НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
В.о. завідувача кафедри
_____ Михайло НОВОТАРСЬКИЙ
(підпис)

“__” _____2025 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою “Комп’ютерні системи та мережі”

спеціальності 123 “Комп’ютерна інженерія”

на тему: Банківська система на основі мікросервісної архітектури

Виконав: студент __4__ курсу, групи __ІО-13__
(шифр групи)

_____Казак Вадим Сергійович _____
(прізвище, ім’я, по батькові) (підпис)

Керівник _____асистент, Русінов В. В._____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) асистент, Пономаренко А. М._____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент __доцент, Шимкович В. М._____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2025 р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Комп’ютерні системи та мережі”

спеціальності 123 “Комп’ютерна інженерія”

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ Михайло НОВОТАРСЬКИЙ
(підпис)

“__” _____2025 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Казака Вадима Сергійовича

1. Тема проєкту *Банківська система на основі мікросервісної архітектури*
керівник проєкту Русінов Володимир Володимирович, асистент,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від 23 травня 2025 року №1705-с

2. Термін здачі студентом закінченого проєкту *7 червня 2025 р.*

3. Вихідні дані до проєкту *технічна документація, теоретичні дані.*

4. Зміст пояснювальної записки (перелік питань, які розробляються)

Розділ 1. Огляд та аналіз предметної області.

Розділ 2. Етап проєктування системи.

Розділ 3. Етап розробки системи.

Розділ 4. Дослідження та тестування розробленої системи.

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) схема архітектури системи, схема баз даних, алгоритм роботи системи.

6. Консультанти розділів проєкту

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Пономаренко А.М.		

7. Дата видачі завдання 24 січня 2025 р.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	*24.10.2024 - 26.10.2024*	
2.	*Вивчення та аналіз завдання*	*14.04.2025 - 21.04.2025*	
3.	*Розробка архітектури та загальної структури системи*	*22.04.2025 - 28.04.2025*	
4.	*Розробка структур окремих підсистем*	*29.04.2025 - 03.05.2025*	
5.	*Програмна реалізація системи*	*04.05.2025 - 18.05.2025*	
6.	*Оформлення пояснювальної*	*11.05.2025 - 01.06.2025*	
7.	*Захист програмного продукту*	*18.05.2025*	
8.	*Передзахист*	*02.06.2025*	
9.	*Захист*	*16.06.2025*	

Студент _____ Вадим КАЗАК
(підпис)

Керівник проєкту _____ Володимир РУСІНОВ
(підпис)

АНОТАЦІЯ

У даній дипломній роботі розглянуто процес проєктування та розробки сучасної банківської інформаційної системи на основі мікросервісної архітектури. Проведено аналіз сучасного стану банківського сектору, тенденцій цифрової трансформації та вимог до фінансового програмного забезпечення. Обґрунтовано доцільність використання сервісно-орієнтованого підходу в умовах високої динаміки ринку та зростання вимог до масштабованості, безпеки й доступності систем. Було спроєктовано архітектуру системи, яка складається з незалежних мікросервісів. Реалізовано механізми автентифікації із використанням сесій, зберігання даних у PostgreSQL, кешування у Redis, обмін повідомленнями через RabbitMQ. Основна частина готового продукту була написана за допомогою TypeScript та фреймворку NestJS.

Ключові слова: банківська система, мікросервісна архітектура, TypeScript, NestJS, автентифікація, Redis, PostgreSQL, RabbitMQ, безпека, KYC, Nginx.

ANNOTATION

This diploma work describes the process of designing and developing a modern banking system based on microservice architecture. An analysis of the current state of the banking sector, digital transformation trends, and requirements for financial software was carried out. The expediency of using a service-oriented approach in the context of high market dynamics and growing requirements for scalability, security and availability of systems is substantiated. The system architecture consisting of independent microservices was designed. Authentication mechanisms by using sessions, data storage in PostgreSQL, caching in Redis and messaging via RabbitMQ were implemented. The main part of the finished product was written using TypeScript and the NestJS framework.

Keywords: banking system, microservice architecture, TypeScript, NestJS, authentication, Redis, PostgreSQL, RabbitMQ, security, KYC, Nginx.

ОПИС АЛЬБОМУ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Банківська система на основі мікросервісної архітектури*»

Київ – 2025

Довідки	Формат	*Значення*	*Найменування*	Кіл. листів	№ екземпляра	*Додаток*
			Документація загальна			
			Знову розроблена			
	А4	***ІАЛЦ.467200.002 ТЗ***	Банківська система на основі	4		
			мікросервісної архітектури			
			Технічне завдання			
	А4	***ІАЛЦ.467200.003 ПЗ***	Банківська система на основі	66		
			мікросервісної архітектури			
			Пояснювальна записка			
	А4	***ІАЛЦ.467200.004 Д1***	Банківська система на основі	1		
			мікросервісної архітектури			
			Схема архітектури системи			
	А4	***ІАЛЦ.467200.005 Д2***	Банківська система на основі	1		
			мікросервісної архітектури			
			Схема баз даних			
	А4	***ІАЛЦ.467200.006 Д3***	Банківська система на основі	1		
			мікросервісної архітектури			
			Алгоритм роботи системи			
	А4	***ІАЛЦ.467200.007 Д4***	Банківська система на основі	44		
			мікросервісної архітектури			
			Текст програмного коду			

					ІАЛЦ.467200.001 ОА	
Зм.	*Лист*	*№ докум.*	*Підп.*	*Дата*		
Розроб.		Казак В.С.			*Банківська система на основі мікросервісної архітектури* Опис альбому	Літ. / Аркуш: 1 / Аркушів: 1
Перев.		Русінов В.В.				
Реценз..		Шимкович В.М.				
Н. Контр.		Пономаренко А.М.				***“КПІ ім. Ігоря Сікорського” ФІОТ ІО-13***
Затвердив						

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Банківська система на основі мікросервісної архітектури*»

Київ – 2025

ЗМІСТ

1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання визначає розробку мікросервісної банківської системи. Ця система створена для гарантування безпечного та ефективного оперування даними користувачів та їх платіжними картками.

Галузь застосування цієї системи охоплює банківський сектор, де вона забезпечує цілісний процес банківських операцій. Це включає в собі управління дебетовими та кредитними картками, верифікацію клієнтів, обслуговування рахунків та проведення фінансових транзакцій.

2. ПРИЧИНИ ДЛЯ РОЗРОБКИ

Підставою розробки такої системи є завдання для виконання бакалаврської роботи освітньо-професійної програми «Комп’ютерні системи та мережі» спеціальності 123 «Комп’ютерна інженерія», затверджене кафедрою «Обчислювальної техніки» Національного технічного Університету України «Київський Політехнічний інститут ім. Ігоря Сікорського».

3. МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою даної дипломної роботи є розробка сучасної банківської системи, що функціонує на основі мікросервісної архітектури. Така система має забезпечувати ефективне управління банківськими операціями та обслуговування клієнтів.

4. ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки дипломного проєкту є навчальні ресурси, наукова література, офіційна документація технологій.

5. ТЕХНІЧНІ ВИМОГИ

5.1 Вимоги до продукту, що розробляється

Розроблена система має виконувати такі вимоги:

- Управління клієнтськими профілями з можливістю реєстрації, верифікації та редагування персональних даних.
- Управління платіжними картками з можливістю випуску, блокування, розблокування та закриття карток.
- Проведення фінансових операцій, таких як перекази з картки на картку.
- Отримання історії операцій переказів коштів.
- Система безпеки та автентифікації з двофакторною верифікацією та захистом від несанкціонованого доступу.
- Масштабованість архітектури з можливістю горизонтального масштабування та додавання нових сервісів.

5.2 Вимоги до програмного забезпечення

- ОС Windows, Mac або Linux.
- Node.js v18.0.0+.
- Docker (WSL).
- PostgreSQL v14.0+.
- Git.
- Visual Studio Code або WebStorm IDE.
- Postman.

5.3 Вимоги до апаратної частини

- ЦП мінімум Intel Core i5 або AMD Ryzen 5 (2.5GHz+)
- Оперативна пам’ять мінімум 8GB (рекомендовано 16GB)
- Вільний дисковий простір мінімально 50GB

6. ЕТАПИ РОЗРОБКИ

№ п/п	Назва етапів виконання	Термін виконання етапу
1.	Затвердження теми проєкту	24.10.2024-26.10.2024
2.	Вивчення та аналіз завдання	14.04.2025-21.04.2025
3.	Розробка архітектури та загальної структури системи	22.04.2025-28.04.2025
4.	Розробка структур окремих частин системи	29.04.2025-03.05.2025
5.	Програмна реалізація системи	04.05.2025-18.05.2025
6.	Оформлення пояснювальної записки	11.05.2025-01.06.2025

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Банківська система на основі мікросервісної архітектури*»

Київ – 2025

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

JS	(Javascript) Мова програмування Javascript
TS	(Typescript) Мова програмування Typescript
API	(Application Programming Interface) Програмний інтерфейс застосунку
ORM	(Object-Relational Mapping) Об’єктно-реляційне відображення
HTTP	(Hypertext Transfer Protocol) Протокол передачі гіпертексту
JSON	(Javascript Object Notation) Формат обміну даними
2FA	(Two-Factor Authentication) Двофакторна авторизація
SOA	(Service-Oriented Architecture) Сервісно-орієнтована архітектура
REST	(Representational State Transfer) Підхід до архітектури мережевих протоколів, які надають доступ до інформаційних ресурсів
TPS	(Transaction Per Second) Максимальну кількість транзакцій, які може обробити система протягом певного часу
KYC	(Know Your Customer) Перевірка інформації про приватних осіб, що відкривають рахунок та проводять грошові операції
PCI DSS	(Payment Card Industry Data Security Standard) Стандарт безпеки даних індустрії платіжних карток, розроблений Радою зі стандартів безпеки індустрії платіжних карток
TTL	(Time To Live) максимальний період часу, за який дані можуть існувати до свого зникнення

ВСТУП

Сучасний банківський сектор перебуває в стані динамічного розвитку та цифрової трансформації. Зростаюча конкуренція, зміна клієнтських очікувань і поява новітніх технологій спонукають фінансові установи впроваджувати інноваційні рішення, спрямовані на підвищення ефективності роботи та покращення якості обслуговування. Важливою умовою сталого розвитку банків є впровадження інформаційних систем, що дозволяють автоматизувати ключові бізнес-процеси, забезпечити надання сучасних цифрових послуг та відповідати зростаючим вимогам до надійності, масштабованості й безпеки.

Актуальність теми дипломної роботи зумовлена потребою у створенні гнучкої та надійної банківської платформи, здатної ефективно обробляти фінансову інформацію, гарантувати захист даних і швидко адаптуватися до змін ринку. Особливе значення має використання сучасних підходів до архітектури програмного забезпечення – зокрема, мікросервісного підходу, який дозволяє забезпечити гнучкість у розробці, підтримці та масштабуванні системи.

Метою дипломної роботи є розробка банківської системи на основі мікросервісної архітектури, яка забезпечуватиме ефективне управління клієнтськими даними, платіжними картками та фінансовими операціями. Система має бути захищеною та продуктивною.

Для досягнення поставленої мети необхідно вирішити наступні завдання:

1. Провести аналіз існуючих банківських систем та визначити їх переваги та недоліки.
2. Розробити архітектуру мікросервісної банківської системи.
3. Реалізувати основні компоненти системи, включаючи сервіси управління клієнтами, рахунками, картками та фінансовими операціями.

4. Забезпечити безпеку системи, включаючи автентифікацію, авторизацію та захист даних.
5. Провести тестування системи та перевірити її відповідність вимогам безпеки, продуктивності та масштабованості.

Об'єктом дослідження є процес розробки сучасної банківської системи на основі мікросервісної архітектури.

Практична значущість роботи полягає в створенні готової до використання банківської системи, призначеної для автоматизації основних банківських процесів і надання сучасних цифрових послуг клієнтам. Розроблене рішення може бути адаптоване до потреб конкретної фінансової установи та легко доповнене додатковими функціональними модулями.

Методологічною основою дослідження є системний підхід, який дає змогу розглядати банківську систему як сукупність взаємопов’язаних компонентів, що спільно забезпечують ефективне функціонування фінансової установи. У процесі розробки та перевірки системи буде застосовано методи аналізу, синтезу, моделювання та тестування.

Структура дипломної роботи складається з чотирьох розділів. У першому розглядаються теоретичні засади побудови банківських інформаційних систем. Другий виділено для опису архітектури та ключових компонентів розробленої системи. У третьому йдеться про реалізацію програмного забезпечення та етапи впровадження. Четвертий розділ містить результати тестування, а також аналіз ефективності та відповідності системи поставленим вимогам. Кожен розділ завершується висновками.

РОЗДІЛ 1
ОГЛЯД ТА АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Огляд предметної області

1.1.1 Сучасний стан банківського сектору

Сьогодні банківський сектор перебуває в стані постійних змін, що зумовлені розвитком цифрових технологій, новими вимогами клієнтів та зростанням конкуренції, зокрема з боку фінтех-компаній. Щоб не втратити позиції на ринку, банки переосмислюють свої стратегії, впроваджують нові сервіси та оновлюють технічну інфраструктуру.

Важливу роль відіграє діджиталізація. При чому не просто як модний тренд, а як реальна необхідність. Онлайн-банкінг, мобільні додатки, можливість оформити кредит або відкрити рахунок за кілька кліків – усе це вже не перевага, а норма. Наприклад, український Monobank став прикладом банку без відділень, який повністю працює в смартфоні.

Паралельно з цим змінюється і поведінка клієнтів. Люди очікують простого, швидкого та персоналізованого обслуговування. Якщо банк не відповідає цим очікуванням, користувач легко змінює його на зручнішу альтернативу.

На цьому тлі традиційні банки стикаються з конкуренцією не лише між собою, а й із новими гравцями – фінтехами, які мають перевагу у швидкості впровадження інновацій. Щоб не втратити клієнтів, класичні банки змушені інвестувати в діджитал-рішення та впроваджувати нові формати взаємодії.

Технології також відіграють вирішальну роль. Штучний інтелект, блокчейн, хмарні обчислення, мікросервісна архітектура – усі ці інструменти стають основою для побудови більш ефективних, масштабованих та клієнт орієнтованих банківських платформ.

У центрі змін є клієнт. Сучасні банки все активніше аналізують потреби користувачів, використовуючи аналітичні інструменти для персоналізації послуг. Це підвищує лояльність клієнтів і дає змогу створювати конкурентоспроможні продукти.

Окрему увагу банки приділяють безпеці. В умовах зростання кібератак інвестиції в кіберзахист і захист персональних даних є пріоритетом для всіх фінансових установ.

1.1.2 Цифрова трансформація банківської галузі

Цифрова трансформація – один із головних чинників змін у банківському секторі. Це не просто модернізація технологій, а глибока перебудова всіх процесів: від способу надання послуг до внутрішньої організації роботи.

Один із ключових елементів – перехід від фізичних відділень до цифрових каналів. Онлайн-банкінг, мобільні застосунки, чат-боти, дистанційна ідентифікація стало невід'ємною частиною обслуговування клієнтів. Тепер користувач може керувати фінансами, не виходячи з дому.

Другий важливий напрямок – робота з даними. Банки накопичують величезні масиви інформації про поведінку та потреби клієнтів. На основі цієї аналітики вони створюють індивідуальні продукти, оптимізують маркетингові кампанії та прогнозують попит.

Автоматизація є ще одним кроком уперед. Штучний інтелект і машинне навчання використовуються для обробки заявок, оцінки ризиків, виявлення шахрайства. Наприклад, системи автоматично блокують підозрілі транзакції або рекомендують продукти на основі поведінки користувача.

Чат-боти на основі AI вже здатні відповідати на типові питання клієнтів, допомагати з оформленням послуг і навіть консультувати з фінансових питань. Це не лише знижує навантаження на операторів, а й прискорює обслуговування.

Модернізація архітектури є важливим аспектом трансформації. Банки переходять на хмарні сервіси, мікросервісну структуру, API-орієнтовану модель, що дозволяє швидко масштабуватися, інтегруватися з партнерами та запускати нові продукти.

1.1.3 Тенденції розвитку банківських послуг

Останні роки показали, що банки більше не можуть діяти за старими схемами. Щоб залишатися актуальними, їм доводиться гнучко реагувати на зміни – від уподобань клієнтів до нових технологій. Саме тому все більшого значення набуває персоналізований підхід, зручність та інтеграція сервісів.

Однією з ключових тенденцій є персоналізація банківських послуг. Фінансові установи все частіше використовують великі обсяги даних про поведінку клієнтів для створення індивідуальних пропозицій і сервісів. Такий підхід забезпечує точніше задоволення потреб користувачів та підвищує рівень їхньої лояльності.

Інша важлива тенденція – активний розвиток цифрових рішень. Банки інвестують у вдосконалення мобільних додатків, онлайн-банкінгу та інших цифрових каналів, що забезпечують швидкий та зручний доступ до фінансових послуг.

Безконтактні платежі та технології NFC набувають все більшого поширення. Вони дозволяють здійснювати транзакції швидко й без фізичного контакту, що стало особливо актуальним в умовах пандемії та підвищеного попиту на гігієнічні рішення.

Один з найцікавіших напрямків – Open Banking. Ця модель передбачає, що банки відкривають доступ до своїх API для сторонніх розробників. У результаті виникають сервіси, де користувач може керувати рахунками з різних банків в одному додатку. Наприклад, у Європі так працюють додатки

типу Tink або Yapily, а в Україні цю концепцію поступово впроваджують деякі необанки.

1.2 Аналіз існуючих банківських систем

1.2.1 Огляд популярних банківських платформ

Банківський сектор України активно впроваджує сучасні цифрові технології, що сприяє появі інноваційних банківських платформ, орієнтованих на зручність, безпеку та широкий функціонал для клієнтів. Серед найбільш популярних банківських платформ в Україні можна виділити такі:

1. *Приват24 (ПриватБанк)*

Приват24 – це флагманська онлайн-платформа найбільшого банку України, яка обслуговує мільйони фізичних та юридичних осіб. Платформа доступна у вигляді веб-версії та мобільного додатку для iOS та Android [1].

Основні можливості:

- Перекази коштів між картками та рахунками, у тому числі міжнародні SWIFT-перекази.
- Оплата комунальних послуг, штрафів, податків, поповнення мобільного.
- Відкриття депозитів, оформлення кредитів, управління кредитними лімітами.
- Інтеграція з Apple Pay, Google Pay.
- Можливість створення віртуальних карток для онлайн-платежів.
- Високий рівень безпеки: двофакторна автентифікація, push-повідомлення про всі операції, можливість блокування картки в один клік.

2. *monobank*

monobank – перший в Україні мобільний банк без фізичних відділень, який працює на базі Universal Bank [2].

Особливості платформи:

- Відкриття рахунку та отримання картки повністю онлайн, без відвідування відділення.
- Зручний мобільний додаток з інтуїтивним інтерфейсом.
- Миттєві перекази, оплата послуг, поповнення мобільного, оплата штрафів та податків.
- Кредитні ліміти, розстрочка, кешбек за покупки, накопичувальні рахунки.
- Вбудований фінансовий менеджер: категоризація витрат, аналітика, цілі накопичення.
- Високий рівень клієнтської підтримки через чат у додатку. Інтеграція з Apple Pay, Google Pay, можливість випуску віртуальних карток.
- Оригінальні фішки: «банка» для спільних накопичень, ігрові механіки, персоналізовані картки.

3. *Sense Bank*

Sense Bank – це сучасна цифрова платформа, яка поєднує банківські та небанківські сервіси [3].

Особливості:

- Миттєве відкриття рахунків, оформлення карток, кредитів, депозитів.
- Платежі, перекази, оплата послуг, інвестиції у державні облігації.
- Додаткові сервіси: купівля квитків, оплата штрафів, замовлення таксі, бронювання готелів.
- Персоналізовані пропозиції, кешбек, програми лояльності.
- Високий рівень безпеки: біометрична автентифікація, push-сповіщення.

- Інтеграція з Apple Pay, Google Pay, можливість випуску віртуальних карток.

4. *Ощад 24/7*

Ощад 24/7 – це онлайн-платформа одного з найбільших державних банків України [4].

Ключові функції:

- Доступ до рахунків, карток, депозитів, кредитів через веб-інтерфейс та мобільний додаток.
- Оплата комунальних послуг, перекази між картками, поповнення мобільного.
- Можливість відкриття депозитів та оформлення кредитів онлайн. Інтеграція з державними сервісами.
- Високий рівень захисту даних, SMS- та push-повідомлення про операції. Можливість налаштування регулярних платежів та автоплатежів.

1.2.2 Порівняльний аналіз архітектурних рішень

Архітектура банківських платформ є одним із ключових чинників, що визначає гнучкість, масштабованість, надійність системи та її здатність до впровадження інновацій. У сучасному банківському секторі України переважають два основні підходи до побудови архітектури – монолітна та мікросервісна [5]. Кожен із них має свої сильні та слабкі сторони. У таблицях нижче розглянуті основні переваги та недоліки обох підходів.

1. *Монолітна архітектура*

Традиційно більшість банківських систем будувалися за монолітною архітектурою, коли всі функціональні модулі (клієнтський кабінет, платіжний шлюз, управління рахунками, аналітика тощо) інтегровані в єдину програму.

Монолітна архітектура передбачає, що всі частини системи розробляються, деплояться і масштабуються як одне ціле [6]. Такий підхід забезпечує простішу початкову реалізацію та зручність взаємодії між модулями на ранніх етапах. Однак зі зростанням обсягу коду та кількості функціональних компонентів система стає складнішою в обслуговуванні. Зміна або помилка в одному компоненті може вплинути на стабільність усієї системи. Переваги та недоліки такої архітектури наведені в табл. 1.1.

Таблиця 1.1 – Переваги та недоліки монолітної архітектури

Переваги	Недоліки
1. Простота розгортання на початкових етапах 2. Відсутність складної міжсервісної взаємодії 3. Може використовуватись одна база даних 4. Простота розробки	1. Складність масштабування окремих компонентів 2. Висока залежність між модулями 3. Ускладнення інтеграції із зовнішніми API 4. Вразливість до збоїв

2. *Мікросервісна архітектура (SOA)*

У відповідь на виклики масштабованості, продуктивності та частих оновлень, сучасні банки, такі як Monobank, Sense Bank, ПриватБанк, активно впроваджують мікросервісну архітектуру. Вона базується на ідеї розділення всієї системи на окремі сервіси, кожен з яких виконує чітко визначену бізнес-функцію (наприклад, сервіс платежів, сервіс користувачів, сервіс аналітики тощо). Кожен мікросервіс розгортається незалежно, може мати власну базу даних, та спілкується з іншими сервісами через стандартизовані API [7]. Такий підхід спрощує масштабування, прискорює розгортання оновлень та зменшує залежність між командами розробки.

Таблиця 1.2 – Переваги та недоліки мікросервісної архітектури

Переваги	Недоліки
1. Гнучке горизонтальне масштабування 2. Незалежне розгортання та оновлення окремих компонентів без зупинки всієї системи 3. Вища стійкість до збоїв 4. Легкість інтеграції з зовнішніми системами через API 5. Можливість використання різних технологій для різних сервісів	1. Ускладнення адміністрування, потреба у впровадженні систем оркестрації 2. Необхідність забезпечення надійної міжсервісної взаємодії 3. Складність масштабування окремих компонентів 4. Складніша організація безпеки та моніторингу

1.2.3 Переваги та недоліки існуючих банківських систем

Банківські системи в Україні демонструють значний рівень розвитку, активну цифрову трансформацію та впровадження інновацій. Попри це, між різними платформами існують суттєві відмінності щодо рівня автоматизації, функціональності, стабільності та клієнтського досвіду. Узагальнено основні переваги та недоліки таких систем у таблиці 1.3.

Таблиця 1.3 – Переваги та недоліки існуючих банківських систем

Переваги	Недоліки
1. Розвинена цифрова інфраструктура	1. Складність масштабування в деяких банках

Продовження таблиці 1.3

Переваги	Недоліки
2. Розвинена цифрова інфраструктура 3. Високий рівень доступності 4. Інтеграція з державними сервісами 5. Активне впровадження інновацій 6. Високий рівень безпеки даних	2. Складність масштабування в деяких банках 3. Високі вимоги до інфраструктури 4. Високі витрати на підтримку й оновлення великого коду в застарілих системах. 5. Складність підтримки та модернізації

1.3 Вимоги до сучасних банківських систем

1.3.1 Функціональні вимоги до банківських систем

Функціональні вимоги до сучасних банківських систем визначаються необхідністю забезпечення широкого спектра фінансових послуг, високого рівня безпеки, можливості інтеграції з іншими сервісами. Відповідність цим вимогам дозволяє банкам ефективно конкурувати на ринку, підвищувати лояльність клієнтів і забезпечувати стабільну роботу навіть за умов зростання навантаження.

1. *Управління клієнтськими профілями*

Система має підтримувати повний цикл роботи з клієнтами. Зокрема, повинна бути реалізована можливість реєстрації нових користувачів, зберігання та оновлення їх персональних даних, а також процес верифікації особи для забезпечення безпеки та відповідності нормативним вимогам. Користувач повинен мати змогу здійснювати вхід до особистого кабінету,

завершувати сеанс роботи, а також, за потреби, ініціювати закриття свого облікового запису в банківській системі.

2. *Управління рахунками та картками*

Банківська система повинна забезпечувати повний цикл управління платіжними картками – як дебетовими, так і кредитними. Зокрема, має бути реалізовано функціонал для відкриття, обслуговування, блокування та закриття карткових рахунків. Користувачі повинні мати змогу переглядати поточний баланс, а також доступ до історії.

3. *Проведення фінансових операцій*

Система повинна підтримувати здійснення грошових переказів між картками як всередині банку, так і на картки інших банків. Повинна бути можливість коректно обробляти більше одного платежу одночасно.

4. *Безпека та автентифікація*

Банківська система повинна впроваджувати сучасні механізми безпеки, зокрема багаторівневу автентифікацію користувачів, шифрування чутливих даних як у стані спокою, так і при передачі, безпечне збереження сесій із захистом від атак типу CSRF та XSS.

5. *Безпека та автентифікація*

Система повинна забезпечувати стабільну роботу при зростанні кількості користувачів, підтримувати резервне копіювання, відновлення після збоїв та автоматичного реагування на інциденти.

1.3.2 Вимоги до безпеки та захисту даних

Безпека та захист даних є одними з ключових вимог до сучасних банківських систем, оскільки ці системи оперують великою кількістю конфіденційної інформації та фінансових активів клієнтів. Будь-яке порушення безпеки може призвести до значних фінансових збитків, втрати довіри з боку клієнтів і юридичних наслідків для банку. Саме тому питання

інформаційної безпеки та захисту даних мають бути пріоритетними на всіх етапах – від розробки і тестування до впровадження та експлуатації системи.

1. *Шифрування даних*

Всі конфіденційні дані, що зберігаються або передаються, повинні бути захищені сучасними алгоритмами шифрування. Це стосується як даних у базах, так і інформації, що передається між клієнтом і сервером.

2. *Захист від кібератак*

Банківська система повинна бути захищена від основних типів кібератак: фішинг, DDoS, SQL-ін'єкції, XSS, CSRF тощо. Для цього впроваджуються міжмережеві екрани, регулярне оновлення програмного забезпечення та проведення аудиту безпеки.

3. *Відповідність стандартам і регуляторним вимогам*

Банківські системи повинні відповідати міжнародним стандартам безпеки (наприклад, PCI DSS для платіжних карток, ISO/IEC 27001) та вимогам законодавства щодо захисту персональних даних (GDPR, Закон України «Про захист персональних даних» [8]).

1.3.3 Вимоги до продуктивності та масштабованості

Продуктивність і масштабованість є критично важливими характеристиками сучасних банківських систем, оскільки саме вони визначають здатність платформи ефективно обробляти великі обсяги транзакцій, забезпечувати стабільну роботу під час пікових навантажень і підтримувати зростання кількості користувачів та сервісів. Високі показники продуктивності та масштабованості дозволяють банкам гарантувати безперервність обслуговування, мінімізувати ризик простоїв і втрати даних, а також швидко адаптуватися до змін ринку та зростаючих вимог бізнесу.

1. *Висока пропускна здатність*

Банківська система повинна забезпечувати обробку великої кількості транзакцій у реальному часі, включаючи платежі, перекази, запити до бази даних, оновлення балансу тощо. Продуктивність системи вимірюється кількістю транзакцій, які можуть бути виконані за одиницю часу (TPS).

2. *Мінімальний час відгуку*

Система повинна забезпечувати швидкий час відгуку на запити користувачів, незалежно від навантаження. Зазвичай прийнятним вважається час відгуку не більше 1-2 секунд для основних операцій, що підвищує задоволеність клієнтів і знижує ризик відмови від сервісу.

3. *Горизонтальна та вертикальна масштабованість*

Сучасна банківська система має підтримувати як вертикальне, так і горизонтальне масштабування. Це дозволяє ефективно розподіляти навантаження, уникати «вузьких місць» і забезпечувати безперервну роботу навіть при різкому зростанні кількості користувачів.

4. *Балансування навантаження*

Важливою вимогою є впровадження механізмів балансування навантаження між різними компонентами системи, серверами та базами даних. Це дозволяє уникати перевантаження окремих вузлів і забезпечує рівномірний розподіл ресурсів.

ВИСНОВОК ДО РОЗДІЛУ 1

У першому розділі було здійснено всебічний аналіз сучасного стану банківського сектору, основних напрямів його цифрової трансформації, актуальних тенденцій у сфері фінансових послуг, а також архітектурних підходів до побудови банківських систем. Розгляд і порівняння існуючих платформ в Україні таких як Приват24, monobank, Sense Bank та Ощад 24/7 – дозволили зробити висновки про високий рівень діджиталізації фінансової галузі.

Здійснений аналіз підтверджує, що цифрова трансформація є не просто трендом, а критично важливою умовою конкурентоспроможності банків у сучасному середовищі. Активне впровадження мобільних додатків, онлайн-сервісів, систем автоматизованої верифікації, інтеграція з державними API – усе це свідчить про швидкий перехід банків до гнучких і масштабованих технологічних рішень. Успішність цього переходу залежить не лише від технічної реалізації, а й від здатності банків відповідати на зміну поведінки клієнтів, забезпечуючи швидкий, зручний і безпечний доступ до фінансових послуг.

Визначено два головні архітектурні підходи, що використовуються при побудові банківських платформ: монолітна та мікросервісна архітектура. Монолітна структура, хоч і залишається поширеною, втрачає актуальність через обмеження масштабованості та складність оновлення. Натомість мікросервісна архітектура забезпечує гнучкість, незалежність модулів, зниження ризиків при оновленнях та полегшення масштабування системи, що є критично важливим для фінансових установ з великою кількістю користувачів і високими навантаженнями. Саме тому провідні банки України дедалі частіше переходять на сервісно-орієнтовану архітектуру.

Також було систематизовано основні переваги та недоліки існуючих банківських платформ в Україні. Незважаючи на високий рівень цифровізації,

деякі системи мають труднощі з масштабуванням, страждають від складної підтримки застарілих компонентів і несуть великі витрати на оновлення інфраструктури. Водночас, загальна тенденція вказує на активне впровадження сучасних технологій.

У підрозділі 1.3 окреслено функціональні, безпекові та нефункціональні вимоги до сучасної банківської системи. Ці вимоги лягли в основу подальшого проєктування та реалізації розроблюваної платформи, яка повинна відповідати всім критичним стандартам надійності, високій пропускній здатності та захисту персональних даних.

РОЗДІЛ 2
ЕТАП ПРОЄКТУВАННЯ СИСТЕМИ

2.1 Вибір технологій

У сучасних умовах стрімкого розвитку цифрових технологій, банківські системи повинні забезпечувати не тільки надійну обробку транзакцій, а й гарантувати безпеку, масштабованість, гнучкість та високу доступність сервісів. Зважаючи на суворі вимоги до фінансових застосунків, зокрема щодо захисту персональних та платіжних даних, важливо ретельно обирати технології та архітектурні підходи, що лежать в основі розробки таких систем.

Однією з найефективніших сучасних архітектурних моделей, що широко застосовується у фінансовій сфері, є мікросервісна архітектура. Основна її перевага полягає у поділі системи на певну кількість незалежних сервісів, кожен з яких відповідає за окрему частину функціональності: автентифікація, управління користувачами, верифікація, обробка карткових операцій тощо. Кожен мікросервіс є ізольованим процесом, який може бути розроблений, протестований, масштабований та оновлений незалежно від інших. У контексті банківських систем такий підхід не лише спрощує їхнє обслуговування, а й дозволяє уникнути простоїв при внесенні змін, що критично для систем, які повинні працювати без збоїв.

В основу розробки буде покладено NestJS – сучасний фреймворк на базі Node.js з підтримкою TypeScript, що поєднує у собі переваги об'єктно-орієнтованого, функціонального та реактивного програмування [9]. NestJS дозволяє створювати структуровані додатки, орієнтовані на модульність та незалежність один від одного. Його інструментарій сприяє швидкій розробці, а використання TypeScript забезпечує статичну типізацію, що знижує ймовірність помилок на етапі розробки, підвищуючи надійність у роботі з фінансовими даними.

У контексті мікросервісної взаємодії, NestJS підтримує реалізацію сервісів через брокери повідомлень, наприклад таких як RabbitMQ [10]. Це дозволяє створити асинхронну архітектуру з реакцією на події, де сервіси обмінюються повідомленнями, не блокуючи один одного. Наприклад, після реєстрації нового користувача сервіс автентифікації може надіслати повідомлення про створення облікового запису до інших сервісів, таких як сервіс верифікації або сервіс обліку карток, які самостійно виконають необхідні дії.

Для взаємодії з базами даних у проєкті буде використано PostgreSQL, яка є перевіреним рішенням для фінансових систем [11]. Її відповідність принципам ACID, підтримка складних транзакцій та вбудовані механізми безпеки дозволяють надійно зберігати чутливу інформацію. У поєднанні з ORM-інструментом Prisma, який автоматично генерує типи моделей, спрощує написання SQL-запитів та забезпечує міграції бази даних, така технічна комбінація забезпечуватиме високу продуктивність та безпеку.

Для підвищення ефективності обробки запитів та зниження навантаження на основну базу даних буде застосовуватися Redis – високошвидкісний інструмент кешування [12]. Він дозволяє зберігати тимчасові дані, такі як токени доступу, коди підтвердження або ж сесії користувачів. Redis ефективний у задачах, де необхідно забезпечити миттєвий доступ до невеликого обсягу даних із регулярними зверненнями.

Ще однією перевагою мікросервісної архітектури є можливість гнучкої контейнеризації сервісів. Для цього в проєкті використовуватиметься Docker, який дозволяє створити ізольовані середовища для кожного компонента [13]. Це спрощує розгортання, пришвидшує CI/CD-процеси та унеможливлює конфлікти між залежностями. У майбутньому, для production-середовища може бути впроваджено Kubernetes для автоматизованого масштабування, балансування навантаження та моніторингу стану мікросервісів.

Для маршрутизації запитів, балансування навантаження та обробки статичних ресурсів використовуватиметься Nginx як HTTP-сервер та зворотний проксі [14]. У рамках мікросервісної архітектури Nginx виконуватиме роль API Gateway, перенаправляючи запити до відповідних сервісів, застосовуючи правила доступу, а також забезпечуючи термінацію SSL-з'єднань. Його використання дозволяє підвищити безпеку, продуктивність та масштабованість системи. Крім того, Nginx ефективно кешує відповіді від мікросервісів, знижуючи навантаження на них при повторних зверненнях.

Отже, загальний список технологій, що було обрано, описано в табл. 2.1.

Таблиця 2.1 – Загальна таблиця обраних основних технологій

Технологія	Призначення	Причина вибору
NestJS	Серверна платформа для побудови модульної архітектури та REST API	Забезпечує масштабованість, модульність, підтримку TypeScript і зручну інтеграцію з RabbitMQ та іншими сервісами
TypeScript	Мова програмування, що компілюється у JavaScript	Підвищує надійність коду за рахунок типізації, зменшує кількість помилок під час розробки
PostgreSQL	Реляційна система керування базами даних	Потужна, стабільна, з розширеною підтримкою транзакцій і зв'язків між таблицями

Продовження таблиці 2.1

Технологія	Призначення	Причина вибору
Prisma ORM	Інструмент для взаємодії з базою даних	Спрощує роботу з БД, забезпечує типобезпечність і автоматичне генерування типів моделей
RabbitMQ	Система обміну повідомленнями між сервісами (message broker)	Підходить для мікросервісної архітектури, забезпечує асинхронний обмін повідомленнями
Docker	Контейнеризація додатків	Забезпечує ізоляцію сервісів, спрощує розгортання і тестування
Redis	Кешування, зберігання сесій та тимчасових даних	Висока швидкодія, підходить для зберігання тимчасових даних або керування чергами
Nginx	Зворотний проксі, маршрутизація, балансування, SSL, кешування	Висока продуктивність, стабільність, підтримка зворотного проксі та безпеки
S3 (Minio)	Зберігання файлів (документів)	Горизонтальне масштабування, висока пропускна здатність, розподілене зберігання

2.2 Архітектура системи

Для реалізації даного проєкту було розроблено архітектурне рішення, що базується на мікросервісній парадигмі проєктування програмного забезпечення. Обрана архітектурна модель передбачає поділ єдиного монолітного застосунку на функціонально відокремлені компоненти.

Кожен мікросервіс у спроєктованій системі відповідатиме за окремий домен бізнес-логіки, матиме власну модель даних, а також чітко визначені інтерфейси для взаємодії з іншими сервісами. Такий підхід забезпечуватиме високу гнучкість, масштабованість і незалежність у розробці, розгортанні та супроводі кожного компонента системи.

Загалом, можна виділити наступні структурні компоненти архітектури:

1. *Сервіс автентифікації (Auth Service)*

Сервіс автентифікації відповідатиме за реєстрацію користувачів та управління сесіями. Також у даному сервісі буде реалізовано підтримку двофакторної автентифікації (2FA), що підвищуватиме рівень безпеки доступу до системи, а також механізми відновлення доступу у разі втрати облікових даних.

2. *Сервіс користувачів (User Service)*

Сервіс користувачів забезпечуватиме управління обліковими записами, включаючи зберігання та оновлення персональних даних. Завдяки цьому сервісу інші компоненти зможуть отримувати необхідну інформацію про користувача.

3. *Сервіс верифікації (Verification Service)*

Сервіс верифікації буде реалізовувати процеси ідентифікації та верифікації особи клієнта відповідно до вимог KYC, які регламентуються чинним законодавством у сфері фінансових послуг. Він забезпечуватиме перевірку достовірності особистих даних та документів.

4. *Сервіс карток (Card Service)*

Сервіс керування платіжними картками здійснюватиме управління життєвим циклом банківських карток, включаючи відкриття, блокування, розблокування та видалення. Сервіс взаємодіятиме з верифікаційним та користувацьким сервісами для перевірки статусу клієнта та його прав на отримання картки.

5. *Сервіс платіжок (Payment Service)*

Даний сервіс функціонуватиме як спеціалізований компонент, відповідальний за обробку фінансових транзакцій та управління платіжними операціями.

6. *Сервіс історії операцій (History Service)*

Сервіс історії операцій є компонентом, що відповідатиме за зберігання, обробку та надання доступу до хронологічних даних усіх операцій у системи. Це включатиме: вхід до системи або вихід з системи, зміна персональних даних, будь-які операції із картками, переміщення коштів у системі тощо. Це дозволить вирішувати певні технічні проблеми або проблеми, що пов’язані із фінансовою складовою.

Загальна схема архітектури зображена на креслені у додатку Д1.

Розглянемо її більш детально.

Так як користувач не має знати дані прямо до мікросервісів, куди входить порт тощо, то використовуватиметься API Gateway. API Gateway виконуватиме роль єдиної точки входу для клієнтських запитів і маршрутизуватиме їх до мікросервісів на основі їх URL-шляхів:

- /auth/* -> Auth Service (3000 PORT)
- /users/* -> User Service (3001 PORT)
- /verification/* -> Verification Service (3002 PORT)
- /cards/* -> Card Service (3003 PORT)
- /payment/* -> Payment Service (3004 PORT)

- /history/* -> History Service (3005 PORT)

Сам Nginx забезпечуватиме додавання заголовків до запитів, передачу оригінальних IP-адрес клієнтів, передачу контексту HTTP-запитів, перенаправлення OPTIONS запитів для підтримки CORS preflight. Також, конфігурація Nginx має Load Balancer, який дозволяє масштабувати мікросервіси горизонтально. Тобто, він допомагає, коли треба підняти більше 1 інстанса мікросервіса, наприклад при великій навантаженості.

Мікросервіси спілкуватимуться за допомогою RabbitMQ та протоколу AMQP. З прикладного рівня можна виділити Redis Protocol для взаємодії із Redis DB, S3 API для взаємодії із об'єктним сховищем.

Кожен сервіс матиме власну чергу для отримання повідомлень:

- auth_queue – для Auth Service
- user_queue – для User Service
- verification_queue – для Verification Service
- card_queue – для Card Service
- payment_queue – для Payment Service
- history_queue – для History Service

Патерни обміну повідомленнями [15].

1. Request-Response (запит-відповідь)

Цей патерн використовується для синхронної взаємодії між сервісами, коли один сервіс надсилає запит, а інший повертає відповідь. Такий підхід дозволяє отримати результат у реальному часі та використовується для критичних операцій, де важлива негайна відповідь.

2. Publish/Subscribe (публікація/підписка)

Цей патерн застосовується для асинхронної взаємодії, коли один сервіс публікує подію, а інші сервіси, які підписані на цю подію, отримують повідомлення та виконують відповідні дії. Це дозволяє розділити відповідальність між сервісами та забезпечити масштабованість системи

Також кожен мікросервіс матиме свою власну базу даних у PostgreSQL. Сам же Redis буде використаний для зберігання активних сесій користувачів. Файли документів зберігатимуться у S3 сховищі (MinIO).

2.3 Проєктування баз даних

2.3.1 Загальні принципи проєктування

Проєктування баз даних є критично важливим етапом у розробці системи, оскільки саме воно визначає фундамент для надійного зберігання, обробки та захисту фінансової інформації. У системах, що працюють із чутливими персональними та транзакційними даними, помилки на цьому етапі можуть призвести до серйозних ризиків – як технічних, так і правових. З одного боку, кожен мікросервіс повинен мати власну, ізольовану модель даних, що забезпечує автономність і незалежність у розробці та масштабуванні. З іншої сторони, система в цілому має зберігати логічну цілісність і узгодженість інформації, що вимагає продуманої стратегії взаємодії між сервісами, у тому числі через події або черги повідомлень.

У розробленій системі буде застосовано шаблон «Database per Service», де кожен мікросервіс має власну базу даних, що забезпечує:

- Логічну ізоляцію даних, де кожен сервіс відповідатиме за власні дані
- Незалежне масштабування, де кожна база може масштабуватися окремо
- Підвищену відмовостійкість, коли збій в одній з баз даних не впливатиме на інші.

2.3.2 Структура баз даних PostgreSQL

1. *База даних сервісу автентифікації (Auth Service)*

База даних сервісу автентифікації спроєктована для забезпечення безпечного процесу входу, двофакторної автентифікації та процедур відновлення доступу. Особливість цієї бази даних – фізичне відокремлення чутливих даних від основної інформації користувача, що відповідає принципу розділення відповідальності.

Сутності PasswordResetToken та TwoFactorSecret реалізують основні функції безпеки. Токени для скидання паролю генеруються з обмеженим часом життя, обов'язковою перевіркою унікальності та можливістю лише одноразового використання. Це захищає від атак повторного використання або підбору токенів. Секретні коди двофакторної автентифікації зберігаються з можливістю їх тимчасової деактивації без повного видалення.

Важливим аспектом є організація процесу очищення даних: застарілі токени автоматично видаляються через регулярні задачі обслуговування, що запобігає перевантаженню бази даних зайвими даними. Моделі PasswordResetToken та TwoFactorSecret показані на рисунку 2.1.

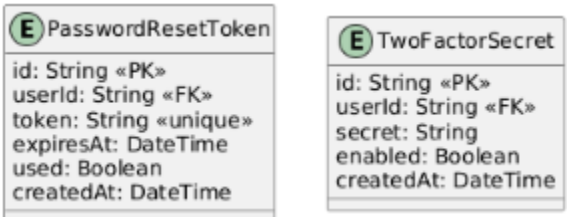


Рисунок 2.1 – Моделі PasswordResetToken та TwoFactorSecret

2. *База даних сервісу користувачів (User Service)*

База даних у сервісі користувачів є центральною у всій архітектурі. Клієнт є ядром, навколо нього буде будуватися уся бізнес логіка проєкту. Ця база зберігає усю важливу інформацію про користувача системи: персональні дані, контактну інформацію, роль користувача та поточний статус. При проєктуванні бази даних користувачів було приділено особливу увагу безпеці та відповідності вимогам GDPR.

Модель даних User містить основні поля для ідентифікації користувача та забезпечення його взаємодії з системою. Паролі зберігаються лише у вигляді хешів, що підвищує безпеку системи. Унікальні індекси на полях email та phone забезпечують швидкий пошук та гарантують, що в системі не буде дублікатів користувачів. Для підтримки бізнес-логіки введено поле status з переліком можливих станів у вигляді Enum: активний, заблокований або неверифікований, а також значення ролі користувача, що може бути або звичайним клієнтом, або адміністратором. Модель User та Enum із статусами користувача зображені на рисунку 2.2.

Для оптимізації продуктивності додано індексування полів, які часто використовуються для пошуку та фільтрації.

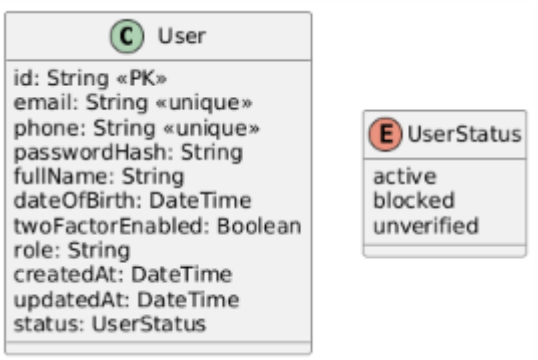


Рисунок 2.2 – Модель User та Enum UserStatus

3. *База даних сервісу верифікації (Verification Service)*

База даних сервісу верифікації створена для підтримки процесів KYC і забезпечення відповідності регуляторним вимогам. У неї зберігається інформація про завантажені користувачами документи.

Така структура дозволяє зберігати метадані ID-карток або ж водійських посвідчень. При цьому самі документи фізично зберігаються в S3-сумісному сховищі, а сама база даних містить лише посилання на них, що оптимізує використання ресурсів і відповідає принципам проєктування для хмарних середовищ. Модель для сервісу верифікації та Enum’и зображені на рисунку 2.3.

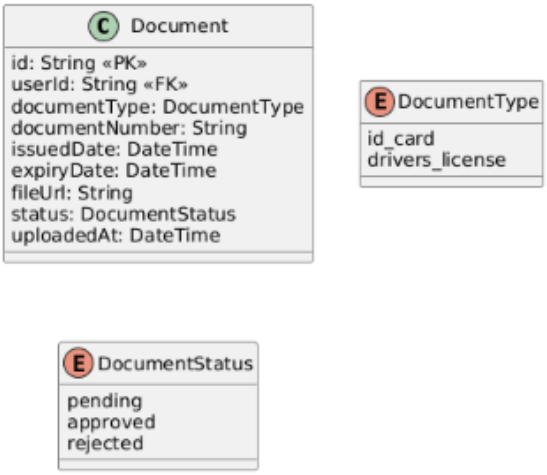


Рисунок 2.3 – Модель Document та Enums для сервісу верифікації

4. *База даних сервісу карток (Card Service)*

База даних для сервісу карток розроблена з урахуванням вимог PCI DSS для безпечного зберігання та обробки даних платіжних карток. Структура бази даних сервісу карток включає дві основні сутності: картки та заявки на випуск карток.

Важливо реалізувати, щоб CVV-код зберігався виключно в хешованому форматі й ніколи не міг бути відновлений у відкритому вигляді. Кожна картка має визначений життєвий цикл статусів – від активної до заблокованої чи закритої, що дозволяє коректно керувати їх використанням. Моделі для сервісу карток та Enum’и зображені на рисунку 2.4.

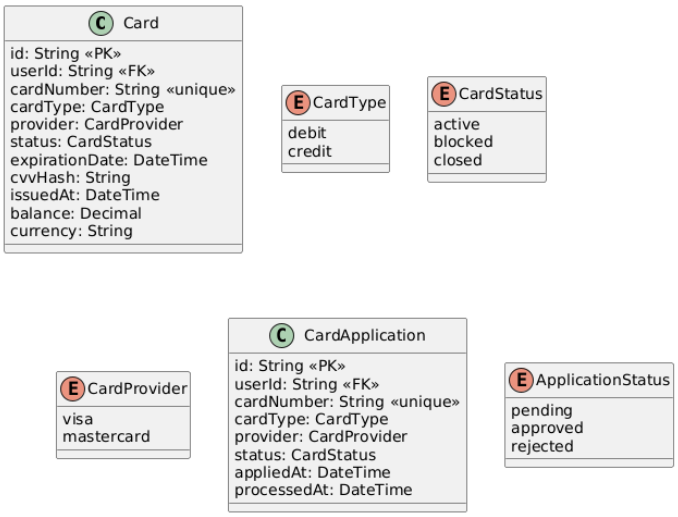


Рисунок 2.4 – Моделі та Enums для сервісу карток

5. *База даних сервісу транзакцій (Payment Service)*

База даних сервісу платежів є дуже важливою у банківській системі, оскільки забезпечує зберігання історії фінансових транзакцій. Вона спроєктована з урахуванням необхідності підтримки високої доступності.

Основною сутністю є модель Transfer, а також Enum – PaymentStatus. При проєктуванні особлива увага приділена точності фінансових розрахунків – для зберігання грошових сум використовується тип Decimal, що забезпечує точність без проблем із плаваючою комою.

Виникає логічне питання: навіщо зберігати щось у базі даних для переказів, якщо ми матимемо базу даних із історією всіх транзакцій, дій користувача. Відповідь на це питання є цілком логічною. Тому що ця база використовуватиметься для відстеження статусу активних платежів, а також для забезпечення атомарності операції. Тобто фактично у цій базі даних будуть зберігатися операційні дані, які будуть видалені протягом певного часового проміжку, а у History Database – історичні дані.

Моделі для сервісу карток та Enum TransferStatus зображені на рисунку 2.5.

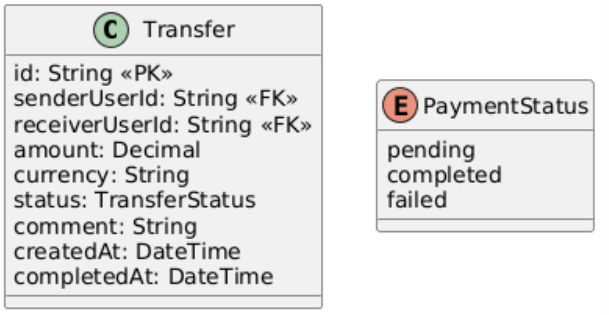


Рисунок 2.5 – Модель Transfer та Enum PaymentStatus

6. *База даних сервісу історії операцій (History Service)*

База даних сервісу історії операцій зберігає інформацію про будь-які дії користувача у системі. Це може бути як вхід в систему або вихід з неї, відправлення або отримання коштів тощо. Завдяки зберіганню такої

інформації можна виявляти факти шахрайства, несанкціонованого входу до системи тощо.

Модель Event містить ключову інформацію: типу івенту та метадані що передаються до зберігання. Enum EventType зберігає основні типи івентів які потрібно логувати до бази. Модель та Enum зображені на рисунку 2.6.

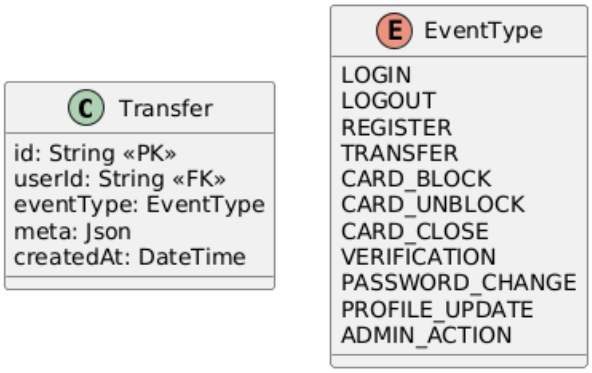


Рисунок 2.6 – Модель Event та Enum EventType

Загальна діаграма спроєктованих баз даних зображена у додатку Д2.

2.4 Безпека

2.4.1 Архітектурні принципи безпеки

Проєкт буде побудовано за принципами сервісно-орієнтованої архітектури, що дозволить ізолювати логіку окремих підсистем і зменшити ризики потенційних атак. Кожен мікросервіс розгортатиметься у власному контейнері та матиме доступ лише до тих ресурсів, які безпосередньо необхідні для його функціонування. Таким чином, у разі успішної атаки на один із сервісів, шкода не поширюватиметься на всю систему.

Особливу увагу буде приділено тому, щоб сервіси спілкувалися виключно через захищені канали – наприклад, RabbitMQ із TLS. Усі критичні секретні дані зберігатимуться у змінних середовища й ніколи не потраплятимуть до вільного доступу.

2.4.2 Аутентифікація та авторизація

Для безпечного користування системою буде реалізовано механізм аутентифікації на основі сесій із використанням Redis як сховища сесійних даних. Такий підхід забезпечуватиме високу безпеку, масштабованість і стабільність у комплексі. Після успішної автентифікації користувача сервер створюватиме унікальний ідентифікатор сесії, який зберігатиметься у Redis разом із відповідними даними користувача. Клієнт отримуватиме HTTP cookie з цим ідентифікатором, який автоматично передаватиметься з кожним наступним запитом.

Зберігання сесій у Redis дозволить досягти централізованого керування сесіями, що буде критично важливо при горизонтальному масштабуванні системи. Це забезпечуватиме узгодженість стану користувача незалежно від того, до якого саме екземпляра сервісу надійде запит. Крім того, Redis забезпечуватиме високу швидкодію й підтримуватиме механізми автоматичного очищення застарілих сесій завдяки TTL механізму.

Такий підхід дозволятиме ефективно захищати сеанси користувачів від атак типу session fixation, session hijacking та CSRF. Для додаткового захисту cookie матимуть прапорці HttpOnly та Secure.

Також буде реалізовано функціонал двофакторної аутентифікації. Це означатиме, що якщо користувач налаштує Google Auth, то для доступу до особистого кабінету, захищених ендпоінтів тощо, йому потрібно буде ввести код з додатку, який система валідуватиме.

2.4.3 Захист паролів

Паролі користувачів зберігатимуться у базі лише у вигляді хешів, сформованих за допомогою алгоритму bcrypt. Цей алгоритм буде криптографічно стійким і автоматично додаватиме «сіль», що унеможливить атаки типу rainbow table. Вибір bcrypt також зумовлюватиметься його

здатністю додавати затримку при хешуванні, що ускладнить brute-force атаки. Збереження паролів у такому вигляді дозволить бути впевненим, що навіть у разі витоку даних з бази їх буде важко відновити.

2.4.4 Безпека API Gateway

Безпека API Gateway буде реалізована на основі Nginx, який виконуватиме функції зворотного проксі-сервера. Усі запити клієнтів спочатку надходитимуть до шлюзу, де перевірятимуться, фільтруватимуться, доповнюватимуться заголовками безпеки й лише після цього перенаправлятимуться до внутрішніх мікросервісів. Це дозволятиме централізовано керувати політиками доступу й зменшуватиме ризики атак.

Буде реалізовано встановлення безпечних HTTP-заголовків (X-Forwarded-For, X-Real-IP, X-Forwarded-Proto, Host, Cookie) для збереження достовірної інформації про клієнта. Налаштовуватимуться таймаути та буде вимкнене кешування проксі, щоб запобігти атакам типу Slowloris або маніпуляціям із буфером. Також буде впроваджено CORS-політику з підтримкою preflight-запитів для обмеження доступу до API з небажаних джерел. Кожен маршрут проксуватиметься лише до відповідного мікросервісу, ізольованого в межах внутрішньої Docker-мережі, що унеможливить прямий зовнішній доступ.

2.4.5 Безпека Docker-контейнерів

Кожен мікросервіс працюватиме в ізольованому Docker-контейнері. Усі образи базуватимуться на перевірених офіційних базових образах із регулярними оновленнями безпеки. Контейнери буде об’єднано в приватну мережу, недоступну ззовні, а доступ до портів дозволятиметься лише іншим контейнерам, яким необхідна взаємодія. Це обмежить доступ до внутрішніх сервісів і унеможливить їхнє сканування ззовні.

2.4.6 Безпека Redis

Redis у проєкті буде використовуватися для кешування сесій та зберігання короткострокових даних. Його буде розгорнуто так, щоб він був доступний лише з внутрішньої мережі, з автентифікацією по паролю та обмеженим набором дозволених команд. Оскільки Redis за замовчуванням не має вбудованої автентифікації, налаштування описані вище суттєво зменшать ризики зовнішнього впливу на сховище.

2.4.7 Безпека RabbitMQ

RabbitMQ буде використовуватися як транспортний шар між сервісами. Комунікація між мікросервісами відбуватиметься через обмін повідомленнями, які можуть містити критичні або конфіденційні дані. Щоб запобігти перехопленню, RabbitMQ буде налаштовано на приймання з'єднань лише з внутрішньої мережі, із підтримкою TLS-шифрування каналів.

2.5 Масштабованість

Масштабованість є однією з ключових вимог до майбутньої банківської системи, оскільки вона має забезпечити здатність ефективно обслуговувати зростаючу кількість користувачів, обробляти великі обсяги транзакцій та оперативно впроваджувати нові сервіси. Під час розробки мікросервісної банківської платформи буде враховано низку викликів, які можуть виникнути у процесі масштабування, а також передбачено відповідні рішення для забезпечення стабільної та ефективної роботи системи.

Однією з очікуваних проблем стане обмеженість монолітної архітектури, яка унеможливлює масштабування. У таких системах зростання навантаження на один модуль змусить масштабувати всю систему повністю, що є неефективним і фінансово затратним. Вирішенням цього питання стане

реалізація мікросервісної архітектури, в рамках якої кожен сервіс буде окремим незалежним додатком. Це дозволить масштабувати лише ті компоненти, які зазнають підвищеного навантаження.

Зі зростанням кількості користувачів і транзакцій виникатиме потреба в ефективному розподілі навантаження між сервісами. У проєктному рішенні для балансування трафіку планується використання зворотного проксі-сервера, який перенаправлятиме запити до відповідних інстансів сервісів. Завдяки можливості запуску кількох копій одного сервісу та ручному масштабуванню вдасться досягти базового рівня розподілення навантаження.

Наступним важливим аспектом стане масштабування бази даних. В умовах постійного зростання транзакцій існуватиме ризик перевантаження сховища даних, що може негативно вплинути на продуктивність. Основною базою даних планується використання PostgreSQL з налаштуванням реплікації за схемою master-slave. Це дозволить розподіляти навантаження між різними вузлами – операції читання передаватимуться на репліки, а запис – здійснюватиметься на головному сервері. Для пришвидшення доступу до часто використовуваних даних буде застосовано Redis, що суттєво зменшить навантаження на основну базу. Окрім того, використання ORM забезпечить оптимізацію запитів та ефективну роботу з великими обсягами інформації.

Організація взаємодії між сервісами в умовах масштабування стане ще одним важливим викликом. Із зростанням кількості інстансів комунікація між сервісами ускладнюватиметься, особливо в разі синхронної взаємодії. У архітектурі проєкту буде впроваджено асинхронну модель з використанням RabbitMQ. Цей брокер повідомлень дозволить організувати надійні черги, забезпечуючи гарантовану доставку подій навіть у випадку пікового навантаження. RabbitMQ також підтримуватиме кластеризацію, що дасть змогу масштабувати систему обміну повідомленнями та підвищити її стійкість до збоїв.

Особливу увагу буде приділено роботі системи в умовах пікового навантаження, яке є типовим для фінансових платформ – наприклад, під час масових виплат чи одночасного виконання великої кількості транзакцій. У таких випадках планується заздалегідь запускати додаткові екземпляри критичних сервісів і збільшувати ресурси Redis та RabbitMQ для обробки зростаючої кількості запитів. Завдяки гнучкій структурі мікросервісів і можливості незалежного масштабування компонентів система зможе оперативно адаптуватися до зростаючого навантаження.

ВИСНОВОК ДО РОЗДІЛУ 2

У другому розділі здійснено обґрунтування ключових архітектурних та технологічних рішень, що формують основу системи. Попередньо до цього, було прийнято рішення про використання мікросервісної архітектури. Такий підхід дозволяє кожному сервісу функціонувати автономно, розгортатися незалежно та виконувати конкретну бізнес-функцію.

Було детально оглянуто вибір стеку технологій. Було обґрунтовано вибір кожного з інструментів і технологій. Кожен компонент розглядався не лише через призму зручності чи популярності, а з точки зору здатності ефективно функціонувати в умовах навантаження, відповідати вимогам безпеки, і водночас залишатися гнучким до змін. Особливо важливо, що система передбачає впровадження асинхронного обміну даними, ізоляції середовищ, централізованого управління сесіями та механізмів, спрямованих на захист персональної інформації.

Розроблено архітектуру системи, яка охоплює окремі мікросервіси для автентифікації, управління користувачами, верифікації тощо. Усі сервіси взаємодіють між собою через централізований API Gateway. Це рішення дозволяє централізовано контролювати маршрутизацію запитів, обробку навантаження.

Виконано проєктування баз даних кожного сервісу відповідно до підходу «Database per Service». Такий підхід дозволяє зберігати логічну ізоляцію, зменшити зв’язаність компонентів і полегшити масштабування. Було описано призначення кожної з баз даних.

Розглянуто елементи безпеки кожного елементу системи. Прикладами такого є: реалізація механізмів аутентифікації через сесії, двофакторна автентифікація, шифрування з'єднань, ізоляція контейнерів, обмеження доступу до даних.

Були описані виклики та спроєктовані відповідні рішення у плані масштабованості, що дозволять системі адаптуватися наприклад, до зростання навантаження, збільшення кількості користувачів тощо.

РОЗДІЛ 3
ЕТАП РОЗРОБКИ СИСТЕМИ

3.1 Організація процесу розробки

3.1.1 Ініціалізація та структурування проєкту

На початковому етапі розробки системи важливо правильно спланувати структуру проєкту. Це потрібно для того, щоб у подальшому було зручно працювати з проєктом. Для цього було створено окремий репозиторій у системі контролю версій Git. Це дає змогу зберігати всі зміни, які відбувалися з кодом, і при потребі повертатися до попередніх версій. Такий підхід також допомагає уникати втрати важливих частин коду та краще організовувати роботу під час розробки.

Структура проєкту була організована за принципом «monorepo», де кожен мікросервіс розміщено у власній директорії (auth-service, user-service, card-service, payment-service, history-service, verification-service). Monorepo було обрано тому, що він полегшує синхронізацію змін між модулями, спрощує розробку, а також дозволяє ефективніше контролювати версії та залежності в одному спільному середовищі [16].

У кожному сервісі виділено окремі папки для основних компонентів:

- /src – директорія із вихідним кодом сервісу
- /prisma – директорія із схемами баз даних та міграції
- /common – директорія із спільними файлами, що перевикористовуються (наприклад фільтри, гарди тощо)

Також кожен сервіс має конфігураційні файли:

- tsconfig.ts – файл з налаштуваннями параметрів компіляції TypeScript
- package.json – файл, що зберігає інформацію про проєкт, його залежності, скрипти та конфігурації

- package-lock.json – файл, що фіксує точні версії встановлених залежностей
- jest.config.js – файл із конфігурацією, що використовується для налаштування поведінки фреймворку для тестування Jest
- Dockerfile – файл, що містить інструкції для створення Docker-образу, для розгортання застосунку в ізольованому середовищі
- docker-compose.yaml – файл, що описує і координує запуск кількох контейнерів Docker як єдиного сервісу з їхніми налаштуваннями та мережевими зв’язками
- .prettierrc – файл, що містить налаштування для автоматичного форматування коду за допомогою інструменту Prettier
- .dockerignore – файл, що описує файли та директорії, які будуть проігноровані при створені Docker образу
- .env.example – файл із прикладами секретних даних що мають зберігатися у .env файлі

Так як було обрано структуру одного репозиторію, то є спільні файли (файли та директорії, що були описані раніше не будуть продубльовані) такі як:

- .gitignore – файл, який визначає, які файли та папки не слід відстежувати в Git
- nest-cli.json – конфігураційний файл для NestJS CLI
- /config – директорія, що зазвичай містить конфігураційні файли для застосунку (Nginx, RabbitMQ)

На цьому етапі також було визначено основні правила для іменування файлів, структурування модулів, організації DTO, сервісів, контролерів, що забезпечує єдиний стиль коду для всього проєкту.

Щодо Git Workflow, то його можна наступним чином: існує головна гілка проєкту та гілка dev для реалізації певного функціоналу. Впевнившись, що

немає жодних помилок і все працює так як потрібно, можна робити pull request до main гілки та робити merge гілок. Git Workflow зображено на рисунку 3.1.

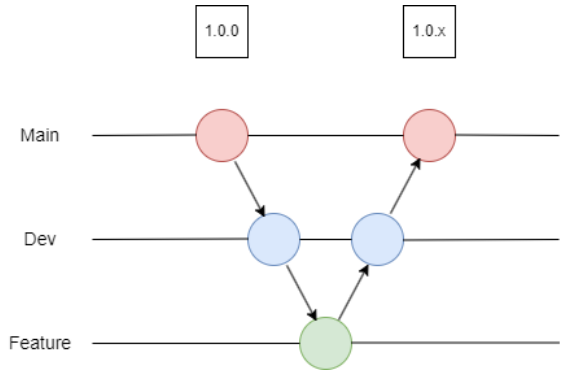


Рисунок 3.1 – Git Workflow Structure

3.1.2 Контейнеризація, налаштування Docker, Nginx та RabbitMQ

Для створення ізольованого, стабільного та зручного середовища розробки у проєкті використовується Docker. Для кожного мікросервісу був створений власний Dockerfile, де описано процес збирання контейнера: встановлення залежностей, копіювання коду, компіляцію TypeScript та запуск відповідного сервісу. Це дозволяє запускати кожен сервіс незалежно від операційної системи чи конфігурації машини, на якій працює проєкт. Dockerfile зображено на рисунку 3.2.

```
1   FROM node:20-alpine
2
3   WORKDIR /app
4
5   COPY package*.json ./
6   RUN npm install
7   RUN npm install -g @nestjs/cli
8
9   COPY . .
10
11  RUN npm run build
12  RUN npx prisma generate
13
14  EXPOSE 3001
15
16  CMD ["npm", "run", "start:dev"]
```

Рисунок 3.2 – Dockerfile

Щоб запускати всю систему одразу разом із необхідними сервісами, такими як RabbitMQ, PostgreSQL, Redis, тощо, було налаштовано файл docker-compose.yaml. У ньому вказано, як пов’язані між собою мікросервіси, які змінні середовища потрібні, яка мережа використовується, та як зберігаються дані. Завдяки цьому можна швидко запускати всю інфраструктуру на будь-якому комп’ютері, не налаштовуючи кожен компонент окремо. Docker-compose зображено на рисунку 3.3.

```yaml
version: '3.8'

services:
  auth-service:
    build: .
    container_name: auth-service
    environment:
      DATABASE_URL: ${DATABASE_URL}
      RABBITMQ_URL: ${RABBITMQ_URL}
      REDIS_URL: ${REDIS_URL}
    volumes:
      - ./:/app
    working_dir: /app
    command: sh -c "npm install && npm run start:dev"
    ports:
      - "3000:3000"
    networks:
      - postgres-net
      - rabbitmq-net
      - redis-net
      - nginx-gateway
  auth-worker:
    build:
      context: .
    container_name: auth-worker
    command: npm run start:worker
    environment:
      NODE_ENV: production
      RABBITMQ_URL: ${RABBITMQ_URL}
    depends_on:
      - auth-service
    networks:
      - rabbitmq-net
      - postgres-net
      - redis-net

networks:
  postgres-net:
    external: true
  rabbitmq-net:
    external: true
  redis-net:
    external: true
  nginx-gateway:
    external: true
```

Рисунок 3.3 – docker-compose.yaml

Вхідною точкою для всіх зовнішніх запитів у системі виступає Nginx. Він маршрутизує запити до відповідних мікросервісів згідно з прописаними правилами, приховуючи внутрішню структуру проєкту. Крім того, Nginx дозволяє налаштувати балансування навантаження, кешування, SSL-термінацію та базовий захист від небажаних запитів. У конфігураційному файлі nginx.conf описані маршрути до кожного сервісу, таймаути, обмеження розміру запитів та базові параметри безпеки, наприклад, для захисту від простих DDoS-атак. Уся конфігурація Nginx знаходиться у файлі nginx.conf. Nginx.conf зображено на рисунку 3.4.

```
events {}

http {
  # Global proxy headers
  proxy_set_header Host $host;
  proxy_set_header X-Real-IP $remote_addr;
  proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
  proxy_set_header X-Forwarded-Proto $scheme;
  proxy_set_header Cookie $http_cookie;

  # Timeouts and buffers
  proxy_connect_timeout       10s;
  proxy_send_timeout          30s;
  proxy_read_timeout          30s;
  send_timeout                30s;
  proxy_buffering             off;

  # CORS headers
  map $http_origin $cors_origin {
    default "*";
  }

  upstream auth_service {
    server auth-service:3000;
  }

  server {
    listen 80;

    # Preflight CORS
    location / {
      if ($request_method = 'OPTIONS') {
        add_header Access-Control-Allow-Origin "$cors_origin" always;
        add_header Access-Control-Allow-Methods "GET, POST, OPTIONS, PUT, DELETE" always;
        add_header Access-Control-Allow-Headers "Authorization, Content-Type" always;
        add_header Content-Length 0;
        add_header Content-Type text/plain;
        return 204;
      }
    }

    location /auth/ {
      add_header Access-Control-Allow-Origin "$cors_origin" always;
      add_header Access-Control-Allow-Methods "GET, POST, OPTIONS, PUT, DELETE" always;
      add_header Access-Control-Allow-Headers "Authorization, Content-Type" always;
      proxy_pass http://auth_service/;
    }
```

Рисунок 3.4 – nginx.conf

RabbitMQ запускається як окремий контейнер, а його налаштування описано у файлі rabbitmq.conf. Кожен мікросервіс підключається до RabbitMQ через змінні середовища, що дозволяє гнучко налаштовувати підключення і не зберігати конфіденційні дані у відкритому коді. Обмін повідомленнями побудовано таким чином, що для кожного типу події використовується окрема черга – це дозволяє легко масштабувати окремі сервіси, зменшити кількість зайвих повідомлень і зробити взаємодію між сервісами більш передбачуваною. Приклад rabbitmq.conf зображено на рисунку 3.5.

Рисунок 3.5 – Приклад rabbitmq.conf

3.2 Реалізація сервісів

3.2.1 Створення основних мікросервісів

Процес створення основних мікросервісів базується на принципах модульності, ізоляції та незалежності кожного компонента.

Список основних елементів, що будуть використовуватися у кожному сервісі:

- Контролери (*.controller.ts) – елементи, що відповідають за прийом і валідацію зовнішніх запитів, маршрутизацію та передачу даних у сервіси
- Сервіси (*.service.ts) – елементи, що містять бізнес-логіку
- DTO (*.dto.ts) – класи, що визначають структуру вхідних даних у запиті, валідують поля та їхні типи
- Модулі (*.module.ts) – елементи, що об’єднують контролери, сервіси, провайдери та інші залежності в єдину логічну одиницю
- Контролери івентів (*.events.ts) – елементи, що відповідають за прийом та обробку івентів RabbitMQф

У кожному мікросервісі проєкту є файл main.ts, у якому реалізовано початкову ініціалізацію застосунку. Саме тут підключаються всі необхідні сервіси, налаштовуються глобальні middleware, фільтри обробки помилок, а також конфігурації для роботи з повідомленнями. Завдяки тому, що використовується NestJS, підключення мікросервісного транспорту відбувається досить просто, що дозволяє зробити систему асинхронною та зручною для масштабування.

Кожен мікросервіс має власну базу даних, описану через файл schema.prisma. Такий підхід дозволяє уникнути конфліктів між сервісами, забезпечити повну незалежність міграцій і підвищити загальну безпеку

системи. ORM Prisma використовується для типобезпечної взаємодії з базою – це дозволяє швидко створювати моделі, генерувати та виконувати міграції.

NestJS має механізм Dependency Injection, який активно застосовуються в усіх модулях, завдяки якому можна легко підключати сервіси, репозиторії тощо та й при цьому зберігати модульність і гнучкість архітектури. У кожному модулі використовується лише ті залежності, які реально потрібні, що робить спрощення тестування.

Розробка кожного мікросервісу проводиться поступово – від найпростішого до складнішого. Спочатку створювався базовий каркас із кількома маршрутами та підключенням до бази. Потім поетапно додавались DTO, сервіси, інтеграції з RabbitMQ, взаємодію з іншими мікросервісами та інші потрібні компоненти. Потім додавалися інші необхідні деталі, util-функції, що необхідні для бізнес логіки тощо.

3.2.2 Реалізація бізнес-логіки

У кожному мікросервісі банківської системи було реалізовано бізнес-логіку відповідно до його функціонального призначення. Наприклад, у сервісі користувачів основний фокус – це перевірка унікальності даних під час реєстрації, валідація профілю та підтримка різних статусів. Уся ця логіка є розділеною від інших частин, що дозволяє легко змінювати або розширювати функціонал без впливу на інші елементи.

Під час розробки система слідує підходу коли контролери відповідають лише за прийом запитів і передачу даних, а вся логіка виконується у сервісах. Наприклад, при оновленні профілю контролер просто передає DTO у відповідний метод сервісу, де відбувається перевірка доступу, валідація даних, оновлення запису в базі.

У сервісі керування картками логіка зосереджена на таких речах, як перевірка лімітів користувача, генерація унікального номера картки, зміна її

статусу, а також контроль атомарності операцій під час виконання переказів грошей.

У сервісі для платежів критично важливо забезпечити правильне збереження інформації переводів. Потрібно, щоб під час неуспішного переказу користувач, який приймає, не бачив цю операцію, лише той, хто намагається відправити гроші.

Сервіс історії відповідає за зберігання всієї історії дій, що відбуваються у системі. Сервіс допомагає зберігати дію у базу даних, отримувати усю історію та отримувати певну історію за ID.

Важливо ще те, що налаштовано централізовану обробку помилок у всіх сервісах за допомогою глобальних фільтрів NestJS. Це дає змогу логувати всі винятки та повертати користувачу зрозумілі відповіді.

3.2.3 Забезпечення безпеки та захисту даних

Питання безпеки в цьому проєкті розглядається не просто як набір окремих технічних рішень, а як цілісний підхід до розробки, який охоплює всі рівні системи. Ще на етапі проєктування було поставлено за мету створити таку архітектуру, де захист даних буде вбудований у кожен компонент.

Щодо автентифікації, то замість відкритих токенів реалізовано сесійну модель із використанням Redis, де зберігаються сесії. Двофакторна автентифікація додає додатковий рівень захисту і допомагає зменшити ризики, пов’язані з компрометацією облікових записів. Для зберігання паролів використовується хешування за допомогою bcrypt – це бібліотека, яка унеможливлює розкриття паролів навіть у випадку витоку бази шляхом одностороннього шифрування.

Всі запити до API проходять валідацію через DTO й декоратори class-validator, що допомагає уникати SQL-ін’єкцій, XSS та інших типових атак.

Система доступу до маршрутів реалізована через Guard-и NestJS. Вони перевіряють роль користувача та його права доступу. Завдяки цьому приватні маршрути, наприклад, оновлення профілю або створення транзакції, доступні лише тим, хто має відповідні права.

Усі важливі дії – такі як логін, зміна пароля, фінансові транзакції чи інші критичні запити – логуються у централізовану систему. Це дозволяє відстежувати підозрілу активність та швидко реагувати на потенційні загрози.

3.3 Взаємодія між сервісами

3.3.1 Вибір протоколів та механізмів комунікації

Взаємодія між мікросервісами є частиною розподіленої банківської системи. У проєкті вирішено поєднати як синхронні, так і асинхронні механізми комунікації, щоб досягти балансу між простотою, швидкістю та надійністю обміну даними між компонентами системи.

Для синхронної взаємодії використовується HTTP – це класичний REST API, який працює через API Gateway на базі Nginx. Такий підхід дозволяє зручно організувати запити до сервісів, стандартизувати формат передачі даних у вигляді JSON і легко масштабувати систему за рахунок балансування запитів на рівні шлюзу.

Але в деяких випадках синхронність є не найкращим рішенням. Наприклад, коли потрібно обробити подію або транзакцію у фоновому режимі, краще підходить асинхронна модель. Тому у таких випадках на допомогу приходить брокер повідомлень, який дозволяє сервісам обмінюватися подіями незалежно одне від одного. Для кожного сервісу створено окрему чергу, тому можна уникати зайвих затримок, блокувань і зробити обробку більш стабільною. RabbitMQ дозволяє реалізувати шаблони, як-от «publish/subscribe» або «work queue», що дає змогу рівномірно

розподіляти навантаження, особливо коли сервісів багато або є декілька екземплярів одного сервісу.

У результаті обрана архітектура дозволяє гнучко керувати взаємодією між сервісами, легко масштабувати окремі частини системи та безболісно додавати нові мікросервіси в майбутньому, що є важливим фактором правильно масштабованої системи.

3.3.2 Інтеграція та обробка міжсервісних запитів

Інтеграцію між мікросервісами в межах системи реалізовано через чітко визначені API та канали івентів. Кожен сервіс має свій власний набір REST-ендпоінтів.

Асинхронна взаємодія між сервісами реалізується через RabbitMQ. Наприклад, коли користувач здійснює транзакцію, сервіс платежів просто надсилає подію в чергу RabbitMQ, а вже інший сервіс отримує цю подію й зберігає інформацію про неї. Такий підхід дозволяє уникати сильних зв’язків між компонентами: сервіси не залежать один від одного напряму, що позитивно впливає на стійкість системи.

Також реалізовано трасування запитів через так звані correlation IDs – це унікальні ідентифікатори, які приходять разом із запитом між сервісами. Це дуже допомагає, коли треба зрозуміти, що саме відбулося у системі – наприклад, якщо користувач має проблемо, що «щось не працює», то можна за одним ідентифікатором простежити весь ланцюжок дій.

3.4 Алгоритм роботи системи

Нижче описаний загальний алгоритм роботи системи. Тобто немає алгоритму із залученням усіх сервісів, є лише основні: Auth, History та будь-який інший сервіс. Діаграма загального алгоритму роботи системи зображена у додатку Д3.

1. Користувач взаємодіє із системою, надсилає HTTP-запит до API Gateway

2. Запит потрапляє до API Gateway, який визначає, до якого мікросервісу його потрібно перенаправити, і здійснює маршрутизацію

3. Вибраний мікросервіс приймає запит, зчитує sessionId з cookie або заголовка, і ініціює перевірку сесії через Auth Service за допомогою RabbitMQ, якщо це потрібно

4. Auth Service отримує запит, перевіряє sessionId у Redis, повертає результат перевірки у відповідь через RabbitMQ

5. Якщо операція вимагає додаткової перевірки, мікросервіс надсилає запит до необхідного сервісу через RabbitMQ, наприклад для перевірки статусу користувача у Verification Service

6. Після проходження всіх перевірок мікросервіс виконує основну бізнес-операцію

7. Якщо в процесі виконання операції виникає подія, що потребує логування, мікросервіс надсилає повідомлення через RabbitMQ до History Service

8. History Service приймає повідомлення, зберігаючи інформацію про подію у своїй базі даних

9. Після завершення всіх операцій мікросервіс формує відповідь, яка повертається через API Gateway користувачу

ВИСНОВОК ДО РОЗДІЛУ 3

У третьому розділі було зосереджено увагу на практичній реалізації розробки банківської системи. Основні завдання, реалізовані в межах цього етапу, охоплюють структурування проєкту, налаштування інфраструктури, контейнеризацію, створення мікросервісів, реалізацію бізнес-логіки, механізмів безпеки та міжсервісної взаємодії.

Побудова проєкту на основі мікросервісної архітектури у форматі monorepo продемонструвала свою доцільність, оскільки дала змогу чітко розділити відповідальність між сервісами, одночасно зберігаючи централізовану структуру зручного управління залежностями. Такий підхід також забезпечив спрощене налаштування середовища розробки.

Налаштовано контейнеризацію середовища за допомогою Docker, що дозволило створити універсальну платформу для розгортання всіх сервісів незалежно від операційної системи. Інструмент docker-compose дозволив спростити процес одночасного запуску як основних сервісів, так і допоміжних компонентів.

У результаті, було досягнуто чіткого розмежування між обов’язками кожного мікросервісу: автентифікація, управління користувачами, верифікація, керування картками, платежі, логування історії операцій. Використання принципу зберігання усієї бізнес-логіки дозволило досягти більшої модульності та простоти підтримки коду.

Увагу приділено і захисту даних та безпеці, які були закладені у фундамент системи ще на етапі проєктування. Було впроваджено сучасні підходи до автентифікації користувачів із використанням сесій у Redis, а також реалізовано двофакторну автентифікацію для додаткового захисту облікових записів. Стандарти валідації даних, шифрування паролів, аудит дій користувачів та обмеження прав доступу дозволили досягти комплексного рівня захисту.

Ще одним критичним аспектом стала міжсервісна взаємодія, реалізована за допомогою синхронних REST API і асинхронної обробки подій за допомогою RabbitMQ. Такий підхід забезпечив не лише стабільну комунікацію між сервісами, але й можливість їх незалежного масштабування без порушення роботи всієї системи.

Таким чином, реалізація усіх перелічених технічних рішень та підходів створила фундамент для безпечної, гнучкої й масштабованої банківської системи.

РОЗДІЛ 4
ТЕСТУВАННЯ РОЗРОБЛЕНОЇ СИСТЕМИ

4.1 Мета та завдання тестування

Тестування є невід'ємною складовою процесу розробки сучасних програмних систем, особливо коли мова йдеться про критично важливі сфери, зокрема фінансові платформи. У межах розробки мікросервісної системи питання якості, надійності, безпеки та продуктивності є пріоритетними. Тому тестування розглядається не як фінальний етап, а як безперервний процес, інтегрований у всі фази життєвого циклу програмного забезпечення – від початкового проєктування до етапу експлуатації.

Головна мета тестування полягає у виявленні дефектів, перевірці відповідності реалізованої функціональності вимогам, а також у забезпеченні стабільної та передбачуваної роботи системи в умовах реального використання. Належне тестове покриття дозволяє мінімізувати ризики виникнення критичних помилок, які можуть призвести до фінансових втрат або ж компрометації безпеки.

Основні завдання тестування системи:

- Перевірка бізнес-логіки кожного сервісу
- Виявлення дефектів у міжсервісній взаємодії
- Перевірка стійкості до збоїв та навантажень

4.2 Огляд типів тестувань

У сучасній розробці програмного забезпечення широко використовується концепція «піраміди тестування». Вона ілюструє оптимальний розподіл різних типів тестів у проєкті, дозволяючи досягти балансу між швидкістю виконання, вартістю підтримки та глибиною перевірки системи.

Піраміда тестування складається з трьох основних рівнів [17]:

1. *Модульні тести (Unit Tests)* – перший рівень

Модульні тести є найчисленнішим і найшвидшим типом тестів. Вони перевіряють окремі функції, методи або класи в ізоляції – без залежності від зовнішніх сервісів чи бази даних. Такий підхід дозволяє ефективно перевіряти внутрішню бізнес-логіку, виявляти помилки ще на ранніх етапах розробки та досягати високого рівня покриття коду. Завдяки швидкому виконанню модульних тестів можна оперативно перевіряти зміни в коді, що є важливою умовою для підтримки стабільності системи під час активної розробки.

2. *Інтеграційні тести (Integration Tests)* – другий рівень

Інтеграційні тести забезпечують перевірку коректності взаємодії між модулями всередині одного мікросервісу або між кількома сервісами системи. На відміну від модульних тестів, ці тести охоплюють роботу із зовнішніми залежностями – зокрема, з базами даних, чергами повідомлень, зовнішніми API тощо. Інтеграційні тести допомагають виявити помилки у взаємодії компонентів, які не можна знайти на рівні модульних тестів. Такий підхід дозволяє зменшити ризики, пов’язані з інтеграційними помилками, і покращує загальну надійність платформи.

3. *End-to-end тести (e2e Tests)* – третій рівень

Це найменш численний, але найцінніший тип тестів, які перевіряють роботу всієї системи в цілому, включаючи всі сервіси, інфраструктурні компоненти та зовнішні інтеграції. End-to-end тести імітують реальні сценарії використання платформи з точки зору кінцевого користувача. Вони дозволяють переконатися, що всі частини системи працюють разом коректно, але виконуються повільніше та складніші у підтримці.

Загалом, така піраміда існує не просто так, у ній є певна залежність. Чим рівнем нижчі тести – тим вони швидше виконуються і дешевше коштують. І навпаки. Приклад такої піраміди зображено на рисунку 4.1.

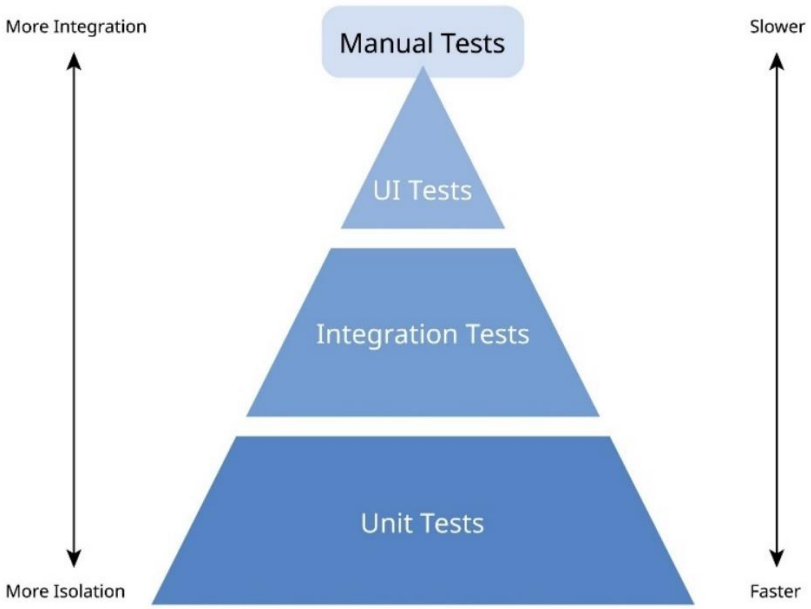


Рисунок 4.1 – Піраміда тестування

4.3 Тестування системи

У процесі розробки системи було обрано модульне тестування. Основною причиною такого вибору є те, що модульні тести дозволяють ізолювати та перевірити кожен окремий компонент системи. У мікросервісній архітектурі, де кожен сервіс виконує чітко визначену бізнес-функцію, таке тестування є найефективнішим способом гарантувати коректність реалізації логіки на ранніх етапах розробки.

Перевагами Unit тестування є:

- Швидкість виконання
- Локалізація помилок
- Високе покриття коду

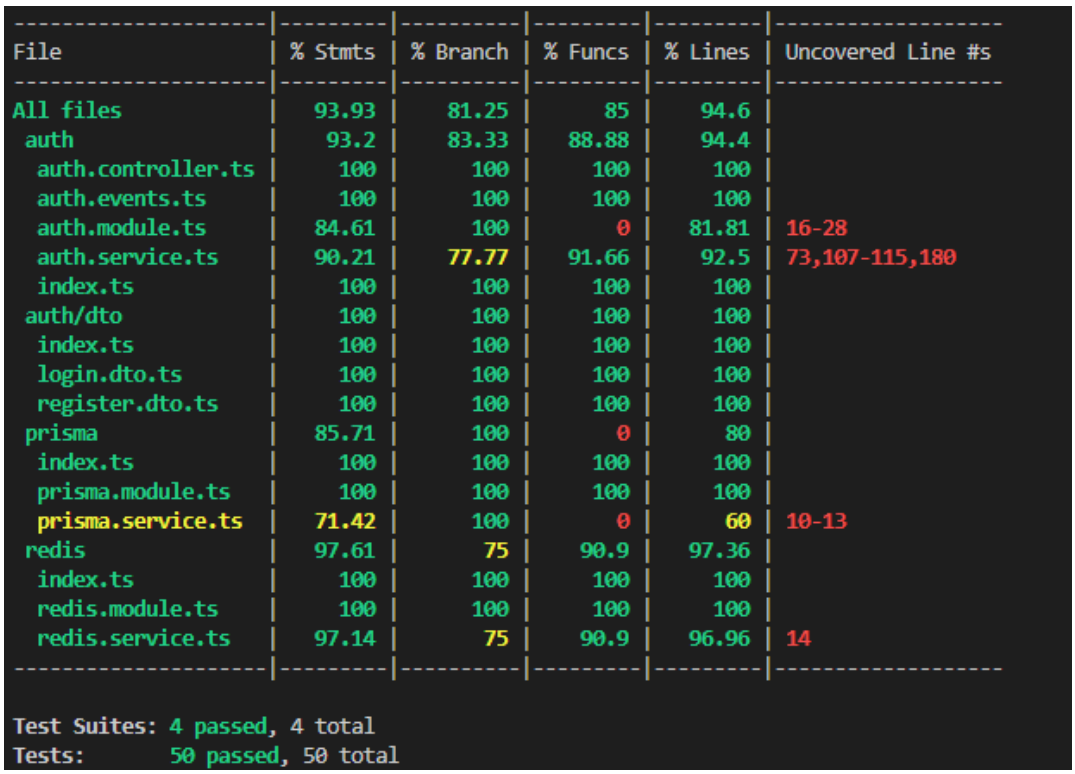
У дипломній роботі для написання тестів використовується фреймворк Jest, який є стандартом для NestJS [18, 19]. Тести розміщуються у папках __tests__ кожного мікросервісу та охоплюють основні сервіси, контролери, утиліти та допоміжні функції.

У процесі реалізації тестування для кожного мікросервісу проєкту створюються окремі файли з тестами, наприклад auth.service.spec.ts, user.service.spec.ts, payment.service.spec.ts та інші. Така структура дозволяє логічно розділити тести відповідно до функціональних областей системи та спрощує підтримку коду. У тестових сценаріях активно використовуються mock-об’єкти, які імітують поведінку зовнішніх залежностей, таких як база даних, сторонні API чи брокер повідомлень RabbitMQ. Це дозволяє ізолювати логіку окремих компонентів і перевіряти їхню роботу незалежно від зовнішнього середовища.

Тестові кейси охоплюють як позитивні, так і негативні сценарії. Перевіряється не лише правильна обробка коректних даних, а й реакція системи на некоректні вхідні параметри, невалідні запити чи виняткові ситуації.

Для складніших бізнес-процесів, наприклад, валідації платіжної інформації, генерації токенів чи обробки транзакцій, створюються окремі тестові сценарії з чіткою логікою перевірок. Це дозволяє впевнено оцінити коректність реалізації критичних операцій і зменшити ризик помилок у роботі з фінансовими даними.

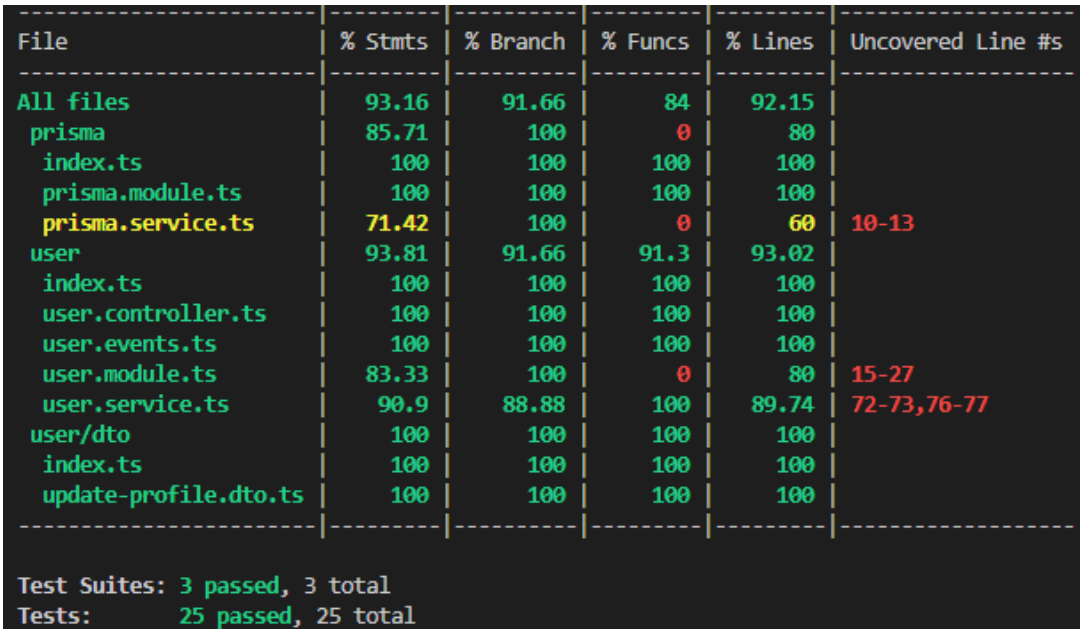
Приклади тестування системи і покриття сервісів зображені на рисунках 4.2 – 4.6.



File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	93.93	81.25	85	94.6	
auth	93.2	83.33	88.88	94.4	
auth.controller.ts	100	100	100	100	
auth.events.ts	100	100	100	100	
auth.module.ts	84.61	100	0	81.81	16-28
auth.service.ts	90.21	77.77	91.66	92.5	73,107-115,180
index.ts	100	100	100	100	
auth/dto	100	100	100	100	
index.ts	100	100	100	100	
login.dto.ts	100	100	100	100	
register.dto.ts	100	100	100	100	
prisma	85.71	100	0	80	
index.ts	100	100	100	100	
prisma.module.ts	100	100	100	100	
prisma.service.ts	71.42	100	0	60	10-13
redis	97.61	75	90.9	97.36	
index.ts	100	100	100	100	
redis.module.ts	100	100	100	100	
redis.service.ts	97.14	75	90.9	96.96	14

Test Suites: 4 passed, 4 total
Tests: 50 passed, 50 total

Рисунок 4.2 – Покриття тестами мікросервісу Auth



File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	93.16	91.66	84	92.15	
prisma	85.71	100	0	80	
index.ts	100	100	100	100	
prisma.module.ts	100	100	100	100	
prisma.service.ts	71.42	100	0	60	10-13
user	93.81	91.66	91.3	93.02	
index.ts	100	100	100	100	
user.controller.ts	100	100	100	100	
user.events.ts	100	100	100	100	
user.module.ts	83.33	100	0	80	15-27
user.service.ts	90.9	88.88	100	89.74	72-73,76-77
user/dto	100	100	100	100	
index.ts	100	100	100	100	
update-profile.dto.ts	100	100	100	100	

Test Suites: 3 passed, 3 total
Tests: 25 passed, 25 total

Рисунок 4.3 – Покриття тестами мікросервісу User

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	95.52	94.87	73.91	94.78	
prisma	85.71	100	0	80	
index.ts	100	100	100	100	
prisma.module.ts	100	100	100	100	
prisma.service.ts	71.42	100	0	60	10-13
s3	100	100	100	100	
index.ts	100	100	100	100	
s3.service.ts	100	100	100	100	
verification	95.18	89.47	66.66	94.52	
index.ts	100	100	100	100	
verification.controller.ts	100	83.33	100	100	26
verification.events.ts	77.77	100	0	71.42	7,11
verification.module.ts	83.33	100	0	80	15-27
verification.service.ts	100	100	100	100	
verification/dto	100	100	100	100	
admin-verify.dto.ts	100	100	100	100	
index.ts	100	100	100	100	

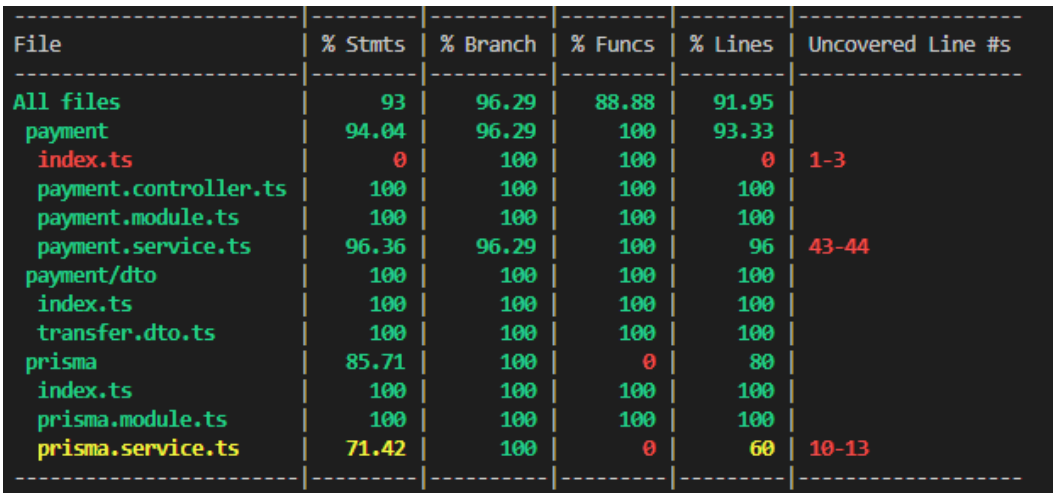
Test Suites: 3 passed, 3 total
Tests: 31 passed, 31 total

Рисунок 4.4 – Покриття тестами мікросервісу Verification

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	81.48	90	80.64	81.19	
card	80.67	90	86.2	80.95	
card.controller.ts	100	100	100	100	
card.events.ts	100	100	100	100	
card.module.ts	0	100	0	0	1-52
card.service.ts	87.09	90	92.3	87.03	17-35
index.ts	0	100	100	0	1-3
card/dto	100	100	100	100	
index.ts	100	100	100	100	
open-card.dto.ts	100	100	100	100	
prisma	85.71	100	0	80	
index.ts	100	100	100	100	
prisma.module.ts	100	100	100	100	
prisma.service.ts	71.42	100	0	60	10-13

Test Suites: 3 passed, 3 total
Tests: 32 passed, 32 total

Рисунок 4.5 – Покриття тестами мікросервісу Card



File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	93	96.29	88.88	91.95	
payment	94.04	96.29	100	93.33	
index.ts	0	100	100	0	1-3
payment.controller.ts	100	100	100	100	
payment.module.ts	100	100	100	100	
payment.service.ts	96.36	96.29	100	96	43-44
payment/dto	100	100	100	100	
index.ts	100	100	100	100	
transfer.dto.ts	100	100	100	100	
prisma	85.71	100	0	80	
index.ts	100	100	100	100	
prisma.module.ts	100	100	100	100	
prisma.service.ts	71.42	100	0	60	10-13

Рисунок 4.6 – Покриття тестами мікросервісу Payment

Отже, якщо подивитися на загальні результати покриття тестами коду, то майже увесь код покривається повністю. Загалом, прийнятий відсоток покриття є відносним. Кожна людина або компанія встановлює його під свої потреби. Згідно загальним значенням, прийнятним є 60%, достойним – 75%, зразковим – 90% [20]. Тим не менше, розробники стараються встановлювати рамки «80/20», де 80% коду є покритим. Тому і для тестування системи було прийнято порогове значення у вигляді 75-80%.

ВИСНОВОК ДО РОЗДІЛУ 4

Четвертий розділ, що відноситься до етапу тестування, засвідчив важливість безперервного контролю якості програмного забезпечення під час розробки системи.

Ретельно побудована стратегія тестування, що ґрунтується на принципах піраміди тестування, дозволила досягти балансу між покриттям, глибиною перевірки та продуктивністю. Основна увага була зосереджена на модульному тестуванні, як на найбільш доступному, швидкому й стабільному інструменті контролю за коректністю реалізації окремих логічних блоків. Застосування Jest як тестового фреймворку забезпечило легкість написання та підтримки тестів у середовищі NestJS.

Особливої уваги заслуговує підхід до структурування тестових кейсів. Вони були розроблені як для позитивних сценаріїв – із валідними вхідними даними, так і для обробки винятків, помилкових запитів, порушень авторизації та інших аномальних ситуацій. Така стратегія дозволила перевірити не лише коректність поведінки, але й її передбачуваність у різних умовах.

Рівень покриття коду, що становить близько 75–80%, є достатнім для впевненості в правильності роботи частин системи, що підлягалися тестуванню. Це значення підтверджує, що основні логічні гілки протестовані, а відсутність сліпих зон у критичних модулях знижує ризик неочікуваних помилок.

ВИСНОВКИ

У межах виконання дипломної роботи було проведено комплексне дослідження проблематики створення сучасної банківської системи з урахуванням актуальних тенденцій цифрової трансформації, високих вимог до надійності, масштабованості, продуктивності та безпеки. Результатом цього дослідження стала повноцінна реалізація банківської платформи, побудованої за мікросервісною архітектурою.

У першому розділі було обґрунтовано аналіз предметної області, огляду сучасного стану банківського сектору та порівнянню існуючих платформ. Це дало змогу сформувати чітке розуміння вимог до функціональності, архітектури та захисту даних. Виявлено, що саме мікросервісний підхід сьогодні найкраще відповідає потребам фінансових установ, дозволяючи ефективно масштабувати систему та гнучко адаптувати її до нових бізнес-вимог.

Другий розділ був присвячений проєктуванню системи: обґрунтовано вибір технологій, визначено загальну архітектуру, спроєктовано бази даних, реалізовано стратегії інформаційної безпеки та масштабування. Особлива увага приділялася дотриманню принципів ізоляції сервісів, їх автономності, а також ефективному управлінню станом користувачів. Ретельно продумана структура дозволяє адаптувати платформу до вимог конкретної фінансової установи.

У третьому розділі висвітлено практичні аспекти розробки системи: реалізовано ключові мікросервіси, описано особливості їхньої взаємодії через RabbitMQ та API Gateway, налаштовано контейнеризацію через Docker та створено повноцінну інфраструктуру розгортання. Було дотримано принципів модульності, повторного використання коду та мінімізації зв’язності між сервісами, що відповідає кращим практикам інженерії програмного забезпечення.

Четвертий розділ був присвячений етапу перевірки якості реалізованої системи. Тестування здійснювалося у форматі модульного тестування, що дозволило перевірити коректність бізнес-логіки кожного окремого сервісу. Для цього було використано Jest – сучасний інструмент тестування в середовищі Node.js. Основна увага приділялася сценаріям з коректними та некоректними вхідними даними, а також перевірці виняткових ситуацій. Такий підхід забезпечив необхідний рівень впевненості в стабільності системи, дозволяючи швидко виявляти та усувати логічні помилки.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. PrivatBank About the Bank [Електронний ресурс] – Режим доступу до ресурсу: https://privatbank.ua/en/about.
2. Monobank First Mobile Bank [Електронний ресурс] – Режим доступу до ресурсу: https://techukraine.org/portfolio/monobank/.
3. SenseBank Про Банк [Електронний ресурс] – Режим доступу до ресурсу: https://sensebank.ua/about.
4. Ощадбанк Про Банк [Електронний ресурс] – Режим доступу до ресурсу: https://www.oschadbank.ua/about.
5. Microservices vs. monolithic architecture [Електронний ресурс] – Режим доступу до ресурсу: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith.
6. Monolithic architecture [Електронний ресурс] – Режим доступу до ресурсу: https://www.techtarget.com/whatis/definition/monolithic-architecture.
7. Microservices architecture [Електронний ресурс] – Режим доступу до ресурсу: https://www.atlassian.com/microservices/microservices-architecture.
8. Закон України «Про захист персональних даних» [Електронний ресурс] – Режим доступу до ресурсу: https://zakon.rada.gov.ua/laws/show/2297-17#Text.
9. NestJS Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://docs.nestjs.com/.
10. RabbitMQ Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://www.rabbitmq.com/docs.
11. PostgreSQL Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://www.postgresql.org/docs/.

12. Redis Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://martinfowler.com/articles/microservices.html
13. Docker Guides [Електронний ресурс] – Режим доступу до ресурсу: https://docs.docker.com/guides/.
14. Nginx Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://nginx.org/en/docs/.
15. RabbitMQ Patterns [Електронний ресурс] – Режим доступу до ресурсу: https://danmartensen.svbtle.com/rabbitmq-message-broker-patterns.
16. Monorepo vs. Multi-Repo [Електронний ресурс] – Режим доступу до ресурсу: https://kinsta.com/blog/monorepo-vs-multi-repo/.
17. What is Testing Pyramid? [Електронний ресурс] – Режим доступу до ресурсу: https://testsigma.com/blog/testing-pyramid/.
18. Jest Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://jestjs.io/docs/getting-started.
19. Microservices Testing [Електронний ресурс] – Режим доступу до ресурсу: https://www.lambdatest.com/learning-hub/microservices-testing
20. How to Decide on Unit Test Coverage [Електронний ресурс] – Режим доступу до ресурсу: https://softteco.com/blog/unit-test-coverage

ДОДАТОК 1

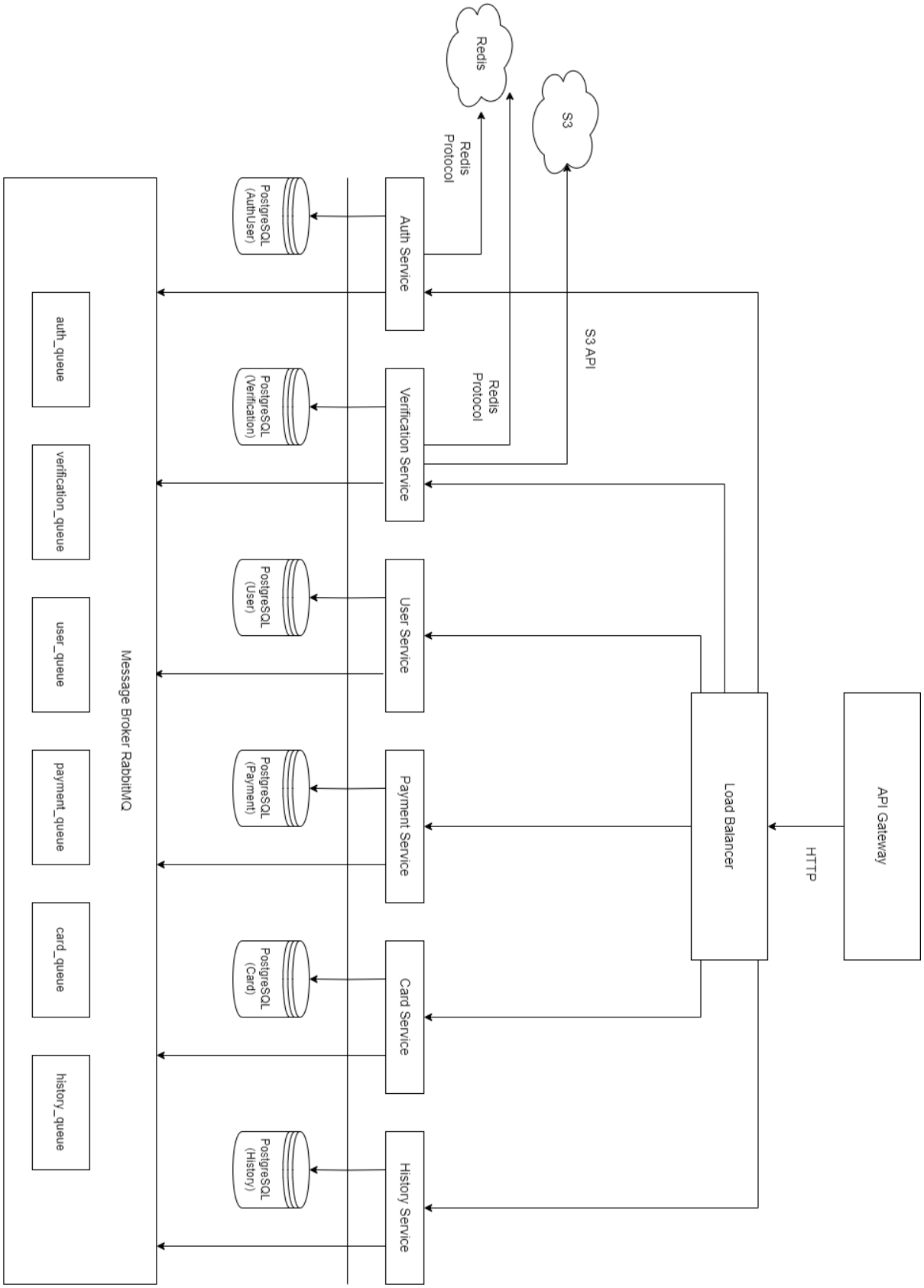
Банківська система на основі мікросервісної архітектури

Схема архітектури системи

ІАЛЦ.467200.004 Д1

Аркушів 1

Київ - 2025 р



					ІАЛЦ.467200.004 Д1	
		№ докум.	Підпис	Дата		
Розробив		Казак В.С.			**Банківська система на основі мікросервісної архітектури** Схема архітектури системи	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Русінов В.В.				
Реценз.		Шимкович В. М.				КПІ ім. Ігоря Сікорського, ФІОТ, ІО-13
Н. Контр.		Пономаренко А.М.				
Затвердив						

ДОДАТОК 2

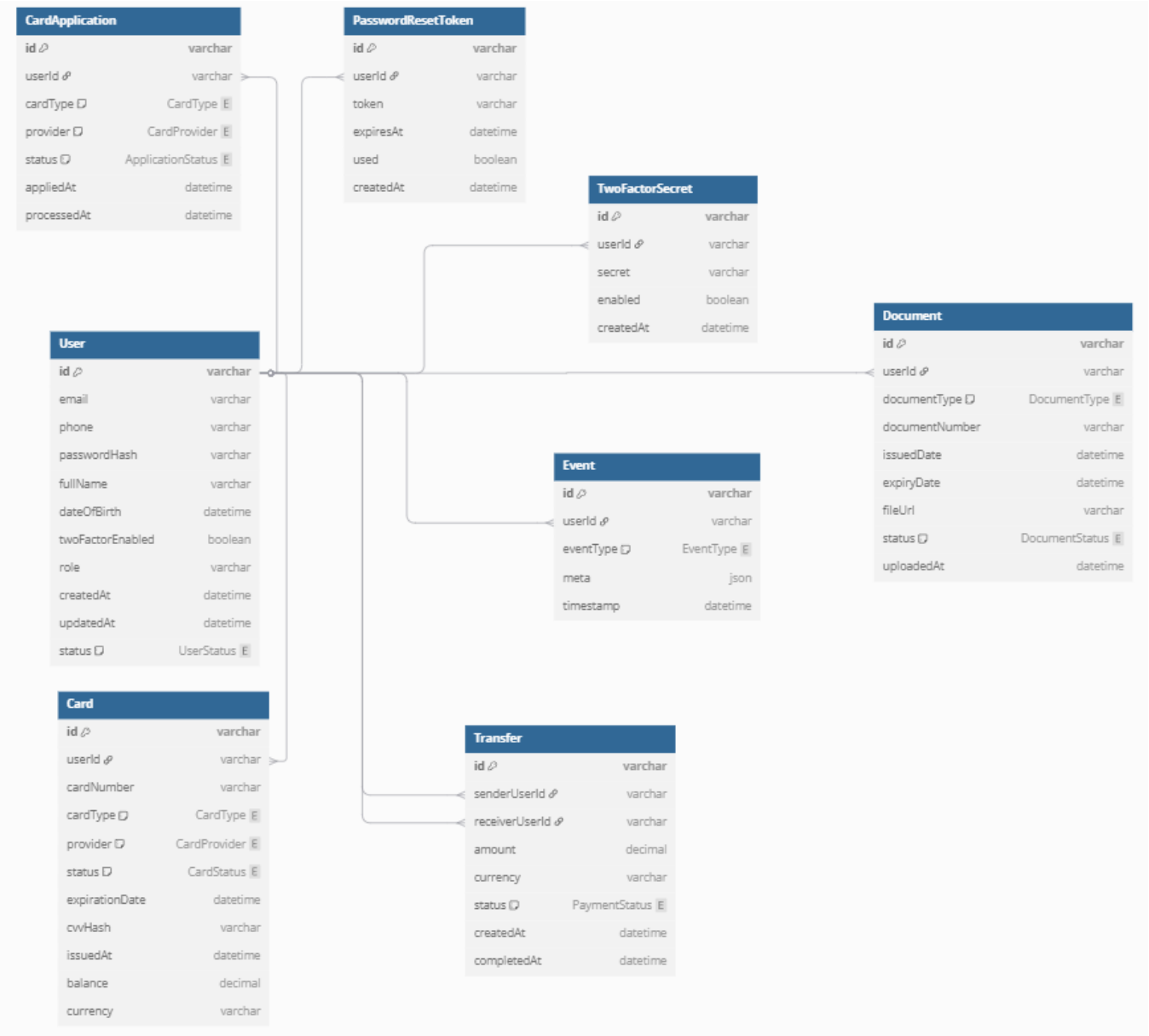
Банківська система на основі мікросервісної архітектури

Схема баз даних

ІАЛЦ.467200.005 Д2

Аркушів 1

Київ - 2025 р



					ІАЛЦ.467200.005 Д2	
		№ докум.	Підпис	Дата		
Розробив		Казак В.С.			**Банківська система на основі мікросервісної архітектури** Схема баз даних	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Русінов В.В.				
Реценз.		Шимкович В. М.				КПІ ім. Ігоря Сікорського, ФІОТ, ІО-13
Н. Контр.		Пономаренко А.М.				
Затвердив						

ДОДАТОК 3

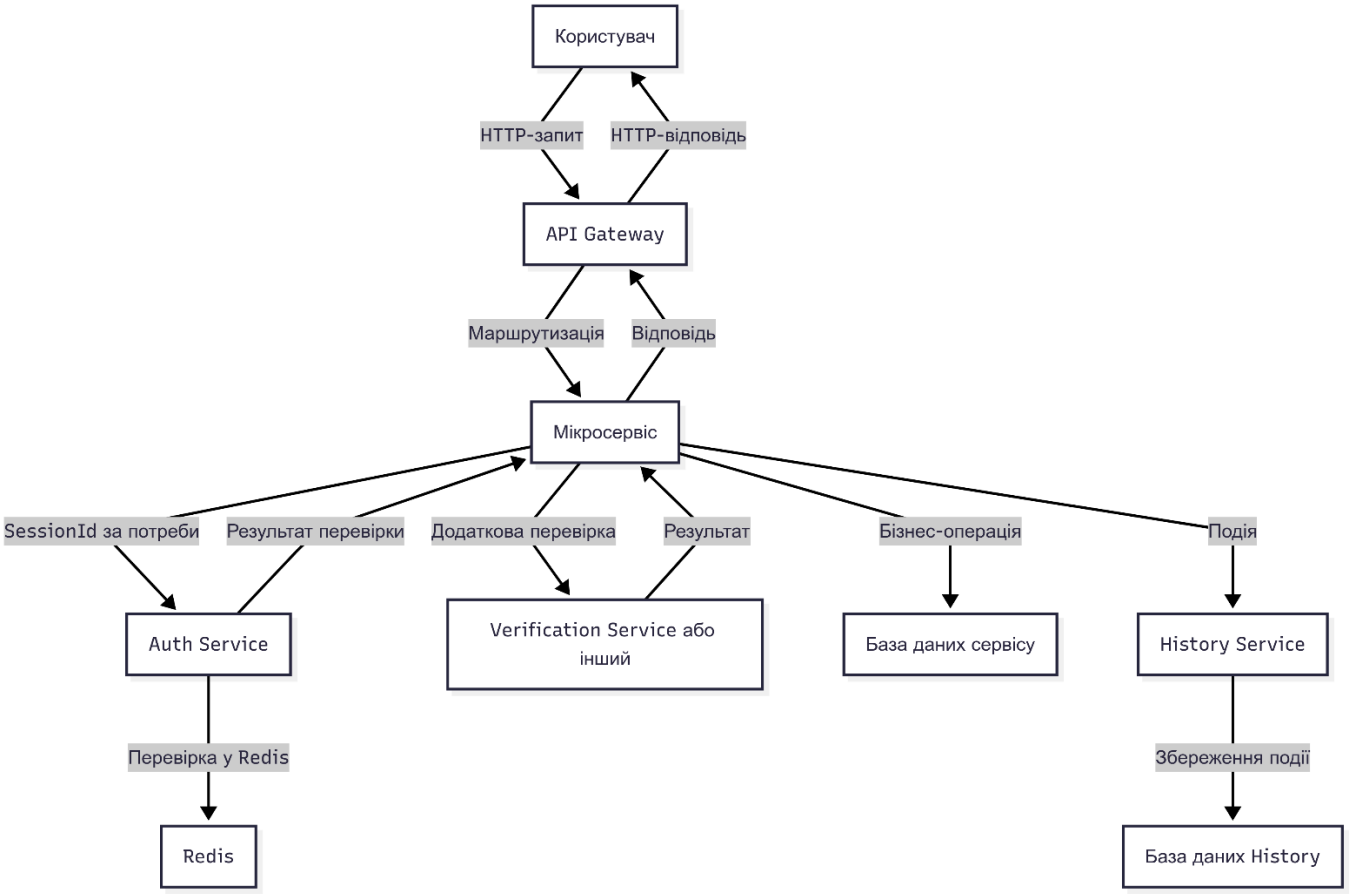
Банківська система на основі мікросервісної архітектури

Алгоритм роботи системи

ІАЛЦ.467200.006 Д3

Аркушів 1

Київ - 2025 р



					ІАЛЦ.467200.006 Д3	
		№ докум.	Підпис	Дата		
Розробив		Казак В.С.			**Банківська система на основі мікросервісної архітектури** Алгоритм роботи системи	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Русінов В.В.				
Реценз.		Шимкович В. М.				КПІ ім. Ігоря Сікорського, ФІОТ, ІО-13
Н. Контр.		Пономаренко А.М.				
Затвердив						

ДОДАТОК 4

Банківська система на основі мікросервісної архітектури

Текст програмного коду

ІАЛЦ.467200.007 Д4

Аркушів 44

Київ - 2025 р

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { MicroserviceOptions, Transport } from '@nestjs/microservices';
import * as cookieParser from 'cookie-parser';
import { RpcToHttpExceptionFilter } from '../common/exceptions';
import { ConfigService } from '@nestjs/config';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  const configService = app.get(ConfigService);
  app.use(cookieParser());
  app.useGlobalFilters(new RpcToHttpExceptionFilter());

  app.connectMicroservice<MicroserviceOptions>({
    transport: Transport.RMQ,
    options: {
      urls: [configService.get<string>('RABBITMQ_URL')],
      queue: configService.get<string>('RABBITMQ_QUEUE'),
      queueOptions: {
        durable: true,
      },
    },
  });

  await app.startAllMicroservices();

  const port = configService.get<number>('PORT') ?? 3000;

  await app.listen(port);
  console.log(`Auth Service is running on http://localhost:${port}`);
}

bootstrap();

import { Module } from '@nestjs/common';
import { AuthModule } from './auth';
import { APP_PIPE } from '@nestjs/core';
import { ValidationPipe } from '@nestjs/common'
```

```
import { PrismaModule } from './prisma';
import { RedisModule } from './redis';
import { ConfigModule } from '@nestjs/config';

@Module({
  imports: [
    AuthModule,
    PrismaModule,
    RedisModule,
    ConfigModule.forRoot({
      isGlobal: true,
    }),
  ],
  controllers: [],
  providers: [
    {
      provide: APP_PIPE,
      useClass: ValidationPipe,
    },
  ],
})
export class AppModule {}



import { NestFactory } from '@nestjs/core';
import { MicroserviceOptions, Transport } from '@nestjs/microservices';
import { AppModule } from './app.module';
import { ConfigService } from '@nestjs/config';
import { AmqpTransport, ExchangeType } from '@nestjstools/microservices-rabbitmq';

async function bootstrap() {
  const appContext = await NestFactory.create(AppModule);
  const configService = appContext.get(ConfigService);

  const microservice =
    await NestFactory.createMicroservice<MicroserviceOptions>(AppModule, {
      transport: Transport.RMQ,
      options: {
        urls: [configService.get<string>('RABBITMQ_URL')],
        queue: configService.get<string>('RABBITMQ_QUEUE'),
        queueOptions: { durable: true },
      },
    });
```

```
  await microservice.listen();
  console.log('Auth Worker is listening for RabbitMQ messages...');
}
bootstrap();
```

```
version: '3.8'

services:
  auth-service:
      build: .
      container_name: auth-service
      environment:
        DATABASE_URL: ${DATABASE_URL}
        RABBITMQ_URL: ${RABBITMQ_URL}
        REDIS_URL: ${REDIS_URL}
      volumes:
        - ./:/app
      working_dir: /app
      command: sh -c "npm install && npm run start:dev"
      ports:
        - "3000:3000"
      networks:
        - postgres-net
        - rabbitmq-net
        - redis-net
        - nginx-gateway
  auth-worker:
    build:
      context: .
    container_name: auth-worker
    command: npm run start:worker
    environment:
      NODE_ENV: production
      RABBITMQ_URL: ${RABBITMQ_URL}
    depends_on:
      - auth-service
    networks:
      - rabbitmq-net
      - postgres-net
      - redis-net
```

```
networks:
  postgres-net:
    external: true
  rabbitmq-net:
    external: true
  redis-net:
    external: true
  nginx-gateway:
    external: true
```

```
import { Module } from '@nestjs/common';
import { AuthService } from './auth.service';
import { AuthController } from './auth.controller';
import { RedisService } from '../redis';
import { SessionGuard } from '../../common/guards';
import { ClientsModule, Transport } from '@nestjs/microservices';
import { AuthEventsController } from './auth.events';
import { ConfigService } from '@nestjs/config';

@Module({
  imports: [
    ClientsModule.registerAsync([
      {
        name: 'USER_SERVICE',
        inject: [ConfigService],
        useFactory: (configService: ConfigService) => ({
          transport: Transport.RMQ,
          options: {
            urls: [configService.get<string>('RABBITMQ_URL')],
            queue: configService.get<string>('RABBITMQ_QUEUE_USER'),
            queueOptions: { durable: true },
          },
        }),
      },
    ]),
  ],
  controllers: [AuthController, AuthEventsController],
  providers: [AuthService, RedisService, SessionGuard],
  exports: [AuthService],
})
export class AuthModule {}
```

```
import { Controller, Post, Body, Req, Res, UseGuards } from '@nestjs/common';
import { AuthService } from './auth.service';
import { RegisterDto, LoginDto } from './dto';
import { Request, Response } from 'express';
import { SessionGuard } from '../../common/guards';

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

  @Post('register')
  async register(@Body() dto: RegisterDto) {
    return this.authService.register(dto);
  }

  @Post('login')
  async login(
    @Body() dto: LoginDto,
    @Res({ passthrough: true }) res: Response,
  ) {
    const result = await this.authService.login(dto);
    if (result.require2fa) {
      return result;
    }
    res.cookie('sessionId', result.sessionId, {
      httpOnly: true,
      sameSite: 'lax',
    });

    return { success: true };
  }

  @UseGuards(SessionGuard)
  @Post('2fa/setup')
  async setup2fa(@Req() req: Request) {
    return this.authService.setup2fa(req['user'].userId);
  }

  @UseGuards(SessionGuard)
  @Post('2fa/enable')
  async enable2fa(@Req() req: Request, @Body() body: { code: string }) {
    return this.authService.enable2fa(req['user'].userId, body.code);
  }
```

```
@UseGuards(SessionGuard)
  @Post('2fa/disable')
  async disable2fa(@Req() req: Request) {
    return this.authService.disable2fa(req['user'].userId);
  }

  @Post('2fa/verify')
  async verify2fa(
    @Body() body: { userId: string; code: string },
    @Res({ passthrough: true }) res: Response,
  ) {
    const result = await this.authService.verify2fa(body.userId, body.code);

    res.cookie('sessionId', result.sessionId, {
      httpOnly: true,
      sameSite: 'lax',
    });
    return { success: true };
  }

  @Post('password-reset/request')
  async requestPasswordReset(@Body() body: { email: string }) {
    return this.authService.requestPasswordReset(body.email);
  }

  @Post('password-reset/confirm')
  async resetPassword(@Body() body: { token: string; newPassword: string }) {
    return this.authService.resetPassword(body.token, body.newPassword);
  }

  @UseGuards(SessionGuard)
  @Post('logout')
  async logout(@Req() req: Request, @Res({ passthrough: true }) res: Response) {
    const sessionId = req.cookies?.sessionId;
    await this.authService.logout(sessionId);
    res.clearCookie('sessionId');
    return { success: true, sessionIdEnded: sessionId };
  }

  @UseGuards(SessionGuard)
  @Post('logout-all')
  async logout_all(
    @Req() req: Request,
```

```
    @Res({ passthrough: true }) res: Response,
  ) {
    const userId = req['user'].userId;
    await this.authService.logoutAll(userId);
    res.clearCookie('sessionId');
    return { success: true };
  }
}



import {
  Injectable,
  BadRequestException,
  UnauthorizedException,
  Inject,
  NotFoundException,
  ForbiddenException,
} from '@nestjs/common';
import { RegisterDto, LoginDto } from './dto';
import { RedisService } from '../redis';
import { PrismaService } from '../prisma';
import { randomUUID } from 'crypto';
import { ClientProxy } from '@nestjs/microservices';
import { firstValueFrom, timeout } from 'rxjs';
import * as bcrypt from 'bcryptjs';
import * as speakeasy from 'speakeasy';

@Injectable()
export class AuthService {
  constructor(
    @Inject('USER_SERVICE') private readonly userClient: ClientProxy,
    private readonly redisService: RedisService,
    private readonly prisma: PrismaService,
  ) {}

  async register(dto: RegisterDto) {
    const passwordHash = await bcrypt.hash(dto.password, 10);
    const user = await firstValueFrom(
      this.userClient
        .send({ cmd: 'create-user' }, {
          email: dto.email,
          phone: dto.phone,
          fullName: dto.fullName
```

```
passwordHash,
          dateOfBirth: dto.dateOfBirth,
          createdAt: new Date(),
          updatedAt: new Date(),
          status: 'unverified',
        })
        .pipe(timeout(3000)),
    );
    return { id: user.id, email: user.email, phone: user.phone };
  }

  async login(dto: LoginDto) {
    const user = await firstValueFrom(this.userClient
      .send({ cmd: 'find-user' }, {
        phone: dto.phone,
      }).pipe(timeout(3000)));

    if (!user) throw new UnauthorizedException('User not found');

    const isMatch = await bcrypt.compare(dto.password, user.passwordHash);
    if (!isMatch) throw new UnauthorizedException('Invalid credentials');

    // 2FA check
    const tfa = await this.prisma.twoFactorSecret.findFirst({
      where: { userId: user.id, enabled: true },
    });
    if (tfa) {
      return { require2fa: true };
    }

    const sessionId = randomUUID();
    await this.redisService.setSession(sessionId, { userId: user.id });
    return { sessionId };
  }

  // --- 2FA ---
  async setup2fa(userId: string) {
    await this.prisma.twoFactorSecret.deleteMany({ where: { userId } });

    const secret = speakeasy.generateSecret();
    await this.prisma.twoFactorSecret.create({
      data: { userId, secret: secret.base32, enabled: false },
    });
```

```
return { otpauth_url: secret.otpauth_url, secret: secret.base32 };
}

async enable2fa(userId: string, code: string) {
  const tfa = await this.prisma.twoFactorSecret.findFirst({
    where: { userId, enabled: false },
  });
  if (!tfa) throw new NotFoundException('2FA setup not found');
  const verified = speakeasy.totp.verify({
    secret: tfa.secret,
    encoding: 'base32',
    token: code,
  });
  if (!verified) throw new ForbiddenException('Invalid 2FA code');
  await this.prisma.twoFactorSecret.update({
    where: { id: tfa.id },
    data: { enabled: true },
  });
  await firstValueFrom(this.userClient.send({ cmd: 'enable-2fa' }, { userId }));
  return { success: true };
}

async verify2fa(userId: string, code: string) {
  const tfa = await this.prisma.twoFactorSecret.findFirst({
    where: { userId, enabled: true },
  });
  if (!tfa) throw new NotFoundException('2FA not enabled');
  const verified = speakeasy.totp.verify({
    secret: tfa.secret,
    encoding: 'base32',
    token: code,
  });

  if (!verified) throw new ForbiddenException('Invalid 2FA code');

  const sessionId = randomUUID();
  await this.redisService.setSession(sessionId, { userId });
  return { sessionId };
}

async disable2fa(userId: string) {
  const tfa = await this.prisma.twoFactorSecret.findFirst({
    where: { userId, enabled: true },
```

```
  });
  if (!tfa) throw new NotFoundException('2FA not enabled');
  await this.prisma.twoFactorSecret.deleteMany({ where: { userId } });
  await firstValueFrom(this.userClient.send({ cmd: 'disable-2fa' }, { userId }));
  return { success: true };
}

// --- Password Reset ---
async requestPasswordReset(email: string) {
  const user = await firstValueFrom(this.userClient
    .send({ cmd: 'find-user' }, { email }));
  if (!user) throw new NotFoundException('User not found');
  const token = randomUUID();
  const expiresAt = new Date(Date.now() + 1000 * 60 * 15); // 15 minutes
  await this.prisma.passwordResetToken.create({
    data: { userId: user.id, token, expiresAt },
  });
  // TODO: send email with token
  return { token, expiresAt };
}

async resetPassword(token: string, newPassword: string) {
  const prt = await this.prisma.passwordResetToken.findUnique({
    where: { token },
  });
  if (!prt || prt.used || prt.expiresAt < new Date()) {
    await this.prisma.passwordResetToken
      .delete({ where: { token } })
      .catch(() => {});
    throw new BadRequestException('Invalid or expired token');
  }
  const passwordHash = await bcrypt.hash(newPassword, 10);
  await firstValueFrom(this.userClient
    .send(
      { cmd: 'update-user-password' },
      { userId: prt.userId, passwordHash },
    ));
  await this.prisma.passwordResetToken.delete({ where: { token } });
  return { success: true };
}

async logout(sessionId: string) {
  if (!sessionId) throw new BadRequestException('No sessionId provided');
```

```
    await this.redisService.deleteSession(sessionId);
    return { success: true };
  }

  async logoutAll(userId: string) {
    const sessions = await this.redisService.getSessionsByUserId(userId);

    if (!sessions || sessions.length === 0) {
      return { success: true, message: 'No sessions found for this user' };
    }

    await this.redisService.deleteAllUserSessions(userId);
    return { success: true };
  }
}

// go register
// go login
// if 2fa is not enabled, u can login and cookies are set up
// if 2fa is enabled, u have to verify 2fa code and then cookies are set up on /verify

// to enable 2fa u have to go /2fa/setup
// then u have to scan qr code with authenticator app and enter code from app to /2fa/enable
// after that u can login with 2fa

// to disable 2fa u have to go /2fa/disable


import {
  Injectable,
  Inject,
  BadRequestException,
  ConflictException,
} from '@nestjs/common';
import { PrismaService } from '../prisma';
import { RpcException } from '@nestjs/microservices';
import * as bcrypt from 'bcryptjs';
import { UpdateProfileDto } from './dto';
import { ClientProxy } from '@nestjs/microservices';
import { UserStatus } from '../../common/types/user-status.d';

@Injectable()
export c
```

```
lass UserService {
  constructor(
    private readonly prisma: PrismaService,
    @Inject('AUTH_SERVICE') private readonly authClient: ClientProxy,
  ) {}

  async findByEmail(email: string) {
    return this.prisma.user.findUnique({ where: { email } });
  }

  async findByPhone(phone: string) {
    return this.prisma.user.findUnique({ where: { phone } });
  }

  async create(data: any) {
    const exists = await this.prisma.user.findFirst({
      where: {
        OR: [{ email: data.email }, { phone: data.phone }],
      },
    });

    if (exists)
      throw new RpcException({
        status: 409,
        message: 'User with this email or phone already exists',
      });

    if (data.dateOfBirth) data.dateOfBirth = new Date(data.dateOfBirth);

    return this.prisma.user.create({ data });
  }

  async delete(id: string) {
    return this.prisma.user.delete({ where: { id } });
  }

  async updatePassword(userId: string, passwordHash: string) {
    return this.prisma.user.update({
      where: { id: userId },
      data: { passwordHash },
    });
  }
```

```
 async updateProfile(id: string, data: UpdateProfileDto) {
    const updateData: any = {};
    if (data.email) updateData.email = data.email;
    if (data.phone) {
      const existingUser = await this.prisma.user.findUnique({ where: { phone: data.phone }
});
      if (existingUser && existingUser.id !== id) {
        throw new ConflictException('User with this phone already exists');
      }
      updateData.phone = data.phone;
    }
    if (data.password)
      updateData.passwordHash = await bcrypt.hash(data.password, 10);

    if (Object.keys(updateData).length === 0) {
      throw new BadRequestException('No data to update');
    }
    return this.prisma.user.update({ where: { id }, data: updateData });
  }

  async enable2FA(userId: string) {
    return this.prisma.user.update({
      where: { id: userId },
      data: { twoFactorEnabled: true },
    });
  }

  async disable2FA(userId: string) {
    return this.prisma.user.update({
      where: { id: userId },
      data: { twoFactorEnabled: false },
    });
  }

  async updateUserStatus(userId: string, status: UserStatus): Promise<any> {
    return this.prisma.user.update({
      where: { id: userId },
      data: { status },
    });
  }
}
```

```
import { Injectable, InternalServerErrorException, Inject } from '@nestjs/common';
import { S3Service } from '../s3/s3.service';
import { File } from 'multer';
import { PrismaService } from '../prisma/prisma.service';
import { ClientProxy, ClientProxyFactory, Transport } from '@nestjs/microservices';
import { firstValueFrom } from 'rxjs';

@Injectable()
export class VerificationService {
  constructor(
    private readonly s3Service: S3Service,
    private readonly prisma: PrismaService,
    @Inject('USER_SERVICE') private readonly userClient: ClientProxy,
  ) {}

  async verifyUser(userId: string, file: File, documentType: string): Promise<any> {
    try {
      const fileUrl = await this.s3Service.uploadFile(file);

      const document = await this.prisma.document.create({
        data: {
          userId,
          documentType: documentType as any,
          documentNumber: '',
          issuedDate: new Date(),
          expiryDate: new Date(),
          fileUrl,
          status: 'pending'
        },
      });

      const verificationStatus = await this.prisma.verificationStatus.create({
        data: {
          userId,
          status: 'verified',
          comment: null,
        },
      });

      await firstValueFrom(this.userClient.send({ cmd: 'user_verified' }, { userId, status:
'active' }));

      return { userId, document, verificationStatus };
```

```
   } catch (error: any) {
      throw new InternalServerErrorException(error.message);
    }
  }
}


generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

model PasswordResetToken {
  id        String   @id @default(uuid())
  userId    String
  token     String   @unique
  expiresAt DateTime
  used      Boolean  @default(false)
  createdAt DateTime @default(now())
}

model TwoFactorSecret {
  id        String   @id @default(uuid())
  userId    String
  secret    String
  enabled   Boolean  @default(false)
  createdAt DateTime @default(now())
}



generator client {
  provider = "prisma-client-js"
  output = "../node_modules/.prisma/client"
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
```

```
model User {
  id           String   @id @default(uuid())
  email        String   @unique
  phone        String   @unique
  passwordHash String
  fullName     String
  dateOfBirth  DateTime
  twoFactorEnabled  Boolean  @default(false)
  createdAt    DateTime @default(now())
  updatedAt    DateTime @updatedAt
  status       UserStatus
}

enum UserStatus {
  active
  blocked
  unverified
}



generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

model Card {
  id             String   @id @default(uuid())
  userId         String
  cardNumber     String   @unique
  cardType       CardType
  status         CardStatus
  expirationDate DateTime
  cvvHash        String
  issuedAt       DateTime @default(now())
  transactions   CardTransaction[]
}

enum CardType {
  debit
```

```
 credit
}

enum CardStatus {
  active
  blocked
  closed
}

model CardApplication {
  id          String   @id @default(uuid())
  userId      String
  cardType    CardType
  status      ApplicationStatus
  appliedAt   DateTime @default(now())
  processedAt DateTime?
}

enum ApplicationStatus {
  pending
  approved
  rejected
}

model CardTransaction {
  id              String   @id @default(uuid())
  cardId          String
  amount          Decimal
  currency        String
  transactionType TransactionType
  description     String
  createdAt       DateTime @default(now())
}

enum TransactionType {
  payment
  withdrawal
  deposit
}



generator client {
  provider = "prisma-client-js"
```

```
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

model Document {
  id             String   @id @default(uuid())
  userId         String
  documentType   DocumentType
  documentNumber String
  issuedDate     DateTime
  expiryDate     DateTime
  fileUrl        String
  status         DocumentStatus
  uploadedAt     DateTime @default(now())
}

enum DocumentType {
  passport
  id_card
  drivers_license
}

enum DocumentStatus {
  pending
  approved
  rejected
}

model VerificationStatus {
  id        String   @id @default(uuid())
  userId    String
  status    VerificationState
  updatedAt DateTime @default(now())
  comment   String?
}

enum VerificationState {
  pending
  verified
  rejected
```

```
}


import { Controller, Logger } from '@nestjs/common';
import { MessagePattern } from '@nestjs/microservices';
import { RedisService } from '../redis';

@Controller()
export class AuthEventsController {
  private readonly logger = new Logger(AuthEventsController.name);

  constructor(private readonly redisService: RedisService) {}

  @MessagePattern({ cmd: 'validate-session' })
  async validateSession(sessionId: string) {
    const session = await this.redisService.getSession(sessionId);
    if (!session) return { valid: false };
    return { valid: true, ...session };
  }
}


import { Controller } from '@nestjs/common';
import {
  MessagePattern,
  Payload,
} from '@nestjs/microservices';
import { UserService } from './user.service';
import { UserStatus } from '../../common/types/user-status';

@Controller()
export class UserEventsController {
  constructor(private readonly userService: UserService) {}

  @MessagePattern({ cmd: 'create-user' })
  async createUser(@Payload() data: any) {
    return this.userService.create(data);
  }

  @MessagePattern({ cmd: 'find-user' })
  async findUser(@Payload() data: { email?: string; phone?: string }) {
    if (data.email) {
      return this.userService.findByEmail(data.email);
```

```
}
    if (data.phone) {
      return this.userService.findByPhone(data.phone);
    }
    return null;
  }

  @MessagePattern({ cmd: 'update-user-password' })
  async updateUserPassword(
    @Payload() data: { userId: string; passwordHash: string },
  ) {
    return this.userService.updatePassword(data.userId, data.passwordHash);
  }

  @MessagePattern({ cmd: 'enable-2fa' })
  async enable2FA(@Payload() data: { userId: string }) {
    return this.userService.enable2FA(data.userId);
  }

  @MessagePattern({ cmd: 'disable-2fa' })
  async disable2FA(@Payload() data: { userId: string }) {
    return this.userService.disable2FA(data.userId);
  }

  @MessagePattern({ cmd: 'user_verified' })
  async updateUserStatus(@Payload() data: { userId: string; status: string }) {
    return this.userService.updateUserStatus(data.userId, data.status as UserStatus);
  }
}
```

```
import { CanActivate, ExecutionContext, Injectable, UnauthorizedException } from
'@nestjs/common';
import { RedisService } from '../../src/redis/redis.service';
import { Request } from 'express';

@Injectable()
export class SessionGuard implements CanActivate {
  constructor(private readonly redisService: RedisService) {}

  async canActivate(context: ExecutionContext): Promise<boolean> {
    const req = context.switchToHttp().getRequest<Request>();
    const sessionId = req.cookies?.sessionId || req.headers['x-session-id'];
```

```
 if (!sessionId) throw new UnauthorizedException('No session');
    const session = await this.redisService.getSession(sessionId);
    if (!session) throw new UnauthorizedException('Invalid session');
    req['user'] = session;
    return true;
  }
}
```

```
import { Injectable, OnModuleInit, OnModuleDestroy } from '@nestjs/common';
import Redis from 'ioredis';
import { ConfigService } from '@nestjs/config';

@Injectable()
export class RedisService implements OnModuleInit, OnModuleDestroy {
  private client: Redis;

  constructor(private readonly configService: ConfigService) {}

  onModuleInit() {
    this.client = new Redis(
      this.configService.get<string>('REDIS_URL'),
    );
    this.client.on('error', (err) => {
      console.error('Redis error:', err);
    });
  }

  onModuleDestroy() {
    this.client?.disconnect();
  }

  private sessionKey(sessionId: string) {
    return `session:${sessionId}`;
  }

  private userKey(userId: string) {
    return `user-session:${userId}`;
  }

  async setSession(
    sessionId: string,
    data: any,
```

```
  ttlSeconds = 300,
): Promise<boolean> {
  const pipeline = this.client.pipeline();
  pipeline.set(
    this.sessionKey(sessionId),
    JSON.stringify(data),
    'EX',
    ttlSeconds,
  );
  pipeline.sadd(this.userKey(data.userId), sessionId);
  pipeline.expire(this.userKey(data.userId), ttlSeconds);
  await pipeline.exec();
  return true;
}

async getSession(sessionId: string): Promise<any | null> {
  const data = await this.client.get(this.sessionKey(sessionId));
  return data ? JSON.parse(data) : null;
}

async getSessionsByUserId(userId: string): Promise<string[]> {
  return this.client.smembers(this.userKey(userId));
}

async deleteSession(sessionId: string) {
  const sessionData = await this.client.get(this.sessionKey(sessionId));
  const pipeline = this.client.pipeline();

  if (sessionData) {
    const { userId } = JSON.parse(sessionData);
    pipeline.srem(this.userKey(userId), sessionId);
  }

  pipeline.del(this.sessionKey(sessionId));
  await pipeline.exec();
}

async deleteAllUserSessions(userId: string) {
  const sessions = await this.getSessionsByUserId(userId);
  const pipeline = this.client.pipeline();

  for (const sessionId of sessions) {
    pipeline.del(this.sessionKey(sessionId));
```

```
}

    pipeline.del(this.userKey(userId));
    await pipeline.exec();
  }
}
```

```
version: '3.8'

services:
  rabbitmq:
    image: rabbitmq:3-management
    container_name: rabbitmq
    ports:
      - '5672:5672'    # AMQP
      - '15672:15672'  # Management UI
    environment:
      RABBITMQ_DEFAULT_USER: ${RABBITMQ_USER}
      RABBITMQ_DEFAULT_PASS: ${RABBITMQ_PASSWORD}
      RABBITMQ_LOAD_DEFINITIONS: /etc/rabbitmq/definitions.json
    volumes:
      - ./config/rabbitmq-definitions.json:/etc/rabbitmq/definitions.json
      - ./config/rabbitmq.conf:/etc/rabbitmq/rabbitmq.conf
    restart: unless-stopped
    networks:
      - rabbitmq-net
  postgres:
    image: postgres:15-alpine
    container_name: postgres
    environment:
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
      POSTGRES_DB: ${POSTGRES_DB}
    ports:
      - '5432:5432'
    restart: unless-stopped
    volumes:
      - pgdata:/var/lib/postgresql/data
    networks:
      - postgres-net
  pgadmin:
    image: dpage/pgadmin4
```

```
  container_name: pgadmin
  environment:
    PGADMIN_DEFAULT_EMAIL: ${PGADMIN_EMAIL}
    PGADMIN_DEFAULT_PASSWORD: ${PGADMIN_PASSWORD}
  ports:
    - '5050:80'
  depends_on:
    - postgres
  networks:
    - postgres-net
redis:
  image: redis:7-alpine
  container_name: redis
  ports:
    - '6379:6379'
  restart: unless-stopped
  volumes:
    - redisdata:/data
  networks:
    - redis-net
nginx-gateway:
  image: nginx:alpine
  container_name: nginx-gateway
  ports:
    - '80:80'
  volumes:
    - ./config/nginx.conf:/etc/nginx/nginx.conf:ro
  networks:
    - nginx-gateway
minio:
  image: minio/minio
  container_name: minio
  ports:
    - "9000:9000"
  environment:
    MINIO_ROOT_USER: minioadmin
    MINIO_ROOT_PASSWORD: minioadmin
  command: server /data
  volumes:
    - minio-data:/data
  networks:
    - minio-net
```

```
volumes:
  pgdata:
  redisdata:
  minio-data:

networks:
  postgres-net:
    external: true
  rabbitmq-net:
    external: true
  redis-net:
    external: true
  nginx-gateway:
    external: true
  minio-net:
    external: true
```

```
import {
  Controller,
  Post,
  UseInterceptors,
  UploadedFile,
  Body,
  BadRequestException,
  UseGuards,
  Req,
  Get
} from '@nestjs/common';
import { FileInterceptor } from '@nestjs/platform-express';
import { VerificationService } from './verification.service';
import { File } from 'multer';
import { AdminVerifyDto } from './dto';
import { SessionGuard, AdminRoleGuard } from '../../common/guards';

@Controller('verification')
export class VerificationController {
  constructor(private readonly verificationService: VerificationService) {}

  @Post()
  @UseInterceptors(FileInterceptor('file'))
  @UseGuards(SessionGuard)
  async verify(
```

```
    @UploadedFile() file: File,
    @Req() req,
    @Body('documentType') documentType: string,
  ) {
    const userId = req.user.userId;
    if (!file) {
      throw new BadRequestException('File is required');
    }
    if (!userId || !documentType) {
      throw new BadRequestException('userId and documentType are required');
    }
    if (documentType !== 'id_card' && documentType !== 'drivers_license') {
      throw new BadRequestException('Invalid documentType');
    }
    return await this.verificationService.verifyUser(
      userId,
      file,
      documentType,
    );
  }

  @Post('admin')
  @UseGuards(SessionGuard, AdminRoleGuard)
  async adminVerify(@Body() verifyDto: AdminVerifyDto) {
    if (!verifyDto.documentId) {
      throw new BadRequestException('documentId is required');
    }

    return await this.verificationService.adminVerify(verifyDto);
  }

  @Get('/check-verification')
  @UseGuards(SessionGuard)
  async getVerification(@Req() req) {
    const userId = req.user.userId;
    return await this.verificationService.getVerification(userId);
  }
}



import {
  Injectable,
  InternalServerErrorException
```

```
  Inject,
  ForbiddenException
} from '@nestjs/common';
import { S3Service } from '../s3';
import { File } from 'multer';
import { PrismaService } from '../prisma';
import { ClientProxy } from '@nestjs/microservices';
import { AdminVerifyDto } from './dto';
import { firstValueFrom, timeout } from 'rxjs';

@Injectable()
export class VerificationService {
  constructor(
    private readonly s3Service: S3Service,
    private readonly prisma: PrismaService,
    @Inject('USER_SERVICE') private readonly userClient: ClientProxy,
  ) {}

  async verifyUser(
    userId: string,
    file: File,
    documentType: string,
  ): Promise<any> {
    try {
      const existing = await this.prisma.document.findFirst({
        where: {
          userId,
          status: { in: ['pending', 'approved'] },
        },
      });
      if (existing) {
        throw new InternalServerErrorException('Verification already exists');
      }
      const fileUrl = await this.s3Service.uploadFile(file);

      const document = await this.prisma.document.create({
        data: {
          userId,
          documentType: documentType as any,
          documentNumber: '',
          issuedDate: new Date(),
          expiryDate: new Date(),
          fileUrl
```

```
        status: 'pending',
      },
    });

    return { userId, document };
  } catch (error: any) {
    throw new InternalServerErrorException(error.message);
  }
}

async adminVerify(verifyDto: AdminVerifyDto) {
  const document = await this.prisma.document.findUnique({
    where: { id: verifyDto.documentId },
  });
  if (!document) throw new Error('Document not found');

  const updatedDocument = await this.prisma.document.update({
    where: { id: verifyDto.documentId },
    data: {
      status: verifyDto.action === 'approve' ? 'approved' : 'rejected',
    },
  });

  if (verifyDto.action === 'approve') {
    await firstValueFrom(
      this.userClient.send(
        { cmd: 'change-user-status' },
        { userId: document.userId, status: 'active' }
      ).pipe(timeout(3000))
    );
  }

  return {
    userId: document.userId,
    documentId: verifyDto.documentId,
    document: updatedDocument,
  };
}

async getVerification(userId: string) {
  const verification = await this.prisma.document.findFirst({ where: { userId } });
  if (!verification) return { status: 'not_verified' };
  if (verification.userId !== userId) {
```

```
      throw new ForbiddenException('Access denied: not your verification');
    }
    return { id: verification.id, status: verification.status };
  }
}


import { Controller, Logger } from '@nestjs/common';
import { MessagePattern } from '@nestjs/microservices';
import { VerificationService } from './verification.service';

@Controller()
export class VerificationEventsController {
  constructor(private readonly verificationService: VerificationService) {}

  @MessagePattern({ cmd: 'get-verification-status' })
  async getVerificationStatus(userId: string) {
    return this.verificationService.getVerification(userId);
  }
}


import {
  Controller,
  Post,
  Get,
  Body,
  Param,
  UseGuards,
  Req,
} from '@nestjs/common';
import { CardService } from './card.service';
import { OpenCardDto } from './dto';
import {
  SessionGuard,
  CardOwnerGuard,
  AdminRoleGuard,
  VerificationGuard,
} from '../../common/guards';

@Controller('cards')
export class CardController {
  constructor(private readonly cardService: CardService) {}
```

```
@Post('open')
@UseGuards(SessionGuard, VerificationGuard)
async openCard(@Body() dto: OpenCardDto, @Req() req) {
  const userId = req.user.userId;
  return this.cardService.createCardApplication(userId, dto);
}

@Post(':id/block')
@UseGuards(SessionGuard, CardOwnerGuard, VerificationGuard)
async blockCard(@Param('id') cardId: string) {
  return this.cardService.blockCard(cardId);
}

@Post(':id/unblock')
@UseGuards(SessionGuard, CardOwnerGuard, VerificationGuard)
async unblockCard(@Param('id') cardId: string) {
  return this.cardService.unblockCard(cardId);
}

@Post(':id/close')
@UseGuards(SessionGuard, CardOwnerGuard, VerificationGuard)
async closeCard(@Param('id') cardId: string) {
  return this.cardService.closeCard(cardId);
}

@Get()
@UseGuards(SessionGuard, VerificationGuard)
async getAllByUser(@Req() req) {
  return this.cardService.findAllByUser(req.user.userId);
}

@Get('applications')
@UseGuards(SessionGuard, AdminRoleGuard, VerificationGuard)
async getApplications() {
  return this.cardService.getAllApplications();
}

@Get(':id')
@UseGuards(SessionGuard, CardOwnerGuard, VerificationGuard)
async getOne(@Param('id') cardId: string) {
  return this.cardService.findOne(cardId);
}
```

```
  @Post('applications/:id/approve')
  @UseGuards(SessionGuard, AdminRoleGuard, VerificationGuard)
  async approveApplication(@Param('id') id: string) {
    return this.cardService.approveApplication(id);
  }
}


import { Injectable, Inject } from '@nestjs/common';
import { PrismaService } from '../prisma';
import { OpenCardDto } from './dto';
import {
  generateCardNumber,
  generateCvv,
  generateExpirationDate,
} from '../utils';
import * as bcrypt from 'bcryptjs';
import { ClientProxy } from '@nestjs/microservices';

@Injectable()
export class CardService {
  constructor(private readonly prisma: PrismaService, @Inject('HISTORY_SERVICE') private
readonly historyClient: ClientProxy) {}

  private async openCard(userId: string, dto: OpenCardDto) {
    const cardNumber = generateCardNumber(dto.provider);
    const cvv = generateCvv();
    const cvvHash = await bcrypt.hash(cvv, 10);
    const expirationDate = generateExpirationDate();

    const card = await this.prisma.card.create({
      data: {
        userId: userId,
        cardNumber,
        cardType: dto.cardType,
        provider: dto.provider,
        status: 'active',
        expirationDate,
        cvvHash,
      },
    });
```

```
    await this.historyClient.send({ cmd: 'history.log' }, { userId, eventType: 'ADMIN_ACTION',
meta: { action: 'card.open', cardId: card.id, provider: dto.provider } }).toPromise();
    return { ...card, cvv };
  }

  async blockCard(cardId: string) {
    const card = await this.prisma.card.update({
      where: { id: cardId },
      data: { status: 'blocked' },
    });
    await this.historyClient.send({ cmd: 'history.log' }, { userId: card.userId, eventType:
'CARD_BLOCK', meta: { cardId } }).toPromise();
    return card;
  }

  async closeCard(cardId: string) {
    const card = await this.prisma.card.findUnique({ where: { id: cardId } });
    if (!card) throw new Error('Card not found');
    if (Number(card.balance) !== 0)
      throw new Error('Cannot close card with non-zero balance');
    const closed = await this.prisma.card.update({
      where: { id: cardId },
      data: { status: 'closed' },
    });
    await this.historyClient.send({ cmd: 'history.log' }, { userId: card.userId, eventType:
'CARD_CLOSE', meta: { cardId } }).toPromise();
    return closed;
  }

  async unblockCard(cardId: string) {
    const card = await this.prisma.card.update({
      where: { id: cardId },
      data: { status: 'active' },
    });
    await this.historyClient.send({ cmd: 'history.log' }, { userId: card.userId, eventType:
'CARD_UNBLOCK', meta: { cardId } }).toPromise();
    return card;
  }

  async findAllByUser(userId: string) {
    return this.prisma.card.findMany({ where: { userId } });
  }
```

```
  async findOne(cardId: string) {
    return this.prisma.card.findUnique({ where: { id: cardId } });
  }

  async createCardApplication(userId: string, dto: OpenCardDto) {
    return this.prisma.cardApplication.create({
      data: {
        userId,
        cardType: dto.cardType,
        provider: dto.provider,
        status: 'pending',
      },
    });
  }

  async getAllApplications() {
    return this.prisma.cardApplication.findMany();
  }

  async approveApplication(id: string) {
    const application = await this.prisma.cardApplication.findUnique({
      where: { id },
    });
    if (!application) throw new Error('Application not found');

    if (application.status === 'approved') {
      return { message: 'Application already approved' };
    }

    const updated = await this.prisma.cardApplication.update({
      where: { id },
      data: { status: 'approved', processedAt: new Date() },
    });

    const card = await this.openCard(updated.userId, {
      cardType: updated.cardType,
      provider: updated.provider,
    });
    await this.historyClient.send({ cmd: 'history.log' }, { userId: updated.userId, eventType:
'ADMIN_ACTION',   meta:   {   action:   'card.application.approve',   applicationId:   id   }
}).toPromise();
    return card;
  }
```

```
  async transfer(senderCardId: string, receiverCardId: string, amount: number, currency: string) {
    return this.prisma.$transaction(async (tx) => {
      const senderArr = await tx.$queryRawUnsafe(
        `SELECT "id", "userId", "balance", "currency" FROM "Card" WHERE id = $1 FOR UPDATE`,
        senderCardId
      ) as any[];

      const sender = senderArr[0];

      if (!sender) throw new Error('Card not found');
      if (Number(sender.balance) < amount) throw new Error('Insufficient funds');

      const receiverArr = await tx.$queryRawUnsafe(
        `SELECT "id", "userId", "balance", "currency" FROM "Card" WHERE id = $1 FOR UPDATE`,
        receiverCardId
      ) as any[];

      const receiver = receiverArr[0];

      if (!receiver) throw new Error('Card not found');
      if (sender.currency !== currency || receiver.currency !== currency) {
        throw new Error('Currency mismatch');
      }

      await tx.card.update({
        where: { id: senderCardId },
        data: { balance: { decrement: amount } },
      });

      await tx.card.update({
        where: { id: receiverCardId },
        data: { balance: { increment: amount } },
      });

      await this.historyClient.send({ cmd: 'history.log' }, { userId: sender.userId, eventType: 'TRANSFER', meta: { direction: 'out', senderCardId, receiverCardId, amount, currency } }).toPromise();
      await this.historyClient.send({ cmd: 'history.log' }, { userId: receiver.userId, eventType: 'TRANSFER', meta: { direction: 'in', senderCardId, receiverCardId, amount, currency } }).toPromise();
    });
  }
```

```
  async getCardByNumber(cardNumber: string) {
    return this.prisma.card.findUnique({ where: { cardNumber } });
  }
}


import { Controller, UseGuards } from '@nestjs/common';
import { MessagePattern } from '@nestjs/microservices';
import { CardService } from './card.service';

@Controller()
export class CardEventsController {
  constructor(private readonly cardService: CardService) {}

  @MessagePattern({ cmd: 'transfer' })
  async transfer({ senderCardId, receiverCardId, amount, currency }: { senderCardId: string;
receiverCardId: string; amount: number; currency: string }) {
    try {
      await this.cardService.transfer(senderCardId, receiverCardId, amount, currency);
      return { success: true };
    } catch (e) {
      console.error(e);
      return { success: false, error: e.message };
    }
  }

  @MessagePattern({ cmd: 'get-card-by-number' })
  async getCardByNumber(cardNumber: string) {
    return this.cardService.getCardByNumber(cardNumber);
  }

  @MessagePattern({ cmd: 'get-cards-by-user' })
  async getCardsByUser(userId: string) {
    return this.cardService.findAllByUser(userId);
  }
}


import { Injectable, InternalServerErrorException } from '@nestjs/common';
import * as Minio from 'minio';

interface MulterFile {
  fieldname: string;
```

```
  originalname: string;
  encoding: string;
  mimetype: string;
  buffer: Buffer;
  size: number;
}

@Injectable()
export class S3Service {
  private minioClient: Minio.Client;
  private bucketName: string = process.env.MINIO_BUCKET || 'uploads';

  constructor() {
    this.minioClient = new Minio.Client({
      endPoint: process.env.MINIO_ENDPOINT || 'minio',
      port: parseInt(process.env.MINIO_PORT || '9000', 10),
      useSSL: process.env.MINIO_USE_SSL === 'false',
      accessKey: process.env.MINIO_ACCESS_KEY || 'minioadmin',
      secretKey: process.env.MINIO_SECRET_KEY || 'minioadmin',
    });

    this.initializeBucket();
  }

  public async initializeBucket(): Promise<void> {
    try {
      const exists: boolean = await new Promise((resolve, reject) => {
        (this.minioClient.bucketExists as any)(
          this.bucketName,
          (err: Error, exists: boolean) => {
            if (err) return reject(err);
            resolve(exists);
          },
        );
      });
      if (!exists) {
        await new Promise<void>((resolve, reject) => {
          (this.minioClient.makeBucket as any)(
            this.bucketName,
            'us-east-1',
            (err: Error) => {
              if (err) return reject(err);
              resolve();
```

```
          },
        );
      });
      console.log(`Bucket ${this.bucketName} created successfully`);
    }
  } catch (err) {
    console.error('Error creating bucket:', err);
  }
}

async uploadFile(file: MulterFile): Promise<string> {
  const filename = `${Date.now()}-${file.originalname}`;
  try {
    await new Promise<void>((resolve, reject) => {
      (this.minioClient.putObject as any)(
        this.bucketName,
        filename,
        file.buffer,
        file.buffer.length,
        (err: Error, etag: string) => {
          if (err) return reject(err);
          resolve();
        },
      );
    });
    const protocol = process.env.MINIO_USE_SSL === 'true' ? 'https' : 'http';
    const endpoint = process.env.MINIO_ENDPOINT || 'localhost';
    const port = process.env.MINIO_PORT || '9000';
    return `${protocol}://${endpoint}:${port}/${this.bucketName}/${filename}`;
  } catch (err) {
    throw new InternalServerErrorException('Cannot upload file to Minio');
  }
}
}


export function generateCardNumber(provider: 'visa' | 'mastercard'): string {
  // BIN/IIN
  let bin = '';
  if (provider === 'visa') {
    bin = '400000'; // або інший реальний BIN банку
  } else if (provider === 'mastercard') {
    bin = Math.random() < 0.5 ? '510000' : '220000'; // 5***** або 2*****
```

```
  }

  // 9 випадкових цифр
  const accountIdentifier = Array.from({ length: 9 }, () =>
    Math.floor(Math.random() * 10),
  ).join('');

  // Перші 15 цифр (без контрольної)
  const partial = bin + accountIdentifier;

  // Контрольна цифра (Luhn)
  const checkDigit = getLuhnCheckDigit(partial);

  return partial + checkDigit;
}

function getLuhnCheckDigit(number: string): string {
  let sum = 0;
  let shouldDouble = true;
  for (let i = number.length - 1; i >= 0; i--) {
    let digit = parseInt(number[i], 10);
    if (shouldDouble) {
      digit *= 2;
      if (digit > 9) digit -= 9;
    }
    sum += digit;
    shouldDouble = !shouldDouble;
  }
  const checkDigit = (10 - (sum % 10)) % 10;
  return checkDigit.toString();
}



import {
    Controller,
    Post,
    Get,
    Body,
    Param,
    UseGuards,
    Req,
  } from '@nestjs/common';
  import { PaymentService } from './payment.service';
```

```
  import {
    SessionGuard,
    VerificationGuard,
  } from '../../common/guards';
  import { TransferDto } from './dto';

  @Controller('payment')
  export class PaymentController {
    constructor(private readonly paymentService: PaymentService) {}

    @Post('transfer')
    @UseGuards(SessionGuard, VerificationGuard)
    async transfer(@Body() dto: TransferDto, @Req() req) {
        return this.paymentService.transfer(dto, req.user.userId);
    }

    @Get('history')
    @UseGuards(SessionGuard, VerificationGuard)
    async getHistory(@Req() req) {
        return this.paymentService.getHistory(req.user.userId);
    }

    @Get(':id')
    @UseGuards(SessionGuard, VerificationGuard)
    async getById(@Param('id') id: string, @Req() req) {
        return this.paymentService.getById(id, req.user.userId);
    }
}


import { Inject, Injectable, BadRequestException, ForbiddenException } from '@nestjs/common';
import { PrismaService } from '../prisma';
import { ClientProxy } from '@nestjs/microservices';
import { firstValueFrom } from 'rxjs';
import { TransferDto } from './dto';

@Injectable()
export class PaymentService {
  constructor(
    private readonly prisma: PrismaService,
    @Inject('CARD_SERVICE') private readonly cardClient: ClientProxy,
    @Inject('HISTORY_SERVICE') private readonly historyClient: ClientProxy,
  ) {}
```

```
  async transfer(dto: TransferDto, userId: string) {
    const senderCard = await firstValueFrom(
      this.cardClient.send({ cmd: 'get-card-by-number' }, dto.senderCardNumber)
    );

    const receiverCard = await firstValueFrom(
      this.cardClient.send({ cmd: 'get-card-by-number' }, dto.receiverCardNumber)
    );

    if (!senderCard || !receiverCard) {
      throw new BadRequestException('Card not found');
    }

    if (senderCard.userId !== userId) {
      throw new ForbiddenException('Not your card');
    }

    let transferResult;
    try {
      transferResult = await firstValueFrom(
        this.cardClient.send({ cmd: 'transfer' }, {
          senderCardId: senderCard.id,
          receiverCardId: receiverCard.id,
          amount: dto.amount,
          currency: dto.currency,
        }),
      );
    } catch (e) {
      await this.historyClient.send({ cmd: 'history.log' }, { userId, eventType:
'ADMIN_ACTION', meta: { action: 'payment.transfer.failed', error: e.message, dto }
}).toPromise();
      throw e;
    }
    const transfer = await this.prisma.transfer.create({
      data: {
        senderCardId: senderCard.id,
        receiverCardId: receiverCard.id,
        amount: dto.amount,
        currency: dto.currency,
        status: transferResult.success ? 'completed' : 'failed',
        comment: transferResult.success ? null : (transferResult.error || 'Unknown error'),
        completedAt: transferResult.success ? new Date() : null,
      },
```

```
    });
    await this.historyClient.send({ cmd: 'history.log' }, { userId, eventType: 'TRANSFER',
meta: { transferId: transfer.id, status: transfer.status } }).toPromise();
    return transfer;
  }

  async getHistory(userId: string) {
    const cards = await firstValueFrom(
      this.cardClient.send({ cmd: 'get-cards-by-user' }, userId)
    );
    if (!Array.isArray(cards) || cards.length === 0) {
      return [];
    }
    const cardIds = cards.map(card => card.id);

    const transfers = await this.prisma.transfer.findMany({
      where: {
        OR: [
          { senderCardId: { in: cardIds } },
          { receiverCardId: { in: cardIds } },
        ],
      },
      orderBy: { createdAt: 'desc' },
    });

    await this.historyClient.send({ cmd: 'history.log' }, { userId, eventType: 'ADMIN_ACTION',
meta: { action: 'payment.history' } }).toPromise();

    return transfers
      .map(tr => {
        let type: 'incoming' | 'outgoing';
        let cardId: string;
        if (cardIds.includes(tr.senderCardId)) {
          type = 'outgoing';
          cardId = tr.senderCardId;
        } else {
          type = 'incoming';
          cardId = tr.receiverCardId;
        }
        return {
          ...tr,
          type,
          cardId,
```

```
        };
      })
      .filter(tr =>
        (tr.type === 'outgoing' && cardIds.includes(tr.senderCardId)) ||
        (tr.type === 'incoming' && tr.status === 'completed' &&
cardIds.includes(tr.receiverCardId))
      );
  }

  async getById(id: string, userId: string) {
    const cards = await firstValueFrom(
      this.cardClient.send({ cmd: 'get-cards-by-user' }, userId)
    );
    const cardIds = cards.map(card => card.id);
    const transfer = await this.prisma.transfer.findUnique({ where: { id } });
    if (!transfer) throw new BadRequestException('Transfer not found');

    let type: 'incoming' | 'outgoing' | null = null;
    let cardId: string | null = null;

    if (cardIds.includes(transfer.senderCardId)) {
      type = 'outgoing';
      cardId = transfer.senderCardId;
    } else if (cardIds.includes(transfer.receiverCardId)) {
      type = 'incoming';
      cardId = transfer.receiverCardId;
    } else {
      throw new ForbiddenException('Access denied');
    }

    await this.historyClient.send({ cmd: 'history.log' }, { userId, eventType: 'ADMIN_ACTION',
meta: { action: 'payment.getById', transferId: id } }).toPromise();

    return {
      ...transfer,
      type,
      cardId,
    };
  }
}

import { Controller, Get, Param, UseGuards } from '@nestjs/common';
import { HistoryService } from './history.service';
```

```
import { SessionGuard, VerificationGuard, AdminRoleGuard } from '../../common/guards';

@Controller('history')
export class HistoryController {
  constructor(private readonly historyService: HistoryService) {}

  @UseGuards(SessionGuard, VerificationGuard, AdminRoleGuard)
  @Get('user/:userId')
  async getUserEvents(@Param('userId') userId: string) {
    return this.historyService.getUserEvents(userId);
  }

  @UseGuards(SessionGuard, VerificationGuard, AdminRoleGuard)
  @Get('all')
  async getAllEvents() {
    return this.historyService.getAllEvents();
  }
}


import { Injectable } from '@nestjs/common';
import { PrismaService } from '../prisma/prisma.service';
import { LogEventDto } from './dto';

@Injectable()
export class HistoryService {
  constructor(private readonly prisma: PrismaService) {}

  async logEvent(dto: LogEventDto) {
    return this.prisma.event.create({
      data: {
        userId: dto.userId,
        eventType: dto.eventType,
        meta: dto.meta,
      },
    });
  }

  async getUserEvents(userId: string) {
    return this.prisma.event.findMany({
      where: { userId },
      orderBy: { timestamp: 'desc' },
```

```
    });
  }

  async getAllEvents() {
    return this.prisma.event.findMany({
      orderBy: { timestamp: 'desc' },
    });
  }
}

import { Controller } from '@nestjs/common';
import { MessagePattern, Payload } from '@nestjs/microservices';
import { HistoryService } from './history.service';
import { LogEventDto } from './dto';

@Controller()
export class HistoryEventsController {
  constructor(private readonly historyService: HistoryService) {}

  @MessagePattern({ cmd: 'history.log' })
  async logEvent(@Payload() dto: LogEventDto) {
    return this.historyService.logEvent(dto);
  }
}
```