МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

на тему: **«РОЗРОБКА СИСТЕМИ ДИСТАНЦІЙНОГО БАНКІВСЬКОГО ОБСЛУГОВУВАННЯ ЗАСОБАМИ NESTJS ТА ANGULAR»**

Виконав: студент 2 курсу, групи 8.1214-іпз-1
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми інженерія програмного забезпечення
(назва освітньої програми)
А.М. Данілов
(ініціали та прізвище)

Керівник доцент кафедри фундаментальної та прикладної математики, доцент, к.ф.-м.н. Панасенко Є.В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Рецензент доцент кафедри комп'ютерних наук, доцент, к.т.н. Матвіїшина Н.В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти магістр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2025 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Данілову Антону Миколайовичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка системи дистанційного банківського обслуговування засобами NestJS та Angular

керівник роботи Панасенко Євген Валерійович, к.ф.-м.н., доцент
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 28 » лютого 2025 року № 326-с

2. Строк подання студентом роботи 03.12.2025 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Теоретичний основи системи дистанційного банківського обслуговування.
2. Проєктування та розробка вебзастосунку.
3. Програмування та розробка системи.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація за темою доповіді

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 07.03.2025 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	28.03.2025	
2.	Збір вихідних даних.	25.04.2025	
3.	Обробка методичних та теоретичних	13.06.2025	
	джерел.		
4.	Розробка першого та другого розділу.	29.08.2025	
5.	Розробка третього розділу.	24.11.2025	
6.	Оформлення та нормоконтроль	01.12.2025	
	кваліфікаційної роботи магістра.		
7.	Захист кваліфікаційної роботи.	17.12.2025	

Студент _____ А.М. Данілов
(підпис) (ініціали та прізвище)

Керівник роботи _____ Є.В. Панасенко
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота магістра «Розробка системи дистанційного банківського обслуговування засобами NestJS та Angular»: 71 с., 11 рис., 10 джерел.

АВТЕНТИФІКАЦІЯ, API-ІНТЕГРАЦІЯ, БАЗА ДАНИХ, БЕЗПЕКА, КЛІЄНТСЬКИЙ КАБІНЕТ, СЕРВЕРНА ЧАСТИНА, СИСТЕМА ДІСТАНЦІЙНОГО БАНКІВСЬКОГО ОБСЛУГОВУВАННЯ, ТРАНЗАКЦІЇ, ФІНТЕХ, ANGULAR, NESTJS, NODE.JS, POSTGRESQL, WEB API.

Об’єкт дослідження – процес проєктування та реалізації повноцінної вебсистеми дистанційного банківського обслуговування (ДБО), що автоматизує надання фінансових послуг клієнтам через інтернет.

Мета роботи: розробити безпечну, масштабовану та зручну в обслуговуванні систему ДБО, що інтегрує сучасний клієнтський інтерфейс на Angular та високопродуктивну серверну частину на NestJS.

Методи дослідження: методи проєктування модульної архітектури (NestJS, DDD), моделювання даних (PostgreSQL), розробки API (RESTful), застосування сучасних механізмів безпеки (JWT, 2FA, RBAC), інтеграції з платіжними системами та комплексного тестування.

Результати роботи: розроблено повноцінну вебсистему ДБО, що складається з клієнтського додатку на Angular та серверної платформи на NestJS. Система реалізує ключові модулі: управління користувачами та безпекою, роботу з рахунками, здійснення платежів та переказів, адміністрування. Для зберігання даних створено оптимізовану схему в PostgreSQL. Забезпечено високий рівень безпеки та продуктивності: середній час відповіді на критичні операції не перевищує 300 мс. Система відрізняється чіткою модульною архітектурою, сучасним інтерфейсом та може бути впроваджена як самостійна платформа або інтегрована в існуючу банківську інфраструктуру.

SUMMARY

Master’s qualifying paper “Development of a Remote Banking Service System Using NestJS and Angular”: 71 pages, 11 figures, 10 references.

ANGULAR, API INTEGRATION, AUTHENTICATION, BACKEND, CLIENT PORTAL, DATABASE, FINTECH, NESTJS, NODE.JS, POSTGRESQL, REMOTE BANKING SERVICE SYSTEM, SECURITY, TRANSACTIONS, WEB API.

The object of the study is the process of designing and implementing a full-fledged web-based remote banking service (RBS) system that automates the provision of financial services to customers over the Internet.

The aim of the study is to develop a secure, scalable, and maintainable RBS system that integrates a modern client interface on Angular and a high-performance backend on NestJS.

The methods of research include methods of designing modular architecture (NestJS, DDD), data modeling (PostgreSQL), API development (RESTful), application of modern security mechanisms (JWT, 2FA, RBAC), integration with payment systems, and comprehensive testing.

Results of the work: a full-fledged web-based RBS system was developed, consisting of a client application on Angular and a server platform on NestJS. The system implements key modules: user and security management, account operations, execution of payments and transfers, and administration. An optimized data schema was created in PostgreSQL for data storage. A high level of security and performance was ensured: the average response time for critical operations does not exceed 300 ms. The system is distinguished by its clear modular architecture, modern interface, and can be implemented as a standalone platform or integrated into existing banking infrastructure.

ЗМІСТ

Завдання на кваліфікаційну роботу……...2
Реферат 4
Summary 5
Вступ..... 7
1 Теоретичні основи системи дистанційного банківського обслуговування . 10
1.1 Сучасний стан та тренди ринку банківських послуг 10
1.2 Архітектурні підходи до розробки фінтех-систем 13
1.3 Огляд технологічного стеку для сучасних вебзастосунків..... 16
1.4 Принципи безпеки та захисту даних у банківських вебсистемах..... 18
1.5 Аналіз існуючих рішень та API провідних банків 21
1.6 Висновки до розділу 1 23
2 Проєктування та розробка вебзастосунку 24
2.1 Вибір архітектурного стилю 24
2.2 Проєктування компонентів серверної частини..... 28
2.3 Проєктування схеми бази даних..... 35
2.4 Проєктування компонентів клієнтської частини..... 41
2.5 Висновки до розділу 2 49
3 Програмування та реалізація системи..... 50
3.1 Розробка серверної частини на NestJS..... 50
3.2 Розробка клієнтської частини на Angular..... 58
3.3 Інтеграція компонентів та тестування системи 64
Висновки 69
Перелік посилань..... 71

ВСТУП

Сучасний розвиток цифрових технологій та фінтеху призвів до стрімкого зростання попиту на зручні, безпечні та високопродуктивні системи дистанційного банківського обслуговування (ДБО). Ці системи стають ключовим каналом взаємодії банків з клієнтами, забезпечуючи доступ до рахунків, здійснення транзакцій та управління фінансами онлайн. Прикладом таких рішень є платформи, що використовують сучасні вебтехнології, такі як системи на основі мікросервісної архітектури з інтуїтивними клієнтськими інтерфейсами.

Метою кваліфікаційної роботи є проєктування та реалізація безпечної, масштабованої та зручної у використанні вебсистеми дистанційного банківського обслуговування з використанням NestJS для побудови серверної частини та Angular для розробки клієнтського інтерфейсу, з інтеграцією PostgreSQL для надійного зберігання даних та реалізацією механізмів фінансової безпеки.

Для досягнення поставленої мети сформульовано такі задачі:

– проаналізувати сучасні підходи, архітектурні патерни та вимоги безпеки при розробці фінтех-систем, зокрема систем ДБО;
– сформулювати функціональні та нефункціональні вимоги (безпека, продуктивність, масштабованість) до вебзастосунку;
– спроєктувати модульну архітектуру системи з чітким розмежуванням відповідальності між компонентами;
– реалізувати основний REST API та бізнес-логіку (управління рахунками, транзакції, платежі) на Node.js з використанням фреймворку NestJS;
– розробити інтуїтивний, адаптивний інтерфейс користувача з використанням Angular та TypeScript;
– реалізувати систему безпеки, що включає автентифікацію (JWT),

авторизацію (RBAC), двофакторну аутентифікацію (2FA) та шифрування конфіденційних даних;

– організувати надійне зберігання, цілісність та відновлюваність фінансових даних в СУБД PostgreSQL;
– розробити адміністративну панель для управління користувачами, моніторингу операцій та модерації;
– виконати комплексне тестування системи, включаючи модульне, інтеграційне та навантажувальне тестування.

Структурно робота складається з трьох розділів. У першому розділі проаналізовано сучасний стан ринку дистанційних банківських послуг, розглянуто архітектурні підходи (моноліт, мікросервіси) та технологічні стеки для розробки фінтех-систем. Обґрунтовано вибір основних інструментів: NestJS – для створення структурованого, тестового та ефективного серверного додатку; Angular – для побудови динамічного та безпечного клієнтського інтерфейсу; PostgreSQL – як транзакційної та надійної системи зберігання даних. Також досліджено ключові вимоги безпеки та нормативні аспекти.

У другому розділі виконано аналіз та формалізацію функціональних (керування профілем, переказ коштів, перегляд виписок) та нефункціональних (час відповіді, обробка одночасних запитів, захист від атак) вимог до системи. На основі цих вимог спроєктовано загальну архітектуру системи з використанням UML-діаграм (варіантів використання, послідовності, компонентів), детально розроблено схеми взаємодії між модулями (клієнтський додаток, API-шар, сервіси, база даних) та спроєктовано логічну та фізичну структуру бази даних в PostgreSQL.

У третьому розділі реалізовано основні компоненти системи: серверну частину (RESTful API, модулі безпеки, бізнес-логіку) на NestJS; клієнтський вебзастосунок (SPA) з компонентами, сервісами та маршрутизацією на Angular; інтегровано та налаштовано базу даних PostgreSQL. Проведено комплексне тестування: модульне – окремих сервісів та контролерів; інтеграційне – перевірка спільної роботи компонентів; навантажувальне – для

оцінки продуктивності та стабільності під високим навантаженням; а також перевірку реалізованих механізмів безпеки.

1 ТЕОРЕТИЧНІ ОСНОВИ СИСТЕМИ ДИСТАНЦІЙНОГО БАНКІВСЬКОГО ОБСЛУГОВУВАННЯ

1.1 Сучасний стан та тренди ринку банківських послуг

Сучасний ринок дистанційних банківських обслуговування (ДБО) переживає період глибокої трансформації, спричиненої конвергенцією фінансових технологій, змінами в очікуваннях клієнтів і жорсткою конкуренцією з боку фінтех-стартапів та необанків. Дистанційні сервіси перестали бути лише додатковим каналом та перетворилися на основний спосіб взаємодії для переважної більшості клієнтів, особливо в сегментах роздрібного банкінгу та малого бізнесу [1].

Розглянемо, що включають ключові тренди, які формують сучасний стан ринку.

Безперервна консолідація функціоналу. Сучасна платформа ДБО – це універсальний фінансовий хаб, який об’єднує традиційні операції зі списком послуг, що постійно розширюється. Він охоплює не лише переказ коштів та перегляд балансу, а й інвестиційні продукти, страхування, бухгалтерські сервіси для бізнесу та навіть не фінансові пропозиції в рамках екосистемних моделей.

Абсолютний пріоритет мобільності та досвіду користувача (UX). Концепція «mobile-first» перетворилася на «mobile-only» для значної частини аудиторії. Вимоги до інтерфейсу різко зросли: клієнти очікують інтуїтивної навігації, персональних пропозицій, миттєвих операцій та бездоганної роботи на будь-якому пристрої.

Впровадження принципів Open Banking. Регуляторні ініціативи, такі як PSD2 в Європі, стали каталізатором переходу від замкнутих банківських систем до відкритої API-економіки. Це дозволило клієнтам безпечно надавати стороннім провайдерам доступ до своїх фінансових даних для агрегації

рахунків (AIS) або ініціювання платежів (PIS), стимулюючи інновації та конкуренцію.

Глибока інтеграція штучного інтелекту та аналітики. AI/ML технології переходять від пілотних проєктів до ядра бізнес-процесів. Вони використовуються для проактивного виявлення шахрайства в реальному часі, персоналізації маркетингу, розумного кредитного скорингу, створення чат-ботів з натуральним мовленням та автоматизації обслуговування.

Непохитний акцент на кібербезпеці. Зростання онлайн-операцій призводить до пропорційного зростання кіберзагроз. У відповідь, сучасні системи ДБО посилюють захист за допомогою багатофакторної та біометричної аутентифікації, поведінкового аналізу для виявлення аномалій, продвинутих методів шифрування та технологій безпеки на рівні апаратного забезпечення (наприклад, Secure Enclave).

Розвиток в напрямку екосистем та супердодатків. Провідні гравці прагнуть утримати клієнта, перетворюючи свій банківський додаток на універсальну платформу для життя (lifestyle platform). Це означає інтеграцію послуг від партнерів: від замовлення таксі та бронювання столиків до управління лояльністю та електронною комерцією.

Таким чином, сучасний стан ринку ДБО характеризується поступовим стиранням меж між традиційним банкінгом та цифровими сервісами. Успіх залежить від здатності установи надавати не просто набір функцій, а безпечний, швидкий, персоналізований та всеохоплюючий цифровий досвід, побудований на основі масштабованої, відкритої та інтелектуальної технологічної платформи. Окрім основних напрямів, на ринку дистанційного банкінгу формується низка додаткових тенденцій, що визначають його майбутній розвиток [1].

Схід банківських послуг до хмарних платформ стає ключовою технологічною парадигмою. Традиційна інфраструктура на власних дата-центрах поступається місцем гібридним та публічним хмарним рішенням, таким як AWS, Google Cloud або Microsoft Azure. Цей перехід не лише

зменшує капітальні витрати, але й надає банкам безпрецедентну гнучкість та масштабованість. Він стає фундаментом для швидкого впровадження нових продуктів (скорочуючи Time-to-Market) та впровадження складних обчислювальних навантажень, наприклад, складних моделей штучного інтелекту для обробки великих даних у реальному часі [1].

Паралельно набирає обертів тренд embedded finance (вбудованих фінансів), який радикально змінює саму концепцію надання банківських послуг. Фінансові інструменти все частіше інтегруються безпосередньо в нефінансові платформи та користувацький досвід. Наприклад, клієнт може отримати кредит у кілька кліків прямо в процесі купівлі автомобіля в онлайн-салоні або оформити страхування під час бронювання готелю. Це розмиває традиційні межі банкінгу, перетворюючи його на невидиму, але життєво необхідну функцію в рамках ширших цифрових екосистем [1].

Очікування клієнтів щодо миттєвості операцій переростають в вимогу до роботи в реальному часі (Real-Time) по всіх фронтах. Стандартом стають не лише миттєві платежі за системами на кшталт SEPA Instant, але й одержання миттєвих кредитних рішень, негайне зарахування коштів та онлайн-оновлення фінансової звітності. Відповідно, архітектура сучасних систем ДБО має орієнтуватися на події (event-driven) та забезпечувати глибоку інтеграцію з усіма підсистемами, що працюють у режимі реального часу [1].

Прогресивні фінансові установи також зміщують фокус з виконання транзакцій на підтримку фінансового благополуччя (Financial Wellness) своїх клієнтів. Це означає трансформацію банківського додатку в особистого фінансового помічника. Інструменти для автоматизованого бюджетування, детального аналізу витрат за категоріями, накопичення на конкретні цілі та механізми мікро-заощаджень (наприклад, round-up – заокруглення суми покупки) стають обов’язковими атрибутами конкурентного сервісу, спрямованого на поглиблення довіри та тривалих стосунків [1].

Під впливом суспільних очікувань та регуляторного тиску набуває значення тренд відповідного та зеленого банкінгу (ESG). Клієнти, особливо

молодші покоління, все більше цінують можливість робити свідомі фінансові вибори. У відповідь банки розвивають функціонал, що дозволяє відстежувати екологічний вплив своїх покупок, інвестувати в сталі проєкти або отримувати вигідні умови для «зелених» кредитів на енергоефективність. ДБО стає головним каналом для комунікації цієї цінності та надання відповідних інструментів [1].

Нарешті, спостерігається тенденція до глобалізації цифрових пропозицій. Необанки та великі технологічні гравці все частіше виходять зі своїми мобільними продуктами за національні кордони, створюючи конкуренцію місцевим установам на їхній же території. Це змушує традиційні банки розглядати розвиток своїх ДБО-платформ не лише в локальному, а й в міжнародному контексті, забезпечуючи підтримку мультивалютності, адаптацію інтерфейсу та відповідність різним регуляторним вимогам [1].

Разом ці тенденції свідчать про те, що дистанційне банківське обслуговування еволюціонує від ізольованого транзакційного інструменту до комплексної, інтелектуальної та етично орієнтованої платформи, що глибоко інтегрована в цифрове повсякдення клієнта [1].

1.2 Архітектурні підходи до розробки фінтех-систем

Вибір архітектури для системи дистанційного банківського обслуговування (ДБО) є стратегічним рішенням, що безпосередньо впливає на її безпеку, продуктивність, масштабованість та можливість подальшої модернізації. Фінтех-системи, до яких належить ДБО, пред’являють підвищені вимоги до надійності, обробки транзакцій та відповідності нормативним вимогам. Сучасна практика розробки пропонує кілька ключових архітектурних парадигм, кожна з яких має свої переваги, недоліки та область оптимального застосування [1].

Монолітна архітектура традиційно була основним підходом для банківських систем. У такій моделі всі компоненти додатку (інтерфейс

користувача, бізнес-логіка, логіка доступу до даних) об’єднані в єдину кодобазую і розгортаються як єдине ціле. Основними перевагами моноліту є простота розробки на ранніх етапах, легкість тестування та розгортання, а також висока продуктивність завдяки внутрішнім викликам функцій. Однак, у міру зростання системи моноліт проявляє суттєві недоліки: ускладнення внесення змін через високе зачеплення компонентів, труднощі з масштабуванням окремих навантажених функцій (масштабування лише вертикальне, «в цілому»), довгі цикли випуску оновлень та високий ризик при модернізації технологічного стеку. Для невеликих або доволі статичних систем ДБО моноліт може бути життєздатним рішенням, але для динамічних, складних платформ, що потребують швидкого оновлення, його обмеження стають критичними [1].

Мікросервісна архітектура стала відповіддю на виклики масштабування та гнучкості великих систем. Вона передбачає розбиття додатку на набір дрібних, слабкозв’язаних (loosely coupled) та незалежно розгортаємих (independently deployable) сервісів. Кожен мікросервіс відповідає за окрему бізнес-спроможність (наприклад, «Керування рахунками», «Виконання платежів», «Аутентифікація») та комунікує з іншими через легкі механізми, зазвичай RESTful API або асинхронні повідомлення (message brokers, наприклад, Apache Kafka або RabbitMQ). Ключові переваги такого підходу для ДБО включають незалежну масштабованість кожного сервісу відповідно до його навантаження, можливість використання різних технологій та мов програмування для різних завдань, стійкість до відмов (відмова одного сервісу не «валить» всю систему), а також прискорені цикли розробки окремих команд. Недоліками мікросервісів є значна складність оркестрації, моніторингу та відлагодження розподіленої системи, вищі вимоги до інфраструктури та DevOps-практик, а також потенційні проблеми з узгодженістю даних (data consistency) та додатковою мережевою затримкою [1].

Проміжним рішенням між монолітом і мікросервісами є модульна монолітна архітектура (Modular Monolith). У цій моделі кодова база чітко

розділена на внутрішні модулі з чіткими контрактами та мінімальною залежністю один від одного, проте система все ще компонується і розгортається як єдине ціле. Це дозволяє отримати переваги структурної ясності та легшого тестування, підготувавши ґрунт для можливого майбутнього розбиття на мікросервіси, одночасно уникаючи надмірної складності повної розподіленої архітектури на початкових етапах проєкту [1].

Для конкретної розробки серверної частини системи ДБО у цій роботі обрано фреймворк NestJS, який ідеально підтримує саме модульний підхід. Він надає вбудовані засоби для створення чітко структурованих застосунків з розділенням відповідальності (контролери, провайдери, модулі), що дозволяє будувати архітектуру, близьку до модульного моноліту. Такий вибір дає можливість створити добре організовану, тестовну та підтримувану систему з чіткими межами модулів (наприклад, AuthModule, AccountsModule, PaymentsModule), зберігаючи при цьому відносну простоту розгортання та управління. У разі потреби подальшого масштабування, чітко визначені модулі NestJS можуть бути відносно легко трансформовані в окремі мікросервіси [1].

Сервіс-орієнтована архітектура (SOA) та подієво-орієнтована архітектура (Event-Driven Architecture, EDA) часто використовуються як доповнення до основних підходів, особливо в складних системах. EDA є критично важливою для реалізації таких сценаріїв, як обробка транзакцій у реальному часі, розсилка сповіщень або синхронізація даних між сервісами, забезпечуючи слабку зв’язаність та високу гнучкість [1].

Отже, вибір архітектурного підходу для фінтех-системи, такої як ДБО, є компромісом між складністю, гнучкістю, безпекою та вимогами до часу виходу на ринок. Для цілей даної роботи модульний підхід на основі NestJS визнаний оптимальним, оскільки він дозволяє реалізувати систему, що відповідає основним нефункціональним вимогам (структурованість, тестовність), і закладає фундамент для її еволюції в бік більш розподіленої архітектури у майбутньому [1].

1.3 Огляд технологічного стеку для сучасних вебзастосунків

Вибір технологічного стека для розробки системи дистанційного банківського обслуговування є фундаментальним рішенням, що визначає довгострокову життєздатність, безпеку та конкурентоздатність продукту. Стек повинен забезпечувати не лише функціональну повноту, а й відповідати підвищеним вимогам до продуктивності, транзакційної цілісності та захисту від кіберзагроз, характерним для фінансового сектору [1].

Серверна частина та мова програмування. Серед сучасних рішень провідну позицію для високонавантажених мережевих застосунків займає платформа Node.js. Її асинхронна, неблокуюча архітектура забезпечує високу пропускну здатність при обробці великої кількості одночасних з’єднань, що є критичним для системи, де тисячі користувачів можуть одночасно ініціювати платежі або запитувати виписки. Для структурованої розробки на Node.js існують фреймворки різного рівня абстракції. У цій роботі обрано NestJS – прогресивний фреймворк, що надає готову архітектуру корпоративного рівня. Він пропонує вбудовану модульність, потужну систему Dependency Injection, підтримку TypeScript «з коробки» та інтеграцію з усіма критичними компонентами (бази даних, кешування, черги, автентифікація). Це дозволяє розробникам зосередитися на бізнес-логіці, а не на побудові інфраструктури, забезпечуючи при цьому високу якість коду, тестованість та легкість супроводу [1].

Клієнтська частина та фреймворк інтерфейсу. На стороні користувача основним вибором є між такими фреймворками, як Angular, React та Vue.js. Для складного корпоративного застосунку, такого як ДБО, де важлива сувора структура, типобезпека та наявність вбудованих інструментів для маршрутизації, управління станом та безпеки, оптимальним вибором є Angular. Будучи повноцінною платформою, а не просто бібліотекою, Angular надає чітку архітектурну модель (компоненти, модулі, сервіси), використовує TypeScript для виявлення помилок на етапі компіляції та забезпечує потужні

механізми для створення динамічних, адаптивних інтерфейсів. Його двостороннє зв’язування даних та ін’єкція залежностей значно прискорюють розробку складних форм та бізнес-логіки на стороні клієнта [2].

Система управління базами даних (СУБД). Надійне та консистентне зберігання фінансової інформації є основним вимогою. Реляційні СУБД, такі як PostgreSQL, є стандартом де-факто для подібних систем завдяки підтримці транзакцій ACID (Atomicity, Consistency, Isolation, Durability), що гарантують цілісність даних при одночасних операціях. PostgreSQL відрізняється високою надійністю, розширеною системою дотримання цілісності та підтримкою складних типів даних (наприклад, JSON). Вона ідеально підходить для зберігання структурованих даних про користувачів, рахунки, транзакції та платежні реквізити. Її можливості реплікації та кластеризації забезпечують необхідний рівень доступності та відмовостійкості [2].

Інфраструктура безпеки та додаткові компоненти. Безпека в системі ДБО реалізується на кількох рівнях. Для автентифікації та авторизації використовується механізм JSON Web Tokens (JWT), що дозволяє створювати statless-сесії та безпечно передавати закодовані дані про користувача між клієнтом і сервером. Для захисту від міжсайтової підробки запитів (CSRF) та ін’єкцій вбудовані механізми фреймворків посилюються суворими політиками CORS. Для підвищення продуктивності частих запитів (наприклад, курсів валют, статичних даних) та зменшення навантаження на базу даних застосовується in-memory база даних Redis, що також може використовуватися для керування чергами завдань або тимчасовим зберіганням токенів [2].

Інструменти розробки та інтеграції. Робота в єдиному контурі типобезпеки забезпечується використанням TypeScript як на сервері (NestJS), так і на клієнті (Angular). Для автоматизації процесів збирання, тестування та розгортання використовується CI/CD пайплайн, що може бути реалізований за допомогою GitHub Actions, GitLab CI/CD або Jenkins. Контейнеризація за допомогою Docker забезпечує однаковість середовища виконання від розробки до продакшну, спрощуючи масштабування та розгортання [2].

Обраний технологічний стек NestJS / Angular / PostgreSQL є стратегічно виваженим рішенням для розробки корпоративної системи ДБО. Його основні переваги – типобезпека, чітка архітектура, продуктивність та комплексність – прямо відповідають критичним вимогам фінансового сектора: надійності, безпеці та підтримці. Потенційні виклики, пов'язані зі складністю, можуть бути ефективно мінімізовані завдяки належній документації, кодстайлу та поступовому введенню розробників у проєкт. Цей стек забезпечує міцну основу для створення масштабованого, безпечного та конкурентоздатного продукту [2].

1.4 Принципи безпеки та захисту даних у банківських вебсистемах

Безпека є абсолютним імперативом при розробці будь-якої фінансової системи, особливо системи дистанційного банківського обслуговування (ДБО), яка стає основним вектором атак зловмисників. Захист тут має бути багаторівневим (defense in depth) та охоплювати всі компоненти архітектури – від мережевого рівня до логіки бізнес-додатку. Дотримання суворих принципів безпеки не лише захищає активи клієнтів і репутацію банку, але й є обов'язковою вимогою регуляторних органів (наприклад, НБУ, PCI DSS, GDPR) [2].

Фундаментальним принципом є «нульова довіра» (Zero Trust), згідно з яким жоден користувач, пристрій чи мережевий потік не вважаються надійними за замовчуванням, незалежно від їхнього розташування (всередині чи за межами периметру). Кожен запит на доступ до ресурсу повинен проходити сувору перевірку автентифікації та авторизації. У практичній реалізації це означає відмову від ідеї «безпечної внутрішньої мережі» та застосування мікросегментації, де доступ між компонентами системи (наприктлад, між сервером додатку та базою даних) також контролюється [2].

Надійна автентифікація (Authentication) виходить далеко за рамки простої пари логін-пароль. Сучасний підхід передбачає багатофакторну

автентифікацію (MFA), яка поєднує щонайменше два різні типи факторів: знання (пароль, PIN), володіння (смартфон з TOTP-додатком, SMS-токен, апаратний ключ) та біометрію (відбиток пальця, розпізнавання обличчя). Для сесійного управління використовуються короткоживучі JWT (JSON Web Tokens), які підписуються за допомогою стійких алгоритмів (наприклад, RS256) і зберігаються безпечним чином (httpOnly, Secure cookies). Обов'язковим є захист від атак на сесії, таких як фіксація (session fixation) або перехоплення [2].

Принцип найменших привілеїв (Principle of Least Privilege, PoLP) є основою для системи авторизації (Authorization). Кожен користувач (або системний компонент) отримує мінімальний набір прав, необхідних для виконання своїх функцій. Це реалізується за допомогою моделей RBAC (Role-Based Access Control) або, для більш гнучких сценаріїв, ABAC (Attribute-Based Access Control). Наприклад, звичайний клієнт не може ініціювати внутрішній банківський переказ, а оператор кол-центру не має доступу до адміністративних функцій бек-офісу [2].

Захист даних як у стані спокою, так і під час передачі вимагає обов'язкового використання сильного шифрування. Усі конфіденційні дані (паролі, PIN-коди, номера рахунків, повна фінансова історія) у базі даних мають зберігатися у зашифрованому вигляді. Паролі обов'язково хешуються з використанням стійких адаптивних алгоритмів (наприклад, Argon2, bcrypt). Зв'язок між клієнтом і сервером, а також між внутрішніми сервісами, має захищатися сучасними версіями протоколу TLS (1.3 або вище), що забезпечує конфіденційність та цілісність переданої інформації [3].

Проактивний захист від вебвразливостей є обов'язковою практикою. Система повинна включати захист від усіх критичних загроз, визначених OWASP Top 10, зокрема:

– ін'єкцій (SQL, NoSQL, Command): використання параметризованих запитів ORM (наприклад, TypeORM/Prisma в NestJS) та валідації всіх вхідних даних;

- міжсайтового скриптингу (XSS): екранування виводу, політика безпеки контенту (Content Security Policy – CSP), використання фреймворків, що автоматично ексейплять дані (як Angular);
- міжсайтової підробки запитів (CSRF): використання токенів CSRF та перевірка заголовків Origin/Referer;
- небезпечної десеріалізації: валідація та санітизація всіх серіалізованих об’єктів.

Аудит та моніторинг (Auditing & Logging) формують основу для виявлення інцидентів та розслідування. Усі критичні події (вхід/вихід, фінансові операції, зміни конфігурації, спроби несанкціонованого доступу) мають реєструватися в централізованій системі логування без можливості редагування. Логи мають містити незмінний часовий штамп, ідентифікатор користувача, тип події та всі релевантні деталі. Реалізація SIEM (Security Information and Event Management)-систем дозволяє аналізувати ці логи в реальному часі для виявлення підозрілих патернів поведінки [3].

Управління вразливостями та безперервна оцінка безпеки є процесом, а не разовим заходом. Це включає регулярне сканування коду на наявність вразливостей (SAST), сканування залежностей (SCA) на відомі проблеми, пентестинг інфраструктури та додатку, а також проведення навчань з кібербезпеки для розробників і адміністраторів. Кожен новий реліз має проходити перевірку безпеки перед розгортанням у продуктивне середовище [3].

Таким чином, безпека банківської вебсистеми – це комплексний і безперервний процес, вбудований у кожен рівень архітектури. Він будується на суворих принципах нульової довіри, найменших привілеїв та проактивного захисту, що технічно реалізується через багатофакторну автентифікацію, детальну авторизацію, потужне шифрування, захист від стандартних атак та всебічний аудит. Дотримання цих принципів є обов’язковою умовою для створення довіри клієнтів та забезпечення фінансової стабільності установи [3].

1.5 Аналіз існуючих рішень та API провідних банків

Розгляд досвіду провідних гравців ринку дозволяє сформувати практичне розуміння стандартів, тенденцій та технологічних підходів у розробці систем дистанційного обслуговування. Аналіз охоплює як закриті клієнтські інтерфейси, так і публічні API, що відкривають нові можливості для інтеграції та інновацій [3].

«Монобанк» як еталон мобільного першого (mobile-first) та високої зручності (UX). Цей необанк побудований навколо прямих та швидких мобільних операцій. Його додаток вирізняється надзвичайно простою реєстрацією через «Дію», інтуїтивним інтерфейсом з мінімальною кількістю кліків для виконання стандартних операцій (переказ за номером телефону, створення віртуальних карток) та активною комунікацією через push-сповіщення. Основна перевага – бездоганний користувацький досвід, що формує високу лояльність. Недоліком є відносно обмежений, порівняно з універсальними банками, набір складних фінансових продуктів (наприклад, іпотеки, інвестиційних рахунків) та закритість архітектури, що ускладнює інтеграцію зі сторонніми сервісами для клієнтів [3].

«ПриватБанк» та його екосистемний підхід через API. Цей банк є піонером у розвитку відкритої API-економіки в Україні. Його публічні API дозволяють стороннім розробникам інтегрувати широкий спектр послуг: від переказів та отримання виписок до створення платіжних терміналів. Перевагою є створення потужної екосистеми партнерських сервісів навколо банку, що розширює його присутність далеко за межі власного додатку. Основним недоліком, з точки зору розробника, є архаїчна в частині технічної документації структура деяких API (наприклад, використання XML замість JSON), що може ускладнювати та сповільнювати процес інтеграції порівняно з сучасними REST/GraphQL рішеннями [3].

«Райффайзен Банк» та акцент на безпеку та складні операції. Додаток цього банку орієнтований на клієнтів, які цінують безпеку та потребують

доступу до повного спектру корпоративних і роздрібних продуктів. Він пропонує розширені налаштування безпеки (детальний контроль сесій, обмеження операцій), потужні інструменти для бізнесу (акцепт платежів, масові виплати) та чітку структуру меню. Перевагою є глибина функціоналу та рівень контролю безпеки. Недоліком часто називають перевантажений інтерфейс, що може бути складним для нових користувачів, та іноді більш довгі сценарії виконання простих операцій через додаткові кроки перевірки [3].

Міжнародний досвід: Revolut та N26 і їх глобальна стандартизація. Ці необанки демонструють підхід, орієнтований на міжнародну аудиторію та швидку адаптацію продукту. Їхні додатки пропонують стандартизований, чистий інтерфейс, миттєві валютні обміни за ринковим курсом, зручне керування підписками та інтеграцію з міжнародними фінтех-сервісами (наприклад, криптобіржами). Їхня основна перевага – продуктивність, чіткість та орієнтація на цифрових кочівників. Недоліком для локальних ринків, таких як український, може бути обмежена підтримка національних платіжних систем (наприклад, українських єПідписів для підтвердження операцій) або відсутність інтеграції з місцевими державними та комунальними сервісами [3].

Висновки для проєктування власної системи ДБО. Аналіз показує, що немає єдиного ідеального рішення. Успішна система має поєднувати в собі кращі практики: швидкість та простоту інтерфейсу на кшталт «Монобанку», відкритість та екосистемність API на зразок «ПриватБанку», рівень контролю безпеки та глибину продукту, властиву «Райффайзену», а також інтернаціональність та стандартизацію, як у Revolut. Для розроблюваної системи це означає необхідність проєктування модульної архітектури, де ядро безпеки та бізнес-логіки залишається незмінним, а інтерфейс користувача та публічні API можуть адаптуватися до різних цільових аудиторій та партнерських інтеграцій. Технічною основою для такого підходу є обраний стек з NestJS, що забезпечує структурованість back-end, та Angular, що дозволяє будувати як складні адміністративні, так і легкі клієнтські інтерфейси на єдиній типобезпечній основі [3].

1.6 Висновки до розділу 1

У першому розділі проведено аналіз сучасного стану та тенденцій ринку дистанційного банківського обслуговування, архітектурних підходів та технологічних рішень для фінтех-систем. Обґрунтовано вибір NestJS як основного фреймворку для серверної частини, враховуючи його модульну архітектуру, підтримку TypeScript та вбудовані механізми безпеки, що відповідають високим вимогам фінансового сектору. Для реалізації клієнтського інтерфейсу обрано Angular, який забезпечує створення динамічних, типобезпечних та структурованих односторінкових додатків. Як систему управління базами даних обрано PostgreSQL – надійну та транзакційно-безпечну СУБД, критичну для гарантії цілісності фінансових даних.

Таким чином, в результаті проведеного аналізу сформовано технологічну та архітектурну основу для розробки системи дистанційного банківського обслуговування. Обраний стек технологій забезпечує реалізацію продуктивної, масштабованої та безпечної платформи, що відповідає сучасним вимогам користувачів та регуляторним стандартам.

2 ПРОЄКТУВАННЯ ТА РОЗРОБКА ВЕБЗАСТОСУНКУ

2.1 Вибір архітектурного стилю

Проєктування системи дистанційного банківського обслуговування вимагає вибору архітектурного стилю, який стане основою для реалізації всієї бізнес-логіки та визначить довгострокову життєздатність продукту. Основними альтернативами є традиційний моноліт, мікросервісна архітектура та проміжний варіант – модульний моноліт. Після комплексного аналізу вимог до продуктивності, безпеки та можливості майбутнього масштабування для даної системи обрано саме підхід модульного моноліту, реалізованого за допомогою фреймворку NestJS [4].

Модульний моноліт передбачає організацію коду у вигляді чітко окреслених внутрішніх модулів, кожен з яких відповідає за певний бізнес-домен (наприклад, аутентифікація, управління рахунками, платіжні операції). Усі модулі компілюються та розгортаються як єдина програмна одиниця. Цей стиль ідеально підтримується NestJS, що пропонує вбудовані механізми для створення модулів, ін’єкції залежностей та організації архітектури на основі шаблону контролера, сервісу та репозиторію. Така структура забезпечує високу ясність коду, спрощує навігацію та колективну розробку, а також робить систему значно легшою для модульного тестування [4].

Ключовою перевагою обраного підходу є оптимальне співвідношення складності та гнучкості на старті проєкту. У порівнянні з повноцінною мікросервісною архітектурою модульний моноліт усуває величезний пласт операційної складності: відсутня потреба в системі оркестрації (Kubernetes), складній мережевій конфігурації, розподіленому трасуванню та управлінню транзакціями між сервісами. Це дозволяє команді розробників зосередитися на реалізації безпечної та коректної бізнес-логіки, а не на підтримці інфраструктури розподіленої системи. Одночасно така архітектура готує ґрунт

для майбутньої еволюції: чіткі межі модулів дозволяють при необхідності виділити найбільш навантажений або специфічний функціонал (наприклад, сервіс генерації PDF-виписок або відправки SMS) в окремий мікросервіс з мінімальними змінами в коді [4].

Для фінансової системи критично важливим є забезпечення транзакційної цілісності даних при таких операціях, як переказ коштів, де необхідно атомарно оновити баланси двох рахунків. Модульний моноліт, що працює з єдиною базою даних (PostgreSQL), дозволяє легко використовувати механізми транзакцій ACID на рівні СУБД або оркеструвати їх за допомогою менеджера транзакцій фреймворку. У мікросервісній архітектурі реалізація подібних розподілених транзакцій вимагає значно складніших паттернів, таких як Saga, що підвищує ризик помилок та складність відлагодження [4].

Додатковою перевагою є висока продуктивність внутрішньої взаємодії. Виклики між сервісами різних модулів відбуваються в межах одного процесу як звичайні виклики методів об’єктів. Це на порядки швидше, ніж будь-яка мережева комунікація (HTTP, gRPC) між окремими мікросервісами, що знижує загальну латентність системи та підвищує швидкість обробки запитів [4].

Основним потенційним обмеженням модульного моноліту є масштабування. Якщо один конкретний модуль (наприклад, модуль звітності) створює дуже високу обчислювальну або пам’ятну нагрузку, доведеться масштабувати весь застосунок цілком. Проте цей ризик можна мітигувати на етапі проєктування: ресурсомісткі асинхронні завдання варто відразу виносити в окремі фонові воркери або черги завдань (використовуючи, наприклад, бібліотеку Bull з Redis), а модулі з чіткою специфікою проєктувати з розрахунком на їх потенційне відокремлення [4].

На рисунку 2.1 зображено архітектуру системи дистанційного банківського обслуговування, що реалізована за принципом модульного моноліту з використанням фреймворку NestJS. Діаграма наочно ілюструє три основні рівні системи та їх внутрішню структуру.

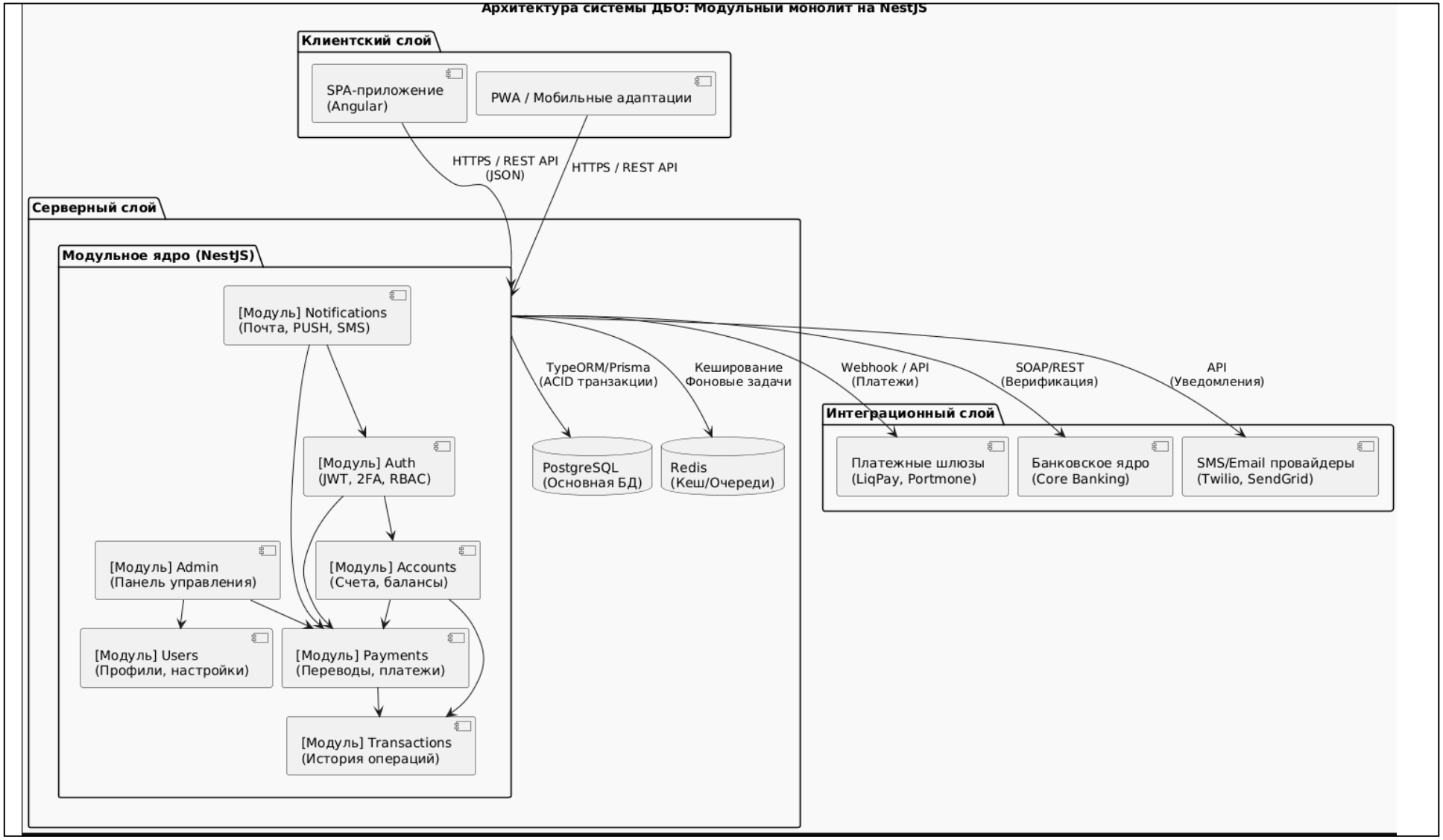

Рисунок 2.1 – Модульний моноліт NestJS

На верхньому рівні розташований клієнтський шар (Frontend), представлений SPA-застосунком, розробленим на Angular, та адаптивними PWA-версіями для мобільних пристроїв. Цей шар відповідає за надання інтуїтивного користувацького інтерфейсу для взаємодії з банківськими послугами [4].

Центральним елементом архітектури є серверний шар (Backend), який складається з модульного ядра на NestJS та систем зберігання даних. Модульне ядро організоване у вигляді семи функціональних модулів:

- AuthModule – відповідає за автентифікацію через JWT, двофакторну аутентифікацію (2FA) та управління ролями (RBAC);
- AccountsModule – забезпечує управління банківськими рахунками та балансами;
- PaymentsModule – реалізує логіку здійснення платежів, переказів та роботи з шаблонами операцій;
- TransactionsModule – відповідає за ведення історії всіх фінансових операцій;
- UsersModule – управляє користувацькими профілями та налаштуваннями;
- AdminModule – надає інструменти адміністративного контролю та модерації;
- NotificationsModule – організовує розсилку сповіщень через email, PUSH та SMS.

Модулі взаємодіють між собою через чітко визначені внутрішні інтерфейси, що забезпечує слабку зв’язаність та високу згуртованість кожної бізнес-одиниці. Для зберігання даних використовуються дві системи: PostgreSQL як основна ACID-сумісна база даних для фінансової інформації та Redis для кешування та обробки фонових завдань [5].

Третій рівень становить інтеграційний шар, що забезпечує зв’язок із зовнішніми системами: платіжними шлюзами (LiqPay, Portmone), банківським ядром (Core Banking) для верифікації операцій та провайдерами розсилки повідомлень (Twilio, SendGrid) [5].

Така архітектура забезпечує оптимальний баланс між структурованістю корпоративної системи та ефективністю монолітного застосунку, дозволяючи легко масштабувати окремі функціональні модулі та інтегрувати нові зовнішні сервіси без суттєвої модифікації ядра системи [5].

2.2 Проєктування компонентів серверної частини

Проєктування компонентів серверної частини системи дистанційного банківського обслуговування виконується з урахуванням принципів модульної монолітної архітектури та можливостей фреймворку NestJS. Архітектура серверного додатку будується на основі чіткого розділення відповідальностей між компонентами, що забезпечує високу тестовість, підтримуваність та можливість подальшого масштабування [6].

Серверна частина реалізована з використанням архітектурного шаблону «Контролер-Сервіс-Репозиторій» (Controller-Service-Repository), адаптованого під парадигму NestJS. Кожен функціональний модуль має однакову внутрішню структуру:

- контролери (Controllers) – відповідають за обробку HTTP-запитів, валідацію вхідних даних та формування відповідей (вони виступають як публічний API-шар модуля);
- сервіси (Services) – інкапсулюють бізнес-логіку, обробляють дані, координують роботу різних репозиторіїв та інших сервісів (вони містять основну логіку предметної області);
- репозиторії (Repositories/Entities) – забезпечують абстракцію доступу до даних, інкапсулюючи роботу з базою даних через ORM (TypeORM або Prisma);
- моделі/Сутності (Entities/DTOs) – описують структуру даних, що передаються між шарами: DTO (Data Transfer Object) для валідації вхідних/вихідних даних та Entity для відображення на таблиці БД;

– модулі (Modules) – організують компоненти в єдину функціональну одиницю з власною ін’єкцією залежностей.

Модуль безпеки та автентифікації (AuthModule) є фундаментальним компонентом, що відповідає за захист доступу до системи. Він забезпечує весь цикл управління ідентифікацією користувачів: від реєстрації з ретельною валідацією вхідних даних до безпечного входу в систему. Ключовою особливістю модуля є генерація та перевірка JWT-токенів з обмеженим терміном дії, що забезпечує безпечне управління сесіями. Для підвищення рівня захисту реалізовано двофакторну аутентифікацію через TOTP, що вимагає від користувача підтвердження через мобільний додаток. Модуль також управляє системою ролей та прав доступу через механізм гардів, що дозволяє гнучко контролювати дозволи кожного користувача. Додаткові функції безпеки включають механізми оновлення паролів, відновлення доступу при його втраті, а також автоматичне блокування підозрілих сесій та захист від атак типу brute-force [6].

Модуль управління користувачами (UsersModule) зосереджений на роботі з профільними даними. Він надає інструменти для керування персональною інформацією, контактними даними та параметрами облікового запису. Модуль дозволяє користувачам завантажувати та зберігати аватари профілів, налаштовувати персональні преференції та параметри сповіщень. Важливою функцією є ведення детальної історії дій користувача, що забезпечує можливість аудиту та аналізу активності в системі для виявлення незвичайних дій або потенційних загроз безпеці [7].

Модуль управління рахунками (AccountsModule) інкапсулює всю логіку роботи з фінансовими рахунками різних типів. Він відповідає за створення нових рахунків, їх закриття та переведення в архівний стан. Модуль підтримує роботу з поточними, депозитними та кредитними рахунками, забезпечуючи кожному типу відповідну бізнес-логіку. Користувачі можуть отримувати актуальну інформацію про баланси, переглядати детальні відомості про кожен рахунок та формувати персоналізовані списки доступних рахунків з можливістю фільтрації за різними критеріями [7].

Модуль обробки транзакцій (TransactionsModule) є ядром фінансової активності системи, відповідаючи за всі операції з коштами. Він забезпечує створення та обробку транзакцій різних категорій: від стандартних переказів до складних платежів та операцій зарахування. Критично важливим аспектом роботи модуля є гарантія атомарності фінансових операцій через механізми транзакцій бази даних, що запобігає частковому виконанню операцій та забезпечує цілісність даних. Модуль веде детальний журнал усіх операцій з підтримкою пагінації для великих наборів даних та різноманітних фільтрів для зручного пошуку. Крім того, він забезпечує генерацію фінансових виписок у різних форматах для подальшого аналізу або надання у звітних цілях [8].

Модуль платіжних операцій (PaymentsModule) спеціалізується на здійсненні різних типів платежів. Він дозволяє виконувати внутрішні перекази між рахунками одного користувача, а також перекази іншим учасникам системи за номером телефону або банківськими реквізитами. Для зручності користувачів реалізовано функціонал роботи з шаблонами платежів та можливість налаштування розкладених платежів. Модуль інтегрується зі зовнішніми платіжними шлюзами для поповнення рахунків, а також включає комплексну систему валідації платіжних реквізитів та перевірки дотримання встановлених лімітів операцій [8].

Модуль сповіщень (NotificationsModule) забезпечує ефективну комунікацію системи з користувачами через різні канали. Він організовує розсилку email-сповіщень про важливі події, такі як здійснення фінансових операцій чи зміни у налаштуваннях безпеки. Для миттєвого інформування модуль інтегрується зі зовнішніми провайдерами SMS-розсилок та генерує PUSH-сповіщення для мобільних додатків. Система дозволяє гнучко налаштовувати шаблони повідомлень та регулювати частотність сповіщень відповідно до переваг кожного користувача [8].

Адміністративний модуль (AdminModule) надає комплексні інструменти для управління системою та її користувачами. Він включає спеціалізовану панель адміністратора з аналітичними інструментами та

візуалізацією ключових показників системи. Модуль забезпечує повний контроль над користувацькими обліковими записами та їх правами доступу, дозволяє здійснювати модерацію транзакцій та розслідування потенційних спорів. Адміністратори можуть конфігурувати системні параметри, встановлювати тарифи та керувати іншими аспектами функціонування платформи [8].

Взаємодія між модулями організована через чітко визначені інтерфейси, що забезпечує ефективну співпрацю компонентів при збереженні їхньої незалежності. Для швидкої синхронної взаємодії в межах одного процесу використовуються прямі виклики методів сервісів. Асинхронна обробка дій, що не вимагають негайної реакції, реалізується через систему подій за допомогою EventEmitter. Ресурсоємні операції, такі як генерація звітів або масова обробка даних, виконуються через черги завдань з використанням Redis та бібліотеки Bull. Спільні допоміжні сервіси забезпечують перевикористання функціоналу, що необхідний у кількох модулях, наприклад, валідація даних або механізми кешування [8].

Інфраструктурні компоненти формують технічну основу системи, забезпечуючи її стабільну роботу та спрощуючи розробку. DatabaseModule відповідає за конфігурацію підключення до PostgreSQL та налаштування об’єктно-реляційного відображення. CacheModule інтегрує систему з Redis для ефективного кешування часто використовуваних даних та організації черг завдань. ConfigModule забезпечує централізоване управління конфігурацією через змінні середовища, що дозволяє легко адаптувати систему до різних умов розгортання. LoggerModule реалізує структуроване логування з різними рівнями деталізації для ефективного моніторингу та діагностики проблем. ValidationPipe забезпечує глобальну валідацію вхідних даних, автоматично перевіряючи коректність інформації, що надходить від клієнтів [8].

Принципи проєктування, застосовані при розробці компонентів, забезпечують високу якість архітектури системи. Принцип єдиного обов’язку гарантує, що кожен компонент виконує чітко визначену функцію, що спрощує

його тестування та супровід. Ін’єкція залежностей дозволяє явно декларувати всі залежності компонентів та керувати ними через конструктори, що підвищує тестовість та гнучкість системи. Імутабельність об’єктів передачі даних запобігає неочікуваним змінам стану та спрощує відлагодження. Уніфікована система обробки помилок забезпечує послідовну поведінку при виникненні виняткових ситуацій та надає зрозумілі відповіді клієнтам. Акцент на тестованість компонентів дозволяє легко створювати модульні та інтеграційні тести, що підвищує надійність системи та спрощує її подальшу модернізацію [8].

На рисунку 2.2 зображено діаграму компонентів серверної частини системи дистанційного банківського обслуговування, яка реалізована з використанням архітектури модульного моноліту на основі фреймворку NestJS. Діаграма наочно демонструє багаторівневу структуру системи та взаємозв’язки між її компонентами.

На верхньому рівні розташований шар контролерів (API Gateway & Controllers), що включає шість основних контролерів: AuthController, UsersController, AccountsController, TransactionsController, PaymentsController та AdminController. Кожен контролер відповідає за обробку HTTP-запитів за конкретними маршрутами REST API, валідацію вхідних даних та формування відповідних HTTP-відповідей. Наприклад, AuthController обробляє запити на реєстрацію (/auth/register), вхід (/auth/login), оновлення токенів (/auth/refresh) та двофакторну аутентифікацію (/auth/2fa) [8].

Центральним рівнем системи є шар бізнес-логіки (Business Logic Layer), представлений сімома сервісами: AuthService, UsersService, AccountsService, TransactionsService, PaymentsService, AdminService та NotificationService. Ці сервіси інкапсулюють основну логіку функціонування системи: AuthService відповідає за автентифікацію та управління JWT-токенами, PaymentsService координує платіжні операції, NotificationService забезпечує розсилку сповіщень. Сервіси активно взаємодіють між собою: наприклад, PaymentsService викликає TransactionsService для створення транзакцій та NotificationService для надсилання сповіщень про платежі [8].

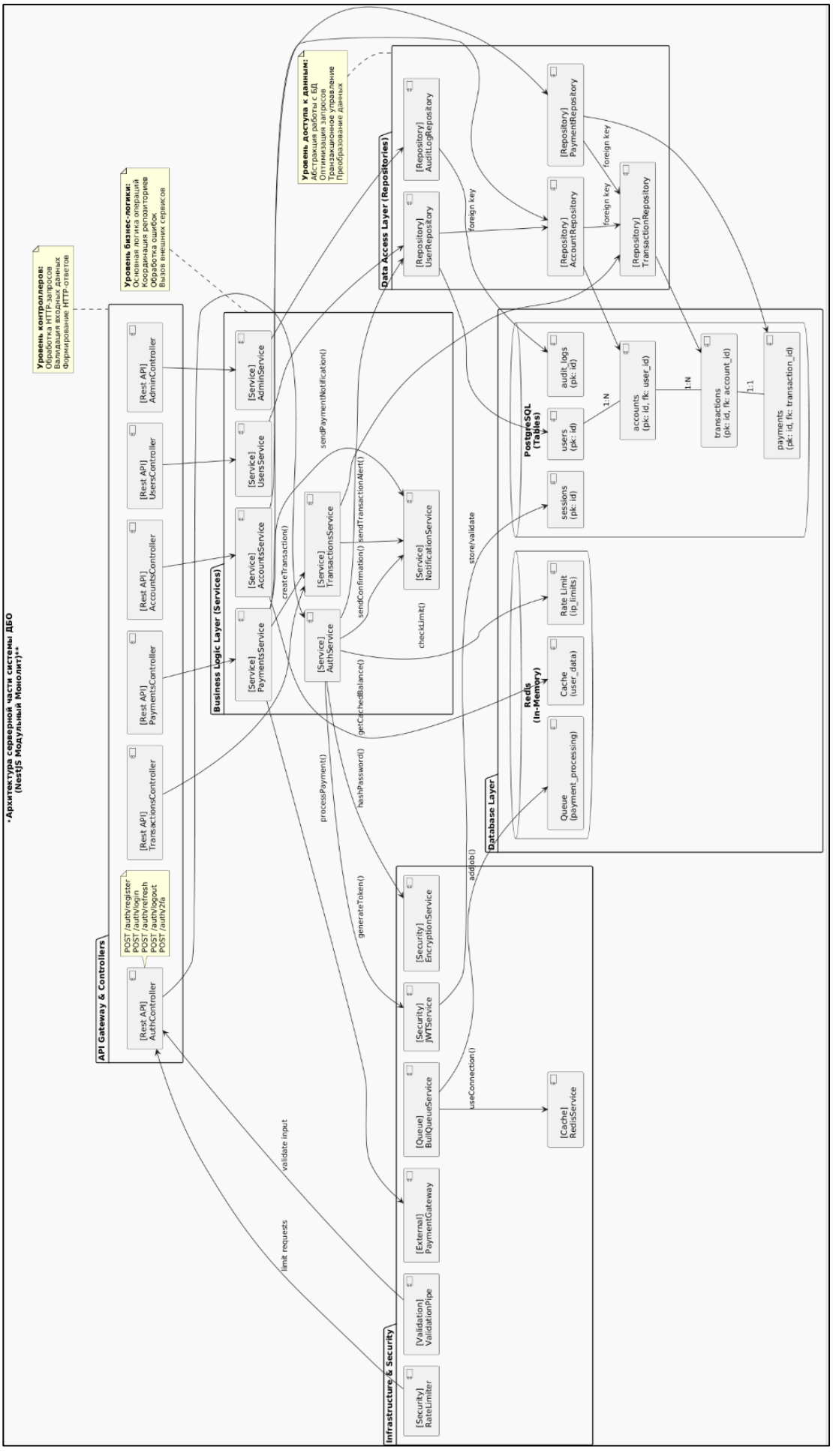

Рисунок 2.2 – Діаграма компонентів серверної частини

Третій рівень становить шар доступу до даних (Data Access Layer), що включає репозиторії: UserRepository, AccountRepository, TransactionRepository, PaymentRepository та AuditLogRepository. Ці компоненти надають абстракцію для роботи з базою даних, інкапсулюючи SQL-запити та ORM-операції. Репозиторії пов’язані між собою через відносини бази даних: UserRepository має зв’язок з AccountRepository через зовнішній ключ, що відображає залежність рахунків від користувачів [8].

В основі архітектури розташований шар бази даних (Database Layer), що включає дві системи зберігання: PostgreSQL для постійного зберігання структурованих даних та Redis для тимчасового кешування та черг завдань. У PostgreSQL організовано таблиці users, accounts, transactions, payments, audit_logs та sessions, пов’язані між собою через первинні та зовнішні ключі. Redis використовується для кешування даних користувачів, організації черг обробки платежів та зберігання лімітів запитів для захисту від перевантаження [8].

Окремим блоком представлені інфраструктурні компоненти (Infrastructure & Security): JWTService для роботи з JSON Web Tokens, EncryptionService для шифрування даних, RateLimiter для обмеження частоти запитів, ValidationPipe для валідації вхідних даних, RedisService для роботи з кешем, BullQueueService для управління чергами завдань та PaymentGateway для інтеграції з платіжними системами [8].

Діаграма також ілюструє ключові взаємозв’язки між компонентами: контролери делегують обробку запитів сервісам, сервіси використовують репозиторії для доступу до даних, репозиторії взаємодіють з таблицями бази даних. Система безпеки інтегрована на всіх рівнях: RateLimiter обмежує запити до AuthController, JWTService перевіряє токени та зберігає інформацію про сесії, EncryptionService забезпечує шифрування конфіденційних даних [8].

Така модульна архітектура забезпечує високу ступінь узгодженості компонентів при збереженні їхньої незалежності, що дозволяє легко тестувати, супроводжувати та масштабувати окремі частини системи без впливу на її загальну функціональність [8].

2.3 Проєктування схеми бази даних

Проєктування схеми бази даних є критично важливим етапом розробки системи дистанційного банківського обслуговування, оскільки від ефективності організації даних залежить продуктивність, надійність та безпека всієї системи. Для зберігання фінансової інформації обрано реляційну систему управління базами даних PostgreSQL, яка забезпечує ACID-сумісність (Atomicity, Consistency, Isolation, Durability) та підтримує складні транзакції, що є обов’язковою вимогою для банківських систем [9].

Концептуальна модель даних будується на основі виділення ключових сутностей предметної області та визначення зв’язків між ними. Основними сутностями системи є користувач (User) як фізична особа, що користується системою; рахунок (Account) як фінансовий інструмент користувача різних типів; транзакція (Transaction) як запис про будь-яку фінансову операцію; платіж (Payment) з деталями конкретного платежу; категорія (Category) для класифікації витрат і надходжень; шаблон платежу (PaymentTemplate) для збереження налаштувань регулярних операцій; та історія дій (AuditLog) для ведення журналу подій з метою аудиту і забезпечення безпеки [9].

Логічна модель даних реалізується у вигляді набору нормалізованих таблиць з чітко визначеними первинними та зовнішніми ключами та обмеженнями цілісності даних. Центральною таблицею системи є таблиця users, яка зберігає основну інформацію про користувачів. Вона містить унікальний ідентифікатор, електронну пошту (яка є унікальною та індексованою для швидкого пошуку), хеш пароля, що генерується за допомогою алгоритму bcrypt для максимального захисту, ім’я та прізвище користувача, номер телефону для комунікації та двофакторної аутентифікації, а також статус облікового запису (активний, заблокований, видалений). Додатково таблиця включає поля для зберігання ролі користувача (user, admin, moderator), часових міток створення та останнього оновлення запису, а також кількості невдалих спроб входу для реалізації захисту від brute-атак [9].

Для зберігання фінансової інформації проєктується таблиця accounts, яка містить дані про всі рахунки користувачів. Кожен рахунок має унікальний ідентифікатор, посилання на користувача-власника через зовнішній ключ, унікальний номер рахунку за міжнародним стандартом IBAN, тип рахунку (поточний, депозитний, кредитний, ощадний), валюту рахунку, поточний баланс з обмеженням на негативні значення для деяких типів рахунків, статус (активний, закритий, заморожений), дату відкриття та, за необхідності, дату закриття. Для прискорення пошуку рахунків за користувачем та їх номером створюються окремі індекси [9].

Всі фінансові операції системи реєструються в таблиці transactions, яка є найбільш інтенсивною за записами. Кожна транзакція містить унікальний ідентифікатор, посилання на рахунок, з якого або на який здійснено операцію, тип операції (переказ, платіж, зарахування, зняття, комісія), суму операції з точністю до двох знаків після коми, валюту операції, поточний баланс рахунку після виконання операції, детальний опис призначення платежу, статус операції (успішна, очікує підтвердження, відхилена, скасована), а також точні часові мітки створення та виконання транзакції. Для ефективного пошуку та формування звітів створюються складені індекси за рахунком та датою, а також окремий індекс за статусом операцій [9].

Для деталізації платіжних операцій використовується таблиця payments, яка розширює інформацію про транзакції. Вона зберігає дані про отримувача платежу (ім’я, номер рахунку або IBAN), код призначення платежу за стандартом УКТЗЕД, інформацію про платіжний шлюз або банк-кореспондент для міжбанківських переказів, референс операції для відстеження, комісію за обслуговування, та додаткові метадані у форматі JSON для зберігання неструктурованої інформації. Ця таблиця має зовнішній ключ на таблицю транзакцій у відношенні один до одного [9].

Для категорізації фінансових операцій та аналітики витрат проєктується таблиця categories, яка містить ієрархічну структуру категорій. Кожна категорія має назву, тип (витрати, надходження, трансфер), посилання на

батьківську категорію для побудови дерева, колір для візуального відображення та підказку для користувача. Категорії пов’язуються з транзакціями через допоміжну таблицю transaction_categories, що реалізує відношення багато до багатьох [9].

Система також включає таблицю payment_templates для зберігання шаблонів регулярних платежів. Шаблони містять назву, посилання на рахунок відправника, дані отримувача, регулярність виконання (щодня, щотижня, щомісяця, щокварталу), дату наступного виконання, активний статус та додаткові налаштування. Для забезпечення цілісності фінансових даних та дотримання законодавчих вимог проєктується таблиця audit_logs, яка фіксує всі важливі дії в системі: хто, коли, що зробив, з якими параметрами та який був результат. Ця таблиця є критично важливою для розслідування інцидентів, аудиту та відповідності нормативним вимогам [9].

Для покращення продуктивності системи та обробки складних запитів створюється комплексна система індексів. Крім стандартних індексів первинних ключів, додаються складені індекси для часто фільтрованих полів, таких як (account_id, created_at) для швидкого отримання історії операцій за період, (user_id, status) для пошуку активних рахунків користувача, та (type, status, created_at) для адміністративних звітів. Для повнотекстового пошуку в описах транзакцій створюються індекси GIN, а для гешування конфіденційних даних використовуються індекси BRIN [9].

Заходи безпеки реалізуються на рівні бази даних через обмеження цілісності, транзакційні механізми та шифрування. Всі зовнішні ключі налаштовані з каскадним видаленням або обмеженням залежно від бізнес-логіки. Для критичних операцій, таких як переказ коштів, використовуються транзакції рівня ізоляції SERIALIZABLE для гарантії консистентності даних. Конфіденційна інформація, така як номера рахунків контрагентів, зберігається у зашифрованому вигляді з використанням pgcrypto, а доступ до даних контролюється через систему ролей та привілеїв PostgreSQL [9].

Архітектура бази даних також передбачає механізми масштабування та резервування. Для великих таблиць, таких як transactions, планується

партиціонування за датою для підвищення продуктивности запитів та спрощення архівування даних. Налаштовується реплікація master-slave для забезпечення відмовостійкості та розподілу навантаження на операції читання. Регулярні бекапи здійснюються за стратегією WAL (Write-Ahead Logging) з можливістю відновлення на будь-яку точку в часі для гарантії цілісності фінансових даних навіть у разі серйозних збоїв [9].

На рисунку 2.3 зображено діаграму бази даних системи дистанційного банківського обслуговування, розроблену на основі реляційної СУБД PostgreSQL. Діаграма наочно демонструє структуру основних таблиць, їх атрибути, типи даних, обмеження та взаємозв’язки між сутностями, що формують цілісну модель даних фінансової системи.

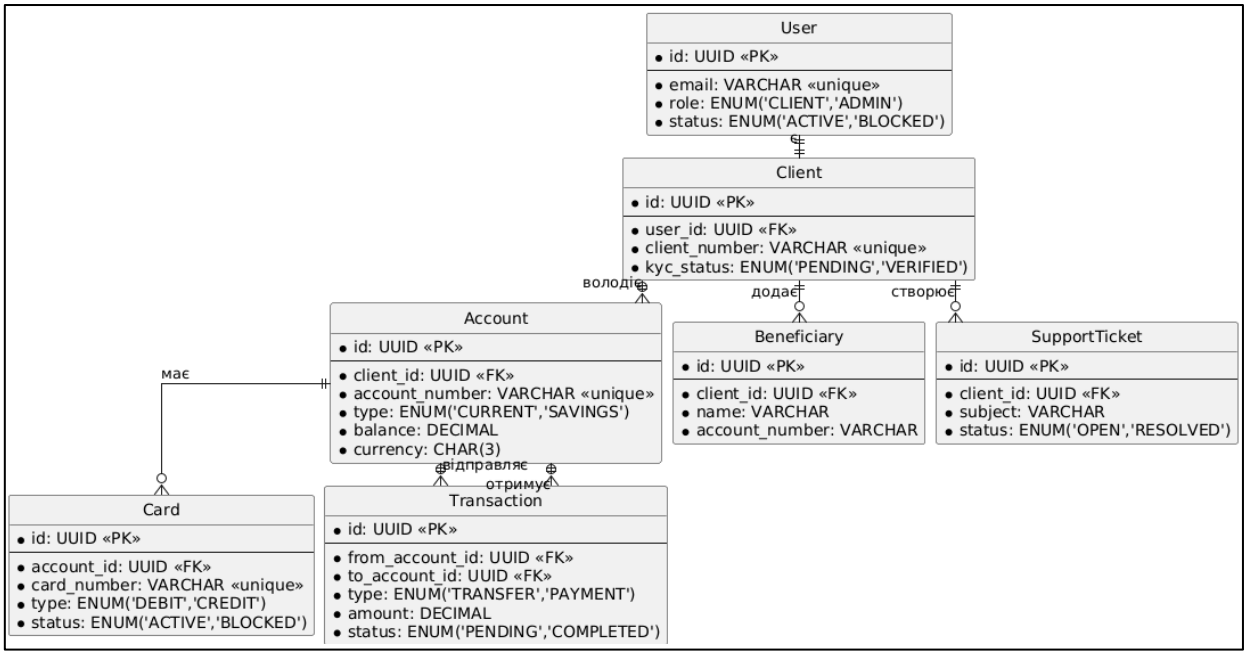

Рисунок 2.3 – Діаграма бази даних

Центральними сутностями схеми є таблиці користувачів, рахунків та транзакцій. Таблиця users містить інформацію про всіх користувачів системи та включає такі критичні поля: унікальний ідентифікатор (id), електронну пошту (email) з обмеженням унікальності, хеш пароля (password_hash), ім’я та прізвище, номер телефону, статус облікового запису, роль користувача, налаштування двофакторної аутентифікації, лічильник невдалих спроб входу

та часові мітки створення та оновлення запису. Для підвищення продуктивності запитів створено індекси за електронною поштою, статусом та роллю, а також датою створення [9].

Таблиця accounts зберігає дані про фінансові рахунки користувачів і містить посилання на власника через зовнішній ключ user_id. Кожен рахунок характеризується унікальним номером за стандартом IBAN, типом (поточний, ощадний, депозитний, кредитний), валютою, поточним та доступним балансами, статусом активності, датами відкриття та закриття, процентною ставкою для депозитних рахунків та кредитним лімітом для кредитних рахунків. Для швидкого пошуку рахунків за користувачем, номером та типом створено відповідні індекси [9].

Серцевиною фінансової системи є таблиця transactions, яка реєструє всі операції з коштами. Кожна транзакція має унікальний ідентифікатор типу UUID, посилання на рахунок, тип операції (переказ, платіж, внесення, зняття, комісія), суму, валюту, баланс після операції, опис, статус виконання, референс-номер, дані контрагента та метадані у форматі JSONB. Для ефективної роботи з великими обсягами транзакційних даних створено складені індекси за рахунком та датою, окремі індекси за типом та статусом операцій, а також повнотекстовий індекс GIN для пошуку в метаданих.

Деталізацію платіжних операцій забезпечує таблиця payments, яка містить специфічну інформацію про кожен платіж: метод оплати, дані отримувача (ім’я та IBAN), код призначення платежу, розмір комісії, курс валюти, ідентифікатор платіжного шлюзу, відповідь шлюзу у форматі JSONB, дату запланованого виконання та ознаку регулярності платежу. Ця таблиця пов’язана з таблицею транзакцій відношенням один до одного [9].

Для забезпечення безпеки та аудиту в системі передбачено спеціальні таблиці. Таблиця sessions відстежує активні сесії користувачів, зберігаючи хеші токенів, інформацію про пристрій, IP-адресу, user-agent та термін дії сесії. Таблиця audit_logs фіксує всі значимі дії в системі для забезпечення відповідності регуляторним вимогам: хто, коли, яку дію виконав, з якими

параметрами та який був результат. Таблиця login_attempts реєструє всі спроби входу в систему для виявлення та блокування підозрілих дій [9].

Система також включає таблиці для підвищення зручності користувачів. Таблиця payment_templates дозволяє зберігати шаблони регулярних платежів з налаштуваннями частоти виконання. Таблиці categories та transaction_categories реалізують механізм категорізації фінансових операцій для аналітики витрат, підтримуючи ієрархічну структуру категорій [9].

Для управління валютними операціями та обмеженнями передбачено таблиці exchange_rates (курси валют з періодами дії), account_limits (ліміти на операції за рахунками) та system_config (системні налаштування). Остання таблиця дозволяє гнучко керувати параметрами системи без необхідності зміни коду [10].

Діаграма також відображає ключові зв’язки між таблицями: користувачі мають один-до-багатьох відношення з рахунками, сесіями, шаблонами платежів та записами аудиту; рахунки пов’язані з транзакціями; транзакції деталізуються через платежі та категоризуються через категорії. Для забезпечення цілісності даних всі зовнішні ключі мають обмеження NOT NULL та налаштовані каскадні операції [10].

Архітектура бази даних передбачає оптимізацію продуктивності через партиціонування таблиці транзакцій за датою, створення BRIN-індексів для великих діапазонів дат, використання перевірочних обмежень (CHECK constraints) для гарантії цілісності даних. Заходи безпеки включають шифрування конфіденційних полів за допомогою модуля pgcrypto, реалізацію захисту на рівні рядків (Row Level Security) для підтримки мультитенантності та повне аудитування всіх змін даних для дотримання вимог GDPR та національного законодавства [10].

Таким чином, розроблена схема бази даних забезпечує не лише ефективне зберігання та обробку фінансової інформації, але й відповідає найвищим стандартам безпеки, надійності та масштабованості, що є критично важливим для банківської системи, яка обробляє конфіденційні дані та здійснює реальні фінансові операції [10].

2.4 Проєктування компонентів клієнтської частини

Проєктування клієнтської частини системи дистанційного банківського обслуговування здійснюється з використанням фреймворку Angular, що дозволяє створити сучасний, адаптивний та безпечний односторінковий застосунок (SPA). Архітектура клієнтської частини будується на принципах компонентно-орієнтованого дизайну, розділення відповідальностей та типобезпеки через TypeScript.

Клієнтська частина організована за модульним принципом з чітким розділенням на функціональні модулі, кожен з яких відповідає за конкретну область бізнес-логіки. Використовується паттерн «розумні та наївні компоненти» (smart/dumb components), де контейнери (розумні компоненти) управляють станом та бізнес-логікою, а презентаційні компоненти (наївні компоненти) відповідають лише за відображення даних. Додаток будується з використанням реактивного підходу через RxJS для обробки асинхронних операцій та управління станом.

Модуль автентифікації та безпеки (AuthModule) є фундаментальним компонентом клієнтської частини, що забезпечує весь цикл управління доступом. Він включає компоненти для реєстрації нових користувачів з валідацією введених даних, входу в систему з підтримкою двофакторної автентифікації, відновлення пароля через email або SMS, а також зміни облікових даних. Модуль інтегрує сервіси для роботи з JWT-токенами, зберігаючи їх у захищеному сховищі (HttpOnly cookies або localStorage з обмеженнями), та реалізує автоматичне оновлення токенів за допомогою refresh-токенів. Додатково реалізовані компоненти для налаштування безпеки: управління активними сесіями, встановлення двофакторної автентифікації через TOTP, налаштування питання безпеки та відстеження історії входів.

Головний модуль та панель управління (DashboardModule) надає користувачам централізований доступ до ключової фінансової інформації. Основним елементом є інтерактивна головна панель (dashboard), яка агрегує дані про всі рахунки користувача: поточні баланси у різних валютах, останні

транзакції з можливістю швидкої фільтрації, активні кредити та депозити, графіки динаміки витрат та надходжень за періодами. Модуль включає віджети для швидкого доступу до найчастіше використовуваних функцій: створення швидкого переказу, поповнення мобільного зв'язку, оплати комунальних послуг. Для візуалізації фінансових даних використовуються бібліотеки Chart.js або D3.js, що дозволяють створювати інтерактивні діаграми та графіки [10].

Модуль управління рахунками (AccountsModule) забезпечує повний контроль над фінансовими рахунками користувача. Він містить компоненти для перегляду детальної інформації по кожному рахунку: історія операцій з пагінацією та фільтрацією, поточні залишки, доступні ліміти, нараховані відсотки. Реалізовано функціонал для створення нових рахунків різних типів (поточних, ощадних, депозитних), закриття існуючих рахунків, відкриття віртуальних карток з моментальною генерацією даних. Окремий компонент відповідає за агрегацію рахунків з різних банків через Open Banking API (PSD2), що дозволяє користувачеві бачити повну фінансову картину в одному інтерфейсі [10].

Модуль платіжів та переказів (PaymentsModule) є найбільш інтенсивним за інтерактивністю компонентом системи. Він включає комплексну форму для здійснення переказів з підтримкою різних сценаріїв: переказ між власними рахунками, переказ іншому користувачеві системи за номером телефону або email, платіж за реквізитами з автоматичною валідацією IBAN, платіж за шаблоном. Форма реалізує покрокову валідацію даних з миттєвим відображенням помилок, автоматичним заповненням даних отримувача при скануванні QR-коду, розрахунком комісії в реальному часі. Додатково модуль містить компоненти для роботи з шаблонами платежів (створення, редагування, видалення), управління регулярними (розкладеними) платежами та історією всіх платіжних операцій [10].

Модуль аналітики та звітів (AnalyticsModule) забезпечує глибокий аналіз фінансової активності користувача. Він включає компоненти для категорізації витрат за автоматично розпізнаними шаблонами або ручним

призначенням категорій, аналізу динаміки витрат за категоріями та періодами, порівняння бюджетів з фактичними витратами. Модуль надає інструменти для генерації звітів у різних форматах (PDF, Excel, CSV): виписки по рахунках, податкові звіти, статистика за категоріями. Реалізовано функціонал для налаштування фінансових цілей (цільового накопичення) з відстеженням прогресу та автоматичними порадами щодо оптимізації витрат [10].

Модуль сповіщень та повідомлень (NotificationsModule) відповідає за комунікацію системи з користувачем. Він включає центр сповіщень, що агрегує всі повідомлення: інформацію про проведені операції, сповіщення про безпеку (підозрілі входи), маркетингові пропозиції, нагадування про регулярні платежі. Реалізовано різні канали доставки: push-сповіщення для браузера та мобільних додатків, email-розсилка, SMS. Користувач може налаштовувати параметри сповіщень: вибирати типи подій для сповіщення, встановлювати час «не турбувати», керувати частотою повідомлень [10].

Сервіс автентифікації (AuthService) інкапсулює всю логіку роботи з токенами, управління сесіями та захистом маршрутів. Він реалізує автоматичне додавання JWT-токена до кожного HTTP-запиту через інтерцептор, обробку помилок автентифікації (401, 403), перенаправлення на сторінку входу при витіканні сесії [10].

Сервіс управління станом (StateService) організує централізоване управління станом додатку. Використовується підхід з використанням сервісів з BehaviorSubject замість повноцінних стейт-менеджерів для збереження простоти архітектури. Сервіс зберігає дані про поточного користувача, активні рахунки, налаштування додатку, забезпечуючи консистентність даних між різними компонентами [10].

Сервіс валідації (ValidationService) надає розширені можливості валідації форм. Він включає кастомні валідатори для банківських даних: перевірка коректності IBAN, валідація номерів карток за алгоритмом Луна, перевірка контрольних сум рахунків, валідація SWIFT-кодів.

Сервіс локалізації (LocalizationService) забезпечує підтримку багатомовності додатку. Він дозволяє динамічно змінювати мову інтерфейсу,

підтримує формати дат, чисел та валют відповідно до локальних налаштувань, забезпечує переклад всіх текстових елементів інтерфейсу [10].

Сервіс роботи з API (ApiService) надає уніфікований інтерфейс для взаємодії з серверною частиною. Він абстрагує HTTP-запити, надаючи типобезпечні методи для роботи з різними endpoint'ами, реалізує повторення запитів при тимчасових помилках мережі, кешування часто запитуваних даних, обробку глобальних помилок [10].

Навігація та маршрутизація реалізована через Angular Router з підтримкою ледачого завантаження модулів (lazy loading) для оптимізації часу завантаження додатку. Маршрути захищені гардами (guards), що перевіряють права доступу користувача та стан автентифікації. Реалізована історія навігації з підтримкою breadcrumbs та кнопкою «Назад» [10].

Адаптивний дизайн забезпечується через використання CSS Grid та Flexbox у поєднанні з бібліотекою Angular Material або Bootstrap. Додаток коректно відображається на пристроях з різними розмірами екрану: від мобільних телефонів до десктопних моніторів. Реалізовані окремі шаблони для мобільних та десктопних версій з оптимізованими інтерфейсами для кожного типу пристроїв [10].

Оптимізація продуктивності досягається через стратегії кешування даних, ледачий завантаження модулів, прелоадинг критичних ресурсів, оптимізацію зображень та асинхронне завантаження скриптів. Додаток використовує Service Workers для реалізації Progressive Web App (PWA) функціоналу, що дозволяє працювати офлайн та отримувати push-сповіщення [10].

Система тестування включає модульні тести компонентів за допомогою Jasmine та Karma, інтеграційні тести сервісів, e2e-тестування користувацьких сценаріїв через Cypress або Protractor. Реалізовано тести доступності (accessibility) для забезпечення відповідності стандартам WCAG [10].

Безпека клієнтської частини забезпечується через захист від XSS-атак через вбудований механізм санітизації Angular, захист від CSRF через додавання токенів до запитів, обмеження кількості спроб автентифікації,

шифрування чутливих даних у локальному сховищі, регулярне оновлення токенів доступу. На рисунку 2.4 зображено діаграму компонентів клієнтської частини системи дистанційного банківського обслуговування, розробленої на основі фреймворку Angular. Діаграма наочно демонструє модульну архітектуру додатку, взаємозв'язки між компонентами та їх функціональні обов'язки [10].

Кореневий модуль додатку є основною точкою входу та включає три ключові елементи: AppComponent як головний компонент, що ініціалізує додаток; AppRoutingModule, який відповідає за навігацію та визначає всі маршрути системи; та AppShell, що задає основний шаблон розмітки додатку. Цей рівень забезпечує базову структуру додатку та координацію всіх функціональних модулів [10].

Функціональні модулі організовані за принципом предметних областей та включають вісім основних модулів. AuthModule відповідає за всю логіку автентифікації та безпеки, включаючи вхід, реєстрацію, двофакторну автентифікацію та відновлення пароля. DashboardModule надає головну панель керування з агрегованою фінансовою інформацією. AccountsModule інкапсулює функціонал роботи з банківськими рахунками. PaymentsModule забезпечує здійснення платежів та переказів. AnalyticsModule надає інструменти аналітики та звітності. AdminModule призначений для адміністраторів системи. SettingsModule дозволяє користувачам керувати налаштуваннями. SharedModule містить спільні компоненти, що використовуються в різних частинах додатку [10].

Компоненти автентифікації формують перший екран взаємодії з користувачем. LoginComponent реалізує форму входу з валідацією даних та обробкою помилок. RegisterComponent забезпечує реєстрацію нових користувачів з покроковим заповненням інформації. TwoFactorComponent реалізує перевірку двофакторної автентифікації через TOTP-коди. PasswordResetComponent дозволяє відновити доступ до облікового запису через email або SMS. EmailVerifyComponent забезпечує підтвердження електронної пошти після реєстрації [10].

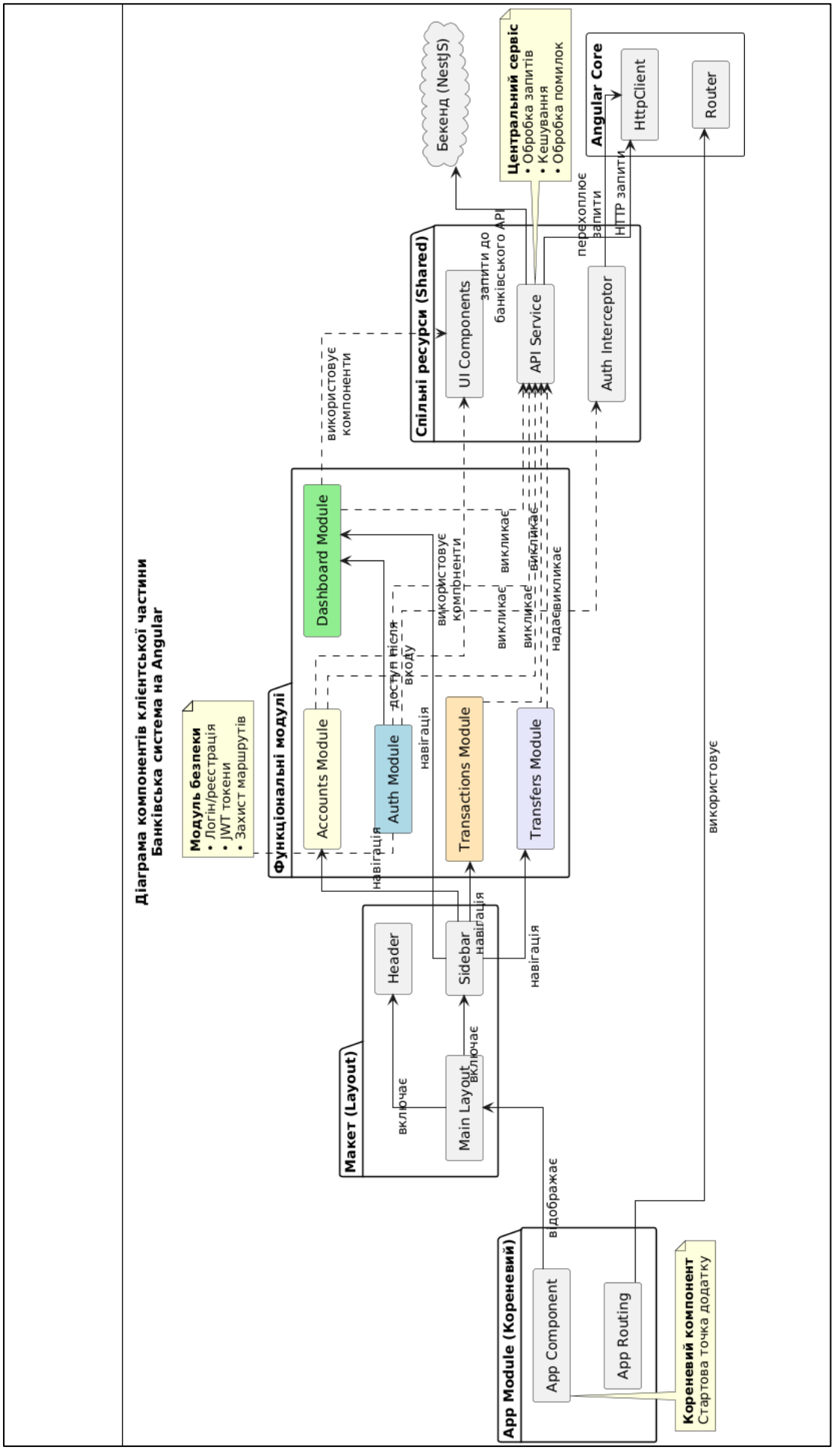

Рисунок 2.4 – Діаграма компонентів клієнтської частини

Компоненти головної панелі створюють інтуїтивний інтерфейс для швидкого доступу до ключової інформації. DashboardComponent є основним контейнером, що організовує розміщення всіх віджетів. AccountsOverviewComponent надає огляд усіх рахунків користувача з балансами у різних валютах. RecentTransactionsComponent відображає останні фінансові операції з можливістю швидкої фільтрації. QuickActionsComponent містить кнопки для найчастіше використовуваних операцій. BalanceChartComponent візуалізує динаміку зміни балансів за обраний період [10].

Компоненти роботи з рахунками забезпечують повний контроль над фінансовими інструментами. AccountsListComponent відображає список усіх рахунків користувача з основними параметрами. AccountDetailComponent показує детальну інформацію про конкретний рахунок, включаючи повну історію операцій. AccountCreateComponent реалізує процес відкриття нових рахунків різних типів. TransactionsHistoryComponent забезпечує розширену фільтрацію та пошук в історії операцій. StatementExportComponent дозволяє генерувати та експортувати виписки у різних форматах [10].

Компоненти платежів реалізують основну фінансову функціональність системи. PaymentFormComponent є складною інтерактивною формою для здійснення переказів та платежів з покроковою валідацією. PaymentTemplatesComponent дозволяє створювати, редагувати та використовувати шаблони для регулярних платежів. ScheduledPaymentsComponent управляє розкладеними та регулярними платежами. PaymentHistoryComponent надає доступ до повної історії всіх платіжних операцій. BeneficiariesComponent дозволяє керувати списком одержувачів для швидкого доступу [10].

Сервіси додатку інкапсулюють бізнес-логіку та взаємодію з серверною частиною. AuthService управляє автентифікацією, токенами та сесіями користувачів. ApiService надає уніфікований інтерфейс для HTTP-запитів до серверу. AccountsService обробляє всі операції з рахунками. PaymentsService

відповідає за здійснення платежів. TransactionsService управляє транзакціями та історією операцій. NotificationsService забезпечує роботу зі сповіщеннями. CacheService реалізує кешування даних на клієнті. StateService управляє глобальним станом додатку [10].

Спільні UI компоненти забезпечують консистентність інтерфейсу та перевикористання коду. HeaderComponent містить верхню навігаційну панель з логотипом, пошуком та профілем користувача. SidebarComponent реалізує бічну панель навігації між розділами додатку. FooterComponent відображає нижній колонтитул з додатковою інформацією. LoadingSpinner показує індикатор завантаження під час виконання асинхронних операцій. ErrorAlert відображає повідомлення про помилки у стандартизованому форматі. ConfirmDialog реалізує діалогові вікна для підтвердження дій. DataTable забезпечує уніфіковане відображення табличних даних з сортуванням та фільтрацією. Pagination реалізує механізм розбиття великих наборів даних на сторінки [10].

Маршрутизація та навігація організовані через Angular Router з підтримкою ледачого завантаження модулів для оптимізації продуктивності. Гарди (guards) забезпечують захист маршрутів: AuthGuard перевіряє наявність активної сесії, RoleGuard контролює доступ на основі ролей користувача, TwoFactorGuard вимагає проходження двофакторної автентифікації для доступу до чутливих розділів.

HTTP інтерцептори розширюють функціональність HTTP-клієнта. AuthInterceptor автоматично додає JWT-токен до кожного запиту на сервер. ErrorInterceptor обробляє всі помилки від сервера та перетворює їх у зрозумілі повідомлення для користувача. LoadingInterceptor керує відображенням індикаторів завантаження під час виконання запитів [10].

Система тем та стилів забезпечує адаптивність та консистентність дизайну. ThemeService дозволяє динамічно змінювати тему додатку (світла/темна). MaterialModule інтегрує компоненти Angular Material для швидкої розробки UI. CustomTheme містить кастомні стилі, що відповідають

фірмовому стилю банку. ResponsiveModule забезпечує коректне відображення на пристроях з різними розмірами екрану.

Така модульна архітектура дозволяє ефективно організувати розробку, забезпечує високу ступінь перевикористання коду, полегшує тестування та супровід додатку. Кожен компонент має чітко визначені обов’язки та інтерфейси взаємодії, що забезпечує масштабованість системи та можливість незалежної розробки різних функціональних блоків [10].

2.5 Висновки до розділу 2

У другому розділі, присвяченому проєктуванню системи дистанційного банківського обслуговування, обґрунтовано архітектурний стиль модульного моноліту на базі NestJS та розроблено детальну архітектуру серверної та клієнтської частин. Визначено компонентну структуру серверної частини з семи функціональних модулів (AuthModule, AccountsModule, PaymentsModule тощо), що забезпечують чітке розмежування відповідальностей та високу тестовість системи.

Спроєктовано схему бази даних PostgreSQL з основними сутностями: користувачі, рахунки, транзакції, платежі, які пов’язані між собою через реляційні зв’язки та ACID-транзакції. Розроблено архітектуру клієнтської частини на Angular з модульною структурою, що включає компоненти автентифікації, головної панелі, управління рахунками та платежів, а також спільні сервіси та UI-компоненти.

В результаті сформовано цілісну технічну документацію та архітектурну основу для подальшої реалізації системи ДБО, що забезпечує високу продуктивність, безпеку, масштабованість та зручність користування відповідно до сучасних стандартів фінтеху.

3 ПРОГРАМУВАННЯ ТА РЕАЛІЗАЦІЯ СИСТЕМИ

3.1 Розробка серверної частини на NestJS

Розробка серверної частини системи дистанційного банківського обслуговування виконувалася з використанням фреймворку NestJS, що дозволило реалізувати модульну монолітну архітектуру з чітким розділенням відповідальностей та високою тестовістю коду.

Проєкт ініціалізовано за допомогою NestJS CLI з генерацією стандартної структури директорій. Основна конфігурація включала налаштування TypeScript з суворими типами, ESLint для забезпечення якості коду та Prettier для форматування. Для керування залежностями використано npm з підтримкою семантичного версіонування. Головний модуль додатку створено з налаштуванням глобальних пайпів для валідації, інтерцепторів для обробки помилок та middleware для логування.

Реалізовано повний цикл управління ідентифікацією користувачів. Для роботи з JWT-токенами інтегровано бібліотеку @nestjs/jwt з налаштуванням секретних ключів через змінні середовища. Паспортні стратегії (Passport.js) забезпечили підтримку локальної автентифікації та JWT-перевірки. Двофакторну автентифікацію реалізовано через TOTP з використанням бібліотеки speakeasy, де секретні ключі шифрувалися перед зберіганням у базі даних. Система RBAC (Role-Based Access Control) реалізована через кастомні декоратори та гарди, що перевіряють права доступу на основі ролей користувача (USER, ADMIN, MODERATOR). Додатково розроблено механізми обмеження запитів (rate limiting) через бібліотеку @nestjs/throttler для захисту від DoS-атак та brute-force.

На рисунку 3.1 зображено структуру проєкту серверної частини системи дистанційного банківського обслуговування, розроблену на основі фреймворку NestJS. Модульна організація коду забезпечує чітке розділення відповідальностей та дотримання принципів SOLID.



Рисунок 3.1 – Структура серверної частини

Коренева структура проєкту містить ключові елементи для запуску та конфігурації додатку. Файл main.ts є точкою входу в програму та відповідає за ініціалізацію NestJS-додатку, налаштування CORS-політик, валідацію, обробку помилок та запуск HTTP-сервера на заданому порту. app.module.ts виконує роль головного модуля, який імпортує та координує всі функціональні модулі системи, а також налаштовує глобальні провайдери та middleware.

Каталог common/ консолідує спільні компоненти, що використовуються в різних частинах системи. Підкаталог decorators/ містить кастомні декоратори для роботи з метаданими, такі як @Roles() для управління доступом на основі

ролей та @CurrentUser() для отримання інформації про автентифікованого користувача. У filters/ розташовані фільтри для обробки винятків, що забезпечують уніфіковане форматування помилок. Каталог guards/ включає гарди безпеки для перевірки JWT-токенів, ролей користувачів та двофакторної автентифікації. Interceptors/ містить HTTP-інтерцептори для додавання заголовків, логування запитів та обробки помилок. Middleware/ включає проміжне програмне забезпечення для аналізу запитів, а pipes/ – кастомні пайпи для трансформації та валідації даних.

Каталог config/ відповідає за централізоване управління конфігурацією додатку. database.config.ts налаштовує параметри підключення до PostgreSQL, включаючи хост, порт, облікові дані, сутності та міграції. redis.config.ts конфігурує з'єднання з Redis для кешування та черг завдань. jwt.config.ts визначає параметри JWT-токенів: секретні ключі, час життя та алгоритми підпису. app.config.ts зберігає загальні налаштування додатку, такі как режим роботи, порт сервера та URL клієнтської частини.

Модульна архітектура реалізована в каталозі modules/, де кожен підкаталог представляє окрему предметну область. auth/ – модуль автентифікації, що включає контролери для реєстрації та входу, сервіси для роботи з JWT-токенами та двофакторною автентифікацією, стратегії для Passport.js, гарди для захисту маршрутів, DTO для валідації вхідних даних та інтерфейси для типізації. users/ – модуль управління користувачами з контролерами, сервісами, сутністю User та DTO. accounts/ – модуль для роботи з банківськими рахунками. transactions/ – модуль обробки фінансових операцій. payments/ – модуль для здійснення платежів. notifications/ – модуль розсилки сповіщень. admin/ – адміністративний модуль для управління системою.

Каталог database/ містить конфігурацію бази даних, міграції для створення та зміни таблиць, а також сиди для наповнення тестовими даними. shared/ включає спільні модулі для кешування, черг завдань та логування.

Така структура проєкту забезпечує високу організованість коду, спрощує розробку, тестування та супровід системи. Кожен модуль має чітко

визначені межі та відповідальності, що дозволяє командам розробників працювати паралельно над різними частинами системи без конфліктів. Модульна архітектура також полегшує майбутнє масштабування та рефакторинг коду.

На рисунку 3.2 зображено головний модуль (app.module.ts) серверної частини системи дистанційного банківського обслуговування, який виконує функцію центрального хабу для імпорту та конфігурації всіх компонентів системи. Цей модуль є кореневим у ієрархії застосунку NestJS та відповідає за ініціалізацію та координацію всіх залежностей.

```
import { Module } from '@nestjs/common';
import { ConfigModule, ConfigService } from '@nestjs/config';
import { TypeOrmModule } from '@nestjs/typeorm';
import { ThrottlerModule } from '@nestjs/throttler';
import { ScheduleModule } from '@nestjs/schedule';

import { AppController } from './app.controller';
import { AppService } from './app.service';

import { AuthModule } from './modules/auth/auth.module';
import { UsersModule } from './modules/users/users.module';
import { AccountsModule } from './modules/accounts/accounts.module';
import { TransactionsModule } from './modules/transactions/transactions.module';
import { PaymentsModule } from './modules/payments/payments.module';
import { NotificationsModule } from './modules/notifications/notifications.module';
import { AdminModule } from './modules/admin/admin.module';
import { DatabaseModule } from './database/database.module';

import { configuration } from './config/app.config';
import { JwtConfigService } from './config/jwt.config';
import { RedisConfigService } from './config/redis.config';

@Module({
  imports: [
    // Конфігурація
    ConfigModule.forRoot({
      isGlobal: true,
      load: [configuration],
    }),

    // База даних
    TypeOrmModule.forRootAsync({
      imports: [ConfigModule],
      useFactory: (configService: ConfigService) => ({
        type: 'postgres',
        host: configService.get('database.host'),
        port: configService.get('database.port'),
        username: configService.get('database.username'),
        password: configService.get('database.password'),
        database: configService.get('database.name'),
        entities: [__dirname + '/**/*.entity{.ts,.js}'],
        migrations: [__dirname + '/database/migrations/*{.ts,.js}'],
        synchronize: configService.get('app.env') === 'development',
```

Рисунок 3.2 – Головний модуль

Секція імпорту модулів демонструє комплексний підхід до конфігурації додатку. Першим імпортується ConfigModule.forRoot(), який забезпечує глобальний доступ до змінних середовища та конфігураційних файлів через ConfigService. Цей модуль налаштований з опцією isGlobal: true, що дозволяє використовувати конфігурацію в будь-якому місці додатку без необхідності повторного імпорту. Завантаження користувацької конфігурації здійснюється через функцію configuration, що обробляє файли з каталогу config.

Інтеграція з базою даних реалізована через TypeOrmModule.forRootAsync(), який налаштовує асинхронне підключення до PostgreSQL. Конфігурація використовує фабричний метод, що отримує параметри з ConfigService: тип бази даних, хост, порт, облікові дані, назву бази даних. Важливою особливістю є автоматичне завантаження сутностей з усіх каталогів проєкту за шаблоном *.entity{.ts,.js} та міграцій з каталогу database/migrations/. Опція synchronize активується лише в режимі розробки для автоматичної синхронізації схеми бази даних, тоді як у продакшені використовуються міграції. Додаткові налаштування включають логування SQL-запитів та обмеження максимальної кількості підключень.

Система захисту від DoS-атак інтегрована через ThrottlerModule.forRootAsync(), який налаштовує обмеження частоти запитів на основі конфігурації. Модуль використовує параметри TTL (time-to-live) та ліміт запитів, що дозволяє захищати критичні endpoint’и від перевантаження та brute-force атак. Це особливо важливо для операцій автентифікації та здійснення платежів.

Планувальник завдань представлений ScheduleModule.forRoot(), що надає можливості для виконання періодичних та відкладених завдань. У контексті банківської системи цей модуль використовується для автоматичного нарахування відсотків по депозитах, закриття прострочених кредитів, генерації щомісячних звітів та синхронізації з зовнішніми платіжними системами.

Імпорт функціональних модулів включає всі основні компоненти системи: AuthModule для автентифікації, UsersModule для управління

користувачами, AccountsModule для роботи з рахунками, TransactionsModule для обробки операцій, PaymentsModule для здійснення платежів, NotificationsModule для розсилки сповіщень, AdminModule для адміністративного управління та DatabaseModule для додаткових налаштувань бази даних. Кожен модуль інкапсулює свою предметну область та надає чітко визначені інтерфейси для взаємодії.

Секція контролерів та провайдерів визначає головний контролер (AppController) та сервіс (AppService) додатку. Додатково реєструються конфігураційні сервіси: JwtConfigService для роботи з JWT-токенами та RedisConfigService для налаштування Redis. Ці провайдери забезпечують централізоване управління критичними параметрами системи.

Ключові архітектурні особливості головного модуля включають використання асинхронної конфігурації для всіх зовнішніх залежностей, що дозволяє динамічно отримувати параметри з змінних середовища або систем зберігання конфігурацій. Застосування фабричних методів (useFactory) забезпечує гнучкість у налаштуванні модулів та дозволяє інжектувати залежності під час ініціалізації. Глобальні налаштування через ConfigModule та TypeOrmModule спрощують доступ до конфігурації та бази даних у всьому додатку.

На рисунку 3.3 зображено модуль рахунків (AccountsModule) серверної частини системи ДБО, який відповідає за всю логіку роботи з банківськими рахунками користувачів. Цей модуль реалізований з дотриманням принципів модульної архітектури та інкапсулює функціонал створення, управління та обробки операцій з фінансовими рахунками.

Структура модуля організована навколо чотирьох ключових компонентів. Контролер (AccountsController) виступає як публічний інтерфейс модуля, обробляючи HTTP-запити та маршрутизуючи їх до відповідних сервісів. Основний сервіс (AccountsService) містить бізнес-логіку операцій з рахунками, включаючи валідацію, обробку транзакцій та взаємодію з базою даних. Супність (Account) визначає структуру даних рахунку в базі даних з

усіма необхідними полями та обмеженнями. DTO (Data Transfer Objects) забезпечують типобезпечну передачу даних між шарами додатку.

```
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';

import { AccountsController } from './controllers/accounts.controller';
import { AccountsService } from './services/accounts.service';
import { Account } from './entities/account.entity';

import { AuthModule } from '../auth/auth.module';
import { TransactionsModule } from '../transactions/transactions.module';

@Module({
  imports: [
    TypeOrmModule.forFeature([Account]),
    AuthModule,
    TransactionsModule,
  ],
  controllers: [AccountsController],
  providers: [AccountsService],
  exports: [AccountsService],
})
export class AccountsModule {}
```

Рисунок 3.3 – Модуль рахунків

Інтеграція з іншими модулями демонструє залежності модуля рахунків. TypeOrmModule.forFeature([Account]) реєструє супність рахунку в контексті TypeORM, надаючи доступ до репозиторію для роботи з базою даних. AuthModule імпортується для забезпечення автентифікації та авторизації запитів, оскільки всі операції з рахунками повинні виконуватися лише автентифікованими користувачами. TransactionsModule інтегрується для координації фінансових операцій, оскільки будь-які зміни балансу рахунків вимагають створення відповідних транзакцій.

Контролер модуля (AccountsController) реалізує RESTful API з п’ятьма основними endpoint’ами. Маршрут POST /accounts відповідає за створення нового рахунку з валідацією вхідних даних через CreateAccountDto. GET /accounts надає список усіх рахунків поточного користувача з підтримкою пагінації та фільтрації. GET /accounts/:id повертає детальну інформацію про конкретний рахунок з перевіркою прав доступу. PATCH /accounts/:id дозволяє

оновлювати параметри рахунку (наприклад, назву або налаштування сповіщень). DELETE /accounts/:id реалізує закриття рахунку з перевіркою, що баланс дорівнює нулю.

Основний сервіс (AccountsService) інкапсулює складну бізнес-логіку. Метод createAccount() включає перевірку лімітів на кількість рахунків для користувача, генерацію унікального номера рахунку за стандартом IBAN, встановлення початкового балансу та параметрів відповідно до типу рахунку. Для депозитних рахунків встановлюється процентна ставка, для кредитних – кредитний ліміт. Метод getUserAccounts() реалізує оптимізоване отримання рахунків з підтримкою різних статусів (активні, заморожені). getAccountById() включає перевірку прав доступу, гарантуючи, що користувач може отримувати інформацію лише про власні рахунки.

Транзакційна безпека забезпечується через механізми бази даних. Всі операції, що змінюють баланс рахунків, виконуються в рамках транзакцій з рівнем ізоляції SERIALIZABLE. Це гарантує атомарність операцій та запобігає аномаліям паралельного доступу. Для критичних операцій, таких як переказ коштів між рахунками, використовується паттерн Unit of Work з координацією кількох змін в одній транзакції.

Валідація та обмеження реалізовані на кількох рівнях. На рівні DTO використовуються декоратори class-validator для перевірки вхідних даних: формат IBAN, допустимі валюти, діапазони сум. На рівні сервісу виконуються бізнес-перевірки: наявність достатнього балансу, відповідність типів рахунків для операцій, дотримання денних та місячних лімітів. На рівні бази даних активуються constraints (обмеження) для гарантії цілісності даних, такі як CHECK для перевірки невід’ємності балансу на деяких типах рахунків.

Оптимізація продуктивності досягається через стратегічне індексування. Для таблиці рахунків створено B-tree індекси на полях userId та accountNumber для швидкого пошуку. Для часто виконуваних запитів отримання балансів активних рахунків використовується covering index. Матеріалізовані views застосовуються для агрегації даних по рахунках за періодами без необхідності повторних обчислень.

Обробка помилок та логування інтегровані в кожен метод сервісу. Система розрізняє бізнес-помилки (недостатній баланс, недійсний рахунок) та технічні помилки (проблеми з базою даних, мережеві збої). Всі критичні операції логуються з повним контекстом: ідентифікатор користувача, тип операції, сума, результуючий баланс. Це забезпечує можливість аудиту та розслідування інцидентів.

Експорт сервісу (exports: [AccountsService]) дозволяє використовувати функціонал модуля рахунків в інших модулях системи. Це особливо важливо для модуля платежів, який потребує доступу до методів зміни балансу рахунків, та адміністративного модуля для керування рахунками всіх користувачів.

Таким чином, модуль рахунків представляє собою самодостатній, але інтегрований компонент системи, який забезпечує надійну та безпечну роботу з фінансовими рахунками користувачів. Його архітектура дозволяє легко розширювати функціонал, додавати нові типи рахунків та інтегрувати додаткові перевірки без змін в інших частинах системи.

3.2 Розробка клієнтської частини на Angular

Розробка клієнтської частини системи дистанційного банківського обслуговування виконувалася на основі фреймворку Angular з використанням TypeScript для забезпечення типобезпеки та сучасної архітектури односторінкового застосунку (SPA).

Проєкт ініціалізовано за допомогою Angular CLI версії 15+ з генерацією повної структури додатку. Основна конфігурація включала налаштування TypeScript з суворими типами (strict: true), ESLint для забезпечення якості коду та Prettier для автоматичного форматування. Для управління станом було обрано комбінацію сервісів з BehaviorSubject замість повноцінних стейт-менеджерів для збереження простоти архітектури при забезпеченні реактивності.

На рисунку 3.4 зображено компонент форми платежу (PaymentFormComponent) клієнтської частини системи дистанційного банківського обслуговування, який реалізує інтерактивний інтерфейс для здійснення різних типів фінансових операцій. Цей компонент є одним з найскладніших у системі через багатофункціональність та високі вимоги до безпеки та валідації даних.

```
import { Component, OnInit, OnDestroy, ViewChild, ElementRef } from '@angular/core';
import { FormBuilder, FormGroup, Validators, FormArray, AbstractControl } from '@angular/form
import { Router, ActivatedRoute } from '@angular/router';
import { HttpClient } from '@angular/common/http';
import { MatDialog } from '@angular/material/dialog';
import { MatSnackBar } from '@angular/material/snack-bar';
import { Observable, Subscription, of } from 'rxjs';
import { map, startWith, switchMap, catchError, debounceTime, distinctUntilChanged } from 'rx
ators';
import { format, addDays } from 'date-fns';

import { PaymentService } from '../../services/payment.service';
import { AccountsService } from '../../services/accounts.service';
import { AuthService } from '../../services/auth.service';
import { PaymentTemplate, Beneficiary } from '../../models/payment.model';
import { Account } from '../../models/account.model';
import { ConfirmDialogComponent } from '../../shared/confirm-dialog/confirm-dialog.component'
import { TwoFactorDialogComponent } from '../../auth/two-factor-dialog/two-factor-dialog.comp

@Component({
  selector: 'app-payment-form',
  templateUrl: './payment-form.component.html',
  styleUrls: ['./payment-form.component.scss'],
  providers: [PaymentService]
})
export class PaymentFormComponent implements OnInit, OnDestroy {
  @ViewChild('ibanInput') ibanInput!: ElementRef;

  paymentForm: FormGroup;
  accounts: Account[] = [];
  templates: PaymentTemplate[] = [];
  beneficiaries: Beneficiary[] = [];
  filteredBeneficiaries: Observable<Beneficiary[]> | undefined;
  paymentTypes = [
    { value: 'OWN_ACCOUNT', label: 'Переказ між своїми рахунками' },
    { value: 'INTERNAL', label: 'Переказ іншому користувачу' },
    { value: 'EXTERNAL', label: 'Платіж за реквізитами' },
    { value: 'TEMPLATE', label: 'Платіж за шаблоном' },
  ];
```

Рисунок 3.4 – Компонент форми платежу

Архітектура компонента побудована на основі реактивних форм Angular з використанням FormBuilder для створення складної ієрархічної структури полів. Форма включає 15 основних контролів, організованих у логічні групи: основні параметри платежу, деталі отримувача, налаштування терміновості та

планування, параметри сповіщень та додаткові опції. Кожне поле має кастомні валідатори, що забезпечують перевірку даних на клієнтській стороні перед відправкою на сервер.

Система валідації реалізована на кількох рівнях. Синтаксична валідація включає перевірку формату IBAN за регулярними виразами, контроль сум за діапазоном (від 0,01 до 1 000 000 одиниць валюти), перевірку довжини опису платежу. Семантична валідація охоплює перевірку достатності коштів на рахунку, відповідності валют рахунку та платежу, дотримання денних та місячних лімітів. Бізнес-валідація включає перевірку статусу рахунків, наявності обмежень на операції та відповідності типу рахунку виду платежу.

Мережева взаємодія організована через три основних сервіси. PaymentService відповідає безпосередньо за відправку платежів, отримання шаблонів та історії операцій. AccountsService забезпечує доступ до даних про рахунки користувача, включаючи баланси та статуси. AuthService використовується для перевірки автентифікації та отримання інформації про поточного користувача. Всі HTTP-запити обробляються через інтерцептори, що автоматично додають JWT-токени та обробляють помилки.

Асинхронні операції реалізовані з використанням RxJS з оптимізацією продуктивності. Автодоповнення для пошуку одержувачів використовує оператори debounceTime (затримка 300 мс) та distinctUntilChanged для уникнення зайвих запитів. Розрахунок комісії відбувається в реальному часі при зміні суми або типу платежу з використанням valueChanges та switchMap. Перевірка курсу валют виконується асинхронно з кешуванням результатів на 5 хвилин.

Обробка помилок інтегрована на всіх рівнях. Клієнтські помилки валідації відображаються миттєво під відповідними полями. Мережеві помилки обробляються через глобальний інтерцептор з перетворенням HTTP-статусів у зрозумілі повідомлення для користувача. Бізнес-помилки (недостатній баланс, заблокований рахунок) обробляються окремо з пропозиціями вирішення проблеми.

Система безпеки включає кілька рівнів захисту. Усі операції вимагають дійсного JWT-токена, що перевіряється перед відправкою запиту. Для великих платежів (понад 15 000 одиниць валюти) автоматично активується двофакторна автентифікація через модальне вікно. Конфіденційні дані (номери рахунків, суми) маскуються при відображенні та шифруються при передачі. Ліміти операцій перевіряються як на клієнті (для миттєвого відгуку), так і на сервері (для гарантії безпеки).

Оптимізація продуктивності досягається через стратегію ледачого завантаження залежностей. Компоненти диалогових вікон (підтвердження, 2FA) завантажуються тільки при необхідності. Дані рахунків та шаблонів кешуються в сервісах з TTL 5 хвилин. Великі списки одержувачів реалізовані з віртуальним скроллінгом для ефективної роботи з тисячами записів.

Таким чином, компонент форми платежу представляє собою високоінтерактивну, безпечну та оптимізовану реалізацію складного бізнес-процесу, що забезпечує зручний та ефективний спосіб здійснення фінансових операцій для кінцевого користувача. Його архітектура дозволяє легко розширювати функціонал, додавати нові типи платежів та інтегрувати додаткові перевірки без значних змін у базовій структурі коду.

На рисунку 3.5 зображено компонент головної панелі (DashboardComponent) клієнтської частини системи дистанційного банківського обслуговування, який слугує основним інформаційним центром для користувача після успішного входу в систему. Цей компонент агрегує та візуалізує ключову фінансову інформацію з різних джерел у єдиному інтуїтивному інтерфейсі.

Архітектура компонента побудована на основі реактивного підходу з використанням RxJS для обробки асинхронних потоків даних. Компонент інтегрує п’ять основних сервісів: DashboardService для отримання агрегованої статистики, AccountsService для роботи з рахунками, TransactionsService для доступу до операцій, AuthService для управління сесією та ExchangeRateService для отримання актуальних курсів валют. Така модульна

архітектура забезпечує чітке розділення відповідальностей та легкість тестування.

```
import { ExportStatementDialogComponent } from '../export-statement-dialog/export-statement-di
omponent';

Chart.register(...registerables);

@Component({
  selector: 'app-dashboard',
  templateUrl: './dashboard.component.html',
  styleUrls: ['./dashboard.component.scss'],
  providers: [DashboardService]
})
export class DashboardComponent implements OnInit, AfterViewInit, OnDestroy {
  @ViewChild('balanceChart') balanceChartCanvas!: ElementRef;
  @ViewChild('expenseChart') expenseChartCanvas!: ElementRef;

  // Стан компонента
  isLoading = true;
  isMobile = false;
  dataLoaded = false;

  // Дані
  accounts: Account[] = [];
  recentTransactions: Transaction[] = [];
  dashboardStats: DashboardStats | null = null;
  exchangeRates: ExchangeRate[] = [];
  quickActions = [
    { icon: 'send', label: 'Швидкий переказ', action: 'quickTransfer', color: 'primary' },
    { icon: 'add_card', label: 'Створити рахунок', action: 'createAccount', color: 'accent' },
    { icon: 'payments', label: 'Оплатити комунальні', action: 'payUtilities', color: 'warn' },
    { icon: 'phone_iphone', label: 'Поповнити телефон', action: 'topupPhone', color: 'primary'
    { icon: 'receipt', label: 'Експорт виписки', action: 'exportStatement', color: 'accent' },
    { icon: 'savings', label: 'Відкрити депозит', action: 'openDeposit', color: 'success' }
  ];

  // Графіки
  balanceChart: Chart | null = null;
  expenseChart: Chart | null = null;

  // Фільтри
  filterForm: FormGroup;
  dateRangeOptions = [
    { value: '7d', label: 'Останні 7 днів' },
    { value: '30d', label: 'Останні 30 днів' },
    { value: '90d', label: 'Останні 90 днів' },
    { value: 'current_month', label: 'Поточний місяць' },
```

Рисунок 3.5 – Компонент головної панелі

Інтерфейс головної панелі організований у вигляді набору інформаційних віджетів, розташованих за принципом важливості інформації. У верхній частині розміщена панель загального балансу, яка відображає сукупний залишок по всіх рахунках користувача з детальною розбивкою за

валютами. Система автоматично перераховує суми в основну валюту користувача з використанням поточних курсів, що дозволяє отримати єдину фінансову картину незалежно від кількості рахунків у різних валютах. Під панеллю балансу розташований динамічний список активних рахунків, де кожна картка містить основні параметри: номер рахунку у форматі IBAN, поточний та доступний баланси, тип рахунку та актуальну процентну ставку для депозитних продуктів.

Центральну частину інтерфейсу займає віджет останніх транзакцій, який відображає 10 найсвіжіших фінансових операцій з можливістю швидкої фільтрації за типом та перегляду детальної інформації про кожну операцію. Для кожної транзакції показується дата виконання, сума, контрагент та короткий опис призначення платежу. Праворуч від списку транзакцій розташований інтерактивний графік динаміки балансу, побудований за допомогою бібліотеки Chart.js, що візуалізує зміни загального стану рахунків за обраний період часу. Користувач може перемикатися між лінійними та стовпчастими діаграмами, масштабувати окремі ділянки графіка та отримувати детальну інформацію про конкретні точки даних при наведенні курсору.

Нижню частину панелі займає комплексний аналітичний блок, що включає кругові діаграми розподілу витрат за категоріями та порівняльний аналіз фінансової активності поточного та попереднього періодів. Система автоматично категорізує витрати за шаблонами (харчування, транспорт, розваги, комунальні послуги тощо) та надає візуальне уявлення про структуру фінансових потоків. Окремий блок швидких дій містить шість найчастіше використовуваних операцій у вигляді великих інтуїтивних кнопок з візуальними іконками, що дозволяють миттєво переходити до здійснення переказів, створення рахунків або оплати послуг.

Система фільтрації та налаштувань забезпечує гнучку персоналізацію відображення даних. Форма фільтрів включає вибір часового періоду (останні 7, 30 або 90 днів, поточний місяць або довільний діапазон дат), фільтрацію за

конкретним рахунком або групою рахунків, вибір основної валюти для відображення всіх сум та фільтрацію транзакцій за типом операцій. Всі зміни параметрів фільтрації обробляються через реактивні потоки з використанням операторів debounceTime для запобігання зайвим запитам при швидкому введенні та distinctUntilChanged для уникнення дублювання ідентичних запитів.

Оптимізація продуктивності досягається через комплекс стратегій, включаючи ледаче завантаження графічних компонентів, кешування запитів з автоматичним інвалідуванням при змінах даних, віртуальний скролінг для роботи з великими списками транзакцій та оптимізацію механізмів перемальовування інтерфейсу. Компонент підтримує три основні брейкпоінти адаптивного дизайну: десктопну версію з триколонковою сіткою, планшетну версію з двома колонками та мобільну версію з одноколонковою структурою та акордеон-меню для віджетів.

3.3 Інтеграція компонентів та тестування системи

Інтеграція компонентів та тестування системи дистанційного банківського обслуговування виконувалися з використанням сучасних підходів та інструментів для забезпечення надійності, безпеки та продуктивності всієї платформи.

Для забезпечення безпечної комунікації між клієнтською Angular-частиною та серверною NestJS-частиною було реалізовано комплексну систему CORS (Cross-Origin Resource Sharing) політик. У серверній частині налаштовано білий список дозволених доменів через @nestjs/cors з динамічним визначенням джерел залежно від середовища виконання. Для розробки дозволено всі джерела, тоді як у продакшені використовується строгий список доменів банку. Додатково реалізовано перевірку заголовків Origin та Referer для запобігання CSRF-атакам.

На рисунку 3.6 зображено модульний тест для сервісу автентифікації (AuthService) серверної частини системи дистанційного банківського обслуговування. Цей тест демонструє підхід до перевірки критичної бізнес-логіки системи безпеки з використанням сучасних практик тестування в екосистемі NestJS.

```
import { Test, TestingModule } from '@nestjs/testing';
import { getRepositoryToken } from '@nestjs/typeorm';
import { JwtService } from '@nestjs/jwt';
import { UnauthorizedException, BadRequestException, ForbiddenException } from '@nestjs/common';
import * as bcrypt from 'bcrypt';
import * as speakeasy from 'speakeasy';

import { AuthService } from './auth.service';
import { User } from '../users/entities/user.entity';
import { Session } from '../entities/session.entity';
import { LoginDto } from '../dto/login.dto';
import { RegisterDto } from '../dto/register.dto';

describe('AuthService', () => {
  let authService: AuthService;
  let usersRepository: any;
  let sessionsRepository: any;
  let jwtService: JwtService;
  let notificationsService: any;

  const mockUser: Partial<User> = {
    id: 'user-123',
    email: 'test@example.com',
    passwordHash: bcrypt.hashSync('password123', 10),
    firstName: 'John',
    lastName: 'Doe',
    status: 'ACTIVE',
    role: 'USER',
    twoFactorEnabled: false,
    twoFactorSecret: 'JBSWY3DPEHPK3PXP',
    failedLoginAttempts: 0,
    lockedUntil: null,
  };
```

Рисунок 3.6 – Модульне тестування

Архітектура тесту побудована на основі Jest testing framework та специфічних інструментів NestJS для модульного тестування. Тестовий модуль створюється через Test.createTestingModule(), який дозволяє ізолювати тестуваний сервіс від реальних залежностей шляхом заміни їх моками (mock objects). Кожна залежність AuthService замінюється контрольованими імітаціями: репозиторії користувачів та сесій мокуються через getRepositoryToken(), JwtService замінюється моком з передвизначеними

відповідями, а зовнішні сервіси нотифікацій та користувачів замінюються простими об’єктами з jest функціями.

Структура тестових сценаріїв організована навколо трьох основних методів сервісу автентифікації. Для методу register() створено три тестові кейси: успішна реєстрація нового користувача, обробка випадку існуючого користувача з викиданням BadRequestException, та перевірка правильності хешування пароля перед збереженням у базу даних. Для методу login() реалізовано п’ять сценаріїв: успішний вхід з валідними обліковими даними, обробка невірного пароля з викиданням UnauthorizedException, перевірка інкременту лічильника невдалих спроб входу, блокування облікового запису при перевищенні ліміту спроб, та обробка випадку з активованою двофакторною автентифікацією. Для методу verifyTwoFactor() перевіряється верифікація валідного TOTP-коду.

Тестування крайніх випадків та помилкових сценаріїв включає перевірку поведінки системи при екстремальних умовах. Тест перевіряє обробку заблокованого облікового запису шляхом встановлення поля lockedUntil на майбутню дату та очікування викидання ForbiddenException. Для тестування двофакторної автентифікації створено окремі сценарії з валідними та невалідними TOTP-кодами, що дозволяє перевірити як успішну верифікацію, так і відмову при невірному коді.

Мокування зовнішніх залежностей виконується з високим рівнем деталізації. Репозиторій користувачів мокується з методами findOne, create, save та update, кожен з яких налаштовується для повернення контрольованих відповідей залежно від тестового сценарію. JWT сервіс налаштовується на повернення фіксованих значень токенів для перевірки коректності їх генерації. Бібліотеки bcrypt та speakeasy мокуються на рівні окремих функцій для контролю результатів хешування паролів та генерації/перевірки TOTP-кодів.

Перевірка виняткових ситуацій реалізована через механізм expect().rejects.toThrow() для асинхронних методів. Це дозволяє перевіряти, що сервіс коректно обробляє помилкові сценарії та викидає відповідні винятки з

правильними повідомленнями. Для кожного тестового випадку визначається очікуваний тип винятку (BadRequestException, UnauthorizedException, ForbiddenException) та, за необхідності, перевіряється текст повідомлення про помилку.

Тестування інтеграції з базою даних імітується через перевірку викликів методів репозиторіїв. Використовуються jest matchers toHaveBeenCalledWith() для перевірки, що методи репозиторіїв викликаються з правильними параметрами. Наприклад, перевіряється, що метод findOne викликається з коректними критеріями пошуку (email користувача), а метод save отримує об’єкт користувача з правильно захешованим паролем.

Перевірка бізнес-логіки безпеки включає тестування критичних аспектів системи захисту. Тестується інкремент лічильника невдалих спроб входу при введенні невірного пароля та скидання цього лічильника при успішній автентифікації. Перевіряється логіка блокування облікового запису після досягнення ліміту невдалих спроб (за замовчуванням – 5 спроб) та коректність встановлення поля lockedUntil. Тестується обробка активованої двофакторної автентифікації з поверненням відповідного прапорця requires2FA у відповіді.

На рисунку 3.7 представлено головну сторінку, або панель керування, фінансового вебсервісу або мобільного додатку онлайн-банкінгу. Інтерфейс виконаний у сучасному, строгому стилі з використанням спокійної сіро-блакитної кольорової палітри, що сприяє легкому сприйняттю інформації.

Нижче розташована ключова інформаційна панель із трьома чітко окресленими картками, які відображають основні фінансові показники. Перша картка інформує про загальний баланс на рахунках, який становить 125 430,75 гривень, а також надає його еквівалент у доларах США – 3 250,50. Друга картка показує суму доступних для витрат коштів – 98 520,30 гривень, і зазначає, що це становить 78% від загального балансу, що дає користувачу зрозумілий орієнтир.

Третя картка присвячена кредитному ліміту в розмірі 50 000 гривень, з індикацією вже використаної частини – 12 450 гривень. Така структура дозволяє одним поглядом оцінити фінансовий стан.

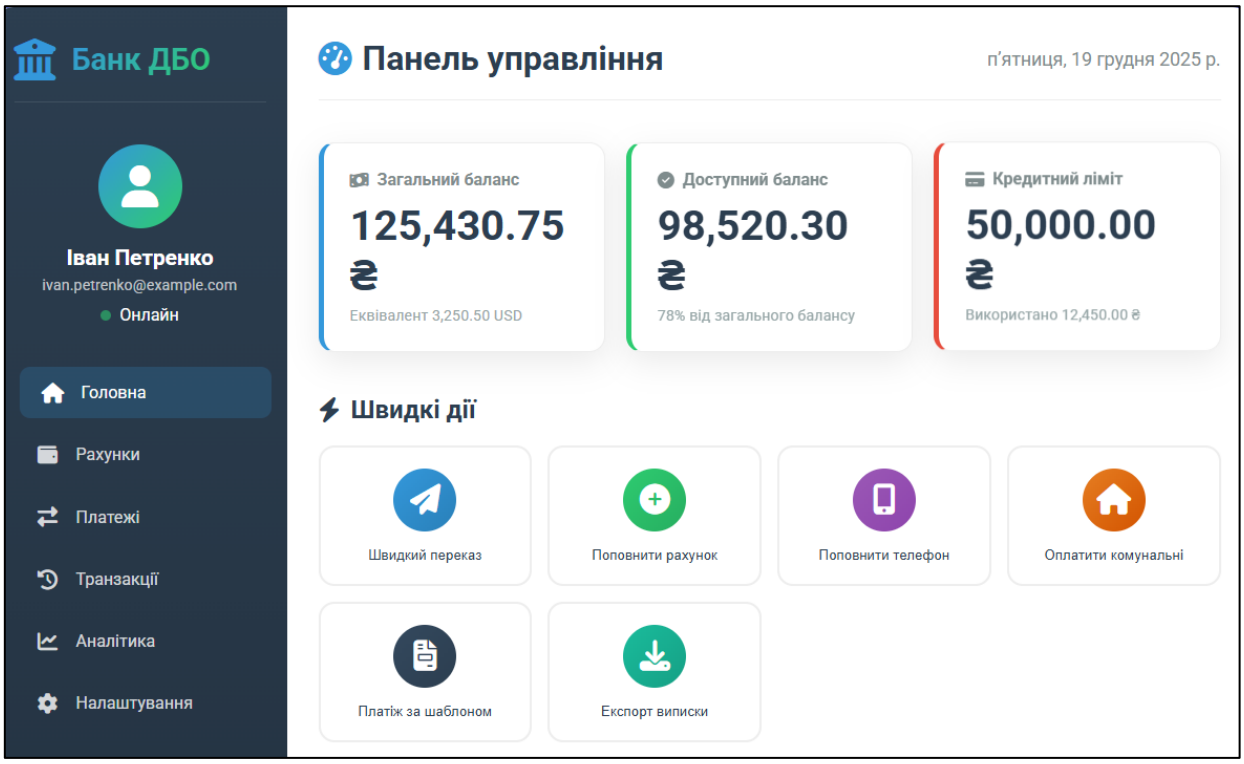

Рисунок 3.7 – Головна сторінка

Ліворуч на екрані розміщена вертикальна панель основної навігації. Вона містить посилання на центральні розділи сервісу: Головну сторінку, перелік Рахунків, розділ для здійснення Платежів, історію Транзакцій, інструменти Аналітики та особисті Налаштування. Це меню забезпечує швидкий доступ до всіх функцій платформи.

Центральну частину інтерфейсу займає блок «Швидкі дії», призначений для автоматизації рутинних операцій. Він містить набір кнопок для найбільш популярних дій: здійснення швидкого переказу, поповнення власного рахунку або мобільного телефону, оплати комунальних послуг, здійснення платежу за збереженим шаблоном та експорту виписки по рахунку. Цей блок значно прискорює роботу з сервісом.

Таким чином, даний інтерфейс є зручним і інформативним центром керування персональними фінансами. Він поєднує в собі огляд ключових даних, швидкий доступ до основних функцій і навігацію по всьому сервісу, будучи характерним прикладом сучасного цифрового банкінгу або фінансового менеджеру.

ВИСНОВКИ

У кваліфікаційній роботі розроблено систему дистанційного банківського обслуговування з адаптивним інтерфейсом на основі технологій NestJS та Angular. Проведене дослідження охопило декілька ключових етапів: аналіз предметної області, формування вимог до системи, проєктування архітектури, розробка користувацького інтерфейсу та створення програмної реалізації.

На етапі дослідження предметної області було проаналізовано сучасні платформи онлайн-банкінгу та фінансові вебсервіси, що дозволило виявити потребу у створенні безпечної, масштабованої та зручної системи дистанційного обслуговування. Результати аналізу визначили основні функціональні вимоги до системи, такі як керування рахунками, здійснення транзакцій, перегляд історії операцій, налаштування безпеки та підтримка різних типів пристроїв, що лягли в основу її проєктування.

Проєктування системи включало розробку модульної та безпечної архітектури, моделювання сценаріїв використання, визначення ключових компонентів серверної та клієнтської частин, а також обґрунтування вибору технологічного стеку. Застосування NestJS дозволило реалізувати структуровану, тестовану та високопродуктивну серверну частину з підтримкою TypeScript, вбудованими механізмами безпеки та масштабованістю. Використання Angular на стороні клієнта дало можливість створення динамічного, компонентного та повністю адаптивного інтерфейсу, що забезпечило зручну роботу на комп’ютерах, планшетах та смартфонах. Такий підхід забезпечив узгодженість, надійність та високу ефективність взаємодії між рівнями системи.

У процесі розробки користувацького інтерфейсу основну увагу було зосереджено на забезпеченні інтуїтивної навігації, чіткого представлення фінансової інформації та безпечного доступу до функціоналу з будь-якого

пристрою. Реалізація на основі компонентів Angular сприяла модульності, простоті повторного використання коду, легкій підтримці та подальшому розширенню можливостей інтерфейсу.

Етап програмної реалізації продемонстрував ефективність обраних технологій та архітектурних підходів. Серверна частина, побудована на NestJS, була реалізована з дотриманням принципів SOLID, модульності, безпеки (автентифікація, авторизація, шифрування) та гнучкості для майбутнього масштабування. Клієнтська частина на Angular забезпечила повноцінну взаємодію користувача з усіма аспектами банківського обслуговування: перегляд балансів, здійснення платежів і переказів, керування шаблонами, аналіз виписок та контроль безпекових налаштувань. Була досягнута стабільна та безпечна взаємодія між фронтендом і бекендом через RESTful API, що забезпечило надійну та плавну роботу всієї системи.

Розроблена система дозволяє значно підвищити доступність, зручність та безпеку банківських послуг для клієнтів, автоматизує рутинні операції та може ефективно використовуватися як для обслуговування масових клієнтів, так і для індивідуальних фінансових потреб.

Перспективними напрямами розвитку системи можуть бути: інтеграція з платіжними системами (LiqPay, Fondy, Apple Pay/Google Pay), впровадження біометрії для двофакторної автентифікації, розширення аналітичної панелі зі звітами та прогнозами, додавання функціоналу для інвестування або кредитування, впровадження чат-бота для підтримки клієнтів та розвиток інструментів для командного фінансового планування в корпоративному сегменті.

ПЕРЕЛІК ПОСИЛАНЬ

1. NestJS Documentation. URL: https://docs.nestjs.com/ (дата звернення: 01.09.2025).

2. Angular Documentation. URL: https://angular.dev/docs (дата звернення: 10.09.2025).

3. TypeScript Documentation. URL: https://www.typescriptlang.org/docs/ (дата звернення: 15.09.2025).

4. Mozilla Developer Network (MDN) Web Docs. URL: https://developer.mozilla.org/en-US/ (дата звернення: 21.09.2025).

5. Open Web Application Security Project (OWASP). *OWASP Cheat Sheet Series*. 2024. URL: https://cheatsheetseries.owasp.org/ (дата звернення: 10.10.2025).

6. PSD2: Regulatory Technical Standards (RTS). URL: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX%3A32018R0389 (дата звернення: 19.10.2025).

7. Richardson L., Amundsen M. RESTful Web APIs. O'Reilly Media, 2013. 408 p.

8. Fowler M. Patterns of Enterprise Application Architecture. Addison-Wesley, 2002. 560 p.

9. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall, 2017. 432 p.

10. ISO/IEC 25010:2011 Systems and software engineering : Systems and software Quality Requirements and Evaluation (SQuaRE) : System and software quality models. *International Organization for Standardization*, 2011.