Міністерство освіти і науки України

Національний технічний університет
«Дніпровська політехніка»

Навчально-науковий
інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)
Кафедра інформаційних технологій та комп'ютерної інженерії
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня бакалавра

здобувача Касумов Давид Уш`ярович
(ПІБ)

академічної групи 126-21-1
(шифр)

спеціальності 126 Інформаційні системи та технології
(код і назва спеціальності)

за освітньо-професійною програмою Інформаційні системи та технології
(офіційна назва)

на тему "Клієнт-серверна система для банківських переказів з підтримкою хешованого зберігання паролів та транзакційних операцій"

(назва за наказом ректора)

Керівники	Прізвище, ініціали	Оцінка за шкалою		Підпис
		рейтинговою	інституційною	
кваліфікаційної роботи	доц. Сушевський Д.В.			
загального розділу	доц. Сушевський Д.В.			
спеціального розділу	доц. Сушевський Д.В.			
Рецензент				
Нормоконтролер	проф. Коротенко Г.М.			

Дніпро
2025

ЗАТВЕРДЖЕНО:
завідувач кафедри
інформаційних технологій
та комп'ютерної інженерії
(повна назва)
Гнатушенко В.В.
(підпис) (прізвище, ініціали)

"25" січня 2025 року

ЗАВДАННЯ
на кваліфікаційну роботу
ступеня бакалавр

здобувача Касумов Д.У. **академічної групи** 126-21-1
(прізвище та ініціали) (шифр)

спеціальності 126 Інформаційні системи та технології

за освітньо-професійною програмою Інформаційні системи та технології
(офіційна назва)

на тему "Клієнт-серверна система для банківських переказів з підтримкою хешованого зберігання паролів та транзакційних операцій"

затверджену наказом ректора НТУ «Дніпровська політехніка» від 05.05.2025 № 336-с

Розділ	Зміст	Термін виконання
Розділ 1. Аналіз стану області рішення завдання	На основі матеріалів виробничих практик, інших науково-технічних джерел показати актуальність завдання, сформулювати мету та задачі виконання кваліфікаційної роботи	10.02.2025
Розділ 2. Вибір та обґрунтування програмно-технічних засобів розробки клієнт-серверних додатків	1 Мова програмування Java та її роль у розробці банківських систем. 2 Захист Java-технологій та їхня еволюція у веб-середовищі. 3 REST Архітектура та веб-сервіси 4 Мобільна операційна система Android 5 Механізми авторизації OAuth 2.0	20.04.2025
Розділ 3. Розробка та тестування програмного забезпечення	1 Опис схеми баз даних для програмного забезпечення 2 Взаємодія програмного забезпечення з базою даних 3 Розробка сервісів (бізнес-логіки) серверу 4 Розробка контролерів 5 Розробка Android-клієнта	31.05.2025

Завдання видано _____ доц. Сушевський Д.В.
(підпис керівника) (прізвище, ініціали)

Дата видачі 25.01.2025

Дата подання до екзаменаційної комісії _____

Прийнято до виконання _____ Касумов Д.У.

РЕФЕРАТ

Пояснювальна записка: 75 с., 25 рис., 1 табл., 1 додаток, 10 джерел.

КЛІЄНТ-СЕРВЕР, КЛІЄНТ-БАНКІВСЬКИЙ ДОДАТОК, БАНКІВСЬКІ ОПЕРАЦІЇ, ТЕХНОЛОГІЯ REST, ANDROID-ДОДАТОК

Об'єкт кваліфікаційної роботи: розробка та функціонування мобільних програмних додатків для взаємодії з розподіленими інформаційними системами.

Предметом кваліфікаційної роботи є методи та технології створення захищеного мобільного банківського додатку на платформі Android з використанням RESTful API для взаємодії з серверною частиною.

Метою роботи є розробка та дослідження принципів побудови захищеного мобільного клієнта для фінансових операцій на базі операційної системи Android, що взаємодіє з серверною системою за допомогою RESTful API, з метою підвищення зручності та доступності банківських послуг для користувачів.

У вступі наведено інформацію про стан проблеми, здійснено аналіз відомих аналогів, визначено об'єкт, предмет, мету та задачі кваліфікаційної роботи.

У першому розділі наведені основні відомості про рівень розвитку інформаційних систем та технологій, зокрема, розглянуто поняття банківської системи та банківських операцій.

У другому розділі наведена проектна складова вирішення завдання. Описані мова програмування Java та її роль у розробці банківських систем, а також розглянута еволюція захисту Java-технологій у веб-середовищі

У третьому розділі подані результати розробки та тестування програмного забезпечення.

У висновках узагальнені результати роботи розробленого програмного забезпечення. Надані пропозиції щодо подальшого розвитку.

Практичне значення кваліфікаційної роботи полягає у створенні функціонального та захищеного мобільного клієнта для фінансових операцій. Розроблене програмне забезпечення може бути впроваджено у діяльність банківських установ для підвищення зручності обслуговування клієнтів.

ABSTRACT

Explanatory note: 75 p., 25 fig., one tab., one app., 10 sources. **Keywords:** CLIENT-SERVER, CLIENT-BANKING APPLICATION, BANKING OPERATIONS, REST TECHNOLOGY, ANDROID-APPLICATION

The object of the qualification work: development and functioning of mobile software applications for interaction with distributed information systems.

The subject of the qualification work: methods and technologies for creating a secure mobile banking application on the Android platform using RESTful API for interaction with the server part.

Purpose of the work: development and research of the principles of building a secure mobile client for financial operations based on the Android operating system, interacting with the server system using RESTful API, to increase the convenience and accessibility of banking services for users.

The **introduction** provides information on the state of the problem, analyzes known analogs, and defines the qualification work's object, subject, purpose, and objectives.

The **first chapter** provides basic information on the development of information systems and technologies; in particular, the concept of the banking system and banking operations is considered.

The **second chapter** presents the project component of the task solution. The Java programming language and its role in developing banking systems are described, and the evolution of Java technology protection in the web environment is considered.

The **third chapter** presents the results of software development and testing.

The **conclusions** summarize the results of the developed software. Proposals for further development are provided.

The **practical significance** of the qualification work lies in creating a functional and secure mobile client for financial operations. The developed software can be implemented in banking institutions' activities to increase customer service convenience.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАК, ОДИНИЦЬ ТА ТЕРМІНІВ

АБС – Автоматизовані банківські системи

API – Application Programming Interface (Інтерфейс програмування застосунків)

ART – Android Runtime

CRUD – Create, Read, Update, Delete (Створити, Прочитати, Оновити, Видалити)

HAL – Hardware Abstraction Layer (Апаратний рівень абстракції)

HTTP – Hypertext Transfer Protocol (Протокол передачі гіпертексту)

HTTPS – Hypertext Transfer Protocol Secure (Захищений протокол передачі гіпертексту)

IDE – Integrated Development Environment (Інтегроване середовище розробки)

JSON – JavaScript Object Notation (Об'єктна нотація JavaScript)

JVM – Java Virtual Machine (Віртуальна машина Java)

KYC – Know Your Customer (Знай свого клієнта)

M2M – Machine-to-Machine (Від машини до машини)

НБУ – Національний банк України

ОС – Операційна система

OHA – Open Handset Alliance (Альянс відкритих мобільних пристроїв)

OWASP – Open Worldwide Application Security Project (Всесвітній проект відкритої безпеки застосунків)

PKCE – Proof Key for Code Exchange (Ключ підтвердження для обміну кодом)

REST – Representational State Transfer (Передача репрезентативного стану)

SDK – Software Development Kit (Комплект для розробки програмного забезпечення)

SQL – Structured Query Language (Мова структурованих запитів)

SSL – Secure Sockets Layer (Рівень захищених сокетів)

TLS – Transport Layer Security (Захист транспортного рівня)

URL – Uniform Resource Locator (Уніфікований покажчик ресурсу)

BBB – World Wide Web

ПВК/ФТ – Протидія відмиванню коштів / Фінансуванню тероризму

ЦБ – Центральний банк

ВСТУП

По суті, система обробки даних — це комплекс, що виконує різноманітні математичні та логічні операції над вхідними даними, трансформуючи їх у потрібний користувачу формат (числовий, текстовий, графічний, звуковий або відео), тим самим роблячи їх інформативними та корисними. До основних функцій такої системи належать: збір інформації, що здійснюється через різноманітні периферійні засоби, такі як модеми, локальні та глобальні комп'ютерні мережі, датчики, а також традиційні клавіатура та монітор; обробка інформації, яка виконується центральним процесором під керуванням відповідного програмного забезпечення для вирішення поставленої проблеми; та представлення даних, що полягає у візуалізації отриманої після обробки інформації у зрозумілому для користувача вигляді [1].

Особливої уваги серед автоматизованих систем потребують ті, що забезпечують грошові операції, зокрема банківські системи. В умовах, коли банківськими послугами користується практично 100% населення, виникає нагальна потреба в автоматизації цих процесів. Це призводить до інтенсивної розробки клієнт-банківських додатків, які надають можливість будь-якій людині, не відстоюючи величезні черги у відділеннях, у будь-який момент скористатися додатком та здійснити необхідні операції. Таке рішення не тільки підвищує ефективність банківського обслуговування, а й значно покращує користувацький досвід, роблячи фінансові послуги доступнішими та зручнішими.

1 АНАЛІЗ СТАНУ ОБЛАСТІ РІШЕННЯ ЗАВДАННЯ

1.1 Поняття банківської системи

У загальному вигляді банківська система — це сукупність різноманітних банків та інших фінансових інститутів, які спільно функціонують для мобілізації коштів і надання клієнтам широкого спектру послуг, від прийому вкладів до надання кредитів. Це не випадкове явище, а інтегрована частина кредитно-грошової системи, що є елементом економічного базису суспільства та розвивається за законами ринкової економіки. Банківська система є внутрішньо організованою, взаємопов'язаною структурою зі спільною метою та завданнями, що існує в будь-якій країні в певний історичний період і є невід'ємною складовою кредитної системи держави.

Для забезпечення ефективного функціонування банківська система має відповідати наступним вимогам:

– повинна бути достатня кількість функціонуючих банків та кредитних установ, які забезпечують конкурентне середовище та покривають потреби всіх секторів економіки;

– система повинна постійно розвиватися, кількісно та якісно змінюватися, адаптуючись до викликів ринку та технологічних інновацій. Це включає впровадження нових фінансових продуктів, цифровізацію послуг та підвищення операційної ефективності.

Відсутність надлишкових елементів: У системі не повинно бути зайвих або неправомірно діючих елементів, таких як банківські установи, що не розпочали діяльність у встановлені строки, працюють без належних ліцензій або створені всупереч чинному законодавству. Це забезпечує стабільність та прозорість фінансового сектора.

Центральне місце в банківській системі посідає Національний банк України (або центральний банк у будь-якій країні), який виступає основним координатором кредитних інститутів та ефективно виконує функції управління грошово-кредитними та фінансовими процесами в економіці. Важливе значення має його

незалежність від уряду, що забезпечується системою призначень функціонерів, взаємовідносинами з урядом, конституційним закріпленням функцій цінової стабільності, обмеженням монетарного фінансування бюджетного дефіциту та самостійністю у використанні монетарних інструментів. Ця незалежність є критично важливою для підтримання макроекономічної стабільності та довіри до національної валюти.

Поряд із центральним банком, в Україні діє розгалужена мережа комерційних банків. Ці установи охоплюють усі сфери національної економіки та зовнішньоекономічні зв'язки, здійснюючи широкий діапазон банківських операцій та фінансових послуг для юридичних і фізичних осіб. Вони не обмежуються лише акумуляцією та розподілом коштів підприємств та організацій, а й активно сприяють накопиченню капіталу та відіграють ключову роль у фінансуванні інвестицій та підтримці економічного зростання. Сучасні комерційні банки постійно впроваджують інноваційні технології, такі як онлайн-банкінг, мобільні додатки та цифрові платежі, щоб відповідати зростаючим потребам клієнтів та викликам глобалізованого фінансового ринку [1].

1.2 Банківські операції

Банківська діяльність, що є основою функціонування фінансової системи, реалізується через виконання різноманітних банківських операцій. Традиційно їх поділяють на три основні категорії: депозитні, кредитні та касово-розрахункові. Водночас, з розвитком фінансових ринків та технологій, спектр послуг банків значно розширився, включаючи так звані нетрадиційні банківські операції, такі як надання в користування індивідуальних сейфів, надання консалтингових послуг, довірчі (трастові) операції, брокерські послуги, лізинг, факторинг, тощо.

Попри таке різноманіття, всі банківські операції можуть бути систематизовані за функціональними ознаками у декілька ключових груп [2]:

– кредитні операції – це фундаментальна функція банку, що полягає у наданні позичальникам (фізичним та юридичним особам) тимчасово вільних

грошових ресурсів на умовах платності, забезпеченості, поворотності, строковості та цільового характеру. Сучасні кредитні продукти включають споживчі кредити, іпотеку, автокредити, кредити для малого та середнього бізнесу (МСБ), корпоративні кредити, овердрафти, відновлювальні кредитні лінії, тощо. В умовах цифровізації, все більшої популярності набувають онлайн-кредити та цифрові платформи для кредитування;

– засновницькі операції – ці операції дозволяють банкам безпосередньо брати участь у створенні нових господарських суб'єктів або реорганізації існуючих шляхом інвестування в їхній статутний капітал. Це сприяє розвитку економіки та створенню нових підприємств;

– розрахунково-касові операції (РКО) – ця група операцій охоплює широкий спектр послуг, пов'язаних з управлінням грошовими потоками клієнтів. До них належать: зберігання коштів на поточних, бюджетних та інших рахунках; прийом платежів на користь клієнтів; здійснення грошових переказів за дорученнями власників коштів (в тому числі через системи інтернет-банкінгу, мобільні додатки та платіжні термінали); прийом готівкових коштів для зарахування на рахунки; та видача готівки з рахунків через операційні каси банку або банкомати. Активний розвиток безготівкових платежів та систем миттєвих розрахунків значно трансформує цю сферу;

– інвестиційні операції – включають вкладання фінансових ресурсів банку у різноманітні корпоративні та державні цінні папери, інші боргові зобов'язання, а також участь у капіталах інших компаній. Це може відбуватися шляхом придбання цінних паперів на фондових біржах та організованому позабіржовому ринку, або шляхом прямих інвестицій. Метою є отримання доходу у вигляді відсотків, дивідендів або приросту капіталу;

– депозитні операції – це операції із залучення тимчасово вільних коштів юридичних та фізичних осіб на різні типи рахунків: поточні, ощадні, строкові вклади (депозити), бюджетні, кодовані, тощо. Депозити є основним джерелом формування ресурсної бази банку для подальшого кредитування та інвестиційної діяльності. Сучасні депозитні продукти часто пропонують гнучкі умови та

можливості онлайн-відкриття;

– міжбанківські операції – ці операції пов'язані з недепозитним залученням та розміщенням ресурсів на міжбанківському ринку. Це може включати отримання кредитів від центрального банку (наприклад, рефінансування), позик від інших комерційних банків, розміщення депозитів в інших банках або центральному банку. Міжбанківський ринок є ключовим для підтримання ліквідності банківської системи;

– емісійні операції – операції банку, пов'язані з формуванням власного капіталу та недепозитного залучення фінансових ресурсів. Здійснюються через випуск банківських акцій, облігацій, векселів та інших боргових зобов'язань. Це дозволяє банку збільшити свою капітальну базу та залучити довгострокове фінансування;

– комісійні операції – це різноманітні послуги, за які банки отримують дохід у вигляді комісії. До них належать консультаційні, інформаційні, аудиторські, облікові, реєстраторські, кастодіальні (зберігання цінних паперів), трастові послуги, послуги андеррайтингу, гарантії та акредитиви. Розширення цього сегмента послуг є одним з основних трендів розвитку банківського бізнесу;

– посередницькі операції – послуги банків з розміщення цінних паперів емітентів на первинному фондовому ринку, брокерські та дилерські послуги за операціями з фондовими цінностями, іноземною валютою, та інші види операцій на грошовому ринку, де банки виступають як посередники, поєднуючи інтереси різних сторін фінансових угод.

За економічною сутністю, всі операції комерційних банків можуть бути класифіковані як:

– активні операції – це операції з розміщення банками власного капіталу та залучених ресурсів з метою отримання доходу, забезпечення діяльності та підтримання необхідного рівня ліквідності. До активних операцій належать кредитні, засновницькі, інвестиційні та міжбанківські операції (у частині наданих позик та розміщених депозитів). Також до активів відносяться операції з придбання необоротних активів (приміщень, обладнання, технічних засобів тощо) та

формування касових залишків і залишків коштів на кореспондентських рахунках банків;

– пасивні операції – ці операції пов'язані з формуванням власного капіталу та ресурсної бази банку. Вони забезпечують проведення активних операцій з метою досягнення запланованих показників дохідності та є запорукою ліквідності й платоспроможності. До пасивних операцій належать емісійні (випуск цінних паперів банку), депозитні (залучення вкладів) та міжбанківські операції (у частині отриманих позик та залучених депозитів);

– комісійно-посередницькі послуги – це операції консультаційного характеру, що виконуються банками завдяки їх високій інформативності, глибоким професійним знанням персоналу та володінню новітніми технологіями. Також до цієї групи належать операції, де банки діють за рахунок та в інтересах клієнтів. Ці операції приносять банкам дохід, але не потребують додаткового залучення та використання наявних ресурсів.

Важливо зазначити, що багато операцій можуть виступати в ролі як активних, так і пасивних, залежно від перспективи. Наприклад, залучення коштів на депозит є пасивною операцією, оскільки збільшує ресурсну базу банку. Водночас, отримані на депозит кошти розміщуються у вигляді залишку на кореспондентському рахунку в центральному банку або в операційній касі банку, що, безумовно, є активною операцією для банку. Це підкреслює складність та взаємопов'язаність банківської діяльності.

1.3 Технологія клієнт/сервер

Сучасний розвиток інформаційних систем значною мірою зумовлений потребою у підвищенні швидкості та ефективності доступу кінцевого користувача до необхідної інформації. Сьогодні більшість інформаційних систем, особливо у фінансових установах, базуються на концепції відкритих систем, невід'ємною складовою частиною яких є технологія клієнт/сервер.

Історично, централізована обробка даних еволюціонувала до розподіленої

обробки. Спочатку це проявлялося в режимі розподілу часу на центральному комп'ютері, а згодом процедури обробки та доступу до даних почали розподілятися між робочими станціями, підключеними до центрального комп'ютера. Таким чином, початково концепція клієнт/сервер означала використання персональних комп'ютерів (клієнтів), об'єднаних локальними чи глобальними мережами з центральним комп'ютером або сервером (рис.1.1) [3].

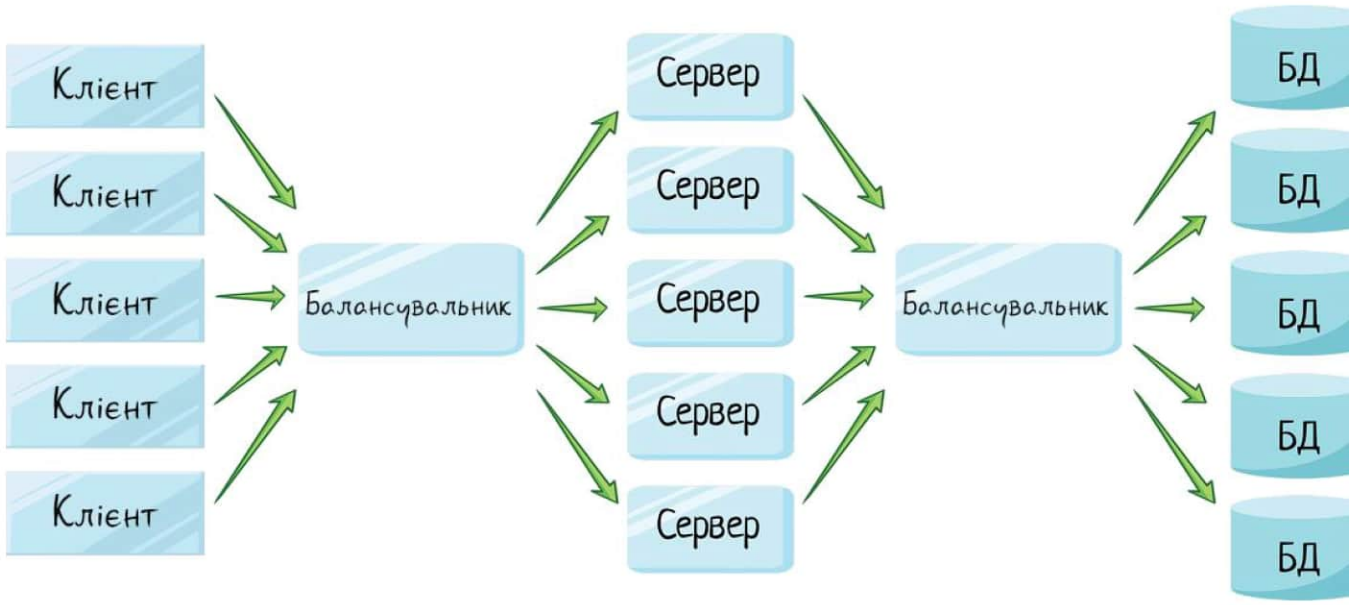


Рисунок 1.1 – Модель технології клієнт/сервер

В основі технології клієнт/сервер лежить чіткий розподіл ролей та функцій між двома ключовими компонентами: клієнтом і сервером.

Клієнт – це програмно-апаратний комплекс (зазвичай робоча станція користувача або мобільний пристрій), що виконує функції взаємодії з користувачем, обробляє запити, здійснює необхідні обчислення та забезпечує підключення до віддалених ресурсів, таких як бази даних та засоби їх обробки.

З технічної точки зору, як клієнт можуть бути використані стандартні персональні комп'ютери, ноутбуки, планшети або смартфони. До них, як правило, не пред'являються специфічні вимоги до обчислювальної потужності, оскільки основна логіка зберігання та обробки даних реалізується на стороні сервера. Основними функціями клієнтського пристрою є: забезпечення введення-виведення інформації

для користувача, локальне збереження особистих даних (кеш, тимчасові файли), використання периферійного устаткування (принтери, сканери) та управління взаємодією з віддаленими серверами через мережу.

Програмне забезпечення клієнта може бути як "тонким" (веб-браузер, що лише відображає дані з сервера), так і "товстим" (настільний додаток, що виконує значну частину логіки локально). Сучасні тенденції, такі як хмарні обчислення та веб-технології, сприяють розвитку "тонких" клієнтів, що значно спрощує розгортання та оновлення програмного забезпечення. Клієнтська програма може запускати процеси на віддалених серверах, дозволяючи не лише зберігати модулі прикладних програм на різних комп'ютерах, але й виконувати їх на різних машинах, оптимізуючи розподіл навантаження. Завантаження операційної системи та програмного забезпечення для "тонких" клієнтів може здійснюватися віддалено з сервера через мережу, зменшуючи вимоги до локального сховища. Взаємодія клієнта з сервером відбувається за допомогою стандартних мережевих протоколів, таких як TCP/IP, HTTP/HTTPS, WebSockets, що забезпечують надійний та захищений обмін даними [4].

Сервер – це потужний комп'ютер (одно- або багатопроцесорний), призначений для зберігання, обробки та надання доступу до ресурсів та послуг для клієнтів. Він характеризується значним обсягом пам'яті, високою обчислювальною потужністю, розвиненими засобами телекомунікації та управління устаткуванням.

У сучасних мережах передачі даних функціонує велика кількість серверів, кожен з яких спеціалізується на певних функціях: файл-сервери (для зберігання файлів), сервери баз даних (для управління базами даних), комунікаційні сервери (для електронної пошти, обміну повідомленнями), веб-сервери (для розміщення веб-сайтів), сервери додатків (для виконання бізнес-логіки) тощо. Для забезпечення високої продуктивності та надійності, як правило, використовуються потужні багатозадачні операційні системи, такі як різні дистрибутиви Linux/Unix (наприклад, Ubuntu Server, Red Hat Enterprise Linux), Microsoft Windows Server, VMware ESXi (для віртуалізації).

Взаємодія сервера з клієнтом відбувається за допомогою механізму

транзакцій. Транзакція — це логічна одиниця роботи, що об'єднує послідовність операцій, які мають бути виконані атомарно: або всі успішно, або жодна. Сервер відстежує запити клієнта, ставить їх у чергу, виконує згідно з розкладом, сповіщає клієнту про виконання та надає результат обробки. Важливою характеристикою транзакцій є відповідність принципам ACID (Atomicity, Consistency, Isolation, Durability), що гарантує надійність та цілісність даних, особливо критично важливих у банківських системах.

Технологія клієнт/сервер забезпечує низку значних переваг [5]:

– корпоративне управління ресурсами дозволяє централізовано управляти всіма ресурсами інформаційної системи, доступними через мережі передачі даних, включаючи дані, додатки, користувачів та безпеку;

– розподіл функцій та доступу чітко розділяє доступ до даних та програм між робочими станціями та серверами, що з'єднані мережами передачі даних. Це підвищує безпеку, гнучкість та ефективність системи;

– організація програмного забезпечення на основі відкритих систем сприяє використанню стандартизованих протоколів та інтерфейсів, що забезпечує сумісність компонентів від різних виробників та полегшує масштабування та інтеграцію;

– системи клієнт/сервер легко масштабуються шляхом додавання нових клієнтських робочих станцій або підвищення потужності серверів, а також впровадження розподілених баз даних та кластерних рішень;

– централізоване зберігання даних на серверах та можливість резервного копіювання значно підвищують надійність. Реалізація комплексних механізмів автентифікації, авторизації та шифрування забезпечує високий рівень безпеки даних.

Таким чином, за технологією клієнт/сервер користувач не керує безпосередньо ходом виконання складної проблеми. Натомість, система автоматично здійснює її вирішення, оптимальним чином використовуючи ресурси технічних засобів, баз даних та засобів телекомунікації. Це означає, що одна задача може вирішуватись багатьма робочими станціями та серверами, які є територіально розподіленими –

розташованими в різних частинах міста, країни чи навіть світу, забезпечуючи глобальний доступ та співпрацю.

1.4 Особливості використання баз даних в мережах

У сучасному світі фінансові установи оперують величезними обсягами інформації, доступ до якої є критично важливим не лише для внутрішніх співробітників, а й для широкого кола зовнішніх зацікавлених сторін. Це можуть бути біржі, яким потрібна актуальна інформація про курси валют та швидкість оформлення угод; підприємства, для яких є життєво важливим моніторинг надходження коштів на поточні рахунки та статус розрахунків; або страхові компанії, що взаємодіють з банками у процесі врегулювання страхових випадків. Ефективне вирішення цих завдань сьогодні неможливе без використання локальних та глобальних мереж передачі даних, а також надання відповідного доступу до баз даних (БД) різним категоріям користувачів [6, 7].

Ці проблеми успішно вирішуються за допомогою телекомунікаційних технологій та архітектури клієнт/сервер. Зокрема, ключову роль у забезпеченні віддаленого доступу до баз даних відіграють SQL-технології (Structured Query Language), що є стандартом для взаємодії з реляційними базами даних.

Центром будь-якої мережевої бази даних є її прикладна частина. Вона складається з сервера БД, самих джерел даних (файлів БД) та мережевого програмного забезпечення, яке забезпечує підключення клієнтів до мережі. Сучасні та широко поширені сервери баз даних включають такі потужні системи, як Oracle Database, Microsoft SQL Server, PostgreSQL, MySQL, IBM Db2, Apache Cassandra (NoSQL) та інші. Сервер БД адмініструється фахівцями (адміністраторами баз даних), які визначають права доступу для клієнтів до таблиць та даних відповідно до посадових обов'язків та статусу кожного користувача [4].

Інтерфейсна частина – це програмне забезпечення, що функціонує на робочому місці користувача. Вона є компонентом автоматизованої системи, розробленим для вирішення конкретних завдань цього користувача. Це може бути

спеціалізоване автоматизоване робоче місце (АРМ) або комплекс програм, створений за допомогою різних мов програмування та фреймворків, таких як C++, Java, Python, .NET (C#), JavaScript (з фреймворками типу React, Angular, Vue.js), Delphi (Object Pascal) та інші. Прикладна частина зазвичай розміщується на сервері разом з даними БД, і її користувачами є адміністратори БД, розробники автоматизованих систем, системні аналітики та системні адміністратори. Інтерфейсна частина розташовується на комп'ютерах кінцевих користувачів – операторів введення даних, бухгалтерів, фінансистів, економістів, менеджерів тощо (рис.1.2).

Рисунок 1.2 – Схема взаємодії прикладної та інтерфейсної частин

База даних може бути локальною, коли користувач підключається до неї безпосередньо на одному пристрої, або віддаленою – у випадку підключення через мережу, незалежно від фізичної відстані. Підключення до віддаленої бази даних

здійснюється за допомогою мережевого забезпечення та відповідних протоколів передачі даних.

Однією з ключових технологій, що забезпечує уніфікований доступ до різнорідних баз даних, є ODBC (Open Database Connectivity) – відкритий інтерфейс доступу до бази даних. Це стандарт, що дозволяє програмам, розробленим на різних мовах, взаємодіяти з різними СУБД через єдиний інтерфейс. Драйвер ODBC виступає як посередник, приймаючи запити від клієнтської програми, перетворюючи їх у формат, зрозумілий конкретній СУБД, та повертаючи результат виконання. На сьогодні ODBC залишається важливим стандартом, який підтримується багатьма виробниками програмного інструментарію (наприклад, Microsoft Visual Studio, PowerBuilder, LibreOffice/OpenOffice Base) та виробниками СУБД, що забезпечує гнучкість та сумісність.

Крім ODBC, виробники СУБД пропонують власні потужні засоби підключення до віддалених баз даних. Наприклад, корпорація Oracle пропонує для підключення до віддалених баз даних продукт Oracle Net (раніше Net8), який може використовуватись з різними мережевими протоколами, такими як TCP/IP, Named Pipes, SPX/IPX (хоча останній вже застарілий), та працює під управлінням поширених операційних систем.

Особливо актуальним сьогодні є доступ до віддалених баз даних за допомогою Web-технологій. У цьому сценарії всі запити до бази даних направляються через Web-сервер. Кінцевий користувач ініціює доступ до віддаленої БД за допомогою Web-браузера, який забезпечує зв'язок за заданою IP-адресою або доменним ім'ям з відповідним Web-сервером.

Процес взаємодії виглядає наступним чином: користувач через Web-браузер надсилає запит до Web-сервера. Web-сервер, отримавши запит, перевіряє ім'я користувача та пароль (якщо потрібна аутентифікація). Після успішної аутентифікації Web-сервер формує запит до системи управління базами даних (СУБД). СУБД, можливо, також запитує реквізити доступу до БД. Сервер БД обробляє запит і повертає результати Web-серверу. Web-сервер форматує отримані дані (наприклад, у HTML) та відображає їх у вікні Web-браузера користувача [8].

Використання Web-технології для доступу до баз даних вимагає надійного захисту інформаційних потоків. Це досягається шляхом застосування брандмауерів (міжмережевих екранів) – апаратно-програмних систем, що контролюють мережевий трафік та захищають від несанкціонованого доступу до сервера. Крім того, використовуються протоколи шифрування, такі як HTTPS (HTTP Secure), що забезпечує конфіденційність та цілісність даних під час передачі. Сучасні системи також інтегрують механізми WAF (Web Application Firewall) для захисту від атак на рівні веб-додатків, систем виявлення та запобігання вторгнень (IDS/IPS) та інші комплексні рішення для кібербезпеки (рис.1.3).

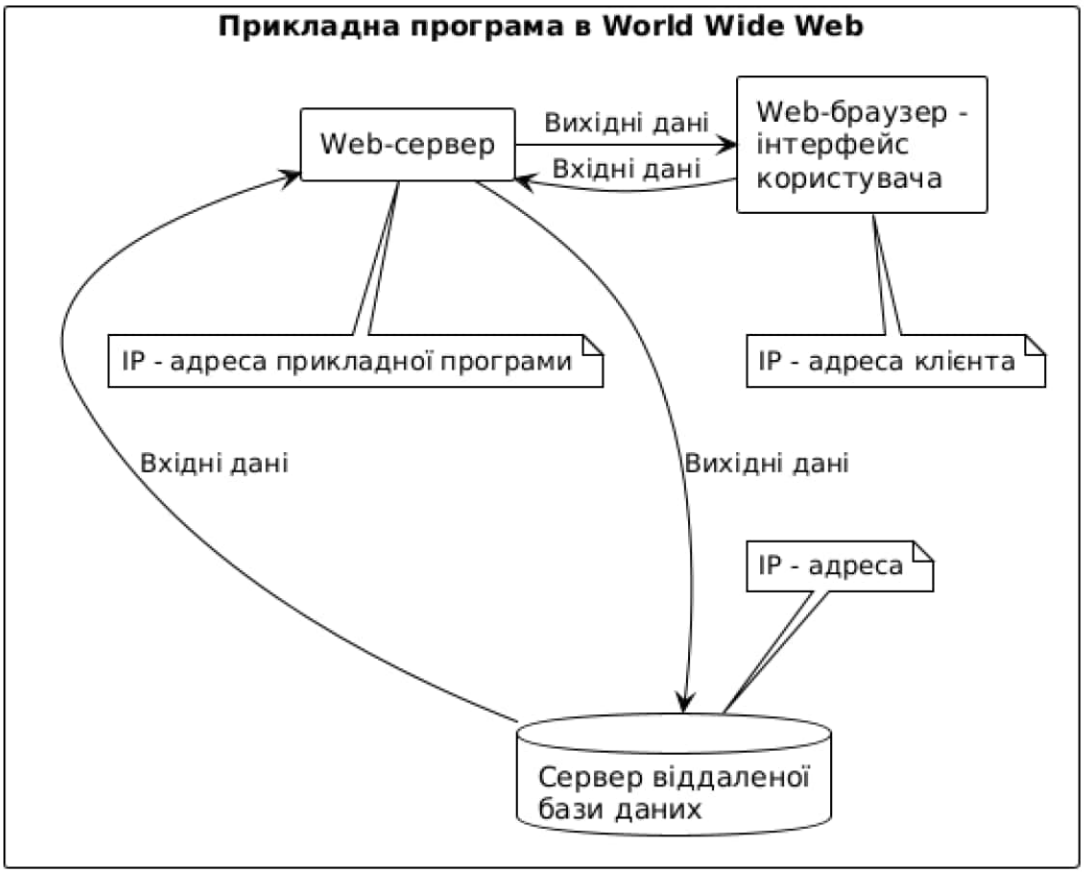


Рисунок 1.3 – Доступ до віддаленої бази даних з допомогою Web - технології

1.5 Транзакції

У контексті систем керування базами даних (СКБД) та розподілених

обчислень, транзакція (від англ. transaction) визначається як дискретна, логічна одиниця роботи, що складається з однієї або кількох послідовних операцій з даними. Ключовою характеристикою транзакції є її властивість виконуватися повністю і успішно (коміт) або ж бути повністю скасованою (відкат), не залишаючи проміжних результатів. Це гарантує цілісність даних та їхню коректність незалежно від паралельних операцій чи системних збоїв. Обробка транзакцій здійснюється транзакційними системами, які підтримують історію їх виконання для забезпечення відстежуваності та відновлення.

Розрізняють декілька типів транзакцій [7]:

– послідовні (локальні) транзакції виконуються в межах однієї транзакційної системи і є найбільш поширеним типом у монолітних додатках;

– паралельні транзакції запускаються одночасно та можуть взаємодіяти з одними й тими ж даними, вимагаючи механізмів управління паралелізмом для забезпечення ізоляції;

– розподілені транзакції охоплюють більше однієї транзакційної системи або вузла мережі, що вимагає значно складнішої логіки координації та фіксації. Прикладом таких протоколів є двофазний протокол фіксації (two-phase commit – 2PC), який забезпечує атомарність операцій у розподіленому середовищі;

– автономні транзакції (під транзакції) являють собою окремі, повністю незалежні одиниці роботи, що виконуються в межах батьківської транзакції, але мають власну атомарність та можуть бути відкачені незалежно.

Розглянемо класичний приклад банківської операції з переказу 10 грошових одиниць з рахунку №5 на рахунок №7. Ця операція є логічною одиницею роботи та реалізується як транзакція за допомогою наступної послідовності дій:

Початок транзакції:

– зчитування поточного балансу рахунку №5;

– зменшення балансу рахунку №5 на 10 одиниць;

– запис оновленого балансу рахунку №5;

– зчитування поточного балансу рахунку №7;

– збільшення балансу рахунку №7 на 10 одиниць;

– запис оновленого балансу рахунку №7.

– завершення транзакції (коміт/відкат):

Якщо всі проміжні операції пройшли успішно, транзакція фіксується (коміт), і зміни стають постійними.

У разі будь-якого збою на будь-якому етапі (наприклад, недостатньо коштів на рахунку №5, збій мережі або системи), транзакція повністю відкочується (відкат), скасовуючи всі виконані зміни.

Важливість транзакцій полягає в гарантуванні цілісності даних. У вищенаведеному прикладі, якщо транзакція буде перервана між списанням з одного рахунку та зарахуванням на інший без механізму відкату, це призведе до некоректного стану системи, коли кошти будуть втрачені без зарахування.

Для забезпечення надійності та цілісності даних, транзакції традиційно відповідають чотирьом ключовим властивостям, відомим як акронім ACID: Atomicity (Атомарність), Consistency (Узгодженість), Isolation (Ізольованість) та Durability (Надійність).

Атомарність гарантує, що транзакція є неподільною одиницею роботи. Це означає, що або всі операції в складі транзакції будуть успішно виконані та зафіксовані в базі даних, або жодна з них не буде зафіксована. У випадку будь-якого збою (системного, апаратного, програмного) під час виконання транзакції, система автоматично виконує "відкат" (rollback), повертаючи базу даних до стану, який передував початку транзакції. Це забезпечує, що база даних ніколи не залишається в частково оновленому, некоректному стані.

Властивість узгодженості вимагає, щоб кожна транзакція переводила базу даних з одного узгодженого стану в інший узгоджений стан. Це означає, що транзакція не може порушувати встановлені правила цілісності даних, бізнес-правила або обмеження, визначені для бази даних (наприклад, унікальні ключі, зовнішні ключі, перевірки доменів). Якщо транзакція намагається виконати операцію, яка порушує ці правила, вона буде відкочена. Важливо розрізняти узгодженість від цілісності даних: цілісність стосується дотримання визначених

обмежень (наприклад, referential integrity), тоді як узгодженість є ширшим поняттям, що включає дотримання всіх бізнес-правил та інваріантів системи. Під час виконання транзакції може спостерігатися тимчасовий неузгоджений стан, проте завдяки властивості ізольованості, цей проміжний стан не є видимим для інших паралельних транзакцій.

Ізольованість гарантує, що виконання паралельних транзакцій не впливає на результати одна одної, створюючи ілюзію послідовного виконання. Кожна транзакція виконується таким чином, ніби вона є єдиною, що працює з базою даних. Це запобігає виникненню аномалій паралельного виконання, таких як "брудне читання" (dirty reads), "неповторюване читання" (non-repeatable reads) або "фантомні читання" (phantom reads). Рівні ізоляції (наприклад, Read Uncommitted, Read Committed, Repeatable Read, Serializable) визначають ступінь захисту від цих аномалій, впливаючи на продуктивність системи. У сучасних розподілених системах досягнення сильної ізоляції може бути складним і часто компромітується на користь доступності або продуктивності (згідно з CAP-теоремою), що призводить до появи концепцій, таких як "базова узгодженість" (eventual consistency).

Надійність гарантує, що зміни, внесені успішно зафіксованою (закоміченою) транзакцією, є постійними та зберігаються в базі даних навіть у випадку подальших збоїв системи (наприклад, відключення електроенергії, апаратних поломок або програмних помилок). Для забезпечення надійності СКБД використовують механізми ведення журналів транзакцій (write-ahead logging - WAL), резервного копіювання та реплікації даних. Це означає, що після підтвердження користувачеві про завершення транзакції, він може бути впевнений, що його зміни не будуть втрачені.

В ідеалі, для забезпечення повної коректності, паралельні транзакції повинні виконуватися таким чином, щоб створювати ілюзію послідовного виконання, тобто кожна транзакція мала б повну ізоляцію від інших. Проте, з міркувань оптимізації продуктивності та для підтримки певних спеціалізованих задач, системи керування базами даних (СКБД) зазвичай надають різні рівні ізоляції транзакцій. Ці рівні визначають ступінь, до якого одна транзакція повинна бути ізольована від змін, що

вносяться іншими паралельно виконуваними транзакціями. Компроміс між ізоляцією та продуктивністю є фундаментальним аспектом проектування транзакційних систем.

Рівні ізоляції, визначені стандартом SQL (ANSI/ISO SQL), описуються в порядку збільшення ізольованості та, відповідно, надійності роботи з даними, проте також зростають і вимоги до системних ресурсів.

0. Читання непідтверджених даних (Read Uncommitted / Dirty Read). На цьому рівні ізоляції транзакція може читати дані, які були змінені іншими транзакціями, але ще не були зафіксовані (комітовані). Це дозволяє "брудне читання" (dirty read), оскільки існує ризик, що ці незафіксовані зміни можуть бути відкочені (rollback), роблячи прочитані дані недійсними. Це найнижчий рівень ізоляції, який максимізує паралелізм, але може призвести до некоректних результатів аналізу. Аномалії, такі як "втрачені оновлення" (lost updates), "неповторювані читання" (non-repeatable reads) та "фантомні читання" (phantom reads), є можливими.

1. Читання підтверджених даних (Read Committed). Цей рівень є найбільш поширеним і часто використовується за замовчуванням у багатьох сучасних СКБД (наприклад, PostgreSQL, SQL Server, Oracle). Транзакція може читати лише ті зміни, які були зафіксовані іншими транзакціями. Це усуває проблему "брудного читання" (dirty reads). Однак, "неповторювані читання" (non-repeatable reads) все ще можливі: якщо транзакція зчитує рядок двічі, і між цими читаннями інша транзакція змінює та фіксує значення цього рядка, перша транзакція побачить різні значення. Також можливі "фантомні читання" (phantom reads).

2. Повторюване читання (Repeatable Read). На цьому рівні ізоляції транзакція гарантує, що будь-які рядки, прочитані нею, залишаться незмінними протягом її виконання. Якщо транзакція зчитує рядок двічі, вона завжди побачить те саме значення, навіть якщо інша транзакція намагалася його змінити та зафіксувати. Це досягається шляхом блокування прочитаних рядків або використання механізмів багатоверсійного керування паралелізмом (MVCC – Multi-Version Concurrency Control), що є актуальним для сучасних СКБД. Рівень Repeatable Read усуває аномалії "неповторюваного читання", проте "фантомні читання" все ще можуть

виникати, коли інші транзакції додають нові рядки, що відповідають умовам запиту поточної транзакції.

3. Серіалізований (Serializable). Це найвищий рівень ізоляції, який забезпечує повну серіалізованість виконання транзакцій. Результат паралельного виконання декількох транзакцій на цьому рівні є логічно еквівалентним результату будь-якого послідовного виконання цих транзакцій. Всі аномалії паралельного виконання (брудні читання, неповторювані читання, фантомні читання) повністю усуваються. Досягнення Serializable ізоляції зазвичай вимагає використання агресивних блокувань або складних алгоритмів керування паралелізмом (наприклад, оптимістичні блокування, серіалізуючі snapshot ізоляції), що може значно знизити продуктивність та паралелізм системи.

Чим вищий рівень ізоляції, тим більші системні ресурси (наприклад, процесорний час, пам'ять, дисковий ввід/вивід) необхідні для його забезпечення. Це пов'язано з тим, що для підтримки вищих рівнів ізоляції СКБД повинні застосовувати більше блокувань, використовувати складніші механізми управління версіями даних або виконувати додаткові перевірки. Відповідно, підвищення ізольованості може призводити до зниження швидкості виконання паралельних транзакцій, що є "платою" за підвищення надійності та коректності даних.

У сучасних СКБД рівень ізоляції транзакцій може бути налаштований як глобально для всієї системи, так і локально для окремої транзакції. Хоча Read Committed є типовим рівнем за замовчуванням, Read Uncommitted може використовуватись для аналітичних запитів, де невелика похибка допустима, а висока швидкість читання є пріоритетом. Рівні Repeatable Read та Serializable застосовуються у випадках, коли до коректності та цілісності даних пред'являються найвищі вимоги, незважаючи на потенційне зниження продуктивності.

1.6 Стан розвитку автоматизованих банківських систем

Еволюція автоматизованих банківських систем (АБС) є відображенням загального прогресу в інформаційних технологіях та специфічних потреб

фінансового сектору. Класифікація АБС за "поколіннями" є до певної міри умовною, проте дозволяє систематизувати підходи до їх архітектури, технологічної платформи та базових структурних елементів.

Процес розробки програмного забезпечення АБС завжди був складним і динамічним, успіх якого значною мірою залежить від обґрунтованого вибору технологічної архітектури. Історично виділяють дві основні архітектурні парадигми:

Файл-серверна архітектура. У цій моделі централізована база даних зберігається на виділеному сервері. Вся обробка інформації, включаючи логіку додатків, виконується на сервері, тоді як клієнтські комп'ютери (робочі станції) функціонують переважно як інтерфейси для введення/виведення даних. Ця архітектура була поширеною на ранніх етапах розвитку АБС.

Клієнт-серверна архітектура. За цією технологією база даних також зберігається на сервері, проте значна частина прикладної логіки та обробки даних переноситься на робочі станції користувачів. Ця архітектура може бути реалізована як дворівнева (безпосередня взаємодія клієнтів з сервером бази даних) або трирівнева (з використанням проміжного "сервера додатків" для розміщення бізнес-логіки та обробки запитів від клієнтів перед зверненням до бази даних). Трирівнева архітектура дозволяє зменшити навантаження на клієнтські машини, забезпечити кращу масштабованість та гнучкість, а також централізувати оновлення та управління бізнес-логікою.

Вибір базового елементу системи також суттєво змінювався. Якщо перші покоління АБС оперували переважно бухгалтерською проводкою як основною структурною одиницею, то сучасні системи переходять до більш абстрактних і комплексних понять. У системах четвертого покоління документу надається центральне значення, а його проведення є невід'ємною частиною. П'яте та шосте покоління АБС оперують угодою як базовим елементом, яка може інтегрувати в собі ряд документів та проводок, відображаючи складні бізнес-процеси.

Аналізуючи етапи розвитку АБС, можна виділити шість умовних поколінь.

Перше покоління (кінець 1980-х – початок 1990-х): Характеризувалося використанням автономних робочих станцій (на базі ПК AT-286, AT-386 під

управлінням MS-DOS), що працювали з локальними базами даних (наприклад, FoxPro, Clipper, Clarion). Обмін інформацією між центральним комп'ютером банку та окремими АРМами здійснювався переважно за допомогою фізичного перенесення файлів (наприклад, на гнучких магнітних дисках). Системи цього покоління мали суттєві недоліки: відсутність механізмів захисту інформації (санкціонованого доступу, розподілу повноважень), складнощі з консолідацією даних та відсутність механізмів транзакцій, що унеможливлювало підтримку цілісності даних. Типовим прикладом в Україні була програма ОДБ "УНІТІБАРС".

Друге покоління (початок 1990-х): Ознаменувалося появою локальних мереж (наприклад, Novell NetWare), що дозволило об'єднувати робочі станції в межах однієї установи за файл-серверною технологією. Це покращило консолідацію даних на рівні філії, проте між філіями обмін все ще відбувався за консолідованою схемою (наприклад, через електронну пошту). Захист інформації забезпечувався переважно засобами мережевих ОС, а проблема цілісності даних залишалася частково вирішеною через відсутність повноцінного механізму транзакцій на рівні СКБД.

Третє покоління (середина 1990-х): Є перехідним від файл-серверної до клієнт-серверної архітектури. Важливим кроком стала інтеграція менеджерів записів з підтримкою транзакцій на рівні ядра (наприклад, Btrieve в АБС "RS-BANK" фірми R-Style). Це забезпечило значне підвищення надійності та безпеки обробки банківської інформації, хоча підтримка цілісності даних все ще була відносно складною. Технічна база включала ПК AT-486 і вище, ОС MS-DOS або Windows-9x.

Четверте покоління (кінець 1990-х – початок 2000-х): Засноване на повноцінній клієнт-серверній архітектурі з використанням професійних реляційних СКБД (наприклад, Paradox, dBase, а згодом Oracle, SQL Server, MySQL) та мереж передачі даних (Novell NetWare, Windows NT, Unix, Linux). Ці системи забезпечували ефективне використання транзакцій, мереж передачі даних та протоколів для забезпечення цілісності та безпеки. В Україні це покоління отримало масовий розвиток після введення обов'язкової сертифікації АБС Національним банком України [7].

П'яте покоління (початок 2000-х – середина 2010-х): Характеризується

переходом до розподілених архітектур з використанням потужних професійних СКБД (ORACLE, IBM DB2, SYBASE, Microsoft SQL Server) та широким застосуванням SQL-запитів. Активно розвивалися локальні та глобальні мережі, що дозволило працювати в режимі реального часу (ON-LINE технологія) з віддаленими філіями та підрозділами. Це покоління забезпечило високий рівень безпеки доступу до інформації та підтримку цілісності бази даних на значних територіальних просторах.

Шосте покоління (середина 2010-х – сучасність): Це поточне та перспективне покоління АБС, що перебуває в стадії активної розробки та впровадження. Воно базується на глобально розподіленій архітектурі, широко використовує Web-технології, хмарні обчислення (Cloud Computing), мікросервісну архітектуру (Microservices), контейнеризацію (Docker, Kubernetes) та API-орієнтований підхід. Дані можуть зберігатися на географічно розподілених серверах, але доступ до них є оперативним та прозорим завдяки розвиненим мережевим технологіям (зокрема Internet) та розподіленим СКБД (як реляційним, так і NoSQL). Це забезпечує високу гнучкість, масштабованість, відмовостійкість та можливість швидкого розгортання нових сервісів. АБС цього покоління активно інтегруються з мобільними додатками, FinTech-сервісами та системами аналізу великих даних.

На сучасному етапі розвитку банківської діяльності пріоритет надається наступним напрямкам:

– розподілені та хмарні архітектури. Відхід від монолітних систем на користь мікросервісів та хмарних рішень для підвищення масштабованості, гнучкості та відмово стійкості;

– забезпечення надійності, збереження та цілісності даних: Критично важливе використання механізмів транзакцій (ACID властивості), а також сучасних підходів до реплікації, резервного копіювання та відновлення даних;

– робота в реальному часі (ON-LINE) з територіально-розподіленою базою даних. Миттєвий доступ до актуальної інформації та оперативне проведення операцій незалежно від фізичного розташування користувача чи даних.

– використання сучасних засобів розробки та проектування. Активне

застосування об'єктно-орієнтованих мов програмування, SQL та NoSQL технологій, фреймворків для швидкої розробки (RAD), а також DevOps-практик;

– комплексна безпека та захист інформації. Реалізація багаторівневих систем безпеки, включаючи розмежування доступу та повноважень, протоколювання всіх операцій, застосування електронних цифрових підписів (ЕЦП), криптографічних методів захисту, систем виявлення вторгнень та моніторингу загроз;

– масштабованість та продуктивність. Відсутність обмежень щодо зростання обсягів баз даних та швидкості обробки транзакцій, здатність ефективно працювати зі зростаючим потоком даних та кількістю користувачів;

– функціональна повнота та інтегрованість. АБС повинна охоплювати всі аспекти банківської діяльності (від операцій до звітності та аналітики) та забезпечувати безшовну взаємодію між усіма інформаційно та функціонально пов'язаними компонентами та зовнішніми системами (API-інтеграції);

– гнучкість та адаптивність. Можливість швидкої модернізації, інтеграції нових сервісів та адаптації до змін у нормативно-правовій базі, бізнес-процесах, а також до зростання кількості клієнтів, філій та обсягів операцій. Це включає підтримку мікросервісної архітектури, що дозволяє незалежне розгортання та оновлення окремих компонентів;

– надійність. Здатність системи забезпечувати безперебійну роботу багатьох користувачів одночасно, без конфліктів та збоїв, гарантуючи достовірний результат обробки інформації та мінімальний час простою (High Availability).

Актуальний розподіл задач в АБС:

– OLTP (On-Line Transaction Processing).Левова частка функціоналу сучасних АБС припадає на оперативну обробку транзакцій у реальному часі. Це включає щоденні банківські операції, такі як перекази, платежі, відкриття/закриття рахунків тощо;

– OLAP (On-Line Analytical Processing). Значна, але менша частка задач присвячена аналітичним функціям. Це включає аналіз ресурсної бази, активів, фінансово-господарської діяльності, оцінку ринків та ризиків, що допомагає

керівництву приймати стратегічні рішення;

– DSS (Decision Support Systems). Хоча це напрям активно розвивається, задачі підтримки прийняття рішень (наприклад, з використанням штучного інтелекту, машинного навчання для прогнозування та оптимізації) все ще менш поширені в АБС порівняно з OLTP та OLAP, але їх роль зростає.

1.7 Мета та задачі кваліфікаційної роботи

Метою даної кваліфікаційної роботи є розробка та реалізація безпечної, масштабованої клієнт-серверної програмної системи, призначеної для здійснення базових банківських операцій, зокрема переказу коштів між рахунками користувачів, з використанням сучасних технологічних рішень та забезпеченням високого рівня цілісності даних.

Для досягнення поставленої мети передбачається сформулювати та вирішити комплекс взаємопов'язаних задач:

1. Провести аналіз теоретичних засад та сучасних тенденцій розвитку автоматизованих банківських систем, що включатиме дослідження архітектурних підходів до побудови АБС, зокрема клієнт-серверних моделей. Також буде проаналізовано основні принципи забезпечення цілісності даних у фінансових системах, з детальним вивченням концепції транзакцій та їхніх ACID властивостей, а також ознайомлення з актуальними методами захисту інформації та авторизації у розподілених системах.

2. Розробити архітектуру програмного забезпечення, що передбачатиме чіткий поділ на серверний (бек-енд) та клієнтський (мобільний Android-додаток) компоненти. У рамках цього завдання буде визначено структуру бази даних для зберігання користувачів, рахунків та транзакцій.

3. Здійснити розробку серверної частини системи. Вона буде реалізована на мові Java з використанням Spring Framework. Передбачається забезпечити функціональність переказу коштів між рахунками користувачів, а також впровадити механізми транзакційної обробки банківських операцій для гарантування

атомарності та цілісності даних. Буде організовано взаємодію з базою даних MySQL, включаючи хешування паролів користувачів для безпечного зберігання. Також планується реалізувати механізм авторизації за протоколом OAuth 2.0 з використанням гранту "Resource Owner Password Credentials" для отримання токенів доступу, а HTTP-запити для комунікації клієнта з сервером будуть налаштовані на передачу інформації у форматі JSON.

4. Виконати розробку клієнтської частини системи. Вона буде створена у вигляді мобільного додатку для платформи Android, що забезпечить зручний інтерфейс для авторизації користувачів та виконання операцій переказу коштів. Буде реалізовано обробку HTTP-запитів до сервера в окремих потоках для підвищення чутливості інтерфейсу. Також передбачається коректна обробка та відображення даних, отриманих від сервера у форматі JSON, та розробка механізмів відображення інформаційних повідомлень (наприклад, про помилки сервера або закінчення терміну дії ключа доступу).

5. Провести тестування та демонстрацію функціональності розробленого програмного забезпечення для перевірки його працездатності, відповідності вимогам безпеки та коректності виконання всіх банківських операцій, з подальшою демонстрацією реалізованих функцій.

2 ВИБІР ТА ОБҐРУНТУВАННЯ ПРОГРАМНО-ТЕХНІЧНИХ ЗАСОБІВ РОЗРОБКИ КЛІЄНТ-СЕРВЕРНИХ ДОДАТКІВ

2.1 Мова програмування Java та її роль у розробці банківських систем

Для розробки програмного забезпечення клієнт-банк, зокрема серверних компонентів та мобільних додатків, мова програмування Java є одним із найбільш обґрунтованих та зручних підходів. Її архітектурні особливості та екосистема дозволяють створювати надійні, масштабовані та безпечні рішення, що є критично важливим для фінансового сектору.

Історично, розвиток мов програмування був зумовлений потребою в абстрагуванні від низькорівневих двійкових кодів, які безпосередньо виконуються комп'ютером. Початкові етапи програмування оперували машинними кодами (послідовності 0 і 1), що було надзвичайно складним і схильним до помилок. Подальша еволюція призвела до появи мов асемблера, а згодом – високорівневих мов програмування, які використовували мнемонічні позначення, слова та вирази. Це значно спростило процес написання програм, використовуючи компілятори та інтерпретатори для перетворення вихідного коду на машинний.

Подальше ускладнення програмних систем зумовило перехід до структурного програмування, де код групувався в логічні блоки, функції та процедури (наприклад, мова C). Це покращило читабельність коду та його модульність. Однак зі зростанням обсягів та складності програмних рішень виникла потреба в ще вищій мірі абстракції та організації коду.

Саме ця потреба спричинила появу об'єктно-орієнтованих мов програмування (ООП), до яких належать Java, C++, C#, Python, Ruby, Swift та інші. ООП базується на концепції об'єктів – програмних сутностей, що інкапсулюють у собі дані (атрибути) та поведінку (методи), які взаємодіють з цими даними. Об'єкти є екземплярами класів, які виступають як шаблони або "креслення" для створення об'єктів, визначаючи їхню структуру та поведінку. Такий підхід дозволяє значно спростити розробку складних систем, оперуючи логічними "будівельними блоками" замість низькорівневих інструкцій.

Ключовими принципами об'єктно-орієнтованого програмування є:

– інкапсуляція – механізм, що дозволяє об'єднувати дані та методи, які маніпулюють цими даними, в єдину сутність (об'єкт) та обмежувати доступ до внутрішнього стану об'єкта ззовні. Це підвищує безпеку даних, запобігає небажаним змінам та спрощує підтримку коду, оскільки зміни в реалізації об'єкта не впливають на зовнішній інтерфейс;

– успадкування (спадкування) – механізм, що дозволяє одному класу (дочірньому або підкласу) отримувати властивості (атрибути та методи) від іншого класу (батьківського або суперкласу). Це сприяє повторному використанню коду, зменшує дублювання та дозволяє будувати ієрархії класів, відображаючи реальні відношення "є видом" (is-a);

– поліморфізм – здатність об'єктів з різними реалізаціями мати спільний інтерфейс, дозволяючи обробляти об'єкти різних класів як об'єкти спільного базового класу. Це спрощує розробку гнучких та розширюваних систем, оскільки один і той же метод може поводитися по-різному залежно від типу об'єкта, до якого він застосовується.

Окрім парадигми ООП, однією з найважливіших переваг Java є її кросплатформність (Write Once, Run Anywhere - WORA). Це досягається завдяки концепції Java Virtual Machine (JVM). Вихідний код Java компілюється в байт-код, який є незалежним від конкретної апаратної платформи чи операційної системи. Цей байт-код потім виконується JVM, яка встановлена на цільовій системі та перетворює байт-код на машинний код, зрозумілий для даної платформи. Ця властивість зробила Java надзвичайно популярною для розробки як серверних додатків, так і мобільних застосунків (зокрема, для Android), забезпечуючи універсальність розгортання [9].

У контексті розробки програмного забезпечення для банківських систем, Java пропонує низку ключових переваг:

– надійність та стабільність – зріла екосистема Java, строга типізація та механізми обробки винятків сприяють створенню високостабільних додатків, що критично важливо для фінансових операцій;

– масштабованість. Java-додатки легко масштабуються як вертикально (більш потужне обладнання), так і горизонтально (додавання нових серверів), що необхідно для обробки зростаючих обсягів транзакцій;

– безпека – Java має вбудовані механізми безпеки на рівні JVM та широку спільноту, яка постійно працює над виявленням та усуненням вразливостей. Це дозволяє реалізовувати високі стандарти безпеки, що вимагаються у банківській сфері;

– підтримка багато потоковості. Java має потужні вбудовані засоби для роботи з багатопотоковістю, що дозволяє ефективно керувати паралельними операціями з базами даних (наприклад, з використанням SQL-запитів до MySQL) та обробляти одночасно велику кількість клієнтських запитів. Це є фундаментальним для забезпечення високої продуктивності та швидкості відгуку в системах онлайн-транзакцій;

– розширена екосистема – наявність величезної кількості фреймворків (наприклад, Spring Framework), бібліотек, інструментів розробки та великої спільноти розробників значно прискорює процес розробки та дозволяє використовувати перевірені рішення;

– інтеграційні можливості – Java надає широкі можливості для інтеграції з різними базами даних, зовнішніми системами (через API, веб-сервіси) та застарілим програмним забезпеченням, що є актуальним для складних банківських інфраструктур.

2.2 Захист Java-технологій та їхня еволюція у веб-середовищі

Історично, Java відігравала значну роль у розвитку динамічного веб-контенту через технологію Java-аплетів. Аплети були невеликими Java-програмами, які вбудовувалися у веб-сторінки та виконувалися безпосередньо у веб-браузері клієнта за допомогою Java Virtual Machine (JVM). Вони використовувалися для додавання інтерактивних елементів, анімації, обчислень на стороні клієнта, а також для візуалізації даних, що раніше було неможливим для статичних HTML-сторінок. На

відміну від інших мов програмування, які компілюються в об'єктний код, залежний від конкретної операційної системи та процесора (наприклад, програма, скомпільована для Windows, не працює на macOS без емуляції), Java використовує проміжний байт-код (bytecode). Цей байт-код є платформонезалежним, а його виконання на будь-якій системі забезпечується встановленою на ній JVM, яка конвертує байт-код у машинний код, зрозумілий для локального процесора.

Процес створення та виконання Java-аплетів традиційно включав наступні етапи [9]:

1. Програміст створював вихідний код Java-аплета.
2. Компілятор Java перетворював вихідний код у байт-код (*.class файл), який був платформонезалежним.
3. Користувач завантажував веб-сторінку, яка містила посилання на аплет.
4. Веб-браузер користувача завантажував байт-код аплета.
5. JVM, інтегрована у веб-браузері (або як плагін), конвертувала байт-код у машинний код і виконувала аплет.

Для забезпечення безпеки, особливо в умовах завантаження аплетів з невідомих джерел, Java-аплетні середовища використовували концепцію "пісочниці" (sandbox). Пісочниця створювала обмежене віртуальне середовище виконання для аплета, суворо контролюючи його доступ до системних ресурсів клієнтського комп'ютера (наприклад, файлової системи, мережевих з'єднань, системних властивостей). JVM виступала посередником між аплетом та системними ресурсами, перехоплюючи, перевіряючи та виконуючи запити аплета лише в межах дозволених правил безпеки. Це дозволяло уникнути зловмисної або випадково некоректної поведінки аплета.

Важливо зазначити, що технологія Java-аплетів значною мірою застаріла і практично не використовується в сучасному веб-розробленні. Це пов'язано з кількома причинами:

– попри концепцію "пісочниці", у минулому було виявлено низку вразливостей, які дозволяли аплетам обходити обмеження безпеки;
– вимагали встановлення та оновлення окремих плагінів JVM у браузері,

що створювало проблеми сумісності та зручності для користувачів;

– поява та розвиток JavaScript, HTML5 (Canvas, WebSockets), CSS3, WebAssembly та інших веб-стандартів надала більш ефективні та безпечні способи створення динамічного та інтерактивного веб-контенту без залежності від сторонніх плагінів. Більшість сучасних браузерів припинили підтримку Java-аплетів (наприклад, Chrome, Firefox, Edge).

У сучасному веброзробленні, Java переважно використовується на серверному боці (бек-енд) для створення потужних, масштабованих та безпечних веб-сервісів та корпоративних додатків. Захист Java-технологій у цьому контексті реалізується на інших рівнях:

– використання захищених конфігурацій серверів (наприклад, Apache Tomcat, Jetty, WildFly, Spring Boot Web Server), регулярне оновлення програмного забезпечення, застосування брандмауерів та систем виявлення/запобігання вторгненням;

– забезпечення безпечної комунікації між клієнтом та сервером, а також між компонентами серверної частини, за допомогою протоколів шифрування, таких як TLS/SSL (Transport Layer Security / Secure Sockets Layer);

– використання надійних механізмів управління доступом, таких як OAuth 2.0 (як зазначено в роботі), OpenID Connect, JWT (JSON Web Tokens) для захисту API-запитів. Це дозволяє розмежувати доступ користувачів до ресурсів відповідно до їхніх ролей та повноважень;

– впровадження практик безпечного кодування для запобігання таким атакам, як ін'єкції SQL, XSS (Cross-Site Scripting), CSRF (Cross-Site Request Forgery), небезпечна десеріалізація тощо. Фреймворки, такі як Spring Security, надають потужні інструменти для реалізації цих механізмів;

– зберігання чутливих даних (наприклад, паролів) у хешованому вигляді (з "сіллю" - salt) та шифрування конфіденційної інформації як у стані спокою (у базі даних), так і під час передачі;

– забезпечення безпечного управління сесіями користувачів для запобігання викраденню сесій;

– конфігурація безпеки СКБД (MySQL), контроль доступу, шифрування даних на рівні сховища, регулярне аудитування;

– інтеграція практик безпеки на всіх етапах життєвого циклу розробки програмного забезпечення, включаючи автоматизоване сканування коду, управління залежностями та моніторинг вразливостей.

З розвитком веб-технологій та підвищенням вимог до безпеки, технологія Java-аплетів була визнана застарілою та виведена з підтримки основними веб-браузерами починаючи з 2015-2017 років. Це було зумовлено низкою причин, включаючи складність підтримки плагінів, часті вразливості безпеки та появу більш ефективних, безпечних та інтегрованих клієнтських веб-технологій (JavaScript, HTML5, WebAssembly).

Незважаючи на це, Java залишається однією з провідних мов для розробки серверних (бек-енд) систем, включаючи складні банківські додатки та мобільні програми для платформи Android. У сучасному контексті, захист Java-технологій фокусується на комплексному підході до безпеки, що охоплює весь стек технологій. Це включає: забезпечення безпеки веб-сервісів та API за допомогою сучасних протоколів авторизації (наприклад, OAuth 2.0, OpenID Connect) та криптографії (TLS/SSL) для захисту даних у транзиті; впровадження надійних механізмів безпеки на рівні додатків (наприклад, з використанням фреймворків безпеки, таких як Spring Security, та дотриманням принципів безпечного кодування згідно з OWASP Top 10); захист даних шляхом шифрування у стані спокою та при передачі, а також хешування чутливої інформації (паролі); розробку деталізованих систем управління доступом на основі ролей та дозволів; забезпечення безпеки інфраструктури (серверів, баз даних, мережевих з'єднань); та інтеграцію практик DevSecOps, що охоплює безпеку на всіх етапах життєвого циклу розробки програмного забезпечення. Таким чином, сучасний підхід до безпеки Java-додатків перейшов від фокусу на "пісочниці" аплетів до побудови багатошарової, глибокоешелонованої системи захисту для корпоративних та розподілених рішень.

2.3 REST Архітектура та веб-сервіси

В основі функціонування переважної більшості сучасних веб-додатків лежить фундаментальна клієнт-серверна модель взаємодії. У цій парадигмі, клієнт представляє собою ту частину додатку, яка відображає користувачеві інтерфейс та обробляє його взаємодію; це може бути веб-браузер, мобільний додаток або десктопна програма. Сервер, у свою чергу, відповідає за зберігання даних, обробку бізнес-логіки, виконання обчислень та підготовку інформації для відображення клієнту. Взаємодія між цими компонентами відбувається за допомогою мережевих протоколів, найпоширенішим з яких є HTTP. При HTTP-взаємодії, статусні коди відіграють ключову роль у індикації результату запиту: наприклад, 200 OK означає успішне виконання, 404 Not Found – ресурс не знайдено, 500 Internal Server Error – критична помилка на сервері, 401 Unauthorized – відсутність аутентифікації, а 403 Forbidden – відсутність дозволу на доступ.

З моменту появи інтернету та стрімкого зростання його складності, еволюціонували й підходи до побудови розподілених систем. У 2000 році Рой Філдінг у своїй докторській дисертації сформулював принципи Representational State Transfer (REST) – архітектурного стилю для розподілених гіпермедіа-систем. REST не є протоколом, а являє собою набір архітектурних обмежень, які, при їх дотриманні, сприяють створенню масштабованих, надійних та гнучких веб-сервісів.

Ключові архітектурні обмеження REST включають: клієнт-серверну архітектуру, що забезпечує чітке розділення відповідальності між компонентами; відсутність стану (stateless), коли кожен запит від клієнта до сервера містить всю необхідну інформацію для обробки без збереження попередніх станів на сервері, що підвищує відмовостійкість та масштабованість; можливість кешування, що дозволяє клієнтам та проміжним серверам зберігати копії відгуків для оптимізації трафіку та продуктивності; багаторівневу систему, яка дозволяє клієнту взаємодіяти з проміжними серверами без усвідомлення цього, покращуючи масштабованість та безпеку. Найважливішим обмеженням є уніфікований інтерфейс, який включає чотири підпринципи: ідентифікація ресурсів за допомогою унікальних URI (наприклад, /users/123); маніпулювання ресурсами через їх представлення, де клієнт

отримує та, за необхідності, модифікує представлення ресурсу; самоописові повідомлення, що містять достатньо інформації для розуміння (наприклад, HTTP-заголовки); та гіпермедіа як рушій стану застосунку (HATEOAS), де сервер надає клієнту гіперпосилання для навігації та подальших дій.

Коли архітектура REST застосовується для створення веб-сервісів, вони називаються RESTful веб-сервісами (або RESTful API). Вони ефективно використовують стандартні HTTP-методи для виконання операцій над ресурсами: GET для отримання даних, POST для створення нових ресурсів, PUT для повної заміни існуючих, PATCH для часткового оновлення та DELETE для видалення ресурсів. Замість традиційного HTML, RESTful API зазвичай обмінюються "сирими" даними у легших для програмної обробки форматах, таких як JSON (JavaScript Object Notation) або XML (Extensible Markup Language), причому JSON є значно більш поширеним у сучасних розробках завдяки своїй простоті та читабельності.

REST залишається домінуючим архітектурним стилем для побудови API у сучасних веб-системах, включаючи мікросервісні архітектури. Це обумовлено його ключовими перевагами: простотою та інтуїтивністю використання стандартних HTTP-методів та URI; високою масштабованістю завдяки stateless-дизайну серверної частини; гнучкістю, що дозволяє незалежну розробку клієнтів та серверів; кросплатформенністю, що забезпечує взаємодію з будь-якої мови програмування та платформи, яка підтримує HTTP; а також ефективністю завдяки можливості кешування та використанню стиснених форматів даних. Ці переваги роблять REST ідеальною архітектурою для створення ефективних та надійних механізмів взаємодії між мобільними додатками, такими як Android-клієнт, та серверними системами у контексті банківських операцій, забезпечуючи швидкий та безпечний обмін даними.

Архітектурний стиль Representational State Transfer (REST) не є конкретною технологією, а скоріше набором фундаментальних принципів та обмежень, дотримання яких дозволяє створювати високоефективні, масштабовані та надійні розподілені системи. Рой Філдінг, який описав REST, наголошував, що ці обмеження визначають загальну архітектуру, залишаючи свободу у виборі

конкретних реалізацій окремих компонентів.

Шість основних архітектурних обмежень REST включають: клієнт-серверну архітектуру, що полягає в чіткому розділенні зон відповідальності між клієнтським та серверним компонентами. Клієнти, відповідальні за користувацький інтерфейс та представлення даних, відокремлюються від серверів, які керують сховищем даних та бізнес-логікою. Такий поділ покращує переносимість клієнтського коду та масштабованість серверів, дозволяючи їх незалежну розробку та вдосконалення, доки не змінюється єдиний інтерфейс взаємодії.

Другим важливим обмеженням є відсутність стану (Stateless). Взаємодія клієнт-сервер обмежується відсутністю збереження контексту клієнта на сервері між окремими запитами. Кожен запит від будь-якого клієнта повинен містити всю необхідну інформацію для його повної обробки, а будь-який стан сесії має зберігатися виключно на стороні клієнта. Це значно підвищує надійність серверів та значно покращує їх масштабованість, оскільки запити можуть бути рівномірно розподілені між будь-якими доступними екземплярами сервера.

Третє обмеження – кешування (Cacheable). Згідно з ним, відповіді від сервера повинні бути явно або неявно позначені як такі, що можуть або не можуть бути кешовані. Це дозволяє клієнтам або проміжним компонентам зберігати копії відповідей та використовувати їх для обробки наступних ідентичних запитів, що суттєво впливає на підвищення масштабованості та продуктивності системи, а також знижує навантаження на серверну інфраструктуру.

Четверте обмеження – багаторівнева система (Layered System). Це означає, що клієнт не має чітко розрізняти, чи підключається він безпосередньо до кінцевого сервера, чи до проміжного посередника. Кожен шар системи може додавати функціональність (наприклад, балансування навантаження, кешування, безпеку), не впливаючи на взаємодію між іншими шарами.

П'яте обмеження – код на вимогу (Code-On-Demand) – є єдиним опціональним. Воно дозволяє серверам тимчасово розширювати або налаштовувати функціональність клієнта шляхом передачі йому виконуваної логіки. У сучасному веброзробленні це в основному стосується JavaScript, що завантажується в браузер

для динамічної обробки та інтерактивності інтерфейсу користувача, підвищуючи гнучкість системи.

Шосте, і, можливо, найважливіше обмеження – єдиний інтерфейс (Uniform Interface). Він спрощує та розділяє архітектуру, дозволяючи кожній частині системи розвиватися незалежно. Єдиний інтерфейс ґрунтується на чотирьох керівних принципах: ідентифікація ресурсів за допомогою унікальних URI; маніпулювання ресурсами через їх представлення; самоописові повідомлення, що містять достатньо інформації для розуміння; та гіпермедіа як рушій стану застосунку (HATEOAS), де сервер надає клієнту гіперпосилання для навігації та подальших дій.

Дотримання цих архітектурних обмежень є ключовим для відповідності архітектурному стилю REST. Якщо сервіс порушує будь-яке з цих обмежень (окрім опціонального "Коду на вимогу"), він не може бути однозначно названий RESTful. Відповідність REST дозволяє будь-якій розподіленій гіпермедіа-системі мати такі бажані властивості, як висока продуктивність, масштабованість, простота розробки, видимість (легкість моніторингу), мобільність компонентів та надійність. Ці властивості роблять REST оптимальним вибором для побудови сучасних веб-сервісів, зокрема у фінансовому секторі.

2.4 Мобільна операційна система Android

Android — це домінуюча відкрита операційна система (ОС) для широкого спектру мобільних пристроїв, включаючи смартфони, планшети, розумні годинники, телевізори, автомобілі та інші IoT-пристрої. Її розробка почалася в компанії Android Inc., яку Google придбала у 2005 році. Основою Android є модифіковане ядро Linux. Згодом, Google у співпраці з іншими учасниками консорціуму Open Handset Alliance (OHA), заснованого 5 листопада 2007 року, спільно розвивали цю нову операційну систему. Сьогодні підтримку та подальший розвиток платформи здійснює Android Open Source Project (AOSP), що забезпечує відкритість та прозорість її розробки. Завдяки цій відкритості, Android має величезну та активну спільноту розробників, яка постійно розширює

функціональність пристроїв та створює інноваційні додатки.

Екосистема Android підтримується офіційним магазином додатків — Google Play Store (який раніше називався Android Market). Google Play пропонує величезний асортимент програм та ігор, включаючи як платні, так і безкоштовні варіанти, доступні для користувачів по всьому світу. Завдяки відкритому характеру ОС Android, користувачам також надається можливість завантажувати та встановлювати додатки з інших джерел ("sideloading"), що надає більшу гнучкість, але вимагає від користувачів підвищеної обережності щодо безпеки.

Основною мовою програмування для розробки нативних Android-додатків історично була Java. Розробники взаємодіють з функціоналом пристрою та операційної системи через багатий набір бібліотек, наданих Google, які становлять Android SDK (Software Development Kit). Хоча Java залишається повністю підтримуваною, починаючи з 2017 року Kotlin стала офіційно рекомендованою мовою для розробки Android-додатків. Kotlin є сучасною, статично типізованою мовою, яка компілюється в байт-код JVM і є повністю сумісною з Java, пропонуючи при цьому більш лаконічний синтаксис, покращені функції безпеки та зручні інструменти для розробників. Це дозволило значно прискорити та спростити розробку.

Офіційно про операційну систему Android було оголошено 5 листопада 2007 року разом із заснуванням Open Handset Alliance, що об'єднав понад 80 компаній, включаючи виробників пристроїв, операторів мобільного зв'язку та розробників програмного забезпечення. Більша частина коду Android була випущена під ліцензією Apache 2.0, що підкреслює її відкритий характер та дозволяє широке використання та модифікацію.

Внутрішня структура Android-додатків раніше базувалася на виконанні Java-додатків та бібліотек віртуальною машиною Dalvik, яка використовувала JIT-компілятор. Однак, починаючи з Android 4.4 (KitKat) і повністю реалізовано в Android 5.0 (Lollipop), Dalvik був замінений на Android Runtime (ART). ART використовує технологію AOT (Ahead-of-Time) компіляції, компілюючи додатки в нативний код під час встановлення, що значно прискорює їх запуск та виконання, а

також покращує енергоефективність. До ключових бібліотек Android входять: система управління (Android Framework), графічні бібліотеки (наприклад, OpenGL ES для 2D/3D графіки, Skia Graphics Engine), двигун WebKit для веб-відображення, бібліотеки SSL для безпечної комунікації, а також бібліотека Bionic (варіант стандартної бібліотеки C). Загалом, Android є складною системою, що налічує мільйони рядків коду, написаних різними мовами, включаючи значні обсяги на XML (для ресурсів та макетів), C, C++ (для низькорівневих компонентів) та Java/Kotlin (для фреймворку та додатків).

Розробка додатків для Android традиційно ведеться на мові Java (з підтримкою версії Java 1.5 і вище), хоча сучасні тенденції вказують на Kotlin як на офіційно рекомендовану мову розробки. Додатки для Android компілюються в спеціальний байт-код, який виконується Android Runtime (ART) – віртуальною машиною, що замінила раніше використовувану Dalvik. ART використовує технологію AOT-компіляції (Ahead-of-Time), перетворюючи байт-код у машинний код безпосередньо під час встановлення додатку, що значно підвищує продуктивність та швидкість запуску.

Google пропонує для вільного завантаження комплексний інструментарій для розробки – Android SDK (Software Development Kit). Цей SDK підтримується на широкому спектрі операційних систем, включаючи Windows (XP, Vista, 7, 8, 10, 11), macOS та Linux, і вимагає встановленого JDK (Java Development Kit) версії 8 або вище.

Історично, для розробки Android-додатків широко використовувалися такі інтегровані середовища розробки (IDE), як Eclipse з плагіном Android Development Tools (ADT) та IntelliJ IDEA, для якої також існував відповідний плагін. Однак, з 2014 року Android Studio (базована на IntelliJ IDEA) стала офіційним і рекомендованим середовищем розробки від Google. Вона надає повний набір інструментів, включаючи візуальний редактор макетів, емулятор, інтеграцію з системою складання Gradle та потужні засоби налагодження, значно спрощуючи та оптимізуючи процес розробки Android-додатків.

Операційна система Android має шарувату архітектуру, яка забезпечує

гнучкість, модульність та ефективне використання ресурсів (рис.2.1). Ця архітектура складається з п'яти основних шарів, кожен з яких базується на функціональності нижчого рівня, надаючи послуги для верхніх шарів. Такий підхід дозволяє розробникам зосереджуватися на створенні додатків, не заглиблюючись у низькорівневі деталі апаратної взаємодії.

Найнижчим рівнем архітектури Android є ядро Linux. Android побудований на базі модифікованого ядра Linux, яке відповідає за управління базовими системними службами, такими як управління пам'яттю, процесами, драйверами пристроїв (камера, Wi-Fi, аудіо, flash-пам'ять), мережеві стеки та управління живленням. Ядро Linux забезпечує апаратну абстракцію, дозволяючи верхнім шарам працювати незалежно від конкретних апаратних реалізацій. Важливо, що ядро Linux постійно оновлюється та оптимізується Google та спільнотою AOSP для підтримки нового апаратного забезпечення, підвищення безпеки та продуктивності.

Над ядром Linux розташований апаратний рівень абстракції (Hardware Abstraction Layer - HAL). HAL є проміжним шаром, який надає стандартизовані інтерфейси. Розробники апаратного забезпечення (виробники чіпсетів та пристроїв) реалізують ці інтерфейси для забезпечення сумісності свого обладнання з Android. Наприклад, існують інтерфейси HAL для камери, Bluetooth, Wi-Fi, GPS. Це дозволяє Android бути незалежним від конкретного апаратного забезпечення та значно полегшує портування ОС на різні пристрої. HAL еволюціонує з кожною версією Android, додаючи підтримку нового обладнання та покращуючи взаємодію, зокрема через ініціативу Project Treble, що прискорює оновлення Android.

Третій шар містить бібліотеки та Android Runtime (ART). Цей шар включає набір бібліотек, написаних на C/C++ (наприклад, libc (Bionic), Media Framework для аудіо/відео, OpenGL ES для графіки, SQLite для баз даних, WebKit для веб-контенту, SSL для безпечних комунікацій). Поруч з цими бібліотеками функціонує Android Runtime (ART) – віртуальна машина, яка є серцем виконання Android-додатків. Вона відповідає за компіляцію байт-коду додатків у нативний машинний код під час встановлення (Ahead-of-Time - AOT компіляція), що забезпечує швидкий запуск та виконання, а також ефективне управління пам'яттю за допомогою "збирача сміття"

(Garbage Collector). ART постійно оптимізується для підвищення продуктивності та енергоефективності.

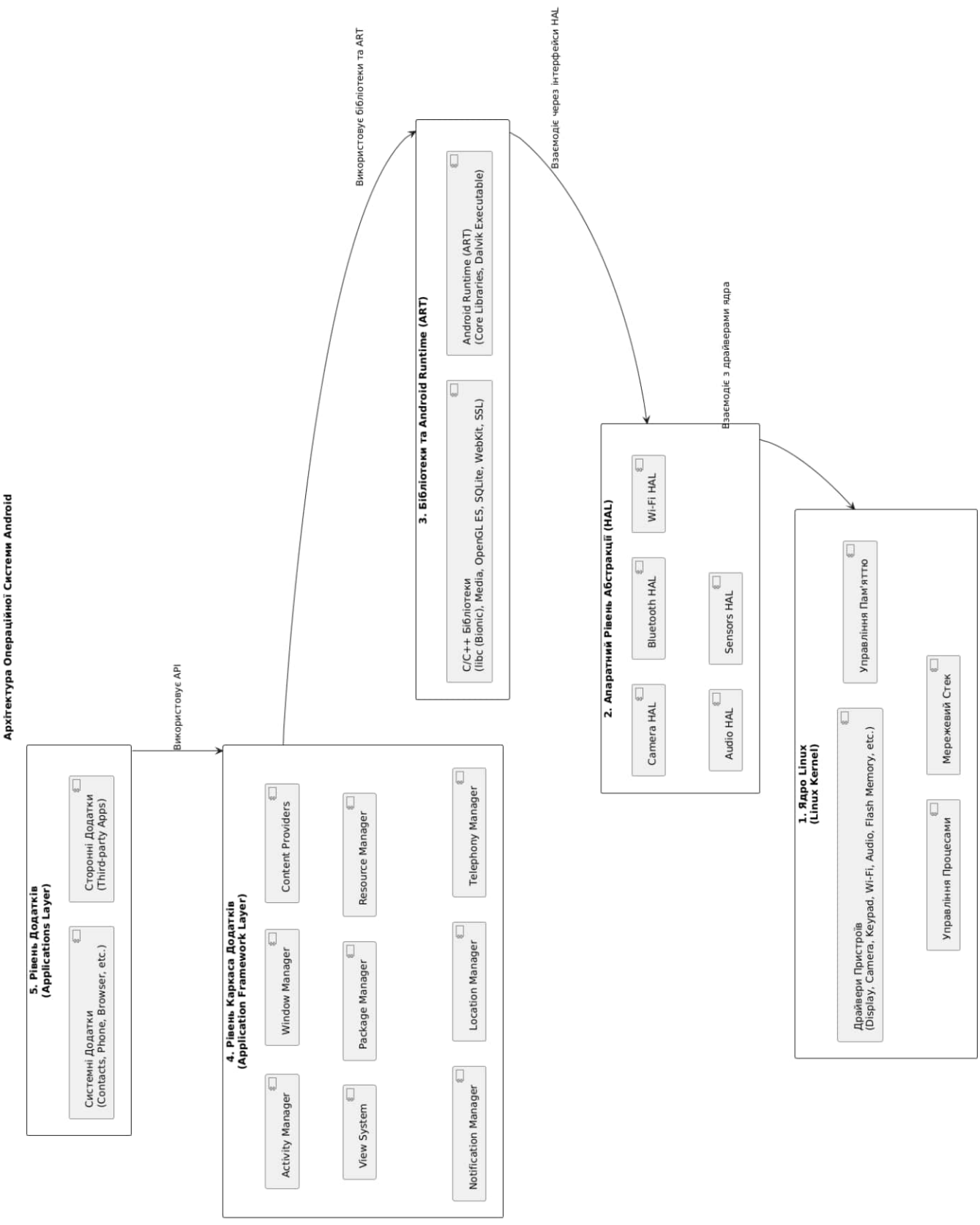


Рисунок 2.1 – Архітектура операційної системи Android [10]

Над бібліотеками та ART розташований рівень каркаса додатків (Application Framework). Цей шар складається з набору API (Application Programming Interfaces) та сервісів, написаних на Java. Він надає розробникам високоінтелектуальні та повторно використовувані компоненти, що значно спрощують розробку додатків. До ключових компонентів фреймворку належать Activity Manager (керує життєвим циклом додатків), Window Manager (управляє вікнами), Content Providers (дозволяють додаткам обмінюватися даними), View System (для побудови інтерфейсу), Package Manager (для управління додатками), Resource Manager (для не-кодових ресурсів), Notification Manager (для сповіщень) та Location Manager (для геолокації). Application Framework є основою для всіх Android-додатків, і Google постійно розширює його, додаючи нові API для підтримки нових функцій ОС та апаратного забезпечення.

Найвищим шаром архітектури є рівень додатків (Applications). Він включає всі користувацькі та системні додатки, такі як Телефон, Контакти, Камера, Галерея, Веб-браузер, а також усі сторонні додатки, які користувач встановлює. Кожен додаток працює у власному процесі та власному екземплярі ART, забезпечуючи ізоляцію та безпеку. Це найдинамічніший шар, що постійно поповнюється новими додатками, які використовують можливості нижчих рівнів для надання функціональності кінцевому користувачеві. Така шарувата архітектура забезпечує значну гнучкість, дозволяючи Google та виробникам пристроїв незалежно оновлювати компоненти, а розробникам – створювати потужні та функціональні додатки, використовуючи добре визначені API.

2.4.1 Грант реквізитів доступу власника ресурсу

Грант реквізитів доступу власника ресурсу (Resource Owner Password Credentials Grant) в OAuth 2.0 є одним з чотирьох стандартних типів надання авторизації, що використовується для отримання ключа доступу (access token). Цей тип гранту застосовується тоді, коли існує високий ступінь довіри між власником ресурсу (користувачем) та клієнтом (додатком). Він є найбільш зручним для

"власних" клієнтів (first-party clients), тобто додатків, розроблених тією ж організацією, що керує сервером авторизації та ресурсами, або для корпоративних клієнтів, які переходять з традиційних методів аутентифікації (наприклад, базової HTTP- або дайджест-аутентифікації) на OAuth 2.0. У таких сценаріях клієнт має прямий доступ до імені користувача та пароля власника ресурсу (рис.2.2).

Рисунок 2.2 – Схема гранту реквізитів доступу власника ресурсу

Процес отримання ключа доступу за допомогою гранту реквізитів власника ресурсу відбувається за такою схемою. Власник ресурсу (користувач) безпосередньо передає свої облікові дані (ім'я користувача та пароль) довіреному клієнту OAuth 2.0 (додатку). Це відбувається, наприклад, через екран входу в мобільний додаток. Клієнт OAuth 2.0, використовуючи надані користувачем облікові дані, формує запит на отримання ключа доступу до відповідної кінцевої точки (endpoint) сервера авторизації. У цьому запиті клієнт повинен вказати тип гранту (grant_type: password), ім'я користувача (username) та пароль (password). За бажанням, може бути вказана область запиту доступу (scope), що визначає дозволи, які запитує клієнт. Якщо клієнт є конфіденційним (тобто має власні секретні реквізити) або до нього застосовуються інші вимоги до аутентифікації, він також проходить аутентифікацію

на сервері авторизації, наприклад, використовуючи HTTP Basic Authentication (як показано в прикладі з заголовком Authorization: Basic ...). Уся комунікація відбувається із застосуванням засобів безпеки транспортного рівня, таких як HTTPS/TLS (рис.2.3).

```
POST /token HTTP/1.1
Host: server.example.com
Authorization: Basic czZCaGRSa33F0MzpnmS3sSL1Jw  // (Base64-кодовані client_id:client_secret)
Content-Type: application/x-www-form-urlencoded

grant_type=password&username=username&password=abcd&scope=read%20write
```

Рисунок 2.3 – Приклад HTTP-запиту

Сервер авторизації отримує запит, перевіряє справжність клієнта OAuth 2.0 (якщо застосовується аутентифікація клієнта) та валідність облікових даних власника ресурсу. У разі успішної перевірки, сервер авторизації видає ключ доступу (access_token) та, за бажанням, ключ оновлення (refresh_token). Ключ оновлення дозволяє клієнту отримати новий ключ доступу після закінчення терміну дії поточного, не вимагаючи повторного введення користувачем облікових даних (рис.2.4).

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "access_token": "2YotnFZFejsrtmOOls",
  "token_type": "Bearer", // Зазвичай "Bearer"
  "expires_in": 3600,      // Час життя ключа в секундах
  "refresh_token": "tGzGdszGZkM" // Необов'язково
}
```

Рисунок 2.4 – Приклад HTTP-запиту

Хоча грант реквізитів доступу власника ресурсу є функціональним, він вважається менш безпечним порівняно з іншими типами грантів OAuth 2.0 (наприклад, Authorization Code Grant або PKCE). Основний ризик полягає в тому, що

клієнт безпосередньо обробляє облікові дані користувача (ім'я користувача та пароль). Якщо клієнт є скомпрометованим або ненадійним, ці реквізити можуть бути перехоплені зловмисниками.

Згідно з рекомендаціями RFC 6819 (OAuth 2.0 Threat Model and Security Considerations) та пізнішими оновленнями, використання цього гранту не рекомендується для загальнодоступних клієнтів (наприклад, сторонніх мобільних додатків або JavaScript-додатків, що працюють у браузері), оскільки вони не можуть надійно зберігати секрет клієнта. Його застосування обмежується сценаріями, де клієнт є високонадійним та повністю контролюється постачальником сервісу (наприклад, офіційний мобільний додаток банку). У таких випадках він може бути зручним для міграції зі старих систем аутентифікації або для надання API існуючим "власним" системам. Для інших сценаріїв, таких як авторизація сторонніх додатків або додатків на базі браузера, рекомендуються більш безпечні потоки, які не вимагають передачі пароля користувача клієнту.

2.4.2 Грант коду авторизації

Грант коду авторизації є одним з найбільш безпечних та широко використовуваних типів надання авторизації в протоколі OAuth 2.0, особливо оптимізованим для конфіденційних клієнтів (наприклад, веб-додатків на стороні сервера, які можуть надійно зберігати свої секрети) та публічних клієнтів (наприклад, мобільних та односторінкових додатків, що не можуть зберігати секрети, у поєднанні з розширенням PKCE – Proof Key for Code Exchange, що буде розглянуто пізніше). Цей потік заснований на переадресації, що вимагає від клієнта здатності взаємодіяти з агентом користувача (зазвичай веб-браузером) для перенаправлення до сервера авторизації та приймати вхідні запити від сервера авторизації через URL переадресації.

Процес отримання ключа доступу за допомогою гранту коду авторизації відбувається у кілька кроків.

Ініціація запиту авторизації (Крок А). Клієнт (наприклад, веб-додаток) ініціює

процес, перенаправляючи агента користувача власника ресурсу (зазвичай веб-браузер) до кінцевої точки авторизації сервера авторизації (рис.2.5). Цей запит містить такі обов'язкові параметри: response_type=code (вказує, що очікується код авторизації), client_id (унікальний ідентифікатор клієнта), та redirect_uri (URL, на який сервер авторизації перенаправить користувача після авторизації). Факультативними, але вкрай рекомендованими, є scope (область запитуваного доступу, наприклад, read profile) та state (непрозоре значення, що використовується для підтримки стану між запитом і зворотним викликом, а також для запобігання атакам CSRF).

```
GET /authorize?response_type=code&client_id=s6BhdRkqt3&state=xyz&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb HTTP/
Host: server.example.com
```

Рисунок 2.5 – Приклад GET-запиту на кінцеву точку авторизації

Після отримання цього запиту, сервер авторизації, як правило, перенаправляє веб-браузер користувача на сторінку входу або надання згоди.

Аутентифікація та авторизація користувача (Крок Б). Власник ресурсу (користувач) проходить автентифікацію на сервері авторизації (вводить логін/пароль, використовує MFA тощо) та надає (або відхиляє) запитуваний клієнтом доступ. Сервер авторизації не ділиться обліковими даними користувача з клієнтом.

Перенаправлення з кодом авторизації (Крок С). Якщо власник ресурсу надає доступ, сервер авторизації перенаправляє агент користувача (веб-браузер) назад до клієнта, використовуючи наданий раніше redirect_uri. Цей URL переадресації включає в себе код авторизації (code) – короткоживучий, одноразовий, непрозорий рядок – та будь-які відомості про локальний стан (state), надані клієнтом. Дуже важливо, що клієнт не повинен використовувати отриманий код авторизації більше одного разу, і будь-які подальші спроби з тим самим кодом мають відхилятися сервером авторизації. Код авторизації пов'язаний з client_id та redirect_uri, забезпечуючи додатковий рівень безпеки (рис.2.6).

```
https://client.example.com/cb?code=SplxlOBeZQQYbYS6WxSbIA&state=xyz
```

Рисунок 2.6 – Приклад URL-переадресації до клієнта

Обмін коду на ключ доступу (Крок Д). Отримавши код авторизації, клієнт (на відміну від попереднього гранту, це відбувається на стороні сервера клієнта, а не в браузері) робить прямий (бек-енд) запит до кінцевої точки видачі ключів (/token endpoint) сервера авторизації. У цьому запиті передаються: grant_type=authorization_code, отриманий code, та той самий redirect_uri, що використовувався на кроці (А). Для конфіденційних клієнтів також обов'язковою є аутентифікація самого клієнта (наприклад, через HTTP Basic Authentication з client_id та client_secret або шляхом їх передачі в тілі запиту).

```
POST /token HTTP/1.1
Host: server.example.com
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW // (Base64-кодовані client_id:client_secret)
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code&code=SplxlOBeZQQYbYS6WxSbIA&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
```

Рисунок 2.7 – Приклад URL-переадресації до клієнта

Верифікація та видача ключів (Крок Е). Сервер авторизації перевіряє справжність клієнта (якщо це конфіденційний клієнт), валідність коду авторизації та відповідність redirect_uri тому, що використовувався на кроці (С). Якщо всі перевірки пройдені успішно, сервер авторизації видає ключ доступу (access_token) та, при необхідності, ключ оновлення (refresh_token), якщо запитується доступ в автономному режимі. Ключ оновлення дозволяє клієнту отримувати нові ключі доступу після закінчення терміну дії поточних без повторної взаємодії з користувачем (рис.2.8).

Грант коду авторизації є найбезпечнішим і рекомендованим потоком для більшості типів клієнтів, оскільки пароль користувача ніколи не передається клієнту, а код авторизації (який тимчасово передається через браузер) не дає доступу до ресурсів сам по собі, а є лише одноразовим обмінником. Завдяки цьому він підходить для захищених веб-додатків, а в поєднанні з розширенням PKCE

(Proof Key for Code Exchange) – також для публічних клієнтів (мобільних та односторінкових додатків), які не можуть безпечно зберігати client_secret. Цей грант є золотим стандартом у сучасній реалізації OAuth 2.0.

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Cache-Control: no-store
Pragma: no-cache
{
  "access_token": "2YotnFZFEjr1zCsicMWpAA",
  "token_type": "Bearer",
  "expires_in": 3600,
  "refresh_token": "tGzv3JOkF0XG5Qx2TlKWIA",
  "scope": "read write" // Опціонально
}
```

Рисунок 2.8 – Приклад URL-переадресації до клієнта

2.4.3 Грант реквізитів клієнта

Грант реквізитів клієнта є типом надання авторизації в OAuth 2.0, призначеним для конфіденційних клієнтів (наприклад, серверних додатків, демонів, фонових сервісів), які бажають отримати доступ до захищених ресурсів, що знаходяться під їхнім власним контролем, або ж до ресурсів, які належать самому сервісу. У цьому сценарії клієнт сам є власником ресурсу або ж діє від імені самого сервісу, і для отримання ключа доступу використовуються лише власні облікові дані клієнта. Взаємодія відбувається без участі кінцевого користувача (власника ресурсу), що робить його ідеальним для міжсервісної комунікації або для виконання автоматизованих завдань (рис.2.9).

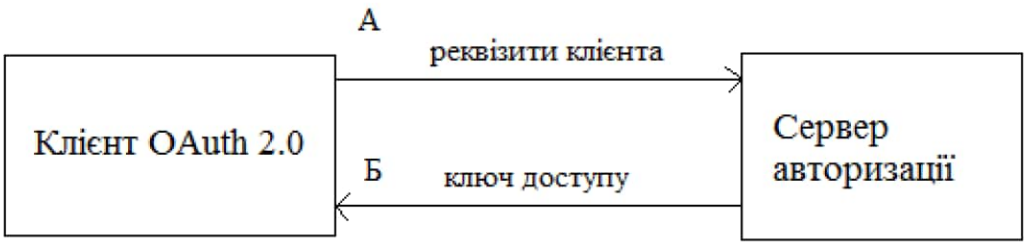


Рисунок 2.9 – Передача реквізитів клієнта

Процес отримання ключа доступу за допомогою гранту реквізитів клієнта відбувається за такою схемою. Запит ключа доступу клієнтом (Крок А). Конфіденційний клієнт OAuth 2.0 проходить автентифікацію на сервері авторизації, використовуючи свої власні реквізити (зазвичай client_id та client_secret, або інші підтримувані засоби автентифікації, такі як взаємний TLS, JWT-бейрер тощо). Клієнт формує запит на отримання ключа доступу до кінцевої точки видачі ключів (/token endpoint) сервера авторизації. У цьому запиті обов'язковим параметром є grant_type: client_credentials. Факультативним параметром є scope, який визначає область запиту доступу. Оскільки в цьому гранті авторизації використовуються лише облікові дані клієнта, жодної додаткової авторизації від кінцевого користувача не потрібно. Уся комунікація відбувається із застосуванням засобів безпеки транспортного рівня, таких як HTTPS/TLS (рис.2.10).

```
POST /token HTTP/1.1
Host: server.example.com
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW // (Base64-кодовані client_id:client_secret)
Content-Type: application/x-www-form-urlencoded

grant_type=client_credentials&scope=api_access
```

Рисунок 2.10 – Приклад HTTP POST-запиту

Перевірка та видача ключа доступу (Крок Б). Сервер авторизації отримує запит, перевіряє справжність клієнта OAuth 2.0 та валідність його реквізитів. Якщо клієнт автентифікований та авторизований для запитуваної області доступу (scope), сервер авторизації видає ключ доступу (access_token). У цьому гранті зазвичай не надається ключ оновлення (refresh_token), оскільки немає кінцевого користувача, який би "авторизував" довготривалу сесію; замість цього клієнт просто повторно автентифікується та запитує новий ключ доступу після закінчення терміну дії поточного.

Якщо запит є недійсним або несанкціонованим, сервер авторизації повертає відповідний HTTP-код помилки (наприклад, 400 Bad Request, 401 Unauthorized) з деталізованим повідомленням про помилку у відповіді.

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "access_token": "2YotnFZFEjr1zCsicMWpAA",
  "token_type": "Bearer",
  "expires_in": 3600, // Час життя ключа в секундах
  "scope": "api_access" // Опціонально
}
```

Рисунок 2.11 – Приклад успішної відповіді сервера авторизації

3 РОЗРОБКА ТА ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Опис схеми баз даних для програмного забезпечення

Для розробки клієнт-банківського додатку спочатку необхідно реалізувати серверну частину. Серверна частина відповідає за відправлення клієнту необхідної інформації. Уся необхідна інформація зберігається у базі даних до якої сервер буде слати запити на отримання необхідної інформації. До необхідної інформації, з точки зору банківської системи, будемо відносити в першу чергу користувача, як власника ресурсу, і в другу чергу його дані, а саме номера рахунків, транзакції, тощо.

Проектування схеми баз даних з користувача обумовлюється розробкою системи захисту інформації, адже безпека – одне з найголовніших вимог щодо автоматизованих систем, які працюють з фінансами. До користувача будемо відносити його персональні дані, а саме: прізвище, ім‘я, ідентифікаційний номер, електронна пошта, та пароль. Звичайно, це не весь набір необхідної інформації для користувача, але достатній для демонстрації роботи системи.

Отже, таблиця users представлена на рисунку 3.1.

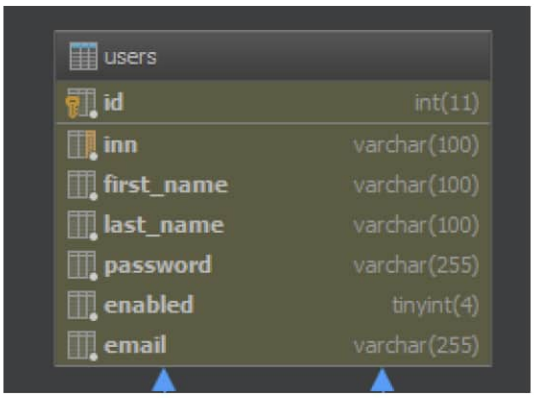


Рисунок 3.1 – Таблиця users

Таблиця буде зберігати всіх зареєстрованих в системі користувачів, та виконувати одну із функцій авторизації користувачів.

Користувачі мають можливість проходження авторизації, отримання необхідної інформації, створення банківських операцій, але необхідно розробити користувача, який матиме змогу адмініструвати системою. Для цього необхідно

розділити права користувачів, та використовувати ці права в якості доступу до тої інформації, та до тих функцій, які будуть в компетенції зареєстрованих користувачів. Наприклад, звичайний користувач додатку не може мати доступу до списку всіх зареєстрованих користувачів, в той же час користувач з правами адміністратора повинен мати такий функціонал для успішного адміністрування системою.

Для вирішення цієї проблеми необхідно створити таблицю user_roles, яка буде зберігати в собі користувача та його роль в системі. Таблиця наверена на рисунку 3.2.

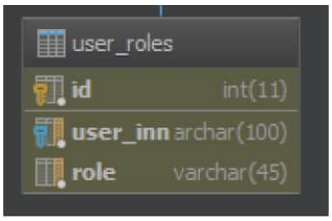


Рисунок 3.2 – Таблиця user_roles

Кожен звичайний користувач повинен мати рахунки, в яких він буде зберігати гроші. Отже рахунку достатньо зберігати інформацію про його власника – користувача, та баланс, тобто сумму грошей. Звичайно, користувач може мати декілька рахунків, отже зв‘язок мід таблицями користувача та рахунку повинна бути один-до-багатьох. Таблиця рахунків представлена на рисунку 3.3.

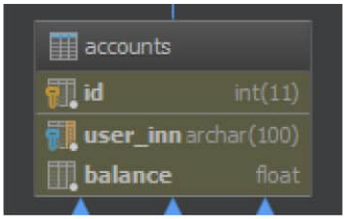


Рисунок 3.3 – Таблиця accounts – рахунки

Для демонстрації роботи банківської інформаційної системи необхідно зберігати інформацію про виконані операції над рахунком користувача. Банківська операція – поняття не очевидне, але в розробленій системі вона може бути як: вхідна операція, вихідна операція та кредитна операція. Оскільки вхідна та вихідна операції

зберігаються одні і ті самі дані, то необхідності виділяти окрему таблицю немає. Операція буде унікальна своїм ідентифікатором, який ій присвоїть СУБД. Але кредитна операція не зовсім відповідає в інформаційному плані вхідній чи вихідній операції, отже необхідно створити дві таблиці: звичайна операція та крединта операція. Кожна операція повинна бути прив‘язана до якогось одного рахунку, отже таблиця рахунку та операції повинні мати зв‘язок один-до-багатьох. Таблиця звичайної операції та кредитної наведені на рисунку 3.4.

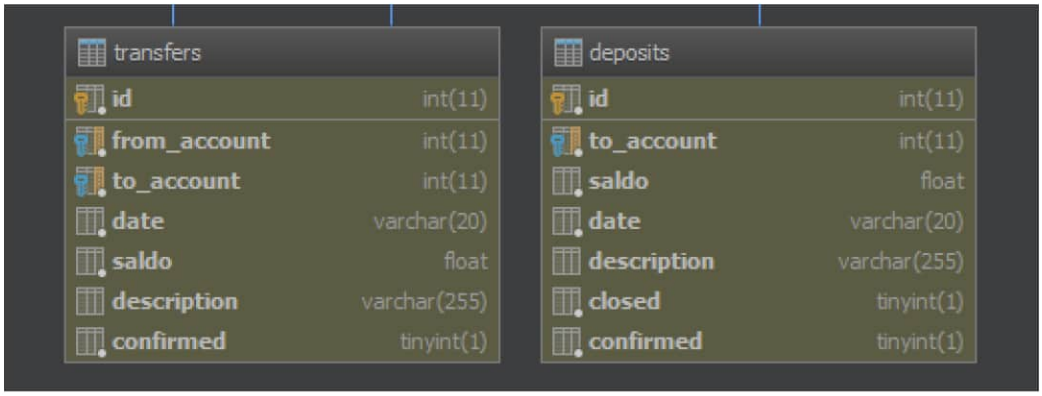


Рисунок 3.4 – Таблиця transfers та deposits – звичайна операція та кредитна

Описаних вище таблиць достатньо для демонстрації роботи функціонування банківської системи. Відповідальність на безпеку системи покладемо на відкритий протокол авторизації OAuth 2.0. Для цього необхыдно розробити необхідні таблиці для його функціонування.

В системі клієнт буде взаємодіяти з сервером через андроід-додаток, отже для забезпечення авторизації достатньо розробити схему для гранту реквізитів доступу власника ресурсів. В першу чергу протокол потребує інформацію про клієнта. Таблиця клієнта зображена на рисунку 3.5.

Для зберігання разових жетонів авторизації користувачів протокол потребує таблицю в базі даних, яка зображена на рисунку 3.6.

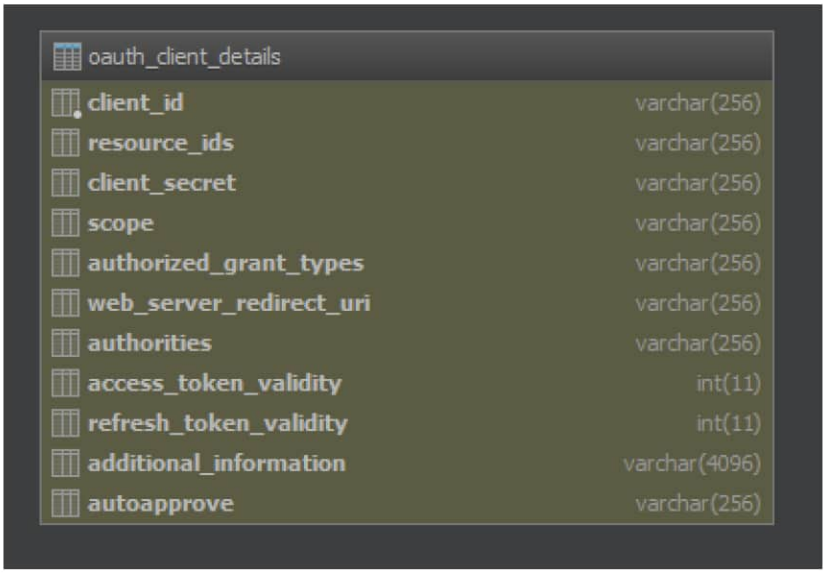


Рисунок 3.5. Таблиця oauth-client-credentials

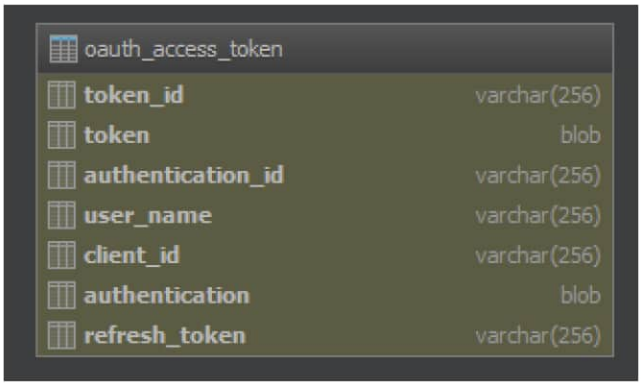


Рисунок 3.6. Таблиця oauth-access-token

3.2 Опис діаграми класів

На основі розробленої схеми бази даних необхідно розробити класи для подальшої розробки функціонування системи. Класи необхідно створювати тільки, якщо вони потрібні для якогось алгоритму функціонування. Отже, мова програмування Java, а саме Spring Framework бере на себе відповідальність впровадження протоколу OAuth 2.0. Програмісту необхідно лише прописати необхідні конфігурації для правильної роботи протоколу.

Всі класи, іх властивості та зв‘язки наведені на рисунку 3.7. Їх кількість не є зразковою, але є достатньою для демонстрації роботи системи. Опис користувача мовою Java наведений у додатку А.

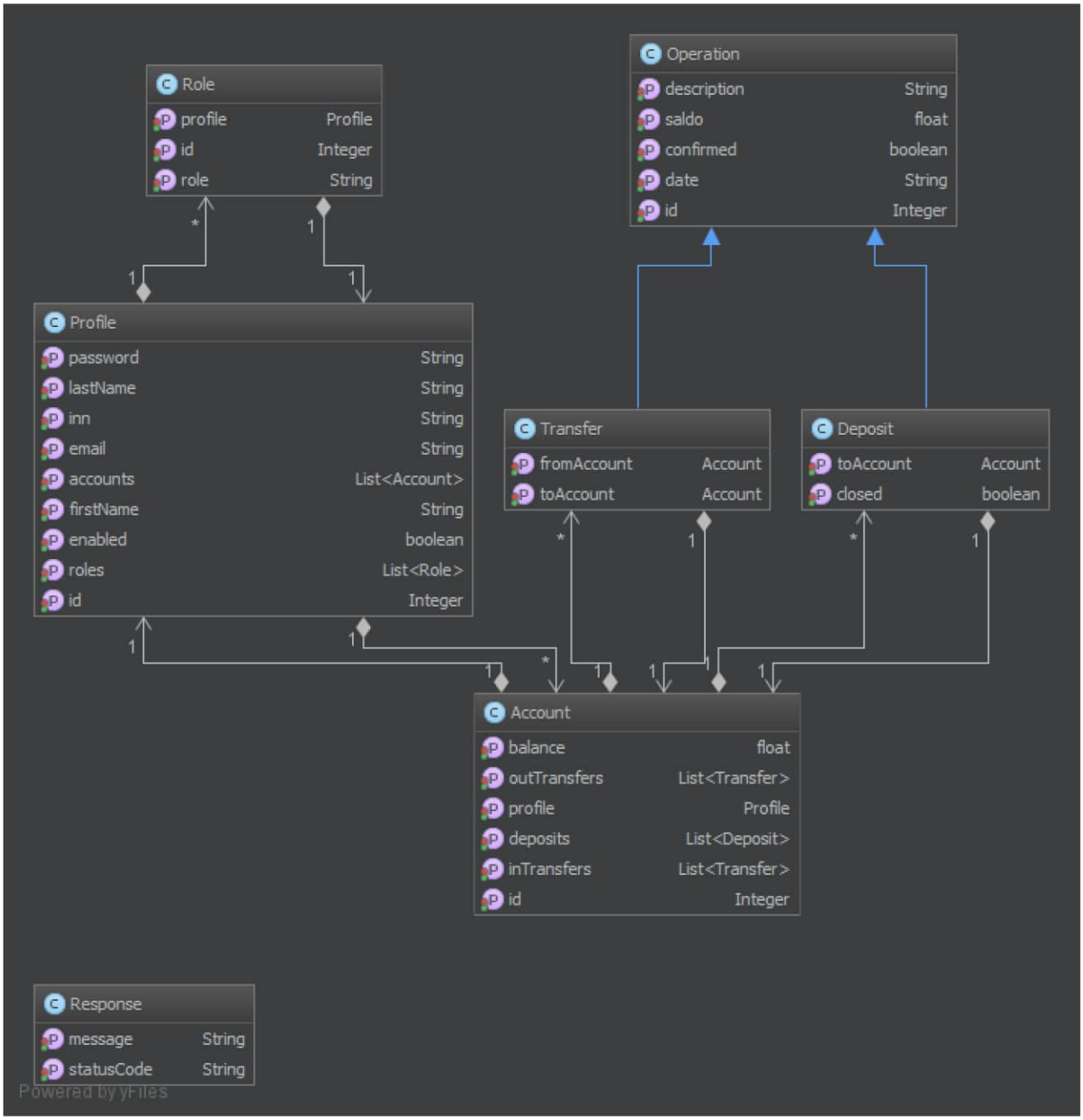


Рисунок 3.7 – Діаграма класів системи

3.3 Взаємодія програмного забезпечення з базою даних

На основі створених сутностей необхідно розробити функціонал, для подальшого ними маніпулювання. Оскільки всі дані в системі зберігаються а базі даних, то необхідно забезпечити зв‘язок між розроблюваним сервером та базою даних. В Java э декілько способів запезпечення такого зв‘язку, одним з яких є використання Spring Framework. Підключення бібліотеки Spring до розроблюваного серверу полегшує рутине програмування. Для забезпечення зв‘язку з базою даних

необхідно просто прописати необхідні конфігурації про розміщення, логін та пароль. Приклад конфігурації підключення до БД описаний в додатку Б.

Для прискорення програмування основних операцій з БД Spring надає модуль Spring Data. Програмісту необхідно лише створити інтерфейс та успадкувати його від інтерфейсу Repository. Таким чином, його тепер можна використовувати для виклику функцій створення, оновлення, читання, видалення сутності с бази даних. Приклад опису інтерфейсу рахунків користувача наведений у додатку В.

Програмісту залишається тільки використовувати ці операції, та сконцентруватися лише на логіці. Такий підхід дає можливість зробити рівень зберігання даних незалежним від рівню його використовування (рис.3.8). Це дає змогу реалізувати паттерн Model View Controller.

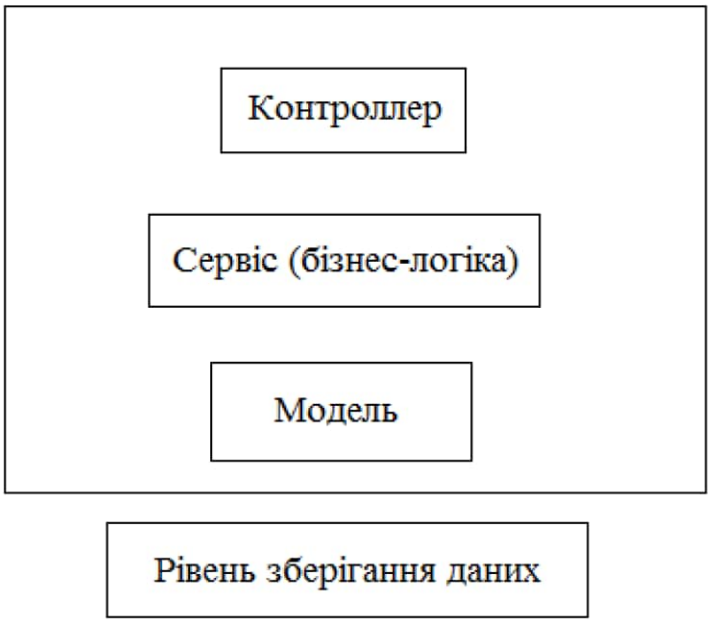


Рисунок 3.8 – Архітектура обробки інформації

На рис.3.8 в якості моделі виступають створені сутності користувача, рахунку, операцій. В якості сервіса виступає елемент, де описується бізнес-логіка використання моделі. А в якості контроллера виступає елемент, який взаємодіє з кінцевим користувачем серверу. Тобто контроллер використовує сервіс, для отримання потрібної інформації, та відправлення її кінцевому користувачу.

Рівень зберігання залишається не залежним від елементів, які відповідають за функціонування системи. Це дає змогу перейти з MySQL до, наприклад, Oracle або MongoDB, не змінюючи сервіс модель та контоллер. В Java є інтерфейси, за допомогою яких можна відокремити рівень зберігання даних та сервіс.

3.4 Розробка сервісів (бізнес-логіки) серверу

Оскільки база даних та необхідні класи створені, то необхідно розробити функціонал, яким може користуватися кінцевий користувач. Отже, сервер має виконувати наступні функції:

Створення користувачів, тобто їх регістрація в системі. Оскільки в розробленій системі користувач має певні ролі, а без ролі, він не матиме можливості отримувати будь-яку інформацію та виконувати будь-які операції, процес реєстрації повинен включати себе не лише створення запису в таблиці users, а і створення запису в таблиці user_roles. Отже процес реєстрації повинен бути обгорнутий в транзакцію, оскільки лише створення запису в таблиці users не дає змогу користуватися системою. Сервіс с реалізацією реєстрації наведений у додатку Д.

Авторизація зареєстрованих користувачів. Користувач вводить свій ідентифікаційний номер в якості логіна, та пароль, введений при реєстрації. Сервер повинен вислати ключ доступу до інформації при успішній авторизації, або вислати повідомлення з описом помилки. Конфігурація мовою Java наведена у додатку Г.

Відкриття рахунку. Користувач, який щойно зареєструвався не має жодного відкритого рахунку, отже необхідно забезпечити можливість його створення. Код Java наведений у додатку Е.

Отримання всіх рахунків. Мається на увазі отримання тільки рахунків авторизованого користувача. Авторизований користувач не повинен мати доступ до рахунків інших користувачів. Код Java наведений у додатку Е.

Отримання усіх вхідних, вихідних та кредитних операцій, які зафіксовані за відповімним рахунком. Іншими словами – це отримання інформації про рахунок. Також необхідно передбачити отримання операцій за певний проміжок часу. Код

Java наведений у додатку Е.

Створення операцій. Цей момент дуже важливий, оскільки банківська операція в системі – переказ коштів. Необхідно передбачити наступні моменти:

– користувач може створити вихідну операцію (переказ коштів на будь-який рахунок по його ідентифікаційному номеру в базі даних);

– користувач може створити крединту операцію, яка не повинна перевищувати 5000 гривень;

– користувач може закрити кредит, маючи на рахунку необхідну суму;

– користувач не може створити крединту операцію, якщо він маж кредит на будь-яких із своїх рахунків;

– користувач не може створити вхідну операцію. Вхідна операція виводиться у інформації про рахунок того користувача, на який була створена вихідна операція;

– користувач не може створити вихідну операцію, не маючи на рахунку необхідних коштів.

Кожна вихідна операція ділиться на: перевірка балансу рахунку відправника, зняття коштів з рахунку відправника, пошук рахунку одержувача, зміна балансу рахунку одержувача. Отже, даний функціонал обгорнутий в транзакцію з рівнем ізоляції – Serializable. Кожна дія всередині транзакції також обгорнута в транзакцію з видимістю Mandatory, який характеризується помилкою, якщо операція почалася не в середині транзакції. Код Java наведений у додатку Е.

Отримання інформації про кожну операцію рахунку. До інформації про операцію відносять:

– її унікальний ідентифікатор;

– дата проведення;

– інформація про відправника (відсутня, якщо операція - кредитна);

– інформація про отримувача;

– опис;

– стан (якщо операція - крединта).

Для забезпечення більшої безпеки, створення кожної операції можна розділити спочатку на запит на створення операції. Мається на увазі підтвердження операції кодом. Створення кожної операції ділиться на створення запиту та підтвердження. Код підтвердження генерується за допомогою генератора випадкових чисел та прив‘язується до операції. Інформація про код підтвердження зберігається у HashMap в пам’яті.

3.5 Розробка контроллерів

Як вже казалося раніше, контроллери виконують функцію комунікації з клієнтом. Комунікація здійснюється мовою HTTP запитів. Тобто виклик кожної функції створеного сервісу супроводжується HTTP запитом.

Усі опреції читання інформації згідно архітектури REST мають супроводжуватися GET запитом. Операції створення – POST запитом, операції зміни – PUT, операції видалення – DELETE запитами.

Отже, для комунікації клієнта с сервером необхідно створити наступні посилання на контроллер які позначені в таблиці 3.1. Контроллер для маніпулювання даними користувача мовою Java показаний у додатку Ж.

Таблиця 3.1. Посилання на сервер для отримання інформації

Доступ	**HTTP метод**	**URI**	**Опис**	**Параметри**
+	POST	*/oauth/token*	отримання ключу доступу до ресурсів	grant_type, client_id, client_secret, username, password
-	GET	*/bank/users/me*	отримання інформації про користувача	
+	POST	*/bank/users/register*	реєстрація користувачів	firstName, lastName, inn, email, password.
-	GET	*/bank/accounts*	отримання інформації про	

			рахунки	
-	GET	*/bank/accounts/$id/user*	отримання інформації про власника рахунку	Id – ідентифікатор рахунку
-	GET	*/bank/accounts/$id/operations/out*	отримання всіх вихідних операцій рахунку	Id – ідентифікатор рахунку
-	GET	*/bank/accounts/$id/operations/in*	отримання всіх вхідних операцій рахунку	Id – ідентифікатор рахунку
-	GET	*/bank/accounts/$id/operations/deposit*	отримання всіх депозитних операцій рахунку	Id – ідентифікатор рахунку
-	POST	*/bank/accounts*	Відкриття рахунку	
-	POST	*/bank/accounts/$id/operations/out*	Запит на створення операції	Id – ідентифікатор рахунку, to_id – ідентифікатор рахунку отримувача, saldo – сумма коштів, description - опис
-	POST	*/bank/accounts/$id/operations/deposit*	Запит на створення кредитної операції	Id – ідентифікатор рахунку, saldo – сумма кредиту
-	PUT	*/bank/accounts/operations/out/$id*	Створення операції	Id – ідентифікатор рахунку, code – код підтвердження
-	PUT	*/bank/accounts/operations/deposit/$id*	Створення крединтої операції	Id – ідентифікатор рахунку, code – код підтвердження
-	POST	*/bank/accounts/$id/operations/in/date*	Отримання всіх вхідних операцій відфільтрованих по даті	Id – ідентифікатор рахунку, from – початкова дата, to – кінцева дата
-	POST	*/bank/accounts/$id/operations/out/date*	Отримання всіх вихідних операцій відфільтрованих по даті	Id – ідентифікатор рахунку, from – початкова дата, to – кінцева дата

-	POST	*/bank/accounts/$id/operations/deposit/date*	Отримання всіх крединтих операцій відфільтрованих по даті	Id – ідентифікатор рахунку, from – початкова дата, to – кінцева дата
-	PUT	*/bank/accounts/$id/operations/deposit/close*	Закриття кредиту	D_id – ідентифікатор кредитної операції

3.6 Розробка android-клієнта

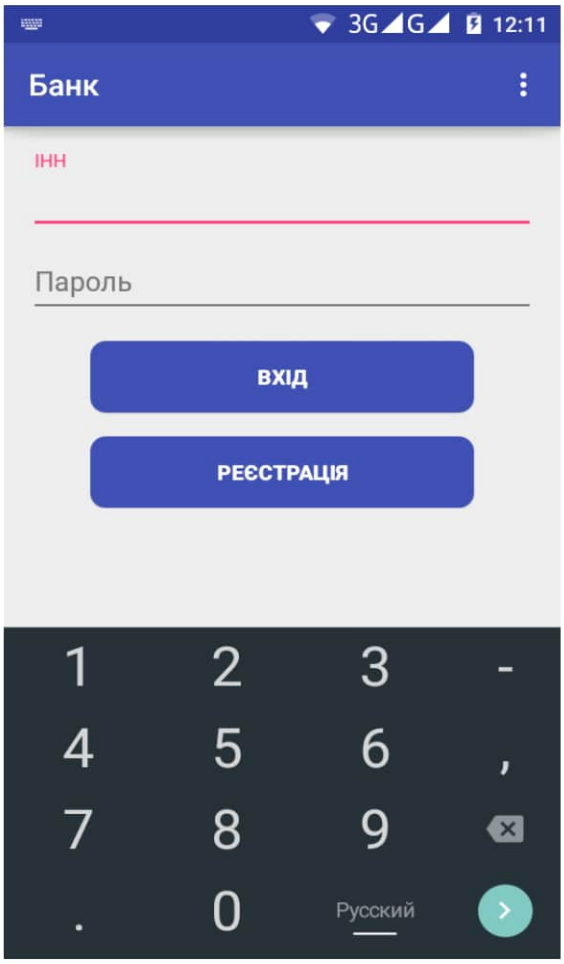
Коли сервер наповнений необхідним функціоналом необхідно розробити клієтна, який буде взаємодіяти з сервером через кінцевого користувача.

Розробка android додатку є дуже простою с точки зору програмування, тому що багато чого в програмуванні додатку лягає на плечі операційної системи. Наприклад – багатопотоковість. Якщо нам необхідно щоб якась функція виконувалась в окремому потоці, то операційна система надає програмісту окремий клас, виклик якого запускає функціонал, який описаний в цьому класі в окремому потоці. Це є необхідним елементом в програмуванні додатків на android оскільки, додаток має графічний інтерфейс, який виконується в головному потоці, і не повинен залежати від виконання функцій, які не пов‘язані з ним. Цей клас називається AsyncTask, який параметризується трьома типа даних. Перший тип даних відповідає за вхідні дані, другий – за дані, які можуть передаватися до інтерфейсу для котролю стану виконання функції, третій – за те, що буде повернено в якості результату виконання функції.

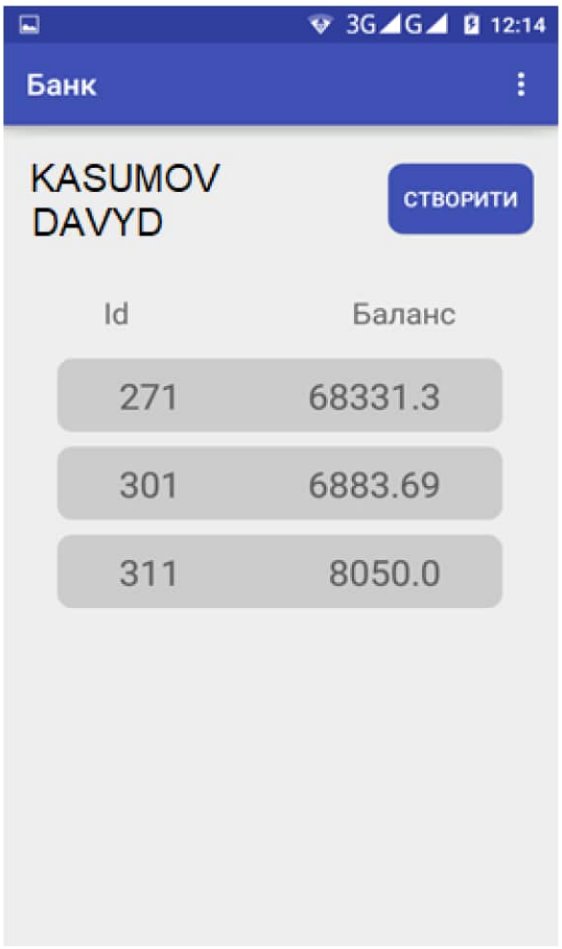
За кожен інтерфейс відповідає клас операційної системи Activity. Щоб створити необхідний інтерфейс екрану необхідно створити клас та успадкуватися від Activity. Усі елементи інтерфейсу задаються у xml файлі, який має суфікс layout. Приклад такого layout наведений у додатку Л.

В першу чергу необхідно реалізувати систему авторизації з сервером. Приклад коду Java для розробки авторизації клієнта з сервером наведений у додатку К. Потім необхідно реалізувати інші інтерфейси, які будуть взаємодіяти з кінцевим

користувачем. Якщо авторизація прошла успішно сервер висилає ключ доступу до ресурсів, який необхідно підставляти в header кожного запиту на отримання інформації. Цей ключ зберігається в пам‘яті додатку. Результат створених інтерфейсів показаний на рисунку 3.8 –рис.3.11.

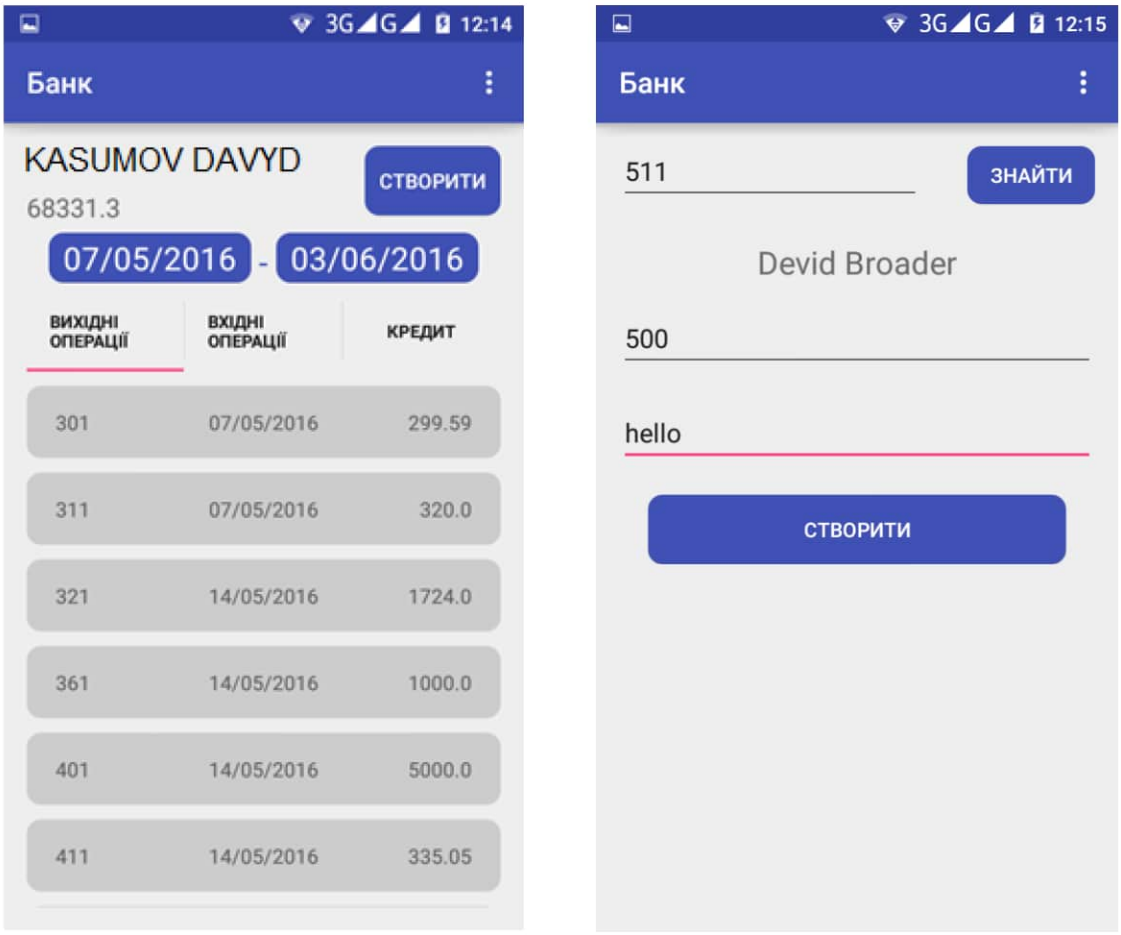


а)



б)

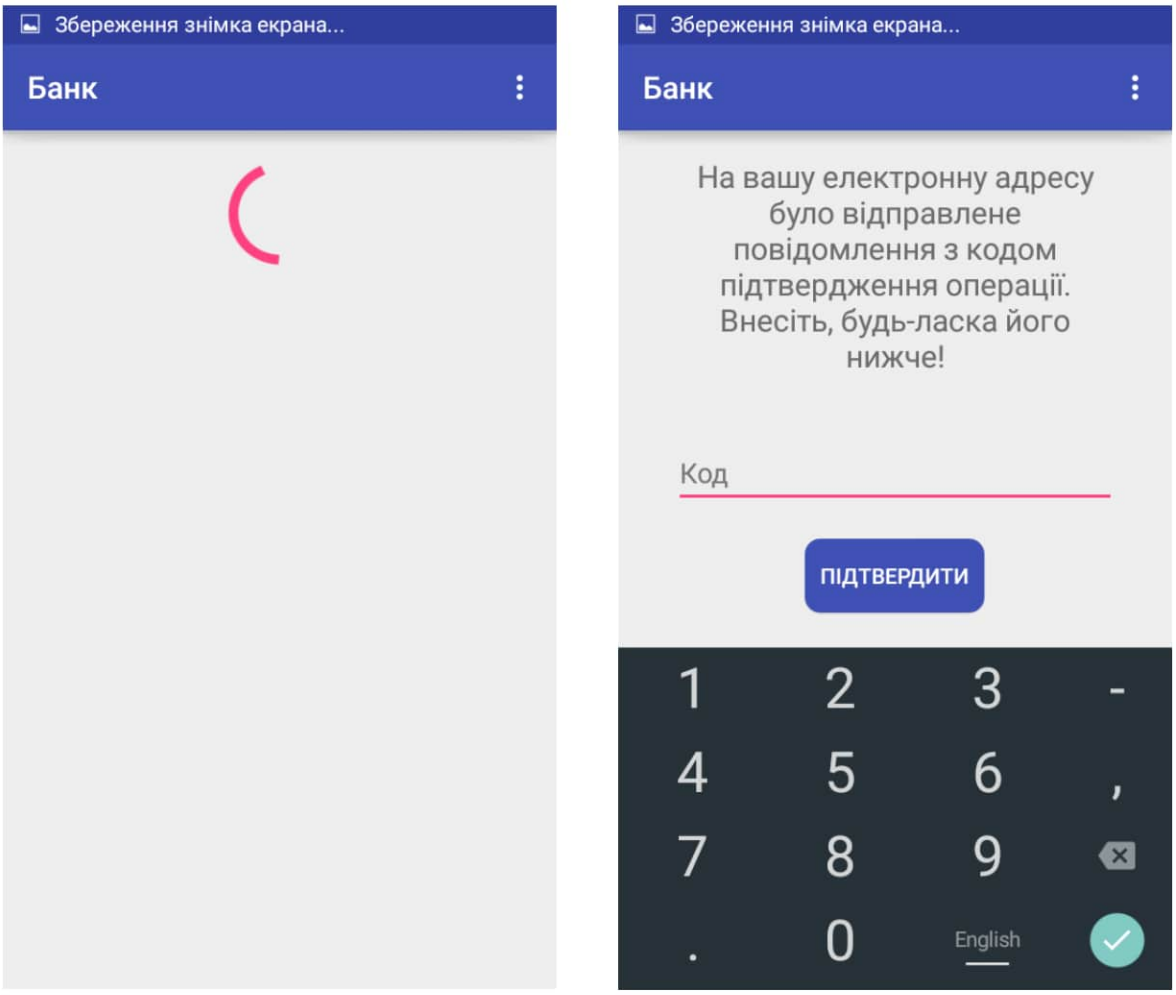
Рисунок 3.8 – Інтерфейси створеного андроід додатку: а) – інтерфейс авторизації користувача з сервером, б) – інтерфейс з інформацією про рахунки користувача, який відкривається після успішної авторизації



а) б)

Рисунок 3.9 – Інтерфейси створеного андроід додатку: а) – інтерфейс з інформацією про операції відповідного рахунку, б) – інтерфейс створення рахунку

Рисунок 3.9 відображає основні відомості про користувача ("Artem Baranovskiy") та його поточний баланс ("683.3"). Нижче розташовані кнопки для фільтрації операцій за періодом ("07/05/2016 - 03/06/2016") та за типом ("Вхідні Операції", "Вихідні Операції", "Кредит"). Основна частина екрану представляє список транзакцій з їх ідентифікаторами (наприклад, "301", "311", "321"), датою операції ("07/05/2016", "14/05/2016") та сумою ("299.59", "320.0", "1724.0", "1000.0", "5000.0", "335.05"). Присутність кнопки "Створити" вказує на можливість ініціювання нових операцій.

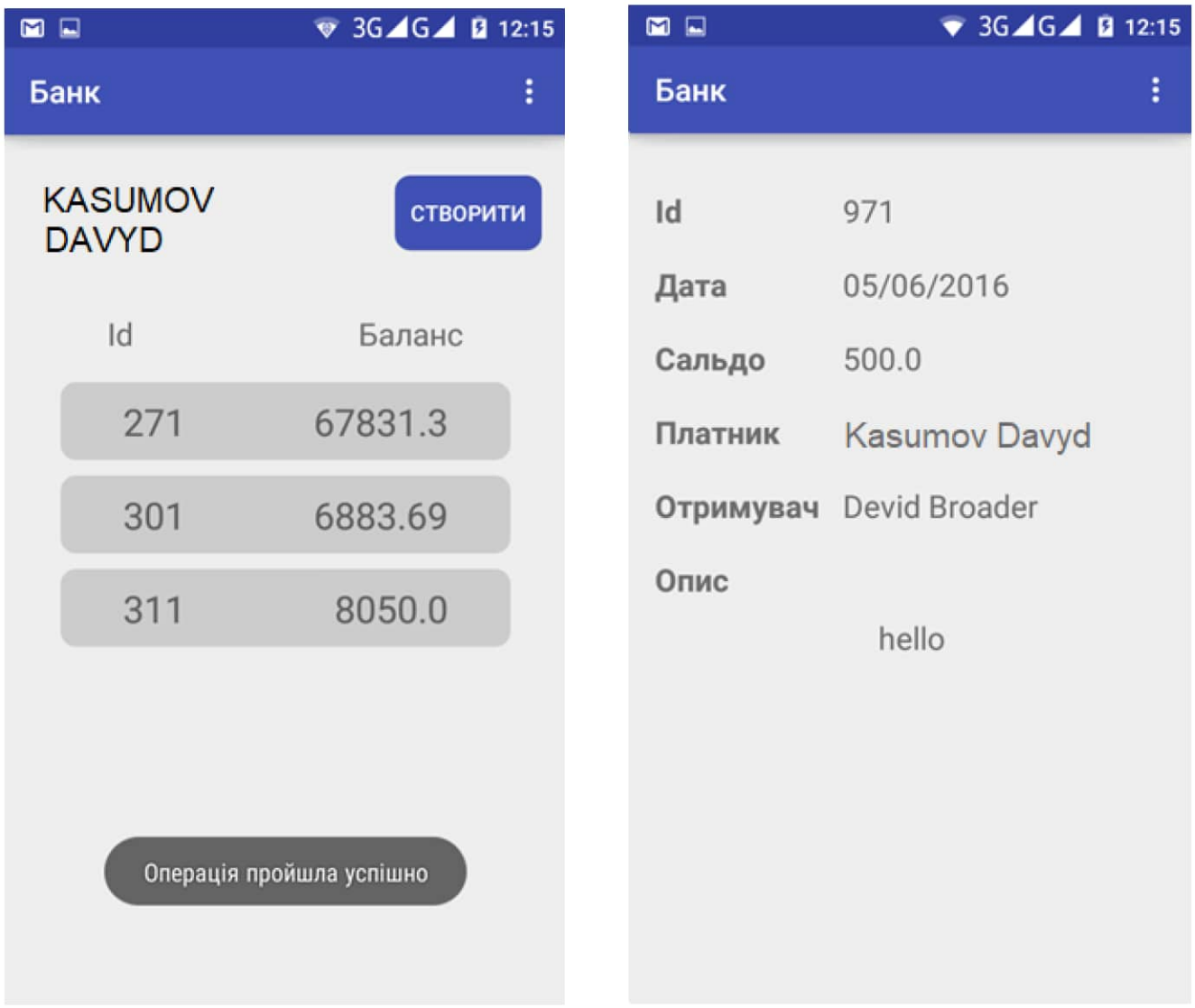


а) б)

Рисунок 3.10 – Інтерфейси створеного андроід додатку: а) – елемент призупинення показу інтерфейсу при виконанні запиту до сервера, б) – інтерфейс вводу кода підтвердження створення вихідної операції

Під час виконання запитів до сервера (наприклад, під час відправлення даних для створення операції або отримання історії транзакцій), додаток відображає елемент завантаження (спінер). Це забезпечує користувача візуальним зворотним зв'язком, інформуючи його про те, що запит обробляється, та запобігаючи подальшим діям до отримання відповіді.

Для підтвердження критичних операцій, таких як створення вихідної операції, реалізовано механізм двофакторної аутентифікації. Користувачеві надсилається код підтвердження на електронну адресу, і цей інтерфейс дозволяє ввести отриманий код для завершення операції. Це значно підвищує безпеку фінансових транзакцій.



а) б)

Рисунок 3.11. Інтерфейси створеного андроід додатку: а) – інтерфейс інформації про рахунки користувача з інформаційним повідомленням про стан операції, б) – інтерфейс інформації про щойно створену вихідну операцію

ВИСНОВКИ

У даній кваліфікаційній роботі бакалавра було розглянуто та реалізовано програмний додаток для мобільної операційної системи Android, призначений для взаємодії з віддаленою банківською системою. Досягнуті результати підтверджують ефективність обраних архітектурних рішень та технологій, а також демонструють практичну цінність розробленого програмного забезпечення.

Проаналізовано та описано ключові аспекти захисту Java-технологій, починаючи від історичного контексту Java-аплетів з їхніми механізмами "пісочниці", верифікації байт-коду та ролі SecurityManager, і закінчуючи актуальними підходами до безпеки у сучасних серверних та мобільних Java-застосуваннях. Цей аналіз підкреслив еволюцію безпеки від клієнтських аплетів до багатошарових систем захисту для корпоративних рішень.

Детально вивчено архітектурний стиль REST, який став фундаментальною основою для взаємодії компонентів розробленого додатка. Описано шість ключових обмежень REST: клієнт-серверна архітектура, відсутність стану, можливість кешування, багаторівнева система, опціональний код на вимогу та уніфікований інтерфейс. Було підкреслено, що дотримання цих принципів є вирішальним для побудови масштабованих, надійних та гнучких веб-сервісів, що особливо актуально для фінансових систем.

Особливу увагу приділено операційній системі Android, її шаруватій архітектурі, що включає ядро Linux, апаратний рівень абстракції (HAL), бібліотеки та Android Runtime (ART), рівень каркаса додатків та рівень додатків. Також було розглянуто ключові мови програмування (Java, Kotlin) та інструменти розробки (Android Studio, Android SDK), які використовувалися в процесі створення мобільного клієнта.

Досліджено різні типи грантів у протоколі OAuth 2.0, зокрема грант реквізитів доступу власника ресурсу, грант коду авторизації та грант реквізитів клієнта. Наголошено на важливості вибору відповідного гранту залежно від довіри до

клієнта та сценарію використання, а також на підвищених вимогах до безпеки у фінансових додатках. Грант коду авторизації виділено як найбільш безпечний та рекомендований для більшості веб- та мобільних клієнтів.

У рамках практичної частини роботи було розроблено програмний додаток для Android, який виступає в ролі клієнтського інтерфейсу для взаємодії з віддаленим банківським сервером. Додаток реалізує стандартну клієнт-серверну модель взаємодії, де Android-клієнт спілкується з Web-сервером (імовірно, через RESTful API), а той, у свою чергу, взаємодіє з сервером віддаленої бази даних. Розроблені інтерфейси включають: інтерфейс відображення інформації про рахунок та операції, інтерфейс створення рахунку, елемент завантаження при виконанні запиту до сервера та інтерфейс введення коду підтвердження для вихідних операцій. Ці результати демонструють успішну реалізацію основних функціональних можливостей для фінансового мобільного додатка з урахуванням аспектів безпеки, таких як підтвердження транзакцій..

ПЕРЕЛІК ПОСИЛАНЬ

1. О. В. Панькова. О. Ю. Касперович.Українське волонтерство в умовах збройної російської агресії. Економічний вісник Донбасу. Цифрова економіка та інформаційні технології. 2022. № 2(68). С. 113-121/

2. Padmapriya N., Tamilarasi K., Kanimozhi P., Kumar T. A., Rajmohan R. and Adeola A. S. «A Secure Trading System using High level Virtual Machine (HLVM) Algorithm», 2022 International Conference on Smart Technologies and Systems for Next Generation Computing (ICSTSN), 2022. Pp. 1–4.

3. Gad A. F. «NumPyCNNAndroid: A Library for Straightforward Implementation of Convolutional Neural Networks for Android Devices», 2019 International Conference on Innovative Trends in Computer Engineering (ITCE), 2019. Pp. 17–22.

4. Bushong V., Das D., Al Maruf A. and Cerny T. «Using Static Analysis to Address Microservice Architecture Reconstruction», 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2021. Pp. 1199–1201.

5. Бан (Інтернет). [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org.

6. IETF RFC 6749. [Електронний ресурс] – Режим доступу до ресурсу: https://www.rfc-editor.org/info/rfc6749.

7. IETF RFC 6819. [Електронний ресурс] – Режим доступу до ресурсу: https://www.rfc-editor.org/info/rfc6819.

8. World Wide Web Consortium [Електронний ресурс] – Режим доступу до ресурсу: https://www.w3.org/Protocols/.

9. Java Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://dev.java/.

10. Android Developers [Електронний ресурс] – Режим доступу до ресурсу: https://developer.android.com/.

ДОДАТОК А.

Програмний код

```
@Entity
@Table(name = "users")
public class Profile implements Serializable {

    @Id @GeneratedValue @Null
    private Integer id;

    @NotNull @Size(min = 2, max = 50)
    @Column(name = "first_name")
    private String firstName;

    @NotNull @Size(min = 2, max = 50)
    @Column(name = "last_name")
    private String lastName;

    @NotNull
    @Column(name = "inn")
    private String inn;

    @NotNull @Size(min = 4)
    @Column(name = "password")
    private String password;

    @NotNull @Email
    @Column(name = "email")
    private String email;

    @Column(name = "enabled")
    private boolean enabled;

    @OneToMany(mappedBy = "profile", fetch = FetchType.EAGER)
    private List<Role> roles;

    @OneToMany(mappedBy = "profile")
    private List<Account> accounts;

    public Profile() {}

    public Profile(String firstName, String lastName, String inn, String
password, String email) {
        setFirstName(firstName);
        setLastName(lastName);
        setInn(inn);
        setEmail(email);
        setPassword(password);
        setEnabled(true);
    }

    public Profile(String firstName, String lastName, String inn,
```

```
String password, List<Role> roles, List<Account> accounts, String email) {
        setFirstName(firstName);
        setLastName(lastName);
        setInn(inn);
        setEmail(email);
        setPassword(password);
        setRoles(roles);
        setAccounts(accounts);
        setEnabled(true);
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getInn() {
        return inn;
    }

    public void setInn(String inn) {
        this.inn = inn;
    }

    public boolean getEnabled() {
        return enabled;
    }

    public void setEnabled(boolean enabled) {
        this.enabled = enabled;
    }

    public String getPassword() {
        return password;
    }
```

```
public void setPassword(String password) {
    this.password = password;
}

public String getEmail() {
    return email;
}

public void setEmail(String email) {
    this.email = email;
}

public List<Role> getRoles() {
    return roles;
}

public void setRoles(List<Role> roles) {
    this.roles = roles;
}

public List<Account> getAccounts() {
    return accounts;
}

public void setAccounts(List<Account> accounts) {
    this.accounts = accounts;
}

@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;

    Profile profile = (Profile) o;

    return inn != null ? inn.equals(profile.inn) : profile.inn == null;

}

@Override
public int hashCode() {
    return inn != null ? inn.hashCode() : 0;
}}
```