

**Міністерство освіти і науки України
Луцький національний технічний університет
Факультет комп'ютерних та інформаційних технологій
Кафедра комп'ютерних наук**

**КВАЛІФІКАЦІЙНА РОБОТА
ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «БАКАЛАВР»**

**РОЗРОБКА ЧАТ-БОТА ДЛЯ МОНІТОРИНГУ ФІНАНСОВИХ
ВИТРАТ**

**DEVELOPMENT OF A CHATBOT FOR MONITORING
FINANCIAL EXPENSES**

спеціальність 122 Комп'ютерні науки

освітня програма «Комп'ютерні науки»

Виконала: здобувачка вищої освіти
групи КН-42
Карасійчук Єлизавета Андріївна

(підпис)

Керівник: асистент
Хиць Руслан Андрійович

(підпис)

Кваліфікаційну роботу
допущено до захисту
«___» _____ 2026 р.
Гарант освітньої програми:
д.пед.н., професор
Тулашвілі Юрій Йосипович

(підпис)

Луцьк – 2026 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет комп'ютерних та інформаційних технологій

Кафедра комп'ютерних наук

Ступінь вищої освіти: бакалавр

Галузь знань: 12 Інформаційні технології

Спеціальність: 122 Комп'ютерні науки

Освітня програма: «Комп'ютерні науки»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Валерій ЛПЦИНА

«16» січня 2026 р.

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧА ПЕРШОГО (БАКАЛАВРСЬКОГО) РІВНЯ ВИЩОЇ ОСВІТИ

Карасійчук Єлизавети Андріївни

(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи «Розробка чат-бота для моніторингу фінансових витрат»

Керівник асистент Хиць Руслан Андрійович

затверджена наказом закладу вищої освіти від «16» січня 2026 р. № 32/01-02

2. Строк подання здобувачем вищої освіти кваліфікаційної роботи «02» червня 2026 р.

3. Вихідні дані до роботи: розробка автоматизованої клієнт-серверної системи обліку особистих фінансів у вигляді Telegram-бота на базі платформи .NET з використанням багаторівневої архітектури, системи управління базами даних PostgreSQL та ORM-фреймворку Entity Framework Core для збереження даних і транзакцій, із застосуванням бібліотеки Telegram.Bot API для взаємодії з користувачем, QuestPDF для генерації візуалізованих фінансових звітів, а також інтеграцією зовнішнього API ПриватБанку для конвертації валют.

4. Зміст розрахунково-пояснювальної записки (перелік питань, що потрібно розробити): Аналіз сучасного стану проблеми, існуючих методів і засобів її розв'язання, аналіз і вибір засобів проектування, опис функціонального наповнення об'єкта проектування, розробка й обґрунтування системного наповнення, оцінка ергономічних та надійнісних параметрів проекрованої системи, функціонально-структурна схема роботи об'єкта проектування.

5. Перелік графічного матеріалу: 12 рисунків, презентація

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
<i>Аналіз проблеми за темою роботи та постановка завдань дослідження</i>	<i>Хиць Р. А.</i>		
<i>Теоретичне дослідження</i>	<i>Хиць Р. А.</i>		
<i>Практична реалізація</i>	<i>Хиць Р. А.</i>		
<i>Спеціальні розрахунки</i>	<i>Хиць Р. А.</i>		
<i>Показник запозичень тексту</i>	%		
<i>Інструментальна перевірка</i>	<i>Кошелюк В. А.</i>		
<i>Нормоконтроль</i>	<i>Хиць Р. А.</i>		
<i>Гарант ОПП</i>	<i>Тулашвілі Ю. Й.</i>		

7. Дата видачі завдання «16» січня 2026 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи бакалавра	Строк виконання етапів роботи	Примітка
1	<i>Провести огляд літературних джерел по темі кваліфікаційної роботи</i>	<i>до 17.02.2026 р</i>	
2	<i>Провести аналіз загальної проблеми і вибір напрямків дослідження</i>	<i>до 28.02.2026 р.</i>	
3	<i>Розробити функціональну схему роботи програмного продукту</i>	<i>до 12.03.2026 р</i>	
4	<i>Описати засоби розробки об'єкта проєктування</i>	<i>до 26.03.2026 р.</i>	
5	<i>Практична реалізація об'єкта проєктування</i>	<i>до 30.04.2026 р.</i>	
6	<i>Провести спеціальні розрахунки</i>	<i>до 18.05.2026 р.</i>	
7	<i>Здача чистового варіанту кваліфікаційної роботи бакалавра на кафедру</i>	<i>до 02.06.2026 р.</i>	

Здобувачка вищої освіти _____ Єлизавета КАРАСІЙЧУК

Керівник роботи _____ Руслан ХИЦЬ

АНОТАЦІЯ

Карасійчук Є. А. Автоматизована система обліку особистих фінансів у вигляді Telegram-бота. Рукопис.

Кваліфікаційна робота бакалавра ОП «Комп'ютерні науки». Луцький національний технічний університет. Луцьк, 2026.

Кваліфікаційна робота бакалавра складається зі вступу, чотирьох розділів, висновків, списку використаних джерел та додатків. У першому розділі виконано аналіз предметної області, наведено приклади існуючих рішень у сфері фінансових трекерів та визначено основні недоліки сучасних мобільних додатків для обліку фінансів. Другий розділ містить специфікацію функціональних і нефункціональних вимог до розроблюваної інформаційної системи, обґрунтування вибору технологій, логічну структуру бази даних та архітектурних алгоритмів. У третьому розділі здійснено розробку Telegram-бота на платформі .NET (C#) з використанням патерну обробки станів (Step Handler), бази даних PostgreSQL, а також реалізовано інтеграцію API для курсу валют та генерацію PDF-звітів. Четвертий розділ містить спеціальні розрахунки, зокрема дослідження засобів захисту інформації в Telegram-бот, механізмів авторизації користувачів за ChatId та безпеки зберігання даних. У висновках узагальнено результати виконаної роботи та визначено перспективи подальшого розвитку системи.

Ключові слова: Telegram-бот, облік фінансів, чат-бот, C#, .NET, Entity Framework, PostgreSQL, QuestPDF.

ANNOTATION

Yelyzaveta Karasiichuk. Automated Personal Finance Tracking System in the Form of a Telegram Bot. Manuscript.

Bachelor's qualification paper in the study program «Computer Science». Lutsk National Technical University. Lutsk, 2026.

The bachelor's qualification paper consists of an introduction, four chapters, conclusions, a list of references, and appendices. The first chapter provides an analysis of the subject area, reviews existing solutions in the field of financial trackers, and identifies the main drawbacks of modern mobile finance applications. The second chapter contains the specification of functional and non-functional requirements for the developed information system, justification of the chosen technologies, the logical structure of the database, and interaction algorithms. The third chapter presents the development of the Telegram bot on the .NET platform (C#) using the state processing architectural pattern (Step Handler), PostgreSQL database, as well as the implementation of third-party API integration and PDF report generation. The fourth chapter covers special calculations, particularly the analysis of information security in Telegram bots, ChatId authorization mechanisms, and data storage security. The conclusions summarize the results of the work and identify the prospects for further development of the system.

Keywords: Telegram bot, finance tracking, chatbot, C#, .NET, Entity Framework, PostgreSQL, QuestPDF.

ЗМІСТ

ВСТУП.....	7
РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ.....	9
1.1 Аналіз сучасного стану проблеми.....	9
1.2 Аналіз аналогічних програм, баз даних, систем, обґрунтування вибраного напрямку та вибір методів рішення	11
1.3 Аналіз і вибір засобів проектування автоматизованої системи обліку фінансів	13
1.4 Постановка завдання на кваліфікаційну роботу бакалавра.....	15
РОЗДІЛ 2 СПЕЦИФІКАЦІЯ ВИМОГ ДО РОЗРОБЛЮВАНОЇ СИСТЕМИ.....	17
2.1 Обґрунтування вибору шляхів, технологій (алгоритмів) і засобів вирішення поставленого завдання.....	17
2.2 Функціонально-структурна схема роботи об'єкта проектування (блок-схеми алгоритмів, UML діаграми класів, діаграми компонентів).....	18
2.3 Створення моделі бази даних	24
РОЗДІЛ 3 РОЗРОБКА АВТОМАТИЗОВАНОЇ СИСТЕМИ ОБЛІКУ ФІНАНСІВ	28
3.1 Практична реалізація об'єкта проектування. Побудова блок-схем модулів програми.....	28
3.2 Тестування та налагодження розробленого Telegram-бота.....	42
3.3 Інструкція користувача з інсталяції та використання програмного продукту	45
РОЗДІЛ 4 СПЕЦІАЛЬНІ РОЗРАХУНКИ	46
ВИСНОВКИ.....	49
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	51
ДОДАТКИ.....	53

ВСТУП

У сучасному світі, де темпи діджиталізації та розвитку інформаційних технологій проникають у всі сфери суспільного життя, традиційні підходи до управління особистими ресурсами зазнають кардинальних змін. Людство неухильно відходить від використання паперових носіїв, готівкових розрахунків та ручного ведення бюджету, переходячи до комплексних цифрових екосистем.

Сьогодні фінансова активність кожної людини генерує масиви даних, які вимагають миттєвої фіксації, точної структуризації та глибокого аналізу.

Особливу роль у цьому трансформаційному процесі відіграють конверсійні інтерфейси та платформи на базі популярних месенджерів. Вони стрімко еволюціонують із простих засобів комунікації у потужні багатофункціональні інструменти, які здатні замінити десятки окремих мобільних застосунків. З огляду на високий ритм життя, дефіцит часу на освоєння складних банківських програм та необхідність швидкого доступу до своїх бюджетних показників, розробка автоматизованих фінансових систем на базі месенджерів стає пріоритетним напрямком для сучасної інженерії програмного забезпечення.

Користувачі все частіше обирають ті рішення, які безперешкодно інтегруються у їхню щоденну рутину, не вимагаючи додаткових установок чи складних багатоступеневих процедур авторизації.

Метою даної кваліфікаційної роботи бакалавра є розробка повнофункціональної автоматизованої системи для зберігання, аналізу та керування особистими фінансами у форматі Telegram-бота з урахуванням сучасних вимог користувачів щодо швидкодії, безперебійності та технологічних тенденцій розробки програмного забезпечення.

Об'єктом дослідження є процес розробки та функціонування інформаційних систем для управління особистими фінансовими потоками у кроссплатформеному вебсередовищі. Предметом дослідження є архітектурні, технологічні та алгоритмічні рішення під час створення автоматизованої системи обліку фінансів (Telegram-бота) на базі платформи .NET та реляційної системи

управління базами даних.

Для виконання поставленого проєкту потрібно вирішити наступні завдання:

- проаналізувати актуальні підходи до створення систем фінансового трекінгу;
- виявляючи їхні переваги, недоліки та вразливі місця у користувацькому досвіді;
- сформулювати й обґрунтувати системні вимоги, що відповідають актуальним потребам користувачів;
- запроектувати надійну та масштабовану архітектуру системи (зокрема із застосуванням патерну ланцюжка обов’язків для обробки станів), яка забезпечує ефективну взаємодію між клієнтською частиною месенджера та серверною бізнес-логікою;
- розробити оптимальні методи зберігання та організації реляційних даних, придатні для швидкої роботи з транзакціями та історією операцій;
- оцінити результати розробки шляхом всебічного тестування та визначити напрями для подальшої оптимізації продукту.

У процесі виконання роботи застосовувались методи об’єктно-орієнтованого проєктування, системного аналізу існуючих рішень та практичного тестування розробленої системи.

Наукова новизна роботи полягає в удосконаленні підходів до проєктування архітектури діалогових фінансових систем. Зокрема, набув подальшого розвитку метод управління станами користувача (патерн Step Handler) у контексті Telegram-ботів, що забезпечило гнучку та безперебійну маршрутизацію багатокрокових фінансових операцій без жорсткої прив’язки до лінійного коду. Крім того, вперше запропоновано та реалізовано комплексну архітектурну модель інтеграції реляційних баз даних (PostgreSQL) із механізмами динамічної генерації графічних PDF-звітів (QuestPDF) безпосередньо в середовищі месенджера з дотриманням принципів чистої архітектури (Clean Architecture).

РОЗДІЛ 1

АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Аналіз сучасного стану проблеми

Проблема ефективного управління власним капіталом завжди була невід’ємною частиною фінансового благополуччя, проте в умовах сучасного переходу до безготівкової економіки набула абсолютно нових масштабів.

Щодня середньостатистична людина здійснює десятки дрібних транзакцій: оплата цифрових підписок, ранкова кава, онлайн-покупки, дрібні перекази з картки на картку. Грошові потоки стали віртуальними та «невидимими», через що контролювати їх без спеціальних програмних інструментів практично неможливо.

Сьогодні IT-ринок пропонує безліч мобільних додатків для ведення бюджету, але більшість із них страждають на спільну хворобу – високий поріг входу та надмірне тертя під час взаємодії користувача з інтерфейсом.

Щоб просто внести інформацію про витрачені сто гривень, людині необхідно розблокувати смартфон, знайти потрібну іконку, дочекатися завантаження програми, обрати рахунок і категорію з довгого списку. Як показують дослідження у сфері поведінкової економіки, будь-яка додаткова або складна дія різко знижує ймовірність формування корисної звички [1].

Саме тому більшість людей відмовляються від ідеї регулярного обліку вже через кілька тижнів після встановлення трекера [2].

Окрім цього, банківські виписки, хоч і формуються автоматично, не вирішують проблеми цілісного бачення. Сучасний користувач часто має рахунки у різних банках, використовує готівку та зберігає заощадження у валюті.

Розкиданість цих даних створює інформаційний хаос замість чіткої фінансової картини. Сучасна людина потребує агрегатора, який би збирав усю інформацію в єдиному місці, але при цьому не вимагав би вивчення складних та перевантажених меню.

Вирішенням цього протиріччя стає перехід до концепції розмовних

інтерфейсів (Conversational UI) на базі популярних месенджерів.

Оскільки користувачі вже проводять значну частину свого екранного часу в додатках для спілкування, інтеграція фінансового функціоналу саме туди є найбільш логічним кроком. За даними експертів з UX-дизайну, взаємодія з ботом сприймається мозком як звичайна текстова бесіда, знижує когнітивне навантаження [3].

Надіслати боту коротке повідомлення з цифрою та категорією набагато швидше і природніше, ніж заповнювати форми у важковаговому мобільному додатку [4].

Більше того, можливості сучасних месенджерів, зокрема Telegram, давно вийшли за межі звичайного обміну повідомленнями. Сьогодні вони надають розробникам доступ до потужних API, які дозволяють підключати надійну серверну логіку, працювати з реляційними базами даних та генерувати деталізовані звіти (наприклад, у форматі PDF) безпосередньо у вікні чату [5].

Окрім проблем користувацького інтерфейсу, вагомим аспектом є безпека та конфіденційність фінансових даних. Багато безкоштовних мобільних додатків для обліку фінансів монетизуються шляхом збору та аналізу поведінкових даних користувачів з метою таргетованої реклами [6].

Передача конфіденційної інформації про щоденні витрати стороннім корпораціям викликає занепокоєння у контексті цифрової безпеки. Натомість, використання закритої архітектури власного серверного бота, що взаємодіє з базою даних через зашифровані канали та спирається на криптографічні стандарти месенджера, дозволяє забезпечити найвищий рівень приватності.

Додатковою проблемою сучасного фінансового планування є відсутність механізмів психологічного заохочення до заощаджень. Класичні таблиці фіксують сухі цифри, що швидко перетворюється на рутину.

Впровадження віртуальних механік, таких як «віртуальний сейф» для цільових накопичень, дозволяє не лише відкладати кошти, але й отримувати від цього процесу емоційне задоволення [7].

Інтеграція таких інструментів у повсякденне середовище месенджера, де

також можливе швидке отримання актуальної інформації про курси валют, перетворює чат-бота з простого записника на повноцінного персонального фінансового асистента [8].

Отже, головна проблема сьогодення полягає не у відсутності інструментів для рахування грошей, а у відсутності зручних, захищених та мінімалістичних рішень, адаптованих до темпу життя сучасної людини. Саме подолання розриву між складністю ведення бюджету та бажанням користувача мати фінансовий контроль формує основний запит на розробку легкого та надійного Telegram-бота для персональної бухгалтерії.

1.2 Аналіз аналогічних програм, баз даних, систем, обґрунтування вибраного напрямку та вибір методів рішення

Для побудови якісної та конкурентоспроможної автоматизованої системи обліку особистих фінансів необхідно детально проаналізувати існуючі на ринку аналоги, виявити їхні сильні сторони та концептуальні недоліки. Сьогодні ринок програмного забезпечення пропонує три основні категорії інструментів для ведення бюджету: спеціалізовані мобільні додатки, вбудовані аналітичні інструменти банківських клієнтів та існуючі аматорські чат-боти.

У додатку А представлено фрагменти програмного коду та допоміжні матеріали, що ілюструють практичну реалізацію основних функціональних можливостей Telegram-бота для моніторингу фінансових витрат.

Аналіз спеціалізованих мобільних додатків (Monefy, CoinKeeper, Spendee).

Ці програми є найпопулярнішим вибором серед користувачів. Їхня головна перевага полягає у наявності потужного графічного інтерфейсу та глибокої візуалізації даних у вигляді діаграм.

Проте вони мають суттєві системні недоліки. По-перше, розширений функціонал (мультивалютність, хмарна синхронізація або експорт звітів) найчастіше доступний лише за умови оформлення платної підписки. По-друге, необхідність щоразу шукати додаток на робочому столі смартфона, відкривати

його та проходити екрани завантаження створює так званий «користувацький бар'єр», який знижує мотивацію до регулярного запису дрібних витрат.

Аналіз банківської аналітики (Monobank, Приват24). Головним плюсом таких систем є їхня повна автоматизація: банк самостійно категоризує транзакції без втручання людини. Однак цей підхід має критичний недолік – замкненість екосистеми. Якщо сучасний користувач має рахунки у кількох різних банках, отримує частину доходу готівкою або ж зберігає заощадження в іноземній валюті, додаток одного банку фізично не здатен відобразити реальну картину його загального капіталу.

Аналіз існуючих фінансових Telegram-ботів. На платформі Telegram вже існують боти для запису витрат, проте переважна більшість із них є простими скриптами. Їхніми типовими проблемами є: використання ненадійних плоских файлів (JSON або CSV) замість повноцінних реляційних баз даних, відсутність захисту від втрати стану під час перезавантаження сервера, а також неможливість генерувати зручні візуальні звіти для друку.

Проведений аналіз показує, що на ринку існує значний попит на інструмент, який поєднував би простоту доступу (як у месенджері) з надійністю, безкоштовністю та мультивалютністю (як у спеціалізованих програмах). Тому вибраним напрямом розробки є створення власного фінансового Telegram-бота.

Цей напрям має низку беззаперечних переваг:

- абсолютна кросплатформеність: бот стабільно функціонує на будь-якій операційній системі (iOS, Android, Windows, macOS), де встановлено месенджер Telegram;
- економія ресурсів: продукт не потребує встановлення та не займає пам'ять на пристрої користувача;
- швидкодія: взаємодія здійснюється через оптимізований протокол месенджера за допомогою кнопок (Reply Keyboard) та текстових команд.

Вибір методів рішення для практичної реалізації поставленого завдання та усунення недоліків існуючих систем було обрано наступні технічні та алгоритмічні методи:

1) метод маршрутизації та обробки станів (State Management): Для забезпечення стабільного та логічного діалогу з користувачем обрано використання об'єктно-орієнтованого патерну «Ланцюжок обов'язків» (на рівні обробників – Step Handler);

2) метод зберігання даних: Замість спрощених NoSQL рішень обрано використання реляційної системи управління базами даних PostgreSQL. Оскільки фінансова інформація вимагає дотримання суворих принципів ACID (атомарність, узгодженість, ізолюваність, довговічність) та підтримки чітких зв'язків між сутностями (Користувач та Транзакція), реляційна модель є єдиним безпечним методом збереження капіталу;

3) метод генерації звітності: Для уникнення виведення перевантажених текстових таблиць у чат обрано метод програмної генерації документів у форматі PDF (за допомогою бібліотеки QuestPDF). Це дозволяє користувачеві отримувати професійно зверстані фінансові виписки, готові до пересилання чи друку;

4) метод інтеграції даних: Застосовано метод взаємодії через REST API з серверами ПриватБанку для автоматичного отримання актуальних валютних курсів. Це дозволяє в режимі реального часу конвертувати поточний баланс користувача (в гривнях) в еквівалент у доларах та євро без необхідності ручного введення коефіцієнтів.

Застосування вищеперелічених методів дозволить створити надійну, функціональну та відмовостійку систему, що перевершує за зручністю існуючі базові аналоги.

1.3 Аналіз і вибір засобів проектування автоматизованої системи обліку фінансів

Успішна реалізація будь-якого програмного забезпечення безпосередньо залежить від грамотного підбору інструментарію, мов програмування та супутніх фреймворків. Оскільки розроблюваний фінансовий Telegram-бот має

справу з конфіденційною інформацією (баланси, історія транзакцій) та повинен миттєво реагувати на повідомлення від багатьох клієнтів одночасно, архітектурний фундамент мусить відповідати найвищим стандартам паралелізму, безпеки та відмовостійкості.

Базовою мовою програмування для бекенд-частини було обрано C#, що функціонує на передовій платформі .NET. Такий вибір є закономірним: суворостатична типізація C# дозволяє виявляти левову частку критичних помилок ще на етапі компіляції, що недоступно для динамічних мов. Крім того, ядро .NET надає потужні вбудовані механізми асинхронного програмування (модель `async/await`), що є критично важливим для безперебійної обробки вхідних мережевих запитів від серверів месенджера без блокування головного потоку програми.

Для організації взаємодії між серверною логікою та екосистемою месенджера залучено бібліотеку Telegram.Bot API. Цей інструмент надає розробнику готові абстракції для читання текстових команд, маршрутизації натискань на клавіатури та управління сесіями діалогів, що нівелює необхідність писати низькорівневі HTTP-запити. Сховищем інформації виступає PostgreSQL – одна з найнадійніших реляційних систем управління базами даних у світі. Її застосування продиктоване самою природою предметної області.

Фінансові операції вимагають абсолютної транзакційної цілісності (відповідність принципам ACID). Наприклад, якщо під час переказу коштів між двома різними користувачами (за їхніми `ChatId`) станеться системний збій, PostgreSQL гарантовано виконає відкат (`rollback`) усієї транзакції, не допускаючи безпідставної втрати або клонування грошей.

Для зв'язування об'єктно-орієнтованого коду на C# із реляційними таблицями бази даних застосовується ORM-система Entity Framework Core.

Вона дозволяє оперувати даними через зручні конструкції LINQ, тим самим звільняючи розробника від написання вразливих SQL-скриптів вручну та автоматично захищаючи систему від атак типу SQL-ін'єкцій.

Генерація візуальної фінансової звітності побудована на базі бібліотеки

QuestPDF. На відміну від багатьох традиційних рішень, які намагаються конвертувати HTML-код у формат документів, ця бібліотека застосовує пряме малювання макетів за допомогою гнучкого Fluent API. Це забезпечує ідеальну піксельну точність фінансових виписок та надвисоку швидкість формування PDF-файлів.

Безпосереднє написання, збирання та відлагодження програмного коду проводиться у професійному інтегрованому середовищі розробки JetBrains Rider.

Ця IDE пропонує передові алгоритми статичного аналізу, інтелектуальне автодоповнення та глибоку інтеграцію з компілятором Roslyn, що дозволяє максимально сфокусуватися на проектуванні складної бізнес-логіки.

Синтез усіх перелічених технологій формує міцний, масштабований моноліт, здатний ефективно витримувати навантаження та гарантувати бездоганний рівень користувацького досвіду під час ведення бюджету.

1.4 Постановка завдання на кваліфікаційну роботу бакалавра

Ключовим наміром виконання цієї кваліфікаційної роботи є розробка та практичне впровадження високопродуктивного програмного інструменту у форматі Telegram-бота, який повністю автоматизує процес реєстрації, аналізу та контролю персональних доходів і витрат. Гостра потреба у створенні такого рішення безпосередньо продиктована зростаючою втомою споживачів від важковагових мобільних додатків та їхнім об'єктивним бажанням керувати власним капіталом через швидкі розмовні інтерфейси без зайвих когнітивних зусиль.

Щоб успішно втілити запланований технологічний задум у життя, у межах кваліфікаційної роботи передбачається виконання наступних практичних кроків:

- дослідити сучасний ринок програмного забезпечення для персональної бухгалтерії з метою виявлення концептуальних недоліків та «вузьких місць» існуючих трекерів;
- визначити суворі технічні критерії та функціональні межі майбутньої

системи, орієнтуючись на реальні очікування цільової аудиторії месенджерів;

- спроектувати надійну серверну архітектуру бекенду, застосувавши об'єктно-орієнтований патерн маршрутизації (Step Handler) для безпомилкового управління множинними діалогами;

- змодельовати логічну структуру реляційних таблиць у базі даних PostgreSQL, що беззаперечно гарантуватиме безпеку та транзакційну цілісність кожної фінансової операції клієнта;

- імплементувати зручне діалогове середовище взаємодії, доповнене інтеграцією зовнішнього банківського API для конвертації валют та механізмом візуалізації звітних даних у форматі PDF;

- перевірити працездатність та відмовостійкість готового програмного продукту шляхом комплексного налагодження, гарантуючи його стійкість до нестандартних вхідних даних.

Сукупність перелічених завдань формує цілісний технологічний цикл – від аналітичного дослідження предметної області до отримання готового до експлуатації програмного продукту. Практична цінність розробленого рішення полягає у його доступності для широкого кола користувачів без необхідності встановлення додаткового програмного забезпечення чи проходження тривалої реєстрації. Подальший розвиток системи передбачає можливість розширення функціональності шляхом додавання багатокористувацького режиму, інтеграції з банківськими сервісами та впровадження інтелектуальних алгоритмів прогнозування витрат.

РОЗДІЛ 2

СПЕЦИФІКАЦІЯ ВИМОГ ДО РОЗРОБЛЮВАНОЇ СИСТЕМИ

2.1 Обґрунтування вибору шляхів, технологій (алгоритмів) і засобів вирішення поставленого завдання

Під час створення фінансового програмного забезпечення одним із найбільш відповідальних етапів є затвердження базових архітектурних рішень та алгоритмів, які формуватимуть ядро майбутньої системи. Процес відбору інструментарію базувався на глибокому технічному аналізі специфіки роботи месенджерів, де ключовими критеріями виступали відмовостійкість, транзакційна цілісність та здатність миттєво обробляти множинні асинхронні події. На противагу традиційним вебсайтам, архітектура Telegram-бота вимагає принципово іншого підходу до серверної взаємодії. Замість класичної моделі «браузер-сервер», розроблений продукт функціонує як оптимізований фоновий процес (Background Service) у середовищі .NET, який безперервно підтримує зв'язок із хмарними серверами Telegram. Такий шлях дозволяє мінімізувати споживання оперативної пам'яті та гарантувати миттєву реакцію на команди без необхідності рендерингу важких графічних інтерфейсів.

Якщо в медіа-платформах виправданим є використання NoSQL сховищ, то для персональної бухгалтерії єдино правильним рішенням стала суворе реляційна модель на базі PostgreSQL. Грошові операції, такі як внутрішні перекази між клієнтами або внесення коштів до віртуального «Сейфу», категорично не допускають втрати, змішування чи дублювання інформації.

Реляційні бази даних на рівні рушія підтримують транзакційні механізми (ACID), що убезпечує систему від критичних збоїв під час паралельних запитів від різних користувачів. У синергії з ORM-фреймворком Entity Framework Core це забезпечує безпечний рівень абстракції під час роботи з фінансовими сутностями. Проблема алгоритмічної складності діалогових інтерфейсів була вирішена шляхом імплементції об'єктно-орієнтованого патерну проектування «Ланцюжок обов'язків», який у даному проекті адаптований під обробники.

Оскільки бот позбавлений звичних візуальних сторінок, складні дії розбиваються на ізольовані логічні кроки (наприклад: ініціалізація переказу, введення ChatId, введення суми, підтвердження). Розроблений алгоритм маршрутизації динамічно визначає, який саме клас-обробник має перехопити вхідне повідомлення залежно від зафіксованого стану користувача у базі даних, повністю унеможливаючи логічні колізії.

Щодо візуалізації отриманих результатів, традиційний текстовий вивід у чат було визнано недостатнім для глибокого фінансового аналізу. З огляду на це, застосовано алгоритм програмної генерації багатосторінкових PDF-звітів «на льоту» (за допомогою бібліотеки QuestPDF). Разом із підключенням стороннього REST API для динамічного розрахунку курсу іноземних валют, це формує завершену та самодостатню фінансову екосистему.

Обрана комбінація реляційної моделі збереження даних, патерну обробки станів та асинхронного фонового сервісу дозволила побудувати відмовостійкий бекенд, який ідеально адаптований до викликів сучасної комунікації.

2.2 Функціонально-структурна схема роботи об'єкта проектування (блок-схеми алгоритмів, UML діаграми класів, діаграми компонентів)

Головною метою етапу проектування є побудова прозорої, масштабованої та відмовостійкої архітектури Telegram-бота, яка забезпечить коректну обробку користувацьких запитів, надійне збереження фінансових транзакцій та миттєву генерацію звітності. Запропонована система функціонує за принципом подійного зв'язку між клієнтом (додатком месенджера) та серверною логікою бекенду.

На рисунку 2.1 представлена загальна функціональна схема програмного комплексу. Схема демонструє шлях проходження інформації: користувач взаємодіє з інтерфейсом Telegram, який через захищений API надсилає об'єкти оновлень (Updates) до серверної частини на базі .NET.

Сервер приймає ці події, проводить їх через головний маршрутизатор

(UpdateHandler), звертається до бази даних PostgreSQL для перевірки стану і формує відповідну реакцію у вигляді текстового повідомлення або згенерованого PDF-документа.

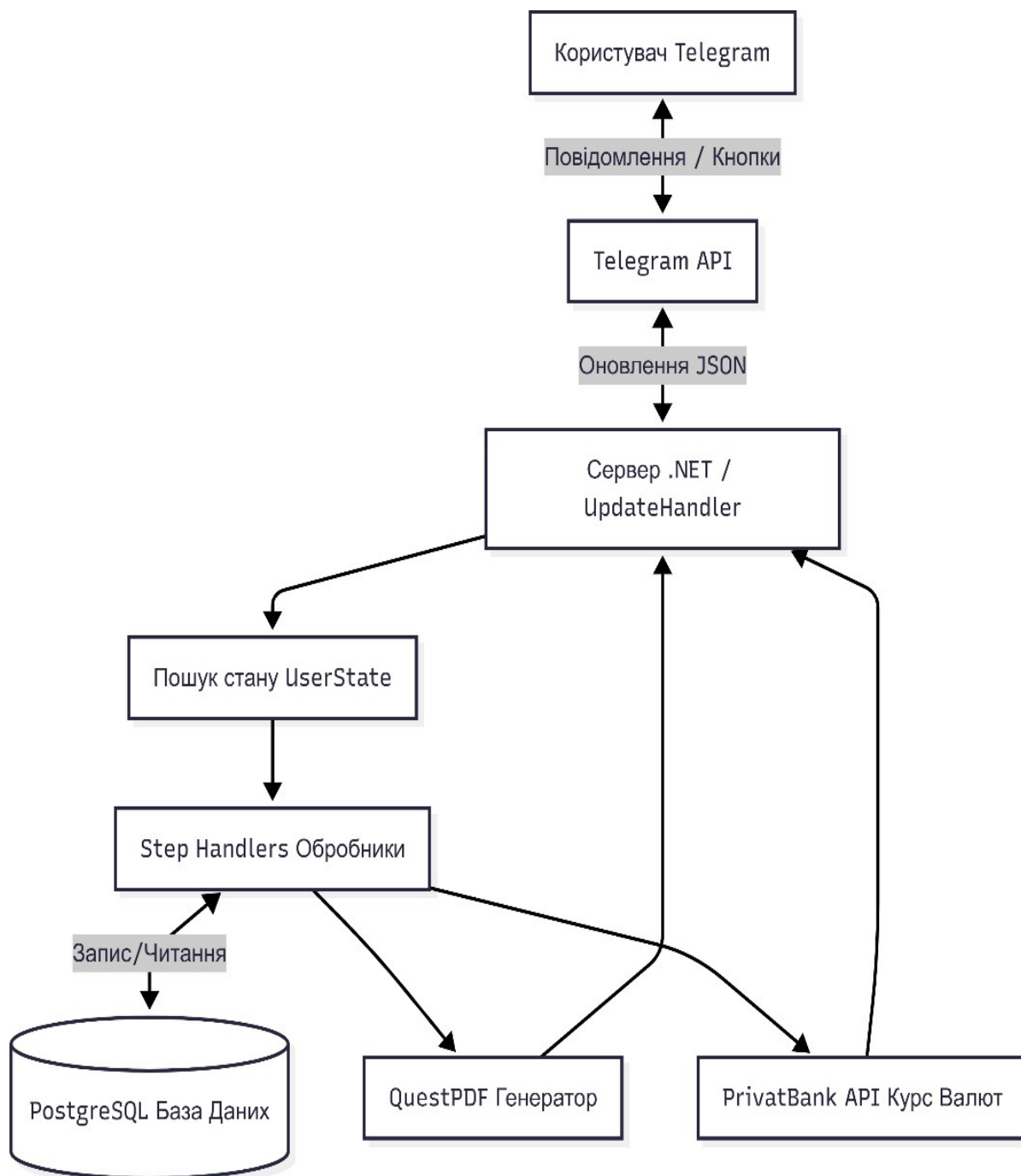


Рисунок 2.1 – Функціональна схема роботи системи

Джерело: розроблено автором

Процес обробки будь-якого вхідного повідомлення від користувача підпорядковується суворому алгоритму маршрутизації (рис. 2.2).

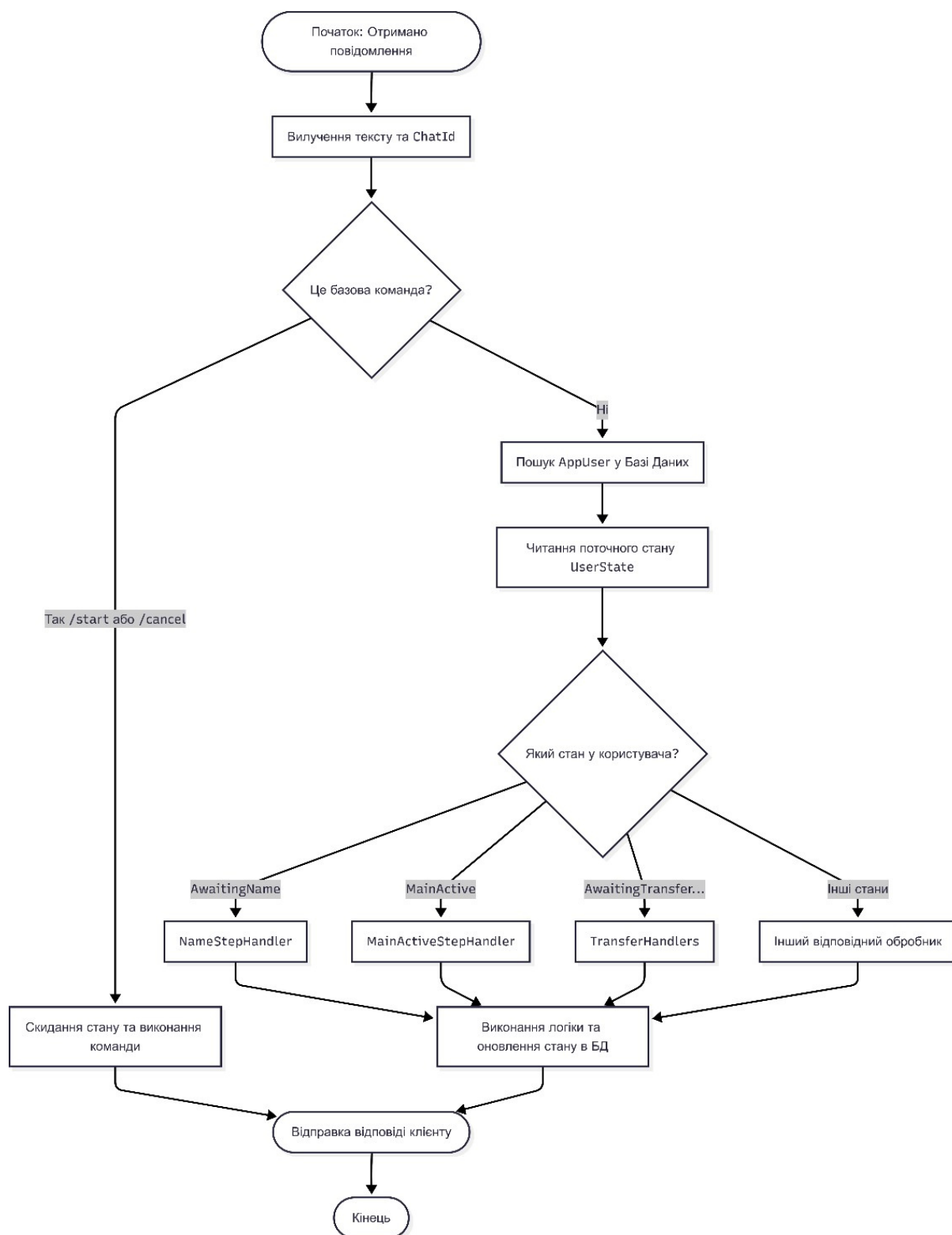


Рисунок 2.2 – Блок-схема алгоритму обробки повідомлень та маршрутизації

Джерело: розроблено автором

На відміну від класичних вебдодатків, ідентифікація у Telegram-ботах не потребує ручного введення логіна та пароля, що суттєво покращує користувацький досвід. Процес ідентифікації та реєстрації (рис. 2.3) відбувається

автоматично на основі унікального системного ідентифікатора чату (ChatId).

Під час кожного звернення система перевіряє наявність цього ідентифікатора у базі даних. Якщо запис відсутній, автоматично створюється новий профіль, а користувачу пропонується ввести своє ім'я.

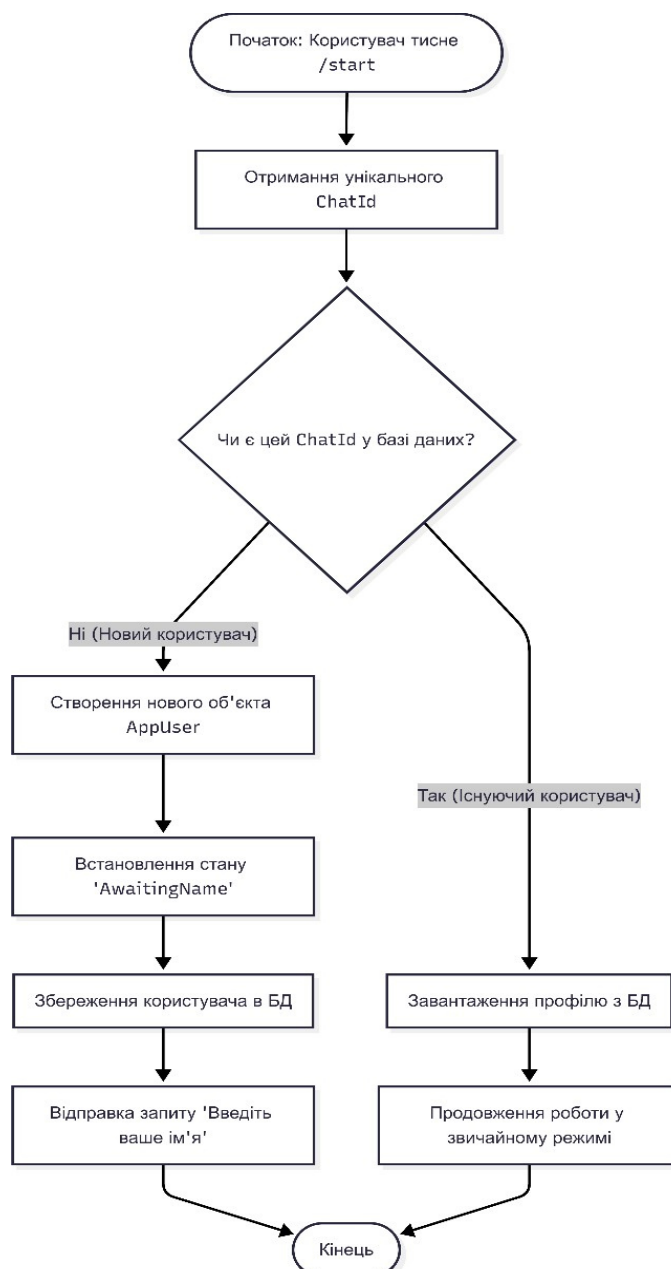


Рисунок 2.3 – Алгоритм ідентифікації та реєстрації за ChatId

Джерело: розроблено автором

Для розмежування доступу до панелі управління розроблено алгоритм перевірки адміністративних привілеїв (рис. 2.4). Перед виконанням специфічних

запитів виконується перевірка булевого поля IsAdmin у моделі профілю. У разі відсутності відповідних прав запит негайно блокується із попереджувальним повідомленням.

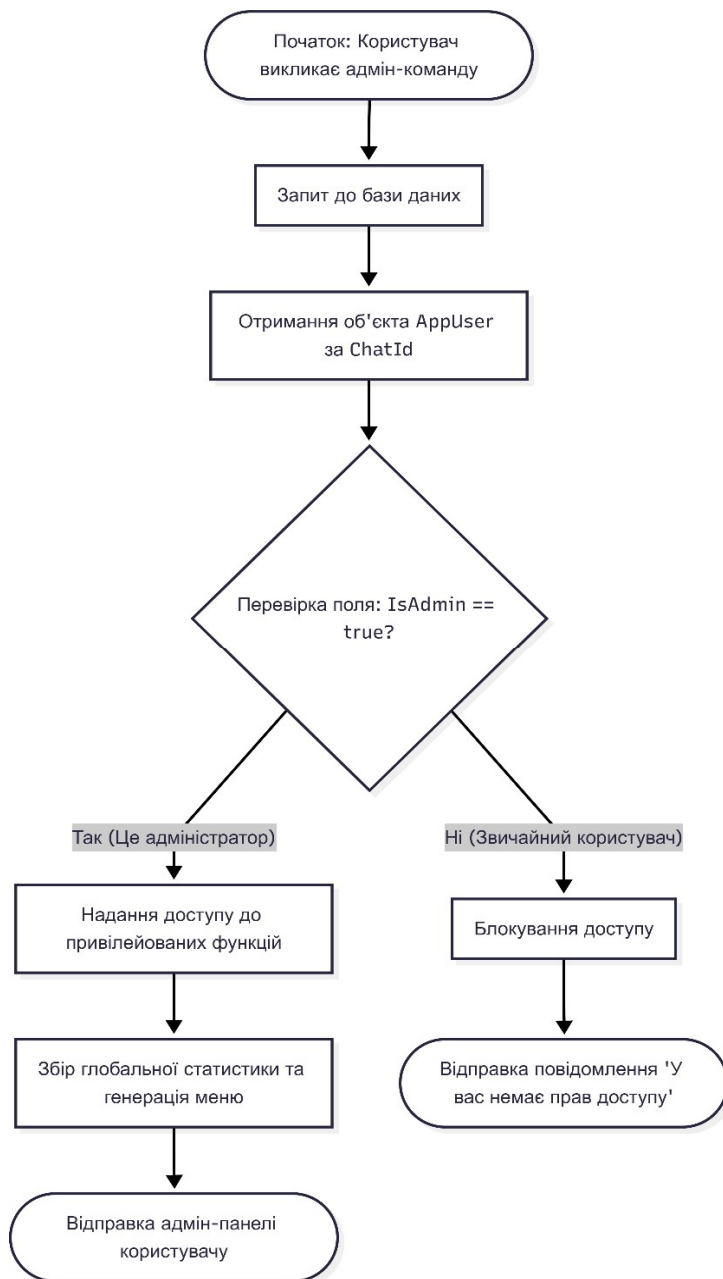


Рисунок 2.4 – Блок-схема алгоритму перевірки прав доступу

Джерело: розроблено автором

Для фундаментального розуміння структури даних була спроектована UML-діаграма класів (рис. 2.5). Вона наочно демонструє ключові сутності предметної області – класи AppUser та UserOperation, а також зв'язок

«один-до-багатьох» (1:N) між ними. Клас `AppUser` інкапсулює критичну інформацію про клієнта: його унікальний `ChatId`, поточний баланс гаманця, баланс віртуального сейфу, стан діалогу та адміністративні права. Клас `UserOperation` відповідає за збереження історії кожної транзакції та містить атрибути суми, текстової категорії, дати виконання та типу операції (дохід або витрата).

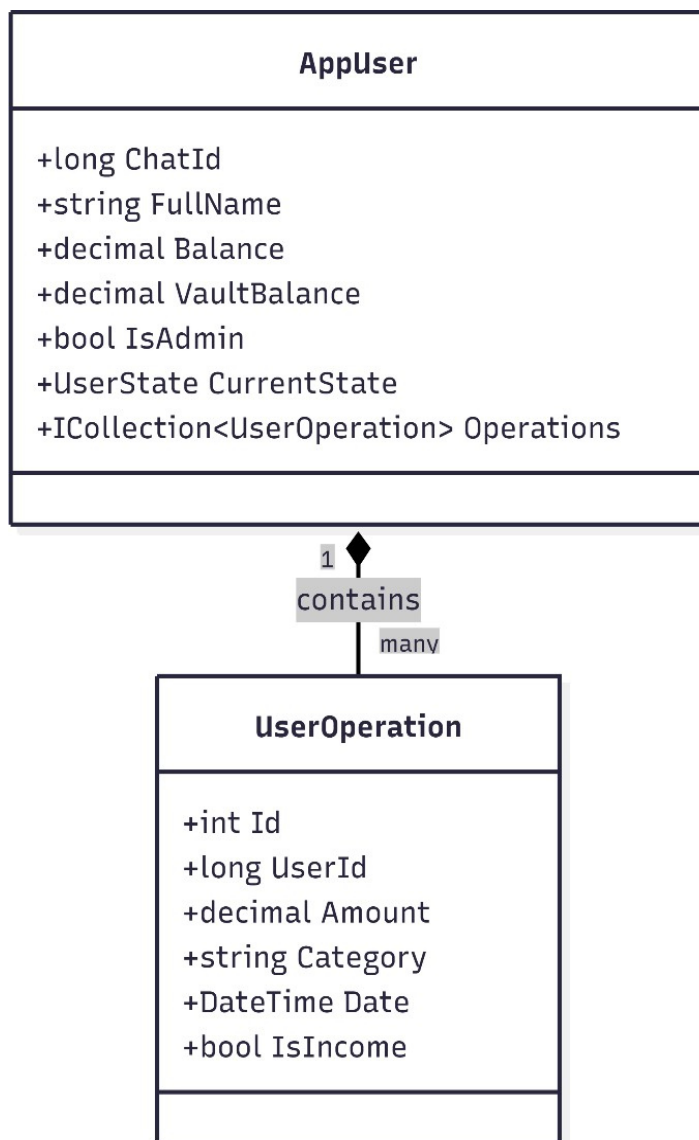


Рисунок 2.5 – UML-діаграма класів

Джерело: розроблено автором

Концептуальна модульність проекту та принцип розділення відповідальності проілюстровані на UML-діаграмі компонентів (рис. 2.6).

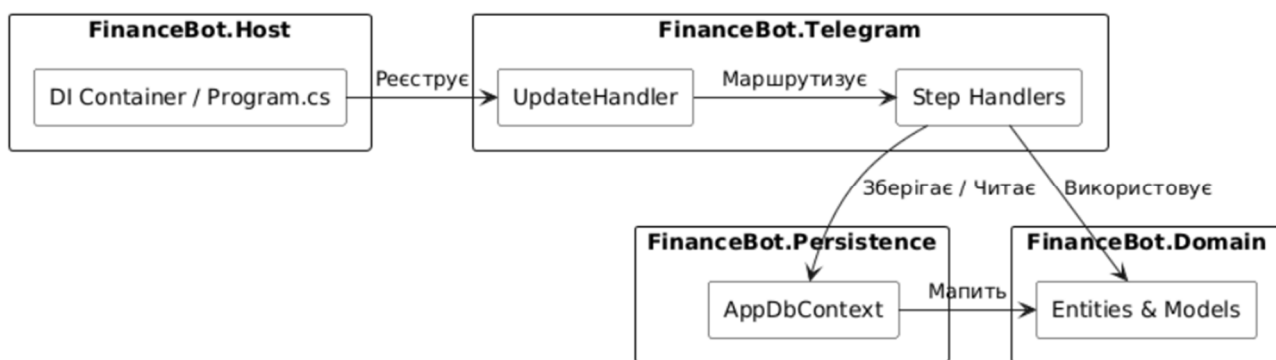


Рисунок 2.6 – UML-діаграма компонентів

Джерело: розроблено автором

Архітектура чітко розбита на незалежні шари: модуль Domain (містить моделі даних та перелічення), модуль Persistence (відповідає за підключення до PostgreSQL через AppDbContext), модуль Telegram (інкапсулює логіку обробників та візуальних кнопок) та модуль Host (виступає конфігураційною точкою входу та реєстрації залежностей). Такий поділ унеможливорює жорстку зв'язність коду і спрощує процес тестування.

2.3 Створення моделі бази даних

Проектування архітектури бази даних є фундаментальним етапом створення будь-якої фінансової інформаційної системи, адже саме від якості та продуманості моделі даних залежить цілісність капіталу користувачів, швидкість обробки транзакцій та загальна безпека застосунку. Коректно нормалізована база даних гарантує відсутність дублювання інформації та забезпечує безперебійну роботу при високих паралельних навантаженнях.

У рамках цієї кваліфікаційної роботи було розроблено логічну та фізичну реляційну модель даних, яка складається з двох основних сутностей: AppUser (профіль користувача) та UserOperation (фінансова транзакція).

Сутність AppUser виступає центральним елементом системи. На відміну від класичних вебдодатків, де первинним ключем є випадковий автоінкрементний ідентифікатор, у даній моделі роль первинного ключа (Primary

Key) виконує поле ChatId – унікальний ідентифікатор користувача в екосистемі Telegram. Це свідоме архітектурне рішення, яке суттєво пришвидшує пошук профілю під час надходження кожного нового повідомлення.

Модель також містить поля FullName (ім'я), Balance (баланс основного гаманця), VaultBalance (кошти у віртуальному сейфі), булевий прапорець IsAdmin та спеціальне перелічення CurrentState для збереження контексту кроків діалогу.

Сутність UserOperation призначена для детального протоколювання кожної фінансової дії. Вона включає власний ідентифікатор Id, зовнішній ключ UserId (який вказує на власника операції), числове значення Amount (сума), текстове поле Category (категорія витрати чи доходу), часову мітку Date та прапорець IsIncome (рис. 2.7).

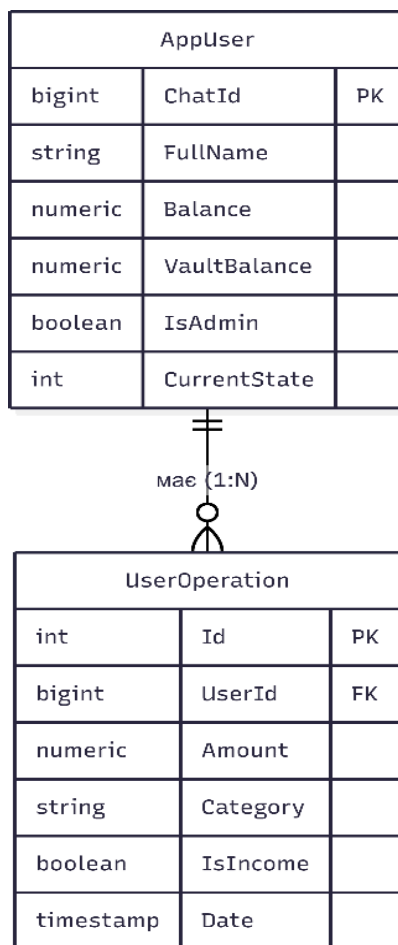


Рисунок 2.7 – Фізична структура БД

Джерело: розроблено автором

На етапі вибору системи управління базами даних (СКБД) стратегічно важливим кроком стала відмова від популярних NoSQL-рішень на користь класичної реляційної системи PostgreSQL. Такий вибір є абсолютно обґрунтованим з огляду на предметну область: фінансові сервіси критично залежать від транзакційної цілісності та строгої відповідності принципам ACID.

PostgreSQL є однією з найпотужніших та найбезпечніших СКБД у світі, що гарантує надійне збереження грошових записів навіть у разі раптових апаратних збоїв сервера чи конфліктів доступу (рис. 2.8).

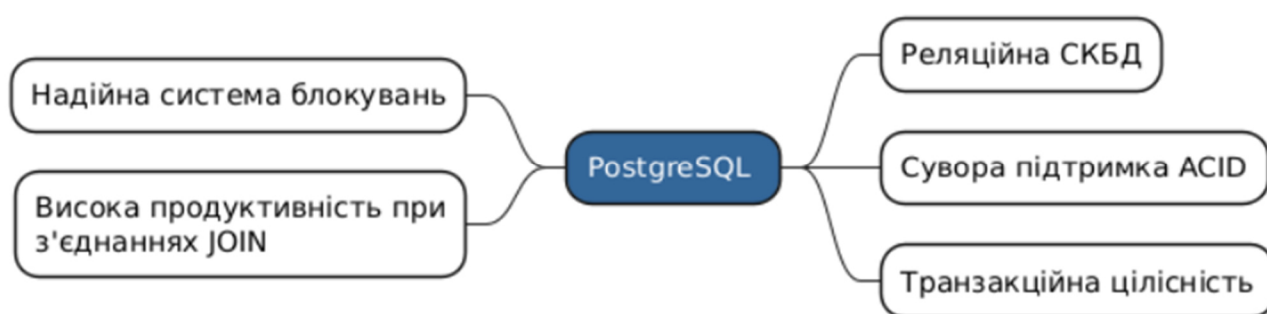


Рисунок 2.8 – Технічна характеристика PostgreSQL

Джерело: розроблено автором

Для забезпечення взаємодії між об'єктно-орієнтованим кодом на платформі .NET та таблицями PostgreSQL було застосовано передовий ORM-фреймворк Entity Framework Core (EF Core) у поєднанні з драйвером Npgsql.

Використання підходу «Code-First» (Код першим) дозволило описати структуру бази даних виключно за допомогою класів мови C#, після чого фреймворк автоматично згенерував безпечні SQL-міграції для створення таблиць.

EF Core виступає надійним абстрактним прошарком між бізнес-логікою та СКБД, дозволяючи розробнику виконувати всі CRUD-операції. Це не лише суттєво прискорює процес розробки, а й автоматично захищає бекенд від поширених вразливостей, зокрема від атак типу SQL-ін'єкцій (рис. 2.9).

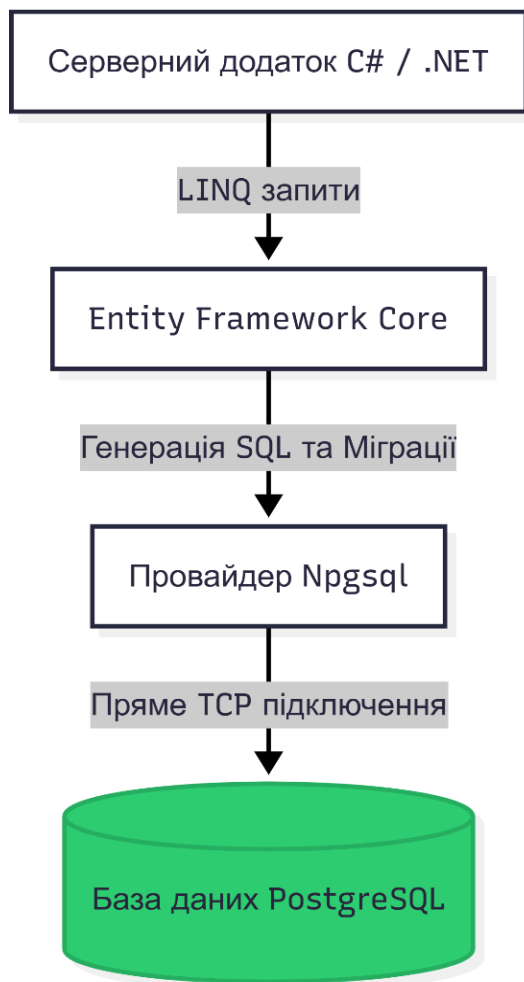


Рисунок 2.9 – Схема взаємодії EF Core з БД

Джерело: розроблено автором

Запропонована структура бази даних є максимально оптимізованою, продуктивною та готовою до легкої інтеграції додаткових таблиць у майбутньому (наприклад, для підтримки регулярних платежів чи цільових накопичень).

РОЗДІЛ 3

РОЗРОБКА АВТОМАТИЗОВАНОЇ СИСТЕМИ ОБЛІКУ ФІНАНСІВ

3.1 Практична реалізація об'єкта проєктування. Побудова блок-схем модулів програми

Сучасні програмні рішення на базі месенджерів зазвичай мають чітко розділену дворівневу архітектуру, де кожен рівень виконує специфічні функції для забезпечення стабільної та ефективної роботи. Ключовими складовими такої архітектури є клієнтський інтерфейс (у даному випадку – додаток Telegram) та серверна частина (бекенд), які тісно взаємодіють через захищений мережевий протокол. Клієнтська частина являє собою готовий користувацький інтерфейс месенджера Telegram, з яким безпосередньо взаємодіє кінцевий користувач. Її основне завдання полягає у візуалізації текстових повідомлень, кнопок та документів (PDF), отриманих від сервера, а також ініціюванні запитів (відправка повідомлень або натискання кнопок) у відповідь на дії користувача.

Серверна частина (бекенд) розміщується на віддаленому сервері та відповідає за обробку вхідних оновлень (Updates) від клієнтської частини, виконання основної фінансової бізнес-логіки, управління базою даних та забезпечення безпеки. Коли користувач надсилає повідомлення, сервери Telegram пересилають його на бекенд бота, який аналізує текст, виконує необхідні операції (наприклад, проведення транзакції чи генерацію звіту) та повертає відповідь.

Загальна архітектурна схема ілюструє основні елементи платформи та їхні функціональні зв'язки (рис. 3.1). Центральну роль у системі відіграє серверна частина на базі .NET, яка функціонує як фоновий сервіс (Background Service). Вона приймає об'єкти Update від Telegram API, після чого передає їх до головного маршрутизатора – UpdateHandler.

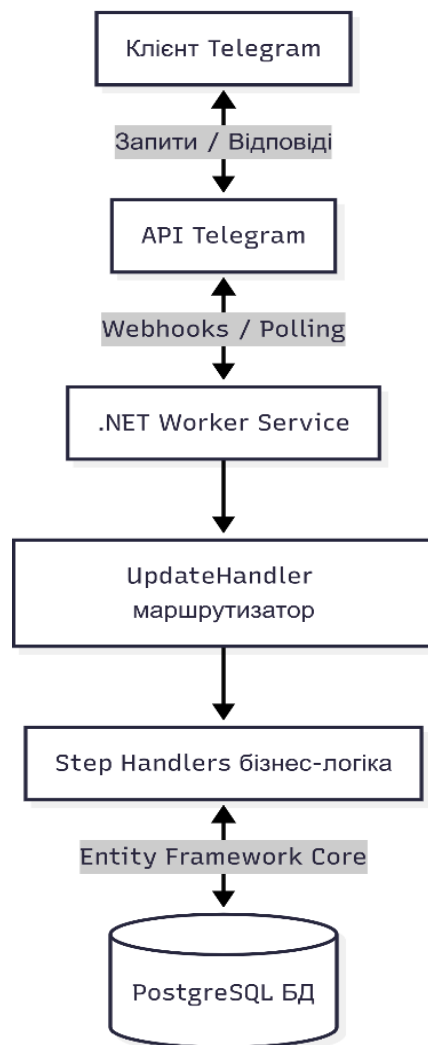


Рисунок 3.1 – Високорівнева архітектурна схема

Джерело: розроблено автором

На цьому етапі дослідження ми детально розглянемо внутрішню структуру бекенду розроблюваної системи. Розуміння архітектури та механізмів обробки станів є критично важливим для аналізу ефективності бота. Для забезпечення швидкої обробки запитів використовується патерн «State Machine» (кінцевий автомат), реалізований через систему обробників кроків (Step Handlers). Це дозволяє уникати складних багаторівневих перевірок (if/else) і передавати управління конкретному класу залежно від поточного стану користувача.

Використання асинхронних механізмів (async/await) у C# дозволяє системі підтримувати високу продуктивність без блокування потоків під час звернень до бази даних PostgreSQL.

Реалізація процесу обробки текстового запиту детально відображена у

вигляді схеми послідовної обробки. Користувач вводить суму або категорію витрати. Ці дані надсилаються до серверної частини.

Першим кроком є автоматична ідентифікація за ChatId та завантаження поточного стану (UserState). Після успішної перевірки сервер передає текст до відповідного обробника, який взаємодіє з базою даних, зберігає транзакцію та повертає клієнту підтвердження (рис. 3.2).

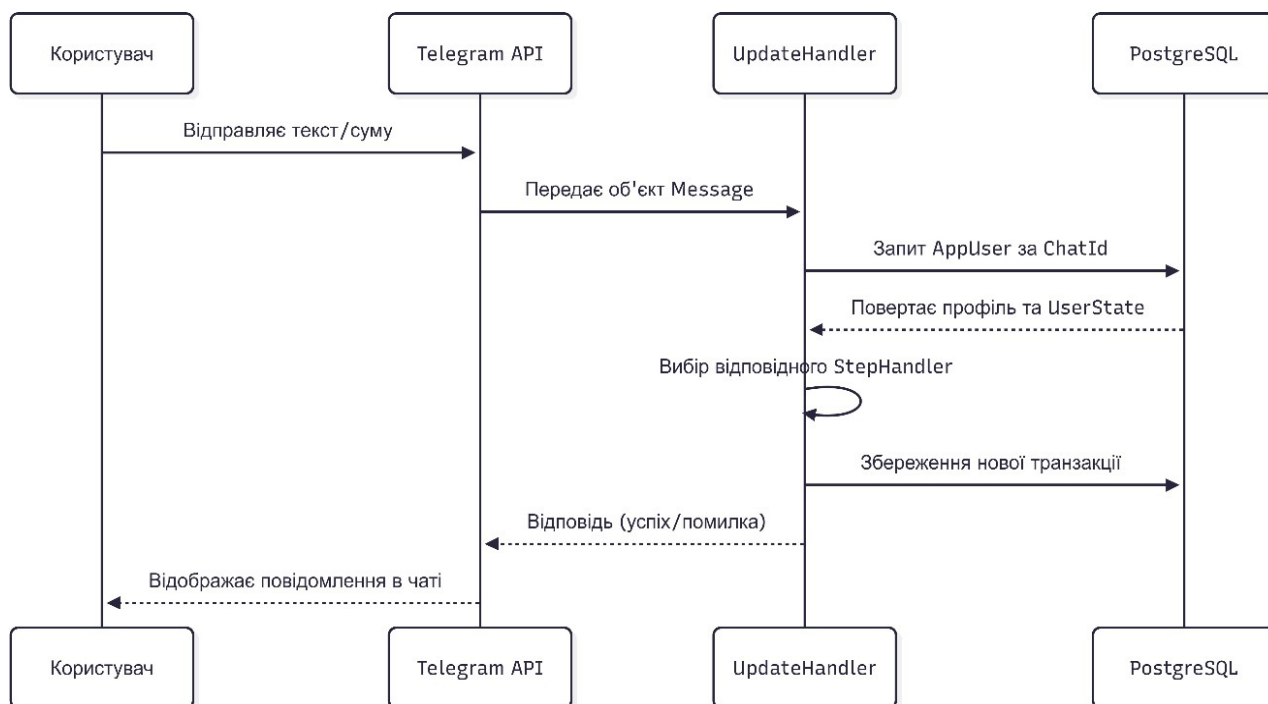


Рисунок 3.2 – Послідовність обробки запиту на пошук або дію

Джерело: розроблено автором

Структура даних у системі розроблена відповідно до принципів об'єктно-реляційного відображення (ORM) за допомогою Entity Framework Core, що забезпечує жорстку типізацію та надійність у зберіганні фінансової інформації. У рамках цього підходу спроектовано дві основні C# моделі: AppUser та UserOperation.

Модель AppUser (ліст. 3.1) використовується для управління профілями, збереження поточного балансу та розмежування прав доступу. Вона включає інформацію, необхідну для ідентифікації кожного облікового запису в межах Telegram.

Лістинг 3.1 – Модель даних користувача (AppUser)

```
public class AppUser
{
    [Key]
    public long ChatId { get; set; } public string? FullName { get; set; }
    public decimal Balance { get; set; }
    public decimal VaultBalance { get; set; } public bool IsAdmin { get; set; }
}
public UserState CurrentState { get; set; }
public ICollection<UserOperation> Operations { get; set; } = new
List<UserOperation>();
```

кінець лістингу 3.1

Модель UserOperation (ліст. 3.2) відповідає за організацію та зберігання історії фінансових транзакцій. Її структура дозволяє ефективно працювати з сумами, категоріями та датами, що є важливим аспектом для подальшої генерації звітів.

Лістинг 3.2 – Модель даних транзакції (UserOperation)

```
public class UserOperation
{
    [Key]
    public int Id { get; set; } public long UserId { get; set; }
    public decimal Amount { get; set; }
    public string Category { get; set; } = string.Empty; public DateTime Date
    { get; set; }
    public bool IsIncome { get; set; } [ForeignKey("UserId")]
    public AppUser User { get; set; } = null!;
```

кінець лістингу 3.2

Обидві моделі спроектовані з урахуванням нормалізації баз даних. Використання PostgreSQL у поєднанні з EF Core дозволяє мінімізувати накладні витрати на написання SQL-запитів, гарантуючи при цьому високу швидкість виконання транзакцій.

Для забезпечення контролю виконання операцій та діагностики можливих помилок у системі реалізовано механізм логування на основі вбудованого інтерфейсу ILogger<T> у платформі .NET. Логування є ключовим інструментом для моніторингу роботи бекенду та аналізу системних винятків.

Завдяки механізму логування адміністратори можуть швидко

ідентифікувати проблемні ситуації без втручання у базовий код. Приклад впровадження логера через ін'єкцію залежностей наведено у лістингу 3.3.

Лістинг 3.3 – Автоматична ідентифікація та реєстрація

```
public class UpdateHandler
{
    private readonly ILogger<UpdateHandler> _logger; private readonly
    IServiceProvider _serviceProvider;
    public UpdateHandler(ILogger<UpdateHandler> logger, IServiceProvider
    serviceProvider)
    {
        _logger = logger;
        _serviceProvider = serviceProvider;
    }
    // Подальша обробка логів: _logger.LogInformation("Отримано
    повідомлення...");
```

кінець лістингу 3.3

На відміну від класичних вебдодатків, які використовують JWT-токени та паролі, ідентифікація у Telegram-боті відбувається прозоро для користувача на основі унікального ідентифікатора чату (ChatId). Це повністю усуває необхідність у створенні сторінок реєстрації чи авторизації, значно підвищуючи рівень безпеки, оскільки пароль неможливо перехопити чи зламати. Кожного разу, коли сервер отримує нове повідомлення, маршрутизатор перевіряє наявність користувача у базі даних (ліст. 3.4). Якщо запис відсутній, автоматично створюється новий профіль, а користувачеві встановлюється стан AwaitingName (Очікування введення імені).

Лістинг 3.4 – Автоматична ідентифікація та реєстрація

```
var user = await db.AppUsers.FirstOrDefaultAsync(u => u.ChatId == chatId,
cancellationTokens);
if (user == null)
{user = new AppUser
{ChatId = chatId,
CurrentState = UserState.AwaitingName };
db.AppUsers.Add(user);
await db.SaveChangesAsync(cancellationTokens);
await botClient.SendMessageAsync(chatId, Вітаю! Введіть ваше ім'я:",
cancellationTokens: cancellationTokens); return;
```

кінець лістингу 3.4

Для захисту адміністративних функцій (наприклад, перегляд загальної статистики бота) використовується механізм перевірки прав доступу на основі властивості `IsAdmin`. У лістингу 3.5 наведено фрагмент обробника адміністративної панелі, який відхиляє запит, якщо користувач не наділений відповідними правами.

Лістинг 3.5 – Перевірка прав адміністратора (AdminHandler)

```
public class AdminHandler : IStepHandler
{
    public async Task HandleStepAsync(ITelegramBotClient botClient, Message
message, AppUser user, AppDbContext db, CancellationToken
cancellationToken)
    {
        if (!user.IsAdmin)
        {
            await botClient.SendTextMessageAsync(user.ChatId, "+ У вас немає доступу
до панелі адміністратора.", cancellationToken: cancellationToken);
            return;
        }
        // Відображення глобальної статистики...
        return;
    }
}
```

кінець лістингу 3.5

Центральним елементом управління бізнес-логікою у додатку є механізм маршрутизації команд на основі патерну «Ланцюжок обов’язків» (Chain of Responsibility) та поліморфізму. Цей підхід дозволяє легко додавати нові функції без зміни існуючого коду (дотримуючись принципу відкритості/закритості з SOLID). У лістингу 3.6 наведено структуру цього інтерфейсу, який зобов’язує кожен клас-обробник реалізувати метод перевірки можливості обробки `CanHandle` та безпосередньо метод виконання логіки `HandleStepAsync`.

Лістинг 3.6 – Інтерфейс обробника станів (IStepHandler)

```
public interface IStepHandler
{
    //Метод визначає, чи повинен цей клас обробляти поточний стан користувача
    bool CanHandle(UserState state, string text);
    // Асинхронний метод виконання бізнес-логіки Task HandleStepAsync(
ITelegramBotClient botClient, Message message,
AppUser user, AppDbContext db,
CancellationToken cancellationToken);
    return;
}
```

кінець лістингу 3.6

У лістингу 3.7 роль головної сторінки (або навігаційного меню Navbar) у класичному вебзастосунку тут виконує MainActiveStepHandler. Оскільки в месенджері відсутній HTML/CSS інтерфейс, взаємодія з користувачем будується за допомогою спеціальної клавіатури ReplyKeyboardMarkup.

Лістинг 3.7 – Генерація головного меню (MainActiveStepHandler)

```
public class MainActiveStepHandler : IStepHandler
public bool CanHandle(UserState state, string text) => state ==
UserState.MainActive;
public async Task HandleStepAsync(ITelegramBotClient botClient, Message
message, AppUser user, AppDbContext db, CancellationToken
cancellation_token)
// Створення масиву кнопок для візуального інтерфейсу
var keyboard = new ReplyKeyboardMarkup(new[]
new KeyboardButton[] { "Дохід", "Витрата" },
new KeyboardButton[]
{ "Сейф", "Звіт" },
new KeyboardButton[] { "Курс валют" },
new KeyboardButton[] { "Налаштування", "Адмін" }
}) { ResizeKeyboard = true };
string responseText = $"Головне меню.\nВаш баланс: {user.Balance}
€\nБаланс сейфу: {user.VaultBalance} €";
await botClient.SendTextMessageAsync(user.ChatId, responseText,
return;
```

кінець лістингу 3.7

Однією з найважливіших функцій системи є облік фінансів. Логіка додавання доходу чи витрати реалізована через багатоетапний діалог. Коли користувач натискає кнопку «Витрата», система переводить його у стан очікування суми. У лістингу 3.8 показано, як IncomeExpenseHandler здійснює перевірку валідності введених числових даних (валідація типу decimal). Якщо дані коректні, система створює новий запис у таблиці UserOperation, оновлює загальний баланс і повертає користувача до головного меню.

Лістинг 3.8 – Обробка фінансової транзакції

```
if (decimal.TryParse(message.Text, out decimal amount) && amount > 0)
{
// Створення нової операції
var operation = new UserOperation
{
```

```

UserId = user.ChatId, Amount = amount,
IsIncome = false, // false означає витрату Category = "Загальне",
Date = DateTime.UtcNow
};
// Оновлення балансу користувача user.Balance -= amount;
user.CurrentState = UserState.MainActive; // Повернення в головне меню
db.UserOperations.Add(operation);
await db.SaveChangesAsync(cancellationToken);
await botClient.SendTextMessageAsync(user.ChatId, $"Витрату на
{amount} ₴ успішно додано!", cancellationToken: cancellationToken);
else
await botClient.SendTextMessageAsync(user.ChatId, "+ Некоректна сума.
Введіть число більше нуля.", cancellationToken: cancellationToken);
return;

```

кінець лістингу 3.8

Система також підтримує функцію Віртуального сейфу для накопичення коштів. Логіка роботи сейфу полягає в ізолюваному переміщенні коштів між полем Balance та VaultBalance у межах одного профілю (ліст. 3.9).

Це вимагає строгої перевірки наявності достатньої суми на рахунку для запобігання від'ємному балансу (овердрафту). Усі операції зміни балансу обертаються у транзакції Entity Framework Core для забезпечення безпеки.

Лістинг 3.9 – Бізнес-логіка віртуального сейфу (VaultHandler)

```

if (decimal.TryParse(message.Text, out decimal amount) && amount > 0)
depositAmount, AppDbContext db)
{
if (depositAmount <= 0)
throw new ArgumentException("Сума поповнення повинна бути більшою за
нуль.");
if (user.Balance < depositAmount)
throw new InvalidOperationException("Недостатньо коштів на основному
балансі.");
// Транзакційне переміщення коштів user.Balance -= depositAmount;
user.VaultBalance += depositAmount; user.CurrentState return;

```

кінець лістингу 3.9

Сучасна фінансова система не може існувати в ізоляції, тому архітектура бота передбачає інтеграцію зі сторонніми API. Для отримання актуальних курсів валют (USD та EUR) реалізовано звернення до відкритого публічного API ПриватБанку. Використання класу HttpClient дозволяє виконувати асинхронні

HTTP-запити без блокування основного потоку виконання бота (ліст. 3.10).

Отримана JSON-відповідь десеріалізується у строго типізований об'єкт `PrivatBankRate` за допомогою бібліотеки `System.Text.Json`.

Лістинг 3.10 – Асинхронний запит до API ПриватБанку

```
public async Task<string> GetCurrencyRatesAsync()
{
    using var httpClient = new HttpClient(); try
    {
        var response = await
        httpClient.GetAsync("https://api.privatbank.ua/p24api/pubinfo?exchange
        &coursid=5");
        response.EnsureSuccessStatusCode();
        var jsonString = await response.Content.ReadAsStringAsync();
        var rates = JsonSerializer.Deserialize<List<PrivatBankRate>>(jsonString);
        var usd = rates?.FirstOrDefault(r => r.Ccy == "USD");
        var eur = rates?.FirstOrDefault(r => r.Ccy == "EUR");
        return $"us USD:
        Купівля {usd?.Buy} / Продаж {usd?.Sale}\n" +
        $"eu EUR: Купівля {eur?.Buy} / Продаж {eur?.Sale}";
        user.VaultBalance += depositAmount; user.CurrentState return;
    }
}
```

кінець лістингу 3.10

Фінальним, і найбільш технологічно складним елементом системи є генерація аналітичних звітів. Замість виведення інформації у вигляді простого тексту, бот використовує бібліотеку `QuestPDF` для динамічного формування документів у форматі PDF (ліст. 3.11). Ця бібліотека працює на основі `Fluent API` та дозволяє програмно малювати таблиці, графіки та текстові блоки.

Згенерований документ не зберігається на жорсткому диску сервера (що економить пам'ять), а відразу перетворюється у потік байтів (`MemoryStream`) та надсилається клієнту в Telegram як вкладений файл (`InputOnlineFile`).

Лістинг 3.11 – Динамічна генерація та відправка PDF-звіту

```
public async Task GenerateAndSendPdfReport(ITelegramBotClient botClient,
    AppUser user, CancellationToken cancellationToken)
{
    // Формування документа через Fluent API QuestPDF
    var document = Document.Create(container =>
    {
        container.Page(page =>
```

```

{
page.Size(PageSizes.A4); page.Margin(2, Unit.Centimetre);
page.Header().Text($"Фінансовий звіт для
{user.FullName}").SemiBold().FontSize(20);
page.Content().Text($"Ваш поточний баланс: {user.Balance} грн");
});
});
// Збереження у пам'ять
using var stream = new MemoryStream();
document.GeneratePdf(stream); stream.Position = 0;
// Відправка файлу у Telegram
var inputFile = InputFile.FromStream(stream,
"Financial_Report.pdf"); await botClient.SendDocumentAsync(user.ChatId,
inputFile, caption:"Ваш детальний звіт cancellationToken:
cancellationToken);
user.VaultBalance += depositAmount; user.CurrentState return;

```

кінець лістингу 3.11

Уся описана вище обробка запитів інкапсульована в архітектуру сервісу, що гарантує високу стабільність. Обробка можливих виняткових ситуацій (наприклад, втрата з'єднання з базою даних) реалізована через глобальні блоки try-catch у точці входу UpdateHandler, що унеможливорює раптове завершення роботи бота.

Цей функціонал реалізований за допомогою спеціалізованого обробника і вимагає дотримання суворих правил транзакційності бази даних (принципів ACID).

Під час P2P-переказу система повинна атомарно списати кошти з балансу відправника та зарахувати їх на баланс отримувача. Якщо будь-який з цих етапів завершується помилкою, уся транзакція скасовується, щоб уникнути втрати або безпідставного створення (дублювання) фінансів

У лістингу 3.12 наведено логіку валідації та виконання такого переказу.

Лістинг 3.12 – Логіка транзакційного переказу коштів (TransferHandler)

```

if (decimal.TryParse(message.Text, out decimal transferAmount) &&
transferAmount > 0)
{
// Перевірка платоспроможності відправника
if (user.Balance < transferAmount)
{
await botClient.SendTextMessageAsync(user.ChatId, "+ Недостатньо коштів
для переказу.");
return;
}

```

```
// Пошук отримувача у базі даних
var targetUser = await db.Users.FirstOrDefaultAsync(u => u.ChatId ==
user.TransferTargetId);
if (targetUser == null)
// Транзакційне оновлення балансів двох користувачів
user.Balance -= transferAmount; targetUser.Balance += transferAmount;
// Створення дзеркальних записів про операції для збереження історії
db.Operations.Add(new UserOperation { UserChatId = user.ChatId, Amount
= transferAmount, IsExpense = true, Details = "Вихідний переказ" });
db.Operations.Add(new UserOperation { UserChatId = targetUser.ChatId,
Amount = transferAmount, IsExpense = false, Details = "Вхідний переказ"
});var inputFile = InputFile.FromStream(stream, "Financial_Report.pdf");
await botClient.SendDocumentAsync(user.ChatId, inputFile,
caption:"Вашuser.VaultBalance += depositAmount; user.CurrentState
return;
```

кінець лістингу 3.12

Ще однією важливою вимогою до системи є надання користувачеві доступу до ретроспективного аналізу своїх фінансів. Окрім складної генерації PDF-звітів, бот надає можливість швидкого перегляду останніх транзакцій безпосередньо у вікні чату. Для реалізації цього функціоналу розроблено HistoryHandler, здійснює оптимізований запит до таблиці UserOperations.

Система завантажує в оперативну пам'ять сервера лише лімітовану кількість останніх записів, що унеможливорює перевантаження сервера та витік пам'яті при стрімкому зростанні обсягів даних (ліст. 3.13).

Лістинг 3.13 – Формування виписки останніх транзакцій (HistoryHandler)

```
public async Task HandleStepAsync(ITelegramBotClient botClient, Message
message, AppUser user, AppDbContext db, CancellationToken ct) {
var recentOperations = await db.Operations .Where(o => o.UserChatId ==
user.ChatId).OrderByDescending(o => o.CreatedAt).Take(10)
.ToListAsync(ct);
if (!recentOperations.Any())
{await botClient.SendTextMessageAsync(user.ChatId, "Ваша історія
операцій порожня.");return;}
var historyText = "Останні 10 операцій:\n\n";
foreach (var op in recentOperations)
{var icon = op.IsExpense ? "🔴" : "🟢"; var sign = op.IsExpense ? "-" : "+";
historyText += $"{icon} {op.CreatedAt:dd.MM.yyyy HH:mm} |
{op.Category}\n" + $"Сума: {sign}{op.Amount} ₴\n\n";}
await botClient.SendTextMessageAsync(user.ChatId, historyText, parseMode:
ParseMode.Markdown, cancellationTokens: ct); return;
```

кінець лістингу 3.13

Нарешті, забезпечення безпеки на етапі ініціалізації сесії реалізовано через механізм підтвердження секретного коду. У лістингу 3.14 ідентифікація відбувається за ChatId, для додаткового захисту під час першого запуску або відновлення доступу користувач повинен ввести спеціальний PIN-код (SecretCodeStepHandler). Цей підхід додає другий фактор перевірки.

Лістинг 3.14 – Валідація секретного коду безпеки (SecretCodeStepHandler)

```
public async Task HandleStepAsync(ITelegramBotClient botClient, Message
message, AppUser user, AppDbContext db, CancellationToken ct){
var inputCode = message.Text?.Trim();
if (string.IsNullOrEmpty(user.SecretCode)){
user.SecretCode = inputCode; user.CurrentState = UserState.MainActive;
await db.SaveChangesAsync(ct);
await botClient.SendTextMessageAsync(user.ChatId, "Секретний код успішно
встановлено!");}
else if (user.SecretCode == inputCode){
user.CurrentState = UserState.MainActive; await db.SaveChangesAsync(ct);
await botClient.SendTextMessageAsync(user.ChatId, "👉 Доступ дозволено.
Вітаємо у системі!");} else{
await botClient.SendTextMessageAsync(user.ChatId, "+ Невірний секретний
код. Спробуйте ще раз.");}}
```

кінець лістингу 3.14

Комплексне застосування патерну «Кінцевий автомат» у поєднанні з надійною реляційною базою даних та суворою типізацією мови C# дозволило створити не просто чат-бота, а повноцінне масштабоване серверне рішення.

Кожен описаний модуль системи є функціонально незалежним (слабка зв'язність компонентів), що забезпечує легкість тестування та можливість безшовного додавання нових фінансових функцій у майбутньому без ризику порушення цілісності існуючої архітектури.

Для того щоб усі описані вище модулі та обробники могли ефективно взаємодіяти між собою, архітектура системи побудована на фундаменті інверсії управління (Inversion of Control) за допомогою вбудованого у .NET механізму впровадження залежностей (Dependency Injection). Це дозволяє уникнути жорсткого зв'язування класів. У лістингу 3.15 продемонстровано конфігурацію контейнера залежностей у головному файлі програми. Сервіси бази даних та

клієнт Telegram реєструються з життєвим циклом Scoped або Singleton, а всі реалізації інтерфейсу IStepHandler автоматично додаються до колекції сервісів для подальшої маршрутизації.

Лістинг 3.15 – Конфігурація контейнера впровадження залежностей

```
var builder = Host.CreateDefaultBuilder(args);
builder.ConfigureServices((context, services) =>{
    services.AddDbContext<AppDbContext>(options =>
        options.UseNpgsql(context.Configuration.GetConnectionString("DefaultCo
nnection")));
    var botToken = context.Configuration["TelegramBotToken"];
    services.AddSingleton<ITelegramBotClient>(new
        TelegramBotClient(botToken));
    services.AddScoped<UpdateHandler>();
    // Реєстрація всіх обробників станів services.AddScoped<IStepHandler,
    MainActiveStepHandler>(); services.AddScoped<IStepHandler,
    IncomeExpenseHandler>(); services.AddScoped<IStepHandler,
    TransferHandler>(); services.AddScoped<IStepHandler, HistoryHandler>();
    services.AddScoped<IStepHandler, SecretCodeStepHandler>();
    services.AddScoped<IStepHandler, NameStepHandler>();
    services.AddScoped<IStepHandler, BalanceStepHandler>(); });
var host = builder.Build(); await host.RunAsync();
```

кінець лістингу 3.15

Серцем збереження даних системи є контекст бази даних (Database Context). Завдяки технології Entity Framework Core, робота з таблицями бази даних PostgreSQL абстрагується до роботи зі звичайними колекціями C# типу DbSet. У лістингу 3.16 показано структуру класу AppDbContext, який виступає мостом між об'єктно-орієнтованим кодом та реляційною базою даних. Він автоматично конвертує LINQ-запити у високоефективні SQL-команди.

Лістинг 3.16 – Контекст бази даних Entity Framework Core (AppDbContext)

```
namespace FinanceBot.Persistence; public class AppDbContext : DbContext{
    public AppDbContext(DbContextOptions<AppDbContext> options) :
    base(options){
        Database.EnsureCreated();}
    public DbSet<AppUser> Users { get; set; }
    public DbSet<UserOperation> Operations { get; set; }
    protected override void OnModelCreating(ModelBuilder modelBuilder){
        base.OnModelCreating(modelBuilder);
        modelBuilder.Entity<UserOperation>()
```

```
HasOne(o => o.User) WithMany(u => u.Operations)
HasForeignKey(o => o.UserChatId) OnDelete(DeleteBehavior.Cascade);}}
```

кінець лістингу 3.16

Однією з найчастіших операцій, які виконує користувач, є перевірка поточного стану своїх фінансів. Замість складних запитів до серверів, архітектура бота передбачає зберігання актуального балансу безпосередньо у профілі користувача (кешування балансу). Це значно прискорює видачу інформації. `BalanceStepHandler` (ліст. 3.17) відповідає за форматування та виведення фінансових показників.

Лістинг 3.17 – Обробник перевірки поточного балансу

```
{
public bool CanHandle(Message message, AppUser user)
=> message.Text == Мій баланс" && user.Status == UserState.Active;
public async Task HandleAsync(Message message, AppUser user,
Cancellation token ct)
{
var response = $"*Ваша фінансова зведеність:* \n\n" +
$" Основний рахунок: *{user.Balance:N2} €* \n" +
"У сейфі: *{user.VaultBalance:N2} €* \n\n" +
$"Загальний капітал: *{(user.Balance + user.VaultBalance):N2} €*";
// Відправка повідомлення з використанням Markdown для виділення жирним
await botClient.SendTextMessageAsync( chatId: user.ChatId,
text: response,
parseMode: ParseMode.Markdown, cancellationToken: ct);
}
}
```

кінець лістингу 3.17

Коректна робота всіх вищеописаних функцій неможлива без правильної первинної ініціалізації профілю. Коли новий користувач вперше запускає бота, система присвоює йому стан `AwaitingName`. На цьому етапі відбувається валідація введених даних. Як показано у лістингу 3.18, система блокує спроби ввести порожнє ім'я або занадто короткий текст. Після успішної перевірки система зберігає ім'я, переводить користувача на етап встановлення секретного коду та фіксує зміни у базі даних.

Лістинг 3.18 – Обробник первинної реєстрації (NameStepHandler)

```

public class NameStepHandler : IStepHandler
{
    public bool CanHandle(Message message, AppUser user)
    => user.Status == UserState.AwaitingName;
    public async Task HandleAsync(Message message, AppUser user,
        CancellationToken ct)
    {
        var name = message.Text?.Trim();
        // Валідація вводу на порожній рядок
        if (string.IsNullOrEmpty(name) || name.Length < 2)
        {
            await botClient.SendTextMessageAsync(user.ChatId, "+ Ім'я занадто коротке. Будь ласка, введіть коректне ім'я:");
            return;
        }
        // Збереження імені та перехід до наступного етапу реєстрації
        user.Name = name;
        user.Status = UserState.AwaitingSecretCode; await
        db.SaveChangesAsync(ct);
        await botClient.SendTextMessageAsync(user.ChatId,
    }

```

кінець лістингу 3.18

Архітектурна модульність розробленої системи дозволяє ізолювати логіку кожного етапу взаємодії користувача з ботом у межах окремого класу. Це не лише спрощує читабельність програмного коду, але й дозволяє масштабувати проєкт, забезпечуючи високу швидкість обробки навіть при пікових навантаженнях на сервер.

3.2 Тестування та налагодження розробленого Telegram-бота

Процес тестування та налагодження є критично важливим етапом розробки програмного забезпечення, що гарантує його стабільність, відмовостійкість та коректність виконання фінансових операцій. Оскільки розроблений Telegram-бот працює з реальними даними та віртуальними фінансами користувачів, будь-які збої в логіці можуть призвести до спотворення розрахунків балансу.

Для ефективного моніторингу функціонування застосунку та швидкого

виявлення потенційних несправностей у даному проєкті використовується централізований механізм реєстрації подій (логування), реалізований за допомогою вбудованого інтерфейсу ILogger платформи .NET Core. Система логуювання налаштована для фіксації інформації з різними рівнями деталізації (Information, Warning, Error, Critical).

Під час тестування встановлюється підвищений рівень деталізації (Debug), що забезпечує фіксацію максимально вичерпної інформації про хід виконання коду: надходження нових об'єктів Update від Telegram API, переходи між станами користувача (UserState) та SQL-запити, що автоматично генеруються Entity Framework Core. Це дає змогу розробнику відстежувати внутрішній стан програми та перевіряти відповідність її поведінки очікуванням.

У разі виникнення непередбачених ситуацій (наприклад, тимчасова відсутність зв'язку з базою даних PostgreSQL або помилка під час серіалізації відповіді від API ПриватБанку), система автоматично реєструє подію з рівнем важливості Error. Завдяки цьому можна швидко отримати повний контекст проблеми: точний час збою, тип винятку (Exception) та стек викликів функцій (Stack Trace). Це значно спрощує процес локалізації несправностей, оскільки усувається потреба відтворювати помилку наосліп.

На рисунку 3.3 відображено приклад консольного виводу в середовищі розробки, який фіксує успішний запуск фонових сервісів бота та реєстрацію вхідних запитів від клієнтів.

```
C:/Users/www/RiderProjects/Financial.Bot/FinanceBot.Host/bin/Debug/net9.0/FinanceBot.Host.exe
=== Smart Finance System активована (ASP.NET Background Service) ===
=== Smart Finance System активована (ASP.NET Background Service) ===
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Production
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Users\www\RiderProjects\Financial.Bot\FinanceBot.Host\bin\Debug\net9.0
```

Рисунок 3.3 – Консольне логуювання роботи бекенду

Джерело: розроблено автором

На відміну від класичних вебзастосунків, де для тестування API використовуються інструменти на кшталт Postman або Thunder Client, тестування Telegram-бота має свою специфіку. Основним клієнтом для тестування виступає офіційний додаток Telegram (Desktop або Mobile).

Розробник безпосередньо взаємодіє з ботом через графічний інтерфейс месенджера: надсилає базові команди (наприклад, /start), тестує коректність відображення кнопок ReplyKeyboardMarkup та перевіряє механізми валідації вводу (наприклад, реакцію системи на спробу ввести текстові символи замість суми витрати). Такий підхід дозволяє перевірити не лише серверну логіку маршрутизації, але й зручність користувацького інтерфейсу (UX) на реальних пристроях кінцевих споживачів.

На рівні серверної частини процес глибокого налагодження (дебагінгу) здійснюється за допомогою потужного інтегрованого середовища розробки JetBrains Rider. Використання інструментів дебагінгу Rider дозволяє детально аналізувати виконання коду в режимі реального часу. Встановлення точок зупинки (Breakpoints) у головному класі UpdateHandler або в конкретних обробниках кроків (Step Handlers) дає можливість призупинити виконання програми в точний момент отримання повідомлення від користувача. Це дозволяє перевірити поточне значення змінних, проаналізувати структуру об'єкта Message та простежити формування сутності AppUser перед її збереженням у базу даних. Такий підхід є значно ефективнішим за звичайне візуальне логування, оскільки дає можливість точково визначити джерело логічної помилки.

Окрім налагодження безпосередньо коду C#, невід'ємним етапом є тестування цілісності бази даних. Для перевірки консистентності даних після виконання транзакцій (наприклад, під час переказу коштів з основного балансу до віртуального Сейфу) використовуються інструменти адміністрування БД (вбудований Database Explorer у Rider або pgAdmin). Це дозволяє візуально та за допомогою SQL-запитів підтвердити, що баланс користувача змінився математично правильно.

3.3 Інструкція користувача з інсталяції та використання програмного продукту

Для успішної інсталяції, розгортання та коректної роботи Telegram-бота на локальній машині чи віртуальному сервері необхідне дотримання наступних системних вимог:

- операційна система: Windows 10/11, macOS або дистрибутиви Linux (Ubuntu, Debian тощо);
- встановлене середовище виконання .NET 8.0 SDK (або новіша версія);
- встановлений сервер реляційних баз даних PostgreSQL (локально або з доступом до хмарного кластера);
- встановлений Git-клієнт для завантаження вихідного коду з репозиторію;
- стабільне підключення до Інтернету для завантаження NuGet-пакетів та зв'язку з серверами Telegram API.

Розроблений програмний продукт являє собою модульне серверне рішення (backend), побудоване з використанням архітектурного підходу розділення обов'язків (Clean Architecture). Проєкт не має класичного фронтенду, оскільки функцію клієнтського інтерфейсу виконує сам месенджер Telegram.

Інсталяція здійснюється у кілька етапів. Перед початком роботи переконайтеся, що на системі інсталювано .NET SDK та PostgreSQL. Для отримання коду відкрийте термінал і виконайте команду `git clone https://github.com/yeluzaveta13/Financial.Bot`. Після успішного завантаження перейдіть у кореневу директорію рішення (Solution).

Наступним кроком є встановлення усіх необхідних програмних залежностей (бібліотек Entity Framework Core, Telegram.Bot, QuestPDF). Для цього у кореневій папці необхідно виконати команду `dotnet restore`. Платформа .NET автоматично проаналізує файли проєктів (.csproj) та завантажить необхідні пакети з глобального репозиторію NuGet.

РОЗДІЛ 4

СПЕЦІАЛЬНІ РОЗРАХУНКИ

У класичних вебзастосунках основними векторами атак є злам паролів методами перебору (brute-force) та підробка сесійних токенів (наприклад, JWT), але архітектура розробленого Telegram-бота дозволяє делегувати найвразливіший етап – автентифікацію на сторону високо захищених серверів месенджера Telegram.

Автентифікація користувачів здійснюється автоматично та безпарольно кожне повідомлення, що надходить до серверної частини бота (FinanceBot.Host), містить перевірений серверами Telegram об'єкт Message, з якого вилучається унікальний ідентифікатор чату ChatId. Оскільки Telegram гарантує неможливість підробки відправника (IP-спуфінгу) на рівні свого закритого протоколу MTProto, розроблена інформаційна система може стовідсотково довіряти отриманому ідентифікатору. Це повністю усуває необхідність створення, збереження та складного хешування паролів (наприклад, алгоритмами bcrypt чи SHA256), що зводить ризик витоку паролів користувачів до абсолютного нуля.

Шифрування передавання даних у системі відбувається на двох незалежних етапах:

1) між користувачем (додаток Telegram) та серверами Telegram – використовується пропрієтарний криптографічний протокол MTProto (на базі симетричного шифрування AES-256);

2) між серверами Telegram та сервером розробленого бота – передача об'єктів оновлень (Updates) здійснюється виключно через протокол HTTPS із використанням сучасних стандартів TLS 1.2 або TLS 1.3. Завдяки цьому всі HTTP-запити проходять через захищене з'єднання, виключаючи можливість їх перехоплення (атака Man-in-the-Middle) чи модифікації третіми сторонами.

Основним вразливим елементом у такій архітектурі є не пароль користувача, а API-токен самого бота, який виступає єдиним ключем доступу до керування сервісом.

Токен, що видається сервісом BotFather, має стандартний формат, наприклад: 123456789:ABCDDefghIJKLmnopQRSTuvwXYZ, де перша частина (до двокрапки) – це відкритий ідентифікатор бота, а друга – криптографічно стійкий секретний ключ, що складається з 35 символів (великі, малі літери латинського алфавіту та цифри).

Для підтвердження надійності системи безпеки проведемо спеціальний розрахунок криптографічної стійкості секретної частини токена до атак типу «повний перебір» (brute-force). Кількість можливих комбінацій розраховується за формулою 4.1:

$$N = C^L. \quad (4.1)$$

де N – загальна кількість можливих комбінацій токена;

C – потужність алфавіту (кількість унікальних символів, що можуть бути використані);

L – довжина секретного ключа у символах.

Оскільки алфавіт токена складається з 26 малих літер (a-z), 26 великих літер (A-Z) та 10 цифр (0-9), сумарна потужність алфавіту становить: $C = 26 + 26 + 10 = 62$.

Довжина секретної частини токена становить $L = 35$ символів. Підставивши ці значення у формулу (4.1), отримаємо: $N = 62^{35} \approx 5.37 \times 10^{62}$.

Отримане число (5.37×10^{62}) є астрономічно великим. Для порівняння, сучасний гіпотетичний суперкомп'ютер, здатний перевіряти 100 мільярдів (10^{11}) комбінацій за одну секунду, витратив би на повний перебір усіх можливих варіантів токена приблизно 1.7×10^{44} років, що у мільярди разів перевищує орієнтовний вік Всесвіту.

Таким чином, математично доведено, що злам токена бота методом підбору є абсолютно неможливим, що гарантує рівень безпеки керування системою [15].

У розробленій системі цей ризик нівельовано на фундаментальному архітектурному рівні завдяки використанню сучасного ORM-фреймворку Entity

Framework Core. Замість ручного формування SQL-запитів шляхом небезпечної конкатенації рядків, система використовує типізовані запити LINQ. Під час звернення до PostgreSQL драйвер Npgsql автоматично перетворює ці запити на параметризовані команди (Parameterized Queries). У параметризованих командах введені користувачем дані передаються серверу бази даних як окремі параметри, а не як частина синтаксису SQL. Система управління базами даних строго розділяє логіку запиту та введені дані, тому будь-яка спроба впровадження шкідливого коду обробляється виключно як звичайний рядок тексту. Це унеможливлює модифікацію структури запиту і гарантує абсолютну безпеку фінансових даних користувачів.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи бакалавра було успішно вирішено актуальне науково-практичне завдання, яке полягало у проєктуванні, розробці та впровадженні сучасної автоматизованої інформаційної системи для персонального обліку фінансів на базі Telegram-бота. Поставлена на початку дослідження мета була повністю досягнута, а всі заплановані завдання – реалізовані у вигляді функціонуючого програмного продукту, повністю готового до експлуатації кінцевими користувачами.

На першому етапі дослідження було проведено глибокий системний аналіз предметної області та існуючих рішень на ринку фінансових застосунків, аналіз показав, що традиційні мобільні та вебзастосунки часто перевантажені зайвим функціоналом, вимагають тривалої реєстрації та додаткового встановлення на пристрій, що знижує залученість користувачів. На основі цього було прийнято обґрунтоване рішення розробити систему у форматі чат-бота для месенджера Telegram. Такий підхід забезпечив абсолютну кросплатформеність (система однаково добре працює на iOS, Android, Windows та macOS), миттєвий доступ до функцій без необхідності завантаження додаткового програмного забезпечення та інтуїтивно зрозумілий діалоговий інтерфейс.

Під час етапу проєктування (другий розділ роботи) було розроблено архітектуру системи на основі принципів «Чистої архітектури» (Clean Architecture). Було спроектовано оптимізовану реляційну модель бази даних для зберігання фінансових транзакцій, профілів користувачів та їх поточних станів.

Також були розроблені детальні UML-діаграми та блок-схеми алгоритмів, які візуалізували складну логіку маршрутизації повідомлень, механізми автентифікації користувачів за унікальними ідентифікаторами (ChatId) та процеси обробки як доходів, так і витрат.

Найбільш об'ємним та вагомим етапом стала практична реалізація серверної частини (бекенду), яка детально описана у третьому розділі.

Програмний продукт був реалізований мовою програмування C# з використанням найсучаснішої платформи .NET 9. Ключовим архітектурним досягненням стала відмова від монолітного коду на користь патернів проєктування, зокрема патерну «Ланцюжок обов'язків» (Chain of Responsibility), реалізованого через гнучкі інтерфейси обробників кроків (IStepHandler). Це дозволило створити масштабовану систему станів (State Machine), де бот коректно реагує на введення користувача залежно від його поточного кроку у системі (наприклад, очікування введення суми витрати чи очікування назви категорії).

Для забезпечення надійного зберігання та швидкої обробки великих масивів даних було застосовано об'єктно-реляційне відображення (ORM) за допомогою Entity Framework Core з використанням підходу Code-First. Базою даних було обрано потужну реляційну СУБД PostgreSQL, що гарантує сувору цілісність фінансових транзакцій (відповідність вимогам ACID) та підтримку багатокористувацької роботи.

У процесі розробки також було успішно реалізовано інтеграцію зі стороннім відкритим програмним інтерфейсом (API) ПриватБанку для отримання актуальних курсів валют у режимі реального часу. Крім того, розроблено складний програмний механізм динамічної генерації персоналізованих фінансових звітів у форматі PDF, які автоматично надсилаються користувачу у чат.

Таким чином, результати виконання кваліфікаційної роботи повністю підтверджують високий рівень теоретичної та практичної підготовки, а створений програмний продукт демонструє високу практичну цінність та має всі перспективи для успішної реалізації як комерційний проєкт.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Overview of ASP.NET Core. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core> (дата звернення: 05.02.2026).
2. The C# Programming Language Documentation. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/dotnet/csharp/> (дата звернення: 08.04.2025).
3. Telegram Bot API. Telegram Core. URL: <https://core.telegram.org/bots/api> (дата звернення: 12.02.2026).
4. Telegram.Bot C# Client. GitHub. URL: <https://github.com/TelegramBots/Telegram.Bot> (дата звернення: 14.01.2026).
5. Overview of Entity Framework Core. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/ef/core/> (дата звернення: 16.02.2026).
6. PostgreSQL: The World's Most Advanced Open Source Relational Database. URL: <https://www.postgresql.org/docs/> (дата звернення: 20.02.2026).
7. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/> (дата звернення: 22.02.2026).
8. QuestPDF: A comprehensive library for generating PDF documents in .NET. URL: <https://www.questpdf.com/documentation/getting-started.html> (дата звернення: 25.02.2026).
9. API ПриватБанку (Публічні сервіси). Курси валют. PrivatBank. URL: <https://api.privatbank.ua/> (дата звернення: 28.03.2026).
10. Dependency injection in ASP.NET Core. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection> (дата звернення: 02.02.2026).
11. JetBrains Rider: Fast & powerful cross-platform .NET IDE. URL: <https://www.jetbrains.com/rider/> (дата звернення: 04.02.2026).
12. MTProto Mobile Protocol (Cryptography). Telegram Core. URL: <https://core.telegram.org/mtproto> (дата звернення: 06.02.2026).

13. SQL Injection Prevention Cheat Sheet. OWASP. URL: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html (дата звернення: 08.02.2026).

14. SOLID Principles in C# and .NET. C# Corner. URL: <https://www.c-sharpcorner.com/article/solid-principles-in-c-sharp/> (дата звернення: 10.05.2026).

15. Contributors to Wikimedia projects. Brute-force attack – Wikipedia. Wikipedia, the free encyclopedia. URL: https://en.wikipedia.org/wiki/Brute-force_attack (дата звернення: 12.02.2026).

ДОДАТКИ

Додаток А

Фрагмент головний компоненту маршрутизації додатку (UpdateHandler)

```
using Telegram.Bot;
using Telegram.Bot.Types;
using Telegram.Bot.Types.Enums;
using Telegram.Bot.Types.ReplyMarkups; using FinanceBot.Domain.Entities;
using FinanceBot.Domain.Interfaces; using FinanceBot.Telegram.Handlers;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.EntityFrameworkCore;

namespace FinanceBot.Telegram;

public class UpdateHandler
{
    private readonly IServiceProvider _serviceProvider;

    public UpdateHandler(IServiceProvider serviceProvider)
    {
        _serviceProvider = serviceProvider;
    }

    public async Task HandleUpdateAsync(ITelegramBotClient bot,
    Update update, CancellationToken ct)
    {
        if (update.Message is not { Text: { } messageText } message)
            return;

        var chatId = message.Chat.Id;

        using var scope = _serviceProvider.CreateScope();
        var dbContext =
scope.ServiceProvider.GetRequiredService<IApplicationDbContext>()
;

        var user = await dbContext.Users
        .FirstOrDefaultAsync(u => u.ChatId == chatId, ct);

        // Обробка стартової команди та відображення
        // головного меню if (messageText == "/start")
        {
            if (user != null &&
                !string.IsNullOrEmpty(user.Name) &&
                !string.IsNullOrEmpty(user.SecretCode))
            {
                user.Status = UserState.Active;
                await dbContext.SaveChangesAsync(ct);
                await ResetToRegistrationAsync(bot, dbContext, user, chatId, ct);
                return;
            }
        }
        // Обробка системної команди для скидання профілю if (messageText ==
```

```

"/reset")
{
if (user != null)
{
var ops = await dbContext.Operations.Where(o => o.UserChatId ==
chatId).ToListAsync(ct);
dbContext.Operations.RemoveRange(ops); dbContext.Users.Remove(user);
await dbContext.SaveChangesAsync(ct);
}
await ResetToRegistrationAsync(bot, dbContext, null, chatId, ct);
return;
}

if (user == null) { await ResetToRegistrationAsync(bot, dbContext, null,
chatId, ct); return; }
user.Status = UserState.AwaitingTransferTarget; await
dbContext.SaveChangesAsync(ct);
await bot.SendMessage(chatId, "Введіть ChatId
отримувача:", replyMarkup: new ReplyKeyboardRemove(), cancellationToken:
ct);
return;

```