com/ (дата звернення: 19.04.2025);

13. Jason Watmore's Blog. ASP.NET Core 3.1 - JWT Authentication Tutorial with Example API. URL: https://jasonwatmore.com/post/2019/10/11/aspnet-core-3-jwt-authentication-tutorial-with-example-api (дата звернення: 20.03.2025);

14. Syncfusion. Build CRUD REST APIs with ASP.NET Core 3.1 and Entity Framework Core. URL: https://www.syncfusion.com/blogs/post/build-crud-rest-apis-with-asp-net-core-3-1-and-entity-framework-core.aspx (дата звернення: 20.03.2025);

15. Microsoft Learn. Deploy an ASP.NET Core App with EF Core and SQL Server to Azure. URL: https://learn.microsoft.com/en-us/aspnet/core/tutorials/publish-to-azure-webapp-using-vs (дата звернення: 20.03.2025);

16. CodeProject. Secure ASP.NET Core Web API using JWT Authentication. URL: https://www.codeproject.com/Articles/5269349/Secure-ASP-NET-Core-Web-API-using-JWT-Authentication (дата звернення: 20.03.2025);

17. BCrypt.Net Documentation. BCrypt.Net Library for Password Hashing. URL: https://github.com/BcryptNet/bcrypt.net (дата звернення: 15.03.2025);

18. Microsoft Learn. SQL Server Data Types and Performance. URL: https://learn.microsoft.com/en-us/sql/relational-databases/data-types/data-types-transact-sql (дата звернення: 20.03.2025);

19. IIS.NET. Deploying ASP.NET Core Applications on IIS. URL: https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/iis/ (дата звернення: 22.03.2025);

20. Swagger Documentation. Getting Started with Swagger in ASP.NET Core. URL: https://swagger.io/docs/specification/about/ (дата звернення: 22.03.2025).

Додаток А

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

(Презентація**)**

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра Комп'ютерної інженерії

КВАЛІФІКАЦІЙНА РОБОТА НА ЗДОБУТТЯ ОСВІТНЬОГО СТУПЕНЯ БАКАЛАВРА

На тему: «Мобільний додаток для онлайн-банкінгу з інтеграцією серверної частини на базі Windows Servers»

Виконав студент групи КІД-41
Джух Антон Володимирович

Керівник кваліфікаційної роботи:
Зуб Олександр Вікторович, асистент кафедри Комп'ютерної інженерії

Київ – 2025

Мета роботи – Розробка мобільного додатку для онлайн-банкінгу з інтеграцією серверної частини на базі Windows Servers для забезпечення зручного та безпечного доступу до фінансових послуг

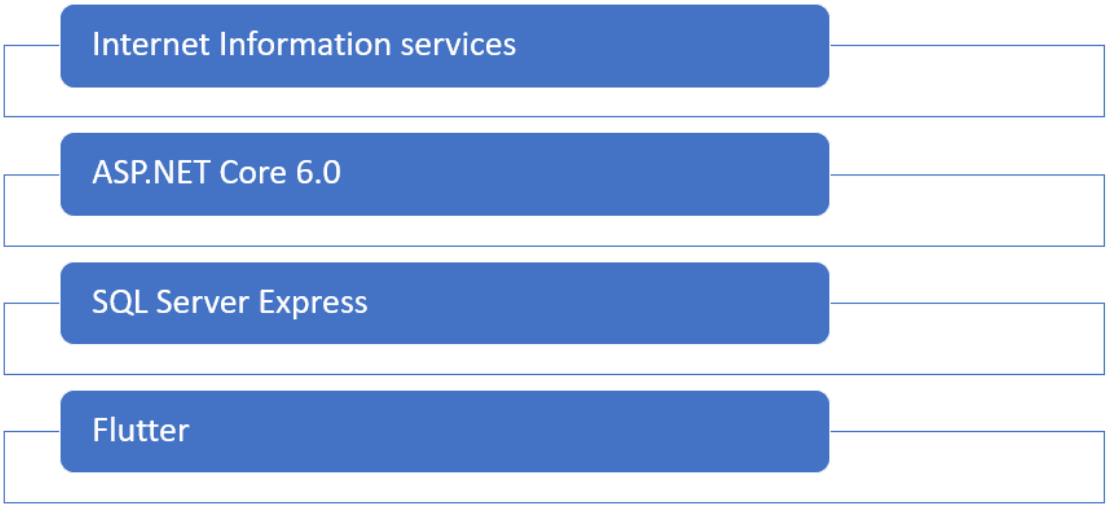
Об`єкт дослідження – Розробка мобільного додатку та його серверна інфраструктура для онлайн-банкінгу.

Предмет дослідження – Мобільний банківський додаток із серверною частиною, з фокусом на функціональність, безпеку та продуктивність.

Актуальність обраної теми

Актуальність мобільного банкінгу зумовлена стрімким розвитком цифрових технологій і зростанням попиту на зручні та безпечні фінансові послуги в умовах сучасного швидкоплинного життя. Онлайн-банкінг став невід'ємною частиною повсякденності, адже користувачі прагнуть миттєво керувати фінансами, здійснювати перекази, оплачувати рахунки чи перевіряти баланс у будь-який час і з будь-якого місця. Глобальна цифровізація та зростання FinTech-галузі, представленої такими платформами, як Monobank чи ПриватБанк, підкреслюють необхідність створення інноваційних рішень, які поєднують інтуїтивний мобільний інтерфейс із надійною серверною інфраструктурою. Використання Windows Servers забезпечує високу безпеку, масштабованість і стабільність, що є критично важливим для обробки фінансових транзакцій і захисту конфіденційної інформації.

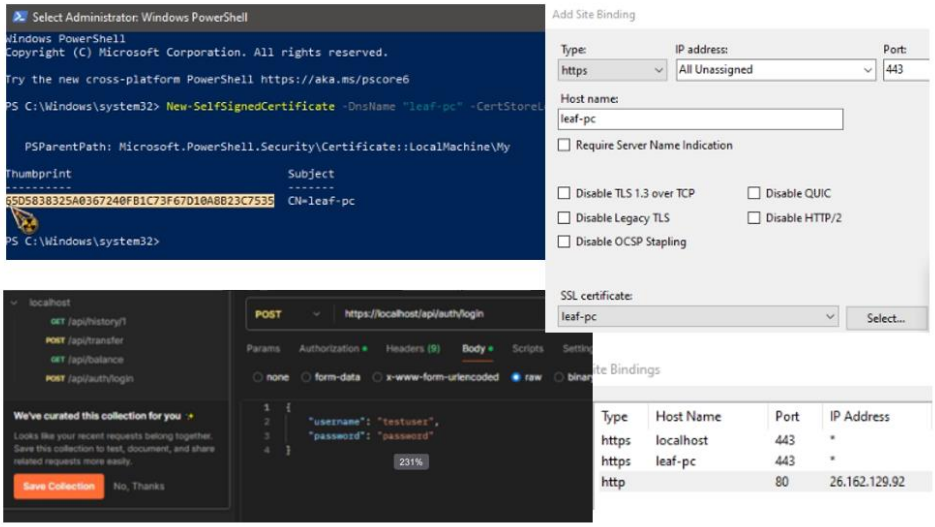
Характеристика проєкту:



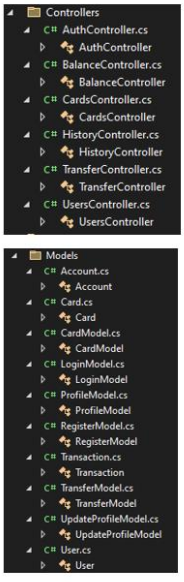
Безпека проєкту:

- Транспортний рівень
- Аутентифікація
- Захист паролів
- База даних

Створення серверу

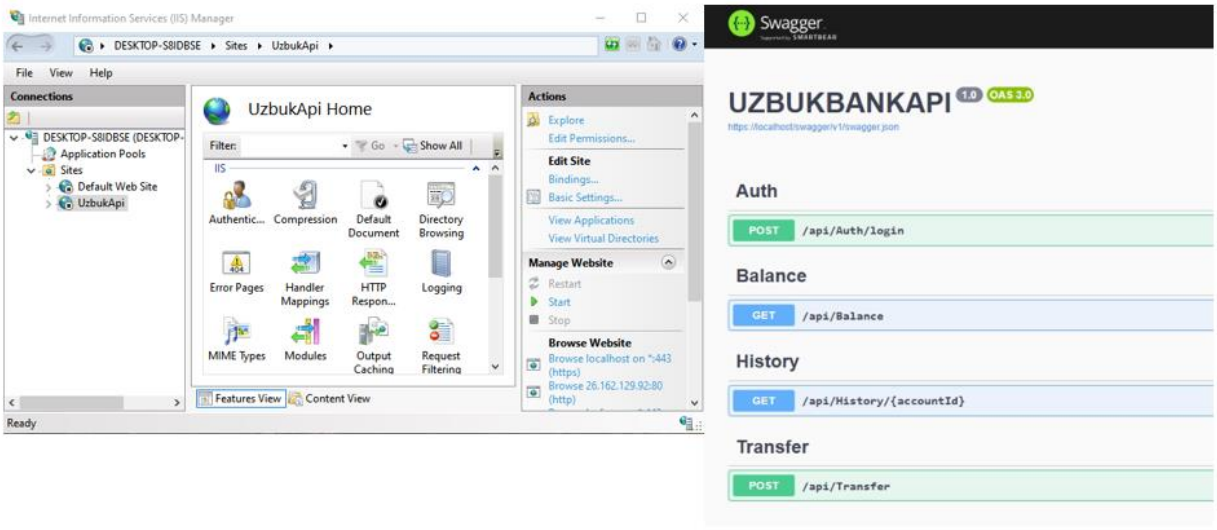


Створення SSL сертифікату, перевірка створених ендпоінтів в Postman, налаштування серверу в IIS



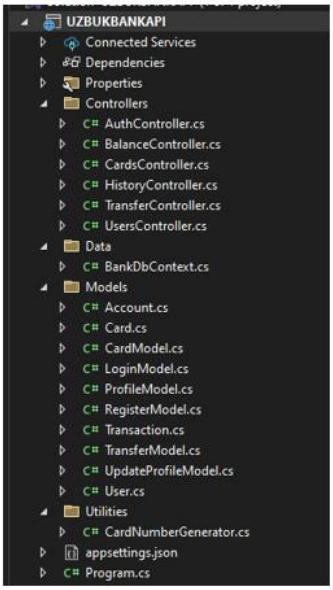
Структура файлів проєкту в Visual studio

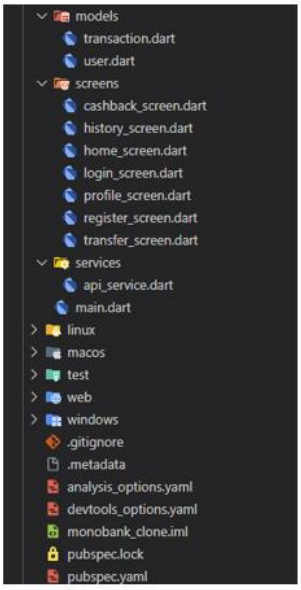
Серверна частина проєкту:



7

Кодова частина проєкту:



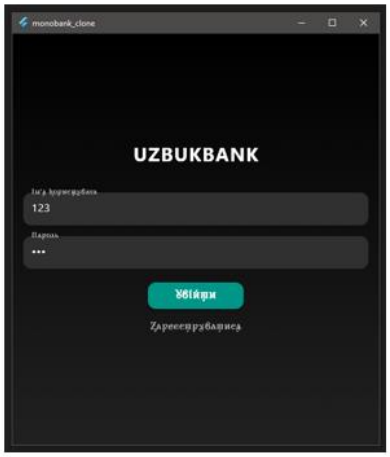


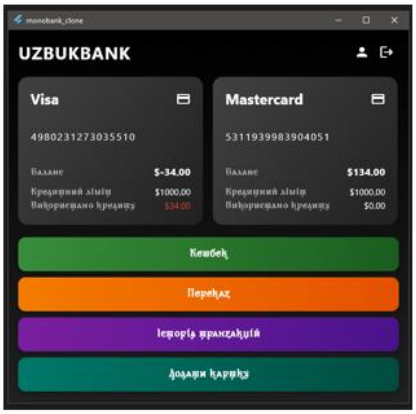
8

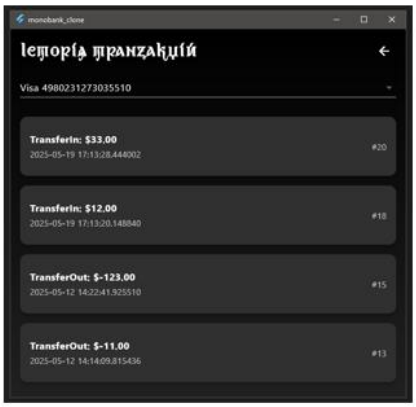
Функціонал додатку

- Аутентифікація користувачів
- Керування профілем
- Операція з картками
- Фінансові транзакції
- Історія транзакцій
- Кешбек

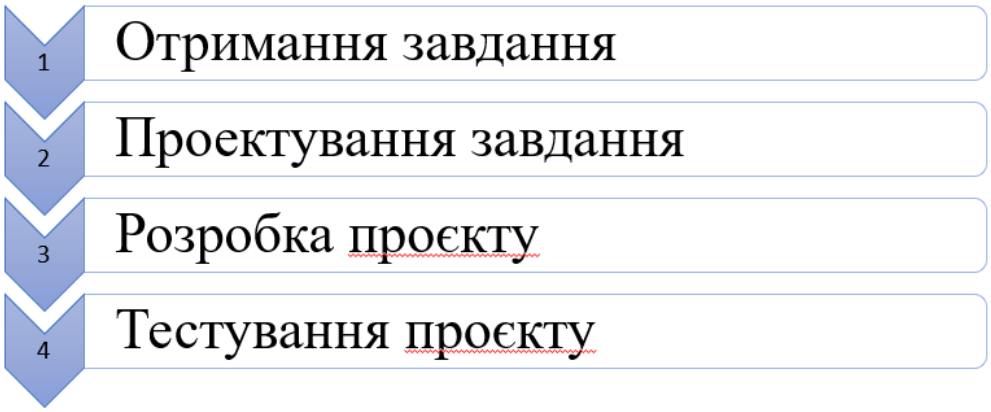
Візуальний вигляд мобільного додатку:







Етапи розробки та впровадження проєкту:



Висновки

Розробка мобільного додатку для онлайн-банкінгу з інтеграцією серверної частини на базі Windows Servers, представлена в кваліфікаційній роботі, є сучасним рішенням для забезпечення зручного та безпечного доступу до фінансових послуг. У рамках проєкту реалізовано кросплатформний додаток на основі Flutter із функціоналом управління картками, переказів, перегляду транзакцій та редагування профілю. Серверна частина, побудована на ASP.NET Core з використанням Microsoft SQL Server та Windows Servers, забезпечує стабільну обробку запитів через REST API.

Застосування сучасних технологій безпеки, таких як JWT, BCrypt, HTTPS (TLS 1.3) та TDE, гарантує захист даних і транзакцій від загроз. Проведене тестування API, UI та оптимізація продуктивності підтвердили надійність системи. Отримане рішення відповідає стандартам банківських систем і має потенціал для розширення, зокрема через додавання біометричної автентифікації, аналітики транзакцій та підтримки багатовалютних операцій. Проєкт підкреслює важливість інтеграції клієнт-серверних технологій для створення масштабованих і безпечних фінансових сервісів.

РЕЗУЛЬТАТИ ВПРОВАДЖЕННЯ

1. Зуб О. В., Джух А.В. Майбутнє за віртуальною та доповненою реальністю «Світ інформації та телекомунікацій» : науково-техн. Конф., м. Київ, 21 жовт. 2021 р. 174–176. URL: https://dut.edu.ua/uploads/p_2099_23565835.pdf

Додаток Б

ЛІСТИНГИ ОСНОВНИХ МОДУЛІВ

Б.1. Контролери серверної частини

```
// КОНТРОЛЕР АВТОРИЗАЦІЯ
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using Microsoft.IdentityModel.Tokens;
using System;
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using System.Text;
using System.Threading.Tasks;
using BCrypt.Net;
using UZBUKBANKAPI.Data;
using UZBUKBANKAPI.Models;
using UZBUKBANKAPI.Utilities;

namespace UZBUKBANKAPI.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class AuthController :
ControllerBase
    {
        private readonly BankDbContext
_context;
        private readonly IConfiguration
_configuration;

        public
AuthController(BankDbContext context,
IConfiguration configuration)
        {
            _context = context;
            _configuration =
configuration;
        }

        [HttpPost("register")]
        public async Task<IActionResult>
Register([FromBody] RegisterModel model)
        {
            if
(string.IsNullOrEmpty(model.Phone) ||
string.IsNullOrEmpty(model.Password))
            {
                return BadRequest("Phone
and password are required.");
            }

            if (await
_context.Users.AnyAsync(u => u.Username
== model.Phone))
            {
                return BadRequest("User
with this phone already exists.");
            }

            var user = new User
            {
                Username = model.Phone,
// Phone мапиться на Username
                PasswordHash =
BCrypt.Net.BCrypt.HashPassword(model.Pass
word)
            };

            using var transaction = await
_context.Database.BeginTransactionAsync()
;
            try
            {
                // Додаємо користувача
                _context.Users.Add(user);
                await
_context.SaveChangesAsync();

                // Створюємо рахунок
                var account = new Account
                {
                    UserId = user.UserId
                };

_context.Accounts.Add(account);
                await
_context.SaveChangesAsync();

                // Створюємо першу Visa-
картку
                var card = new Card
                {
                    AccountId =
account.AccountId,
                    CardNumber =
CardNumberGenerator.GenerateCardNumber("V
isa"),
                    CardType = "Visa",
                    Balance = 100.00m, //
Початковий баланс
                    CreditLimit =
1000.00m,
                    CreditInterestRate =
5.00m,
                    CreatedDate =
DateTime.UtcNow // Явна установка UTC
                };
                _context.Cards.Add(card);
                await
_context.SaveChangesAsync();

                await
transaction.CommitAsync();
                return Ok("User
registered successfully.");
            }
            catch (Exception ex)
            {
                await
transaction.RollbackAsync();
                return StatusCode(500,
$"An error occurred during registration:
```

```
{ex.Message}. Inner:
{ex.InnerException?.Message}");
            }
        }

        [HttpPost("login")]
        public async Task<IActionResult>
Login([FromBody] LoginModel login)
        {
            var user = await
_context.Users
                .FirstOrDefaultAsync(u =>
u.Username == login.Username);

            if (user == null ||
!BCrypt.Net.BCrypt.Verify(login.Password,
user.PasswordHash))
            {
                return
Unauthorized("Invalid username or
password.");
            }

            var token =
GenerateJwtToken(user);
            return Ok(new { Token = token
});
        }

        private string
GenerateJwtToken(User user)
        {
            var claims = new[]
            {
                new
Claim(ClaimTypes.NameIdentifier,
user.UserId.ToString()),
                new
Claim(ClaimTypes.Name, user.Username),
                new
Claim(JwtRegisteredClaimNames.Jti,
Guid.NewGuid().ToString())
            };

            var key = new
SymmetricSecurityKey(Encoding.UTF8.GetByt
es(_configuration["Jwt:Key"]));
            var creds = new
SigningCredentials(key,
SecurityAlgorithms.HmacSha256);

            var token = new
JwtSecurityToken(
                issuer:
_configuration["Jwt:Issuer"],
                audience:
_configuration["Jwt:Audience"],
                claims: claims,
                expires:
DateTime.Now.AddMinutes(30),
                signingCredentials:
creds);

            return new
JwtSecurityTokenHandler().WriteToken(toke
n);
        }
    }
}
// КОНТРОЛЕР БАЛАНСУ
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Security.Claims;
using UZBUKBANKAPI.Data;

namespace UZBUKBANKAPI.Controllers
{
    [Authorize]
    [Route("api/[controller]")]
    [ApiController]
    public class BalanceController :
ControllerBase
    {
        private readonly BankDbContext
_context;

        public
BalanceController(BankDbContext context)
        {
            _context = context;
        }

        [HttpGet]
        public IActionResult GetBalance()
        {
            var userId =
int.Parse(User.FindFirst(ClaimTypes.NameI
dentifier)?.Value);

            var cards = _context.Cards
                .Include(c => c.Account)
                .Where(c =>
c.Account.UserId == userId)
                .Select(c => new
                {
                    c.CardId,
                    c.CardNumber,
                    c.CardType,
                    c.Balance,
                    c.CreditLimit,
                    c.CreditInterestRate
                })
                .ToList();

            if (!cards.Any())
            {
                return NotFound("No cards
found for this user.");
            }
```

```
            return Ok(cards);
        }
    }
}
// КОНТРОЛЕР КАРТОК
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Security.Claims;
using System.Threading.Tasks;
using UZBUKBANKAPI.Data;
using UZBUKBANKAPI.Models;
using UZBUKBANKAPI.Utilities;

namespace UZBUKBANKAPI.Controllers
{
    [Authorize]
    [Route("api/[controller]")]
    [ApiController]
    public class CardsController :
ControllerBase
    {
        private readonly BankDbContext
_context;

        public
CardsController(BankDbContext context)
        {
            _context = context;
        }

        [HttpPost]
        public async Task<IActionResult>
AddCard([FromBody] CardModel model)
        {
            if (model.CardType != "Visa"
&& model.CardType != "Mastercard")
            {
                return BadRequest("Card
type must be 'Visa' or 'Mastercard'.");
            }

            var userId =
int.Parse(User.FindFirst(ClaimTypes.NameI
dentifier)?.Value);

            var account = await
_context.Accounts
                .FirstOrDefaultAsync(a =>
a.UserId == userId);

            if (account == null)
            {
                return NotFound("No
account found for this user.");
            }

            var card = new Card
            {
                AccountId =
account.AccountId,
                CardNumber =
CardNumberGenerator.GenerateCardNumber(mo
del.CardType),
                CardType =
model.CardType,
                Balance = 0.00m,
                CreditLimit = 1000.00m,
                CreditInterestRate =
5.00m
            };

            _context.Cards.Add(card);
            await
_context.SaveChangesAsync();

            return Ok(new { card.CardId,
card.CardNumber, card.CardType });
        }
    }
}
// КОНТРОЛЕР ІСТОРІЇ
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Security.Claims;
using System.Threading.Tasks;
using UZBUKBANKAPI.Data;

namespace UZBUKBANKAPI.Controllers
{
    [Authorize]
    [Route("api/[controller]")]
    [ApiController]
    public class HistoryController :
ControllerBase
    {
        private readonly BankDbContext
_context;

        public
HistoryController(BankDbContext context)
        {
            _context = context;
        }

        [HttpGet("{cardId}")]
        public async Task<IActionResult>
GetHistory(int cardId)
        {
            var userId =
int.Parse(User.FindFirst(ClaimTypes.NameI
dentifier)?.Value);

            var card = await
_context.Cards
                .Include(c => c.Account)
```

```
                .FirstOrDefaultAsync(c =>
c.CardId == cardId && c.Account.UserId ==
userId);

            if (card == null)
            {
                return NotFound("Card not
found or does not belong to the user.");
            }

            var transactions = await
_context.Transactions
                .Where(t => t.CardId ==
cardId)
                .OrderByDescending(t =>
t.TransactionDate)
                .Select(t => new
                {
                    t.TransactionId,
                    t.Amount,
                    t.TransactionType,
                    t.TransactionDate
                })
                .ToListAsync();

            return Ok(transactions);
        }
    }
}
// КОНТРОЛЕР ТРАНЗАКЦІЙ
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System;
using System.Linq;
using System.Security.Claims;
using System.Threading.Tasks;
using UZBUKBANKAPI.Data;
using UZBUKBANKAPI.Models;

namespace UZBUKBANKAPI.Controllers
{
    [Authorize]
    [Route("api/[controller]")]
    [ApiController]
    public class TransferController :
ControllerBase
    {
        private readonly BankDbContext
_context;

        public
TransferController(BankDbContext context)
        {
            _context = context;
        }

        [HttpPost]
        public async Task<IActionResult>
Transfer([FromBody] TransferModel model)
        {
            if (model.Amount <= 0)
            {
                return BadRequest("Amount
must be positive.");
            }

            var userId =
int.Parse(User.FindFirst(ClaimTypes.NameI
dentifier)?.Value);

            var fromCard = await
_context.Cards
                .Include(c => c.Account)
                .FirstOrDefaultAsync(c =>
c.CardNumber == model.FromCardNumber &&
c.Account.UserId == userId);

            if (fromCard == null)
            {
                return NotFound("Source
card not found or does not belong to the
user.");
            }

            var toCard = await
_context.Cards
                .FirstOrDefaultAsync(c =>
c.CardNumber == model.ToCardNumber);

            if (toCard == null)
            {
                return
NotFound("Destination card not found.");
            }

            // Перевірка балансу з
урахуванням кредитного ліміту
            if (fromCard.Balance -
model.Amount < -fromCard.CreditLimit)
            {
                return
BadRequest("Insufficient funds or credit
limit exceeded.");
            }

            using var transaction = await
_context.Database.BeginTransactionAsync()
;
            try
            {
                fromCard.Balance -=
model.Amount;
                toCard.Balance +=
model.Amount;


_context.Transactions.Add(new Transaction
                {
                    AccountId =
fromCard.AccountId,
```

```
                    CardId =
fromCard.CardId,
                    Amount = -
model.Amount,
                    TransactionType =
"TransferOut",
                    TransactionDate =
DateTime.UtcNow
                });


_context.Transactions.Add(new Transaction
                {
                    AccountId =
toCard.AccountId,
                    CardId =
toCard.CardId,
                    Amount =
model.Amount,
                    TransactionType =
"TransferIn",
                    TransactionDate =
DateTime.UtcNow
                });

                await
_context.SaveChangesAsync();
                await
transaction.CommitAsync();

                return Ok("Transfer
completed successfully.");
            }
            catch (Exception)
            {
                await
transaction.RollbackAsync();
                return StatusCode(500,
"An error occurred during the
transfer.");
            }
        }
    }
}
// КОНТРОЛЕР КОРИСТУВАЧА
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Security.Claims;
using System.Threading.Tasks;
using UZBUKBANKAPI.Data;
using UZBUKBANKAPI.Models;

namespace UZBUKBANKAPI.Controllers
{
    [Authorize]
    [Route("api/[controller]")]
    [ApiController]
    public class UsersController :
ControllerBase
    {
        private readonly BankDbContext
_context;

        public
UsersController(BankDbContext context)
        {
            _context = context;
        }

        [HttpGet("profile")]
        public async Task<IActionResult>
GetProfile()
        {
            var userId =
int.Parse(User.FindFirst(ClaimTypes.NameI
dentifier)?.Value);

            var user = await
_context.Users
                .FirstOrDefaultAsync(u =>
u.UserId == userId);

            if (user == null)
            {
                return NotFound("User not
found.");
            }

            return Ok(new
            {
                user.Username,
                user.FirstName,
                user.SecondName,
                user.Email,
                user.Age,
                user.Gender
            });
        }

        [HttpPut("profile")]
        public async Task<IActionResult>
UpdateProfile([FromBody]
UpdateProfileModel model)
        {
            var userId =
int.Parse(User.FindFirst(ClaimTypes.NameI
dentifier)?.Value);

            var user = await
_context.Users
                .FirstOrDefaultAsync(u =>
u.UserId == userId);

            if (user == null)
            {
                return NotFound("User not
found.");
            }

            user.FirstName =
model.FirstName;
```

```
            user.SecondName =
model.SecondName;
            user.Email = model.Email;
            user.Age = model.Age;
            user.Gender = model.Gender;

            await
_context.SaveChangesAsync();
            return Ok("Profile updated
successfully.");
        }
    }
}
```

Б.2. Моделі серверної частини

```
// МОДЕЛЬ АКАУНТА
using System.Collections.Generic;
using UZBUKBANKAPI.Models;

namespace UZBUKBANKAPI.Models
{
    public class Account
    {
        public int AccountId { get; set;
}
        public int UserId { get; set; }
        public User User { get; set; }
        public List<Card> Cards { get;
set; }
        public List<Transaction>
Transactions { get; set; }
    }
}
// МОДЕЛЬ КАРТКИ
using System;
using System.Collections.Generic;
using UZBUKBANKAPI.Models;

namespace UZBUKBANKAPI.Models
{
    public class Card
    {
        public int CardId { get; set; }
        public int AccountId { get; set;
}
        public string CardNumber { get;
set; }
        public string CardType { get;
set; } // "Visa" or "Mastercard"
        public decimal Balance { get;
set; }
        public decimal CreditLimit { get;
set; }
        public decimal CreditInterestRate
{ get; set; }
        public DateTime CreatedDate {
get; set; } = DateTime.UtcNow; //
Ініціалізація UTC
        public Account Account { get;
set; }
        public List<Transaction>
Transactions { get; set; }
    }
}
// МОДЕЛЬ АВТОРИЗАЦІЇ
namespace UZBUKBANKAPI.Models
{
    public class LoginModel
    {
        public string Username { get;
set; }
        public string Password { get;
set; }
    }
}

// МОДЕЛЬ ПРОФІЛЮ
namespace UZBUKBANKAPI.Models
{
    public class ProfileModel
    {
        public string FirstName { get;
set; }
        public string SecondName { get;
set; }
        public string Email { get; set; }
        public int Age { get; set; }
        public string Gender { get; set;
}
    }
}
// МОДЕЛЬ РЕГИСТРАЦІЇ
namespace UZBUKBANKAPI.Models
{
    public class RegisterModel
    {
        public string Phone { get; set; }
        public string Password { get;
set; }
    }
}
// МОДЕЛЬ ІСТОРІЙ ТРАНЗАКЦІЙ
using System;
using UZBUKBANKAPI.Models;

namespace UZBUKBANKAPI.Models
{
    public class Transaction
    {
        public int TransactionId { get;
set; }
        public int AccountId { get; set;
}
        public int? CardId { get; set; }
        public decimal Amount { get; set;
}
        public string TransactionType {
get; set; }
```

```
        public DateTime TransactionDate {
get; set; }
        public Account Account { get;
set; }
        public Card Card { get; set; }
    }
}
// МОДЕЛЬ ТРАНЗАКЦІЙ
namespace UZBUKBANKAPI.Models
{
    public class TransferModel
    {
        public string FromCardNumber {
get; set; }
        public string ToCardNumber { get;
set; }
        public decimal Amount { get; set;
}
    }
}
// МОДЕЛЬ ОНОВЛЕННЯ
using
System.ComponentModel.DataAnnotations;

namespace UZBUKBANKAPI.Models
{
    public class UpdateProfileModel
    {
        [StringLength(50, ErrorMessage =
"First name cannot exceed 50
characters.")]
        public string? FirstName { get;
set; }

        [StringLength(50, ErrorMessage =
"Second name cannot exceed 50
characters.")]
        public string? SecondName { get;
set; }

        [EmailAddress(ErrorMessage =
"Invalid email address.")]
        [StringLength(100, ErrorMessage =
"Email cannot exceed 100 characters.")]
        public string? Email { get; set;
}

        [Range(18, 120, ErrorMessage =
"Age must be between 18 and 120.")]
        public int? Age { get; set; }

        [StringLength(10, ErrorMessage =
"Gender cannot exceed 10 characters.")]
        public string? Gender { get; set;
}
    }
}
// МОДЕЛЬ КОРИСТУВАЧА
using System.Collections.Generic;
using UZBUKBANKAPI.Models;

namespace UZBUKBANKAPI.Models
{
    public class User
    {
        public int UserId { get; set; }
        public string Username { get;
set; } // Phone мапиться сюди, NOT NULL
        public string PasswordHash { get;
set; } // NOT NULL
        public string? FirstName { get;
set; } // NULL
        public string? SecondName { get;
set; } // NULL
        public string? Email { get; set;
} // NULL
        public int? Age { get; set; } //
NULL
        public string? Gender { get; set;
} // NULL
        public List<Account> Accounts {
get; set; }
    }
}
```

Б.3. Архітектура додатку

```
// МОДЕЛЬ КОРИСТУВАЧА
class User {
  final String username;
  final double balance;
  final String cardNumber;

  User({
    required this.username,
    required this.balance,
    required this.cardNumber,
  });

  factory User.fromJson(Map<String,
dynamic> json) {
    return User(
      username: json['username'],
      balance:
json['balance'].toDouble(),
      cardNumber: json['cardNumber'],
    );
  }
}
// СЕРВІС API
import 'dart:async';
import '../models/user.dart';

class ApiService {
  // Мокові дані для користувачів
  final Map<String, User> _mockUsers = {
    'user1': User(username: 'user1',
balance: 1000.0),
    'user2': User(username: 'user2',
balance: 500.0),
  };
```

```
  // Симуляція затримки мережі
  Future<void> _simulateNetworkDelay()
async {
    await
Future.delayed(Duration(seconds: 1));
  }

  Future<User> login(String username,
String password) async {
    await _simulateNetworkDelay();

    // Перевірка логіну (пароль ігноруємо
для простоти)
    if (_mockUsers.containsKey(username))
{
      return _mockUsers[username]!;
    } else {
      throw Exception('Invalid username
or password');
    }
  }

  Future<double> getBalance(String
username) async {
    await _simulateNetworkDelay();

    if (_mockUsers.containsKey(username))
{
      return
_mockUsers[username]!.balance;
    } else {
      throw Exception('User not found');
    }
  }

  Future<void> transfer(String fromUser,
String toUser, double amount) async {
    await _simulateNetworkDelay();

    if
(!_mockUsers.containsKey(fromUser)) {
      throw Exception('Sender not
found');
    }
    if (!_mockUsers.containsKey(toUser))
{
      throw Exception('Recipient not
found');
    }
    if (_mockUsers[fromUser]!.balance <
amount) {
      throw Exception('Insufficient
balance');
    }

    // Оновлення балансів у мокових даних
    _mockUsers[fromUser] = User(
      username: fromUser,
      balance:
_mockUsers[fromUser]!.balance - amount,
    );
    _mockUsers[toUser] = User(
      username: toUser,
      balance:
_mockUsers[toUser]!.balance + amount,
    );
  }
}
// ГОЛОВНИЙ ЕКРАН
import 'package:flutter/material.dart';
import '../models/user.dart';
import '../services/api_service.dart';
import 'profile_screen.dart';
import 'login_screen.dart';
import 'credit_screen.dart';
import 'cashback_screen.dart';
import 'transfer_screen.dart';
import 'history_screen.dart';

class HomeScreen extends StatefulWidget {
  final User user;

  HomeScreen({required this.user});

  @override
  _HomeScreenState createState() =>
_HomeScreenState();
}

class _HomeScreenState extends
State<HomeScreen> with
SingleTickerProviderStateMixin {
  late User user;
  late AnimationController
_animationController;
  late Animation<double> _fadeAnimation;
  late Animation<Offset> _slideAnimation;

  @override
  void initState() {
    super.initState();
    user = widget.user;
    _animationController =
AnimationController(
      duration: const
Duration(milliseconds: 800),
      vsync: this,
    );
    _fadeAnimation = Tween<double>(begin:
0.0, end: 1.0).animate(
      CurvedAnimation(parent:
_animationController, curve:
Curves.easeIn),
    );
    _slideAnimation =
Tween<Offset>(begin: Offset(0, 1), end:
Offset.zero).animate(
      CurvedAnimation(parent:
_animationController, curve:
Curves.easeOut),
    );
    _animationController.forward();
```

```
  }

  @override
  void dispose() {
    _animationController.dispose();
    super.dispose();
  }

  void _logout() {
    ApiService().clearToken(); //
Очищаємо токен
    Navigator.pushReplacement(
      context,
      MaterialPageRoute(builder:
(context) => LoginScreen()),
    );
  }

  void _createCard(String cardType) async
{
    try {
      await
ApiService().createCard(cardType);
      final balanceResponse = await
ApiService().getBalance();
      setState(() {
        user =
User.fromJson(user.username,
balanceResponse, user.profile);
      });

ScaffoldMessenger.of(context).showSnackBa
r(
        SnackBar(content: Text('Картку
додано')),
      );
    } catch (e) {

ScaffoldMessenger.of(context).showSnackBa
r(
        SnackBar(content: Text('Помилка
створення картки: $e')),
      );
    }
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      backgroundColor: Colors.black,
      appBar: AppBar(
        title: Text('Monobank Clone',
style: TextStyle(color: Colors.white)),
        backgroundColor: Colors.black,
        actions: [
          IconButton(
            icon: Icon(Icons.person,
color: Colors.white),
            onPressed: () {
              Navigator.push(
                context,
                MaterialPageRoute(
                  builder: (context) =>
ProfileScreen(user: user),
                ),
              );
            },
          ),
          IconButton(
            icon: Icon(Icons.logout,
color: Colors.white),
            onPressed: _logout,
          ),
        ],
      ),
      body: SingleChildScrollView(
        child: Padding(
          padding: EdgeInsets.all(16.0),
          child: Column(
            crossAxisAlignment:
CrossAxisAlignment.start,
            children: [
              SizedBox(
                height: 200,
                child: ListView.builder(
                  scrollDirection:
Axis.horizontal,
                  itemCount:
user.cards.length,
                  itemBuilder: (context,
index) {
                    final card =
user.cards[index];
                    return
FadeTransition(
                      opacity:
_fadeAnimation,
                      child:
SlideTransition(
                        position:
_slideAnimation,
                        child: Container(
                          width: 300,
                          margin:
EdgeInsets.only(right: 16),
                          decoration:
BoxDecoration(
                            gradient:
LinearGradient(
                              colors:
[Colors.grey[800]!, Colors.grey[900]!],
                              begin:
Alignment.topLeft,
                              end:
Alignment.bottomRight,
                            ),
                            borderRadius:
BorderRadius.circular(16),
                            boxShadow: [
                              BoxShadow(
                                color:
Colors.black26,
```

```
blurRadius: 10,
                                offset:
Offset(0, 4),
                              ),
                            ],
                          ),
                          padding:
EdgeInsets.all(20),
                          child: Column(

crossAxisAlignment:
CrossAxisAlignment.start,

mainAxisAlignment:
MainAxisAlignment.spaceBetween,
                            children: [
                              Row(

mainAxisAlignment:
MainAxisAlignment.spaceBetween,
                                children:
[
                                  Text(

card.cardType,

style: TextStyle(

color: Colors.white,

fontSize: 20,

fontWeight: FontWeight.bold,
                                    ),
                                  ),

Icon(Icons.credit_card, color:
Colors.white),
                                ],
                              ),
                              Text(

card.cardNumber,
                                style:
TextStyle(
                                  color:
Colors.white,

fontSize: 24,

letterSpacing: 2.0,

fontFamily: 'Courier',
                                ),
                              ),
                              Row(

mainAxisAlignment:
MainAxisAlignment.spaceBetween,
                                children:
[
                                  Text(

'Balance',

style: TextStyle(color: Colors.white70,
fontSize: 16),
                                  ),
                                  Text(

'\$${card.balance.toStringAsFixed(2)}',

style: TextStyle(

color: Colors.white,

fontSize: 24,

fontWeight: FontWeight.bold,
                                    ),
                                  ),
                                ],
                              ),
                            ],
                          ),
                        ),
                      );
                    },
                  ),
                ),
                SizedBox(height: 20),
                Column(
                  crossAxisAlignment:
CrossAxisAlignment.stretch,
                  children: [
                    GestureDetector(
                      onTap: () async {
                        final result =
await Navigator.push(
                          context,

MaterialPageRoute(builder: (context) =>
CreditScreen()),
                        );
                        if (result != null
&& result is double) {
                          final
balanceResponse = await
ApiService().getBalance();
                          setState(() {
                            user =
User.fromJson(user.username,
balanceResponse, user.profile);
                          });
                        }
                      },
                      child: Container(
                        padding:
EdgeInsets.symmetric(vertical: 16),
```

```
                    decoration:
BoxDecoration(
                      gradient:
LinearGradient(
                        colors:
[Colors.blue[700]!, Colors.blue[900]!],
                        begin:
Alignment.topLeft,
                        end:
Alignment.bottomRight,
                      ),
                      borderRadius:
BorderRadius.circular(12),
                      boxShadow: [
                        BoxShadow(
                          color:
Colors.black26,
                          blurRadius:
8,
                          offset:
Offset(0, 4),
                        ),
                      ],
                    ),
                    child: Center(
                      child: Text(
                        'Отримати
кредит',
                        style:
TextStyle(
                          color:
Colors.white,
                          fontSize: 16,
                          fontWeight:
FontWeight.bold,
                        ),
                      ),
                    ),
                  ),
                ),
                SizedBox(height: 10),
                GestureDetector(
                  onTap: () {
                    Navigator.push(
                      context,

MaterialPageRoute(builder: (context) =>
CashbackScreen()),
                    );
                  },
                  child: Container(
                    padding:
EdgeInsets.symmetric(vertical: 16),
                    decoration:
BoxDecoration(
                      gradient:
LinearGradient(
                        colors:
[Colors.green[700]!, Colors.green[900]!],
                        begin:
Alignment.topLeft,
                        end:
Alignment.bottomRight,
                      ),
                      borderRadius:
BorderRadius.circular(12),
                      boxShadow: [
                        BoxShadow(
                          color:
Colors.black26,
                          blurRadius:
8,
                          offset:
Offset(0, 4),
                        ),
                      ],
                    ),
                    child: Center(
                      child: Text(
                        'Кешбек',
                        style:
TextStyle(
                          color:
Colors.white,
                          fontSize: 16,
                          fontWeight:
FontWeight.bold,
                        ),
                      ),
                    ),
                  ),
                ),
                SizedBox(height: 10),
                GestureDetector(
                  onTap: () async {
                    final result =
await Navigator.push(
                      context,

MaterialPageRoute(
                        builder:
(context) => TransferScreen(user: user),
                      ),
                    );
                    if (result != null
&& result is List<Card>) {
                      setState(() {
                        user =
User.fromJson(user.username, result,
user.profile);
                      });
                    }
                  },
                  child: Container(
                    padding:
EdgeInsets.symmetric(vertical: 16),
                    decoration:
BoxDecoration(
                      gradient:
LinearGradient(
```

```
                        colors:
[Colors.orange[700]!,
Colors.orange[900]!],
                        begin:
Alignment.topLeft,
                        end:
Alignment.bottomRight,
                      ),
                      borderRadius:
BorderRadius.circular(12),
                      boxShadow: [
                        BoxShadow(
                          color:
Colors.black26,
                          blurRadius:
8,
                          offset:
Offset(0, 4),
                        ),
                      ],
                    ),
                    child: Center(
                      child: Text(
                        'Переказ',
                        style:
TextStyle(
                          color:
Colors.white,
                          fontSize: 16,
                          fontWeight:
FontWeight.bold,
                        ),
                      ),
                    ),
                  ),
                ),
                SizedBox(height: 10),
                GestureDetector(
                  onTap: () {
                    Navigator.push(
                      context,

MaterialPageRoute(
                        builder:
(context) => HistoryScreen(cards:
user.cards),
                      ),
                    );
                  },
                  child: Container(
                    padding:
EdgeInsets.symmetric(vertical: 16),
                    decoration:
BoxDecoration(
                      gradient:
LinearGradient(
                        colors:
[Colors.purple[700]!,
Colors.purple[900]!],
                        begin:
Alignment.topLeft,
                        end:
Alignment.bottomRight,
                      ),
                      borderRadius:
BorderRadius.circular(12),
                      boxShadow: [
                        BoxShadow(
                          color:
Colors.black26,
                          blurRadius:
8,
                          offset:
Offset(0, 4),
                        ),
                      ],
                    ),
                    child: Center(
                      child: Text(
                        'Історія
транзакцій',
                        style:
TextStyle(
                          color:
Colors.white,
                          fontSize: 16,
                          fontWeight:
FontWeight.bold,
                        ),
                      ),
                    ),
                  ),
                ),
              ),
              SizedBox(height: 10),
              GestureDetector(
                onTap: () {
                  showDialog(
                    context: context,
                    builder:
(context) {
                      String
selectedCardType = 'Visa';
                      return
AlertDialog(
                        title:
Text('Додати картку'),
                        content:
DropdownButton<String>(
                          value:
selectedCardType,
                          onChanged:
(value) {

selectedCardType = value!;
                          },
                          items:
['Visa', 'Mastercard']

.map((type) => DropdownMenuItem(

value: type,
```

```
child: Text(type),
                                    ))
.toList(),
                            ),
                            actions: [
                              TextButton(
onPressed: () {
Navigator.pop(context);
                                },
                                child:
Text('Скасувати'),
                              ),
                              TextButton(
onPressed: () {
_createCard(selectedCardType);
Navigator.pop(context);
                                },
                                child:
Text('Додати'),
                              ),
                            ],
                          );
                        },
                      );
                    },
                    child: Container(
                      padding:
EdgeInsets.symmetric(vertical: 16),
                      decoration:
BoxDecoration(
                        gradient:
LinearGradient(
                          colors:
[Colors.teal[700]!, Colors.teal[900]!],
                          begin:
Alignment.topLeft,
                          end:
Alignment.bottomRight,
                        ),
                        borderRadius:
BorderRadius.circular(12),
                        boxShadow: [
                          BoxShadow(
                            color:
Colors.black26,
                            blurRadius:
8,
                            offset:
Offset(0, 4),
                          ),
                        ],
                      ),
                      child: Center(
                        child: Text(
                          'Додати
картку',
                          style:
TextStyle(
                            color:
Colors.white,
                            fontSize: 16,
                            fontWeight:
FontWeight.bold,
                          ),
                        ),
                      ),
                    ),
                  ),
                ),
              ),
            ],
          ),
        ),
      ),
    );
  }
}
// ЕКРАН ПРОФІЛЮ
import 'package:flutter/material.dart';
import '../models/user.dart';
import '../services/api_service.dart';

class ProfileScreen extends
StatefulWidget {
  final User user;

  ProfileScreen({required this.user});

  @override
  _ProfileScreenState createState() =>
_ProfileScreenState();
}

class _ProfileScreenState extends
State<ProfileScreen> {
  final _firstNameController =
TextEditingController();
  final _secondNameController =
TextEditingController();
  final _emailController =
TextEditingController();
  final _ageController =
TextEditingController();
  final _genderController =
TextEditingController();
  bool isEditing = false;

  @override
  void initState() {
    super.initState();
    final profile = widget.user.profile;
    _firstNameController.text =
profile?.firstName ?? '';
    _secondNameController.text =
profile?.secondName ?? '';
```

```
    _emailController.text =
profile?.email ?? '';
    _ageController.text =
profile?.age?.toString() ?? '';
    _genderController.text =
profile?.gender ?? '';
  }

  void _updateProfile() async {
    try {
      final profile = Profile(
        firstName:
_firstNameController.text,
        secondName:
_secondNameController.text,
        email: _emailController.text,
        age:
int.tryParse(_ageController.text),
        gender: _genderController.text,
      );
      await
ApiService().updateProfile(profile);

ScaffoldMessenger.of(context).showSnackBa
r(
        SnackBar(content: Text('Профіль
оновлено')),
      );
      setState(() {
        isEditing = false;
      });
    } catch (e) {

ScaffoldMessenger.of(context).showSnackBa
r(
        SnackBar(content: Text('Помилка
оновлення профілю: $e')),
      );
    }
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      backgroundColor: Colors.black,
      appBar: AppBar(
        title: Text('Профіль', style:
TextStyle(color: Colors.white)),
        backgroundColor: Colors.black,
      ),
      body: Padding(
        padding: EdgeInsets.all(16.0),
        child: Column(
          crossAxisAlignment:
CrossAxisAlignment.start,
          children: [
            Text(
              'Інформація користувача',
              style: TextStyle(color:
Colors.white, fontSize: 20),
            ),
            SizedBox(height: 16),
            TextField(
              controller:
_firstNameController,
              decoration:
InputDecoration(
                labelText: 'Ім\'я',
                labelStyle:
TextStyle(color: Colors.white70),
                enabledBorder:
OutlineInputBorder(
                  borderSide:
BorderSide(color: Colors.white70),
                ),
                focusedBorder:
OutlineInputBorder(
                  borderSide:
BorderSide(color: Colors.white),
                ),
              ),
              style: TextStyle(color:
Colors.white),
              enabled: isEditing,
            ),
            SizedBox(height: 16),
            TextField(
              controller:
_secondNameController,
              decoration:
InputDecoration(
                labelText: 'Прізвище',
                labelStyle:
TextStyle(color: Colors.white70),
                enabledBorder:
OutlineInputBorder(
                  borderSide:
BorderSide(color: Colors.white70),
                ),
                focusedBorder:
OutlineInputBorder(
                  borderSide:
BorderSide(color: Colors.white),
                ),
              ),
              style: TextStyle(color:
Colors.white),
              enabled: isEditing,
            ),
            SizedBox(height: 16),
            TextField(
              controller:
_emailController,
              decoration:
InputDecoration(
                labelText: 'Електронна
пошта',
                labelStyle:
TextStyle(color: Colors.white70),
                enabledBorder:
OutlineInputBorder(
```

```
                  borderSide:
BorderSide(color: Colors.white70),
                ),
                focusedBorder:
OutlineInputBorder(
                  borderSide:
BorderSide(color: Colors.white),
                ),
              ),
              style: TextStyle(color:
Colors.white),
              enabled: isEditing,
            ),
            SizedBox(height: 16),
            TextField(
              controller: _ageController,
              decoration:
InputDecoration(
                labelText: 'Вік',
                labelStyle:
TextStyle(color: Colors.white70),
                enabledBorder:
OutlineInputBorder(
                  borderSide:
BorderSide(color: Colors.white70),
                ),
                focusedBorder:
OutlineInputBorder(
                  borderSide:
BorderSide(color: Colors.white),
                ),
              ),
              style: TextStyle(color:
Colors.white),
              keyboardType:
TextInputType.number,
              enabled: isEditing,
            ),
            SizedBox(height: 16),
            TextField(
              controller:
_genderController,
              decoration:
InputDecoration(
                labelText: 'Стать',
                labelStyle:
TextStyle(color: Colors.white70),
                enabledBorder:
OutlineInputBorder(
                  borderSide:
BorderSide(color: Colors.white70),
                ),
                focusedBorder:
OutlineInputBorder(
                  borderSide:
BorderSide(color: Colors.white),
                ),
              ),
              style: TextStyle(color:
Colors.white),
              enabled: isEditing,
            ),
            SizedBox(height: 20),
            ElevatedButton(
              onPressed: isEditing ?
_updateProfile : () => setState(() =>
isEditing = true),
              style:
ElevatedButton.styleFrom(
                backgroundColor:
Colors.white,
                foregroundColor:
Colors.black,
              ),
              child: Text(isEditing ?
'Зберегти' : 'Редагувати'),
            ),
            SizedBox(height: 10),
            ElevatedButton(
              onPressed: () =>
Navigator.pop(context),
              style:
ElevatedButton.styleFrom(
                backgroundColor:
Colors.white,
                foregroundColor:
Colors.black,
              ),
              child: Text('Назад'),
            ),
          ],
        ),
      ),
    );
  }
}
// МЕЙН ФАЙЛ
import 'package:flutter/material.dart';
import 'screens/login_screen.dart';

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Monobank Clone',
      theme: ThemeData(primarySwatch:
Colors.blue),
      home: LoginScreen(),
    );
  }
}
// ТРАНЗАКЦІЇ
import 'package:flutter/material.dart';
import '../models/user.dart';
import '../services/api_service.dart';

class TransferScreen extends
StatefulWidget {
```

```
  final User user;

  TransferScreen({required this.user});

  @override
  _TransferScreenState createState() =>
_TransferScreenState();
}

class _TransferScreenState extends
State<TransferScreen> {
  final _toCardNumberController =
TextEditingController();
  final _amountController =
TextEditingController();
  String? selectedFromCardNumber;

  void _transfer() async {
    try {
      double amount =
double.parse(_amountController.text);
      if (selectedFromCardNumber == null)
{
        throw Exception('Виберіть картку
для переказу');
      }
      await ApiService().transfer(
        selectedFromCardNumber!,
        _toCardNumberController.text,
        amount,
      );
      final balanceResponse = await
ApiService().getBalance();

ScaffoldMessenger.of(context).showSnackBa
r(
        SnackBar(content: Text('Переказ
виконано успішно')),
      );
      Navigator.pop(context,
balanceResponse);
    } catch (e) {

ScaffoldMessenger.of(context).showSnackBa
r(
        SnackBar(content: Text('Помилка
переказу: $e')),
      );
    }
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      backgroundColor: Colors.black,
      appBar: AppBar(
        title: Text('Переказ коштів',
style: TextStyle(color: Colors.white)),
        backgroundColor: Colors.black,
      ),
      body: Padding(
        padding: EdgeInsets.all(16.0),
        child: Column(
          crossAxisAlignment:
CrossAxisAlignment.start,
          children: [
            DropdownButton<String>(
              value:
selectedFromCardNumber,
              hint: Text('Виберіть
картку', style: TextStyle(color:
Colors.white70)),
              dropdownColor:
Colors.grey[900],
              style: TextStyle(color:
Colors.white),
              onChanged: (value) {
                setState(() {
                  selectedFromCardNumber
= value;
                });
              },
              items: widget.user.cards
                  .map((card) =>
DropdownMenuItem(
                        value:
card.cardNumber,
                        child:
Text('${card.cardType}
${card.cardNumber}'),
                      ))
                  .toList(),
            ),
            SizedBox(height: 20),
            TextField(
              controller:
_toCardNumberController,
              decoration:
InputDecoration(
                labelText: 'Номер картки
отримувача',
                labelStyle:
TextStyle(color: Colors.white70),
                enabledBorder:
OutlineInputBorder(
                  borderSide:
BorderSide(color: Colors.white70),
                ),
                focusedBorder:
OutlineInputBorder(
                  borderSide:
BorderSide(color: Colors.white),
                ),
              ),
              style: TextStyle(color:
Colors.white),
            ),
            SizedBox(height: 16),
            TextField(
              controller:
_amountController,
```

```
              decoration:
InputDecoration(
                labelText: 'Сума',
                labelStyle:
TextStyle(color: Colors.white70),
                enabledBorder:
OutlineInputBorder(
                  borderSide:
BorderSide(color: Colors.white70),
                ),
                focusedBorder:
OutlineInputBorder(
                  borderSide:
BorderSide(color: Colors.white),
                ),
              ),
              keyboardType:
TextInputType.number,
              style: TextStyle(color:
Colors.white),
            ),
            SizedBox(height: 20),
            ElevatedButton(
              onPressed: _transfer,
              style:
ElevatedButton.styleFrom(
                backgroundColor:
Colors.white,
                foregroundColor:
Colors.black,
                padding:
EdgeInsets.symmetric(vertical: 16),
                shape:
RoundedRectangleBorder(
                  borderRadius:
BorderRadius.circular(12),
                ),
              ),
              child: Text('Виконати
переказ'),
            ),
          ],
        ),
      ),
    );
  }
}
// МОДЕЛЬ КАРТКИ
class User {
  final String username;
  final List<Card> cards;
  final Profile? profile;

  User({
    required this.username,
    required this.cards,
    this.profile,
  });

  factory User.fromJson(String username,
List<dynamic> balanceJson, Profile?
profile) {
    return User(
      username: username,
      cards: balanceJson.map((json) =>
Card.fromJson(json)).toList(),
      profile: profile,
    );
  }
}

class Card {
  final int cardId;
  final String cardNumber;
  final double balance;
  final String cardType;
  final double creditLimit;
  final double creditInterestRate;

  Card({
    required this.cardId,
    required this.cardNumber,
    required this.balance,
    required this.cardType,
    required this.creditLimit,
    required this.creditInterestRate,
  });

  // Витрачені кредитні кошти (якщо
баланс від'ємний)
  double get usedCredit => balance < 0 ?
-balance : 0;

  // Доступний кредитний ліміт
  double get availableCredit =>
creditLimit - usedCredit;

  factory Card.fromJson(Map<String,
dynamic> json) {
    return Card(
      cardId: json['cardId'],
      cardNumber: json['cardNumber'],
      balance:
json['balance'].toDouble(),
      cardType: json['cardType'],
      creditLimit:
json['creditLimit'].toDouble(),
      creditInterestRate:
json['creditInterestRate'].toDouble(),
    );
  }
}

class Profile {
  final String? firstName;
  final String? secondName;
  final String? email;
  final int? age;
  final String? gender;
```

```
  Profile({
    this.firstName,
    this.secondName,
    this.email,
    this.age,
    this.gender,
  });

  factory Profile.fromJson(Map<String,
dynamic> json) {
    return Profile(
      firstName: json['firstName'],
      secondName: json['secondName'],
      email: json['email'],
      age: json['age'],
      gender: json['gender'],
    );
  }

  Map<String, dynamic> toJson() {
    return {
      'firstName': firstName,
      'secondName': secondName,
      'email': email,
      'age': age,
      'gender': gender,
    };
  }
}
// РЕГІСТРАЦІЯ
import 'package:flutter/material.dart';
import '../services/api_service.dart';
import 'login_screen.dart';

class RegisterScreen extends
StatefulWidget {
  @override
  _RegisterScreenState createState() =>
_RegisterScreenState();
}

class _RegisterScreenState extends
State<RegisterScreen> {
  final _phoneController =
TextEditingController();
  final _passwordController =
TextEditingController();
  bool isLoading = false;
  String? errorMessage;

  void _register() async {
    setState(() {
      isLoading = true;
      errorMessage = null;
    });
    try {
      await ApiService().register(
        _phoneController.text,
        _passwordController.text,
      );
      Navigator.pushReplacement(
        context,
        MaterialPageRoute(builder:
(context) => LoginScreen()),
      );
    } catch (e) {
      setState(() {
        errorMessage = 'Помилка
реєстрації: $e';
      });
    } finally {
      setState(() {
        isLoading = false;
      });
    }
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      backgroundColor: Colors.black,
      appBar: AppBar(
        title: Text('Реєстрація', style:
TextStyle(color: Colors.white)),
        backgroundColor: Colors.black,
      ),
      body: Padding(
        padding: EdgeInsets.all(16.0),
        child: Column(
          children: [
            TextField(
              controller:
_phoneController,
              decoration:
InputDecoration(
                labelText: 'Номер
телефону',
                labelStyle:
TextStyle(color: Colors.white70),
                enabledBorder:
OutlineInputBorder(
                  borderSide:
BorderSide(color: Colors.white70),
                ),
                focusedBorder:
OutlineInputBorder(
                  borderSide:
BorderSide(color: Colors.white),
                ),
              ),
              style: TextStyle(color:
Colors.white),
              keyboardType:
TextInputType.phone,
            ),
            SizedBox(height: 16),
            TextField(
              controller:
_passwordController,
              decoration:
InputDecoration(
                labelText: 'Пароль',
```

```
                labelStyle:
TextStyle(color: Colors.white70),
                enabledBorder:
OutlineInputBorder(
                  borderSide:
BorderSide(color: Colors.white70),
                ),
                focusedBorder:
OutlineInputBorder(
                  borderSide:
BorderSide(color: Colors.white),
                ),
              ),
              obscureText: true,
              style: TextStyle(color:
Colors.white),
            ),
            SizedBox(height: 20),
            if (errorMessage != null)
              Padding(
                padding:
EdgeInsets.only(bottom: 10),
                child: Text(
                  errorMessage!,
                  style: TextStyle(color:
Colors.red, fontSize: 14),
                ),
              ),
            isLoading
                ?
CircularProgressIndicator(color:
Colors.white)
                : ElevatedButton(
                    onPressed: _register,
                    style:
ElevatedButton.styleFrom(
                      backgroundColor:
Colors.white,
                      foregroundColor:
Colors.black,
                    ),
                    child:
Text('Зареєструватися'),
                  ),
            TextButton(
              onPressed: () {

Navigator.pushReplacement(
                  context,

MaterialPageRoute(builder: (context) =>
LoginScreen()),
                );
              },
              child: Text(
                'Увійти',
                style: TextStyle(color:
Colors.white70),
              ),
            ),
          ],
        ),
      ),
    );
  }
}
// ІСТОРІЯ ТРАНЗАКЦІЙ
import 'package:flutter/material.dart';
import '../services/api_service.dart';
import '../models/transaction.dart';
import '../models/user.dart';

class HistoryScreen extends
StatefulWidget {
  final List<Card> cards;

  HistoryScreen({required this.cards});

  @override
  _HistoryScreenState createState() =>
_HistoryScreenState();
}

class _HistoryScreenState extends
State<HistoryScreen> {
  List<Transaction> transactions = [];
  bool isLoading = true;
  int? selectedCardId;

  @override
  void initState() {
    super.initState();
    if (widget.cards.isNotEmpty) {
      selectedCardId =
widget.cards[0].cardId;
      _loadHistory();
    } else {
      setState(() {
        isLoading = false;
      });
    }
  }

  void _loadHistory() async {
    if (selectedCardId == null) return;
    try {
      final history = await
ApiService().getTransactionHistory(select
edCardId!);
      setState(() {
        transactions = history;
        isLoading = false;
      });
    } catch (e) {

ScaffoldMessenger.of(context).showSnackBa
r(
        SnackBar(content: Text('Помилка
завантаження історії: $e')),
      );
      setState(() {
        isLoading = false;
```

```
      });
    }
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      backgroundColor: Colors.black,
      appBar: AppBar(
        title: Text('Історія транзакцій',
style: TextStyle(color: Colors.white)),
        backgroundColor: Colors.black,
      ),
      body: Column(
        children: [
          if (widget.cards.isNotEmpty)
            DropdownButton<int>(
              value: selectedCardId,
              dropdownColor:
Colors.grey[900],
              style: TextStyle(color:
Colors.white),
              onChanged: (value) {
                setState(() {
                  selectedCardId = value;
                  isLoading = true;
                  _loadHistory();
                });
              },
              items: widget.cards
                  .map((card) =>
DropdownMenuItem(
                        value:
card.cardId,
                        child:
Text('${card.cardType}
${card.cardNumber}'),
                      ))
                  .toList(),
            ),
          Expanded(
            child: isLoading
                ? Center(child:
CircularProgressIndicator(color:
Colors.white))
                : transactions.isEmpty
                    ? Center(
                        child: Text(
                          'Транзакції
відсутні',
                          style:
TextStyle(color: Colors.white, fontSize:
18),
                        ),
                      )
                    : ListView.builder(
                        padding:
EdgeInsets.all(16.0),
                        itemCount:
transactions.length,
                        itemBuilder:
(context, index) {
                          final
transaction = transactions[index];
                          return Card(
                            color:
Colors.grey[850],
                            shape:
RoundedRectangleBorder(

borderRadius: BorderRadius.circular(12),
                            ),
                            child:
ListTile(
                              title:
Text(

'${transaction.transactionType}:
\$${transaction.amount.toStringAsFixed(2)
}',
                                style:
TextStyle(color: Colors.white),
                              ),
                              subtitle:
Text(

transaction.transactionDate.toString(),
                                style:
TextStyle(color: Colors.white70),
                              ),
                              trailing:
Text(

'#${transaction.transactionId}',
                                style:
TextStyle(color: Colors.white70),
                              ),
                            );
                          },
                        ),
          ),
        ],
      ),
    );
  }
}
```